

Introduction to Computer Graphics

(C S 6 0 2)

Lecture No 04

Point

4.1 Pixel

The smallest dot illuminated that can be seen on screen.

4.2 Picture

Composition of pixels makes picture that forms on whole screen

4.3 Resolution

We know that Graphics images on the screen are built up from tiny dots called picture elements or pixels. The display resolution is defined by the number of rows from top to bottom, and number of pixels from left to right on each scan line.

- Since each mode uses a particular resolution. For example mode 19 uses a resolution of 200 scan lines, each containing 320 pixels across. This is often referred to as 320*200 resolution.
- In general, higher the resolution, more pleasing is the picture. Higher resolution means a sharper, clearer picture, with less pronounced 'staircase' effect on lines drawn diagonally and better looking text characters. On the other hand, higher resolution also means more memory requirement for the display.

4.4 Text and Graphics Modes

We discussed different video hardware devices that include VGA cards and monitors. Video cards are responsible to send picture data to monitor each time it refresh itself. Video cards support both different text and graphics modes. Modes consist of their own refresh rate, number of colors and resolutions (number of rows multiply by number of columns). The following famous video modes that we can set in today's VGA cards on different refresh rate:

- 25 * 80 with 16 colors support (text mode)
- 320 * 200 with 8 bit colors support (graphics mode)
- 640 * 480 with 16 colors support (graphics mode)
- 640 * 480 with 8, 16, 24, 32 bit color support (graphics mode)
- 800 * 600 with 8, 16, 24, 32 bit color support (graphics mode)

4.5 Text and Graphics

All modes are fundamentally of two types, text or graphics. Some modes display only text and some are made only for graphics. As seen earlier, the display adapter continuously dumps the contents of the VDU (video display unit) memory on the screen.

The amount of memory required representing a character on screen in text mode and a pixel in graphics mode varies from mode to mode.

Mode No.	Type	Resolution	Memory Required
3	Text	80 x 25	2 bytes per char
6	Graphics	640 x 200	1 bit per pixel
7	Text	80 x 25	2 bytes per char
18	Graphics	640 x 480	1 bit per pixel
19	Graphics	320 x 200	1 byte per pixel

In mode 6 each pixel displayed on the screen occupies one bit in VDU memory. Since this bit can take only two values, either 0 or 1, only two colors can be used with each pixel.

4.6 How text displays

As seen previously text modes need two bytes in VDU memory to represent one character on screen; of these two bytes, the first byte contains the ASCII value of the character being displayed, whereas the second byte is the attribute byte. The attribute byte controls the color in which the character is being displayed.

The ASCII value present in VDU memory must be translated into a character and drawn on the screen. This drawing is done by a character generator this is part of the display adapter or in VBIOS. The CGA has a character generator that uses 8 scan lines and 8 pixels in each of these scan lines to produce a character on screen; whereas the MA's character generator uses 9 scan lines and 14 pixels in each of these scan lines to produce a character. This larger format of MA makes the characters generated by MA much sharper and hence easier to read.

On older display adapters like MA and CGA, the character generator is located in ROM (Read Only Memory). EGA and VGA do not have a character generator ROM. Instead, character generator data is loaded into plane 2 of display RAM. This feature makes it easy for custom character set to be loaded. Multiple character sets (up to 4 for EGA and up to 8 for VGA) may reside in RAM simultaneously.

A set of BIOS services is available for easy loading of character sets. Each character set can contain 256 characters. Either one or two character sets may be active giving these adapters on the screen simultaneously. When two character sets are active, a bit in each character attribute byte selects which

character set will be used for that character.

Using a ROM-BIOS service we can select the active character set. Each character in the standard character set provided with the EGA is 8 pixels wide and 14 pixels tall. Since VGA has higher resolution, it provides a 9 pixel wide by 16 pixels tall character set. Custom character set can also be loaded using BIOS VDU services.

The graphics modes can also display characters, but they are produced quite differently. The graphics modes can only store information bit by bit. The big advantage of this method is that you design characters of desired style, shape and size.

4.7 Text mode colors

In mode 3, for each character on screen there are two bytes in VDU memory, one containing the ASCII value of the character and other containing its attribute. The attribute byte controls the color of the character. The attribute byte contains three components: the foreground color (color of the character itself), the background color (color of the area not covered by the character) and the blinking component of the character. The next slide shows the breakup of the attribute byte.

