摘要

随着硬件设备的不断发展，PC的硬件性能不断提升。人们除了使用PC来进行工作、听音乐、看视频之外，也会选择玩游戏来打发空闲时间。游戏市场的规模随之迅速扩张，游戏行业也得到了飞速的发展。而由于PC、智能手机的操作系统有很多种，一个游戏如果想在多个平台上运行，那么就需要针对每个平台进行独立开发，这样游戏的开发效率就会很低。因此，游戏厂商和游戏研发工程师们需要一款可以跨平台开发游戏的软件。Unity3D就是在这样的环境下出现的。Unity Technologies公司开发的这款游戏引擎能够实现将同一份游戏代码通过打包程序打包成不同平台的安装包或者可执行程序，从而实现跨平台开发游戏。

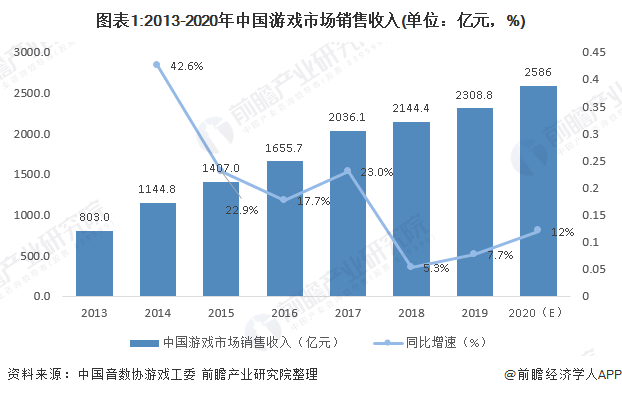
跑酷类游戏是比较常见的游戏题材，流程一般是玩家控制角色跨越重重障碍完成关卡。在此基础上还可以分为2D、3D、休闲、硬核等类别。比较典型的跑酷类游戏有天天酷跑、神庙逃亡、地铁跑酷等等。本文设计的游戏也选择了跑酷题材，玩家需要控制角色熟练通过各种障碍，如掉落的砖块、自动弹起的尖刺等等，拿取所有的宝珠从而过关。

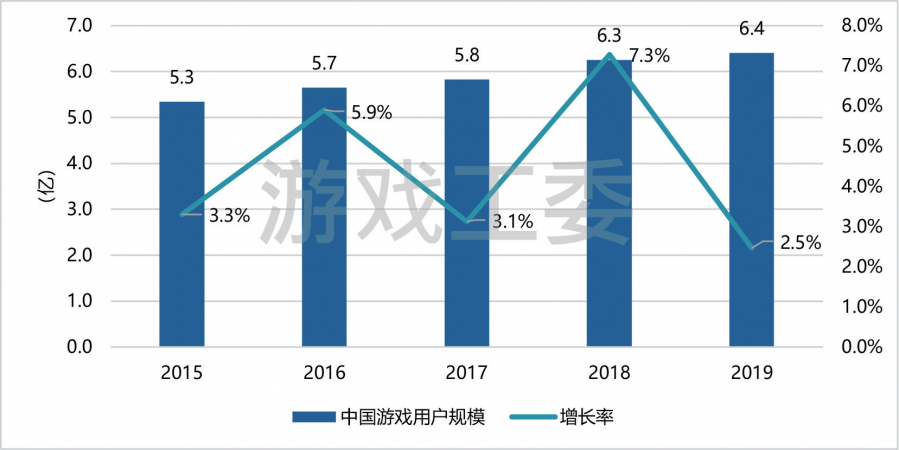
关键词：游戏，Unity，2D，C#

1.1研究背景及意义

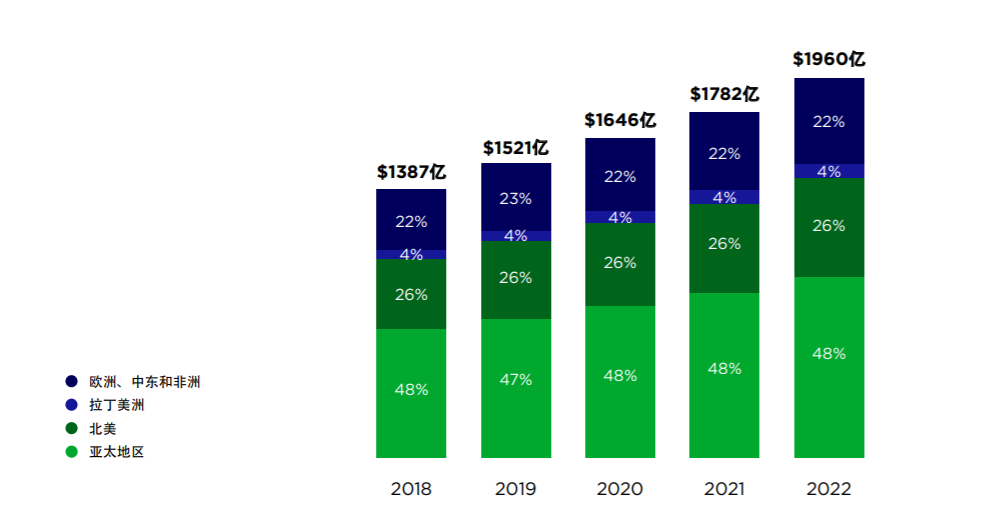
1.1.1研究背景

由于科技水平的进步，现在的PC、手机的性能都有了快速增长，这为游戏行业的火爆奠定了物理基础。另外在当今社会中人们工作都非常忙碌，消遣娱乐的方式也在逐渐由实体娱乐向虚拟游戏靠拢。从2G时代到4G时代，游戏分类也慢慢扩展：网游、主机游戏、端游、页游、手机游戏等游戏如雨后春笋一样蓬勃发展。有关报告显示，2019年中国游戏用户规模达到6.4亿人，较2018年增长2.5%。2019年中国游戏市场实际销售收入2308.8亿元，同比增长7.7%，总游戏用户规模达到6.4亿。而2020年由于疫情爆发，民众大多宅在家中，这对国内的游戏行业反而起到了“助推剂”的作用。如下图所示，数据预测2020年我国游戏市场的销售收入将达到惊人的2586亿元。





如下图所示，2019全球游戏市场收入达到1521亿美元，年同比增长9.6%。到2022年，全球游戏市场将增长至1960亿美元，2018-2022年期间的年复合增长率预计为9.0%。



以上数据充分显示了我国游戏市场的潜力有多么巨大。腾讯、网易这样的老牌游戏厂商在不断扩展自己已有的游戏业务，将业务触手伸向全球市场。就连各大互联网公司也在积极布局游戏业务。像字节跳动这样的信息流企业也已经开始着手进行游戏的发行和研发。游戏研发目前主流的引擎有CryEngine3、Cocos2D、虚幻4等等，但使用最多、最受欢迎的还是Unity3D。该游戏引擎的优点有：易于上手，支持多脚本语言，开发者社区非常强大，并且还拥有着非常出色的跨平台能力。所谓的跨平台能力是指只用一份游戏代码，通过Unity引擎的打包程序就可以打包成能在不同平台上运行的游戏程序。这样不仅可以大大减少研发工程师的工作量、缩小开发周期，也可以大大降低游戏开发成本，为企业节约开支。Unity3D支持三种脚本语言：Boo、JavaScript和C#，这样不同程序员可以选择自己擅长的语言进行开发，最终都能够达到一样的效果，并且程序可以很好地兼容。虽然目前游戏行业整体大趋势是向简单化、休闲化发展，主要目的是让玩家体验到游戏乐趣，而不是增加游戏的难度和操作程度，但本游戏的难度依旧是比较大的，对于玩家的操作水平有较高的要求。另一方面，像本游戏这样对玩家操作有要求的游戏能够给予玩家足够的挑战，激发玩家的胜负心。玩家通过多次练习，能够提高自己的操作水准最终通关，这样得到的成就感也会更强、更真实。

1.1.2研究意义

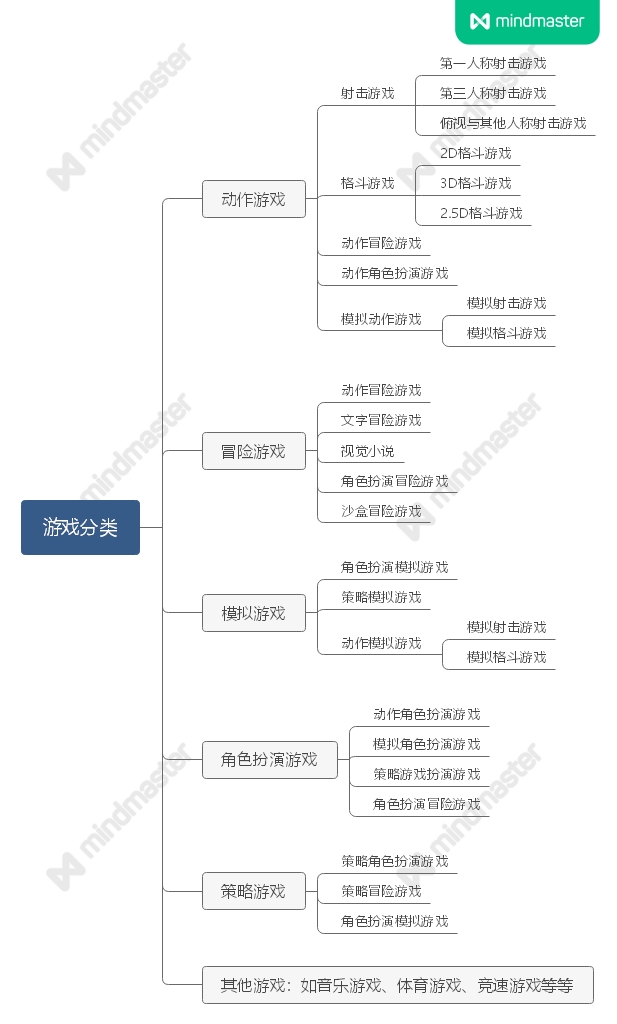
本文研究内容为基于Unity的2D跑酷游戏设计与实现。游戏本身方面，本文的游戏主角在一个阴森的地牢醒来，前方有着重重险阻，一场神奇惊险的地牢闯关之旅也就此展开。整体游戏的画风是偏阴森和黑暗的，这样也和游戏的地牢背景相符。游戏主角能够跑、跳、悬挂（本游戏的特殊机制）以及超级跳，通过跨越各种障碍（包括掉落的砖块、自动弹起的地刺等等）从而通关。通过本游戏的开发过程，能够了解游戏行业的发展历史和前景，了解游戏的制作流程和需要考虑的各种因素。另外还能够深入游戏开发进程，学习游戏中各种常见机制的实现，学习C#脚本开发、音视频特效的实现等等，进而增加自己软件开发的经验并且提高编程能力。

1.2国内外研究情况

上一节中，对本文的背景做了简要介绍，下面分别对国内外游戏产业和游戏引擎的发展情况做简要说明，让读者对国内外游戏产业和游戏引擎的发展历程有一个大概的认识。

1.2.1游戏题材的分类

随着游戏行业的不断发展和更新，电子游戏的类型已经从最初的数种增加到了如今的数十种，甚至同一种游戏类型中还会根据细微的差别细化出不同的子类别。下面是一些常见的游戏题材。

