國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

具備可延展性與 可移植性之圖形介面函式庫

A Scalable and Portable GUI Library

陳世霖

Sei-Lim Chen

指導教授：鄭士康 博士

Advisor: Shyh-Kang Jeng, Ph.D.

中華民國 98 年 6 月

June, 2009

誌謝

誠摯的感謝我的指導老師鄭士康教授，在我的學習過程中，指導我正確的研究方向、最有效率的研究方法，並從中學習到做研究的精神和正確的態度。

感謝口試委員王勝德博士與顏嗣鈞博士，因為有您們的指導，使得本論文能夠更完整。

也感謝博理 510 的諸位伙伴們，因為有你們的陪伴與鼓勵，讓我的實驗室生活變得豐富而精采。

僅以此論文獻給我最親愛的家人與關心我的師長朋友，感謝你們的包容與愛護，願與你們分享這份榮耀。

陳世霖

中華民國九十八年六月

摘要

本研究開發了一套獨立的圖形介面函式庫（GUI library），用來增加圖形介面程式的延展性與移植性，並試圖提高應用程式的跨平台支援程度。在參考了相關的開放原始碼之後，並依循軟體工程的程式開發原則，首先切割物件模型，定義出相關的抽象介面模型，然後實作程式的核心部份。在完成核心程式碼之後，開始設法移植到特定的作業平台移植。選定的是微軟的視窗作業系統（Windows Xp SP2）、Linux（Frame Buffer）與 Bootloader（Non-OS），並整合向量繪圖函式（GPLFlash）做動畫展示。

關鍵字：圖形介面、延展性、移植性、嵌入式、視窗

Abstract

This thesis presents a GUI library used to enhance the scalability and portability of our GUI applications. We also try to solve the problems of software on crossing platforms. As the first step, we refer to some source codes of open source projects. Then following the software engineering priciples, we separate the whole library into pieces of object modules, and define their abstract interfaces. Then it takes some time to implement those core modules. When the kernel part is finished, the next step is to execute the porting procedure for a specific platform. Microsoft Window XP SP2, Linux, and Bootloader are chosen for migration. Besides, it integrates an open source Flash animation playing library for exhibition.

Keywords: GUI, scalability, portability, embedded, window

CONTENTS

LISTS OF FIGURES

Chapter 1 Introduction

1.1 Motivation

In our daily life, we have so many chances to extend our working experience of GUI. But some successful programs are binding on a specific operating system because of their basic GUI toolkit. How can we conquer this restriction? For giving more scalability and portability to GUI programs, we try to satisfy these requirements from the low level library. A new GUI library is needed. The scalability of our GUI library could go through tiny system (non-OS) and large system (Windows XP). The tiny systems, like BIOS or Bootloader, are getting popular on embedded system recently and transiting from text mode to graphic mode which usually supports OSD (On-Screen Display) menu controls. So we develop a simple GUI library to satisfy this requirement.

1.2 Literature Survey

Before a GUI library is really implemented, we have to realize how dose a GUI application work. There is an exhibition of GUI control loop in [1]. Some knowledge of display technology is given in [2] [3].

The component-based approach is adopted for software reuse [4] [5]. This approach reduces development time and improves product quality. A GUI library should be easy to understand, learn, and use; should provide appropriate performance; should be able to control used amount of resource [4].

In [6], the development principle of an interactive program is given. To keep everything simple is always on top priority. By the contributions of [7] [8] [9] [10], we know how many factors would influence the software portability. The types of CPUs, run-time environment, development toolkits, and extra peripheral hardware are all included. Because there are fewer studies on low level GUI since 1980s', we could find only some modern high level GUI reference like [11] [12] [13].

1.3 Approach

A new GUI library is designed to maximize the reuse of modules in [8]. To take off the source code coupling to native GDI is in the first priority in [3] [14] [15] [16]. So the goal is very clear, a common window-oriented GUI library for higher level applications. Following the tradition of GUI libraries, the fundamental unit for display would be a window. There is no complicate algorithm for this GUI library, and every thing is focused on reducing the maintenance efforts and increasing real-time response. It uses the software engineering principles [17] to enhance the scalability and portability of this library [18].

1.4 Contribution

We present a new architecture for GUI object modeling. Here we ignore the traditional realization of window, pixmap, and image. Our design merges the pixel buffer into a window to unify painting behaviors, and to reduce system complexity. Programmers do not have to worry about which object they should use. In addition, here we support window only. A window is what we see exactly on the screen. And we can change the window content by accessing pixel buffer.

Light-weight and easy to use is our goal. We offer a new thought for GUI development, though it is still not so mature.

1.5 Organization of Thesis

This thesis is divided into five chapters: Chapter 1 gives an introduction. Chapter 2 describes the background of our techniques. Chapter 3 explains the design and implementation of our GUI library. The experiment results are shown in Chapter 4. And we would give the conclusions in Chapter 5.

Chapter 2 Background

2.1 Computer Graphics

By referring to the book written by J. Foley in 1995 [19], we could know that modern computers are almost all based on raster display devices like CRT, or LCD monitors. Just like the image shown in Fig.2.1, what we see on our monitor is the rectangular grid which consists of color points. And each point is a ***Pixel***.

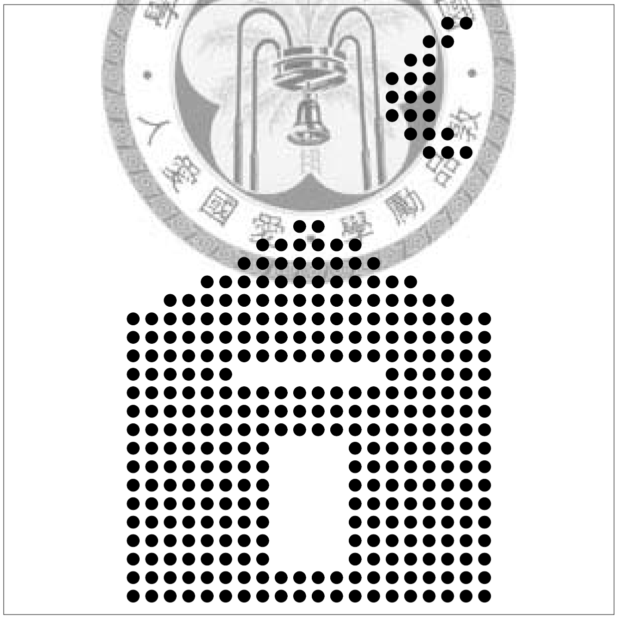


Fig. 2.1 A digital image in dots

2.2 Pixel and Color Model

According to the description of pixel on the famous web site Wikipedia, at the page of “http://en.wikipedia.org/wiki/Pixel”, we got the meaning of pixel. It is a combination of ***picture*** and ***element*** first unveiled on 1930s’.

A pixel is the smallest item of information in an image. In monochrome mode, we do not have to concern about the color of the pixel. Each pixel is represented by either black or white, and 1 bit is enough to store the pixel information. But when it comes to color image, it needs a color model to describe the specific color of a pixel.

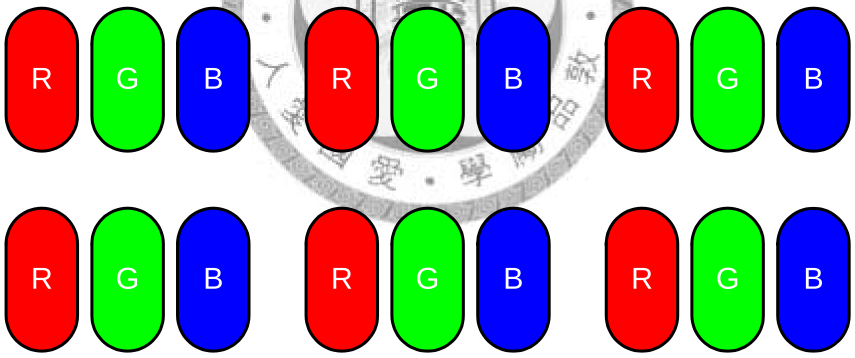


Fig. 2.2 The arrangement of RGB colors in raster display

Figure 2.2 is an example of colorful pixels. The most popular way to use is an additive color model, called RGB model, in which red, green, and blue lights are added together to produce other colors. By varying the luminance of each color, we could reproduce an approximate color for our requirement. When we assign each light luminance as an 8-bit length digital variant, the number of colors could reach 2^{24} = 16,777,216.

Beside 24 bits, we could also have a 16-bit mode for a pixel which could further branches into RGB555 and RGB565. In the mode of RGB555, it uses 5 bits to represent red, 5 bits for green, and 5 bits for blue. And it becomes 5 bits, 6 bits, and 5 bits for RGB565. When we use a 16-bit mode for a pixel, the depth of color would be 16 bpp (bits per pixel).

2.3 Frame Buffer

According to the introduction of [3], "***A frame buffer is basically a multi-ported memory with one of its ports specialized and dedicated to refreshing a raster display***". We could have a brief illustration of frame buffer in Figure 2.3.

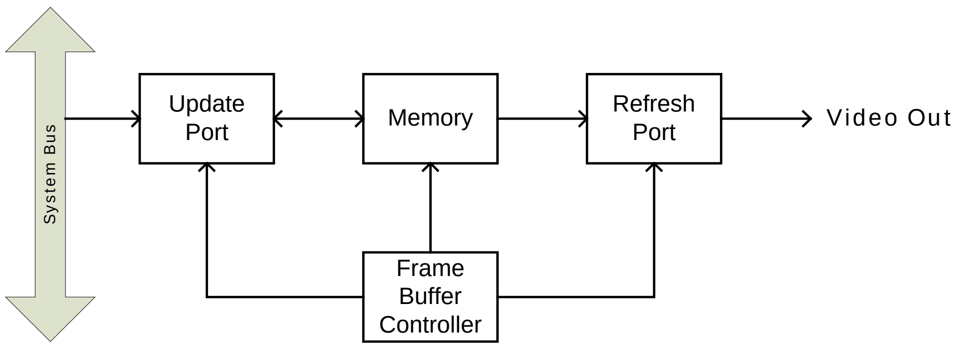


Fig. 2.3 The Frame Buffer architecture

But the system may not be with a dual port memory for cost down in modern design. Here we have an LCD controller block diagram to describe the raster scan of a display device like LCD or CRT in Fig 2.4.

