

Chapter 4

Graphical programming using *windows.h*

When the code and
the comments disagree,
both are probably wrong.

Norm Schryer

This chapter is devoted to beginners who decided to program the graphical applications not in console regime but in screen window. *Microsoft Corporation* introduced an OS named *Windows* in 1985 as an add-on to *MS-DOS*. Main reason of this new system was the market interest in *Graphical User Interfaces* (GUIs).

GUI programming of so called *Window* type was initiated in OS *Windows-95*, but another OS versions *Windows-98/NT/XP/Vista/7* also support C++ programming based on usage of the library `<windows.h>`. All presented below programs are tested in *Windows7* only, without any guarantee to run it in FAT-based *Windows-95/98/Me* as well as NT-based *Windows-NT/2K/XP/Vista*.

However, the significant exceptions or contradictions between graphical programming in different OS are absent and only negligible differences could occur only. For practical purposes, determination of OS *Window* in this chapter means all OS family, where GUI interface is realized.

OS *Windows* presents a GUI who is responsible to recognize the working video-adaptor / hardware graphical interface. It is not necessary for programmer to go into the question how to operate the graphical hardware, because abstract GUI interface creates the conditions and environment for programming at very low level (pixeling level).

Programming of this type must be started from very low level. In some cases, it gives a delight in possibility to control and to manage many small pieces of interface. In otherwords, it requires high qualification of programmer and very conservative view into programming. It also prevents the mobility, fast changes in code etc. For detail description, see excellent publications of LaMothe [22], [23], Williams [52], and Schildt [41].

This chapter is devoted to describe the main possibilities of window graphical interface, which is essential for manipulating by functions of GDI interface in particularly and for understanding of graphic programming altogether:

- i) formation of graphical interface while using requested parameters;
- ii) possibility to create and manipulate the program using menu buttons and mouse manipulator buttons;
- iii) possibility to make the marker positioning at requested screen place;
- iv) possibility to create graphical primitives on screen: pixel, line, rectangle, ellipse;
- v) possibility to clear requested place of screen.

4.1 Graphical interface

Graphical interface is a programming behaviour made for managing, developing the 2D/3D information on the screen of computer. Graphical interface is treated as one of the most important possibility of information exchange between human and computer. Physiologically, any human receives 80% of information through the eyes.

This section is devoted to problems of window interface creation: main window and menu bar. Mouse manipulator operations are also described.

4.1.1 Window formation

1st programming task. Programming task is presented as follows: to create window with the possibility to switch, to destroy this window and nothing else.

Tables 4.1a, 4.1b, 4.1c represent the code for formation of windows like geometrical figure on the monitor screen. Tables 4.1a, 4.1b are devoted for `WinMain()` function. General structure of programming content is presented below.

```
#include <windows.h>

int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
    ...
    // Window structure
    wincl.lpfnWndProc = WindowProc1234; // function called by windows
}
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM)
{
    // how to operate?
}
```

First of all, library `<windows.h>` with implemented GUI interface is necessary. Secondly, function `int WINAPI WinMain()` is necessary. This function organizes creation of window in dynamical (not static) regime using multithreading technologies. Messages from processes (dispatching technique) allow the redrawing of window. After each cycle of windows redrawing, function `int WINAPI WinMain()` returns the management to OS.

Table 4.1a.

Program win01.cpp.

```

#include <windows.h>

// part of global objects
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "WindowsApp"; // class name

int WINAPI WinMain (HINSTANCE hThisInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR      lpszArgument,
                   int         nFunsterStil)
{
    HWND      hwnd;      // handle for window
    MSG        messages; // messages to the application are saved
    WNDCLASSEX wincl;     // data structure for the windowclass

    // Window structure
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName  = szClassName;
    wincl.lpfnWndProc     = WindowProc1234; // function called by windows
    wincl.style           = CS_DBLCLKS;     // Catch double-clicks
    wincl.cbSize          = sizeof (WNDCLASSEX);

    // Use default icon and mouse-pointer
    wincl.hIcon           = LoadIcon      (NULL, IDI_APPLICATION);
    wincl.hIconSm         = LoadIcon      (NULL, IDI_APPLICATION);
    wincl.hCursor         = LoadCursor    (NULL, IDC_ARROW);
    wincl.lpszMenuName    = NULL; // No menu
    wincl.cbClsExtra      = 0;          // No extra bytes after the window class
    wincl.cbWndExtra      = 0;          // structure or the window instance
    // Windows's default color as the background of the window
    wincl.hbrBackground   = (HBRUSH) COLOR_BACKGROUND;

    // Register the window class, and if it fails quit the program
    if (!RegisterClassEx (&wincl))
        return 0;
    // continued to Table 4.1b

```

Function `int WINAPI WinMain()` contains parameters for window structure description, and partially, parameter for associated so called *call-back* function, which is called from OS according to dispatch activity. User-created function `LRESULT CALLBACK WindowProc1234 ()` contains the main programming information "how to operate".

After describing the library `<windows.h>`, part of global objects is present. It is necessary to declare the function `LRESULT CALLBACK WindowProc1234()` (user-created function!) and also class name: `szClassName[ ]` (string in C style), which will be used in all associated files.

Table 4.1b.

Program win01.cpp.

```

// continued from Table 4.1a

// The class is registered, let's create the program
hwnd = CreateWindowEx (
    0,                                // Extended possibilites for variation
    szClassName,                     // Classname
    "Window only...",                // Title Text
    WS_OVERLAPPEDWINDOW,             // default window
    CW_USEDEFAULT,                    // Windows decides the position
    CW_USEDEFAULT,                    // where the window ends up on the screen
    800,                              // The programs width  in pixels
    600,                              // The programs height in pixels
    HWND_DESKTOP,                     // The window is a child-window to desktop
    NULL,                             // No menu
    hThisInstance,                    // Program Instance handler
    NULL                              // No Window Creation data
);

// Make the window visible on the screen
ShowWindow (hwnd, nFunsterStil);

// Run the message loop.
// It will run until GetMessage() returns 0
while (GetMessage (&messages, NULL, 0, 0))
{
    // Translate virtual-key messages into character messages
    TranslateMessage(&messages);
    // Send message to Window Procedure WindowProc1234
    DispatchMessage(&messages);
}

// The program return-value is 0 -
// The value that PostQuitMessage() gave
return messages.wParam;
}

```

```

// part of global objects
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "WindowsApp"; // class name

```

WinMain. Function `int WINAPI WinMain()` contains four input parameters of different types: `HINSTANCE`, `LPSTR` and `int`. Such parameters are necessary for windows manipulating operations. Types `HINSTANCE`, `LPSTR` for `WinMain()` function and `HWND`, `UINT`, `WPARAM`, `LPARAM` for callback function are described in library `<windows.h>`.

```
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR      lpszArgument,  int      nFunsterStil)
```

Function `int WINAPI WinMain()` must solve the following tasks:

- i) determination of window class;
- ii) registration of window class;
- iii) creation of window of required class;
- iv) making the window visible on the screen;
- v) starting the loop of dispatching the messages.

Let's start from determination of window class. The first block of function `int WINAPI WinMain()` contains description of handle for window, messages to the application to be saved and data structure for the window class.

```
HWND      hwnd;      // handle for window
MSG        messages; // messages to the application are saved
WNDCLASSEX wincl;    // data structure for the windowclass
```

The second block of function `int WINAPI WinMain()` contains descriptor (DC) of this application, window class name, name of user-created function called by OS *Windows*, and several window style parameters.

```
// Window structure
wincl.hInstance      = hThisInstance; // DC of this app
wincl.lpszClassName  = szClassName;   // window class name
wincl.lpfnWndProc    = WindowProc1234; // function called by windows
wincl.style          = CS_DBLCLKS;     // Catch double-clicks
wincl.cbSize         = sizeof (WNDCLASSEX);
```

The third block of function `int WINAPI WinMain()` contains parameters of icons and mouse-pointers as standard, default icon and standard pointer for cursor.

```
// Use default icon and mouse-pointer
wincl.hIcon    = LoadIcon (NULL, IDI_APPLICATION);
wincl.hIconSm  = LoadIcon (NULL, IDI_APPLICATION);
wincl.hCursor  = LoadCursor (NULL, IDC_ARROW);
```

The fourth block of function `int WINAPI WinMain()` contains parameters depending menu organization (in that case no menu) and several routine parameters for window organization: no extra bytes after the window class and structure or the window instance. Background of the window must be coloured by selected, user-defined or default colour, and in this case window's default colour could be determined.

```
wincl.lpszMenuName = NULL;      // No menu
wincl.cbClsExtra   = 0;         // No extra bytes after the window class
wincl.cbWndExtra   = 0;         // structure or the window instance
// Windows's default colour as the background of the window
wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;
```

Determination of window class is finished, and let's start to register the window class. Registration of window class must be succesful, and if it fails quitting off the program is compulsory. This condition is realized through `if` statement.

```
// Register the window class, and if it fails quit the program
if (!RegisterClassEx (&wincl)) return 0;
```

In case of successful registration of window class, creation of program (window of required class) is allowed. Function (HWND)CreateWindowEx() is used for such purposes. Function contains 12 input parameters.

The first parameter (0) is presented for advanced users, default value 0 is enough for all routine works. Extended possibilities for variation could be used.

The second parameter (szClassName) requires window class name as a string from global area (made in C style).

The third parameter ("Window only...") represents name of default window, which name user can change according to circumstances.

