**《数字媒体技术综合实训》**

**项目指导书**

**数字媒体技术系**

# 实训项目1 Photon开发框架案例

## 任务1 PUN安装和设置

### 任务内容

1. 完成PUN2网络开发框架的下载与安装；

2. 在Unity中设置PUN2连接参数；

### 提前准备

1. 下载初始项目puntest\_start.zip；

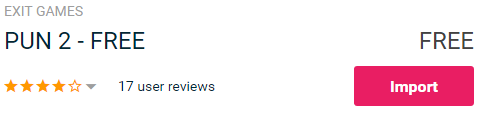
2. 解压缩，并在Unity中打开这个初始项目。

### 任务步骤

1. 在Unity资源商店中下载PUN2

1）打开Unity项目，然后转到资产存储：（window->general->asset store）或按ctrl+9；

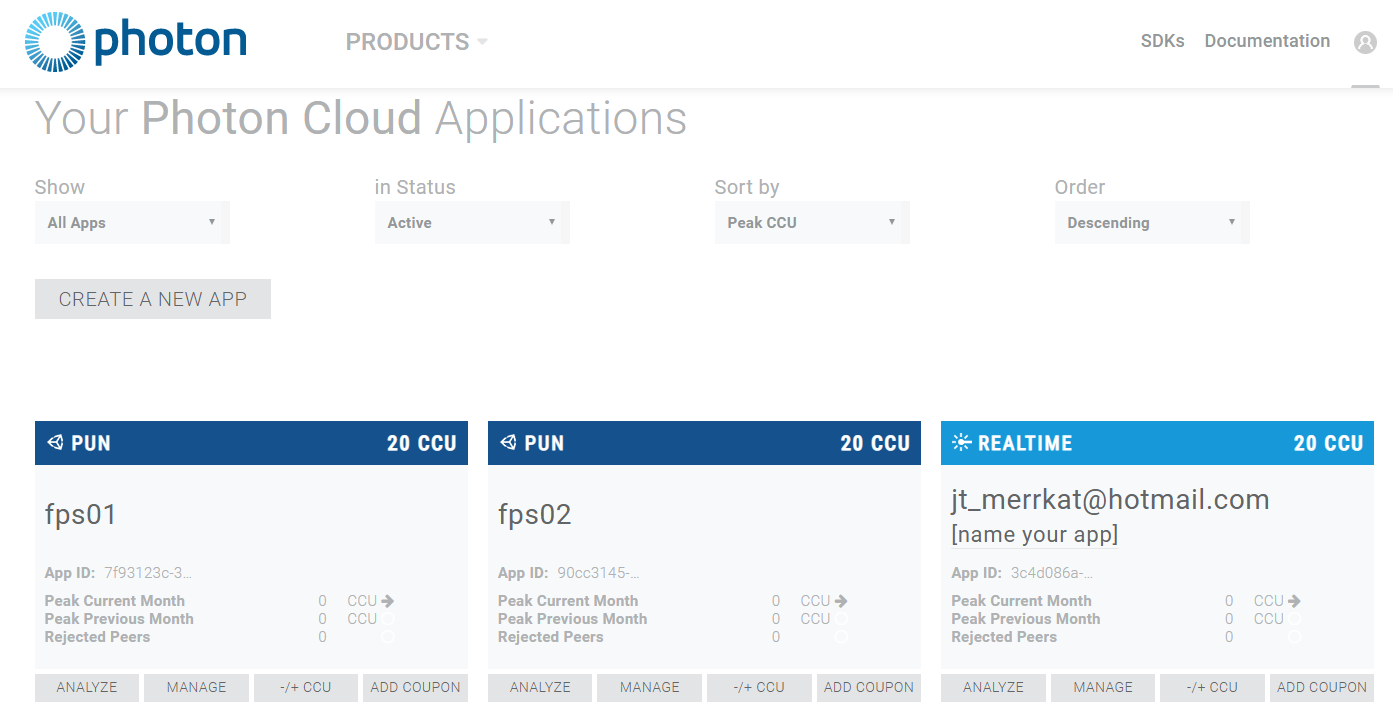
2）搜索“pun 2-free”，然后单击第一个结果或单击此处；



3）下载完成后导入pun 2包。

2.进行pun2联网参数配置

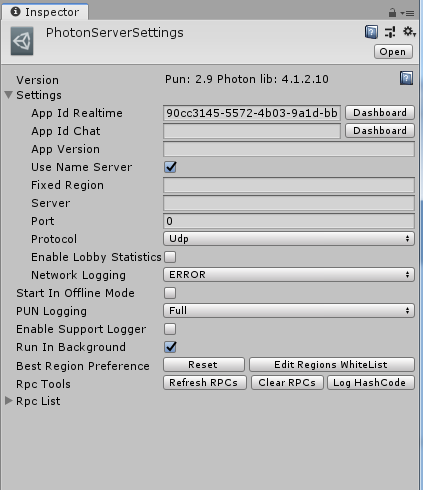
1）在[https://www.photonengine.com/](https://www.photonengine.com/" \t "_blank)网站上找到对应的pun项目，如下图所示：



2）将appid保存在PUN Setup中，如下图所示：



3）也可选择Window -> Photon Unity Networking -> PUN Wizard填入，如下图所示：



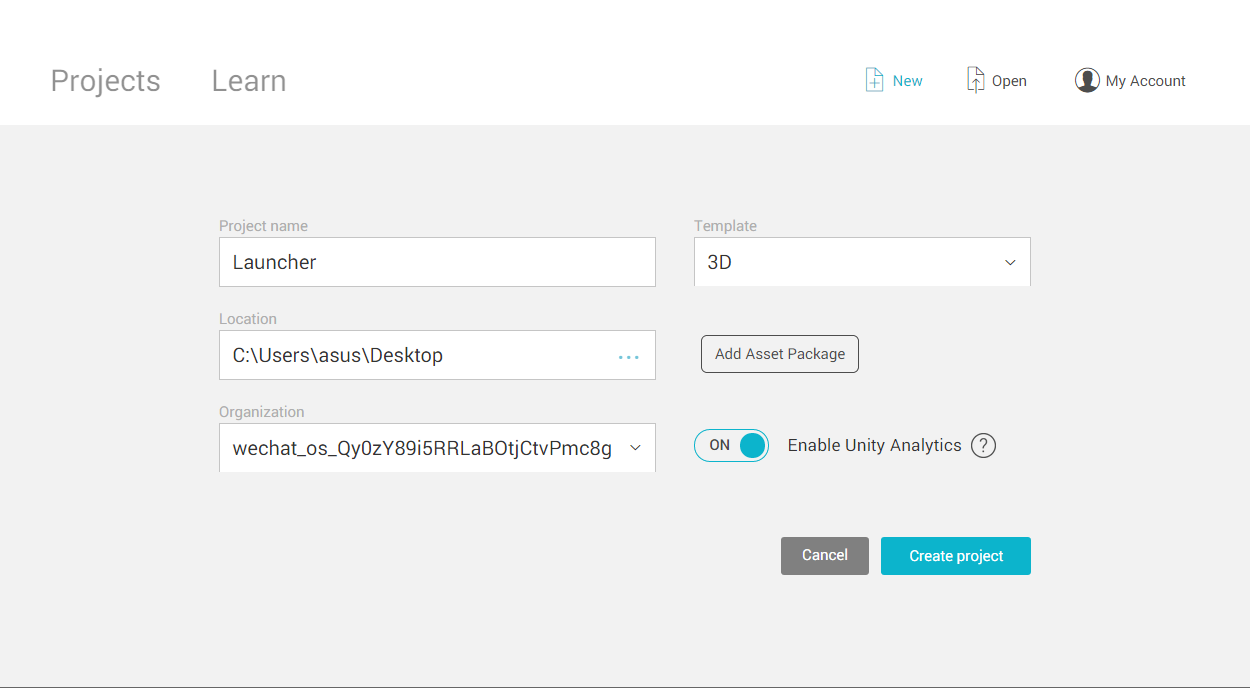
## 任务2 建立大厅

### 任务内容

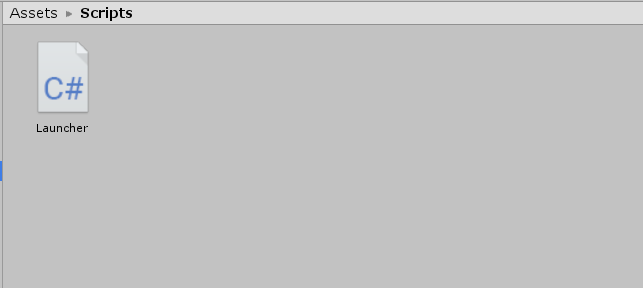
1. 连接服务器
2. 房间的访问和创建

### 提前准备

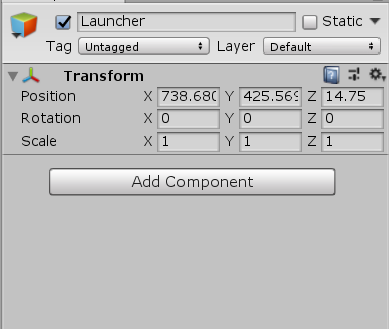
1. 创建一个新的场景.保存为Launcher.unity



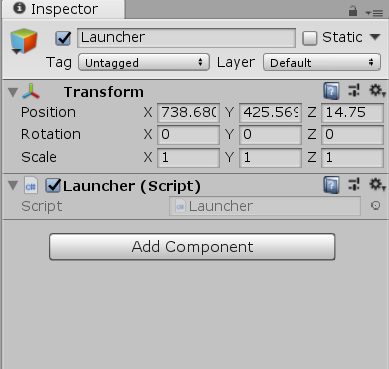
1. 创建新的C#脚本Launcher



1. 创建新的GameObject，命名为Launcher



1. 把C#脚本添加到Launcher对象上



### 任务步骤

1.按照以下的内容编辑C#脚本

Using System.Collections;

Using System.Collections.Generic;

using UnityEngine;

using Photon.Pun;

namespace Com.MyCompany.MyGame

{

public class Launcher : MonoBehaviour

{

string gameVersion = "1";

void Awake()

{

PhotonNetwork.AutomaticallySyncScene = true;

}

void Start()

{

Connect();

}

public void Connect()

{

if (PhotonNetwork.IsConnected)

{

PhotonNetwork.JoinRandomRoom();

}

else

{

PhotonNetwork.GameVersion = gameVersion;

PhotonNetwork.ConnectUsingSettings();

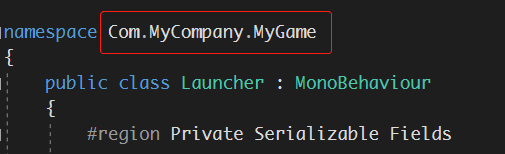
}

}

}

}

<1>Namespace



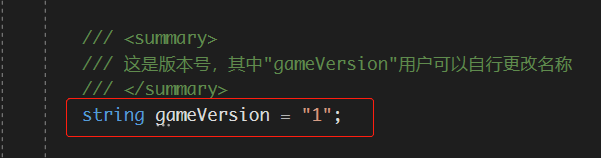
虽然不是强制性的，为脚本提供适当的命名空间可防止与其他Assets和开发人员发生冲突。 如果另一个开发者创建一个类Launcher怎么办？ Unity编译器会报错，你们必须重命名这个的类。 如果冲突来自您从资源商店下载的Assets，这可能很棘手。 现在，Launcher类实际上是在Com.MyCompany.MyGame.launcher这个命名空间下，别人不太可能使用与我们这个完全相同的命名空间，因为你拥有这个域名，所以使用域名逆序作为命名空间，可以使您的工作安全，组织良好。 Com.MyCompany.MyGame应该被替换为你自己的逆序域名和游戏名称，应该遵从这个良好的约定。

<2>MonoBehaviour类



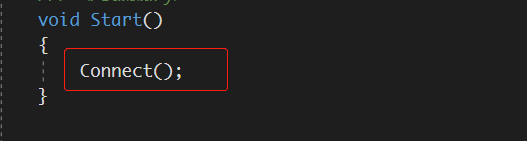
注意，我们使用MonoBehaviour派生类，它可以将我们的类转换为Unity组件，然后可以放到GameObject或Prefab上。 扩展MonoBehaviour的类可以访问许多非常重要的方法和属性。 在这里，我们将使用两个回调方法，Awake()和Start()。

<3>PhotonNetwork.GameVersion:（光子网络游戏版本）



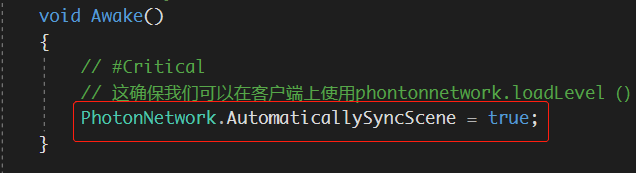
gameVersion变量表示你的游戏版本。 应该将其保留为“1”，直到需要对已经处于活动状态的项目进行大的修改。

<4>PhotonNetwork.ConnectUsingSettings()



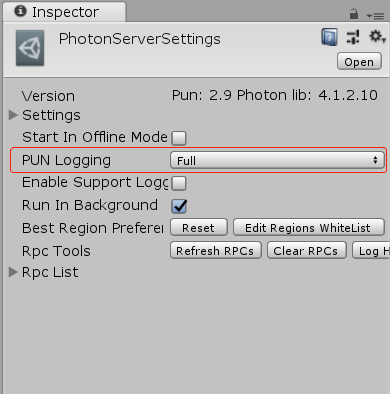
在start（）期间，我们调用公共函数connect（），该函数调用此方法。这里需要记住的重要信息是，该方法是连接光子云的起点。

<5>PhotonNetwork.AutomaticallySyncScene



我们的游戏将有一个根据玩家数量可调整大小的房间，为了确保加载的场景对每个连接的玩家都是相同的，我们将利用光子提供的非常方便的功能：photonnetwork.automaticallySyncScene。当该变量设置为true的时候，MasterClient可以调用PhotonNetwork.LoadLevel()，此时所有连接的玩家都会自动的加载同样的房间

<6>设置日志



此时，您可以保存启动程序场景并打开光子服务器设置（****Window/Photon Unity Networking/Highlight Photon Server Settings****），我们需要将pun日志设置为”full”：

运行程序，就会在Unity控制台中看到一些日志，其中一个应该通知Received your UserID from server”。这已经是连接过程成功的好迹象。我们将很快看到一些其他可能的通知。

编码时的一个好习惯是总是测试潜在的失败。这里我们假设计算机连接到Internet，但是如果计算机没有连接到Internet，会发生什么？让我们看看。关闭计算机上的Internet并播放场景。您应该在Unity控制台中看到这个错误：



理想情况下，我们的脚本应该知道这个问题，并对这些情况作出反应。并且无论什么情况或问题可能出现，都要能够积极响应。

我们现在处理这两种情况，并通知我们的Launcher脚本，我们到底有没有连接上PUN服务器。 这将是对PUN Callbacks的完美介绍。

1. PUN CallBacks:

PUN的回调非常灵活，并提供了两种非常不同的实现。 让我们学习所有的方法，然后根据情况选择使用最适合的一个。

实现回调接口:

你可以在你的类中使用PUN提供的C#接口

IConnectionCallbacks: 连接回调接口

IInRoomCallbacks:房间回调接口

ILobbyCallbacks: 大厅回调接口

IMatchmakingCallbacks: 配对回调接口

IOnEventCallback: 任意事件单个回调接口

IWebRpcCallback: 用于接收WebRPC操作响应的单个回调

IPunInstantiateMagicCallback: 对实例化的PUN预制体的单个回调

IPunObservable: 光子视图序列化回调

IPunOwnershipCallbacks: PUN所有权转让回调

**回调接口必须注册和注销。调用phontonnetwork.addCallbackTarget（this）和phontonnetwork.removeCallbackTarget（this）（可能分别在onEnable（）和onDisable（）中）**

这是确保类符合所有接口的非常安全的方法，但强制开发人员实现所有接口声明。好的IDE将使这个任务非常容易。然而，脚本最终可能会得到很多可能方法，但是必须实现所有的方法才能让Unity编译器满意。所以这就是你的脚本将大量使用所有或大多数PUN功能的时候。

我们确实将使用ipunobservable，在本教程的进一步内容中进行数据序列化。

Extending MonoBehaviourPunCallbacks

另一种技术，我们将经常使用的技术，是最方便的。我们不创建从MonoBehavior派生的类，而是从MonoBehaviorPuncallbacks派生该类，因为它公开了特定的属性和虚拟方法，以便我们方便地使用和重写。这是非常实用的，因为我们可以确保我们没有任何错别字，而且我们不需要实现所有方法。

注意：当重写时，大多数IDE默认情况下将实现一个基调用，并自动为您填充它。在我们的例子中，我们不需要这样做，所以作为MonoBehaviorPuncallbacks的一般规则，除非重写OnEnable（）或OnDisable（），否则永远不要调用基方法。如果重写OnEnable（）和OnDisable（），则始终调用基类方法。

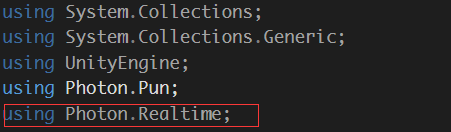
那么，让我们将这个应用于onConnectedTomaster（）和onDisconnected（）pun回调

<1>创建C#脚本Launcher

<2>将基类从MonoBehavior修改为MonoBehaviorPuncallbacks

public class Launcher : MonoBehaviourPunCallbacks{

<3>使用photon.realtime；添加到类定义之前的文件顶部。



<4>为了清晰起见，在类的末尾添加以下两个方法，即区域内的MonoBehaviorPuncallbacks回调。

public override void OnConnectedToMaster(){

Debug.Log("pun基础教程/启动程序：OnConnectedToMaster()由PUN调用");}

public override void OnDisconnected(DisconnectCause cause){

Debug.LogWarningFormat("pun基础教程/启动程序： onDisconnected（）被pun{0}调用", cause);}

<5>保存脚本

现在，如果我们在有或没有互联网的情况下播放这个场景，我们可以采取适当的步骤通知播放器和/或进一步进入逻辑。当我们开始构建UI时，我们将在下一节中处理这个问题。现在我们将处理成功的连接，因此，我们将以下调用附加到onConnectedToMaster（）方法：

PhotonNetwork.JoinRandomRoom();

如果尝试加入一个随机房间失败，我们需要得到通知，在这种情况下，我们需要实际创建一个房间，因此我们在脚本中实现onJoinRandomFailed（）pun回调，并使用photonNetwork.createRoom（）创建一个房间，您已经猜到了，相关的pun回调onJoinedRoom（）将通知您的脚本。当我们有效地加入一个房间时：

public override void OnJoinRandomFailed(short returnCode, string message){

Debug.Log("pun基础教程/启动程序：:OnJoinRandomFailed()由 PUN调用.没有可用的随机房间，随机创建一个 \n 回调: PhotonNetwork.CreateRoom");

