

Topic 3: (Part B)

I/O and Peripheral Devices Management

GRAPHICS in the Nintendo DS

Systemes Embarqués Microprogrammés

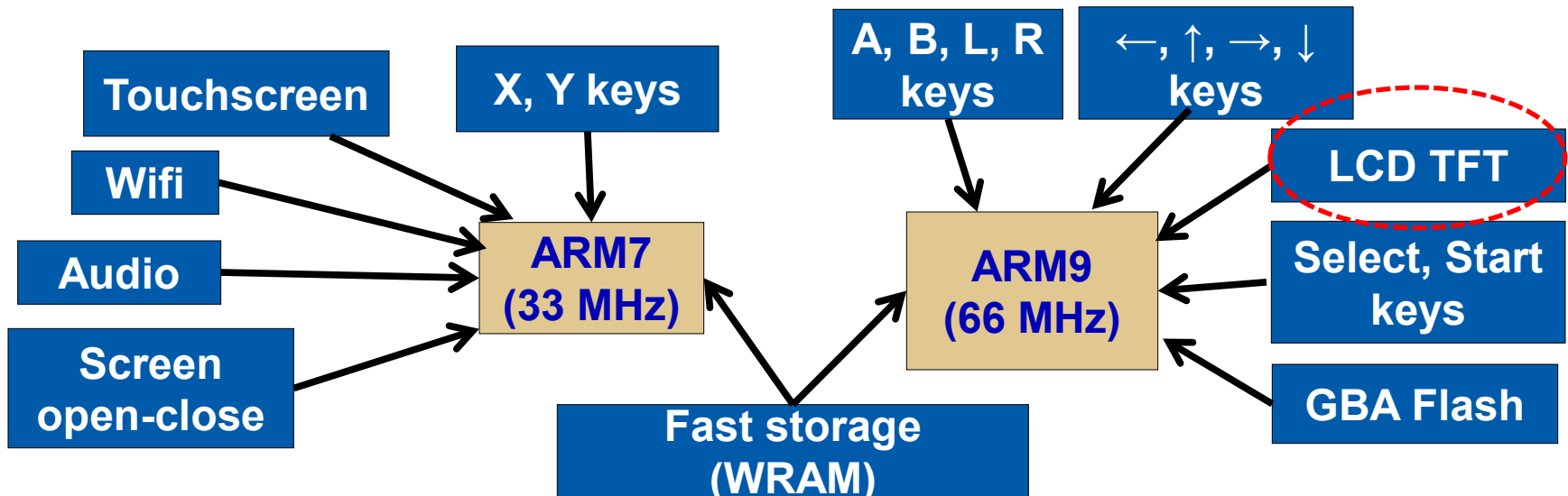
- Fundamental graphics in the NDS
 - Graphics subsystem
 - Concepts: colours and bitmaps
 - NDS video modes
 - Review of the (V)RAM structure

- Drawing graphics in framebuffer mode
 - Implementation of fundamental graphic functions to draw lines and filling colours on areas of the background
 - Drawing basic shapes on the NDS screen (rectangles, lines, triangles) with different colours

Graphics subsystem in the NDS:

The screens and the engines

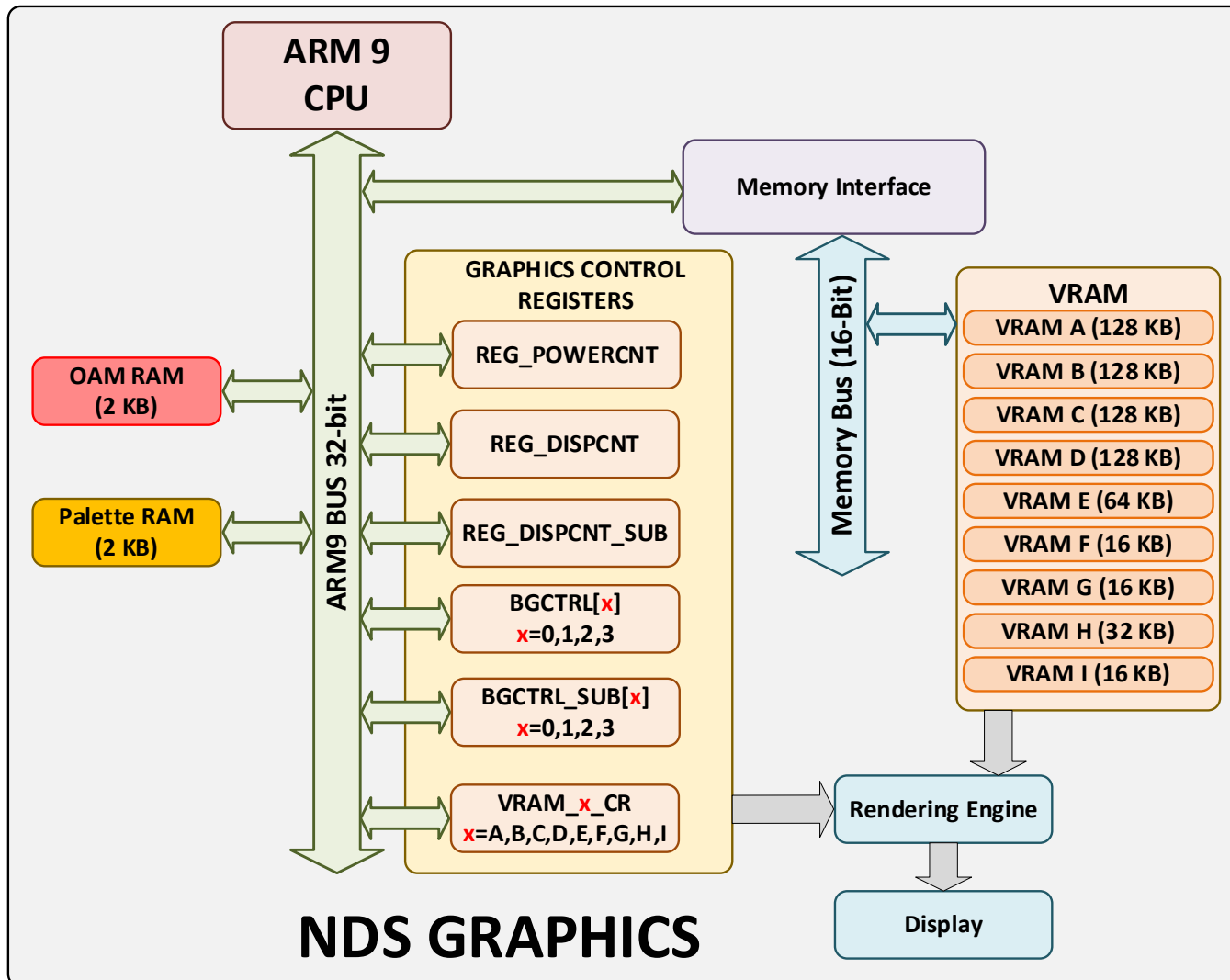
- Two screens on the NDS, both use memory-mapped graphical I/O interface
 - Resolution of 192x256 pixels each screen



- Two 2D graphics engines in the NDS
 - Main: it can display both video memory content or bitmaps of 256 colours
 - Also it can use the proprietary 3D graphics engine for backgrounds
 - Sub: secondary display that can only use the video memory content