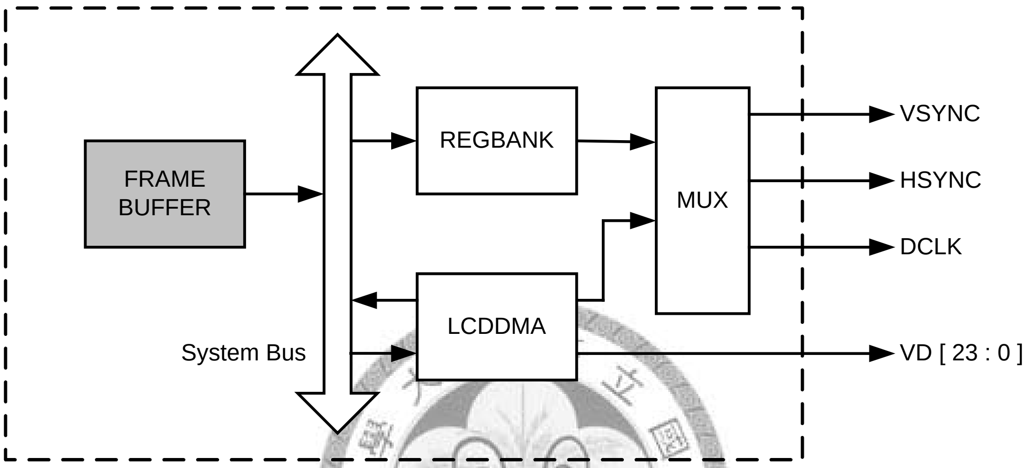


Fig. 2.4 An illustration of LCD controller

Because the display device is treated as a raster grid, the Frame Buffer holding the information of pixels could also be arranged as a 2-D matrix of memory. We can say that is a bitmap.

Fig. 2.5 is used to describe a bitmap composed by a 2-D array of 8 rows and 6 columns. The left side of Fig. 2.5 is the bitmap context stored in memory. And the right side is the result of output display as an 8x6 image.

0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	1	0
0	1	1	0	0	1	1	0

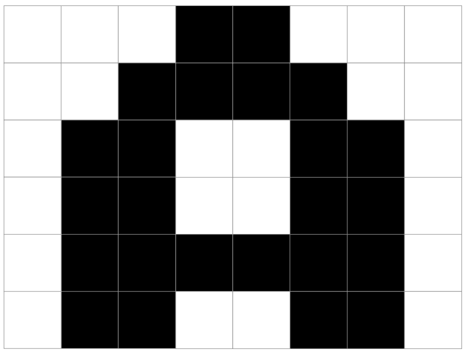


Fig. 2.5 An example for bitmap

It is so convenient to modify the bitmap by indicating the memory address of a specific pixel. But when we want to alter a whole block of pixels, the better way is Bit Block Transfer.

2.4 BitBlt

BitBlt stands for Bit Block Transfer. It is a computer operation to carry one bitmap into another in which two bitmaps may be in different size and the location are also not concrete already. We could realize that it is a process of memory copy from one block to another.

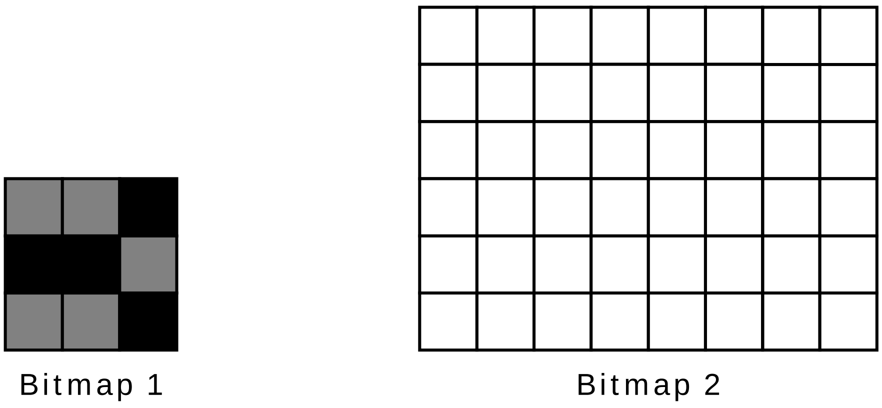


Fig. 2.6 Two bitmaps prepared for BitBlt

Now we could have 2 original bitmaps in Figure 2.6. Bitmap 1 could be located any where in the bitmap2 like in Fig.2.7, as case A and case B.

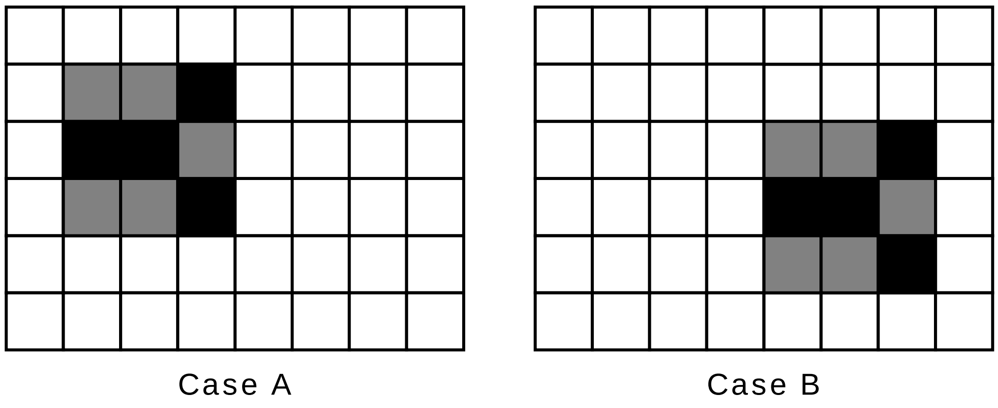
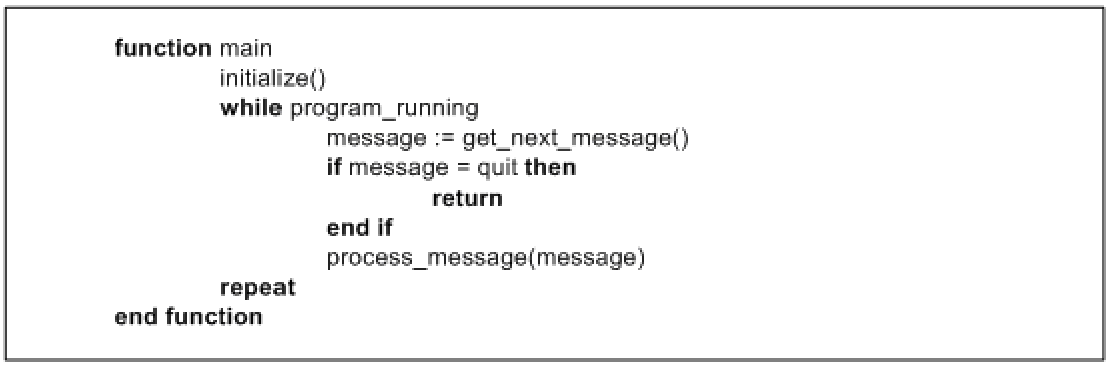


Fig. 2.7 Two different results of BitBlt

2.5 Event Driven

According to [1] [20], an event loop already becomes the standard for dealing users' input in interactive system. For a basic concept of event-loop, we have a set of pseudo code for exhibition in Figure 2.8.



```
function main
    initialize()
    while program_running
        message := get_next_message()
        if message = quit then
            return
        end if
        process_message(message)
    repeat
end function
```

Fig. 2.8 The pseudo code of event loop

2.6 Software Portability

Portability is a software feature supporting developers to reuse their source code in another environment. When we attempt to move the software to another platform, we usually encounter some impediment factors [7] [9].

2.6.1 Processor

The processor architecture occupies the most important place of software porting. It determines the features of the software that could be enabled or not directly. We cite three major issues for various processors.

(A) Basic data type

The instruction and register of the CPU would be possible of 8, 16, 32, or 64 bits length shown in Figure 2.9.

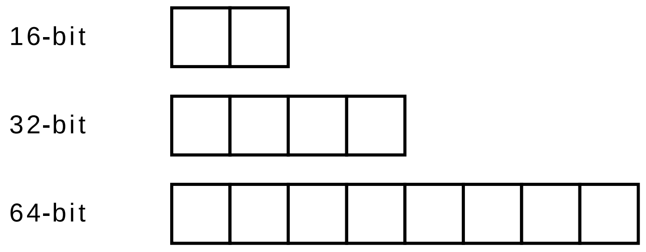


Fig. 2.9 Different kinds of basic data type

(B) Byte Order

There are two kinds of byte order as big-endian and little-endian like those shown in Figure 2.10.

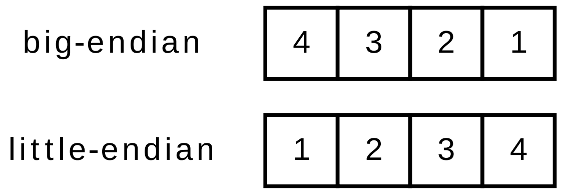


Fig. 2.10 The demonstrations of byte order

(C) Alignment and Padding

For acceleration, machines might access scalar data items in memory by the multiples of basic data type. Each item in memory was all addressed by multiples of bus width. But it usually leaves a “***hole***” for bus alignment. Fig.2.11 shows the situations of a 32-bit aligned memory. Bytes in gray are “***holes***” and not effective at all.

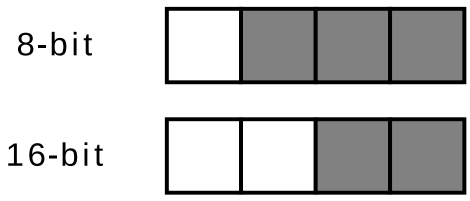


Fig. 2.11 The demonstrations of padding

2.6.2 Operating System

Even by the same OS, the same system call could still be different from machine to machine. And some system call depends on special processors or implementation.

For example, when we want to open a file, in UNIX OS we may use system call "***open***" to easily achieve our goal. But it is not working in the Windows environment, for WIN32 has its own file handle APIs with that "***CreateFile***" is designed to do the same thing as open in UNIX.

2.6.3 Compiler and Build Environment

Even in the most popular C language, all compilers could have different versions which may support different grammars or syntaxes. Going through the history of C language from 1970s, it already has many revisions for improvement.

To overcome the issue of compiler and dependent library, we choose GNU C to be our standard tool chain, and it will be introduced soon in subsection 2.7.1.

2.6.4 Hardware

Some software may rely on a 3D accelerator or other particular hardware device like a camera.

2.7 Related Open Source Projects

Our program depends on GNU Compiler Collection which includes gcc, and make, etc. (GCC – http:// directory.fsf.org/project/gcc/)

The MinGW/MSYS (http://www.mingw.org/) is a GCC port for for WIN32 environment. We use it to compile our GUI library on Windows XP.

