基于 EasyX 和 Win32 的 AI 桌面宠物系统

付灏睿 *Fuhaorui*

2025.9.1

郑州大学 2024 级人工智能 3 班

illustrated by LaTeX

目录

插图

Listings

1 引言

当下社会的科技与经济正在步入高速发展阶段，人们的物质需求的满足和美好生活的建设都冲上了快车道的进程之中，成为了美好社会的基石。然而，极度膨胀的发展速度以及日益增多的工作和生活压力也伴随着经济的进步持续不断地压缩人们满足精神生活需求的空间，让原本丰富充实的娱乐和社交生活一步步走向高压的职场环境和枯燥的办公系统。对社会来说这样的趋势无异于加重了戾气和劣化了环境，而对于个人，更是一场对抗压力与抑郁的搏斗。可见，物质生活的满足和个人所得的提高同精神文化建设并不总是成正相关。

然而，近年来人工智能领域的突破触发了一系列势不可挡的爆炸式飞跃。其中，以大语言模型 (LLM) 为底层框架的智能体出现更是强而有力地冲击着人们的生活，刷新着人们的认知。同时，通过 AI 其卓越的可塑性和客制化能力，人们发现了多模态情感模拟在智能体身上的应用似乎远超传统预期，甚至为主流商业化赛道开辟了一条更为柔性的分支——情感智能。

因此，为满足日常生活的精神需要，给人们带来新奇的体验和排解、中和负能量的出口，本程序将实现一个生动的，可交互的，搭载了 LLM 的，拥有多模态情感的桌面宠物系统，命名为 **Amiya**。

一、在编程语言方面，程序将选择 C/C++ 为语言，来完成对所有模块功能的实现。

二、在图形库和框架方面，程序选择 EasyX 和 Win32(即 Windows API)，一方面是由于 EasyX 拥有易上手，好操作的优点，另一方面，出于对程序功能的实现，单一的 EasyX 在复杂任务的完成情况略显乏力，故引入 Win32 作为辅助，整体协调完成程序功能的落地和实现。

三、在编译器方面，程序选择 TDM-GCC 作为编译器，很好地弥补了 EasyX 图形库在 Visual Studio Code 上的兼容问题，避免了其他编译器诸如 MinGW，MSVC 等因适配性问题出现的 Bug，为后续程序的正常编写和运行保驾护航。

四、在 IDE 方面，本程序的编译在 Visual Studio Code 上实现，其简洁，方便，迅速的优点保障了程序编译的体验。

五、在操作系统方面，程序的编译在 Windows 11 上实现。

最后，本程序设计的希望能够让人们在繁杂的办公环境中，注意到桌宠在屏幕背后的默默陪伴，和与其对话时的全盘包纳与宽容，让人们从数据流中感受到情感的拟态。

关键词：EasyX，Win32，人工智能，大语言模型，LLM，情感智能

2 需求分析

2.1 总体描述

Amiya 是一款在桌面上运行的 AI 桌面宠物系统。程序启动后，在桌面上生成一个可爱的 ACG 角色 (原型为 Arknights 的虚拟角色 Amiya)，并且在相应状态下进行相应动作。起初，她会随机地在桌面上四处走动，展现出宠物这一极富主观能动性的客体性质。用户可以拖动 Amiya 至屏幕上的任意处。并且，Amiya 支持用户的一系列交互行为，比如右键触发交互动作同时打开 Menu 界面。在 Menu 界面中，有锁定功能和聊天功能供用户选择，前者可以通过切换锁定和非锁定状态，可以决定是否让 Amiya “坐” 在桌面上来控制 Amiya 的走动行为，后者则可以唤出聊天框，实现 AI 交流。在 AI 聊天层面，用户能够体会到 Amiya 极具个性的说话方式和拟真性格，让人们体会到程序的多模态情感而不是冷冰冰的机器输入输出。

2.2 功能性需求

2.2.1 初始化程序

1. 初始化随机数种子
   生成随机数参与后续在 Amiya 的随机状态时间，朝向，速度，加速度中的相关运算，模拟 Amiya 的 “主观能动性”，增添其生机和活力。

