基于Android平台的游戏设计与实现

# 摘要

随着智能手机的不断发展和广泛普及，人们对于手机游戏的需求也日益高涨。移动端游戏凭借其便携性、休闲性、能够利用时间碎片等优势抢夺了原本电脑端游戏、主机端游戏持有的游戏市场的主导权。现在，移动端游戏已成为人们移动生活不可分割的一部分，移动端游戏作为一种已经被广泛认可的文化形式，正受到前所未有的关注。所以，设计和实现一款Android平台的游戏很有学习研究的价值。

游戏开发我使用的是Unity游戏引擎。Unity游戏引擎较为成熟，底层代码Bug较少，并且Unity游戏引擎能够提供丰富的功能组件系统：渲染引擎（即“渲染器”，含二维图像引擎和三维图像引擎）、物理引擎、碰撞检测系统、音效、脚本引擎、动画编辑系统、人工智能、网络引擎以及场景管理，大大简化了游戏开发流程、降低了游戏的开发难度。在开发中，学会去运用这些功能系统来开发游戏。

在游戏开发中，使用Unity游戏引擎的MonoBehaviour脚本系统，通过将c#脚本挂载到场景物体上，在MonoBehaviour的各个生命周期中编写代码去实现游戏的逻辑控制。

使用Unity游戏引擎的动画系统，用动画编辑器去制作人物、陷阱、怪物的独立动画，并用动画状态机结合脚本代码去控制人物的动作表现，使人物看起来栩栩如生。

使用Unity游戏引擎的物理系统，给物体绑定刚体组件，从而可以通过刚体组件调用Unity物理引擎提供的API去实现人物运动、陷阱掉落、子弹发射等物理运动效果。

使用Unity游戏引擎的碰撞检测系统，通过给物体绑定碰撞器，去检测碰撞器间的碰撞并触发碰撞回调从而去实现一些如人物踩到陷阱死亡等的功能。

使用Unity游戏引擎的场景编辑器，在可视化视图中搭建游戏场景。

使用Unity游戏引擎的UGUI去完成UI界面的制作、UI界面的屏幕适配。

使用Unity Android Build Support去构建出一个Android平台的原生apk，能使游戏正常在Android平台运行。

通过上面这些Unity游戏引擎提供的功能系统，去实现人物动作、运动、控制，摄像机跟随，怪物、陷阱，场景的搭建、控制切换等游戏功能，开发出一个完成的游戏。

**关键词：**移动端游戏开发、Unity游戏引擎、MonoBehaviour脚本系统、动画系统、物理系统、碰撞检测统、场景编辑器、UGUI

# 绪论

本章主要描述移动端游戏行业分析以及游戏引擎的背景、发展现状，进而提出本文所要研究的内容及目标。

## 1.1 移动端游戏行业分析

随着4G的到来，移动端游戏也能够提供可操作性更强、网络延迟更低以及游戏体验更好的网络游戏。游戏的开发也从早期以JAVA、Flash、Web技术为基础开发的低质粗糙的小型游戏转变为Unreal、Unity、Cocos等功能更加强大的通用游戏引擎，有些大型游戏公司经过长年的技术积累甚至能自主开发出功能更加齐全，性能更高的游戏引擎。因而，通过游戏引擎，游戏开发者们能为玩家提供画面更加精美，玩法更加复杂多样的游戏。

此外，国内经济的良好发展业也为游戏行业的未来提供了强有力的保障。在2012-2015年期间，在国家政策的扶持下，手机游戏很快由萌芽期进入到高速发展期。与手机游戏相关的各类投资商，开发团队大量涌入市场。

然而，手机游戏的飞速发展却引发了市场混乱的问题。2016年之后有关手机游戏行业新政策发布后，手机游戏的审批时间与难度就大大增加了。这次新政策淘汰了近一半的游戏开发团队，结束了手机游戏混乱发展的状态，让手机游戏在更有序的环境里健康发展。

## 1.2 游戏引擎的背景、发展现状

游戏引擎是为方便游戏开发，为开发者提供的一系列类库、开发框架以及可视化操作界面。有了游戏引擎的帮助，游戏的开发将会极大简化，游戏开发者不用每次开发游戏都从最底层的图形渲染做起。

十几年前的游戏现在看来都很简单，容量大小只有几M，但当时通常一款游戏开发周期都在8到10个月左右。因为每款游戏的开发都需要从头开始编写代码，期间存在着大量重复劳动，耗时耗力。于是渐渐地，开发人员发现某些游戏总会有相同的代码。将这些相同的代码抽离出来可以应用在同题材的游戏上，这样就可以大大减少游戏开发流程，缩短游戏开发周期和开发费用。慢慢地，这些通用代码就形成了游戏引擎的雏形。之后，伴随着硬件更新迭代以及游戏开发技术的不断积累，游戏引擎不断完善就成了如今的样子。

如今的游戏引擎，不光提供游戏开发的通用代码供开发人员调用，还提供了友好的可视化操作界面能够供非程序人员使用，以及各种功能系统：渲染引擎（即“渲染器”，含二维图像引擎和三维图像引擎）、物理引擎、碰撞检测系统、音效、脚本引擎、动画编辑系统、人工智能、网络引擎以及场景管理。

## 1.3 本文研究内容及目标

如今游戏的开发已离不开游戏引擎，对于程序员来说熟练地掌握游戏引擎是游戏开发的必经路。所以，本文就通过开发Android平台游戏的机会去深入学习比较成熟的Unity游戏引擎。

### 1.31 研究内容

要求开发一款基于Android平台的游戏，要求游戏中要用到Cocos或者Unity引擎，界面设计合理、美观，游戏场景构建逼真，特效应用合理。

### 1.32 研究目标

使用Unity游戏引擎，从零开始开发一款基于Android平台的游戏。游戏类型是类似猫里奥的2D横板过关游戏，游戏素材从Unity Asset中获取。实现人物各种动作状态，场景中各种地形、陷阱、怪物及其相应的行为逻辑， 以及摄像机的合理跟随、游戏元素的交互。

在开发过程中，学会熟练掌握Unity游戏引擎，学会使用其提供的各种API接口，学会构建良好的代码结构，以及完整体验一个游戏的开发流程。

### 1.33 本文组织结构

本篇论文总共分为五章。

第一章主要介绍移动端游戏的行业情况、游戏引擎的出现背景、发展和现状，并提出了本文的研究内容和研究目标。

第二章讲解了必要开发环境的搭建，包括了Unity游戏引擎的安装与配置、Mono开发框架的安装，代码编辑器Visual Studio Code的安装与配置，JDK的安装与配置，Android Studio的安装配置，SourceTree的安装与使用。

第三章是总体设计，大致讲述了人物的设计，场景的设计，摄像机的设计，陷阱、怪物、道具的设计以及UI界面的设计。

第四章详细设计与实现是整篇论文的重点，在第四章会详细讲解整个游戏的实现过程，包括人物相关实现，摄像机相关实现，陷阱、怪物相关实现、场景相关实现。

第五章对全文进行总结，归纳了使用Unity游戏引擎的开发流程、开发游戏遇到的难点并提出需要改进的地方。

# 开发环境的搭建

## 2.1 Unity游戏引擎安装

2019版本的Unity游戏引擎官方推荐通过Unity Hub来管理安装。通过Unity Hub可以下载安装任何版本的Unity游戏引擎或者相关组件，还可以管理磁盘中所有不同版本的Unity游戏引擎以及所有Unity工程项目。去Unity官网下载安装Unity Hub，再启动Unity Hub并登录Unity账号去安装最新稳定版本的Unity游戏引擎，相关组件勾选Android Build Support以及中文语言支持（VS Community也可勾选），之后Unity Hub就会自动后台下载并安装。安装完，打开Unity Hub在Projects选项卡下点击new就可以新建工程。

## 2.2 Mono开发框架的安装

因为Unity引擎的多平台发布依赖于Mono对于.net框架的可跨平台的实现，并且Mono中还包含了C#的编译器，所以Mono也是必须要安装的。Mono的安装可直接在Mono官网下载安装即可。

为何Unity能够跨平台发布原生APP？ 其原理就在于Mono重新实现了.net标准中叫做CIL（Common Intermediate Language，微软通用中间语言）的一种代码指令集的编译与反编译。CLI可以在任何支持CLI（Common Language Infrastructure，通用语言基础结构）的环境中运行。这样跨平台只需将代码编译成CIL，然后再在各个平台运行时，用各个平台的CLI去解释运行CIL或者将其编译为该平台下的原生代码，而Mono就是实现了不同平台下的CLI。

## 2.3 代码编辑器VS Code安装与配置

代码编辑器我选用的是Visual Studio Code，因为其比较轻量，占用内存比较小，开启关闭速度快。当然也可以用Visual Studio，VS能够提供更完整的代码提示补全机制。VS Code可直接在官网下载安装，安装完成后还需去安装Debugger for Unity插件用于Unity运行调试，以及C#支持插件用于让编辑器能识别C#代码从而可以实现多个脚本代码关联、查找引用、代码格式规范、代码提示补全等功能。因为该C#支持插件是基于OminSharp的并且在windows下依赖于.net Framework，所以还需安装.net Framework Dev Pack，并且不能安装最新的4.8版本（因为c#插件不支持4.8版本，4.7版本为宜）。