Bits								Purpose
7	6	5	4	3	2	1		
X	x	x	x	x	x	x	1	Blue component of foreground color
X	x	x	x	x	x	1	x	Green component of foreground color
X	x	x	x	x	1	x	x	Red component of foreground color
X	x	x	x	1	x	x	x	Intensity component of foreground color
X	x	x	1	x	x	x	x	Blue component of background color
X	x	1	x	x	x	x	x	Green component of background color
X	1	x	x	x	x	x	x	Red component of background color
1	x	x	x	x	x	x	x	Blinking component

4.8 Graphics Mode colors

So far we have seen how to set color in text modes. Setting color in graphics modes is quite different. In the graphics mode each pixel on the screen has a color associated with it. There are important differences here as compared to setting color in text mode. First, the pixels cannot blink. Second, each pixel is a discrete dot of color, there is no foreground and background. Each pixel is simply one color or another. The number of colors that each adapter can support and the way each adapter generates these colors is drastically different. But we will only discuss here colors in VGA.

4.9 Colors in VGA

IBM first introduced the VGA card in April 1987. VGA has 4 color planes – red, green, blue and intensity, with one bit from each of these planes contributing towards 1 pixel value.

There are lots of ways that you can write pixel on screen. You can write pixel on screen by using one of the following methods:

- Using video bios services to write pixel
- Accessing memory and registers directly to write pixel on screen.
- Using library functions to write pixel on screen

4.10 Practical approach to write pixel on screen

As we have discussed three ways to write pixel on screen. Here we will discuss all these ways practically and see how the pixel is displayed on screen. For that we will have to write code in Assembly and C languages. So Get ready with these languages

4.11 Writing pixel Using Video BIOS

The following steps are involved to write pixel using video BIOS services.

- Setting desired video mode
- Using bios service to set color of a screen pixel
- Calling bios interrupt to execute the process of writing pixel.

Source code

Below are the three lines written in assembly language that can set graphics mode 19(13h). You can use this for assembler or you can embed this code in C language using '*asm*' keyword

```
MOV AH,0
MOV AL,13h ;;mode number from 0-19
INT 10H
```

To insert in C language above code will be inserted with key word *asm* and curly braces.

```
asm{
    MOV AH,0
    MOV AL,13h ;;mode number from 0-19
    INT 10H
}
```

Description

- Line #1: `mov ah,0` is the service number for setting video mode that is in register ah
- Line #2: `mov al,13h` is the mode number that is in register al

- Line #3: int 10h is the video bios interrupt number that will set mode 13h

Source code for writing pixel

The following code can be used to write pixel using video bios interrupt 10h and service number 0ch.

```
MOV AH,0Ch
MOV AL,COLOR_NUM
MOV BH,0
MOV CX,ROW_NUM
MOV DX,COLUMN_NUM
INT 10h
```

Description

- Line#1: service number in register aH
- Line#2: color value, since it is 13h mode so it has 0-255 colors range. You can assign any color number between 0 to 255 to al register. Color will be selected from default palette setting against the number you have used.
- Line#3: page number in bH register. This mode supports only one page. So 0 is used in bH register. 0 mean default page.
- Line#4: column number will be used in CX register
- Line#5: row number will be used in DX register
- Line#6: BIOS interrupt number 10h

4.12 Writing pixel by accessing memory directly

So far we used BIOS to draw pixel. Here we will draw pixel by accessing direct pointer to the video memory and write color value. The following steps are involved to write direct pixel without using BIOS:

- Set video mode by using video BIOS routine as discussed earlier
- Set any pointer to the video graphics memory address 0x0A0000.
- Now write any color value in the video memory addressing

Direct Graphics Memory Access Code

```
Mov ax,0a000h
Mov ds,ax      ;;segment address changed
Mov si,10      ;; column number
Mov [si],COLOR_NUM
```

- Work to do:
 - Write pixel at 12th row and 15th column
- *Hint:* use formula (row * 320 + column) in si register.

4.13 Writing character directly on screen

You can also write direct text by setting any text mode using BIOS service and then setting direct pointer at text memory address 0x0b8000.

Example

Set mode Number 3. using BIOS service and then use this code to write character

```
Mov ax,0b8000h
Mov ds,ax
Mov si,10           ;;column number
Mov [si],'a'       ;;character to write
```

4.14 Using Library functions

While working in C language, you can use graphics library functions to write pixel on screen. These graphics library functions then use BIOS routines or use direct memory access drivers to draw pixel on screen.

```
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk)
/* an error occurred */
{
    printf("Graphics error: %s\n", getch());    exit(1);
    /* return with error code */
}
/* draw a pixel on 10th row and 10 column */
putpixel(10, 10, BLUE);
/* clean up */
closegraph();
```

4.15 Steps in C language

- First call Initgraph() function
- and then call putpixel() function to draw pixel on screen. It takes row, column and color value as parameters.
- after drawing pixel use closegraph() function to close the graphics routines provided by built in driver by Borland.