The fourth parameter (WS_OVERLAPPEDWINDOW) represents style of default window: it will be presented in overlapping style to the other windows.

The fifth and the sixth parameters (CW_USEDEFAULT) represents the position of starting window. In that case, OS *Windows* decides on the initial position (x_0, y_0) of window as geometrical figure on the screen. Fig. 4.1 represents distribution of window coordinates on screen. Initial point (0,0) is presented at top left.

The seventh and the eighth parameters (800,600) represents the width and height (x and y , respectively) of windows as geometrical figure in pixels. Thus it is also possible to use macros CW_USEDEFAULT as in previous positions in order to receive the default window values.

The ninth parameter (HWND_DESKTOP) is devoted to window style determination: the window is a child-window to desktop.

The tenth parameter (NULL) shows the information that menu will be not organized (no menu).

The eleventh parameter (hThisInstance) shows, which program *Instance handler* will be used (handler described in WINAPI WinMain() function as one of four input parameters).

The twelfth parameter (NULL) says that no window creation data will be used (this parameter is useful for advanced users).

Next function ShowWindow () is devoted to window drawing: it makes the window visible on the screen. Two input parameters - descriptor hwnd from function (HWND)CreateWindowEx() and descriptor nFunsterStil as one of four input parameters of WINAPI WinMain() function must be initialized.

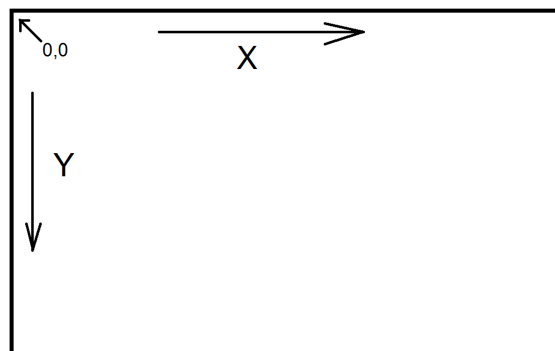


Fig. 4.1. Coordinates of window as a geometrical figure. Initial point (0,0) at top left.

```

// The class is registered, let's create the program
hwnd = CreateWindowEx (
    0,                                // Extended possibilities for variation
    szClassName,                      // Classname
    "Window only...",                 // Title Text
    WS_OVERLAPPEDWINDOW,              // default window
    CW_USEDEFAULT,                    // Windows decides the position
    CW_USEDEFAULT,                    // where the window ends up on the screen
    800,                              // The programs width in pixels
    600,                              // The programs height in pixels
    HWND_DESKTOP,                     // The window is a child-window to desktop
    NULL,                             // No menu
    hThisInstance,                    // Program Instance handler
    NULL );                           // No Window Creation data
ShowWindow (hwnd, nFunsterStil); // Make the window visible on the screen

```

Finally, starting the loop of dispatching the messages must be done. Loop `while()` contains two functions.

The first function `TranslateMessage(&messages)` is devoted to translate the virtual-key messages into character messages.

The second function `DispatchMessage(&messages)` is devoted to send messages to user-created window procedure `WindowProc1234()` (in that case).

```

// Run the message loop.
// It will run until GetMessage() returns 0
while (GetMessage (&messages, NULL, 0, 0))
{
    // Translate virtual-key messages into character messages
    TranslateMessage(&messages);
    // Send message to Window Procedure WindowProc1234
    DispatchMessage(&messages);
}

```

Program finishes, when function `PostQuitMessage()` gives return message.

```

// The program return-value is 0 -
// The value that PostQuitMessage() gives
return messages.wParam;

```

CALLBACK. Table 4.1c represents user-created callback function called `WindowProc1234()`. Due to programming task (to create window with the possibility to switch, to destroy this window and nothing else) callback function `WindowProc1234()` contains one `switch()` in loop. According to handle of messages `message` two different switch cases are presented.

The first case as a regular requesting for window destroying could be organized by user (through window managing button). In that case, handle `message` containing `WM_DESTROY` occurs. Function `PostQuitMessage (0)` realizes sending a `WM_QUIT` to the message queue.

The second case as a default case is realized through function `DefWindowProc()`.

Table 4.1c. Program win01.cpp.

```
// This function is called by the Windows function DispatchMessage()
LRESULT CALLBACK WindowProc1234 (HWND hwnd,
                                UINT message,
                                WPARAM wParam,
                                LPARAM lParam)
{
    switch (message)                // handle the messages
    {
        case WM_DESTROY:
            PostQuitMessage (0);   // send a WM_QUIT to the message queue
            break;
        default:                    // for messages that we don't deal with
            return DefWindowProc (hwnd, message, wParam, lParam);
    };
    return 0;
}
```

```
switch (message)                // handle the messages
{
    case WM_DESTROY:
        PostQuitMessage (0); // send a WM_QUIT to the message queue
        break;
    default:                    // for messages that we don't deal with
        return DefWindowProc (hwnd, message, wParam, lParam);
};
return 0;
```

Fig. 4.2 represents output of program win01.cpp. Window is presented as a rectangle with background filled with gray colour.

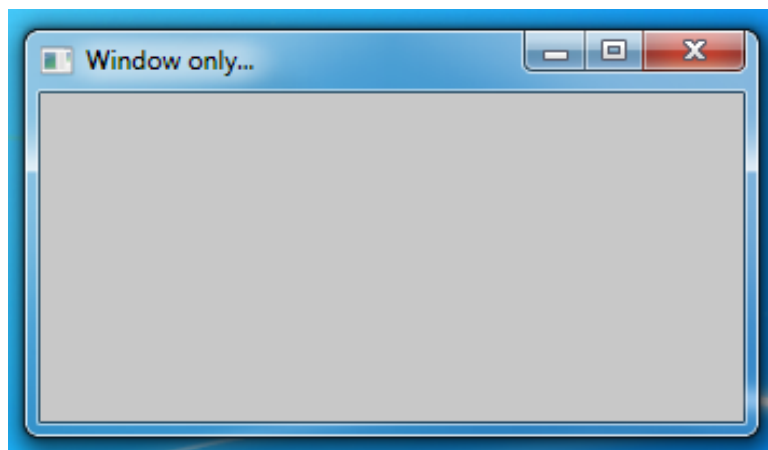


Fig. 4.2. Output of win01.cpp. Window only.

Right top angle contains three system buttons for window management: *Minimize*, *Maximize*, *Close*. Left top angle contains three system buttons built up in *Windows-95* style - *Restore*, *Move*, *Size*, *Minimize*, *Maximize*, *Close*. Such possibilities are not presented in Fig. 4.2. Window etiquette containing user-suggested phrase "Window only..." is presented on the windows top. Finishing of program and closing of window occurs by pushing red button *Close* from right top angle.

4.1.2 Dialog and manipulator operations

Dialog represents a behaviour of programming tools for exchanging management information between human and machine (computer). One of the best choices of dialog is the organization of menu buttons. Mouse plays a role of manipulator in modern computers.

2nd programming task. Programming task represents window creation, user-requested menu bar creation and realizations of mouse manipulator operations.

Tables 4.2a, 4.2b represent `int WINAPI WinMain()` function, Tables 4.2c, 4.2d represent user-created call-back function `LRESULT CALLBACK WindowProc2345()`. Table 4.3 shows corresponding header file and Table 4.4 represents resource script file.

Function `int WINAPI WinMain()` devoted for presented programming task is very similar to previous one which is presented in Tables 4.1a, 4.1b. Main code sentences and comments of code correspond to each other. Significant differences are observed below.

Additional libraries are present. In addition to `<windows.h>`, libraries for input operations `<string.h>` and `<stdio.h>` are necessary. Header file "win02.h" must be also included here (presented in Table 4.3).

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win02.h"
```

WinMain. That part of windows class parameters contains:

- i) determination of user-created call-back function `WindowProc2345()`;
- ii) determination of main menu name `MYMENU_002`.

Menu is described in resource script file as presented in Table 4.4.

```
// Windows class determination
wcl.lpfnWndProc = WindowProc2345; // function of window
wcl.lpszMenuName = "MYMENU_002"; // main menu
```

User-delimited colour of window background could be established by function `(HBRUSH)GetStockObject()`. According to user suggestions, input parameters `WHITE_BRUSH` or `BLACK_BRUSH` - prelimit white and black colour of background respectively.

```
// to fill window by white colour
wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
//wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);
```

Table 4.2a.

Program win02.cpp.

```

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win02.h"

// global objects
LRESULT CALLBACK WindowProc2345(HWND,UINT,WPARAM,LPARAM);
char szWinName[] = "MyWindow";

int WINAPI WinMain (HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR     lpzArgs,  int         nWinMode)
{
    HWND hwnd;
    MSG msg;
    WNDCLASS wcl;
    HACCEL hAccel;

    // Windows class determination
    wcl.hInstance      = hThisInst;      // descriptor of application
    wcl.lpszClassName = szWinName;      // name of windows class
    wcl.lpfnWndProc    = WindowProc2345; // function of window
    wcl.style          = 0;              // default style

    wcl.hIcon    = LoadIcon (NULL, IDI_APPLICATION); // standard icon
    wcl.hCursor  = LoadCursor (NULL, IDC_ARROW);      // standard cursor

    wcl.lpszMenuName="MYMENU_002"; // main menu
    //wcl.lpszMenuName=NULL;        // no menu
    wcl.cbClsExtra= 0; // no additional
    wcl.cbWndExtra= 0; // information

    // to fill window by white color
    wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
    //wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);

    // to register the window class
    if ( !RegisterClass (&wcl)) return 0;

    // continued to Table 4.2b

```

In case of successful registration of window class, the creation of program (window of required class) is allowed. Function (HWND)CreateWindow() is used for such purpose. Function contains eleven input parameters.

The first parameter (szWinName) requires class name, declared before WinMain() function, in global part.

Table 4.2b.

Program win02.cpp.