## 2.4 JDK的安装

因为该项目最终要发布到Android平台，所以必须要安装JDK。JDK直接在官网下载安装即可。安装后还需配置环境变量，以windows为例：

在系统变量ixa新建一个名为JAVA\_HOME的变量，变量值为jdk的安装目录。（用于指定jdk的安装路径，很多Java程序会用到这个变量，接下来的PATH和CLASSPATH的变量也会使用到该变量）

在系统变量下查看PATH变量，如果没有就新建一个。点击编辑，在变量值文本框的末尾添加上“%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;”。（Path指定了一个路径列表，当执行一个可执行文件时，如果在当前路径下不能找到这个文件，就会依次安装Path下的路径去一一查找，直到找到为止，否则就会报错。因为Java的编译命令（javac)，执行命令（java）和一些工具命令（javadoc，jdb等）都在其安装目录的bin下以及其安装目录下的jre的bin下，因此将这两个路径添加到PATH变量中）

在系统变量下查看CLASSPATH变量，如果没有同样新建一个。点击编辑，在变量值文本框的起始位置添加上“.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar;”（CLASSPATH也是指定了一个路径列表，是用来搜索Java编译时需要用到的类。最前面的“.”代表了当前目录，包含了该目录后，就可以到任意目录下去执行需要用到该目录下某个类的 Java 程序，即便该路径并未包含在 CLASSPATH 中。

## 2.5 Android Studio的安装

因为最新版本的Unity引擎构建Android apk舍弃了原先的内置构建，转用了Android Studiod的gradle构建，同时发布到Android平台还需要用到Android SDK，所以要去安装Android Studio。Android Studio在官网上下载安装。在安装过程中，设置安装路径不应带有中文字符以及空格。安装完成后，紧接着会提示安装Android SDK，我因为自己手机是Android 6.0的缘故选择的是API 21，高版本的API会导致无法发布到低版本的Android手机上。安装完成后，需要新建一个项目并打开，这时右下角会显示正在下载安装gradle，gradle下载安装完，并且空项目构建完就可以关闭Android Studio了。

## 2.6 Unity Android Build Support 配置

Unity还需设置发布平台，打开Unity创建完工程项目后，选择左上角File->Build Settings->Android->Switch Platform即可切换发布平台到Android。此外，接下来点击android下player settings，在弹出来的Project Settings的窗口里选择Player选项卡，设置好Company Name、Product Name、Default Icon后，打开下面的Resolution and Presentation选项卡设置Default Orientation为Landscape Left（左侧横屏）。这样发布到Android平台的所有准备就全都做好了。

## 2.7 Source Tree的安装与配置

这个游戏项目是交给GitHub管理的。Sourcetree是一款GitHub代码管理工具，有了这个工具可以不用敲Git指令很方便地实现创建、克隆、提交、push、pull、合并等操作。需要先去官网下载安装git，然后再去下载安装Sourcetree，安装过程中还需登录BitBucket账户。安装完成后，还需给本机提交权限。首先要去生成本机的ssh密钥，通过在cmd中输入指令“ssh-keygen -t rsa -C ["youremail@example.com"](mailto:%22youremail@example.com%22) ”来完成，之后登录GitHub，打开setting->SSH keys，点击右上角 New SSH key，把生成好的公钥id\_rsa.pub放进 key输入框中，最后在SourceTree中配置好SSH密钥和SSH客户端(SSH客户端选择OpenSSH)即可。

# 总体设计

## 3.1 人物设计

### 3.11 人物形象设计

人物的形象由几张身体、手、脚、眼睛表示人物身体部分的sprite图片拼凑而成。这样，可以通过改变人物身体各个部位的位置、旋转角度、缩放大小，让人物做出不同的姿势、动作。

### 3.12 人物动作设计

人物的动作使用Unity的动画编辑器来完成。相比较于在人物脚本代码里通过FixedUpdate去每帧切换人物的Sprite图片的方式，这样可以不再依赖于帧动画图片素材。Unity的动画编辑器提供了一个帧时间轴，我们只需给场景中结点在这个时间轴的某个帧上定义关键帧即可，当播放这个动画时，Unity会自动计算出从一个关键帧到另一个关键帧的过渡帧。

通过Unity的动画编辑系统，需要制作出人物的若干个独立动作：待机动作、开始奔跑、奔跑中、刹车、跳跃、死亡。人物不同动作之间切换还需要做过渡，这个就依赖于Unity提供的动画融合功能，通过设定一些参数就实现同一个物体的不同动画间的流畅转换（这个在实现里会详细阐述）。此外，这些人物动作的切换还需脚本代码控制，Unity的动画控制系统会提供设置动作转换的条件参数，我们需要在脚本代码里设定这些参数，来起到控制切换的作用。

### 3.13 人物运动控制设计

人物运动的实现有两种方式，一个是在脚本里每帧根据玩家的按键去刷新人物位置，另一种是通过Unity物理引擎来实现。我选择是后者，因为物理引擎能够提供更加丰富多彩的人物运动表现，比如人物奔跑起来后不能立马停下的加速度、角速度引起的翻滚、重力的加持影响等等，这些都只要调用人物的刚体组件提供的物理引擎相关的方法即可。

### 3.14 人物影子设计

这个游戏还有个人物影子的设定，用于复现上一次死亡前玩家操作的人物的行为，用于提醒玩家上一把遇到的坑以及增加游戏趣味性。这个通过每帧去记录人物的帧数据，然后在下次重生时将这些帧数据应用到人物影子上播放出来即可。

## 3.2 游戏场景设计

## 3.2.1 场景的视图布置

场景的视图主要考虑到背景、人物、活动地面、地面装饰。我们将背景的sprite图片层级设为最低，循环铺设在场景的最后面，人物各个部位的sprite层级设为最高来避免遮挡，然后就可以根据设计依次铺上地、水以及树、花等地面装饰来完成一个美观关卡视图。

### 3.2.2 人物活动地面

地面是场景中支持人物运动的关键所在，所有地面必须绑定上2D碰撞器组件来支撑人物，除此之外，不同坡度的地面决定了人物站在这块地面上运动时所施加的力的方向和大小，所以还需绑定一个地面脚本来修改人物移动时所施加的力。

### 3.2.3 场景限制区域

场景不是无限大的，所以为避免人物走到非法区域，所以左右两边需要设置空气墙，底下需要设置一个死亡线，当人物落到这个死亡线时会置人物死亡并重生。右边的空气墙还有触发切换到下一关的作用，当人物碰到右边空气墙时，就会触发碰撞回调调用切换场景的方法。

### 3.2.4 场景切换

场景的切换使用异步加载的方式，这样可以让两个关卡切换衔接起来更加自然流畅，当触发切换关卡时，原场景会还在，并且会慢慢变黑，此时第二个场景在后台加载，加载完成后才将黑色遮罩慢慢变透明。

## 3.3 摄像机设计

因为是2D游戏，所以摄像机选用正交摄像机，它能忽视场景物体的远近，呈现出2D画面。

### 3.3.1 摄像机跟随

摄像机需要去跟随人物移动，那么只需在每帧里去调整摄像机到相对于人物的一个固定位置即可。为了实现美观，我在每帧中将摄像机调整到刚好能使人物在屏幕中处于黄金比例的一个位置上。

除此之外，为避免摄像机照摄到非法区域而导致穿帮，摄像机的移动还需设定区域限制，摄像机的区域限定需要通过接下来要介绍摄像机动态的正交大小来计算出具体值。

### 3.3.2 正交摄像机的手机适配

为了能使游戏场景在各个手机上都能按照合适的比例显示出来，在摄像机的脚本里需要根据当前手机屏幕的分辨率去动态调整摄像机的正交大小，并根据调整好的摄像机正交大小去计算出摄像机的移动左右区域边界以及人物跟随时相对与人物的位置。

## 3.4 陷阱、怪物、道具设计

### 3.4.1 陷阱、怪物触发人物死亡

触发人物死亡都写在对应陷阱、怪物脚本中的碰撞、触发回调中，当人物碰到陷阱、怪物的碰撞器时就会去调用人物脚本提供的死亡方法。人物的死亡动画的末尾绑定一个复活的回调方法，当人物死亡动画播放完就会自动去调用复活方法。

### 3.4.2 人物复活时陷阱、怪物的重置

人物复活时，场景中所有的物体都要重置回原先的状态，所以所有的陷阱、怪物的脚本都要提供一个reset方法去恢复状态，reset方法会在初始化的一开始利用c#的事件与委托机制绑定在人物脚本的一个事件上，而这个事件会在人物复活方法中被调用，这样就实现了人物重生时的陷阱、怪物重置。

### 3.4.3 陷阱、怪物的触发唤醒设计

对于需要触发的陷阱、怪物，可以利用双重碰撞器来实现，最里面层的碰撞器包裹陷阱、怪物体积用来代表实体，来触发人物碰到死亡的回调，之后再嵌套一层碰撞器勾选上trigger用于触发调用陷阱、怪物唤醒的方法。这样做的好处是，可以在可视化场景编辑器中直观地配置陷阱、怪物的触发半径。

### 3.4.4 陷阱的寻路方式

陷阱的寻路，可以为陷阱配置一个坐标点队列与移动速度，然后让陷阱根据配置的速度依次沿着这个队列中点去移动，这样就能很简单地去规划陷阱的移动路径。地面也可以用这个方法，实现移动载人的效果。