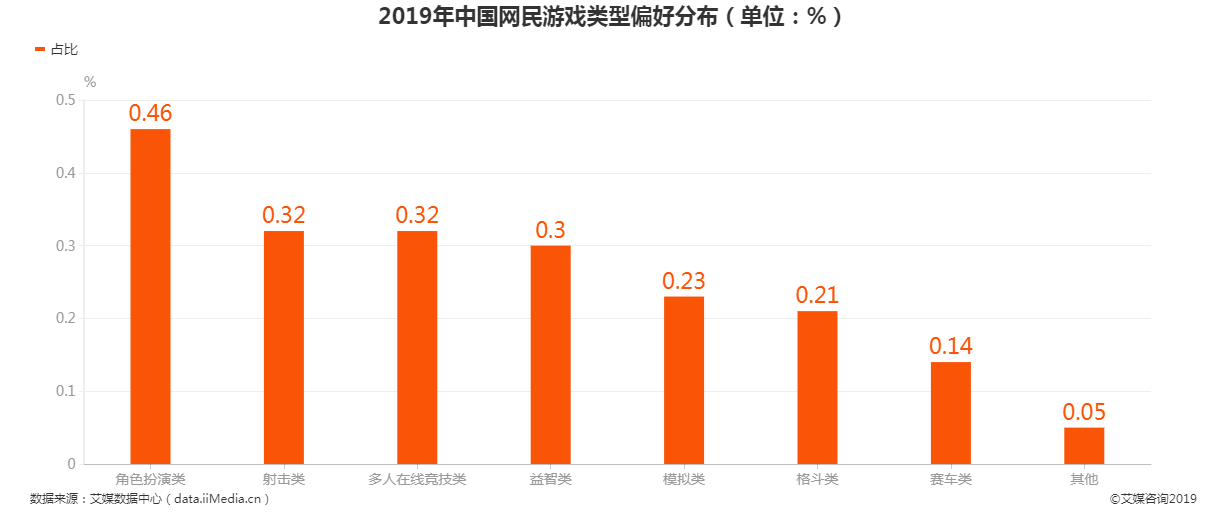


可以看到，各个游戏题材之间是有重叠的，这并不是偶然现象。首先由于游戏行业已经发展数十年，很多游戏类别当初的界定在现在看来并不严谨，造成了这种同一款游戏分属好几种类别的情况。另外游戏的创新性使然，如今的不少游戏在其中刻意加入了其它游戏类别的元素，使得玩家能够得到更新奇刺激的游玩体验。比如GTA5这款游戏，它分线上线下两种模式。线下模式为玩家控制三个主角进行种种眼花缭乱的犯罪任务，这是角色扮演游戏的特征。其中还会涉及到大量的枪战场面，玩家可以选择第一人称或者第三人称来体验游戏，这又包含了射击游戏的特征。而线上模式更多集中在玩家的运营上，如何从一个默默无闻的打手慢慢打拼从而构建自己的商业帝国，这又是模拟游戏的特征。最终GTA5这款游戏看起来像一个大杂烩，但是包罗万象，让玩家沉浸其中，成为了有史以来最成功的跨平台游戏之一。这也说明了游戏研发不能局限于之前的思维，要勇于创新，给予玩家新鲜感。

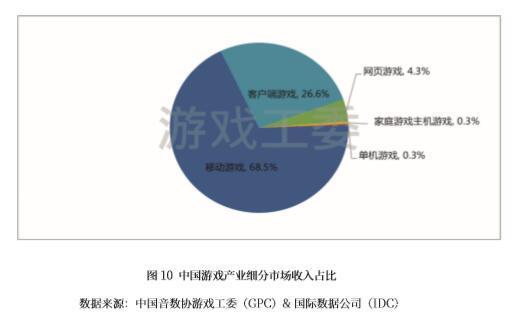
1.2.2国内外游戏产业现状

数据显示，2019年中国游戏产业用户规模同比增长1.6%至6.4亿人。相关分析指出，当前我国消费群体向90后转移，消费群体呈现年轻化态势，基于市场的垂直化发展，女性和二次元消费群体将逐步崛起，并加入游戏市场。所以在此基础上，我国游戏用户规模将进一步扩大。

如下图所示，从我国游戏玩家的游戏偏好分布角度来看，偏好角色扮演类的玩家最多，占比达到了46%，典型的像网易推出的逆水寒、剑三、DNF等等。喜爱射击类和多人在线竞技类的玩家也分别占到了32%，这其中就包含了如今最火的王者荣耀和和平精英。偏好益智类游戏的玩家占30%，模拟游戏占23%。另外格斗类游戏占21%，赛车类占剩余的14%。



而从游戏平台的角度统计，移动端游戏占据了国内游戏市场的主力，达到了68.5%，这与国内的通信产业极度发达不无关系。客户端游戏次之，占据26.6%的市场份额。而至于其它的游戏平台，如网页、主机和单机游戏的份额几乎可以忽略不计。



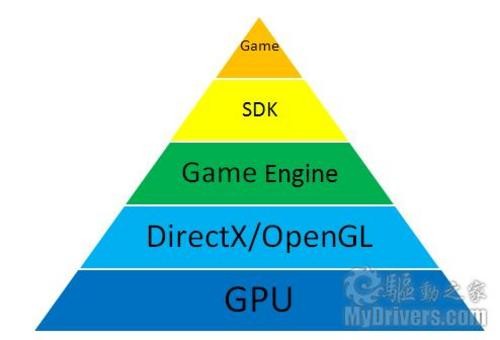
全球游戏市场的情况和国内有所不同。如下图所示，国外游戏玩家更多地还是选择主机和PC平台进行游戏，这主要和国情与国外的经济发达程度有关。中国作为发展中国家，高性能PC和游戏主机对于多数国民还不在消费承受范围以内，但手机基本已经做到了全覆盖。所以国内的游戏玩家更偏好移动端游戏，国外游戏玩家相对来说更偏向主机和PC游戏也就理所当然了。



2.1主流游戏引擎概要

2.1.1游戏引擎定义

汽车引擎赋予汽车动力，使之能够移动。游戏引擎虽然名字也带有“引擎”，但其实际上却是行“大脑”之实，控制协调游戏中的各种资源。游戏引擎的准确定义是：一些已编写好的可编辑电脑游戏系统或者一些交互式实时图像应用程序的核心组件，用于控制所有游戏功能的主程序，从计算碰撞、物理系统和物体的相对位置，到接受玩家的输入，以及按照正确的音量输出声音等。从计算机系统的层面看，最底层的是GPU（即显卡），上面一层是各种图形API（如DX9、DX11、OpenGL等）。而游戏引擎则是建立在这些图形API基础之上的，控制各种游戏资源反馈给玩家不同的效果。从GPU到实际的游戏可分为如下五层，可以看到游戏引擎将硬件和软件联系起来，起着非常重要的作用。



首先游戏引擎要负责光影效果的绘制，即场景中的不同光源对其中人和物的影响方式，比如折射、反射等基本的光学效果，以及动态和彩色光源等。其次是动画系统，一般分为模型动画系统和骨骼动画系统。

另外游戏引擎要提供物理系统，最常见的比如重力、摩擦力，这可以使游戏中的物体看起来和现实中遵循一样的物理定律，更加真实。物理系统的核心部分是碰撞探测，这样角色和墙面边缘相遇的时候就会停下，以防止穿模。同时游戏引擎还可以探测并记录碰撞以产生碰撞效果以及调用其它事件。

渲染是游戏引擎最重要的功能，没有之一。美工会为游戏的不同物体制作基础的模型，以及相应的贴图、材质，游戏引擎再通过把模型、动画、光影、特效等所有效果汇总到一起实时计算出来并展示到显示器上。一般游戏每秒都有至少60帧画面，这也就意味着游戏引擎每秒要计算60次画面并输出，这其中的工作量非常大，过程极其复杂。游戏引擎的强大与否直接决定了最终的游戏画面质量。

2.1.2 Unreal Engine（虚幻引擎）

Unreal是由波兰EPIC公司开发的游戏引擎，提供了游戏开发者需要的大量核心技术、数据生成工具和基础支持。第一代于1998年推出，开发的游戏有《魔域幻境》和《虚幻竞技场》。之后虚幻引擎还经历过两代的发展，直到2008年发布了第四代，也就是风靡全球的“虚幻四”。绝地求生、方舟、战争机器等等3A大作都是由虚幻4引擎开发的。该引擎的优点是其具有非常强势的实时渲染系统，能够输出非常惊人的CG级画面，并且源代码是对公众开放的。缺点是其对硬件配置要求非常高，以至于当初有一部分使用虚幻3引擎做出的游戏都不能在主机平台上游玩，使得下一代引擎虚幻4不得不为主机平台重新做了适配。

2.1.3 Cocos2d-x

cocos2d-x是一个开源的移动2D游戏框架，基于MIT协议的免费开源框架。支持Windows、Mac和Linux等操作系统。其优势在于允许开发人员利用C++、Lua及Javascript来进行跨平台部署，包括iOS、Android、、Windows和Mac OS等等。该引擎方便拓展，易于集成第三方库。常见的利用cocos2d-x开发的游戏有捕鱼达人、魂斗罗和卧龙吟等。

但另一方面，其无法避免的劣势就是其只适合做2D游戏，而目前游戏更多是在向3D方向发展，所以其越发显得吃力。

2.2 Unity3D

2.2.1 Unity3D介绍

Unity3D是用Unity Technologies开发的一个让玩家轻松创建诸如三维视频游戏、建筑可视化、实时三维动画等类型互动内容的多平台的综合型游戏开发工具，是一个全面整合的专业游戏引擎。2004年，Unity3D诞生于丹麦，最初它只能应用于Mac平台。2008年，推出Windows版本。2010年适配Android，2011年适配PS3和XBOX360。至此Unity3D真正成为跨平台的游戏引擎。

