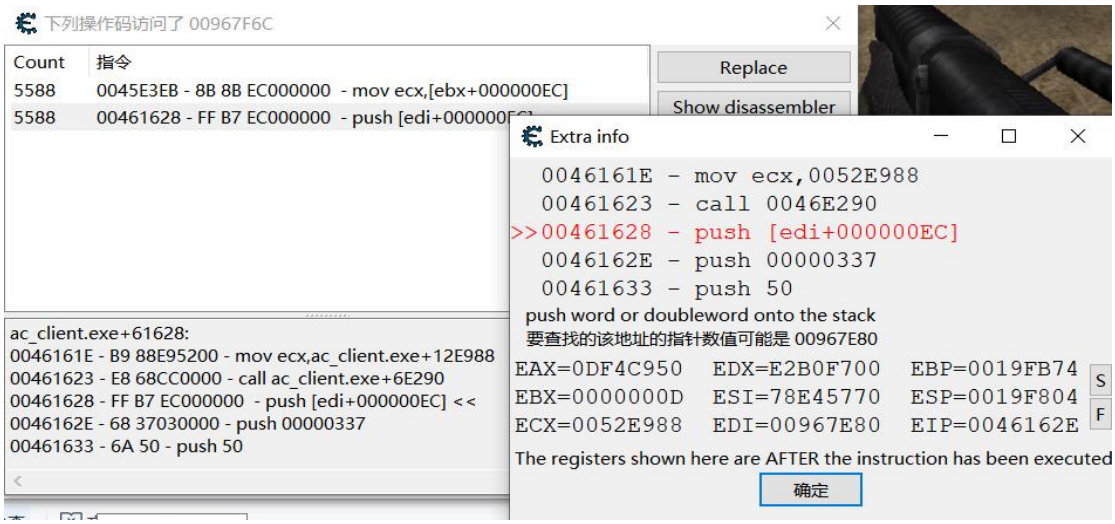
首先在 CE 中查找控制玩家血量的地址

通过查看是谁访问了该地址，可以猜测出血量在 Player 结构体中的偏移是 0xEC

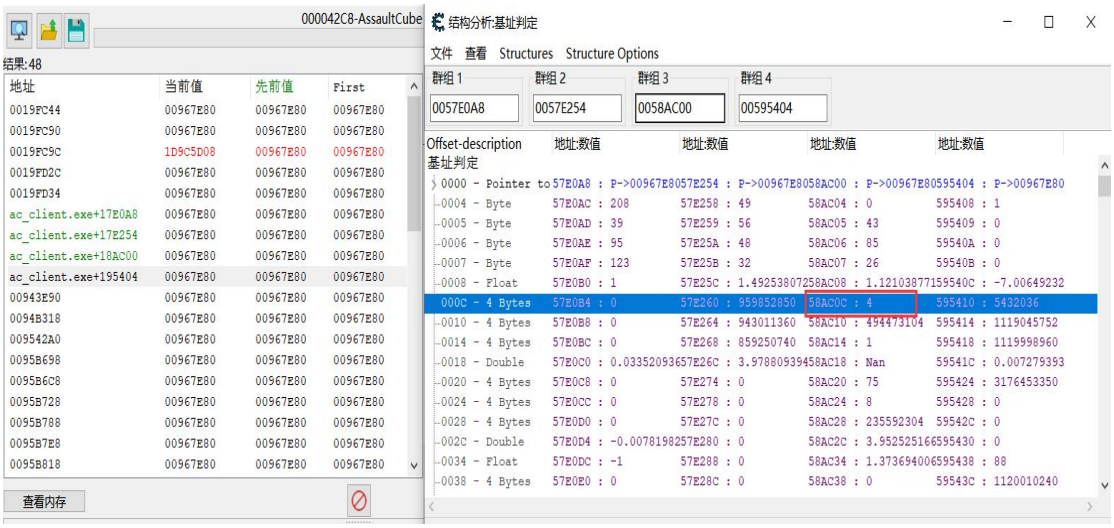


然后查找玩家对象的静态基址指针，将扫描数值设置为 00967E80，寻找绿色的地址数据即为所求，以便在程序每次启动时都能找到对象基址

以上四个绿色地址均可作为游戏对象基址的指针，游戏设计师在设计游戏时通常会将游戏玩家数量，玩家基地址，entity 基地址在一片连续的内存区域进行设定，所以可以对这四个静态地址所在内存的上下文环境进行观察比对，选择出一个更有关联价值的静态地址