```

// continued from Table 4.2a

// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "MENU AND MANIPULATOR", // window header
    WS_OVERLAPPEDWINDOW, // window style
    CW_USEDEFAULT,       // X-coordinate, from Windows
    CW_USEDEFAULT,       // Y-coordinate, from Windows
    CW_USEDEFAULT,       // width, from Windows
    CW_USEDEFAULT,       // high, from Windows
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,        // use class menu
    hThisInst,           // descriptor
    NULL);               // no additional arguments

// to load accelerator
hAccel = LoadAccelerators (hThisInst,"MYMENU_002");

//to show window and redraw content
ShowWindow (hwnd,nWinMode);
UpdateWindow(hwnd);

// cycle of annonses
while ( GetMessage (&msg, NULL,0,0))
{
    if (!TranslateAccelerator (hwnd,hAccel,&msg))
    {
        TranslateMessage (&msg); // to use keyboard
        DispatchMessage (&msg); // to return direction from Windows
    }
};
return msg.wParam;
}

```

The second parameter ("MENU AND MANIPULATOR") represents string of window header. This string will be presented on window top as an window etiquette.

All the other parameters are commented in the previous subsection, for function `CreateWindowEx()`.

```

// to create window
hwnd = CreateWindow (
    szWinName,           // class name
    "MENU AND MANIPULATOR", // window header
    WS_OVERLAPPEDWINDOW, // window style
    CW_USEDEFAULT,       // X-coordinate, from Windows

```

```

CW_USEDEFAULT,          // Y-coordinate, from Windows
CW_USEDEFAULT,          // width, from Windows
CW_USEDEFAULT,          // high, from Windows
HWND_DESKTOP,           // no additional window
(HMENU) NULL,           // use class menu
hThisInst,              // descriptor
NULL);                  // no additional arguments

```

Table 4.2c.

Program win02.cpp.

```

LRESULT CALLBACK WindowProc2345 (HWND    hwnd,
                                UINT     message,
                                WPARAM   wParam,
                                LPARAM   lParam)
{
    int response;
    switch (message)
    {
    case WM_COMMAND: //
        switch ( LOWORD (wParam))
        {
            case IDM_ALPHA:  MessageBox (hwnd, "Alpha",  "",    MB_OK);
                                break;
            case IDM_BETA:   MessageBox (hwnd, "Beta",   "",    MB_OK);
                                break;
            case IDM_GAMMA:  MessageBox (hwnd, "Gamma",  "",    MB_OK);
                                break;
            case IDM_EPSILON: MessageBox (hwnd, "Epsilon", "",    MB_OK);
                                break;
            case IDM_ZETA:   MessageBox (hwnd, "Zeta",   "",    MB_OK);
                                break;
            case IDM_ETA:    MessageBox (hwnd, "Eta",    "",    MB_OK);
                                break;
            case IDM_THETA:  MessageBox (hwnd, "Theta",  "",    MB_OK);
                                break;
            case IDM_HELP:   MessageBox (hwnd, "No Help", "Help", MB_OK);
                                break;
        }
    };
    break;

    // continued to Table 4.2d

```

Table 4.2d.

Program win02.cpp.

```

// continued from Table 4.2c

case WM_RBUTTONDOWN: // right button of mouse
    response = MessageBox (hwnd, "Select action:",
                           "Right button", MB_ABORTRETRYIGNORE);
    switch (response)
    {
        case IDABORT:  MessageBox (hwnd, "", "Abort",  MB_OK); break;
        case IDRETRY:  MessageBox (hwnd, "", "Retry",  MB_OK); break;
        case IDIGNORE: MessageBox (hwnd, "", "Ignore", MB_OK); break;
    };

case WM_LBUTTONDOWN: // left button of mouse
    response = MessageBox (hwnd, "Continue ?",
                           "Left button", MB_ICONHAND | MB_YESNO);
    switch (response)
    {
        case IDYES:    MessageBox (hwnd, "Press", "YES", MB_OK); break;
        case IDNO:     MessageBox (hwnd, "Press", "NO",  MB_OK); break;
    };

case WM_DESTROY: // to end program
    PostQuitMessage (0); break;
default:
    return DefWindowProc (hwnd, message, wParam, lParam);
}
return 0;
}

```

Additional possibilities of manipulating could be received from keyboard. Function `LoadAccelerators()` is presented for such purpose.

This function requires two input parameters - descriptor of application `hThisInst` and menu name "MYMENU_002". Menu is presented in resource script file, where keyboard buttons are associated with the corresponding identifiers, see Table 4.4.

```

// to load accelerator
hAccel = LoadAccelerators (hThisInst,"MYMENU_002");

```

CALLBACK. Call-back function `LRESULT CALLBACK WindowProc2345()` (see Tables 4.2c, 4.2d) contains the main programming code according to user request. Managing of program could be done by several different ways: by virtual machine command, by pushing the button of mouse manipulator (right as well as left) or using using system window button *Close* for virtual machine destroying. All such cases are presented in `switch()` structure according to the input parameter `message` request.

```

switch (message)
{
    case WM_COMMAND:    {...}
    case WM_RBUTTONDOWN:{...}    // right button of mouse
    case WM_LBUTTONDOWN:{...}    // left button of mouse
    case WM_DESTROY:    {...}    // to end program
    default:
}

```

Let's go to the first case `case WM_COMMAND:` - virtual machine command. According to the user request, this part contains eight menu button positions such as: `IDM_ALPHA`, `IDM_BETA`, ..., `IDM_THETA`, `IDM_HELP`. In that case, selecting of each position is confirmed by creation of message box with corresponding etiquette. All presented identifiers must be described in header file "win02.h" (Table 4.3) which is included into the main program file (Table 4.2a) as well as in resource script file (Table 4.4).

Function `MessageBox()` requires four input parameters: i) handle of windows `hwnd`; ii) middle and top etiquettes to message box - as the strings of C type, for example "No Help" and "Help"; iii) managing button, in that case master button `MB_OK`.

User-requested programming code could be placed between `case` and `break` operators.

```

switch ( LOWORD (wParam))
{
    case IDM_ALPHA:    MessageBox (hwnd, "Alpha",    "",    MB_OK); break;
    case IDM_BETA:    MessageBox (hwnd, "Beta",    "",    MB_OK); break;
    case IDM_GAMMA:    MessageBox (hwnd, "Gamma",    "",    MB_OK); break;
    case IDM_EPSILON:  MessageBox (hwnd, "Epsilon",    "",    MB_OK); break;
    case IDM_ZETA:    MessageBox (hwnd, "Zeta",    "",    MB_OK); break;
    case IDM_ETA:    MessageBox (hwnd, "Eta",    "",    MB_OK); break;
    case IDM_THETA:    MessageBox (hwnd, "Theta",    "",    MB_OK); break;
    case IDM_HELP:    MessageBox (hwnd, "No Help", "Help", MB_OK); break;
    // case IDM_XXX:    /* any user-created code */    break;
};
break;

```

Let's go to the second case `case WM_RBUTTONDOWN` which corresponds to the manipulation by right button of mouse. First of all, it is necessary to select manipulation type. For such purposes, function `MessageBox()` through master button group `MB_ABORTRETRYIGNORE` offers the selection of three positions "Abort", "Retry", "Ignore". After that, selecting of each position by click of mouse button - `IDABORT`, `IDRETRY`, `IDIGNORE` - is confirmed by creation of message box with corresponding etiquette. User-requested programming code could be placed between `case` and `break` operators.

```

int response = MessageBox (hwnd, "Select action:", "Right button",
                           MB_ABORTRETRYIGNORE);

switch (response)
{
    case IDABORT:  MessageBox (hwnd, "", "Abort", MB_OK); break;
    case IDRETRY:  MessageBox (hwnd, "", "Retry", MB_OK); break;
    case IDIGNORE: MessageBox (hwnd, "", "Ignore", MB_OK); break;
};

```

Let's go to the third case `case WM_LBUTTONDOWN` which corresponds to the manipulation by left button of mouse. First of all, it is necessary to select manipulation type. For such purposes, function `MessageBox()` through master button group `MB_ICONHAND | MB_YESNO` offers the selection of position "Left button". After that, selecting of each position by click of mouse button `-IDYES`, `IDNO` - is confirmed by creation of message box with corresponding etiquette. User-requested programming code could be placed between `case` and `break` operators.

```
int response = MessageBox (hwnd, "Continue ?", "Left button",
                           MB_ICONHAND | MB_YESNO);

switch (response)
{
    case IDYES:    MessageBox (hwnd, "Press", "YES", MB_OK); break;
    case IDNO:     MessageBox (hwnd, "Press", "NO",  MB_OK); break;
};
```

Let's go to the fourth case `case WM_DESTROY`. Regular requesting for window destroying could be organized by user (by pushing window managing button). In that case, handle message containing `WM_DESTROY` occurs, and function `PostQuitMessage(0)` realizes the sending a `WM_QUIT` to the message queue.

Let's go to the fifth case `case default`. Default case is realized through function `DefWindowProc()`.

HEADER FILE. Table 4.3 represents the header file `win02.h` which must be included into `win02.cpp` and `win02.rc` files. Content of header file represents an association of `case` identifiers in `WM_COMMAND` expression only to the integer numbers through operator `define`. Integer number could be chosen freely, but it must be unique for each identifier.

