Unity 2D 像素风格视频录制大纲

1. 导论

本课程是使用 unity 来进行独立游戏开发的一个基础入门的教程。游戏是电脑单机的一个 Topdown（俯视角）的 RPG 类的像素风格的 2D 冒险游戏，我们的主角名叫 DuDu,所以游戏名：DuDu Go!适合的人群是那些正在学习 unity 基础的那些小伙伴们，课程中有很多内容都是我自己的理解，也不免有些错误，大家可以一起讨论下。

本课程主要有 3 个部分：

第一部分会介绍如何使用 asprite 像素画的软件来绘制游戏的美术资源，这部分仅作参考；

第二部分我会使用 C#的来写一个简单的运行上有点类似 Unity 的控制台的小程序，就是也有 start().update()函数以及组件的概念，然后让程序跑起来，将 C#的基本语法都贯穿在其中，比如面向编程中类的使用，类的继承和多态，静态成员，还有 C#的委托，协程的原理，C#的反射等等的内容，希望大家对 C#的基础语法有个基本的全面的了解；

最后一大部分使用 unity 来开发一个 2D 的独立游戏，也只是做了一个比较简单的游戏原型，以讲解 Unity 的相关知识点为主，涉及到的是 unity 游戏开发中比较基础的内容：比如使用精灵片和 tileMap 工具搭建我们的游戏场景；通过对输入模块和动画模块的学习来掌握如何控制我们的主角，以及基本的动画切换；制作场景中的道具，来了解主角如何与场景中的其他物体进行交互，并学习到 unity 提供的 scriptable object 这个数据容器；通过 unity 的 UI 系统显示主角的能量和生命值，以及搭建一个小型的背包系统，来了解游戏 UI 元素的用法；之后创建我们的 NPC，就是敌人角色与我们的主角进行交互，敌人有漫游场景和攻击角色的功能，以及在场景中的某些地点按一定频率进行生成，我们的角色通过抛掷投掷物来反击敌人，这其中我们会创建一个投掷物的 object pool;还写了一个管理场景的单例的 GameManager 类，里面写了我们游戏胜利和游戏失败的触发显示；之后加上各类音效使得我们的游戏更加生动，最后做了简单的 MainMenu 菜单使我们的游戏更加完整。

Unity 版本：2018.4.18 ； VS 版本：2017

参考教程：
《Developing 2D Games with Unity》
《Unity 中 2D 动作游戏开发艺术训练视频教程》
C#:刘铁猛老师的 C#基础编程入门
《C#图解教程》（C# 5.0）

2. 像素风格素材制作

软件使用：专门用于像素画（Aseprite,piskel）,通用图像软件 photoShop；有手绘板会更方便；
像素画参考教程：https://www.bilibili.com/video/BV144411v7iS?share_source=copy_web

场景的绘制目标：一个简单的类似公园，花园一样的小场景像素风格，感觉美术要求不是太

高，PS 中场景草图的绘制。

参考的图片收集，收集动画类型的素材（这边是主要参考了宫崎骏的一些动画片，比如崖上的波妞，龙猫等），不要使用真实照片，将动画素材进行下采样可以直接下采样加以利用或者进行参考。

aseprite 源码地址：https://github.com/aseprite/aseprite
Piskel:https://www.piskelapp.com/
本教程使用 Aseprite：

操作时要切换到英文输入法；
画笔时上勾上 pixel perfect
常用快捷键：铅笔工具(pencil tool)【B】，画线工具(line tool)【L】， 矩形形状工具（rectangle tool）【U/shift+U 椭圆】，轮廓工具(contour tool)【D】，吸管工具(EyeDropper tool)【I 或者按住 ALT】,油漆桶工具(paint bucket tool)【G】，移动(move tool)【按住 CTRL】，添加图层(new layer)【shift+N】,添加帧(new frame)【Alt+N】
选取工具(Rectangular Marquee Tool)【M】,选取后镜像操作（shift + h）,取消选择 ctrl +D,按住 shift 再框选的话就加选，鼠标右键减选。
鼠标左键前景色，鼠标右键背景色

人物设定为 32x32 像素，其他场景元素依据人物尺寸做参考。
(1) 背景地形元素的制作（土地，草地，水面），使用 tileMap 的方式绘制：
Tile:瓦片，本质上就是无缝贴图，对应于 unity 的 tileMap 的功能；View ->TileMode,；制作类似九宫格的瓦片单元,这里绘制 3 类一共六个九宫格，方便进行地形形状块多样化的绘制；单独的 tile 绘制； tile 大小 16X16; aseprite 背景参考网格的大小的调整：Edit->Prefererences->Background->Size->16*16
土地与草地（外围土地），草地与土地（外围草地）：交界处草的部分要画的暗一点；
土地与水面（外围土地），水面与土地（外围水面）：交界处土地外面有一层白色
草地与水面（外围草地），水面与草地（外围水面）：交界处草的外围有一层白色

(2) 植被和道具的制作：树木，花，金币，能量瓶，血瓶等道具；以及建筑物的制作：房子，围栏等；
导入参考图：直接将图片拖进去就可以了。
参考图中相关元素进行裁剪，之后再进行下采样（图片缩放），放入一个图层中进行参考，然后新建一个图层进行绘制。改变画布（canvas）大小:Sprite->Canvas Size,可用于裁剪画布；改变图片大小：Sprite->Sprite Size
绘制时需要绘制出亮部，中间调，暗部即可。
增加阴影

(3) 相关 UI 元素制作:背包,血条，按钮等等

(4)主角参考：崖上的波妞，角色名：DuDu；分层(layer)绘制:头部，衣服，手部和脚。将参考图的相关部位进行截取下采样。需要绘制 DuDu 正面，侧面和背面。

(4) DuDu 的动画序列制作：人物角色拥有四个方向(东南西北)的 Idle（东南西北的 idle 都是两帧），正面的 idle 动画：身体头部上下摆动；侧面的 idle:也是身体头部上下摆动
walk（东和西（侧面方向）4 帧，南和北（正面和反面）2 帧
正面的 walk 动画，脚步一高一低，手部左右轻微摇摆，摇摆需要手绘，这里不能用选区+旋转的方法；
侧面的 walk 动画：腿部向前旋转一点，可能还是需要使用旋转工具试一下，旋转的轴心点可以放在左上角，胳膊也是前后摆动。
attack（东和西（侧面方向）3 或 4 帧，南和北（正面和反面）2 帧）攻击（只设计静止时的 attack）；
正面的 attack 进攻：将手部举起来和手部放下，这来个动作，中间可以插一帧挥动时的那种气流（拖尾）；
侧面的 attack 进攻:手部举起放下，有那种气流；

选中当前帧，Alt + N，生成下一帧，生成的下一帧复制了上一帧的内容。New Tag 命令可以为不同的动作序列打上标签，相当于分组的作用。
动画序列输出：File->Export Sprite Sheet 可以输出成一整张序列表
敌人角色：参考《鱿鱼游戏》小兵角色的 Q 版图；一个朝向南面的 Idle 和 walk,都是两帧;

素材文件导入到 photoshop 中进行排版，可能会调整部分元素的比例大小，各美术元素组合成大图，各美术资源排布尽量紧凑但不能出现边界框交叉即可，导出成 png 格式，导出成 3 张大图，地形 tileSet,场景元素（房子，树木等），人物角色（主角和敌人）这三张大图。
之所以要拼接成大图的原因：
1. 方便开发者分享和使用
2. 载入到内存和显存的时候，系统直接分配一块连续的内存，优化存储和读取的操作
3. 图形学渲染时，减少 drawCall（可以认为是模型渲染到屏幕上的过程）命令的调用。一张图-->一个 texture--> 一个 shader-->一个 drawCall,drawCall 调用频繁的话影响渲染效率，降低帧率(FPS:Frames per second)，unity 对 sprite 的切割内部应该最后优化成纹理 UV 坐标的偏移。

3. C#基础入门概览

参考教程：
刘铁猛老师的 C#基础入门教程,bilibili
《C#图解教程》（C# 5.0）

课程内容是用 C#程序语言编写一个运行上有点类似 unity 的控制台小程序，比如具有 start()和 update()成员函数的组件编写，GameObject 的编写，协程函数的编写等，希望能将 C#的大部分语法都贯穿其中，与 unity 的程序有不同，所以这个小程序仅作参考。

C#编程准备：
.NET framework ,.NET core,Visual Studio,C#：（平台，环境，系统，框架？？？）
.NET framework: .NET 框架，微软提供了在 windows 系统上进行程序开发和程序执行的

底层服务，统一了之前的桌面程序，web 程序甚至移动端程序的开发平台，方便开发者进行程序开发。.NET framework 主要包含两个部分：

1. CLR(Common Language Runtime)公共语言运行时，运行时管理程序的执行，包括内存管理和垃圾回收，线程管理和异常处理，代码的类型安全检查等；
2. FCL(Framework class Library,FCL,有些地方称为 BCL,Base Class Library,基类库)框架类库，一组 DLL（Dynamic Link Library 动态库）程序集的集合，包括通用基础类（文件、字符串操作等），集合类，线程类等；

C#（Csharp）是微软专门为.Net 配备的一门新的编程语言，是一种面向对象，运行在.NET framework 上的一种编程语言,它基本类似于 Java,也吸收了 C/C++语言的特点。

C#的编译：与 C/C++有不同，先将源代码编译成 CIL（Common Intermediate language，公共中间语言），然后 CLR（运行时）编译成本机代码，给 CPU 去执行，并且运行时受到 CLR 管理，这就有了托管代码(managed code)的概念,反之，不在 CLR 控制之下运动的代码，称为非托管代码（unmanaged code）。C++的话，编译器都是直接编译成本机代码了。

需要安装的工具：

Visual Studio: IDE（integrated development environment 集成开发环境），是一个可视化的编程工具，方便+强大。安装时勾上桌面开发，对于 unity 游戏开发，务必勾上使用 Unity 的游戏开发。

.Net Core:.NET framework 只能在 windows 系统下使用，.NetCore 是微软推出的新的跨平台框架，在 mac,linux 操作系统下都能使用，且是开源的。

Mono: .Net Core 和.NET framework 都是微软推出的，Mono 则是非官方（第三方）推出的一个跨平台的.Net FrameWork 的实现，Unity 就是用的这个 mono 的框架来编译执行。

Visual Studio 创建 C#控制台程序，打开 Visual Studio,菜单栏：文件->项目->Windows 桌面，控制台应用（.Net Framework）,起名为 miniYou,控制台程序的入口点 Main 方法（函数）；

using+命名空间,声明命名空间 namespace,命名空间可视为一个字符串（字符串中可以使用点，此外如果是多个命名空间的嵌套也是用”.”），是相关的类的范围限定，其作用就是组织和管理类，防止不同类库中，有相同类型名称而引起的冲突；

类库 DLL(Dynamic Link Library)的引用；如果是编译成类库的话，选择类库的模板，没有 Main 函数；

第三方类库的引用:解决方案管理器中，右键“引用”，选择“添加引用”，打开引用管理器，选择“浏览”，点击右下角的“浏览”按钮就可以引入第三方的 dll.，也可以在这个引用管理器的程序集中查看.Net 提供给我们的那些类库。在 C#中编译出来的 dll 和 exe 都称为程序集(Assembly)

引入 dll 的准则：不是越多越好，用哪个引入哪个；

书写 helloWorld：Console.WriteLine(“Hello World”);

C#面向过程的编程方式介绍变量和函数：
语句或者语句块注释：// 或者 /* */；///主要用于文档注释
系统自带的基本的数据类型，系统自带预定义的数据类型和自定义的数据类型：
基本的数据 int,uint(无符号整型),char,float（单精度浮点数）, double,bool(true ,false)等，系统自带预定义的数据类型有 string,object 等，
C#强类型语言，变量的作用：存储数据；声明变量时需要类型名+变量名(例如 int healthAmount)，变量名首字母一般都小写,两个单词组合第二个单词大写，避免与关键字冲突。类型名代表着分配的内存的大小，计算机的最小存储单位是字节（byte），每个字节有 8 个比特（bit，二进制 1 和 0）,int 4 字节,float 4 字节（符号位，指数位，底数位），double 8 字节; 可以用 sizeof(类型名)进行查看；声明基本类型+赋值（int healthAmount=100,这里=等号表示赋值）相当于在内存中开辟了一个 4 个字节的空间，在里面存了 100 的整数，变量名隐含着这个内存的地址信息，让我们能获取这个变量内保存的数值。 string 字符串类型，系统提供给我们的，让我们能像基本类型那样使用。变量名的类型前加上 const 表示该变量只读，不能修改。
格式字符串输出：Console.WriteLine(“ {0},{1}”,变量 1，变量 2)

基本运算符：+ - * / < > == && || ！;流程控制语句：判断语句（if else;switch），循环语句(for ,while，foreach)；跳转语句 continue,break,return

函数是一些实现某个算法的代码块的封装使用，方便代码在其他地方调用；函数声明： 返回值 函数名（函数形参），参数列表称为函数形参，函数名一般首字母大写；调用函数时，函数的参数列表填入的参数称为实参，实参的值是复制给形参的。局部变量，变量作用域（可以认为是{}的包含范围）；
函数形参前加上 ref 关键字，表示传入值参数的引用，可以改变传入进来的实参的值，本质是传入的实参的地址；调用时实参前也需要加上 ref 关键字。相当于 C/C++的指针。
加上 out 关键字，称为输出参数，与加 ref 的效果一样，但是必须在函数代码中对其赋值；调用时实参前也需要加上 out 关键字；
加上 params，参数类型前加上 [ ] 表示数组,比如 params int[] args 表示某某类型的参数数组,可以传入一个该类型的数组或者直接写上任意数量的该类型的实参变量。只能有一个 params，且只能放在最后一个参数列表上。调用中不需要加修饰符。数组可以认为是相同类型的数的集合，会用序号去获取每个数组元素。

函数重载：函数名相同，参数不同（参数类型或者参数个数）构成函数重载，跟返回值没有关系。将函数光标放置于函数的()内，Shift + ctrl+space 查看重载函数。
举例说明：受到攻击减血，吃到血瓶加血
Int healthAmount=100；
int damageAmount=5；
int bloodAmount = 10;
int AddhealthAmount(int amount,int blood);
int SubhealthAmount(int amount,int damage);
void SubhealthAmount(int amount,int damage);(不是重载)
int SubhealthAmount(int amount,int damage,string sentence)

自定义类型：类 class,结构 struct,数组 int[]等,枚举 enum,委托 delegate,接口 interface
自定义类型的数据：分为定义和声明（创建）两部分。

C#是面向对象编程的一种程序语言（OOP:object-oriented programming）,面向对象编程是对现实物体进行计算机语言的抽象，一切 C#的数据类型都有一个万物基类：object.
类 class 的定义（相当于定义了一种新的数据类型）：

内存四区：栈，堆，代码区，全局静态区。

值类型和引用类型：值类型只需要一段内存即可，这段内存直接存储变量的值；引用类型有两段内存，一段存储变量的值，一段存储变量的地址（即引用）；在值类型在内存栈区，引用类型数据是存储在堆区，在栈区存了一个数据的地址即引用。C/C++堆内的内存都要自己释放（内存泄漏），C#有垃圾回收机制（CLR 管理）会自动释放堆内的没有引用的内存。
C#中的值类型：基本的数据类型（int float bool 等），struct,enum
C#中的引用类型：object, string ,class, interface,delegate,array

class 类名
{成员变量；构造函数；成员方法}
成员变量和成员方法的修饰符（访问修饰符）： public（公有的，类内类外（继承类，类对象）都能访问） ,private（私有的，默认的访问级别，类内才能访问）,protected（保护的），internal（内部的），protected internal（受保护内部的）；internal 感觉用的较少，表示只有在同一个程序集中可以访问。

成员变量：类的一些变量特性，C#中声明时可以直接初始化，也可以在构造函数中初始化。
构造函数：又称为构造器，名字与类名一样，没有返回值，默认程序提供了一个无参的构造函数，可以自己写有参构造函数，可以有多个构造函数（重载）。在类的实例化时调用，初始化对象的作用，在内存中为对应的实例对象分配空间。
C#有垃圾回收机制，可以不写析构函数。
C#成员变量分字段（Fields）和属性(Properties)，字段就是成员变量私有化，作用：保存类的数据；
属性：读取和设置字段的值，是一个成员方法，相当于字段的 get 和 set 函数（访问器），一般写法：类型修饰符 public + 变量类型 相同的字段变量名（但首字母大写），函数体中一般写{get{return} set{ = value}}；value 是系统提供给 set 访问器的一个隐式值参，是个关键字，用来接收外部传来的数据。在 Unity 的组件脚本的写法中似乎不常用属性，而是直接将字段进行 public，这是因为需要将字段展现在参数面板上供开发人员调试。

成员方法：类的行为，与普通函数方法差不多，函数前需要加上 public 或者 private 的修饰符来区别该方法是只能类内调用还是类内外都能调用。方法内能访问类的成员变量，方法内也可以声明专门自己时候的局部变量，函数形参就是局部变量

类的实例化，又叫创建类的对象，会调用类的构造函数， 类名 xxx = new 类名（构造函数参数）。类的实例对象通过“.”点运算符来调用类的 public 的成员属性和成员方法。

代码内容：模仿 Unity 程序的初始化(在游戏中对应加载各种数据等)以及每一帧的循环（游戏中对应渲染每一帧画面，检测用户输入等）的框架
定义一个 Class Game，
成员变量 bool m_isRunning= true(演示字段和属性)，来控制循环帧的结束。
构造函数。
三个 public 函数，
bool Initialize(),游戏的初始化
void RunLoop(),游戏循环每一帧
void ShutDown(),游戏退出，
private 函数：
StartGame()，Initialize()内调用,
processInput()和 UpdateGame()， RunLoop()的 while 循环内调用，
processInput()作用是键盘输入触发，当按了 Q 键后退出循环，具体代码如下

```
While(Console.KeyAvailable)  //while 的()填入表达式，可以是 bool 变量，计算表达式等
{
ConsoleKey ck = Console.ReadKey(true).Key;
If(ck==ConsoleKey.Q)  //表示条件选择
{
M_isRunninig = false;
}
}
```

在 Main 函数中，“.”

```
Game  game = new Game();
Bool success= game.Initialize();
If(success)
{
Game.RunLoop();
}
Game.ShutDown();
```

类的特性：封装，继承和多态
封装：我认为有两层含义：1 是相对于面向过程编程来说，把变量和函数都封装进了一个类中；2 在类中，将一些变量和一些方法进行私有化进行的封装

类的继承：C#所有类都派生自 object;作用：在原有类的功能上增加新的功能，代码复用；
自定义继承的写法：class 派生类：基类，冒号后面是基类的名字，或者称为子类：父类；
派生类继承基类的全部成员变量和成员方法，但是基类私有 private 的成员不能在子类中使用,protected 和 public 都可以使用；C#只能单继承,但可以有继承链，要多继承的话在 C#中是通过继承接口 interface 实现的；
继承的使用：在编程中可以根据现实情况来规划哪些类有继承关系，比如写个人物类 Character,然后让主角 Player 和敌人 Enemy 都继承 Character;或者有些类有很多公共部分，把

相同部分都提取出来作为一个基类；

若派生类这边调用的是无参构造，那么执行过程是先执行基类的无参构造函数，再执行派生类的无参构造函数（这里是隐式调用，相当于构造函数的合并吧）。

现在有两个类，会出现一种情况：若派生类中有成员（通常就特指成员函数）与基类中的成员函数签名一致，这里的签名指的是函数的名字和参数的类型和个数都一致（不包括返回值），这会发生什么情况呢？还有基类引用引向派生类对象。

1. 派生类隐藏基类方法：然后在变量和函数签名前再加上 new 关键字，不写 new 的话也是这种情况，它会有些警告，让你明确你的编程目的。那么子类要调用的话，就调用自己的变量或者成员函数。子类要调用父类中的相同成员的话，需要用 base.的方式，base 就表示基类或者父类。类型转换：在变量名加上（）然后填上要转换成的对象类型；这时基类对象引用派生类对象的话，再调用这个同名的隐藏方法的话，就调用的是基类的方法
2. 派生类重写基类方法：基类的相同签名相同方法前加上 virtual 的关键字，那么派生类相同的要写上 override 的关键字（必须写），此时的现象是：基类对象引用派生类对象的话，基类引用变量调用这个重写函数的话，调用到的是派生类的函数。这么一种用法就叫做类的多态。学术化的定义：相同类型的对象调用相同的方法表现出不同行为的现象。

代码部分：
Unity 的游戏对象（GameObject）是通过挂上组件的方式来增加不同的功能，以便完成相关的游戏逻辑，GameObject 中组件都是继承自 MonoBehaviour 的。
代码 Component 类，Component 类构造函数,public start()和 public update()，派生出子类 SphereComponent 类和 TransformComponent 类，,两个子类各自写上构造函数（unity 的继承 MonoBehaviour 的组件不要写构造函数），两个子类在 Game 类的 StartGame()中分别进行实例化查看执行效果，然后给基类的 start 和 update 函数加上 virtual ,派生类重写这两个函数，加上关键字 override，运行代码测试，查看运行结果。Unity 中的派生类的组件中，start 和 update 函数没有 override 的关键字，它应该用其他方法实现了这种多态的行为，具体这块我不太清楚。