### 3.4.5 怪物的跟踪方式

怪物的跟踪取决于人物距怪物的方位，因为这是一个2D横板游戏，所以可以将方位简化为只有左和右，通过每一帧去判断人物距怪物的左侧还是右侧从而让怪物往相应的位置移动，一旦人物进入怪物的攻击范围，怪物就会立即发动攻击。此外，怪物的跟踪不能是全图的，所以还需设定一个跟踪范围，或者当人物一旦脱离怪物的检测范围，怪物就会返回原先的导航寻路点。

### 3.4.6 射击型怪物的射击方式

射击型怪物子弹的射击通过物理引擎来实现，通过给子弹刚体以冲量而将子弹射击出去，射击的角度由当前帧怪物与人物间的方向向量来决定，子弹在生成好后需要设定多少时间后销毁，防止其一直占用内存。怪物的射击可以先调用射击动画，在射击动画的末尾绑定射击方法的回调，这样可以实现怪物将射击动作做完后才将子弹射击出去。

### 3.4.7 下落型陷阱

下落型陷阱可以通过在触发回调里改变陷阱刚体所受重力倍数来实现，坍塌的地面、高空落刺都是这样实现的。

### 3.4.8 加速、减速陷阱

加速、减速陷阱通过在人物触发陷阱时，每帧里去将人物刚体的速度乘以倍率来实现，在人物离开陷阱范围后恢复人物速度来实现。

### 3.4.9 中途存档点

当人物触碰到中途存档点时，触发trigger回调，在那里面会重设人物的复活位置以此来实现中途存档点的功能。

## 3.5 UI界面设计

### 3.5.1 人物控制的虚拟按键

用于代替键盘控制人物左右移动、跳跃，为了能有按下反馈，需要在PointerDown、PointerUp、PointerExit事件绑定的回调中去切换它的图片。（具体如何控制人物移动会在详细实现中说明）

### 3.5.2 暂停键

用于暂停游戏，通过修改Unity的系统时间为0来实现。当点击暂停后，会产生一层灰色遮罩，暂停键也会变为红色，当再次点击暂停键时，就会继续游戏。

### 3.5.3 上、下一关

用于切换关卡，点击后会切换到上、下一关，方便调试关卡。

### 3.5.4 开始页面

游戏的启动页面，设计为游戏标题的浮动，以及点击任意处标题飞出，人物跑着出现并开始游戏。这些通过动画编辑器以及脚本里绑定点击事件triggerEvent回调来实现。

# 详细设计与实现

## 4.1 人物的相关实现

### 4.11 人物单个动作的实现

要完整地实现人物的动作表现，首先要确定人物一共有哪几个动作，根据上一章所说我所设计的人物动作有6个：待机动作、开始奔跑、奔跑中、刹车、跳跃、死亡。那么，在确定好所有的人物动作后，需要去利用Unity的动画编辑器去把这些单个动作一一实现出来。要编辑人物动画，必须先要有个人物动画的控制器。首先需要在资源文件夹下新建一个动画控制器文件取名为CharacterAnimController，接着在人物结点上绑定动画控制器组件，将CharacterAnimController拖入到这个组件中，这样人物的动画控制器就准备就绪了。选中人物结点，然后点击工具栏中的Windows -> Animation -> Animator调出动画控制器视图，接下来我们就可以看到Unity的动画控制是通过动画状态机来实现的，我们需要为每个独立的动画创建一个状态并取好相应的名称。

在动画控制视图中创建好动画状态后，选中一个状态，可以看到在右侧Inspector窗口中Motion为空，Motion就是该状态的动画文件。以跳跃动画为例，在资源文件夹下新建一个Animation文件取名为CharacterJump，接着将CharacterJump文件拖入Jump状态的Motion中，之后通过点击工具栏Windows -> Animation -> Animation打开动画编辑窗口就可以开始编辑动画了。

选中人物结点，我们可以在动画编辑窗口中看到之前创建的CharaceterJump的动画，点击Add Property可以为其添加人物结点或其子结点的属性。添加完动画需要的属性后，在时间轴相应的位置上右键选择添加关键帧，接着将表示播放位置的竖线移到刚刚创建的关键帧位置上，之后就可修改该关键帧上左侧结点属性的值了。或者，也可以通过点击动画编辑窗口左上角的小红开启录制模式，在录制模式下一切对人物结点属性的修改都会被录入到这个动画中，因而可以直接对场景中人物的各个子结点拖动、旋转来完成动画。

### 4.12 人物动作状态机

根据上述方法，为6个动画状态完成单独动画的制作，之后需要为这6个状态建立状态转换。状态转换需要转换条件，所以必须明确人物各个动作间的转换前提。

待机状态是默认状态，所以右击Entry连线至待机状态，表示将待机状态设为默认状态，这样游戏一运行，状态机就会自动转移到待机状态播放待机动画。

从待机状态能够转换到开始奔跑状态，转换条件是人物在地上、移动键被按下，所以我们需要为这两个条件创建相应的变量，在动画控制窗口左上角有个Parameters的子窗口，就在这里点击+键分别建立两个bool类型的变量：isGround表示人物是否在地上、isBtnRun表示移动键是否被按下。这两个变量可以为整个人物动画控制器所使用。创建好两个变量后，点击从待机状态到开始奔跑状态引出的边，在右侧的Inspector中可以看到有一栏Conditions，在这里可以点击“+”键可以添加刚刚创建好两个变量isGround、isBtnRun右侧的值都设为true。当人物在地上，并且移动键被按下时，动画状态机才会从待机状态转换到开始奔跑状态。而开始奔跑状态到奔跑中状态的转换，不需要任何转换条件，开始奔跑动画播放完之后即可开始循环播放奔跑中动画。

当处于奔跑中状态时，当同时满足人物在地面、移动键松开，水平速度大于4时转换为刹车状态，同样新建一个表示人物水平速度的float变量horSpeed，将条件加入到转换边的Conidtions中。当处于奔跑状态时可能会切换到跳跃状态，条件是不处于地面、跳跃键被按住了。奔跑状态也会切回到待机状态，条件是移动键松开并且人物处于地面上。

刹车状态也会转换到奔跑中状态，条件是人物在地上并且移动键被按住。刹车状态同样也会切换到待机状态、跳跃状态，条件和从奔跑种状态切换过去的一样。这时，我们发现开始奔跑、奔跑中、刹车三个状态的状态转移目标和条件都一样，为了简化状态机我们可以给这三个状态建立一个Sub-State Machine取名为Run，将这三个状态全部放入Run中，Run中包含了内部三个状态的关联，而Run这个外层状态负责和外部状态的关联。引入子状态机可以有效地减少关联边，使状态机图看起来更加清晰。

待机状态也可以转换到跳跃状态，条件是人物不在地面上并且跳跃键被按下了，跳跃状态也可以切换回待机状态，条件刚好相反。

任何状态都可以转换到高空掉落、死亡状态，这就要用到在人物动画控制器从创建以来一直存在的一个状态Any State，它代表图中的任意一个状态。创建从Any State到高空掉落状态的转换边，条件是人物不在地上，垂直高度大于4，跳跃键没按住。从Any State到死亡状态的转换条件是一个trigger类型，trigger类型会在值为true时触发转换，同时自动将值置回false。高空掉落、死亡状态也都能转换到待机状态，两个条件分别为人物在地面上、triggerRebirth重生。