```
#define IDM_ALPHA    100
```

Table 4.3. Header file.	Program win02.h.
<pre>// win02.h // #define IDM_ALPHA 100 #define IDM_BETA 101 #define IDM_GAMMA 102 #define IDM_DELTA 103 #define IDM_EPSILON 104 #define IDM_ZETA 105 #define IDM_ETA 106 #define IDM_THETA 107 #define IDM_HELP 108</pre>	

RC FILE. Table 4.4 represents resource script (RC) file win02.rc which is responsible for program management organization: menu buttons and accelerators from keyboard. Resource script (RC) file is written using script language, but not C/C++ language. First of all, this file contains two `include` statements pointed to library `<windows.h>` and header file `"win02.h"`.

```
#include "windows.h"
#include "win02.h"
```

The second part of resource script file contains description of menu buttons. Statement containing menu name `MYMENU_002` is responsible for menu recognising in `WinMain()` function.

```
MYMENU_002 MENU DISCARDABLE
```

After that, distribution of menu items is presented. Each menu item must be associated with the identifier, previously described in header file. After that, this menu item could be used in call-back function. For example, menu items `ALPHA` and `BETA` could be grouped into pop-up item `FIRST`.

```
POPUP "&FIRST"
BEGIN
    MENUITEM "&ALPHA\tF2",          IDM_ALPHA
    MENUITEM "&BETA\tF3",          IDM_BETA
END
```

Also, pop-up item `SECOND` could be organized as collection of menu item `GAMMA` and second level pop-up item `DELTA` which contains menu item `EPSILON` and menu item `ZETA`

```
POPUP "&SECOND"
BEGIN
    MENUITEM "&GAMMA\tShift+G",      IDM_GAMMA
    POPUP "&DELTA"
    BEGIN
        MENUITEM "&EPSILON\tCtrl+E",  IDM_EPSILON
        MENUITEM "&ZETA\tCtrl+Z",    IDM_ZETA
    END
END
```

The third part of resource script (RC) file contains description of accelerators. Statement containing menu name `MYMENU_002` is responsible for accelerators recognising in `WinMain()` function.

```
MYMENU_002    ACCELERATORS MOVEABLE PURE
```

Keyboard accelerators are presented using definition of three type: virtual key notation `VK_F2`, ASCII notation `"G"` or `"Ê"` (one key or `CTRL+E`, respectively). Semantics of such type associations is very strong: virtual key could be associated to the identifier, previously described in header file. After that, this accelerator could be used in call-back function.

```
VK_F2,          IDM_ALPHA,          VIRTKEY
"G",            IDM_GAMMA,          ASCII
"^E",           IDM_EPSILON,        ASCII
```

Figs. 4.3÷4.5 represent the output of win02 program. New window is created and menu containing two pop-up items is organized.

Table 4.4. RC file.

Program win02.rc.

```

#include "windows.h"
#include "win02.h"

// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

// Menu
MYMENU_002 MENU DISCARDABLE
BEGIN
    POPUP "&FIRST"
    BEGIN
        MENUITEM "&ALPHA\tF2",          IDM_ALPHA
        MENUITEM "&BETA\tF3",          IDM_BETA
    END
    POPUP "&SECOND"
    BEGIN
        MENUITEM "&GAMMA\tShift+G",      IDM_GAMMA
        POPUP "&DELTA"
        BEGIN
            MENUITEM "&EPSILON\tCtrl+E",  IDM_EPSILON
            MENUITEM "&ZETA\tCtrl+Z",      IDM_ZETA
        END
        MENUITEM "&ETA\tCtrl+F4",        IDM_ETA
        MENUITEM "&THETA\tF5",          IDM_THETA
    END
    MENUITEM "&HELP",                  IDM_HELP
END

// Accelerator
MYMENU_002 ACCELERATORS MOVEABLE PURE
BEGIN
    VK_F2,          IDM_ALPHA,          VIRTKEY
    VK_F3,          IDM_BETA,           VIRTKEY
    "G",            IDM_GAMMA,          ASCII
    "^E",           IDM_EPSILON,        ASCII
    "^Z",           IDM_ZETA,           ASCII
    VK_F4,          IDM_ETA,            VIRTKEY, CONTROL
    VK_F5,          IDM_THETA,          VIRTKEY
    VK_F1,          IDM_HELP,           VIRTKEY
END

#endif // English (U.S.) resources

```

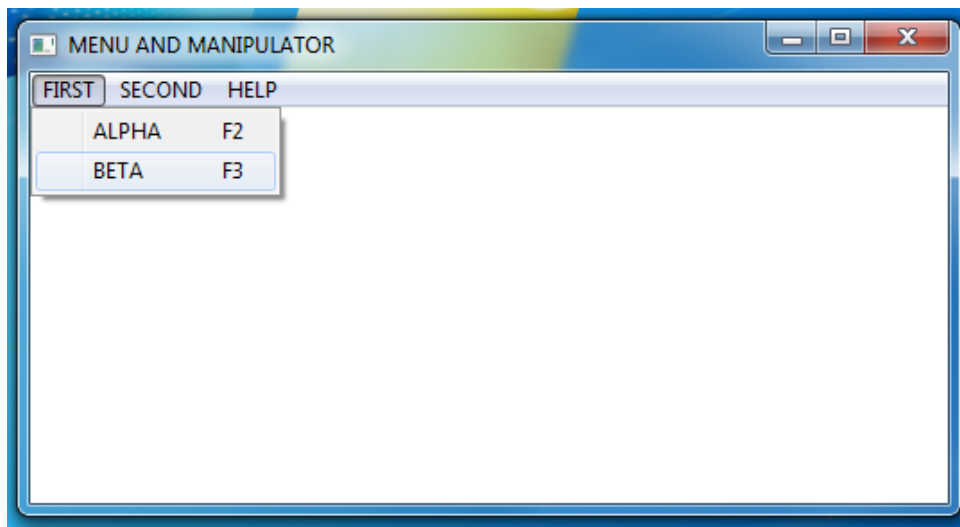


Fig. 4.3. Output of win02. Menu in window.

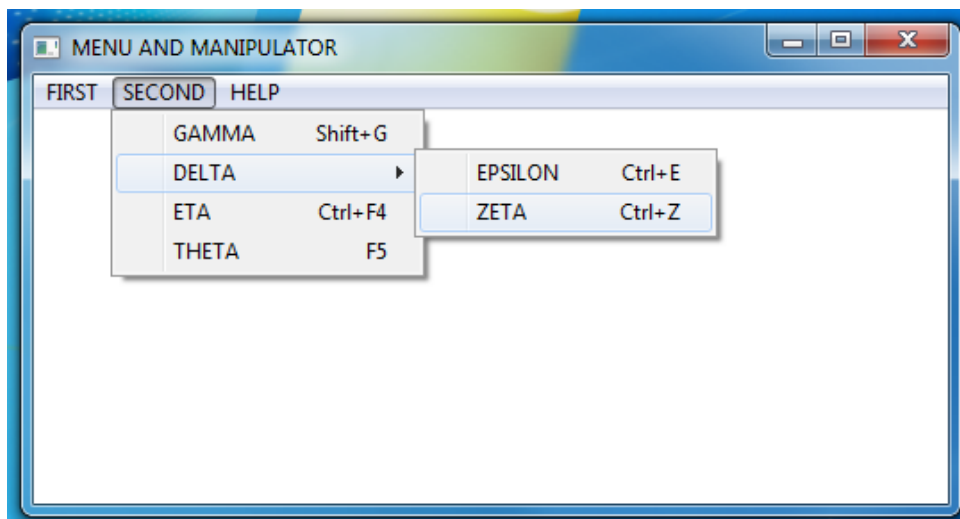


Fig. 4.4. Output of win02. Menu in window.

In Fig. 4.3, pop-up item **FIRST** is activated using mouse manipulator. According to the menu description in resource script file, two menu items **ALPHA** and **BETA** are present. Accelerators are presented on button etiquettes.

In Fig. 4.4, pop-up item **SECOND** is activated using mouse manipulator, and pop-up item of the second level **DELTA** is also activated. According to the menu description in resource script file, two columns of menu items are present. Accelerators are presented on button etiquettes.

In Fig. 4.5, right button of mouse manipulator is pushed, and corresponding message box occurs. This box contains the information for following selection of the next step: *Abort*, *Retry*, *Ignore*.

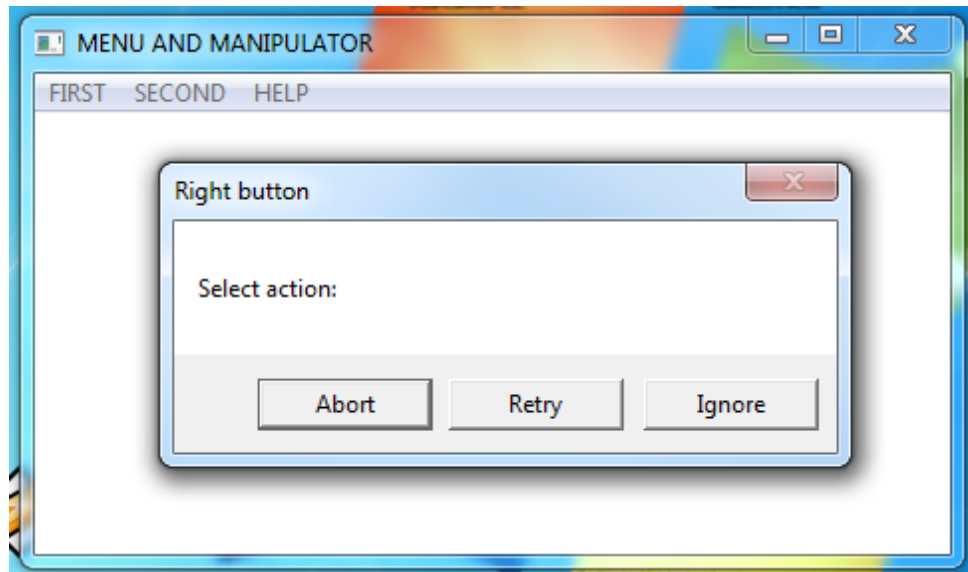


Fig. 4.5. Output of win02. Menu in window.

4.2 Figure drawing

C Code.
 C code run.
 Run code run...
 PLEASE !!!

Barbara Tongue

3rd programming task. Programming task represents window creation, user-requested menu bar creation and realization of several primitive drawings on screen such as a line, a rectangle, an ellipse and a pie. Tables 4.5a, 4.5b represent `int WINAPI WinMain ()` function, Tables 4.5c, 4.5d represent call-back function `LRESULT CALLBACK WindowProc3456()`. Table 4.6 represents corresponding header file, and Table 4.7 shows resource script file. Figs. 4.6÷4.8 show outputs of program.

WinMain. Function `int WINAPI WinMain()` prepared according to the presented programming task is very similar to the previous one, presented in Table 4.1a as well as Table 4.2a. Main code sentences and comments of code corresponds each to other. Negligible differences are observed below.

Additional library functions are necessary. In addition to `<windows.h>`, several libraries for input operations: `<string.h>` and `<stdio.h>` are necessary. Header file "win03.h" must be included here (presented in Table 4.6).

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win03.h"
```

