上海大学

SHANGHAIUNIVERSITY

毕业设计（论文）

UNDERGRADUATEPROJECT(THESIS)

题目:基于 Android 平台的游戏设计与实现

学院	计算机工程与科学学院
专业	计算机科学与技术
学号	15123125
学生姓名	陈天瑜
指导教师	陈圣波
起讫日期	2019.02.25 – 2019.06.07

目录

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 脚本引擎，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各个动画状态间的转换，使动画看起来栩栩如生。

使用 Unity 游戏引擎的物理引擎，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 脚本引擎、动画系统、物理引擎、碰撞检测系统、场景编辑器、UGUI、Android Build Support

Design And Implementation Of The Mobile Game Based On Android Platform

ABSTRACT

The continuous development and wide spread of the smartphone are creating an explosion of demand on mobile game. Mobile game has looted the game market which computer game and console game occupied by its portability, amusement and the ability of using time fragment. Up to now, mobile game has become an indivisible part in our mobile life. Mobile game as a kind of generally recognized cultural pattern is receiving tremendous attention. That is, it will be of great value for study to design and implement a mobile game based on Android Platform.

I will develop the game with Unity Engine for its maturity and less bug on bottom implementation. What's more, Unity Engine provides abundant functional units: Frame Rendering, Physical Engine, Collision Detection System, Sound Effect, Script Engine, AI, Network Engine and Scene Management, which simplifies the development process and reduce the development difficulty greatly. In development, we need to learn to use these functional units.

In development, through MonoBehaviour Script Engine, hanging the C# Script on the game object, we can implement the game control logic in MonoBehaviour lifecycle.

Through Unity Animation System, we can editor role, trap, monster animation with a Animation Editor, binding with Animation State Machine to control animation transition which makes animation more natural.

Through Unity Physical Engine, binding game object with Rigidbody Component, we can invoke rigidbody's physical API to implement some functions such as role movement, trap dropping, bullet shooting.

Through Unity Collison Detection System, binding game object with Collision Component, we can implement some collision call-back, such as triggering trips. When collision happen, the call-back will be invoke。

Through Unity Scene Editor, we can build the scene with visible view.

Through UGUI, we can implement and adaptive UI conveniently.

Through Unity Android Build Support, we can build a Android APK which run

on the Android Platform easily.

With Unity functional units above, this project will implement all game functions such as role animation and movement, camera follow, trap, monster, scene build and switch, to develop a perfect mobile game.

Keywords: development of mobile game, Unity Engine, MonoBehaviour Script Engine, Animation System, Physical Engine, Collision Detection System, Scene Editor, UGUI, Android Build Support

第一章 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[2]，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[3]。

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[4]。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[5]。

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[7]。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[6]。

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。

1.3.1 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。

1.3.2 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。游戏类型是类似马里奥的 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。

1.3.3 本文组织结构

本篇论文总共分为五章。

第 1 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 2 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 3 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 4 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 5 章对游戏成品进行截图展示。

第 6 章对全文进行总结，总结了使用 Unity 游戏引擎的开发工作流程、开发游戏遇到的难点以及个人心得体会。

第2章 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[8]。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[9]。

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 系统为例：

在系统变量下新建一个名为 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[10]）

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[10]）

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 Studio 的 Gradle 构建[11]，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）[12]。这样发布到 Android 平台的所有准备就全都做好了。

2.7 Sourcetree 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" ”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。