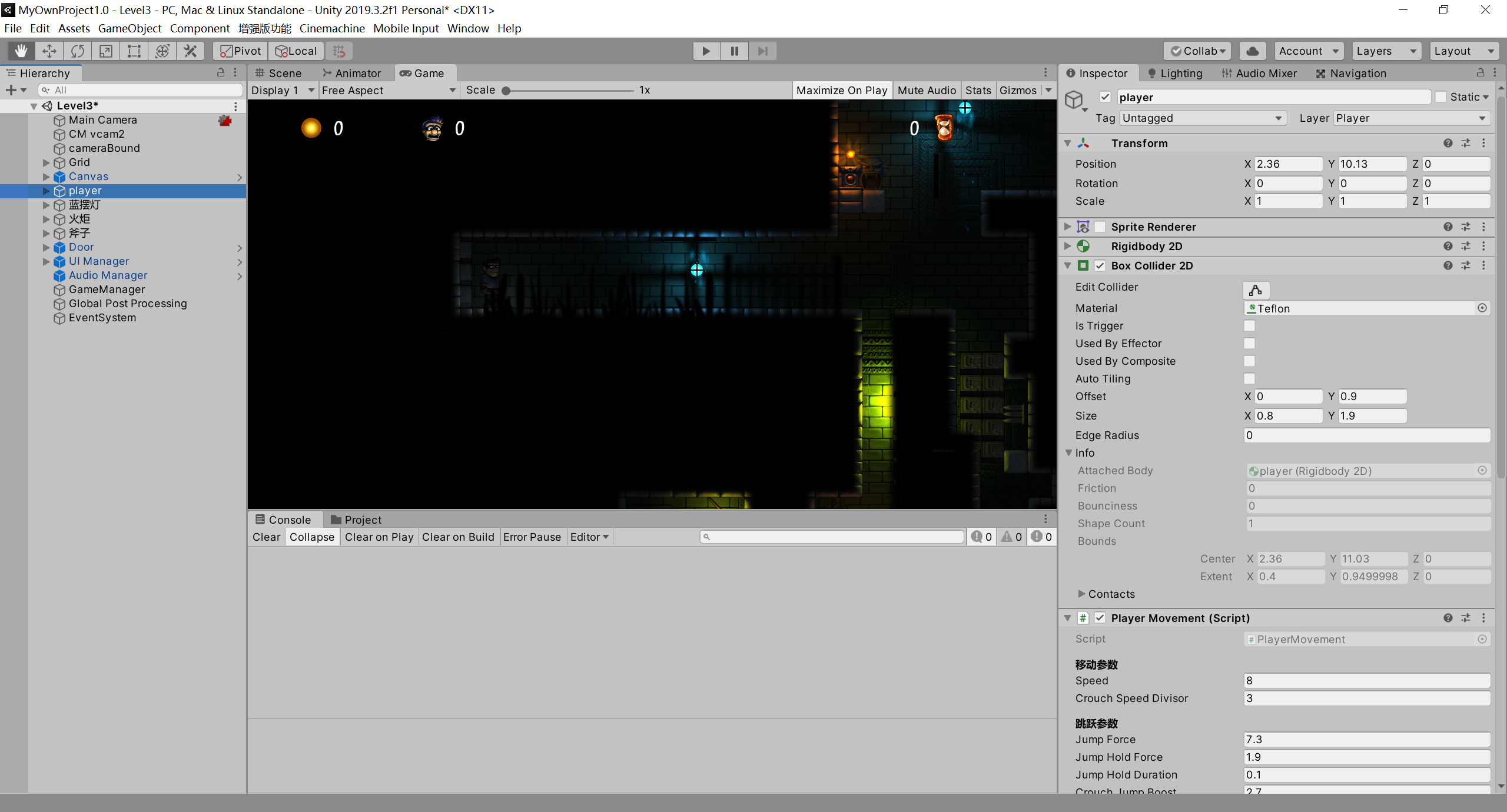
Unity3D的优势首先是其真正的跨平台开发特性，可以做到一次开发，多平台发布。第二，Unity3D的入门门槛比较低，具有轻松友好的可视化编辑界面，容易上手。第三，Unity3D的配套资源极其丰富。因为Unity3D是目前最火爆的游戏引擎，用的人非常多，所以其配套的开发者社区非常完备，也有非常详尽的官方文档。另外其内置的Asset Store提供了大量游戏必备的资源，如声光特效、美术材质、资源模型等，这样没有美工的中小型游戏开发者也可以方便地从Asset Store购买自己所需要的各种资源和游戏功能模块，极大地提高了游戏的开发效率。

Unity3D不仅可以开发3D游戏，2D游戏也完全能够胜任。另外Unity3D还可以做AR、VR、实时动画、室内设计、数控仿真和教学软件等等。

基于以上原因，本文最终选择Unity3D引擎进行开发。

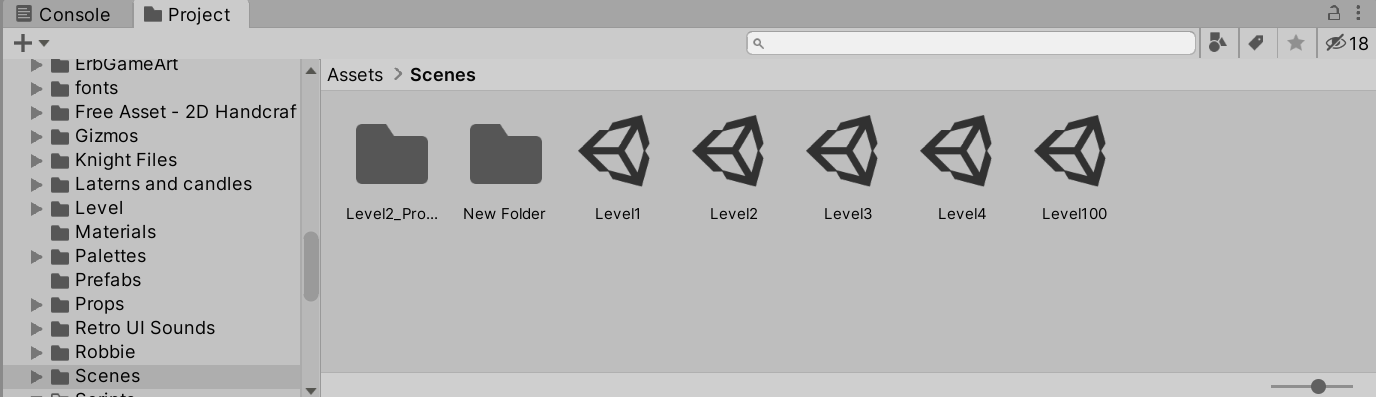
2.2.2 Unity3D开发界面

Unity3D引擎的主体开发界面如下图所示。

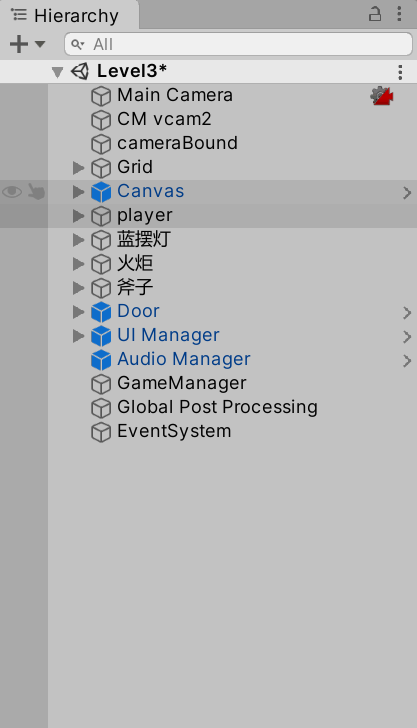


开发界面主要由五个部分组成，分别是Project、Hierarchy、Scene、Game、Inspector视图。

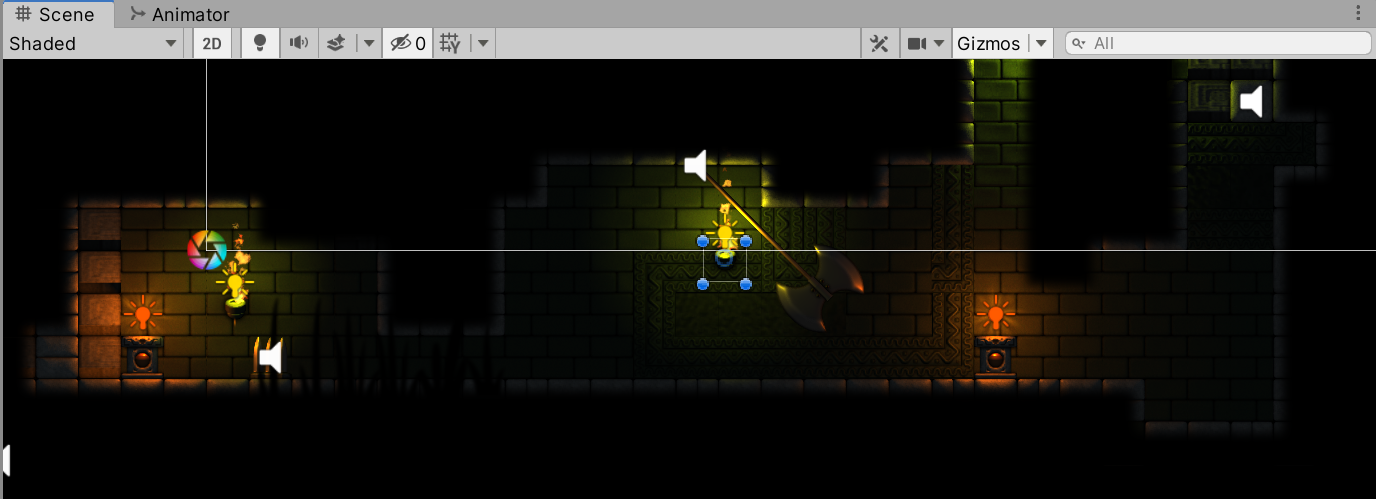
1. Project（游戏工程视图）：包含所有游戏工程文件夹中的文件、资源、代码等等。从Asset Store下载的asset或者从外部导入的package都会存放在Project视图中。开发者可以直接在其中新建或删除文件，相应的改动会直接反映到文件系统中，非常方便。Project视图如下图所示。



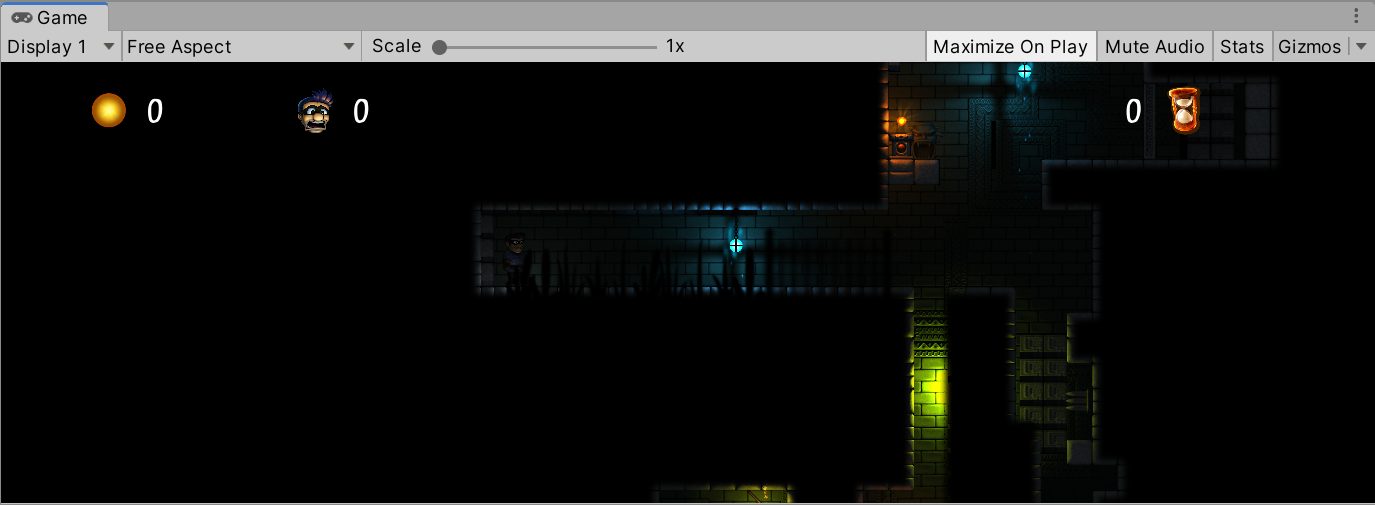
1. Hierarchy（游戏层次视图）：其包含所有游戏场景中的物体，也就是GameObject。其中的对象都具有一定的层次，一个游戏对象可以有很多子对象。开发者还可以对这些物体进行归档分类以更易读。Hierarchy视图如下所示。