PhotonNetwork.CreateRoom(null, new RoomOptions());}

public override void OnJoinedRoom(){

Debug.Log("pun基础教程/启动程序：onJoinedRoom（）由pun调用。 进入房间");}

现在，如果您运行场景，您应该按照逻辑顺序完成连接到pun、尝试加入现有房间，或者创建一个房间并加入新创建的房间。

在本教程的这一点上，由于我们现在已经讨论了连接和加入房间的关键方面，有一些事情不太方便，需要尽早解决。这些与学习PUN没有真正的关系，但从整体上来说很重要。

Expose Fields in Unity Inspector

你可能已经知道这一点，但是如果你不知道，单性行为可以自动地将字段暴露给统一检查器。默认情况下，所有公共字段都是公开的，除非它们标记为[HideInInspector]。如果我们想公开非公共字段，我们可以使用属性[serializefield]。这是Unity中一个非常重要的概念，在我们的例子中，我们将修改每个房间的最大玩家数，并在Inspector中公开它，这样我们就可以在不接触代码本身的情况下设置它。

我们将对每个房间的最大玩家数进行同样的操作。在代码中硬编码并不是最佳实践，相反，让我们把它作为一个公共变量，这样我们以后就可以决定和玩弄这个数字，而不需要重新编译。

在类声明的开头，在私有可序列化字段区域中，让我们添加：

[Tooltip("The maximum number of players per room. When a room is full, it can't be joined by new players, and so new room will be created")][SerializeField]private byte maxPlayersPerRoom = 4;

然后我们修改phontonnetwork.createroom（）调用并使用这个新字段，而不是我们以前使用的harcoded数字。

PhotonNetwork.CreateRoom(null, new RoomOptions { MaxPlayers = maxPlayersPerRoom });

所以，现在我们不强制脚本使用静态的maxplayers值，我们只需要在Unity检查器中设置它，然后点击Run，不需要打开脚本，编辑它，保存它，等待Unity重新编译并最终运行。这样做更有效率和灵活性。

## 任务3 大厅界面

### 任务内容

1.play按钮

2.玩家名字

3.连接进度

### 提前准备

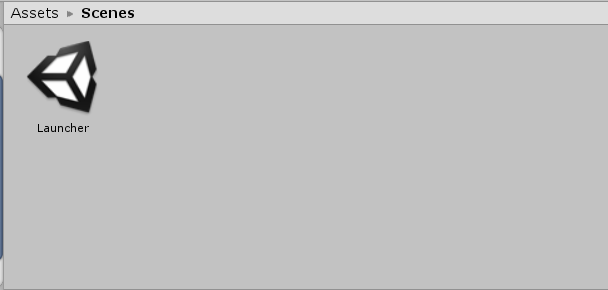
1.打开unity项目文件

2.打开Launcher脚本

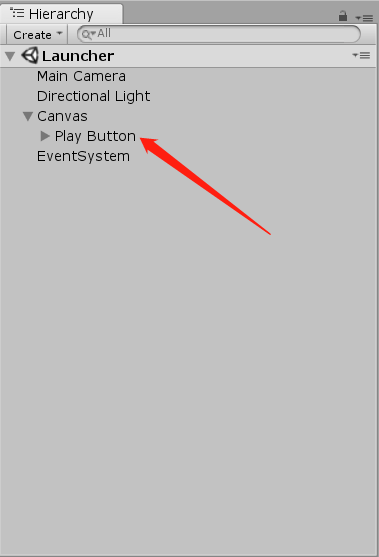
### 任务步骤

1. Play按钮

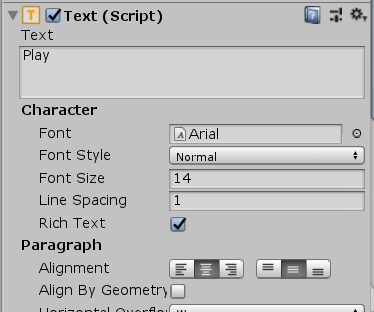
<1>打开场景Launcher



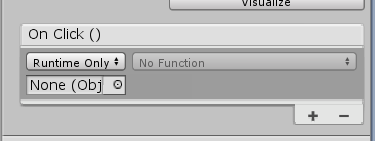
<2>创建命名”PlayButton”的按钮



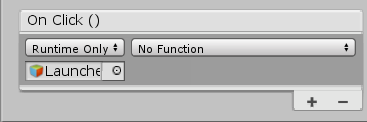
<3>编辑Play Button的Text值为“Play”



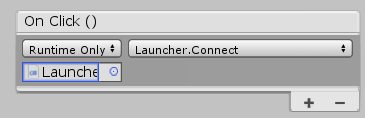
<4>选择Play Button然后定位到按钮组建的On Click()部分



<5>把Launcher对象从Hierachy面板中拖过来



<6>在下拉列表中选择Launcher.connect()函数。我们现在把按钮和Launcher脚本连起来了，当用户点击按钮的时候，将会调用Launcher脚本的connect()函数



<7>打开Launcher脚本,把Start()函数中的connect()函数删除,保存Launcher脚本和场景

现在点击Play，你会发现需要点击按钮才能进行连接了。

1. 玩家名字

典型游戏的另一个重要的最低要求是让用户输入他们的名字，以便其他玩家知道他们正在和谁玩。 我们将实现这个简单的任务，通过使用PlayerPrefs记住名字，以便当用户打开游戏，我们可以知道名字是什么。 要为您的游戏创建一个伟大的用户体验的话，这是一个非常方便和相当重要的功能。

让我们先创建一个脚本来管理和记住玩家的名字，然后创建相关的UI。

<1>创建玩家名字输入框

1. 创建一个新的C#脚本，命名为PlayerNameInputField
2. 以下是脚本的全部内容，编辑然后保存

using UnityEngine;using UnityEngine.UI;

using Photon.Pun;using Photon.Realtime;

using System.Collections;

namespace Com.MyCompany.MyGame{

[RequireComponent(typeof(InputField))]

public class PlayerNameInputField : MonoBehaviour

{

const string playerNamePrefKey = "PlayerName";

void Start () {

string defaultName = string.Empty;

InputField \_inputField = this.GetComponent<InputField>();

if (\_inputField!=null)

{

if (PlayerPrefs.HasKey(playerNamePrefKey))

{

defaultName = PlayerPrefs.GetString(playerNamePrefKey);

\_inputField.text = defaultName;

}

}

PhotonNetwork.NickName = defaultName;

}

public void SetPlayerName(string value)

{

if (string.IsNullOrEmpty(value))

{

Debug.LogError("Player Name is null or empty");

return;

}

PhotonNetwork.NickName = value;

PlayerPrefs.SetString(playerNamePrefKey,value);

}

}}

1. 分析脚本

·RequireComponent(typeof(InputField))

我们首先要确保这个脚本强制执行inputfield，因为我们需要它，这是一个非常方便和快速的方法，可以保证这个脚本的无故障使用。

·PlayerPrefs.HasKey(), PlayerPrefs.GetString() and PlayerPrefs.SetString():

PlayerPrefs是一个简单的配对条目的查找列表（像一个excel表有两列），一个是键，一个是值。 Key是一个字符串，是完全任意的，你决定如何命名，你需要在整个开发过程中记住它。因此，有必要总是将PlayerPrefs键存储在一个地方，一个方便的方法是使用Static变量声明，因为它不会随着时间的推移在游戏过程中改变，并且每次都是相同的。

所以，逻辑非常简单。如果PlayerPrefs有一个给定的键，我们可以得到它，并当我们要用的时候直接赋值。在我们的案例中，我们在启动时填充InputField时，在编辑过程中，我们把当前InputField的值设置给PlayerPref键，然后我们确定它被存储在用户设备上以供稍后检索（下一次用户打开此游戏）。

·PhotonNetwork.NickName

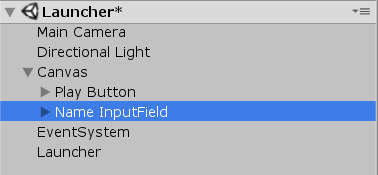
这是这个脚本的重点，通过网络设置玩家的名称。该脚本在两个地方使用它，一个是在start（）期间，在检查名称是否存储在playerrefs中以及在public方法setplayername（）中之后。现在，没有人调用这个方法，我们需要绑定inputfield onValueChange（）来调用setPlayerName（），这样每当用户编辑inputfield时，我们都会记录它。我们只能在用户按下Play键时执行此操作。它还意味着，当用户键入信息时都会被记录下来。

<2>为玩家的名字创建UI

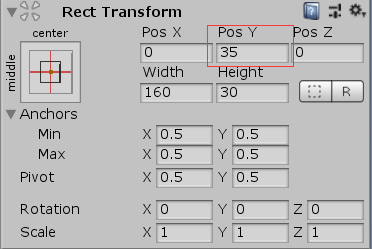
1. 打开Launcher场景



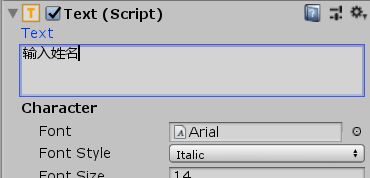
1. 使用Unity菜单'GameObject/UI/InputField'创建InputField，命名为Name InputField



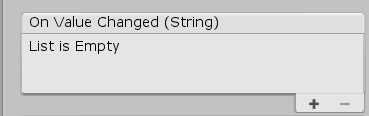
1. 把RectTransform中的PosY值设置为35，它会在PlayButton的上面



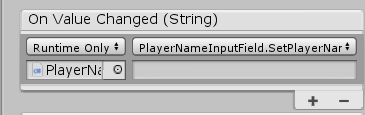
1. 定位到Name InputField的子对象PlaceHolder，设置它的文本值为"输入姓名"



1. 选择Name InputField对象
2. 把我们刚才创建的PlayerNamerInputField脚本给它加上
3. 定位到InputField组件的On Value Change (String)部分



1. 把PlayerNamerInputField拖拽到框里
2. 下拉列表中选择PlayerNameInputField.SetPlayerName()



1. 保存场景

好了，你可以点击play运行，输入你的名字，然后停止play，再次点击play启动，你刚才输入的名字就会有了。

我们实现了功能，然而在用户体验方面，我们缺少连接进度的反馈，还缺少当连接期间和加入房间时出现问题时的反馈。

1. 连接进度

我们在这里尽量保持简单，隐藏名称字段和play按钮，并在连接期间将其替换为简单的文本“正在连接...”，并在需要时将其切换回来。

为此，我们把播放按钮和名称字段做成一个组，以便我们只需要激活和停用该组。 后来更多的功能可以添加到组，它不会影响我们的逻辑。

<1>使用unity菜单'GameObject/UI/Panel'创建UI面板，命名为Control Panel

<2>删除Control Panel的Image和Canvas Renderer组件，我们不需要任何可视元素，我们只关心它的内容

<3>把Play Button 和 Name InputField拖拽到Control Panel对象上去

<4>使用unity菜单'GameObject/UI/Text'创建UI文字，命名为Progress Label，不用关心它影响了显示，我们将在运行时激活和停用它们

<5>选择Progress Label的Text组件

<6>设置对齐方式为center align和middle align

<7>设置文字为“Connecting...”

<8>设置颜色为白色或者其他和背景有区别

<9>保存场景

此时，为了测试，您可以简单地启用/禁用Control Panel和Progress Label，以查看各种连接阶段的情况。 现在让我们编辑脚本以控制这两个GameObjects激活。

<1>在Public Properties区块添加下面两个属性

<2>把下面的代码加入Start函数

<3>在Connect方法开头添加下面的代码

<4>在OnDisconnectedFromPhoton方法开头添加下面的代码

<5>保存Launcher脚本,确保打开场景Launcher

<6>在Hierarchy中选中Launcher对象

<7>把Hierarchy中的Control Panel和Progress Label拖拽到对应的Launcher中的组件

<8>保存场景

现在，如果你运行场景。 您将看到只有控制面板，可见，一旦您单击播放，将显示Progres标签。

到此为止，我们做好了Lobby部分。 为了进一步增加Lobby的功能，我们需要切换到游戏本身，并创建各种场景，以便我们可以在加入房间时最终加载正确的级别。 我们将在接下来的部分完成，然后，我们将完成Lobby系统。

## 任务4 创建游戏场景

### 任务内容

1.第一个房间创建

2.Game Manager

3.退出房间按钮

4.创建其他房间

5.Build Settings 场景列表

### 提前准备

### 任务步骤

1. 第一个房间的创建

<1>首先创建一个新的场景，保存，命名为“Room for 1”

<2>创建一个Cube然后命名为floor

<3>把它放到0,0,0.这很重要，因为我们的逻辑系统将会在中心位置上方(0,x,0)生成玩家

缩放floor到20,0,20

这样的场景肯定能够玩了，但是需要一些墙壁将保持玩家在地板区域内。 只需创建更多的立方体和位置，旋转和缩放作为墙壁。下面是所有四个墙壁的位置和缩放，匹配floor对象。

不要忘记保存Room for 1场景。

2.Game Manager Prefab

<1>创建一个新的C#脚本GameManager

<2>创建一个空的游戏对象，命名为Game Manager

<3>把GameManager脚本拖拽到Game Manager对象上

<4>把Game Manager对象从场景Hierarchy拖拽到Assets里面转成prefab，它会变成蓝色

<5>编辑GameManager脚本

<6>替换为如下脚本

using System;using System.Collections;

using UnityEngine;using UnityEngine.SceneManagement;

using Photon.Pun;using Photon.Realtime;

namespace Com.MyCompany.MyGame{

public class GameManager : MonoBehaviourPunCallbacks

{

public override void OnLeftRoom()

{

SceneManager.LoadScene(0);

}

public void LeaveRoom()

{

PhotonNetwork.LeaveRoom();

}

}}

<7>保存GameManager脚本

所以，我们创建了一个公共方法LeaveRoom()。它的作用是明确地让内部玩家离开Photon Room，我们将它放在Public Methods区块。我们可能希望在稍后阶段实现更多功能，例如保存数据，或插入用户将离开游戏的确认步骤等。

根据我们的游戏要求，如果我们不在房间里，我们需要显示Launcher场景，所以我们要监听OnLeftRoom()回调并加载Lobby场景Launcher，它在Build Settings场景列表中被索引为0，我们将在本部分的Build Settings场景列表部分中设置。

但是为什么要做一个prefab呢？ 因为我们的游戏需求意味着同一游戏的几个场景，需要重用这个Game Manager对象。在Unity中，重用GameObjects的最好方法是将它们转换为Prefabs。

下面，我们来创建一个UI按钮来调用GameManager LeaveRoom()方法。

1. 退出房间按钮

同样，就像Game Manager一样，在我们将有许多不同的场景需要这个功能，提前计划是有意义的，使Button成为Prefab，这样我们需要在以后使用的话，我们可以重用它并在只一个地方修改它。

<1>确保你在Room for 1场景中

<2>使用unity菜单'GameObject/UI/Panel'，创建一个UI Panel，命名为Top Panel

<3>去掉Panel的Image和Canvas Renderer组件

<4>设置垂直锚点为top，水平锚点为stretch，同时按住Shift和Alt键

<5>设置RectTransform高度为50

<6>右击Top Panel对象，添加UI按钮，命名为Leave button

<7>选择Leave button的文字子节点，设置文本为Leave Game

<8>把按钮的OnClick时间连接到Game Manager实例的Leave Room函数

1. 把Leave button也转变成prefab
2. 保存场景，保存工程
3. 创建其他房间

我们已经创建好了一个房间，我们可以复制3份，然后给它们合适的名字。

1. Build Settings 场景列表

当发布时，我们需要在构建设置中添加所有这些场景，以便Unity在构建应用程序时包括它们。

<1>通过"File/Build Settings"打开Build Settings

<2>把所有的场景拖拽进来，Launcher场景必须是第一个，因为Unity默认会加载第一个场景

## 任务5 游戏管理

### 任务内容

1. 加载Arena Routine
2. 观察玩家连接
3. 从大厅加载Arena

### 提前准备

### 任务步骤

1. 加载Arena Routine

我们创造了4个不同的房间，并且按照约定命名他们，最后一个字符是玩家的数量，所以现在很容易绑定当前房间和关联场景球员数量。这是一种被称为“convention over configuration”的非常有效的技术，基于“Configuration”的方法，举例来说，关于房间中给定数量的玩家，已经维护了一个场景名字查找表。我们的脚本然后会查看该列表，并返回一个无关的场景。 “Configuration”一般需要更多的代码，这就是为什么我们将在这里使用“Convention”，让我们更容易实现想要的功能，而不会让无关的功能污染我们的代码。

<1>打开GameManager脚本

<2>在private methods区块内添加一个新方法，不要忘记保存

#region Private Methods

void LoadArena()

{

if (!PhotonNetwork.isMasterClient)

{

Debug.LogError("PhotonNetwork : Trying to Load a level but we are not the master Client");

}

Debug.Log("PhotonNetwork : Loading Level : " + PhotonNetwork.room.playerCount);

PhotonNetwork.LoadLevel("Room for " + PhotonNetwork.room.playerCount);

}

#endregion

<3>保存GameManager脚本

当我们调用这个方法时，我们将根据我们所在的房间的playerCount属性加载适当的房间。

在这里有两件事要注意，这是非常重要的

<1>如果我们是master,PhotonNetwork.LoadLevel()才应该被调用。所以我们使用PhotonNetwork.isMasterClient首先检查是否是master。检查这一点是调用者的责任，我们将在本节的下一部分中介绍。

<2>我们使用PhotonNetwork.LoadLevel()加载我们想要的级别，我们不直接使用Unity，因为我们想依靠Photon加载这个级别，使在房间中所有连接的客户端都生效，因为我们在这个游戏中启用了PhotonNetwork.automaticallySyncScene。

现在我们可以加载正确的级别，让我们绑定到玩家的连接和断开上面。

1. 观察玩家连接

目前，我们的GameManager脚本是一个常规的MonoBehaviour，我们在前面的教程中学习了使用Photon回调的各种方法，现在GameManager需要监听玩家的连接和断开连接。让我们实现这个。

<1>打开GameManager脚本

<2>把当前的基类MonoBehaviour修改为Photon.PunBehaviour

public class GameManager : Photon.PunBehaviour {

<3>添加下面的Photon回调信息

public override void OnPhotonPlayerConnected(PhotonPlayer other)

{

Debug.Log("OnPhotonPlayerConnected() " + other.name); // not seen if you're the player connecting

if (PhotonNetwork.isMasterClient)

{

Debug.Log("OnPhotonPlayerConnected isMasterClient " + PhotonNetwork.isMasterClient); // called before OnPhotonPlayerDisconnected

LoadArena();

}

}

public override void OnPhotonPlayerDisconnected(PhotonPlayer other)

{

Debug.Log("OnPhotonPlayerDisconnected() " + other.name); // seen when other disconnects

if (PhotonNetwork.isMasterClient)

{

Debug.Log("OnPhotonPlayerConnected isMasterClient " + PhotonNetwork.isMasterClient); // called before OnPhotonPlayerDisconnected

LoadArena();

}

}

<4>保存GameManager脚本

现在，我们有了一个完整的设置。每当玩家加入或离开房间时，我们都会被通知，我们将调用我们刚才创建的LoadArena()方法。但是，只有当我们是PhotonNetwork.isMasterClient的情况下，我们才调用LoadArena()。

让我们现在回到Lobby，最终加入房间的时候能够加载正确的场景。

1. 从大厅加载Arena

<1>编辑脚本Launcher

<2>把下面的脚本添加到OnJoinedRoom方法

// #Critical: We only load if we are the first player, else we rely on PhotonNetwork.automaticallySyncScene to sync our instance scene.

if (PhotonNetwork.room.playerCount == 1)

{

Debug.Log("We load the 'Room for 1' ");

// #Critical

// Load the Room Level.