第3章 总体设计

这次开发的游戏我取名为 Red Man Adventure。要完成这个游戏，首先需要对游戏的各个组成部分进行总体的规划设计，以方便之后的实现。

在这个游戏设计中，整个游戏由人物、游戏场景、摄像机、怪物陷阱、UI 界面组成，如图 3-1 所示。

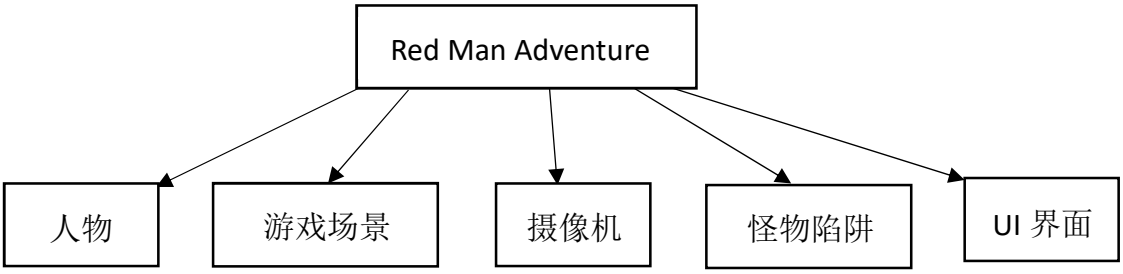

图 3-1 总体设计结构图

人物需要实现其形象、动作、运动、影子，如图 3-2 所示。

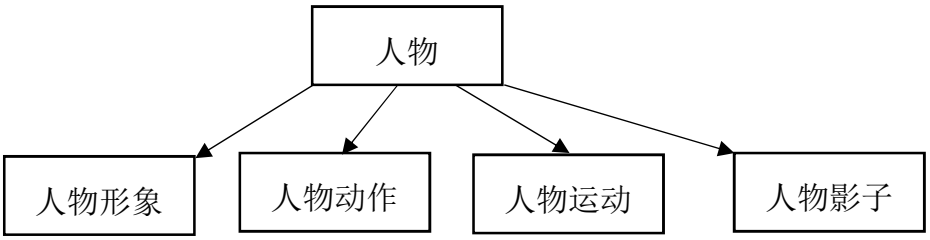

图 3-2 人物设计结构图

游戏场景需要实现视图布置、人物活动地面、场景限制区域、场景切换，如图 3-3 所示。

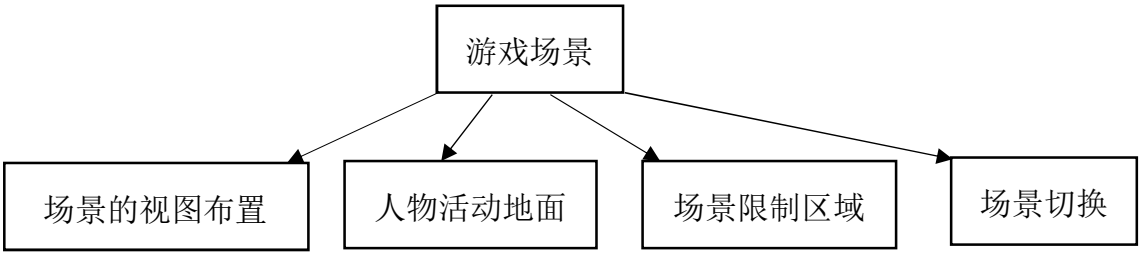

图 3-3 游戏场景设计结构图

摄像机需要实现区域限制、跟随、道具以及正交摄像机的屏幕适配，如图

3-4 所示。

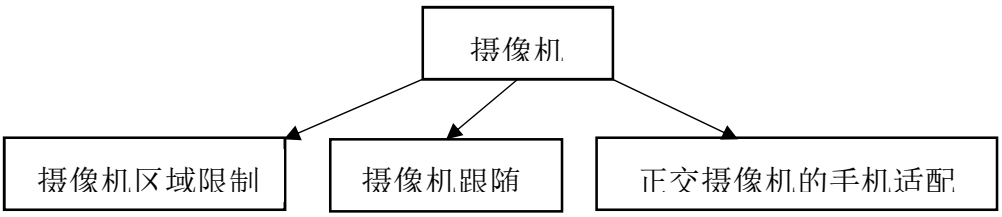

图 3.4 摄像机设计结构图

陷阱、怪物、道具需要实现触发人物死亡、人物复活时的重置、触发式唤醒、寻路方式、跟踪方式、射击方式、下落方式、影响人物运动速度以及中途存档点，如图 3-5 所示。