Table 4.5a.

Program win03.cpp.

```

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win03.h"

// global objects
LRESULT CALLBACK WindowProc3456(HWND,UINT,WPARAM,LPARAM);
char szWinName[] = "MyWindow"; // window class
int iXmax, iYmax;               // window width and hight

HDC      memdc;                 // DC of virtual window
HBITMAP  hbit;                  // rastr in virtual window
HBRUSH   hbrush, hOldbrush;     // DC of pen
HPEN     hOldpen;               // DC of old pen
HPEN     hRedpen, hGreenpen, hBluepen, // DC of color pen
         hYellowpen, hCyanopen,hMargentapen;

int WINAPI WinMain (HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR      lpszArgs,  int      nWinMode)
{
    HWND hwnd;  MSG msg;  WNDCLASS wcl;  HACCEL hAccel;

    // Windows class determination
    wcl.hInstance      = hThisInst;      // DC of application
    wcl.lpszClassName  = szWinName;      // windows class
    wcl.lpfnWndProc    = WindowProc3456; // function of window
    wcl.style          = 0;              // default style

    wcl.hIcon    = LoadIcon (NULL, IDI_APPLICATION); // standard icon
    wcl.hCursor  = LoadCursor (NULL, IDC_ARROW);      // standard cursor
    wcl.lpszMenuName  = "MYMENU_003";                 // main menu
    //wcl.lpszMenuName = NULL;                         // no menu
    wcl.cbClsExtra = 0; // no additional
    wcl.cbWndExtra = 0; // information

    // to fill window by color
    //wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
    wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);

    // to register the window class
    if ( !RegisterClass (&wcl)) return 0;

    // continued to Table 4.5b

```

Table 4.5b.

Program win03.cpp.

```

// continued from Table 4.5a

// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "FIGURES",           // header
    WS_OVERLAPPEDWINDOW, // window style
    0,    // CW_USEDEFAULT, // X-0-coordinate
    0,    // CW_USEDEFAULT, // Y-0-coordinate
    1200, // CW_USEDEFAULT, // width
    800,  // CW_USEDEFAULT, // hight
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,         // use class menu
    hThisInst,           // descriptor
    NULL);               // no additional arguments

// to show window and redraw content
ShowWindow (hwnd, nWinMode);
UpdateWindow (hwnd);

// cycle of annonses
while ( GetMessage (&msg, NULL,0,0))
{
    if (!TranslateAccelerator (hwnd,hAccel,&msg))
    {
        TranslateMessage (&msg); // to use keyboard
        DispatchMessage (&msg); // to return direction from Windows
    }
}
return msg.wParam;
}

```

Block of global objects contains the following parts: declaration and initialization of window class, declaration of integer variables devoted for visualization on screen: window width and height, declaration of descriptor (DC) of virtual window, bitmap in virtual window, DC of pen, DC of old pen, and six DCs of colour pen. All classes named HDC, HBITMAP, HBRUSH, HPEN are described in library <windows.h>.

```

char szWinName[] = "MyWindow";           // window class
int iXmax, iYmax;                        // window width and hight
HDC memdc;                               // DC of virtual window
HBITMAP hbit;                             // bitmap in virtual window
HBRUSH hbrush, hOldbrush;                 // DC of pen
HPEN hOldpen;                             // DC of old pen
HPEN hRedpen, hGreenpen, hBluepen,        // DCs of color pens
    hYellowpen, hCyanopen, hMargentapen;

```

Part of windows class parameters contains determination of user-created call-back function `WindowProc3456()` and determination of main menu name `MYMENU_003`. Menu is described in resource script file as presented in Table 4.7.

```
// Windows class determination
wcl.lpfWndProc   = WindowProc3456; // function of window
wcl.lpszMenuName = "MYMENU_003";  // main menu
```

User-delimited colour of window background could be established by function `(HBRUSH)GetStockObject()`. According to user suggestions, input parameters `WHITE_BRUSH` or `BLACK_BRUSH` predefine white and black colour respectively.

```
// to fill window by colour
// wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
wcl.hbrBackground   = (HBRUSH)GetStockObject (BLACK_BRUSH);
```

In case of successful registration of window class, the creation of window of required class is allowed. Function `(HWND)CreateWindow()` was used for such purpose. Function contains eleven input parameters.

The first parameter (`szWinName`) requires class name. This parameter is declared before `WinMain()` function, in global part.

The second parameter ("`FIGURES`") represents string of window header. This string will be presented on window top as an etiquette.

The third parameter (`WS_OVERLAPPEDWINDOW`) represents style of default window: it will be presented in overlapping style to the other windows.

The fourth and the fifth parameters (`0,0`) represent the position of starting window as the geometrical figure on the screen. In that case, user-requested parameters of window will be used. According to scheme presented in Fig. 4.1, left top angle of window starts at position (`0,0`) of screen.

The sixth and the seventh parameters (`1200,800`) represent the width and height (x and y respectively) of windows as geometrical figure (in pixels).

Generally, it is not necessary to use an absolute values, because some risk to overload the real screen exist. In many cases, it is preferable to use macros `CW_USEDEFAULT` for fourth, fifth, sixth and seventh positions in order to receive the default window values.

All the other parameters are commented in the previous subsection, for function `CreateWindowEx()`.

```
// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "FIGURES",           // header
    WS_OVERLAPPEDWINDOW, // window style
    0,                   // CW_USEDEFAULT, // X-0-coordinate
    0,                   // CW_USEDEFAULT, // Y-0-coordinate
    1200,                // CW_USEDEFAULT, // width
    800,                 // CW_USEDEFAULT, // height
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,        // use class menu
    hThisInst,           // descriptor
    NULL);               // no additional arguments
```

CALLBACK. Call-back function `LRESULT CALLBACK WindowProc3456()` (see Tables 4.5c, 4.5d) contains the main program code. Managing of program could be done using several different ways: by virtual machine command, or using system window button *Close* for virtual machine destroying.

```
switch (message)
{
    case WM_CREATE: // to prepare tools
    case WM_DESTROY: // to end program
    case WM_PAINT: // to redraw window
    case WM_COMMAND: //
        switch ( LOWORD (wParam))
        {
            case IDM_01: // to draw pixel
            case IDM_02: // to draw line
            case IDM_03: // to draw ellipse
            case IDM_04: // to draw rectangle
            case IDM_05: // to clear
            case IDM_HELP: // to show comments
        }
    default:
}
```

All such cases are presented in `switch()` statement according to the input parameter `message` request. Several systemic cases represent preparation of tools (`WM_CREATE`), destruction of tools `WM_DESTROY` and usage of tools `WM_PAINT`. Several user-defined cases (through `WM_COMMAND`) represent drawing of the geometric primitives: pixel, line, ellipse, rectangle as well as window clearing and output of comments (traditional help).

Code of function `LRESULT CALLBACK WindowProc3456()` is very long compared to the previous `WindowProc1234()` function. Several code blocks belonging to the function `WindowProc3456()` are presented in partitioned form of included files (initial file divided into six parts) for the reader's convenience. It is difficult to present the whole file of quite large amount in the plain user-readable form.

```
#include "win03_create.h"
```

All cases will be described separately.

WM_CREATE. Code block, devoted to `case WM_CREATE`, is realized in file "win03_create.h" - see Table 4.5e.

Function `GetSystemMetrics()` is used in order to get window screen parameters `iXmax` and `iYmax` previously described in global section.

```
iXmax = GetSystemMetrics (SM_CXSCREEN);
iYmax = GetSystemMetrics (SM_CYSCREEN);
```

Three functions such as `CreateCompatibleBitmap()`, `GetDC()` and also `CreateCompatibleDC()` are used for receiving descriptors which will be used in operation to create bitmap in windows. `SelectObject()` function selects an object into the specified device context (according to the descriptor `hbit`).