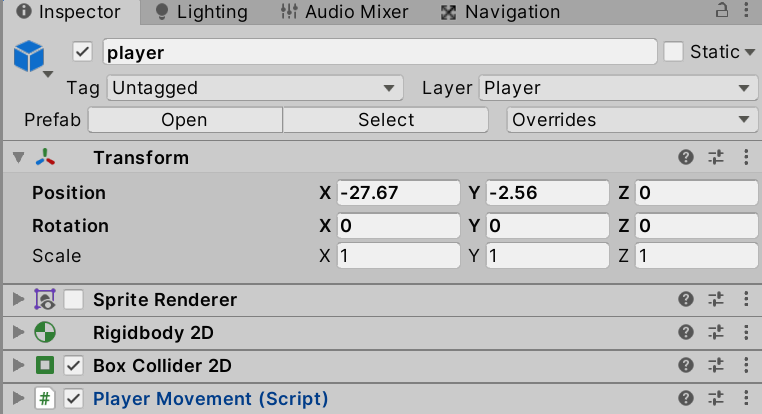
1. Scene（游戏场景视图）：该视图为开发者摆放游戏画面、场景的主要区域。开发者可以调节各个游戏对象的位置、旋转角度，能够实现真正的“所见即所得”。Scene视图如下所示。



1. Game（游戏预览视图）：Game窗口会展示游戏主摄像头拍摄到的实时画面，如果开发者在代码中有摄像机的切换，也会按照代码切换不同摄像机的视野。该窗口可以用作游戏预览，让开发者可以在开发过程中随时预览游戏画面并做出调整。Game视图如下所示。

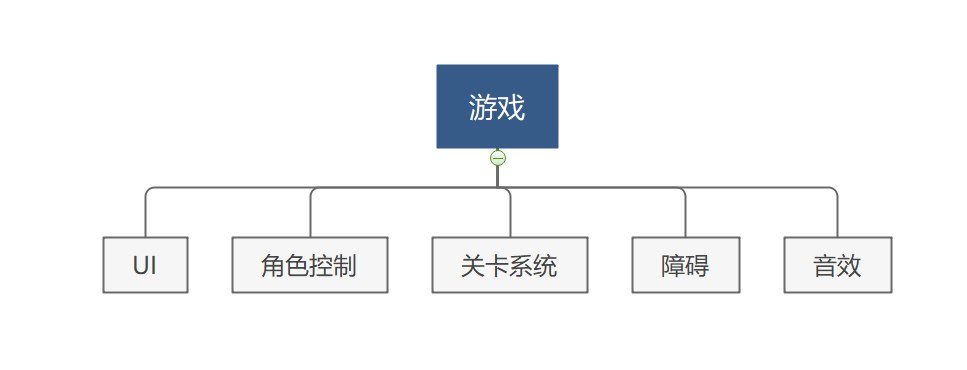


1. Inspector（检视器视图）：当开发者在Scene视图中选定一个游戏对象时Inspector视图中就会显示该对象的所有属性和可调整选项。开发者可以通过拖拽来为游戏对象添加组件，或者直接改动某些属性。Inspector视图如下所示。



3.1游戏整体设计思路

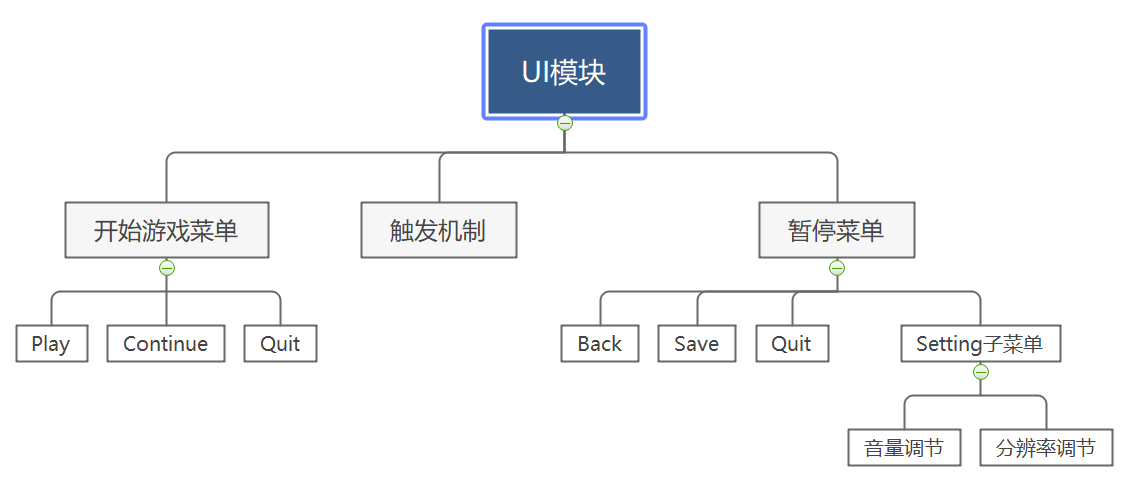
本游戏整体设计图如下。



下面分别对各模块进行概述。

3.2 UI模块

UI模块如下图所示，包括开始游戏菜单和暂停菜单。其中开始游戏菜单需要实现开始游玩、读取已存储的游戏、关闭游戏等功能。暂停菜单需要实现恢复游戏、存储游戏、退出游戏等功能。另外还需要内嵌设置功能在暂停菜单中，这样玩家可以在游玩过程中暂停并调节一些设置（如音量、分辨率等）来达到更好的游戏体验。另外UI系统还需要有一套触发机制，默认情况下是隐藏的，只有需要的时候才会显示。



3.3 角色控制模块

本游戏为2D游戏，所以角色的移动限制在2D层面。最基础的2D移动：前进后退跳跃下蹲是必须要实现的。并且由于本游戏是利用TileMap实现的，所以全部地形都是由相同大小的“块”组成的。基于此，本游戏还实现了一个特殊功能：悬挂。该功能使得角色可以悬挂在障碍物边缘，并且还外延出两个子功能：悬挂时下落和悬挂时跳跃。另外还实现了超级跳的功能，角色通过下蹲蓄力就可以跳的更高。

3.4 关卡系统模块

作为2D闯关游戏，关卡系统是必须拥有的。Unity3D引擎中可以利用Scene来很方便地构建不同关卡，开发者只需要编写脚本实现关卡之间的切换即可。另外该模块内还实现了新手教学功能，在第一关帮助玩家快速上手。

3.5 障碍系统模块

一般来说，跑酷游戏都需要玩家控制角色翻越障碍，或者击杀怪物从而过关。障碍系统是为了给玩家的游戏增加难度，这样玩家在不断尝试不断提高自身操作的过程中会得到更大的成就感。本游戏实现的障碍包括掉落的砖块、凸起的地刺、摇摆的斧头和滚动的锯盘等。玩家必须控制角色在适当时候跳跃、下蹲来快速通过这些障碍，否则就只能死亡重来。

3.6 音效系统模块

游戏中有很多声音，比如：角色走动的声音、背景音、道具音效、其它特效音等等。这些声音如果每个都独自播放，那么会导致一个声音播放了一点点马上就被切断，切换到下一个声音还是如此。这样就会生成非常奇怪的效果：所有声音都完全听不清，因为所有音源都在快速不停地切换。这就势必需要一个整体的声音管理器，通过不同通道并行播放不同声音来达到正常的效果。这个声音管理器应当是全局的，并且拥有多个通道，可以通过代码实时控制每个通道播放的音源以及相应的音量大小。

4 游戏的具体实现和测试

4.1 游戏开发环境

本游戏采用Unity3D引擎开发，使用C#作为脚本语言。Unity3D的版本为2019.3.2f1。C#使用Visual studio 2019（后文简称VS）作为编辑器。这主要是由于VS对于Unity3D有特殊的适配。一方面可以在Unity中直接新建代码并在VS中打开进行编辑，另一方面还可以在VS中打断点进行调试，并且Unity3D中运行游戏也会触发相应断点，这样就大大提高了开发者的调试效率。

4.2.UI模块的实现

4.2.1 开始菜单

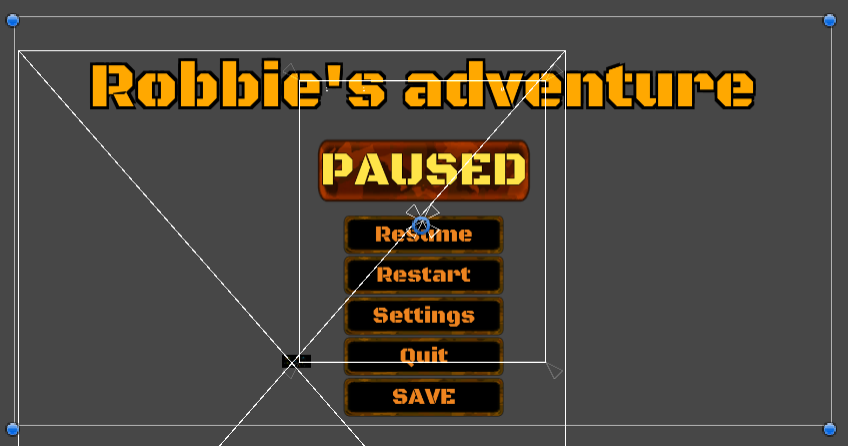
开始游戏菜单如下所示。由两部分构成，上面一行为游戏名字：Robbie’s adventure。

下方为三个按钮，功能分别为开始游戏（Play）、从存储处继续（Continue）和退出（Quit）。



4.2.2 暂停菜单

游戏中的暂停菜单如下所示，整体布局和开始游戏菜单类似，多了三个功能：Restart（重新开始这一关）、Save（存储游戏）和Setting（跳转设置菜单）。下面详细讲解UI模块不同功能的具体实现。



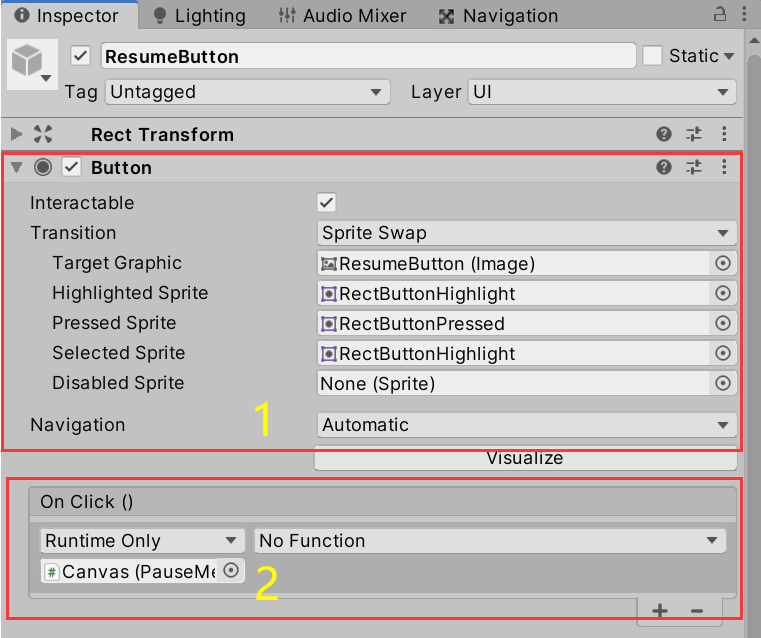
4.2.3 UI模块各功能的详细实现

1. 静态文字的实现

游戏场景中有许多静态文字，比如游戏开头显示的标题、新手教学的文字和其它提示语等。使用Unity3D提供的UI.Text可以自定义显示的内容、文字样式和大小等等。并且所有UI类的实例都是GameObject，所以都可以在不需要时隐藏，在需要时通过SetActive(true)来使其显示在画面中。