现在重开一把 2v2 的游戏，游戏玩家总数为 4。借助 CE 的结构分析工具查找它

们的上下文环境哪个有 4 这个值



通过上述比对我们可以发现，ac_client.exe+0x18AC00 这个地址的上下文环境对我们获得其他游戏信息非常有益，所以我们将采用 ac_client.exe+0x18AC00 作为玩家基址指针，读取 ac_client.exe+0x18AC00+0xC 的值即为玩家数量

下面进行查找敌人基地址，方法与上面查找玩家基地址类似，锁定一个敌人，对他进行攻击，通过血量变化找到存储该敌人血量的地址，由于敌人作为一个游戏实体的存在，它的结构和玩家的设计结构应该是相同的，所以直接减去 0xEC 即可得到该敌人对象的基址

敌人的血量	1DC6741C		4字节	87
敌人基地址	1DC67330		4字节	0054D07C

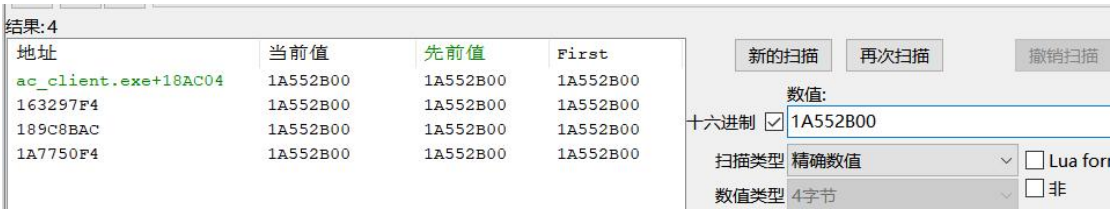
也可以通过该方法获得队友和其他敌人的动态基址，另外也可以直接在 CE 扫描值为 54D07C 的地址，即可一次性显示出其他 entity 的基址

接下来需要做的工作是如何查找 entity 数组的基地址,可以将这些敌人基地址作为扫描值进行扫描，发现得到的地址结果都是动态地址，结合实际开发经验，entity 数组存放的可能是加载 entity 的指针，即二级地址的形式，下面将这三个地址分别进行扫描，并将扫描结果记录在 excel 表格中。观察扫描结果，寻找其中的连续地址（即相差为 4 的等差数列），观察统计结果：entity 数组的动态基址很有可能为 1A552B00 或 1A552B24

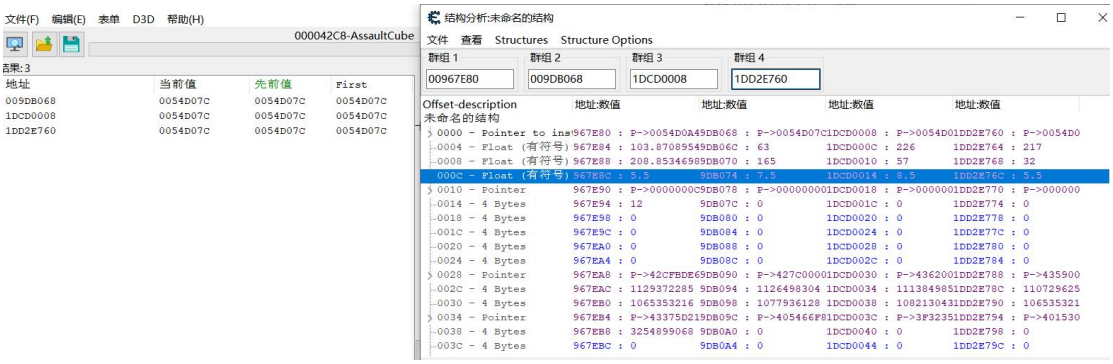
	A	B	C	D	E
1	009D0F18	1A552B08	1A552B0C		
2	009D0F34	1A552B2C	1A552B30		
3	1A552B04	1A559518	1A55951C		
4	1A552B28	1A559658	1A55965C		
5	1A559450	1A5598D8	1A5598DC		
6	1A559478	1A559B08	1A559B0C		
7	1A559630	1A559C48	1A559C4C		
8	1A559928	1A559C98	1A559C9C		
9	1A559BA8	1D5C3CA8	1D5C3CE8		
10	1A559C20	1D61A1D0	1D61A200		
11	1D5C3FE8	1D61A320	1D61A230		
12	1D61A1A0	1D61A440	1D61A260		
13	1D61A380	1D61A4A0	1D61A290		
14	1D61A410	1D61A500	1D61A3B0		
15	1D61A470	1D61A530	1D61A4D0		
16	1D797218	1D797480	1D67300C		
17	1D9C9290	1DA574EC	1D797678		
18	1D9C92AC	1DCFBD90	1DCFBDD8		
19	1DA4EBE0				
20	1DA4ECD0				
21	1DA8D01C				
22	1DC92228				
23	1DC92244				
24	1DD00AA0				
25					