Table 4.5c.

Program win03.cpp.

```

LRESULT CALLBACK WindowProc3456 (HWND  hwnd,      UINT  message,
                                WPARAM wParam,  LPARAM lParam)
{
    HDC hdc; PAINTSTRUCT paintstruct; int response;
    switch (message)
    {
        case WM_CREATE:
            #include "win03_create.h"
            break;

        case WM_DESTROY: // to end program
            #include "win03_destroy.h"
            break;

        case WM_PAINT: // to redraw window
            hdc = BeginPaint (hwnd, &paintstruct);    // to get DC
            BitBlt (hdc, 0, 0, iXmax, iYmax, memdc, 0, 0, SRCCOPY);
            EndPaint (hwnd, &paintstruct);            // to free DC
            break;

        case WM_COMMAND: //
            switch ( LOWORD (wParam))
            {
                case IDM_01: // pixel
                    #include "win03_command_01.h"
                    break;

                case IDM_02: // line
                    #include "win03_command_02.h"
                    break;

                case IDM_03: // ellipse
                    #include "win03_command_03.h"
                    break;

                case IDM_04: // rectangle
                    #include "win03_command_04.h"
                    break;

                case IDM_05:
                    MessageBox (hwnd, "", "Clear", MB_OK);
                    PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
                    InvalidateRect (hwnd, NULL, 1);
                    break;

                // continued to Table 4.5d
            }
    }
}

```

Table 4.5d.

Program win03.cpp.

```

        // continued from Table 4.5c

        case IDM_HELP:
            char *cc;
            cc="Pixels, Lines, Ellipses, Rectanges.\n\nEnjoy Window graphics!";
            MessageBox (hwnd,cc, "Help", MB_OK);
            break;
        };
        break;
    default:
        return DefWindowProc (hwnd, message, wParam, lParam);
    }
    return 0;
}

```

Table 4.5e.

Program win03_create.h.

```

// get window screen parameters:
iXmax = GetSystemMetrics (SM_CXSCREEN);
iYmax = GetSystemMetrics (SM_CYSCREEN);

// to create bitmap in windows
hdc  = GetDC (hwnd);
memdc = CreateCompatibleDC (hdc);
hbit  = CreateCompatibleBitmap (hdc, iXmax, iYmax);
SelectObject (memdc, hbit);

hbrush = (HBRUSH)GetStockObject (WHITE_BRUSH);
SelectObject (memdc, hbrush);
PatBlt (memdc,0,0,iXmax, iYmax,PATCOPY);

hRedpen      = CreatePen(PS_SOLID, 1, RGB(0xff, 0, 0));
hGreenpen    = CreatePen(PS_SOLID, 2, RGB( 0, 0xff, 0));
hBluepen     = CreatePen(PS_SOLID, 3, RGB( 0, 0, 0xff));
hYellowpen   = CreatePen(PS_SOLID, 4, RGB(0xff, 0xff, 0));
hCyanopen    = CreatePen(PS_SOLID, 5, RGB( 0, 0xff, 0xff));
hMargentapen = CreatePen(PS_SOLID, 6, RGB(0xff, 0, 0xff));

// too keep default pen
hOldpen = (HPEN)SelectObject (memdc, hYellowpen);
SelectObject (memdc, hOldpen);
ReleaseDC (hwnd, hdc);

```

Table 4.5f. RGB coding.

Colour	Red	Green	Blue	Yellow	Cyano	Margenta
RED	0xff	0	0	0xff	0	0xff
GREEN	0	0xff	0	0xff	0xff	0
BLUE	0	0	0xff	0	0xff	0xff

```
// to create bitmap in windows
hdc      = GetDC (hwnd);
memdc    = CreateCompatibleDC (hdc);
hbit     = CreateCompatibleBitmap (hdc, iXmax, iYmax);
SelectObject (memdc, hbit);
```

Function `GetStockObject()` retrieves a handle to one of the stock pens, brushes, fonts, or palettes (in that case - brush). `SelectObject()` function selects an object into the specified device context (according to the descriptor `hbrush`). Function `PatBlt()` paints the specified rectangle using the brush that is currently selected into the specified device context.

It means that the screen part will be coloured in white, or in other words so called screen initialization will be done. Screen part is limited by rectangle (left top angle at (0,0) position and right bottom angle at (iXmax, iYmax) position.

```
hbrush = (HBRUSH)GetStockObject (WHITE_BRUSH);
SelectObject (memdc, hbrush);
PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
```

Now is necessary to prepare the tools. All pens (`hRedpen`, `hGreenpen`,...), previously declared in global section, must be associated to the colour parameters. Function `CreatePen()` creates a logical pen that has the specified style (`PS_SOLID`), width(1..6), and colour (macros of RGB). The pen can subsequently be selected into a device context and used to draw lines and curves. Table 4.5f represents coding schema for several RGB colours: three pure and three mixed. Value `0xff` in hexadecimal format corresponds to number 255 in usual decimal format. Positioning of tree components in values from interval (0÷255) allows to reveice full palette pure and mixed colours.

```
hRedpen      = CreatePen(PS_SOLID, 1, RGB(0xff, 0, 0));
hGreenpen    = CreatePen(PS_SOLID, 2, RGB( 0, 0xff, 0));
hBluepen     = CreatePen(PS_SOLID, 3, RGB( 0, 0, 0xff));
hYellowpen   = CreatePen(PS_SOLID, 4, RGB(0xff, 0xff, 0));
hCyanopen    = CreatePen(PS_SOLID, 5, RGB( 0, 0xff, 0xff));
hMargentapen = CreatePen(PS_SOLID, 6, RGB(0xff, 0, 0xff));
```

After selecting the new tools it is always necessary to keep old tools. In that case, `hOldpen` is associated with the yellow pen. `SelectObject()` function selects an object into the specified device context (according to the descriptor `hYellowpen`). `ReleaseDC()` function releases a device context (according to DC), freeing it for use by other applications. It frees only common and window DCs.

```
// too keep default pen
hOldpen = (HPEN)SelectObject (memdc, hYellowpen);
SelectObject (memdc, hOldpen);
ReleaseDC (hwnd, hdc);
```

Table 4.5g.

Program win03_destroy.h.

```

DeleteObject (hRedpen);
DeleteObject (hGreenpen);
DeleteObject (hBluepen);
DeleteObject (hYellowpen);
DeleteObject (hCyanopen);
DeleteObject (hMargentapen);
DeleteDC (memdc);
PostQuitMessage (0);

```

WM_DESTROY. Code block, devoted for case `WM_DESTROY`, is realized in file "win03_destroy.h" - Table 4.5g. All previously created objects from descriptors `hRedpen`, `hGreenpen`, `hBluepen`, `hYellowpen`, `hCyanopen`, `hMargentapen`, `memdc` must be destroyed, using function `DeleteObject()`. Function `PostQuitMessage(0)` indicates to the OS that a thread has made a request to terminate (quit). It is typically used in response to a `WM_DESTROY` message.

WM_PAINT. Code block, devoted for case `WM_PAINT` is aimed to redraw window - see Table 4.5c. Three functions are used.

`BeginPaint()` function prepares the specified window for painting and fills a `PAINTSTRUCT` structure with information about the painting. This function requires two input parameters: (`hwnd`, handle to the window to be repainted; `&paintstruct`, pointer to the `PAINTSTRUCT` structure that will receive painting information). If the function succeeds, the return value is the handle (DC) to a display device context for the specified window.

```
hdc = BeginPaint (hwnd, &paintstruct);    // to get DC
```

`BitBlt()` function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context. Function requires nine input parameters.

The first parameter - `hwnd`, handle to the destination device context.

The second parameter - 0, x -coordinate of the upper-left corner of the destination rectangle.

The third parameter - 0, y -coordinate of the upper-left corner of the destination rectangle.

The fourth parameter - `iXmax`, width of the source and destination rectangles.

The fifth parameter - `iYmax`, height of the source and the destination rectangles.

The sixth parameter - `memdc`, handle to the source device context.

The seventh parameter - 0, x -coordinate of the upper-left corner of the source rectangle.

The eighth parameter - 0, y -coordinate, in logical units, of the upper-left corner of the source rectangle.

The ninth parameter - `SRCCOPY`, a raster-operation code. These codes define how the colour data for the source rectangle will be combined with the colour data for the destination rectangle to achieve the final colour.

Table 4.5h.