2. 初始化宠物程序窗口
   要求窗口背景和 title bar 透明且不可见，在视觉效果上呈现出只有 Amiya 的动画显示在桌面上。同时，宠物窗口第一次出现在屏幕正中央，便于用户定位到宠物位置。

3. 初始化动画资源
   从同根文件夹 \assets 中提取一系列动画帧 png 图片，用变量加载到内存之中，同时预留可拓展空间，增强程序的适应性和可拓展性。

4. 初始化全局鼠标钩子
   安装全局鼠标钩子来实现系统级的鼠标行为监控，为正确识别鼠标在 Amiya 身上的作用提供技术支撑：执行相应的回调函数来完成对用户鼠标行为的响应。

5. 初始化 AI 模块
   建立与外部大语言模型 API 接口的链接，同时读取并加载 Prompt 配置，为后续 Amiya 保持特定人设的回答提供技术保障。同时初始化历史对话队列，使得用户与 Amiya 的聊天更加自然。

2.2.2 状态机管理

1. 状态机类型

   - 待机状态 (relax) 如图(1)所示
   - 活动状态 (move) 如图(2)所示
   - 交互状态 (interact) 如图(3)所示

- 坐下状态 (sit) 如图(4)所示


图 1: relax 状态


图 2: move 状态


图 3: interact 状态


图 4: sit 状态

2. 状态机切换
   初始状态为 relax，自动运行随机走动代码，在 relax 与 move 之间切换，同时播放相应的 relax 和 move 动画。当用户右键点击 Amiya，触发 interact，完成整个 interact 播放后，回到 relax 状态。若右键唤出菜单点击 Sit 按钮，Amiya 进入持续的 sit 状态。

2.2.3 用户交互

1. 唤出菜单 (Menu)
   当用户右键点击 Amiya 时，唤出菜单，同时 Amiya 进入 interact 状态。菜单上预设了两个交互按钮，分别为切换坐下状态和开启聊天框进行与 Amiya 的聊天。菜单可以通过鼠标左键点击除菜单以外的地方来实现关闭操作。菜单在开启状态下，当 interact 状态结束时，Amiya 进入 relax 状态以及后续的 move 状态，菜单窗口会固定与 Amiya 的相对位置，跟随宠物移动。

2. 切换坐下状态 (Sit)
   当用户左键点击菜单上的坐下按钮时，Amiya 进入坐下状态，即持续的 sit 状态，此时 Amiya 不会随机切换到 move 状态，在视觉上呈现出 Amiya 坐在了屏幕的指定位置，而不会乱跑。此外，在坐下状态，Amiya 依旧可以被鼠标右键点击触发 interact 状态和唤出菜单。

3. 聊天 (Chat)
   当用户左键点击菜单上的聊天按钮时，出现可以输入文字的聊天框。用户把想要表达的文字输在聊天框中并发送，Amiya 内置的 LLM 模型会根据用户的输入作出相应输出。值得注意的是，LLM 模型会读取设置的 Prompt 来模拟 Amiya 原型的说话习惯，充分拟合 Amiya 的性格与情感。

4. 拖动宠物
   当用户左键按住宠物时，再移动鼠标，Amiya 会“附着”在鼠标上，实现拖动效果，便于用户改变宠物的位置，把 Amiya 放置在屏幕上想要的位置。

2.3 非功能性需求

2.3.1 动画素材获取

声明：根据《中华人民共和国著作权法》，本程序所用形象“Amiya”仅用于为个人学习、研究或欣赏，为个人范畴，版权版本归上海鹰角网络科技有限公司所有。

通过解包游戏文件，获取动画素材，储存在内存数组中，便于后续的快速调用。

2.3.2 动画切换流畅自然

状态机的切换过程是一个连续，流畅，自然的过渡，因此为了实现视觉层面的完美呈现，需要确保相应状态下的动画周期的播放完整，以及一个动画周期播放之时不能被另一动画周期覆盖，来贴合宠物的真实感。

