目录：

1. 项目方案

2.1 项目特色

2.2 游戏基本功能及规则介绍

2.3 关卡异步加载技术简介

2.4 Object Pool(对象池)技术简介

2.5 Unity2D动画技术简介

2.6 EasyTouch技术简介

2.7 开发工具及运行环境

1. 项目设计及实现

3.1 系统整体框图

3.2 功能模块的设计与实现

3.2.1 玩家控制模块的设计与实现

3.2.2 技能系统的设计与实现

3.2.3 关卡加载模块的设计与实现

3.2.4 对象池的设计与实现

1. 特殊问题及解决方案

4.1 UI布局问题

4.2 脚本引用丢失问题

4.3 游戏控制器的耦合问题

4.4 进度条过渡生硬问题

**2.1 项目特色**

1) 良好的平台移植性：本项目采用Unity3D作为开发引擎，Unity3D的C#基础脚本模块是通过Mono来实现的，Unity通过Mono将C#脚本代码编译成CLI，然后Mono运行时利用JIT或者AOT将CLI编译成目标平台的原生代码实现跨平台。在构建项目打包过程中，无论基于Android、ios还是webGL等各大主流平台，都能通过Unity直接导出指定文件，如.apk和.xproject项目等，因此从项目开发到构建导出整个过程，都能保持其良好的平台移植性。

2) 丰富的敌人类型：本游戏中敌人类型共6种，包括4种小型飞行器、陨石以及最终Boss，不同敌人类型除了在基本属性，如外观、生命值等方面各不相同外，还具有不同的行为属性，主要包括运动行为与攻击行为。运动行为方面，不同的敌人在移动速度、运动轨迹、出现概率等数值方面均有差异；攻击行为方面，差异主要集中包括在射击概率、发射的子弹类型、射击频率等等。不同敌人无论在状态还是行为上，都各有特色，增强游戏的趣味性和可玩性。

3) 技能系统：游戏中共有4种可供玩家选择的技能，各技能在状态、行为上为玩家提供了不同程度上的增益效果，玩家可选择恰当的时机通过对不同技能进行组合释放，产生各种丰富多彩、意想不到的游戏效果。技能系统作为本游戏中最主要的交互方式，从数值、表现等方面均为游戏增加了更强的表现力和游戏性。此外，本项目根据实际需求编写了一套基于点击触发式的简易技能系统框架，耦合度较低，可复用性强。

**2.2 游戏基本功能及规则介绍**

本游戏是一款基于Android手机平台的飞行射击类游戏。

游戏总共分为3关，各关卡间主要以本关的敌人数量、敌人类型、行为模式、攻击模式，包括可供玩家选择的技能数量、技能类型及技能属性等具体游戏数值和关卡内容上的差异作为区分。3关的难度依次递增，其中前两关需要玩家与小型飞行器进行作战，玩家需要在所有敌人都生成完毕、且完成其攻击行为后依然存活下来，方能进入下一关；第3关中，除了需要对抗小型飞行器外，关卡还将生成一个具有更加丰富的运动模式、攻击模式，并且生命值远远大于其他小型飞行器生命值的大怪Boss，玩家需要在存活的条件下击败大Boss，方能通过本关。而在通过第3关，即最后一关后，玩家将获得最终胜利。

每当玩家进入新的一关，其生命值将被重置为满血，但可用的技能将发生变化，随着关卡难度递增，可选技能的数量、威力值都将得到提升。每个技能需要在拾取到对应的道具才能获得，每次技能释放完毕后，均有指定的冷却时间，在冷却时间内技能无法再次释放，直至冷却时间结束后方能再次激活。

本游戏中敌人类型共6种，包括4种小型飞行器、陨石以及最终Boss，游戏的第一关将出现其中的2种小型飞行器，第二关开始将增加另外两种飞行器和陨石，最后一关将在第二关的基础上增加最终Boss。不同敌人类型除了在基本属性，如外观、生命值等方面各不相同外，还具有不同的行为属性，主要包括运动行为与攻击行为。运动行为方面，不同的敌人在移动速度、运动轨迹、出现概率等数值方面均有差异；攻击行为方面，差异主要集中包括在射击概率、发射的子弹类型、射击频率等等。