After tracing Qt (http://www.qtsoftware.com/) source code in C++, we decided to develop an easier package. For the consistence of our GUI library, SDL (http://www.libsdl.org/) is so important because we refer to its surface definitions and rapid BitBlt functions. And we get the layering idea from GTK (http://www.gtk.org/).

Chapter 3 Design

3.1 Architecture

The GUI program could be simplified into a cycle of request and response [20]. Fig. 3.1 is a simple interaction cycle loop.

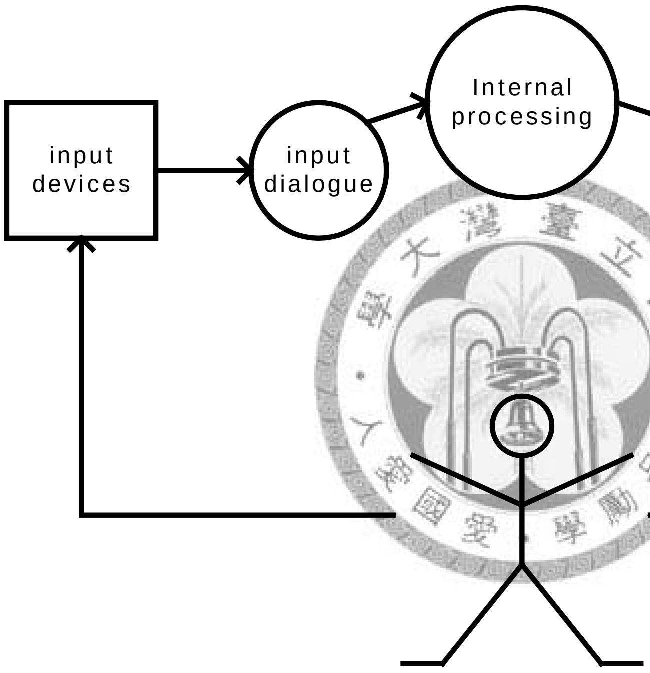


Fig. 3.1 The interaction of a GUI [1]

Then we can rotate the diagram and get more details in Fig. 3.2. The scope in gray is usually the GDI supported by each platform or OS. Modern smart GUI libraries often design an abstract layer or object model to raise the source code reusability of application.

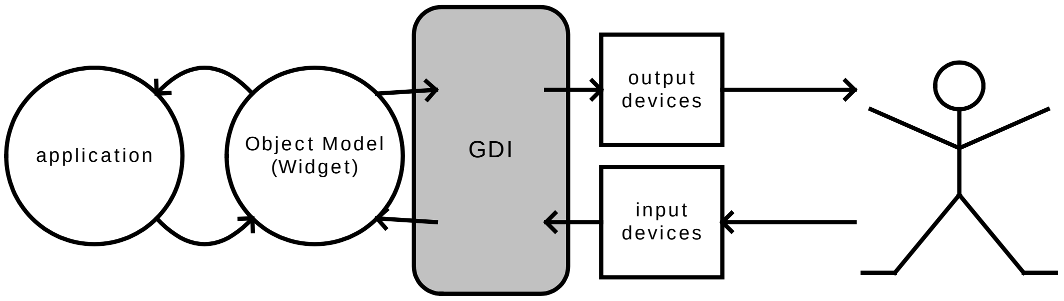


Fig. 3.2 To get more details of GUI

To replace the functionality of ***GDI***, our GUI library makes a window to carry a pixel buffer by itself to unify the programming style and simplify porting procedure.

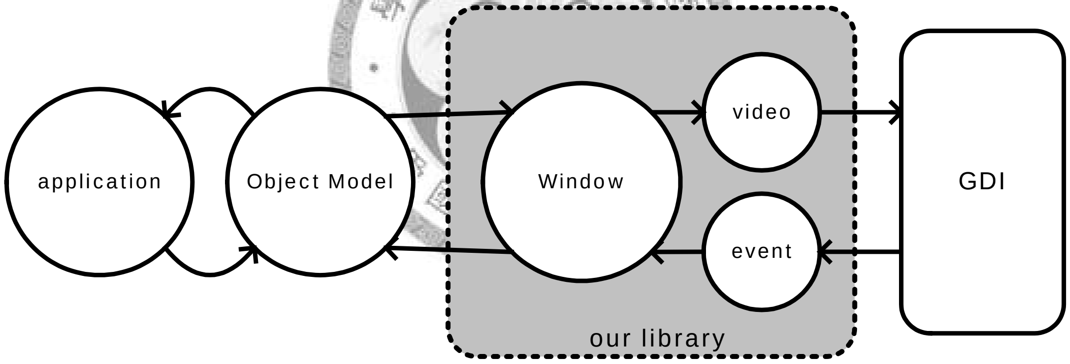


Fig. 3.3 To insert a new glue layer into GUI

Our library eliminates all things out of our interest for simplicity, like traditional GDI paintings which are in charge of drawing lines or polygons, etc. We deal with basic window behaviors only, including ***size***, ***show***, ***hide***, ***move***, and ***draw***, etc. We reserve the flexibility for varieties of optimized rendering engines.

To raise the portability of our library, we separate it into 4 parts of modules. They are Video, Event, Window, and BitBlt as what we show in Fig. 3.4. Our idea is to keep Window permanent and to switch others by platforms.

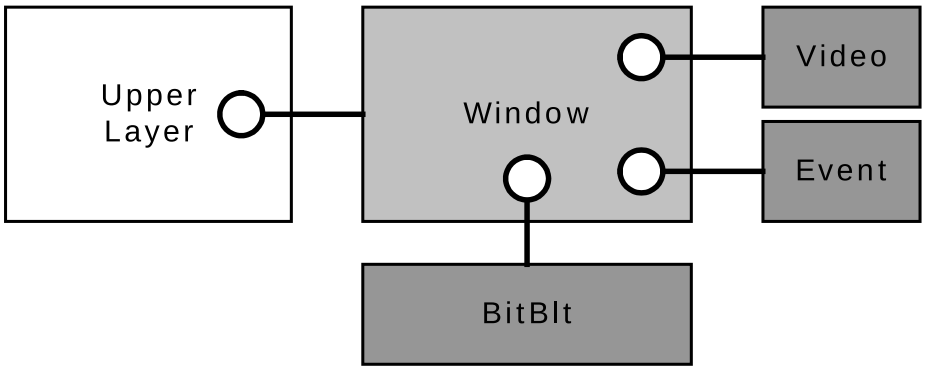


Fig. 3.4 The diagram of our GUI modules

3.2 Screen

Screen is the target for real display. It might be a Frame Buffer or just a sub-window in native GDI environment. Figure 3.5 shows its relationship to upper layer shadow (See next section).

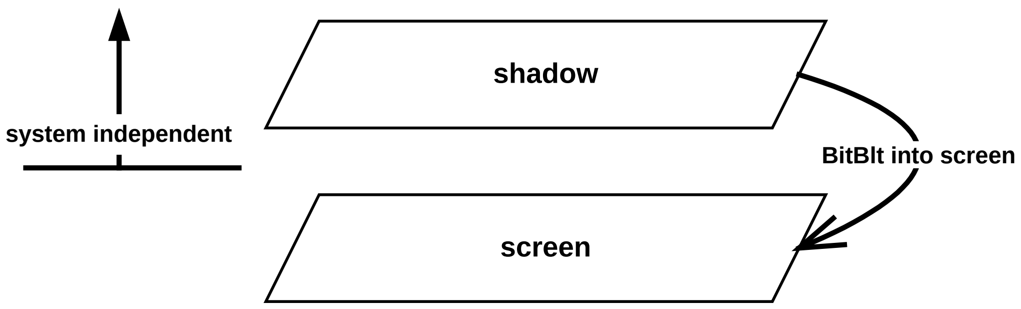


Fig. 3.5 The role of screen

3.3 Shadow

Because the operation of bit block transferring copies only a complete rectangular bitmap each time, we have to concern the refresh blinking of the bottom window. To avoid those covered area appearing on the screen, we need a hidden bitmap named Shadow to handle the progress of BitBlt. Figure 3.6 is the concept of shadow.

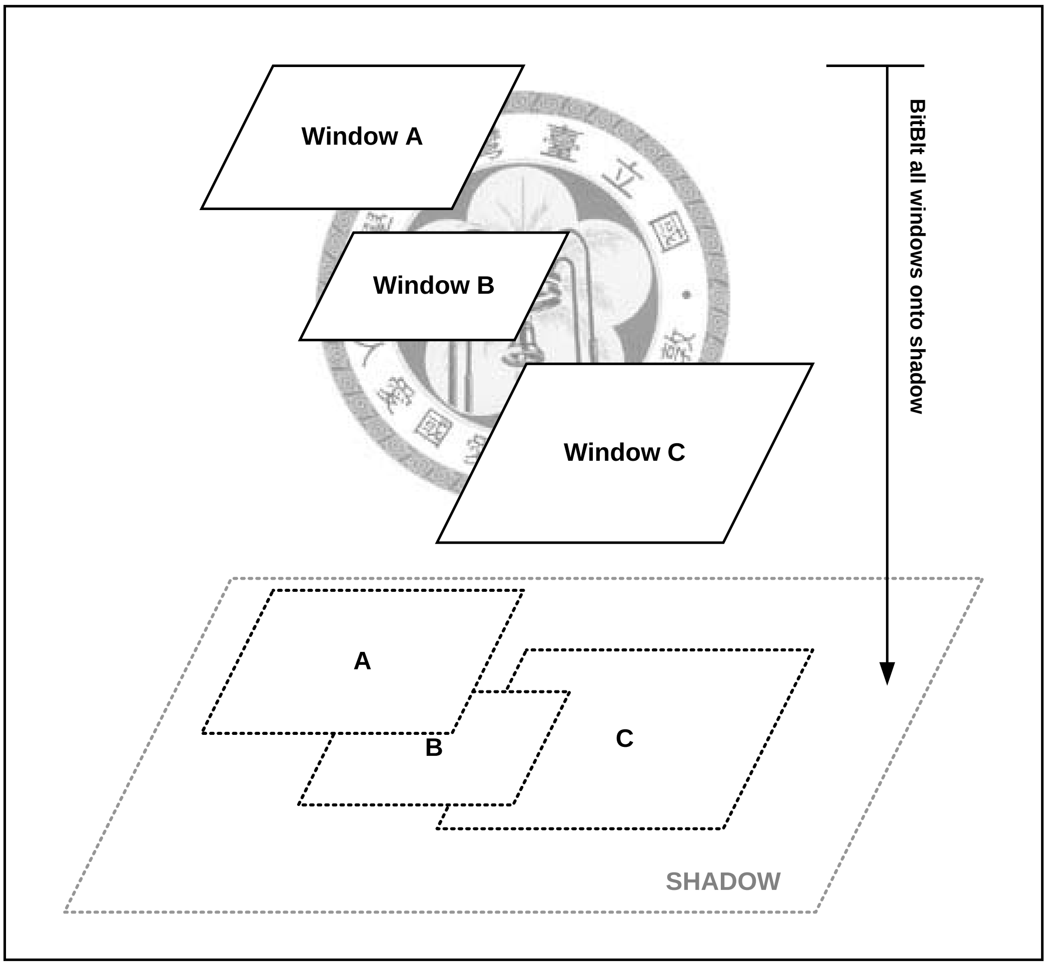


Fig. 3.6 The behavior of shadow

Chapter 4 Implementation

For scalability issue, we use an individual BitBlt, which is to be introduced in subsection 4.4, to engage windows in different formats together. It is convenient for programmers to allocate their applications on a proper formatted window. By universal Window behavior, we could guarantee the operating efficiency to developers.

