Victor Ioan BÂCU
Teodor Traian ȘTEFĂNUȚ
Dănuț Vasile MIHON
Dorian GORGAN

# ELEMENTS OF COMPUTER ASSISTED GRAPHICS

## *Laboratory works*

UTPRESS
Cluj-Napoca, 2015
ISBN 978-606-737-058-4

**Victor Ioan BÂCU**

**Teodor Traian ȘTEFĂNUȚ**

**Dănuț Vasile MIHON**

**Dorian GORGAN**

# Elements of Computer Assisted Graphics

**Laboratory works**

**U.T. PRESS**
**Cluj-Napoca, 2015**
**ISBN 978-606-737-058-4**

Editura U.T.PRESS
Str.Observatorului nr. 34
C.P.42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: Prof.dr.ing. Daniela Manea
Consilier editorial: Ing. Călin D. Câmpean


Multiplicareaă executat la Editura U.T.PRESS.

ISBN 978-606-737-058-4
Bun de tipar: 04.06.2015

# Preface

This book contains 11 laboratory works related to the computer graphics domain. Its main focus are the students from the second year of Computer Science department from the Computer Science and Automation faculty, Technical University of Cluj-Napoca, but it can be used by any engineer interested in this domain.

The content follows the structure of the Elements of Computer Assisted Graphics course taught at Technical University of Cluj-Napoca.

Each laboratory work is structured into four main sections. The first section presents the objectives and what is supposed to be learnt by students, the second section offers an overview of the theoretical background supporting the presented material. The last two sections present some practical examples and some assignments.

Cluj-Napoca, Authors

10.04.2015

# Table of content

# Laboratory work 1: Win32 applications

## 1 Objectives

The objective of this laboratory is to describe very briefly the code structure of a Win32 application and to exemplify the theoretical notions presented through a sample paint application.

## 2 Theoretical background

Windows provides an API called the **Graphics Device Interface**, or **GDI**. Whenever an action occurs, the Windows sends to the application different messages. All the messages are processed in the **WndProc** function. The **WM_PAINT** represents the message that is sent to the application when the repaint of the window is necessary. This message has low priority and is the last message that will be processed.

A **Device Context (DC)** is used to define the attributes of text and images that are output to the screen or printer. The actual context is maintained by GDI. A handle to the Device Context (HDC) is obtained before the output is written and then released after elements have been written. The **HDC (Handle to Device Context)** represents a handle to something you can draw on. A HDC can represent the entire screen, an entire window, the client area of a window, a bitmap stored in memory, or a printer.

When you create a new Win32 project the following basic WndProc function is generated.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
```

```
      case WM_COMMAND:
         wmId = LOWORD(wParam);
         wmEvent = HIWORD(wParam);
         // Parse the menu selections:
         switch (wmId)
         {
            case IDM_ABOUT:
               DialogBox(hInst,
                         MAKEINTRESOURCE(IDD_ABOUTBOX),
                         hWnd, 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...
         EndPaint(hWnd, &ps);
      break;

      case WM_DESTROY:
         PostQuitMessage(0);
      break;

      default:
         return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

The messages that are processed in this function are the following:

- **WM_COMMAND** – is sent whenever a user selects a menu item;
- **WM_PAINT** – is sent when is necessary to repaint the window, the message could be sent by the system or by another application;
- **WM_DESTROY** – is sent when the window will be destroyed.

Depending on the application requirements, you can manage other messages. The messages that you will be using in this laboratory are the following:

- **WM_LBUTTONDOWN** – is sent when the left mouse button is pressed and the mouse cursor is in the client area of a window;
- **WM_RBUTTONDOWN** - is sent when the right mouse button is pressed and the mouse cursor is in the client area of a window.

You can retrieve the mouse coordinates in two ways:

```
x = LOWORD(lParam);
y = HIWORD(lParam);
```

```
x = GET_X_LPARAM(lParam)
y = GET_Y_LPARAM(lParam)
```

For the **GET_X_LPARAM** and **GET_Y_LPARAM** macros you have to include the WindowsX.h header.

You can force the repaint of the window using the InvalidateRect function that will send a WM_PAINT message. If the last parameter is **TRUE** then the old window content is erased, otherwise the invalidated region is accumulated. If the second parameter is **NULL** then the entire window region is invalidated, otherwise you can specify the region that will be invalidated.

## 2.1 Specify colors

To define colors you need to define a variable of type **COLORREF**. The color values can be specified by using the **RGB** macros (you have to specify the values for the *red*, *green* and *blue* components). The values are in the 0-255 interval. To extract the individual values for the red, green, and blue components you can use the **GetRValue**, **GetGValue**, and **GetBValue** macros.

## 2.2 Define a pen

When you define a new pen you need to specify the *style*, *width*, and *color*.

```
HPEN hPen;
hPen = CreatePen(PS_DASH, 1, colorPen);
```

After you have created a new pen you need to specify to what device context you will use it.

```
SelectObject(hdc, hPen);
```

You should delete the pen after you have used it.

```
DeleteObject(hPen);
```

## 2.3 Define a brush

When you define a new brush you need to specify the color of the brush.

```
COLORREF colorBrush;
colorBrush = RGB(rand() % 255, rand() % 255, rand() % 255);

HBRUSH hBrush;
hBrush = CreateSolidBrush(colorBrush);

SelectObject(hdc, hBrush);
```

You should delete the brush after you have used it.

```
DeleteObject(hBrush);
```

## 2.4 Define a hatch brush

When you define a new hatch brush you need to specify the color of the brush and in addition the hatch pattern.

```
HBRUSH myHatchBrush;
myHatchBrush = CreateHatchBrush(HS_CROSS, RGB(255, 255, 0));

SelectObject(hdc, myHatchBrush);
```

## 2.5 Drawing graphical primitives

### 2.5.1 Line

In order to display a line you have to specify the line coordinates. The **MoveToEx** function updates the current position to the specified point. The **LineTo** function draws a line from the current position up to the specified point.

```
MoveToEx(hdc, x, y, NULL);
LineTo(hdc, x + rand() % 200, y + rand() % 200);
```

In the Visual Studio 2010 before you draw a line you have to draw a pixel using the following function:

```
SetPixel(hdc, 0, 0, RGB(255, 255, 255));
```

### 2.5.2 Rectangle

The rectangle function draws a rectangle specified by the corner coordinates.

```
Rectangle(hdc, x, y, x + rand() % 200, y + rand() % 200);
```

### 2.5.3 Ellipse

The ellipse function draws an ellipse specified by the corner coordinates.

```
Ellipse(hdc, x, y, x + rand() % 200, y + rand() % 200);
```

## 2.6 Polyline

The **Polyline** function draws a series of line segments by connecting the points in the specified array.

```
#define verticesNr 8
POINT vertices[verticesNr];
…
//generate polyline vertices
for(int i = 0; i < verticesNr; i++)
{
   vertices [i].x = rand() % 1000;
   vertices [i].y = rand() % 1000;
}
…
Polyline(hdc, vertices, verticesNr);
```

### 2.6.1 Display text

To display a text message you can use one of the following two methods:

```
LPCSTR text1 = "Display a sample message";
…
//first method
TextOut(hdc,100,200,TEXT("Text"),strlen("Text"));
//second method
TextOutA(hdc,100,300,(LPCSTR)(text1),strlen(text1));
```

# 3 Tutorial

## 3.1 Create a new Win32 project

From the main menu, select **File->New->Project**. Specify the name and the location of the project. In the *New Project* window, select **Visual C++->Win32->Win32 Project**.