代码部分新建一个 GameObject 类，模拟 unity 的 GameObject（游戏对象），
涉及到的知识点：
List:列表，一种集合的类型；类似于数组，但增加了很多操作数组的功能；数组代表了相同类型元素的集合,它也是一种引用类型,声明时通常时类型名加上”[]”,例如 int[] myArray = new int[100]，初始化时需要明确指出数据的个数，进行在内存中会分配一个固定大小内存来存储这些数据，获取或者设置数组元素的方式：数据变量名[index],index 从 0 开始。List<>是系统类库提供的一个泛型的集合类：列表，它的本质也是个数组，但是这个类给我们提供了很多方便的数据元素的操作方法，比如增加一个元素 add,删除一个元素，remove,还有当超出内存的存储范围时，它会自动增加内存大小来存储新的数，而单纯的数组是没有这类特性的，。

泛型（generic）：使变量的类型的参数化，在定义类的时候，对于某些成员变量的类型用通用的类型符号<T>表示，在实例化具体对象的时候再用具体的数据类型代替，也是实现代码

重用的一种方式；where 关键字来约束泛型的类型

for/foreach 循环的用法：for 循环有种遍历的含义：它的用法通常如下：i:循环变量,i=0 循环变量初始化，只执行一次，i<100 循环条件，满足这个条件，执行 {}代码；循环变量 i 自增，i++ =>i = i+1，这条语句在每次执行{}代码后调用;

```
For(int i=0;i<100;i++)
{
}
```

For 循环转化 while 循环的写法：

```
Int i=0
While(i<100)
{
i++
}
```

foreach 语句通常用于遍历数组或对象集合中的元素。写法 foreach(var comp in comps){},隐含了一个迭代器。

关键字 var 让系统自动去推荐变量类型，应该会消耗性能。

This 关键字：在类中使用，是对当前的所实例化后的对象的引用。

代码部分：GameObject 类中加上一个组件列表（泛型的集合列表），表示这个游戏对象的有多少组件。

Private List<Component> comps = new List<Component>()集合类的泛型

GameObject 类中的成员方法 public 的，主要是添加组件，销毁组件和返回组件的函数

Public void AddComponent(Comp) ->comps.Add()

Public void RemoveComponet(comp)->comps.remove()

Public Comp GetComponent(string) ->return comp;

GameObject 类再添加相关的 start 和 update 的函数，主要是遍历它的组件列表来调用组件的 start 函数和 update 函数：

Public void Start()->foreach{ comp.Start()}

Public void Update()->foreach(comp.update())

Component 组件的构造函数中组件中依赖注入 GameObject，就是增加一个 GameObject 变量的引用，在 C#中类的变量就是引用，所以参数前不需要加 ref 关键字了；Component 的成员变量中可以增加一个 protected 的成员变量：GameObject m_owner;之所以需要引入这个参数，是因为一个 GameObject 间的组件之间是有数据传递的，这个参数表示这个组件的 owner（拥有者）;此时 Component 的构造函数中就需要修改，变成有参构造函数,并且加上 gameObject.AddComponent(this),这句话表示将这个组件加入到 GameObject 组件列表中去。Component 子类的写法也变为：SphereComponent(GameObject owner):base(owner)

Game 类中也声明一个 List<GameObject> gobjects= new List<GameObject>() 和对应的 AddGameObject（GameObject）和 RemoveGameObject(GameObject)函数，来模拟游戏场景中有很多个游戏对象；然后在初始化函数 StartGame 函数中实例化两个 GameObject,然后对每个 GameObject，各自实例化两个 SphereComponent 和 TransformComponent,具体代码如下：

都是局部变量：

```
GameObject go1 = new GameObject();
GameObject go2= new GameObject();
AddGameObject(go1);
AddGameObject(go2);
TranformComponent tc1 = new TransformComponent(go1);
SphereComponent sc1= new TransformComponent(go1)
TranformComponent tc2 = new TransformComponent(go2);
SphereComponent sc2= new TransformComponent(go2)
```

这里虽然都是局部变量，但是他们的引用都已经赋值到 list<>的集合列表中去了。
此外也别忘了在 Game 的 StartGame 函数中增加对 gameobject 的 Start 函数的调用

```
Foreach(var go in goObjects)
{
Go.Start();
}
```

在 Game 的 UpdateGame 函数中增加对 gameobject 的 Update 函数的调用

```
Foreach(var go in goObjects)
{
Go.Update();
}
```

类的修饰符：
类的访问修饰符：public ,internal，前者各个程序集间可访问，后者只能在同一个程序集间访问
其他修饰符：abstract:抽象类：不能被实例化，只能作为基类被其他类继承；抽象类中含有抽象方法（abstract 修饰的函数成员），派生类（不是抽象类）继承后需要实现这个抽象方法（需要带上 override 关键字）
Sealed:密封类，不能被继承的类
Static:静态类，类中所有的成员（包括数据和函数）都要静态，都要加上 static 关键字；Console 类就是一个静态类；无法实例化和被继承

一般的类中，在变量或者函数前也可加上 static 的关键字，变成静态变量和静态方法；相对于静态，其他普通的变量和函数就称为实例方法：

静态变量存放在内存的静态区，类内其他函数可以调用这个静态变量，类外（与这个类有没有实例化就无关了）调用 public 属性的静态变量：类名.静态变量名
静态方法还是放在代码区，是一组操作指令，调用 public 成员方法方式：类名.静态方法。
注意：静态方法不能调用普通方法和普通变量，只能调用静态变量和静态方法，这是为什么一开始在执行 Main 函数中调用的函数前需要加上 static.

当程序开始加载起来（可能是还没执行 Main 函数的语句之前），静态变量就在静态区域中分配了内存了，而且这个变量可以认为是具有全局性；静态函数也是在类实例化之前就可以用（类名.静态方法）调用了，那么此时那些类实例中的普通变量还没有生成，普通函数实

际上隐藏了这一个实例对象的引用参数，即 this；那么就说没有实例化对象之前，这些普通的函数是没法调用的，所以静态函数中不能有普通变量和普通函数，要调用的化，就需要传入一个类的对象作为形参，或者在静态函数内实例化这个类。

自定义结构体 struct Name{},结构体与类的定义方式几乎差不多，可以有构造器，但是结构体是值类型，也不能派生，通常都是用作一些不同类型变量的数据集合。比如：

```
Struct PersonInfo
{
String name;
Int age;
Bool gender;
Float healthAmount;
}
```

代码实例，写一个 MathTools 的类，写一个两个二维矢量相加的静态方法，让其能在 TransformComponent 调用，显示函数调用的过程。
创建结构体，需要与类同级，不能放在类中创建

```
Struct Vector2
{
Public Float x;
Public Float y;
Public Vector2(float x,float y)
{
This.x = x;
This.y = y;
}
}
```

静态函数

```
static public Vector2 AddVector2(Vector2 v1,Vector2 v2)
{
Return Vector2(v1.x+v2.x,v1.y+v2.y);
}
```

委托的使用：
delegate 函数回调，C/C++函数指针，函数委托，实质上就是将函数作为一个变量;本来函数名就是函数的地址。委托是一个引用函数的变量。委托是一个变量，使用方法可以类比与普通的变量的使用。有了委托之后，函数就可以像变量一样赋值，也可以将函数作为参数变量传给其他函数方法，提高了方法的扩展性。C#的事件（event）就是基于委托的.

Delegate 也是一个自定义类型，跟 class 同一个级别，所以要定义委托类型也要写在类外；用 delegate 关键字自定义一个 delegate 的类型： delegate void DelTrigger（int val），需要明确指出函数的返回类型和函数的形参变量。DelTrigger 就是委托类型的名字。
声明委托的方法和初始化：

自定义委托类型 变量名 = new 自定义委托类型(相同类型函数的名称)
委托可以挂上多个函数，可以用+=添加，-=减掉

Unity 中是将碰撞触发的相关代码写在它给定的 OnEnterTrigger()函数中，这边我不知道它的具体实现。这里模拟 UE4 中 sphereComponent 组件发生 trigger 或者 collision 时，会挂上相应的委托函数。
代码部分：
自定义一个 delegate void DelTrigger(int val)的托管类型
SphereComponent 成员变量中声明一个这个类型的委托变量： public DelTrigger OnBeginOverLap 的委托变量，并将这个委托变量放在这个 Update()函数中，实际中其实这个调用是封装隐藏的，不暴露给我们的，可以加个 count 变量，让其 count++累加，让其 2000 次调一次。这个委托变量在 SphereComponent 的函数进行初始化，先可以挂上一个空的函数；
在 GameObject 的类中定义一个 OnEnterTrigger 函数，在这个类的 start 函数，调用 GetComponent()的函数，让其遍历整个组件列表，然后 Comp.ToString()==”那个组件的名（命名空间.组件类型名）”，返回那个 SphereComponent 的组件引用，之前需要声明一个 SphereComponent sc 的变量 ，然后将其挂上即可 sc.OnBeginOverLap += OnEnterTrigger.

Unity 协程 Coroutine 的底层原理：https://www.cnblogs.com/yespi/p/9847533.html
会涉及到枚举器和迭代器，接口（StartCoroutine,IEnumerator）
枚举器和迭代器的学习中会涉及到接口：IEnumerable 和 IEnumerator

接口：C#类是单继承，用接口可以实现多继承；接口是从抽象类（abstract）进化过来的，接口的关键字 interface，按照编程习惯，自定义的接口类型的名称通常是 I 开头 ；
接口定义中只能包含非静态的成员函数，且不能有代码实现，函数前不能写访问修饰符，它是隐式 public 的。感觉继承接口后，将接口中函数的代码实现转移到了派生类当中。
接口使用方法：在基类列表中包含接口名称，就像继承一个基类一样，冒号后面写上接口名，然后派生类必须实现接口的所有成员方法。这边的术语一般都不叫“继承某某接口”，就叫“实现某某接口”
接口的作用：是一组行为的规范；解耦，类之间关系松耦合，代码扩展，(个体认为)使用起来有点像类的多态，也有点像委托；

接口比较常用的方法：比如某个函数调用某个类中的方法：Class A{ void fun()}
Void TestFun(A a),a.fun() 那么这个时候只能调用 A 这个类了，如果 Class B{void fun()},Class C{void fun()}...的类似功能的方法，那么这个时候就可以用上接口，就可以 iterface IfunInterface{void fun},然后 class A:IfunInterface;class B:IfunInterface;class C:IfunInterface;然后函数可以变为 void TestFun(IfunInterface ifun);ifun.fun();这样的话就相关与 IfunInterface 基类引用各自的子类，然后调用一致的一个函数 fun(),接口没有实现，只能用各自子类的实现，这有点像多态的功能，但不是传统的 virtual 和 override，并且基类中也没有实现；这也像一个委托的行为，先用接口在某个函数中方式一个函数，等到实际调用的时候再将具体的函数实现放进去，但这里没有涉及到将函数作为一个变量来传递。此外这样也将传入的函数的参数没有关联到一个具体的类，实现了与具体类的解耦，实现了松耦合。

枚举类型,枚举器（enumerator），可枚举类型(enumerable)，迭代器:
枚举(一一列举，遍历)类型 enum，自定义类型，值引用，通常是用作对状态的列举，比如角色的状态，有时候跟 switch 选择语句一起使用挺方便：
比如：

```
Enum MainPlayerState
{
MPS_Idle,
MPS_Walk,
MPS_Attack,
MPS_Dead
}
```

声明并初始化：MainPlayerState mps = MainPlayerState.MPS_Idle;

```
Switch (mps)
{
Case MainPlayerState.MPS_Idle:
Break;
Case MainPlayerState.MPS_Walk:
Break;
Case MainPlayerState.MPS_Attack:
Break;
Case MainPlayerState.MPS_Dead:
Break;
Default:
Break;
}
```

枚举器，实现了 IEnumerator 的类叫做枚举器，
可枚举类型，实现了 IEnumerable 的类叫做可枚举类型的类
迭代器：有些地方将枚举器就称为迭代器；有些地方：因为引入了 yield return 来简化枚举器原来的代码，然后将相关的部分称为了迭代器

Foreach 语句能对一个数据集合类进行遍历，就需要这个类实现 IEnumerable 的接口，这个接口是让其返回一个 IEnumertator 类型即可
代码举例：

```
Using System.Collections
Class Erable:IEnumerable  这个接口需要实现 GetEnumerator()这个函数
{
Public IEnumerator GetEnumerator()
{
return
}
Return new
}
Class Erator:IEnumerator 这个接口需要实现 Current,MoveNext 和 Reset()这个三个函数
{
```

```
    Private int index=-1
    Public object Current
{
Get{return index;}
}
MoveNext(){   if(index<5)return true else false}
Reset(){ index = -1}
}
```

Foreach 接受的是 Erable 实例，通过 GetEnumerator 来获取一个枚举器，然后调用枚举器的 MoveNext,然后 Current,然后 MoveNext,Current...一直到 MoveNext　　返回 false 遍历结束；可以在这些函数中填入相关的打印语句查看相关的流程。

它里面的枚举器的运行方法

```
Erator   e =new Erator()
While (e.moveNext)
{
   Console.Write(e.Current);
}
```

后来为了简化代码，引入了 yield return,将 Class Erator:IEnumerator 可以进行简化

简化成一个成员函数 Erable 中的成员函数

```
IEnumerator TestErator()
{
  For(int index =0;index<5;index++)
{
Yield return index;
}
}
```

然后 GetEnumerator()函数 return TestErator.碰到 yield return 程序先返回，迭代器移动到下一个，然后再从上次 yield return 的语句执行下面的语句，可以把 For 循环拆开来解释。

unity 协程：

协同程序，unity 中通过 StartCoroutine()开启一个协程,不是开启一个新的线程，其运行也是在主线程上，可查看 unity 的执行顺序：https://docs.unity3d.com/Manual/ExecutionOrder.html 实质就是每个 MoveNext()运行在 Update（）函数的一个帧，执行到 yield return，协程的调用返回，协程函数挂起，下一次从上一次返回的地方进入，一直到所有的 yield return 都执行完，或者碰到 yield break,在 unity 中是调用 StopCoroutine（）函数就停止了；作用：延时调用，分帧调用等

代码实例：模型 unity 的协程的使用：

在 TransformComponent 中加上一个

```
  Private IEnumerator e = null;
```

定义一个协程函数：

```
IEnumerator TestCoroutine()
```

```
{
For(int index = 0;index <100000;index++)
{
Console.Write()
Yield return null;
}
}
```

设置 void StartCoroutine()在 start（）函数中调用

```
{
e = TestCoroutine();
}
```

Void RunCoroutine()在 update 函数中调用

```
{
If(!e)
{
If(!e.MoveNext())
{
E=null;
}
}
}
```

C#反射（Reflection）机制：

元数据(metadata)：有关程序及其类型的数据，它们保存在程序的程序集中。

反射:一个运行的程序查看本身的元数据或者其他程序的行为。

反射是.NetFramework 提供的功能 ，跟 CLR（公共语言运行时）相关，C#代码是先编译成 CLI 语言，然后在 CLR 进行托管运行的，可以在运行的时候加载程序集（dll 或者 exe）来获取相关的数据类型（class 等）信息，然后根据获取的信息可以动态的创建一个对象。一般用 new 类型（）这种方式都是在编译时帮你创建好了对象的。运行的时候，就不能用 new 的方式来创建对象了，只能靠反射的方式来创建对象了。

一般用到的地方比如：

在 Visual Studio 中我们按 F12 查看系统类的定义时，打开的文件都有[从元数据]的标识；在 unity 中写了一个脚本组件后，挂到游戏物体上，在 inspector(检视面板，或者称为参数面板)会将脚本中的 public 变量给显示出来，供开发者调试。

.....等等

要使用反射，需要 using System.Reflection

Type 类型的变量（反射中最重要的一个变量了），Type 类是保存其他类的类型信息的（元数据），元数据就包括这个类的数据成员和函数方法；

例如：Type t = typeof(Game)，t 变量保存了 Game 类型的信息，之后就可以从这个变量中获取类的构造函数 t.GetConstructor()以此来创建 Game 类的对象（实例化）;方法 t.GetMethods()/GetMethod();变量 t.GetMembers()/GetMember()...默认情况下，只能获得那些公共的成员变量 public；在 Unity 中在私有(private)变量前加上 [SerializeField]就可以通过反射获取私有变量了，这个语法在 C#叫特性，我们可以自己给我们的元数据增加特性，在

unity 中系统已经提供了很多特性了。
变量的可以用对应的 GetValue(),SetValue()这样的函数获取和设置，函数调用的话可以用 Invoke()函数来调用，第一参数需要填上类的对象，类的对象可以通过调用构造函数或者 Activator.CreateInstance()函数来创建。

反射提供了另外一种写代码的方式，这里将原来 Main 函数中的那么执行代码，全部用反射的方法重新写一遍：先获取程序集，然后从程序集中获取 Game 的构造函数实例化创建对象，然后获取其初始化函数，设置 bool 变量的值，running 函数和关闭函数，分别运行 inVoke 来运行。这里的运行时调用可以这么来理解:在程序运行的时候，我们可以 Console.RealLine 来获取相应的字符串，然后获取字符串的内容，填入到对应的函数上 GetMethod(“ ”)，那么就会运行这个函数，如果是 new 那一串代码的话，就事先已经安排好要运行怎么样的代码了。
具体代码如下：

```
Assembly asm = Assembly.Load(“MiniYou”);
可以先用 Type[] ts = asm.GetTypes();
Foreach (var st in types)
{
CW(st);查看这个数据集中有哪些数据类型可以调用；
}
Type t = asm.GetType(“MiniYou.Game”);
ConstructorInfo cInfo = t.GetConstructor(new Type[0]);//new Type[0]可以表示无输入参数
Object obj = cInfo.Invoke(null);创建实例对象
MethodInfo initialInfo = t.GetMethod(“Initialize”);
Object success = initialInfo.Invoke(obj,null);
Bool bSuccess= (bool)success;
PropertyInfo pInfo = t.GetProperty(“M_isRunning”);
pInfo .SetValue(obj,true);
If(bSuccess)
{
MethodInfo runLoopInfo = t.GetMethod(“RunLoop”);
runLoopInfo.Invoke(obj,null);
}
MethodInfo shutInfo = t.GetMethdo(“ShutDown”);
shutInfo.Invoke(obj,null);
```

4. Unity 游戏编写

参考教程：
《Developing 2D Games with Unity》一个 2D 的 topDown 的小游戏
《Unity 中 2D 动作游戏开发艺术训练视频教程》
https://www.rrcg.cn/thread-16756347-1-1.html

目标：
制作一个 TopDown 的 2D 游戏原型（prototype,不打磨游戏玩法上的细节，对 unity 的基本使用以及相关的技术点进行讲解），以此了解 unity 制作游戏的整个流程，熟练掌握 unity 这款游戏引擎的使用方法。用 2D 游戏去入门 unity 比较合理。

Unity 简介：

Unity 游戏引擎可以认为是一个游戏开发的软件平台或者工具，它集成了游戏开发所需要的各模块，例如渲染模块，物理仿真模块，音频处理，以及网络开发等，也给出了很多数学和算法上的各类程序库和工具插件。它给我们提供了一整套相对固定但又非常灵活的游戏开发流程。游戏开发过程中可以使用引擎提供的 C#语言编写游戏对象的运行逻辑，但是引擎不提供 3D 建模，3D 动画，纹理贴图等方面的功能，这些都属于游戏美术资源，通常都是通过专门的软件，然后导入到 Unity 中的。总而言之 unity 就是方便我们游戏开发。

unity 安装：可在其官网（网站需要注册）上下载 unity Hub 或直接下载 unity 对应的版本都可以。一般建议先下载 unity hub 然后在 hub 上安装,它具有管理 unity 上的游戏项目，unity 软件版本等功能的。演示 hub 上安装 unity ,installs->Add，让你选择安装的版本，选择 Unity LTS（Long term support）的版本。点击 Next，让其勾上选择发布的平台模块，然后安装即可。安装完之后，打开软件，或者在 Projects 项目中创建工程的话，会要求使用证书的认证来激活 unity，这个时候选择 personal,表示仅个人学习使用，是免费版本。

创建工程项目：unity Hub->Projects->New(它的下拉列表上可以选择 unity 的版本，这里选择 unity2018)，打开新窗口后选择 2D 模板，填上项目名 DuDuGo 和保存路径即可。

界面的布局排版（layout）可以调整，里面的窗口都是可以随意拖动的，界面会自动去吸附,可以将自定义的界面进行保存，在 window->Layouts->Save Layouts 或者软件右上角的标签按钮可以选择你要的布局。
可能你那边默认的皮肤是银白色的，我这边是黑灰的；对游戏项目相关的一些属性设置，和对游戏编辑器的一些设置；可以打开菜单栏的 Edit->Project settings 或者 Edit->Preferences,
更换 unity 皮肤颜色：Preferences->General->Editor Skin:Professional

