

UEFI Development Exploration 45 – GuiLite Overview

原创

luobing4365

Posted on 2020-02-28 11:42:47

Read 1.7k

Collection 5

Likes 1

copyright

Category columns:


UEFI Development

Article Tags:

GUIList

Graphics Library

Graphics Programming

UEFI Development

This column includes this content

503 Subscribe

104 articles

Subscribe to our column

(Please keep it-> Author: Luo Bing <https://blog.csdn.net/luobing4365>)

When I started this exploration series, I planned to port a complete GUI library to UEFI.

In fact, a large part of the code for developing various graphics programming and special effects comes from another project of mine - Foxdisk. In this project, task switching is implemented to some extent, which can be regarded as a small shell interface with graphical display.

However, the keyboard processing in Foxdisk is completely separated from the graphics, and the mouse processing is not implemented. In short, this is a very loose and modular GUI library. It is far from what I expect, similar to MFC, QT, JUCE and other libraries.

UEFI is not a complete operating system, and the GUI library you choose is actually better to be embedded. After a brief review of several open source GUI libraries, including LearningGUI, GuiLite, littlevgl, etc., I chose to start this work from GuiLite.

On the one hand, the amount of code does not seem to be large, about 6,000 lines, so it should not take too much time; on the other hand, this open source library was developed by a Chinese, and I have also joined the QQ group created by the author, so I can easily ask any questions I have.

1 GuiLite class organization

According to its documentation, GuiLite only does two things: interface element management and graphics drawing. Graphics drawing does not depend on interface management and can exist independently to cope with the need to be ported to a single-chip environment with limited resources.

The class organization chart is as follows:

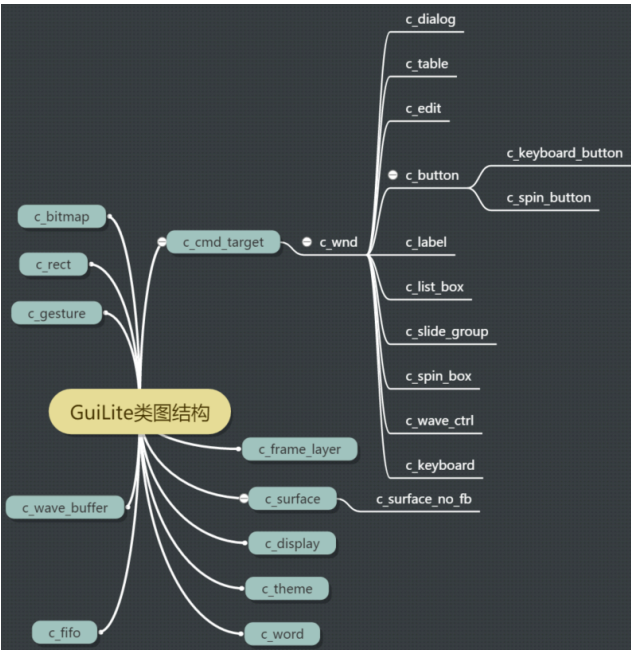


Figure 1 GuiLite class diagram

The class diagram of GuiLite is a bit like the structure of MFC, and if you go deeper, you will find that its message processing mechanism is also very similar to MFC. The author has refined the necessary parts and made a streamlined graphics library suitable for multiple platforms, which is amazing.

The `c_cmd_target` class is mainly used to handle message positioning, and the core function is `find_msg_entry()`;

The `c_wnd` class defines the basic framework of the window. Most control classes, including `c_dialog` and `c_button`, are derived from it.


`c_display` and `c_surface` are mainly used to implement virtual drawing of graphics. The so-called virtual drawing is not directly drawing on the hardware, but a memory buffer is defined and pixels are arranged in the buffer. In other words, it provides a virtual framebuffer, and application developers can directly write buffer data to the hardware.


The `c_bitmap` and `c_word` classes implement the drawing of bmp images (still writing data into the memory buffer). The core message transmission and graphics drawing. If you understand these classes, there will be no major problems.


2 Message Mechanism


GuiLite's window (`c_wnd`) instances are all linked together at runtime. This is a large linked list that all messages (including mouse events, etc.) can be traced back to their subordinate windows in this linked list.

After logging in, you can enjoy the following benefits:

Free Copy Code

Interact with bloggers

Download massive resources

Post updates/write articles/join the community

Sign in now

The `c_wnd` class contains the following member variables:

```
c_wnd* m_parent;
c_wnd* m_top_child;
c_wnd* m_prev_sibling;
c_wnd* m_next_sibling;
```

They are the parent window, the first child window, the front sibling window and the back sibling window respectively. The linked list is implemented based on these class pointers.

Another key data structure related to the message mechanism is `WND_TREE`, which is defined as follows:

```
typedef struct struct_wnd_tree{    c_wnd* p_wnd;    unsigned int resource_id;    const char* str;    short x;    short y;    short width;
short height;    struct struct_wnd_tree* p_child_tree; }WND_TREE;
```

It is equivalent to the resource file of the window, which contains the pointer of the window instance, resource ID, string and other information. The construction of the window list is based on this information provided by the user. The core implementation function is `c_wnd::connect()`.