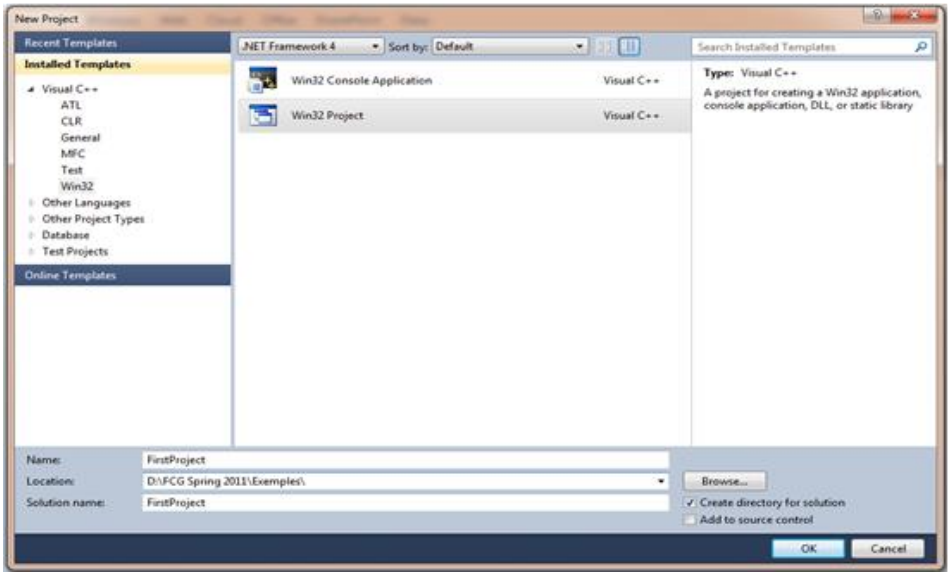


Figure 1.1: Create a new Win32 project

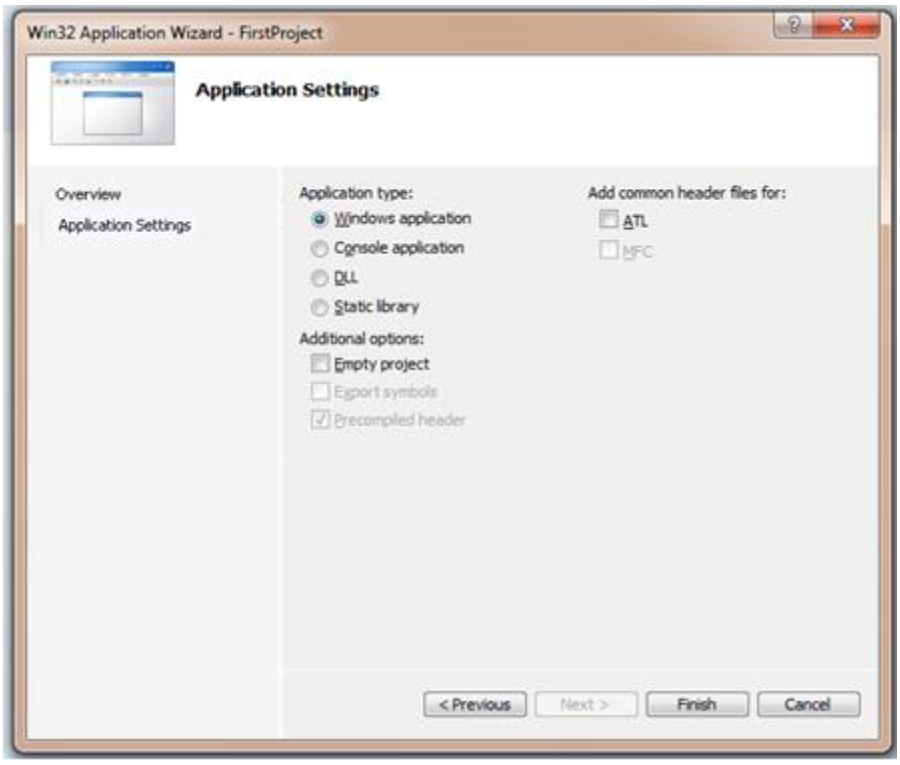


Figure 1.2: Establish application settings

## 3.2 Draw a rectangle onto the window

Add the following lines in the WM_PAINT message:

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
Rectangle(hdc, 100, 100, 400, 400);
EndPaint(hWnd, &ps);
break;
```

Compile and run the application to see the effect.

## 3.3 Define the pen style

Define a new variable, called myPen.

```
HPEN myPen;
```

Add the following lines in the WM_PAINT message:

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
myPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
SelectObject(hdc, myPen);
Rectangle(hdc, 100, 100, 400, 400);
DeleteObject(myPen);
EndPaint(hWnd, &ps);
break;
```

Compile and run the application to see the effect.

## 3.4 Define the brush style

Define a new variable, called myBrush.

```
HBRUSH myBrush;
```

Add the following lines in the WM_PAINT message:

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
myPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
SelectObject(hdc, myPen);
myBrush = CreateSolidBrush(RGB(255, 255, 0));
SelectObject(hdc, myBrush);
Rectangle(hdc, 100, 100, 400, 400);
DeleteObject(myPen);
DeleteObject(myBrush);
EndPaint(hWnd, &ps);
break;
```

Compile and run the application to see the effect.

## 3.5 Add mouse interaction

Define two global variables, x1 and y1, which will be used to store mouse coordinates.