Graphics subsystem in the NDS: Registers and Memories

- Graphics Control Registers and VRAM memory banks



Configuration Sequence of Framebuffer Mode

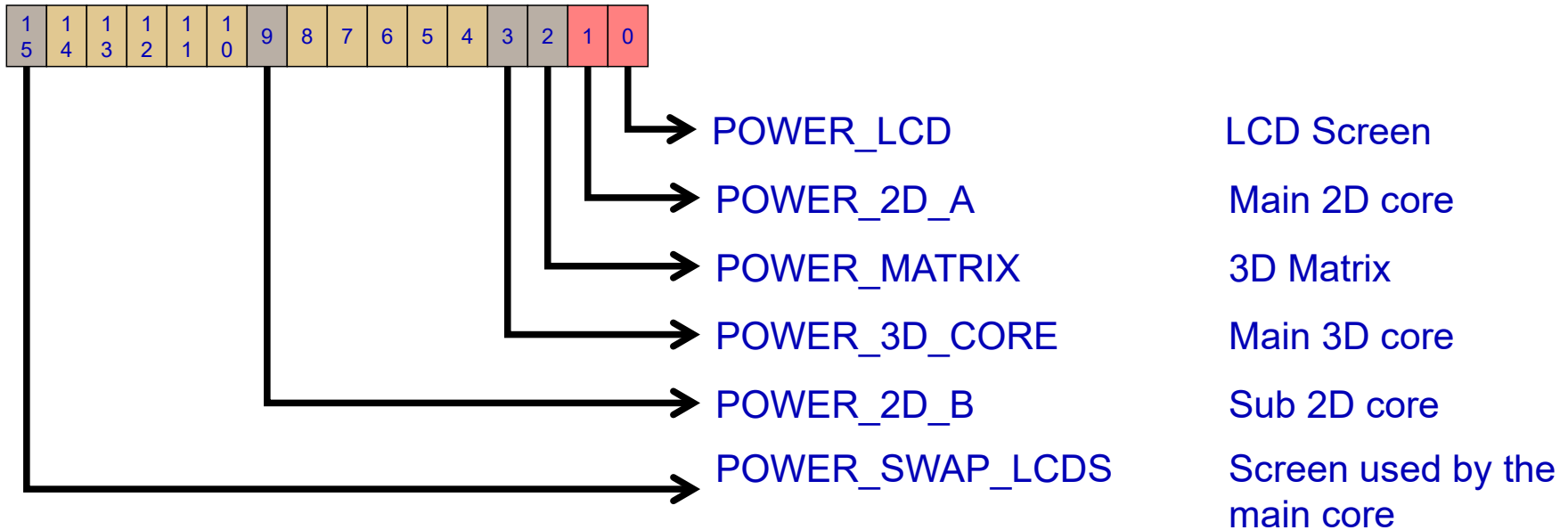
1. Power Manager configuration (REG_POWERCNT)
 - Performed with default settings in system boot-up
2. Graphical engines configuration (REG_DISPCNT)
 - Configure mode
3. VRAM configuration (VRAM_x_CR)
 - Activate banks and configure them (depending on used framebuffer mode)

Ready to filling the screen canvas with graphics!

Graphics subsystem in the NDS: Activating screens and graphic coprocessors

- Control register for powering up I/O NDS devices: REG_POWERCNT
 - It is mapped on the memory address: 0x4000304

REG_POWERCNT



- In libnds, activating the LCD and engines can be done with macros:

```
REG_POWERCNT = POWER_LCD | POWER_2D_A;
```

- Pixels: the basic screen drawing elements
 - Each pixel has an RGB (Red-Green-Blue) representation of 16 bits
 - 5 bits to show the intensity of each colour (0: none, 31: maximum value)
 - 1 bit for transparency (0: pixel is transparent, 1: pixel is opaque)

- In libnds, macros *ARGB16()* and *RGB15()* can create each pixel:

```
static uint16 shape_color = ARGB16(1, 31, 0, 0);
static uint16 shape_color = RGB15(31, 0, 0);
```

Libnds: RGB15	Colour
RGB15(31,0,0)	Red
RGB15(0,31,0)	Green
RGB15(0,0,31)	Blue
RGB15(0,0,0)	Black

- Resolution relates to draw (render) raster graphic bitmaps: matrix of pixels
 - 49152 pixels: 192 rows of 256 points each (96KB)
 - The screen draws sequentially each point from left to right and up to down
 - Two interrupts occurs from the screen drawing

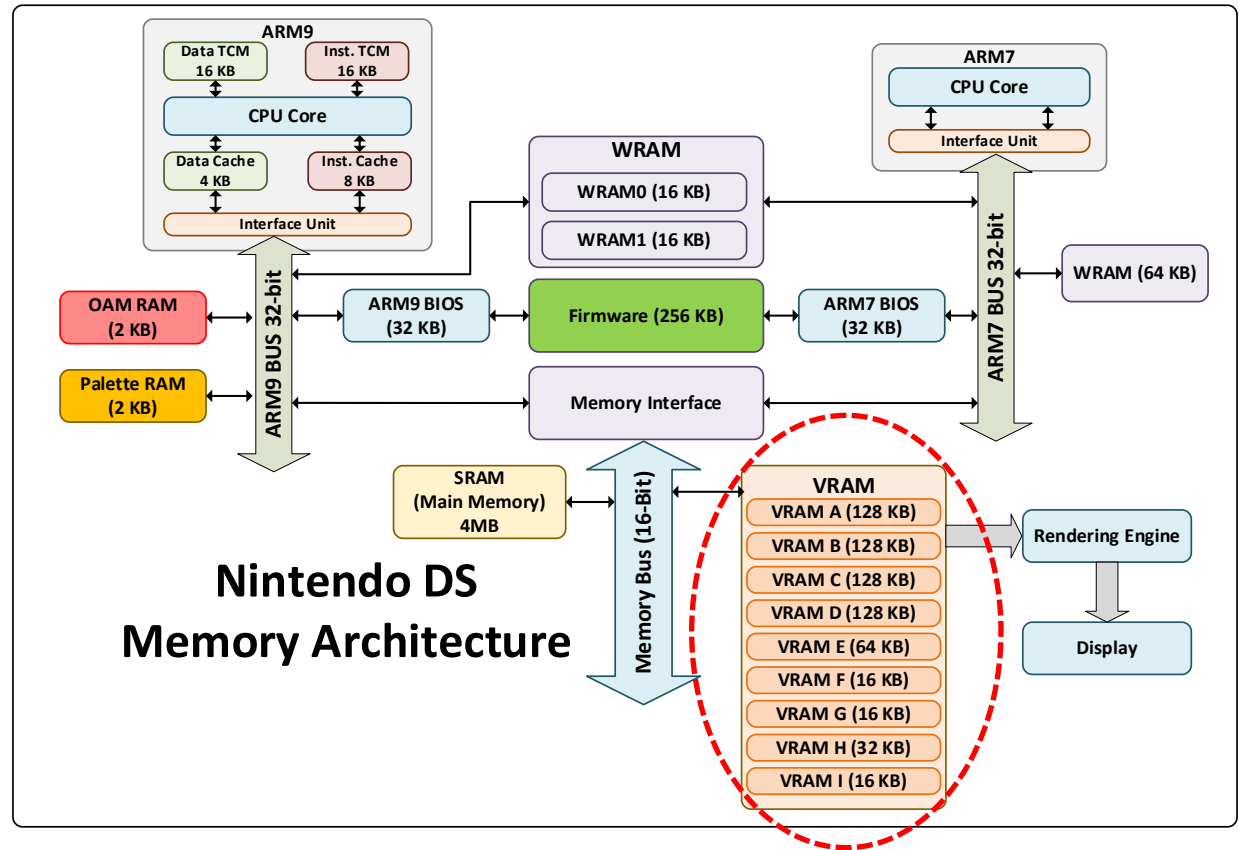
- When do we change the bitmap content to draw?
 - In the interval between VBLANK and start redrawing again from the top-left pixel

Libnds: IRQ_MASK	Description
IRQ_VBLANK	Vertical blank
IRQ_HBLANK	Horizontal blank

But where are the pixels stored?