2.3.3 AI 对话的一致性

为保证用户的体验，以及切实模拟和 Amiya 的对话，AI 需要根据 Prompt 的内容保证与原型人物设定一致，避免出现回复的内容与角色性格之间的差值过大，造成尖锐的冲突感受。

2.3.4 程序模块化封装

将程序划分为低耦合的逻辑模块，每个模块执行相应的功能。同时将数据和执行函数封装在模块内部，对外开放 API 接口，便于主程序的调用。此外，对于功能较为独立的模块，要做到高内聚，低耦合。而对于有耦合情况的模块组，要做到模块之间逻辑划分明确，API 调用清晰。整体上要提高程序的可维护性，可拓展性和可复用性。

3 概要设计

3.1 总体设计思路

本程序采用**事件驱动 + 状态机管理 +LLM 回复**的设计模式，以“用户输入—> 触发相应状态/AI 处理文本-> 输出相应状态/AI 回复”为主要逻辑链，来实现 AI 桌面宠物系统。

3.1.1 事件驱动

“事件驱动”主要识别用户对桌宠和菜单进行的鼠标操作，即左键点击，右键点击和长按左键，以及简单的键盘操作 (不包含文本的输入)——按下 ESC 关闭宠物窗口。通过识别用户的输出，驱动事件的发展，比如状态机的切换，和菜单的唤出等事件。

3.1.2 状态机管理

“状态机管理”则是事件驱动的主要结果和输出。事件驱动发出指令，由相关变量进行接收，状态机管理通过实时调用变量识别当前状态以及下一步状态，作出在 relax，move，interact，sit 四个状态之间的来回切换或保持不变。同时，真正将宠物的状态呈现在屏幕之上的，是状态机调用当前状态相应的动画帧图集，循环播放 (比如 move，relax，sit) 和单次播放 (比如 interact，一次右键点击只触发一次 interact 动画)。

3.1.3 LLM 回复

“LLM 回复”即指通过用户在聊天框输入并发出的文字，作出相应的回答。LLM 回复有两个需要注重的方面，一方面，考虑到硬件性能和时间成本问题，LLM 仅调用市面上的 Api 端口进行对用户输入的回复，这要求程序能够进行的 Api 调用和相关数据的传入传出。另一方面，简单输出 LLM 大模型的回复并不完全满足“情感智能”的需求，为实现宠物的亲和力和共情能力，需要使用 Prompt 为 LLM 的回复提出相应要求，同时接入多模态情感功能，进一步确保其回复针对人物原型人格的高拟合。

3.2 系统模块划分

3.2.1 基础层

1. 窗口管理模块：负责窗口的创建以及样式配置，主要服务对象为主窗口宠物窗口，子窗口菜单窗口和聊天窗口。
2. 线程同步模块：确保线程按预想状态下安全进行，避免线程耦合。
3. 资源加载模块：加载动画图片资源和配置参数。
4. 日志与调试模块：记录程序运行状态，输出错误信息和日志报告，便于排查和维护。

3.2.2 业务层

1. 宠物动画模块：管理 Amiya 的状态机切换的视觉效果，实时渲染。

2. AI 聊天模块：实现用户和 Amiya 之间的对话交互，包含 http 请求，响应与显示。
3. 打字机效果模块：拟合说话的逐字显示的视觉效果，模拟真实对话，提升用户体验。

3.2.3 控制层

1. 鼠标交互模块：监听全局鼠标事件，实现宠物的右键交互并唤出菜单，按住左键的开始拖动等交互效果。
2. 键盘交互模块：处理键盘输入，实现程序的退出和聊天内容的输入。
3. 状态联动模块：协调模块之间的状态同步，避免功能冲突。