PhotonNetwork.LoadLevel("Room for 1");

}

<3>保存脚本

让我们测试一下，打开Launcher场景，运行它。点击“play”，让系统连接并加入一个房间。就这样，Lobby没有问题。但如果你离开房间，你会注意到，当回到大厅时，它会自动重新加入房间。噢，让我们解决一下这个问题。

如果你还不知道为什么，简单地分析一下日志。我只是简单地说说一下，因为这需要实践和经验来解决这个问题，知道问题出在哪里以及如何调试它。

你自己尝试一下，如果你仍然找不到问题的根源，让我们一起做。

<1>运行Launcher场景

<2>点击Play按钮，等到你加入了一个房间，"Room for 1"加载完毕

<3>清空Unity Console

<4>点击“Leave Room”

<5>研究一下Unity Console，注意记录中有这句话 "DemoAnimator/Launcher: OnConnectedToMaster() was called by PUN"

<6>停止Launcher场景

<7>再Unity Console中双击这句话 "DemoAnimator/Launcher: OnConnectedToMaster() was called by PUN"，会打开脚本并指向调用的那一行

<8>嗯...每次我们连接上的时候，都会自动的加入一个房间JoinRandomRoom，但是这并不是我们想要的。

要解决这个问题，我们需要知道上下文。当用户点击“Play”按钮时，我们应该记下一个标志，以便知道连接过程源于用户。然后我们可以在各种Photon回调中，检查这个标志，执行相应代码。

<1>编辑Launcher脚本

<2>在Private Variables部分创建一个新的属性

bool isConnecting;

<3>在connect()方法开头加入下面代码

isConnecting = true;

<4>在OnConnectedMaster()方法中，在PhotonNetwork.JoinRandomRoom()外面加一个if语句

if (isConnecting)

{

PhotonNetwork.JoinRandomRoom();

}

<5>保存脚本

现在，如果我们再次测试并运行启动场景，并在大厅和游戏之间来回切换，一切都很好:)为了测试场景的自动同步，您需要发布应用程序（发布桌面，它运行测试最快），并在Unity之外运行它，所以你有了两个玩家，将连接和加入一个房间。 如果Unity Editor首先创建房间，它将是MasterClient，您将能够在Unity控制台中验证您在连接时获得“PhotonNetwork：Loading Level：1”和更高版本“PhotonNetwork：Loading Level：2” 发布的实例。

## 任务6 本地游戏角色创建

### 任务内容

1. prefab基础
2. CharacterController
3. 动画设置
4. 用户输入
5. 相机设置
6. 光束设置
7. 健康设置

### 提前准备

### 任务步骤

1. Prefab基础

了解PUN的第一个重要约定是，对于一个要通过网络实例化的Prefab，它需要保存在Resources文件夹中，否则不行。

在Resources中使用Prefabs的第二个重要的副作用，是你需要监视他们的名字。在Assets Resources中不应该有相同名字的Prefab，因为Unity会选择它找到的第一个，因此请务必确保在您的项目资源中，Resources路径中没有两个Prefab命名相同。

我们将使用Unity提供的Kyle Robot作为一个自由资产。它作为一个Fbx文件，它是由3d软件生成的，例如3ds Max，Maya，cinema4d。使用这些软件创建网格和动画超出了本教程的范围，但是对于创建自己的角色和动画来说至关重要。这个机器人Kyle.fbx位于/Assets/Photon Unity Networking/Demos/Shared Assets/。

这里有一种方法开始使用Kyle Robot.fbx为你的玩家：

<1>在项目浏览器中，创建一个名为“Resources”的文件夹

<2>创建一个新的空场景，并保存为Kyle Test，放在这个文件夹/PunBasics\_tutorial/Scenes/

<3>将Robot Kyle拖放到场景Hierarchy上。

<4>将刚刚在Hierarchy中创建的GameObject重命名为My Robot Kyle

<5>将我的机器人Kyle拖放到/PunBasics\_tutorial/Resources/

1. CharacterController

<1>让我们在层次结构中添加一个CharacterController组件到我的Kyle Robot实例。你可以直接在Prefab本身上这样做，但我们需要调整它，所以这是更快的这种方式。

<2>双击My Robot Kyle让场景视图放大。注意Capsule Collider在脚中间; 我们需要Capsule Collider来正确匹配角色。

<3>在Capsule Collider组件中把Center.y属性改成1。

<4>点击Apply使改变对prefab生效。这不很重要，因为我们编辑了My Kyle Robot prefab，我们想要所有的实例都生效，不只是这一个，所以点击Apply。

1. 动画设置

<1>分配动画控制器

<2>尝试控制器参数

<3>Animator Manager 脚本

<4>Animator Manager : 速度控制

<5>Animator Manager 脚本：方向控制

<6>Animator Manager 脚本：跳跃

5.相机设置

在本节中，我们将使用CameraWork脚本，以保持专注于Player Prefab整体创建过程。如果你想从头开始写CameraWork，请去下一部分，完成后回到这里。

<1>将组件CameraWork添加到My Kyle Robot Prefab

<2>打开属性Follow on Start，可以有效地使照相机即时跟随角色。当我们开始网络实现时，我们将关闭它

<3>设置属性Center Offset为0,4,0，这使得相机看起来更高，从而给出了一个更好的视角的环境比，如果相机直视玩家，我们会看到太多的地面什么都没有。

<4>运行场景Kyle Test，并移动角色，以验证相机正确跟随角色。

6.光束设置

我们的机器人角色还没有武器，让我们创造一些可以从它的眼睛中发出来的激光束。

<1>添加光束模型

为了简单起见，我们将使用简单的立方体并将它们缩放为非常瘦长。有一些技巧来快速做到这一点：不要直接添加一个Cube作为头部节点的子节点，而是创建它移动它，并放大，然后将其附加到头，这将防止猜测正确旋转值让你的光束与眼睛对齐。

另一个重要的技巧是，对两个光束只使用一个碰撞器。这是为了让物理引擎更好地工作，瘦的碰撞器从来不是一个好主意，它不可靠，所以我们将制作一个大盒子碰撞器，以确保可靠地击中目标。

(1)打开Kyle test场景

(2)添加一个Cube,命名为Beam Left

(3)把它修改成一个长的光束，放到左眼的位置

(4)在Hierarchy中选中My Kyle Robot

(5)选中Head子节点

(6)给Head对象添加一个空白对象，命名为Beams

(7)把Beam Left拖拽到Beams下面

(8)复制Beams Left，命名为Beams Right

(9)把它放到右眼的位置上

(10)去掉Beams Right的碰撞体

(11)调整Beams Right的碰撞体，让它包括两个Beam对象

(12)把Beams Left碰撞体的IsTrigger属性设置为True，我们只想知道光束接触到的玩家，而不是碰撞体

(13)撞见一个新的材质，命名为Red Beam，保存

(14)把Red Beam赋值给两个Beams

(15)对prefab执行Apply

<2>通过用户输入控制Beams

好了，既然我们有了激光束，让我们使用Fire键来触发他们。

创建一个C#脚本，命名为PlayerManager。下面是该脚本第一个版本的完整内容：

using UnityEngine;using UnityEngine.EventSystems;

using System.Collections;

namespace Com.MyCompany.MyGame{

/// <summary>

/// Player manager.

/// Handles fire Input and Beams.

/// </summary>

public class PlayerManager : MonoBehaviour

{

#region Private Fields

[Tooltip("The Beams GameObject to control")]

[SerializeField]

private GameObject beams;

//True, when the user is firing

bool IsFiring;

#endregion

#region MonoBehaviour CallBacks

/// <summary>

/// MonoBehaviour method called on GameObject by Unity during early initialization phase.

/// </summary>

void Awake()

{

if (beams == null)

{

Debug.LogError("<Color=Red><a>Missing</a></Color> Beams Reference.", this);

}

else

{

beams.SetActive(false);

}

}

/// <summary>

/// MonoBehaviour method called on GameObject by Unity on every frame.

/// </summary>

void Update()

{

ProcessInputs ();

// trigger Beams active state

if (beams != null && IsFiring != beams.activeSelf)

{

beams.SetActive(IsFiring);

}

}

#endregion

#region Custom

/// <summary>

/// Processes the inputs. Maintain a flag representing when the user is pressing Fire.

/// </summary>

void ProcessInputs()

{

if (Input.GetButtonDown("Fire1"))

{

if (!IsFiring)

{

IsFiring = true;

}

}

if (Input.GetButtonUp("Fire1"))

{

if (IsFiring)

{

IsFiring = false;

}

}

}

#endregion

}}

这个脚本在这个阶段的要点是激活或停用激光束。当激活时，激光束将有效地触发与其他模型发生碰撞，因此我们将在后面利用这些触发器来影响每个角色的健康值。

我们还暴露了一个公共属性Beams，它将让我们在My Kyle Robot Prefab的层次结构中引用确切的对象。让我们看看我们如何工作来连接Beams，因为在Assets浏览器中，Prefabs只暴露第一个子节点，而不是所有子节点，而且我们的Beams确实埋在Prefab层次结构中，因此，我们需要从场景中的一个实例执行此操作，然后将其应用回Prefab本身。  
 (1)打开Kyle Test场景

(2)在场景Hierachy中选择我的Kyle Robot

(3)将PlayerManager组件添加到My Kyle Robot

(4)将My Kyle Robot/Root/Ribs/Neck/Head/Beams拖放到Inspector中的PlayerManager Beams属性中

(5)将实例中的更改应用到Prefab

如果你点击play，并按Fire1输入（默认情况下是左鼠标或左ctrl键），Beams将显示，并立即隐藏时释放时。

7.健康设置

让我们实现一个非常简单的健康系统，当光束击中玩家时会减少生命。由于它不是子弹，而是一个恒定的能量流，我们需要以两种方式考虑健康损害，当我们受到光束撞击时，以及在整个时间射束撞击我们。

<1>打开PlayerManager脚本

<2>为了暴露PhotonView组件，把PlayerManager改变成Photon.PunBehaviour的子对象，

<3>将playerManager转换为单行为puncallbacks以暴露PUN组件

using Photon.Pun;public class PlayerManager : MonoBehaviourPunCallbacks

{

<4>在公共字段区域内添加公共卫生属性

[Tooltip("The current Health of our player")]

public float Health = 1f;

<5>将以下两种方法添加到单行为回调区域。然后保存playermanager脚本。

void OnTriggerEnter(Collider other){

if (!photonView.IsMine)

{

return;

}

if (!other.name.Contains("Beam"))

{

return;

}

Health -= 0.1f;}

void OnTriggerStay(Collider other){

if (! photonView.IsMine)

{

return;

}

if (!other.name.Contains("Beam"))

{

return;

}

Health -= 0.1f\*Time.deltaTime;}

<6>保存PlayerManager脚本

首先，这两种方法几乎是相同的，唯一的区别是，我们在TriggerStay期间使用Deltatime减少健康，减量的速度不取决于帧速率。这是一个重要的概念，通常适用于动画，但在这里，我们也需要这样，我们希望Health在所有设备上以可预测的方式减少，在更快的计算机上这是不公平的，你的健康下降更快:) Deltatime在这里是为了保证一致性。如果您有问题，并通过搜索Unity社区了解DeltaTime，直到您完全吸收这个概念，然后回来，这是至关重要的。

第二个重要的方面，现在应该明白，我们只影响本地玩家的健康，这就是为什么我们前面退出方法的条件PhotonView不是Mine。

最后，如果击中我们的对象是一个Beam，我们只想影响健康，所以我们使用标签“Beam”检查这点，这是我们为何标记我们的Beam对象。

为了便于调试，我们使Health float作为一个公共浮动，以便在等待UI构建时轻松检查其值。

好吧，这看起来一切正确吗？健康系统是不完整的，当健康是0时，没有考虑到玩家的游戏结束状态，让我们现在做到这一点。

游戏结束健康检查

为了保持简单，当玩家的健康达到0时，我们就离开房间。如果你还记得，我们已经在GameManager Script中创建了一个离开房间的方法。如果我们可以重用这个方法而不是重写一遍，这是不错的主意。 相同结果的重复代码是你应该尽一切代价避免的。这也将是一个好时机，介绍一个非常方便的编程概念，“Singleton”。 虽然这个主题本身可以写满几个教程，我们将只实现极小的“单例”。了解Singleton，它们在Unity上下文中的变体以及它们如何帮助创建强大的功能是非常重要的，并将为您节省很多麻烦。所以，不要犹豫，把时间放在这个教程来了解更多。

<1>打开GameManager脚本

<2>在Public Properties区块添加这个变量

public static GameManager Instance;

<3>在Start函数中添加这行代码

void Start(){

Instance = this;

}

<4>保存GameManager脚本

注意，我们使用[static]关键字修饰了Instance变量，这意味着，不必持有一个指向GameManager实例的指针，就可以使用这个变量，所以你可以在代码中的任何地方做一个简单的GameManager.instance.xxx()。这是非常实用的！让我们看看如何用于我们的游戏结束逻辑管理。

<1>打开PlayerManager脚本

<2>在Update函数中，ProcessInput之后，加入这些代码

if (Health <= 0f){

GameManager.Instance.LeaveRoom();}

<3>保存PlayerManager脚本

注意，我们考虑到健康可能是负面的，因为激光束造成的损害在强度上是不同的。我们调用了GameManager实例的LeaveRoom()公共方法，而实际上不需要获取组件或任何东西，我们仅仅依赖于我们假设GameManager组件在当前场景中某个GameObject上的事实。

## 任务7 相机设置

### 任务内容

1.创建CameraWork脚本

### 提前准备

### 任务步骤

1. 创建CameraWork脚本

<1>新建一个C#脚本，命名为CameraWork

<2>把CameraWork的内容替换成下面的代码

using UnityEngine;using System.Collections;

namespace Com.MyCompany.MyGame{

/// <summary>

/// Camera work. Follow a target

/// </summary>

public class CameraWork : MonoBehaviour

{

#region Private Fields

[Tooltip("The distance in the local x-z plane to the target")]

[SerializeField]

private float distance = 7.0f;

[Tooltip("The height we want the camera to be above the target")]

[SerializeField]

private float height = 3.0f;

[Tooltip("The Smooth time lag for the height of the camera.")]

[SerializeField]

private float heightSmoothLag = 0.3f;

[Tooltip("Allow the camera to be offseted vertically from the target, for example giving more view of the sceneray and less ground.")]

[SerializeField]

private Vector3 centerOffset = Vector3.zero;

[Tooltip("Set this as false if a component of a prefab being instanciated by Photon Network, and manually call OnStartFollowing() when and if needed.")]

[SerializeField]

private bool followOnStart = false;

// cached transform of the target

Transform cameraTransform;

// maintain a flag internally to reconnect if target is lost or camera is switched

bool isFollowing;

// Represents the current velocity, this value is modified by SmoothDamp() every time you call it.

private float heightVelocity;

// Represents the position we are trying to reach using SmoothDamp()

private float targetHeight = 100000.0f;

#endregion

#region MonoBehaviour Callbacks

/// <summary>

/// MonoBehaviour method called on GameObject by Unity during initialization phase

/// </summary>

void Start()

{

// Start following the target if wanted.

if (followOnStart)

{

OnStartFollowing();

}

}

/// <summary>

/// MonoBehaviour method called after all Update functions have been called. This is useful to order script execution. For example a follow camera should always be implemented in LateUpdate because it tracks objects that might have moved inside Update.

/// </summary>

void LateUpdate()

{

// The transform target may not destroy on level load,

// so we need to cover corner cases where the Main Camera is different everytime we load a new scene, and reconnect when that happens

if (cameraTransform == null && isFollowing)

{

OnStartFollowing();

}

// only follow is explicitly declared

if (isFollowing)

{

Apply();

}

}

#endregion

#region Public Methods

/// <summary>

/// Raises the start following event.

/// Use this when you don't know at the time of editing what to follow, typically instances managed by the photon network.

/// </summary>

public void OnStartFollowing()

{

cameraTransform = Camera.main.transform;

isFollowing = true;

// we don't smooth anything, we go straight to the right camera shot

Cut();

}

#endregion

#region Private Methods

/// <summary>

/// Follow the target smoothly

/// </summary>

void Apply()

{

Vector3 targetCenter = transform.position + centerOffset;

// Calculate the current & target rotation angles

float originalTargetAngle = transform.eulerAngles.y;

float currentAngle = cameraTransform.eulerAngles.y;

// Adjust real target angle when camera is locked

float targetAngle = originalTargetAngle;

currentAngle = targetAngle;

targetHeight = targetCenter.y + height;

// Damp the height

float currentHeight = cameraTransform.position.y;

currentHeight = Mathf.SmoothDamp( currentHeight, targetHeight, ref heightVelocity, heightSmoothLag );

// Convert the angle into a rotation, by which we then reposition the camera

Quaternion currentRotation = Quaternion.Euler( 0, currentAngle, 0 );

// Set the position of the camera on the x-z plane to:

// distance meters behind the target

cameraTransform.position = targetCenter;

cameraTransform.position += currentRotation \* Vector3.back \* distance;

// Set the height of the camera

cameraTransform.position = new Vector3( cameraTransform.position.x, currentHeight, cameraTransform.position.z );

// Always look at the target

SetUpRotation(targetCenter);

}

/// <summary>

/// Directly position the camera to a the specified Target and center.

