

# Computer Labs: Lab5

## Video Card in Graphics Mode

### 2º MIEIC

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

November 15, 2020

# Contents

## Graphics Adapter/Video Card

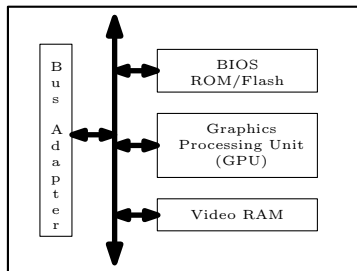
### Video Card in Graphics Mode

### Lab 5

### BIOS and VBE

### Accessing VRAM

# Graphics Adapter/Video Card



**GPU** Earlier known as the Graphics Controller:

- ▶ Controls the display hardware (CRT vs. LCD)
- ▶ Performs 2D and 3D rendering algorithms, offloading the CPU and accelerating graphics applications

**BIOS ROM/Flash** ROM/Flash Memory with firmware. Includes code that performs some standardized basic video I/O operations, such as the Video BIOS Extension (VBE)

**Video RAM** Stores the data that is rendered on the screen.

- ▶ It is accessible also by the CPU (at least part of it)

# Video Modes

## Text Mode

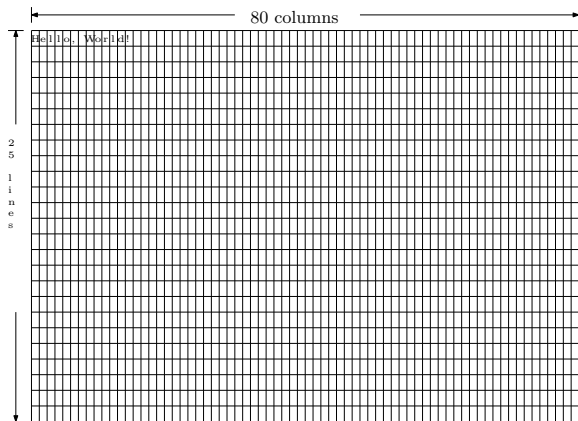
- ▶ Mode used by Minix 3 by default

## Graphics Mode

- ▶ Mode you will use in Lab 5

# PC's Graphics Adapter Text Modes

- ▶ Used to render mostly text
- ▶ Abstracts the screen as a matrix of characters (row x cols)
  - ▶ E.g. **25x80**, 25x40, 50x80, 25x132
  - ▶ Black and white vs color (16 colors)



# Contents

Graphics Adapter/Video Card

**Video Card in Graphics Mode**

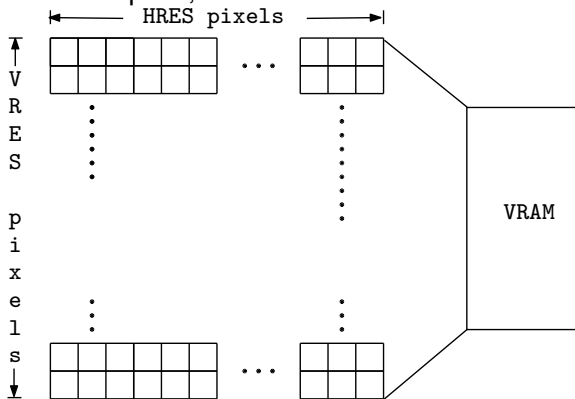
Lab 5

BIOS and VBE

Accessing VRAM

# Video Card in Graphics Mode

- ▶ The screen is abstracted as a matrix of points, or **pixels**
  - ▶ 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



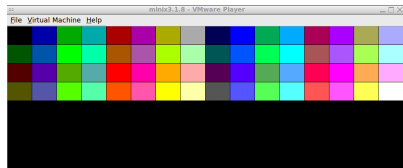
# How Are Colors Encoded? (2/2)

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

- ▶ For 8 bits per primary color, if we use a resolution of  $1024 \times 768$  we need 3 MB (assuming 4 bytes per pixel)

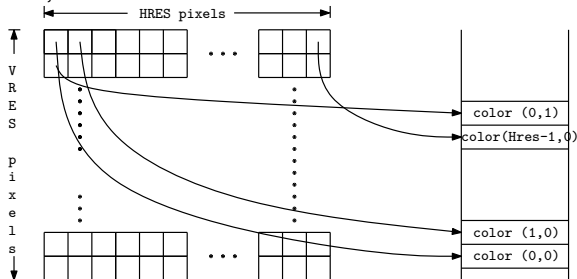
**Indexed color** Rather than storing the color per pixel, it stores an index into a table – the **palette/color map** – with the definition, i.e. the intensity of the 3 primary colors, of each color