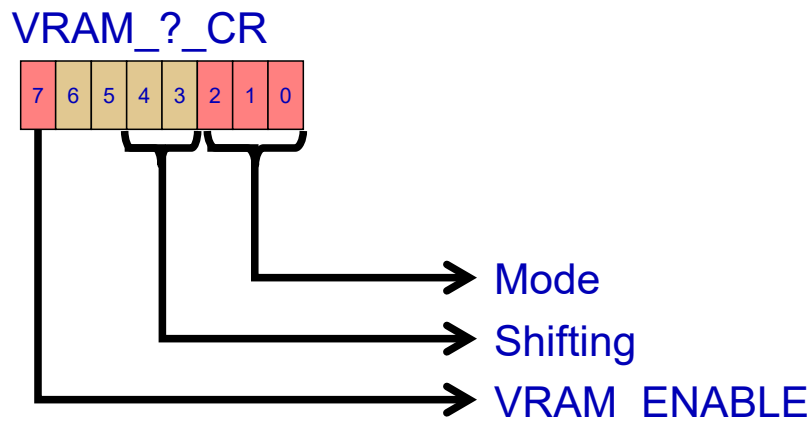
Nintendo DS memory range: Video memory or VRAM

- The bitmaps are stored in a fast RAM memory: VRAM
 - ARM9: **0x06000000 - 0x068A0000** (656KB);
- Divided in 9 banks
 - Different uses/modes
 - Backgrounds or layers
 - Different bank sizes



- Control register for each bank: control select and activate: **VRAM_?_CR**

-VRAM_A	(128KiB)
-VRAM_B	(128KiB)
-VRAM_C	(128KiB)
-VRAM_D	(128KiB)
-VRAM_E	(64KiB)
-VRAM_F	(16KiB)
-VRAM_G	(16KiB)
-VRAM_H	(32KiB)
-VRAM_I	(16KiB)



- Example: activate bank A and map it to the 2D background representation with the main 2D core

```
VRAM_A_CR = VRAM_ENABLE | VRAM_A_LCD;
```

- Set functions are also available: `/opt/devkitPro/libnds/include/nds/arm9/video.h`
 - Example: VRAM_A bank mapped to main 2D core background

```
vramSetBankA(VRAM_A_LCD);
```

- Each engine has four backgrounds (or layers): BG0, BG1, BG2 and BG3
 - Final view on the screen will be their combination based on used graphic mode
- Each 2D engine has different sets of four possible modes:
 - Tiled
 - Rotation or rotoscale
 - Extended rotation
 - Framebuffer (special rendering mode, without backgrounds)

■ Main engine

- 7 modes and framebuffer

Mode	BG0	BG1	BG2	BG3
0	Tiled/3D	Tiled	Tiled	Tiled
1	Tiled/3D	Tiled	Tiled	Rotoscale
2	Tiled/3D	Tiled	Rotoscale	Rotoscale
3	Tiled/3D	Tiled	Tiled	Ext. Rotoscale
4	Tiled/3D	Tiled	Rotoscale	Ext. Rotoscale
5	Tiled/3D	Tiled	Ext. Rotoscale	Ext. Rotoscale
6	3D	N/A	Large Bitmap	N/A
FrameBuf.	Direct VRAM display as a bitmap			

■ Sub engine

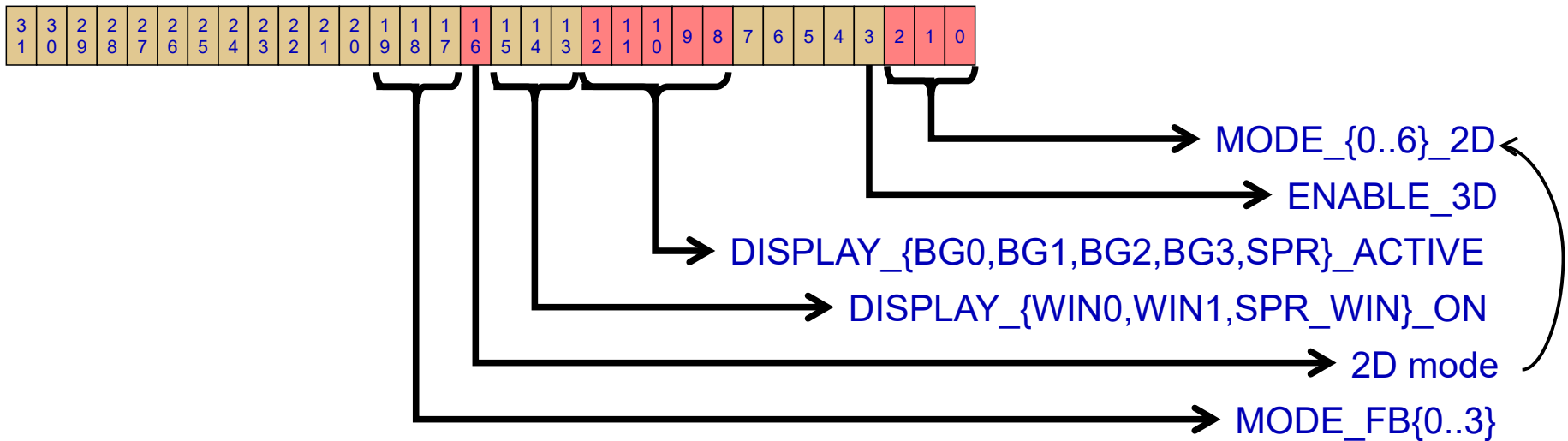
- 6 modes, without 3D or framebuffer

Mode	BG0	BG1	BG2	BG3
0	Tiled	Tiled	Tiled	Tiled
1	Tiled	Tiled	Tiled	Rotoscale
2	Tiled	Tiled	Rotoscale	Rotoscale
3	Tiled	Tiled	Tiled	Ext. Rotoscale
4	Tiled	Tiled	Rotoscale	Ext. Rotoscale
5	Tiled	Tiled	Ext. Rotoscale	Ext. Rotoscale

NDS screen mode control

- REG_DISPCNT: display register to control mode and active backgrounds

REG_DISPCNT



- Example: activate mode 0 and background 1 (BG1)
REG_DISPCNT = MODE_0_2D | DISPLAY_BG1_ACTIVE;

Configuring Memory-Mapped I/O Peripheral Registers

- LibNDS library under /opt/devkitPro defines the registers ID

```
#define REG_DISPCNT *(vu32*)0x04000000
```

```
*(vu32*)0x04000000
```