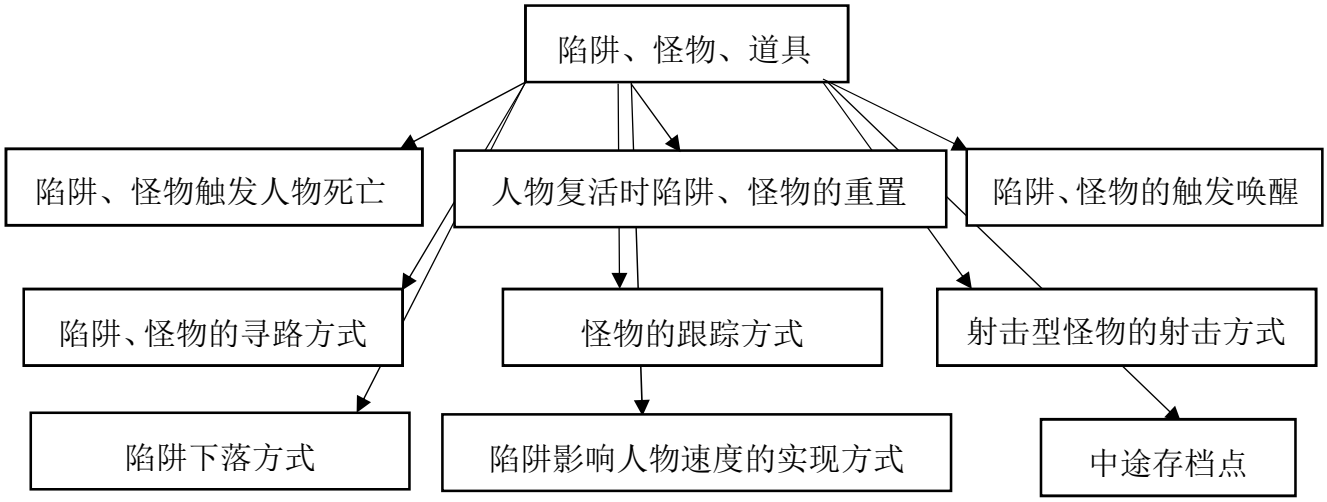

图 3-5 陷阱、怪物、道具设计结构图

UI 需要实现上、下一关键、人物控制虚拟按键、暂停键、开始页面，如图 3-6 所示。

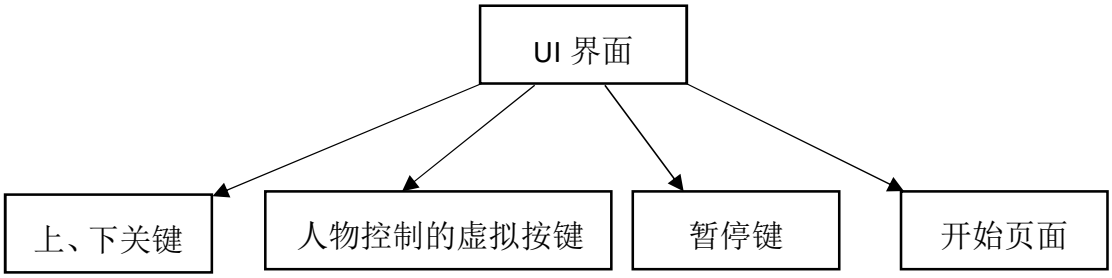

图 3-6 UI 界面设计结构图

3.1 人物设计

3.1.1 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。

3.1.2 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。

3.1.3 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。

3.1.4 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 摄像机区域限制

为避免摄像机照摄到非法区域而导致穿帮，摄像机的移动需要设定区域限制，摄像机的移动区域限制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 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。