1. 按钮的实现

游戏中有很多按钮，都同样使用UI.Button - TextMeshPro实现。TextMeshPro是Unity推出的默认文本组件替代品，由于使用了Signed Distance Field技术，使得它拥有和默认UI组件近似的高性能，但它相比默认组件提供了更加高级的控制功能和更多的效果。下图为游戏中一个按钮的属性面板。其中第1部分为Button的交互设置，Interactable指明该按钮可以交互，如果这里不打勾那么下面所有设置都没有效果。Button有点击、悬浮（高亮）、选择等状态。玩家点击不同按钮，按钮就会显示不同的样式来提示玩家。Transition为按钮切换方式，一般选择Sprite（图片切换）或者Animation（动画切换）。本文选择Sprite实现，原理就是对应Button不同的状态，选择不同的背景图片，在切换状态时切换不同的背景图片来达到想要的效果。第2部分OnClick属性可以绑定函数，在该按钮被点击的时候就会调用该函数。所以对于不同按钮绑定不同的函数以实现差异化的功能。



1. 滑动条的实现

Setting菜单如下，其中内置了5个滑动条，分别控制五个声音通道的音量大小。滑动条使用UI.Slider实现。Slider也在点击、悬浮等不同状态有不同的显示效果，在滑动的时候会调用OnValueChanged属性设置的函数，从而改变对应通道的音量。



1. UIManager（下称UM）的实现

UM是一个类，负责宝珠数量、玩家复活次数、已花费时间和FPS等信息的实时更新和显示。UM类的实现中利用了单例模式，只生成了一个全局静态实例，这样在切换关卡时就不会重新生成UM对象，节省了无谓的资源销毁和重新生成。另外这样做还有一个显著的优点：如果上一关卡中玩家在Setting菜单中调整了相应的声音设置，在下一关卡中就会直接延用之前的设置，这是因为之前的UM对象没有销毁。而如果不使用单例模式的话，想实现这样的功能就会比较困难。UM管理宝珠数量、玩家复活次数、已花费时间这三个功能都是一样的设计思路，通过一个图标和一个动态Text实现。图标直接指明后面数字对应的意义，之后紧跟的Text通过挂载的脚本进行实时更新。FPS显示功能实现代码如下。

void Update ()

{

deltaTime += (Time.unscaledDeltaTime - deltaTime) \* 0.1f;

SetFPS();

if(Time.time-lastchangedtime>1f && Math.Abs(fps - lastfps) > 5)

{

lastchangedtime = Time.time;

lastfps = fps;

fpsText.text = string.Format("FPS: {0:000}", lastfps);

}

}

void SetFPS()

{

float msec = deltaTime \* 1000.0f;

fps = 1.0f / deltaTime;

}

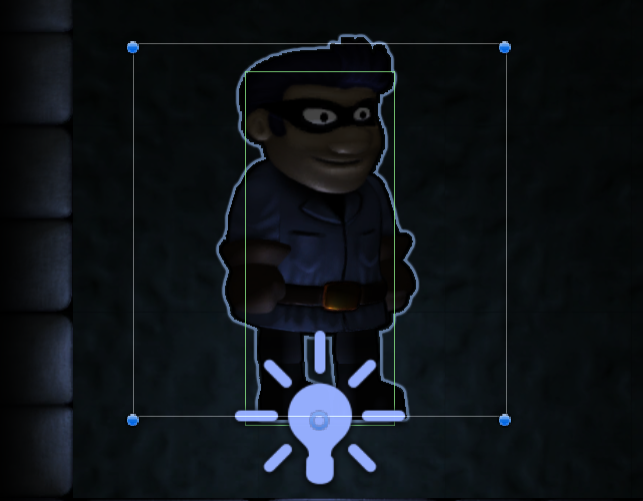
整体思路是利用Time.unscaledDeltaTime得到上一帧到当前帧所经历的时间，取其倒数即可进行格式化输出。但有一个小细节需要注意，通过这种方式算出来的FPS值每秒会刷新几十次（Unity3D默认为50次），这会导致玩家完全看不清FPS的值。所以实现中另外生成了一个变量lastchangedtime记录上次更改FPS值的时间戳，间隔1秒以内不再更改FPS值。

1. 触发机制

玩家在游戏过程中直接按esc键即可调出暂停菜单，实际的游戏“暂停”效果通过更改Time.Scale=0f来实现。菜单中不同按钮的切换通过方向键和Enter键来完成，切换和点击按钮还有对应的反馈音效。

4.3 角色控制模块的实现

角色的Scene视图如下，首先角色外表是一张贴图，贴图没有物理效果，也没有碰撞体积。为了实现物理效果，需要为角色添加一个RigidBody2D组件。RigidBody2D是Unity中专门用于2D游戏的刚体组件，该组件能够赋予绑定的父对象各种实际的物理效果。比如重力、摩擦力、速度等等。另外还可以指定该刚体是否为Static的，如果是Static的话该刚体就会保持静止状态，如果是Dynamic，那么就会受到其它力的作用效果，在这里我们的角色需要是Dynamic的。另外由于是2D游戏，所以RigidBody2D中的Freeze Rotation选项中要勾选Z轴，这样角色就不会旋转了。



第二，为了和其它物体（如地面和障碍）产生碰撞效果，角色的GameObject上还绑定了一个Box Collider2D。这是一个长方体的碰撞体，在实际的碰撞事件中只有这个Collider是有效的，角色表面覆盖的贴图并没有碰撞体积。一般来讲，人物的碰撞体可以由两个碰撞体实现，一个Box Collider2D负责上半身，一个Circle Collider2D负责下半身。单长方体碰撞体方案的劣势有：如果地面不平或者有缝隙，就会发生角色水平移动时被卡住的情况；另外如果地面存在斜面，那么角色在斜面走到平面上的时候，就会有一个短暂的腾空，这可能会导致角色动画的突然切换，显得比较突兀。但本文对角色的实现中依然选择Box Collider2D实现，这主要是基于三点考虑：第一，游戏地形和所有的障碍都是基于TileMap实现的，大小都是一样的1\*1；第二，游戏地形中不存在斜面；第三，由于角色实现了一个“悬挂”的操作，多碰撞体实现这个功能会比较麻烦。经过测试，单碰撞体实现方案运行效果良好。

4.3.1判断是否在地面上

这个功能在PhysicsCheck()函数中实现，它会被很多其它功能调用，所以非常重要。例如角色不在地面上的话就不能跑动和跳跃。本文实现该功能时使用了Unity3D的射线功能。在Unity3D中，射线就是从一个点向另一个点发射的一条无限长的线，在飞行过程中如果击中了某一碰撞体就会停止，并通过相应API反馈击中的碰撞体信息。一般游戏开发者会使用射线来制作子弹击中效果和碰撞效果等，本部分核心代码如下。

RaycastHit2D leftCheck = RayCast(new Vector2(-footOffset, 0f), Vector2.down, groundDistance, groundLayer);

RaycastHit2D rightCheck = RayCast(new Vector2(footOffset, 0f), Vector2.down, groundDistance, groundLayer);

isOnGround = leftCheck || rightCheck;

原理就是在角色左右脚位置建立两条垂直向下的射线，只要有一条射线击中了碰撞体就说明角色正站在站在地面上，效果图如下所示，绿色为未击中，红色为已击中。



4.3.2 水平移动的实现

该功能在GroundMovement()函数中实现。如下图所示，首先要识别键盘输入，一旦识别到玩家按下了水平方向键，那么就进入了整个流程。

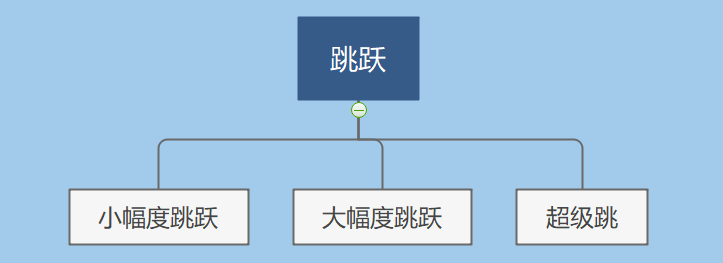


首先要判断一系列的条件：角色需要在地面上，这部分由isOnGround变量的值指示。如果目前为下蹲状态，那么只能蹲下行走；如果角色为站立状态，那么可以跑动；如果角色当前为悬空状态，那么无法进行水平移动。脚本中通过isHanging、isCrouching变量来区分这几种不同情况。

在通过上一个阶段后，角色是能够行走/跑动的。通过Input.GetAxis("Horizontal")取得玩家按下的键，如果是左方向键就会返回-1，右方向键则返回1。其中Horizontal对应的键位需要在Project Settings中的Input Manager中进行设置。接下来如果要蹲下行走，将速度变量除以一个divisor，实际的移动操作则直接对rigidBody2D赋予一个Vector2类型的速度跑动值即可。

4.3.3 跳跃的实现

该功能在MidAirMovement()函数中实现，整体结构如下所示。



1. 小幅度跳跃（以下简称“小跳”）

玩家只按一下“Jump”按键激发小跳。对应的条件变量是jumpPressed，在update函数中调用jumpPressed = Input.GetButtonDown("Jump")得到当前帧内是否按下跳跃按键。如果该Bool值为真，赋予角色刚体组件一个向上的力并更改相应状态指示变量，这部分不再赘述。

1. 大幅度跳跃（以下简称“大跳”）

大跳是在小跳的基础上实现的。玩家按下跳跃键直接激活小跳，接下来在FixedUpdate函数中考察是否跳跃键还是按下的状态，记录到Bool变量jumpHeld中。由于该函数是固定时间刷新的，所以如果jumpHeld为真，就说明从上一个Frame（有效帧）到当前FixedUpdate刷新的这段时间内，玩家一直按着跳跃键。直接在小跳的基础上再为角色刚体组件额外添加一个跳跃力，这样角色就能够跳的更高。

经过反复测试，该额外跳跃力维持0.1s左右效果最好，所以脚本中还另外生成了一个Float变量用来记录赋予该力的已持续时间，一旦超过时限就取消该力，让角色开始自由落体。

1. 超级跳

角色如果在下蹲状态进行跳跃，就会激活超级跳。同样地，首先需要一个Bool变量isCrouching记录是否下蹲，如果isCrouching为真，并且按下了跳跃键，就添加一个额外的跳跃力以跳的更高。游戏中设置了相当多的地形，只有使用超级跳才能跳上去或者刚好挂住。