- ▶ With an 8 bit index we can represent 256 colors, each of which may have 8 bits 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 both VMware Player and VirtualBox
  - ▶ 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 how video memory is organized, i.e., where 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 **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

# Contents

Graphics Adapter/Video Card

Video Card in Graphics Mode

## Lab 5

BIOS and VBE

Accessing VRAM

## Lab5: Video Card in Graphics Mode

- ▶ Write a set of functions:

```
void video_test_init(unsigned short mode,
                    unsigned short delay)
int  video_test_rectangle(uint16_t mode, uint16_t x,
                        uint16_t y, ...)
int  video_test_pattern(uint16_t mode, uint16_t no_rectangles,
                        uint32_t first)
```

to set the screen to graphics mode and to change the display in that mode

- ▶ These are only the functions to implement by the first class
- ▶ Essentially you have to:
  1. Configure the video card for the desired graphics mode
    - ▶ Minix 3 boots in text mode, not in graphics mode
  2. Write to VRAM to display on the screen what is requested
    - ▶ Map VRAM to the process' address space
  3. Reset the video card to the text mode used by Minix
    - ▶ You need only to call a function that we provide you

# Video Card Configuration (`video_test_init()`)

**Problem** 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

Graphics Adapter/Video Card

Video Card in Graphics Mode

Lab 5

**BIOS and VBE**

Accessing VRAM

# PC BIOS

- ▶ Basic Input-Output System is:
  1. A firmware interface for accessing PC HW resources
  2. The implementation of this interface
  3. 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
- ▶ Nowadays, most PCs use the "Unified Extensible Firmware Interface" (UEFI)

# 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

# How to make a BIOS Call in Minix 3.1.x?

## 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 call 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.”

# How to make a BIOS Call in Minix 3.4.x?

## Problem

- ▶ Kernel call `SYS_INT86` was dropped in Minix 3.2

## Solution

- ▶ Pedro Silva has added this kernel call to Minix 3.4.0rc6, by porting `libx86emu`, a small library that emulates some key x86 instructions.
  - ▶ Essentially, we can use `sys_int86` in Minix 3.4.x, as we did in Minix 3.1.8.
  - ▶ The implementation though is at user level.

# BIOS Call in Minix 3: Example

```
#include <machine/int86.h> // /usr/src/include/arch/i386
int vg_exit() {
    reg86_t reg86;

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

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

- ▶ `reg86_t` is a struct with a union of anonymous structs that allow access the IA32 registers as
  - ▶ 8-bit registers
  - ▶ 16-bit registers
  - ▶ 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.
- ▶ Unfortunately, neither VirtualBox nor VMwarePlayer support the protected mode interface

# 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:

```
reg86_t r;  
r.ax = 0x4F02; // VBE call, function 02 -- set VBE mode  
r.bx = 1<<14|0x105; // set bit 14: linear framebuffer  
r.intno = 0x10;  
if( sys_int86(&r) != OK ) {  
    printf("set_vbe_mode: sys_int86() failed \n");  
    return 1;  
}
```

**You should use symbolic constants.**

## video\_test\_rectangle()

- ▶ Draw a rectangle on the screen in the desired mode