界面的介绍：
菜单栏:上面的一些命令基本都可以在其他界面点击右键获得；工具栏（手型 icon：平移视图，后面几个是对场景元素的位移姿态的操作，位移，旋转，缩放，对 2D 元素的操作，复合操作，快捷键分别是 Q,W,E,R,T,Y）；Center <-> Pivot ,中心点（物体的 bounding box 的中心点）与轴心点（模型空间的坐标系原点）的切换，Global<->local：世界坐标系与自身坐标系的切换。注意，unity 是左手坐标系（大拇指 X 轴，食指 Y 轴，无名指 Z 轴）。2D 游戏缺 Z 轴；播放/暂停 /每一帧：控制进入游戏模式。

视图面板有场景视图（scene）和游戏视图（game）：
场景视图就可以认为是编辑（操作）视图，会显示参考网格，比如搭建游戏场景之类的，面板的 2D 标记可以 2D 和 3D 视图的转换；
三维视图的基本操作，旋转：Alt+鼠标左键；平移：点击鼠标滚轮或者选择工具栏上的手型

图标后，用鼠标左键拖动；缩放：鼠标滚轮或者鼠标右键，F 键：focus（聚焦）单个物体，围绕一个物体旋转的话就需要先 F 然后在视图旋转。

游戏视图：场景的渲染视图，显示在游戏摄像机下的经过渲染算法渲染（render）出来的画面，也就是游戏成品后的画面，同时进行人机交互和物理仿真。按下工具栏上的播放键就进入 GameMode，设置 playerMode 的颜色下，在渲染模式下修改游戏对象的属性参数，编辑器不保存。Preferences->Colors->General/Playmode tint 设置一下它的颜色，表示 Game 模式下软件界面的颜色，以此来提醒开发者是在哪个模式下进行测试；。

Hierachy:层级面板，以树形这种数据结构的方式列出了场景视图的游戏对象（gameObject）和 UI 元素,点击相关的物体，场景中的物体就会高亮，反之也一样。场景视图中的物体最后都会运行到游戏上去。这里的 SampleScene 是关卡文件，这边也称为场景，保存游戏每一关游戏对象。

Inspector:检视面板：可以认为是各个游戏元素的参数面板，例如选择一个场景中的 GameObject（在场景中创建一个对象，相当于实例化代码：GameObect xxx=new GameObject(), 这个对象里面可以挂很多组件 component）,就能在它上面看到组成这个物体的相关组件以及各组件暴露出来的参数。Unity 中定义游戏对象的方式，是纯组件的那种方式，系统提供了很多内置组件，我们也可以用 C#来自定义组件。UE4 除了组件，各个游戏对象之间还可以继承。
C#代码书写一般都用的 Visual Studio 这个 IDE 工具，可在进行设置：Preference->External Tools->External Script Editor 可以设置脚本语言的编辑器。

Project window:项目面板，游戏的 assets（数字资产，或者可称为游戏素材）都显示在这个面板下，包含模型，材质，动画，游戏特效，各种逻辑脚本等等，显示在这里的文件若不放进场景视图中去的话，仅仅是个素材，要在游戏中使用的话就要在游戏场景中生成，或者挂给场景中物体。这里需要注意的是，对于资源一定要进行分门别类，不然会很难查找（是人很难查找，不是计算机），默认情况下有个 Scenes,保存关卡文件,选择菜单栏 File->New Scene 新建一个关卡，然后在保存 ctrl + S,命名为 Level01,保存在 Scenes 中。一般的话，从其他人那里获取的 unity 项目，可以直接点击这个关卡文件来打开工程。

Console:控制台显示面板，会将游戏运行时，程序中错误的信息或者 Debug.Log 的那些信息打印在上面，供检查程序的语法错误以及算法的逻辑错误。

自动保存的功能？？？设置中没有看到有这个选项，需要自己写一些脚本代码，所以要多保存，多多 ctrl + S.

Unity 场景绘制/搭建：
按照场景草图绘制：绘制背景地形，在地形上摆放场景元素；
方便各类游戏资源的管理，在 project 面板中创建几个文件夹：Sprites->GameEnv 和 Charactor, 将环境瓦片地图 tileMap.png 和场景的物体 SpriteObject.png 这两个素材导入到 GameEnv 中，

将 SpriteAnim.png 导入到 Charactor 这个文件夹中。

通常情况下，2D 游戏中，这一张张的 2D 图片都称为 sprite(精灵片)；之所以这么称呼我觉得还是需要一些图形学的知识，在图形学中，都有模型，纹理的概念。那么精灵片就被可以认为就是一个平板的模型和它的纹理，模型和纹理是通过 UV 坐标联系的。

2D 精灵片切割：点击 tileMap.png 素材，在 inspector 参数面板（对于素材参数的调整也在这里）中设置相应的参数，
Texture Type:Sprite(2D and UI);
sprite Mode:Multiple;
Pixels Per Unit:16(每个单位包含图片的多少个像素;场景中的单位网格相当于 1*1 米的模型面片(物理上的 tile)，这个参数可以认为调整的面片与精灵 tile(图片的 tile)的映射关系，比如单位面片是 1*1，tile 像素是 16*16，若这个参数填 16，那么一个面片对应一个精灵 tile；若填 8，那么 4 个面片对应 1 个精灵 tile，若填 32，一个面片可以包含 4 个精灵 tile，这个就有点像是在改变模型对应的 UV 坐标);
Filter Mode:point;（邻近点滤波，边界就没有模糊的效果）
Compression:None，点击 Apply;
之后打开 Sprite Editor 面板进行切割，slice:Type:Grid By Cell Size;Pixel Size:X:16,y:16,点击 Apply.

使用 tileMap（瓦片地图）工具绘制游戏场景：
Tilemap 和 Tile Palettes（画布和调色板）:这里 tilemap 这里认为就是一块画布，将一单位格子的 sprite（tile 瓦片）作为一个绘图元素，绘制到画布上，每单位格的精灵片（tile）就成了瓦片地图的调色板中的元素。

在 hierarchy 面板中，右键 2D Object->Tilemap，会创建出创建 Grid/tilemap.这个 tileMap 就是那张画布，可以在上面放置多张画布，将 tilemap 命名为 tilemap_Ground;再创建一个 tilemap 命名为 tilemap_water,water 对主角有阻挡，单独画在一个画纸上。
创建 Tile palette(瓦片调色板)：window->2D->Tile palette;Asset 文件夹下创建 TilePalettes.

Tile palette 面板上点击“Create New Palette”，名为 Ground,点击 create，保存在刚才所长创建的 TilePalette 文件夹下。将文件 Sprites/GameEnv(已做过切割)，拖入到 TilePalette 中去，然后 unity 会要求你将切割好的 tiles 进行保存,就保存到刚才的 TilePalette 中去就行。

点击画笔工具 paint Brush（铅笔图标）的那个，点中一个 tile 调色板中的移动一个 tile,就可以在场景中绘制了，要擦除场景中的 tile，同时按住 shift 键即可。”[”，”]”可以旋转选中的 tile.
tilePalette 其他按钮的使用：
指针箭头：选择调色板或者场景中的一个区域 tiles;
十字箭头：将上个命令选择的区域进行移动
白板按钮：绘制时可以拖出一个区域框来填充选中的 tile 元素；
吸管工具：吸取场景中的一个 tile 元素，接着可以进行绘制，可以按住 ALT 去拾取 tile 块
油桶工具：选中一个 tile,填充整个区域

设置 TileMap_Water 场景的碰撞；
选中 tilemap_water 的层，在其 inspect 参数栏中，添加 tileMap collider 2D 组件，这个组件

为每一个单独的 tile 都会加上各自的碰撞体，这在进行碰撞检测和物理仿真时效率为非常的不高。我们需要的结果是那些挨着的那些 tile 的碰撞体要合成一个大的碰撞体。再增加一个 Composite Collider 2D 的组件，unity 会同时给它加上一个 rigidBody 2D 的组件,BodyType 需要设置成 static .然后原来的 tileMap collider 2D 需要勾上 Used by Composite;

场景中环境元素的搭建：
对 Sprites/GameEnv/SpriteObject 的精灵表进行切割：
基本参数设置于上节课的瓦片进行分割一样：
Pixels PerUnit:16
Fiter Mode:Point(像素分割都是选这种滤波模式);Bilinear(双线性，采样，亚像素，周围四个点，图片模糊，若是其他风格的都采用这种滤波模式);Trilinear（三线性，周围 9 个点）
点击 Apply.
打开 Sprite Editor 对精灵图片进行切割，对于这种类型精灵表的切割选择：Slice->Type->Automatic,让软件自动切割。要注意的是，这个边界不是程序自动判断出来的，是程序根据图片的 Alpha 来切割的，可以点击 Apply 右边的三原色小标记来查看图片的 Alpha 通道；一般比如 png,tga 图片的才会有 Alpha（透明通道）。若是那种不带透明通道的图片比如加入的图片是个纯白底的 jpg 图片，是无法用这种方法切割的。即使用 cell size 的方法去切割，拖入到场景是带有白底的。

切好的精灵片进行重命名

对角色的 sprite 表进行切割，Pixels PerUnit:26，角色大概 1 米多点的样子。
打开 sprite editor 进行切割：Type:Grid By cell size 32X32;

将 spriteObject 中的房子和树拖入场景中，若游戏对象未出现在 Game 视图中，在 Scene 视图先调好观察视角，然后记得在 Hierarchy 面板中选中相机，菜单栏 GameObject->Move To View(相机位置的移动)或者 Align With View（相机位置和旋转都变化），这里的 2D 话相机原来就没有旋转，都可用。
Game 视图中可以修改渲染图片分辨率大小，可以选择 HD1920X1080

若此时托进去的武器没有渲染出来，需要调整 sorting Layer 这个是跟渲染相关的层，表示图片渲染的先后顺序，通常在 renderer 组件中，不是最上面的 tag，Layer 中的那个 layer(这个 layer 是跟物体间的碰撞有关)，点开这个标签选择 Add Sorting layer, 以此填上 Background,SpriteObject,Character.选择上节课绘制的底层背景，在相关的 Renderer 组件的 Sorting Layer 中选择 Background,对 spriteObject 那一类的物体选择 SpriteObject，然后放入我们的角色，sorting Layer 填上 Character.这个表示渲染的顺序，程序会从上到下以此渲染，后渲染的会遮挡前面渲染的物体（这跟我们画画一样），然后同一层有遮挡，设置下面的 Order in Layer 参数，值大的渲染在上面，值小的渲染在下面。

拖入场景中的各个物体，inspector 参数栏的每一栏就是组成这个游戏对象(GameObject)的每个组件.

要调整环境物体的大小，可以调这个游戏对象的 transform 的 scale 参数；

再提组件：游戏场景中的每个物体都可以认为是一个 GameObject。一般在编写组件脚本的时候，要获取游戏对象的话，都是 GameObject 这个数据类型去引用的。每个 gameObject 具有什么样的功能或者表现出什么样的行为都是这个 gameObject 下的组件 component 决定的，在一个 gameObject 下可以挂载系统给我们的组件，比如 Collier 组件，renderer 组件等等，也可以根据 Unity 提供的自定义脚本（script）的组件方式，挂载到 gameObject 上去，现在常用的就是用 C#语言编写的脚本。可以说组件是 unity 游戏编程的灵魂，这里 GameObject 可看做是一个组件容器。

每个 GameObject 都会默认有一个 Transform 组件，控制这个 gameObject 在场景中的位置关系，以及这个 gameObject 自身的父子层级关系。

这边给我们的场景对象加上碰撞器组件 Collider 组件，是让物理引擎来检测两个物体是否碰撞的，然后会引发阻碍进而引起碰撞事件，或者不阻碍而引起触发事件；对于 2D 物体，系统提供有基本的碰撞器，比如 Box Collider2D,Circle Collider2D 等，还有复杂一点的，Polygon Collider2D 等，对于复杂的碰撞器可能会消耗 CPU 的计算，会算的久一点，对于简单的碰撞体就计算的快一点，比如判断两个圆形是否碰撞，只要计算两个圆心的距离是否小于两圆的半径之和即可。这里的场景物体都选用简单的碰撞体即可，对于稍微复杂的可以用多个几何体拼合即可，就是多挂几个碰撞组件。

引入 Prefab（预制件）的概念：
Project 面板中，新建一个 Prefabs 文件夹，再建两个 Character 和 EnvObject 文件夹，将 hierarchy 中的 bush 对象，拖入到这个文件夹下，就生成了一个这个对象的预制件；Prefab 可以认为是一类 asset。
Prefab 预制件的作用:若场景中有多个这一类的物体，可以在场景中通过 ctrl +d 复制生成，如果有需要修改，那么就得一个一个改过去。但有了预制体之后，我们只需要修改预制体就可以改变场景中所有的对象，预制体都关联着场景中的此类物体；还有一种情况，如果一开始某个游戏对象是 在游戏中生成的，如果写代码 new GameObjet(),AddComponent()...会显得非常麻烦。这个时候如果我们把这个游戏对象先做成预制件，然后再通过 instantiate()函数生成即可。

通过预制件实例化到场景中的游戏对象，可以修改自己的属性，若将这个修改的游戏对象要替换掉原来的预制件的话，点击 inspect 参数面板头上的 prefab->overrides->apply all

Prefab 底层：使用了 C#的序列化(serialize)和反序列化（deserialize）的知识点。序列化：将一个实例化的游戏对象转化成数据流（这里数据流，可以是 xml,二进制等，在程序语言中似乎对这种写入写出，传来传去的数据都称为流，C#，C++那边提供文件读写的类就是什么 iostream,fstream 之类的，stream：流），所以序列化的作用就是对数据的存储和传输。这里的 prefab 是将场景中的游戏对象保存成了一个文件，里面包含了游戏对象相关信息，我们可以用文本编辑器查看，选中文件，右键 show In explorer,是一个 YAML，里面保存着对象的成员变量的数据。这个文件也可以保存成二进制的文件，Edit->Project Settings->Editor->Asset Serialization Mode 改为 Force Binary 即可，Serialization 就是序列化的意思。
有了序列化，就有反序列化，我们将 project 面板中的 prefab 拖入到场景中就是一个反序列

化的过程，就是通过保存下来的数据流来生成一个游戏对象。

Unity Prefab 的必须性:通过 prefab 来生成游戏对象就好比我们写 C#代码时，自定义了一个类然后实例化类的过程；而 unity 这边的操作更像是先实例化了一个游戏对象，然后将它保存成预制件，然后实例化其他对象，那个函数的名称就是 instantiate()（实例化的意思）。在 unity 中一般写的就是组件代码，基本其他代码也是服务于组件的。没有这种 Player:GameObject 这样的类，GameObject 是个 sealed 的封装类，无法继承，那么要创建多个游戏对象来统一管理就需要采用这个 prefab 的方式，UE4 中有 Actor 对标于这里的 GameObject,但是在 UE4 可以继承 Actor 派生出各种游戏对象类（对标于这里的 prefab），然后进行实例化对象。

主角 DuDu 的制作：

主角的移动控制和与环境物体的碰撞：
Hierarchy 中创建一个空的 GameObject,重命名为 PlayerObject,在 Inspector 面板中，Add Component 选择 Sprite Renderer，这个 Renderer 负责将模型或者精灵片渲染到场景中，Sprite 先载入一张正面的精灵图片，Sorting Layer:Charactor;

创建一个脚本来控制角色移动：
Project 的 Assets 中新建 Scripts 文件夹，然后在里面创建 MonoBehaviours,将自定义组件的相关代码都放在这个文件夹里。进入 MonoBehaviors 右键 Create->C#,重命名为 PlayerMovementController,双击打开，需要注意的地方：文件名要保持一致，不然出错；从脚本给出的模板来看：引入了 using UnityEngine 这个库，类是继承自 MonoBehaviour 这个基类的，mono 指的是一个第三方的.Net 平台（可以认为给 C#程序提供各种功能函数以及运行的开发平台，mono 是跨系统平台的），后面的 behaviour 应该是在提示我们编写的是一个游戏对象的行为。只有继承了这个基类的类才能作为组件挂载到游戏对象上。类中有 Start()和 Update()两个函数，引擎会自动调用这个两个函数，start 在程序初始化的时候会被调用一次，Update 是每个渲染帧都会被调用，除了这两个还有 Awake(),FixedUpdate(),lateUpdate()，还有涉及碰撞的相关函数如 OnCollisionEnter()等函数，可以供开发者在提供的函数中写相关的逻辑。引擎暴露给我们的方法都是运行在一个主线程上的，调用有先后顺序。可参考链接中的执行顺序图：
https://docs.unity3d.com/cn/current/Manual/ExecutionOrder.html

浅谈脚本 Script，unity 这个引擎内部是用 C/C++,然后暴露出一些可编程的接口来让开发者写自己的逻辑程序，C/C++语言是直接跟硬件打交道的，所以它写出来的代码执行效率高，但是使用起来难度大。所以 unity 在这块就加入了一个脚本系统（脚本引擎），然后加入了脚本语言，这一层面的语言简单易学，方便使用，unity 用的是 C#，感觉就像是 C/C++语言包封装成了 C#可以调用的库让外面的 C#脚本语言调用；unity 中提到的脚本语言似乎与普通意义上的脚本语言不同，一般说的脚本语言是 python，shell 这类的，他们都不需要编译成 exe，直接用解释器一条条执行下来的。但 C#不是。但两者作用似乎一致，就是方便我们的开发，脚本语言就好比程序开发的 UI 界面，我们写了一个程序，让别人去使用它就会加个 UI 界面；那么可以类比，我们在 unity 的内部写逻辑代码，里面的 C/C++太难了，不好开发，那么我们选一个简单的容易学的语言，作为这个开发的“UI 界面”，方便我们开发；

在移动我们的角色之前需要完成跟输入设备的交互，因为我们都是靠外部设备（比如键盘）来操纵角色的移动的。
这边 unity 已经帮我们做好了输入设备与输入控制代码这块的联系，首先，对一些按键的操作做了一些分类： Project Settings -> Input 中的列表，比如这里的 Horizontal 关联着 a,d,Left,right 键，然后在代码中就有个 Input 类来捕获我们的 Horizontal 标识下的按键。
代码部分：
成员变量(字段)：

```
Public float moveSpeed= 3.0f;//public 修饰的字段会在 inspect 窗口暴露出来
Vector2 moveDir = new Vector2();
```

成员函数：

```
Private void MoveCharacter()
{
    Movement.x = Input.GetAxisRaw(“Horizontal”);
    Movement.y = Input.GetAxisRaw(“Vertical”);
    Movement.Normalize();//向量的归一化，保证每个方向的移动速度一致。

}
```

Input.GetAxisRaw(“Horizontal”):捕获 Horizontal 下的按键，当按到 A 或者 Left 时返回 1，当按到 D 或者 Right 时返回-1；操作按键的时候，要把鼠标聚焦到 Game 视图，就是需要将鼠标移到 Game 视图然后再点击一下。

在 Update 函数调用 MoveCharacter 函数
还有：在 unity 的那些继承自中 monobehavior 的脚本中，这个类的实例化是由系统帮我们操作的，因此我们不要显示去写它的构造函数；还有脚本中的都是字段，不需要写这个字段的属性，属性无法在 inspect 窗口进行调试。

Unity C#脚本的调试，通过查询一些变量的值来排除错误：
1. Debug.Log();用于打印，就是 ConsoleWriteLine()；填入字符串即可。
2. 断点调式：VS 菜单栏：调试-> 附加 unity 调试程序，弹出一个选择 unity 实例，然后点中按确定即可，unity 窗口中按播放，进入断点调试，F10,F11 逐过程，逐语句调试。

组件脚本代码写完后，可以在编辑器中直接托给游戏角色对象，或者在 AddComponent 按钮上也能找到写好的组件。需要等代码编译完后，才能按 Game 窗口的执行按钮（工具栏上播放键），不然会报错，选中脚本右键 Reimport 重新导入等待编译完成即可。

在 unity 移动的方法有很多种，但主要是分为两类；
1. 基于 transform 组件的移动，可以通过修改 Transform 的 position 的值来移动物体，Transform 组件的变量为 transform，可以直接获取，获取其他组件就必须用 GetComponent<组件名>()

```
transform.position += new Vector3(deltaX,deltaY,deltaZ)*moveSpeed*Time.deltaTime;
```

deltaTime 表示每帧的渲染时间，值约等于 1/FPS（framerate per second）,一般的话 40FPS，

就表示每秒钟对调用 40 次 update 函数，所以在 update()函数在运动的话，都会乘上 deltaTime，一是设置的像移动速度 moveSpeed 这类值的时候更合理，有依可循，二是电脑帧率不一样的话，乘上这个 deltaTime 可以保证，运动的节奏是一致的。

使用 Vector2.MoveTowards()函数,这个函数与 Vector2.Lerp()函数（插值）用法基本一致。 它的用法是给定一个起始点和终点，和一个权重值，让游戏对象沿着目标点移动，这种方式适合那种鼠标点击哪里，然后让对象移动过去，或者敌人之类的去追逐玩家。有种跟随的感觉。
transform.position = Vector2.MoveTowards(transform.position,target.Position,deltaTime);