For portability issues, Video and Event in subsections 4.1 and 4.2 are abstract layers to take off Window in subsection 4.3 from OS coupling. Both Video and Event are as simple as possible to reduce the porting effort.

We specify the pixel format definition in subsection 4.1. That improves the adaption of our GUI library to diversified processors. The GNU tool chain helps us reserving our existing source codes and make files. For other factors like hardware, we develop a 2D accelerator enabled bit block transfer function to support hardware BitBlt.

4.1 Video

The registers and address wires are all 32 bits wide in a 32-bit host machine, but we can not make sure where the colors of red, green, and blue are located. We even have no idea about the color depth of each pixel by accessing the pixel buffer only. We need a type which could cover the depth of colors and the locations of each color of red, green, and blue. Figure 4.1 includes the major definition of Video. Fig. 4.7 is the methods of Video.

4.1.1 Properties of Video

```
typedef struct
{
        UINT32          Bpp;
        UINT8           Aloss;
        UINT8           Rloss;
        UINT8           Gloss;
        UINT8           Bloss;
        UINT8           Ashift;
        UINT8           Rshift;
        UINT8           Gshift;
        UINT8           Bshift;
} xFormat;
```

Fig. 4.1 The definition of structure xFormat

Bpp

It means ***bytes*** per pixel.

When we have a 24-bit pixel and each color has 8 bits length are shown in Figure 4.2.

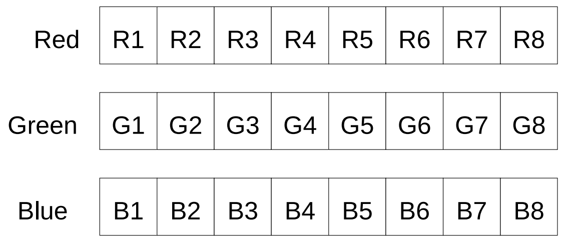


Fig. 4.2 A pixel with 24 bits

We want to convert it into another pixel format, like **RGB565** in Figure 4.3

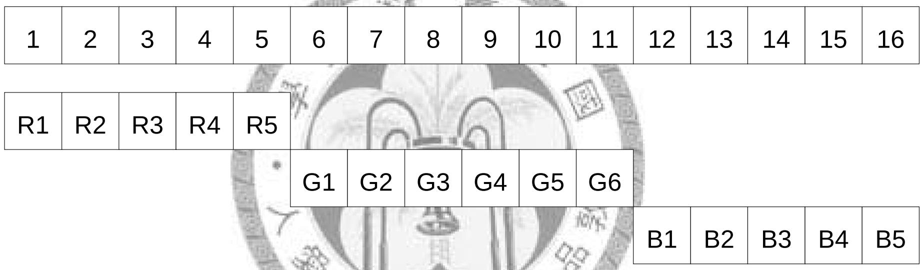


Fig. 4.3 The bits sequence of RGB565

Or **RGB555** in Figure 4.4

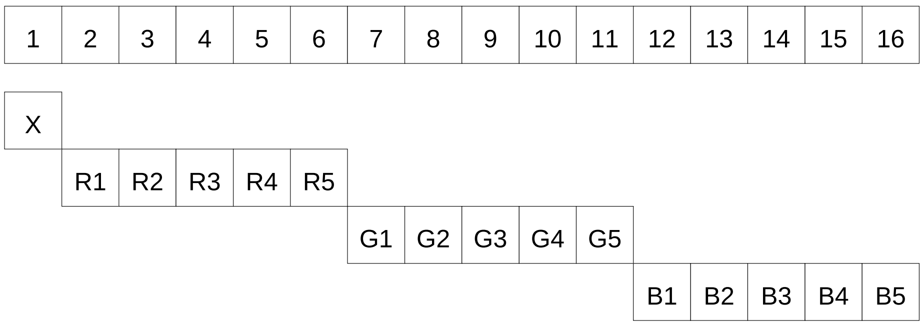


Fig. 4.4 The bits sequence of RGB555

Fig. 4.5 is the algorithm that could transform the incoming RGB color bits from one to another by referring to a specific format.

```
return   (r >> format->Rloss) << format->Rshift
     | (g >> format->Gloss) << format->Gshift
     | (b >> format->Bloss) << format->Bshift
     ;
```

Fig. 4.5 The statements of color conversion

And we could list an example for RGB565 and RGB555 in Fig. 4.6.

```
static xFormat
pool_format[2] = {
        { 2, 8, 3, 3, 3,  0, 10, 5, 0 },   // RGB555
        { 2, 8, 3, 2, 3,  0, 11, 5, 0 }    // RGB565
};
```

Fig. 4.6 The statements of format definition

By this color-bit definition, we can avoid problems of ***Word Order*** and ***Word Alignment***, mentioned in Chapter 2, on crossing different platforms.

Then for BitBlt functions, most conditions of platform porting are included in these use cases.

4.1.2 Methods of Video

```
        extern int      oem_video_init (UINT32 mode, xFormat *format);

        extern int      oem_video_size (xWindow *window,
                                        UINT32 width, UINT32  heigth);

        extern int      oem_video_free (xWindow *window);

        extern int      oem_video_draw (xWindow *window,
                                        INT32 x, INT32 y, UINT32  w, UINT32  h);
```

Fig. 4.7 The methods of Video

oem_video_init ()

It determines whether the wanted mode could be satisfied. If the mode is accepted, it would then fill the “format” structure.

oem_video_size ()

They are used to allocate and delete bitmap buffer for a window. This pair of functions could keep the flexibility of hardware acceleration by using contiguous physical address memory buffer.

oem_video_free ()

It is used to collect allocated resources for the indicated window.

oem_video_draw ()

It is used to transfer the pixel buffer in “Shadow” to video screen for double buffering architecture shown in Fig. 4.8. Sometimes we call it “synchronizing”.

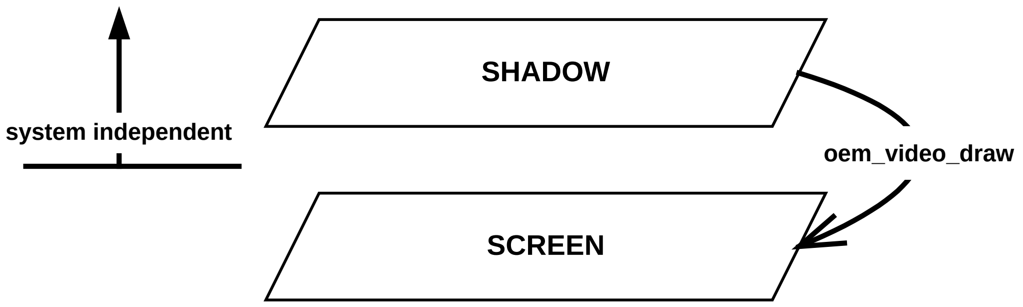


Fig. 4.8 The relationship between screen and shadow

4.2 Event

The basic definitions of Event are introduced in Fig. 4.9, Fig 4.10, and Fig. 4.11.

4.2.1 Global Variables

```
static void          (*dtk_main_proc) (xEvent *, void *) = NULL;
static void *        dtk_main_data = NULL;
```

Fig. 4.9 The global variables of Event

4.2.2 Properties of Event

```
typedef struct _xEvent
{
            UINT32              type;
            UINT32              code;
            void *              data;
} xEvent;
```

Fig. 4.10 The definition of structure xEvent

type

Each Input category has its own ID number to identify what is carried.

code

When it comes to a keypad or keyboard input, a 32-bit variable is usually large enough to carry the input value. But in some other types it could also be used to represent other property.

data

It is the context of an event addressed by a void pointer.

4.2.3 Methods of Event

```
extern int          oem_event_init (void);
extern int          oem_event_func (void *proc, void *data);
extern int          oem_event_loop (void);
```

Fig. 4.11 The methods of Event

oem_event_init ()

It will do some preparation for the event loop.

oem_event_func ()

It is used to assign the callback function and permanent referred data for event loop.

oem_event_loop ()

It is an end-less loop function while a GUI program is executing, unless we want to terminate this program. It is usually presented in a while loop like the following statements. What we see in Fig. 4.12 is the sample code.

```
while (1) {
        event = GetNextEvent ();

        if (dtk_main_proc != NULL)
                dtk_main_proc (event, dtk_main_data);
}
```

Fig. 4.12 The statements of our event loop

4.3 Window

Fig 4.13, and Fig. 4.20 shows the properties and methods of a window object.

4.3.1 Properties of Window

```
typedef struct _xWindow xWindow;
struct _xWindow
{
        xRect           area;
        UINT8 *         pixel;
        xFormat *       format;
        UINT32          keycolor;
        UINT16          alpha;
        UINT8           flags;
        UINT8           type;
        UINT8 *         name;
        Atom *          atom;
        xWindow *       prev;
        xWindow *       next;
        xBlit *         blit;
};
```

Fig. 4.13 The definition of structure xWindow

When a window is presented as a digital image like Fig. 4.14, the position of a window means the coordinate to locate this bitmap in the screen. And Fig. 4.15 is a structure of area definition.