4.16 Discussion on pixel drawing methods

BIOS routines are standard routines built in VGA cards but these routines are very much slow. You will use pixel to draw filled triangle, rectangles and circles and these all will be much slower than direct memory access method. Direct memory access method allows you to write pixel directly by passing the complex BIOS routines. It is easy and faster but its programming is only convenient in mode 13h. Library functions are easier to use and even faster because these are optimized and provided with special drivers by different companies.

4.17 Drawing pixel in Microsoft Windows

So far we have been discussing writing pixel in DOS. Here we will discuss briefly how to write pixel in Microsoft Windows. Microsoft windows is complete graphical operating system but it does not allow you to access BIOS or direct memory easily. It provides library functions (APIs) that can be used to write graphics. By working in graphics in windows one must have knowledge about Windows GDI (graphics device interface) system.

4.18 Windows GDI functions

Here are some windows GDI functions that can be used to draw pixel e.g SetPixel and SetPixelV. Both are used to draw pixel on screen. The example and source code of writing pixel in windows will be available.

Window Code Example:

```
// a.cpp : Defines the entry point for the application.
//

#include "stdafx.h"
#include "resource.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                //
current instance
TCHAR szTitle[MAX_LOADSTRING];
        // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];
        // The title bar text

// Forward declarations of functions included in this code
module:
ATOM                MyRegisterClass(HINSTANCE
hInstance);
BOOL                InitInstance(HINSTANCE, int);
```

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR      lpCmdLine,
                    int        nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance,      IDS_APP_TITLE,      szTitle,
MAX_LOADSTRING);
    LoadString(hInstance,      IDC_A,              szWindowClass,
MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    hAccelTable      =      LoadAccelerators(hInstance,
(LPCTSTR)IDC_A);

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable,
&msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return msg.wParam;
}

//
// FUNCTION: MyRegisterClass()
//
```

```
// PURPOSE: Registers the window class.
//
// COMMENTS:
//
//     This function and its usage is only necessary if
// you want this code
//     to be compatible with Win32 systems prior to the
// 'RegisterClassEx'
//     function that was added to Windows 95. It is
// important to call this function
//     so that the application will get 'well formed'
// small icons associated
//     with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = (WNDPROC)WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance     = hInstance;
    wcex.hIcon          = LoadIcon(hInstance,
(LPCTSTR)IDI_A);
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = (LPCSTR)IDC_A;
    wcex.lpszClassName = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance,
(LPCTSTR)IDI_SMALL);

    return RegisterClassEx(&wcex);
}

//
// FUNCTION: InitInstance(HANDLE, int)
//
// PURPOSE: Saves instance handle and creates main
// window
//
// COMMENTS:
//
//     In this function, we save the instance handle
// in a global variable and
```

```
//          create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;

    hInst = hInstance; // Store instance handle in our
    global variable

    hWnd = CreateWindow(szWindowClass, szTitle,
    WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL,
    hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

//
// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
// PURPOSE: Processes messages for the main window.
//
// WM_COMMAND - process the application menu
// WM_PAINT - Paint the main window
// WM_DESTROY - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR szHello[MAX_LOADSTRING];
    LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);

    switch (message)
    {
        case WM_COMMAND:
```

```
        wmId      = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
            case IDM_ABOUT:
                DialogBox(hInst,
(LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
                break;
            case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
            default:
                return DefWindowProc(hWnd, message,
wParam, lParam);
        }
        break;
    case WM_PAINT:
        {
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...
            RECT rt;
            GetClientRect(hWnd, &rt);
            int j=0;
            //To draw some pixels of RED colour on the
screen
            for(int i=0;i<100;i++)
            {
                SetPixel(hdc,i+j,10,RGB(255,0,0));
                j+=6;
            }

            EndPaint(hWnd, &ps);
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam,
lParam);
    }
    return 0;
}

// Mesage handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM
```

```
wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:
            if (LOWORD(wParam) == IDOK || LOWORD(wParam)
== IDCANCEL)
            {
                EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
            break;
    }
    return FALSE;
}
```