C1 用Microbit实现“雷霆战机”

姚琦 刘佳铭

摘要：本文分析microbit实现小游戏“雷霆战机”的过程，并且对提供了一个实现方案，对源代码进行分析，后续我们将对其进行更进一步的拓展，以增强其可玩性。

1. 编码之前的分析：

我们注意到，在经典的小游戏“雷霆战机”中存在着三个主要对象：代表飞机主体的 **plane** 对象、代表飞机所发射的激光的**plane\_razor** 对象、代表敌人的**razor**对象。同时，为了增加游戏的乐趣，我们考虑增加一个 **buff** 对象，当玩家获得buff时，就可以获得对应的收益。有了这些分析，我们所要做的第一步就是构造这四个类，对其进行封装，在后续的主循环中调用其接口从而实现该小游戏



“雷霆战机“图片

1. 四个主要类的构建

（1）我们首先来构建**plane**这个类，我们需要借助一个像素点来代表玩家所操控的飞机。而我们展示像素点时，需要调用二个参数：代表飞机坐标的location 与 代表飞机亮度的light。代表飞机坐标的location应该借助列表进行实现，其中的第一个对象代表飞机横坐标的值，第二个则代表其纵坐标。

借助以上分析，我们可以写出plane的\_\_init\_\_ 方法中的一部分，与展示其坐标的show 方法：

class plane:  
 def \_\_init\_\_(self):  
 self.Light = 9  
 self.Location = [2,4]  
 def show(self):  
 display.set\_pixel(self.Location[0], self.Location[1],self.Light)

初始化使得飞机恰好处在microbit 板的下方的正中间。

在完成这个初始化构建后，我们可以开始考虑飞机的移动问题。我们注意到，microbit 板上实际仅仅存在三个按键：A、B

与Logo。所以，想要实现四个方向上的移动，我们可以使飞机在Logo不被触摸时一直向下移动，而在Logo被触摸时向上移动。

另外，在实现这几个接口时，还要注意边界条件。例如其横坐标的取值范围是（0，1，2，3，4），我们的飞机坐标不能超出这些范围。具体的代码实现如下：

def Left\_go(self):  
 if button\_a.is\_pressed():  
 if self.Location[0] > 0:  
 self.Location[0] -= 1  
  
def Right\_go(self):  
 if button\_b.is\_pressed():  
 if self.Location[0] < 4:  
 self.Location[0] += 1  
  
def Upwards(self):  
 if pin\_logo.is\_touched():  
 if self.Location[1] > 0:  
 self.Location[1] -= 1  
  
def Downwards(self):  
 if not pin\_logo.is\_touched():  
 if self.Location[1] < 4:  
 self.Location[1] += 1

由于在接口中已经加入了条件判断，所以后续仅需在主循环中调用这几个接口即可。

另外，我们需要考虑飞机的血量问题，使得主循环能在飞机血量为0时结束，所以我们考虑用飞机的亮度self.Light代表飞机的血量。当飞机被击中时，其亮度减少，减少到一定值时，循环结束。

为了实现上述功能，我们考虑在\_\_init\_\_中加入如下参数：

self.outed = False

同时，我们也需要定义一个out接口，使得当生命值降低到一定程度时，将上述参数修改为True，从而结束主循环，该接口代码实现如下：

def Out(self):  
 if self.Light == 1:  
 display.scroll('Out')

display.scroll(self.count)  
 self.outed = True

而使飞机血量减少，则需要另外一个接口来实现，这个接口在飞机被敌方发射的razor击中时，使飞机血量减少，该接口实现如下：

def getHitted(self,razor):  
 if self.Location == [razor.x,razor.y]:  
 self.Light -= 2  
 razor.disappear()  
 return True  
 else:  
 return False

razor是我们接下来要实现的类对象。

到此，我们对plane类的构建基本完成，后续在我们完成剩余三个类的定义后，我们再对该类进行进一步的修正，并且添加相应的接口与参数。

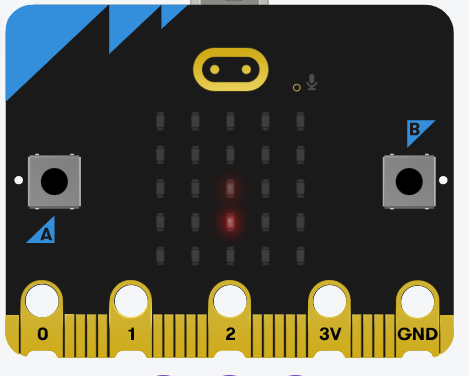
（2）接下来我们来进行razor类的构建。我们设想，razor应当从microbit板的最上方随机生成，并且不断的下落，于是这个类的几个接口所要实现的功能就很清楚了。除了\_\_init\_\_外，剩下的几个接口分别要实现razor的展示、消失与下降，于是这个类的实现如下：