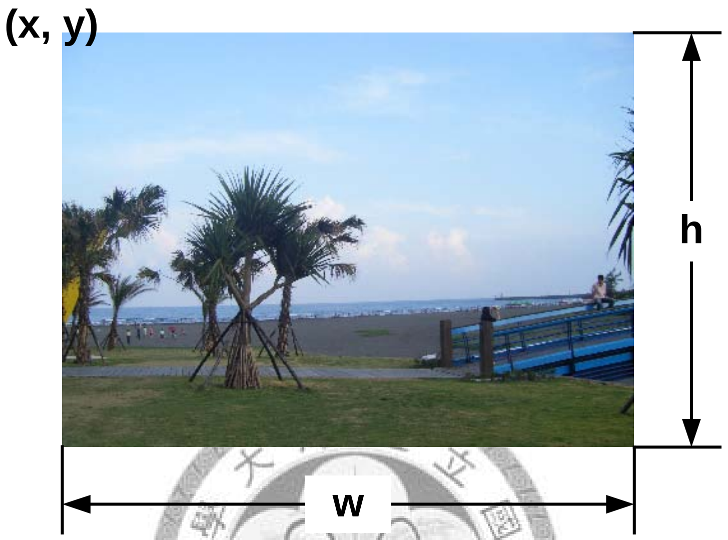


Fig. 4.14 A digital image

area

```
typedef struct _xRect
{
            INT32           x;
            INT32           y;
            UINT32          w;
            UINT32          h;
} xRect;
```

Fig. 4.15 The definition of structure xRect

The positive part of INT32 is an integer ranging from 0 to (2^31-1). Today it is still huge enough to satisfy the maximum resolution of cutting edge display device. Here we use two different types of integer variable, INT32 and UINT32, to represent position and size. Why should we declare them in this way?

The reason is that a window may appear in any corner of the physical display screen. Case A in Fig. 4.16 is the placement in normal style.

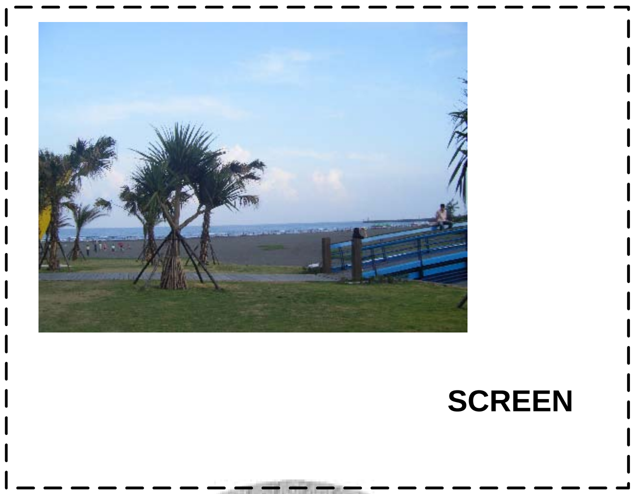


Fig. 4.16 The Case A

But Case B in Fig. 4.17, is a tough condition that the image is not fully placed in the screen.

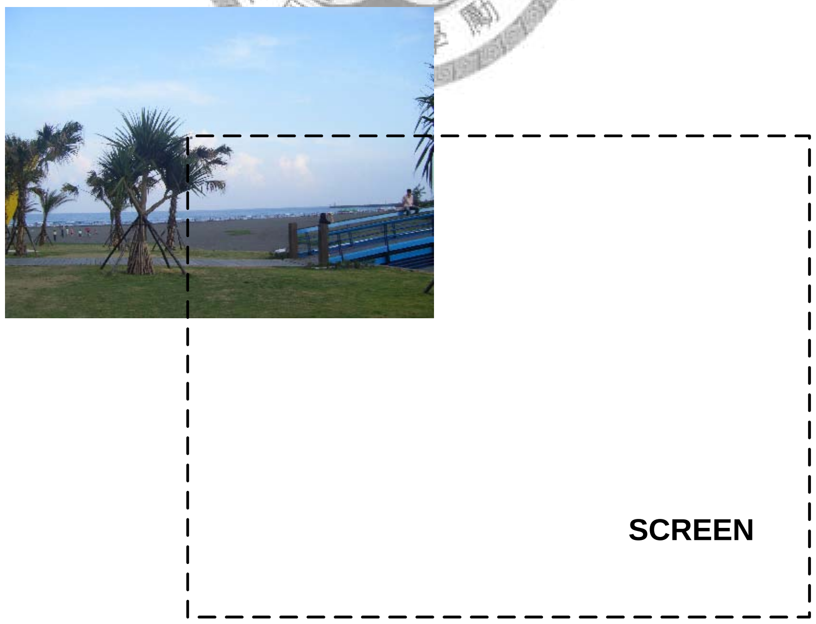


Fig. 4.17 The Case B

The left and top points of the image in case B should be both negative. For reserving this capability to describe this situation, we declare variables x, and y in type INT32, a signed integer for length of 32 bits.

keycolor

We might want our window to have a special shape other than a rectangle. It is implemented by using the reserved Transparent Color.

alpha

It is the Alpha channel for a window. A bitblt function would read this value to perform a weighted pixel blending.

flags

Other options of a window, like visible, enabling keycolor, or having alpha channel.

type

Other options of an existing window

name

We can assign a specific string as a name for each window.

atom

It is reserved for upper layer object model that enables event loop to deliver the message back to the atom object immediately.

blit

Please refer to the introduction to "BitBlt".

prev, next

It is a logical translation to represent the sequence of a window located on the screen in Fig. 4.18.

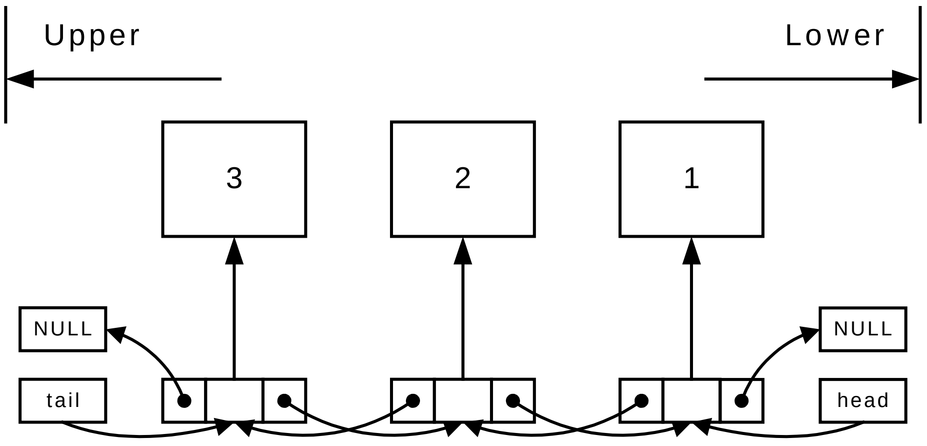


Fig. 4.18 Use double linked-list to remain sequence of windows

A sort of all windows in the screen could be treated as a linked-list from the lower side to upper side. Upper means more close to us and lower means the opposite direction. Here a double linked-list is convenient for insertion or removal of any node. In some special case, we can also merge shadow into the linked-list, too, just like Fig. 4.19.

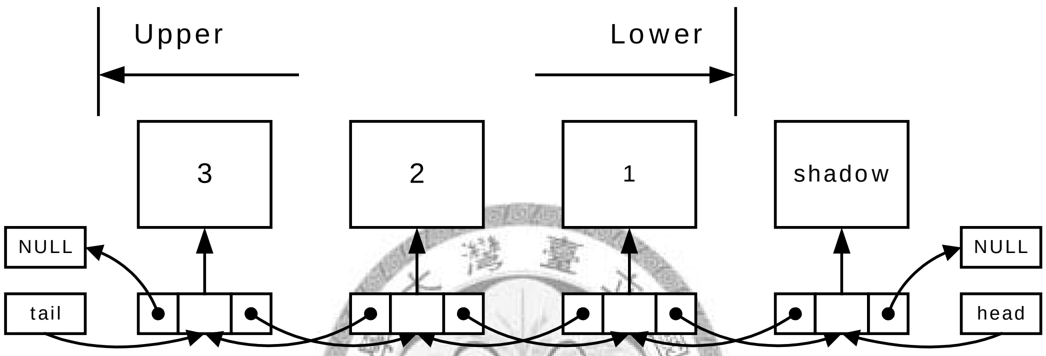


Fig. 4.19 The Shadow takes a place in the sort of windows

4.3.2 Methods of Window

```
extern int         ddk_window_init (UINT32 mode);
extern xWindow*    ddk_window_new (UINT32 type);
extern int         ddk_window_size (xWindow *window, UINT32 w, UINT32 h);
extern int         ddk_window_free (xWindow *window);
extern int         ddk_window_show (xWindow *window);
extern int         ddk_window_hide (xWindow *window);
extern int         ddk_window_move (xWindow *window, INT32 x, INT32 y);
extern int         ddk_window_draw (xWindow *window,
                              INT32 x, INT32 y, UINT32 width, UINT32 height);
```

Fig. 4.20 The methods of Window

ddk_window_init ()

It initializes the “screen” and “shadow”

ddk_window_new ()

It allocates the memory space for window structure and also determines its bliter.

ddk_window_free ()

This function will free all relative memory space indicated by this pointer.

ddk_window_size (),

It alters property and allocates pixel buffer for this window.

ddk_window_draw ()

It is used to draw pixels of a determined area back to the screen.

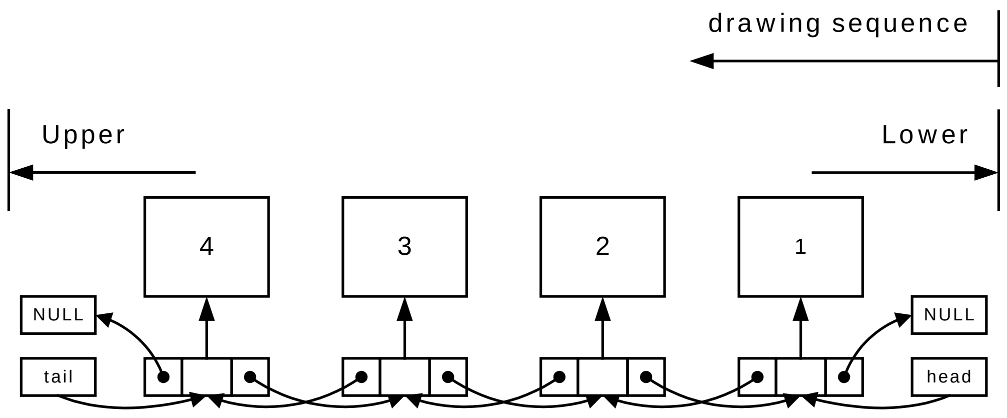


Fig. 4.21 a sort of windows

It is so simple to transfer all windows into screen by using a while loop for a sort of windows like Fig. 4.21. Fig. 4.22 is the corresponding C language statements.

```
w = head;
while (w) {
        ddk_blit (shadow, w, w->area.x, w->area.y, 0, 0, w->area.w, w->area.h);
        w = w->next;
}
```

Fig. 4.22 statements of simple bitblt

But is that **CORRECT**? No, at least two points will challenge this algorithm.

1. It may cause some unpredictable errors if the drawing area exceeds the screen boundary.
2. It wastes too many system computing powers, even if we want to draw a tiny area only.

So we are looking for a solution for solving these problems. It requires intersection calculation for filtering the overlapping area. First, we have to make sure that the drawing area is not out of the shadow/screen boundary.

