北京理工大學

汇编小组实验报告

汇编小组实验报告

Assembly Language Lab Report

学　　院：	计算机学院
专　　业：	计算机科学与技术
学生姓名：	索一贺、朱函琪、孙嘉昕、尤乐淳
学　　号：	1120191187、1120190807、 1120191173、1120190855
指导教师：	李元章

2022 年 6 月 24 日

目 录

第 1 章 游戏概述

1.1 游戏介绍

这是一款我们原创的游戏，名为“橄榄球、死亡与机器人”，简称“橄死机”。命名模仿了当下热播的Netflix网剧《爱，死亡与机器人》。

“橄死机”游戏是一款闯关游戏，玩家通过控制橄榄球进攻球员的移动，躲避防守球员的移动路线，最终成功达阵的游戏。

由于关卡一共有三个主题，分别是橄榄球、丧失、机器人，因而命名为“橄榄球、死亡与机器人”。

1.2 游戏机制

玩家按“W”“A”“S”“D”操纵进攻队员的移动，防守队员会向玩家控制的进攻队员的位置以在一定范围内随机的速度移动。玩家控制的进攻队员与防守队员相撞后，闯关失败。防守队员移动出下边界后，会在上边界处随机的位置重新生成。每闯过一关后，游戏的速度会增加。游戏最开始和每闯过一关后，需要按回车键开始新一关的游戏。

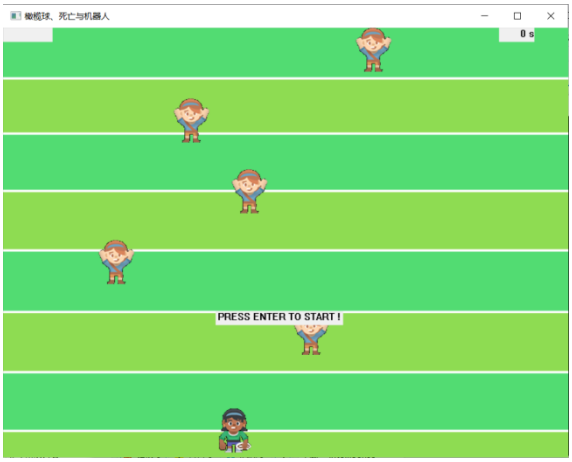

图 1-1 开始界面

图 1-2 结束界面

第2章 界面设计

2.1 资源文件

资源文件包括进攻方和防守方的图片、背景图片、背景音乐等。以背景图片为例，资源标识在 resource.h 中定义：

```
#define IDB_BKG1                        100
#define IDB_BKG2                        101
#define IDB_BKG3                        102
```

在主过程中加载：

```
invoke LoadBitmap,hInstance,IDB_BKG1
mov hBmbkg1,eax
invoke LoadBitmap,hInstance,IDB_BKG2
mov hBmbkg2,eax
invoke LoadBitmap,hInstance,IDB_BKG3
mov hBmbkg3,eax
```

2.2 界面设计

游戏界面共有三种场景，分别是橄榄球、僵尸和机器人，还包括关卡和时间显示（更新时通过SendMessage实现）。三种场景按照关卡round取模轮替，每通过一关就更换场景。

```
ChangeRound proc uses eax
            ;改敌人和背景，三种场景轮替
            mov edx, 0
            mov eax, round
            mov ebx, 3
            div ebx
            .if edx == 1
                mov eax, hBmbkg1
                mov hBmbkg, eax
                mov eax, hBmgirl
                mov hBmenemy, eax
            .elseif edx == 2
                mov eax, hBmbkg2
                mov hBmbkg,eax
                mov eax, hBmzombie
                mov hBmenemy, eax
            .elseif edx == 0
```

```
                mov eax, hBmbkg3
                mov hBmbkg,eax
                mov eax, hBmrobot
                mov hBmenemy, eax
            .endif
            ret
ChangeRound endp
```