3.3.3 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、怪物的寻路方式

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 陷阱的下落方式

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 陷阱影响人物速度的实现方式

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回调来实现。

第4章 详细设计与实现

上一章，对游戏各个功能模块进行了总体设计上的介绍，这章就开始深入去讲解这些功能模块的详细设计与实现。

4.1 人物的相关实现

4.1.1 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。

4.1.2 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 4-1 以及图 4-2 所示。

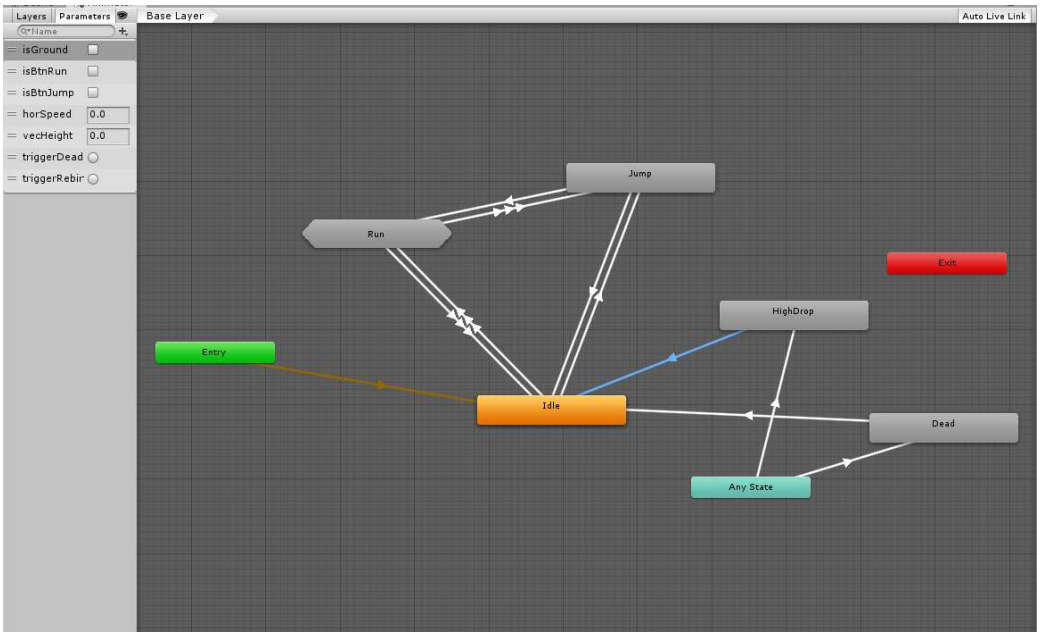

图 4-1 人物动作状态机图（外层）

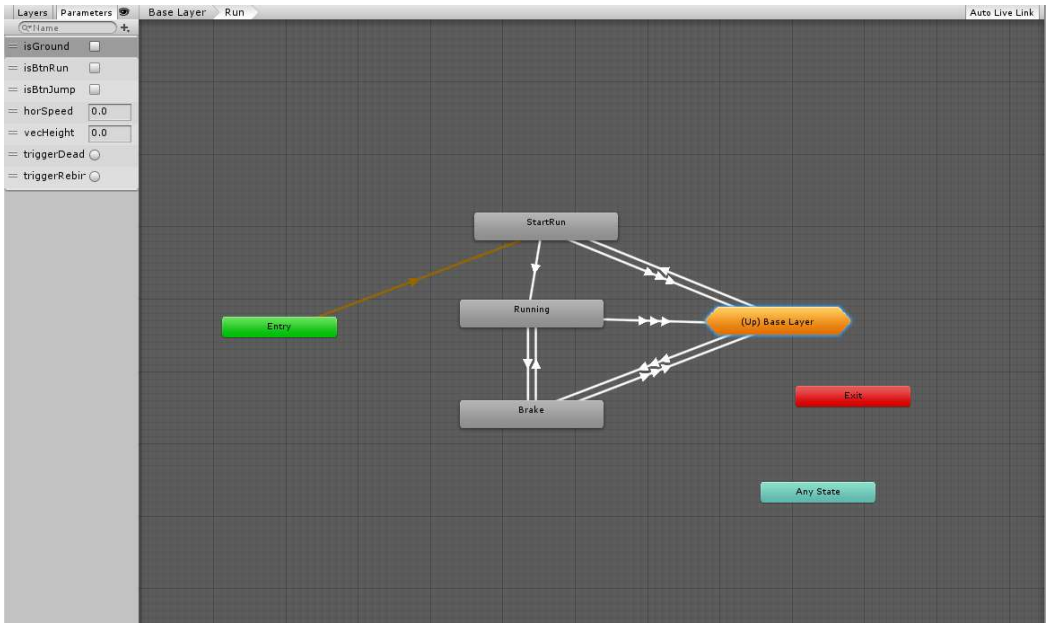

图 4-2 人物动作状态机图（Run 子状态机内部图）

4.1.3 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。

4.1.4 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。

4.1.5 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。

4.1.6 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。

4.1.7 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;
}
```

需要注意的是，人物旋转信息存的是四元数而不是欧拉角，是因为欧拉角转

出来的最终角度会根据 x,y,z 三轴旋转顺序的不同而不同，甚至当 y 轴转到 90 度会发生万向节锁的问题，而四元数则完全没有这种问题[14]。

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.ToArr
ay());
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

4.3.1 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。

4.3.2 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[13]。

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。

4.3.3 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。

4.3.4 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 4-3 所示。

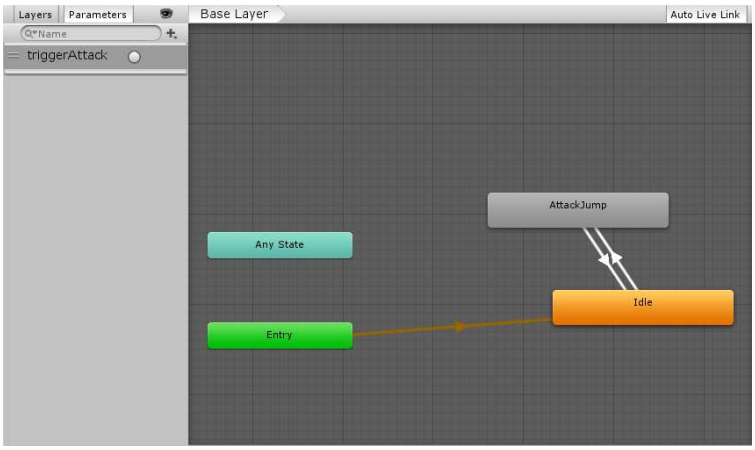

图 4-3 JumpMace 动作状态机图