扫描 1A552B00 发现一个绿色地址，所以 entity 数组的基址存放在 ac_client.exe+0x18AC04 中

（也即存放玩家基地址的指针的下一条地址）



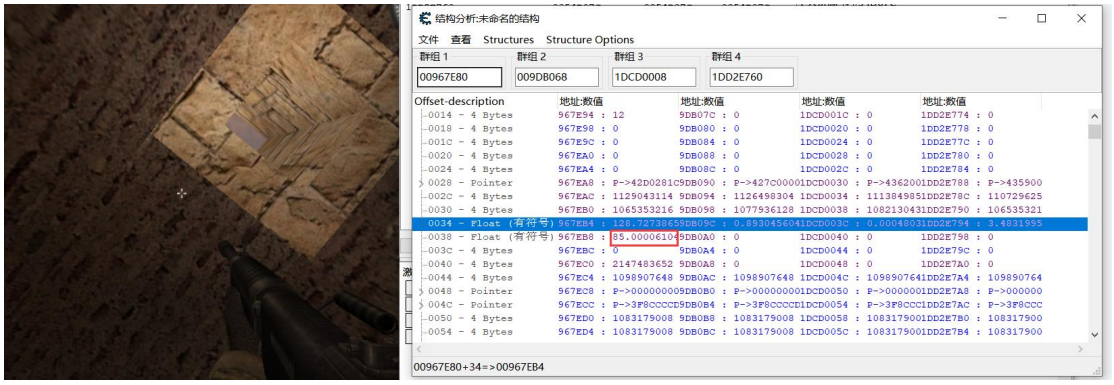
然后分析对比敌人和玩家的结构体，从中找出游戏实体的坐标信息，偏角信息，昵称，阵营



