Computer Labs: Lab2
Video Card in Graphics Mode
2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

September 27, 2012

Lab2: Video Card in Graphics Mode

- Write a set of functions:

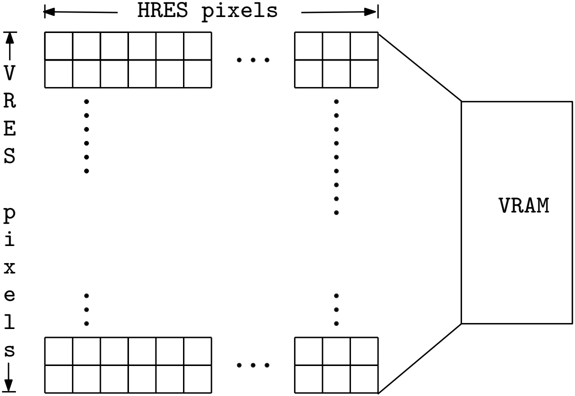
```
int vg_fill_screen(unsigned long color)
int vg_set_pixel(unsigned long x, unsigned long y,
                 unsigned long color)
int vg_get_pixel(unsigned long x, unsigned long y)
nt vg_draw_line(unsigned long xi, unsigned long yi,
                unsigned long xf, unsigned long yf,
                unsigned long color)
```

  to change the screen in graphics mode
- Like in Lab1, you output something to the screen by writing to VRAM
- Unlike in Lab1, you'll have to configure the graphics mode that you'll use:
  - Minix 3 boots in text mode, not in graphics mode

Contents

Video Card in Graphics Mode

- Like in text mode, the screen can be abstracted as a matrix
  - Now, a matrix of points, or **pixels**, instead of characters
    - With `HRES` pixels per line
    - With `VRES` pixels per column
  - For each pixel, the VRAM holds its color



How Are Colors Encoded? (1/2)

- Most electronic display devices use the RGB color model
  - A color is obtained by adding 3 primary colors – red, green, blue – each of which with its own intensity
  - This model is related to the physiology of the human eye
- One way to represent a color is to use a triple, with a given intensity per primary color
  - Depending on the number of bits used to represent the intensity of each primary color, we have a different number of colors
  - E.g., if we use 8 bits per primary color, we are able to represent $2^{24} = 16777216$ colors

How Are Colors Encoded? (2/2)

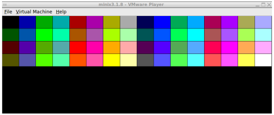
Direct-color mode Store the color of each pixel in the VRAM

- For 8 bits per primary color, if we use a resolution of 1024×768 we need a little bit more than 2 MB per screen

Indexed color Rather than store the color per pixel store an index into a table – the **palette/color map** – with the color definition, i.e. the intensity of the 3 primary colors.

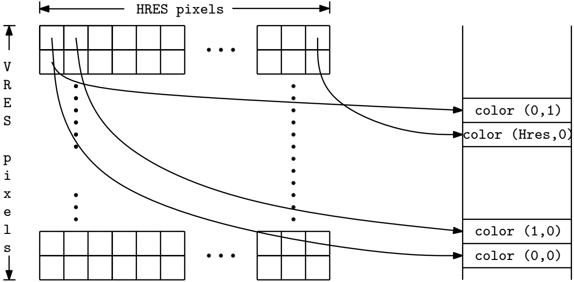
- With an 8 bit index we can represent 256 colors, each of which may have 8 bits or more per primary color
- By changing the **palette** it is possible to render more than 256 colors

- In the lab you'll use a palette with up to 256 colors, whose default initialization on VMware Player
  - Uses only the first 64 of the 256 elements
    - The first time it switches to the mode, the colors are not as bright – don't ask me why.



Memory Models

- The memory model determines the way the value of each pixel is stored in VRAM
  - Different graphics modes use different memory models
- The simplest mode, and the one that will be used in the lab, is the linear mode:



All we need to know is:

  - The base address of the frame buffer
  - The coordinates of the pixel
  - The number of bytes used to encode the color

Video Card Configuration

Problem (s)

1. How do you know the base address of the frame buffer?
2. How do you configure the desired graphics mode?

NO Solution Read/write directly the GPU registers

- GPU manufacturers usually do not provide the details necessary for that level of programming

Solution Use the VESA Video Bios Extension (VBE)

Contents

PC BIOS

- Basic Input-Output System is:
  - A firmware interface for accessing PC HW resources
  - The implementation of this interface
  - The non-volatile memory (ROM, more recently flash-RAM) containing that implementation
- It is used mostly when a PC starts up
  - It is 16-bits: even IA-32 processors start in real-mode
  - It is used essentially to load the OS (or part of it)
  - Once the OS is loaded, it usually uses its own code to access the HW not the BIOS

BIOS Calls

- Access to BIOS services is via the SW interrupt instruction `INT xx`
  - The `xx` is 8 bit and specifies the service.
  - Any arguments required are passed via the processor registers
- Standard BIOS services:

Interrupt vector (`xx`)	Service
`10h`	video card
`11h`	PC configuration
`12h`	memory configuration
`16h`	keyboard

BIOS Call: Example

- Set Video Mode: `INT 10h`, function `00h`

```
; set video mode

MOV AH, 0        ; function
MOV AL, 3        ; text, 25 lines X 80 columns, 16 colors
INT 10h
```

BIOS Call: From Minix 3

Problem

- The previous example is in real address mode
- Minix 3 uses protected mode with 32-bit

Solution

- Use Minix 3 kernel call `SYS_INT86`

  "Make a real-mode BIOS on behalf of a user-space device driver. This temporarily switches from 32-bit protected mode to 16-bit real-mode to access the BIOS calls."