**2.3 关卡异步加载技术简介**

1. 背景：

游戏中经常需要进行场景的加载，而有时候有些场景因为需要加载大量的资源，导致在加载场景时感觉很卡；另一方面，在场景后台进行耗时较长的资源加载的过程中，若始终不给予玩家以必要的视觉反馈，玩家将在这段时间内的等待中感到疑惑和不确定，产生糟糕的游戏体验。这时候需要使用异步场景加载来解决这样的问题。

1. 异步加载场景：

**Func**:

public static AsyncOperation LoadSceneAsync(sceneName, mode)

public static AsyncOperation LoadSceneAsync(sceneIndex, mode)

**Parameters**:

sceneName: Name or path of the Scene to load.

sceneBuildIndex: Index of the Scene in the Build Settings to load.

Mode: If LoadSceneMode.Single then all current Scenes will be unloaded before loading.

* 1. 描述：

若调用同步加载场景API：LoadScene(int sceneBuildIndex)，则当前场景中的所有游戏逻辑都将停止，以进行目标场景资源的加载过程，但若调用后台异步加载功能，则当前场景的游戏逻辑正常执行，资源加载过程在后台进行异步执行，并可以实时获取加载进度，还可对是否需要在目标场景资源加载完毕后进行自动跳转。

在后台异步加载场景，给定的场景名称既可以是完整的场景路径，也可以是构建设置窗口中显示的路径，也可以是场景名称或场景下标值。 如果只给出场景名称，则会加载匹配列表中的第一个场景。 如果有多个具有相同名称但路径不同的场景，则应在构建设置中使用完整的场景路径。

* 1. 应用：

通常游戏的主场景包含的资源较多，这会导致加载场景的时间较长。为了避免这个问题，可以首先加载Loading场景，然后再通过Loading场景来加载主场景。因为Loading场景包含的资源较少，所以加载速度快。在加载主场景的时候，由于耗时可能较长，一般会在Loading界面中显示一个进度条来告知玩家当前加载的进度，以获得更佳的游戏体验与视觉效果。在Unity中可以通过调用SceneManager.LoadSceneAsync函数来异步加载游戏场景，通过查询AsyncOperation.progress的值来得到场景加载的进度。

float displayProgress, destProgress;

displayProgress = destProgress = 0;

while (operation.progress < 0.9f) {

destProgress = operation.progress \* 100f;

while (displayProgress < destProgress) {

// 对于当前进度数值，每一帧在实际的基础上+1

loadingBar.transform.Find("LoadingProgText").GetComponent<Text>().text = ++displayProgress + "%";

loadingBar.value = displayProgress / 100f; // 设置进度条数值

yield return new WaitForEndOfFrame();

}

}

**2.4 Object Pool(对象池)技术简介**

在常见的游戏开发过程中，Instatiate() 和 Destroy() 都是十分常用的方法，分别控制着游戏物体的创建与销毁，对单个游戏物体来说，每一次的创建或销毁仅需要耗费极少的CPU时间。但是，对于一些生命周期较短、且每秒可能会被大量次数地销毁的对象来说，频繁的创建与销毁是一个极大的开销，需要占用大量的CPU用时。此外，Unity使用GC(Garbage Collection)，即垃圾回收机制来释放不再被使用的内存，而对Destroy()的重复调用将频繁地触发GC，这将减缓CPU、使游戏运行具有很大的内存隐患，尤其是对于资源相对受限的运行环境，如移动设备和Web平台等。

对象池技术，是以一部分内存为代价，将游戏过程中将频繁使用的对象，进行预先的实例化，并贮存至“池”中。需要时即从“池”中取出，不需要时则放回。有了对象池以后，在整个游戏过程中，就不再需要对对象进行任何的创建和销毁，一切该类对象的使用均从“池”中获取，有效地提高了对象的利用率，将降低了对象的重复创建和销毁所带来的内存损耗的同时，对性能提升有很大的帮助。