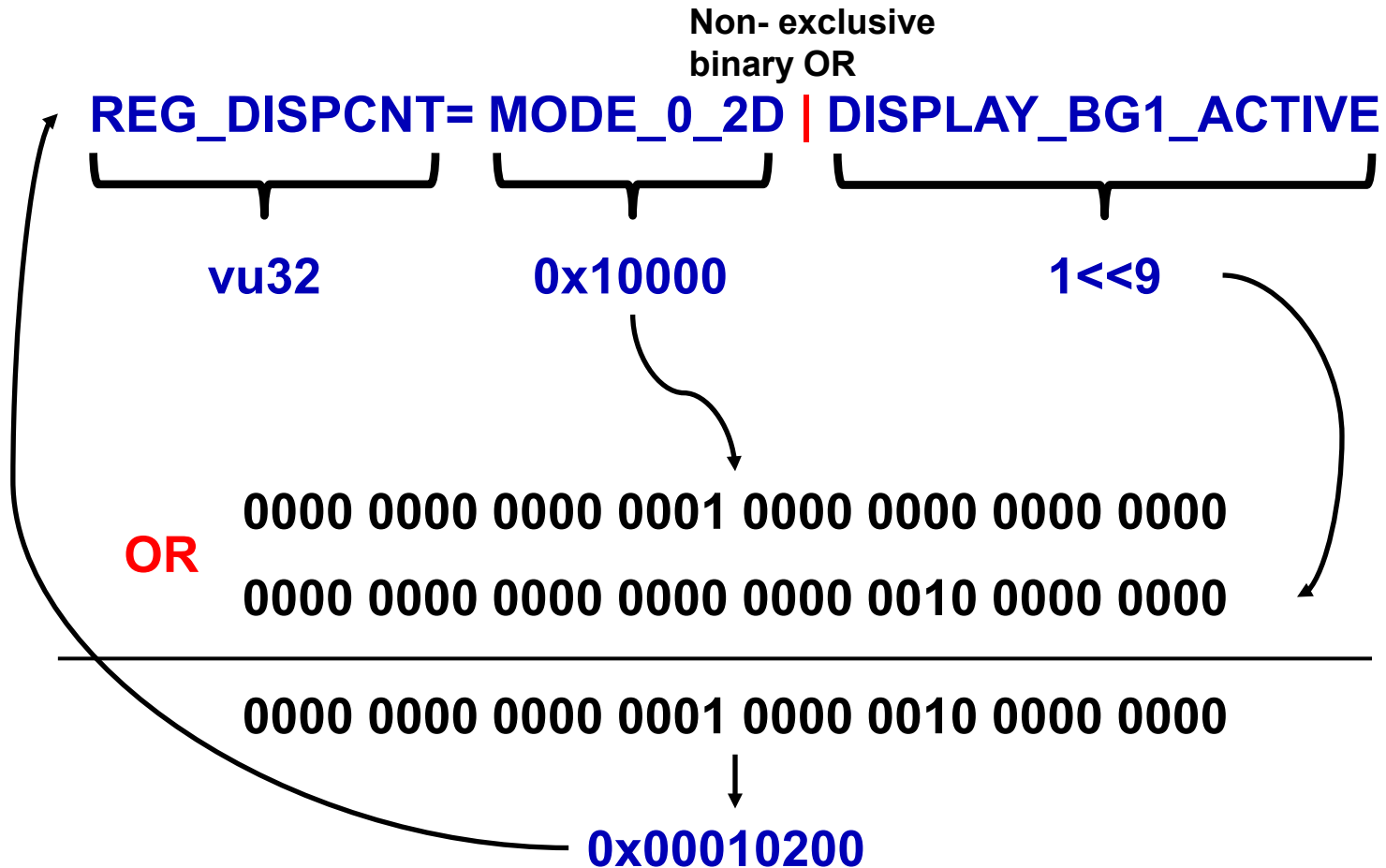
Volatile 32 bit
Unsigned integer

0000 0100 0000 0000 0000 0000 0000 0000

- Hint: Use the following command in the virtual image terminal to find any macro's value:
\$: grep -rnw '/opt/devkitPro/libnds/include/nds' -e 'MACRO_NAME'

Example: Enabling and Assigning VRAM Block A to Main Screen in Mode 0

- C language operation meaning:



The framebuffer mode

- Framebuffer mode: drawing the map of pixels directly
 - Screen mapped to a portion of memory: bitmap or matrix of pixels (192 x 256)
 - Writing data to this memory results in data represented onto the screen

- Four different framebuffers: FB0.. FB3
 - Mapped to the 128KiB VRAM banks
 - Support for *double buffering*:
hiding changes in the pixels content
 - One buffer is being read to write on the screen, while another one is being written in memory
 - Exchange during VBLANK interrupt

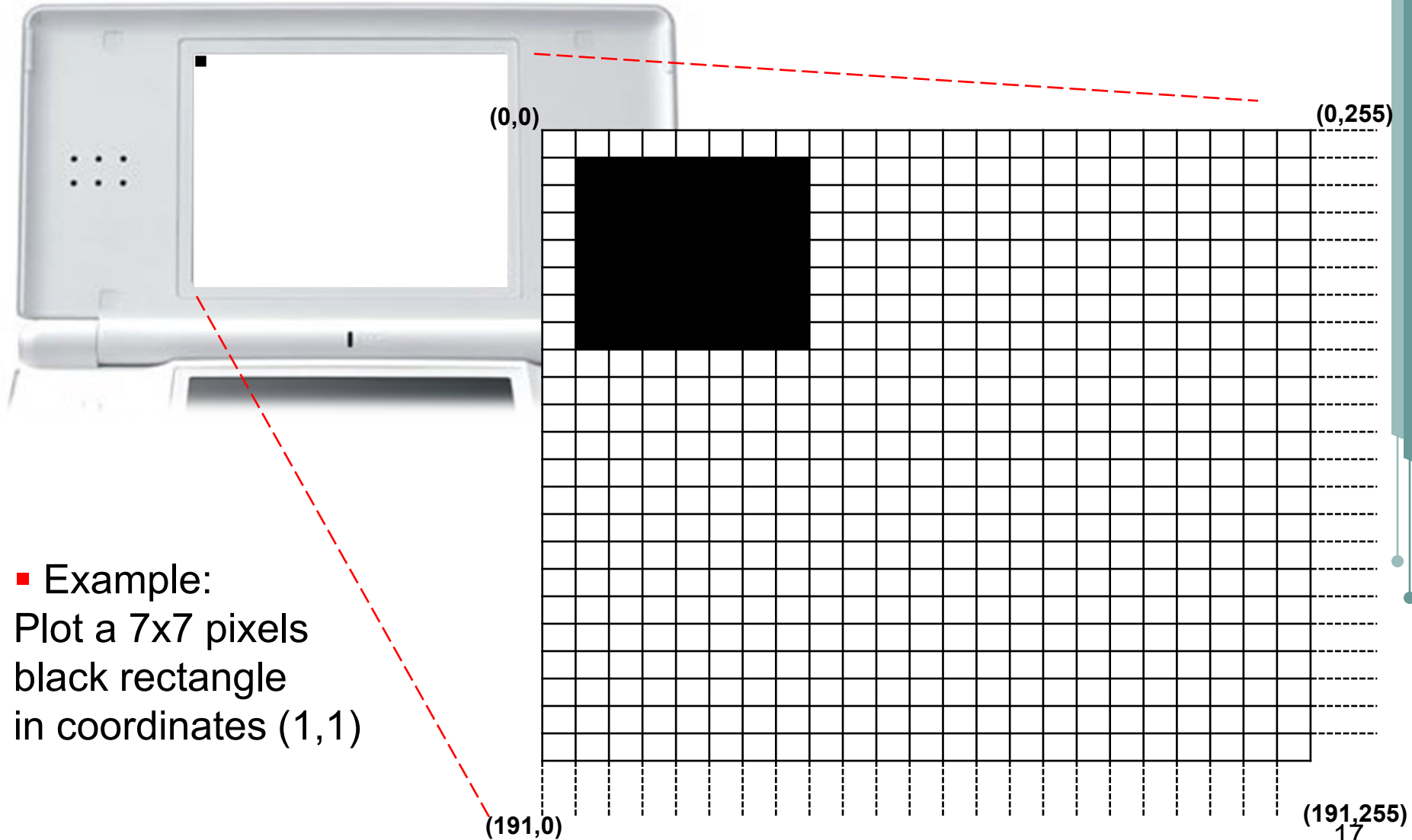
- Example of use: Use of FB0, so two steps:
 1. Framebuffer configured in VRAM bank A (VRAM_A)
 2. VRAM_A activated and configured to work with the LCD screen