三种跳跃方式的高度对比如下：



4.3.4 下蹲的实现

1. 下蹲

要实现下蹲效果，角色的碰撞体需要改动。这是因为角色和其它物体的交互实际依赖于碰撞体之间的碰撞事件。一般来说有两种实现方案：使用多个碰撞体，一大一小，在下蹲时进行切换；或者直接改变已有碰撞体的size和offset。本文选择第二种实现方式，相关实现代码如下，实现在Crouch()函数里。其中size为碰撞体的大小，offset为碰撞体的偏移位置。如果只改size，Collider会基于原来的中心进行等比缩小，所以还需要更改Collider的offset。

isCrouching = true;

playerBoxCollider2D.size = colliderCrouchSize;

playerBoxCollider2D.offset = colliderCrouchOffset;

2. 站起

和下蹲相反，角色站起时将碰撞体的size和offset恢复原值即可，相应实现在StandUp()函数里。

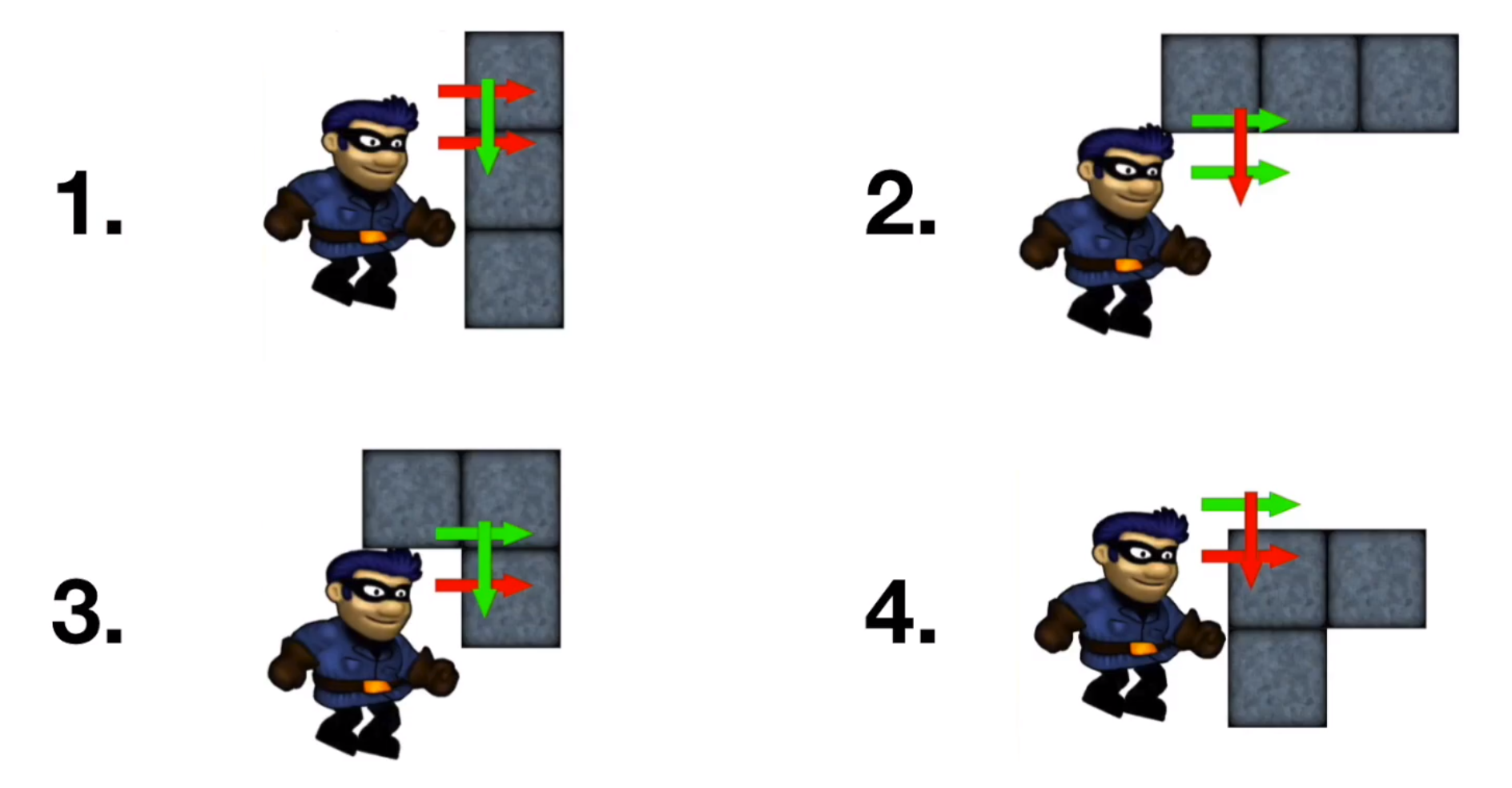
3. 相关细节条件判断

首先，如果角色不站在地面上就不能执行下蹲和站起操作。角色正在下蹲通过障碍时松开了下蹲键，这种情况需要单独处理。这是因为如果此时角色站起，Collider恢复之前的属性，这样就会和地形Collider产生冲突，很有可能会把角色弹飞或者“鬼畜”在原地。所以这种情况下不能执行站起操作。另外一种情况是：如果角色下蹲通过障碍后松开下蹲键，此时角色又应该自动站起。这里的判断也是通过射线实现，在玩家头顶向上生成一条射线，如果击中了其它Collider就说明此时角色头顶有障碍物，此时无法执行站起操作。以上所有这些情况都需要在脚本中加以区分。

4.3.5 悬挂的实现

悬挂为本游戏实现的一个特殊操作，如下图所示，角色可以实现单手攀附在地形边缘的效果。由于本游戏的地形依靠TileMap设计，所以所有的地形都是按照“单位块”的方式搭建起来的，这为本功能的实现提供了前提条件。如果不是基于TileMap实现的地形，那么地形边界都是不规则图形，根本无法实现类似的操作。



该功能的实现较为复杂，示意图如下。在角色头部位置放置了三条射线，利用射线的击中状态来判断相关条件。首先要明确Unity中射线的击中机制：从射线起点开始的路径上如果遇到了新的Collider，或者移出了之前的Collider就会反馈击中状态；但是如果射线起点本身就位于该Collider中那么该射线是不会反馈击中的。在角色和地形距离很近时，会有以下几种情况出现。第一种：角色紧靠垂直地形下落，这种情况下两条水平射线都会击中，而垂直射线不会击中。第二种：角色在地形边缘下方下落，只有垂直射线击中。第三种：角色头顶和前方都有地形阻挡，只有下方的水平射线会击中。第四种：角色在地形边缘上方下落，只有上方的水平射线不会击中。只有这种情况才可以悬挂，其余情况都没有悬挂的前提条件。实际的悬挂操作只需要把角色刚体RigidBody2D组件更改为Static，解除悬挂状态时再改回Dynamic就可以了。  


概括来说，悬挂操作的实现逻辑并不难，难点主要在于条件的判断，核心代码如下。首先保存角色之前的位置，接下来将角色刚体固定。这里对于固定位置做了限制，每次悬挂时角色和攀附地形的相对坐标都强制保持一致，这样除了整齐一致的好处以外，还能够使得后续悬挂时跳跃和下落的操作更加稳定。否则若每次悬挂时角色和攀附地形的相对距离存在一定误差，可能就会出现两次悬挂同一位置，但一次能够跳跃到下一更高地形，另一次却不行的情况，这样会使游戏性大打折扣，所以通过锁定相对坐标的方式来规避这种情况的发生。

Vector3 pos = transform.position;

pos.x += (grabEyeCheck.distance - 0.05f) \* direction;

pos.y -= grabDownCheck.distance;

transform.position = pos;

playerRb.bodyType = RigidbodyType2D.Static;

isHanging = true;

4.3.6 小跳和大跳的实现区别

Unity3D的MonoBehavior类中，Update函数每一帧执行一次，FixedUpdate函数每隔固定时间执行一次，这个时间间隔默认是0.02s。对于跳跃功能，如果想达到无延迟的跳跃效果，那么就一定需要在Update函数中进行跳跃按键的识别。如果在FixedUpdate()函数中实现按键识别的话，可能会出现当前帧并没有调用FixedUpdate()函数的情况，就会造成玩家跳跃键无反应的错觉。另外相邻两帧之间的间隔时间是不固定的（和FPS有关），所以Update()函数中无法完成需要一定持续时间的操作。

综合前面所述，小跳操作实现在Update()函数中，这是因为它需要即时响应性。并且小跳是后续大跳的基础，如果它不灵敏，那么会影响很多操作的顺畅程度。在小跳的前提下，玩家按住跳跃键不松开再激活大跳，所以它需要一定的持续时间，实现在FixedUpdate()函数里。小跳使用Input.GetButtonDown()函数识别输入，该函数的作用为：如果当前帧内按下对应键就返回True，着重点在于按键的“按下”过程。大跳使用Input.GetButton()函数识别输入，该函数的作用为：如果当前按住了对应键就返回True，着重点在于保持“按住”状态。这两个函数各自的特点决定了它们不同的使用场景。

4.3 游戏管理逻辑的实现

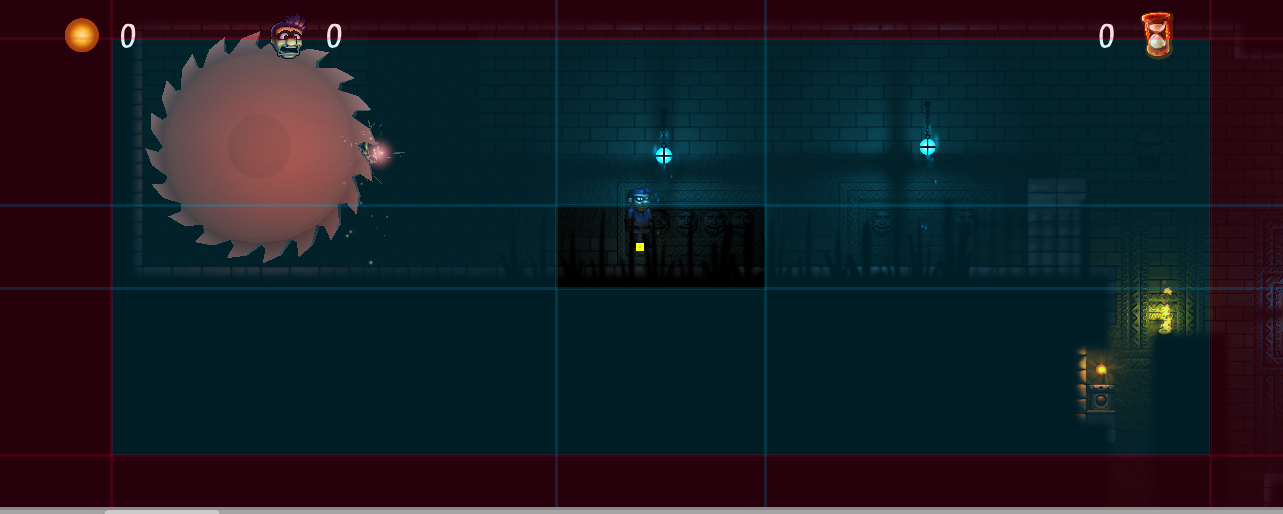
游戏管理需要负责很多方面，包括画面和摄像机的设置、关卡间的切换、游戏的储存和读取以及游戏运行中信息的记录等等。游戏作为一个音视频、图片、脚本等各种元素合并成的整体，其中不同功能有很多子系统去实现，但仍然需要被连接到一起从而实现信息互通。这和如今互联网企业的“数据中台”的概念很相似，虽然远没有那么复杂，但原理是类似的。