/// </summary>

void Cut()

{

float oldHeightSmooth = heightSmoothLag;

heightSmoothLag = 0.001f;

Apply();

heightSmoothLag = oldHeightSmooth;

}

/// <summary>

/// Sets up the rotation of the camera to always be behind the target

/// </summary>

/// <param name="centerPos">Center position.</param>

void SetUpRotation( Vector3 centerPos )

{

Vector3 cameraPos = cameraTransform.position;

Vector3 offsetToCenter = centerPos - cameraPos;

// Generate base rotation only around y-axis

Quaternion yRotation = Quaternion.LookRotation( new Vector3( offsetToCenter.x, 0, offsetToCenter.z ) );

Vector3 relativeOffset = Vector3.forward \* distance + Vector3.down \* height;

cameraTransform.rotation = yRotation \* Quaternion.LookRotation( relativeOffset );

}

#endregion

}}

<3>保存脚本

## 任务8 角色网络同步

### 任务内容

1.PhotonView 组件

2.Transform 同步

3.Animator 同步

4.用户输入管理

5.Camera 控制

6.Beams 开火控制

7.Health 同步

### 提前准备

### 任务步骤

1. PhotonView 组件

首先，我们需要在Prefab上添加一个PhotonView组件。PhotonView将每个计算机上的各种实例连接在一起，并定义要观察的组件以及如何观察这些组件。

<1>添加PhotonView组件到My Robot Kyle

<2>将Observe Option设置为Unreliable On Change

<3>注意，PhotonView警告你，要起作用的话你需要观察一些东西

让我们设置我们要观察的对象，然后我们将回到这个PhotonView组件，并完成它的设置。

1. Transform同步

我们想要同步的明显特征是角色的位置和旋转，使得当Player移动时，其他计算机上的Player也以类似的方式移动和旋转。

你可以直接观察自己脚本中的Transform组件，但是由于网络延迟和数据同步的有效性，你会遇到很多麻烦。幸运的是，为了使这个常见的任务更容易，我们将使用[Photon Transform View]组件，作为变换组件和PhotonView之间的中间人(middleMan)。基本上，这个组件已经为你做了所有的工作。

<1>给'My Robot Kyle'Prefab添加PhotonTransformView

<2>拖拽PhotonTransformView的标题栏到PhotonView组件的第一个Observable组件上

<3>在PhotonTransformView组件上勾选Synchronize Position

<4>在Synchronize Position里面，Interpolation Option选择“Lerp”

<5>把Lerp Speed设置为10（数值越大插值越快）

<6>勾选SynchronizeRotation

1. Animator 同步

PhotonAnimatorView也使得网络设置变得轻而易举，将为您节省大量的时间和麻烦。它允许您定义哪些层权重和要同步的参数。层权重只有在游戏过程中改变了才需要同步，并且可以不同步它们。参数也是如此。有时可以从其他因素导出动画值。速度值是一个很好的例子，你不一定需要让这个值完全同步，但你可以使用同步的位置更新来估计它的值。如果可能，尝试同步尽可能少的参数。

<1>给My Robot Kyle Prefab添加一个PhotonAnimatorView

<2>拖拽PhotonAnimatorView的标题栏到PhotonView组件的Observable组件上  
<3>同步参数中设置Speed为Discrete

<4>设置Direction为Discrete

<5>设置Jump为Discrete

<6>设置Hi为Disabled

每个值都可以是disabled，或者以离散(discretely)或连续(continuously)的方式同步(synchronized)。在我们的例子中，由于我们不使用Hi参数，我们将禁用它，从而节省带宽。

离散同步(Discrete synchronization)意味着值每秒发送10次（在OnPhotonSerializeView中）。 接收客户端将值传递到他们的本地Animator。

连续同步(Continuous synchronization)意味着PhotonAnimatorView每帧都要运行。当调用OnPhotonSerializeView（每秒10次）时，自上次调用以来记录的值一起发送。接收客户端然后按顺序应用值以保持平滑过渡。虽然此模式更平滑，但它为了实现此效果发送了更多数据。

1. 用户输入管理

用户控制网络的一个关键方面是，相同的Prefab将为所有玩家实例化，但是其中只有一个是实际在计算机前的用户控制的，所有其他实例代表的是在其他计算机上的其他用户。因此，这一点的第一个障碍是输入管理。我们如何能够在一个实例上而不是在其他实例上启用输入，以及如何知道哪个是正确的？这就需要isMine的概念。

让我们编辑我们之前创建的PlayerAnimatorManager脚本。在目前的形式中，这个脚本不知道这个区别，让我们实现它。

<1>打开脚本PlayerAnimatorManager

<2>把PlayerAnimatorManager的基类改为Photon.MonoBehaviour，这个类很方便的暴露photonView的组件

<3>在Update函数的开头加入下面代码

if (photonView.IsMine == false && PhotonNetwork.IsConnected == true){

return;}

<4>保存脚本

Ok，如果实例由“客户端”应用程序控制，PhotonView.isMine将为true，意味着此实例表示在此应用程序中在此计算机上正在play的玩家。因此，如果它是假的，我们不想做任何事情，只依靠PhotonView组件来同步我们之前设置的变换和动画组件。

但是，为什么在我们的if语句中强制执行PhotonNetwork.connected == true？因为在开发期间，我们可能想要测试这个prefab，而不连接。在虚拟场景中，例如，只是创建和验证与网络功能无关的代码。因此，使用这个附加表达式，如果我们没有连接，我们将允许使用输入。这是一个非常简单的伎俩，并将大大改善您的开发过程中的工作流。

5.Camera 控制

它和输入一样，Player只有一个游戏视图，所以我们需要CameraWork脚本只跟随本地Player，而不是其他Player。这就是为什么CameraWork脚本有这个能力来定义什么时候跟随。

让我们修改PlayerManager脚本来控制CameraWork组件。

<1>打开PlayerManager脚本

<2>在Awake()和Update()函数之间插入下面的代码

void Start(){

CameraWork \_cameraWork = this.gameObject.GetComponent<CameraWork>();

if (\_cameraWork != null)

{

if (photonView.IsMine)

{

\_cameraWork.OnStartFollowing();

}

}

else

{

Debug.LogError("<Color=Red><a>Missing</a></Color> CameraWork Component on playerPrefab.", this);

}}

<3>保存脚本

首先，它获取CameraWork组件，我们期望这样，所以如果我们没有找到它，就记录一个错误。然后，如果photonView.isMine为true，这意味着我们需要跟随这个实例，因此我们调用\_cameraWork.OnStartFollowing（），它有效地使相机跟随场景中的那个实例。

所有其他Player实例的photonView.isMine将设置为false，因此它们各自的\_cameraWork将不会做任何事情。

下面的一个改变可以使这点生效

在Robot animator prefab上，CameraWork组件中禁用Follow on Start属性

PlayerManager脚本将会像上面描述的那样调用\_cameraWork.OnStartFollowing()，现在这有效的处理了跟随Player的逻辑。

6.Beams 开火控制

开火也同样遵循上面的输入原则，它只在photonView.isMine是true的情况下工作。

<1>打开PlayerManager脚本

<2>用一个if语句包住输入处理

if (photonView.IsMine){

ProcessInputs ();}

<3>保存脚本

然而，当测试这个的时候，我们只看到本地Player开火。我们需要看看其他实例何时开火！我们需要一种用于在网络上同步开火的机制。为此，我们将手动同步IsFiring布尔值，直到现在，我们离开了PhotonTransformView和PhotonAnimatorView来为我们进行变量的所有内部同步，我们只需要调整通过Unity Inspector方便地暴露给我们的参数，但在这里我们需要的是，针对你的具体游戏，所以我们需要手动这样做。

<1>打开PlayerManager脚本

<2>实现IPunObservable接口

public class PlayerManager : MonoBehaviourPunCallbacks, IPunObservable{

#region IPunObservable implementation

public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)

{

}

#endregion

<3>在IPunObservable.OnPhotonSerializeView函数中添加下面代码

if (stream.IsWriting){

// We own this player: send the others our data

stream.SendNext(IsFiring);}else{

// Network player, receive data

this.IsFiring = (bool)stream.ReceiveNext();}

<4>保存脚本

<5>回到Unity编辑器，在assets中选择Robot Animator prefab，在PhotonView组件中添加一条监视记录，然后把PlayerManager组件拖拽到上面

没有上一步的话，IPunObservable.OnPhotonSerializeView永远不会调用，因为它没有被PhotonView监视。

在这个IPunObservable.OnPhotonSerializeView方法中，我们传递了一个变量stream，这是将通过网络发送的，并且这个调用是我们读写数据的机会。当我们是本地Player的时候（PhotonView.isMine == true）才能写入数据，否则是读数据。

由于stream类自己知道该怎么处理数据，所以我们只需要简单的利用stream.isWriting，就可以知道当前实例情况下要做什么。

如果我们期望写入数据，我们使用stream.SendNext()附加到数据流的IsFiring值，这是一个非常方便的方法，隐藏了数据序列化的所有辛苦工作。如果我们希望读数据，那就使用stream.ReceiveNext()。

7.Health 同步

好的，为了完成更新Player的功能，我们将同步Health值，以便Player的每个实例都有正确的Health值。这与我们刚刚介绍的IsFiring值使用完全相同的原则。

<1>打开脚本PlayerManager

<2>在IPunObservable.OnPhotonSerializeView中，SendNext和ReceiveNext处理IsFiring变量之后，同样的处理一下Health

if (stream.IsWriting){

// We own this player: send the others our data

stream.SendNext(IsFiring);

stream.SendNext(Health);}else{

// Network player, receive data

this.IsFiring = (bool)stream.ReceiveNext();

this.Health = (float)stream.ReceiveNext();}

<3>保存PlayerManager

这样就完成了同步Health变量。

## 任务9 游戏角色创建同步

### 任务内容

1. 实例化Player
2. 追踪Player实例
3. 在竞技场外时管理Player位置

### 提前准备

### 任务步骤

1. 实例化Player

实际上很容易实例化我们的Player Prefab。我们需要在刚刚进入房间时实例化它，我们可以依靠GameManager脚本的Start()回调，这将表明我们加载了Arena，这意味着我们在一个房间里了。

<1>打开GameManager脚本

<2>在Public Variables区块，添加下面的变量

[Tooltip("The prefab to use for representing the player")]

public GameObject playerPrefab;

<3>在Start()函数中，添加下面的代码

if (playerPrefab == null){

Debug.LogError("<Color=Red><a>Missing</a></Color> playerPrefab Reference. Please set it up in GameObject 'Game Manager'",this);}

else{

Debug.LogFormat("We are Instantiating LocalPlayer from {0}", Application.loadedLevelName);

PhotonNetwork.Instantiate(this.playerPrefab.name, new Vector3(0f,5f,0f), Quaternion.identity, 0);}

<4>保存脚本

这暴露了一个公共字段让你引用Player Prefab，它很方便，因为在这个特别的，我们可以直接拖放在GameManager Prefab上，而不是在每个场景，因为Player Prefab是一个Asset，所以引用将保持完好（与引用层次结构中的GameObject相反，Prefab只能在同一场景中实例化时）。

****警告：****一定要确保要通过网络实例化的Prefabs是放在Resources文件夹里面的，这是Photon的要求。

然后，在Start()中，我们实例化它（在检查我们有一个正确Prefab Player引用之后）。

注意，我们实例化在地板上方（5个单位以上，而Player只有2个单位高）。当新player加入房间时防止碰撞，Player可能已经在围绕舞台的中心移动，并且因此避免突然的碰撞。“下降”Player也是一个很好的清晰的指示，在游戏中引入了一个新的实体。

然而，这是不够的我们的情况下，我们有一个扭曲:)当其他Player将加入的时候，不同的场景将加载，我们想保持一致性，不能只是因为其中一个离开就破坏现有的Player。因此，我们需要告诉Unity不要销毁我们创建的实例，这反过来意味着，我们需要检查在加载场景时是否需要实例化。

1. 追踪Player实例

<1>打开PlayerManager脚本

<2>在Public Variables区块，添加下面的代码

[Tooltip("The local player instance. Use this to know if the local player is represented in the Scene")]

public static GameObject LocalPlayerInstance;

<3>在Awake()中添加下面的代码

if (photonView.IsMine){

PlayerManager.LocalPlayerInstance = this.gameObject;

}

DontDestroyOnLoad(this.gameObject);

<4>保存脚本

修改完这些，然后我们在GameManager脚本内部实现只在必要时实例化。

<1>打开GameManager脚本

<2>把实例化调用部分放到if语句中

if (PlayerManager.LocalPlayerInstance == null){

Debug.LogFormat("We are Instantiating LocalPlayer from {0}", SceneManagerHelper.ActiveSceneName);

PhotonNetwork.Instantiate(this.playerPrefab.name, new Vector3(0f, 5f, 0f), Quaternion.identity, 0);

}

else{

Debug.LogFormat("Ignoring scene load for {0}", SceneManagerHelper.ActiveSceneName);

}

<3>保存脚本

这样的话，如果PlayerManager中LocalPlayerInstance为空的话，才会实例化。

1. 在竞技场外时管理Player位置

我们还有一件事要注意。竞技场的尺寸基于玩家的数量而改变，这意味着存在如下情况：如果一个玩家离开，并且其他玩家接近当前竞技场尺寸的边界，那么在加载完小场景之后，他们将发现自己在较小的竞技场之外，我们需要考虑到这一点，并且在这种情况下简单地将Player重新定位到竞技场的中心。这在你的游戏和级别设计时是一个问题。

目前有一个额外的复杂性，因为Unity已经改进了场景管理，并且Unity 5.4已经弃用了一些回调，要创建一个兼容所有Unity版本（从Unity 4.7到最新）的代码，会稍微复杂一些。所以我们需要基于Unity不同版本写不同的代码。它与Photon Networking无关，但无论如何对你掌握项目更新很重要。

<1>打开PlayerManager脚本

<2>在顶部添加下面的代码

#if UNITY\_5\_4\_OR\_NEWER

void OnSceneLoaded(UnityEngine.SceneManagement.Scene scene, UnityEngine.SceneManagement.LoadSceneMode loadingMode)

{

this.CalledOnLevelWasLoaded(scene.buildIndex);

}

#endif

<3>在Start()方法末尾，添加下面的代码

#if UNITY\_5\_4\_OR\_NEWER

UnityEngine.SceneManagement.SceneManager.sceneLoaded += OnSceneLoaded;#endif

<4>在MonoBehaviour CallBacks区块，添加下面两个方法

#if !UNITY\_5\_4\_OR\_NEWER/// <summary>See CalledOnLevelWasLoaded. Outdated in Unity 5.4.</summary>void OnLevelWasLoaded(int level){

this.CalledOnLevelWasLoaded(level);}#endif

void CalledOnLevelWasLoaded(int level){

// check if we are outside the Arena and if it's the case, spawn around the center of the arena in a safe zone

if (!Physics.Raycast(transform.position, -Vector3.up, 5f))

{

transform.position = new Vector3(0f, 5f, 0f);

}}

<5>重写OnDisable方法

#if UNITY\_5\_4\_OR\_NEWERpublic override void OnDisable(){

base.OnDisable ();

UnityEngine.SceneManagement.SceneManager.sceneLoaded -= OnSceneLoaded;}#endif

<6>保存PlayerManager脚本

这个新的代码正在监听加载一个级别，从当前玩家的位置和向下发射射线，看看我们是否击中任何东西。如果没有，这意味着我们不在竞技场的地面上，我们需要重新定位回中心，正如我们第一次进入房间。

如果你的Unity版本低于Unity 5.4，我们将使用Unity的回调OnLevelWasLoaded。如果你是Unity 5.4或更高版本，OnLevelWasLoaded不再可用，而是必须使用新的SceneManagement系统。最后，为了避免重复代码，我们只需要调用CalledOnLevelWasLoaded()方法，该方法将从OnLevelWasLoaded或SceneManager sceneLoaded回调中调用。

## 任务10 玩家血条

### 任务内容

1.创建UI Prefab

2.PlayerUI脚本基础

3.实例化和Player绑定

4.跟随目标Player

### 提前准备

### 任务步骤

1. 创建UI Prefab

<1>打开任何一个有UI Canvas的场景

<2>向canvas添加Slider UI对象，将其命名为Player UI

<3>将RectTransform垂直锚点设置为居中，将水平锚点设置为中心

<4>将RectTransform宽度设置为80，高度设置为15

<5>选择background子节点，将其图像组件颜色设置为红色

<6>选择子节点"Fill Area/Fill"，将其图像颜色设置为绿色

<7>添加一个文本UI对象作为Player UI的子节点，将其命名为Player Name Text

<8>将Player UI从Hierarchy拖动到Assets中的Prefab文件夹中，生成Prefab

<9>删除场景中的实例，我们不再需要它了。

1. PlayerUI脚本基础

<1>创建新的C#脚本，命名为PlayerUI

<2>下面是PlayerUI脚本的框架

using UnityEngine;using UnityEngine.UI;

using System.Collections;

namespace Com.MyCompany.MyGame{

public class PlayerUI : MonoBehaviour

{

[Tooltip("UI Text to display Player's Name")]

[SerializeField]

private Text playerNameText;

[Tooltip("UI Slider to display Player's Health")]

[SerializeField]

private Slider playerHealthSlider;

}}

<3>保存脚本

现在让我们编辑Prefab。

<1>把PlayerUI脚本添加到PayerUI prefab上

<2>把 "Player Name Text"对象拖拽到PlayerNameText公共字段上面

<3>把Slider组件拖拽到公共字段PlayerHealthSlider上面

3.实例化和Player绑定

(1)绑定PlayerUI到Player

PlayerUI脚本需要知道它代表了哪个Player，因为其中的一个原因是：能够显示它的健康和名称，让我们创建一个公共的方法来完成这个绑定。

<1>打开PlayerUI脚本

<2>在私有属性区块添加一个私有属性

private PlayerManager target;

我们在这里需要思考，我们会定期寻找Health，所以缓存一个Player Manager的引用以提高效率是有意义的。

<3>在公共方法区块添加下面的代码

public void SetTarget(PlayerManager \_target){

if (\_target == null)

{

Debug.LogError("<Color=Red><a>Missing</a></Color> PlayMakerManager target for PlayerUI.SetTarget.", this);

return;

}

// Cache references for efficiency

target = \_target;

if (playerNameText != null)

{

playerNameText.text = target.photonView.Owner.NickName;

}}

<4>在MonoBehaviour Messages区块添加这个方法

void Update(){

// Reflect the Player Health

if (playerHealthSlider != null)

{

playerHealthSlider.value = target.Health;

}}

<5>保存代码

有了这些，我们就可以显示目标Player的名字和health了

(2)实例化

OK，所以我们已经知道如何实例化这个Prefab。每次我们实例化，最好的方法是在PlayerManager的初始化过程中。

<1>打开脚本PlayerManager

<2>添加公共字段来持有Player UI的引用

[Tooltip("The Player's UI GameObject Prefab")]

[SerializeField]

public GameObject PlayerUiPrefab;

<3>在Start()方法中添加下面的代码

if (PlayerUiPrefab != null){

GameObject \_uiGo = Instantiate(PlayerUiPrefab);

\_uiGo.SendMessage ("SetTarget", this, SendMessageOptions.RequireReceiver);

}

else{

Debug.LogWarning("<Color=Red><a>Missing</a></Color> PlayerUiPrefab reference on player Prefab.", this);

}

<4>保存脚本

所有这些都是标准的Unity编码。但请注意，我们正在向刚刚创建的实例发送消息。我们需要一个接收器，这意味着如果SetTarget没有找到响应它的组件，我们将被警告。另一种方法是从实例中获取PlayerUI组件，然后直接调用SetTarget。通常的建议是，直接使用组件，但是知道你可以以不同的方式实现同样的事情也很好。

然而这远远不够，我们需要处理删除Player时，不能有孤独的UI实例，所以当UI实例发现被分配到的目标不存在的话，我们就需要销毁它。

<1>打开PlayerUI脚本

<2>在Update()中添加代码

// Destroy itself if the target is null, It's a fail safe when Photon is destroying Instances of a Player over the networkif (target == null){

Destroy(this.gameObject);

return;}

<3>保存PlayerUI脚本

这个代码，虽然容易，但实际上是相当方便。由于Photon删除联网实例的方式，如果发现目标引用为空，则UI实例更容易简单地销毁自身。这避免了很多潜在的问题，并且非常安全，无论为什么目标丢失的原因，相关的UI会自动破坏自己，非常方便快捷。

但等等...当一个新的级别被加载，UI正在被销毁，但我们的Player还在...所以我们需要实例化它，当我们知道一个级别被加载时，让我们这样做：

<4>打开PlayerManager脚本

<5>在CalledOnLevelWasLoaded()方法中添加

GameObject \_uiGo = Instantiate(this.PlayerUiPrefab);