这样，人物动作的状态转换机就全部完成了，如图：

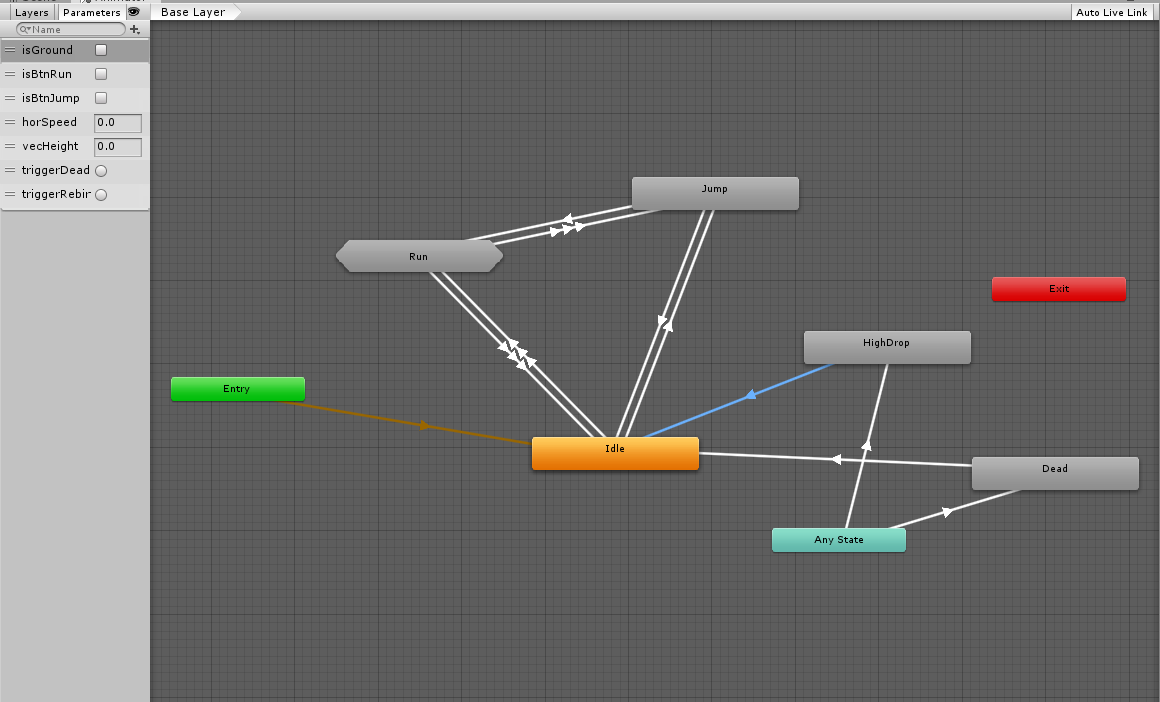


图1 人物动作状态机图（外层）

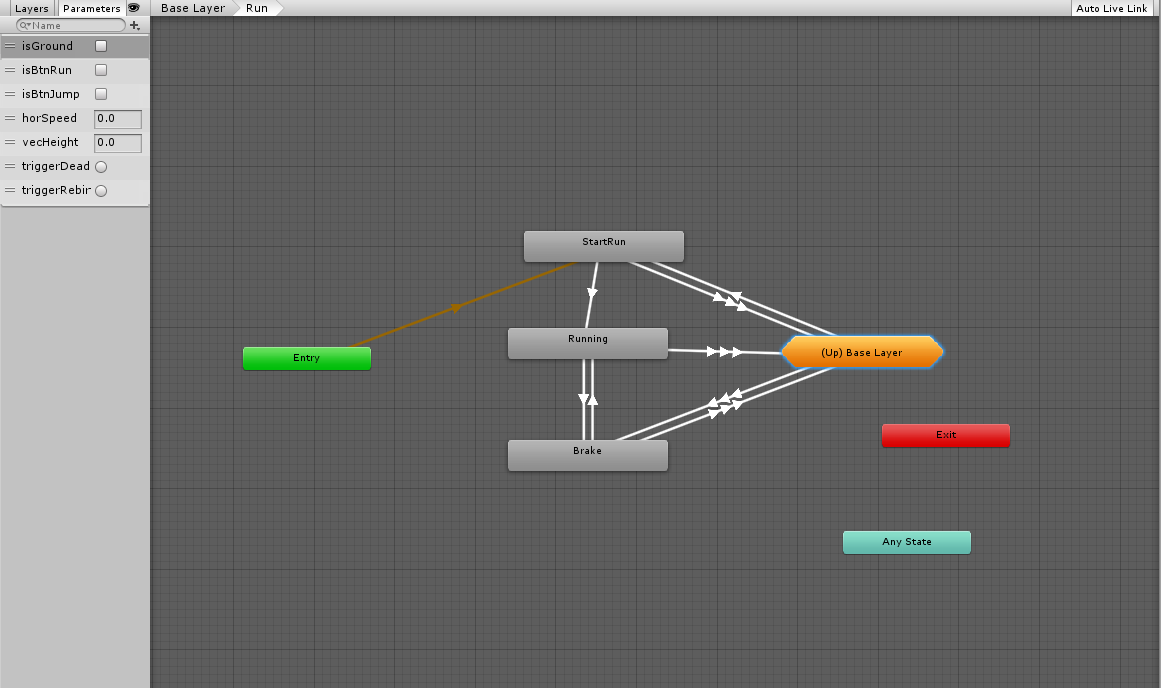


图2 人物动作状态机图（Run子状态机内部图）

### 4.13 人物动作融合过渡部分参数说明

在转换边的Inspector窗口，我们可以看到在Conditions上面还有很多参数，这些参数也是必须要去了解的。

Has Exit Time规定了状态A处于动画的特定时刻才能转换到B，而Exit Time就是设定的该时间。若Has Exit Time未被勾选，而转换可在源状态动画的任意时刻发生转换。在StartRun切换到Running的转换边上，Has Exit Time被勾选是因为只有StartRun的动画全部播放完才可以去播放Running的动画。

Transition Duration状态转换时动画融合过渡的时长，我这里都是设置的0.2s。

Transition Offset转换目标动画的偏移，即发生转换时动画融合从目标动画的哪个位置开始，我设的都是0。

### 4.14 人物运动的实现

之前做好的人物动作，以及人物动作状态转换机需要通过人物脚本去控制它。为此，需要在资源文件夹下新建一个人物脚本CharacterBehaviour.cs，并将它拖到人物结点上作为人物结点的组件。

人物的运动是利用Unity物理引擎实现的。在人物脚本里，通过获取按键的输入，去改变bool变量isRun的值，再在Update生命周期里根据isRun调用物理引擎方法使人物左右移动。

Update中有关人物左右移动的详细代码如下：

|  |
| --- |
| if (this.isRun)  {  if (System.Math.Abs(this.rigidBody.velocity.x) < this.maxRunSpeed || this.isReleaseSpeed)  {  if (this.isRightDir)  {  Vector2 force;  if (!this.isJump && this.isGround)  {  switch (this.pathType)  {  case PathBlockType.FlatPath:  force = this.flatRunForce \* Vector2.right;  break;  case PathBlockType.LeftHillPath:  force = this.upHillRunForce \* (new Vector2(1, 1).normalized);  break;  default:  force = this.downHillRunForce \* (new Vector2(1, -1).normalized);  break;  }  }  else  {  force = this.flatRunForce \* Vector2.right;  }  this.rigidBody.AddForce(force);  }  else  {  Vector2 force;  if (!this.isJump && this.isGround)  {  switch (this.pathType)  {  case PathBlockType.FlatPath:  force = this.flatRunForce \* Vector2.left;  break;  case PathBlockType.LeftHillPath:  force = this.downHillRunForce \* (new Vector2(-1, -1).normalized);  break;  default:  force = this.upHillRunForce \* (new Vector2(-1, 1).normalized);  break;  }  }  else  {  force = this.flatRunForce \* Vector2.left;  }  this.rigidBody.AddForce(force);  }  }  else  {  if (this.rigidBody.velocity.x >= 0)  {  this.rigidBody.velocity = new Vector2(this.maxRunSpeed, this.rigidBody.velocity.y);  }  else  {  this.rigidBody.velocity = new Vector2(-this.maxRunSpeed, this.rigidBody.velocity.y);  }  }  } |

当左右移动键被按下时，变量isRun会被置为true，此时在Update里会触发上述代码。先检测此时人物的水平速度是否已达到最大值maxRunSpeed，若已达到最大值，则根据按下移动键的方向将人物水平速度保持在最大值，否则根据人物当前帧所在的地形与按下移动键方向，给人物施加特定方向的力（还需判断人物是否处于空中，若在空中则移动的力是水平方向的）。

当左右移动键被松开时，isRun则会被置为false，此时不会在Update里触发上述代码，也就不会给人物施加力，人物会在物理引擎摩擦力的作用下停下来。

而有关于人物跳跃的详细代码如下：

|  |
| --- |
| private void onBtnJumpUp(BaseEventData eventData = null)  {  Image btnImg = this.jumpTrigger.GetComponent<Image>();  btnImg.sprite = this.btnJumpImgs[0];  if (this.isJump)  {  this.rigidBody.AddForce(new Vector2(0, -180));  this.isJump = false;  }  } |

当按下跳跃键时，判断人物跳跃剩下段数大于0、人物在地面上并且isJump为false，这样才能去执行人物的跳跃，人物的跳跃通过物理引擎给人物一个垂直方向跳跃初速度来实现，同时还需要将人物的跳跃段数-1，让人物接下来不能再去跳跃。而跳跃段数的恢复，如下代码：

|  |
| --- |
| void OnCollisionEnter2D(Collision2D collision)  {  if (this.isGround)  this.jumpLeftSeg = this.jumpMaxSeg;  } |

OnCollisionEnter2D是带有碰撞器的结点碰撞到其他碰撞器时会被回调的方法。当人物落到地面时，跳跃段数重置为设定的最大值。

人物的跳跃还有一个设定是当玩家按住跳跃键时能跳得更高。这个实现其实在跳跃键松开的回调处理里：

|  |
| --- |
| private void onBtnJumpUp(BaseEventData eventData = null)  {  Image btnImg = this.jumpTrigger.GetComponent<Image>();  btnImg.sprite = this.btnJumpImgs[0];  if (this.isJump)  {  this.rigidBody.AddForce(new Vector2(0, -180));  this.isJump = false;  }  } |

按住跳跃键能跳得更高，换种思维就是跳跃过程中当跳跃键松开时，人物就要下落，所以只需在跳跃键松开时，给人物施加一个向下的力就可以了，同时还要把isJump置为false表示人物不再处于跳跃状态。一直按住跳跃键，由于受到重力的影响，跳跃也会到达一个最大高度，此时也需要将isJump置为false，这个在Update里去检查：

|  |
| --- |
| if (this.isJump && this.rigidBody.velocity.y <= 0)  {  this.isJump = false;  } |

