大作业实验指导书

大作业框架面向《程序设计基础》课程的大作业实验项目，原则上所有同学必须使用该框架完成指定的大作业题目，如果有同学想要使用其他框架完成大作业或者选择其他的大作业题目，请提前联系助教确认。

本次大作业实验的题目是《飞机大战》，一款耳熟能详的游戏。想必各位同学从小到大或多或少玩过类似的游戏，比如“QQ雷电”等。在本次大作业实验中，同学们将基于Windows桌面应用程序利用Win32 API实现一款《飞机大战》游戏，提升自身的编程能力。

这是同学们为数不多能学习并编写较大桌面应用程序的机会，和在OJ上写代码完全不一样，请同学们认真对待，抓住这次机会。在本次大作业实验中，同学们将掌握以下能力：

- 使用Win32 API来编写桌面应用程序，理解事件驱动的程序设计方式
- 了解预编译的相关内容（宏），熟练使用结构体和指针，掌握Debug方式
- 如何进行多文件编程，如何编写头文件和源文件，如何高效、内聚地组织代码
- 了解最基本的游戏运行逻辑，体会一个简单的游戏框架
- 大幅度提高编程能力，将自身所想实际实现出来

0、写在前面

在开始大作业之前，请同学先学习以下前置知识。

1）文档查询（MSDN）

请同学们掌握文档查询能力，尤其是在编写大作业的过程中，可能会遇到许多没用过的函数（Win32 API函数等），这种时候可以用鼠标点击让光标停留在该函数上，然后按键盘上`F1`键，这将跳转到对应的微软MSDN文档函数说明，在文档中你将看到函数的参数、返回值，已经产生的效果。

2）预编译（宏）

预编译做的只有一件事——字符串替换！牢记这件事，看下面的例子。

> `#include`就是把头文件原封不动地粘贴到这里。
>
> 库文件`#include`用尖括号：`#include <math.h>`
>
> 自己写的头文件用引号：`#include "util.h"`
>
> `##`可以在预编译时进行字符串拼接。

```
#define ID 1
int id = ID;  // 预编译之后就是 int id = 1;

#define M
#ifdef M
int m = 1;
#else
int m = 2;
#endif
// 预编译之后就是 int m = 1;

#define ADD(a, b) a + b
int x = ADD(1, 2);  // 预编译之后就是 int x = 1 + 2;
```

```
#define CONCATENATE(STR1, STR2)  STR1##STR2
const char* str = CONCATENATE("Hello, ", "World!");
// 预编译之后就是 const char* str = "Hello, ""World!"

#define Funf(FunctionName)  FunctionName##f
Funf(print)("%d", 1);
// 预编译之后就是 printf("%d", 1);
```

> 有疑惑的同学可以自己先试一试！

3）函数先声明再调用

程序编译是自上而下的，函数应该先声明再调用，否则在调用时，并不知道函数长什么样。

```
// main.cpp
int fun(int param);  // 声明一个函数

int main() {
    fun(0);  // 声明后才能调用
    return 0;
}

// 任何.cpp
// 函数定义可以在任何地方
int fun(int param) {
    printf("%d", param);
    return 0;
}
```

函数必须先声明（原型）再调用，函数定义（实现）可以在**任何地方**，包括声明时直接定义，或者定义在另一个`.cpp`文件中，这是多文件编程的基础。

声明只是为了能正常**编译**，`.cpp`文件**编译**后还会进行**链接**，此时会把函数调用定位到具体的定义位置，因此函数定义可以放在任何地方。

> 感兴趣的可以去整体了解下 预编译->编译->链接 可执行文件生成过程。

头文件与源文件

因为函数必须先声明再调用，而函数定义可以在任何地方，所有在多文件编程中，通常把函数的声明放在头文件里，而定义放在源文件里。

这样，任何要调用该函数的源文件，只需要包含该函数声明的头文件即可。

例如：

```
// fun.h
void fun();

// fun.cpp
void fun() {}

// main.cpp
#include "fun.h"

int main() {
    fun();
}
```

再次提醒，预编译指令`#include`只做了复制粘贴，`main.cpp`实际上预编译后就是以下内容。

```
// main.cpp
void fun();

int main() {
    fun();
}
```

4）语言关键字

`static`关键字

`static`关键字在不同地方起到的作用不一样：

`static`关键字如果在函数里，将声明一个静态变量，只在函数第一次运行的时候执行变量初始化，后面变量将不再初始化，可以用这个特性，在多次调用函数时共享某个值。

> 实质上类似全局变量，但只能在函数内部访问。

```
int fun() {
    static int a = 0;  // 初始化只进行一次!
    a = a + 1;
    return a; // 返回的结果将是1、2、3、4、5、……
}
```

`static`关键字如果作用于全局变量或者函数声明之前，将限制该全局变量或者函数只能在本`.cpp`文件中调用，可以用这个特性来阻止其他文件调用某函数。

> 大作业框架中使用了这个特性，同学们能看懂即可，是否使用不作强制要求。

```
static int a = 0;  // 这个全局变量在其他文件中不能访问
int b = 0;  // 这个全局变量可以在其他文件中访问

static void fun1();  // 这个函数不能在其他文件中调用
void fun2();  // 这个函数可以在其他文件中调用
```

`extern`关键字

`extern`关键字用来声明访问其他`.cpp`文件中的没有被`static`修饰的全局变量。

> 这个也是多文件编程的基础。

```
// A.cpp
int a = 0;

// B.cpp
extern int a;  // 和A.cpp文件中的a是同一个变量，指向相同的内存空间，修改任意一个，另一个的值也会变
```

`new`关键字

虽然可以用`malloc`分配空间，但是用`new`会更现代一点。

```
struct A {
    int a;
}

A* a = new A();  // 创建一个结构体
```

`delete`关键字

正如`free`对应`malloc`分配空间，用`delete`来删除`new`出来的对象。

```
A* a = new A();
delete a;

int* arr = new int[10]();
delete[] arr;
```

5）Visual Studio使用方式

声明定义快速预览

在Visual Studio中，按住键盘Ctrl键再用鼠标点击某个函数，可以快速转到这个函数的声明或者定义。

或者右键点击这个函数，选择“转到声明”或者“转到定义”，也可以快速切换。

或者按键盘上F12按键，可以快速切换

查找所有引用

右击某个函数，选择“查找所有引用”可以快速查找该函数的所有引用。

类似地，可以右击某个函数，可以“查找调用层次结构”、“速览、切换标题/代码文件”等。

> Ctrl + K 然后 T、Ctrl + K 然后 J、Ctrl + K 然后 O 等快捷键都可以试试看

编译运行调试

编译 - Ctrl + Shift + B

开始执行（不调试）- Ctrl + F5

开始调试 - F5（**建议使用这个，如果出错能得到更多信息**）

断点 - F9 或者 点击代码行最左边

逐行运行（步过） - F10

逐行运行（步入） - F11

可以触摸变量查看变量的值，或者在下面变量窗口查看

> 以上快捷键同学们可以尽量熟悉，提高效率

预编译头加速

预编译头是Visual Studio提供的一项功能，通过把一系列通常不变的头文件整合到一个头文件里单独编译，以此加速，参考该文档。

大作业框架使用了预编译头加速功能，因此所有的源文件第一个包含的头文件必须是：

```
#include "stdafx.h"
```

6）C++ STL

大作业框架为了方便不可避免地使用了一些C++ STL相关的内容，包括：

- `std::set<T>`：用法，有序集合，用于存储无重复的内容，默认升序存储
- `std::vector<T>`：用法，向量，可以自动扩容的数组

虽然C++ STL不在本课程要求范围内，但是仍然估计同学了解学习。`<T>`模板参数，代表实际存储的类型。`std::`是命名空间，STL的内容大都在`std`命名空间中，命名空间目的防止函数重名。

（至少需要掌握以下内容）关于`std::set`的用法：

```
#include <set>

int main() {
    std::set<int> s;
    s.insert(3);  // 插入一个元素
    s.insert(4);  // 插入一个元素
    s.erase(3);  // 删除一个元素
    int n = s.size();  // 集合大小
    // 遍历方式（将是升序的顺序）
    for (int num : s) {
        printf("%d\n", num);
    }
}
```

（至少需要掌握以下内容）关于`std::vector`的用法：

```
#include <vector>

int main() {
    std::vector<int> v;
```

```
    v.push_back(3);  // 往最后插入一个元素
    v.push_back(4);  // 往最后插入一个元素
    v.pop_back(3);  // 弹出最后一个元素
    int n = v.size(); // 数组大小
    int a = v[0];  // 第i个元素（下标从0开始）
    // 遍历方式1
    for (int num : v) {
        printf("%d\n", num);
    }
    // 遍历方式2
    for (int i = 0; i < n; i++) {
        printf("%d\n", v[i]);
    }
}
```

7）所谓游戏

游戏究根到底并没有同学想的那么复杂，不过是按一定的时间间隔更新游戏逻辑并把游戏画面在屏幕上绘制出来。同学们要做的就是游戏逻辑如何更新，与游戏画面如何绘制。于是这里有一些术语需要提前了解：

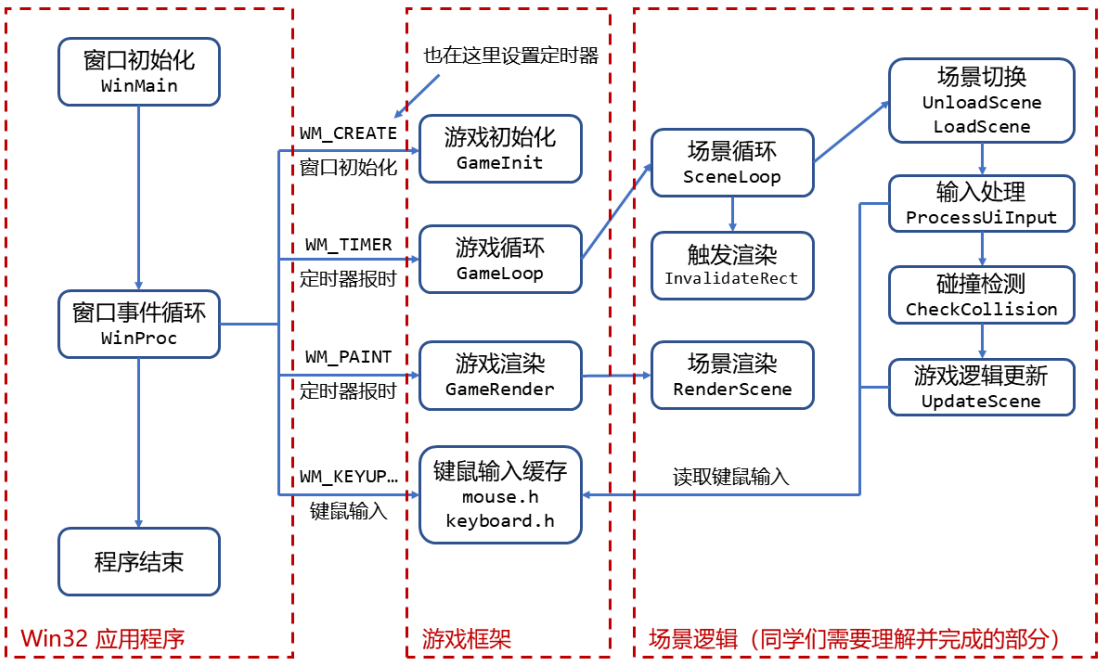
- 帧：某一时刻的画面
- 帧率：1秒中有多少画面
- 帧间隔时间（帧耗时）：顾名思义，相邻两帧的间隔时间
- 游戏时间：游戏实际运行的时间，打开游戏的时刻定为0
- 场景：当前的所有内容认为是包含在一个场景里

游戏很卡就是指帧率低，帧率低的原因是帧耗时过长，可能是这一帧更新游戏逻辑耗时过长，也可能是绘制画面耗时过长。

一、框架预览

同学们下载完框架代码后，可以双击`Game.sln`打开解决方案。在“解决方案管理器”中，你可以看到头文件和源文件分别创建了一些筛选器（类似文件夹），方便同学理解不同的代码文件所属的分类。

1）框架结构



这个大作业框架的代码结构如上图所示，分为Win32应用程序部分、游戏框架部分和场景逻辑部分，接下来逐个进行介绍。请注意，这部分代码可能略复杂，但分层设计的思想是解耦的重要方式，对大家编程能力和思想大有脾益。

不失一般性，下面在提到`XXX`文件的时候，将同时包括`XXX.h`、`XXX.cpp`，`XXX.h`中是函数的声明，`XXX.cpp`中是函数的定义。

2）Win32 应用程序层

学习这一部分，请看`main.cpp`文件。这个文件绝大部分内容都是Visual Studio在创建"Windows桌面应用程序"时自动生成的，进行了一系列Windows桌面应用程序的初始化操作。

`WinMain`函数是Windows桌面应用程序的入口函数，对应于之前写控制台程序时的`main`函数，这个函数只需要关注一个部分，也就是主消息循环：

```
// main.cpp

// 主消息循环:
while (GetMessage(&msg, nullptr, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

这个函数表示了程序和Windows操作系统的交互，也是**事件驱动**的程序设计模式。简单来讲就是，程序轮询操作系统，是否有某个事件发生，如果发生了某个事件，就对这个事件进行响应。

实际的事件响应函数是`WndProc`函数，它的函数原型是：

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
```

当操作系统检测到事件发生时，就会调用这个函数对事件进行响应，其中的`HWND`函数是窗口句柄（指针），`message`是消息的类型，`wParam`和`lParam`是消息对应的内容，至于具体如何解释这个内容，请选择对应的消息类型并按`F1`查看文档。

`WndProc`函数用`switch-case`对类型进行了选择，我们可以看到以下几个比较重要的事件类型：

- `WM_CREATE`：这是窗口创建后的事件，在这里我们调用了游戏初始化`GameInit`函数，并设置定时器
- `WM_TIMER`：定时器会以固定间隔产生该定时事件，我们在定时事件中调用游戏主循环函数`GameLoop`，**计算一帧**
- `WM_PAINT`：这是绘图事件，我们在该事件中绘制游戏画面，调用游戏渲染函数`GameRender`
- `WM_KEY_DOWN`、`WM_KEY_OUT`、`WM_MOUSEMOVE`、`WM_LBUTTONDOWN`、`WM_LBUTTONUP`：这是一系列键盘和鼠标事件，我们在对应事件中设置键鼠输入

除此之外的事件类型我们在框架中没有使用，比如`WM_COMMAND`、`WM_DESTROY`等，如果需要使用这些事件也可以查看文档，至于其他没有出现但可用的事件，也请查看文档。