\_uiGo.SendMessage("SetTarget", this, SendMessageOptions.RequireReceiver);

<6>保存PlayerManager脚本

注意，有更复杂并且强大的方法来处理这一点，UI可以使用单例模式实现，但它会很快变得复杂，因为其他Players加入和离开房间也需要处理他们的UI。在我们的实现中，这是直接的，代价是我们实例化我们的UI预制的重复。作为一个简单的练习，您可以创建一个私有方法来实例化和从各个地方发送“SetTarget”消息，调用该方法，而不是复制代码。

1. Player绑定

Unity UI系统的一个非常重要的约束是，任何UI元素必须放置在Canvas对象下面。因此当这个PlayerUI Prefab被实例化时，我们需要处理这个约束，我们将在PlayerUI的初始化期间这样做。

<1>打开PlayerUI脚本

<2>在MonoBehaviour Messages区块添加下面的方法

void Awake(){

this.transform.SetParent(GameObject.Find("Canvas").GetComponent<Transform>(), false);}

<3>保存PlayerUI脚本

为什么要蛮力查找Canvas这种方式？ 因为当场景要被加载和卸载时，我们的Prefab和Canvas每一次都不相同。为了避免更复杂的代码结构，我们将采用最快的方式。真的不建议使用“Find”，因为这是一个缓慢的操作。实现更复杂的处理方法超出了本教程的范围。当你觉得习惯于Unity和脚本，并能找到编写更好的管理Canvas元素引用、并考虑到加载和卸载的方法，这会是一个很好的练习。

4.跟随目标Player

<1>打开PlayerUI脚本

<2>在Public Fields 区块添加公共属性

[Tooltip("Pixel offset from the player target")]

[SerializeField]

private Vector3 screenOffset = new Vector3(0f,30f,0f);

<3>在Public Fields 区块添加下面私有属性

float characterControllerHeight = 0f;Transform targetTransform;Renderer targetRenderer;Vector3 targetPosition;

<4>在SetTarget()方法中\_target设置之后，添加下面代码

targetTransform = this.target.GetComponent<Transform>();

targetRenderer = this.target.GetComponent<Renderer>();CharacterController characterController = \_target.GetComponent<CharacterController> ();

if (characterController != null){

characterControllerHeight = characterController.height;

}

我们知道我们的Player是基于一个CharacterController，它有一个Height属性，我们需要这个做一个适当的偏移，使得UI元素在player上面。

<5>在Public Methods区块添加公共方法

void LateUpdate(){// Do not show the UI if we are not visible to the camera, thus avoid potential bugs with seeing the UI, but not the player itself.if (targetRenderer != null){

this.gameObject.SetActive(targetRenderer.isVisible);}

// #Critical// Follow the Target GameObject on screen.if (targetTransform != null){

targetPosition = targetTransform.position;

targetPosition.y += characterControllerHeight;

this.transform.position = Camera.main.WorldToScreenPoint (targetPosition) + screenOffset;}}

<6>保存PlayerUI脚本

因此，将2d位置与3d位置匹配的诀窍是使用Camera的WorldtoScreenPoint函数。由于在我们的游戏中我们只有一个相机，所以可以依赖于访问Unity场景的默认设置主相机 。

注意，我们如何在几个步骤中设置偏移：首先我们获得目标的实际位置，然后添加\_characterControllerHeight，最后，在我们推导出Player顶部的屏幕位置之后，我们添加屏幕偏移。

# 实训项目2 双人对战小项目

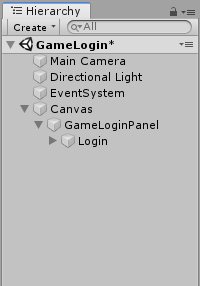
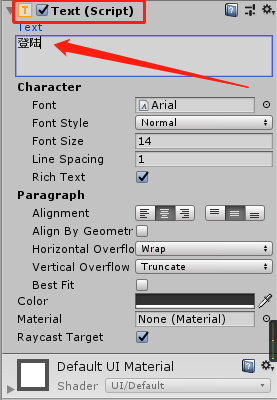
## 任务1 搭建场景

## 任务内容

**1. 搭建登陆场景**

**2. 搭建沙盒场景**

搭建我们需要用的场景，包括两个主要场景：登陆场景和沙盒场景。现在，我们新建一个Unity3D项目，至于保存位置就按照我们自己的意愿吧，OK，建立成功后，我们就从登陆场景开始吧。新建一个场景，嗯... ...重新命名为GameBegin , 你可以自定义你想要的名字，但我觉得这个很有说明性。在Hierarchy面板中右键Creat->UI->Panel, 然后就会出现一个半透明面板，这个作为登陆面板，为这个面板命名：GameLoginPanel,然后我们再新建一个Button并命名为Login, 确保这个Button是正常的。然后展开这个Button的物体，会找到Text,单击Text,在Inspector面板中修改它的显示内容。

然后让我们搭建沙盒场景：

## 任务2 编写登陆脚本

## 任务内容

**编写登陆脚本**

**测试登陆流程**

打开上次的Unity3D项目，首先新建一个对象，叫它GameBegin,再新建一个C# Scripts也叫它GameBegin,并打开它。这个脚本主要用Photon的一些函数来实现连接服务器和推送连接状态等。

首先我们需要把原有的MonoBehavior改成MonoBehaviourPunCallbacks。 MonoBehaviourPunCallbacks是Photon的一个类，这个类里主要包含了很多回掉函数，顾名思义就是然我们通过这些回调函数来检测目前的状态也可以是我们的客户端逻辑会在这些回掉函数里进行体现。对了，我们还需要引进两个头文件：

1. **using** Photon.Pun;
2. **using** Photon.Realtime;

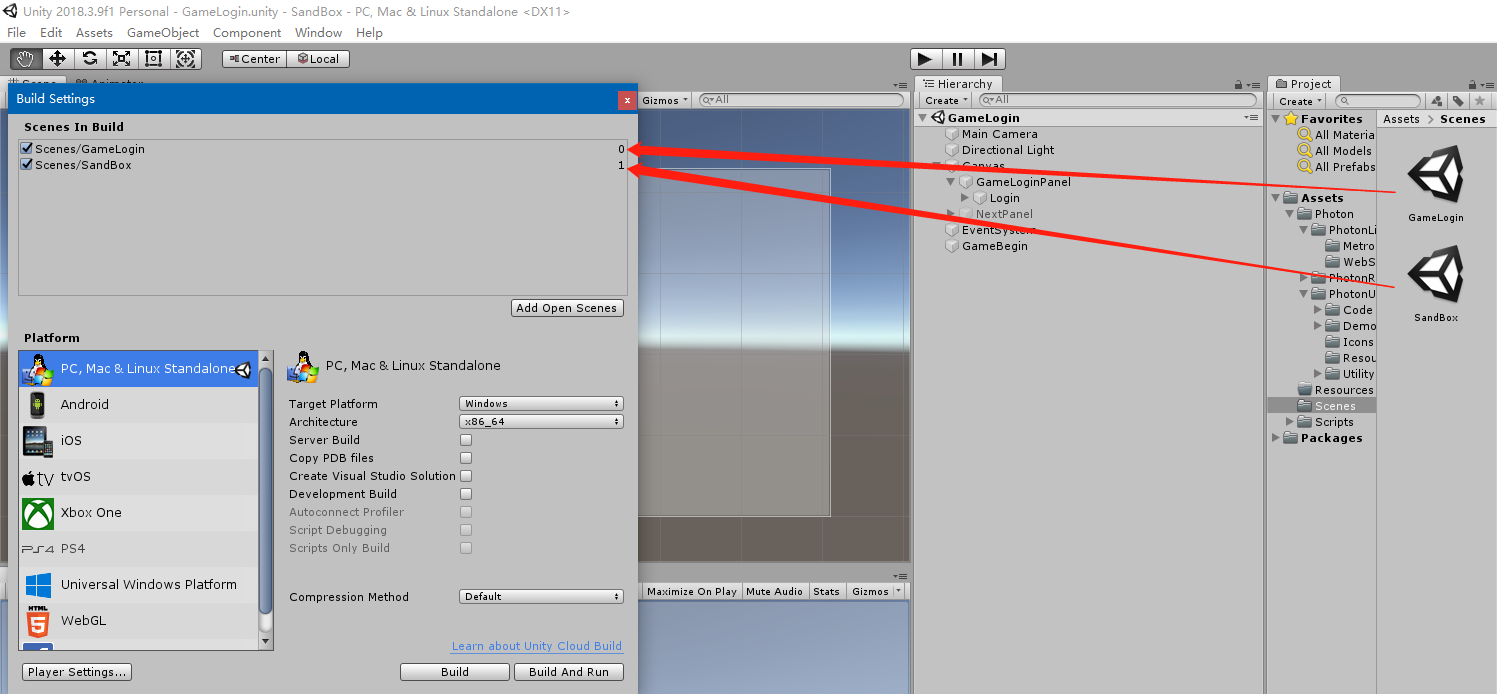
然后，就可以开始编写了：

1. **using** Photon.Pun;
2. **using** Photon.Realtime;
3. **using** System.Collections;
4. **using** UnityEngine;
5. **using** UnityEngine.UI;
7. **public** **class** GameBegin : MonoBehaviourPunCallbacks
8. {
9. //登陆Button
10. **public** Button Login;
11. //创建房间Button
12. **public** Button CreatRoom;
13. //加入房间Button
14. **public** Button JoinRoom;
15. //是否我们准备好加入一个房间
16. **private** **bool** joinRoomIs;
18. **private** **void** Start()
19. {
20. //保持让所有的玩家的加载场景为同一个级别
21. PhotonNetwork.AutomaticallySyncScene = **true**;
22. //如过连不上，就用Pun自带的默认连接配置文件来连接
23. PhotonNetwork.ConnectUsingSettings();
24. }
26. /// <summary>
27. /// 在成功连接服务器后调用
28. /// </summary>
29. **public** **override** **void** OnConnectedToMaster()
30. {
32. }
34. /// <summary>
35. /// 成功加入房间后调用
36. /// </summary>
37. **public** **override** **void** OnJoinedRoom()
38. {
40. }
42. /// <summary>
43. /// 在加入房间失败后调用
44. /// </summary>
45. /// <param name="returnCode"></param>
46. /// <param name="message"></param>
47. **public** **override** **void** OnJoinRandomFailed(**short** returnCode, **string** message)
48. {
50. }
52. /// <summary>
53. /// 在成功创建房间后调用
54. /// </summary>
55. **public** **override** **void** OnCreatedRoom()
56. {
58. }
59. }

可以看到有很多override的样子的函数，这些就是回调函数。我们只需要在里面写我们想的就可以了，不需要在Update()和Start()的里调用。我们接着写：

1. **using** Photon.Pun;
2. **using** Photon.Realtime;
3. **using** System.Collections;
4. **using** UnityEngine;
5. **using** UnityEngine.UI;
7. **public** **class** GameBegin : MonoBehaviourPunCallbacks
8. {
9. //登陆Button
10. **public** Button Login;
11. //创建房间Button
12. **public** Button CreatRoom;
13. //加入房间Button
14. **public** Button JoinRoom;
15. //是否我们准备好加入一个房间
16. **private** **bool** joinRoomIs;
17. //登陆成功后的面板
18. **public** GameObject nextPanel;
20. **private** **void** Start()
21. {
22. //保持让所有的玩家的加载场景为同一个级别
23. PhotonNetwork.AutomaticallySyncScene = **true**;
25. //如过连不上，就用Pun自带的默认连接配置文件来连接
26. Login.onClick.AddListener(() => { PhotonNetwork.ConnectUsingSettings(); });
27. //创建一个房间
28. CreatRoom.onClick.AddListener(() => { PhotonNetwork.CreateRoom(**null**, **new** RoomOptions { MaxPlayers = 10 }, **null**, **null**); });
29. //加入一个随机的房间
30. JoinRoom.onClick.AddListener(() => { PhotonNetwork.JoinRandomRoom(); });
31. }
33. /// <summary>
34. /// 在成功连接服务器后调用
35. /// </summary>
36. **public** **override** **void** OnConnectedToMaster()
37. {
38. joinRoomIs = **true**;
39. nextPanel.SetActive(**true**);
40. }
42. /// <summary>
43. /// 成功加入房间后调用
44. /// </summary>
45. **public** **override** **void** OnJoinedRoom()
46. {
47. PhotonNetwork.LoadLevel("SandBox");
48. }
50. /// <summary>
51. /// 在加入房间失败后调用
52. /// </summary>
53. /// <param name="returnCode"></param>
54. /// <param name="message"></param>
55. **public** **override** **void** OnJoinRandomFailed(**short** returnCode, **string** message)
56. {
57. Debug.Log("加入房间失败");
58. }
60. /// <summary>
61. /// 在成功创建房间后调用
62. /// </summary>
63. **public** **override** **void** OnCreatedRoom()
64. {
65. Debug.Log("创建房间成功");
66. }
67. }

到这里我们已经编写完了登陆的代码，现在切换到Unity3D项目中，在左上角的File中Build Settings中把我们需要的两个场景托进去。



然后我们点击Play就可以进行登陆游戏了。

## 任务4 角色生成与管理

## 任务内容

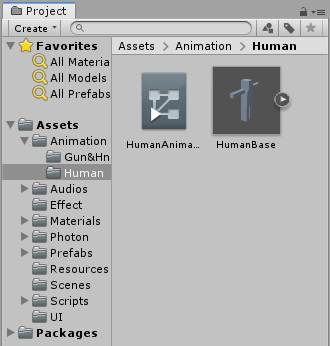
**编写GameManager脚本**

**绑定角色预制体**

在第二个场景我们创建一个空物体，叫它GameManager，然后为它绑定一个脚本：新建一个GameManager的脚本，然后我们打开它：

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
4. **using** Photon.Pun;
6. **namespace** SandBox
7. {
8. **public** **class** GameManager : MonoBehaviour
9. {
10. //角色预制体
11. **public** GameObject playerPrefabs;
13. **void** Start()
14. {
15. //进入房间就会生成一个角色
16. PhotonNetwork.Instantiate(playerPrefabs.name, Vector3.zero, Quaternion.identity, 0);
17. }
18. }
19. }

现在我们需要一个角色供我们加入或创建房间后用它来战斗，导入角色的素材包：



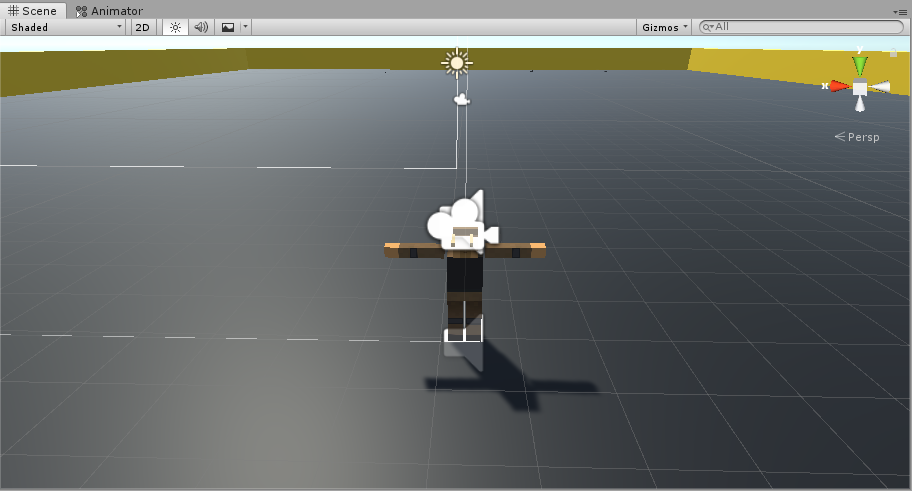
## 任务5 角色控制

## 任务内容

**创建角色预制体**

**让角色移动**

将我们的已经导入好的角色素材拖曳到游戏窗口里：



在Assest store 中下载Standard Assest包

删减FirstPersonController脚本到如下：

1. **using** Photon.Pun;
2. **using** UnityEngine;
3. **using** Random = UnityEngine.Random;
5. [RequireComponent(**typeof**(CharacterController))]
6. [RequireComponent(**typeof**(AudioSource))]
7. **public** **class** FirstPersonController : MonoBehaviour
8. {
9. [SerializeField] **private** **bool** m\_IsWalking;
10. **public** **float** m\_WalkSpeed;
11. **public** **float** m\_RunSpeed;
12. [SerializeField] [Range(0f, 1f)] **private** **float** m\_RunstepLenghten;
13. [SerializeField] **private** **float** m\_JumpSpeed;
14. [SerializeField] **private** **float** m\_StickToGroundForce;
15. [SerializeField] **private** **float** m\_GravityMultiplier;
16. [SerializeField] **private** **bool** m\_UseFovKick;
17. [SerializeField] **private** **bool** m\_UseHeadBob;
18. [SerializeField] **private** **float** m\_StepInterval;
19. [SerializeField] **private** AudioClip[] m\_FootstepSounds;    // an array of footstep sounds that will be randomly selected from.
20. [SerializeField] **private** AudioClip m\_JumpSound;           // the sound played when character leaves the ground.
21. [SerializeField] **private** AudioClip m\_LandSound;           // the sound played when character touches back on ground.
23. **private** Camera m\_Camera;
24. **private** **bool** m\_Jump;
25. **private** Vector2 m\_Input;
26. **private** Vector3 m\_MoveDir = Vector3.zero;
27. **private** CharacterController m\_CharacterController;
28. **private** CollisionFlags m\_CollisionFlags;
29. **private** **bool** m\_PreviouslyGrounded;
30. **private** Vector3 m\_OriginalCameraPosition;
31. **private** **float** m\_StepCycle;
32. **private** **float** m\_NextStep;
33. **private** **bool** m\_Jumping;
34. **private** AudioSource m\_AudioSource;
35. **private** PhotonView pv;
37. // Use this for initialization
38. **private** **void** Start()
39. {
40. pv = **this**.GetComponent<PhotonView>();
41. m\_CharacterController = GetComponent<CharacterController>();
42. m\_Camera = Camera.main;
43. m\_OriginalCameraPosition = m\_Camera.transform.localPosition;
44. m\_StepCycle = 0f;
45. m\_NextStep = m\_StepCycle / 2f;
46. m\_Jumping = **false**;
47. m\_AudioSource = GetComponent<AudioSource>();
48. }

51. // Update is called once per frame
52. **private** **void** Update()
53. {
54. RotateView();
55. // the jump state needs to read here to make sure it is not missed
56. **if** (!m\_Jump)
57. {
58. m\_Jump = Input.GetButtonDown("Jump");
59. }
61. **if** (!m\_PreviouslyGrounded && m\_CharacterController.isGrounded)
62. {
63. PlayLandingSound();
64. m\_MoveDir.y = 0f;
65. m\_Jumping = **false**;
66. }
67. **if** (!m\_CharacterController.isGrounded && !m\_Jumping && m\_PreviouslyGrounded)
68. {
69. m\_MoveDir.y = 0f;
70. }
72. m\_PreviouslyGrounded = m\_CharacterController.isGrounded;
73. }