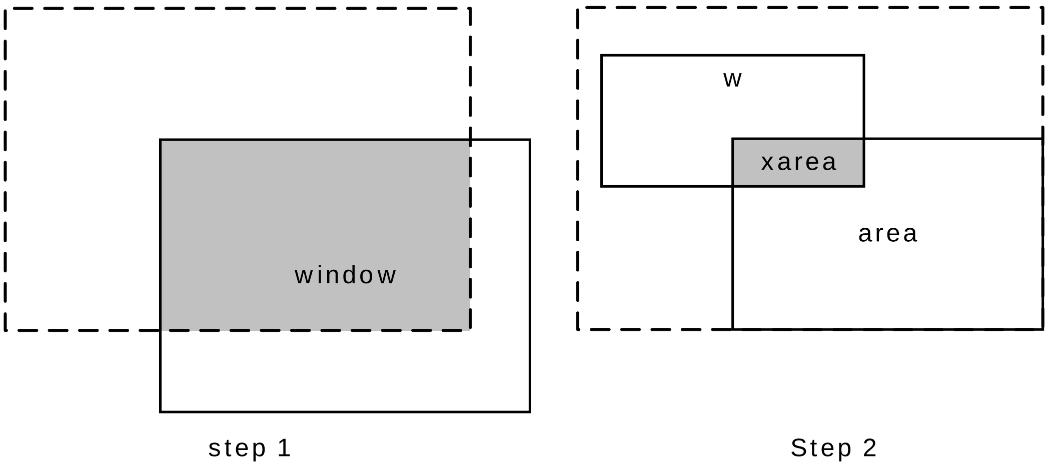


Fig. 4.23 two steps of real window bitblt

Fig. 4.23 shows the selecting steps for an intersection.

Step 1: Extract the intersection of window and screen by statements in Fig 4.24.

```
xRect     area;

ret = ddk_rect_intersect (&shadow->area, &window->area, &area);
```

Fig. 4.24 statements of scope select

Step 2: Use this inside area to calculate overlapping with other windows by statements in Fig.4.25.

```
w = head;
while (w) {
        ret = ddk_rect_intersect (&w->area, &area, &xarea);

        if (ret == TRUE)
                ddk_blit (shadow, w,
                        xarea.x, xarea.y,
                        xarea.x - w->area.x, xarea.y - w->area.y,
                        xarea.w, xarea.h);
        w = w->next;
}
```

Fig. 4.25 statements of bitblt iteration loop

ddk_window_show ()

It is the original case of screen linked list in Fig. 4.26.

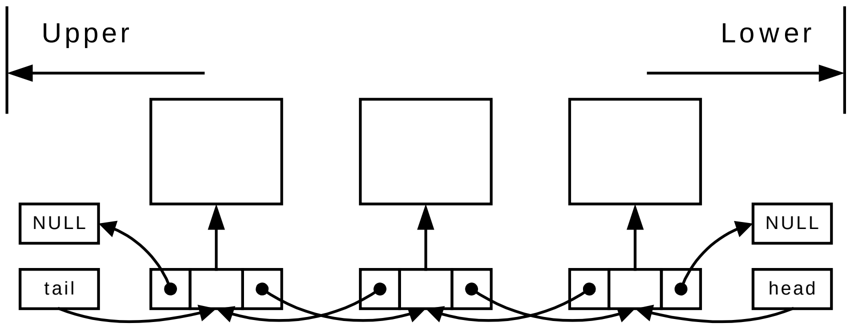


Fig. 4.26 a sort of windows

Append the window into the screen linked list as shown in Fig. 4.27.

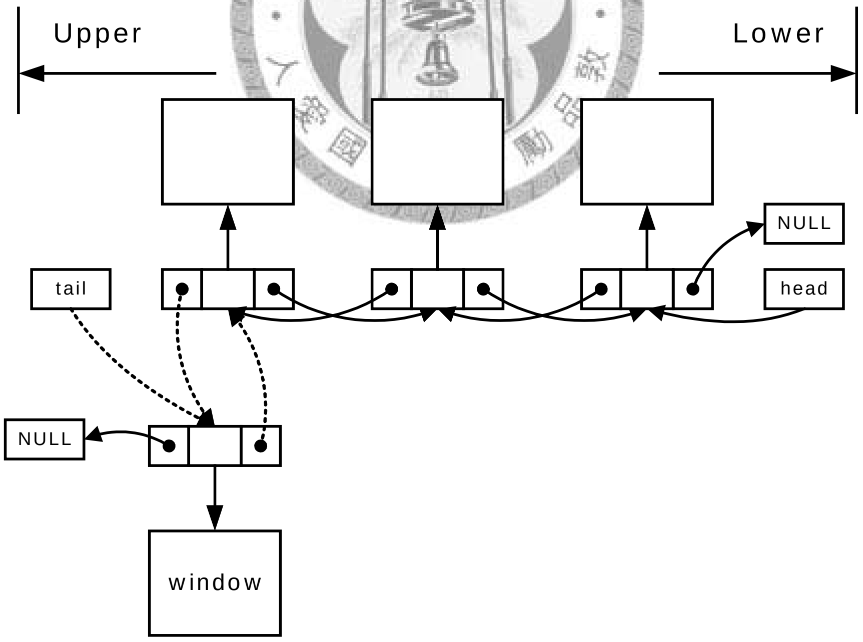


Fig. 4.27 insert a window

Then redraw the area of this window.

ddk_window_hide ()

It is the original case of screen linked list in Fig. 4.28.

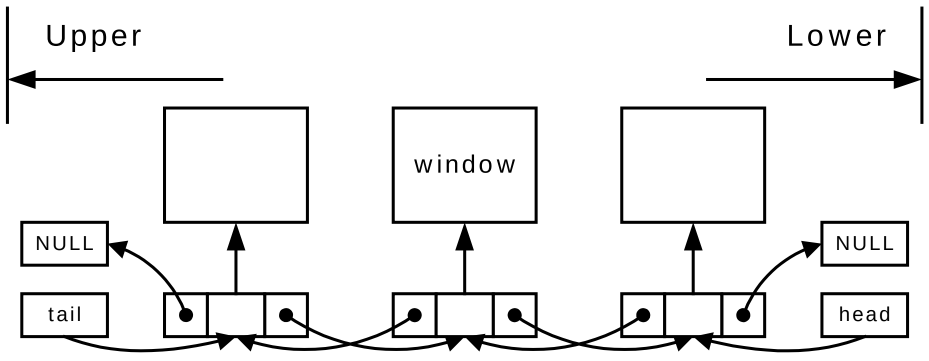


Fig. 4.28 a sort of windows

Remove the window from the screen linked list shown as Fig. 4.29.

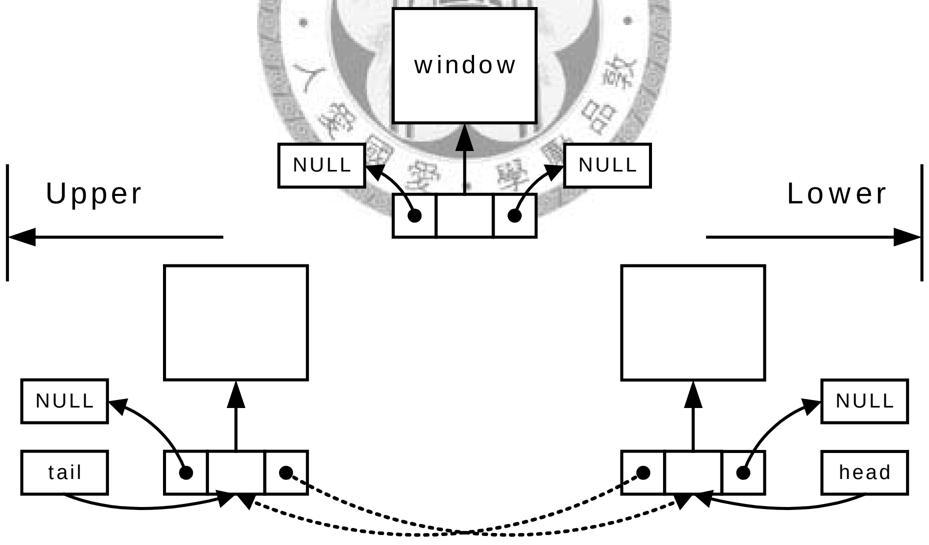


Fig. 4.29 remove a window

Then redraw the area of this window.

ddk_window_move ()

It is an atomic operation of two steps. Fig.4.30 is the original status of windows.

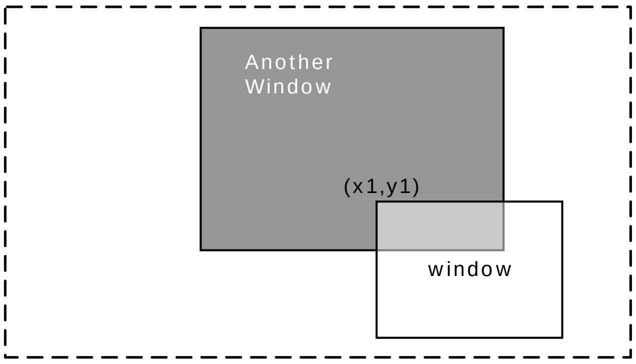


Fig. 4.30 the original status before move

1. Flush original area. See Fig. 4.31.

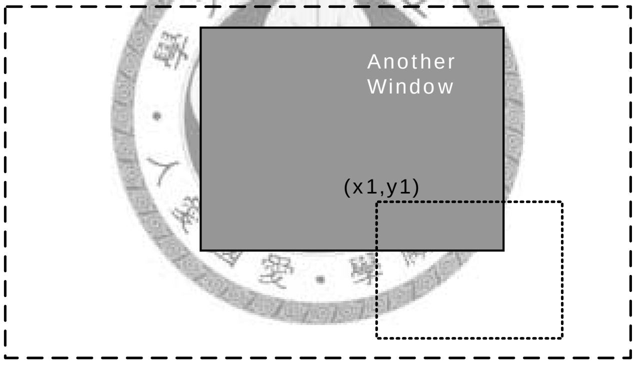


Fig. 4.31 step 1 of move: hide

2. Redraw the window in new position. See Fig. 4.32.

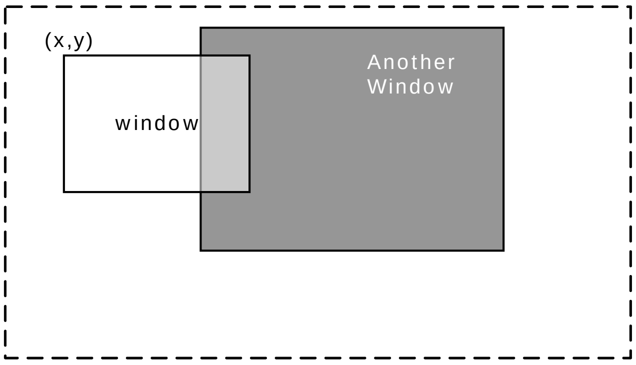


Fig. 4.32 step 2 of move: show

4.4 BitBlt

We list the properties and methods by Fig. 4.33 and Fig. 4.35. Fig.4.34 is a bliter example for RGB565. The bliter means a BitBlt executor.