3.3 数据结构

3.3.1 窗口创建相关

代码如 Listing(1)所示。

```
// 聊天窗口过程函数
LRESULT CALLBACK ChatWindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

// 聊天线程数据结构
struct ChatThreadData {
    DesktopPet* pet;
    HWND parentHwnd;
    HINSTANCE hInstance;
};

// 窗口相关成员变量
class DesktopPet {
private:
    HWND hwnd;                          // 宠物窗口句柄
    int windowWidth, windowHeight;   // 窗口宽高
    POINT pt_src;                       // 图层更新源点
    SIZE size_wnd;                      // 窗口大小
    BLENDFUNCTION blend;                // 透明度混合结构
    std::atomic<HWND> hChatWnd{NULL};   // 聊天窗口句柄
};
```

Listing 1: 窗口创建相关数据结构

3.3.2 线程同步相关

1. 聊天窗口线程参数
   作为传递给聊天窗口线程的初始化参数，包含宠物对象指针（用于访问全局状态与方法）、宠物窗口句柄（用于设置窗口层级）和应用实例句柄（用于注册窗口类与创建控件）, 代码如 Listing(2)所示。

```
struct ChatThreadData {
    // 宠物对象指针，用于访问全局状态与方法
    DesktopPet* pet;
    // 宠物窗口句柄，用于设置窗口层级
    HWND parentHwnd;
    // 应用实例句柄，用于注册窗口类与创建控件
    HINSTANCE hInstance;
};
```

Listing 2: 聊天窗口线程参数

2. 原子状态标志
   封装程序核心状态的原子变量，确保多线程环境下的状态同步，避免数据竞争。包括 AI 是否正在显示回复、是否正在处理 AI 请求、聊天窗口线程是否运行、是否显示聊天窗口以及最后一次触发互动 / AI 请求的时间等标志, 代码如 Listing(3)所示。

```
struct AtomicFlags {
    // AI是否正在显示回复
    std::atomic<bool> isTyping{false};
    // 是否正在处理AI请求
    std::atomic<bool> isProcessingMessage{false};
    // 聊天窗口线程是否运行
    std::atomic<bool> chatThreadRunning{false};
    // 是否显示聊天窗口
    std::atomic<bool> showChatWindow{false};
    // 最后一次触发互动/AI请求的时间
    std::atomic<uint64_t> lastTriggerTime{0};
};
```

Listing 3: 原子状态标志

3.3.3 宠物动画相关

1. 宠物状态
   枚举宠物的所有行为状态，作为核心标志参与状态机的切换。包括放松（relax）、移动（move）、交互（interact）和坐下（sit）四种状态，代码如 Listing(4)所示。

```
enum class PetState {
    relax,    // 放松状态
    move,     // 移动状态
    interact,// 交互状态
    sit       // 坐下状态
};
```

Listing 4: 宠物状态

2. 宠物属性

存储宠物的运动属性与状态参数，用于物理运动模拟与动画控制。位置与运动参数包括 x 和 y 坐标（屏幕坐标）、x 和 y 方向的速度与加速度、最大速度等；状态参数包括当前状态、当前状态剩余时间（帧）、是否朝右（控制动画方向）以及是否处于坐下状态，代码如 Listing(5)所示。

```
struct PetAttribute {
    // 位置与运动参数
    float x;           // 宠物窗口左上角x坐标（屏幕坐标）
    float y;           // 宠物窗口左上角y坐标（屏幕坐标）
    float vx;          // x方向速度（像素/帧）
    float vy;          // y方向速度（像素/帧）
    float vmax;        // 最大速度（默认3.0f）
    float ax;          // x方向加速度（默认0.3f）
    float ay;          // y方向加速度（默认0.3f）
    // 状态参数
    PetState state;    // 当前状态
    int stateTimeLeft;// 当前状态剩余时间（帧）
    bool facingRight; // 是否朝右（控制动画方向）
    bool isSitting;    // 是否处于坐下状态
};
```

Listing 5: 宠物属性

3. 宠物动画配置

存储各状态的动画帧数量与帧率，用于统一管理动画播放参数。包括各状态（relax、move、interact、sit）的帧数量、帧间隔时间以及宠物窗口的宽度和高度，代码如 Listing(6)所示。

```
struct AnimationConfig {
    // 各状态帧数量（relax/move/interact/sit）
    const int frameCount[4] = {40, 46, 40, 320};
    // 帧间隔
    const int frameDelay = 1000 / 40;
    // 宠物窗口宽度
```

```
    const int windowWidth = 450;
    // 宠物窗口高度
    const int windowHeight = 450;
};
```