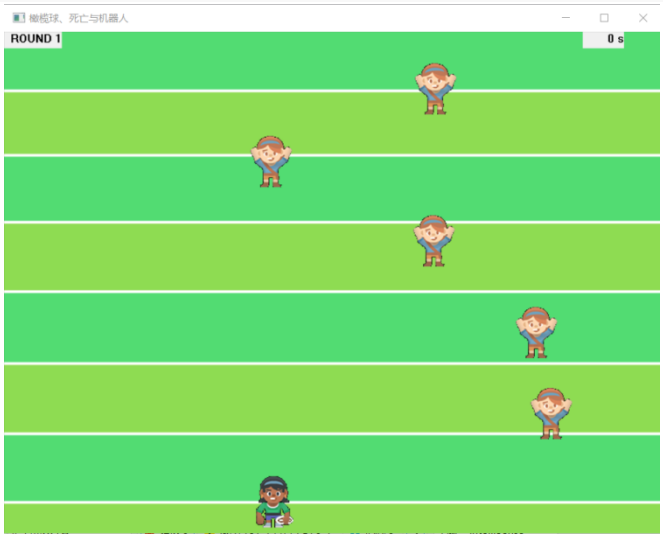

图2-1 场景1

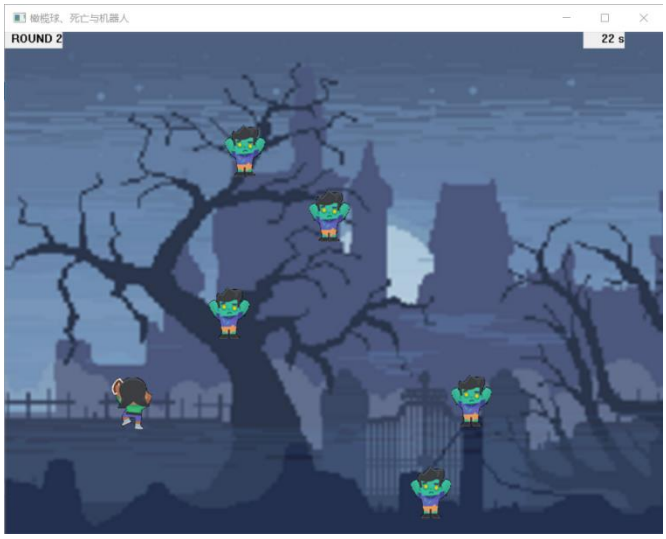

图2-2 场景2

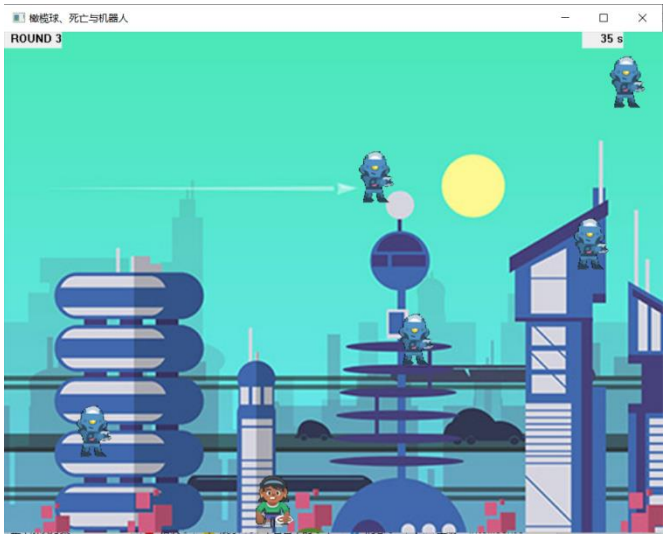

图2-3 场景3

2.3 人物显示

2.3.1 人物动态

玩家控制的进攻方共有 9 张图片（包括初始状态、上下左右各两张），当前显示的图片根据当前坐标取模决定，达到动态的效果。former 记录上一个动作，以在不按键时保持上一动作。

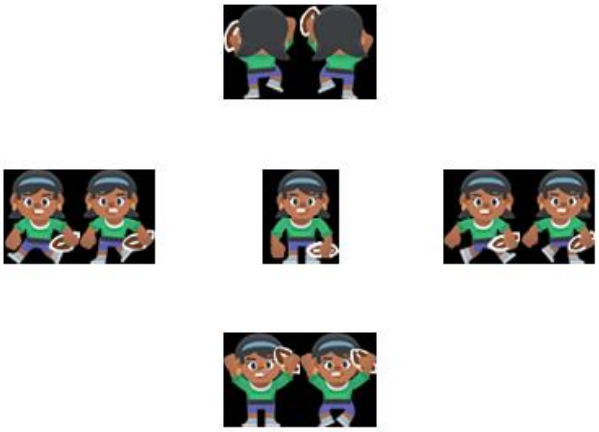

图 2-4 人物动态

以向下（按向下键/上一个动作为向下）为例（包括位置的更新）：

```
.if isdown == 1 || former == 1
    mov former, 1
    mov eax, player.y
    mov ebx, 2
    div ebx
```

```
    .if edx == 1
        invoke CreateBitmapMask,  hBmdown1
        mov hBmpMask,edx
        invoke SelectObject, mDc, hBmpMask
        invoke BitBlt, @hDc, player.x, player.y, bm.bmWidth, bm.bmHeight, mDc, 0, 0, SRCAND
        invoke SelectObject, mDc, hBmdown1
        invoke GetObject, hBmdown1, sizeof BITMAP, addr bm
    .else
        invoke CreateBitmapMask,  hBmdown2
        mov hBmpMask,edx
        invoke SelectObject, mDc, hBmpMask
        invoke BitBlt, @hDc, player.x, player.y, bm.bmWidth, bm.bmHeight, mDc, 0, 0, SRCAND
        invoke SelectObject, mDc, hBmdown2
        invoke GetObject, hBmdown2, sizeof BITMAP, addr bm
    .endif
```