4.4.1 Properties of BitBlt

```
typedef struct {
        UINT32   flag;
        UINT32   src_mode;
        UINT32   dst_mode;
        void     (*func)(xWindow *dst, xWindow *src,
                 UINT16 dx, UINT16 dy, UINT16 sx, UINT16 sy,
                 UINT16 width, UINT16 height);
} xBlit;
```

Fig. 4.33 The definition of structure xBlit

flag

It is a combination of “keycolor” and “alpha” option.

src_mode, dst_mode

It is the format definition of source and destination window pixel buffer.

func

It is the callback function we register in.

```
static xBlit
pool_blit16[] = {
        { 00, 565, 565, Blit16bpp },
        { 01, 565, 565, Blit565to565Alpha },
        { 10, 565, 565, Blit16bppKeycolor },
        { 11, 565, 565, Blit565to565AlphaKeycolor },
        { 0 }
};
```

Fig. 4.34 The methods of BitBlt

4.4.2 Methods of BitBlt

```
extern void         ddk_blit (xWindow *dst, xWindow *src, INT16 dx, INT16 dy,
                             INT16 sx, INT16 sy, UINT16 width, UINT16 height);
```

Fig. 4.35 The method BitBlt

dst

It is the destination window. Normally, it is the “shadow” window.

src

It is the source window.

dx, dy, sx, sy

It is the left-top point of the destination and source window.

width,height

The range of buffers we want to transfer into shadow buffer.

The related language C statements should be like those in Fig.4.36.

```
xBlit      *bliter = src->blit;
if (bliter) {
           bliter->func (dst, src, dx, dy, sx, sy, width, height);
}
```

Fig. 4.36 The usage example for BitBlt

We determine the bliter when a window is created as shown in Fig. 4.37.

```
src = window;
dst = shadow;

src_mode = (8 - src->format->Aloss)*1000 +
           (8 - src->format->Rloss)*100 +
           (8 - src->format->Gloss)*10 +
           (8 - src->format->Bloss);

dst_mode = (8 - dst->format->Aloss)*1000 +
           (8 - dst->format->Rloss)*100 +
           (8 - dst->format->Gloss)*10 +
           (8 - dst->format->Bloss);

flag  =    src->flags;

for (i = 0; pool_blit16[i].src_mode; i++) {
        if (flag == pool_blit16[i].flag &&
                src_mode == pool_blit16[i].src_mode &&
                dst_mode == pool_blit16[i].dst_mode)
        {
                src->blit = &pool_blit16[i];
                return TRUE;
        }
}

return FALSE;
```

Fig. 4.37 The statements for looking up a proper bliter

Then we can give an example for a 16-bits bliting like Fig. 4.38.

```
srcp = &((UINT16*)src->pixel)[sx + src->area.w*sy];
dstp = &((UINT16*)dst->pixel)[dx + dst->area.w*dy];

sskip = src->area.w - width;
dskip = dst->area.w - width;

for (j = 0; j < height; j++) {
        for (i = 0; i < width; i++) {
                UINT32 s = *srcp++;

                *dstp = s;
                dstp++;
        }

        srcp += sskip;
        dstp += dskip;
}
```

Fig. 4.38 An example for 16 bits bliting

Chapter 5 Experiments and Discussions

5.1 Porting to Windows XP

For porting to a new target platform, we have to perform an investigation first. So we want to know how dose a Microsoft Windows GUI program work. After referring to the open documents on internet, we know that a windows program is based on WIN32 API and use an event loop for fetching users' input commands.

The major job for porting is to make an integration of Event, and Video which we mentioned in Chapter 3.

5.1.1 Global variables

In Fig. 5.1, we define a global variable "***pool_format***" for the supported pixel formats. In Windows XP, the system supports RGB555 only for the 16-bit color mode.

```
static xFormat
pool_format[4] = {
        { 2, 8, 3, 3, 3,  0, 10, 5, 0 }   // RGB555
        { 2, 8, 3, 2, 3,  0, 11, 5, 0 },  // RGB565
        { 3, 8, 0, 0, 0,  0, 16, 8, 0 },  // RGB888
        { 4, 0, 0, 0, 0, 24, 16, 8, 0 }   // ARGB
};
```

Fig. 5.1 An example of pixel format declaration

5.1.2 Video Methods

From Fig. 5.2 to Fig. 5.5 are all C language statements for WIN32 porting.

oem_video_init ( )

It takes two major missions.

1. Determines whether the mode asked is supported

2. Duplicate the content of the format which was chosen.

oem_video_size ( )

```
window->pixel = (UINT8*) malloc (width * heigth * root_format->Bpp);
```

Fig. 5.2 To allocate the pixel buffers for a type of WINDOW

```
hdc = GetDC (hwnd_screen);
hbmp_screen = CreateDIBSection (hdc,
            binfo, DIB_RGB_COLORS, (void**) &window->pixel, NULL, 0);
ReleaseDC (hwnd_screen, hdc);
```

Fig. 5.3 To realize a native GDI window for a type of SCREEN

oem_video_free ( )

```
free (window->pixel);
```

Fig. 5.4 Statemet to free the pixel buffers

oem_video_draw ( )

To synchronize the pixels held in Shadow into Screen.

```
BitBlt (hdc, x, y, w, h, mdc, x, y, SRCCOPY);
```

Fig. 5.5 The WIN32 API "*BitBlt*"

5.1.3 Event Methods

oem_event_init ( )

In this function, we should assign a callback function for WIN32 message loop at the stage "***DispatchMessage*". ** Fig. 5.6 is the declaration of callback function.

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

Fig. 5.6 The declaration of WIN32 callback

Fig. 5.7 shows how this callback function is registered.

```
wcex.lpfnWndProc = (WNDPROC) WndProc;
RegisterClass (&wcex);
```

Fig. 5.7 To register the callback function "*WndProc*"

We have to call a WIN32API "***CreateWindow***", shown in Fig. 5.8, to get a window handler and copy its value to a global variable named "***hwnd_screen***".

```
hwnd_screen = CreateWindow (szWindowClass,
        szWindowClass,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0,
        CW_USEDEFAULT, 0, NULL, NULL, hInst, NULL);
```

Fig. 5.8 How to call the WIN32API "CreateWindow"

oem_event_func ( )

Fig. 5.9 is the registry of callback function called each time on dispatching.

```
dtk_main_proc = proc;
dtk_main_data = data;
```

Fig. 5.9 The assignment of callback function

oem_event_loop ( )

Because WIN32 API has a special message loop shown in Fig. 5.10, the statement of "***DispatchMessage***" would call the function ***WndProc*** we registered. Fig. 5.11 is the statements we add in WndProc.

```
while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
};
```

Fig. 5.10 The message loop for WIN32 API

```
if (dtk_main_proc)
        (*dtk_main_proc) (&event, dtk_main_data);
```

Fig. 5.11 The callback function is in the bottom of "*WndProc*"

5.1.4 Integration

We choose a flash movie playback library "GPLFlash" as the upper layer to be integrated in. Because the library itself is not a portable design, it is for UNIX and X Window only. So it took some time to understand the dependencies of GPLFlash.

First, drop the parts that couples with Xlib in Makefile. See Fig. 5.12.

```
SUBDIRS = lib #player plugin
```

Fig. 5.12 To comment the unused parts

Then we can build the library into an archive of object codes.

Second, link the archive into our program. We need a faked function at the linking stage. See Fig. 5.13.

```
int
gettimeofday (struct timeval *tp, void *tzp)
{
        DWORD tv = 0;

        tv = timeGetTime ();
        tp->tv_sec = tv / 1000;
        tp->tv_usec = tv % 1000;

        return 0;
}
```

Fig. 5.13 The faked function for GPLFlash

5.2 Porting to Linux Frame Buffer

The Linux Frame Buffer is only a memory segment mapping to raster display without supported GDI.

5.2.1 Global variables

We need global variables listed in Fig. 5.14 to keep the attributes of Frame Buffer.

```
static xWindow      screen;
static xFormat      screen_format;
static char *       fbdev = "/dev/fb0";
```

Fig. 5.14 Global Variables for Linux porting

5.2.2 Video Methods

oem_video_init ( )

Linux Frame buffer is implemented as a character device. We have to open it and go through system calls for extracting memory mapping to Frame Buffer.

Step1. Open the device file of Linux Frame Buffer (see Fig. 5.15).

```
if ((fd = open (fbdev, O_RDWR, 0)) == -1) {
        fprintf(stderr, "open: %s failed, errno=%d-\n", fbdev, errno);
        return NULL;
}
```

Fig. 5.15 Open Frame Buffer device file

Step 2. Use ioctl to get the parameters of Frame Buffer driver (see Fig. 5.16).

```
struct      fb_fix_screeninfo fix;
struct      fb_var_screeninfo var;

if (ioctl (fd, FBIOGET_FSCREENINFO, (void*)(&fix)) == -1) {
            fprintf (stderr, "ioctl: FBIOGET_FSCREENINFO, errno=%d-\n", errno);
            close (fd);
            return NULL;
}

if (ioctl (fd, FBIOGET_VSCREENINFO, (void*)(&var)) == -1) {
            fprintf (stderr, "ioctl: FBIOGET_VSCREENINFO, errno=%d-\n", errno);
            close (fd);
            return NULL;
}
```

Fig. 5.16 System calls to read variables

Assign those values to global variables. (see Fig.5.17)

```
screen_format.Bpp = var.bits_per_pixel / 8;

screen_format.Rloss = 8 - var.red.length;
screen_format.Gloss = 8 - var.green.length;
screen_format.Bloss = 8 - var.blue.length;

screen_format.Rshift = var.red.offset;
screen_format.Gshift = var.green.offset;
screen_format.Bshift = var.blue.offset;

screen.area.x = 0;
screen.area.y = 0;
screen.area.w = var.xres;
screen.area.h = var.yres;

screen.pixel = (UINT8*) m;

fbuf_sz = screen.area.w * screen.area.h * screen_format.Bpp;

memset (m, 0x00, fbuf_sz);
```

Fig. 5.17 Copy Frame Buffer variables

Step 3. Attach kernel space into user land. (see Fig. 5.18).

```
void        *m = NULL;

m = mmap (NULL, fix.smem_len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Fig. 5.18 Use system call "mmap"

Step 4. Close the device file. (see Fig. 5.19).