```
static int x1, y1;
```

Modify the following lines in the WM_PAINT message:

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
myPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
SelectObject(hdc, myPen);
myBrush = CreateSolidBrush(RGB(255, 255, 0));
SelectObject(hdc, myBrush);
Rectangle(hdc, x1, y1, x1 + 200, y1 + 200);
EndPaint(hWnd, &ps);
break;
```

Add the WM_LBUTTONDOWN message:

```
case WM_LBUTTONDOWN:
x1 = LOWORD(lParam);
y1 = HIWORD(lParam);

InvalidateRect(hWnd, NULL, FALSE);
break;
```

Compile and run the application to see the effect.

# 4 Assignment

Extend the application with the following functionality:

- Using the right mouse button click select the type of figure you will draw on the screen (line, polyline, rectangle and ellipse)
- Display in the upper-right corner your name
- Display the graphical figures (line, polyline, rectangle and ellipse) with random colors and random line styles (solid, dash, dot, etc.)

# Laboratory work 2: Mouse inputs

## 1 Objectives

This laboratory work will present methods of using **WM_MOUSEMOVE** message by teaching you to draw basic geometrical figures (ex. rectangles) through drag-and-drop interaction. For a better display performance, the double buffering technique will also be shortly described.

## 2 Theoretical background

As already presented in Laboratory Work 1, the mouse input from the user is sent by the Windows operating system to a Win23 application in the form of messages. Drag-and-drop user interaction technique is implemented through the combination of three mouse messages:

- **WM_LBUTTONDOWN** – signals the start of drag-and-drop process
- **WM_MOUSEMOVE** – triggered by the system during the mouse move action
- **WM_LBUTTONUP** – represents the end of the drag-and-drop operation

These messages can be processed in the WndProc function, as presented below

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
                         wParam, LPARAM lParam)
{
int wmId, wmEvent;
PAINTSTRUCT ps;
HDC hdc;

switch (message)
{
   case WM_LBUTTONDOWN:
      // Start drag-and-drop action
   break;

   case WM_MOUSEMOVE:
      // Process mouse move message by updating the display
            accordingly
   break;
```

```
    case WM_LBUTTONUP:
       // End of drag-and-drop action
    break;

    case WM_CREATE:
       // Operations to be done only once, when the
             application is launched
    break;

    default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

As you remember from Laboratory Work 1, the mouse coordinates can be retrieved in two ways:

```
x = LOWORD(lParam)
y = HIWORD(lParam)
```

```
x = GET_X_LPARAM(lParam)
y = GET_Y_LPARAM(lParam)
```

For the **GET_X_LPARAM** and **GET_Y_LPARAM** macros you have to include the WindowsX.h header.

When creating user interactions that require continuous display (like drag-and-drop), the update of the screen must be performed in real time, at the best possible refresh rate. Due to the limitations of the graphical hardware, when drawing directly on the screen some flicker may appear, affecting the clarity of the display. In order to solve these issues, the most common solution is to use the technique called Double Buffering.

## 2.1 Double Buffering concept

The purpose of Double Buffering is to keep the user from seeing the progress of the display process, until it is finalized. Instead of drawing directly to video memory, the application draws everything to a second buffer that is not visible. When finished, everything from the second buffer is copied to video memory all at once. At that point the application clears the double buffer (if necessary) and the process starts over.

## 2.2 Create the second buffer

The creation of a second buffer is a two steps process. First, is necessary to create a compatible Drawing Context:

```
HDC hdcBack;
hdcBack = CreateCompatibleDC(hdc);
```

After you have created the Compatible DC, it is necessary to create and assign to it a bitmap that will store all graphical elements you will draw. The bitmap should have the dimensions of the application window. Therefore, we will define a RECT structure to store this information:

```
RECT windowRect;
```

Then we can create and assign the second buffer:

```
HBITMAP backBuffer;
GetClientRect(hWnd, &windowRect);
backBuffer = CreateCompatibleBitmap(hdc, windowRect.right,
                                    windowRect.bottom);
SelectObject(hdcBack, backBuffer);
```

## 2.3 Draw on the second buffer

When created, the bitmap associated with the second hdc (hdcBack) is black. As our application uses a white background for drawing, we will have to adjust the color:

```
FloodFill(hdcBack, 0, 0, RGB(255, 255, 255));
```

All the drawings that would normally take place directly on the video memory will now be redirected to the second buffer. For example, drawing a rectangle with a specific brush:

```
COLORREF color;
color = RGB(rand() % 255, rand() % 255, rand() % 255);

HBRUSH hBrush;
hBrush = CreateSolidBrush(color);

SelectObject(hdcBack, hBrush);

Rectangle(hdcBack, x1, y1, x2, y2);
```

## 2.4 Display the content of the second buffer

When all the drawing operations are finalized, in the second buffer should be available the final version of the graphical display that will be shown to the user. Nevertheless, it is still not visible until it is loaded into the video memory.

```
BitBlt(hdc, 0, 0, windowRect.right, windowRect.bottom,
       hdcBack, 0, 0, SRCCOPY);
```

If the second buffer's content is not required for the following operations (most of the time the drawing will start from the beginning), the memory should be freed up:

```
DeleteDC(hdcBack);
DeleteObject(backBuffer);
```

# 3 Tutorial

In order to exemplify the notions presented in this paper, we will create a basic Paint like application that will allow the user to draw basic geometrical shapes using drag-and-drop user interaction. To improve the display quality, a basic implementation of the double buffering technique will be described.

## 3.1 Create a new Win32 project

Please refer to the Laboratory Work 1 for instructions on how to create a new Win32 project.

## 3.2 Detect the left mouse button down event, to start the action

Add the following lines in the WM_LBUTTONDOWN message:

```
case WM_LBUTTONDOWN:
x1 = LOWORD(lParam);
y1 = HIWORD(lParam);
break;
```

At this point, we save the mouse coordinates as being one corner of the rectangle enclosing the shape (top left / bottom right).

## 3.3 Detect mouse move event, to draw the shape

Add the following lines in the WM_MOUSEMOVE message:

```
if(wParam & MK_LBUTTON)
{
x2 = LOWORD(lParam);
y2 = HIWORD(lParam);

InvalidateRect(hWnd, NULL, TRUE);
}
```

As the mouse moves, we have to read the pointer coordinates only if the left mouse button is pressed, as this will indicate that we are in the process of drawing a new geometrical shape. The point with the coordinates (x2, y2) represents the opposite corner of the rectangle enclosing the shape, relative to (x1, y1).

Although the WM_LBUTTONUP message is not explicitly processed, the drawing operation will stop automatically when the left mouse button is released.