以下是Win32应用程序层的其他相关文件：

- `info.h`和`main.cpp`中其他的一些额外函数处理了状态栏和日志输出的内容，有兴趣的同学可以自行学习了解。
- `timer`中配置了定时器相关的内容，有兴趣的同学可以自行学习了解。

- `stdafx`是预编译加速文件，把一些公共的头文件都放在了里面，`targetver.h`是Visual Studio自动生成的文件。
- `config.h`是配置文件，会定义一些程序的属性，比如窗口大小等

顺便强调一下，因为涉及中文，游戏中所有用到的字符串类型并非`char`而是`TCHAR`，所有字符串字面量都需要用`TEXT`宏包围起来，例如`TEXT("游戏开始")`。

在`WndProc`函数中，我们调用了游戏框架层对应的函数，接下来我们移步到游戏框架层。

3）游戏框架层

游戏框架层，主要包括`core`、`resource.cpp`、`keyboard`和`mouse`。

游戏时间

在`core`中，实现了一套游戏时间管理，包括游戏时间和帧间隔的统计以及帧率的显示等等，这部分有兴趣的同学可以自行学习了解。

游戏初始化

游戏初始化函数`GameInit`定义在`core`中，函数比较短，抛开游戏时间统计，目前只做了两件事：

```
// core.cpp

void GameInit(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    // 初始化游戏资源
    GameResourceInit(hWnd, wParam, lParam);
    // 切换到开始场景
    ChangeScene(StartScene);
}
```

游戏资源初始化函数`GameResourceInit()`定义在`resource.cpp`中，这个函数负责初始化游戏中用到的资源，例如图片等，另外一些字符串资源也可以放在这里。

```
// resource.cpp

// 图片资源
HBITMAP bmp_StartButton;      // 开始按钮图片

void GameResourceInit(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    bmp_WhiteBackground = CreateBackground(hWnd, RGB(255, 255, 255));

    // TODO: 引入其他的静态资源
}
```

关于图片资源的使用更多内容请参考附一。

初始化游戏资源后会切换到开始场景。

游戏循环

游戏循环函数`GameLoop`定义在`core`中，函数中会统计帧耗时并刷新帧率显示。除此之外，调用`SceneLoop`把控制权交给场景层，最后发送更新游戏画面的通知。

```
// core.cpp

void GameLoop(HWND hWnd)
{
    // 场景循环更新
    SceneLoop();

    // 最后进行渲染，实际的渲染函数是GameRender，只重绘画面部分
    RECT rect = {0, 0, WINDOW_WIDTH, WINDOW_HEIGHT};
    InvalidateRect(hWnd, &rect, FALSE);
}
```

`InvalidateRect`是Win32的API函数，调用这个函数后，操作系统便会产生一个`WM_PAINT`事件，在`WM_PAINT`事件里再调用`GameRender`。

游戏渲染

游戏渲染函数`GameRender`定义在`core`中，函数会初始化Win32绘图的基本内容，比如画笔等，然后创建两个缓存，一个用来绘制画面，一个用来加载图片；绘制完场景之后，把缓存交换到屏幕上（也就是双缓冲绘图），再完成收尾工作。

```
// core.cpp

void GameRender(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    // 开始绘制
    PAINTSTRUCT ps;
    HDC hdc_window = BeginPaint(hWnd, &ps);

    // 创建缓存
    HDC hdc_memBuffer = CreateCompatibleDC(hdc_window);
    HDC hdc_loadBmp = CreateCompatibleDC(hdc_window);

    // 初始化缓存
    HBITMAP blankBmp = CreateCompatibleBitmap(hdc_window, WINDOW_WIDTH,
WINDOW_HEIGHT);
    SelectObject(hdc_memBuffer, blankBmp);

    // 清空背景
    SelectObject(hdc_loadBmp, bmp_WhiteBackground);
    BitBlt(hdc_memBuffer, 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT, hdc_loadBmp, 0, 0,
SRCCOPY);

    // 绘制场景
    RenderScene(hdc_memBuffer, hdc_loadBmp);

    // 最后将所有的信息绘制到屏幕上
    BitBlt(hdc_window, 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT, hdc_memBuffer, 0, 0,
SRCCOPY);

    // 回收资源所占的内存（非常重要）
```

```
    DeleteObject(blankBmp);
    DeleteDC(hdc_loadBmp);
    DeleteDC(hdc_memBuffer);

    // 结束绘制
    EndPaint(hWnd, &ps);
}
```

在期间，调用`RenderScene`把控制权交给场景层。

关于绘制画面的更多内容请参考附一，这里同学们记住一点，在绘制画面时，坐标原点位于左上角，`x`轴朝右，`y`轴朝下。

键盘、鼠标

键盘相关函数定义在`keyboard`中、鼠标相关函数定义在`mouse`中，比较简单，同学自行了解即可。

```
// keyboard.h

// Windows应用程序框架的写入函数
void KeyDown(HWND hWnd, WPARAM wParam, LPARAM lParam);
void KeyUp(HWND hWnd, WPARAM wParam, LPARAM lParam);

// 游戏逻辑的读取函数
bool GetKeyDown(int keycode);
```

```
// mouse.h

// Windows应用程序框架的写入函数
void MouseMove(HWND hWnd, WPARAM wParam, LPARAM lParam);
void LButtonDown(HWND hWnd, WPARAM wParam, LPARAM lParam);
void LButtonUp(HWND hWnd, WPARAM wParam, LPARAM lParam);

// 游戏逻辑的读取函数
bool IsMouseLButtonDown();
int GetMouseX();
int GetMouseY();
```

4）场景逻辑层

场景层的函数定义在`scene`中，而实际每个场景的函数实现在`scene1`、`scene2`中。

场景有一个结构体用来保存当前场景的信息，可以通过`GetCurrentScene()`来获取。其中，`SceneId`枚举类型定义了游戏中的各种场景。

```
// scene.h

// 游戏场景
enum SceneId
{
    None = 0,       // 没有场景
    StartScene = 1, // 开始场景
    GameScene = 2   // 游戏场景
};

struct Scene
```

```
{
    SceneId sceneId; // 游戏场景的编号

    // TODO: 如果场景需要保存更多信息，添加在这里
};

// 获取当前场景
Scene *GetCurrentScene();
```

场景循环

对于场景循环函数 `SceneLoop` 和场景渲染函数 `RenderScene`，会依次调用 `LoadScene`、`UnloadScene`、`ProcessUiInput`、`CheckCollision` 和 `UpdateScene` 和 `RenderScene`。

```
// scene.cpp

// 场景循环
void SceneLoop(double deltaTime)
{
    if (_newSceneId != None)
    {
        // 卸载当前场景
        ROUTE_SCENE_FUNCTION(UnloadScene);
        // 切换场景ID
        currentScene->sceneId = _newSceneId;
        _newSceneId = None;
        // 加载新场景
        ROUTE_SCENE_FUNCTION(LoadScene);
    }

    // 先处理UI输入
    ROUTE_SCENE_FUNCTION(ProcessUiInput);

    // 再计算碰撞
    ROUTE_SCENE_FUNCTION(CheckCollision);

    // 然后运行游戏逻辑
    ROUTE_SCENE_FUNCTION_OneParam(UpdateScene, deltaTime);
}

// 渲染场景
void RenderScene(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    ROUTE_SCENE_FUNCTION_TwoParam(RenderScene, hdc_memBuffer, hdc_loadBmp);
}
```

每一个函数都有其特定的含义：

- `LoadScene`：在场景加载时应该做的初始化操作，比如创建一些对象、初始化一些变量等
- `UnloadScene`：在场景卸载时应该做的资源清理操作，比如销毁一些对象等
- `ProcessUiInput`：在场景循环开始的时候先判断当前的Ui操作，例如按钮点击等
- `CheckCollision`：在场景循环中接着判断当前场景中的游戏对象是否产生碰撞，如果碰撞应该跑什么逻辑
- `UpdateScene`：在场景循环中最后更新当前场景中的游戏对象，例如移动等
- `RenderScene`：在场景中如何绘制当前场景

这里的`ROUTE_SCENE_FUNCTION`是一个宏函数：

```
// 宏函数 - 路由场景函数调用，如果有新的场景需要添加，在这里添加对应的 case 分支
#define ROUTE_SCENE_FUNCTION(FUNCTION_NAME) \
    switch (GetCurrentScene()->sceneId)     \
    {                                       \
    case None:                              \
        break;                              \
    case StartScene:                        \
        FUNCTION_NAME##_StartScene();       \
        break;                              \
    case GameScene:                         \
        FUNCTION_NAME##_GameScene();        \
        break;                              \
    default:                                \
        break;                              \
    }
```

> 记住宏函数是字符串替换。

比如上面`SceneLoop`中的片段：

```
if (_newSceneId != None)
{
    // 卸载当前场景
    ROUTE_SCENE_FUNCTION(UnloadScene);
    // 切换场景ID
    currentScene->sceneId = _newSceneId;
    _newSceneId = None;
    // 加载新场景
    ROUTE_SCENE_FUNCTION(LoadScene);
}
```

在经过宏展开后等价于：

```
 if (_newSceneId != None)
 {
   // 卸载当前场景
-  ROUTE_SCENE_FUNCTION(UnloadScene);
+  switch (GetCurrentScene()->sceneId)
+  {
+    case None:
+      break;
+    case StartScene:
+      UnloadScene_StartScene();
+      break;
+    case GameScene:
+      UnloadScene_GameScene();
+      break;
+    default:
+      break;
+  }
   // 切换场景ID
   currentScene->sceneId = _newSceneId;
   _newSceneId = None;
   // 加载新场景
```

```
-   ROUTE_SCENE_FUNCTION(LoadScene);
+   switch (GetCurrentScene()->sceneId)
+   {
+     case None:
+       break;
+     case StartScene:
+       LoadScene_StartScene();
+       break;
+     case GameScene:
+       LoadScene_GameScene();
+       break;
+     default:
+       break;
+   }
  }
```

然后实际调用的函数就是`scene1`、`scene2`里面对应名称的函数。

切换场景

切换场景时就是设置`_newSceneId`，然后在下一帧最开始的时候切换。

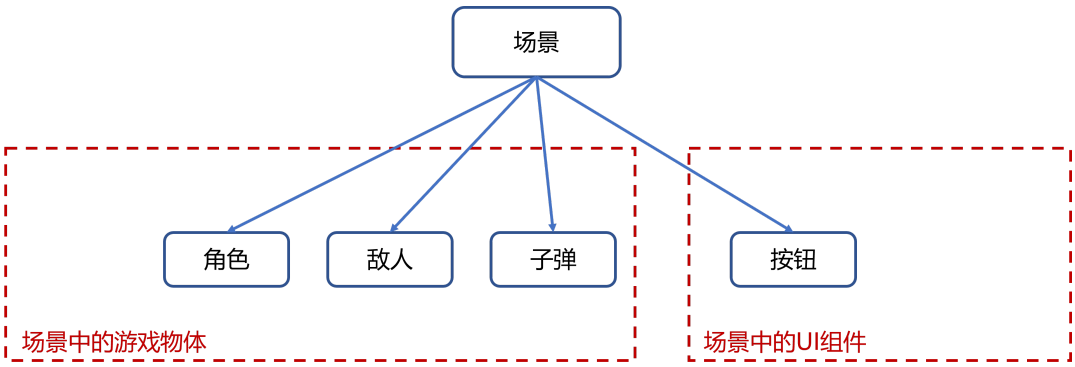
```
// scene.cpp

// 切换场景
void ChangeScene(SceneId newSceneId)
{
    _newSceneId = newSceneId;
}
```

每个场景

每个场景都必须在头文件声明相关的函数并进行实现，例如框架中的开始游戏场景。

```
// scene.h
// 加载场景
void LoadScene_StartScene();

// 卸载场景
void UnloadScene_StartScene();

// 处理UI输入
void ProcessUiInput_StartScene();

// 碰撞检测
void CheckCollision_StartScene();

// 更新场景
void UpdateScene_StartScene(double deltaTime);

// 渲染场景
void RenderScene_StartScene(HDC hdc_memBuffer, HDC hdc_loadBmp);
```

在这六个函数中，都需要同时考虑游戏对象和UI组件两个部分（除了`ProcessUiInput`只针对Ui、`CheckCollision`只针对游戏对象外），例如对于目前的游戏场景，结构如下图所示：



场景要负责这些物体的创建、销毁、碰撞检测、更新和渲染等等

其中更新函数为：

```
// scene2.cpp
void UpdateScene_GameScene(double deltaTime)
{
    /* UI组件更新 */

    /* 游戏对象更新 */
    // 更新角色对象
    UpdatePlayer(deltaTime);
    // 更新敌人对象
    UpdateEnemies(deltaTime);
    // 更新子弹对象
    UpdateBullets(deltaTime);
    // TODO: 游戏场景中需要更新的游戏对象
}
```