76. **private** **void** PlayLandingSound()
77. {
78. m\_AudioSource.clip = m\_LandSound;
79. m\_AudioSource.Play();
80. m\_NextStep = m\_StepCycle + .5f;
81. }

84. **private** **void** FixedUpdate()
85. {
86. **float** speed;
87. GetInput(**out** speed);
88. // always move along the camera forward as it is the direction that it being aimed at
89. Vector3 desiredMove = transform.forward \* m\_Input.y + transform.right \* m\_Input.x;
91. // get a normal for the surface that is being touched to move along it
92. RaycastHit hitInfo;
93. Physics.SphereCast(transform.position, m\_CharacterController.radius, Vector3.down, **out** hitInfo,
94. m\_CharacterController.height / 2f, Physics.AllLayers, QueryTriggerInteraction.Ignore);
95. desiredMove = Vector3.ProjectOnPlane(desiredMove, hitInfo.normal).normalized;
97. m\_MoveDir.x = desiredMove.x \* speed;
98. m\_MoveDir.z = desiredMove.z \* speed;
100. //Debug.Log("m\_MoveDir.x" + m\_MoveDir.x);
101. //Debug.Log("m\_MoveDir.z" + m\_MoveDir.z);
102. //currentGun.SetFloat("Run", Mathf.Abs(m\_MoveDir.x + m\_MoveDir.z));
103. **if** (m\_CharacterController.isGrounded)
104. {
105. m\_MoveDir.y = -m\_StickToGroundForce;
107. **if** (m\_Jump)
108. {
109. m\_MoveDir.y = m\_JumpSpeed;
110. PlayJumpSound();
111. m\_Jump = **false**;
112. m\_Jumping = **true**;
113. }
114. }
115. **else**
116. {
117. m\_MoveDir += Physics.gravity \* m\_GravityMultiplier \* Time.fixedDeltaTime;
118. }
119. m\_CollisionFlags = m\_CharacterController.Move(m\_MoveDir \* Time.fixedDeltaTime);
121. ProgressStepCycle(speed);
122. UpdateCameraPosition(speed);
123. }
124. **private** **void** PlayJumpSound()
125. {
126. m\_AudioSource.clip = m\_JumpSound;
127. m\_AudioSource.Play();
128. }

131. **private** **void** ProgressStepCycle(**float** speed)
132. {
133. **if** (m\_CharacterController.velocity.sqrMagnitude > 0 && (m\_Input.x != 0 || m\_Input.y != 0))
134. {
135. m\_StepCycle += (m\_CharacterController.velocity.magnitude + (speed \* (m\_IsWalking ? 1f : m\_RunstepLenghten))) \*
136. Time.fixedDeltaTime;
137. }
139. **if** (!(m\_StepCycle > m\_NextStep))
140. {
141. **return**;
142. }
144. m\_NextStep = m\_StepCycle + m\_StepInterval;
146. PlayFootStepAudio();
147. //pv.RPC("PlayFootStepAudio", RpcTarget.AllBuffered);
148. }
150. [PunRPC]
151. **private** **void** PlayFootStepAudio()
152. {
153. **if** (!m\_CharacterController.isGrounded)
154. {
155. **return**;
156. }
157. // pick & play a random footstep sound from the array,
158. // excluding sound at index 0
159. **int** n = Random.Range(1, m\_FootstepSounds.Length);
160. m\_AudioSource.clip = m\_FootstepSounds[n];
161. m\_AudioSource.PlayOneShot(m\_AudioSource.clip);
162. m\_FootstepSounds[n] = m\_FootstepSounds[0];
163. m\_FootstepSounds[0] = m\_AudioSource.clip;
164. }

167. **private** **void** UpdateCameraPosition(**float** speed)
168. {
169. Vector3 newCameraPosition;
170. **if** (!m\_UseHeadBob)
171. {
172. **return**;
173. }
174. **if** (m\_CharacterController.velocity.magnitude > 0 && m\_CharacterController.isGrounded)
175. {
176. newCameraPosition = m\_Camera.transform.localPosition;
177. }
178. **else**
179. {
180. newCameraPosition = m\_Camera.transform.localPosition;
181. }
182. m\_Camera.transform.localPosition = newCameraPosition;
183. }

186. **private** **void** GetInput(**out** **float** speed)
187. {
188. //读取输入
189. **float** horizontal = Input.GetAxis("Horizontal");
190. **float** vertical = Input.GetAxis("Vertical");
192. **bool** waswalking = m\_IsWalking;
194. #if !MOBILE\_INPUT
195. // On standalone builds, walk/run speed is modified by a key press.
196. // keep track of whether or not the character is walking or running
197. m\_IsWalking = !Input.GetKey(KeyCode.LeftShift);
198. #endif
199. // set the desired speed to be walking or running
200. speed = m\_IsWalking ? m\_WalkSpeed : m\_RunSpeed;
201. m\_Input = **new** Vector2(horizontal, vertical);
203. // normalize input if it exceeds 1 in combined length:
204. **if** (m\_Input.sqrMagnitude > 1)
205. {
206. m\_Input.Normalize();
207. }
209. // handle speed change to give an fov kick
210. // only if the player is going to a run, is running and the fovkick is to be used
211. **if** (m\_IsWalking != waswalking && m\_UseFovKick && m\_CharacterController.velocity.sqrMagnitude > 0)
212. {
213. StopAllCoroutines();
214. }
215. }

218. **private** **void** RotateView()
219. {
221. }

224. **private** **void** OnControllerColliderHit(ControllerColliderHit hit)
225. {
226. Rigidbody body = hit.collider.attachedRigidbody;
227. //dont move the rigidbody if the character is on top of it
228. **if** (m\_CollisionFlags == CollisionFlags.Below)
229. {
230. **return**;
231. }
233. **if** (body == **null** || body.isKinematic)
234. {
235. **return**;
236. }
237. body.AddForceAtPosition(m\_CharacterController.velocity \* 0.1f, hit.point, ForceMode.Impulse);
238. }
239. }

然后把这个脚本附加到我们的角色身上。

## 任务6 视角控制

## 任务内容

**编写RotateView脚本**

有了角色可以移动的脚本，我们还需要视角控制的脚本，我们需要用鼠标来控制角色的视角，也就是对摄像机进行操作：

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
5. **public** **class** RotateView : MonoBehaviour
6. {
7. //private:
8. //旋转的临时变量
9. **private** **float** rotationX = 0f;
10. **private** **float** humanRotationX = 0f;
11. //摄像机
12. **private** Camera myCamera;
13. //public:
14. //鼠标水平速度
15. **public** **float** sensitivityHor = 2.0f;
16. //鼠标垂直速度
17. **public** **float** sensitivityVert = 2.0f;
18. //鼠标向上最小翻转角度
19. **public** **float** minimumVert = -45.0f;
20. //鼠标向上最大翻转角度
21. **public** **float** maximumVert = 45.0f;
22. //平滑差值系数
23. **public** **float** smoothTime = 2f;
24. **public** GameObject human;
26. **void** Start()
27. {
28. Cursor.lockState = CursorLockMode.Locked;
29. Cursor.visible =**false**;
30. myCamera = **this**.GetComponentInChildren<Camera>();
31. }
33. **void** Update()
34. {
35. Rotate();
37. **if** (Input.GetKeyDown(KeyCode.Escape))
38. {
39. Cursor.lockState = CursorLockMode.None;
40. Cursor.visible = **true**;
41. }
42. **if**(Input.GetKeyUp(KeyCode.Escape))
43. {
44. Cursor.lockState = CursorLockMode.Locked;
45. Cursor.visible = **false**;
46. }
47. }
49. **private** **void** Rotate()
50. {
51. **float** x = Input.GetAxis("Mouse Y");
52. **float** y = Input.GetAxis("Mouse X");
54. rotationX -= x \* sensitivityVert;
55. humanRotationX-= x \* sensitivityVert;
57. rotationX = Mathf.Clamp(rotationX, minimumVert, maximumVert);
58. humanRotationX = Mathf.Clamp(humanRotationX, -10f, 10f);
59. **float** delta = y \* sensitivityHor;
60. **float** rotationY = transform.localEulerAngles.y + delta;
61. **float** smoothX = Smooth(rotationX);
62. **float** smoothXHuman = Smooth(humanRotationX);
63. transform.localEulerAngles = **new** Vector3(0f, rotationY, 0);
64. myCamera.transform.localEulerAngles = **new** Vector3(smoothX, 0, 0);
65. human.transform.localEulerAngles= **new** Vector3(smoothXHuman, 0, 0);
66. }
68. **private** **float** Smooth(**float** x)
69. {
70. **float** smooth = Mathf.Lerp(x, x, 2f \* Time.deltaTime);
71. **return** smooth;
72. }
73. }

## 任务7 射击实现

## 任务内容

**编写ControlGuns脚本**

**使用射线模拟子弹**

让我们实现射击，但是这个脚本会与其他的脚本相连，所以我们可以暂时不用理解这个脚本，等到进行到每个枪械的相关脚本的编写时再回来看也不迟，首先我们实现射击，采用射线的方式来模拟射击会很好，因为子弹的速度很快，所以可以理解成一条光线，然后用这条光线来检测击中物体的信息。

现在编写这个脚本：

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
4. **using** Photon.Pun;
6. //只控制射击
7. **public** **class** ControlGuns : MonoBehaviour
8. {
9. **private** PhotonView pv;
10. //输入管理
11. **private** InputManager inputManager;
12. //模拟射击子弹的射线的摄像机
13. **public** Camera myCamera;
14. //射线位置
15. **private** Vector3 rayPos;
16. //主武器
17. **public** Transform handgun1;
18. //副武器
19. **public** Transform handgun2;
20. //徒手动画
21. **public** Animator hands;
22. //上一次的射击时间
23. **private** **float** lastFired;
24. //开火速率
25. **public** **float** fireRate;
26. //是否自动开火
27. **private** **bool** isAutoFire = **false**;
28. //弹痕
29. **public** GameObject bulletHole;
30. //是否可以射击
31. **private** **bool** isFire;
32. //脱离瞄准时的摄像机
33. **private** Camera nowCamera;
34. //控制开关
35. **private** **bool** clickGo = **true**;
37. **private** **void** Start()
38. {
39. inputManager = **this**.GetComponent<InputManager>();
40. pv = **this**.GetComponent<PhotonView>();
41. nowCamera = myCamera;
42. isFire = **true**;
43. }
45. **private** **void** Update()
46. {
47. **if** (pv.IsMine)
48. {
49. rayPos = myCamera.ViewportToWorldPoint(**new** Vector3(0.5f, 0.5f, 0f));
50. **if** (Input.GetMouseButtonDown(0) || handgun1.childCount != 0 || handgun2.childCount != 0)
51. {
52. hands.SetTrigger("Hand");
53. }
54. Shoot();
55. AutoFireControl();
56. **if** (Input.GetKeyDown(KeyCode.R))
57. {
58. isFire = **false**;
59. Reload();
60. StartCoroutine("ReloadTime");
61. }
62. }
63. }
64. /// <summary>
65. /// 实例射击
66. /// </summary>
67. **private** **void** AppleShoot()
68. {
69. **if** (handgun1.childCount != 0 || handgun2.childCount != 0)
70. {
71. **if** (handgun1.GetChild(0).gameObject.activeInHierarchy == **true**)
72. {
73. **if** (handgun1.GetChild(0).gameObject.tag == "AK47")
74. {
75. **if** (handgun1.GetChild(0).gameObject.GetComponent<Akm>().bulletsAmount > 0)
76. {
77. handgun1.GetChild(0).gameObject.GetComponent<Akm>().useBullets();
78. ShootRay(100, 25);
79. handgun1.GetChild(0).gameObject.GetComponent<Animator>().SetTrigger("AimFire");
80. }
81. }
82. **else** **if** (handgun1.GetChild(0).gameObject.tag == "M4A1")
83. {
84. **if** (handgun1.GetChild(0).gameObject.GetComponent<Scar>().bulletsAmount > 0)
85. {
86. handgun1.GetChild(0).gameObject.GetComponent<Scar>().useBullets();
87. ShootRay(100, 20);
88. }
89. }
90. **else** **if** (handgun1.GetChild(0).gameObject.tag == "Barrett")
91. {
92. **if** (handgun1.GetChild(0).gameObject.GetComponent<Sinper>().bulletsAmount > 0)
93. {
94. handgun1.GetChild(0).gameObject.GetComponent<Sinper>().useBullets();
95. ShootRay(1000, 90);
96. }
97. }
98. **else** **if** (handgun1.GetChild(0).gameObject.tag == "Shotgun")
99. {
100. **if** (handgun1.GetChild(0).gameObject.GetComponent<Lever>().bulletsAmount > 0)
101. {
102. handgun1.GetChild(0).gameObject.GetComponent<Lever>().useBullets();
103. ShootRay(10, 50);
104. }
105. }
106. **else** **if** (handgun1.GetChild(0).gameObject.tag == "SMG")
107. {
108. **if** (handgun1.GetChild(0).gameObject.GetComponent<Smg>().bulletsAmount > 0)
109. {
110. handgun1.GetChild(0).gameObject.GetComponent<Smg>().UseBullets();
111. ShootRay(50, 10);
112. }
113. }
114. }
115. **else** **if** (handgun2.GetChild(0).gameObject.activeInHierarchy == **true**)
116. {
117. **if** (handgun2.GetChild(0).gameObject.tag == "AK47")
118. {
119. **if** (handgun2.GetChild(0).gameObject.GetComponent<Akm>().bulletsAmount > 0)
120. {
121. handgun2.GetChild(0).gameObject.GetComponent<Akm>().useBullets();
122. ShootRay(100, 25);
123. }
124. }
125. **else** **if** (handgun2.GetChild(0).gameObject.tag == "M4A1")
126. {
127. **if** (handgun2.GetChild(0).gameObject.GetComponent<Scar>().bulletsAmount > 0)
128. {
129. handgun2.GetChild(0).gameObject.GetComponent<Scar>().useBullets();
130. ShootRay(100, 20);
131. }
132. }
133. **else** **if** (handgun2.GetChild(0).gameObject.tag == "Barrett")
134. {
135. **if** (handgun2.GetChild(0).gameObject.GetComponent<Sinper>().bulletsAmount > 0)
136. {
137. handgun2.GetChild(0).gameObject.GetComponent<Sinper>().useBullets();
138. ShootRay(1000, 90);
139. }
140. }
141. **else** **if** (handgun2.GetChild(0).gameObject.tag == "Shotgun")
142. {
143. **if** (handgun2.GetChild(0).gameObject.GetComponent<Lever>().bulletsAmount > 0)
144. {
145. handgun2.GetChild(0).gameObject.GetComponent<Lever>().useBullets();
146. ShootRay(10, 50);
147. }
148. }
149. **else** **if** (handgun2.GetChild(0).gameObject.tag == "SMG")
150. {
151. **if** (handgun2.GetChild(0).gameObject.GetComponent<Smg>().bulletsAmount > 0)
152. {
153. handgun2.GetChild(0).gameObject.GetComponent<Smg>().UseBullets();
154. ShootRay(50, 10);
155. }
156. }
157. }
158. }
159. }
160. /// <summary>
161. /// 开始射击
162. /// </summary>
163. **private** **void** Shoot()
164. {
165. //单发模式
166. **if** (Input.GetMouseButtonDown(0) && !isAutoFire && isFire == **true**)
167. {
168. AppleShoot();
169. }
170. //全自动模式
171. **if** (Input.GetMouseButton(0) && isAutoFire == **true** && isFire == **true**)
172. {
173. **if** (Time.time - lastFired > 1 / fireRate)
174. {
175. AppleShoot();
176. lastFired = Time.time;
177. }
178. }
179. **else**
180. {
181. Debug.Log("Not");
182. }
183. }
184. /// <summary>
185. /// 模拟子弹的射线
186. /// </summary>
187. /// <param name="distance"></param>
188. /// <param name="power"></param>
189. **private** **void** ShootRay(**float** distance, **int** power)
190. {
191. rayPos = nowCamera.ViewportToWorldPoint(**new** Vector3(0.5f, Random.Range(0.5f, 1.0f), 0.0f));
192. Ray ray = **new** Ray(rayPos, nowCamera.transform.forward);
193. **if** (Physics.Raycast(ray, **out** RaycastHit hit, distance))
194. {
195. //如果击中敌人
196. **if** (hit.collider.gameObject.tag == "Player")
197. {
198. //调用敌人的减血代码
199. hit.transform.GetComponent<PhotonView>().RPC("DamageGet", RpcTarget.AllBuffered, power, hit.point);
200. //生成一个临时弹孔
201. GameObject tempHole = Instantiate(bulletHole, hit.point, Quaternion.FromToRotation(Vector3.forward, hit.normal));
202. //0.3s后销毁弹孔
203. Destroy(tempHole, 0.3f);
204. }
205. //如果击中建筑物
206. **else** **if** (hit.collider.gameObject.tag == "buildings")
207. {
208. //生成一个临时弹孔
209. GameObject tempHole = Instantiate(bulletHole, hit.point, Quaternion.FromToRotation(Vector3.forward, hit.normal));
210. Destroy(tempHole, 0.3f);
211. }
212. }
213. }
214. /// <summary>
215. /// 开火模式
216. /// </summary>
217. **private** **void** AutoFireControl()
218. {
219. **if** (Input.GetKeyDown(KeyCode.B))
220. {
221. **if** (clickGo == **true**)
222. {
223. isAutoFire = **true**;
224. clickGo = **false**;
225. Debug.Log("自动");
226. **this**.GetComponent<UIManager>().isAuto.text = "自动";
227. }
228. **else**
229. {
230. isAutoFire = **false**;
231. clickGo = **true**;
232. Debug.Log("单点");
233. **this**.GetComponent<UIManager>().isAuto.text = "单发";
234. }
235. }
236. }
237. /// <summary>
238. /// 重新装弹
239. /// </summary>
240. **private** **void** Reload()
241. {
242. **if** (handgun1.childCount != 0 || handgun2.childCount != 0)
243. {
244. **if** (handgun1.GetChild(0).gameObject.activeInHierarchy == **true**)
245. {
246. **if** (handgun1.GetChild(0).gameObject.tag == "AK47")
247. {
248. **if** (handgun1.GetChild(0).gameObject.GetComponent<Akm>().bulletsAmount < 30)
249. {
250. handgun1.GetChild(0).gameObject.GetComponent<Akm>().Reload();
251. }
252. }
253. **else** **if** (handgun1.GetChild(0).gameObject.tag == "M4A1")
254. {
255. **if** (handgun1.GetChild(0).gameObject.GetComponent<Scar>().bulletsAmount < 30)
256. {
257. handgun1.GetChild(0).gameObject.GetComponent<Scar>().Reload();
258. }
259. }
260. **else** **if** (handgun1.GetChild(0).gameObject.tag == "Barrett")
261. {
262. **if** (handgun1.GetChild(0).gameObject.GetComponent<Sinper>().bulletsAmount < 5)
263. {
264. handgun1.GetChild(0).gameObject.GetComponent<Sinper>().Reload();
265. }
266. }
267. **else** **if** (handgun1.GetChild(0).gameObject.tag == "Shotgun")
268. {
269. **if** (handgun1.GetChild(0).gameObject.GetComponent<Lever>().bulletsAmount < 2)
270. {
271. handgun1.GetChild(0).gameObject.GetComponent<Lever>().Reload();
272. }
273. }
274. **else** **if** (handgun1.GetChild(0).gameObject.tag == "SMG")
275. {
276. **if** (handgun1.GetChild(0).gameObject.GetComponent<Smg>().bulletsAmount < 25)
277. {
278. handgun1.GetChild(0).gameObject.GetComponent<Smg>().Reload();
279. }
280. }
281. }
282. **else** **if** (handgun2.GetChild(0).gameObject.activeInHierarchy == **true**)
283. {
284. **if** (handgun2.GetChild(0).gameObject.tag == "AK47")
285. {
286. **if** (handgun2.GetChild(0).gameObject.GetComponent<Akm>().bulletsAmount < 30)
287. {
288. handgun2.GetChild(0).gameObject.GetComponent<Akm>().Reload();
289. }
290. }
291. **else** **if** (handgun2.GetChild(0).gameObject.tag == "M4A1")
292. {
293. **if** (handgun2.GetChild(0).gameObject.GetComponent<Scar>().bulletsAmount < 30)
294. {
295. handgun2.GetChild(0).gameObject.GetComponent<Scar>().Reload();
296. }
297. }
298. **else** **if** (handgun2.GetChild(0).gameObject.tag == "Barrett")
299. {
300. **if** (handgun2.GetChild(0).gameObject.GetComponent<Sinper>().bulletsAmount < 5)
301. {
302. handgun2.GetChild(0).gameObject.GetComponent<Sinper>().Reload();
303. }
304. }
305. **else** **if** (handgun2.GetChild(0).gameObject.tag == "Shotgun")
306. {
307. **if** (handgun2.GetChild(0).gameObject.GetComponent<Lever>().bulletsAmount < 2)
308. {
309. handgun2.GetChild(0).gameObject.GetComponent<Lever>().Reload();
310. }
311. }
312. **else** **if** (handgun2.GetChild(0).gameObject.tag == "SMG")
313. {
314. **if** (handgun2.GetChild(0).gameObject.GetComponent<Smg>().bulletsAmount < 25)
315. {
316. handgun2.GetChild(0).gameObject.GetComponent<Smg>().Reload();
317. }
318. }
319. }
320. }
321. }
322. /// <summary>
323. /// 子弹填充的时间
324. /// </summary>
325. /// <returns></returns>
326. IEnumerator ReloadTime()
327. {
328. yield **return** **new** WaitForSeconds(1.4f);
329. isFire = **true**;
330. }
331. }