BIOS Call in Minix 3: Example

```
#include <machine/int86.h>
int vg_exit() {
  struct reg86u reg86;

  reg86.u.b.intno = 0x10;
  reg86.u.b.ah = 0x00;
  reg86.u.b.al = 0x03;

  if( sys_int86(&reg86) != OK ) {
    printf("vg_exit(): sys_int86() failed \n");
    return 1;
  }
  return 0;
}
```

- `struct reg86u` is a struct with a union of structs
  `b` is the member to access 8-bit registers
  `w` is the member to access 16-bit registers
  `l` is the member to access 32-bit registers
- The names of the members of the structs are the standard names of IA-32 registers.

Video BIOS Extension (VBE)

- The BIOS specification supports only VGA graphics modes
  - VGA stands for Video Graphics Adapter
  - Specifies very low resolution: 640x480 @ 16 colors and 320x240 @ 256 colors
- The Video Electronics Standards Association (VESA) developed the Video BIOS Extension (VBE) standards in order to make programming with higher resolutions portable
- Early VBE versions specify only a real-mode interface
- Later versions added a protected-mode interface, but:
  - In version 2, only for some time-critical functions;
  - In version 3, supports more functions, but they are optional.

VBE `INT 0x10` Interface

- VBE still uses `INT 0x10`, but to distinguish it from basic video BIOS services
  - `AH = 4Fh` - BIOS uses `AH` for the function
  - `AL = function`
- VBE graphics mode 105h, 1024x768@256, **linear** mode:

```
struct reg86u r;
r.u.w.ax = 0x4F02; // VBE call, function 02 -- set VBE mode
r.u.w.bx = 1<<14|0x105; // set bit 14: linear framebuffer
r.u.b.intno = 0x10;
if( sys_int86(&r) != OK ) {
    printf("set_vbe_mode: sys_int86() failed \n");
    return 1;
}
```

You should use symbolic constants.

Accessing the Linear Frame Buffer

1. Obtain the physical memory address
   1.1 Using a hard-coded address (`0xD0000000`), first;
   1.2 Using Function `0x01` Return VBE Mode Information, once everything else has been completed.
2. Map the physical memory region into the process' address space

- Steps 2 was already described in the Lab 1 slides

Obtaining the Physical Memory Address with VBE (1/5)

- VBE Function 01h - Return VBE Mode Information:

  Input

AX	=	4F01h	Return VBE Mode Information
CX	=		Mode number
ES:DI	=		Pointer to ModeInfoBlock structure

  Ouput

AX	=	VBE return status

- The ModeInfoBlock includes among other information:
  1. The mode attributes, which comprise a set of bits that describe some general characteristics of the mode, including whether:
     - it is supported by the adapter
     - the linear frame buffer is available
  2. The screen resolution of the mode
  3. The physical address of the linear frame buffer

Obtaining the Physical Memory Address with VBE (2/5)

Problem

- The ModeInfoBlock structure must be accessible both in protected mode and in real mode
  - VBE Function 01h is a real mode function
  - Real mode addresses are only 20-bit long (must be in the lower 1MiB).

Solution

- Use the `liblm.a` library
  - Provides a simple interface for applications:

    ```
    lm_init()
    lm_alloc()
    lm_free()
    ```

  - Hides some non-documented functions provided by Minix 3
- The `mmap_t` (already used in Lab 1) includes both:
  - The physical address, for use by VBE
  - The virtual address, for use in Minix 3

Obtaining the Physical Memory Address with VBE (3/5)

```
int get_vbe_mode_info(unsigned short mode, phys_bytes buf) {
  struct reg86u r;

  r.u.w.ax = 0x4F01;          /* VBE get mode info */
  /* translate the buffer linear address to a far pointer */
  r.u.w.es = PB2BASE(buf);    /* set a segment base */
  r.u.w.di = PB2OFF(buf);     /* set the offset accordingly */
  r.u.w.cx = mode;
  r.u.b.intno = 0x10;
  if( sys_int86(&r) != OK ) { /* call BIOS  */
```

`PB2BASE` Is a macro for computing the base of a segment, a 16-bit value, given a 32-bit linear address;

`PB2OFF` Is a macro for computing the offset with respect to the base of a segment, a 16-bit value, given a 32-bit linear address;

Obtaining the Physical Memory Address with VBE (4/5)

Problem The parameters contained in the buffer returned by VBE function 0x01 are layed out sequentially, with no holes between them

- Simply defining a C struct with one member per parameter with an appropriate type, is not enough
- C compilers layout the members of a struct in order and place them in memory positions whose address is aligned according to their type

Solution Use GCC's `__attribute__((packed))`

- In principle, this should be handled by the `#pragma pack` directives, but it is not supported by this version of GCC

Note that this attribute must appear immediately after the `}`, otherwise it has no effect

- You need not do anything, as I've already defined the struct in `vbe.h`

Obtaining the Physical Memory Address with VBE (5/5)

```
#include <stdint.h>

typedef struct
{
   uint16_t ModeAttributes;
   [...]
   uint16_t XResolution;
   uint16_t YResolution;
   [...]
   uint8_t BitsPerPixel;
   [...]
   uint32_t  PhysBasePtr;
   [...]
} __attribute__((packed)) vbe_mode_info_t;
```