基址+0x4: pos_x　基址+0x8:pos_y　基址+0xc: pos_z

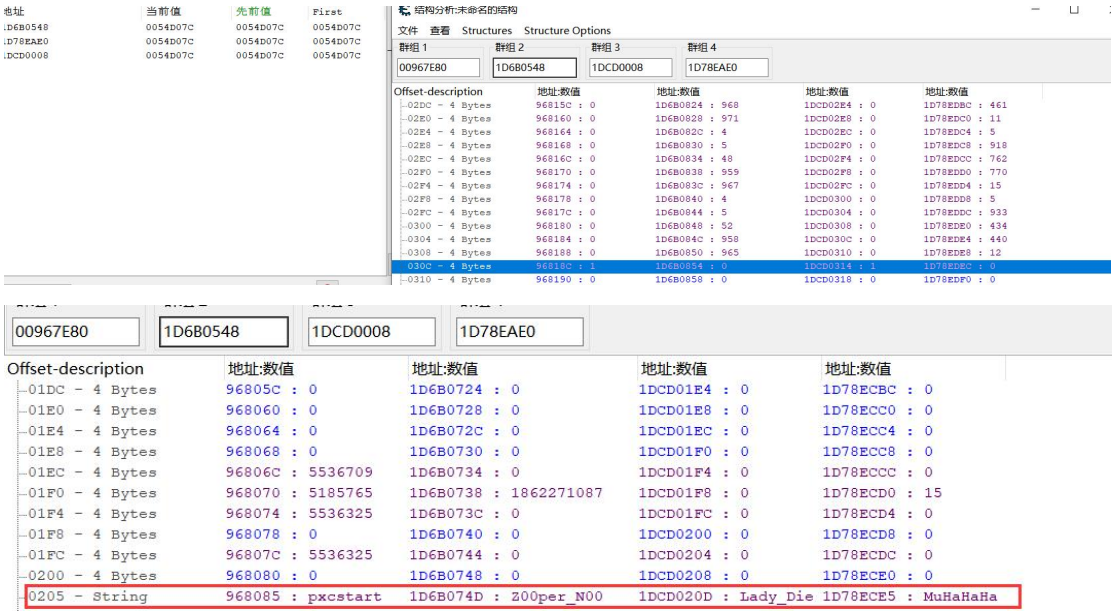
基址+0x205:昵称

关于准心偏角的查找有两种方法，一种是不断移动鼠标改变游戏镜头，当枪支向上移动时准心偏角的 y 值会变大，向下移动时 y 值会减小，找到绿色地址后计算与玩家基址的偏移即可，但搜索过程比较繁琐；另一种比较快速的方式是让准心顶到最上方，此时在结构体中查找接近 90 的值，再让准心吵到最下方，监测此时是否接近-90（上下偏角的范围-90~90）