## 任务8 射击模式设置

## 任务内容

**实现射击模式的切换**

我们不可能捡到枪后就一直摁一下鼠标就发射一颗子弹，这样很不方便，所以，我们要让手中的枪变成连发的模式，想想，不累手的操作是最好的，现在打开上次人物的脚本，继续编写：

我们加一个可以函数就可以了，把它放在Update里做每一帧都检测目前的开火模式：

1. /// <summary>
2. /// 开火模式
3. /// </summary>
4. **private** **void** AutoFireControl()
5. {
6. **if** (Input.GetKeyDown(KeyCode.B))
7. {
8. **if** (clickGo == **true**)
9. {
10. isAutoFire = **true**;
11. clickGo = **false**;
12. Debug.Log("自动");
13. **this**.GetComponent<UIManager>().isAuto.text = "自动";
14. }
15. **else**
16. {
17. isAutoFire = **false**;
18. clickGo = **true**;
19. Debug.Log("单点");
20. **this**.GetComponent<UIManager>().isAuto.text = "单发";
21. }
22. }
23. }

## 任务9 武器和物品拾取

**制作武器的预制**

**编写PickItems脚本**

为了让游戏更公平一些，我们让角色一出生就没有任何武器可以用，直到捡到武器。在第7次任务时，你可能对ControlGuns脚本里的一些变量的指定不太明确，比如：handGuns1和handGuns2，其实ControlGuns脚本和PickItems脚本里都有这两个变量，它们的作用是判断主武器和副武器的位置上是否有枪存在，只有这样我们才能进行注入切枪和扔枪的操作。开始实现武器的拾取：

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
4. **using** UnityEngine.UI;
5. **using** Photon.Pun;
7. //只控制拾取物品
9. **public** **class** PickItem : MonoBehaviour
10. {
11. #region private
12. **private** Vector3 rayPos;
13. //绷带数量
14. **private** **int** bandageAmount;
15. //饮料数量
16. **private** **int** drinkAmount;
17. #endregion
18. #region public
19. **public** Camera myCamera;
20. //头盔
21. **public** GameObject helmet;
22. //防弹衣
23. **public** GameObject armor;
24. //背包
25. **public** GameObject backpack;
27. **public** GameObject[] guns;
29. **private** PhotonView pv;
30. //所有的物品，包括枪
31. **public** GameObject[] items;
32. **public** Transform handgun1;
33. **public** Transform handgun2;
34. **public** Transform handAndGuns;
35. **private** **bool** gun1State; //判断1号位是否有枪
36. **private** **bool** gun2State; //判断1号位是否有枪
37. **public** Animator hands;
38. //远程玩家的枪
39. **public** GameObject[] gunRemove;
40. **public** GameObject handsMelle;
41. #endregion
42. // Start is called before the first frame update
43. **void** Start()
44. {
45. pv = **this**.GetComponent<PhotonView>();
46. bandageAmount = 0;
47. drinkAmount = 0;
48. }
50. // Update is called once per frame
51. **void** Update()
52. {
53. **if** (pv.IsMine)
54. {
55. AroundWeapons();
56. SwitchGuns();
57. **this**.GetComponent<UIManager>().bandageAmountText.text = "绷带 x " + bandageAmount.ToString();
58. **this**.GetComponent<UIManager>().drinkAmountText.text = "饮料 x " + drinkAmount.ToString();
59. rayPos = myCamera.ViewportToWorldPoint(**new** Vector3(0.5f, 0.5f, 0f));
60. **if** (Input.GetKeyDown(KeyCode.F))
61. {
62. Ray ray = **new** Ray(rayPos, myCamera.transform.forward);
63. **if** (Physics.Raycast(ray, **out** RaycastHit hit, 3, 1 << 12))
64. {
65. PickItems(hit.collider.gameObject.tag);
66. hit.collider.gameObject.GetComponent<CanPickObject>().DestoryThisObject();
67. }
68. }
69. }
70. }
71. /// <summary>
72. /// 拾取物品
73. /// </summary>
74. **public** **void** AroundWeapons()
75. {
76. rayPos = myCamera.ViewportToWorldPoint(**new** Vector3(0.5f, 0.5f, 0f));
77. Ray ray = **new** Ray(rayPos, myCamera.transform.forward);
78. **if** (Physics.Raycast(ray, **out** RaycastHit hit, 3, 1 << 12))
79. {
80. **this**.GetComponent<UIManager>().itemText.gameObject.SetActive(**true**);
81. **switch** (hit.collider.gameObject.tag)
82. {
83. **case** "helmet":
84. **this**.GetComponent<UIManager>().itemText.text = "拾取头盔";
85. **if** (Input.GetKeyDown(KeyCode.F))
86. {
87. //仅远程玩家可以看到头盔
88. pv.RPC("RemovePlayerPickItem", RpcTarget.AllBuffered, hit.collider.gameObject.tag);
89. hit.collider.gameObject.GetComponent<PhotonView>().RPC("DestoryThisObject", RpcTarget.AllBuffered);
90. }
91. **break**;
92. **case** "armor":
93. **this**.GetComponent<UIManager>().itemText.text = "拾取防弹衣";
94. **if** (Input.GetKeyDown(KeyCode.F))
95. {
96. pv.RPC("RemovePlayerPickItem", RpcTarget.AllBuffered, hit.collider.gameObject.tag);
97. hit.collider.gameObject.GetComponent<PhotonView>().RPC("DestoryThisObject", RpcTarget.AllBuffered);
98. }
99. **break**;
100. **case** "Bandage":
101. **this**.GetComponent<UIManager>().itemText.text = "拾取绷带";
102. **if** (Input.GetKeyDown(KeyCode.F))
103. {
104. pv.RPC("RemovePlayerPickItem", RpcTarget.AllBuffered, hit.collider.gameObject.tag);
105. hit.collider.gameObject.GetComponent<PhotonView>().RPC("DestoryThisObject", RpcTarget.AllBuffered);
106. Debug.Log("已经拾取了绷带");
107. //背包绷带数加一
108. bandageAmount++;
109. }
110. **break**;
111. **case** "Backpack":
112. **this**.GetComponent<UIManager>().itemText.text = "拾取背包";
113. **if** (Input.GetKeyDown(KeyCode.F))
114. {
115. pv.RPC("RemovePlayerPickItem", RpcTarget.AllBuffered, hit.collider.gameObject.tag);
116. hit.collider.gameObject.GetComponent<PhotonView>().RPC("DestoryThisObject", RpcTarget.AllBuffered);
117. }
118. **break**;
119. **case** "Drink":
120. **this**.GetComponent<UIManager>().itemText.text = "拾取饮料";
121. **if** (Input.GetKeyDown(KeyCode.F))
122. {
123. pv.RPC("RemovePlayerPickItem", RpcTarget.AllBuffered, hit.collider.gameObject.tag);
124. hit.collider.gameObject.GetComponent<PhotonView>().RPC("DestoryThisObject", RpcTarget.AllBuffered);
125. drinkAmount++;
126. }
127. **break**;
128. **case** "Barrett":
129. **this**.GetComponent<UIManager>().itemText.text = "拾取Barrett";
130. **break**;
131. **case** "AK47":
132. **this**.GetComponent<UIManager>().itemText.text = "拾取AK47";
133. **break**;
134. **case** "SMG":
135. **this**.GetComponent<UIManager>().itemText.text = "拾取SMG";
136. **break**;
137. **case** "M4A1":
138. **this**.GetComponent<UIManager>().itemText.text = "拾取M4A1";
139. **break**;
140. **case** "Shotgun":
141. **this**.GetComponent<UIManager>().itemText.text = "拾取Shotgun";
142. **break**;
143. **case** "556":
144. **this**.GetComponent<UIManager>().itemText.text = "拾取5.56子弹";
145. **if** (Input.GetKeyDown(KeyCode.F))
146. {
147. guns[0].GetComponentInChildren<Akm>().backupBullets = 60;
148. guns[1].GetComponentInChildren<Sinper>().backupBullets = 30;
149. guns[2].GetComponentInChildren<Scar>().backupBullets = 60;
150. guns[3].GetComponentInChildren<Lever>().backupBullets = 60;
151. guns[4].GetComponentInChildren<Smg>().backupBullets = 60;
152. }
153. **break**;
154. }
155. }
156. **else**
157. {
158. **this**.GetComponent<UIManager>().itemText.text = **null**;
159. **this**.GetComponent<UIManager>().itemText.gameObject.SetActive(**false**);
160. }
161. }
162. /// <summary>
163. /// 拾取枪
164. /// </summary>
165. /// <param name="itemName"></param>
166. **private** **void** PickItems(**string** itemName)
167. {
168. **for** (**int** i = 0; i < items.Length; i++)
169. {
170. **if** (items[i].tag == itemName)
171. {
172. **if** (gun1State == **false**)
173. {
174. //把捡到的枪放在手中一号枪位
175. items[i].transform.SetParent(handgun1);
176. //开启与拾取到的东西相同的物品
177. items[i].SetActive(**true**);
178. gun1State = **true**; //此时1号位有一把枪
179. GunImageControl(itemName);
180. pv.RPC("RemovePlayerPick", RpcTarget.AllBuffered, itemName);
181. **this**.GetComponent<UIManager>().isAuto.gameObject.SetActive(**true**);
182. handsMelle.SetActive(**false**); //关闭手
183. **this**.GetComponent<UIManager>().bulletsControl.SetActive(**true**);
184. }
185. **else** **if** (gun1State == **true** && gun2State == **false**)
186. {
187. **if** (handgun1.GetChild(0).gameObject.tag == itemName)
188. {
189. Debug.Log("有一把相同的枪");
190. }
191. **else**
192. {
193. //把捡到的枪放在手中一号枪位
194. items[i].transform.SetParent(handgun2);
195. //关闭与拾取到的东西相同的物品
196. items[i].SetActive(**false**);
197. gun2State = **true**;
198. }
199. }
200. **else** **if** (gun1State == **true** && gun2State == **true**)
201. {
202. **if** (items[i].tag == handgun1.GetChild(0).gameObject.tag)
203. {
204. Debug.Log("相同的枪");
205. }
206. **else** **if** (items[i].tag == handgun1.GetChild(0).gameObject.tag && items[i].tag == handgun2.GetChild(0).gameObject.tag)
207. {
208. handgun1.GetChild(0).gameObject.SetActive(**false**);
209. handgun2.GetChild(0).gameObject.SetActive(**true**);
210. }
211. **else** **if** (items[i].tag != handgun1.GetChild(0).gameObject.tag && items[i].tag != handgun2.GetChild(0).gameObject.tag)
212. {
213. **if** (handgun2.GetChild(0).gameObject.activeInHierarchy == **true**)
214. {
215. handgun1.GetChild(0).gameObject.SetActive(**false**);
216. handgun1.GetChild(0).gameObject.transform.SetParent(handAndGuns);
217. //把捡到的枪放在手中一号枪位
218. items[i].transform.SetParent(handgun1);
219. //关闭与拾取到的东西相同的物品
220. items[i].SetActive(**false**);
221. }
222. **else** **if** (handgun1.GetChild(0).gameObject.activeInHierarchy == **true**)
223. {
224. handgun2.GetChild(0).gameObject.SetActive(**false**);
225. handgun2.GetChild(0).gameObject.transform.SetParent(handAndGuns);
226. //把捡到的枪放在手中一号枪位
227. items[i].transform.SetParent(handgun2);
228. //关闭与拾取到的东西相同的物品
229. items[i].SetActive(**false**);
230. }
231. }
232. }
233. }
234. }
235. }
236. /// <summary>
237. /// 切换枪
238. /// </summary>
239. **private** **void** SwitchGuns()
240. {
241. **if** (Input.GetKeyDown(KeyCode.Alpha1))
242. {
243. **if** (handgun1.childCount > 0 && handgun2.childCount > 0)
244. {
245. handgun1.GetChild(0).gameObject.SetActive(**true**);
246. handgun2.GetChild(0).gameObject.SetActive(**false**);
247. GunImageControl(handgun1.GetChild(0).gameObject.tag);
248. handsMelle.SetActive(**false**);
249. pv.RPC("RemovePlayerPick", RpcTarget.AllBuffered, handgun1.GetChild(0).gameObject.tag);
250. }
251. }
252. **if** (Input.GetKeyDown(KeyCode.Alpha2))
253. {
254. **if** (handgun1.childCount > 0 && handgun2.childCount > 0)
255. {
256. handgun1.GetChild(0).gameObject.SetActive(**false**);
257. handgun2.GetChild(0).gameObject.SetActive(**true**);
258. GunImageControl(handgun2.GetChild(0).gameObject.tag);
259. handsMelle.SetActive(**false**);
260. pv.RPC("RemovePlayerPick", RpcTarget.AllBuffered, handgun2.GetChild(0).gameObject.tag);
261. }
262. }
263. }
264. /// <summary>
265. /// 枪械图标显示
266. /// </summary>
267. **private** **void** GunImageControl()
268. {
269. **this**.GetComponent<UIManager>().gunImage[0].gameObject.SetActive(**false**);
270. **this**.GetComponent<UIManager>().gunImage[1].gameObject.SetActive(**false**);
271. **this**.GetComponent<UIManager>().gunImage[2].gameObject.SetActive(**false**);
272. **this**.GetComponent<UIManager>().gunImage[3].gameObject.SetActive(**false**);
273. **this**.GetComponent<UIManager>().gunImage[4].gameObject.SetActive(**false**);
274. }
275. /// <summary>
276. /// 枪械图标控制
277. /// </summary>
278. /// <param name="itemName"></param>
279. **private** **void** GunImageControl(**string** itemName)
280. {
281. **switch** (itemName)
282. {
283. **case** "AK47":
284. GunImageControl();
285. **this**.GetComponent<UIManager>().gunImage[1].gameObject.SetActive(**true**);
286. **break**;
287. **case** "Barrett":
288. GunImageControl();
289. **this**.GetComponent<UIManager>().gunImage[0].gameObject.SetActive(**true**);
290. **break**;
291. **case** "ShotGun":
292. GunImageControl();
293. **this**.GetComponent<UIManager>().gunImage[4].gameObject.SetActive(**true**);
294. **break**;
295. **case** "M4A1":
296. GunImageControl();
297. **this**.GetComponent<UIManager>().gunImage[2].gameObject.SetActive(**true**);
298. **break**;
299. **case** "SMG":
300. GunImageControl();
301. **this**.GetComponent<UIManager>().gunImage[3].gameObject.SetActive(**true**);
302. **break**;
303. **case** "Pistol":
304. GunImageControl();
305. **this**.GetComponent<UIManager>().gunImage[2].gameObject.SetActive(**true**);
306. **break**;
307. **case** "SandEagle":
308. GunImageControl();
309. **this**.GetComponent<UIManager>().gunImage[2].gameObject.SetActive(**true**);
310. **break**;
311. }
312. }
313. /// <summary>
314. /// 远程玩家拾取头盔
315. /// </summary>
316. /// <param name="item"></param>
317. [PunRPC]
318. **private** **void** RemovePlayerPickItem(**string** item)
319. {
320. **if** (item == "helmet")
321. {
322. helmet.SetActive(**true**);
323. **this**.GetComponent<UIManager>().helemet.gameObject.SetActive(**true**);
324. }
325. **else** **if** (item == "Backpack")
326. {
327. backpack.SetActive(**true**);
328. **this**.GetComponent<UIManager>().backpage.gameObject.SetActive(**true**);
329. }
330. **else** **if** (item == "armor")
331. {
332. armor.SetActive(**true**);
333. **this**.GetComponent<UIManager>().aromor.gameObject.SetActive(**true**);
334. }
335. }
336. /// <summary>
337. /// 远程玩家拾取枪
338. /// </summary>
339. /// <param name="name"></param>
340. [PunRPC]
341. **private** **void** RemovePlayerPick(**string** name)
342. {
343. **switch** (name)
344. {
345. **case** "AK47":
346. {
347. gunRemove[2].SetActive(**true**);
348. gunRemove[0].SetActive(**false**);
349. gunRemove[1].SetActive(**false**);
350. gunRemove[3].SetActive(**false**);
351. gunRemove[4].SetActive(**false**);
352. }
353. **break**;
354. **case** "M4A1":
355. {
356. gunRemove[1].SetActive(**true**);
357. gunRemove[0].SetActive(**false**);
358. gunRemove[2].SetActive(**false**);
359. gunRemove[3].SetActive(**false**);
360. gunRemove[4].SetActive(**false**);
361. }
362. **break**;
363. **case** "Barrett":
364. {
365. gunRemove[0].SetActive(**true**);
366. gunRemove[1].SetActive(**false**);
367. gunRemove[2].SetActive(**false**);
368. gunRemove[3].SetActive(**false**);
369. gunRemove[4].SetActive(**false**);
370. }
371. **break**;
372. **case** "SMG":
373. {
374. gunRemove[3].SetActive(**true**);
375. gunRemove[0].SetActive(**false**);
376. gunRemove[1].SetActive(**false**);
377. gunRemove[2].SetActive(**false**);
378. gunRemove[4].SetActive(**false**);
379. }
380. **break**;
381. **case** "Shotgun":
382. {
383. gunRemove[4].SetActive(**true**);
384. gunRemove[0].SetActive(**false**);
385. gunRemove[1].SetActive(**false**);
386. gunRemove[2].SetActive(**false**);
387. gunRemove[3].SetActive(**false**);
388. }
389. **break**;
390. }
391. }
392. }