Mode	VRAM bank
FB0	VRAM_A
FB1	VRAM_B
FB2	VRAM_C
FB3	VRAM_D

```
REG_DISPCNT = MODE_FB0;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_LCD;
```

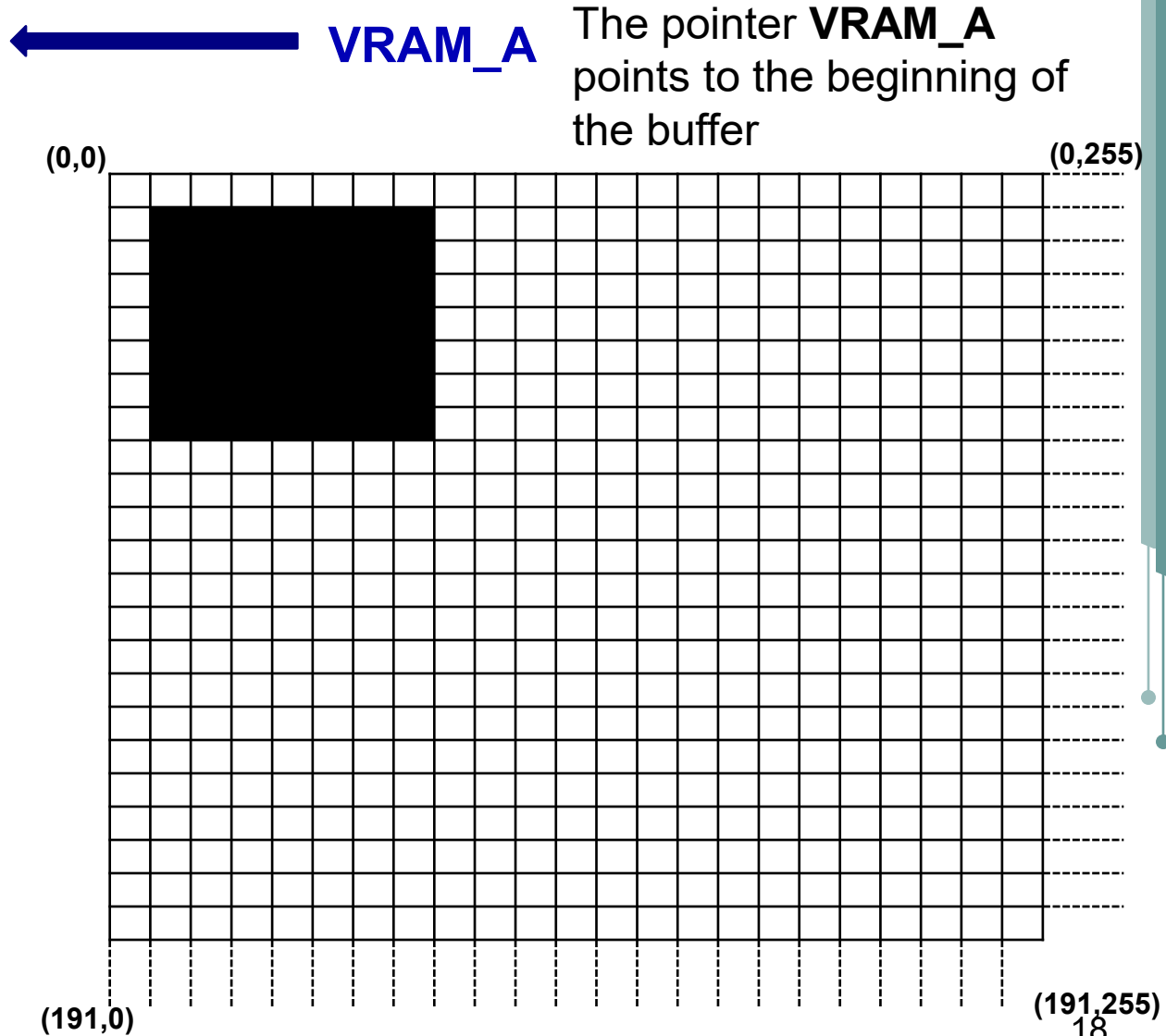
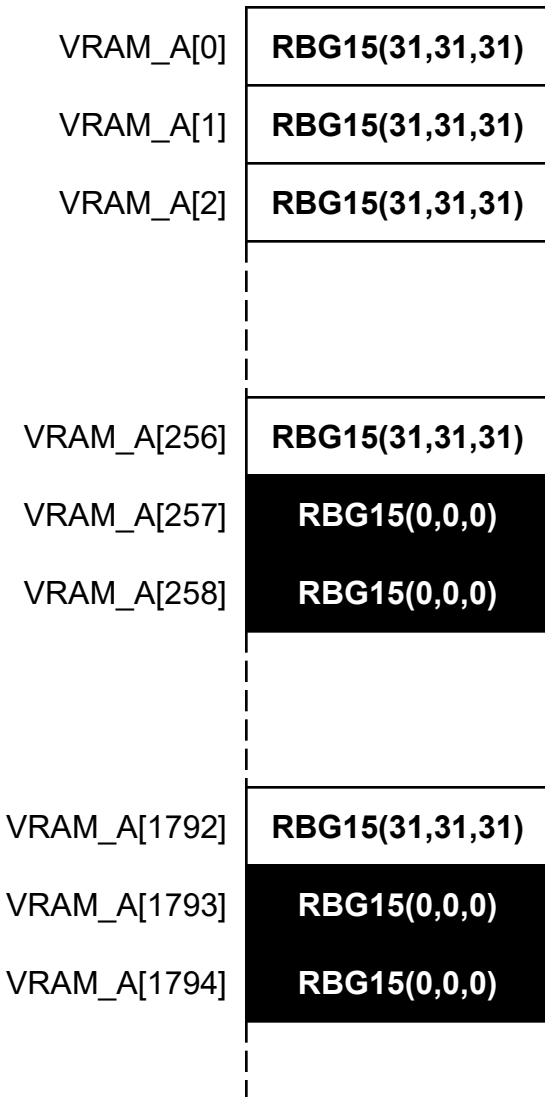
- A framebuffer is characterized by three elements:
 - The memory portion where it is mapped to
 - FB0 in VRAM_A, FB1 in VRAM_B, ...
 - The line length
 - In the NDS it is of 256 pixels per line
 - The pixel format:
 - RGB15(r,g,b)
 - R, G, B: 5 bits per channel (0..31)
 - Most significant bit (Bit15) is not used: pixels are always opaque