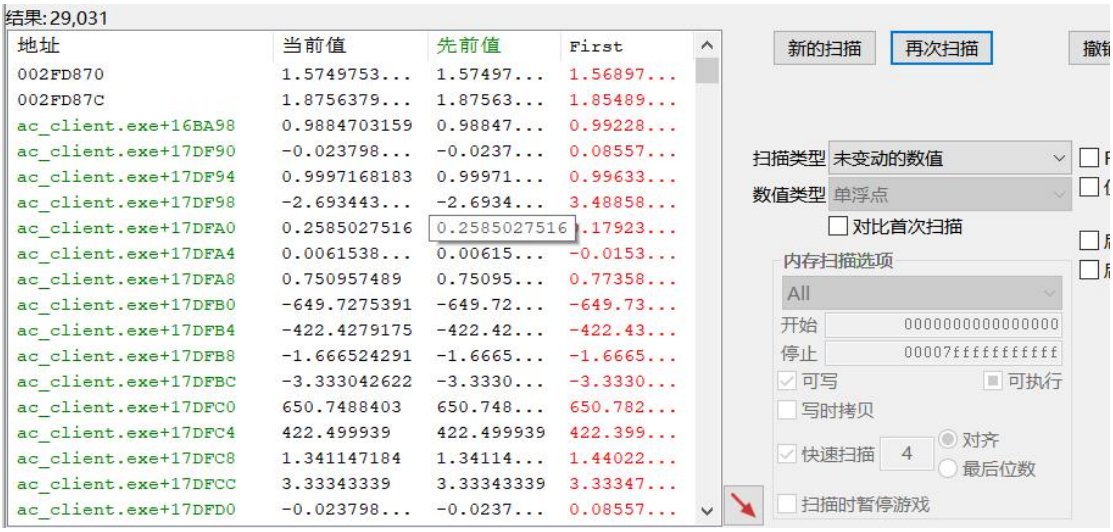


找左右偏角也是类似的道理，让人物面向地图的 N 端，经测试此时应为 0 度（S 端为 180 度，E 端为 90 度，W 端为 270 度）。

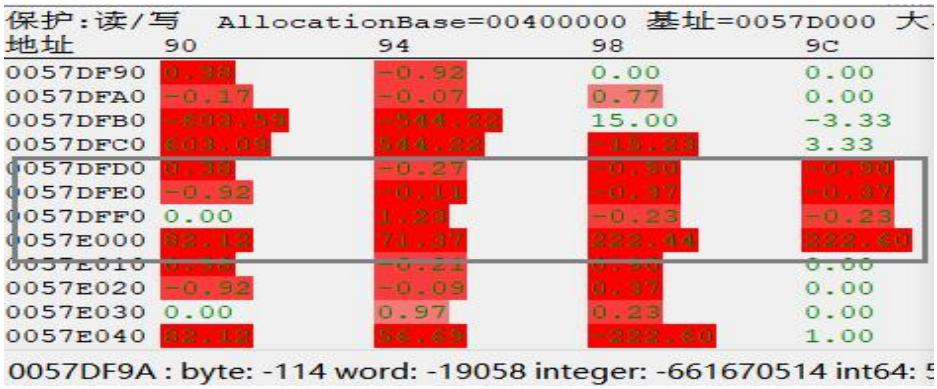
综上，基址+0x34:左右方向的偏航角 基址+0x38:上下方向的俯仰角



从上图可以看出，基址+0x30C:阵营
最后再查找交换矩阵（应用于世界坐标转换为屏幕坐标）
扫描数据时首先将数据类型选择为单精度浮点数，未知初始值；然后不断更改镜头角度--变化的数值 or 不对游戏做操作--未变化的数值，最终筛选出以下可能符合结果的数据



根据游戏的 about 选项查看配置信息，Assaultcube 游戏的图形库为 OpenGL，4x4 坐标变换矩阵的特点为某一行的数字特别大，其他行的数字均在-2~2 之间。并且该矩阵信息一定是记录在一个静态地址中，因此从绿色地址中查看内存，变化镜头查看数据变化（若 4x4 的数据都在变化，且数据值满足以上特质，即可作为备选项）



最终可以看出，交换矩阵记录在 0x57DFD0（即 ac_client.exe+0x17DFD0 中）

代码实现：
定义游戏实体的结构体：

```
typedef struct PlayerData
{
    DWORD BaseEntity;
    float Position[3];
    int HP;
    int TeamFlag;
    char Name[32];
}PD;
```

将刚才通过 CE 捕获到的偏移信息进行填写

```
///OFFSET///  注：使用的assaultcube的版本为v1.3.0.2
static DWORD base_address = 0x00400000;
static DWORD offset_address = 0x18AC00;
static DWORD player_base = base_address + offset_address; //玩家基地址
static DWORD entity_base = player_base + 0x4; //entity基地址
static DWORD offset_playersnum = player_base + 0xC; //游戏中的玩家数量
static DWORD offset_viewmatrix = 0x57DFD0; //交换矩阵
//player结构体内部成员变量偏移
static DWORD offset_health = 0xEC;
static DWORD offset_name = 0x205;
static DWORD offset_posx = 0x4;
static DWORD offset_posy = 0x8;
static DWORD offset_posz = 0xC;
static DWORD offset_team = 0x30C;
static DWORD offset_ang_left_right = 0x34;
static DWORD offset_ang_up_down = 0x38;
```

为了实现透视的方框，先创建一个蒙版窗口：
主要操作步骤为：1）设计一个窗口类 2）注册窗口类 3）创建窗口 4）显示及窗口更新 5）消息循环 6）构造窗口过程函数

```
void CreateOverlayWindow()  //创建蒙版窗口
{
    WNDCLASSEX wc;  //step1:设计一个窗口类
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc_Overlay;  //窗口过程函数
    wc.hInstance = GetModuleHandle(0);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)RGB(0, 0, 0);
    wc.lpszClassName = "Overlay";
    RegisterClassEx(&wc); //step2:注册窗口类
    //step3:创建窗口  注：由于创建的是蒙版窗口，用WS_EX_TRANSPARENT将窗口设为透明
    g_hWnd_Overlay = CreateWindowEx(WS_EX_LAYERED | WS_EX_TRANSPARENT, wc.lpszClassName, "OverlayWindow", WS_POPUP, g_winRect.left, g_winRect.top,
        (g_winRect.right - g_winRect.left), (g_winRect.bottom - g_winRect.top), NULL, NULL, wc.hInstance, NULL);
    //设置窗口透明度
    SetLayeredWindowAttributes(g_hWnd_Overlay, RGB(0, 0, 0), 0, ULW_COLORKEY);
    //step4:显示和更新
    ShowWindow(g_hWnd_Overlay, SW_SHOW);
    UpdateWindow(g_hWnd_Overlay);
}
```

窗口处理函数：

```
LRESULT CALLBACK WindowProc_Overlay(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  //窗口处理函数
{
    switch (message)
    {
    case WM_PAINT:
        break;
    case WM_CREATE:
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    case WM_CLOSE:
        DestroyWindow(g_hWnd_Overlay);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
        break;
    }
    return 0;
}
```