Program win03_command_01.h.

```

MessageBox (hwnd,"","Pixels",MB_OK);
for (int i=0; i<=200; i++)
{
    SetPixel (
        memdc,    // handle to the device context
        40,       // x-coord. of the point to be set
        i,        // y-coord. of the point to be set
        RGB( 0,   0, 0xff)); //blue

        SetPixel (memdc, 50, i, RGB( 0, 0xff, 0)); //green
        SetPixel (memdc, 60, i, RGB( 0, 0xff, 0xff)); //cyano
};
InvalidateRect (hwnd, NULL, 1);

```

```

BitBlt (hdc, 0, 0, iXmax, iYmax, memdc, 0, 0, SRCCOPY);

```

EndPaint() function marks the end of painting in the specified window. This function is required for each call to the **BeginPaint()** function, but only after painting is completed. This function requires two input parameters: **hwnd**, handle to the window to be repainted; **&paintstruct**, pointer to the **PAINTSTRUCT** structure that will receive painting information.

```

EndPaint (hwnd, &paintstruct);    // to free DC

```

IDM_01 - PIXEL. `case WM_COMMAND` part represents user-created programing code, which could be used by selecting several `case` statements. Code block, devoted for `case IDM_01`, is realized in file "win03_command_01.h" (Table 4.5h). Pixeling operation is provided in loop `for()` in order to receive *y*-coordinate of the point to be set in interval $(0 \div 200)$.

```

for (int i=0; i<=200; i++)
{
    SetPixel (
        memdc,    // handle to the device context
        40,       // x-coordinate of the point to be set
        i,        // y-coordinate of the point to be set
        RGB(0,0,0xff)); //blue
};

```

The **SetPixel()** function sets the pixel at the specified coordinates (x,y) to the specified colour. Function requires four input parameters: **memdc**, handle to the device context; **40**, *x*-coordinate of the point to be set; **i**, *y*-coordinate of the point to be set; the colour to be used to paint the point. In order to create a colour value, **RGB** macro was used. Colour request according to RGB value is presented in the Table 4.5f.

If the function succeeds, the return value is the RGB value that the function sets the pixel to. This value may differ from the color specified by user; that occurs when an exact match for the specified colour cannot be found. If the function fails, the return value is equal to (-1).

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. `1`, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Pixels* vertical lines formed from pixels are presented on the left.

IDM_02 - LINE. Code block, devoted for `case IDM_02`, is implemented in file "`win03.command_02.h`" - Table 4.5i. Line drawing (not using pixel context) is realized as a sequence of three operations: i) selecting of tool (pen), generally is possible to use default pen, (function `SelectObject()`; ii) establishing the initial position (function `MoveToEx()`) and iii) lining to final position (function `LineTo()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hGreenpen`).

```
SelectObject (memdc, hGreenpen);
```

Function `MoveToEx()` updates the current position to the specified point and optionally returns the previous position. This function requires three parameters: `memdc`, handle to a device context; `200`, x-coordinate of the new position; `200`, y-coordinate of the new position; `NULL`, pointer to a `POINT` structure; for `NULL` pointer, the previous position is not returned.

```
MoveToEx (memdc, 200, 200, NULL);
```

Function `LineTo()` draws a line from the current position up to but not including the specified point. This function requires three parameters: `memdc`, handle to a device context; `520`, x-coordinate of the line's ending point; `220`, y-coordinate of the line's ending point;

```
LineTo      (memdc, 520, 220);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Lines*, corresponding graphical primitives are presented: six lines as a horizontal lines at the center are drawn on different width and colour according to scheme presented in case `case WM_CREATE`.

IDM_03 - ELLIPSE. Code block, devoted for `case IDM_03`, is implemented in file "`win03.command_03.h`" - Table 4.5j. Ellipse and pie as a geometrical figures will be drawn using as a sequence of two operations:

- i) selecting of tool (pen) or default pen; (function `SelectObject ()`);
- ii) drawing ellipse (function `Ellipse ()`) or drawing cut of ellipse (pie) (function `Pie ()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hBluepen`).

```
SelectObject (memdc, hBluepen);
```

Table 4.5i.

Program win03_command_02.h.

```

MessageBox (hwnd, "", "Lines", MB_OK);
MoveToEx (memdc, 200, 200, NULL);

LineTo (
    memdc, // handle to the device context
    500,   // x-coord. of the line's ending point
    200); // y-coord. of the line's ending point
MoveToEx (
    memdc, // handle to the device context
    220,   // x-coord. of the new position
    220,   // y-coord. of the new position
    NULL); // previous position is not returned

//green
SelectObject (memdc, hGreenpen);
LineTo      (memdc, 520, 220);
MoveToEx    (memdc, 240, 240, NULL);
//red
SelectObject (memdc, hRedpen);
LineTo      (memdc, 540, 240);
MoveToEx    (memdc, 260, 260, NULL);
//blue
SelectObject (memdc, hBluepen);
LineTo      (memdc, 560, 260);
MoveToEx    (memdc, 280, 280, NULL);
//yellow
SelectObject (memdc, hYellowpen);
LineTo      (memdc, 580, 280);
MoveToEx    (memdc, 300, 300, NULL);
//cyano
SelectObject (memdc, hCyanopen);
LineTo      (memdc, 600, 300);
MoveToEx    (memdc, 320, 320, NULL);
//magenta
SelectObject (memdc, hMargentapen);
LineTo      (memdc, 620, 320);
MoveToEx    (memdc, 340, 340, NULL);

InvalidateRect (hwnd, NULL, 1);

```

`Ellipse()` function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle. The ellipse is outlined by using the current pen and is filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 10, x-coordinate of upper-left corner; 200, y-coordinate of upper-left corner; 200, x-coordinate of lower-right corner; 350, y-coordinate of lower-right corner.

Table 4.5j.

Program win03_command_03.h.

```

MessageBox (hwnd, "", "Ellipses", MB_OK);

SelectObject (memdc, hBluepen);
Ellipse (
    memdc, // handle to the device context
    10,    // x-coord., upper-left corner
    200,   // y-coord., upper-left corner
    200,   // x-coord., lower-right corner
    350);  // y-coord., lower-right corner

SelectObject (memdc, hBluepen);
Pie (
    memdc, // handle to the device context
    550,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    700,   // x-coord., lower-right corner
    250,   // y-coord., lower-right corner
    600,   // x-coord., endpoint of first radial
    15,    // y-coord., endpoint of first radial
    650,   // x-coord., endpoint of second radial
    150);  // y-coord., endpoint of second radial

SelectObject (memdc, hOldpen);
InvalidateRect (hwnd, NULL, 1);

```

```

Ellipse (
    memdc, // handle to the device context
    10,    // x-coordinate of upper-left corner
    200,   // y-coordinate of upper-left corner
    200,   // x-coordinate of lower-right corner
    350);  // y-coordinate of lower-right corner

```

`Pie()` function draws a pie-shaped wedge bounded by the intersection of an ellipse and two radials. The pie is outlined by using the current pen and filled by using the current brush.

```

Pie (
    memdc, // handle to the device context
    550,   // x-coordinate of upper-left corner
    10,    // y-coordinate of upper-left corner
    700,   // x-coordinate of lower-right corner
    250,   // y-coordinate of lower-right corner
    600,   // x-coordinate of endpoint of first radial
    15,    // y-coordinate of endpoint of first radial
    650,   // x-coordinate of endpoint of second radial
    150);  // y-coordinate of endpoint of second radial

```

`Pie()` function requires nine parameters: `memdc`, handle to the device context; 550, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 700, x -coordinate of lower-right corner; 250, y -coordinate of lower-right corner; 600, x -coordinate of endpoint of first radial; 15, y -coordinate of endpoint of first radial; 650, x -coordinate of endpoint of second radial; 150, y -coordinate of endpoint of second radial.

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. Function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. 1, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Ellipses*, corresponding graphical primitives are presented: ellipse and pie at left bottom and right bottom respectively.

IDM_04 - RECTANGLE. Code block, devoted for case `IDM_04`, is implemented in file "`win03_command_04.h`" - Table 4.5k. Rectangle as a geometrical figure will be drawn using a sequence of two operations:

- i) selecting of tool (pen) or default pen, (function `SelectObject()`);
- ii) draw geometrically not-damaged rectangle (function `Rectangle ()`) or draw rectangle with rounded corners (function `RoundRect ()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hRedpen`).

```
SelectObject (memdc, hRedpen);
```

`Rectangle()` function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 100, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 150, x -coordinate of lower-right corner; 150, y -coordinate of lower-right corner.

```
Rectangle (
    memdc, // handle to the device context
    100,    // x-coordinate of upper-left corner
    10,     // y-coordinate of upper-left corner
    150,    // x-coordinate of lower-right corner
    150);   // y-coordinate of lower-right corner
```

`RoundRect()` function draws a geometrically damaged figure - rectangle with rounded corners. The rectangle is outlined by using the current pen and filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 350, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 180, x -coordinate of lower-right corner; 180, y -coordinate of lower-right corner; 15, width of ellipse to draw the rounded corners; 13, height of ellipse to draw the rounded corners.

Table 4.5k.

Program win03_command_04.h.

```

MessageBox (hwnd, "", "Rectangles", MB_OK);

SelectObject (memdc, hRedpen);
Rectangle (
    memdc, // handle to the device context
    100,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    150,   // x-coord., lower-right corner
    150);  // y-coord., lower-right corner

SelectObject (memdc, hCyanopen);
RoundRect (
    memdc, // handle to the device context
    350,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    180,   // x-coord., lower-right corner
    180,   // y-coord., lower-right corner
    15,    // width of ellipse to draw the rounded corners
    13);   // height of ellipse to draw the rounded corners

SelectObject (memdc, hBluepen);
Rectangle (memdc, 500, 50, 390, 130);

SelectObject (memdc, hOldpen);
InvalidateRect (hwnd, NULL, 1);

```

```

RoundRect (
    memdc, // handle to the device context
    350,   // x-coordinate of upper-left corner
    10,    // y-coordinate of upper-left corner
    180,   // x-coordinate of lower-right corner
    180,   // y-coordinate of lower-right corner
    15,    // width of ellipse to draw the rounded corners
    13);   // height of ellipse to draw the rounded corners

```

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. `1`, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of win03. After pushing of menu button *Rectangles* corresponding graphical primitives are presented: three rectangles in the center.

IDM_05 - CLEAR. Code block, devoted for `case IDM_05` is aimed for clearing operations. Function `PatBlt()` paints the specified rectangle using the brush that is currently selected into the specified device context. It means that screen part starting from left top angle (0,0) until right bottom angle (iXmax, iYmax) will be coloured by white (so called screen incialization). `InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. 1, background is erased when the `BeginPaint()` function is called.