当Update里判断出当前处于跳跃状态并且人物垂直方向速度小于等于0时，就将isJump置为false。

### 4.15 人物是否在地面上的判断

在之前的叙述中，代表人物是否处于地面上的变量isGround经常会被使用到，而isGround又是何处被赋值的？请看Update中的这段代码：

|  |
| --- |
| RaycastHit2D hit2D = Physics2D.Raycast(this.transform.position, Vector2.down, 50, 1024);  if (hit2D.collider)  {  \*\*\*  this.isGround = (hit2D.distance <= 1.25) ? true : false;  \*\*\*  }  else  {  \*\*\*  this.shadowFrameData.isGround = false;  \*\*\*  } |

游戏的每帧都会从人物中心发射一条垂直向下发射一条射线，如果射线发射碰撞，并且碰撞距离小于等于1.25，则表示人物踩在地面上从而将isGround置为true，否则人物不在地面上isGround置为false。

### 4.16 人物动作状态机的脚本控制

人物动作状态机的控制需要人物动作状态机、人物运动实现完成之后才能去做。人物动作的控制就是操作之前人物状态机定义的那几个状态转换条件变量，所以我们先要在人物脚本里能够获取到那几个变量。为了能够更高效地获取到控制变量，Unity官方推荐使用Hash值去获取而不是直接通过变量名。因而我们需要新建一个专门获取Hash值的脚本AniHashCode.cs，在那里面定义静态变量，静态变量在定义时就被赋了经过Animator.StringToHash方法转换好的Hash值，我们只需要通过这些静态变量去获取人物状态机的控制变量即可。详细请看代码：

|  |
| --- |
| // Character  #region  public static readonly int isGround = Animator.StringToHash("isGround");  public static readonly int isBtnRun = Animator.StringToHash("isBtnRun");  public static readonly int isBtnJump = Animator.StringToHash("isBtnJump");  public static readonly int vecHeight = Animator.StringToHash("vecHeight");  public static readonly int horSpeed = Animator.StringToHash("horSpeed");  public static readonly int triggerDead = Animator.StringToHash("triggerDead");  public static readonly int triggerRebirth = Animator.StringToHash("triggerRebirth");  #endregion |

在有了人物状态机控制变量的hash值后，在人物脚本中可以通过SetBool、SetFloat、SetTrigger等方法去修改对应的控制变量，如下代码：

|  |
| --- |
| this.animator.SetBool(AniHashCode.isBtnRun, this.isRun); |

那么，之后只需在Update生命周期里去给所有人物状态机的控制变量设定好在这帧里的实时值就可以了。

|  |
| --- |
| // 给动画状态机参数赋值  this.animator.SetBool(AniHashCode.isBtnRun, this.isRun);  this.animator.SetBool(AniHashCode.isBtnJump, this.isJump);  this.animator.SetFloat(AniHashCode.horSpeed, System.Math.Abs(this.rigidBody.velocity.x));  RaycastHit2D hit2D = Physics2D.Raycast(this.transform.position, Vector2.down, 50, 1024);  if (hit2D.collider)  {  this.animator.SetFloat(AniHashCode.vecHeight, hit2D.distance);  this.isGround = (hit2D.distance <= 1.25) ? true : false;  this.animator.SetBool(AniHashCode.isGround, this.isGround);  }  else  {  this.animator.SetFloat(AniHashCode.vecHeight, 10000);  this.shadowFrameData.vecHight = 10000;  this.animator.SetBool(AniHashCode.isGround, false);  this.shadowFrameData.isGround = false;  }  this.vecheight = hit2D.distance; |

只要人物状态机构建的没有问题并且所有控制变量都能取到每一帧的实时值，那么人物状态机就会正常工作，人物各种动作也会在相应的时机播放出来。

### 4.17 人物影子的实现

在实现了人物状态机的控制之后，人物影子也可以实现出来了。要做人物影子，首先需要一个人物影子的结点，我将人物结点复制了一份，并将颜色调整为半透明黑色，这样人物影子结点就完成了。人物影子的行为完全依赖为收录的帧数据，所以需要定义帧数据结构。在资源管理器下，新建一个脚本CharacterFrameData.cs，类结构如下：

|  |
| --- |
| public class CharacterFrameData  {  public Vector3 pos;  public Quaternion rot;  // 动画状态机相关  public bool isRun;  public bool isJump;  public float horSpeed;  public float vecHight;  public bool isGround;  public bool triggerDead;  } |

之后，需要为人物影子绑定一个脚本CharacterShadow.cs，在这脚本里面实现播放帧数据的方法，代码如下：

|  |
| --- |
| void FixedUpdate()  {  if (!this.isAllowPlay || this.frameDatas == null || this.playIndex >= this.frameDatas.Length)  return;  var frameData = this.frameDatas[this.playIndex];  this.transform.SetPositionAndRotation(frameData.pos, frameData.rot);  this.animator.SetBool(AniHashCode.isBtnRun, frameData.isRun);  this.animator.SetBool(AniHashCode.isBtnJump, frameData.isJump);  this.animator.SetFloat(AniHashCode.horSpeed, frameData.horSpeed);  this.animator.SetFloat(AniHashCode.vecHeight, frameData.vecHight);  this.animator.SetBool(AniHashCode.isGround, frameData.isGround);  if (frameData.triggerDead)  {  this.animator.SetTrigger(AniHashCode.triggerDead);  }  this.playIndex++;  } |

FixedUpdate生命周期会固定时间间隔被调用。影子的运动完全由帧数据里记录的位置、旋转来实现。因为人物影子是复制人物结点的，所以人物影子也有人物动作状态机，对于这些状态机也同样通过帧数据里记录的控制变量值来控制。在播放方法的一开始，如果isAllowPlay为false或播放队列为空，或播放下标超出播放队列长度都会阻止播放。

接下来要去做的就是去收集这些帧数据，收集帧数据在人物脚本里完成。

|  |
| --- |
| void FixedUpdate()  {  if (this.isCollectFrameData)  {  this.shadowFrameData.pos = this.transform.position;  this.shadowFrameData.rot = this.transform.rotation;  this.shadowFrameData.isRun = this.isRun;  this.shadowFrameData.isJump = this.isJump;  this.shadowFrameData.horSpeed = System.Math.Abs(this.rigidBody.velocity.x);  this.shadowFrameData.vecHight =this.vecheight;  this.shadowFrameData.isGround = this.isGround;  this.shadowFrameDatas.Add(this.shadowFrameData);  this.shadowFrameData = new CharacterFrameData();  if (this.shadowFrameDatas.Count >= this.frameDataMaxSize)  {  this.isCollectFrameData = false;  }  }  } |

通过在人物脚本的FixedUpdate生命周期里，去收集人物相关的帧数据shadowFrameData然后将其置入队列shadowFrameDatas中。为防止其无限制收集帧数据导致内存溢出，还需设定一个收集的最大帧数量frameDataMaxSize。（isCollectFrameData变量用于控制收集帧数据与否）。‘

人物死亡setCharacterDead方法中：

|  |
| --- |
| // 注入最后一帧  this.shadowFrameData.pos = this.transform.position;  this.shadowFrameData.rot = this.transform.rotation;  this.shadowFrameData.isRun = this.isRun;  this.shadowFrameData.isJump = this.isJump;  this.shadowFrameData.horSpeed = System.Math.Abs(this.rigidBody.velocity.x);  this.shadowFrameData.vecHight = this.vecheight;  this.shadowFrameData.isGround = this.isGround;  this.shadowFrameData.triggerDead = true;  this.shadowFrameDatas.Add(this.shadowFrameData);  this.characterShadow.setCharacterFrameDatas(this.shadowFrameDatas.ToArray());  this.shadowFrameDatas = new List<CharacterFrameData>();  this.shadowFrameData = new CharacterFrameData();  this.isCollectFrameData = false; |

在人物死亡时，需要将最后一帧的数据收集完后将这个帧数据队列通过this.characterShadow.setCharacterFrameDatas方法传给人物影子脚本，此时人物影子脚本的播放帧数据会被暂停（isAllowPlay被置为false）。

|  |
| --- |
| public void setCharacterFrameDatas(CharacterFrameData[] frameDatas)  {  this.frameDatas = frameDatas;  this.isAllowPlay = false;  } |

直到人物复活时，影子脚本中绑定事件resetMissionEvent的方法reset被回调：

|  |
| --- |
| void resetShadow()  {  this.gameObject.SetActive(true);  this.playIndex = 0;  this.animator.CrossFade(AniHashCode.Idle, 0, 0, 0, 0);  this.isAllowPlay = true;  this.animator.ResetTrigger(AniHashCode.triggerDead);  } |

将播放下标重置为0，人物动作状态强制置为待机状态后才会去播放新获取到的队列。

收集帧数据、播放帧数据都是在FixedUpdate生命周期内完成的而不是Update生命周期。因为Update是每帧都会调用，与画面的帧率有关，所以当收集、播放时帧率差别较大时，会发生播放出来的画面抖动的情况，故收集、播放帧数据操作放在固定时间调用的FixedUpdate生命周期里较为合适。

## 4.2 摄像机的相关实现

### 4.21 正交摄像机的屏幕适配

2D游戏使用的是正交摄像机来渲染，它能使照出来的场景物体不会近大远小，呈现平面的效果。正交摄像机有个Size的属性，它决定了正交摄像机照摄框的高度，而照摄框的宽度则是Unity会根据当前屏幕去自动计算的。为了能使游戏适配任意尺寸的屏幕，正交摄像机的照摄框的大小应与屏幕的大小一致，因而我们需要根据当前屏幕的分辨率去动态调整正交摄像机的size属性。为此，我们需要先给摄像机绑定一个脚本CharacterCamera.cs。