class razor:  
 def \_\_init\_\_(self):  
 self.x = randint(0,4)  
 self.y = 0  
  
 def show(self):  
 if self.y <= 4:  
 display.set\_pixel(self.x,self.y,1)  
   
  
 def disappear(self):  
 display.set\_pixel(self.x,self.y,0)  
  
 def godown(self):  
 self.y += 1

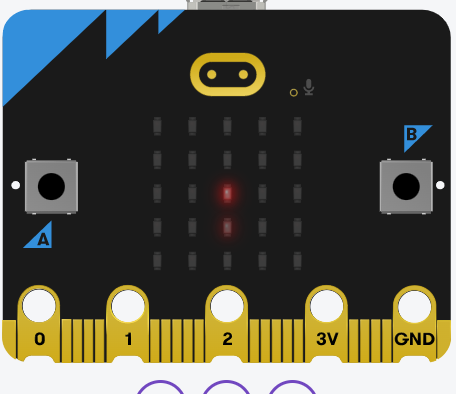
（3）而对于plane\_razor类的构建，与razor类的构建类似，但是不同的是，在初始化时，plane\_razor对象的坐标应当调用plane的坐标。而且，我们需要额外实现一个新的接口，来判断plane\_razor对象是否与razor对象相撞。依据以上分析，plane\_razor的代码实现如下：

class plane\_razor:  
 def \_\_init\_\_(self,plane):  
 self.x = plane.Location[0]  
 self.y = plane.Location[1]  
   
  
 def show(self):  
 if self.y >= 0:  
 display.set\_pixel(self.x,self.y,3)  
   
   
 def goup(self):  
 self.y -= 1  
  
 def disappear(self):  
 display.set\_pixel(self.x,self.y,0)  
  
 def Hitted(self,razor):  
 if self.x == razor.x and self.y <= razor.y :  
 self.disappear()  
 razor.disappear()  
 return True  
 else:  
 return False

其中hitted这个接口，如果采用横纵坐标相等时判断两者相等，很容易出现如下情况：



此时是移动之前的情况，当发生移动后，情况如下：



此时两者仅仅互换位置而不发生相撞判定，所以我们决定采用plane\_razor在razor之上来进行判定，从而实现更为精确的判断。

（4）Buff类的构建，为了增加游戏的趣味性与随机性，我们考虑在游戏中加入buff。Buff的增益效果有三类：增加得分，进入无敌状态，回复血量。而这些都与plane自身的性质有关，于是，构建Buff类必然无法回避在plane中添加接口与参数的问题。

Buff类本身的构建与razor类类似，仅需改变其亮度来对其进行区分。具体实现如下：

class Buff:  
 def \_\_init\_\_(self):  
 self.Location = [randint(0,4),0]  
  
 def disappear(self):  
 display.set\_pixel(self.Location[0],self.Location[1],0)  
  
 def godown(self):  
 self.Location[1] += 1  
   
 def show(self):  
 if self.Location[1] <= 4:  
 display.set\_pixel(self.Location[0],self.Location[1],7)

重点应当放在对plane功能的完善上。

我们考虑对plane增加一个buff\_get方法，每次调用时，检查是否获得buff，并随机选择buff所带来的增益。同时，我们需要为plane增加一个count参数，来表示所得分数。以及一个参数wuDi，来表示其是否进入无敌时间。具体实现如下：

def get\_buff(self,buff):  
 if self.Location == buff.Location:  
 choice\_num = randint(0,2)  
 if choice\_num == 0:  
 if self.Light != 9:  
 self.Light += 2  
 else:  
 self.count += 5  
 elif choice\_num == 1:  
 self.wuDi = True  
 self.wuDitime = 0  
 else:  
 self.count += 5  
 return True  
 else:  
 return False

而改写后的\_\_init\_\_变为：

def \_\_init\_\_(self):  
 self.Light = 9  
 self.Location = [2,4]  
 self.wuDi = False  
 self.outed = False  
 self.count = 0  
 self.wuDitime = 0  
 self.temp = self.Light

其中，无敌时间并不是无限的，我们需要一个接口来实现，当飞机在无敌时间内时的效果，并且在一定的回合数后退出无敌时间，该接口定义如下：

def wudiTime(self):  
 if self.wuDi:  
 self.wuDitime += 1  
 self.Light = 0  
 if self.wuDitime > 50:  
 self.wuDi = False  
 self.wuDitime = 0  
 self.Light = self.temp

而在无敌时间内getHitted接口并不生效，我们仅需考虑加入条件判断语句即可：

def getHitted(self,razor):  
 if not self.wuDi:  
 if self.Location == [razor.x,razor.y]:  
 self.Light -= 2  
 razor.disappear()  
 return True  
 else:  
 return False  
 else:  
 return False

至此，我们完成了对四个主要类的构建，接下来进入主循环

1. 主循环

在编写主循环之前，我们需要构造一个plane对象，以及盛装plane\_razor、razor、buff的三个数组，代码实现如下：  
i 代表回合数

Plane = plane()  
i = 0  
razor\_lst = []  
plane\_razor\_lst = []  
Buff\_lst = []