另外，虽然对象池能够有效提升对象的利用率，但也以一部分提前分配的内存作为代价，因此对象池不能无上限地存储对象，否则将对内存使用造成爆炸性影响。所以，在使用对象池时，需要对对象池进行上限值的设置，并根据实际情况，对对象池进行动态收缩。

### 2.5 Unity2D动画技术简介

Unity主要使用帧动画技术来制作AnimationClip(动画剪辑)。帧动画技术，就是通过在关键帧设置该帧对应的精灵图片，将多帧连接起来，则形成由图片序列组成的2D帧动画。一般使用方法如下：导入所需动画的精灵图片序列，通过Unity自带的动画编辑器，选取关键帧，并在该帧从导入的动画图片序列中选取并设置欲表现的单张精灵图片，完成对该帧的编辑。对每一帧做相同步骤的操作，再播放此动画剪辑，则形成由精灵图片序列组成的2D帧动画。

在制作完需要的动画剪辑后，不同的动画剪辑还需要进行切换，Unity中通过AnimationCotroller(动画控制器)来对同一物体上的不同动画的切换进行状态的管理和触发条件的设置，以规则的方式实现不同动画的切换和过渡效果。

在创建好动画剪辑及其对应的动画控制器后，还需要为物体添加Animator(动画组件)，并将创建好的动画控制器属性添加至动画组件中。若不同的物体具有相同的动画切换逻辑，但是其实际动画表现又各不相同，则可创建一个Animator Override Controller(动画重写控制器)，类似于一个动画控制器的派生类，通过继承具有目标切换逻辑的动画控制器，可对其中的每个动画进行重写，实现用相同的切换逻辑控制具有不同表现的动画状态。

**2.6 EasyTouch技术简介**

EasyTouch是一款解决移动设备触摸按钮、虚拟摇杆等功能的插件。包含EasyButton、EasyJoystick、EasyTouch三个部分。通过使用该插件，可在场景中不受限制地添加具有移动设备触摸功能的按钮或虚拟摇杆，并且无论最终运行平台的分辨率如何，EasyTouch将在其内部控制分辨率的大小，以保持相同的位置和尺寸。同时，对于通过该插件创建的按钮或虚拟摇杆等交互元素，可自行定义其贴图与交互方式。

本项目中主要利用了EasyJoystick部分的功能，为游戏场景添加一个虚拟摇杆，以满足玩家在移动设备上对player的运动控制。

对于EasyJoystick的使用，在交互模式上，EasyJoystick支持通过四种不同的方法与对象进行交互：

1. Direct：EasyJoystick将直接使用参数控制对象；
2. Event：通过事件的形式，发送带有移动参数的消息通知指定方法当前摇杆的控制情况；
3. Include：可将EasyJoystick对象作为参数集成到脚本中，以直接方位虚拟摇杆的相关属性；
4. Direct Event：Direct模式和Event模式的组合。