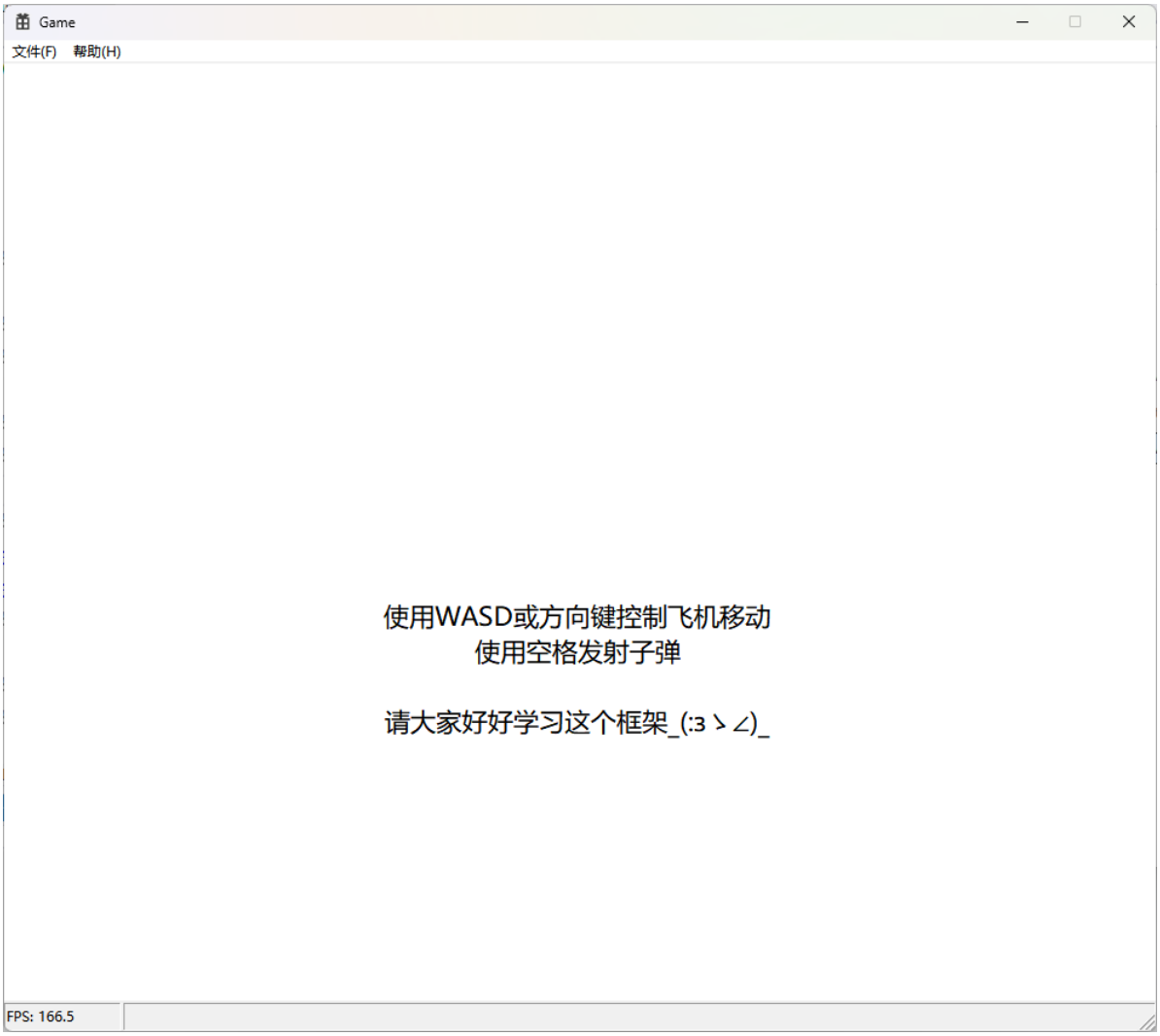
游戏对象和UI组件

现在每一个游戏对象和和UI组件的逻辑都写在了对应的文件里，例如`button`、`player`、`enemy`、`bullet`，这样编写程序可以高内聚、低耦合，逻辑也更加清晰，同学们可以看一下这些文件，在接下来的章节中，将带着大家编写出一个最简单的《飞机大战》游戏。

二、上手游戏

先试着编译当前的项目吧，这应该会成功，如果不成功你可能得安抚下你的电脑了。

此时运行程序，你将看到以下画面。



1）加上开始按钮

这个场景是开始场景`scene1`，当游戏在`GameInit`中初始化资源完成后，就切换到了这个场景`StartScene`。

关注`scene1.cpp`中的六个函数，现在场景中有一段文字，这是在`RenderScene_StartScene`函数中画出来。

现在让我们添加一个开始按钮，先思考按钮应该在什么时候添加？我们可以在加载开始场景`LoadScene_StartScene`中创建一个按钮。

创建按钮

按钮的创建可以直接使用`CreateButton`函数，这定义在`button`中，你需要在该文件中`#include "button.h"`来提供这些按钮函数的声明，然后你就可以创建一个按钮。框架里已经设置了按钮的位置，创建在这个位置即可。

```
// scene1.cpp
void LoadScene_StartScene()
{
    // 创建按钮
    const int width = 300;
    const int height = 200;
    const int x = (WINDOW_WIDTH - width) / 2 - 10;
    const int y = 196;
    ButtonId startButtonId = CreateButton(x, y, width, height,
RenderStartButton, OnStartButtonClick);
    EnableButton(startButtonId);
}
```

`CreateButton`函数需要传入六个参数：

- `x`：`x`坐标
- `y`：`y`坐标
- `width`：宽度
- `height`：高度
- `renderButtonFunc`：渲染函数（函数指针）
- `onClickButtonFunc`：点击函数（函数指针）

`RenderStartButton`和`OnStartButtonClick`定义在了同文件的`#pragma region 按钮逻辑`中，这个预编译指令只是方便折叠代码用的。这两个函数分别实现该按钮如何绘制与点击之后如何响应，现在这两个函数都是空的，我们会在后面再介绍如何实现这两个函数。需要注意的是，创建每个按钮都需要定义渲染函数和点击函数，来针对性处理。

`CreateButton`函数会返回一个`ButtonId`，如果其他地方需要用到的话，你可以用一个全局变量把它保存下来。现在只需要在接下来的启用按钮中使用这个ID，因此用局部变量保存即可。至此，我们创建了一个按钮并启用了它。

销毁按钮

有创建就有销毁，必须时刻牢记，关注对象的生命周期。在加载场景时创建，就立刻在卸载场景`UnloadScene`时把销毁。这里`button`提供了一个`DestroyButtons`函数来销毁创建的所有按钮，调用即可。

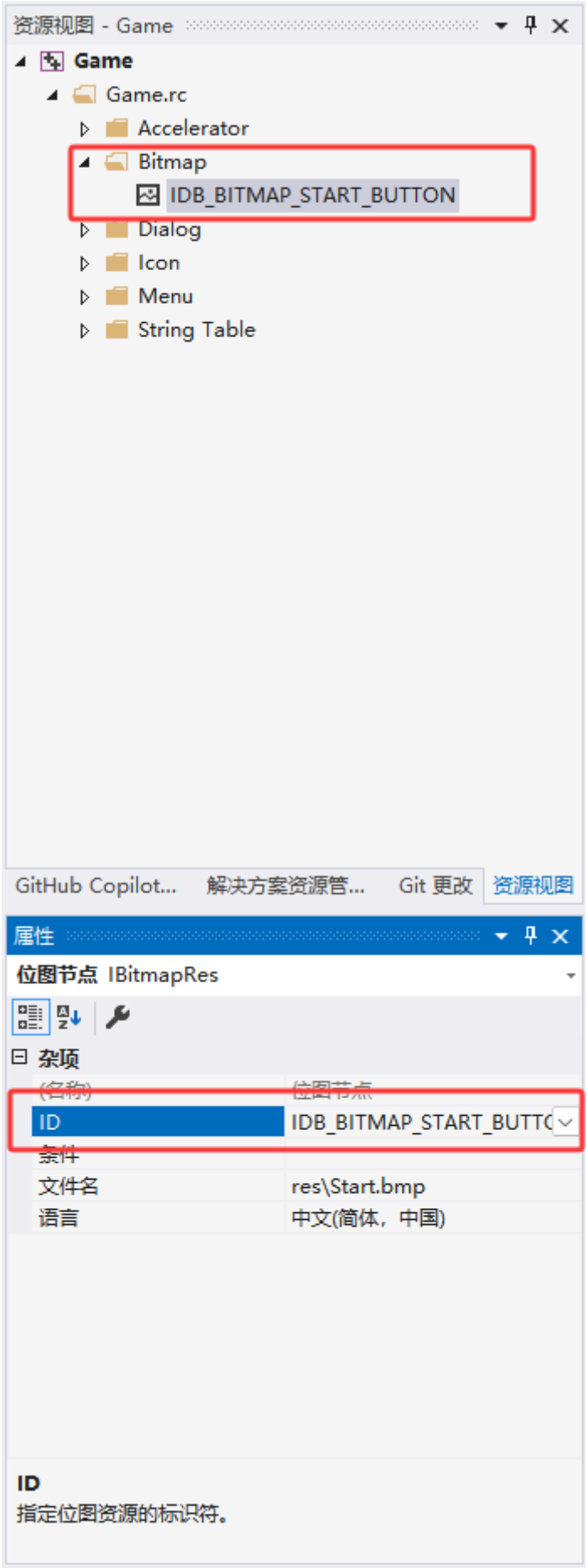
```
// scene1.cpp
void UnloadScene_StartScene()
{
    // 销毁按钮
    DestroyButtons();
}
```

渲染按钮

现在按钮确实已经被创建了出来，但是你运行程序仍然看不到。这是当然的，因为你还没有把它在画面中画出来。接下来，让我们把它画出来。渲染按钮需要我们实现按钮的渲染函数`RenderStartButton`，在`scene1`中找到这个函数，然后实现绘制按钮的逻辑。

在绘制按钮之前，建议先了解学习一下附一部分。

首先双击`Game.rc`打开资源管理器，然后右击最上面的`Game.rc`添加资源，选择Bitmap，导入按钮图片的位图资源`Start.bmp`，在`res`文件夹下面。然后在Bitmap筛选器下选中刚才添加的位图，在下方的属性栏内把ID改为`IDB_BITMAP_START_BUTTON`。



> 其实改ID并非必要，但是更好找到自己想要的图片。

这样，就导入了开始按钮的图片资源。接下来，需要在程序中加载该图片资源，转到`resource.cpp`文件中的初始化资源函数`GameResourceInit`函数，把资源加载并赋值到已经准备好的变量`bmp_StartButton`中：

```
// resource.cpp
HBITMAP bmp_StartButton;      // 开始按钮图片

void GameResourceInit(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    bmp_StartButton = LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,
MAKEINTRESOURCE(IDB_BITMAP_START_BUTTON));

    // TODO: 引入其他的静态资源
}
```

至此，该图片资源便可以在程序中调用了。现在让我们实现开始按钮的渲染函数`RenderStartButton`，这个函数有两个参数，分别是`hdc_memBuffer`和`hdc_loadBmp`，表示我们要绘制到的缓冲区和用来加载位图的缓冲区。注意到在这个函数前面我们使用`extern`关键字声明了在`resource.cpp`中定义的位图资源变量，这使得我们可以使用该位图。绘制函数如下：

```
// scene1.cpp
extern HBITMAP bmp_StartButton;

void RenderStartButton(Button *button, HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // TODO: 绘制开始按钮
    SelectObject(hdc_loadBmp, bmp_StartButton);
    TransparentBlt(
        hdc_memBuffer, (int)button->position.x, (int)button->position.y,
        button->width, button->height,
        hdc_loadBmp, 0, 0, button->width, button->height,
        RGB(255, 255, 255));
}
```

我们用`hdc_loadBmp`加载位图，然后绘制到`hdc_memBuffer`上，具体的位置和大小在创建按钮的时候给出了。

> 注意这里的按钮大小和位图大小刚好一致，若不一致需要处理，详情请看`TransparentBlt`函数的文档。

现在运行程序你就可以看到开始按钮了，但是显然，因为我们没有实现任何按钮点击逻辑，因此点击按钮是没有用的。

注意一下，这里之所以不需要在`RenderScene`调用该函数，是因为`RenderScene`当中已经调用了`RenderButtons`，其中会自动调用我们实现的`RenderStartButton`；在大多数情况下，你需要手动在`RenderScene`当中去绘制每一个物体。

点击按钮

接下来让我们实现按钮的点击逻辑，我们希望点击按钮后能够跳转到游戏场景，在框架中是`GameScene`，即`scene2`，这就需要我们实现`OnStartButtonClick`。这个函数很简单，只是切换场景，直接`#include "scene.h"`然后调用`ChangeScene`即可。当然，如果我们想的话，也可以打印一条日志，使用`Log`函数。

```
// scene1.cpp
void OnStartButtonClick(Button *button)
{
    // TODO: 开始按钮点击事件处理
    Log(1, TEXT("游戏开始！"));
    ChangeScene(GameScene);
}
```

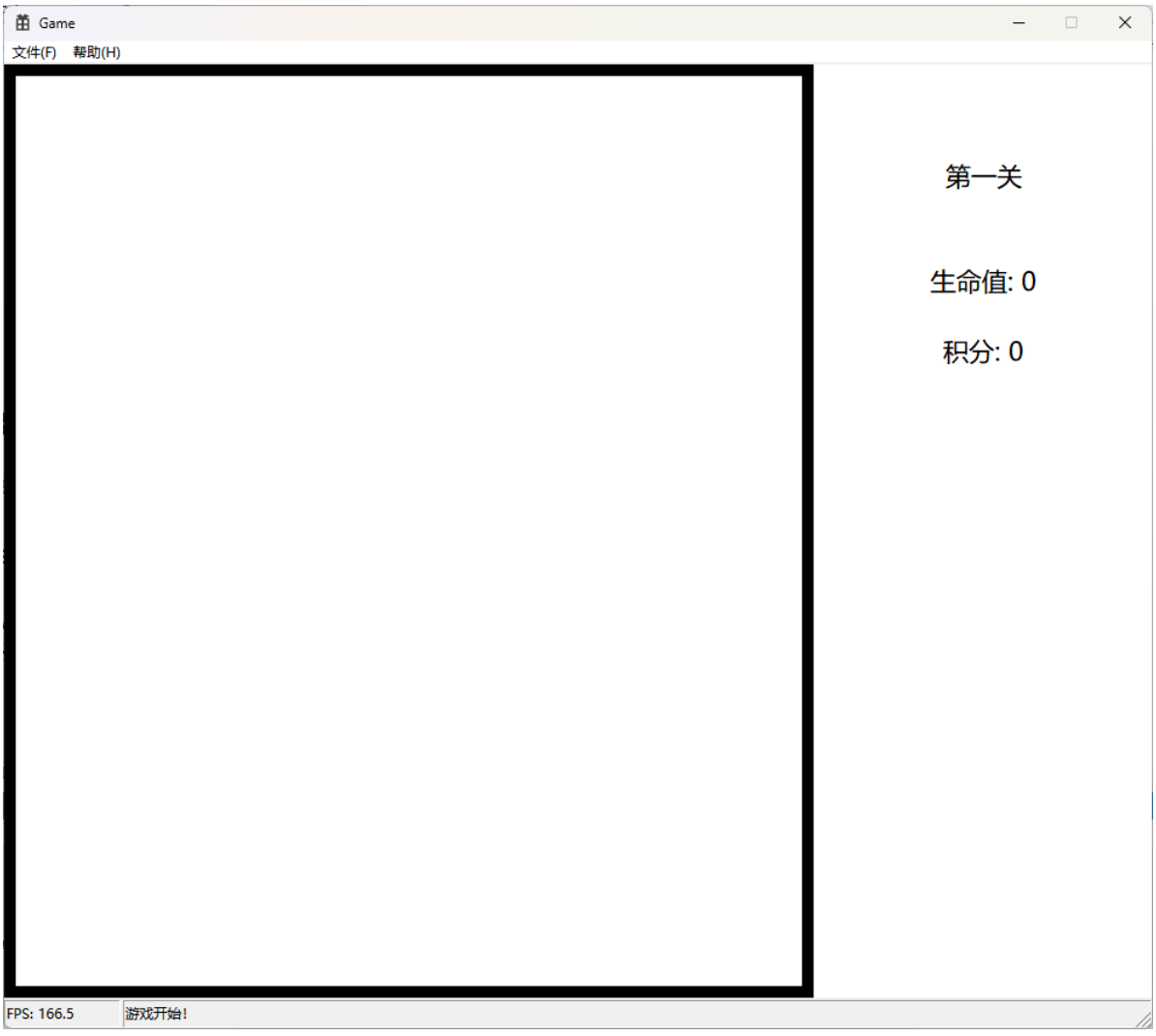
> `Log`函数用法和`printf`一致，第一个参数是状态栏的位置，现在0栏用来显示`FPS`了，所以用1栏。当然你也可以修改`config.h`中的相关配置增加新的栏。第二个参数是格式化字符串，记得用`TEXT`宏包起来。用法可以是：`Log(1, TEXT("我的分数：%d"), score)`。