```
close (fd);
```

Fig. 5.19 Close a file descriptor

oem_video_size ( )

Fig.5.20 and Fig5.21 are couples for memory management.

```
window->pixel = (UINT8*) malloc (width * heigth * root_format->Bpp);
```

Fig. 5.20 To allocate pixel buffer

oem_video_free ( )

```
free (window->pixel);
```

Fig. 5.21 To release pixel buffer

oem_video_draw ( )

Because Frame Buffer is a pixel buffer same as window, we could use common bliter to handle it like Fig.5.22.

```
ddk_blit (&screen, shadow, x, y, x, y, w, h);
```

Fig. 5.22 The BitBlt function

5.2.3 Event Methods

oem_event_init ( )

For multiple input devices, we define a local structure "***Handle***".(see Fig.5.23)

```
typedef struct _Handle Handle {
        int         fd;
        char *      name;
        int         (*func) (int, void *, xEvent *);
        void *      data;
};
```

Fig. 5.23 The structure "*Handle*"

We could open all devices in a loop of handle array shown in Fig.5.24.

```
static Handle *     pool_handle[MAX_HANDLE];

for (i = 0; pool_handle[i]; i++) {
        handle = pool_handle[i];

        handle->fd = open (handle->name, O_RDONLY);
}
```

Fig. 5.24 A loop for input device files opening

oem_event_func ( )

Same as WIN32 porting (see Fig.5.9)

oem_event_loop ( )

Here we use UNIX system call “***select***” for various input multiplexing.

```
static fd_set           r_copy;

while (1) {
        FD_ZERO (&r_copy);

        for (i = 0; pool_handle[i]; i++) {
                handle = pool_handle[i];

                if (handle->fd >= 0) {
                        FD_SET (handle->fd, &r_copy);
                }
        }

        n = select (maxfds + 1, &r_copy, NULL, NULL, NULL);

        if (n < 0) continue;

        for (i = 0; pool_handle[i]; i++) {
                handle = pool_handle[i];

                if (FD_ISSET (handle->fd, &r_copy)) {
                        handle->func (handle->fd, handle->data, &event);
                }
        }

        if (dtk_main_proc)
                (*dtk_main_proc) (&event, dtk_main_data);

}
```

Fig. 5.25 Event loop for Linux

5.3 Porting to Non-OS system (Bootloader)

The system we use is a S3C2410 based development board. The processor is an ARM9 architecture SoC. We will describe how to put our GUI library into the bootloader at this subsection.

5.3.1 Global variables

```
static xWindow      screen;
static xFormat      screen_format;
```

Fig. 5.26 Global Variables for Non-OS porting

5.3.2 Video Methods

oem_video_init ( )

Fig.5.27 is a set of macros for LCD Controller.

```
#define LCD_XRES                240
#define LCD_YRES                320
#define LCD_VBPD                ((2-1)&0xff)
#define LCD_VFPD                ((3-1)&0xff)
#define LCD_VSPW                ((2-1)&0x3f)
#define LCD_HBPD                ((40-1)&0x7f)
#define LCD_HFPD                ((20-1)&0xff)
#define LCD_HSPW                ((20-1)&0xff)
#define LCD_CLKVAL_TFT          (7)
#define MVAL                    (13)
#define MVAL_USED               (0)
#define M5D(n)                  ((n) & 0x1fffff)
```

Fig. 5.27 The macros of LCD Controller

First we have to fill the registers of LCD controller to assign the Frame Buffer start address ***SCREEN_DMA***, and resolution of LCD panel. See Fig.5.28.

```
rLCDCON1 = (LCD_CLKVAL_TFT<<8)|(MVAL_USED<<7)|(3<<5)|(12<<1)|0;
rLCDCON2 = (LCD_VBPD<<24)|((var->yres-1)<<14)|(LCD_VFPD<<6)|(LCD_VSPW);
rLCDCON3 = (LCD_HBPD<<19)|((var->xres-1)<<8)|(LCD_HFPD);
rLCDCON4 = (MVAL<<8)|(LCD_HSPW);
rLCDCON5 = (1<<11)|(1<<9)|(1<<8)|(1<<3)|1;

rLCDSADDR1 = (((unsigned long)SCREEN_DMA>>22)<<21)|
                    M5D((unsigned long)SCREEN_DMA>>1);

rLCDSADDR2 = M5D(((unsigned long)SCREEN_DMA+(var->xres*var->yres*2))>>1);
rLCDSADDR3 = (((var->xres - var->xres)/1)<<11)|(var->xres/1);
```

Fig. 5.28 Initialize the LCD Controller

Then initialize the pixel format of the screen. See Fig.5.29.

```
screen_format.Bpp = 2;

screen_format.Rloss = 3;
screen_format.Gloss = 2;
screen_format.Bloss = 3;

screen_format.Rshift = 11;
screen_format.Gshift = 5;
screen_format.Bshift = 0;

screen.area.x = 0;
screen.area.y = 0;
screen.area.w = 240;
screen.area.h = 320;

screen.pixel = (void*) SCREEN_DMA;
```

Fig. 5.29 To Initialize Global Variables

The display panel we choose is a 3.5” TFT LCD module. According to the document, we know the resolution of this panel is 240 x 320.

oem_video_size ( )

```
        window->pixel = (UINT8*) malloc (width * heigth * root_format->Bpp);
```

Fig. 5.30 To allocate pixel buffer

Because the dynamic memory allocation is already implemented by common library of bootloader, we could use it directly by the same function name as “malloc”. See Fig. 5.30.

oem_video_free ( )

```
        free (window->pixel);
```

Fig. 5.31 To release pixel buffer

We could call “free” to release the memory allocated. See Fig. 5.31.

oem_video_draw ( )

```
        ddk_blit (&screen, shadow, x, y, x, y, w, h);
```

Fig. 5.32 The BitBlt function

The drawing function is a simple action of memory copy. We could transfer memory block directly. See Fig. 5.32.

5.3.3 Event Methodss

oem_event_init ( )

The touch screen needs A/D convertor. Fig. 5.33 shows the registers of ADC.

```
rADCCON = (1<<14)|(7<<3)|(0<<2)|(0<<1)|(0);
rADCTSC = (0<<8)|(1<<7)|(1<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(3);
```

Fig. 5.33 To Initialize A/D Convertor

oem_event_func ( )

Same as WIN32 porting (see Fig.5.9)

oem_event_loop ( )

Fig.5.34 is the GUI event loop.

```
int        pen, key;
xEvent     event_pen, event_key;

while (1) {
           do {
                      pen = PEN_input (&event_pen);
                      key = KEY_input (&event_key);

                      Delay (100);
           } while (!pen || !key);

           if (dtk_main_proc) {
                      if (pen)
                                 (*dtk_main_proc) (&event_pen, dtk_main_data);
                      if (key)
                                 (*dtk_main_proc) (&event_key, dtk_main_data);
           }
}
```

Fig. 5.34 Firmware style Event loop

5.4 Comparison

The GUI library we present uses modules of Video and Event to isolate the core functions of Window from OS dependency. The upper layers like GPLFlash could easily render in the pixel buffer of Window. Following is a brief description of the advantages of our library.

1. It has no dependency on other libraries, and is convenient for porting to other system.
2. Unlike other solutions that need extra pixmap in the midway for graphical presentation. In our library, a window already includes a pixel buffer for rendering.

Chapter 6 Conclusions

A GUI library is designed for lifting up the limitations of OS dependency. But it also exposes some problems. The major one is that the shadow object may take lots of memory resources for complete screen mapping by static memory allocation. The memory space and CPU time are always a trade off. Maybe we can try dynamic allocation on the memory copy stage only.

References

[1] H. Hartson and D. Hix, "Human-computer interface development: concepts and systems for its management," *ACM Computing Surveys (CSUR),* vol. 21, pp. 5-92, 1989.

[2] R. Baecker, "Digital video display systems and dynamic graphics," in *International Conference on Computer Graphics and Interactive Techniques,* 1979, pp. 48-56.

[3] F. Crow and M. Howard, "A frame buffer system with enhanced functionality," *ACM SIGGRAPH Computer Graphics,* vol. 15, pp. 63-69, 1981.

[4] B. Boehm, J. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *International Conference on Software Engineering,* 1976, pp. 592-605.

[5] A. Alvaro, E. Almeida, and S. Meira, "Quality Attributes for a Component Quality Model," in *International Workshop on Component-Oriented Programming (WCOP),* 2005.

[6] A. Wasserman, "User Software Engineering and the design of interactive systems," in *International Conference on Software Engineering,*

1981, pp. 387-393.

[7] M. Hakuta and M. Ohminami, "A study of software portability evaluation," *The Journal of Systems & Software*, vol. 38, pp. 145-154, 1997.

[8] C. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, pp. 131-183, 1992.

[9] J. Mooney, "A course in software portability," in *Technical Symposium on Computer Science Education*, 1992, pp. 53-56.

[10] A. Tanenbaum, P. Klint, and W. Bohm, "Guidelines for software portability," *Software: Practice and Experience*, vol. 8, 1978.

[11] E. Lecolinet, "A molecular architecture for creating advanced GUIs," 2003, pp. 135-144.

[12] J. Bishop and N. Horspool, "Developing principles of GUI programming using views," *ACM SIGCSE Bulletin*, vol. 36, pp. 373-377, 2004.

[13] J. Bishop, "Multi-platform user interface construction: a challenge for software engineering-in-the-small," 2006, pp. 751-760.

[14] J. Acquah, J. Foley, J. Sibert, and P. Wenner, "A conceptual model of raster graphics systems," *ACM SIGGRAPH Computer Graphics*, vol. 16, pp. 321-328, 1982.

[15] W. Newman, "Display procedures," *Communications of the ACM*, vol. 14, pp. 651-660, 1971.

[16] A. Kamran and M. Feldman, "Graphics programming independent of interaction techniques and styles," *ACM SIGGRAPH Computer Graphics*, vol. 17, pp. 58-66, 1983.

[17] S. Nakkrasae and P. Sophatsathit, "A formal approach for specification and classification of software components," in *SEKE*, 2002, pp. 773-780.

[18] J. Jeng and B. Cheng, "Specification matching for software reuse: a foundation," *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 97-105, 1995.

[19] J. Foley, *Computer graphics: principles and practice*: Addison-Wesley Professional, 1995.

[20] J. Thomas and G. Hamlin, "Graphical input interaction technique (GIIT)," *ACM SIGGRAPH Computer Graphics*, vol. 17, pp. 5-30, 1983.