```
MessageBox (hwnd, "", "Clear", MB_OK);
PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
InvalidateRect (hwnd, NULL, 1);
```

IDM_HELP - HELP. Code block, devoted for `case IDM_HELP` traditionally is aimed for user-required information. String of *C* type contains information about used graphical primitives and welcome sentence, which will be presented in message box.

```
char *cc;
cc="Pixels, Lines, Ellipses, Rectanges.\n\nEnjoy Window graphics! ";
MessageBox (hwnd, cc, "Help", MB_OK);
```

Fig. 4.8 represents output of win03. After pushing of menu buttons *Clear* and *Help*, message box with the welcome sentence is presented.

HEADER FILE. Table 4.6 represents header file win03.h which must be included into win03.cpp and win03.rc files. The content of header file represents an association of `case` identifiers in `WM_COMMAND` expression only to the integer numbers through operator `define`. Integer number could be chosen freely, but it must be unique for each identifier.

```
#define IDM_01      100
```

RC FILE. Table 4.7 represents resource script (RC) file win03.rc which is responsible for program management organization using menu buttons. First of all, this file contains two `include` statements: library `<windows.h>` and header file `"win03.h"`.

Table 4.6. Header file.	Program win03.h.
<pre>// win03.h // #define IDM_01 100 #define IDM_02 101 #define IDM_03 102 #define IDM_04 103 #define IDM_05 104 #define IDM_HELP 110</pre>	

Table 4.7. RC file.

Program win03.rc.

```

//win03.rc, resource script.
#include "windows.h"
#include "win03.h"

// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

// Menu
MYMENU_003 MENU DISCARDABLE
BEGIN
    MENUITEM "&PIXELS",      IDM_01
    MENUITEM "&LINES",       IDM_02
    MENUITEM "&ELLIPSES",    IDM_03
    MENUITEM "&RECTANGLES",  IDM_04
    MENUITEM "&CLEAR",       IDM_05
    MENUITEM "&HELP",        IDM_HELP
END

#endif    // English (U.S.) resources

```

```

#include "windows.h"
#include "win03.h"

```

The second part of resource script (RC) file contains the description of menu buttons. Statement containing menu name MYMENU_003 is responsible for menu recognition in WinMain() function.

```
MYMENU_003 MENU DISCARDABLE
```

After that, distribution of menu items is presented. Each menu item must be associated with the identifier, previously described in header file. After that, this menu item could be used in call-back function.

```

MENUITEM "&PIXELS",      IDM_01
MENUITEM "&LINES",       IDM_02
MENUITEM "&ELLIPSES",    IDM_03
MENUITEM "&RECTANGLES",  IDM_04
MENUITEM "&CLEAR",       IDM_05
MENUITEM "&HELP",        IDM_HELP

```

Figs. 4.6-4.8 represent the output of win03. Window and menu are present according to the task requirements.

In Fig. 4.7, pushing of menu buttons *Pixels*, *Lines*, *Ellipses*, *Rectangles* was provided. Corresponding graphical primitives occur on screen: vertical lines formed using function `pixel()` on the left; horizontal lines formed using function `pixel()` in the center; ellipses and cut of ellipses (pie) formed using functions `ellipse()` and `pie()` on the left bottom and on the right bottom respectively; rectangles formed using functions `rectangles()` in the center.

In Fig. 4.8, pushing of menu buttons *Clear* and *Help* was provided. Geometrical primitives are absent, message box with the welcome sentence is present. Enjoy window graphics!

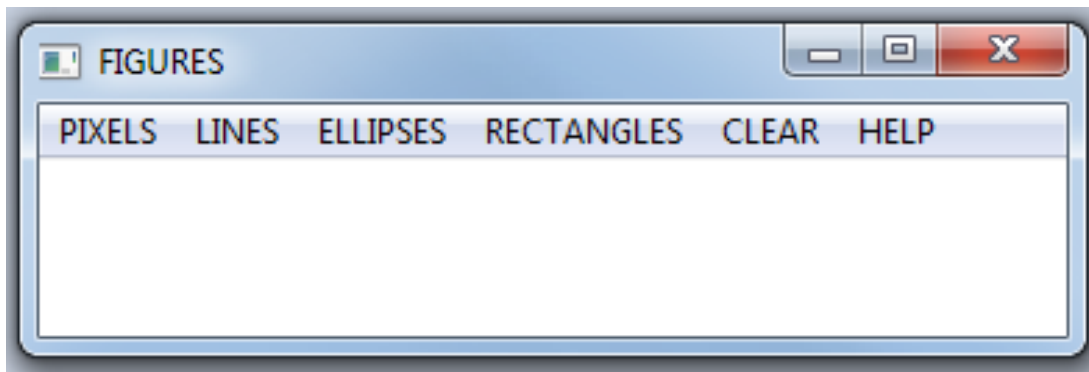


Fig. 4.6. Output of win03. Window and menu.

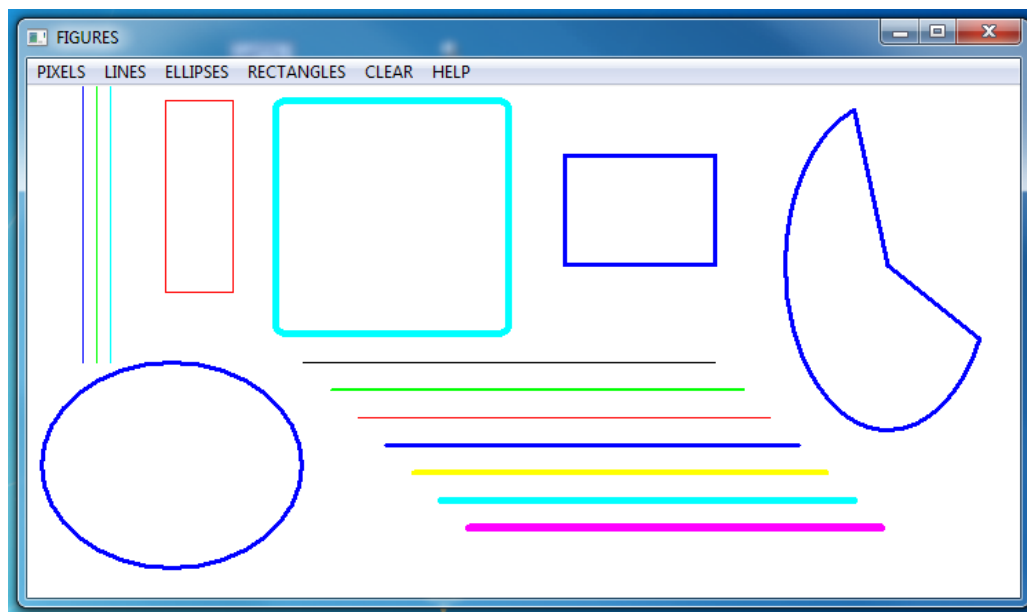


Fig. 4.7. Output of win03. Several graphical primitives.

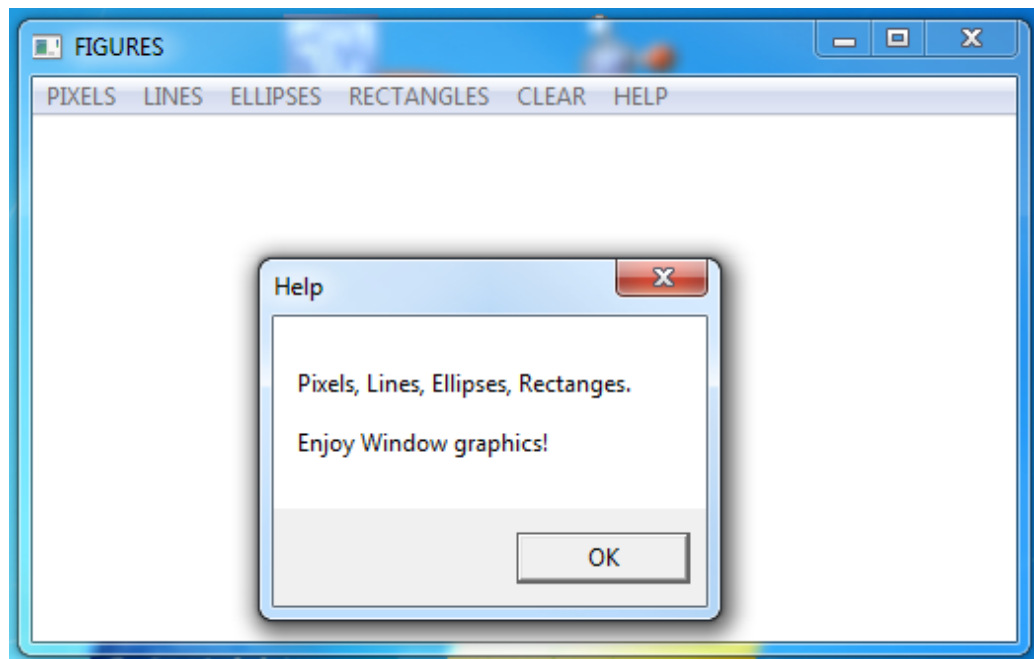


Fig. 4.8. Output of win03. Message box for help.