现在编译运行会发现按钮仍然没有响应点击，因为我们还没有处理UI输入。让我们看向六个函数中的`processUiInput`，在这里我们应该实现按钮点击：

```
// scene1.cpp
void ProcessUiInput_StartScene()
{
    // TODO: 处理鼠标点击按钮
    if (IsMouseLButtonDown())
    {
        PressButtons(GetMouseX(), GetMouseY());
    }
}
```

这里需要`#include "mouse.h"`来获取处理鼠标输入函数声明。字面解释是，如果鼠标左键按下了，然后就尝试按鼠标位置的按钮。这些函数框架已经实现，可以Ctrl + 鼠标左键点击查看这些函数的实现方法了解学习，记住如果多个按钮重叠在一起，行为是不确定了。

至此，我们成功通过点击按钮切换到了游戏场景，但是现在游戏场景是完全空着的，我们将逐个添加游戏对象。



2）加上玩家

让我们加上玩家，同样地，从创建、销毁和绘制开始。

创建玩家

创建玩家同样在`scene2`加载场景`LoadScene`当中。

因为`player`中定义了创建玩家函数`CreatePlayer`，直接调用即可。

```
// scene2.cpp
void LoadScene_GameScene()
{
    /* 游戏对象创建 */
    // 创建玩家对象
    CreatePlayer();
}
```

至于究竟怎么创建呢，那就需要看`CreatePlayer`的实现，很简单，`new`一个新对象就行。可以在`player`头文件中查看玩家结构体的具体定义，这里我们需要给玩家对象设置一些初始值：

```
// player.cpp
void CreatePlayer()
{
    player = new Player();
    // 玩家初始位置
    player->position.x = (GAME_WIDTH - PLAYER_WIDTH) / 2;
    player->position.y = GAME_HEIGHT - PLAYER_HEIGHT - 20;
    player->width = PLAYER_WIDTH;
    player->height = PLAYER_HEIGHT;
    // 玩家初始属性
    player->attributes.health = 3;
    player->attributes.score = 0;
    player->attributes.speed = 500;
    player->attributes.maxBulletCd = 0.1;
    player->attributes.bulletCd = 0.0;
}
```

> 出生点等具体的参数涉及到窗口大小，需要计算，或者以宏方式定义了，参看`config.h`。
>
> Attribute定义在了`type.h`中，因为会被多个地方使用。

这些属性的理解就顾名思义吧。

销毁玩家

同样创建后就记得销毁，在`UnloadScene`当中销毁玩家。

```
// scene2.cpp
void UnloadScene_GameScene()
{
    /* 游戏对象销毁 */
    // 销毁角色对象
    DestroyPlayer();
}
```

`DestroyPlayer`同样定义在`player`中，查看该函数定义会发现只是删除了之前`new`出来的玩家对象，并设置成空指针。

渲染玩家

与渲染按钮类似，导入位图资源`plane.bmp`（不是`player.bmp`），在`GameResourceInit`函数中加载，然后在绘制玩家时使用。

```
// resource.cpp
HBITMAP bmp_Player;           // 角色图片
void GameResourceInit(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    // 其他的
    bmp_Player = LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,
MAKEINTRESOURCE(IDB_BITMAP_PLANE));

    // TODO: 引入其他的静态资源
}
```

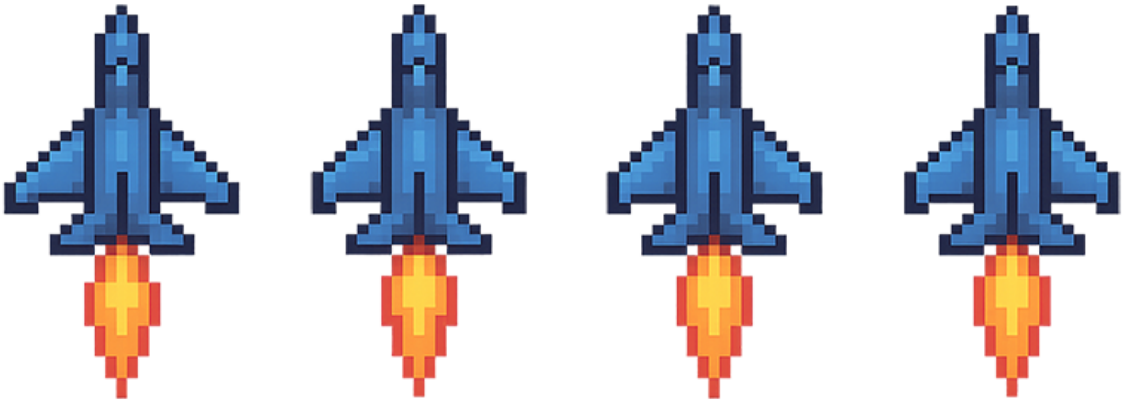
在`scene2`的`RenderScene`中，调用`player`中定义的`RenderPlayer`。

```
// scene2.cpp
void RenderScene_GameScene(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    /* 游戏对象绘制 */
    // 绘制角色对象
    RenderPlayer(hdc_memBuffer, hdc_loadBmp);
}
```

然后在`RenderPlayer`中实际去绘制角色。

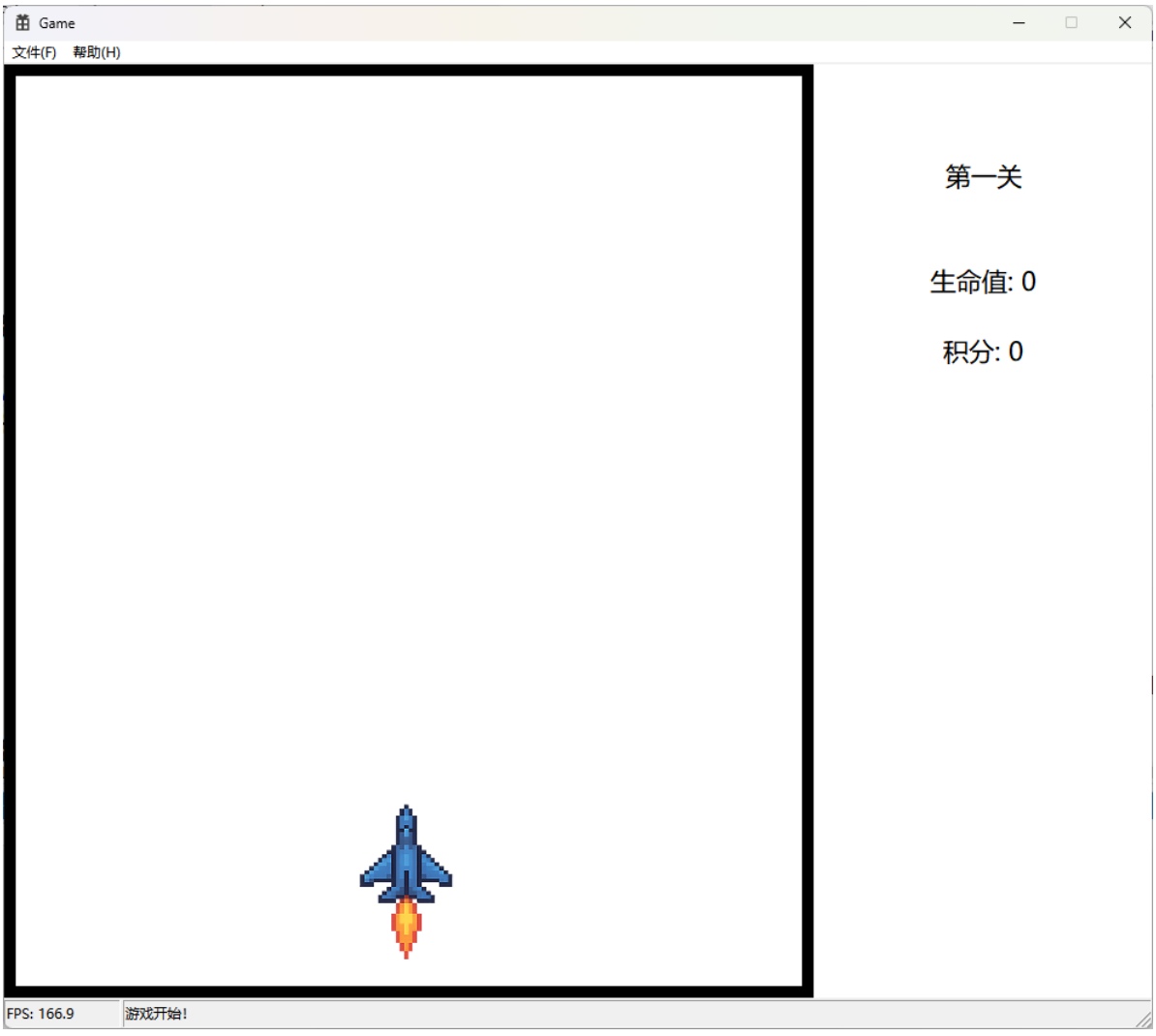
```
// player.cpp
// 渲染资源
extern HBITMAP bmp_Player;
static int frameIndex = 0;
static const int bmp_RowSize = 1;
static const int bmp_ColSize = 4;
static const int bmp_CellWidth = 200;
static const int bmp_CellHeight = 300;

// 中间是其他函数

void RenderPlayer(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // 绘制玩家
    SelectObject(hdc_loadBmp, bmp_Player);
    const int frameRowIndex = frameIndex / bmp_ColSize;
    const int frameColIndex = frameIndex % bmp_ColSize;
    TransparentBlt(
        hdc_memBuffer, (int)player->position.x, (int)player->position.y,
        player->width, player->height,
        hdc_loadBmp, frameColIndex * bmp_CellWidth, frameRowIndex *
bmp_CellHeight, bmp_CellWidth, bmp_CellHeight,
        RGB(255, 255, 255));
}
```

注意下，这里和绘制按钮不同，更复杂一些，这是因为玩家是一张**可以动起来**的序列图片，只要轮播就会动起来。



`frameIndex`指示当前播放到第几张图片单元，`bmp`相关的参数表示这张图片是1行4列，每个单元格宽200高300。

现在运行程序并点击开始按钮，就可以看到玩家飞机出现在了画面当中。



让我们让它动起来，也就是改变`frameIndex`。这个参数该在什么地方改呢？终于到了游戏的核心逻辑部分，游戏更新`UpdateScene`。这个函数游戏每帧都会执行，你应该在这个函数中改变游戏对象与时间相关的变量。

在`scene2`中添加角色更新：

```
// scene2.cpp

void UpdateScene_GameScene(double deltaTime)
{
    /* 游戏对象更新 */
    // 更新角色对象
    UpdatePlayer(deltaTime);
}
```

于是，你可以在`UpdatePlayer`中开始更新玩家角色：

```
void UpdatePlayer(double deltaTime)
{
    // 更新角色帧动画（假设1s播放完全部的动画）
    frameIndex = (int)(GetGameTime() * bmp_RowSize * bmp_ColSize) % (bmp_RowSize
* bmp_ColSize);
}
```

计算方式多种多样，这里随便写了一种。注意这里用到了`GetGameTime()`，这个函数返回游戏开始到现在的时间（单位：s），可以在任何地方调用。

现在在运行程序，你将得到一个动起来的玩家飞机。

让玩家动起来

现在真的该让玩家动起来了！让我们用WASD或者上下左右方向键控制玩家移动！这显然也是在更新玩家对象，在`UpdatePlayer`当中实现即可。判断按钮是否按下，然后根据按钮按下的结果移动玩家就可以了！

```
// scene2.cpp
void UpdatePlayer(double deltaTime)
{
    // 读取键盘输入，然后控制角色位置
    Vector2 direction = { 0, 0 };
    if (GetKeyDown(VK_W) || GetKeyDown(VK_UP))
    {
        direction.y -= 1;
    }
    if (GetKeyDown(VK_S) || GetKeyDown(VK_DOWN))
    {
        direction.y += 1;
    }
    if (GetKeyDown(VK_A) || GetKeyDown(VK_LEFT))
    {
        direction.x -= 1;
    }
    if (GetKeyDown(VK_D) || GetKeyDown(VK_RIGHT))
    {
        direction.x += 1;
    }
    // 归一化方向向量，保证所有方向移动速度一致
    direction = Normalize(direction);
    player->position.x += direction.x * player->attributes.speed * deltaTime;
    player->position.y += direction.y * player->attributes.speed * deltaTime;
}
```

请同学们理解这个地方的代码！这里用了一些技巧让玩家运动所有方向移动速度保持一致，并且用了`deltaTime`变量保证运动是均匀的。`delteTime`是相邻两帧的时间，使用这个变量就会在相邻两帧间隔较短时移动较短的距离，相邻两帧间隔较长时移动较长距离，从而保证运动是均匀的。

> 速度×时间=路程

现在运行程序，玩家飞机便可以正常运动，但是此时仍然有一个问题，就是飞机可以飞到它不该飞到的地方，因此我们需要限制玩家的运动范围，当玩家超出应用范围时掰回来。

```
// scene2.cpp

void UpdatePlayer(double deltaTime)
{
    // 限制角色在屏幕内
    if (player->position.x < 0)
    {
        player->position.x = 0;
    }
    if (player->position.x > GAME_WIDTH - player->width)
    {
        player->position.x = GAME_WIDTH - player->width;
    }
    if (player->position.y < 0)
    {
        player->position.y = 0;
    }
    if (player->position.y > GAME_HEIGHT - player->height)
    {
        player->position.y = GAME_HEIGHT - player->height;
    }
}
```

至此，你将得到一个符合预期的玩家运动逻辑。

3）加上敌人

接下来让我们在场景中加上敌人，老样子，从创建、销毁和绘制开始。

先在`scene2.cpp`中`#include "enemy.h"`吧。你可以随时在这里查看`Enemy`结构体的定义。

创建敌人

创建敌人与之前略有不同，想想《飞机大战》，场景中并不是一开始就有敌人的，而是随时间随机出现在画面的上方并向下移动。

因此我们无需在`LoadScene`中创建敌人：

```
// scene2.cpp
void LoadScene_GameScene()
{
    /* 游戏对象创建 */
    // 敌人将在游戏过程中动态创建
}
```