## 任务10 武器切换

## 任务内容

**编写武器切换的脚本**

我们继续上次任务中的PickItems脚本编写，我解释一下为什捡枪和切换武器要在一个脚本中，我的理解是这些操作都属于人物的动作，是人来操作的，所以可以类亏到一起。我们继续编写这个脚本

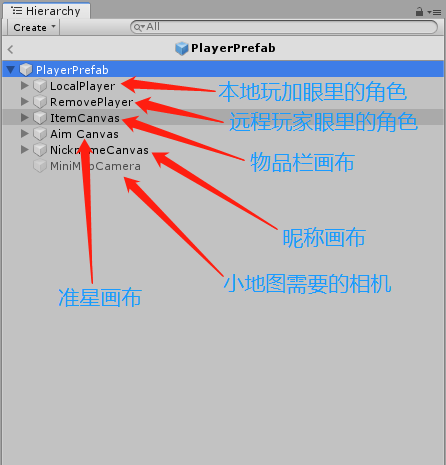
1. /// <summary>
2. /// 切换枪
3. /// </summary>
4. **private** **void** SwitchGuns()
5. {
6. **if** (Input.GetKeyDown(KeyCode.Alpha1))
7. {
8. **if** (handgun1.childCount > 0 && handgun2.childCount > 0)
9. {
10. handgun1.GetChild(0).gameObject.SetActive(**true**);
11. handgun2.GetChild(0).gameObject.SetActive(**false**);
12. GunImageControl(handgun1.GetChild(0).gameObject.tag);
13. handsMelle.SetActive(**false**);
14. pv.RPC("RemovePlayerPick", RpcTarget.AllBuffered, handgun1.GetChild(0).gameObject.tag);
15. }
16. }
17. **if** (Input.GetKeyDown(KeyCode.Alpha2))
18. {
19. **if** (handgun1.childCount > 0 && handgun2.childCount > 0)
20. {
21. handgun1.GetChild(0).gameObject.SetActive(**false**);
22. handgun2.GetChild(0).gameObject.SetActive(**true**);
23. GunImageControl(handgun2.GetChild(0).gameObject.tag);
24. handsMelle.SetActive(**false**);
25. pv.RPC("RemovePlayerPick", RpcTarget.AllBuffered, handgun2.GetChild(0).gameObject.tag);
26. }
27. }
28. }

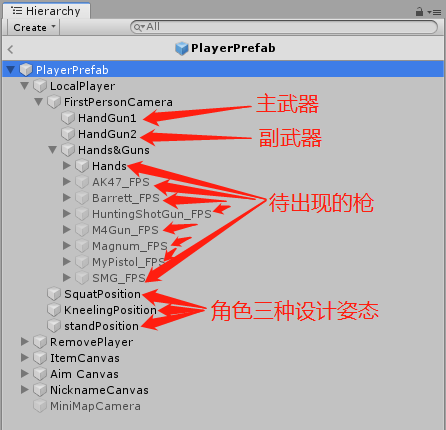
## 任务11 枪械和人物分离

## 任务内容

## **实现枪械与人物分离（人触发枪的行为）**

首先，枪不能自己开枪，只有人的手指触发扳机枪才会开火，所以我们可以把枪单独新建一个类：专门控制枪械开火的。当然，这个类肯定是依附在枪实体上的。而人只要触发就可以开火了。到目前位置我们的人物子物体层级关系还没有展现出来，来设计一下人物的层级关系：





现在我们开始编写每个枪的属性脚本：

拿Ak47的枪械属性脚本来说:

1. **using** System.Collections;
2. **using** System.Collections.Generic;
3. **using** UnityEngine;
4. **using** Photon.Pun;
6. **public** **class** Akm : MonoBehaviour
7. {
8. //30发子弹
9. **public** **int** bulletsAmount = 30;
10. //备用子弹数
11. **public** **int** backupBullets = 30;
12. **public** AudioClip[] scarAudioClips;
13. **public** AudioSource scarSource;
14. **public** Animator gunAnimator;
15. **public** Animator gunAnimatorRemove;
16. **public** PhotonView pv;
17. **public** **int** ClipSize = 30;
18. **void** Update()
19. {
20. **this**.GetComponentInParent<UIManager>().bulletsAmountText.text = bulletsAmount.ToString();
21. **this**.GetComponentInParent<UIManager>().backupBulletsAmounts.text = backupBullets.ToString();
22. }
23. **public** **void** useBullets()
24. {
25. **if** (bulletsAmount <= 0)
26. {
27. scarSource.clip = scarAudioClips[1];
28. scarSource.Play();
29. }
30. **else**
31. {
32. bulletsAmount--;
33. scarSource.clip = scarAudioClips[0];
34. scarSource.Play();
35. gunAnimator.SetTrigger("Shoot");
36. gunAnimatorRemove.SetTrigger("Shoot");
37. }
38. }
40. **public** **void** Reload()
41. {
42. **if** (backupBullets <= 0)
43. {
44. Debug.Log("无法换弹");
45. **return**;
46. }
47. **else** **if** (backupBullets > 0 && backupBullets < 30)
48. {
49. bulletsAmount = backupBullets;
50. }
51. **else**
52. {
53. bulletsAmount = backupBullets - (backupBullets - 30);
54. backupBullets = backupBullets - 30;
56. }
57. gunAnimator.SetTrigger("Reload");
58. gunAnimatorRemove.SetTrigger("Reload");
59. pv.RPC("PlayAudio", RpcTarget.AllBuffered, 2);
60. }
62. [PunRPC]
63. **public** **void** PlayAudio(**int** state)
64. {
65. **if** (state == 0)
66. {
67. scarSource.clip = scarAudioClips[1];
68. scarSource.Play();
69. }
70. **else** **if** (state == 1)
71. {
72. scarSource.clip = scarAudioClips[0];
73. scarSource.Play();
74. }
75. **else** **if** (state == 2)
76. {
77. scarSource.clip = scarAudioClips[2];
78. scarSource.Play();
79. }
80. }
81. }

其他的枪械与Ak47的属性都是一样的结构。

## 任务12 子弹设置

## 任务内容

**设置一下子弹的控制**

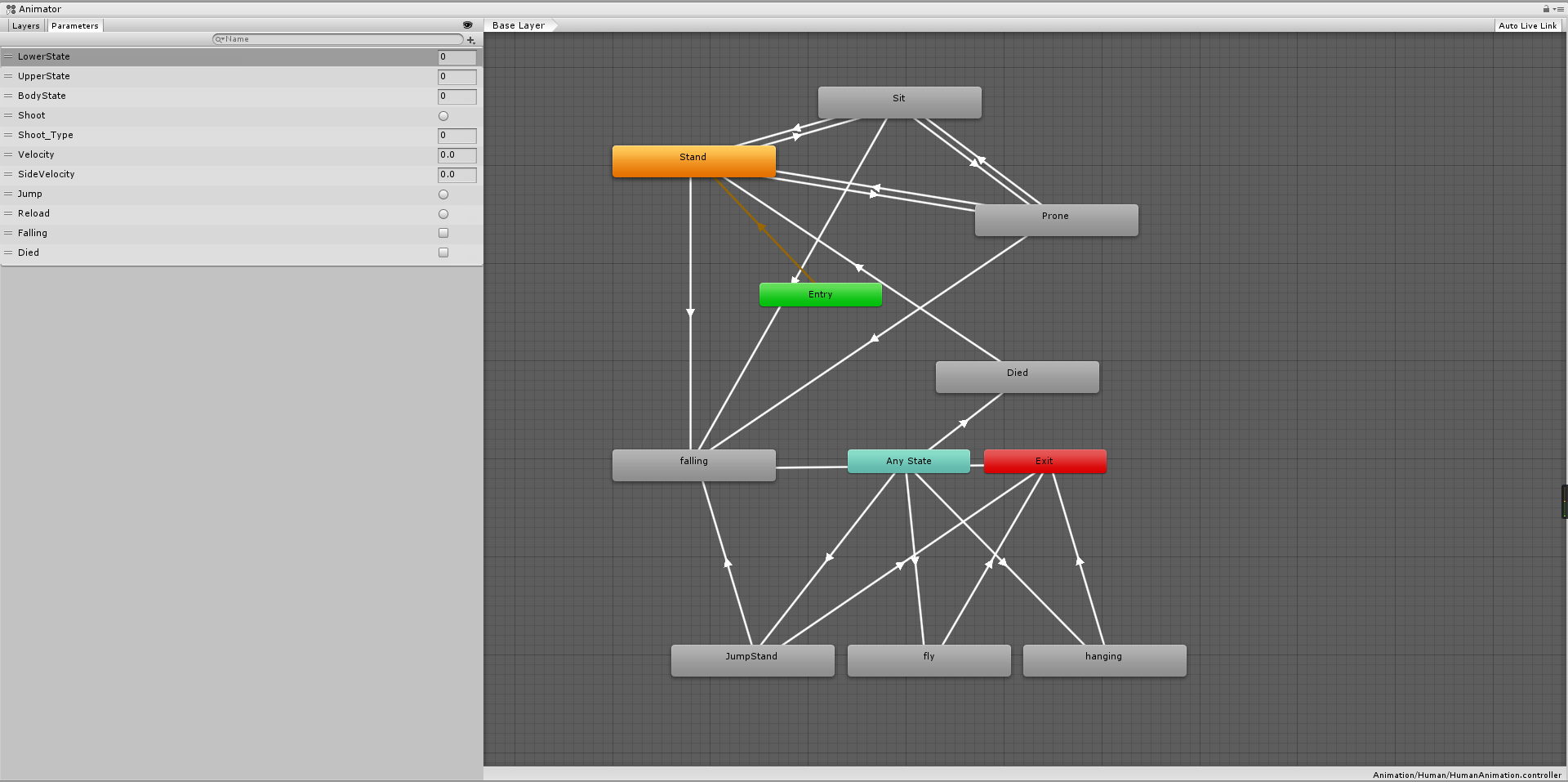
1. //30发子弹
2. **public** **int** bulletsAmount = 30;
3. //备用子弹数
4. **public** **int** backupBullets = 30;
5. **public** AudioClip[] scarAudioClips;
6. **public** AudioSource scarSource;
7. **public** Animator gunAnimator;
8. **public** Animator gunAnimatorRemove;
9. **public** PhotonView pv;
10. **public** **int** ClipSize = 30;

## 任务13 动画设置

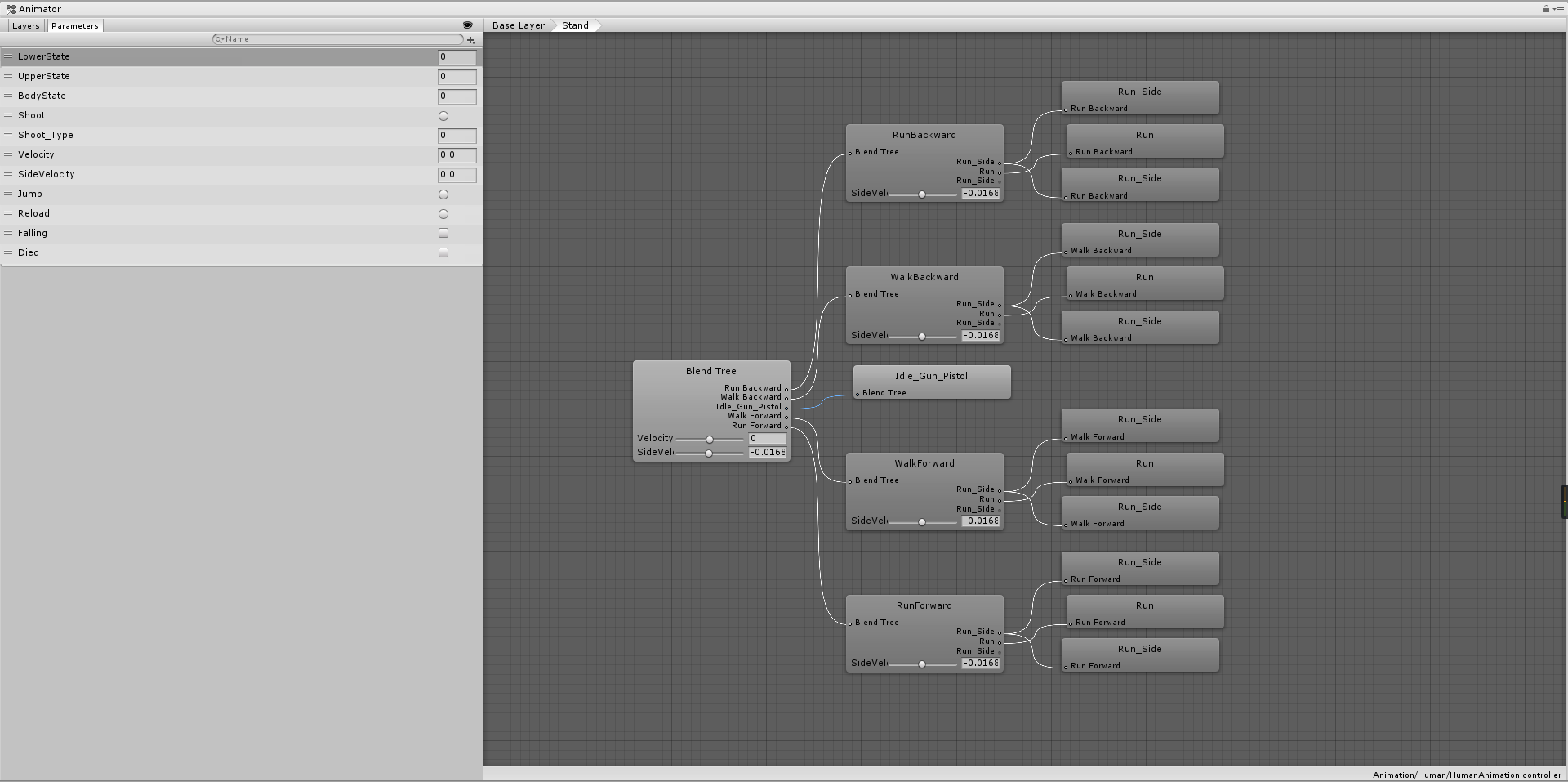
## 任务内容

**实现角色的动画**

**打开Unity3D的我们项目，在Windows->Animation找到Animator并打开：**

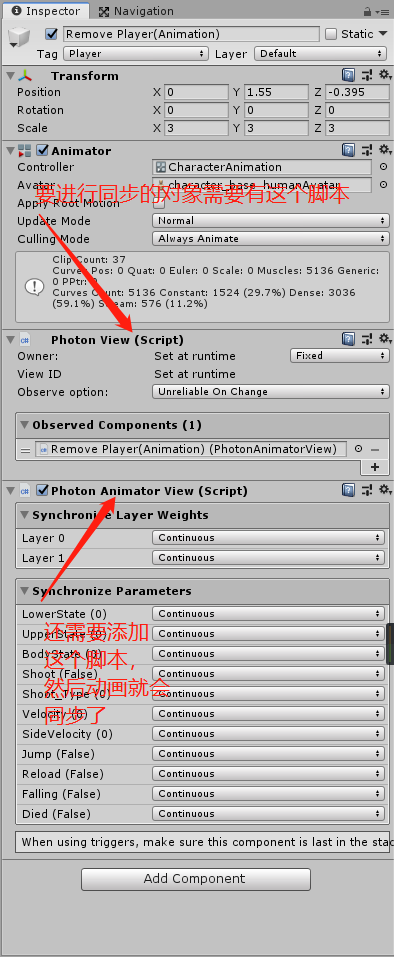


可以看到这个界面，这就是动画状态机，只要我们设置一些动作的触发参数我们就可以实现角色动作的控制。为了让角色更好的模拟现实所以我们使用状态混合树来控制可以更加精确。你双击那个黄色的按钮就会额外打开一个界面：



这个就是状态混合树，这里面的参数需要我们用脚本来实现触发，开始编写角色的动画脚本：

## 任务14 角色位置和动画同步



## 任务15 底牌“PunRPC”



如何调用这个函数呢？



使用上面的图片里的函数，第一个参数：调用的函数名、要发送的对象（就是想让谁知道）、函数的参数

## 任务16 枪械同步

枪械同步依然使用[PunRPC]的方式：

[PunRPC]

private void RemovePlayerPick(string name)

{

switch (name)

{

case "AK47":

{

gunRemove[2].SetActive(true);

gunRemove[0].SetActive(false);

gunRemove[1].SetActive(false);

gunRemove[3].SetActive(false);

gunRemove[4].SetActive(false);

}

break;

case "M4A1":

{

gunRemove[1].SetActive(true);

gunRemove[0].SetActive(false);

gunRemove[2].SetActive(false);

gunRemove[3].SetActive(false);

gunRemove[4].SetActive(false);

}

break;

case "Barrett":

{

gunRemove[0].SetActive(true);

gunRemove[1].SetActive(false);

gunRemove[2].SetActive(false);

gunRemove[3].SetActive(false);

gunRemove[4].SetActive(false);

}

break;

case "SMG":

{

gunRemove[3].SetActive(true);

gunRemove[0].SetActive(false);

gunRemove[1].SetActive(false);

gunRemove[2].SetActive(false);

gunRemove[4].SetActive(false);

}

break;

case "Shotgun":

{

gunRemove[4].SetActive(true);

gunRemove[0].SetActive(false);

gunRemove[1].SetActive(false);

gunRemove[2].SetActive(false);

gunRemove[3].SetActive(false);

}

break;

}

}

## 任务17 生命值同步

/// <summary>

/// 自身受到伤害

/// </summary>

[PunRPC]

public void DamageGet(int dmg, Vector3 hitPoint)

{

bloodVolume -= dmg;

this.GetComponent<UIManager>().healthSlider.value = bloodVolume;

this.GetComponent<UIManager>().damagePanel.SetActive(true);

Instantiate(blood, hitPoint, Quaternion.identity);

StartCoroutine("hideDamage");

if (this.GetComponent<UIManager>().healthSlider.value <= 0)

{

pv.RPC("Died", RpcTarget.AllBuffered);

pv.RPC("DiedAnimator", RpcTarget.AllBuffered);

}

}

[PunRPC]

public void AddHealth(int amt)

{

bloodVolume += amt;

this.GetComponent<UIManager>().healthSlider.value = bloodVolume;

}

# 实训项目3 荒野逃生一级项目