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 4-4 的结点结构：

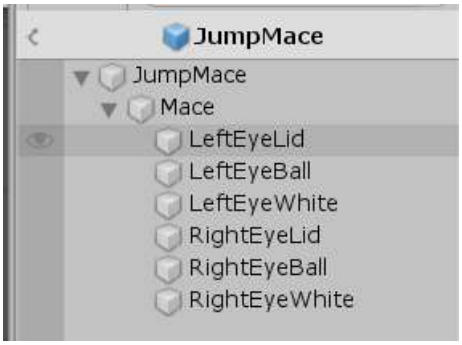

图 4-4 JumpMace 的结点层级

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。

4.3.5 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。

4.3.6 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。

4.3.7 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。

4.3.8 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。

4.3.9 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

4.4.1 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。

4.4.2 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。异步加载能使加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。

第 5 章 成品展示

本章节会通过游戏截图加文字说明的方式去展示游戏项目成品。因成品内容过多，所以这里就只展示游戏第一关的部分截图。

图 5-1 游戏启动页

如图 5-1 所示为游戏的启动页，打开游戏后看到的第一个页面。游戏标题 Red Man Advanture 的红色艺术字在中间漂浮，底下“点击任意处开始游戏”的白色字体若隐若现。

图 5-2 游戏开始

点击游戏开始，图 5-1 的红色标题就会飘离屏幕，同时如图 5-2 可以看到人物会从场景左边跑向屏幕，并用刹车的动作停了下来，人物的右侧还有个尖刺陷阱。当人物一进入屏幕时，界面的上的按钮就会显示出来，这时候玩家就可以按动操作键去控制人物。

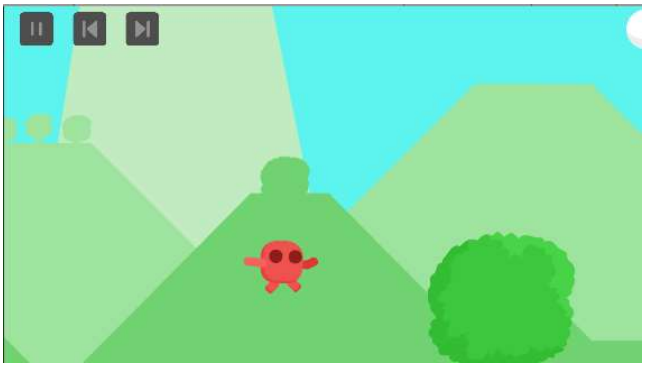

图 5-3 人物死亡

如图 5-3 为人物踩到尖刺陷阱死亡时的画面，人物动作转换到死亡动画的同时，还给人物刚体施加了一个垂直向上的冲量，让人物看起来因疼痛而飞到空中。

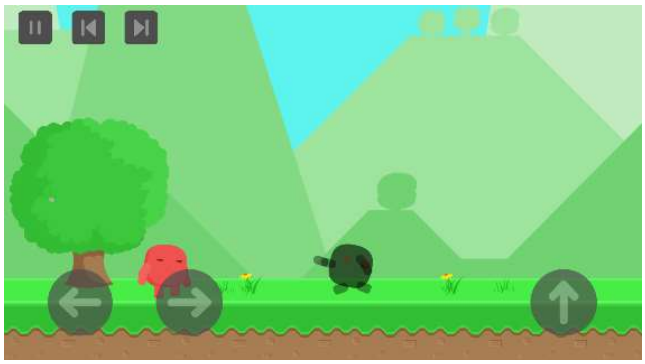

图 5-4 人物影子

如图 5-4 为人物死亡重生时的画面，人物在复活点重生并且复活点位置还生成了一个黑色影子重复人物上一把的行为。

图 5-5 地面塌陷

如图 5-5 为塌陷陷阱的画面，人物踩上去一大片地面产生塌陷，人物掉下去就会死亡

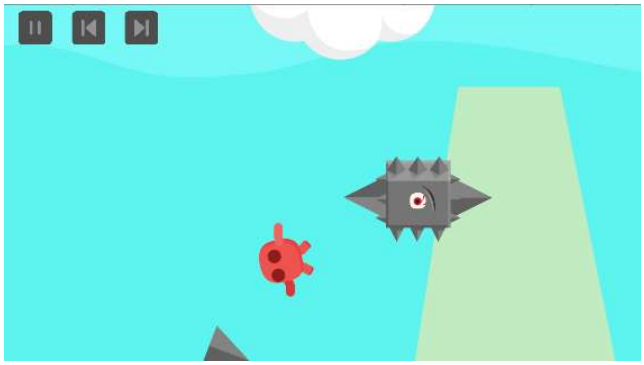

图 5-6 怪物 ShootMace

如图 5-6 为怪物 ShootMace 的截图，人物靠近时 SootMace 会慢慢上升，当上升到最高点时，ShootMace 就会像图中一样向人物射击。

图 5-7 中途存档点

如图 5-7 为存档点，因为缺少存档点的图片素材，所以存档点的形象就由宝箱来代替了。当人物触碰到存档点时，会将人物的复活点设为存档点位置。

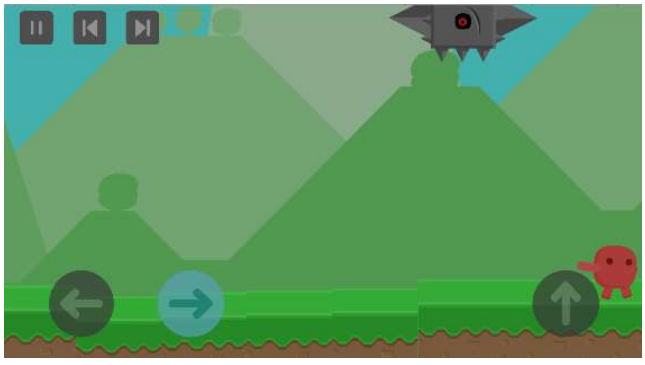

图 5-8 切换场景

如图 5-8 为切换场景，当人物抵达最右边时通关了本关，场景会慢慢变黑过渡到第二关。