使用 transform.Translate 函数 ，参数中填入相关的增加即可.
transform.Translate(new Vector3(deltaX,deltaY,deltaZ));与上面的第一种差不多

2. 基于 rigidbody（或者说是基于物理引擎）的移动，在物理引擎中这个物体就成了一个刚体，可以通过施加力来移动它，也可以设置这个刚体的速度和位置来移动它。在 playerObject 中添加 rigidBody 2D 的组件， Body Type:Dynamic;Gravity Scale 设置为 0，不然我们的角色会掉下去；

关于动力学的更新都在 FixedUpdate(),这个更新函数引擎默认的 0.02s(可用变量 fixedDeltaTime 来获取)调用一次，就是 50fps.
一般都比渲染帧率要高；适合这种物理引擎的模拟的调用频率。

使用 rigidBody.AddForce(Vector2()):通过施加一个力，linear drag（阻力）不能为 0，不然会一直匀速运动下去，在运动方向的话有那种滑动的效果，感觉这个函数适用于角色起跳中。
移动的脚本中获取获取这个刚体组件：
Private Rigidbody2D rd2D;
在 Start()函数中写上：rb2D = GetComponent<Rigidbody2D>();//调用一次
MoveCharacter()函数中加上：
Rb2D.AddForce(new Vector2(moveDir.x.moveDir.y));没有阻力（drag）的话,施加之后物体会一致运动下去；作为移动不好控制，这种 AddForce 的方式，适合作为角色跳跃的时候施加的一个力，不过我们这里是 TopDown 模式的，不需要跳跃。

改变刚体的位置，rigidBody.MovePosition()或者 rigidBody.position +=:，参数带入的是刚体的具体位置，不是增量位置，似乎与 transform 的 position 一致。
rigidBody.MovePosition(new Vector2(transform.position.x,transform.position.y)+ moveDir*moveSpeed*Time.FixedTimeDelta)
rigidBody.position += moveDir*moveSpeed*Time.FixedTimeDelta;因为这边速度一般都是每秒的速度，所以需要乘上 Time.FixedTimeDelta。

修改刚体的速度:rigidBody.Velocity = moveDir*moveSpeed； 不需要乘上 Time.FixedTimeDelta，因为每一帧移动的距离就是 s = v*t = velocity * FixedTimeDelta;教程这边就修改这个刚体速度的方法来控制我们角色的移动。

另外，还有一个组件 CharacterController 的组件，也可以控制角色的移动，但似乎是用于 3D

角色的。

增加角色与物体的碰撞：给角色增加一个 box Collision2D 的碰撞体，rigidBody 中冻结刚体的沿着 z 轴方向的旋转，Rigidbody 2D->Constraints->Freeze Rotation Z 加上勾。

碰撞发生的条件：
两边的物体都有碰撞器，动态的物体要挂上 rigidbody，rigidBody typeBody:Dynamic;场景中的环境物体似乎不需要挂上 rigidBody 就有这种阻碍的效果了。
RigidBody :Body Type:Kinematic（运动学），就是这个刚体就不受力的作用。可以使用上面的 Transform 组件的方式移动物体，或者也可以用 rigidBody.MovePosition()。可以与 Dynamic 类型的物体发生碰撞。这个属性可以进行游戏对象的受力运动与动画之间的切换，比如抛东西。

一般碰撞的有两种现象：一种是阻碍，产生一个 OnCollision 的事件；另一种是 collider 的组件上勾上 IsTrigger，那么此时不阻碍，产生一个 OnTrigger 的事件；

PlayerObject 的 Tag 选成 Player, 设置成这个的话，方便其他脚本找到场景中的角色，用 GameObject.FindGameObjectWithTag()函数。Layer 是设置游戏对象的碰撞检测的管理，可在 Project Setting->Physics 2D->Layer Collision Matrix，打勾的表示这些碰撞组可以相互碰撞。不打勾的话即使有碰撞器和刚体就不会发生碰撞作用。

制作主角 DuDu 的动画
Project 面板下新建一个 Animation 文件夹，其下再建两个文件夹 Animations 和 Controller,
一般动画模块这部分有两个概念，一个是动画片段（对应于 Animation），表示角色的每个动作序列帧，一个是动画控制器 Animator（也可以就称为动画状态机），管理动画片段，实现动画片段间的切换。

我们选择 DuDu 角色动画片段序列，按住 shift 选择正面的那一组 idle 动画，直接拖给 hierarchy 上面的角色 playerObject,弹出对象框进行动画片段的保存,命名为 player_idle_south，此时系统会给我们的 playerObject 添加一个 Animator 的组件，组件第一个参数 Controller 指向的就是这个动画角色的动画控制器（是刚保存动画片段时一起创建的）。

在刚保存的文件中找到这个 playerObject 的控制器，重名为 PlayerAnimCtrl,并把它拖入 Controller 的文件夹内，方便管理（这个 Animator Controller 节点也可以自己创建，projector 面板右键，create ->Animator Controller）。

双击 PlayerAnimCtrl，会打开一个 Animator 的动画面板，负责管理角色动画片段的切换。
可以看到里面的动画片段（动画片段可以连线进行切换）以及 Entry,Any,exit 节点，Entry 节点控制角色初始化的动画状态，与它相连的动画状态的节点显示为金黄色；Any 节点代表面板中的任何节点，这个 Any 节点通常用于这种情况，若我们的角色有好几个动画状态，比如 idle,walk,attack,这几个状态都直接切换到我们的角色的死亡状态，就可以用 Any 连给角色的死亡片段。Exit 节点不知道怎么用？其他都是动画片段节点，名字与动画片段的名字保存一

致，点中查看参数面板，motion 就是它 Animation 中的动画片段，speed 可以调整动画片段的播放速度。右键某个动画片段的节点 Set as Layer Default State

双击 player_idle_south 不会弹出动画片段的编辑框。此时需要，打开 Window->Animation->Animation 面板框，然后需要在 hierarchy 中选中 playerObject 角色物体，就能在 Animation 窗口中看到这个动画片段了，其实这个动画就是对游戏对象的 SpriteRenderer 这个节点中 Sprite 进行了 K 帧，其实动画的实质就是对游戏对象的某些属性进行 K 帧，也可以点击 AddProperty 来选择游戏对象的其他属性 K 动画。也可以创建点击 player_idle_south 所在的动画片段显示栏创建新的动画片段 create New clip,点击创建我们的第二个状态 player_walk_south,Add Property->Spriter Renderer->Sprite 右边的加号，将正面走路的动画序列帧拖入进去，把动画动画片段放入第 1 帧和第 2 帧。按播放查看动画，samples 参数表示动画帧率，可以调整这个参数来改变动画的节奏，默认的每秒采样 60 次，太快可能导致无法播放，改成 12。

角色 idle 动画和 walk 动画的切换： 选中 idle 动画，右键 make Transtion 会连出一条线拖给 walk 动画，walk 动画也可以拖出一条线连给 idle 动画。动画的切换需要通过代码去控制，这里点击连线 可以看到 inspect 参数面板中显示了相关的参数，一个状态向另一状态切换可以多连几条，表示有多个切换条件，这些条件是“或”的关系，就是有一个条件满足就可以进行切换。选中连线，可以在参数面板的 Transitions 查看切换条件，每一条切换条件对应这下面的 Condition 条件，可以选中切换条件然后按右下角的减号键进行删除；如果是要多个条件同时满足才能切换的话，那么就只需要一条切换线，然后在 conditional 中设置多个条件，这里的条件是需要同时满足的，在逻辑运算中是“且”的关系；

动画切换操作：
Animator 面板的左上角 Parameters 标签下，选择“+”，添加一个 Bool 变量，名字命名为 isWalking,旁边有一个小勾，勾上 isWalking 的默认值为 true，不勾为 false;然后点击 idle->walk 的连线，在参数面板的 Conditions，点击“+”选择 isWalking,然后旁边的值选择 true，反之选择 walk->idle 的连线，也是同上的操作，isWalking 选择 false;
PlayerMovementController 的代码中来控制这 isWalking 的变量
脚本中获取动画组件的引用：
Private Animator anim;
Start()函数中进行获取：
Anim = GetComponent<Animator>();
增加函数来控制 Animator Controller 中的 isWalking 变量
Private void UpdateState(),这个函数将其放在 Update 函数调用即可；
函数内部：

```
{
If(Mathf.Approximately(movement.x,0) &&Mathf.Approximately(movement.y,0))
{
Anim.SetBool(“isWalking”,false);
}else
{
Anim.SetBool(“isWalking”,true);
```

}
}
这个函数的意思是当 movement.x 或者 movement.y 等于 0 的时候，即玩家没有按下控制键，那么角色是禁止的的，isWalking 的变量就设置为 false,这里用 Mathf.Approximately(movement.x,0)有单词意思上看是约等于 0，这是因为对于 float 类型的浮点数有精度误差，可能不能到绝对的 0，所以一般不用 movement.x == 0 这种表达式，一般都用 abs(movement.x -很小很小的接近于 0 的数)<一个很小容差值即可。
程序那边也在按一定的调用频率在调用那个动画状态机的状态，当它检测到 isWalking 等于 true 的时候，就将 idle 动画切换到 walk 动画。

看到的效果是切换不流畅：当我按下移动按键时，没有立刻切换到移动的动画：这就需要设置下切换连线上的一些参数了：
Has Exit Time/Exit Time:去掉勾：这个参数可以理解为：当前动画要播放百分之多少再开始切换成另一个状态，设置成 1 的话就是表示当前动画要全播完后才能进行切换。取消这个勾的话，就表明可以随时打断这个动画；一般的话，攻击的动画切换成其他动画应该适合这个参数；
Fixed Duration/Transition Duration:控制两个动画切换过程中的融合的时间的控制，Fixed Duration 打上勾，Transition Duration 以 s（秒）作为标准进行计算，不打勾以%已经计算，下面的图标有显示，上面的数字表示帧，然后以 1s 为整个单位，按%的话，是以动画的%为单位。这个参数应该都是用在 3D 骨骼动画比较多。这里就将 Transition Dutation(s)设置为 0 即可，图片不需要融合。
Interruption Source:一般用于有多个切换条件的情况；可以参考下面的博客：
https://blog.csdn.net/linjf520/article/details/90301909

主角动画 BlendTree
主角有 4 个东南西北方向的 idle,walk 和 attack 的动画，对应的动画片段有
Idle:player_idle_east,player_idle_south,player_idle_west,player_idle_north
Walk:player_walk_east,player_walk_south,player_walk_west,player_walk_north
Attack:player_attack_east,player_attack_south,player_attack_west,player_attack_north
这几个动画片段之间都可以相互转换，如果用上节课的方法，会使得整个动画状态机中的节点图非常复杂。

Unity 中的 blend tree(混合树，融合树)非常适合用于这种 4 个方向的动画，而且使得动画状态机里的节点非常简洁，易于管理，其实我们的动画的本质就是 idle，walk 和 attack 这三个动画的转换。Blend tree 节点，可以在 Animator 面板中，右键 CreateState->From new Blend Tree,跟上节课用的动画 state 节点是同一层级，不同的是动画 state 只包含一个动画序列，而 blendTree 里面可以包含多个动画序列，但是 blendTree 使用时也只输出一个动画序列，
Blend tree 直意就是两个或者多个动画之间的融合，所以 blendTree 会通过某些参数对动画进行融合然后输出一个动画序列，在我们这里的 2D 动画中会将同一个类型(比如都是 idle 的动画)的动画放在一个 blendTree 中，那么我们会创建 3 个 blendTree,
BT_player_idle,BT_player_walk,BT_player_attack,在这个的 Blend tree 中，2D 动画序列之间应该不是融合，而是选择，我们对设置对应的参数（可以有多个）来选择出一个动画序列，比如用方向按键的值之类的。

具体操作：这里只显示 BT_player_idle,BT_player_walk 的制作，以及相互的切换，
双击进入 BT_player_idle，blend Type 选择 2D Simple Diectional,因为我们这边的动画就是简单的四个方向的动画切换；Motion 中点击四次下面的“+”，选择 Add motion field 然后分别载入对应的 player_idle_east,player_idle_south,player_idle_west,player_idle_north,Animator 面板上增加两个 float 类型的参数 xDir,yDir，这两个参数会与玩家的控制输入移动的值挂钩，BlendTree 的 BlendType 参数下的 Parameters 选择 xDir,yDir，这两个参数代表下面的图标的 X，Y 轴。Motion 中 east,south,east,north 分别在 Pos X 和 Pos Y 中填入 1，0；0，-1；-1，0；0，1；在这个 2D 坐标上分别坐落在东南西北四个方位；
点击面板上的 BaseLayer 退出 BT_player_idle 这个混合树，进入 BT_player_walk 的 blend Tree，然后跟 idle 一样的设置，载入的动画都换成 walk 的动画就可以了。
BT_player_idle 和 BT_player_walk 相互 make transition,然后切换的条件和各参数设置跟上节课一样。
playerMovementController 脚本的代码中补上：
在 anim.SetBool(“isWalking”,true)的前面写上下面两句即可：
anim.SetFloat(“xDir”,moveDir.x);
anim.SetFloat(“yDir”,moveDir.y);

这样就能保证朝某个方向走动玩之后，如果此时释放按钮，角色的 idle 方向保留在走动的方向，这是因为当我们释放按钮完按钮，代码就不执行 anim.SetFloat 这两句了，那么动画状态机那边保留的 xDir 和 yDir 就是走动时候的值。

进入游戏的时候角色朝向了 east 方向，这应该是系统随机选择的，如果要将它朝向 正面的南方，可以在动画状态机面板上 xDir 和 yDir 的初始值设置为 0 和-1。

相机跟随：
正交（Orthographic）相机和透视(Perspective)相机：我们现实中的相机就是个透视相机，成像物体有近大远小的效应，它的视角是一个立方梯形（有近平面和远平面）的光柱，距离越远成像面越大，所以远处的物体在成像平面上就越小。正交相机它的视角就是一个长的立方体，成像物体没有近大远小的效应，一般的 2D 平面游戏用的正交相机。

Hierarchy 中选择 Main Camera，在 inspector 面板中需要关注的是 size 这个参数，它设置的是相机的高度的视角范围，相机的高度的视角在场景视图中的范围是（size 的值乘以 2）标准网格的数量 ；然后宽度的视角范围就可以由渲染视图的分辨率大小决定了。

相机跟随角色的运动：

相机拖给角色，做成父子关系，相机就能跟着角色走了

自己写脚本的方式：
Hierarchy 中 create empty 创建一个空的 GameObject,命名为 CameraFollow,transform 组件 reset，表示参数归 0，Main Camera 参数也归 0，然后 MainCamera 拖入到 CameraFollow 中

去作为它的子物体，此时 MainCamera 的 Z 轴数值跟场景的 Z 轴数值一样的话就无法在渲染出场景了， cameraZ 轴调成负值，在 Scene 视图下选个好的视角，然后选中 hierarchy 中的 CameraFollow 的物体，最后 GameObject->MoveToView;

创建脚本 CameraFollow.cs
Private Transform playerTransform;
Public float followSpeed=2.0f;
Start()函数中：
playerTransform = GameObject.FindGameObjectByTag("Player").transform;
Transform.position = playerTransform.position;

lateUpdate()函数，这个函数也是每帧调用，在 Update 之后调用
If(playerTransform!=null)
{
Transform.position =
Vector2.Lerp(transform.position,playerTransform.position,followSpeed*Time.deltaTime);
}
对于有些场景，边界区域没有相关的美术资源，就希望相机的边框到边界之后相机能停下来，角色还能走动一段距离。代码实现：
需要预先准备好相机的能移动的最左 minX,最右 maxX,最上 maxY,以及最下 minY 的数值
Public float minX;
Public float maxX;
Public float maxY;
Public float minY;
再声明一个目标向量：
Vector2 targetPosition = new Vector2();
在 lateUpdate 的 Transform.position 前加上一个 clamp 函数：
TargetPosition.x = Mathf.Clamp(playerTransform.position.x,minX,maxX);
TargetPosition.y = Mathf.Clamp(playerTransform.position.x,minY,maxY);
Lerp()中的 playerTransform.position 改为 targetPosition 即可。
然后在 inspect 面板中设置 minX，maxX，maxY，minY 即可
Clamp()是个夹逼函数，函数原型是 float clamp(float value ,float min,float max)
当 min<=value<=max,返回 value
当 value < min ,返回 min
当 value >max，返回 max

使用 Cinemachine 的插件，cinemachine 是一个很强大的操作镜头画面的工具集，应该是模拟电影中的相机功能，提供了很多很好用的功能，比如镜头融合和镜头切换之类的，我们这里就只用它的相机跟随的功能。
需要先安装下这个插件：2018 版本是在 Window->Package Manager,可以搜索 cinemachine,然后点击 install.

unity 的插件系统是一块很强大的存在，可以让开发者自己定制相关的功能，比如一些硬件设备要搭配一些软件平台，就可以使用 unity，通过 unity 提供的一些编程接口来制作对应的插件，我们就能够在 unity 这个平台上使用我们的设备了。或者我们也可以使用其他人开发好的分享出来的插件来方便我们的开发。

一般情况下，插件什么的，包括各种游戏资源都在 asset store 下载，但是 unity 似乎会把一些好的实用的插件吸收进自己的软件里的，这些插件就放在 Window->Package Manager，Package 包管理器中。

安装完后就可以在 project/Packages 中看到这个 cinemachine 的 package 包，菜单栏上会有一个 cineMachine 的标签，插件的话很多都会跟着版本修改而修改，包括它自身的版本和 unity 的版本，所以使用之前需要读一下它在下载页面上的说明，现在这个版本的相关的创建都在 GameObject/Cinemachine 上了。

把上节课的相机归位，从 CameraFollow 中移出来，然后 moveToView 到角色所在的视图位置；GameObject/Cinmachine，创建 2D camera.Hierachy 视图中创建了一个 CM vcam1 的虚拟相机（virtual camera）,然后 Main Camera 中多了一个 Cinemachine Brain 组件。

Virtual camera 可以认为是 Main Camera 相机的一个控制器，原来相机的很多设置可以在这里设置了，比如位置，视角参数等，follow 中拖入我们的 playerObject,作为相机的跟随目标。这样相机就能跟随我们的角色了。
Main Camera 的 Cinemachine Brain 组件将 Main Camera 和 Virtual Camera 联系了起来；

CM vcam 游戏对象上的 Body 栏的参数，Dead zone width/Height 可以设置角色相对静止的区域，在这个区域移动时，相机并没有跟随运动。Damping 的那个参数是给相机移动加上阻力，使得相机移动的不能那么顺滑。

为场景增加边界，就是上节课的 minX,maxX,minY 和 maxY 设置的效果，
使用 Cinemachine Confiner 组件，confiner 原意就是边界的意思。
在 CM vcam1 的 Extension->Add Extension->Cinemachine Confiner ,confineMode：Confine2D;
场景中创建一个 create empty，命名为 BackgroundBound,然后加入一个 polygon collider2d 的组件，点击 Edit 进行编辑，拖动 polygon 上面的点来拖出场景的边界范围，还有这个 collider 要勾上 isTrigger 变成触发器，将 BackgroundBound 的游戏对象拖拽给 confiner 组件的 Bounding shape 2D 这个参数

游戏道具的制作以及与游戏角色的交互：
会创建一下几个道具：
金币：CoinObject:角色搜集金币，现游戏的任务暂定为需要收集光场景中的金币。
红血瓶：补充角色的 healthPoint 的生命值
蓝血瓶：补充角色的 staminaPoint 的能量值，这个值影响角色投掷物的远近程度
普通投掷物，火焰投掷物，水色投掷物：用于角色攻击敌人

上面的各个道具都分别在 Hierachy 中进行创建，然后最后都制成 prefab 预制体。

制作金币道具：
创建一个 CoinObject，像之前那样载入素材，对精灵图片进行切割；然后将图片序列拖给 Hierachy 创建一个空物体，命名为 CoinObject，将素材序列的金币动拖给它，保存成 coinSpin 的动画片段在 Animation/Aimations 文件夹内，把金币的动画控制器重命名为 CoinAnimCtrl 拖到 Animation/Controller 的控制器中。双击 CoinAnimCtrl 进入 Animator 的面板，选择动画状态节点，调整动画播放速度 Speed：0.5，

Renderer 组件中 sorting layers 中，Add Sorting layers 增加一个 PickUpObject，先将 Character 调整到 SpriteObject 的上面，然后将 PickUpObject,放在 SpriteObject 和 Character 之间。
场景视图中若没有显示可以拖入一个金币的在精灵图片给 SpriteRenderer 的 Sprite 参数，transform 组件中调整大小 x:0.6,y:0.6;若在 Game 视图中没有显示出来，可能是 transform 中的 z 轴超出了相机的可视范围，切换到 3D 视图进行调整；

CoinObject 组件中添加一个 Circle Collider 2D 的组件，点击 Edit 按钮，调成这个碰撞体形状使得跟金币的大小差不多即可或者直接调整 Radius。勾上碰撞器组件的 IsTrigger，我们的道具不会阻碍主角或者敌人，并且只会跟我们的主角发生触碰时产生 Trigger 事件。