The steps to establish the entire mechanism are as follows:

1) Construct an array of `WND_TREE` type , for example:

```
WND_TREE s_dialog_widgets[] =
{
    { &s_dialog_button, ID_DIALOG_BUTTON, "Button", 100, 100, 100, 50},
    { &s_dialog_exit_button, ID_DIALOG_EXIT_BUTTON, "Exit", 100, 200, 100, 50},
    {NULL, 0, 0, 0, 0, 0, 0}
};

WND_TREE s_main_widgets[] =
{
    { &s_edit1, ID_EDIT_1, "ABC", 150, 50, 100, 50},
    { &s_edit2, ID_EDIT_2, "123", 400, 10, 100, 50},

    { &s_label_1, ID_LABEL_1, "label 1", 150, 100, 100, 50},
    { &s_label_2, ID_LABEL_2, "label 2", 150, 170, 100, 50},
    { &s_label_3, ID_LABEL_3, "label 3", 150, 240, 100, 50},

    { &s_button, ID_BUTTON, "Dialog", 400, 100, 100, 50},
    { &s_spin_box, ID_SPIN_BOX, "spinBox", 400, 170, 100, 50},
    { &s_list_box, ID_LIST_BOX, "listBox", 400, 240, 100, 50},

    { &s_my_dialog, ID_DIALOG, "Dialog", 200, 100, 280, 312, s_dialog_widgets},
    {NULL, 0, 0, 0, 0, 0, 0}
};
```

Figure 2 Window tree example (from the HelloWidget example)

It should be noted that the last item of the array must be empty, and the program uses this as the end judgment during the scanning process.

2) Use the `connect()` function to connect all windows. For example:

```
s_my_ui.connect(NULL, ID_ROOT, 0, 0, 0, UI_WIDTH, UI_HEIGHT, s_main_widgets); (Excerpted from HelloWidget's Uicode.cpp line166).
```

After this function is executed, `s_my_ui` is used as the parent window, and all the above windows (`s_edit1`, `s_edit2`, etc.) are connected through their own member variables `m_parent`, `m_top_child`, `m_prev_sibling`, and `m_next_sibling`.

3) Message processing.

GuiLite has prepared a callback mechanism for message processing, which is mainly implemented through the following macros and processing function `find_msg_entry()` (member function of `c_cmd_target` class):

```
#define GL_BEGIN_MESSAGE_MAP(theClass) \
const GL_MSG_ENTRY* theClass::get_msg_entries() const \
{ \
    return theClass::m_msg_entries; \
} \
const GL_MSG_ENTRY theClass::m_msg_entries[] = \
{ \
#define GL_END_MESSAGE_MAP() \
{MSG_TYPE_INVALID, 0, 0, 0}};
```

Figure 3 Message macro (from the HelloWidget example)

After logging in, you can enjoy the following benefits:

- Free Copy Code
- Interact with bloggers
- Download massive resources
- Post updates/write articles/join the community

The data structure `GL_MSG_ENTRY` contains the message ID and the corresponding callback function. In the member function `notify_parent` of `c_wnd`, `find_msg_entry` is called to call the callback function according to the message ID.

After the above window list and message mechanism are established, all key and gesture messages (such as mouse, touchpad, etc.) can be traced back in the list until the subordinate window is found and the corresponding processing function is called.

3. Key points of programming

GuiLite is still under maintenance, so it may be updated at any time. However, most of the mechanisms have been established, and the subsequent application code writing process is similar.

Generally, you can follow the following steps to write:

- 1) Build your own window class and display the required window elements, including various controls, and connect these windows through the connect function;
- 2) Implement message processing functions in the written window class, especially the processing of mouse and keyboard messages;
- 3) Write platform-related processing functions, including docking with the platform's mouse messages, keyboard messages, and graphics drawing. Taking the MFC mouse left-click message as an example, it establishes the processing function `OnLButtonDown` of the `WM_LBUTTONDOWN` message, and the corresponding relationship with the `on_touch` of the GuiLite window class should be established in this function.

The rest of the details are not explained one by one. After learning this, I think I can start the UEFI porting work. In the next article, I will first build a UEFI program framework that supports C++ to complete the initial idea.

(I asked a lot of questions during the learning process. I would like to thank the author for his patient guidance^^ [Open source](#) code address: <https://gitee.com/idea4good/GuiLite/blob/master/documents/HowToWork-cn.md> and <https://github.com/idea4good/GuiLite>)

about Us Careers Business Cooperation Seeking coverage 400-660-0108 kefu@csdn.net Online Customer Service Working hours 8:30-22:00
Public Security Registration Number 11010502030143 Beijing ICP No. 19004658 Beijing Internet Publishing House [2020] No. 1039-165
Commercial website registration information Beijing Internet Illegal and Harmful Information Reporting Center Parental Control
Online 110 Alarm Service China Internet Reporting Center Chrome Store Download Account Management Specifications
Copyright and Disclaimer Copyright Complaints Publication License Business license
©1999-2025 Beijing Innovation Lezhi Network Technology Co., Ltd.

After logging in, you can enjoy the following benefits:

- Free Copy Code
- Interact with bloggers
- Download massive resources
- Post updates/write articles/join the community