## 3.4 Draw the shape using mouse coordinates

The mouse coordinates gathered using WM_LBUTTONDOWN and WM_MOUSEMOVE messages can be utilized in WM_PAINT section to draw the graphical shape. For example, drawing a rectangle:

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
color = RGB(rand() % 255, rand() % 255, rand() % 255);
hBrush = CreateSolidBrush(color);
SelectObject (hdc, hBrush);
Rectangle (hdc, x1, y1, x2, y2);
DeleteObject(hBrush);
EndPaint(hWnd, &ps);
break;
```

Compile and run the application to see the effect.

## 3.5 Improve display quality by adding a second buffer

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);

hdcBack = CreateCompatibleDC(hdc);
GetClientRect(hWnd, &windowRect);
backBuffer = CreateCompatibleBitmap(hdc, windowRect.right,
                                    windowRect.bottom);
```

```
SelectObject(hdcBack, backBuffer);
FloodFill(hdcBack, 0, 0, RGB(255, 255, 255));

color = RGB(rand() % 255, rand() % 255, rand() % 255);
hBrush = CreateSolidBrush(color);
SelectObject (hdcBack, hBrush);
Rectangle (hdcBack, x1, y1, x2, y2);

BitBlt(hdc, 0, 0, windowRect.right, windowRect.bottom,
          hdcBack, 0, 0, SRCCOPY);

DeleteObject(hBrush);

DeleteDC(hdcBack);
DeleteObject(backBuffer);

EndPaint(hWnd, &ps);
break;
```

In order to have a correct behavior for the application, it is necessary to change also the InvalidateRect instruction from the WM_MOUSEMOVE message processing section:

```
InvalidateRect(hWnd, NULL, FALSE);
```

Compile and run the application to see the effect.

# 4 Assignment

- Modify the application to allow the user to draw a rectangle by indicating the two corners (top left / bottom right) using two separate left mouse button clicks.
- Switch between the two drawing methods (drag-and-drop / two clicks) by pressing right mouse button.

# Laboratory work 3: Menus and dialog windows

## 1 Objectives

The main goal of this laboratory is the usage of Win32 menu features. A menu could be defined as one of the most important components in a Graphical User Interface application containing a list of services that facilitates the user interaction with the system.

## 2 Theoretical background

In order to create custom menus inside Win32 projects, the resource file (.rc) should be used. This file is located in the **Resource Files** directory, listed in the *Visual Studio Solution Explorer*, among **Header Files** and **Source Files**.

The user interactivity with the menu items generates a WM_COMMAND message that is sent by the Windows operating system to a Win32 application. Each menu item has a unique identifier (called IDM) that could be retrieved by using the lower half value of the wParam.

When creating a new Win32 project, a default menu is automatically generated, that contains the File and Help options. New added menu items could be processed inside the same switch block as the default ones:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
      case WM_COMMAND:
         wmId = LOWORD(wParam);
         wmEvent = HIWORD(wParam);

         // Parse the menu selections:
         switch (wmId)
```

```
            {
               //What to do when using the About menu item
               case IDM_ABOUT:
                  DialogBox(hInst,
                            MAKEINTRESOURCE(IDD_ABOUTBOX),
                            hWnd, About);
               break;

               //What to do when using the Exit menu item
               case IDM_EXIT:
                  DestroyWindow(hWnd);
               break;

               default:
                  return DefWindowProc(hWnd, message, wParam,
                                       lParam);
            }
         break;
      }
}
```

## 2.1 Add new menu items

In Visual Studio 2010 you can create a menu by using the built-in menu editor. In this case, the menu labels would be automatically added to the .rc file.

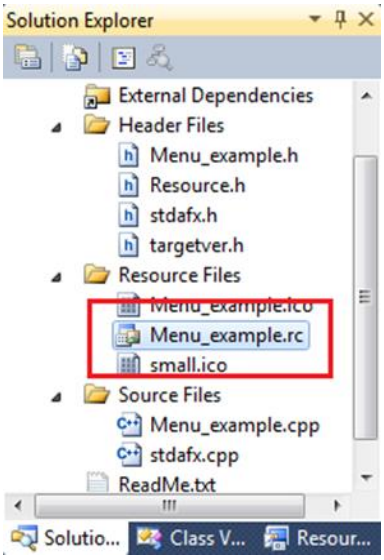


Figure 3.1: Example of project files structure

On an already created Win32 project (see Laboratory Work 1) open the Resource Files folder from the Solution Explorer. An .rc file with the same name as the Win32 project should be located in this folder (see Figure 3.1). This example uses the Menu_example Win32 project.

Double clicking the .rc file displays the available resources for this project (see Figure 3.2Figure).

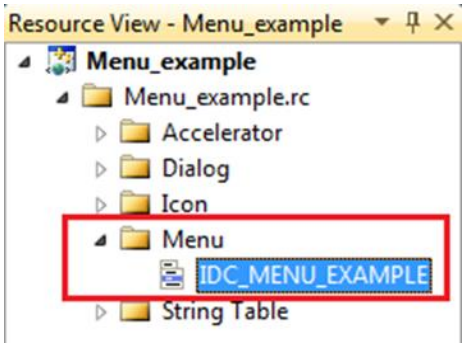


**Figure 3.2: Resources available in the project**

Any other resources of the project can also be customized by the user, such as Icons, mouse cursor or dialog boxes. If no Menu folder is present, right click on the Menu_example.rc and select "Add resource…" option (see Figure 3.3) and then add a menu resource type (see Figure 3.4).

The entire project menu has a unique identifier generated on the basis of the following rule:

```
IDC_<project name>
```

In this example the menu could be programmatically referenced as IDC_MENU_EXAMPLE.

Double clicking on the IDC_MENU_EXAMPLE (see Figure 3.2) opens a new window where we could add or remove menu items using a built-in editor. For example we can create a menu that allows the selection of a geometrical figure to be drawn on the active HDC, as showcased in Figure 3.5.

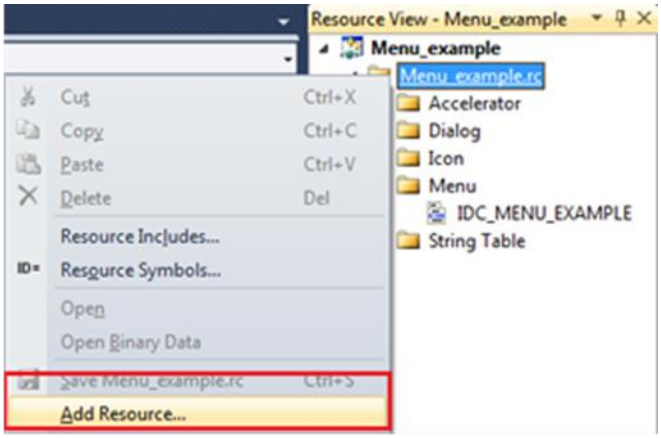


Figure 3.3: Add a new resource to the project

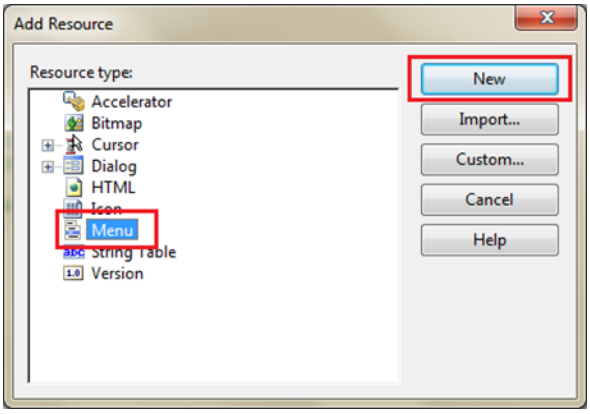


Figure 3.4: Create a new menu resource

It is very important to maintain the uniqueness of the menu items. Each option inside the menu generates its ID based on the following rule:

```
ID_<top menu option>_< menu option>
```

For example the Filled item has ID_RECTANGLE_FILLED as a unique identifier and the Ellipse item has the value ID_FIGURES_ELLIPSE for the ID field. The ID of each menu item is placed in the Properties window of the project (see Figure

3.5Figure). In order to display this window, just right click one of the menu options and then select the Properties option.

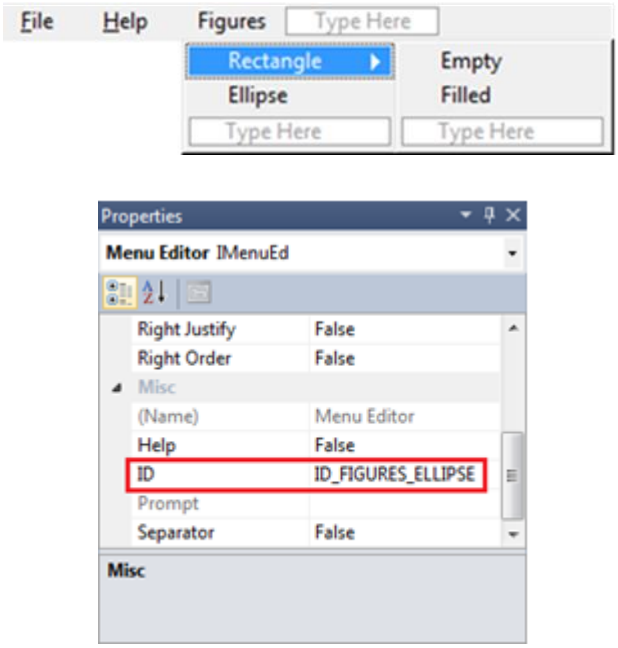


**Figure 3.5: An example of a menu that allows the selection of a geometrical figure**

One of the window class attributes is the menu ID related to this object, stored in the lpszMenuName variable.

```
wcex.lpszMenuName = MAKEINTRESOURCE(IDC_MENU_EXAMPLE);
```

If the menu is defined through an integer identifier, the window class menu is specified using the MAKEINTRESOURCE macro. This macro is used to convert an integer value to a string, representing the menu identifier.

```
wcex.lpszMenuName = MAKEINTRESOURCE(100);
```

## 2.2 Display a popup menu

You could display a popup menu using the following functions:

```
HMENU Popup;
POINT pt;
...
Popup = LoadMenu(hInst, MAKEINTRESOURCE(MENU_ID));
```

```
Popup = GetSubMenu(Popup, 2);
GetCursorPos(&pt);
TrackPopupMenuEx(Popup, TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                 pt.x, pt.y, hWnd, NULL);
```

# 3 Tutorial

The example described in section 3 (Add new menu items) is generally applicable to any kind of application that requires a simple menu. But there are cases where the menu should represent more than a few lines of text. That is why in the following we are trying to create a more complex menu, involving more interactivity on the user side. In order to illustrate the menu concepts, presented above, we propose to draw a rectangle on mouse click event that will be filled with different colors chosen from the interactive menu.

## 3.1 Create a new Win32 project

Please refer to the Laboratory Work 1 for instructions on how to create a new Win32 project.

## 3.2 Add a new menu item

Add a new menu item, among the default ones, called Paint, as presented in Figure 3.6. Make sure that the identifier for the Brush color option is **ID_PAINT_BRUSHCOLOR**. We want to display a new dialog window, as in Figure 3.7, when the Brush color option is used. In order to do this, expand the Dialog folder and add a new dialog resource, by right clicking on the Dialog option and then selecting the Insert dialog. Rename this resource to IDD_SELECT_COLOR.

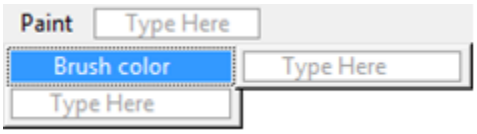


Figure 3.6: Menu item for displaying a new dialog window

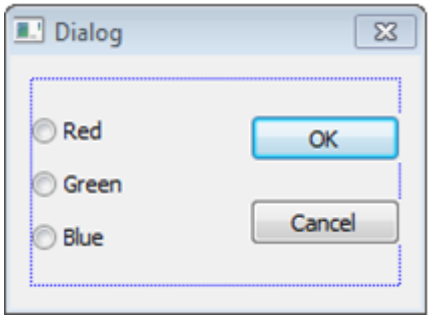


Figure 3.7: Dialog window to select the color of the pen

In order to build a new menu window, like the one presented in Figure 3.7, double click on the IDD_SELECT_COLOR, located in the Dialog folder. This will open a blank window that has to be populated with visual components from the project toolbox, located on the left hand side of the screen (see Figure 3.8Figure).

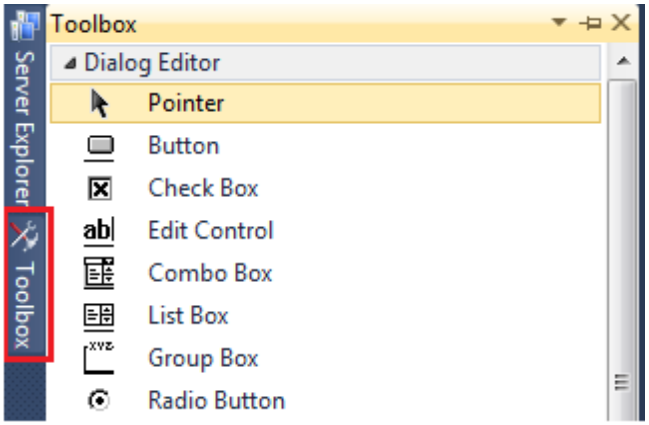


Figure 3.8: Toolbox example

Select from the toolbox the radio button component and with a drag and drop action place it on the blank window. Repeat this operation two more times. Rename the components to Red, Green and Blue. In order to rename a component, make sure that the component is selected, and then go to **Properties** panel and in the **Appearance** section change the **Caption** attribute to your own value.

Because we want to activate only one of these three radio buttons at a specific moment, we must create a radio button group first. In order to do this, select the Red visual component and open the Properties panel. In here go to Misc section and set the Group attribute to true. For the other two radio buttons, set this attribute to false. Also, make sure that the Red, Green and Blue identifiers are as follows: IDC_RADIO1, IDC_RADIO2 and IDC_RADIO3.

Now the menu component is done, and the only thing left is to add the backhand functionality for the menu items. We can now close the Resource View panel and return to the Solution Explorer panel (Figure 3.9). In here open the .cpp file located in the Source Files directory.

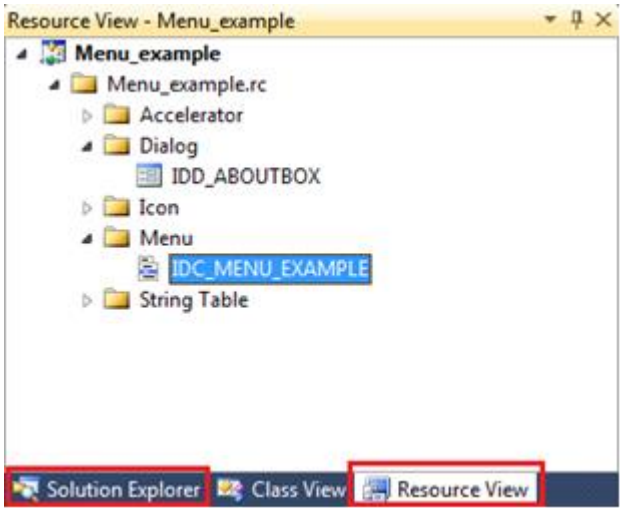


**Figure 3.9: Switch between different views of the project**

## 3.3 Write the functionality for the added menu item

Add the following code lines to the WndProc function:

```
switch (wmId)
{
   case IDM_ABOUT:
      DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd,
                About);
   break;

   case IDM_EXIT:
      DestroyWindow(hWnd);
   break;
```

```
   //Open a dialog box similar to the one presented in
     Figure
   case ID_PAINT_BRUSHCOLOR:
      DialogBox(hInst, MAKEINTRESOURCE(IDD_SELECT_COLOR),
                hWnd, SelectColorDialogBox);
   break;

   default:
      return DefWindowProc(hWnd, message, wParam, lParam);
}
```

The DialogBox function allows us to display a pop up window, inside a parent window, specified by the parent window handler (hWnd in this case). The basic functionality of the pop up is assured by the SelectColorDialogBox function.

At the end of the program add the SelectColorDialogBox function, just after the About function:

```
INT_PTR CALLBACK SelectColorDialogBox(HWND hDlg,
                                      UINT message,
                                      WPARAM wParam,
                                      LPARAM lParam)
{
   UNREFERENCED_PARAMETER(lParam);
   switch (message)
   {
      case WM_INITDIALOG:
         int selection;
         for(selection = IDC_RADIO1; selection < IDC_RADIO3;
             selection++)
         {
            CheckDlgButton(hDlg, selection, BST_UNCHECKED);
         }

         CheckDlgButton(hDlg, currentCheckBoxSelection,
                        BST_CHECKED);

         return (INT_PTR)TRUE;
      break;
      case WM_COMMAND:
         //OK button clicked
         if (LOWORD(wParam) == IDOK)
         {
            EndDialog(hDlg, LOWORD(wParam));

            //Red color selected
```

```
            if(IsDlgButtonChecked(hDlg, IDC_RADIO1) ==
               BST_CHECKED)
            {
               brushColor = RGB(255, 0, 0);
               currentCheckBoxSelection = IDC_RADIO1;
            }

            //Green color selected
            if(IsDlgButtonChecked(hDlg, IDC_RADIO2) ==
               BST_CHECKED)
            {
               brushColor = RGB(0, 255, 0);
               currentCheckBoxSelection = IDC_RADIO2;
            }

            //Blue color selected
            if(IsDlgButtonChecked(hDlg, IDC_RADIO3) ==
               BST_CHECKED)
            {
               brushColor = RGB(0, 0, 255);
               currentCheckBoxSelection = IDC_RADIO3;
            }

            return (INT_PTR)TRUE;
         }

         //Cancel button clicked
         else if(LOWORD(wParam) == IDCANCEL)
         {
            EndDialog(hDlg, LOWORD(wParam));
         }
      break;
   }

   return (INT_PTR)FALSE;
}
```

## 3.4 Define global variables

Some global variables must be declared:

```
COLORREF brushColor = RGB(255, 0, 0);
//At the beginning the Red radio button is selected
int currentCheckBoxSelection = IDC_RADIO1;
int x1, y1;
HPEN myPen;
HBRUSH myBrush;
```

Also add the following line at the beginning of the project, bellow the global variables declaration section:

```
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    SelectColorDialogBox(HWND hDlg,
                                         UINT message,
                                         WPARAM wParam,
                                         LPARAM lParam);
```

## 3.5 Get the mouse coordinates

In order to draw the rectangle at the mouse cursor position, first the mouse coordinates must be identifed. This is done by retrieving the lower and upper half value of the lParam:

```
case WM_LBUTTONDOWN:
x1 = GET_X_LPARAM(lParam);
y1 = GET_Y_LPARAM(lParam);

InvalidateRect(hWnd, NULL, FALSE);
break;
```

## 3.6 Draw the rectangle using the WM_PAINT message

```
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);

myPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
SelectObject(hdc, myPen);

myBrush = CreateSolidBrush(brushColor);
SelectObject(hdc, myBrush);

Rectangle(hdc, x1, y1, x1 + 200, y1 + 200);

EndPaint(hWnd, &ps);
break;
```

Compile and run the application to see the effect.

# 4 Assignment

- Based on the example described in the **Example application** section, add a check box that sets the last parameter of the InvalidateRect function. For example a selected check box is equivalent to the InvalidateRect(hWnd, NULL, TRUE) call.
- Based on the example described in the **Example application** section, add another menu item, called Figures that allow switching between geometrical figures, like: Line, Rectangle, Circle, Polyline.

# Laboratory work 4: Bitmaps, timers and mouse cursors

## 1 Objectives

This laboratory presents some introductory notions on bitmap loading and displaying in a Win32 application. Other subjects that will be presented are the specification of mouse cursors (how you create and display them) and the way in which you can define timers to schedule some events at regular time intervals.

## 2 Theoretical background

In order to display mouse cursors and to define timers some new messages must be processed: **WM_SETCURSOR** and **WM_TIMER**.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
      ...

      case WM_SETCURSOR:
         //display the cursor
      break;

      case WM_TIMER:
         //process
      break;

      ...
   }
}
```

### 2.1 Loading bitmaps

The LoadImage function can be used to load bitmaps from BMP files. Beside loading bitmaps this function can be used to load icons, cursors or animated

cursors. In this laboratory we are focusing only on loading bitmaps using this function.