在Start生命周期中：

|  |
| --- |
| this.characterCamera.orthographicSize = (float)Screen.height / 100 / 2; |

OnAwake是所有生命周期中首先被调用的，物体被激活就会被调用。通过Screen.height获取到屏幕的垂直分辨率，除以100化为Unity单位，100为每单元的像素数（在各图片资源的属性处设置），因为size为照摄框高度的一半因而还需除以2。这样就能把正交相机的照摄框大小设置得和手机屏幕大小一致了，实现了手机屏幕的适配。

### 4.22 摄像机活动区域限制

游戏中，摄像机会跟随人物运动，和人物活动范围限制一样，摄像机的运动同样有区域的限制，需要根据上一节所设置的正交摄像机Size属性去规定摄像机的活动范围。

在Start生命周期中：

|  |
| --- |
| this.startPosLimitX = this.deathLineLeft.transform.position.x + (float)Screen.width / 100 / 2;  this.endPosLimitX = this.deathLineRight.transform.position.x - (float)Screen.width / 100 / 2;  this.maxLimitY -= this.characterCamera.orthographicSize - defaultSize;  this.minLimitY += this.characterCamera.orthographicSize - defaultSize; |

startPosLimitX和endPosLimitX是摄像机水平方向的活动限制，通过左右两边空气墙的位置再减去/加上摄像机照摄框宽度的一半就可以确定。而maxLimitY和minLimitY是摄像机垂直方向的活动限制，它们的初始值在场景里确定后通过Inspector窗口填入到脚本，因为摄像机照摄框的高度是根据屏幕高度去适配的，所以还需在初始值的基础上加上/减去摄像机调整好的size值与初始size值的差值。

### 4.23 摄像机人物跟随

在确定了摄像机的size、活动范围限制后，摄像机跟随还需确定每帧摄像机离人物的相对位置。

在Start生命周期中：

|  |
| --- |
| this.cameraCharacterDistanceX = (0.5f - 0.382f) \* this.characterCamera.orthographicSize \* 2;  this.cameraCharacterDistanceY = this.characterCamera.transform.position.y - this.character.transform.position.y; |

cameraCharacterDistanceX为人物到摄像机的水平距离，因为人物要呈现在屏幕的水平黄金比例位置处，所以人物和摄像机的水平距离为（0.5 – 0.382）\* 摄像机照摄范围宽度。CameraCharacterDistanceY为人物摄像机的垂直距离，在场景中摆好摄像机的高度，然后再代码中计算出人物和摄像机的垂直距离差值即可。

之后，在Update生命周期中：

|  |
| --- |
| void Update()  {  float nextX = this.character.transform.position.x + this.cameraCharacterDistanceX;  float nextY = this.cameraCharacterDistanceY + this.character.transform.position.y;  float deltaMoveX;  float deltaMoveY;  if (nextY <= this.minLimitY)  {  deltaMoveY = this.minLimitY - this.transform.position.y;  }  else if (nextY > this.maxLimitY)  {  deltaMoveY = this.maxLimitY - this.transform.position.y;  }  else  {  deltaMoveY = nextY - this.transform.position.y;  }  if (nextX <= this.startPosLimitX)  {  deltaMoveX = this.startPosLimitX - this.transform.position.x;  }  else if (nextX > this.endPosLimitX)  {  deltaMoveX = this.endPosLimitX - this.transform.position.x;  }  else  {  deltaMoveX = nextX - this.transform.position.x;  }  this.transform.Translate(new Vector2(deltaMoveX, deltaMoveY));  } |

每帧检查这帧若置到之前算好的人物相对位置上是否超过摄像机活动边界，若超出则将视角固定在边界上，否则就将摄像机移动人物的相对位置上。（注：摄像机的移动必须要通过transform.translate方法，直接修改摄像机的坐标会导致画面无法渲染）

## 4.3 陷阱、怪物的相关实现

### 4.31 尖刺陷阱的实现

尖刺陷阱是由一个个刺排列而成，当人物走进时，尖刺会突出或者掉落。先需要给单个尖刺结点创建一个scale缩放从小变大的动画来模拟尖刺突出以及相应的动画状态机控制。之后，给尖刺陷阱结点绑定两个矩形碰撞器组件，第一个碰撞器组件和陷阱同样大小用于陷阱的实体体积，第二个用于范围检测人物靠近，后者必须启用isTrigger。碰撞器的isTrigger打上勾后，不会和其他碰撞器发物理碰撞效果，也就是说人物能穿过第二个碰撞器同时能触发陷阱trigger回调让陷阱察觉人物的靠近。接下来为陷阱创建脚本控制SpikeTrap.cs。

在脚本的trigger回调中：

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.transform.name == "characterCollider")  {  switch (this.trapType)  {  case SpikeTrapType.Motionless:  this.character.setCharacterDead();  break;  case SpikeTrapType.Hiden:  if (!this.colliders[1].enabled)  {  this.character.setCharacterDead();  return;  }  this.colliders[1].enabled = false;  for (int i = 0; i < this.transform.childCount; i++)  {  var obj = this.transform.GetChild(i).gameObject;  obj.SetActive(true);  var animator = obj.GetComponent<Animator>();  animator.SetTrigger(AniHashCode.triggerPopUp);  }  break;  case SpikeTrapType.DropDown:  if (!this.colliders[1].enabled)  {  this.character.setCharacterDead();  return;  }  this.colliders[1].enabled = false;  var rigidBody = this.gameObject.GetComponent<Rigidbody2D>();  rigidBody.gravityScale = 1;  break;  }  }  } |

当有物体和范围检测碰撞器发生碰撞时，因为范围检测碰撞器启用了trigger所以会回调脚本里的OnTriggerEnter2D方法。在这个方法体里，首先会判断什么物体进入了碰撞体内，如果是人物就会根据陷阱被设定的行为方式进行不同的处理。如果陷阱被设定为静止Motionless，就没有范围检测碰撞器去检测人物靠近故人物碰到就必定是陷阱实体碰撞器，调用setCharacterDead方法置人物死亡。如果陷阱被设定为隐藏Hiden，先判断范围检测碰撞器是否启用，若未启用要遍历陷阱下所有的尖刺结点去调用冒出的动画，之后将范围检测碰撞器关掉，否则说明尖刺已冒出，人物触发的是实体碰撞器，置人物死亡。如果陷阱被设定为掉落DropDown，处理和Hiden相似，唯一区别在于当检测到人物靠近时不是调用动画使尖刺冒出，而是将陷阱所受重力倍数从0改为1，使陷阱在重力影响下掉落。

所有游戏中会动态改变位置、旋转或其它状态的场景物体都必须有个reset方法绑定resetMission事件。人物死亡重生时，会调用绑定resetMission的所有方法来重置场景。尖刺陷阱也是如此，需要提供一个reset方法。在reset方法，根据尖刺陷阱被设定的行为模式去恢复状态，若是隐藏型尖刺陷阱，就将每一根刺通过动画状态机将其会回到Idle状态，并将范围检测碰撞器打开。若是掉落型尖刺陷阱，就将其刚体的重力设回为0，速率设为0，位置设为初始位置，并将范围检测碰撞器打开即可。

### 4.32 巡航锯轮的实现

先用Unity动画编辑器为锯轮创建一个旋转的动画，让其在游戏启动时就一直旋转。之后，锯轮按照设定会根据指定路线来回移动，所以需要创建一个锯轮的脚本Saw.cs，在脚本里提供一个路线点队列。在场景中拖动锯轮结点来获取关键点的坐标并按路线顺序填入到队列中。

在Start生命周期中：

|  |
| --- |
| if (this.transform.parent)  {  for (int i = 0; i < this.routePoints.Length; i++)  {  this.routePoints[i] = this.transform.parent.TransformPoint(this.routePoints[i]);  }  } |

因为在场景编辑器里显示并填入路线点队列的坐标是enemy结点下的局部坐标（在场景中我将所有陷阱、怪物都放在enemy结点下），在之后控制锯轮按照路线点队列移动时使用的是世界坐标，所以必须要在脚本启动的一开始将队列中的所有坐标都转换为世界坐标，TransformPoint方法能将任意结点下的本地坐标转换为世界坐标。

在Update生命周期中:

|  |
| --- |
| void Update()  {  switch (this.sawType)  {  case SawType.DirectByRoute:  var direct = (this.routePoints[routeTarget] - (Vector2)this.transform.position).normalized;  var delta = direct \* this.speed \* Time.deltaTime;  var nextPos = (Vector2)this.transform.position + delta;  var nextDirect = (this.routePoints[routeTarget] - nextPos).normalized;  // 到达目标点  if (nextDirect != direct || nextDirect == Vector2.zero)  {  this.transform.Translate(this.routePoints[routeTarget] - (Vector2)this.transform.position, Space.World);  if (this.routeTarget == this.routePoints.Length - 1)  {  this.routeTarget--;  this.isForward = false;  }  else if (this.routeTarget == 0)  {  this.routeTarget++;  this.isForward = true;  }  else  {  if (this.isForward)  this.routeTarget++;  else  this.routeTarget--;  }  }  else  {  this.transform.Translate(delta, Space.World);  }  break;  }  } |