Practical Work 6: Graphics (Part 1): Framebuffer Mode

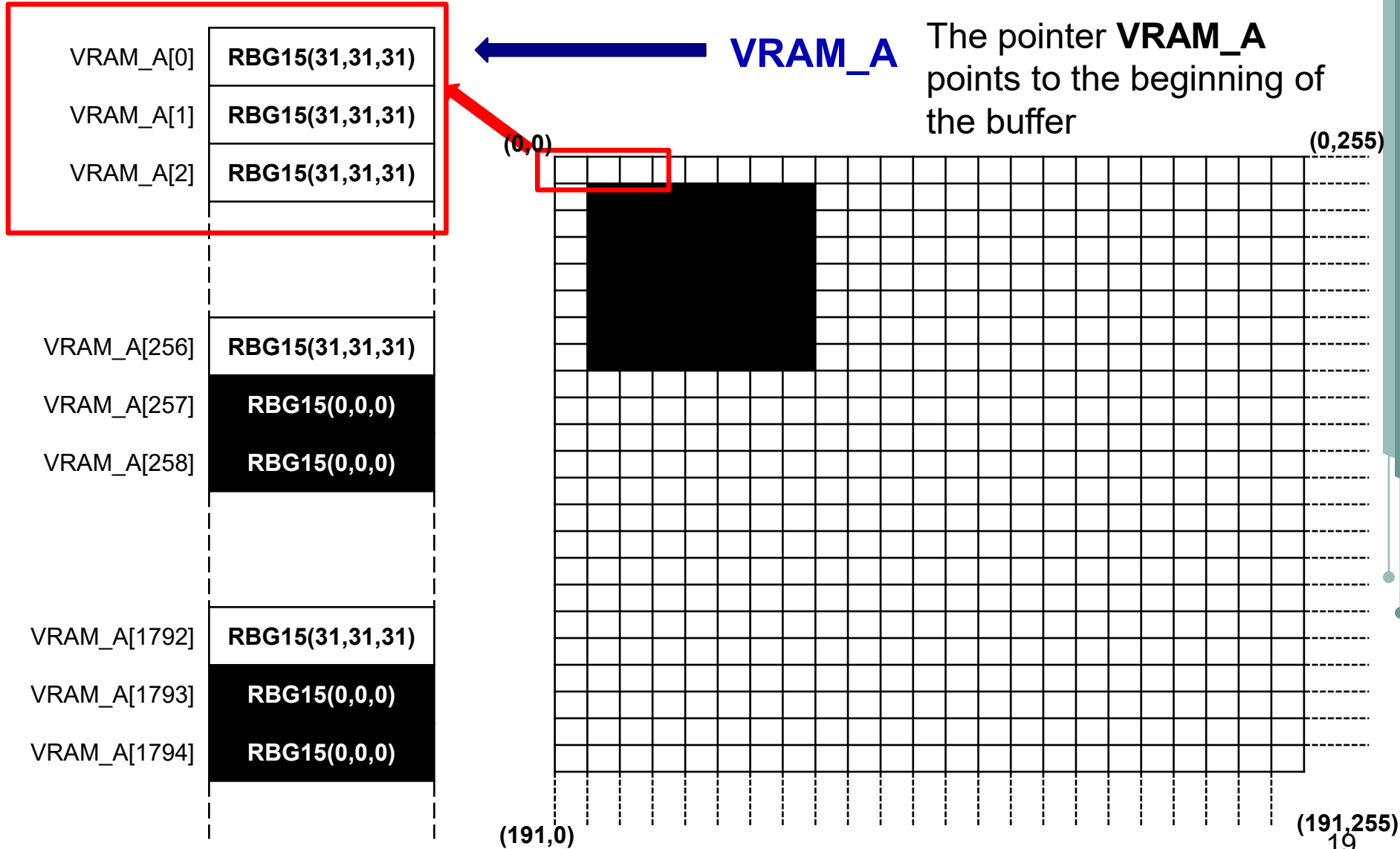


- Example:
Plot a 7x7 pixels
black rectangle
in coordinates (1,1)

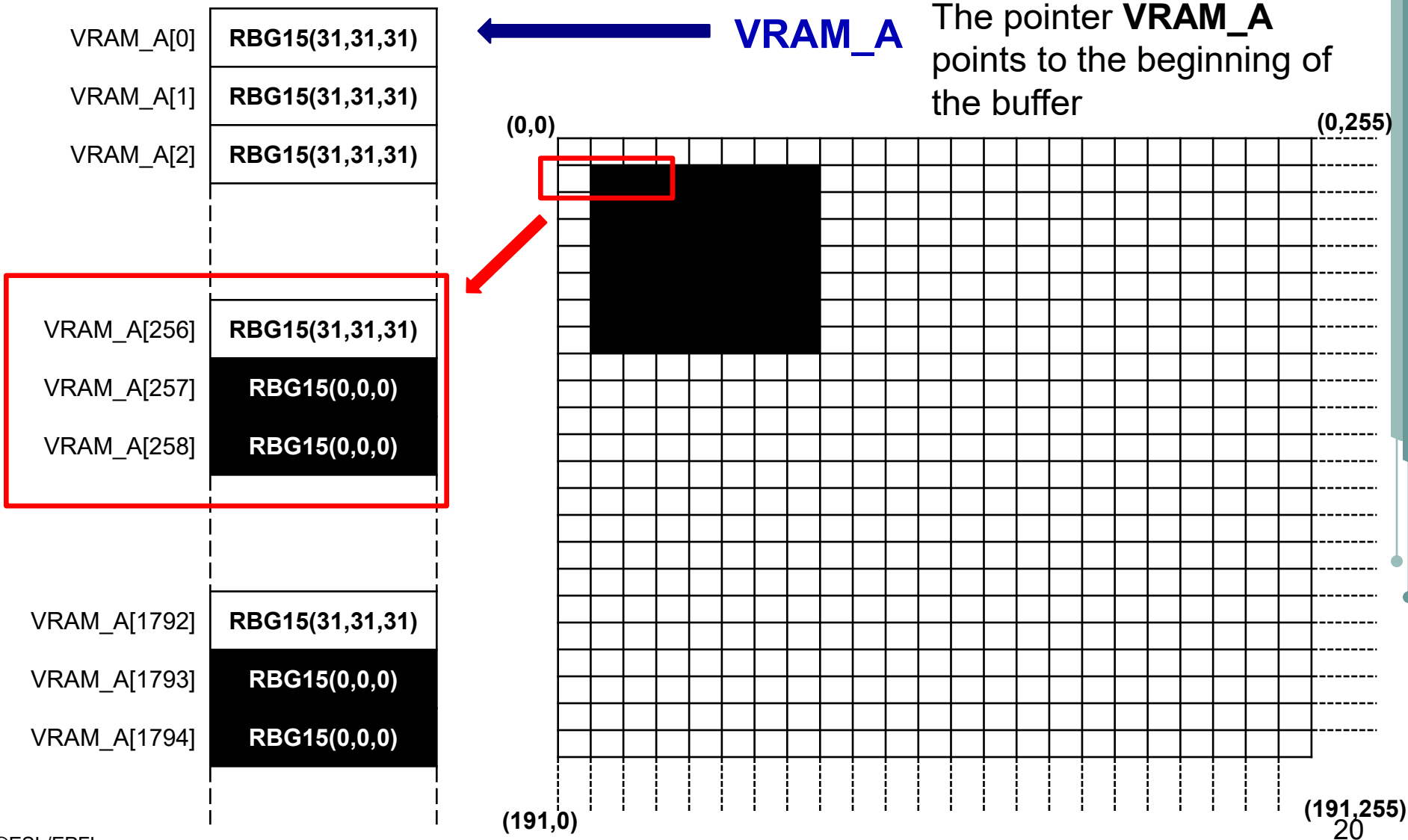
Practical Work 6: Graphics (Part 1): Framebuffer Mode