.....

```
.endif
          invoke BitBlt, @hDc, player.x, player.y, bm.bmWidth, bm.bmHeight, mDc, 0, 0, SRCPAINT
          invoke BitBlt, @hDc_old, 0, 0,@stRect.right, @stRect.bottom, @hDc, 0, 0, SRCCOPY
```

2.3.2 透明背景

进攻方和防守方原图背景都是黑色，都通过创建 Mask 来显示透明背景。创建 Mask 方法如下：

```
CreateBitmapMask proc hbmp:HBITMAP
                local hdcm1:HDC
                local hdcm2:HDC
                local bm1:BITMAP
                local hbmMask:HBITMAP

                invoke GetObject, hbmp, sizeof BITMAP, addr bm1
                invoke CreateBitmap, bm1.bmWidth, bm1.bmHeight, 1, 1, NULL
                mov hbmMask, eax
                invoke CreateCompatibleDC, 0
                mov hdcm1, eax
                invoke CreateCompatibleDC, 0
                mov hdcm2, eax
                INVOKE SelectObject, hdcm1, hbmp
                invoke SelectObject, hdcm2, hbmMask
                invoke SetBkColor, hdcm1, 0
                invoke BitBlt, hdcm2, 0, 0, bm1.bmWidth, bm1.bmHeight, hdcm1, 0, 0, SRCCOPY
                invoke BitBlt, hdcm1, 0, 0, bm1.bmWidth, bm1.bmHeight, hdcm2, 0, 0, SRCINVERT
```

```
                invoke DeleteDC, hdcm1
                invoke DeleteDC, hdcm2
                mov edx, hbmMask
                ret
CreateBitmapMask endp
```

显示 Mask 方法如下（以防守方为例），其中 SRCAND 代表与操作，SRCPAINT 代表或操作。

```
invoke SelectObject, mDc, hBmpMask
invoke BitBlt, @hDc, dotb2.x, dotb2.y, bm.bmWidth, bm.bmHeight, mDc, 0, 0, SRCAND
invoke SelectObject, mDc, hBmenemy
invoke GetObject,hBmenemy, sizeof BITMAP, addr bm
invoke BitBlt, @hDc, dotb2.x, dotb2.y, bm.bmWidth, bm.bmHeight, mDc, 0, 0, SRCPAINT
```

图 2-5 进攻方原图

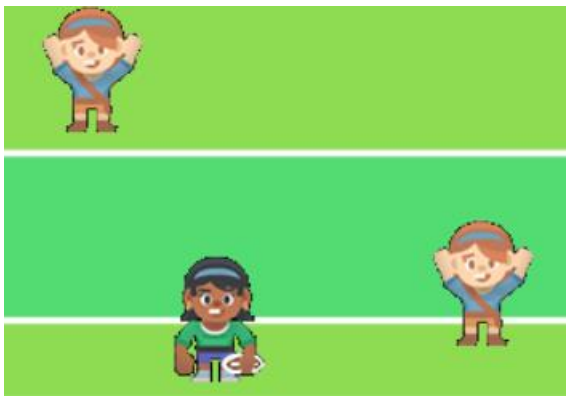

图 2-6 实际显示效果

第3章 人物移动

3.1 人物位置

使用游戏界面内一个点的坐标来记录人物的当前位置，x, y 分别代表人物在水平方向上的位置和竖直方向上的位置。

3.2 人物移动

想要通过 wasd 键控制人物移动，就要知道在计算人物下一位置时知道当前键盘按键的按下状态，本游戏采用 GetAsyncKeyState 来检测某一按键的状态，接着根据 wasd 的按下状态，对人物坐标做相对应的修改。同时实现了同时按下多个按键可以让人物斜向移动，人物的移动方向共 8 个。

以识别按下 a 键人物位置向左为例，识别到 a 键按下后，横坐标减少一个单位值

```
INVOKE  GetAsyncKeyState, VK_A
                test    eax,8000h
                .if     !ZERO?
                        .if     player.x >= ebx
                            sub     player.x, ebx
                            mov     isleft, 1
                            mov     former, 3
                            .endif
                        .endif
```

第4章 游戏时间

4.1 游戏时间

游戏中包含有两个时间，一个是和游戏速度相关的游戏内时间，即游戏中各物体位置的刷新时间，另一个是游戏中显示的玩家游戏时间，与现实时间相同，以秒为单位。

4.2 具体实现

使用 SetTimer 建立计时器，其中一个的计时器时间间隔是变化的，控制游戏时间流速的快慢，想要游戏速度加快时，就减小此计时器的时间间隔，物体位置的变化会加快。

```
invoke  SetTimer, hWinMain, ID_TIMER, gameSpeed, NULL
```