4.3.1 游戏摄像机

在游戏中Camera就像是玩家的眼睛，玩家通过它来观察整个游戏世界，所有炫酷的场景、特效都需要通过Camera渲染出来。一个场景至少需要一个Camera，也可以设置多个，在不同环节切换显示。

本游戏的摄像机只有一个，即跟随角色的主摄像机。要实现跟随功能，简单粗暴的方法可以直接把Camara的transform.position设置成与角色的一样。这样做的确可以做到跟随角色的效果，但问题也很多：没有缓冲，玩家一点点的移动都会反映到摄像机身上；跳起和下落的时候整个画面会快速滑动，对人眼不够友好；角色在画面中的位置是固定的，但有些场景需要角色在上方或者上方，这种情况没法解决。

本文使用Cinemachine（下面简称“CM”）插件实现跟随效果。CM是Unity3D推出的相机管理工具，大大拓展了默认Camera的功能。如下图所示，CM中可以指定要follow和lookat的GameObject。除了常见的视野调整功能，CM最重要的功能就是可以实现带Damping的相机跟随效果，Damping的直翻意思是阻尼，在CM中指的是画面的移动会延迟于follow对象的移动，并且这个延迟是开发者可控的。CM中定义了Dead Zone和Soft Zone，角色在Dead Zone范围中移动时，Camera不会移动。移出该范围就会进入Soft Zone，这时Camera会略微延迟地开始跟随角色移动。

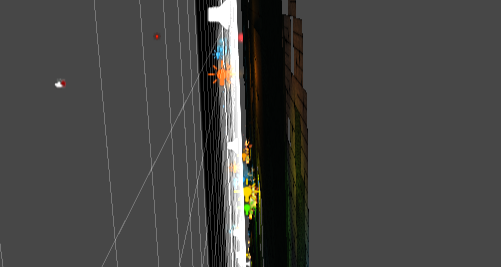


另外还有一个小问题，由于游戏地形存在边界，需要对相机的范围作出限制。CM中的Confiner属性提供了该功能，通过接收一个Polygon Collider（多边形碰撞体），可以把相机视野限制在该区域内。

4.3.2 游戏地形

本文选择TileMap实现游戏地形。TileMap即瓦片地图，它包含四个部分：Sprite、Tile、Palette和Brush。Sprite即精灵，是Unity中2D图片的简称；Tile包含一个Sprite，以及Color和Collider属性；Palette是画板；Brush用来绘画；通过TileMap方法，可以把不同的地形块用刷子刷到游戏场景中，非常地方便。整个游戏场景是由许许多多的Tile组成的。

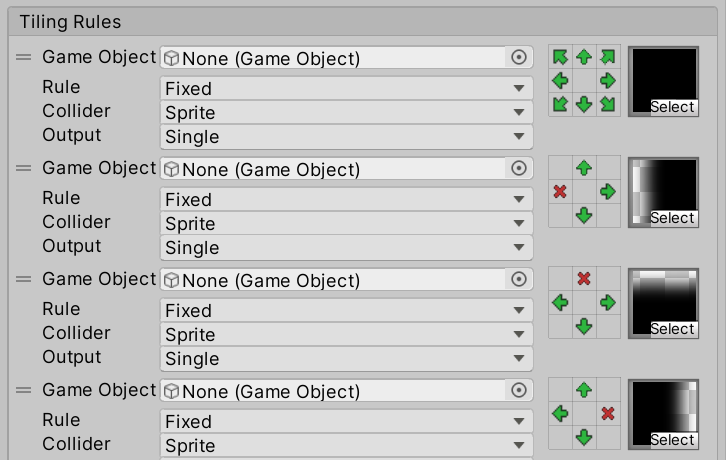
游戏背景一般由多层画面构成，在角色移动的时候，各层次画面移动的速度会有所差别，使游戏画面富有层次感。本游戏实现了多个Layer（图层），分为前景、中景、背景等。它们各自的Z轴坐标都有所差别，角色在前景层中移动的时候，由于距离的差异，中景物体的移动速度会稍慢，远景则更慢，达到Parallex的效果。

/

除了地形是用砖块的Tile实现，本游戏中的一些特殊障碍道具也是用TileMap实现。如下图所示，宝珠、掉落的砖块和地刺其实都是Prefab，利用TileMap自定义Brush的功能生成对应不同的Brush。这样放置这些道具的时候，就可以直接像刷Tile一样直接刷在场景中就可以了。

TileMap还有一个很方便的功能：Rule Tile，它可以按照一些预制的规则自动生成Tile。

如下图所示，其中的九宫格表示以当前块为中心的9个Tile位置。绿色箭头表示该方向上有其它Tile，红色表示该位置无Tile。每一个项目表示一个状态，符合该状态的Tile会自动生成，比如在墙边或者地面，这些Tile就会整齐地自动对齐，极大地提高了开发效率。



4.3.3 GameManager类

游戏内部有许多类，分别实现不同的功能。为了使这些功能实现一定程度的消息互通，实现了GameManager（下简称“GM”）类。GM类和UIManager类一样使用了单例模式，全局范围内只有一个Static实例，所以在初始化时也要加上DontDestroyOnLoad(this)以保证不会在切换关卡时销毁该对象。以下是GM类实现的功能。

1. 注册功能

为了通关玩家需要拿取所有宝珠，所以在每一关开始时需要统计所有宝珠的数量。如果进行全局搜索的话（比如GameObject.Find()），效率就会比较低，而且每次复活之后都要重新进行搜索，这样可能会给数字的显示带来延迟，玩家体验不够好。本文选择的方式是将宝珠作为一个Prefab，在其挂载的脚本的start()函数中实现一个注册（Register）功能。这样每次关卡初始化或者玩家复活时，每个宝珠就可以自己“主动”地进行注册，从而高效率地确定宝珠的数量。对于其它道具也可以做相似处理。

1. 拿取宝珠和过关的实现

GM类中声明一个全局List，其中存储该场景中的所有宝珠。当玩家拿取该宝珠时，即把其从List中remove掉。同时将信息同步给UI模块，更改屏幕上当前剩余的宝珠数量。当List为空时则抬起大门通往下一关。相关代码实现如下。

if (gameManager.orbs.Contains(orb))

{

gameManager.orbs.Remove(orb);

--gameManager.orbNum;

UIManager.UpdateOrbUI(gameManager.orbNum);

}

if (gameManager.orbNum == 0)

{

gameManager.lockedDoor.openDoor();

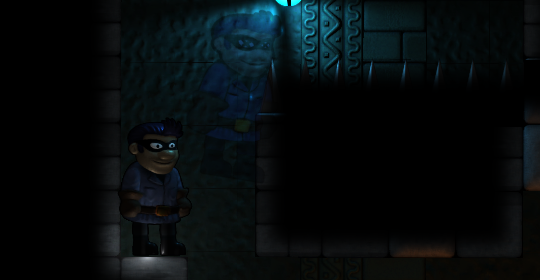
AudioManager.PlayDoorOpenAudio();

}

1. 角色死亡的实现

Unity中的每个GameObject都有属性Tag和Layer。Tag是一对一的，表明你为该物体做了一个独一无二的标记，可以通过它来定位该对象。Layer则是一对多的，有很多GameObject可以同属于一个Layer。只要将所有障碍道具的Layer都设置在同一Layer中，通过判断角色是否碰到了属于该Layer的Collider就可以判断角色是否要死亡。该功能在PlayerHealth脚本中判断，并通过public变量的方式告知GM类。

如下所示，角色死亡后会在原地留下一个映像以提示玩家此前在此处死亡过，提示玩家注意操作。



角色死亡后会设置Time.scale为0，以防止其它效果干扰后续关卡的切换。接下来需要重新开始本关卡，这个功能实现在RestartScene()函数中。本游戏退出关卡时有淡出动画，大概持续1s，所以在角色死亡的1s内画面需要保持静止，等待淡出动画结束，才能刷新整体UI，如恢复宝珠数量和增加已死亡次数等。MonoBehaviour类提供了Invoke()函数可以实现延迟调用函数的功能，但由于此时Time.Scale为0，所以游戏时间是静止的，故无法使用Invoke完成。幸运的是，Unity中有一个很强大的工具：协程。协程本质上是一个函数，由程序员代码进行控制，每次执行到yield关键字停下，下次调用协程函数时则会从上次退出的地方继续开始执行。脚本中加入yield return WaitForSecondsRealtime(1f)，其意义是等待现实世界中的1s之后继续向下执行，这样就完成了淡出时画面的静止效果。

1. 存储游戏和读取游戏的实现

游玩游戏时，游戏程序是打开在内存中的，一旦断电或者退出游戏，相关数据就会消失。为了持久化数据，必然要将数据写到硬盘上。一般有三种方式：二进制写入、json格式写入和key-value格式写入。本文选择二进制写入，这主要是因为需要存储的数据不多，用后两种方式显得有些浪费。实现代码如下，首先打开存储路径的文件流，利用BinaryFormatter的序列化方法将数据转化为二进制写入文，最后关闭文件流，如果catch到异常则输出错误信息并退出。

try

{

int currentSceneID = SceneManager.GetActiveScene().buildIndex;

string dir = Application.dataPath + "/save.txt";

BinaryFormatter bf = new BinaryFormatter();

FileStream fileStream = File.Create(dir);

bf.Serialize(fileStream, currentSceneID);

fileStream.Close();

return true;

}

catch (Exception e)

{

Console.Write("出错！",e.Message);

return false;

}

读取游戏的时候恰好相反，利用BinaryFormatter的反序列化功能将二进制数据还原为有效数据即可。

4.4 障碍模块的实现

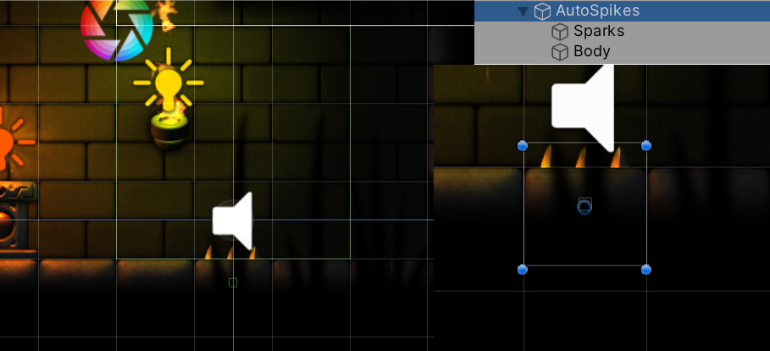
任何闯关游戏中都会有敌人、怪物来阻挡主角前进，主角需要通过击杀或者躲避它们来过关。本游戏中实现了以下几种障碍：角色靠近时会掉落的砖块、自动弹起的地刺、摇摆的摆斧和滚动的锯盘。通过在特定区域放置这些道具，能够显著地提高游戏的操作难度。