The syntax of the LoadImage function:

```
HANDLE WINAPI LoadImage(
  __in_opt  HINSTANCE hinst,
  __in      LPCTSTR lpszName,
  __in      UINT uType,
  __in      int cxDesired,
  __in      int cyDesired,
  __in      UINT fuLoad
);
```

You can load a bitmap file in the following way:

```
HBITMAP hBitmap;
...
hBitmap = (HBITMAP)LoadImage(0, TEXT("images/img.bmp"),
                             IMAGE_BITMAP, 0, 0,
                             LR_LOADFROMFILE);
```

If you load a bitmap from a file then the first parameter is NULL. The second parameter specifies the file path. The file path can be specified relative to the project location (like in the example) or as an absolute path.

After you have used the bitmap you should delete it using the DeleteObject function.

```
DeleteObject(hBitmap);
```

In order to display the bitmap you must create an additional HDC from where you will copy the bitmap to the main window.

```
static BITMAP bm;
...
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   GetClientRect(hWnd, &window);

   hdcDB = CreateCompatibleDC(hdc);
   SelectObject(hdcDB, hBitmap);

   GetObject(hBitmap, sizeof(BITMAP), &bm);
   BitBlt(hdc, bitmapStartX, 0, bm.bmWidth, bm.bmHeight,
          hdcDB, 0, 0, SRCCOPY);
```

```
    DeleteDC(hdcDB);
    EndPaint(hWnd, &ps);
break;
```

The BITMAP structure defines the type, width, height, color format, and bit values of a bitmap.

## 2.2 The BitBlt function

The BitBlt function is used to copy the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context (from a memory buffer to the window).

```
BOOL BitBlt(
  __in  HDC hdcDest,
  __in  int nXDest,
  __in  int nYDest,
  __in  int nWidth,
  __in  int nHeight,
  __in  HDC hdcSrc,
  __in  int nXSrc,
  __in  int nYSrc,
  __in  DWORD dwRop
);
```

An example of usage of the BitBlt function:

```
BitBlt(hdc, bitmapStartX, 0, bm.bmWidth, bm.bmHeight, hdcDB,
       0, 0, SRCCOPY);
```

## 2.3 StretchBlt function

The StretchBlt function copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the dimensions of the destination rectangle, if necessary.

```
BOOL StretchBlt(
  __in  HDC hdcDest,
  __in  int nXOriginDest,
  __in  int nYOriginDest,
  __in  int nWidthDest,
  __in  int nHeightDest,
  __in  HDC hdcSrc,
  __in  int nXOriginSrc,
  __in  int nYOriginSrc,
  __in  int nWidthSrc,
  __in  int nHeightSrc,
  __in  DWORD dwRop
```

```
);
```

An example of usage of the StretchBlt function:

```
StretchBlt(hdc, 10, 300, 512, 512, hdcDB, 0, 0, 256, 256,
           SRCCOPY);
```

## 2.4 Cursors

The current position of the mouse is represented through a cursor. You can choose from a set of predefined cursors or you could create new cursors. The process of creating a new cursor is exemplified in the tutorial section. Each cursor has a special area, called hot spot, where the mouse behavior is attached.

The LoadCursor function loads the specified cursor, a predefined one or a custom cursor.

### 2.4.1 Load a custom cursor

In this case the first parameter of the LoadCursor function is the handle to an instance that contains the cursor to be loaded. The second parameter is the name of the cursor resource to be loaded.

```
HCURSOR hCursor;
hCursor = LoadCursor(hInst, MAKEINTRESOURCE(IDC_CURSOR1));
SetCursor(hCursor);
```

### 2.4.2 Load a predifined cursor

In this case the first parameter of the LoadCursor function is NULL. The second parameter is the name of the cursor resource to be loaded.

```
SetCursor(LoadCursor(NULL, IDC_ARROW));
```

## 2.5 Timers

The system can send messages at predefined moments, regularly, using timers. In this way you can schedule an event for a window after a specified time has elapsed.

### 2.5.1 Define a new timer

The first step is to define an identifier for the timer. The identifier value should be unique.

```
#define TIMER_1 1001
```

### 2.5.2 Set the timer

The SetTimer function creates the timer. The first parameter represents the handle to the window that will be associated with the timer. The second parameter is the timer identifier and the third parameter is the time-out value, defined in milliseconds. The last parameter can be used to specify the callback function used to process the timer event. If this parameter is NULL then the system sends a WM_TIMER message to the window attached to the timer.

```
SetTimer(hWnd, TIMER_1, 30, NULL);
```

### 2.5.3 Process timer messages

After the timer interval expires the system sends this WM_TIMER message. The wParam contains the timer identifier.

```
case WM_TIMER:
   switch(wParam)
   {
      case TIMER_1:
         //do something
      break;
   }
break;
```

### 2.5.4 Destroy the timer

The KillTimer function destroys the specified timer. The first parameter is the handle to the window that has attached the timer and the second parameter is the timer identifier.

```
case WM_DESTROY:
   KillTimer(hWnd, TIMER_1);
   PostQuitMessage(0);
break;
```

## 2.6 Check if a point is inside a rectangular region

You can check if a point is inside a rectangular region in a very simple way. First, you have to define a region (which can be rectangle, polygon, or ellipse). You create a rectangular region using the **CreateRectRgn** function. The parameters for this function are the coordinates (X and Y) of the upper-left corner and lower-right corner. The **PtInRegion** function determines whether the specified point is inside the specified region.

```
static HRGN imgRgn;
imgRgn = CreateRectRgn(0, 0, 512, 512);
PtInRegion(imgRgn, cursorPosX, cursorPosY);
```

# 3 Tutorial

## 3.1 Load a bitmap file

The loading of the bitmap file is done in this example in the WM_CREATE message. In this way, you load the bitmap from the file only once, when the window is initialized.

```
static HBITMAP hBitmap;
static BITMAP bm;
...
case WM_CREATE:
   hBitmap = (HBITMAP)LoadImage(0, TEXT("images/img.bmp"),
                                IMAGE_BITMAP, 0, 0,
                                LR_LOADFROMFILE);
   GetObject(hBitmap, sizeof(BITMAP), &bm);
break;
```

## 3.2 Destroy the bitmap object

In the WM_DESTROY message you should delete the bitmap object. Modify the source in the following way:

```
case WM_DESTROY:
   DeleteObject(hBitmap);
   PostQuitMessage(0);
break;
```

## 3.3 Display the bitmap object

A bitmap object cannot be displayed directly onto the screen. We have to define a new HDC and use it for Bitmap representation. From this HDC we will copy on the screen the content of the Bitmap.