计时器会根据时间间隔发送 WM_TIMER 信息，窗口接收到这个信息时，进行整个

窗口内容更新，游戏不断进行。

另一个计时器的时间间隔默认设定为 1s，用于记录玩家挑战关卡的时间，使用一个变量记录时间，每次到达设定的时间间隔时，此计时器会调用_ProcTimer 更新时间变量值（加 1），玩家的游戏时间会在窗口刷新时更新在窗口中。

```
invoke  SetTimer, hWnd, ID_TIMER2, 1000, addr _ProcTimer
```

```
_ProcTimer  proc _hWnd, uMsg, _idEvent, _dwTime
            pushad

            ;只需要修改这里
            add     playingTime,1
            invoke  wsprintf, addr text_time, addr formatTime, playingTime
            invoke  GetDlgItem, hWinMain, TIME_TEXT
            invoke  SendMessage, eax, WM_SETTEXT, NULL, addr text_time


            popad
            ret
_ProcTimer  endp
```

4.3 游戏加速

游戏中，随着人物不断闯关，关卡的难度也会提高，游戏速度会加快。在每次判定过关之后，需要将原来的游戏刷新计时器删除，重新设置一个时间间隔较小的计时器作为游戏的刷新计时器。

```
invoke KillTimer, hWinMain, ID_TIMER
mov     eax, gameSpeed
shr     eax, 1
mov     gameSpeed, eax
invoke SetTimer, hWinMain, ID_TIMER, gameSpeed, NULL
```

第5章 障碍物移动

5.1 初始位置设置

设置五个障碍物，将其纵坐标分别设为 0、100、200、300、400、500。通过调用 Random 随机函数，产生窗口长度范围内的随机数作为其横坐标。

```
mov    ebx, windowW
sub    ebx, defenW
invoke Random,        ebx
mov         dotb1.x,  eax
mov         dotb1.y,  0
invoke Random,        ebx
mov         dotb2.x,  eax
mov         dotb2.y,  100
invoke Random,        ebx
mov         dotb3.x,  eax
mov         dotb3.y,  200
invoke Random,        ebx
mov         dotb4.x,  eax
mov         dotb4.y,  300
invoke Random,        ebx
mov         dotb5.x,  eax
mov         dotb5.y,  400
```

5.2 障碍物纵向移动

通过调用 Random 随机函数获取一个一定范围内的随机值，再把这个随机值加到某个障碍物的纵坐标中。

```
invoke Random, speed
add    dotb1.y,  eax
```

5.3 障碍物横向移动

我们需要让障碍物朝着进攻队员的方向移动，因此我们首先同样通过调用 Random 随机函数获取一个一定范围内的随机值，再判断该障碍物在进攻队员的左边还是右边。如果在左边就在障碍物的横坐标加上这个随机的距离，反之就减去这个随

机的距离。

```
invoke Random, speed
mov    ebx, player.x
.if    ebx > dotb1.x
       add  dotb1.x,  eax
.else
       sub  dotb1.x,  eax
.endif
```

第6章 碰撞检测与过关检测

6.1 碰撞检测

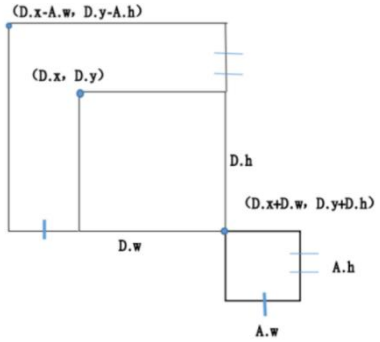

图 6-1 碰撞检测示意图(其中 D.x、D.y 分别为障碍物的横纵坐标，D.w、D.h 为障碍物的宽和高，A.w、A.h 分别为进攻队员的宽和高)

如图所示，左边的矩形是障碍物，右下方的小矩形是进攻队员。因为采用他们左上角的顶点作为其坐标，故碰撞的范围为左上角的大矩形所包括的范围。当进攻队员的左上顶点处于此范围内，则发生碰撞。

```
mov    eax, dotb1.x
mov    ebx, dotb1.y
sub    ebx, plyH
.if    player.y >= ebx
       add  ebx, plyH
       add  ebx, defenH
       .if  player.y <= ebx
            sub  eax, plyW
            .if  player.x >= eax
                 add  eax, plyW
                 add  eax, defenW
```

```
                .if  player.x <= eax
                    mov state,3       ;发生碰撞
                    ret
                .endif
            .endif
        .endif
    .endif
```

6.2 过关检测

当进攻队员的纵坐标小于 10 时，就触发碰撞检测，重新设置新的关卡。