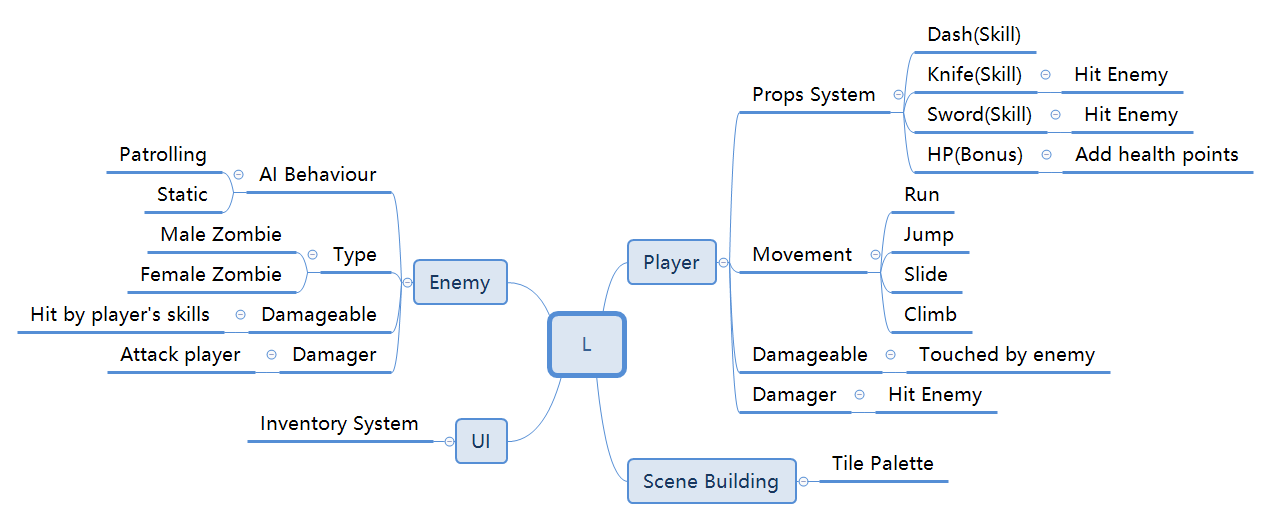
**2.7 开发工具及运行环境**

1. 系统开发工具和开发环境如表2‑1所示。

表2‑1 开发工具表

|  |  |
| --- | --- |
| **开发工具名称** | **软件用途** |
| **Unity5.6.0f** | 开发平台 |
| **Windows 10 Pro** | 运行环境 |
| **应用宝** | 移动端版本打包工具 |
| **XMind** | 系统设计，创建流程图等 |
| **Adobe Photoshop CS6** | 美术素材制作与编辑 |

3.1 系统整体框图



**3.2.1 玩家控制模块的设计与实现**

移动控制：

Player的移动控制部分，在Unity编辑器下采用获取键盘水平和垂直输入来对玩家位置进行移动，但由于本项目导出平台为Android平台，无法读取到键盘输入值，因此需要通过编译指令，根据当前运行环境，执行不同部分的代码。关键代码如下：

#if UNITY\_EDITOR

if (controlIsActive) {

if (Input.GetAxisRaw("Horizontal") != 0 || Input.GetAxisRaw("Vertical") != 0) {

Vector2 keyboardAxis = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));

transform.Translate(keyboardAxis \* moveSpeed \* Time.deltaTime);

ClampPosition();

}

}

#endif

}

上述代码表示若当前运行环境为Unity编辑器，则执行#if与#endif之间的代码，即调用Input.GetAxisRaw(“Horizontal”)来获取键盘的水平轴输入值，调用Input.GetAxisRaw(“Vertical”)获取垂直轴输入值，在Unity的输入设置中，”Horizontal”输入默认对应键盘上的左右方向键或A、D，”Vertical”输入默认对应上下方向键或W、S。在获取到上述两个轴的用户输入值后，利用这两个值创建一个二维向量，由于GetAxisRaw()只返回从-1到1范围内的值，故该二维向量即可代表player的运动方向的单位向量，调用transform.Translate()，将运动方向的单位向量乘以速度值，则控制玩家以特定的速度朝该运动方向上进行平移。

位置限制：

上述代码中最后一行ClampPosition()表示对player位置进行边界限制的方法，关键代码如下：

transform.position = new Vector2

(

Mathf.Clamp(transform.position.x, borders.minX, borders.maxX),

Mathf.Clamp(transform.position.y, borders.minY, borders.maxY)

);

上述代码主要功能在于控制player的位置在一个指定的范围内，防止player运动至相机视角外。调用Mathf.Clamp(value, min, max)，通过传入最小值min和最大值max，将value限制在min与max之间：若value大于max，则返回max；若value小于min，则返回min；否则直接返回value值本身。