```
int video_test_rectangle(uint16_t mode, uint16_t x, uint16_t y,  
                        uint16_t width, uint16_t height, uint32_t color)
```

- ▶ Unlike in previous years, we will test your code for different graphical modes, i.e. different:

**Resolution** both horizontal and vertical

**Bits per pixel** And color models

**Indexed color modes** also called packed-pixel by VBE,  
appear to have only 8 bits per pixel

**Direct color modes** May use a different number of bits per  
pixel

- ▶ And sometimes, the number of bits per component may be different, even if the number of bits per pixel is the same.
- ▶ These affect the offset, with respect to the frame-buffer base address, of the memory location with the color value of a pixel, or of one of its RGB components.
- ▶ The goal is that your code be parametric, so that it can easily handle these differences

# Contents

Graphics Adapter/Video Card

Video Card in Graphics Mode

Lab 5

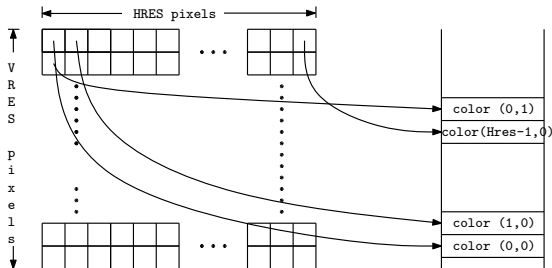
BIOS and VBE

Accessing VRAM



# Mapping the Linear Frame Buffer

- ▶ Before you can write to the frame buffer.

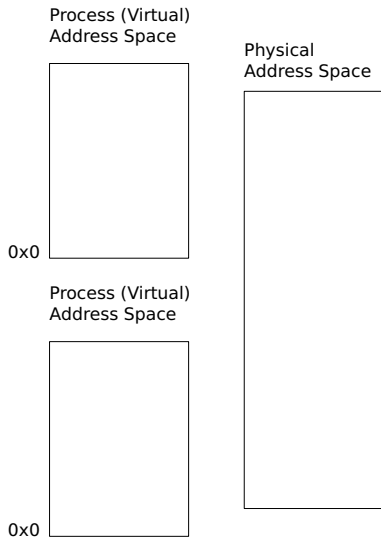


1. Obtain the physical memory address
  - 1.1 Use `vbe_get_mode_info()` that **we provide as part of the LCF**
    - ▶ This function retrieves information about the input VBE mode, including the physical address of the frame buffer
  - 1.2 Should provide your own implementation, using VBE function `0x01 Return VBE Mode Information`, once everything else has been completed.
2. Map the physical memory region into the process' (virtual) address space

# Virtual and Physical Address Spaces

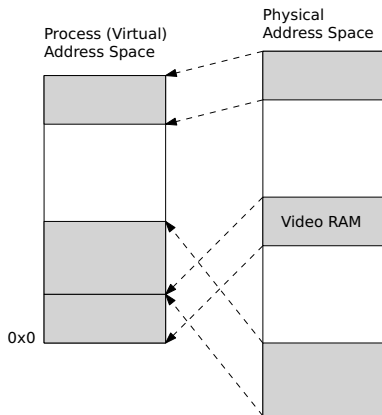
**Issue** Most computer architectures support a **virtual address space** that is decoupled from the **physical address space**

- ▶ Processes can access (physical) memory using a **logical** address that is independent of the physical address (determined by the address bus decoding circuitry)
- ▶ Most modern operating systems, including Minix, take advantage of this feature to simplify memory management.



# Mapping Physical Memory to Virtual Address Space

- ▶ Each process has its own virtual address space, whose size is usually determined by the processor architecture (32-bit for IA-32)
- ▶ The operating system maps regions of the physical memory in the computer to the virtual address spaces of the different processes
  - ▶ The details of how this is done are studied in the Operating Systems course.



# Mapping VRAM in Minix (1/2)

```
int r;
struct minix_mem_range mr; /* physical memory range */
unsigned int vram_base; /* VRAM's physical addresss */
unsigned int vram_size; /* VRAM's size, but you can use
                        the frame-buffer size, instead */
void *video_mem; /* frame-buffer VM address */

/* Allow memory mapping */

mr.mr_base = (phys_bytes) vram_base;
mr.mr_limit = mr.mr_base + vram_size;

if( OK != (r = sys_privctl(SELF, SYS_PRIV_ADD_MEM, &mr)))
    panic("sys_privctl (ADD_MEM) failed: %d\n", r);

/* Map memory */

video_mem = vm_map_phys(SELF, (void *)mr.mr_base, vram_size);

if(video_mem == MAP_FAILED)
    panic("couldn't map video memory");
```

# Mapping VRAM in Minix (2/2)

**Question** What is the following code about?

```
/* Allow memory mapping */  
  
mr.mr_base = (phys_bytes) vram_base;  
mr.mr_limit = mr.mr_base + vram_size;  
  
if( OK != (r = sys_privctl(SELF, SYS_PRIV_ADD_MEM, &mr)))  
    panic("sys_privctl (ADD_MEM) failed: %d\n", r);
```

**Answer** In modern operating systems, **user-level processes** cannot access **directly** HW resources, including physical memory and VRAM

- ▶ Minix 3 handles this by allowing to grant **privileged** user-level processes the permissions they require to perform their tasks

## Lab 5 - Part 1: Key Programming Issue

**Issue** Given a virtual address, how can a program access the physical memory mapped to that virtual address?

**Solution** Use C pointers

- ▶ Pay attention to the size of a pixel.