主循环的基本结束条件是飞机的outed参数为False，所以，主循环的判断条件应当写作

while not Plane.outed:

首先应当调用飞机的移动方法，使得飞机能够移动。

Plane.Left\_go()  
Plane.Right\_go()  
Plane.Downwards()  
Plane.Upwards()

然后我们考虑每三回合，飞机发出一个射线，每两回合，从上方随机生成一个敌方目标，并且以1/50 的概率随机生成一个buff，

这些对象均加入到相应的列表中，具体实现如下：

i += 1  
if i % 3 == 0:  
 pr = plane\_razor(Plane)  
 plane\_razor\_lst.append(pr)  
if i % 2 == 0:  
 r = razor()  
 razor\_lst.append(r)  
buffnum = randint(0,50)  
  
if buffnum == 10:  
 b = Buff()  
 Buff\_lst.append(b)

接下来我们需要维护这三个列表，使得其中的元素不出现坐标越界的情况，即将其中越界的元素删除，具体实现如下：

if plane\_razor\_lst:  
 for m in range(len(plane\_razor\_lst)-1,-1,-1):  
 if plane\_razor\_lst[m].y == -1:  
 plane\_razor\_lst.pop(m)  
   
if razor\_lst:  
 for m in range(len(razor\_lst)-1,-1,-1):  
 if razor\_lst[m].y == 5:  
 razor\_lst.pop(m)  
   
if Buff\_lst:  
 for m in range(len(Buff\_lst)-1,-1,-1):  
 if Buff\_lst[m].Location[1] == 5:  
 Buff\_lst.pop(m)

采用逆向遍历数组的方法，从而消除删除某一元素对后面元素的影响。

接下来我们需要考虑两个相撞的情况，即plane与razor相撞，razor与plane\_razor相撞。调用相应的hit接口即可实现，同时，我们需要将相撞的元素移除列表，具体实现如下：

if razor\_lst and plane\_razor\_lst:  
 for n in range(len(razor\_lst)-1,-1,-1):  
 for m in range(len(plane\_razor\_lst)-1,-1,-1):  
 if plane\_razor\_lst[m].Hitted(razor\_lst[n]):  
 plane\_razor\_lst.pop(m)  
 razor\_lst.pop(n)  
 Plane.count += 1  
 break  
   
if razor\_lst :  
 for m in range(len(razor\_lst)-1,-1,-1):  
 if Plane.getHitted(razor\_lst[m]):  
 razor\_lst.pop(m)

接下来我们需要考虑获得buff的情况，调用get\_buff接口，并遍历整个buff数组即可

if Buff\_lst:  
 for j in range(len(Buff\_lst)-1,-1,-1):  
 if Plane.get\_buff(Buff\_lst[j]):  
 Buff\_lst.pop(j)

另外我们要调用wuDiTime方法为其计数

Plane.wudiTime()

接下来就是展示问题，我们需要遍历三个数组，调用其show方法，使得其在microbit上展示出来。在遍历之前，我们需要清空上一个循环的展示，否则会出现重叠现象，代码实现如下：

for pr in plane\_razor\_lst:  
 pr.show()  
   
for r in razor\_lst:  
 r.show()  
   
for b in Buff\_lst:  
 b.show()  
   
Plane.show()  
Plane.Out()

我们同时调用了Out方法来判断飞机是否死亡。

接下来我们遍历三个数组，使得其中的对象不断地移动

for pr in plane\_razor\_lst:  
 pr.goup()  
   
for r in razor\_lst:  
 r.godown()  
   
if i % 3 == 0:  
 for b in Buff\_lst:  
 b.godown()

在这些执行完毕后，程序需要停顿一段时间再进入下一个循环，这个停顿时间应当与回合数有关，回合数越高，休眠时间越短，具体代码实现如下：

if i <= 200:  
 sleep(200)  
  
elif 200<i<400:  
 sleep(100)  
else:  
 sleep(50)

至此主循环完成

1. 遇到的问题

单片机内存问题困扰了我们，因为在模拟器上运行，会存在一定的缓存，所以，当我们在调试程序的时候，会出现当我们反复执行程序多次后，某一次执行速度突然减慢。为此，我们考虑在microbit实机（上学期计概C1奖品）上进行调试，问题成功得到解决

另外，这个程序中的数值规划并没有进行详细的思考，所以在实际体验过程中，可能我们无法反应过来，或者会出现一直待在原地不动即可的情况。

1. 后续展望

后续我们考虑进一步优化该程序的数值规划，从而实现更好的游戏性，同时我们考虑加入更加丰富的Buff与敌人，从而使得游戏的乐趣增加

同时我们考虑用pygame模块为这个程序设计一个简单的图形，使得其更加贴近原作。