Listing 6: 宠物动画配置

4. 图像资源路径配置

存储不同状态和方向的宠物图像资源路径，用于统一加载和管理动画帧。路径按方向（0 代表朝右，1 代表朝左）和状态（0 代表 relax、1 代表 move、2 代表 interact、3 代表 sit）进行二维数组组织，便于根据宠物当前状态和朝向快速定位并加载对应的动画帧图像, 代码如 Listing(7)所示。

```
// 图像资源路径配置结构体
struct ImagePathConfig {
    // 图像资源路径，二维数组结构：[方向][状态]
    // 方向：0=朝右，1=朝左
    // 状态：0=relax，1=move，2=interact，3=sit
    const char* path[2][4] = {
        {
            "assets\\relax\\relax1\\build_char_002_amiya_R_25_",
            "assets\\move\\move1\\build_char_002_amiya_M_25_",
            "assets\\interact\\interact1\\build_char_002_amiya_I_25_",
            "assets\\sit\\sit1\\build_char_002_amiya_S_25_"
        },
        {
            "assets\\relax\\relax2\\build_char_002_amiya_R_w_25_",
            "assets\\move\\move2\\build_char_002_amiya_M_w_25_",
            "assets\\interact\\interact2\\build_char_002_amiya_I_w_25_",
            "assets\\sit\\sit2\\build_char_002_amiya_S_w_25_"
        }
    };

    // 生成完整图像文件名（拼接路径+帧序号+扩展名）
    std::string GetImageFilename(int dir, int state, int frameIndex)
    const {
        char filename[256];
        // 格式化路径：基础路径 + 三位数字帧号 + .png扩展名
        sprintf(filename, "%s%03d.png", path[dir][state], frameIndex);
        return std::string(filename);
    }
```

```
};
```

Listing 7: 图像资源路径配置

3.3.4 AI 聊天相关

1. 聊天消息参数
   存储单条聊天消息的完整信息，支持打字机效果的逐字符显示。包含消息完整文本（用户输入或 AI 完整回复）、当前显示文本（打字机效果中逐字符更新）、是否为用户发送的消息、消息是否显示完成以及当前显示到的字符索引（用于打字机效果）, 代码如 Listing(8)所示。

```
struct ChatMessage {
    // 消息完整文本（用户输入/AI完整回复）
    std::string fullText;
    // 当前显示文本（打字机效果中逐字符更新）
    std::string currentText;
    // 是否为用户发送（true=用户，false=AI）
    bool isUser;
    // 消息是否显示完成（打字机效果结束后设为true）
    bool isComplete;
    // 当前显示到的字符索引（用于打字机效果）
    int charIndex;
    // 构造函数：初始化用户消息（直接显示完整文本）
    ChatMessage(const std::string& text, bool isUser)
        : fullText(text), currentText(isUser ? text : ""),
          isUser(isUser), isComplete(isUser),
          charIndex(isUser ? text.length() : 0) {}
};
```