转而在`UpdateScene`时动态创建敌人：

```
// scene2.cpp
void UpdateScene_GameScene(double deltaTime)
{
    // 更新敌人对象
    UpdateEnemies(deltaTime);
}
```

```
// enemy.cpp
void UpdateEnemies(double deltaTime)
{
    // 每隔一定时间在随机位置创建一个敌人
    double gameTime = GetGameTime();
    if (gameTime - lastGenerateTime > deltaGenerateTime)
    {
        CreateRandomEnemy();
        lastGenerateTime = gameTime;
    }
}
```

这个函数的意思是每隔一段时间在随机位置创建敌人，当然，不使用游戏时间`GetGameTime`而直接用传入的间隔时间`DeltaTime`也可以达成相同的目标。`CreateRandomEnemy`这个函数已经实现，其实就是在随机位置生成了一个敌人。注意我们让这个敌人飞机在稍高的地方生成。

销毁敌人

同样创建后就记得销毁，这个都必须在`UnloadScene`当中做。

```
// scene2.cpp
void UnloadScene_GameScene()
{
    /* 游戏对象销毁 */
    // 清空敌人对象
    DestroyEnemies();
}
```

`DestroyEnemies`定义在`enemies`中，查看该函数定义会发现它会清空当前所有的敌人对象。其实在游戏过程中，我们既然会动态创建敌人，也肯定会动态清空敌人，所以这里在卸载场景清空的是剩余的敌人飞机，以防出现内存泄漏。

渲染敌人

与之前类似，打开`Game.rc`导入位图资源`enemy.bmp`，在`GameResourceInit`函数中加载，然后在绘制敌人时使用。

```
// resource.cpp
HBITMAP bmp_Enemy;          // 角色图片
void GameResourceInit(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    // 其他的
    bmp_Enemy = LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,
MAKEINTRESOURCE(IDB_BITMAP_ENEMY));

    // TODO: 引入其他的静态资源
}
```

在`scene2`的`RenderScene`中，调用`enemy`中定义的`RenderEnemy`。

```
// scene2.cpp
void RenderScene_GameScene(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    /* 游戏对象绘制 */
    // 绘制敌人对象
    RenderEnemy(hdc_memBuffer, hdc_loadBmp);
}
```

然后在`RenderEnemy`中实际去绘制角色。

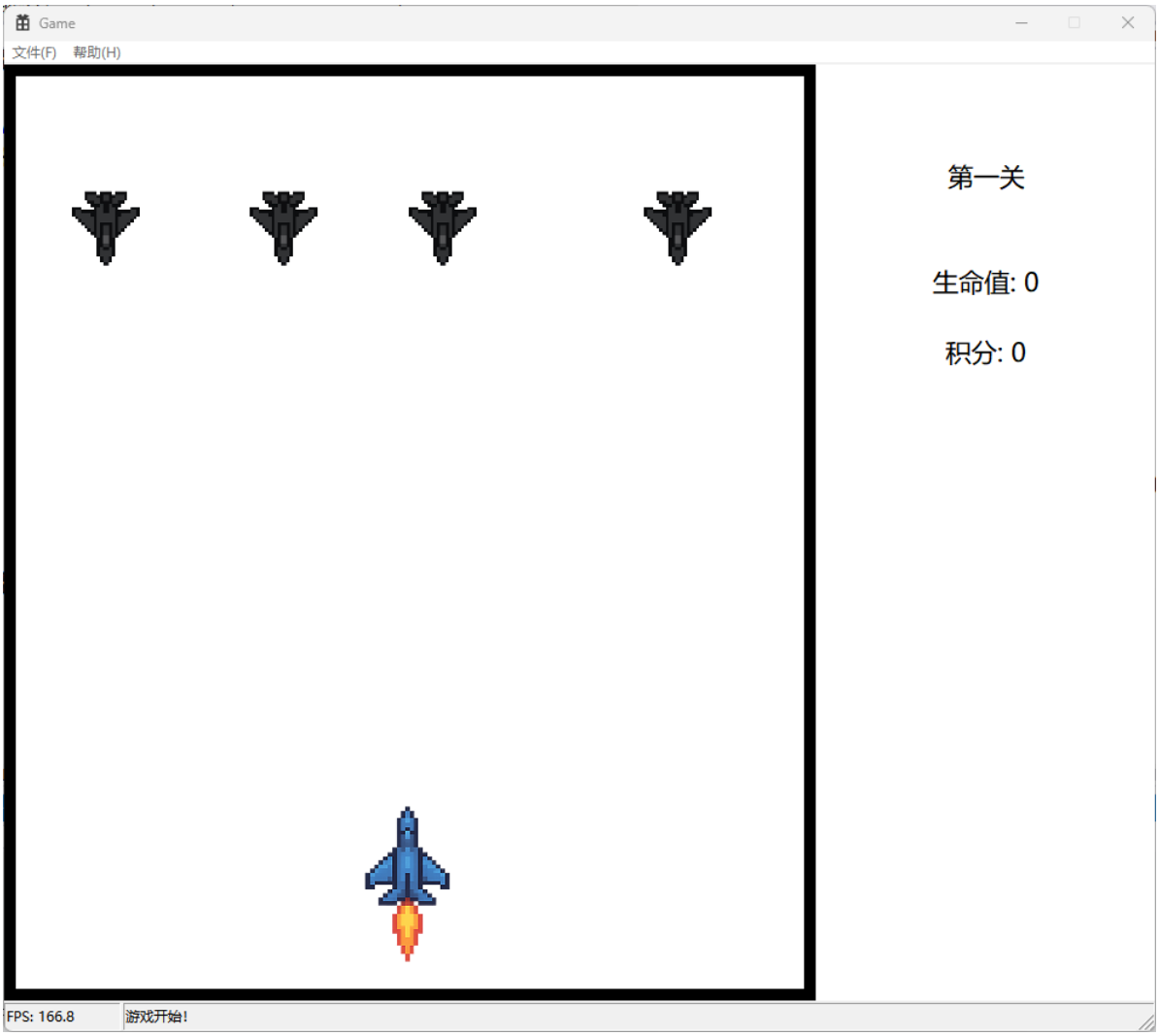
```
// enemy.cpp
// 渲染资源
extern HBITMAP bmp_Enemy;
static int frameIndex = 0;
static const int bmp_RowSize = 1;
static const int bmp_ColSize = 1;
static const int bmp_CellWidth = 200;
static const int bmp_CellHeight = 200;

// 中间是其他函数

void RenderEnemies(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // 画出敌人
    const int frameRowIndex = frameIndex / bmp_ColSize;
    const int frameColIndex = frameIndex % bmp_ColSize;
    for (Enemy* enemy : enemies)
    {
        SelectObject(hdc_loadBmp, bmp_Enemy);
        TransparentBlt(
            hdc_memBuffer, (int)enemy->position.x, (int)enemy->position.y,
            enemy->width, enemy->height,
            hdc_loadBmp, frameColIndex * bmp_CellWidth, frameRowIndex *
bmp_CellHeight, bmp_CellWidth, bmp_CellHeight,
            RGB(255, 255, 255));
    }
}
```

这里复用了绘制玩家的代码，不过因为敌机只有一张图片，所以参数设置固定就好，1行1列，`frameIndex`是0即可。

现在运行程序并点击开始按钮，但是仍然看不到敌人飞机出现在了画面当中。为什么呢，因为我们把敌机创建在了屏幕外面。

```
// enemy.cpp
void CreateRandomEnemy()
{
    CreateEnemy(
        GetRandomDouble(30, GAME_WIDTH - ENEMY_WIDTH - 30),
        -100 // 从上方稍微高一点的位置生成
    );
}
```

把这个-100改成100，你就可以看到飞机生成在画面当中了！



> 记得改回去。

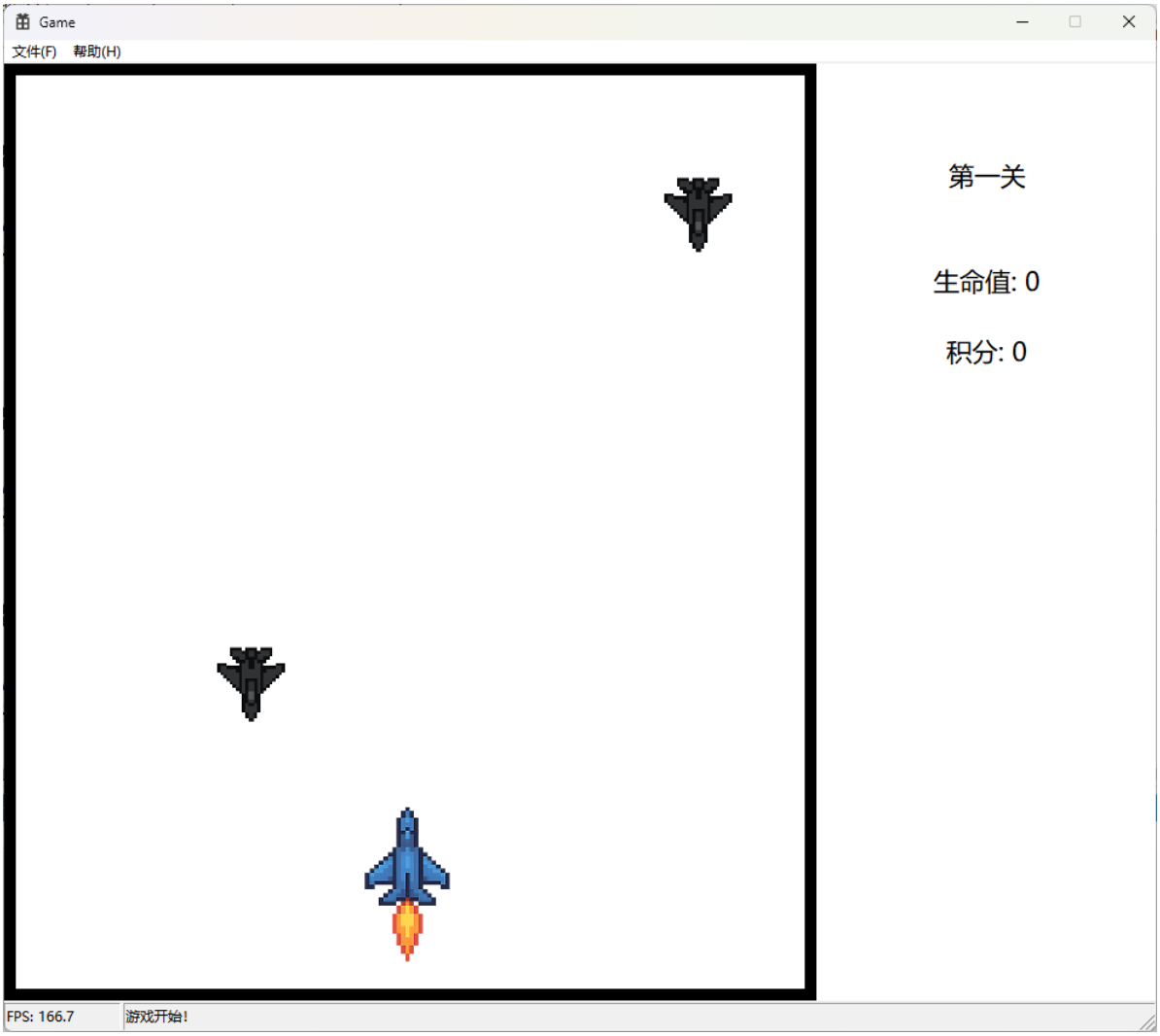
让敌机动起来

让飞机动起来的逻辑相比大家应该能猜到到在什么地方实现了，就是`UpdateEnemy`中。通常在飞机大战中，敌机就是从上方往下方匀速飞行，我们就先这样实现吧。

```
// enemy.cpp
void UpdateEnemies(double deltaTime)
{
    // 每隔一定时间在随机位置创建一个敌人
    // ......

    // 敌人的移动逻辑
    for (Enemy* enemy : GetEnemies())
    {
        // 敌人向下移动
        enemy->position.y += enemy->attributes.speed * deltaTime;
        // 超出屏幕的敌人删除
        if (enemy->position.y > GAME_HEIGHT + 50)
        {
            DestroyEnemy(enemy);
        }
    }
}
```

因为场景中有很多敌机，所以我们需要遍历所有的敌机并更新他们的位置，向下移动只要增加他们的`y`值即可。当然，我们这里还做了动态删除，如果敌机位置超出了屏幕一定距离，就把它删除。

现在运行程序，你可以看到敌人从屏幕上方一直往下移动直到屏幕下方了。



4）加上子弹

激动人心的《飞机大战》怎么可以没有子弹呢？让我们加上子弹！

记得加上`#include "bullet.h"`。你可以随时在这里查看子弹结构体`Bullet`的定义。

创建子弹

第一个问题是什么时候创建子弹？显然不是在场景加载的时候`LoadScene`创建，这里提供过一种创建方式，在玩家按空格键的时候发射子弹。

```
// scene2.cpp
void LoadScene_GameScene()
{
    /* 游戏对象创建 */
    // 初始化玩家对象
    CreatePlayer();
    // 敌人将在游戏过程中动态创建
    // 子弹将在游戏过程中动态创建
    // TODO: 游戏场景中需要创建的游戏对象
}
```

```
// player.cpp
void UpdatePlayer(double deltaTime)
{
    // 发射子弹
    if (GetKeyDown(VK_SPACE))
    {
        // 创建子弹，子弹从飞机顶部中央位置发射
        CreateBullet(
            player->position.x + player->width / 2.0,
            player->position.y,
            1,    // 伤害
            800.0 // 速度
        );
    }
}
```

子弹的位置计算请同学们理解。同样，有动态创建通常会有动态销毁，后面会看到。

销毁子弹

静态销毁仍然很重要，这是防止内存泄漏的重要方式！

总之调用`DestroyBullets`就可以，这个实现方式和`DestroyButtons`或者`DestroyEnemies`都是一样的。

```
// scene2.cpp
void UnloadScene_GameScene()
{
    /* 游戏对象销毁 */
    // 清空子弹对象
    DestroyBullets();
}
```