第 6 章 毕设总结

6.1 开发流程总结

在经过了一个完整的游戏开发过程后，我将开发的总体流程总结归纳为如下：

（1） 搭建开发环境
（2） 构建一个 demo 场景，用作功能测试
（3） 实现人物的各种动画以及动画状态机
（4） 利用 Unity 物理引擎实现人物的运动以及控制
（5） 在人物运动的基础上，将人物动画状态机的控制加上去
（6） 实现摄像机的跟随
（7） 实现各种陷阱、怪物
（8） 实现人物影子
（9） 开始构建配置正式场景
（10） 实现游戏场景的切换
（11） 实现游戏开始页面
（12） 实现暂停、下/上一关按钮
（13） 测试修复 Bug

6.2 开发难点总结

在游戏开发过程中，我遇到了很多的难点。人物动画状态机的控制就是其中最大的一个难点，我一开始将人物状态机的状态切换控制变量都设为 trigger 类型，这样的话人物每次要切换动作都要在脚本里去判断该帧是否要去切换动作状态，并且还要设置 trigger 为 true 让状态机去转换，这样的做法不仅代码复杂，而且有时候状态切换时会重复置 trigger 变量导致人物动作切换异常。后面通过将人物状态机的控制变量改为用 int、float、bool 等表示人物状态的变量后才完美解决了。

如何使子结点的碰撞器触发父节点脚本中的碰撞回调也是我前期遇到的难点之一。当时，有些陷阱、怪物的子结点也会绑定碰撞器，也需要有碰撞的回调处理，但给子结点再绑定一个脚本的话就显得太多余了，后面经过实验才发现当父节点绑定刚体组件时，子结点的碰撞器才可以触发父节点脚本里的碰撞回调。

如何使人物倒下后还控制能再翻回来？因为人物的运动是物理引擎作用的，所以人物在运动时 z 轴会有旋转，这样难免人物会倒下，一开始我使用人物身体的形状作为人物的碰撞器，这样一旦人物倒下就很难再翻回来了。我也考虑过设置人物刚体，冻结其 z 轴旋转，但这样做的话，人物就无法上坡（因为上坡需要人物 z 轴旋转）。后来，我裁剪了一个平底圆形作为人物的碰撞体积才解决了这个问题。

人物影子播放时行为有偏差。人物影子的刚开始我在帧数据里记录的是玩家对按钮的操作，但游戏中帧率是有偏差的，在帧率不同的情况下执行相同帧数间隔相同的操作会导致两次的人物的位置不一样。于是，之后我将帧数据改成记录人物的位置、旋转以及动画动态机控制变量，但这又有了新的问题——人物影子播放时会有抖动。后面经过查阅了相关资料后我得知 Unity 物理引擎对于 transform 值的修改，只在 FixedUpdate 中才会应用，以此为根据，我推测，在 Update 中获取到受到物理引擎作用的坐标值可能是物理引擎计算的中间值，将这些中间值播放出来看起来就是抖动的。后来，我将收集人物帧数据、播放人物帧数据的操作放到了都放到了 FixedUpdate 中，这才顺利解决了人物影子抖动的问题。

6.3 心得体会

在经历了这次毕业设计后，我的收获颇多，感叹也颇多。

游戏开发的流程（从最开始的游戏需求制定，游戏总体设计到游戏各个功能模块的实现，游戏 demo 的制作再到后来游戏关卡、UI 界面的制作）以及 Unity 游戏引擎的基本使用（从场景的搭建、MonoBehavior 脚本的编写、动画的编辑、动画状态机的控制到碰撞器组件、刚体组件、物理引擎、UGUI 的使用再到 Android 工程的发布），这些无不是我在这次的游戏开发实践中学会的。

同时，Unity 游戏引擎的强大也让我感到惊叹，有了游戏引擎这个工具能够让游戏开发者不用再去关注图形的底层渲染，让原本极为复杂的游戏开发变得简单很多，这不禁让我对游戏引擎的图形底层渲染方面产生了浓厚的兴趣，Unity 是如何渲染出游戏这样如此精细复杂的画面的。