在Update中每帧都先计算当前点到队列中目标点的方向向量direct，根据direct、移动速度、两帧间隔时间算出这帧的移动差值向量delta，根据当前坐标、delta算出的这帧移动好的坐标nextPos，以及当前点到nextPos的方向向量nextDirect。之后判断两方向向量是否值相等，如果不相等则说明这帧会到达当前目标点，故需要将锯轮移动到当前目标点位置并把当前目标点设定为队列中的下个点。isForward用来表示当前取队列下一点的方式是从前往后还是从后往前。若当前目标点已经是队列末尾并且isForward为true，则当前目标点设定为前一个点并把isForward改为false，若当前目标点已经是队列开头并且isForward为false，则将当前目标点点设定为后一个点并把isForward改为true。若两方向向量的值相等则表示这一帧没有到达当前目标点，将锯轮移动到nextPos位置即可。

这里锯轮的移动用的是世界坐标而不是本地坐标是因为，游戏中锯轮一直处于旋转状态故本地坐标的方向一直在变化所以不能使用本地坐标来移动。

锯轮提供的reset方法内只需将其设为初始位置并且当前目标设为队列中第一个点即可。

### 4.33 怪物ShootMace的实现

在设计中，ShootMace平常会上下浮动在空中，因而需要为先ShootMace制作一个上下浮动的循环动画。ShootMace在形象结点外需要有个空结点来包裹它，根结点作用怪物移动，子结点绑定浮动动画。当人物靠近ShootMace时，ShootMace会有一个慢慢上升的启动过程。

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.name == "characterCollider")  {  switch (status)  {  case ShootMaceStatus.sleeped:  this.animator.SetTrigger(AniHashCode.triggerAwake);  this.status = ShootMaceStatus.Awaking;  break;  }  this.isInRange = true;  }  } |

当人物触发范围检测触发器时，会先检查怪物状态，若怪物处于沉睡状态则先将其唤醒把状态置为Awaking，并把isInRange设为true表示人物在其攻击范围内。

在Update生命周期内：

|  |
| --- |
| case ShootMaceStatus.Awaking:  var direct = (this.awakeMoveTarget - (Vector2)this.transform.position).normalized;  var delta = direct \* this.awakeMoveSpeed \* Time.deltaTime;  var nextPos = (Vector2)this.transform.position + delta;  var nextDirect = (this.awakeMoveTarget - nextPos).normalized;  // 到达目标点  if (nextDirect != direct)  {  this.transform.Translate(this.awakeMoveTarget- (Vector2)this.transform.position, Space.World);  this.status = ShootMaceStatus.Awaked;  }  else  {  this.transform.Translate(delta, Space.World);  }  break; |

Update中每帧都会检查当前ShootMace的状态，当前状态为Awaking时，就用和之前巡航锯轮一样的定点移动方式将怪物移动至目标点。

|  |
| --- |
| case ShootMaceStatus.Awaked:  if (isInRange && this.shootLeftTime <= 0)  {  this.shootLeftTime = this.shootInterval;  GameObject spike = (GameObject)Instantiate(this.prefabSpike, this.transform.position, this.transform.rotation, this.transform);  }  else  {  this.shootLeftTime -= Time.deltaTime;  } |

当检查到当前状态为Awaked时，就要去判断下次攻击剩余时间是否小于等于0，若是则实例化一个子弹，并把攻击剩余时间重置。

子弹也需要一个脚本来控制其发射。

|  |
| --- |
| public void shoot()  {  var direct = (this.character.transform.position - this.transform.position).normalized;  this.rigidBody.AddForce(this.force \* direct, ForceMode2D.Impulse);  this.rigidBody.AddTorque(1000);  } |

在Start生命周期里立即就调用shoot方法使用物理引擎的冲量将子弹射击出去。子弹生成过一段时间之后需要销毁，以此来实现子弹的射程。如下代码：

|  |
| --- |
| void Update()  {  if (Time.time >= this.existTime)  {  Destroy(this.gameObject, 1);  }  } |

除此之外，ShootMace还可以开启追踪人物。

|  |
| --- |
| if (this.isTrace)  {  int traceDirect = 0;  if (this.character.transform.position.x - this.transform.position.x > 1)  {  traceDirect = 1;  }  else if (this.character.transform.position.x - this.transform.position.x < -1)  {  traceDirect = -1;  }  var dist = this.traceSpeed \* Time.deltaTime \* traceDirect;  var nextPointX = this.transform.position.x + dist;  if (nextPointX < this.traceLeftLimitX)  {  nextPointX = this.traceLeftLimitX;  }  else if (nextPointX > this.traceRightLimitX)  {  nextPointX = this.traceRightLimitX;  }    this.transform.Translate(new Vector2(nextPointX - this.transform.position.x, 0), Space.World);  } |

在Update中，每帧都去检查人物距ShootMace的水平方位（左侧还是右侧），然后这帧让ShootMace根据所得的方位以及设定的追踪速度去移动。怪物的追踪不是全图的，所以还需设定两个水平位置去限制怪物移动。

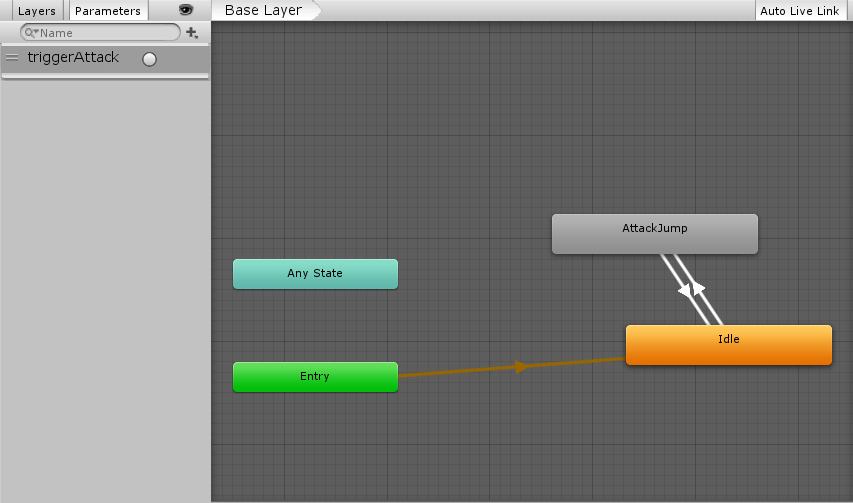
|  |
| --- |
| void OnTriggerExit2D(Collider2D collider)  {  if (collider.name == "characterCollider")  {  this.isInRange = false;  }  } |

当人物离开ShootMace的检测范围时，需要将isInRange置为false，不再追踪。

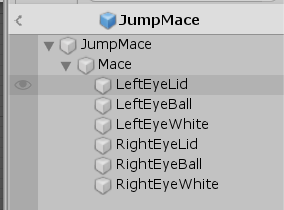
ShootMace的reset方法中，需要将ShootMace的标记状态设回sleeped，动画状态设回Idle，位置设为初始位置。

### 4.34 怪物JumpMace的实现

JumpMace常态会处于一跳一跳的状态下，当人物进入攻击范围时，会高跳然后落下去踩人物。先完成JumpMace动画状态机，如图：



为实现JumpMace能一边跳一边移动，甚至能高跳到人物头顶。JumpMace需要如图的结点结构：



和ShootMace的结点结构一样，JumpMace根节点用来移动怪物，取名Mace的形象结点上绑定动画控制器在局部空间内实现动画。

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.transform.name == "characterCollider")  {  this.isInRange = true;  this.moveSpeed = this.attackSpeed;  }  } |

当人物被JumpMace看到（触发区域检测碰撞器）时，同样将isInRange置为true，并把移动速度设置为更快的attackSpeed。

在Updata生命周期中：

|  |
| --- |
| int status = this.animator.GetCurrentAnimatorStateInfo(0).shortNameHash;  int trans = this.animator.GetAnimatorTransitionInfo(0).nameHash;  if (this.isInRange)  {  if (status == AniHashCode.Idle && trans != AniHashCode.IdleToAttackJump)  {  this.attackTargetPos = new Vector2(this.character.transform.position.x, this.routePoints[0].y);  this.animator.SetTrigger(AniHashCode.triggerAttack);  }  target = this.attackTargetPos;  }  else  {  target = this.routePoints[this.routeTarget];  } |

JumpMace的寻路方式也是用的定点移动算法，只不过当人物在其视线范围内时，寻路方式会有少许的不同。JumpMace的实现里怪物的状态不是由枚举来标注，而是通过动画状态机。通过GetCurrentAnimatorStateInfo方法来获取动画状态机当前所在的状态，以及GetAnimatorTransitionInfo方法来获取当前的正在进行的转换边。当人物在视线范围内时，判断当前状态是在Idle状态并且不在进行切换到攻击状态的转换中时，将定点寻路的目标点改为人物的坐标并将动画状态切换到攻击状态，这样怪物会一边高跳一边去追赶人物。

|  |
| --- |
| void OnTriggerExit2D(Collider2D collider)  {  if (collider.transform.name == "characterCollider")  {  this.isInRange = false;  this.moveSpeed = this.iniMoveSpeed;  }  } |