渲染子弹

子弹渲染就不那么麻烦了，直接当作红色的圆形绘制即可。

在`RenderScene`中加上`RenderBullets`，然后在`RenderBullets`中写好具体的逻辑。

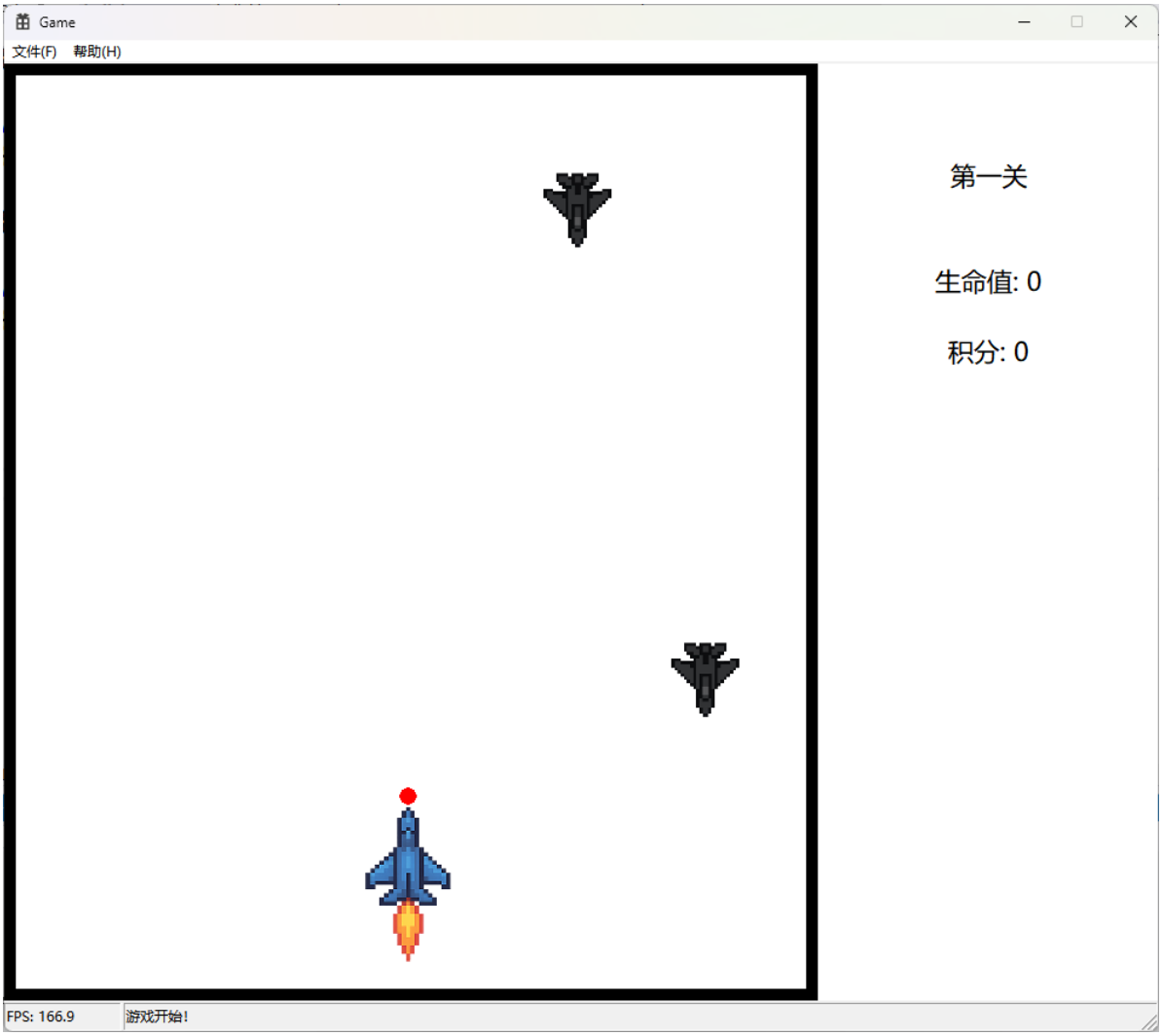
```
// scene2.cpp
void RenderScene_GameScene(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // 绘制子弹对象
    RenderBullets(hdc_memBuffer, hdc_loadBmp);
}
```

```
// bullet.cpp
void RenderBullets(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // 创建红色实心画刷
    HBRUSH hBrush = CreateSolidBrush(RGB(255, 0, 0));
    HBRUSH oldBrush = (HBRUSH)SelectObject(hdc_memBuffer, hBrush);

    // 设置画笔为空，避免边框影响
    HPEN hPen = (HPEN)SelectObject(hdc_memBuffer, GetStockObject(NULL_PEN));
```

```
    // 绘制子弹
    for (Bullet* bullet : bullets)
    {
        Ellipse(
            hdc_memBuffer,
            (int)(bullet->position.x - bullet->radius),
            (int)(bullet->position.y - bullet->radius),
            (int)(bullet->position.x + bullet->radius),
            (int)(bullet->position.y + bullet->radius));
    }

    // 恢复 GDI 对象
    SelectObject(hdc_memBuffer, oldBrush);
    SelectObject(hdc_memBuffer, hPen);
    DeleteObject(hBrush);
}
```

为什么这么画就请看官方文档吧!

现在运行，按空格键你将能看到子弹



但是子弹堆积在了同一个地方，那就让它们动起来!

让子弹动起来

敌机向下移动，子弹向上移动，是这样的。

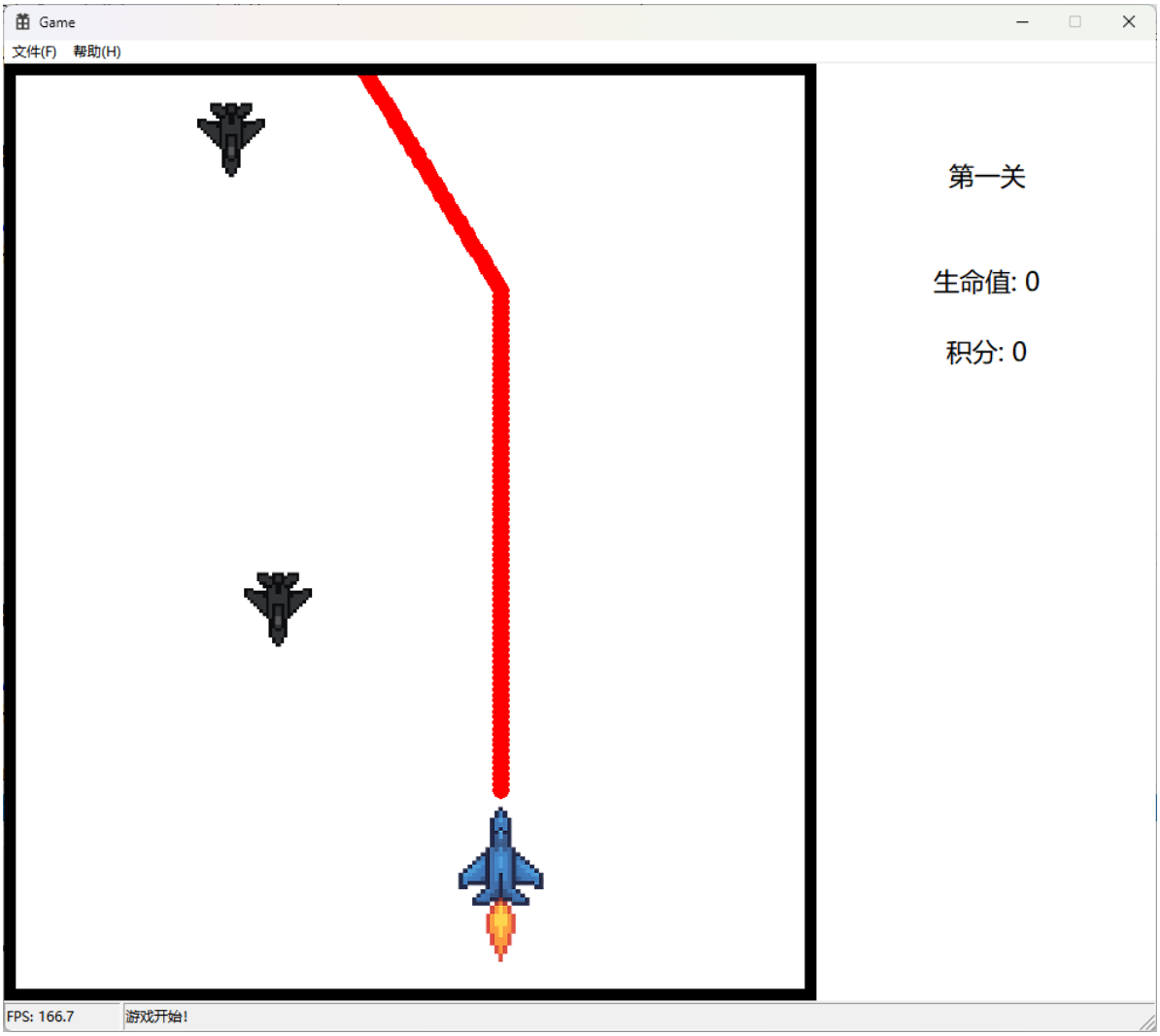
在`UpdateScene`里面加上`UpdateBullets`，然后在`UpdateBullets`中实现这个逻辑。

```
void UpdateScene_GameScene(double deltaTime)
{
    /* 游戏对象更新 */
    // 更新角色对象
    UpdatePlayer(deltaTime);
    // 更新敌人对象
    UpdateEnemies(deltaTime);
    // 更新子弹对象
    UpdateBullets(deltaTime);
    // TODO: 游戏场景中需要更新的游戏对象
}
```

```
// bullet.cpp
void UpdateBullets(double deltaTime)
{
    for (Bullet* bullet : GetBullets())
    {
        // 子弹向上移动
        bullet->position.y -= bullet->speed * deltaTime;
        // 超出屏幕的子弹删除
        if (bullet->position.y + bullet->radius < 0)
        {
            DestroyBullet(bullet);
        }
    }
}
```

和敌机移动类似，因为有很多子弹，需要遍历修改；向上移动就是`y`值减小，用`deltaTime`保证速度均匀；当子弹超出屏幕，将它销毁。

现在你终于可以发射子弹了。



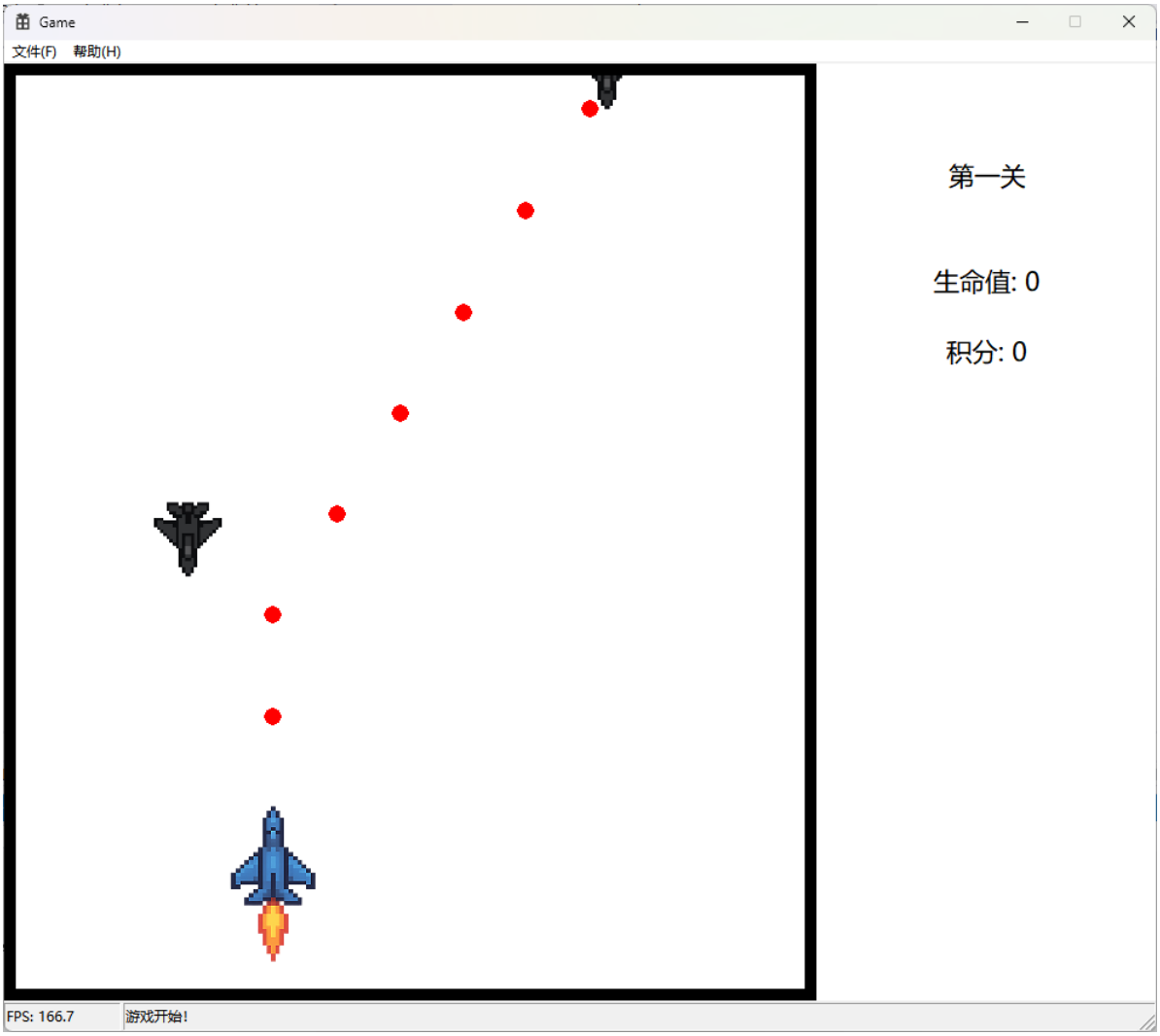
当好像哪里不太对，因为子弹发射速度太快了，那就在发射时加点限制。`Attribute`里面其实提前加了这个`CD`属性值噢。

```
// player.cpp
// 发射子弹
// player.cpp
void UpdatePlayer(double deltaTime)
{
    // 发射子弹
    if (GetKeyDown(VK_SPACE))
    {
        // 控制子弹发射间隔
        if (player->attributes.bulletCd <= 0.0)
        {
            // 创建子弹，子弹从飞机顶部中央位置发射
            CreateBullet(
                player->position.x + player->width / 2.0,
                player->position.y,
                1,    // 伤害
                800.0 // 速度
            );
            // 0.3秒发射一次
            player->attributes.bulletCd = player->attributes.maxBulletCd;
        }
        else
        {
            player->attributes.bulletCd -= deltaTime;
```

```
        }
    }
}
```

这个控制CD的例子请同学们好好理解，**定时**是游戏中很重要的一部分！

现在子弹的发射正常多了。



5）加上碰撞

现在游戏仍然没有完全成形，因为玩家、敌人和子弹都各干各的，需要把他们联系起来。

碰撞相关的函数`CheckCollision`实现在对应的场景中，可以拆成不同的函数用以区分不同的作用。

玩家与敌人

首先是玩家与敌人，如果这两者碰撞，我们可以认为玩家撞碎了敌人，获得了分数，但是会损失血量。

```
// scene2.cpp
void CheckCollision_GameScene()
{
    // 玩家和敌人的碰撞
    CheckCollision_GameScene_Player_Enemies();
}
```