致谢

感谢陈圣波导师几个月来，从选题，构思到实践、定稿各个环节中给予我悉心的指导。

这份论文的完成意味着四年的大学学习生活已迎来终点，四年间在计算机学院我学到了很多、收获了很多，不光是计算机方面的知识，还有孜孜不倦求学态度、坚忍不拔的求学精神。感谢你们，所有教导过我的老师，所有陪伴过我的同学！

参考文献

[1] Oliver Rutz, Ashwin Aravindakshan, Olivier Rubel. Measuring and forecasting mobile game app engagement[J]. International Journal of Research in Marketing, 2019 (Article In Press):1-15.

[2] Shchiglik,Barnes,Scornavacca. The Development of an Instrument to Measure Mobile Game Quality[J]. Journal of Computer Information Systems, 2016, 56(2):97-105.

[3] 宋迪. 正在向移动端转移的游戏产业[J]. 中国传媒科技, 2013(15):101.

[4] 海川. 网络游戏悖论:高速增长与乱象丛生[J]. 新经济导刊, 2018(03):58-62.

[5] 刘晓. 移动端小游戏的发展与问题应对[J]. 传媒, 2018(23):50-52.

[6] Sung Lae Kim, Hae Jung Suk, Jeong Hwa Kang, Jun Mo Jung, Teemu H. Laine,Joonas Westlin. Using Unity 3D to Facilitate Mobile Augmented Reality Game Development[C]. In Proceedings of 2014 IEEE World Forum on Internet of Things (WF-IoT). Seoul, South Korea, 2014:21-26.

[7] Jingming Xie. Research on key technologies base Unity3D game engine[C]. In Proceedings of 2012 7th International Conference on Computer Science & Education (ICCSE), Melbourne, VIC, Australia, 2012:695-699.

[8] 赵懋骏. 移动游戏快速开发平台设计与实现[D].电子科技大学, 2016.

[9] 张博. 基于 Android 平台的手机游戏开发分析[J]. 信息与电脑(理论版), 2018, (20):79-82.

[10] 李婧. 基于 Android 平台的手机游戏设计与实现[D]. 东南大学, 2017.

[11] 李旭. 多平台移动游戏移植方案设计与实现[D]. 北京交通大学, 2014.

[12] 陈洁. 基于 Android 的手机游戏引擎的设计与实现[D]. 吉林大学, 2016.

[13] 庞钦存.四元数在游戏引擎 Unity3D 中的应用[J]. 产业与科技论坛, 2015, 14(17):63-64.

[14] 黄嘉伟. 基于 Unity 的 MMORPG 移动寻路系统的设计与实现[D]. 南京大学, 2018.

附录：部分源代码清单

[1] 人物运动、动作控制