其中的min和max值，是根据视口矩形(viewport rect)的顶点值，先从视口坐标转换为世界坐标，再对其进行一部分偏移来确定的。由于视口坐标的范围在(0,0)至(1,1)之间，即视口矩形的左下角至右上角，因此最终的边界值需根据这两个坐标值确定。关键代码如下：

borders.minX = mainCamera.ViewportToWorldPoint(Vector2.zero).x + borders.minXOffset;

borders.minY = mainCamera.ViewportToWorldPoint(Vector2.zero).y + borders.minYOffset;

borders.maxX=mainCamera.ViewportToWorldPoint(Vector2.right).x-borders.maxXOffset; borders.maxY = mainCamera.ViewportToWorldPoint(Vector2.up).y - borders.maxYOffset;

以上四行代码分别表示边界在水平和数值方向上的min和max值，分别通过视口矩形的左下角和右上角的x坐标和y坐标来确定。

**3.2.2 道具与技能系统的设计与实现**

本游戏中释放技能的前提是该技能已被激活，而激活技能的唯一途径是拾取对应的道具。 对于每种道具，需要定义以下两个关键属性：道具对应的技能类型skillType、拾取道具后的事件OnGetSkill<SkillType>。通过碰撞触发器，来检测player对道具的拾取行为，接着触发OnGetSkill(skillType)，并传入本道具对应的技能类型，而该事件上绑定的方法，其功能就是根据接收到的技能类型参数，对该技能进行激活。技能被激活后，才能通过用户输入进行释放。

对于每种技能，需定义关键属性如下：技能类型skillType、冷却时间coolingDuration、是否激活skillEnbaled、技能释放后的回调方法OnFireSkill()、冷却功能方法Cooling()、激活方法EnableSkill()、关闭方法DisableSkill()。上述属性为每种技能都必备的属性，故可抽象为基类SkillControl，使所有技能类均继承于SkillControl，再在其基础上进行重写，由于每种技能释放效果各不相同，故需要将其中的OnFireSkill()声明为抽象方法，每个派生类都必须对其进行实现。所有技能在初始时默认处于关闭状态，而其激活方法EnableSkill()则绑定在道具的拾取事件上，一旦道具被拾取，事件被触发，则该方法被调用，该技能被激活。

关于技能的释放，在交互层面，可创建相应数量的按钮，对各按钮的点击事件做监听处理，每个按钮的监听事件上各绑定一个指定技能的释放方法，当按钮被点击后，若该技能处于激活状态，则触发该技能的释放，调用其冷却方法Cooling()和技能释放后的回调方法OnFireSkill()，使技能表现出其释放效果，并进行冷却。

关于技能的冷却，主要是在UI交互层面做一些视觉上的表现，并通过代码根据相应的冷却时间，控制其在整个过程中的状态变化。具体来说，对每种具有冷却效果的技能来说，在视觉层面，需要三层贴图来实现冷却效果，第一是渲染层级在最上层的，技能本身的主图标，默认状态下的fillAmount为1，进入冷却时间后，立刻将fillAmount置为0，并根据冷却已经过的时间动态改变它的fillAmount，使其再从0变化至1，实现冷却遮罩的填充变化效果；第二是中层半透明的灰色遮罩，在冷却过程中，主图标的fillAmount由0变为1，该遮罩图片相对而言则是由百分百填充变化为0填充；第三是最下层的技能图标，由于技能的冷却效果并非技能图标被遮罩完全遮挡，而是带有不完全透明度的遮挡，因此需要在最下层再垫一张技能图标。

**3.2.3 关卡加载模块的设计与实现**

异步加载：不影响当前游戏场景的前提下加载新场景。通常异步加载的方式分为两种：第一种是异步加载新游戏场景，当新场景加载完成后进入新场景并且销毁之前的场景。第二种，同样异步加载新场景，新场景加载完毕后，保留旧场景的游戏对象并且进入新场景。本项目使用第一种方式进行场景的异步加载。

加载界面：通常游戏的主场景包含的资源较多，这会导致加载场景的时间较长。为了避免这个问题，可以首先加载Loading场景，然后再通过Loading场景来加载主场景。因为Loading场景包含的资源较少，所以加载速度快。在加载主场景的时候一般会在Loading界面中显示一个进度条来告知玩家当前加载的进度。