在消息循环中不断根据偏移值使用内存读取函数读取玩家数据和每一个 entity 实体的数据

```
//消息循环
MSG msg;
ZeroMemory(&msg, sizeof(MSG));
while (msg.message != WM_QUIT)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    if (GameIsForegroundWindow() == TRUE)  //即判断游戏进程是否在前台进行，若游戏被切到后台，则蒙版应暂停工作
        SetWindowPos(g_hWnd_Overlay, HWND_TOPMOST, g_winRect.left, g_winRect.top, (g_winRect.right - g_winRect.left), (g_winRect.bottom - g_winRect.top),
    else
        SetWindowPos(g_hWnd_Overlay, HWND_BOTTOM, g_winRect.left, g_winRect.top, (g_winRect.right - g_winRect.left), (g_winRect.bottom - g_winRect.top),
    //根据当前消息更新窗口屏幕坐标并适时移动蒙版
    GetWindowRect(hWnd, &g_winRect);
    g_winRect.top += TOP_HEIGHT;
    MoveWindow(g_hWnd_Overlay, g_winRect.left, g_winRect.top, (g_winRect.right - g_winRect.left), (g_winRect.bottom - g_winRect.top), TRUE);
    Clear();
    //获得玩家自己的信息
    PD player;
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player_base),&player.BaseEntity,sizeof(player.BaseEntity),0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + offset_health), &player.HP, sizeof(player.HP), 0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + offset_team), &player.TeamFlag, sizeof(player.TeamFlag), 0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + offset_name), &player.Name, sizeof(player.Name), 0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + 0x4), &player.Position[0], sizeof(player.Position[0]), 0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + 0x8), &player.Position[1], sizeof(player.Position[1]), 0);
    ReadProcessMemory(g_hProcess, (LPCVOID*)(player.BaseEntity + 0xC), &player.Position[2], sizeof(player.Position[2]), 0);
    int playersnum;
    ReadProcessMemory(g_hProcess, (LPCVOID*)(offset_playersnum), &playersnum, sizeof(int), 0);
    PD entitylist[50];
    float xyDistance[50] = { 0 }; //敌人距离屏幕中心的坐标
    for (int i = 0; i < playersnum; i++)
```

实现透视逻辑：
在每次消息循环中读取每个 entity 实体的阵营，坐标，昵称等信息，若阵营和玩家所在阵营不同，则说明是敌人，将敌人的坐标信息（世界坐标）先转换为屏幕坐标，然后根据敌人的世界坐标和玩家的世界坐标计算出敌人与玩家的距离（在游戏世界中的距离），根据该距离的远近调控方框的大小。
坐标转换逻辑实现：

```
BOOL WorldToScreen(float from[3], float to[2]) //世界坐标转为屏幕坐标
{
    int width = g_winRect.right - g_winRect.left;
    int height = g_winRect.bottom - g_winRect.top;
    float camX = width / 2.0;
    float camY = height / 2.0;
    float screenX = g_Matrix[0][0] * from[0] + g_Matrix[1][0] * from[1] + g_Matrix[2][0] * from[2] + g_Matrix[3][0];
    float screenY = g_Matrix[0][1] * from[0] + g_Matrix[1][1] * from[1] + g_Matrix[2][1] * from[2] + g_Matrix[3][1];
    float screenW = g_Matrix[0][3] * from[0] + g_Matrix[1][3] * from[1] + g_Matrix[2][3] * from[2] + g_Matrix[3][3];
    if (screenW > 0.001f)
    {
        to[0] = camX + (camX * screenX / screenW);
        to[1] = camY - (camY * screenY / screenW);
        return true;
    }
    return false;

}
```