```
void Update()
{
    if (this.isJump && this.rigidBody.velocity.y <= 0)
    {
        this.isJump = false;
    }

    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);
            }

        }
```

```
    }

    // 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;
}
```

[2] 设置人物死亡

```
public void setCharacterDead()
{
    if (this.isDeading)
    {
        return;
    }
    Handheld.Vibrate();
    this.rigidBody.AddForce(Vector2.up * 20, ForceMode2D.Impulse);
    this.isDeading = true;
    this.animator.SetTrigger(AniHashCode.triggerDead);
    this.leftTrigger.transform.parent.gameObject.SetActive(false);
    this.rigidBody.angularVelocity = 0;
    this.isRun = false;
    this.isJump = false;

    // 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.ToArr
ay());
    this.shadowFrameDatas = new List<CharacterFrameData>();
    this.shadowFrameData = new CharacterFrameData();
    this.isCollectFrameData = false;
}
```

[3] 人物复活，场景物体状态初始化

```
public void resetMission()
{
    this.rigidBody.velocity = Vector2.zero;
    this.rigidBody.angularVelocity = 0;
    this.isRightDir = true;
    transform.SetPositionAndRotation(this.characterIniPos,
this.characterIniRot);
    this.animator.SetTrigger(AniHashCode.triggerRebirth);
    this.leftTrigger.transform.parent.gameObject.SetActive(true);

    this.onBtnJumpUp();
    this.onBtnLeftUp();
    this.onBtnRightUp();

    this.resetMissionEvent();
    this.isDeading = false;
    this.isCollectFrameData = true;
}
```

[4] 收集人物帧数据

```
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;
        }
    }
}
```

[5] 人物影子帧数据播放

```
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++;
}
```

[6] 摄像机跟随

```
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));
}
```