触发机制实现

4.4.1 触发机制的实现

用自动弹起的地刺举例，需要达到的效果是这样的：角色远离地刺时，地刺不会弹起。一旦角色进入某个触发范围，则地刺马上弹起。为了达到这样的效果，Unity3D提供了许多方法。比如Physics2D.Overlap函数族，可以返回某个点周围指定范围内是否触碰到某个Layer的物体。本文选择了另一种简单且高效的解决办法。

如下图所示，地刺由两部分构成，第一部分Sparks是一个Partical System，负责弹出时的火花效果。第二部分Body是地刺的实体部分。父对象AutoSpikes上绑定了一个大Box Collider，Body对象上绑定了一个小Box Collider。较大的Collider的is Triger属性是勾选的，较小的Collider不勾选。

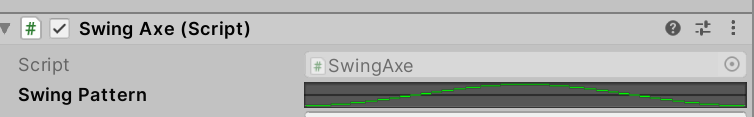


带Trigger和非Trigger的Collider相比：不会和其它Collider产生实际的碰撞，就好像这个Collider不存在一样。普通的Collider碰撞事件通过OnCollisionEnter()和OnCollisionExit()函数来取得状态，而带有Triiger的Collider通过OnTriggerEnter()和OnTriggerExit()函数来取得进入和离开触发范围状态。这样，只需要通过绑定脚本中的OnTriggerEnter()函数来识别进入范围的Collider，如果其Tag是“Player”，则弹起地刺，并把小Collider的size和offset做相应调整来配合地刺贴图的弹出效果。角色离开触发范围时，利用OnTriggerExit()函数识别到角色已经离开触发范围，则放下地刺并调整小Collider。

这种使用两个Collider的方法比较好理解，脚本编写也较简单。本游戏中的另一个障碍道具掉落的砖块也使用了该方法，只不过其Triiger换到了Body的下面。

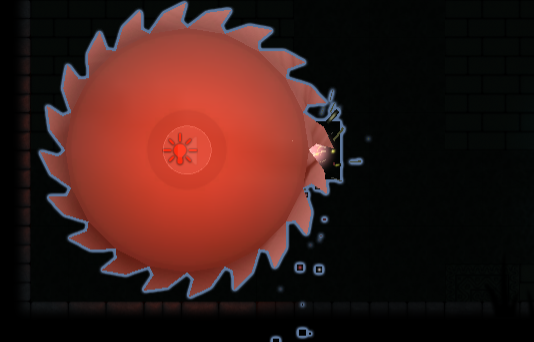
4.4.2 摆斧的实现

摆斧由一个Handle（斧柄）和Axe（斧头）组成，会以固定频率围绕斧柄做钟摆运动。本文对其的实现中使用了AnimationCurve。AnimationCurve是Unity3D提供的一个非常实用的功能，它可以编辑一条任意变化的曲线，能够用于地形调节和绘制轨迹等。如下图所示，其摆动轨迹实际是一个正弦函数，通过调整其相位、频率和振幅就可以改变摆动的轨迹。实际的旋转通过Quaternion.Euler()函数实现，该函数接收一个Vector3类型的值，被调用的对象会按照该值指明的三维角度进行旋转。斧柄是没有Collider的，斧头有一个带Triiger的Polygon Collider，当角色触碰到斧头时就会死亡。



4.4.3 锯盘的实现

如下图所示，锯盘由锯盘本体加上粒子系统以及光源组成。游戏中，锯盘会按照预定路线滚动，如果玩家没有在一定时间内通关就会被锯盘追上，这使得游戏更加的刺激，增加了游戏的挑战性。



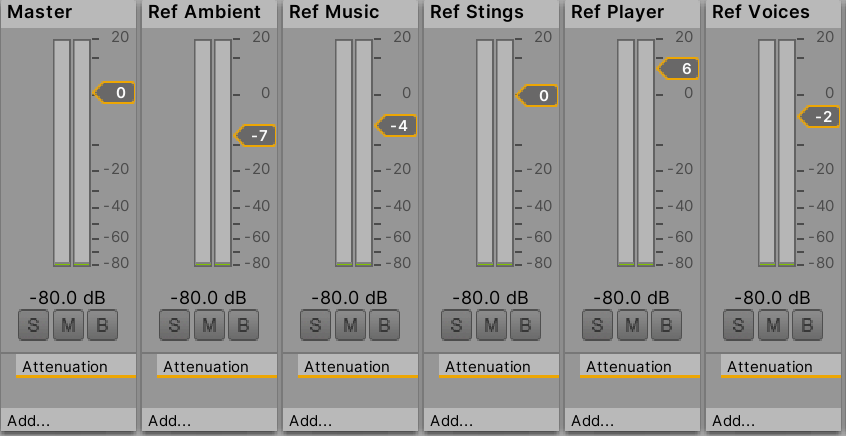
锯盘的碰撞体设置和死亡触发机制和4.4.1节中所述基本相同，滚动机制和4.4.2节中也基本相同，故不予赘述。特殊的是，锯盘会按照预制路径循环移动，到达一个节点会暂停1s，之后提高速度并向下一个节点移动。这样的步步紧逼机制，配合上紧张的音乐和其它效果可以大大提升游戏的紧张感和紧凑感。寻路机制是依靠存储节点坐标的List实现的，锯盘循环按照该List指示的坐标顺序循环移动，通过维护一个当前已经经过的节点序号来记录接下来应该向哪个坐标移动。当锯盘到达一个节点附近时，将锯盘的transform.position设置为该节点的坐标。这是因为在update()函数中判断锯盘是否到达某节点是存在一定误差的，如果不设置误差范围，可能会导致update()中没有抓取到二者坐标重合的瞬间。重设位置也可以减少接下来向下一个节点的移动的误差，这部分在SawFollow脚本中实现，直接挂载在父对象上。

锯盘在移动的过程中，可以摧毁沿途遇到的所有Tile。为了实现此效果，单独命名了一个名为destroyable的Layer，该Layer整体添加了一个Tilemap Collider2D组件。通过OnTriggerEnter()函数，如果识别到遇到了该Layer的Collider，则销毁该Tile。

4.5 音效模块的实现

Unity3D中的声音一般是指Audio Source，其更像是一个播放器，需要给它添加一个Audio Clip（声音片段）才能够播放。一个Audio Source同一时间只能播放一个声音，如果有多个声音需要同时播放，那么就需要多个Audio Source。本游戏中的声音很多，如果全部通过不同的Audio Source挂载在GameObject上的话，非常混乱而且难以管理。所以本文通过设置一个AudioManager（下面简称“AM”）类来对所有声音进行全局管理。AM类同样使用了单例模式，以保证声音设置不会随关卡切换而重置。

诚然可以通过在脚本中生成所有的Audio Source，在需要的时候直接对其Clip进行替换或者更改其音量，但Unity3D中提供了另一个很方便的功能：Audio Mixer，它可以协助我们实现音频管理。Audio Mixer意为混音器，其中含有多条音轨，各音轨之间不发生干涉，可以单独对每个音轨进行控制和调节。如下图所示，在Audio Mixer中添加了六条音轨，分别为Ambient（控制环境音）、Music（背景音乐）、Stings（特效音）、Player（角色移动音效）和Voices（角色语音）。其中Master为主音轨，通过它可以限制所有其它音轨的音量。



通过Expose Parameter功能，可以将每条音轨中的音量参数提取出来，这样就可以通过脚本来控制每条音轨的音量大小。UI模块中的Setting菜单中有五个滑动条，分别对应这五条音轨，每个滑动条的OnValueChanged分别绑定对应的音量参数。这样玩家就可以通过滑动来调节自己喜欢的声音设置。

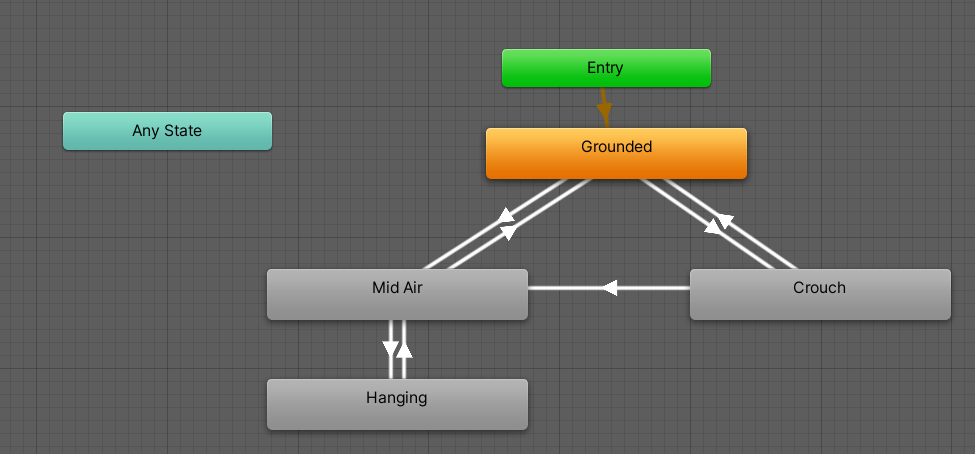
Ambient音轨即风声，music音轨即背景音乐，Stings音轨包括拿取宝珠时的音效，Player音轨包括角色走动和跳跃时发出的声音，Voices音轨则包括拿取宝珠时的欢呼声、角色跑动时的喘气声等。所有这些Audio Clip在需要时会被置入相应的Audio Source，并归入相应的AudioMixerGroup（音轨）中进行播放。

4.6 动画模块的实现

游戏主角是用骨骼动画实现的。模型都是由一个个小三角形组成的，学名叫做“Mesh”。而骨骼则是驱动模型运动的根本。通过绑定两者，即为某块骨骼和相关的Mesh建立联系，并给予不同的权重，就可以让模型动起来。

Unity3D中的动画可以通过两种方式来实现。一种叫Animations，这种方式比较简单，通过指定某个动画进行Play()就可以播放动画。另一种叫Animator，可以通过混合和参数控制来设计更为复杂的动画，并且切换更加平滑。本文选择Animator来实现动画系统。

如下图所示，角色动画一共分为4部分，分别是Grounded（地面姿态），Mid Air（空中姿态）、Crouch（下蹲姿态）和Hanging（悬挂姿态）。它们之间相连的线称为transition，表示从一个动画切换为另一个动画。每个transition都有自己独特的参数值，当某些参数改变时，Animator就会切换到相应的其它动画，否则一直循环播放当前动画。Entry连接的动画表示系统初始化时的默认状态。



其中Mid Air负责空中的动画效果，使用了BlendTree的技术。BlendTree意为混合树，用于允许通过按不同程度组合所有动画的各个部分来平滑过渡多个动画。和Transition的区别是，Transition为定量转化，不存在中间状态，BlendTree则在过渡状态下有多个动画共同组成，各个动画会按照权重大小对整体的呈现效果产生影响。如下所示，通过verticalVelocity变量来控制整个混合树。该变量指示的是角色RigidBody的Y轴Velocity，该值大于0时则角色在跳跃的上升阶段，小于0则位于下落阶段。混合树可以通过读取该变量的值自动切换和过渡动画。另外Grounded也是一个BlendTree，通过一个水平方向的速度变量来平滑过渡走和跑的动画。