Listing 8: 聊天消息参数

2. 聊天历史管理器
   封装聊天历史的核心操作（增删查、状态同步），统一管理所有 ChatMessage 实例，避免数据散落在业务逻辑中, 代码如 Listing(9)所示。

```
class ChatHistoryManager {
private:
    // 存储所有聊天消息的容器
    std::vector<ChatMessage> messageList;
    // 聊天记录最大缓存数量（避免内存溢出）
    size_t maxHistorySize;
    // 当前待编辑消息的索引（如 AI 续写场景）
```

```
8       int currentEditingIndex;
9
10  public:
11      // 构造函数：初始化最大缓存容量
12      ChatHistoryManager(size_t maxSize) :
13          maxHistorySize(maxSize), currentEditingIndex(-1) {}
14
15      // 添加新消息（超过最大容量时删除最早消息）
16      void addMessage(const ChatMessage& msg) {
17          if (messageList.size() >= maxHistorySize) {
18              messageList.erase(messageList.begin());
19          }
20          messageList.push_back(msg);
21          // 新消息默认可编辑（如 AI 回复中）
22          currentEditingIndex = messageList.size() - 1;
23      }
24
25      // 获取指定索引的消息（用于渲染或更新）
26      ChatMessage& getMessageAt(size_t index) {
27          if (index >= messageList.size())
28          throw std::out_of_range("Message index out of range");
29          return messageList[index];
30      }
31
32      // 获取消息总数（用于渲染时计算列表长度）
33      size_t getMessageCount()
34          const { return messageList.size(); }
35
36      // 标记指定消息为“已完成”（打字机效果结束后调用）
37      void markMessageComplete(size_t index) {
38          if (index >= messageList.size()) return;
39          messageList[index].isComplete = true;
40          currentEditingIndex = -1; // 无正在编辑的消息
41      }
42  };
```

Listing 9: 聊天历史管理器

3. 聊天上下文
   存储聊天的全局上下文信息（如会话 ID、用户配置、上下文窗口），为消息生成和状态管理提供全局依赖, 代码如 Listing(10)所示。

```cpp
struct ChatContext {
    // 会话唯一 ID（用于本地缓存、多会话切换）
    std::string sessionId;
    // 用户 ID（区分多用户场景）
    std::string userId;
    // 是否启用打字机效果（全局配置开关）
    bool enableTypewriter;
    // 上下文窗口大小（AI 生成时参考的历史消息数量）
    int contextWindowSize;
    // 最后一条消息的时间戳（用于超时清理或排序）
    std::time_t lastMessageTime;
    // 扩展配置（如 AI 模型版本、消息字体）
    std::map<std::string, std::string> extraConfig;

    // 构造函数：初始化基础会话信息
    ChatContext(const std::string& sid, const std::string& uid)
        : sessionId(sid), userId(uid), enableTypewriter(true),
          contextWindowSize(20), lastMessageTime(std::time(nullptr)) {}

    // 更新最后消息时间戳
    void updateLastMessageTime()
        { lastMessageTime = std::time(nullptr); }

    // 设置扩展配置（如切换 AI 模型）
    void setExtraConfig(const std::string& key, const std::string& value) {
        extraConfig[key] = value;
    }

    // 获取扩展配置（如渲染时读取字体设置）
    std::string getExtraConfig(const std::string& key) const {
        auto it = extraConfig.find(key);
        return it != extraConfig.end() ? it->second : "";
    }
};
```

Listing 10: 聊天上下文

4. API 请求参数

封装 AI 请求的核心参数,用于构造符合 API 格式的 JSON 请求体。包括模型名称(如"Qwen/QwQ-32B")、API 密钥、API 请求地址、系统提示和用户输入的 prompt 等，提供设置 API 密钥和用户输入的方法, 代码如 Listing(11)所示。

```cpp
struct APIRequestParam {
    // 模型名称（"Qwen/QwQ-32B"）
    std::string modelName;
    // API密钥
    std::string apiKey;
    // API请求地址
    // "https://api.siliconflow.cn/v1/chat/completions"
    std::string apiUrl;
    // 系统提示
    std::string systemPrompt;
    // 用户输入的prompt
    std::string userPrompt;
    // 构造函数：使用默认配置初始化（可通过setter修改）
    APIRequestParam()
        : modelName("Qwen/QwQ-32B"),
          apiUrl("https://..."),
          systemPrompt("你是明日方舟的阿米娅，...") {}
    // Setter方法：修改API密钥
    void SetApiKey(const std::string& key) { apiKey = key; }
    // Setter方法：修改用户输入
    void SetUserPrompt(const std::string& prompt)
        { userPrompt = prompt; }
};
```