在 Tag 标签中点击 AddTag，然后在对应的面板上点击“+”增加 CoinPickupObject 和 PickupObject 这个标签，将 CoinObject 的 Tag 标为 CoinPickupObject;在旁边的 Layer 标签中也选择 Add Layer,添加 Pickup 和 Enemy 的 layer，在 project settings->Physice 2D->Layer Collision Matrix 设置 去掉 Pickup 与 Enemy 的勾。

Assets/Prefabs 文件夹内创建一个新文件夹 Pickup,将 CoinObject 拖到这个文件夹下。

红血瓶，蓝血瓶，普通投掷物，火焰投掷物，水色投掷物这几个的设置都与上面的金币道具一样，分别命名成
BloodObject,StaminaObject,CommonBallObject,FireBallObject,WaterBallObject 这几个道具都没有动画，因此需要先创建 Sprite Renderer 组件，拖入对应的素材给 Sprite 参数即可，然后 tag 都选择成 PickupObject,之后在 Layer 标签中 Add layer 增加一个 Ammo 的 layer，将红血瓶，蓝血瓶的 layer 设置成 Pickup，将普通投掷物，火焰投掷物，水色投掷物的 layer 设置成 Ammo。在 project settings->Physics 2D->Layer Collision Matrix 设置 保留 Ammo 与 Enemy 的勾，去掉 Ammo 与其他的勾。
然后将红血瓶，蓝血瓶，普通投掷物，火焰投掷物，水色投掷物分别拖入到 Prefabs/Pickup 文件夹中做成预制体。

做成预制体后，可以将原来的那些 pickup 道具删除掉，然后从预制体重新拖入场景，这里需要注意的是：投入场景的普通投掷物，火焰投掷物，水色投掷物的 layer 要改为 Pickup，表示这几个投掷物再场景中作为拾取物的时候是不和敌人发生作用的，我们的投掷物预制体从我们角色抛出才会与敌人发生作用。

角色增加新的自定义脚本，来处理与道具以及之后与敌人的一些交互逻辑，我们可以写在之前的 PlayerMovementController.cs 的组件中，但这里一般都是建议将自定义的组件做成一个个的独立的功能，所以这里就另外写一个自定义的类 ，命名为 Player.cs。

这里我们先创建一个 Character.cs 的父类，Player 继承于 Character.cs,之后创建的敌人的类 Enemy 也会继承于 Character，或者之后还有什么其他的角色类都可以继承于 Character，我们会将游戏任务角色的共有的属性都写在这个 Character 中。主角和敌人都有生命值，所以在 Character 类中就加上 healthPoints 和 maxHealthPoints 表示当前生命值和最大生命值。

```
Class Character:MonoBehaviour
{
    Public float healthPoints;
    Public float maxHealthPoints;
}
```

让 Player 类继承 Character 类,然后加上能量值，这个值是控制投掷物的远近的，然后加上 OnTriggerEnter2D(Collider2D collision)的，这个函数系统自动调用，当角色与其他物体发生 Trigger 时会调用这个里面的函数。触发满足的条件，函数只调用一次，当两者的碰撞器相互接触的时候调用这个函数，其中一个是勾上 isTrigger;这个 ontrigger 函数写在两个物体的任何一个都行，参数 Collider2D collision 是碰撞的另一方的碰撞器，从这个碰撞器能获取另一个的游戏对象 collision.gameObject 了，从而能知道另一方的所有信息。

```
Class Player:Character
{
    Public float staminaPoints;
    Public float maxStaminaPoints;

  Void OnTriggerEnter2D(Collider2D collision)
  {
  If(collision.gameObject.ComTag(“PickupObject”)||collision.gameObject.ComTag(“CoinPickupObject”))
  {
  Collision.gameObject.SetActive(false);//SetActive 函数表示激活或者不激活这个游戏对象，当不激活时，游戏对象在场景中不显示。相当于在 inspector 面板上勾掉这个游戏对象名字旁边的勾。这就话性能的损耗要比直接 destroy 要小。
  }
  }
}
```

将 Player 脚本挂到 playerObject 上进行测试。

Scriptable Object 的引入和使用：

脚本化对象，可编程对象

Scriptable 对象是一个数据容器，它是一个保存在 project/assets 中一种数字资产（asset）文件，它就像一个类似于 XML 或者其他格式的配置文件，使用时都需要加载它来将数据读进内存中。

作用和特点：

1. 一般通过 prefab 预制件的方式来实例化对象的话，prefab 里面若有对某段数据（可能内存占用很大）的引用的话，那么用 prefab 创建的实实例都会复制出这段数据，那么就会消

耗很大的内存。如果这段数据把它做成 Scriptable 对象引用的话，那么这段数据就只有一份，所有的 prefab 实例都指向这份数据，节省了内存。
2. 使得程序的架构更加简洁干净低耦合。这里我们的游戏中通常把它定义成对物件（item）进行描述的数据集，然后可以让我们的角色，背包系统，或者商店系统对它里面的数据进行读取。
3. 它是一种数据，继承自 ScriptableObject，不是一种行为（monobehavior），不能作为组件挂载在物体上的，只能作为一个数据让其他对象或者脚本引用。
4. 若是需要修改 ScriptableObject 的数据，建议时复制出一份来，在编辑器模式下运行游戏的话，对 ScriptableObject 的数据修改，退出运行后是保存下来的。将程序进行打包后再运行，这个修改不保存。

课程主要针对游戏道具创建 Scriptable Object，存储游戏道具的一些相关的信息数据；project/Scripts 创建新文件夹 ScriptableObjects,，创建 C#脚本命名为 Item.cs，让它继承自 ScriptableObject，模板中的 Start 和 update 函数就需要删除;类中主要设置游戏道具的一些共有信息字段

```
Public string objectName;
Public Sprite sprite;
Public int quantity;
Public float amount;
Public bool stackable;
Public enum ItemType
{
COIN,
HEALTH，
STAMINA,
COMMONBALL,
FIREBALL
WATERBALL,
}
Public ItemType itemType;
```

在这个类前加上下面的特性(Attribute),
[CreateAssetMenu(menuName=”Item”)]；C#中特性也是一个类，特性是给类，方法，属性等增加一些额外信息，常通过反射获取这些特性，然后做些特殊处理。这里就相当于在调用这个类的构造函数。

这个 ScriptableObject 的实例化有点特别，它一般不是在代码中 new 出实例对象，它实例出的对象是个资源文件，就像 prefab 一样保存在资源文件下，然后应该是通过序列化/ 反序列化的那一套用法在代码中使用。

实例化出一个 ScriptableObject 类型的对象，project 鼠标面板上右键 Create->Item,就实例化出一个 ScriptableObject 类型的数字资产,命名为 CoinItem，在旁边的 inspector 面板中对应的

参数需要设置下对应的字段的数值
objectName:coin
Sprite:拖入一张金币的正面的精灵素材
Quantity:0
Amount:0//这个值给血瓶和投掷物使用。
Stackable:true
Item Type:COIN

在 scripts/monobehaviours 文件夹下创建一个 Pickup 的脚本继承自 monobehaviour：声明一个对 Item 类型的引用。
Public Item item;
将这个 Pickup 的脚本挂载给 Coin 的 Prefab。Item 这个变量拖入之前创建的 CoinItem。这个 Coin 的 prefab 就包换了这个 CoinItem 的引用，而且之后用 coin 的 prefab 实例化出来的游戏对象都会引用这一份数据。

当我们的角色与金币相关触碰时：
在 Player.cs 的脚本组件的 OnTriggerEnter2D 函数中获取这个 CoinObject 的 CoinItem，
在 TriggerEnter2D 加入如下代码：

```
        if
(collision.gameObject.CompareTag("PickupObject")||collision.gameObject.CompareTag("Coin
PickupObject"))
        {
            Item hitObject = collision.gameObject.GetComponent<Pickup>().item;
            if (hitObject != null) //这句话要写全了
            {
                Debug.Log(hitObject.name);
                hitObject.quantity +=1;
                collision.gameObject.SetActive(false);
            }
        }
```

用 Item 的 scriptable object 为其他游戏道具创建 item 的实例对象，来保存格子的游戏对象的一些数据信息，都是右键 create ->Item，分别命名为 HealthItem,StaminaItem, CommonBallItem,FireBallItem,WaterBallItem，然后给对应的字段附上相关的参数即可，这边话，投掷物的道具的 stackable 的字段都是 false;之后点击 Add Component 将那个 Pickup 的脚本加入到对应游戏道具的 prefab，然后将这些 scriptable object 的实例数据都赋值给那个 item 的值即可。

OnTriggerEnter 的代码可以增加些代码，用 switch 语句将收集到的道具，通过 itemType 来分类，执行相应的逻辑：在 if(hitObject !=null)的代码中加上：

```
Switch (hitObject.itemType)
{
Case Item.ItemType.COIN:
```

```
Break;
Case Item.ItemType.HEALTH:
AdjustHealthPoints(hitObject.amount);
Break;
Case Item.ItemType.STAMINA:
AdjustStaminaPoints(hitObject.amount);
Break;
.
.
.
Default:
Break;
}
```

AdjustHealthPoints(float amount)和 AdjustStaminaPoints(float amount)是 player 类中的自定义函数,里面的代码都是 xxxPoints+=amount；然后打印下输出的数值即可；

制作 UI 相关显示：血条，能量条，和金币数量
设计样式：放置在屏幕的右上角，一个大的主角头像 Logo，然后旁边并排放置血条和能量条，然后下面显示金币的收集数量

这边在原来的 SpriteObject.png 素材中，我又重新加了一张 32x32 的角色头像的 Logo,然后将整张图片覆盖了 unity 原来的那张。找到这个素材的文件位置，然后将新的图覆盖上，如果 unity 页面中没换过来的话，可以点中图片 Reimport 一下。然后打开 Sprite Editor 框框选头像 Logo 即可，取名 HeadLogo 即可。

Hierachy 右键 UI->Canvas,可以重命名 PlayerCanvas 会同时创建一个 EventSystem.Canvas 原意就是画布的意思，就是说我们的 UI 元素就要画在这个 Canvas 上，场景中表现为其他 UI 元素要作为 Canvas 的子物体。EventSystem 会处理 UI 元素的点击事件，我们之后需要给一些 UI 元素加入点击事件，所以这里不要删除。

UI 基本元素包含有 button(按钮),image(图片),Slider(滑杆),label(标签),input field(输入框)等，右击 Hierachy->UI 都能看到。UI 元素也是 GameObject 类型的。

这里的 UI 元素都是在屏幕空间的，所以 Canvas 的 Render Mode 要选择 Screen Space Overlay，然后也勾上 Pixel Perfect；我们的场景可以认为是在三维空间中的，然后经相机投影投影到屏幕空间中的，这里的 UI 元素是不会随相机的移动而移动，一直是显示屏幕空间的最上层，此外，也有 3D 空间中的 UI 元素（比如常看到的敌人头上的血条），不过这里的课程不会用到。

当屏幕的分辨率发生变化时，Canvas 会跟着进行缩放，里面的 UI 元素也会根据相对位置进行调整和缩放。在 Canvas Scaler 组件中设置 UI Scale Mode:Scale With Screen Size;然后

Reference Resolution 可以填我们现在用的分辨率 1920 和 1080，Reference pixels Per Unit 填 32，这个参数跟我们提供的素材的大小有关，也决定了 UI 元素在屏幕上的大小了。

选中 PlayerCanvas 的情况下，右键 UI->Image,重命名为 HeadLogoBK_img,图片的大小和锚点进行设置，大小调整成 120*120，在 UI 中用 Rect transform 来进行调整，对应的调整工具是工具栏的 Rect tool 工具，锚点 Anchors，选择左上角的锚点位置，视图中有个花状的标志就是锚点，它的作用是，当屏幕的分辨率（长宽比）改变的时候，屏幕中的 UI 元素会保持住与锚点的相对位置。

在背景图片的下面创建一个 Image,HeadLogo_img,子物体的锚点都是相对于在物体的，在父物体的中心即可；聚焦图片的话，场景视图会切换到 UI 编辑视图，这个空间就代表着屏幕的位置，这边我们就在这个视图中移动 UI 元素，然后在 Game 视图中查看显示效果。HeadLogoBK_img 的 Image 组件的 SourceImage 载入背景图片(那张白色的矩形框)，HeadLogo_img 载入角色头像的 Logo，背景图片 color 调整:（226.88.86），场景中的图片的颜色是这个 color 颜色和原来的图片颜色相乘的结果，所有我这里背景图片准备的都是白色的。
ImageType：simple（默认）,sliced（切边的方式，在 sprite editor 中对图片有个绿色的边框可以调整来对图片进行切边，可用在圆角矩阵的图片，将各个圆角和其他部分切割开来，这样图片在进行缩放拉伸的时候，可以保持住这个圆角不拉伸，与前面的 simple 相对比），filled（填充模式，有个 filled amount 可以控制对图片进行裁切，Fill Method 可以选择裁切的方式，可以用于我们的血条的制作）。

血条的制作：选中 PlayerCanvas,右键 UI->Image,命名为 healthBar_img,调整大小：320*20，再复制出三个，分别命名为 healthBarBK_img,staminaBarBK_img 和 staminaBarBK_img 选中长条的背景图片，调整相应的位置，healthBar_img 和 healthBarBK_img 要重合再一起，图层的顺序保证 healthBarBK_img 在 healthBar_img 的上面，UI 似乎没有 sorting layer 的选择，顺序排在上面的先渲染，拍下下面的后渲染，下面的覆盖上面的。healthBar_img 的 color 选成红色，ImageType 选成 filled,file Method :Horizontal,fill origin:Left,fillAmount:0.5;healthBar_img 的 color 选成蓝色，其他设置与 healthBar 一样；
选中 PlayerCanvas，UI->Image:Coin_img,载入金币的图片，调整大小 60*60，锚点也设置在屏幕的右上角，在 Coin_img 下面添加一个 Text 的物体，命名为 CoinCount_txt，Project/Assets 中新建一个文件夹 Fonts，载入参考教程中的字体，slkscr.Text 组件进行参数设置：Text:00;Font:slkscr;
Font Style:Bold;Font Size:38.Alginment 都选为居中（都点中中间的那个标志）。

新建一个组件脚本 PlayerUI.cs，挂在 PlayerCanvas 的游戏对象上。脚本的话，主要是将 UI 中的 healthBar 和 staminaBar 的进度条中 fillAmount 分别受到 player 的血条值和能量值控制，UI 中的文本受到金币数量的控制。
需要这几个成员变量：
Public Item coinItem;//金币的 scriptable object 的数据
Private Player player;//获取角色的血量和能量。
Public Image healthMeter;
Public Image staminaMeter;//获取 UI 中的 fillAmount 的值,
Public Text coinText; //记录金币数量

Start 函数：

```
Player = GameObject.FindGameObjectWithTag(“Player”).GetComponent<Player>();
```

//GameObject 中有好几个 Find 一类的函数，似乎都是从整个场景中寻找的，若是要寻找某个游戏对象的子物体的话，可以使用 transform.Find 的函数。

Update 函数：

```
If(player != null)
{
healthMeter.fillAmount = player.healthPoints/maxHealthPoints;
staminaMeter.fillAmount = player.staminaPoints/maxStamina;
coinText.text =””+coinItem.quantity;
}else
{
Player = GameObject.FindGameObjectWithTag(“Player”).GetComponent<Player>();
}
```

在 inspector 参数面板上分别给 Public 类型的成员变量挂上各自的引用对象，测试。

可以将这个 PlayerCanvas 做成预制件 prefab，可以用在不同的关卡，也可以先把场景中 PlayerCanvas 删掉，然后在角色加载的时候 instantiate 实例化，这里值得注意的是，instantiate 这个游戏对象中（不包括子物体的）的任一个组件，都会将这个游戏对象给实例化出来。

制作一个简易的背包系统：

游戏中准备做一个简易的背包系统，直接将背包的槽(slot)显示在屏幕的左上角。这边会将道具中的血瓶（red bottle）,能量瓶（blue bottle）,火焰投掷物（fire ball）和水球投掷物（water ball）这四个道具，所以这边就准备 4 个背包的槽显示在屏幕的左上角。这里可以另外创建一个 canvas，命名为 InventoryObject,选中 canvas,鼠标右键 create empty,取名 BackGround,在 BackGround 的对象上添加 Horizontal Layout Group 组件，这个组件是可以排序放在它后面(作为子物体)的背包的的槽(slot)，Spacing 可以调整背包间距。移动这个 BackGround，让他放置在屏幕的左上角，锚点也选在左上角。

每一个背包系统中的每一个槽的 UI 显示：BackGround 下创建空对象，命名为 Slot，然后在 Slot 下面放置一张 image,命名为 Background_img,调整大小 150*150,source 下面载入一张 UI 的圆角矩阵的背景图，color 的颜色调成为深蓝色，color(37.57.183);在背景图下在放置一样显示道具贴图的 image,调整大小 120*120,先 enable(false),就是勾调 Image 组件前面的勾；同时在背景图下增加另一张 Image,命名为 Tray_img，调整大小 80*50，放在背景图的右下角，在其下面放置一个 Text 的游戏物体，取名为 Qty_txt，使用之前的字体，调整大小，居中，颜色改为白色，text 的默认值为空，将其放置在 Tray_img 图片的中间。slot 下增加一个 Button，调整尺寸与背景图片大小一致即可，其对应的 Image 组件中的 Color 的那个 Alpha 参数要设置为 0，使得这个按钮变为透明，去掉它的 text 组件。

显示在 slot 中的 Item 中的道具贴图，需要重新到场景物体那种精灵图片中去割取，原来的

是按照道具的 Alpha 通道的边界来割取每个贴图的，显示在 slot 的话，显示道具图像会发生形变。选中 Sprites/GameEnv/SpriteObject,点开 Sprite Editor 编辑器，只需要在道具的图像周围画出一个框来就可以了，框的大小为 16*16，为各自的道具再起个名字即可。都割取完之后，点击 Apply 退出，然后对应的道具的 Scriptable object 的 sprite 属性都换成这一组精灵片。

背包系统的功能有：
主角碰到场景中的血瓶（red bottle）,能量瓶（blue bottle）,火焰投掷物（fire ball）和水球投掷物（water ball）这四个道具，会显示在相应的背包系统的槽的上面，对于血瓶（red bottle）,能量瓶（blue bottle）（这两个 stackable 变量都是设为 true 的）会在右下角的数字显示栏显示相关的道具数量，对于火焰投掷物（fire ball）和水球投掷物（water ball）右下角的数量为空；
用鼠标点击对应的槽：点击空的槽的话，没有任何反应，点击的显示火焰投掷物（fire ball）和水球投掷物（water ball）的槽，此时会变化场景中主角的投掷物，点击后这个槽就显示为空。点击的显示血瓶（red bottle）,能量瓶（blue bottle）的槽的话，他们右下角显示的数字会减少 1，此外场景中角色的血量和能量会相应的增加，减到 0 的话这个槽就显示为空。

创建一个 Slot.cs 的组件类，里面设置两个成员变量：
Public Image itemImage;
Public Text qtyText; //指向 slot 中的 qty_txt
Public Button slotBtn;
将这个组件挂在 slot 的游戏对象上。

创建一个 Inventory.cs 的脚本，将其挂在 InventoryObject 上。代码如下：
Public GameObject slotPrefab;//将 slot 都做成了预制件
Public const int numSlots =4;//槽的数量在运行前就定好了
GameObject[] slots= new GameObject[numSlots];

Image[] itemImages = new Image[numSlots];//这里用数组来简单化了，也可以用 list<>列表的
Item[] items = new Item[numSlots];
Button[] slotBtn = new Button[numSlots];
Text[] slotTxts = new Text[numSlots];
设置一个
Private void createSlots()函数，让其在程序运行时在屏幕的生成 4 个槽
{
if(slotPrefab != null)
{
For(int i=0;i<numSlots;i++)
{
GameObject newSlot = Instantiate(slotPrefab);
newSlot.name = “ItemSlot_”+ i;//在 hierachy 上显示的名字
newSlot.transform.SetParent(gameObject.transform.GetChild(0).transform)；
//transform 组件掌管着游戏对象间的父子关系，这句话就是将 newSlot 这个槽物体作为
//gameObject(这个组件挂在 inventoryObject 上，那么这里 gameObject 指代的就是这个
//inventoryObject)，gameObject.transform.GetChild(0)指的是 inventoryObject 的第一个子物体