同样，当人物离开JumpMace的视线范围时，会将isInRange置为false，怪物移动速度恢复为平常速度。此时，在Update中，因为isInRange为false，怪物也会将目标点置回触发追踪前记录的目标点。这样，人物一旦逃脱了怪物的追赶，怪物就会回到原来的位置。

JumpMace的reset方法需要将JumpMace刚体的速度和角速度都设为0（因为JumpMace设定还会受到重力影响并且会跌落悬崖）防止其恢复状态时会翻滚，位置和旋转都设为初始值，将寻路目标设为队列中第一个。

### 4.35 怪物Mace的实现

Mace的实现和前面两种的实现思路大同小异。Mace拥有三个碰撞器：唤醒范围碰撞器、攻击范围碰撞器，怪物实体碰撞器。

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.transform.name == "characterCollider")  {  switch (this.status)  {  case MaceStatus.Untriggered:  this.status = MaceStatus.InTrace;  this.triggerCollider.enabled = false;  break;  case MaceStatus.InTrace:  this.status = MaceStatus.InAttack;  break;  }  }  } |

因为三个碰撞器都是一层层包含嵌套的，所以人物进入哪个碰撞器可以用当前状态来区分。人物触发碰撞器时，检查当前状态。如果当前状态为Untriggered状态时，人物必定进入的时最外层的唤醒范围碰撞器，此时将怪物状态修改为InTrace，并把唤醒范围碰撞器关掉（因为一旦唤醒，Mace就会全图追踪）；如果当前状态为InTrace，那么人物触发的一定是攻击范围碰撞器，因而将怪物状态修改InAttack表示进入攻击。

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.transform.name == "characterCollider")  {  switch (this.status)  case MaceStatus.InTrace:  this.status = MaceStatus.InAttack;  break;  }  }  } |

当人物离开攻击范围时，怪物需要恢复InTrace状态。

Update生命周期中：

|  |
| --- |
| void Update()  {  switch (this.status)  {  case MaceStatus.InTrace:  var direct = (this.character.transform.position - this.transform.position).normalized;  this.transform.Translate(direct \* this.traceSpeed \* Time.deltaTime);  break;  case MaceStatus.InAttack:  var direct2 = (this.character.transform.position - this.transform.position).normalized;  this.transform.Translate(direct2 \* this.attackSpeed \* Time.deltaTime);  break;  }  } |

和JumpMace一样，Mace追踪玩家通过每帧里计算和玩家的方向向量，再根据方向向量、设定的速度来移动。Mace的攻击方式为对攻击范围内的人物使出快速冲撞，这个的实现其实和追踪的实现是一样的，只不过冲撞的速度会设定得更大一点。

Mace的reset方法需要将位置置为初始值，状态置回Untrigger，并把最外层的范围检测碰撞器重新打开。

### 4.36 陨石陷阱的实现

当人物踩到云朵或者地面时，可能会触发陨石下落。

|  |
| --- |
| case CloudType.CallMeteor:  if (!this.isMeteor)  {  var meteor = Instantiate(this.meteorPrefab, this.meteorPos, Quaternion.Euler(0, 0, 0));  Meteor meteorScript = meteor.GetComponent<Meteor>();  meteorScript.target = this.character.transform.position;  meteorScript.impulse = this.meteorImpulse;  meteorScript.isShootMeteor = true;  this.isMeteor = true;  }  break; |

isMeteor记录陨石是否已经掉落。人物踩到设定有CallMeteor行为方式的地面或云朵时，会通过预制在指定位置生成一个陨石，将设定的陨石下落的相关参数传给动态生成的陨石的脚本。

|  |
| --- |
| var direct = (this.target - this.transform.position).normalized;  this.rigidBody.AddForce(direct \* this.impulse, ForceMode2D.Impulse);  this.cameraAnim.SetBool(AniHashCode.triggerEndShake, false);  this.cameraAnim.SetTrigger(AniHashCode.triggerShake); |

在陨石脚本Meteor.cs中计算出陨石到人物的方向向量，利用物理引擎往该方向向量处应用一个冲量，这样陨石就能从空中下落到触发时人物的位置。此外，当触发陨石时，还需有个摄像机抖动的效果。因而，需要给摄像机创建一个动画状态机。当生成陨石时，同样在陨石脚本通过摄像机状态机的控制变量triggerShake控制其抖动。当陨石撞到物体时，摄像机的抖动就应停止。

动态生成的陨石也需要提供reset方法来置回摄像机Idle动画状态，以及销毁本身。

### 4.37 减速/加速球陷阱的实现

减速/加速球就是对在其内部的人物、子弹施加速度上的影响。

|  |
| --- |
| void OnTriggerEnter2D(Collider2D collider)  {  if (collider.name == "characterCollider")  {  this.isInRange = true;  this.character.isReleaseSpeed = true;  this.rigidbodies.Add(this.characterRigidBody);  }  else if (collider.name == "bullet")  {  this.isInRange = true;  var rigidbody = collider.GetComponent<Rigidbody2D>();  if (rigidbody != null)  this.rigidbodies.Add(rigidbody);  }  } |

每次有人物、子弹进入到碰撞器时，需要将这些物体的刚体加入到rigidbodies列表中。人物因为有速度的限制，所以还需通过设定isReleaseSpeed为true将人物速度解放，以达到在高速球中人物能高速运动的目的。同样，当人物、子弹离开碰撞器时，需要将这些物体的刚体从rigidbodies列表中移除，如果是人物的话还需将isReleaseSpeed设回false

在Update生命周期中：

|  |
| --- |
| void Update()  {  if (this.isInRange)  {  foreach (Rigidbody2D rigidbody in this.rigidbodies)  {  if (rigidbody != null)  rigidbody.velocity \*= this.speedRate;  }  }  } |

每帧去遍历rigidbodies列表，将列表中每一个刚体的速率都乘以设定好的倍数。这样就实现了加速/减速球陷阱。

加速、减速陷阱因为没有状态变化，故不需要提供reset方法。

### 4.38 怪物SniperMace的实现

SniperMace设定是一个能够远距离狙击的怪物。和之前的怪物一样，它通过范围检测碰撞器去检测狙击方向上是否有人物。SniperMace每次射击前都会有一个射击动作，所以需要给SniperMaces建立一个动画状态机。将射击的shoot方法绑定在射击动画的末尾，只有当动作做完了才会回调射击方法。SniperMace和ShootMace一样都有射击间隔，当射击间隔一到就会去播放射击动作。SniperMace射出的子弹也会受减速/加速球陷阱影响，所以需要将射出的子弹命名为bullet。SniperMace的实现方式和之前的ShootMace很相似，所以代码就不粘贴出来了。

### 4.39 塌陷地面陷阱的实现

塌陷地面的实现非常简单，给地面绑定刚体，当人物踩上去触发碰撞回调时，在回调里将地面刚体所受的重力调大即可。

塌陷地面的reset方法中需将重力改回初值，并将地面置回原来的位置。

## 4.4 场景相关的实现

### 4.41 地面的实现

每块地面绑定碰撞器组件，然后在场景中一块一块铺设，这样人物就可以在这些地面上行走。但有些地面存在坡度，这样的地面如果不修改作用在人物身上的力的话就没有办法上去。因而，需要给每块地面都挂一个脚本，在这个脚本里去指定人物走在这个地面上应该施加的力，相关代码在人物运动章节有提及。此外，这个脚本还可以实现一些地面按定点导航移动、召唤陨石、塌陷等陷阱功能。

### 4.42 场景切换

当人物到达场景的右空气墙，或者按下ui界面的上/下一关按钮时，需要进入到上/下一关。为了能有更好的切换关卡表现，我使用的是异步加载方式去加载场景。异步加载能使记载场景的过程在后台进行，原来场景仍会保留在前台。在空气墙绑定的脚本DeathLine.cs中：

|  |
| --- |
| public void loadNextScene()  {  if (this.isLoading)  return;  this.asyncOperation = SceneManager.LoadSceneAsync(this.nextScene);  this.asyncOperation.allowSceneActivation = false;  this.isLoading = true;  this.touchBlock.gameObject.SetActive(true);  } |

当触碰到右空气墙时，就会调用loadNextScene方法，在这里进行异步加载。将allowSceneActivation设为false，这样当异步加载完就不会立即进入下一个场景。

|  |
| --- |
| if (!this.isLoading)  return;  if (this.asyncOperation.progress == 0.9f && this.touchBlock.color.a == 1f)  {  this.asyncOperation.allowSceneActivation = true;  }  else if (this.touchBlock.color.a < 1)  {  if (this.touchBlock.color.a + this.fadeInSpeed > 1)  {  this.touchBlock.color = Color.black;  }  else  {  var color = this.touchBlock.color;  this.touchBlock.color = new Color(color.r, color.g, color.b, (float)(color.a + this.fadeInSpeed));  }  } |

在Update中，检查到isLoading为true时正在加载时，就要修改遮罩透明度让遮罩逐渐变黑，当遮罩透明度为1并且加载进度到达0.9时，修改allowSceneActivation为true启动切换场景。同时，启动那个场景一开始需要用黑色遮罩覆盖，然后用同样方法将将黑色遮罩透明度慢慢变为0。这样就实现了切换场景时淡入淡出的效果。