Listing 11: API 请求参数

3.3.5 交互控制相关

1. 鼠标事件信息参数
   封装鼠标事件的关键信息，用于统一处理鼠标输入（如拖动、点击）。包括鼠标消息类型（如左键按下、右键抬起）、鼠标在屏幕上的坐标、在父窗口中的客户端坐标以及当前指向的窗口句柄（区分宠物窗口和聊天窗口），并提供判断是否为左键或右键按下的辅助方法, 代码如 Listing(12)所示。

```cpp
struct MouseEventInfo {
    // 鼠标消息类型（WM_LBUTTONDOWN、WM_RBUTTONUP）
    WPARAM wParam;
    // 鼠标在屏幕上的坐标
    POINT screenPos;
    // 鼠标在父窗口中的客户端坐标
    POINT clientPos;
    // 鼠标当前指向的窗口句柄（区分宠物窗口/聊天窗口）
```

```
    HWND targetHwnd;
    // 辅助方法：判断是否为左键按下
    bool IsLButtonDown() const
        { return wParam == WM_LBUTTONDOWN; }
    // 辅助方法：判断是否为右键按下
    bool IsRButtonDown() const
        { return wParam == WM_RBUTTONDOWN; }
};
```

Listing 12: 鼠标事件信息参数

2. 鼠标钩子配置参数
存储全局鼠标事件的详细信息，用于捕获和处理鼠标与宠物窗口的交互, 代码如 Listing(13)所示。

```
// 通过钩子参数转换获取
MOUSEHOOKSTRUCT* p = (MOUSEHOOKSTRUCT*)lParam;
// 核心字段
typedef struct {
    POINT pt;             // 鼠标在屏幕上的坐标
    HWND hwnd;            // 鼠标指向的窗口句柄
    UINT wHitTestCode;    // 命中测试代码
    LPARAM dwExtraInfo;// 额外信息
} MOUSEHOOKSTRUCT;
```

Listing 13: 鼠标钩子配置参数

3. 右键菜单配置参数
存储右键菜单的控件布局与文本，用于统一管理菜单样式。包括菜单的宽度和高度，Sit 按钮和 Chat 按钮相对于菜单左上角的位置，按钮的文本（Sit 按钮文本根据状态切换），以及菜单和按钮的颜色配置（如菜单背景色、按钮默认色、激活色和 Chat 按钮颜色）, 代码如 Listing(14)所示。

```
struct MenuConfig {
    const int menuWidth = 180;  // 菜单宽度（像素）
    const int menuHeight = 135; // 菜单高度（像素）
    // 按钮相对位置（相对于菜单左上角）
    const WindowRect sitButtonRel{15, 15, 165, 60};  // Sit按钮
    const WindowRect chatButtonRel{15, 75, 165, 120};// Chat按钮
    // 按钮文本（根据状态切换）
    // 未坐下/已坐下
    const wchar_t* sitBtnText[2] = {L"Sit", L"Sitting"};
    // Chat按钮固定文本
    const wchar_t* chatBtnText = L"Chat";
```

```
12        // 颜色配置
13        // 菜单背景色
14        const COLORREF menuBgColor = RGB(240, 240, 240);
15        // 按钮默认色（Sit）
16        const COLORREF btnNormalColor = RGB(200, 255, 200);
17        // 按钮激活色（Sitting）
18        const COLORREF btnActiveColor = RGB(255, 200, 200);
19        // Chat按钮颜色
20        const COLORREF chatBtnColor = RGB(200, 200, 255);
21    };
```

Listing 14: 右键菜单配置参数

3.4 接口设计

3.4.1 窗口控制模块

1. 初始化窗口
   初始化宠物窗口（无框、分层透明、置顶样式），创建窗口句柄，代码如 Listing(15)所示。

```
1  void InitializeWindow(
2      HWND& hwnd,
3      int width,
4      int height,
5      HINSTANCE hInst
6  )
```

Listing 15: 初始化宠物窗口接口

2. 创建聊天窗口
   创建聊天窗口（含历史显示区、输入区、发送按钮），并绑定到宠物窗口作为父窗口, 代码如 Listing(16)所示。

```
1  HWND CreateChatWindow(
2      HWND parentHwnd,
3      HINSTANCE hInst,
4      DesktopPet* pet
5  )
```