绘画方框的逻辑实现：

```
void DrawEsp(float Enemy_x, float Enemy_y, float distance, PD enemy)
{
    float Rect_w = 2100 / distance;
    float Rect_h = 4000 / distance;
    float Rect_x = (Enemy_x - (Rect_w / 2));
    float Rect_y = Enemy_y;
    float Line_x = (g_winRect.right - g_winRect.left) / 2;
    float Line_y = g_winRect.bottom - g_winRect.top;
    HWND hWnd = GetForegroundWindow();//获取当前窗口
    if (hWnd == FindWindow(NULL, "AssaultCube"))
    {
        HDC hDc = GetDC(g_hWnd_Overlay);//!此处为overlay窗口句柄
        HPEN hPen = CreatePen(PS_SOLID, 2, 0x0000FF);//画笔一旦创建后就无法修改
        HBRUSH hBrush = (HBRUSH)GetStockObject(NULL_BRUSH);//空画刷
        SelectObject(hDc, hPen);
        SelectObject(hDc, hBrush);

        Rectangle(hDc, Rect_x, Rect_y, Rect_x + Rect_w, Rect_y + Rect_h);
        MoveToEx(hDc, Line_x, Line_y, NULL);
        LineTo(hDc, Enemy_x, Enemy_y);

        SetTextColor(hDc, RGB(0, 0, 255));
        SetBkMode(hDc, NULL_BRUSH);     //设置字体
        TextOutA(hDc, Rect_x, Rect_y - 20, enemy.Name, strlen(enemy.Name));
        DeleteObject(hBrush);
        DeleteObject(hPen);

        ReleaseDC(g_hWnd_Overlay, hDc);//!此处为overlay窗口句柄
    }
}
```

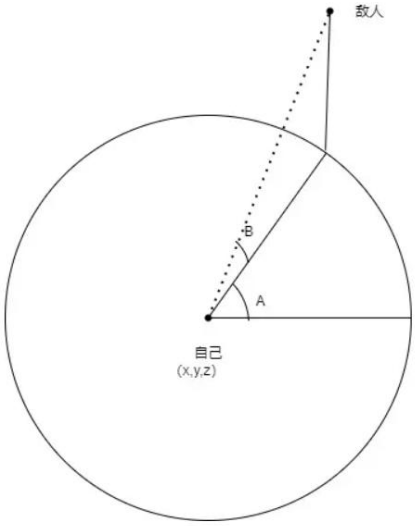
实现自瞄逻辑：

首先选择一个敌人作为自瞄的对象，在这里我选择的是距离屏幕中心最近的敌人，首先利用 GetAsyncKeyState 函数设计一个自瞄快捷键（程序中设计的是 ctrl 键），然后根据敌人的世界坐标和玩家的世界坐标计算出敌人与玩家在 x,y,z 轴三个维度的相对距离，实现自瞄就是要让自己面向敌人，也就是要求出下图中的角 A 也就是偏航角，然后我们计算仰角 B 也就是俯仰角，最后将这两个角度写入人物视角人物准心就能正对敌人了

设计快捷键的逻辑实现：

```
void __stdcall KeyHandlerThread() //定义自瞄快捷键
{
    while (1)
    {
        if (GetAsyncKeyState(VK_CONTROL) < 0)
            IsAuto = true;
        else IsAuto = false;
        Sleep(1u);
    }
}
```

fps 自瞄原理图：



实现自瞄的逻辑：

```
void AutoCollimation(float MY_Position[3], float EL_Position[3], PD player)  //自动瞄准
{
    float relative_x = EL_Position[0] - MY_Position[0];
    float relative_y = EL_Position[1] - MY_Position[1];
    float relative_z = EL_Position[2] - MY_Position[2];
    float dist = sqrt(pow(relative_x, 2) + pow(relative_y, 2)+pow(relative_z,2));

    float ang1 = atan(relative_x / relative_y) * 180 / M_PI;
    float ang2 = asin(relative_z / dist) * 180 / M_PI;
    float yaw, pitch;  //yaw的范围为0~360，pitch的范围为-90~90
    /*if (ang1 > 0)
    {
        if (relative_x >= 0 && relative_y >= 0) yaw = ang1;
        else yaw = 180 + ang1;
    }
    else
    {
        if (relative_x >= 0 && relative_y <= 0) yaw = 90 + ang1;
        else yaw = 360 - ang1;
    }
    yaw += 90;
    pitch = ang2;*/
    yaw = atan(relative_y / relative_x) * 180 / M_PI + 90;
    pitch = asin(relative_z/dist) * 180 / M_PI;
    WriteProcessMemory(g_hProcess, (LPVOID)(player.BaseEntity + offset_ang_left_right), &yaw, sizeof(yaw), nullptr);
    WriteProcessMemory(g_hProcess, (LPVOID)(player.BaseEntity + offset_ang_up_down), &pitch, sizeof(pitch), nullptr);
}
```

最终实现的效果如图：