再求解碰撞的时候，我们可以把玩家和敌人都抽象成简单的矩形，简化计算。一些求解碰撞的函数定义在了`util.cpp`当中。

```
// scene2.cpp

// 检查角色和敌人的碰撞
void CheckCollision_GameScene_Player_Enemies()
{
    // 玩家用简单矩形表示
    Player* player = GetPlayer();
    Rect rect1{};
    rect1.left = player->position.x;
    rect1.right = player->position.x + player->width;
    rect1.top = player->position.y;
    rect1.bottom = player->position.y + player->height;
    // 敌人用简单矩形表示
    std::vector<Enemy*> enemies = GetEnemies();
    Rect rect2{};
    for (Enemy* enemy : enemies)
    {
        rect2.left = enemy->position.x;
        rect2.right = enemy->position.x + enemy->width;
        rect2.top = enemy->position.y;
        rect2.bottom = enemy->position.y + enemy->height;
        if (IsRectRectCollision(rect1, rect2))
        {
            // 碰撞后扣血、加分摧毁敌人
            player->attributes.health--;
            player->attributes.score += enemy->attributes.score;
            DestroyEnemy(enemy);
            if (player->attributes.health <= 0)
            {
                Log(1, TEXT("游戏结束！"));
                ChangeScene(SceneId::StartScene);
            }
        }
    }
}
```

创建两个矩形，填入信息，然后调用矩形碰撞函数`IsRectRectCollision`，如果碰撞了，就扣血加分，并且摧毁敌人。这里还额外做了一件事情，就是如果玩家的血量归零，结束游戏并返回开始界面。

现在运行程序，和敌人碰撞3次之后，游戏就会结束。

敌人和子弹

既然玩家可以发射子弹，那么子弹应该可以命中敌人。如果子弹命中敌人，我们就对敌人进行扣血，销毁子弹；当敌人血量削减为0，就销毁它，并加分给玩家。

```
// scene2.cpp
void CheckCollision_GameScene()
{
    // 玩家和敌人的碰撞
    CheckCollision_GameScene_Player_Enemies();
    // 敌人和子弹的碰撞
    CheckCollision_GameScene_Enemies_Bullets();
}
```

```
// scene2.cpp
```

```
void CheckCollision_GameScene_Enemies_Bullets()
{
    // 敌人用简单矩形表示
    std::vector<Enemy *> enemies = GetEnemies();
    Rect rect{};
    // 子弹用简单圆形表示
    std::vector<Bullet *> bullets = GetBullets();
    Circle circle{};
    for (Enemy *enemy : enemies)
    {
        rect.left = enemy->position.x;
        rect.right = enemy->position.x + enemy->width;
        rect.top = enemy->position.y;
        rect.bottom = enemy->position.y + enemy->height;
        for (Bullet *bullet : bullets)
        {
            circle.center = bullet->position;
            circle.radius = bullet->radius;
            if (IsRectCircleCollision(rect, circle))
            {
                // 碰撞后扣血、加分摧毁敌人和子弹
                enemy->attributes.health -= bullet->damage;
                if (enemy->attributes.health <= 0)
                {
                    GetPlayer()->attributes.score += enemy->attributes.score;
                    DestroyEnemy(enemy);
                }
                DestroyBullet(bullet);
            }
        }
    }
}
```

这里是把敌人抽象成一个矩形，子弹抽象成一个圆形，然后计算碰撞。这里需要双层循环，要计算每一个敌人和每一个子弹的碰撞。

> 双重循环是很慢的！所以如果子弹和敌人太多，就有可能出现卡顿问题。有其他的算法可以加速求解碰撞的过程，感兴趣的同学可以自行了解。不过大作业里直接用双重循环应该会不会遇到效率问题。

现在运行程序，就可以用子弹击毁敌机了。

6）加上UI

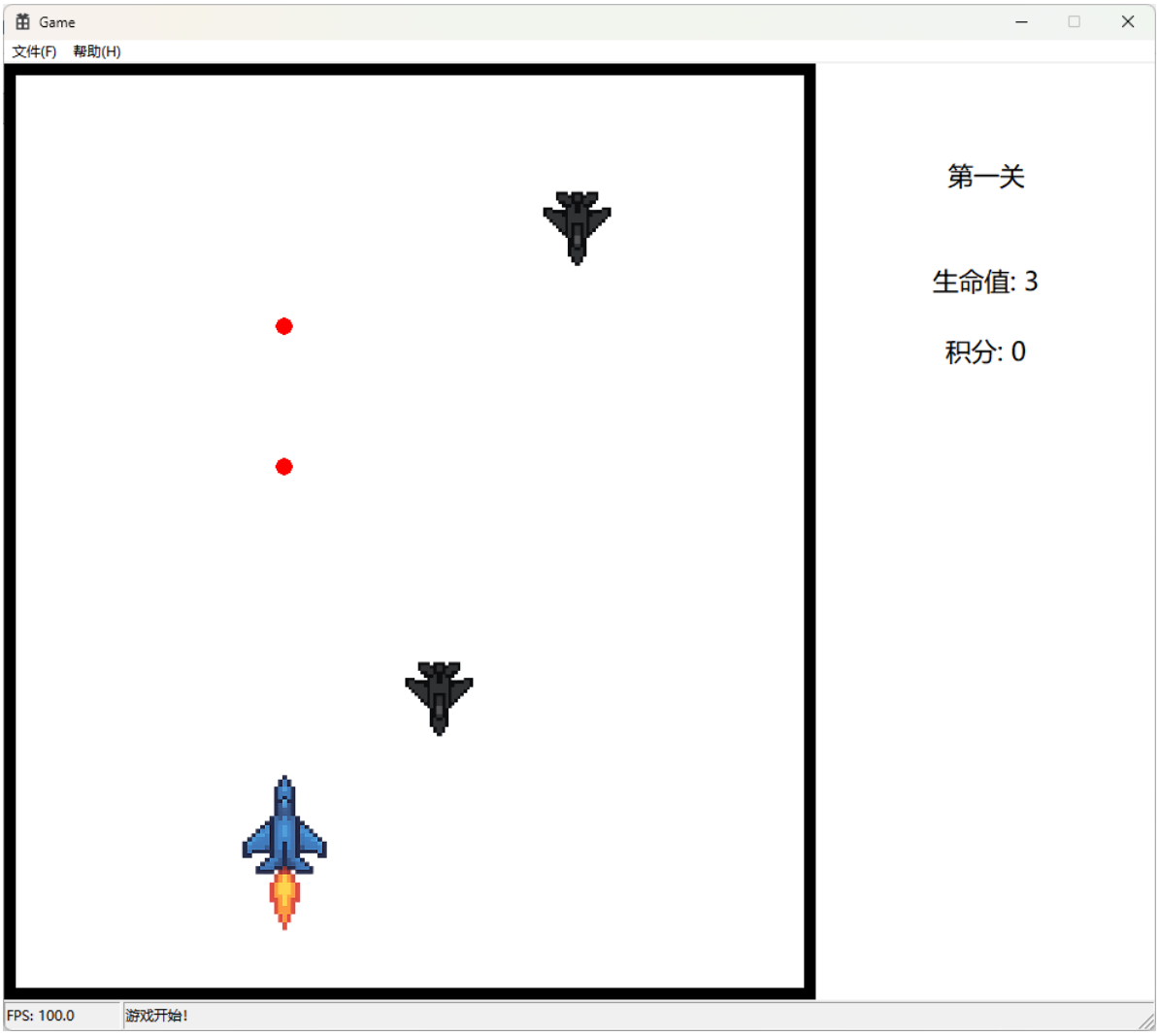
细心的同学应该已经注意到右边应该要显示玩家的分数和玩家的血量，但是并没有显示。这是因为现在UI上默认显示的是0。

同学可以在`RenderScene`中找到绘制该段文字的地方，或者Ctrl + F搜索这段话，然后把对应的信息改成玩家的信息。

```
// scene2.cpp
void RenderScene_GameScene(HDC hdc_memBuffer, HDC hdc_loadBmp)
{
    // 绘制
    TCHAR buffer[128];
    swprintf_s(buffer, sizeof(buffer) / sizeof(TCHAR),
               TEXT("第一关\n\n\n生命值: %d\n\n积分: %d"),
               GetPlayer()->attributes.health,
               GetPlayer()->attributes.score);
    DrawText(hdc_memBuffer, buffer, -1, &rect, DT_CENTER);
}
```

这里其实相当重要——如何在界面上显示变量的值。首先是创建一个缓冲区`buffer`，然后使用`swprintf_s`把变量打印到缓冲区里，格式化输出和`printf`函数一样。

恭喜同学们，现在同学们再运行程序，将得到一个完整的、可以游玩的简单飞机大战了！可以开始游戏，有玩家和敌机，可以发射子弹，可以击毁敌人，能够显示生命值和分数，还有结束游戏。



三、接下来的事情

同学们接下来就可以按照我们给出的Excel得分表选择实现某些功能了，下面再写一些提示。

1）把绘制文字单独拎出来

能不能把绘制文字单独拎出来？现在`scene1`和`scene2`里面绘制场景时，会有很长的代码是为了绘制文字，是否可以写成函数甚至放到新的`cpp`文件里面来减少重复的代码？

2）暂停游戏

想想暂停游戏应该怎么实现，总之不是切换场景吧？因为切换场景的时候，原先游戏里的东西都会被卸载掉，所以并非切换场景。那么该怎么实现呢？我们可以用一个标志为来表示游戏是否暂停，这个可以保存在Scene结构体里：

```
struct Scene
{
    SceneId sceneId; // 游戏场景的编号

    bool isPaused; // 打个比方
    // TODO: 如果场景需要保存更多信息，添加在这里
};
```

然后就可以根据根据这个标志位来切换游戏的运行逻辑：

```
void UpdateScene(double deltaTime) {
    if (GetCurrentScene()->isPaused) {
        // 暂停了就启用暂停界面的UI的按钮
        EnableButton(continueButtonId); // 打个比方，已经创建但是隐藏掉的继续按钮
        //
    }
    else {
        // 没有暂停则运行游戏逻辑
        UpdatePlayer(deltaTime);
        // 打个比方
    }
}

void RenderScene() {
    if (GetCurrentScene()->isPaused) {
        // 绘制暂停界面的UI
    }
}
```

这样就可以完成暂停界面。

3）添加新的场景

添加新的场景，你需要创建一个`scene3.h`和`scene3.cpp`，在`scene.h`的枚举类`SceneId`里添加新的场景名字，在路由宏函数里加上新的`case`，在`scene.cpp`中`#include "scene3.h"`，在`scene3.h`中声明上面讲的六个函数，在`scene3.cpp`中实现上面讲的六个函数。总之，模仿`scene1`和`scene2`中做的事情。

4）添加新的对象

添加新的对象，你需要创建一个对应对象的头文件和源文件，至于具体这个对象要实现哪些功能取决于你。单个对象可以模仿`player`，多个对象可以模仿`enemy`。时刻谨记对象的创建、销毁、渲染与逻辑更新这些相关的内容。

5）绘制新的东西

绘制相关的内容也已经讲了，同学们可以自己找一些资源甚至让GPT生成都可以，来让你的游戏画面更加好看。框架里的所有图片资源都不强制要求使用，你可以换成自己喜欢的东西。界面设计同学们也可以自己进行，不必被框架限制，这里的原则就是，想清楚要画什么、怎么画，并稍微注意一下绘制顺序。

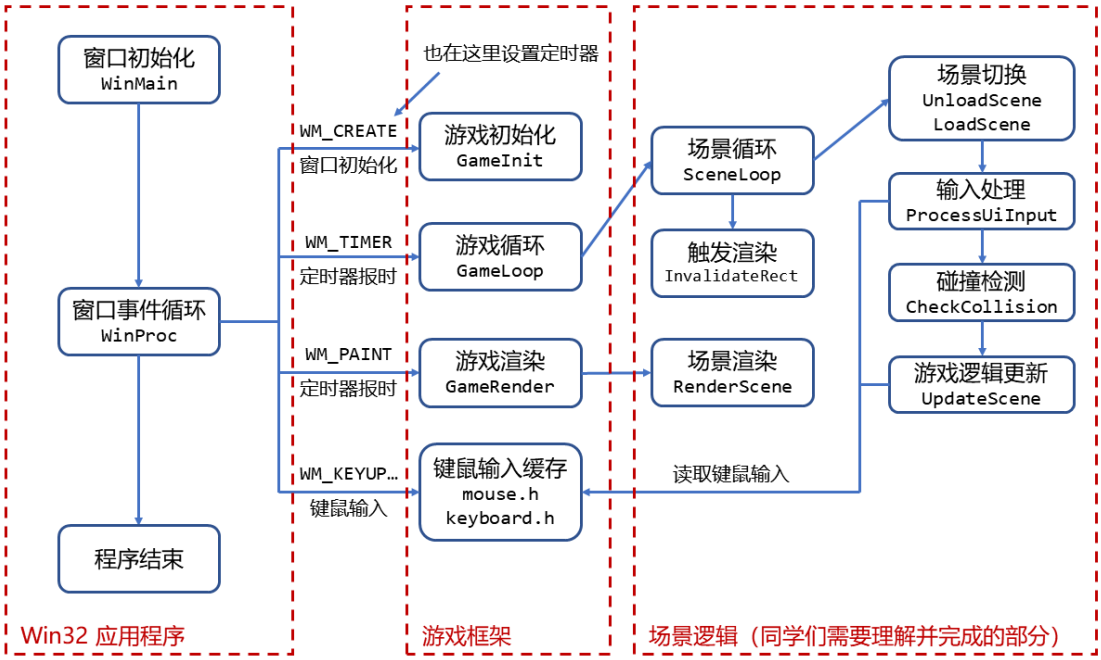
6）背景音乐和音效

给游戏加上背景音乐和音效可以极大地提高游戏的体验，这一部分同学们可以自己去网络上找找该怎么实现，想想相关的音乐播放应该放在游戏初始化或游戏循环中的什么地方。

最后，期待同学们完成的大作业。

附一、Win32 绘图

我们再回顾框架示意图，关注其中的`WM_PAINT`部分。



可以发现，绘图也是众多「事件」中的一种事件——它同样依靠操作系统事件循环一次次地触发。这个触发频率可能很高，例如达到每秒平均60次（注意这60次并不一定匀速触发），那么我们就说这个游戏的刷新帧率达到「60fps」。