Listing 16: 创建聊天窗口接口

3. 其他

   - 配置窗口透明度
     SetWindowTransparent

- 移动窗口到指定屏幕坐标，并设置是否置顶
  MoveWindowPos
- 显示 / 隐藏窗口（隐藏时不销毁句柄）
  ShowOrHideWindow
- 销毁聊天窗口，释放资源并置空句柄
  DestroyChatWindow

3.4.2 宠物动画模块

1. 加载宠物动画
   加载所有状态（relax/move/interact/sit）、所有朝向（朝右 / 朝左）的动画帧图像，代码如 Listing(17)所示。

```
bool LoadImageResources(
    const ImagePathConfig\& pathConfig,
    IMAGE* (&petImages)[2][4][320],
    const AnimationConfig& animConfig
)
```

Listing 17: 加载宠物动画接口

2. 绘制动画帧
   将当前动画帧绘制到宠物窗口，应用透明混合效果，代码如 Listing(18)所示。

```
bool DrawAnimationFrame(
    HWND hwnd,
    const IMAGE* (&petImages)[2][4][320],
    const PetAttribute& petAttr,
    int frameIndex,
    const BLENDFUNCTION& blend
)
```

Listing 18: 绘制动画帧接口

3. 其他
   - 切换宠物状态，重置状态剩余时间与帧索引
     SwitchPetState
   - 更新帧索引（循环播放，超出帧数量时重置为 0）
     UpdateAnimationFrame
   - 释放已加载的图像资源，避免内存泄漏
     ReleaseImageResources

3.4.3 AI 聊天模块

1. 添加新消息
   向聊天历史添加新消息（用户消息直接完整显示，AI 消息初始为空以支持打字机效果）, 代码如 Listing(19)所示。

```
bool AddChatMessage(
    ChatHistoryManager& historyMgr,
    std::mutex& historyMutex,
    const std::string& text,
    bool isUser
)
```

Listing 19: 添加新消息接口

2. 发送 HTTP POST 请求
   向 AI 接口发送 HTTP POST 请求，获取原始响应（基于 libcurl 封装）, 代码如 Listing(20)所示。

```
std::string SendAPIRequest(
    const std::string& jsonReq,
    const APIRequestParam& reqParam
)
```

Listing 20: 发送 HTTP POST 请求接口

3. 其他

   - 更新 AI 消息的显示文本（打字机效果，按字符索引截取）
     UpdateAIMessage
   - 渲染聊天历史到聊天窗口的“历史区”控件
     RenderChatHistory
   - 构造 AI 请求的 JSON 体（含系统提示、历史上下文）
     BuildAPIRequest
   - 解析 AI 接口返回的 JSON，提取回复文本
     ParseAPIResponse

3.4.4 交互控制模块

1. 设置全局鼠标钩子设置全局鼠标钩子（WH_MOUSE_LL），捕获所有鼠标事件（如点击、移动）, 代码如 Listing(21)所示。

```
HHOOK SetMouseHook(
    HINSTANCE hInst,
    HOOKPROC hookProc
```

```
)
```

Listing 21: 设置全局鼠标钩子接口

2. 处理右键菜单按钮点击事件处理右键菜单按钮点击事件，判断点击位置是否在按钮区域内，并更新对应状态（坐下 / 显示聊天窗口）, 代码如 Listing(22)所示。

```
bool HandleMenuClick(
    const POINT& clientPos,
    const MenuConfig& menuConfig,
    bool& isSitting,
    bool& needShowChat
)
```

Listing 22: 处理右键菜单按钮点击事件接口

3. 其他

   - 卸载全局鼠标钩子，释放系统资源
     UnsetMouseHook
   - 处理窗口拖拽逻辑（左键按下开始拖拽，移动时更新位置）
     HandleDragEvent
   - 在鼠标位置显示右键菜单（含 Sit/Chat 按钮）
     ShowRightMenu
   - 隐藏右键菜单（清空绘制区域并重绘窗口）
     HideRightMenu

4 编码实现

Listing 23: 编码实现

5 测试