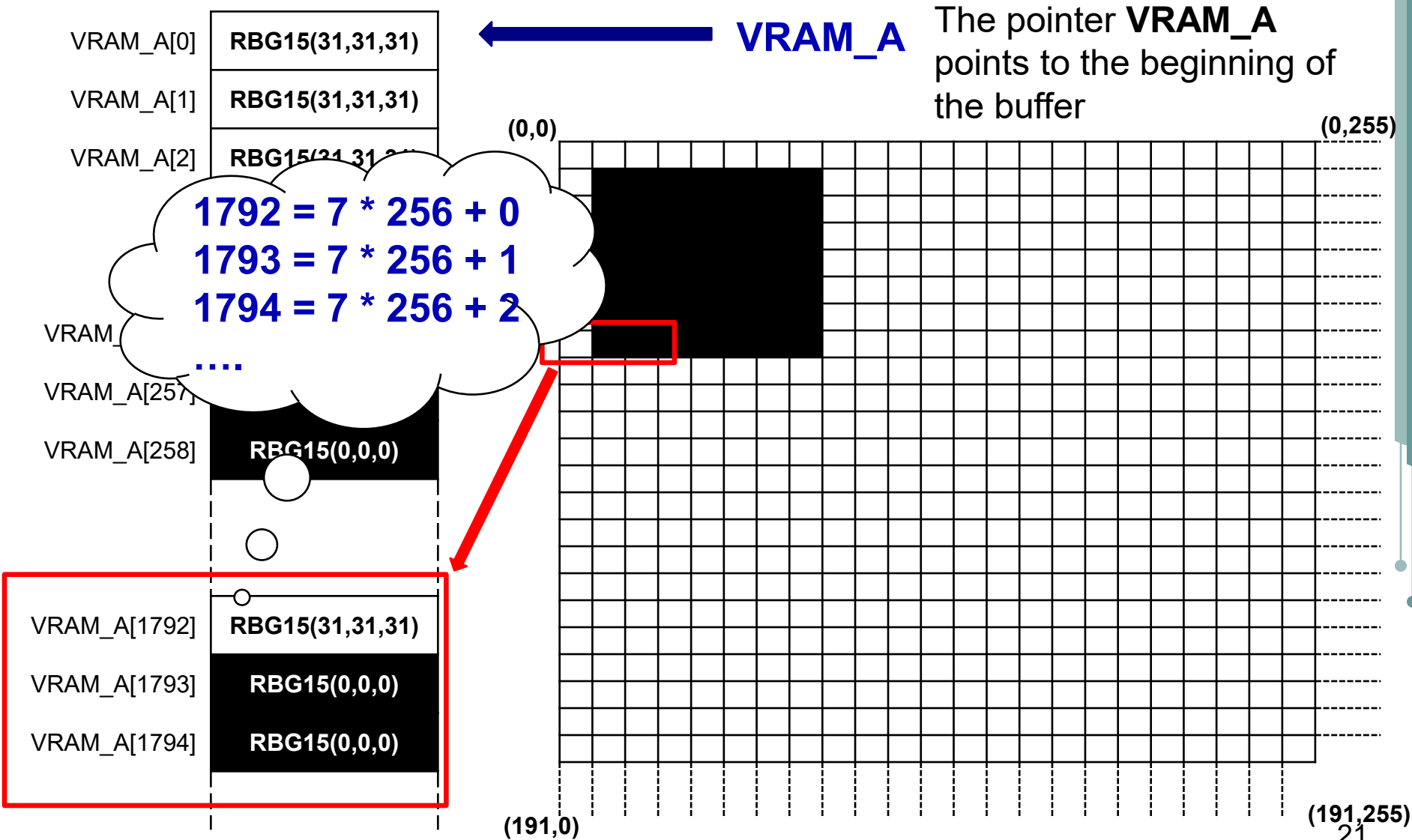
Practical Work 6: Graphics (Part 1): Framebuffer Mode



Practical Work 6: Graphics (Part 1): Framebuffer Mode



Practical Work 6: Graphics (Part 1): Framebuffer Mode



Practical Work 6: Graphics (Part 1): Framebuffer Mode

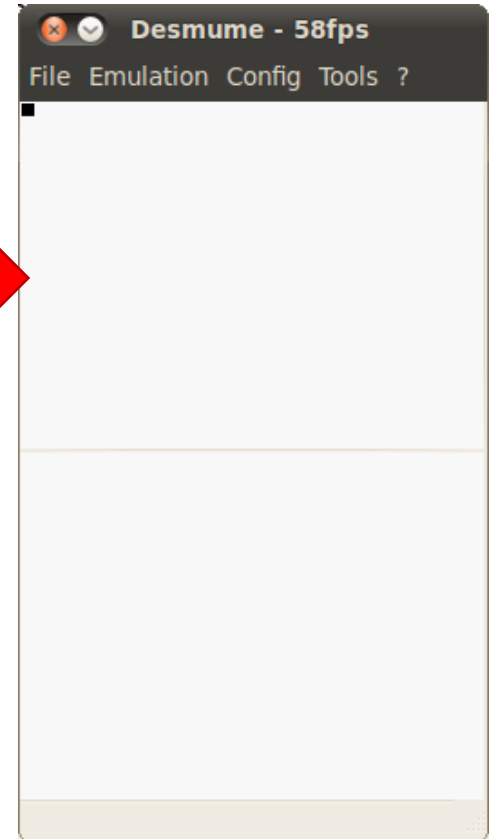
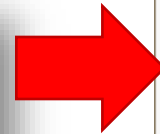
- Solution to the example: code that fills the screen with white, and then draws a black-filled rectangle
 - White => RGB15(31,31,31)
 - Black => RGB15(0,0,0)

```
//counters for the for loops
int row, column;

//Pointer to the buffer
u16* buff = (u16*) VRAM_A;

//Fill the screen with white
for(row = 0; row < 192; row++)
    for(column = 0; column < 256; column++)
        buff[row * 256 + column] = RGB15(31,31,31);

//Draw the 7x7 pixels rectangle
for(row = 1; row < 8; row++)
    for(column = 1; column < 8; column++)
        buff[row * 256 + column] = RGB15(0,0,0);
```