在Unity中可以通过调用SceneManager.LoadSceneAsync函数来异步加载游戏场景；进度条：通过查询AsyncOperation.progress的值来得到场景加载的进度。通过Slider组件实现加载界面。

float displayProgress, destProgress;

displayProgress = destProgress = 0;

while (operation.progress < 0.9f) {

destProgress = operation.progress \* 100f;

while (displayProgress < destProgress) {

// 对于当前进度数值，每一帧在实际的基础上+1

loadingBar.transform.Find("LoadingProgText").GetComponent<Text>().text = ++displayProgress + "%";

loadingBar.value = displayProgress / 100f; // 设置进度条数值

yield return new WaitForEndOfFrame();

}

}

destProgress = 100f;

while (displayProgress < destProgress) {

loadingBar.transform.Find("LoadingProgText").GetComponent<Text>().text = ++displayProgress + "%";

loadingBar.value = displayProgress / 100f;

yield return new WaitForEndOfFrame();

}

operation.allowSceneActivation = true;

**3.2.4 对象池的设计与实现**

首先设计对象池框架必需的管理模块与基类：PoolManager, PoolSpawner, PoolObject。

PoolManager：对所有存放着不同类型对象的对象池，进行全局统一的管理。其中的关键属性：

对象池字典poolDictionary<int, Queue<ObjectInstance>>：每条记录的键值对的含义为：key表示预制体的实例id，由于每个对象池中存放的对象实体不同，该id也可看作是对象池id；value表示该对象池队列，存放着池中所有对象。池中每个对象以队列的结构进行存放，根据FIFO的原则，最先被使用的对象也应最先归还至对象池。

创建对象池方法CreatePool(GameObject prefab, int poolSize)：该方法接收两个参数：对象预制体和对象池大小，prefab即创建的对象模版，poolSize规定该对象池的分配上限，防止出现内存急剧增加的状况。创建对象池的过程主要分两步：将该对象池所代表的键值对，作为一条记录存入对象池字典中；根据poolSize，实例化指定数量的对象，并一一入队。

重用对象方法ReuseObject(GameObject prefab, Vector3 position, Quaternion rotation)：该方法接收三个参数：对象预制体、实例化位置和实例化旋转量。该方法主要负责对指定对象的重用，由于对象池中的所有对象均以队列结构存储，因此在玩家需要“创建”更多的对象实例时，调用该方法，就会从该对象所属对象池的队列中，将最先被使用的对象出队再入队，模拟将该对象放回对象池中再次使用的过程。

PoolSpawner：对PoolManager中的CreatePool()方法的封装，通过在该类声明对象池信息：prefab和poolSize，再调用PoolManager.CreatePool(prefab, poolSize)创建对象池。

PoolObject：该类添加在每个对象池中的对象上，类内主要是声明了两个虚方法：OnObjectReuse()和Destroy()。OnObjectReuse()是对象被重用后的回调，可用于对象每次重用后相关属性的重置等；Destroy()是在不需要该对象显示时，对其进行隐藏，实际上并未真正销毁该对象，而只是调用了gameObject.SetActive(false)使其在游戏场景中不可见。

**4.1 UI布局问题**

4.1.1问题描述

若将游戏分辨率切换至另一不同于初始设置的分辨率，则UI布局杂乱或不可见。

4.1.2解决方案

1)自适应：游戏中的分辨率自适应主要包括两部分：一是在不同尺寸的屏幕下，整体缩放比例的计算方式；二是在不同比例（宽高比）的屏幕下，UI控件所处的位置关系，也称为布局。