//Background,这句代码的意思就是将实例化的 slot 作为 Background 的子物体；
Slot[i] = newSlot;
itemImages[i]= newSlot.GetComponent<Slot>().itemImage;
slotBtns[i] = newSlot.GetComponent<Slot>().slotBtn;
slotTxts[i] = newSlot.GetComponent<Slot>().qtyText;
Int temp = i;//需要这个临时变量 temp 存一下这个变量值 i,
slotBtn[i].onClick.AddListener(delegate()
{
DropItem(temp);//这个的参数不能直接写 i,i 这个数值传不进去？？？
});
}
}
}
这个 createSlot 函数在 Start()函数内调用
增加一个 public void AddItem(Item itemToAdd)的函数
{
If(int i=0;i<items.Length;i++)
{
//需要先遍历一遍是否有这个 item,若这个道具 slot 已经填上了道具的话，并且又接触到这个道具的话，那么就判断这个道具是不是 stackable，是的话就 quantity 加 1， text 框的数字相应的增加。
If(items[i]!=null && items[i].itemType == itemToAdd.itemType)
{
if(itemToAdd.stackable == true)
{
Items[i].quantity = items[i].quantity +1;
slotTxts[i].text = items[i].quantity.ToString();
Return;
}else
{
Return;
}
}
//如果没有的话，再遍历一遍整个槽，当遇到第一个 items[i]还没有填上道具显示的时候，这里就需要进行显示，增加 sprite 图像的显示，若是 stackable 物体，txt 标签需要显示 item.quantity 的数量。不是 stackable 物体可以不显示。
For(int i=0;i<items.Length;i++)
{
If(items[i] == null)
{
Items[i]=Instantiate(itemToAdd);//这里需要修改 item 的 scriptable object 的数值，可以复制一份出来
Items[i]. quantity +=1;
itemImages[i].sprite = itemToAdd.sprite;

```
itemImages[i].enabled = true;
If(itemToAdd.stackable)
{
slotTxts[i].text = items[i].quantity.ToString();
}
Return;
}
}
Return;
}
```

这段代码 AddItem 函数需要在主角碰到道具的时候调用，打开 Player.cs 的组件代码，
Player.cs 中需要增加 Inventory 的成员变量：
Public Inventory playerInventory;然后在 OnEnterTrigger2D 中 switch 语句中的
Case Item.ItemType.HEALTH
Case Item.ItemType.STAMINA
Case Item.ItemType.FIREBALL
Case Item.ItemType.WATERBALL 中调用
playerInventory.AddItem()这个函数；

我们点击槽 slot，如果是空的槽就没有任何反应，如果点击血瓶或者能量瓶，那么相应的主角的生命值和能量就会增加，同时 quantity 的数量减少 1，或减少到等于小于 0 的话，那么此时这个 slot 的 items[index]就设置为空,
itemImages[index].sprite=null,itemImages[index].enable=false 图片为空
字符也为空；
若是点击的是水球投掷物或者火焰投掷物的话，此时场景的主角的相关的武器也会发生变化，item[index]设置为空，itemImages[index].sprite=null 为空，itemImages[index].enabled=false;
首先需要成员变量中增加一个 Player player 的成员变量，在 start 函数上用 findGameObjectWithTag 来获取；
增加要给 public void DropItem(int index)的函数

```
//index 表示了点中的是第几个槽
{
If(items[index]==null)//这个槽是空的话，直接 return
{return;}
Else
{
If(items[index].stackable == true)
{
Items[index].quantity-=1;
Switch(items[index].itemType)
{
Case Item.ItemType.HEALTH:
Player.AdjustHealthPoints(items[index].amount);//这两个函数在 player.cs 中都是设置成 public
Break;
```

```
Case Item.ItemType.STAMINA:
Player.AdjustStaminaPoints(items[index].amount);
Break;
Default:
Break;
}
If(items[index].quantity<=0)
{
Items[index]=null;
itemImages[index].sprite=null;
slotTxts[index].text = “”;

}else
{
slotTxts[index].text = items[i].quantity.ToString();

}
}else
{
Switch(items[index].itemType)
{
Case Item.ItemType.FIREBALL:
暂时 Debug.Log();
Break;
Case Item.ItemType.WATERBALL:
暂时 Debug.Log();
Break;
Default:
Break;
}
Items[index] = null;
itemImages[index].sprite=null;
itemImages[index].enabled = false;
}
}
```

单例类 RPGGameManager 的创建：
单例(Singletons,单例模式，一种软件设计模式),软件设计模式（software design pattern）：可以认为是前人在写软件代码的时候总结下来的通用的，可重复使用的一些解决方案或者设计框架。有很多设计模式，需要平时的积累与学习。

Manager 类通常是对游戏（一般都是指某个场景或者某个关卡）的一些逻辑从整体上进行管理和把控，比如进入游戏时，可以指定我们的主角在那个地方出生,让游戏退出之类的等等；

一般情况下，这种类都是采用单例模式的。

单例就是设计模式的一种，单例模式表示在整个程序的运行过程中这个类只有一个实例（即一个对象），并且需要提供一个访问它的全局访问点。

一般的单例模式需要满足的条件：1.保证类的构造函数是私有的；2.让类自身保存它的唯一实例，类内创建一个该类的 private 静态变量 instance，引用指向这个实例；3.提供一个全局的访问该实例的静态方法 GetInstance()，一般都在这个静态方法内实例化的;

Unity 中的单例：unity 中 Mananger 类一般也是继承自 monobehaviour 的类（即可以挂在到游戏对象上的类），这些类的构造函数都是系统在调用的，我们没法私有化。要让这个 Manager 类有单例的性质，类内我们需要可以声明一个这个类的全局静态变量，在系统调用 Awake()（在 start()前调用，也只调用一次，静态变量能早点指向它的实例）函数的时候，将系统实例化出的这个类的实例对象引用给这个静态变量。我们这边只需要保证整个场景中只有一个包含该类的游戏对象即可。我觉的 unity 的单例指的是单个游戏对象，这就保证了这个游戏对象中的 Manager 类也是单例的。

场景中创建一个 new GameObject,命名为 RPGGameManager,在 Scripts 中创建 Manager 文件夹，专门放置一些管理类，然后创建 RPGGameManager.cs 的组件，
代码中设置一个该类的全局的静态变量，作为访问的入口点
Public static RPGGameManager sharedInstance=null;给这个类创建一个全局的静态变量；静态变量只能声明一次。
Awake()也是系统调用的，在 start()函数前调用的
可查看程序的那副流程图：
https://docs.unity3d.com/Manual/ExecutionOrder.html

Awake 函数中 //防止生成第二个带有这个组件的 gameObject
```
If(sharedInstance !=null && sharedInstance !=this)
{
Destroy(gameObject);
}else
{
sharedInstance = this;
}
```
将这个管理类拖入到 Prefab 文件夹中做成预制件。

敌人 Enemy 的制作：
先在场景中创建敌人游戏对象的基本形态：
在 hierarchy->CreateEmpty,命名为 EnemyObject,将素材中的 idle 动画状态的序列帧选中托给 EnemyObject.保存动画片段和动画控制器，并放在相应的文件夹内，将 walk 动画片段也进行保存。Sorting layer 设置为 character;EnemyObject 游戏对象上挂上 boxCollider 碰撞器组件，调整碰撞器的大小来贴合敌人角色；挂上 rigidBody2D 组件，重力调整为 0，Constrains 勾上 Freeze Rotation 的 Z,Linear Drag 这个阻力参数可以增加带你数值 1。Tag 标签增加一个 Enemy,

将 EnemyObject 选为 Enemy.Layer 层也选为 Enemy;

创建一个 Enemy.cs 的自定义脚本，让它也继承自 Character；
Character 中增加两个方法，这两个方法都设置为 virtual 让 player 和 enemy 都能调用。

```
virtual public void CharacterDie()//角色死亡
{
 Destroy（gameObject）;
}
virtual public void TakeDamage(float damageAmount) //角色受伤减血
{
      healthPoints -= damageAmount;
      If(healthPoints<=float.Epsilon)
      {
            CharacterDie();
      }
}
```

游戏中我们设定当敌人角色碰到玩家时就让玩家受伤：
所以这边在敌人的 Enemy .cs 的脚本中添加成员变量：

```
Public float damageStrength=10.0f;
```

然后在 OnCollisionEnter2D()这个函数与 OnTriggerEnter2D()用法一致，这边敌人角色的碰撞没有勾上 IsTrigger,这边将 OnCollisionEnter2D()这个函数写在敌人这边的脚本中；

```
Void OnCollisionEnter2D(Collision2D collision)
{
If(collision.gameObject.CompareTag(“Player”))
{
Player player = collision.gameObject.GetComponent<Player>();
Player.TakeDamage(damageStrength);
}
}
```

优化一下，Enemy 角色对 Player 玩家角色的伤害：
OnCollisionEnter2D 也是碰撞后函数会调用一次，我们这边想让我们的角色能受到持续性的伤害，相当于碰撞后，进入一个跟游戏同步的 while 循环，有一个持续性的每隔一段时间的伤害；可以引入的 Coroutine 协程来实现这个效果；

Coroutine 底层原理参考：https://www.cnblogs.com/yespi/p/9847533.html

一般的代码写了一个 for 循环或者 while 循环的话，就是一帧内将这个循环执行完毕的。但是不是，协程的表现行为像多线程，协程中的 while 语句（这个 while 里的代码要符合协程的语法规则）可以与程序的那个最外层的一层 while 同时进行运行；但它内部其实是将协程中的 while 的每一次循环都是合并到程序的大的循环中的去的。

在前面的 C#课程中提到：Coroutine 协程的本质就是将迭代器的每一项的执行都分配到程序的循环帧中去了，实现了那种分帧执行的效果。

协程代码中的每一个 yield return 就是迭代器中的一项，每遇到这个 yield return，协程中的代码在这一帧就不往下执行，程序挂起，等到下一帧或者等待多少秒（这是由 yield return 后面的表达式决定的）再执行。Yield return 后面常跟的表达式有：
Yield return null 或者 1，“sss”这类的;那么 yield return 后面的语句在下一帧执行
Yield return new WaitForSeconds(waitTime):等待 waitTime 的时间后，再执行 yield return 后面的语句。
Yield return StartCoroutine(AnotherCoroutine()):或者另外的一个协程执行完后再回来执行 yield return 后面的语句
.....
不过我们一般关心的还是 yield return 前的那些语句的执行。

协程常用到的地方：需要多帧执行的行为，比如移动到某个地方；某个变量的值改变后，需要等一段时间在让它变回来，比如受伤闪一下；.....

Coroutine 协程的调用方法：写一个返回值为 IEnumerator 的函数，函数体内有 yield return 的语句，用 StartCoroutine()调用这个函数，用 StopCoroutine()结束这个协程；StartCoroutine()函数调用后会返回一个 Coroutine 类型的引用，这个控制器作为 StopCoroutine 函数的参数可以停止这个协程。

将我们角色的受伤用协程协成可以持续受伤的代码：
Character 的代码 TakeDamage 修改为引入协程后的代码：
virtual public IEnumerator TakeDamage(float damageAmount,float interval)
参数：float interval 表示隔一段再受到伤害,当 interval 是 0 的话，就表示受到一次伤害。
```
{
While(true)
{
healthPoints -=damageAmount;
if(healthPoints<=float.Epsilon)
{
CharacterDie();
Break;
}
If(interval >float.Epsilon)
{
Yield return new WaitForSeconds(interval);
}else
{
Break;
}
}
}
```
敌人 enemy.cs 代码中需要修改：
增加一个 Coroutine damageCoroutine;

和 float damageInterval = 1.0f;
与 OnCollisionEnter2D 相对应的还有一个 OnCollisionExit2D 的函数，表示当角色和敌人进入碰撞然后又退出碰撞后将执行的事件。这边 Exit 的代码中我们会停止这个 damageCoroutine;

```
OnCollisionEnter2D
{
//获取 player 组件后，调用 TakeDamage 的协程
If(damageCoroutine == null)
{
damageCoroutine = StartCoroutine(player.DamageCharacter(damageStrength,1.0f));
}

}
OnCollisionExit2D(Collsion2D collision)
{
If(collision.gameObject.CompareTag("Player"))
{
If(damageCoroutine != null)
{
StopCoroutine(damageCoroutine);
damageCoroutine=null;
}
}
}
```

敌人场景漫游和追逐玩家：
这一块通常就是指 NPC（non player character）,非游戏玩家的人工智能，一般的 RPG 游戏的人工智能中，那些 NPC 会在场景中进行漫游，然后会自动躲避那些障碍物，就是就基本的寻路的功能，在漫游过程中发现了玩家角色会进行攻击。这一块的主要的技术点还是在寻路这一块，相关的方案有很多，网上也有很多相关的插件来实现寻路的功能。

游戏中的 Enemy 就是 NPC，游戏中我们会写一个 wander 的算法，让他有一个简易的漫游场景的功能，但不做复杂的寻路系统了，就是没有躲避障碍物的功能。

游戏中敌人的场景漫游和追逐玩家的方案：敌人角色在游戏场景中，wander 算法会每隔几秒在敌人的附近随机找一个点，让敌人移动过去，然后在敌人移动到这个点静止等待下一次的漫游的调用，一般情况下，移动到目标点的时间都是要小于这个 wander 算法的调用时间的。敌人角色会有一个触发的 Collider，当我们的玩家角色触碰到这个 Collider 的时候，敌人就将我们的玩家角色的位置当做移动的目标点，然后向玩家角色移动过来，进而碰到玩家角色而造成伤害；当我们的玩家角色出了敌人的触碰框的话，敌人角色进行场景漫游。

因为敌人角色没有躲避障碍物自动寻路的功能，那么这就要求我们的环境碰撞物体要尽量的少，或者将这些环境碰撞物体与敌人没有那种阻碍碰撞的效果。

敌人漫游的实现方法主要方法：
需要间隔某个时间段的调用某个函数：主要有几下几种方法：
1. 协程中的 yield return new WaitForSeconds(interval);
2. 设置一个时间基准点，初始值为 0，在 update 中累加这个时间基准点，然后这个时间超过调用间隔时间 interval 的话就执行对应的代码，并且将这个时间基准点再次设置为 0；
3，使用 invokeRepeating()函数；

此外，敌人朝着目标点移动也不是一帧完成的，是有一个过程的，这个也可以用 Coroutine 的方法，只要协程函数中用 yield return null 之类的，就可以实现每帧执行的效果。或者将移动代码也可以写在 update 函数也可以。

教程用都用 Coroutine 协程的方法。

Enemy 漫游的脚本 Wander.cs
Public float wanderIntervalTime = 3.0f;
Public float wanderSpeed = 2.0f;
Private Coroutine moveCoroutine;
Private Vector endPointPosition;
Private Rigidbody2D rb2D;
Start()函数中：
Rb2D = gameObject.GetComponent<Rigidbody2D>();
StartCoroutine(WanderCoroutine());
Update 函数中：
Debug.DrawLine(transform.position,endPointPosition,Color.red)，视图中可视化移动到的目标点，需要在 Game 视图中按下 Gizmos 的按钮；
需要创建的函数：

```
IEnumerator WanderCoroutine()
{
While(true)
{
ChooseNewEndPoint();
If(moveCoroutine !=null)
{
StopCoroutine(moveCoroutine);
moveCoroutine= StartCoroutine(MoveCoroutine());
Yield return new WaitForSeconds(wanderIntervalTime);
}
}
}
Void ChooseNewEndPoint()
{
Float wanderAngle = Random.Range(0,360);
Float wanderRadius = Random.Range(2,5);
```

```
wanderAngle = wanderAngle * Mathf.Deg2Rad;//180->pi
endPointPosition= rb2D.position + new
Vector2(Mathf.Cos(wanderAngle),Mathf.Sin(wanderAngle))*wanderRadius ;
}

IEnumerator MoveCoroutine()
{
Float remainning Distance =(rb2d.position-endPointPosition).sqrMagnitude;
While(remainingDistance>float.Epsilon)
{
Vector2 newPosition =
Vector2.MoveTowards(rb2D.position,endPointPosition,wanderSpeed*Time.fixedDeltaTime);
Rb2D.MovePosition(newPosition);
remainning Distance =(rb2d.position-endPointPosition).sqrMagnitude;
Yield return new WaitForFixedUpdate();
}
}
```

为敌人角色添加 idle 动画和 walk 动画的切换，Assets/Animation/Controller 双击 EnemyAnimCtrl 进入 Enemy 角色的动画控制器；Parameters 参数增加 bool isWalking,面板中 make transition idle->walk:isWalking = true;walk->idle:isWalking= false;切换参数的面板上 Has Exit Time 的勾去掉，Fixed Duration 改为 0；

Wander.cs 脚本中增加 Animator anim;
Start 函数中
Anim = gameObject.GetComponent<Animator>();
在 MoveCoroutine()的 while(remainingDistance>float.Epsilon)函数内
加上：anim.SetBool("isWalking",true)
在 while 外
加上：anim.SetBool("isWalking",false)

Enemy 角色添加一个 CircleCollider2D 组件，勾上 Is Trigger,作为感知角色的触发器，调整 circle 的 radius 半径值，表示敌人感知的感知范围；
可以在程序中添加 CircleCollider2D 这个变量：将这个感知区域可视化

```
CircleCollider2D circleCollider;
```

在 Start()函数中进行获取：

```
circleCollider = gameObject.GetComponent<CircleCollider2D>();
```

调用

```
Void OnDrawGizmos()
{
If(circleCollider != null)
{
Gizmos.DrawWireSphere(transform.position,circleCollider.radius)
}
}
```

只要我们的玩家进入的敌人的感知区域，敌人就向我们跑过来，这里面其实就将那个敌人移动的目标 endPointPosition 改为角色的位置即可；
代码中加入一个 Transform playerTransform = null 变量即可。
在 MoveCoroutine 的 while 循环中，在 Vector2 newPosition=的代码前加上如下代码即可：

```
if(playerTransform!=null)
{
endPointPosition = playerTeansform.position;
}
```

在代码中加入 OnTriggerEnter2D()和 OnTriggerExit2D()的事件，前一个是给 playerTransform 赋值，后一个是给 playerTransform = null;

```
OnTriggerEnter2D(Collider2D collision)
{
If(collision.gameObject.CompareTag(“Player”))
{
playerTransform = collision.gameObject.GetComponent<Transform>();
If(moveCoroutine != null)
{
StopCoroutine(moveCoroutine);
}
moveCoroutine =StartCoroutine(MoveCoroutine());
}
}
```

这边是当敌人立即感知到玩家角色的话就向玩家跑过去了。
当玩家角色离开了敌人的感知区域的话，就让 playerTransform = null,此时敌人应该在静止状态等待漫游

```
onTriggerExit2D(collider2D collision)
{
If(collision.gameObject.CompareTag(“Player”))
{
Anim.SetBool(“isWalking”,false);
playerTransform=null;
If(moveCoroutine!=null)
{
StopCoroutine(moveCoroutine);
}
}
}
```

优化：可以增加一个变量：public bool followPlayer；可以设置 NPC 是否会追逐玩家，可以用来设置其他的只有漫游功能的 NPC。这个变量指默认为 true,将其加在 triggerEnter 条件的判断中&&followPlayer
还有就是将敌人漫游的速度和追逐玩家的速度设置的不一样，player 的速度是 3
我们将敌人漫游的速度设为 1.5，追逐的速度设置为 2.5
增加一个 public float pursuitSpeed =2.5f;原来的 wanderSpeed = 2;
再增加一个 private float currentSpeed

然后在 MoveCoroutine 的 while 循环内
的开头加上 currentSpeed = wanderSpeed;
在 if(playerTransform != null)内加上 currentSpeed = pursuitSpeed;
原来的 MoveTowards 中的 wanderSpeed 改为 currentSpeed,可以在 ontriggerExit 中也加上 currentSpeed = wanderSpeed

场景中敌人角色的生成：
我希望在场景中随机的分布一些生成点，然后能让我们的敌人角色能隔一段时间在这个生成点上生成一个。此外我们也可以控制敌人的大致数量。我们会写一个生成的脚本让我们的敌人角色能在某个位置，隔一段时间进行生成。

先将敌人角色拖入 prefab 文件夹做成预制件，Hierarchy->Create empty ,命名为 SpawnPointObject,因为没有模型或者精灵片，所以这个 SpawnPointObject 在场景中是没有形态的，我们可以点击他在 Inspector 最右上角的标签显示，选择任意一个颜色的标签即可，场景视图中就显示出这个标签，方便开发者查看。
创建一个 SpawnPoint.cs 的脚本：代码如下：

```
Public GameObject prefabToSpawn;//表示要 spawn 出来的游戏对象
Public float repeatInterval;
Public float delayTime;
Start 函数中：//invokeRepeating 调用的方法似乎不能有参数。
InvokeRepeating(“SpawnObject”,delayTime,repeatInterval);
}
Public GameObjectSpawnObject()
{
{if(prefabToSpawn !=null)
{
GameObject prefabSpawn =
   Instantiate(prefabToSpawn,transform.position,Quaternion.identity)
Return prefabSpawn;
}
}
}
```

我们可以控制要生成的 prefab 的数量，在代码中加入

```
Public int numToSpawn=1;//需要生成的 prefab 的数量
Private int numCount =0;//用于在场景中计数
```

在 SpawnObject 的 instantiate 函数下面写上

```
numCount++;
If(numCount>=numToSpawn)
{
CancelInvoke(“SpawnObject”);
}
```