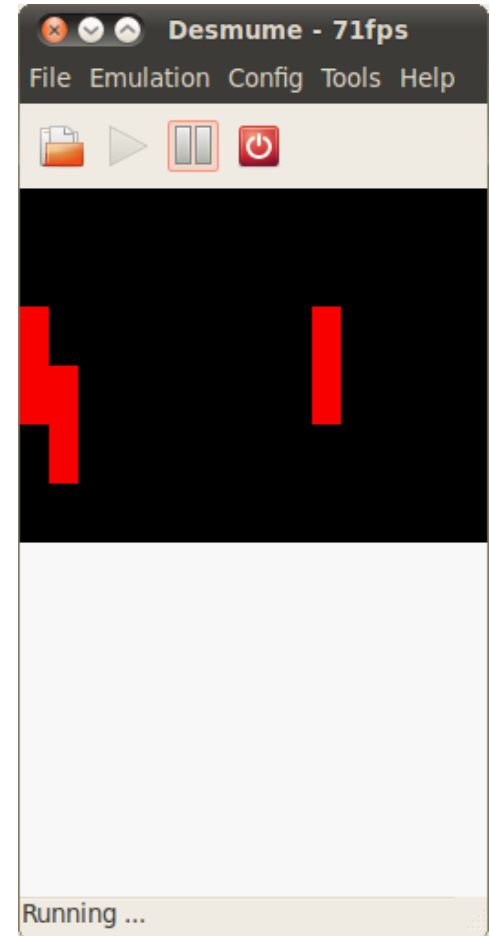


Practical Work 5: Graphics (Part 1): Framebuffer Mode

- Exercises (and homework)
 - Exercise 1 – Initializing the main graphical engine
 - Exercise 2 – Changing the color of the screen
 - Exercise 3 – Filling a rectangle
 - Exercise 4 – Drawing an horizontal line
 - Exercise 5 – Drawing a vertical line
 - Exercise 6 – Drawing a rectangle - reusing code
 - Exercise 7 – Integrating the code (Tetris tiles)

 - *Exercise 8 – Color degradation
 - *Exercise 9 – Shifting data
 - *Exercise 10 – Grayscale transformation

* Additional exercises



Questions?



Let's use the screen of the NDS!

Reminder: Static vs. Dynamic Memory Management

- **Static allocation:** The size of the object is known before execution

```
int array[10];
```

- **Dynamic allocation:** When the size is known only at run-time

```
void *malloc(size_t size) ; /* similar to new int [ ] in C++*/
```

- Dynamically allocated memory must be freed once it is not required

```
void free(void *pointer); /* similar to delete [ ] pointer in C++*/
```

```
#include <stdlib.h>  
//Static memory  
int StaticArray[10];  
  
//Dynamic memory  
int *DynamicArray; //Static pointer  
  
...  
DynamicArray = malloc( n*sizeof(int) ); //n=size of the  
array  
ProcessArray(DynamicArray);  
free(DynamicArray); //de-allocate memory
```

Dynamic Memory: Vectors with Unknown Size

```
#include <stdlib.h>
#define NUM_VEC 10

// Static vector of pointers
int *p[NUM_VEC];
int *q;

for (int i = 0; i < NUM_VEC; ++ i)
    p[i] = malloc((i+1) * sizeof(int));

q = p[0];
*q = 1;

q = p[1];
*q = 3;
*(q+1) = 5;
*(q+2) = 5; // Error!

for (int i = 0; i < NUM_VEC; ++ i)
    free(p[i]);
```

The size of each vector is unknown before run-time, but the number of vectors is known.

```
#include <stdlib.h>
```

```
int numVec; // Init somewhere else.
int **p; // Pointer-to pointer-to integer.
int *q;
```

```
p = malloc(numVec * sizeof(int*));
```

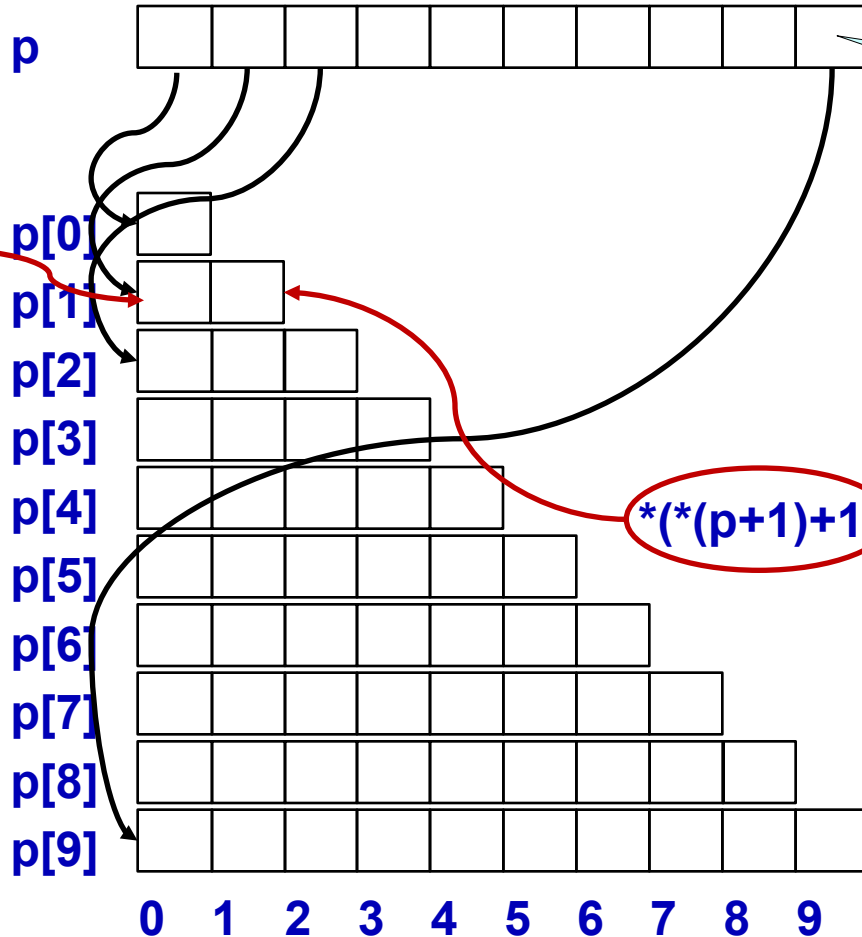
```
for (int i = 0; i < numVec; ++ i)
    p[i] = malloc((i+1) * sizeof(int));
```

```
q = p[0];
*q = 1;
q = p[1];
*q = 3;
*(q+1) = 2;
*(*(p + 1) + 1) = 5;
```

```
for (int i = 0; i < numVec; ++ i)
    free(p[i]); // Free each vector
free(p); // Free the vector of pointers
```

Neither the size of the vectors nor their number is known before run-time.

Example of Dynamic Memory: Pointer to Vector of Pointers



The data structure is similar in both cases, but, in the second case, the size of p is decided at run-time.

In the case of dynamically allocated vectors, it is not possible to use 2D matrix notation!

Important use when you have lots of backgrounds to save and not enough space in VRAM memory!