2)UGUI缩放机制：

a)Constant Pixel Size：固定像素尺寸，即按素材的“固定像素”渲染；

i.Scale Factor：缩放比例，在素材原尺寸上的缩放比例，默认值是1；

ii.Reference Pixels Per Unit：每个unity单位对应的像素数；

b)Scale With Screen Size：根据屏幕尺寸缩放；

i.Reference Resolution：标准分辨率，这是我们提供给美术做图的标准分辨率，所有的UI素材都应该按这个分辨率去做；

ii.Screen Match Mode：Match Width Or Height 以宽高权重匹配；

iii.Match：宽高所占权重，默认值是0，相当于以“标准分辨率的宽”和“实际屏幕的宽”的比例作为缩放比例。同理，如果值是1，相当于以“标准分辨率的高”和“实际屏幕的高”的比例作为缩放比例。如果值是0.5，则相当于宽和高的比例权重相等，最终的缩放比=宽缩放比\*宽权重+高缩放比\*高权重；

iv.Reference Pixels Per Unit：每个unity单位对应的像素数；

c)Constant Physical Size：固定物理尺寸；

i.Physical Unit：物理单位，包括点，英寸，厘米，毫米等；

ii.Fallback Screen DPI：对应物理单位的像素密度；

iii.Default Sprite DPI：默认精灵的像素密度；

iv.Reference Pixels Per Unit：每个unity单位对应的像素数

3)使用方法：

a.Canvas Scaler 选择 Scale With Screen Size；

b.Screen Match Mode 选择 Match Width Or Height，比例设为1，即只和高度进行适配；

4.1.3结果

在不同分辨率下，UI元素可根据实际情况进行自适应：

1)保持相对位置不变；

2)保持宽高比例不变。

**4.2脚本引用丢失问题**

4.2.1问题描述

对脚本资源进行修改后，部分游戏物体上的某些组件提示“Nothing Selected”，脚本引用发生丢失。导致子弹爆炸效果未能正常销毁。

4.2.2解决方案

将版本回退至对脚本资源进行修改前的正常版本，Unity将自动导入先前丢失的资源与引用。

4.2.3结果

引用被重新定位，组件恢复正常，爆炸效果贴图被正常销毁。

**4.3进度条过渡生硬问题**

4.3.1问题描述

通过LoadSceneAysnc进行异步加载时，进度条的数值更新不连续的，效果生硬不自然。

4.3.2解决方案

每一次更新进度条的时候插入过渡数值。当获得AsyncOperation.progress的值后，不立即更新进度条的数值，而是每一帧在原有的数值上加1，这样就会产生数字不停滚动的动画效果。

while (operation.progress < 0.9f) {

destProgress = operation.progress \* 100f;

while (displayProgress < destProgress) {

// 对于当前进度数值，每一帧在实际的基础上+1

loadingBar.transform.Find("LoadingProgText").GetComponent<Text>().text = ++displayProgress + "%";

loadingBar.value = displayProgress / 100f; // 设置进度条数值

yield return new WaitForEndOfFrame();

}

}

4.3.3 结果

经过插值操作之后，进度条数值每一帧都在实际加载进度的范围内进行更新，过渡自然，实现了柔和的动画效果。

**4.4 对象池自行重用问题**

4.4.1 问题描述

对于用到了对象池技术的陨石物体，需要其在触碰到场景边界后自动重用，重新从场景最顶端的一个位置范围内随机出现，但实际情况下，陨石触碰到边界后未能自行重用，而是一直不停向下运动。

4.4.2 解决方案

主要问题在于陨石物体用到了Rigidbody组件，受物理引擎驱动，在不对其运动属性进行干预的情况下将一直保持原来的运动状态。因此，在触发重用回调方法时，需先重置其运动状态，即直接设置其运动速度和角速度均为0，再进行重用方法的调用，问题即得到解决。主要代码如下：

public override void OnObjectReuse() {

ResetMotion();

base.OnObjectReuse();

}

private void ResetMotion() {

transform.rotation = Quaternion.identity;

GetComponent<Rigidbody>().angularVelocity = Vector3.zero;

GetComponent<Rigidbody>().velocity = Vector3.zero;

}

4.4.3 结果

陨石正常运动，并在接触到边界时自动重置其物理状态，再进行自动重用，实现陨石无限循环随机出现的效果。