主角的反击：

我们的主角是通过向敌人抛掷子弹来对敌人进行反击的。玩家用鼠标点击游戏屏幕，然后游戏中主角朝着这个方向来投掷一个子弹。

还是重新制作一组武器的预制件，原来的那一组的投掷物 CommonBallObject FireBallObject,WaterBallObject,将它们的预制件的 layer 都该为 Pickup ，只用作在场景中作为 PickUp 让我们的主角拾取。
另外做一组投掷物的预制件 CommonAttackObject,FireAttackObject 和 WaterAttackObject 让我们的主角进行投掷攻击敌人；只要原来的那一组复制一下，然后 layer 选为 Ammo 即可，后续会在这组 prefab 上增加其他的自定义组件。

我们的投掷物为挂上一个自定义的 Ammo.cs，负责碰到敌人时会对敌人产生伤害。

玩家的游戏角色需要一个武器组件：Weapon.cs,这个武器组件的主要功能是：生成我们的玩家的投掷物，朝着敌人的方向投掷。攻击方案：我们用鼠标左键点击屏幕上的点，然后让投射物沿着这个点的方向飞行，我们可以快速点击鼠标，然后瞬时产生大量的投掷物飞射过去。

这就需要为我们的投掷物创建一个 Object pool(对象池)，一般的投掷物我们都是打到敌人后，直接对这个投掷物进行 Destroy(gameObject)即可，然后系统的垃圾回收系统会在后台清理这些内存。Destroy 本身是很占有系统资源的，我们可能会快速抛出很多投掷物，那么此时系统后台会消耗很多的性能来清理那些内存，从而造成我们的游戏出现卡顿。

Object pool 的方法是我们预先在内存中保存一定数量的投掷物，可以保存在一个集合中，且都是在 deactive 的状态下(处于这一状态在场景中不显示，也不于场景中的其他物体进行交互)，当我们需要使用的时候就 Active 激活它，打到敌人后再 deactive，中间都没有 Destroy() 这种消耗游戏性能命令。如果玩家疯狂点击鼠标，即疯狂发射投掷物的话，就只需将集合中的投掷物都激活即可，但是如果集合中的都使用完，那么此时再按鼠标就不会出现投掷物，等有投掷物 deactive 才能发出新的投掷物。这个与事先制定好 object pool 的 size 的大小有关；

Weapon.cs 中需要事先保存一个投掷物的 object pool，代码中用 List<>保存即可，list 动态是动态数组：
代码如下：ammoPool 设置成静态变量，让这个 ammo 尽快的内存中存在

```
Public GameObject ammoPrefab;
Static List<GameObject> ammoPool ;
Public int poolSize=5;
Void Awake()
{
If(ammoPool ==null)
{
ammoPool = new List<GameObject>();
}
For(int i=0;i<poolSize;i++)
{
    GameObject ammoObject = Instantiate(ammoPrefab);
    ammoObject.SetActive(false);
```

```
        ammoPool.Add(ammoObject);
}
}
```
在 update()函数中，需要写上点击鼠标左键抛出投掷物的逻辑：
```
if(Input.GetMouseButtonDown(0)) // 0:鼠标左键，1：鼠标右键
{
FireAmmo();
}
```
FireAmmo 函数中，我们代码的相关逻辑是：

1. 获取鼠标点击的位置，需要从画面的像素坐标转化到 3D 空间中的坐标

```
Vector2 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
```

2. 根据这个鼠标位置计算出投掷物的运动方向

```
Vector2 direction = mousePosition - new Vector2(transform.position.x,transform.position.y);
Float angle = Mathf.Atan2(direction.y,direction.x)*Mathf.Rad2Deg;
```
反正切函数，这里返回的 -180~180 的角度
```
Quaternion rotation = Quaternion.AngleAxis(angle-90,Vector2.forward);
```
Vector2.forward 是（0，0，1）表示沿着 Z 轴进行旋转，发现上面的角度要减去 90，可能需要的角度是与 Y 轴的夹角。

3. 需要一个 GameObject SpawnAmmo(Vector3 location,Quaternion rotation)的函数，表示从 object pool 中获取一个物体,位置是 location，角度方向是 rotation

```
{
Foreach(GameObject ammo in ammoPool)
{
If(ammo.activeSelf==false)
{
Ammo.SetActive(true);
Ammo.transform.position = location;
Ammo.transform.rotation = rotation;
Return ammo;
}
}
Return null;
}
```
然后在 FireAmmo()进行调用。

给我们的投掷物添加一个自定义的脚本 Ammo.cs,是我们的投射物的运动；

代码如下：
```
public float ammoSpeed = 8.0f;
Public float ammoLifeTime = 3.0f;
Private float timeBase = 0.0f;
```
Update 函数中
```
Transform.Translate(Vetctor2.up*ammoSpeed*Time.deltaTime);//Translate 默认移动的是它的局部坐标系，local
timeBase += Time.deltaTime;
If(timeBase > ammoLifeTime)
```

{
gameObject.SetActive(false);
}
Void OnEnable()
{
timeBase = 0.0f;
}
这个 OnEnable 函数，系统会在游戏对象进行 SetActive(true)的时候调用；

投射物对敌人造成伤害：
这边的投掷物的碰撞框都选为了 IsTrigger 的，
这边就可以调用 OnTriggerEnter2D 的函数
内部代码：

```
if (collision.gameObject.CompareTag("Enemy"))
{
    if (collision is BoxCollider2D)
    {
        Enemy enemy = collision.gameObject.GetComponent<Enemy>();
        StartCoroutine(enemy.TakeDamage(ammoItem.amount, 0.0f));
        gameObject.SetActive(false);
    }
}
```

这里需要获取 ammo 的 scriptable Object 中的伤害值数据，所以代码中需要增加一个 private Item ammoItem 的成员变量。
Start 函数中：

```
ammoItem = gameObject.GetComponent<Pickup>().item;
```

TakeDamage 的第二个参数设为 0 表示受到一次伤害。

这边就可以因为敌人有两个碰撞器，一个碰撞器是 circle 的，一个是 Box，这里就可以根据碰撞器的类型去选择碰到了那个碰撞器。这个是判断对方的碰撞器，可以通过传入进来的 Collider 类型来碰撞到了是哪一个，不过这边好像各个碰撞器没有什么标签什么的，通过一些碰撞器的参数来设置。

假如我们的一个游戏对象挂上很多的碰撞器，每一个游戏对象都会有各自有一个 OnCollision 和 OnTrigger 的事件调用，似乎在自己的对象中没法判断对方物体碰撞到了自己的哪一个碰撞体；这边似乎只能在游戏对象的下面挂上子物体，然后将碰撞器加在这个子物体上，因为子物体也是一个游戏对象，就可以根据游戏对象角色来判断跟哪个碰撞器相碰。

优化：这边点击到背包系统的槽 slot 的时候，也会发射投掷物，这不是我们想要的效果，我们希望，这边需要判断点击到的是我们的 inventory。
可以直接在点击的时候，FireAmmo()外面判断下即可，系统给了我们一个很简单的函数调用：
If(!EventSystem.current.IsPointerOverGameObject())
{

FireAmmo();
}
调用这个函数需要引入 using UnityEngine.EventSystems 的命名空间，一般的话，鼠标点击判断检测到场景中的某个物体的话可以用 RayCast 碰撞检测来实现，不过这边系统对 UI 的检测已经包装好了一个函数，我们只需要简单的调用一下即可。

将我们主角的能量值与投掷物的射程挂钩：
每点击一次我们的 stamina 会减去 1，然后当前的 stamina/maxStamina 的比值会影响投射物的寿命。
Ammo 的脚本中增加一个 public float maxAmmoLifeTime=5.0f 的变量
和 lifeWeight 的权重值：public float lifeWeight=1.0f;这个权重变量需要加上[HideInInspector]的特性，表示在 inspector 不可以，但是在类外可见。
将原来的 ammoLiftTime 变量的 public 改为 private;
Start()中：
ammoLiftTime = maxAmmoLifeTime；
OnEnable 中增加：
ammoLiftTime=maxAmmoLifeTime*lifeWeight;

Weapon.cs 增加一个 Player 成员变量：
Private Player player;
Player =gameObject.GetComponent<Player>();
FireAmmo 的函数中写上：
Player.staminaPoints--;
If(Player.staminaPoints<=10.0f)
{
Player.staminaPoints =10.0f;
};//还是让它有一个最小值，不然就没法进行攻击了。在 Player.cs 的脚本的调整生命值和能量值这块也调整下最大值。
在 SpawnAmmo()函数的 ammo.SetActive(true)前面，
增加这两行代码，来修改 ammo 的 lifeWeight 的值：

```
Ammo ammoScript = ammo.GetComponent<Ammo>();
ammoScript.lifeWeight = player.staminaPoints / player.maxStaminaPoints;
```

投掷物的切换的实现：
点击背包系统中的投掷物，让角色投掷的子弹发生变化。先将 Ammo.cs 这个脚本先挂在上其他的投掷物体上；
Weapon.cs 的脚本中
原来是一个 ammoPrefab，现在改为一个数组
Public const int ammoCount =3;
Public GameObject[] ammoPrefabs = new GameObject[ammoCount];
Awake() 函数中原来的 for 循环中，改为
GameObject ammoObject = Instantiate(ammoPrefabs[0]);
那么在 Inspector 面板中，务必将 CommonAttackAmmo 托给数组的第一个。
增加一个 public void ChangeAmmoPool(Item ammoItem)的函数

当我们点击 Inventory 上的投掷物的时候，会调用这个 ChangeAmmoPool()函数，并且将这个投射物 Item 的信息传递给他它。
代码如下：
先判断点击的投掷物在数组中是在哪一个位置：

```
Int index =0;
For(int i=0;i<ammoCount;i++)
{
Item ammoObjectItem = ammoPrefabs[i].GetComponent<Pickup>().item;
If(ammoObjectItem.itemType == ammoItem.itemType)
{
Index=i;
Break;
}
}
```

然后可以将原来的 ammoPool 中预制件先删除

```
For(int i=0;i<poolSize;i++)
{
Destroy(ammoPool[i]);
}
ammoPool.Clear();// 引用都置成空；
```

然后将新的投掷物加到 ammoPool 中去就可以了。

```
for (int i = 0; i < poolSize; i++)
{
    GameObject ammoObject = Instantiate(ammoPrefabs[index]);
    ammoObject.SetActive(false);
    ammoPool.Add(ammoObject);
}
```

其实这边可以做三条 ammoPool，这样可以事先将所有的武器的 object pool 都预制好，这样这里就不需要 destroy.但因为这里武器也不是很频繁的切换，这里我就这样写了，应该也没事。

在 Inventory.cs 的代码中：
将 DropItem()函数中的原来的关于 FIREBALL /WATERBALL 的那段 switch 给注释掉，
加上

```
Weapon weaponScript = player.GetComponent<Weapon>();
weaponScript.ChangeAmmoPool(item[index]);
```

增加角色的攻击动画：
现在角色抛掷投掷物的时候，身体没有相应的转向，也没有加入抛掷的动作，这就显得看上去非常的奇怪。这边需要加入各个朝向的攻击动作，然后根据不同的方位动画切换成不同方位的攻击动画。

Assets/Animation/Controller 双击角色的动画控制的 PlayerAnimCtrl，打开动画 Animator 面板。右键 create state ->From New Blend Tree,创建一个 blend Tree，重命名为 BT_Fire,先点击参数 parameters 增加几个参数 bool isFiring ,float fireXDir,float fireYDir;双击进入 BT_Fire 的 Blend Tree;Blend Type:2D Simple Directional;Parameters:fireXDir,fireYDir.Motion 中点击”+”选择 Add MotionField,将东南西北四个方法的攻击动画依次加入进去，posX 和 posY 分别设置为：1，0；0，-1；-1，0；0，1；选择每一个攻击动画的节点，将 Loop Time 的勾去掉，让动画只播放一次；

BT_Idle make transition ->BT_Fire ;BT_Fire make transition ->BT_Idle ;Idle 和 Fire 状态之间的转换连线。
BT_Walk make transition->BT_Fire, BT_Fire 到 BT_Walk 的切换就不设置了。这边想在角色攻击的时候不能移动。

BT_Idle->BT_Fire 的切换条件：isFiring :true;Has Exit Time 的勾去掉，Transition Duration:0
BT_Fire->BT_Idle 的切换条件：isFiring :true;Has Exit Time 的勾不要去掉，ExitTime:1;Transition Duration:0;这里是要播放完整个攻击动画一圈。
BT_Walk->BT_Fire 的切换条件：isFiring :true;Has Exit Time 的勾去掉，Transition Duration:0.

动画切换的主要部分是需要根据鼠标的按键，来判断角色是向那个方向攻击，我们这里就用之前的从那个鼠标点击的方向通过 Atan2 计算得到的夹角来设置对应的方向，这个夹角对应的范围是 0~180 和 0~-180，然后将其分为东南西北四个区域分别设置对应的 fireXDir,fireYDir。区域的划分以 45 度，-45 度，135 度，-135 度为界。

Weapon.cs 中加入对应的代码：
成员变量：
Private bool isFiring = false;
[HideInInspector]//inspector 面板不显示，在其他组件脚本中可调用这个参数。
Public bool beCanWalk= true;//设置攻击的时候让角色能暂停一会。
Private Animator anim;//控制动画切换需要有动画控制器
Private float angleDir = 0.0f； //将原来是局部变量的 angle（鼠标点击方向的夹角）提升为 Weapon 类的成员变量。
Start()函数中
Anim =gameObject.GetCoomponent<Animator>();
Update()函数中：
鼠标点击内：
isFiring = true;
bCanWalk= false;
然后在 update 中再加上一个 UpdateAnimState()用来控制动画的切换
先在原来的 FireAngle 内将原来的计算方向角度的那个局部变量 angle 改为成员变量 angleDir
UpdateAnimState()函数内：
If(isFiring)
{
Anim.SetBool(“isFiring”,true);

```
            if (angleDir <= 45 && angleDir >= -45))
              {//当 angleDir 是在 0~45 和 0~-45 之间是播放像东边攻击的动画的话，攻击完之之后
                    回到 Idle 状态也是朝向东面的，所以这里就一起设置影响 idle 的 xDir 和 yDir
                    这两个参数。
                  anim.SetFloat("fireXDir", 1.0f);
                  anim.SetFloat("fireYDir", 0.0f);
                  anim.SetFloat("xDir", 1.0f);
                  anim.SetFloat("yDir", 0.0f);
              }
              else if (angleDir < -45 && angleDir >= -135)
              {南面
                  anim.SetFloat("fireXDir", 0.0f);
                  anim.SetFloat("fireYDir", -1.0f);
                  anim.SetFloat("xDir", 0.0f);
                  anim.SetFloat("yDir", -1.0f);

              }
              else if ((angleDir < -135 && angleDir >= -180) || (angleDir > 135 && angleDir
<= 180))
              {
                  anim.SetFloat("fireXDir", -1.0f);
                  anim.SetFloat("fireYDir", 0.0f);
                  anim.SetFloat("xDir", -1.0f);
                  anim.SetFloat("yDir", 0.0f);
              }
              else
              {
                  anim.SetFloat("fireXDir", 0.0f);
                  anim.SetFloat("fireYDir", 1.0f);
                  anim.SetFloat("xDir", 0.0f);
                  anim.SetFloat("yDir", 1.0f);
              }

              isFiring= false; //这里紧接着将 isFiring 设置为 false;但是我们动画机那边的
isFiring 是在下一帧才能判断到 Anim.SetBool("isFiring",false);，所以动画状态机的动画会切换
到攻击动画。这里如果加上 Anim.SetBool("isFiring",false);就播放不出攻击动画了。

}else
{
Anim.SetBool("isFiring",false);
}
```

当攻击的时候我们不希望角色行走，所以我们使用了一个新的变量 bCanWalk，当鼠标按下时，bCanWalk= false 角色就设置角色不能被控制走动了。

在控制玩家输入行走的 PlayerMovementController.cs 加入一个 Weapon weapon 组件的成员变

量；
在 Start()函数内获取
Weapon = GetComponent<Weapon>();
然后在 MoveCharacter()的 rb2D.velocity= moveDir*moveSpeed 这句话的前面加上：
if(!(weapon.beCanWalk))
{
moveDir.x=0.0f;
moveDir.y = 0.0f;
}
这里不能用 weapon 中的 isFiring 变量，那个 isFiring 变量时在一帧之内由 true 变为了 false;
这个变量 beCanWalk 可以控制成过了大概多少时间后，beCanWalk=true，可以使用协程函数来实现这个效果。

```
IEnumerator SetCanWalk()
{
    yield return new WaitForSeconds(0.3f);
    beCanWalk = true;
}
```

这里进入协程函数后马上让他等个 0.3 秒后，再执行 beCanWalk.
在 Weapon.cs 的 UpdateAnimState()中的 isFiring = false 后面调用
StartCoroutine(SetCanWalk());

攻击和受伤的效果优化：
角色和敌人受伤的时候身体闪一下红色：
思路：主角和敌人的身上都有一个渲染组件 SpriteRenderer,将里面的 Color 的成员变量设置成红色即可，游戏画面中角色的显示的颜色是角色本身的精灵片乘以这个 color 的颜色，这边我们要的是闪一下的效果，那么此时我们可以用协程函数的功能，过某个时间后再将 color 变为原来的颜色，这边我们原来 color 的颜色都是白色。
代码如下，在 character.cs 的类中写一个 CharacterFlick()的协程函数，
IEnumerator CharacterFlick()
{
GetComponent<SpriteRender>().color = Color.red;
Yield return new WaitForSeconds(0.1f);
GetComponent<SpriteRender>().color = Color.white;
}
然后这个函数在 TakeDamage 的 while(true)循环里的第一句调用即可。

敌人和角色死亡后会在地面上留下一滩血迹：
绘制血迹的美术资源，将它放在 spriteObject.png 的大图中，重新导入那张大图，覆盖掉原来的图即可，选择大图，打开 spriteEditor 面板，鼠标拖出血迹图片的框即可，设置大小 32x32，命名为 blood Stain 即可；
场景中做一个 bloodStainObject 的游戏对象，然后做成 prefab，当角色死亡的时候生成。
Hierarchy 中 Create Empty，命名 bloodStainObject，加上 spriteRenderer 组件，并将血迹的精

灵片拖到 Sprite 参数中，调整血迹的大概的大小， sorting layer 选择 character,order in layer 设置为 -2；血迹的显示要在角色的下面。
Character.cs 的代码中，加入 public GameObject bloodStainPrefab;
然后在 CharacterDie()函数中，在 destroy 前加上如下代码：

```
if(bloodStainPrefab)
{
Instantiate(bloodStainPrefab,transform.position,Quaternion.identity);
}
```

在主角和敌人的预制件中的参数栏中拖入血迹的 prefab 即可。

粒子特效的添加，主要投掷物攻击到敌人时，有一个散开成烟花状的粒子特效生成
Hierarchy 中 Effects->Particle System，命名为 FireParticle
Inspector 中调整粒子的各模块的参数，调整前需要给模块打上勾：
Renderer 模块:Material :Sprites-Default;Sorting Layer ID:SpriteObject，Order in Layer 可以设置的大点，比如 5，主要是让粒子显示在最上面的那一层；
Texture Sheet Animation 模块:Mode:Sprites,然后选择一张粒子形态的精灵图片，
Shape 模块：Shape:Circle，然后视图中显示成了一条线，其实可以切换到 3D 视图，它是在 XZ 平面上进行扩散的。这里 transform 组件，rotation X =0 就可以了。调整下 circle 的半径，这边根据场景的大小，可以调节成 0.15 左右，表明粒子在这个形状内产生的。
基本设置中（Particle System 下的那些设置），Looping 的勾去掉，Duration 设置成 0.1，这个参数表示发射粒子的时间，这里就让它在一瞬间发射；
Emission 模块：Bursts 添加“+”，在 Count 一栏下拉选择 Random between Two Constants: 设置 40，60；这个参数表示粒子发射的数量。
基本参数 starting color 下拉选择 Random between Two Colors，这里选择火球的两个颜色，可用吸管工具在视图中火焰投掷物上吸取一下。
Starting lifeTime 选择 Random between Two Constants:0.5，1.5;这个参数表示粒子的寿命；
Start Speed 选择 Random between Two Constants:1.5,2.5;粒子的初始速度
Start Size 选择 Random between Two Constants:0.2,0.3;粒子的初始大小
Stop Action 选择 Destroy，表示粒子消失时将它销毁。
Size over Lifetime 模块，设置粒子的大小感觉它的生命时间会变小（出生时 size 很大，死亡时 size 很小），点击这里的 curve，选择一个随着时间变小的曲线即可。
Noise 模块：稍微设置下粒子运动的混乱度，strength:0.1

Hierarchy 中复制出一个 FireParticle，重名为 WaterParticle，将颜色改为水的蓝色即可；
Prefab 文件中新增 VFX 文件夹，将这两个粒子特效做成预制件，在场景中删除。

代码部分，在 Ammo 中 OnTriggerEnter2D 函数中生成这两个粒子特效。当 fireBall 打到敌人时生成 FireParticle 的粒子特效，当 waterBall 打到敌人时生成 waterParticle 的粒子特效，当 commonBall 打到敌人时没有粒子特效。

Ammo.cs 中加入这两个预制件：

```
Public GameObject fireParticle;
Public GameObject waterParticle;
```