也正是在整个`WM_PAINT`链条中，我们把一个个抽象的变量变成画布上一个个鲜活的画面。

```
// main.cpp
// WndProc()函数
case WM_PAINT:
{
    // 游戏渲染逻辑
    GameRender(hWnd, wParam, lParam);
}
```

`GameRender()`：一切绘制，从这里开始

在`GameRender()`中，我们使用了Win32中最原始，也是最直接的绘制方式「GDI」，但麻雀虽小，五脏俱全。

```
void GameRender(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc_window = BeginPaint(hWnd, &ps);

    // 具体绘制逻辑

    EndPaint(hWnd, &ps);
}
```

这段逻辑的意思是，我们在`hWnd`这个窗口上开始绘制，而绘制通过`hdc_window`进行。（`HDC`是绘制中所有操作的上下文所在，所有对主窗口的绘制，都是对其`HDC`，即`hdc_window`调用函数来实现的）

画个矩形

例如，我们可以在`// 具体绘制逻辑`部分加上绘制矩形的代码：

```
void GameRender(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc_window = BeginPaint(hWnd, &ps);

    Rectangle(hdc_window, 50, 50, 200, 100);

    EndPaint(hWnd, &ps);
}
```

这样，就可以画出一个左上角坐标`(50,50)`，右下角坐标`(200,100)`的矩形。

如果我们想把矩形变成实心的，或者换个颜色，也是可以实现的，需要用到`FillRect()`、`PEN`和`BRUSH`，具体用法请参考GDI的文档：https://learn.microsoft.com/en-us/windows/win32/gdi/windows-gdi，但我们在此不多赘述。

如果你想画其他的基本图元，例如圆、椭圆、线段等，也可以在文档中找到相应的方法。

绘制文字也是类似，使用`DrawText`或`TextOut`等函数，同样可以参考文档：https://learn.microsoft.com/en-us/windows/win32/gdi/fonts-and-text。

位图

在我们的大作业中，这些基本图元难以组成精美的画面，相反，我们会用到的绝大多数资源都是**图片**，因此，我们将在这部分详细讲解图片的绘制。

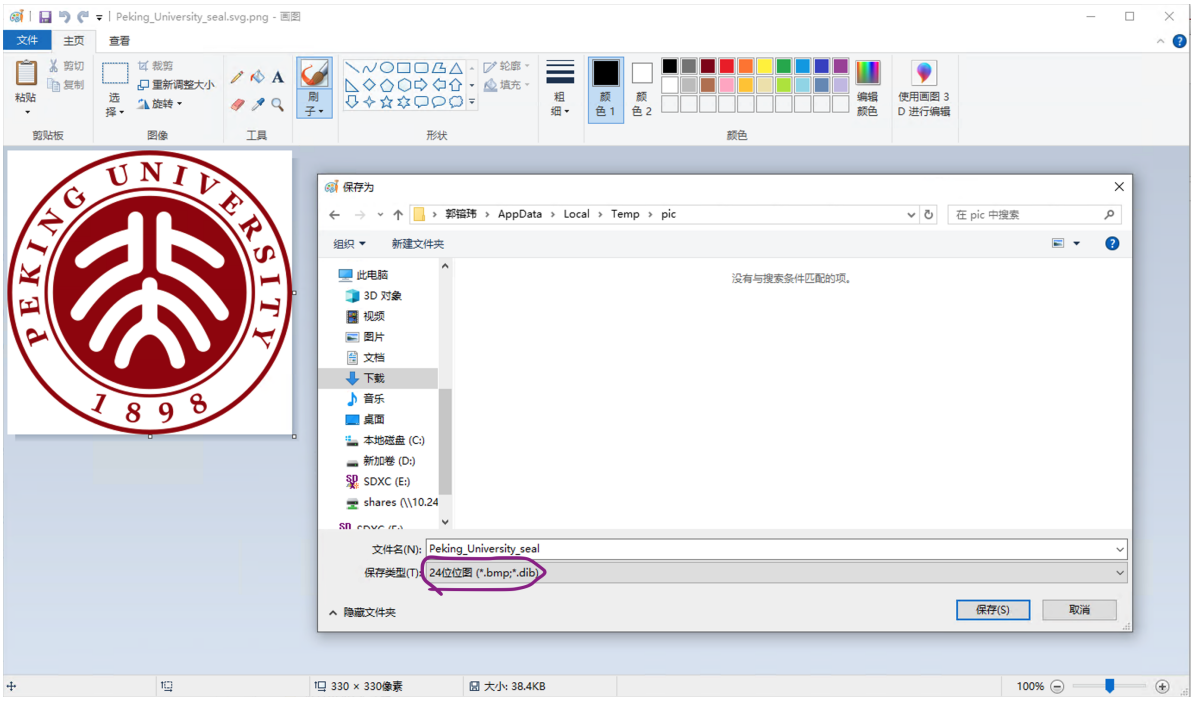
我们日常所见的`jpg`、`png`等格式的文件都是压缩的图片，**无法**被GDI直接使用；而只有**位图**格式，通常后缀为`bmp`，这样格式的文件才能被直接使用。

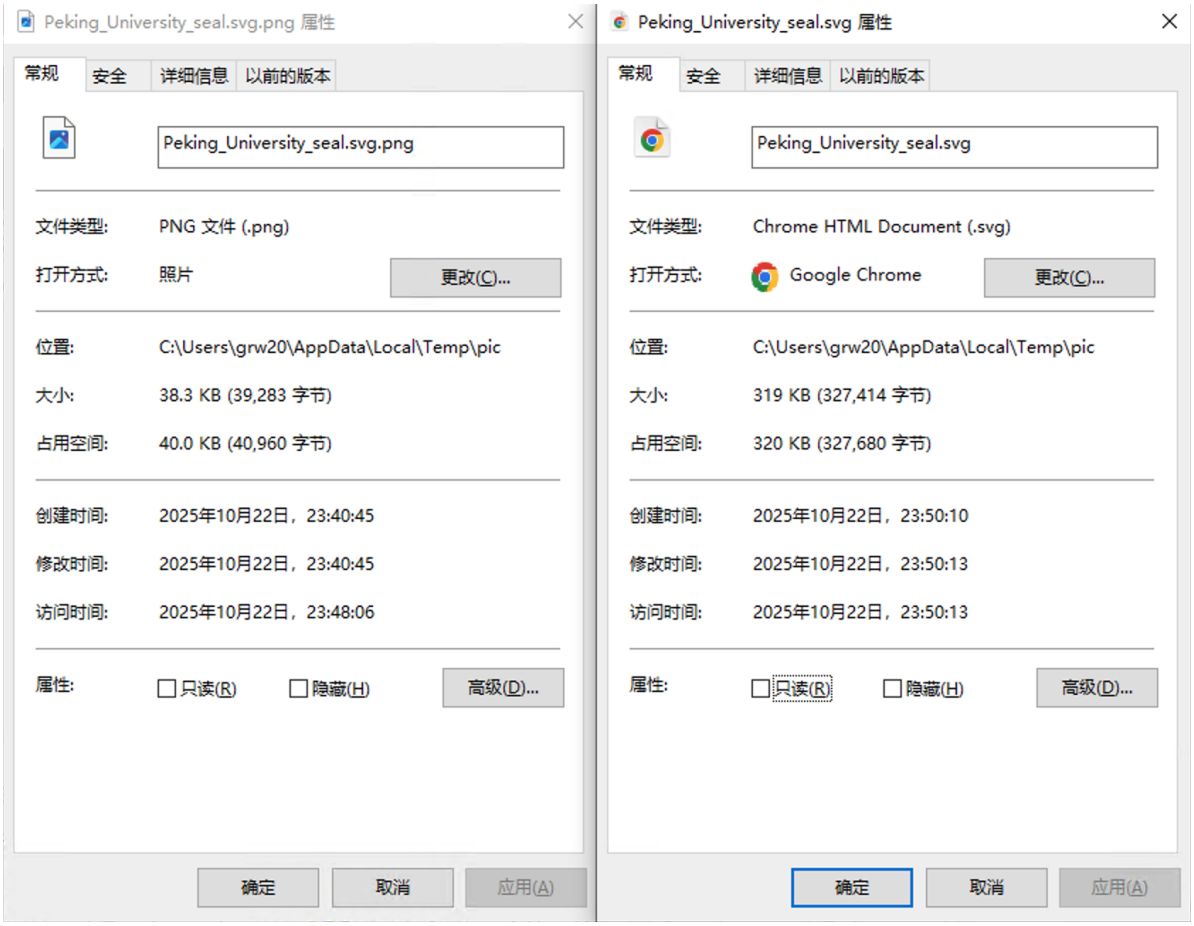
感受位图

位图是完全没有压缩的图片，最直观反映在其文件大小上：我们知道，计算机上每一个像素是由红、绿、蓝三种颜色按比例混合而来的，例如，可以用每3个字节分别表示一个像素红、绿、蓝的亮度值（范围分别是`0~255`），比如`(255, 0,0)`代表纯红、`(255,255,255)`代表纯白、`(128,128,128)`代表一种灰色、`(0,0,0)`代表纯黑。

对于一张图片，我们把代表所有像素的3个字节全部堆叠在一起（像数组一样）写入文件，就构成了一幅位图文件（此外，在文件开头需要一些信息注明其分辨率等信息）。

例如，下图是一幅来自互联网的`png`格式压缩的图片，如果用「画图」打开，并且「另存为」`bmp`格式后，可以发现其文件大小显著增加了。





我们还可以做一个小计算：

> 图片尺寸：330×330像素
>
> 每个像素：3字节
>
> 一共大小：$330 \times 330 \times 3 = 326,700$字节

与上图显示的接近（因为文件开头还需要保存一些文件有关的信息，所以会大一些）。

由此可见，位图格式的存储空间有极大的浪费，所以通常并不会直接存储或传输位图；然而，当一个程序需要将图像加载到内存之中时，图像都是以**位图**的形式存在的（即上面说的像素数组）。

由于加载`jpg`、`png`需要引入额外的解码库，为了方便，我们推荐大家将所有的图像转为位图的形式进行调用。

使用位图

加载位图到`Game.rc`

找到项目中`Game.rc`文件，双击打开后进入`Bitmap`文件夹，可以看到目前所有的位图；选中每一个位图后，可以在「属性」窗格中更改其`ID`。

在`Game.rc`上右键「添加资源」，可以将找到的其他资源加入进来。

在`GameResourceInit()`中加载位图

找到`资源\resource.cpp`文件，在`GameResourceInit`中使用`LoadBitmap`函数将位图加载到程序中。

你可以仿照已有的代码加载更多位图。

把它画到 `hdc_window` 上

还记得刚才我们用的画布 `hdc_window` 吗?

我们使用 `BitBlt()` 和 `TransparentBlt()` 函数来绘制位图，但与刚才不一样的是，这里操作多了一步：

```
void GameRender(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc_window = BeginPaint(hWnd, &ps);

    HDC hdc_loadBmp = CreateCompatibleDC(hdc_window);
        SelectObject(hdc_loadBmp, bmp_Player);
    BitBlt(hdc_window, 0, 0, 200, 300, hdc_loadBmp, 0, 0, SRC_COPY);

    EndPaint(hWnd, &ps);
}
```

我们先不管 `BitBlt` 的调用参数（都是些和坐标变换有关的），先关注那个多出来的 `hdc_loadBmp`。

原来，`BitBlt` 要求我们不能直接把位图画到 `hdc_window` 上，而需要通过一个中间层 `hdc_loadBmp` 来帮忙。

这个 `hdc_loadBmp` 的类型同样是 `HDC`，只不过它并没有和一个实际的窗口相关联，相反，它是一张仅存在于内存中的“虚拟画布”，我们先用 `SelectObject` 将位图加载到这张虚拟画布 `hdc_loadBmp` 上，再将这张虚拟画布画到窗口上。

> `SelectObject` 这个函数功能非常多，在这里作用是将位图加载到 `hdc_loadBmp` 上，在绘制图元的时候也用来选择画笔和画刷的颜色。

在实际代码中（请参考 `scene1.cpp`、`player.cpp`、`enemy.cpp` 等对象的 `Render` 部分），我们会将框架 `core.cpp` 内创建好的一个 `hdc_loadBmp` 通过函数参数不断地传递下去，这是因为频繁地创建 `hdc_loadBmp` 会导致严重的性能问题，我们复用这样一个虚拟画布以优化性能。

双缓冲绘图

你可能会想，`BitBlt()` 不能直接画位图，而要通过一个内存中的虚拟画布 `HDC hdc_loadBmp` 来中转这样的设计是多此一举，但事实是，位图绘制只是 `BitBlt()` 的功能之一，它更重要的是借此设计实现了双缓冲绘图。

双缓冲绘图：抵抗画面撕裂

计算机的指令是一条条执行的，而当我们的图元太多的时候，绘制的过程就需要一定的时间；对于用户来说，它看到的画面就是一点一点地画出来的，就像「大屁股」电视一样，画面是一行一行扫描出来的，十分影响观感。

「双缓冲绘图」思想便应运而生：既然最终显示在显示器上的画面**本质上就是一幅由很多像素组成的位图**，我们于是可以先在内存中创建一幅位图，把全部要绘制的内容绘制到它上面之后，再一次性将这张代表着这一帧内容的位图显示出去，于是就可以解决画面撕裂的问题了。

实现：`hdc_memBuffer`

而实现这张「虚拟位图」的，就是刚刚提到的虚拟画布，只不过我们重新创建一个并另取名字：

```
void GameRender(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc_window = BeginPaint(hWnd, &ps);

    // 创建虚拟画布，并初始化缓存
    HDC hdc_memBuffer = CreateCompatibleDC(hdc_window);
    HBITMAP blankBmp = CreateCompatibleBitmap(hdc_window, WINDOW_WIDTH,
WINDOW_HEIGHT);
    SelectObject(hdc_memBuffer, blankBmp);

    /* 绘制部分
    HDC hdc_loadBmp = CreateCompatibleDC(hdc_window);
        SelectObject(hdc_loadBmp, bmp_Player);
    BitBlt(hdc_memBuffer, 0, 0, 200, 300, hdc_loadBmp, 0, 0, SRC_COPY);
    */

    // 把虚拟画布画到窗口上
    BitBlt(hdc_window, 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT, hdc_memBuffer, 0, 0,
SRCCOPY);

    EndPaint(hWnd, &ps);
}
```

这里的`HDC hdc_memBuffer = CreateCompatibleDC(hdc_window);`，就是我们用来画图的虚拟画布。

如果你去翻阅后续的所有渲染部分，会发现所有的绘制都是通过各种函数绘制在`hdc_memBuffer`上，然后当一些绘制结束，程序回到`core.cpp`时，通过`BitBlt`会知道窗口`hdc_window`上。

由此，我们的绘制过程就全部完成了。