然后在 OnTriggerEnter2D 函数里写上：

```
if (ammoItem.itemType == Item.ItemType.FIREBALL)
{
    Instantiate(fireParticle, transform.position, Quaternion.identity);
}else if(ammoItem.itemType == Item.ItemType.WATERBALL)
{
    Instantiate(waterParticle, transform.position, Quaternion.identity);
}
```

武器的预制件在面板上都加上这两个粒子特效。

制作各类提示界面，设置游戏的胜利法则，失败界面和胜利界面
场景中存在一些物体，他们起到提示的作用，比如，提示玩家需要做什么任务，提示玩家怎么玩这个游戏等等。
这边的话我们会在场景就做几个提示器：
第一个提示器时放在我们角色出生地的附近，然后当我们玩家触碰到它（trigger）的时候，画面上显示一个 UI 界面，告诉玩家这个游戏的任务是什么和我们要怎么样才能获取游戏的胜利等等的信息。游戏中我们会做两个提示用的路标,第一个是提示玩家需要做什么任务，第二个是提示玩家是否完成了任务与否；

Hierachy 右键 create empty 重命名为 TaskHintObject,挂上 sprite renderer,Sprite 参数载入那个标有“SEE”的路牌，sorting Layer:SpriteObject;替换掉原来的场景中 SEE 的路牌。
Hierachy 右键 UI->Canvas,重命名为 HintCanvas,组件 Canvas Scaler 跟之前一样， UI Scale Mode:Scale With Screen Size;reference resolution X:1920 Y:1080.Reference Pixels Per Unit:32;
在其下面创建 Panel 命名为 TaskPanel,panel 默认是扩充到整个屏幕的，这边锚点参数选成中心，然后调整这个 panel 的大小，然后基本居中在屏幕中间即可。Panel 的背景色调整为水蓝色，然后透明度调整下，有点透明的效果即可。Panel 下挂上两个文本，Task_txt 和 Hint_txt
Task_Txt 输入 TASK,font:slkscr,fontStyle:Bold.Font Size:100; font Color 调成粉红色
Hint_Txt 输入： Collect all the coins 空一行 Then find the God statue. Font size:44
然后在这个 panel 下再加入两个 image,命名为 Status_img 和 Array_img,分别载入雕像和箭头的精灵片，用于指示需要找到这个雕像。
制作完这个 panel 后，将其左上角的勾点掉，表示 setActive(false)

接着需要自定义脚本组件，当主角进入路标的话会显示对应的提示面板，这边创建 HintTrigger.cs 的脚本，这个类作为父类，可以延伸出其他的不同的用于各种提示的子类。
HintTrigger.cs 代码如下：
成员变量:
Public GameObject hintPanel;
Public virtual void OnTriggerEnter2D(Collider2D collision)在 unity 中是可以重写父类中的那些事件函数的，所以这里加上了 virtual;此外也可以在这里写上一些公用的功能，比如播放提示音等；
新建一个 TaskGiverTrigger.cs 的脚本继承自 HintTrigger
其代码如下：

```
override public void OnTriggerEnter2D(Collider2D collision)
{
    base.OnTriggerEnter2D(collision);
    if (collision.gameObject.CompareTag("Player"))
    {
        hintPanel.SetActive(true);
    }
}

private void OnTriggerExit2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        hintPanel.SetActive(false);
    }
}
```

这边需要改写父类对应的函数，所以 OnTriggerEnter2D 需要 override,先调用父类的这个函数，用 base.OnTriggerEnter2D,base 是个关键字，指代的是父类的实例，this 指代的是自己。然后我们要的效果的角色进入这个提示板，然后屏幕上显示出提示内容；出了这个提示板，提示内容就消失。

从 TaskHintObject 复制出新的一个，重名为 TaskFinishObject,用于提示主角是否完成了任务。这个游戏对象的 Sprite 显示选择标有“DES”的路牌。替换掉原来的场景中 DES 的路牌。
UI 中的 TaskPanel 也复制出两个来分别重名为 TaskFinishedPanel 和 TaskNotFinshedPanel,里面的图片对象都不要，文字换上：
TaskFinishedPanel： Good: You have collected all the coins Please go to see the God Statue
TaskNotFinshedPanel： Not Good:You haven’t collected all the coins please collect the remaining coins.
这些 UI 面板的默认情况下都是 setActive(false);

Player.cs 的脚本中加入一个是否游戏胜利的 bool bFinishGame =false 的变量进行测试。
自定义一个 TaskFinishTrigger.cs 的脚本，因为这边需要两个 Panel 显示，所以这边再增加一个成员变量 public GameObject hintPanel2;
相应的走近走出的代码如下：

```
override public void OnTriggerEnter2D(Collider2D collision)
{
    base.OnTriggerEnter2D(collision);
    Player player = collision.gameObject.GetComponent<Player>();
    if (player.bFinishGame)
    {
        hintPanel.SetActive(true);
    }
    else
    {
```

```
            hintPanel2.SetActive(true);
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            Player player = collision.gameObject.GetComponent<Player>();
            if (player.bFinishGame)
            {
                hintPanel.SetActive(false);
            }
            else
            {
                hintPanel2.SetActive(false);
            }

        }
    }
```

场景中通过调整 Player.cs 中的 bool bFinishGame 的值进行测试即可。

游戏的胜利的法则：主角吃光场景中的所有的金币，然后跑到雕像那里，游戏就胜利，显示游戏胜利的界面，可以进入下一关什么的。
Player.cs 中加上一个表示场景中有多少金币的成员变量：
pubic int coinAllCount = 0;
在 Start 函数中：获取这个场景中金币的总数量
GameObject[] coins =GameObject.FindGameObjectsWithTag(“CoinPickupObject”);
coinAllCount = coins.Length;
在 OnTriggerEnter2D 函数中，switch 语句中的
Case Item.ItemType.COIN:加上
if(hitObject.quantity>=coinAllCount)
{
bFinishGame = true;
}
每次在编辑器里面的测试的时候，要将 Coin 的 Item 中 quantity 数据改为 0；
在 UI 显示的 PlayerUI 的脚本中，将 UI 显示金币这块也加上总金币的显示，就以“吃到的金币数量/金币的总数”这种方式进行显示，只要修改下代码即可：
coinText.text =””+coinItem.quantity+”/”+player.coinAllCount;

制作胜利画面和失败的画面：
HintCanvas 中从原来的提示用的 panel 复制出两个显示面板，命名为 WinPanel 和 LosePanel panel 改成粉红色的背景，尺寸再调得大一点，然后位置居中即可。字体的颜色改成水蓝色。

WinPanel 中字体框里的内容：You Win;Congratulations;字体都改的大一点，醒目一点
LosePanel 中字体框里的内容：You Lose;What a pity,come on next time;
WinPanel 和 LosePanel 面板都增加两个个按钮：
WinPanel 中增加 NextLevel ,MainMenu 的按钮，按钮颜色为蓝色，字体为白色
LosePanel 中增加 ReStart,MainMenu 的按钮。
Button 组件中 Highlighted Color 和 Pressed Color 都改成深蓝色，吸取 waterBall 精灵片中的最外层颜色。这个是表示当鼠标移动到按钮上和按钮被按下时会显示的颜色。

制作神像 statue 的触发器对象，当我们的主角完成任务后，然后进入的神像的触发范围，我们的游戏的胜利了。
复制出原来的 TaskHintObject,重命名为 StatueObject，Sprite 换成雕像精灵片，替换掉原来的雕像的位置.
自定义 StatueTrigger.cs 让他继承自 HintTrigger.cs;

这边是我们的角色的 bFinishGame 这个变量变成 true 的时候会弹出胜利的界面，当我们角色死亡的时候会显示失败的界面，我们这边可以把胜利和失败的界面以及显示的代码都放在 RPGGameManager 这个类中，这个类有个全局的静态的单例，当我们游戏胜利或者失败的时候就从这个单例中调用相关的胜利以及失败的函数即可。
RPGGameManager.cs 中加入
Public GameObject winPanel;
Public GameObject losePanel;
然后增加一个 public void GameOver(bool playerWin)
我们希望是停个 0.几秒再显示相关的胜利界面和失败的界面，可以用协程。
代码如下：

```
public void GameOver(bool playerWin)
{

    StartCoroutine(DelayGameOver(playerWin));
}

IEnumerator DelayGameOver(bool playerWin)
{
    yield return new WaitForSeconds(0.5f);
    if (playerWin)
    {
        winPanel.SetActive(true);

    }
    else
    {
        losePanel.SetActive(true);
    }
    Time.timeScale = 0;
}
```

这个 Time.timeScale=0 有暂停游戏的作用。这边要放在最后，暂停后它的代码似乎不执行。
StatueTrigger.cs 的代码加上：

```
override public void OnTriggerEnter2D(Collider2D collision)
{
    base.OnTriggerEnter2D(collision);
    Player player = collision.gameObject.GetComponent<Player>();
    if (player.bFinishGame)
    {

        RPGGameManager.sharedInstance.GameOver(true);
    }
}
```

然后在 player.cs 的代码中需要重写 CharacterDie 这个函数：

```
public override void CharacterDie()
{
    base.CharacterDie();
    RPGGameManager.sharedInstance.GameOver(false);
}
```

最后的话，在课下整理下场景中各种元素，下节课做一下开始菜单，然后给游戏再上点音效，然后对相关的按钮写上相应的功能，能进行场景切换。

给游戏加入音效：
增加游戏音效后会使得游戏更加生动活泼；
游戏中增加音效的地方：
1. 发射投掷物的时候会发出“嗖”的一声；
2. 主角，敌人受伤的时候叫一声
3. 投掷物触碰到敌人变成烟花特效的时候有个爆炸的声音
4. 主角拾取金币和其他道具的时候发出的声音
5. 主角和敌人死亡后会留在一滩血迹是时候发出那一滩的声音
6. 游戏失败和胜利时，弹出对应的 UI 显示时，会播放失败和胜利的音乐

暂时没有的音效：走路的音效：因为我这个走路有点太快，加入走路后有点声音吵杂；游戏背景音乐的声音：还没有找到合适的背景音乐，可以留一个播放口。

播放声音涉及的相关组件：
Main Camera 会默认有一个 Audio Listener 的组件，表示听众，一般场景中只有一个 Audio Listener.有了这个 Listener，那么场景中就可以实现那种空间位置的声音的效果了，即离摄像机远的声音游戏玩家听到的就比较小，近的话声音会较大。

Audio source：声音播放的组件，相当于一个声音播放器，给游戏对象加上这个组件，然后在组件上拖入相应的音频文件就能播放。

一般情况下，这边就根据这个声音是来源于哪里的，就在这个游戏对象上挂载一个 Audio source 的声音播放组件，比如粒子烟花和一滩血迹，我们就先给这两个物体挂上音效。

Assets 文件夹中创建 Audio 文件夹，import new asset 载入所需要的音频文件；找到粒子烟花和血迹的预制件，然后都挂上 Audio source 组件，在 AudioClip 中拖入对应的音频文件，然后勾上 Play On Awake,这个参数的意思应该时我们的游戏角色在执行 Awake 事件的时候播放了这个声音，这种情况就比较使用于这里的粒子特效和血迹的声音效果，他们是在某个时刻才生成的，然后一声就可以播放声音特效了。参数 Volume 是控制音量的大小，Pitch 可以控制声音频率，可以控制声音低沉和尖锐。

我们可以自己写代码在脚本中控制声音的播放，此时要勾掉 Play on Awake 这个勾，比如这边我们做一下发射投掷物的音效，我们的投掷物都是预先生成的，存放在对象池中的，然后每次使用时都是 setActive(true),而不是每次 Awake;这边找到 3 个投掷物的 prefab,都加上 Audio Source 组件，clip 中拖入声音片段，去掉 play on Awake 的勾，打开 Ammo.cs:
加上成员变量

```
Private AudioSource aSource;
```

Awake 函数中

```
aSource= GetComponent<AudioSource>();
```

OnEnable()函数中

```
If(aSource !=null)
{
aSource.Play();
}
```

然后每次投掷物 setActive(true)的时候会播放这个音效。

这里需要注意的是，我们不能在游戏对象 destroy()或者 setActive(false)的时候调用 aSource.play(),这个时候游戏对象销毁或者冻结，那么它上面的组件是不起作用的。比如这里的游戏道具，我们这边想要吃游戏道具的时候有个音效的话，就不能把这个游戏音效挂载在这个游戏对象上，我们只能把这个游戏音效挂载在主角上，当主角拾取到游戏道具的时候就播放相关音效，因为我们的主角在那个时候不会 destroy 或者 setActive(false);

这边主角的话暂时不加上走路音效的话，就有两个音效，一个是受伤时候的音效，一个是拾取游戏道具的音效，这边将拾取金币和拾取其他道具分开两个音效，所以在主角身上会有好几个音效。然后敌人角色的话也有一个受伤的音效。
我们这边做一个假设，人物角色的话在某一个时刻只能发出一种音效，那么我们可以给主角只需要加上一个 AudioSource 组件即可，然后音频片段 audio clip 也设置成变量，然后播放不同的音效时，改变这个 audio clip 即可，player.cs 增加代码：现在其父类 Character 中增加

```
Protected AudioSource aSource;
Public AudioClip hurtClip;
```

因为主角和敌人都是有受伤的音效的。
Player.cs 和 Enmey.cs 各自的 start()函数来获取这个 AudioSource 组件

```
aSource = GetComponent<AudioSource>();
```

Character 类中的 TakeDamage()函数中 while(true)加上：

If(aSource!=null && hurtClip!=null)
{
aSource.clip=hurtClip;
aSource.Play();
}
在 Player.cs 中再加上拾取道具的音效：
Public AudioClip pickCoinAudio;
Public AudioClip pickOtherAudio;

在 onEnterTrigger2D 的 switch 中加上：
Coin:

```
if (aSource != null && pickCoinAudio != null)
{
    aSource.clip = pickCoinAudio;
    aSource.Play();
}
```

其他拾取物都加上：

```
if (aSource != null && pickOtherAudio != null)
{
    aSource.clip = pickOtherAudio;
    aSource.Play();
}
```

然后在面板上添加上相应的 audioClip 即可。

游戏胜利和游戏失败，以及背景音乐的代码可以添加在 RPGGameManager 中：
然后在 RPGGameManager.cs 的代码中填上：
Private AudioSource aSource;
Public AudioClip backgoundMusic;
Public AudioClip winAudio;
Public AudioClio loseAudio;
Start 函数中：
aSource = GetComponent<AudioSource>();
If(backgoundMusic != null)
{
aSource.clip=backgroundMusic;
aSource.Play();
}
然后是 DelayGameOver 的函数中，分别在胜利和失败的分支加上代码：
If（winAudio!=null）
{
aSource.clip = winAudio;
aSource.Play();
}
If(loseAudio!=null)

```
{
aSource.clip = loseAudio;
aSource.Play();

}
```

增加菜单 MainMenu 的新场景，完成场景切换的功能，游戏发布

新增加一个场景,菜单栏 File->New Scene,然后 Ctrl+S,命名成 MainMenu，保存在 Scenes 文件夹中。创建一个 UI->Canvas,UI Scale Mode:Scale With Screen Size;Reference Resolution:1920,1080;然后创建一个背景 Panel 命名为 backgroundPanel,将背景色调整成蓝色。之后拖入相关的 image 游戏对象和 text 游戏对象进行图片显示和文字说明即可，这边注意下每个 UI 的锚点设置下就行，最后拖入两个按键，play 和 quit 的按键，设置下他们的高亮和按下的颜色提示即可。

UI 完成后，可以在 Game Scene 切换分辨率看一下，可能有些 UI 元素的锚点设置得不合理。在 UI 的 button 中会加载上 OnClick 的点击事件来切换场景。

设置场景切换：首先需要需要打包的关卡放到 build settings，打开 File->build settings，将要打包的游戏场景 scene 拖入到 Scenes In Build，右边还显示了场景的序号。

场景中拖入一个 RPGGameManager,我们将场景切换的代码写到 RPGGameManager.cs 中，打开这个脚本，引入命名空间。

Using UnityEngine.SceneManagement;

加入下面两个函数：

```
    public void LoadNextLevel(int index)
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(index);
    }

    public void QuitGame()
    {
        Application.Quit();
    }
```

切换场景的函数 SceneManager.LoadScene(int index),这里的 index 就是前面在 build settings 中的场景序号，也可以写场景的名字。在这个函数前需要加上 Time.timeScale=1;因为之前的场景结束的时候加上了 Time.timeScale = 0;需要把它调回来。

MainMenuCanvas 中选中 play_btn,OnClick 选中“+”，拖入场景中的 RPGGameManager 游戏对象，然后选择 LoadNextLevel 函数，参数填上 1；

Quit_btn,OnClick 选中“+”，拖入场景中的 RPGGameManager 游戏对象，然后选择 QuitGame() 的函数。

这个 RPGGameManager 应该不会被带到下一个场景中，需要将一个游戏对象带到下一个场景中去的话，需要用 DontDestroyOnLoad();

双击 Level01 的关卡，对游戏胜利和游戏失败的界面中的按钮也设置一下相关的函数，
winPanel:
NextLevel_btn:不设置；MainMenu_btn:拖入 RPGGameManager 游戏对象，选择 LoadNextLevel 函数，参数填上 0；
losePanel:
Restart_btn:拖入 RPGGameManager 游戏对象，选择 LoadNextLevel 函数，参数填上 1；
MainMenu_btn:拖入 RPGGameManager 游戏对象，选择 LoadNextLevel 函数，参数填上 0；

Weapon .cs 的函数需要修改：
void awake（）中
将 if(ammoPool==null)这个判断去掉，场景切换的时候似乎会保留原有的 ammopool 导致发不出投掷物；

当游戏失败点击场景的 restart 的时候，发现上一次吃到的游戏金币的数量还是被保存下来了，只有在退出这关然后再进入这关，或者退出程序，再进入的话才不会被保存下来。看来需要修改 Scriptable object 的数据的话还是再复制一份相关的数据再进行修改比较好。背包里的数据就是复制一份出来的，所以 restart 的情况下是重新一份数据的。先回到 MainMenu 然后在从 MainMenu 进入的话这个数据就不保存了。 这边相关代码我暂时不修改了（其实就想把 restart 去掉）。打包时金币的 Scriptable object 的数据的 quantity 清零之后再打包。

生成的背包系统的几个 slot 的显示在不同的分辨率下似乎有问题，跟实际在场景中拖上去的效果似乎不太一样。这边 instantiate 之后再挂上去的话，似乎这里 localScale 的大小会变，这边我就在代码中设置为 1，在 Inventory.cs 的 createSlots 函数中在 new .transform.SetParent() 的后面加上下面这句：
newSlot.transform.localScale=new Vector3(1.0f,1.0f.1.0f);

游戏打包：直接可以在刚才的 Build Settings 中，将要打包的游戏场景拖入，选择平台 Platform，我们这边就是 PC 平台,旁边的参数默认即可，然后点开这个 Player Setting 有其他一些画质啊，想能啊之类的参数可以设置。点击 Build 按钮，选择要打包到的文件夹即可。打包好后就可以试玩游戏了。

课程目录：
000_课程简介
001_像素画介绍
002_Aseprite 像素画软件简介
003_Aseprite 地形瓦片集(TileSets)绘制
004_Aseprite 房屋绘制
005_Aseprite 植被绘制
006_Aseprite 人物正面绘制
007_Aseprite 人物侧面绘制
008_Aseprite 人物正面动画制作
009_Aseprite 人物侧面动画制作
010_美术资源整理
011_C#编程准备

012_C#HelloWorld
013_C#变量和函数
014_C#面向对象编程 1
015_C#面向对象编程 2
016_C#类的封装继承和多态 1
017_C#类的封装继承和多态 2
018_C#静态 static
019_C#委托 Delegate
020_C#协程 Coroutine 原理 1
021_C#协程 Coroutine 原理 2
022_C#反射 Reflection
023_Unity 初步介绍 1
024_Unity 初步介绍 2
025_Unity 使用 TileMap 搭建游戏场景
026_Unity 场景添加环境元素 1
027_Unity 场景添加环境元素 2
028_Unity 控制主角移动和碰撞 1
029_Unity 控制主角移动和碰撞 2
030_Unity 主角动画切换
031_Unity 使用动画 BlendTree
032_Unity 相机跟随
033_Unity 使用 Cinemachine 插件进行相机跟随
034_Unity 道具制作和主角交互
035_Unity 使用 ScriptableObject
036_Unity 制作 UI 显示
037_Unity 制作简易背包系统 1
038_Unity 制作简易背包系统 2
039_Unity 制作简易背包系统 3
040_Unity 单例 Manager 类
041_Unity 制作敌人角色
042_Unity 敌人场景漫游和追逐玩家 1
043_Unity 敌人场景漫游和追逐玩家 2
044_Unity 制作角色出生点
045_Unity 主角攻击 1
046_Unity 主角攻击 2
047_Unity 增加主角攻击动画
048_Unity 优化角色攻击和受伤
049_Unity 场景优化 1(增加提示框，设置胜利条件)
050_Unity 场景优化 2(增加提示框，设置胜利条件)
051 _Unity 增加游戏音效
052_Unity 增加 MainMenu 和游戏打包