Modify the WM_PAINT message in the following way:

```
static RECT window;
HDC hdcDB;
...
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   GetClientRect(hWnd, &window);
```

```
   hdcDB = CreateCompatibleDC(hdc);
   SelectObject(hdcDB, hBitmap);
   BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcDB, 0, 0,
          SRCCOPY);

   DeleteDC(hdcDB);
   EndPaint(hWnd, &ps);
break;
```

## 3.4 Create a new mouse cursor

The cursors are added as resources. From the “Add Resource” dialog box choose ”cursor”.

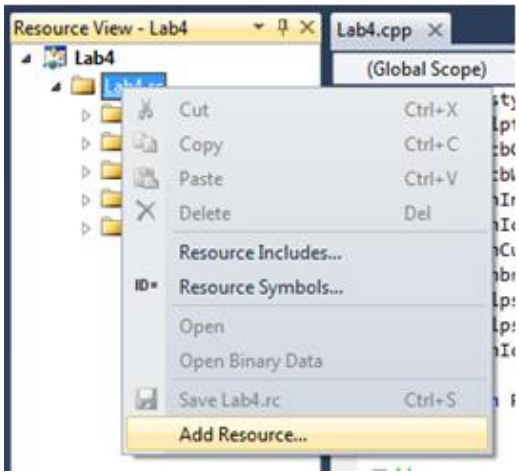


Figure 4.1: Adding a new resource to the project

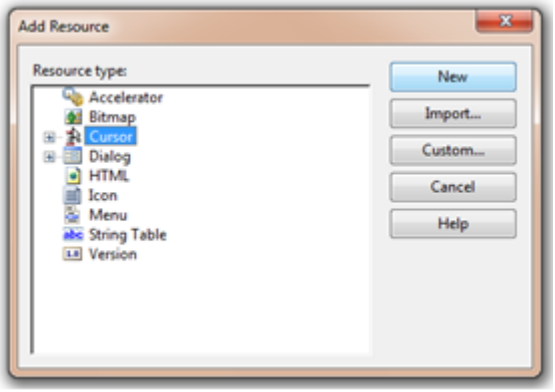


Figure 4.2: Select resource type to be added

You can draw the cursor using the tools that are available in the top part of the screen.

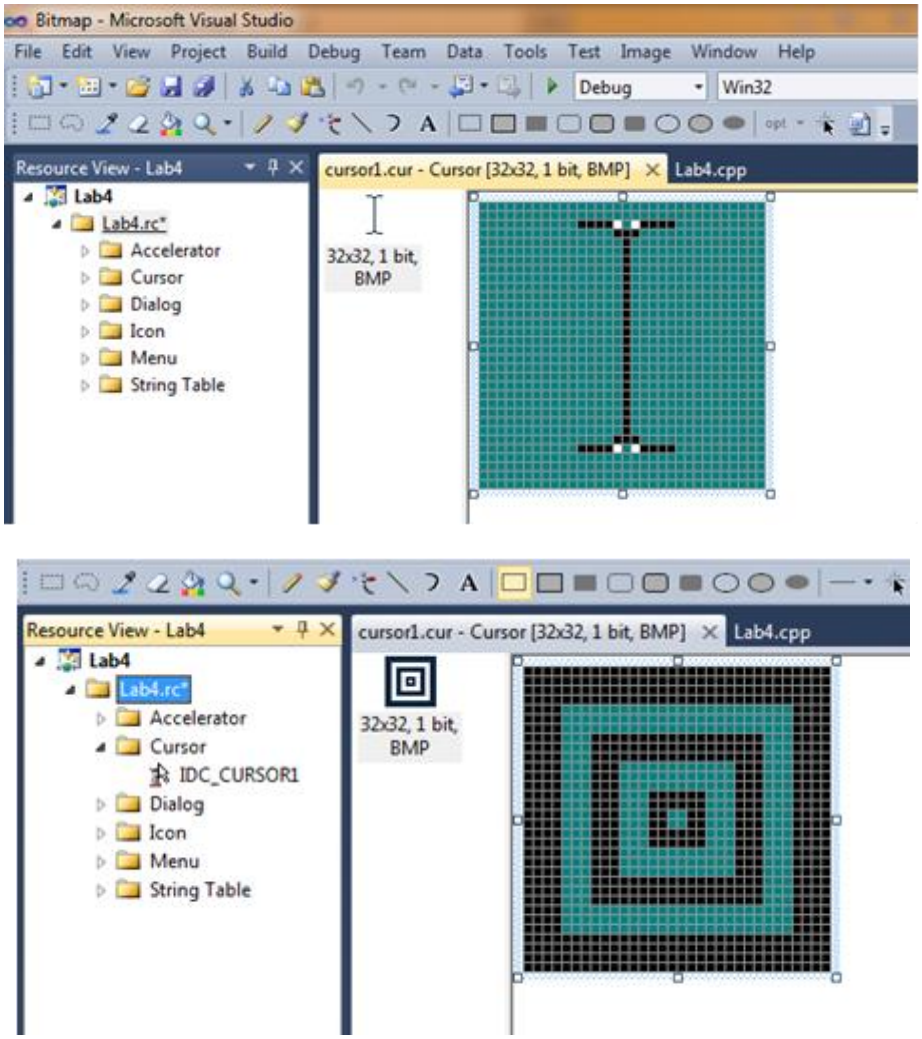


Figure 4.3: Create custom cursor shapes

## 3.5 Define a rectangular region

```
static HRGN imgRgn;
static int cursorPosX, cursorPosY;
HCURSOR hCursor;
...
case WM_CREATE:
```

```
   hBitmap = (HBITMAP)LoadImage(0, TEXT("images/img.bmp"),
                                IMAGE_BITMAP, 0, 0,
                                LR_LOADFROMFILE);
   GetObject(hBitmap, sizeof(BITMAP), &bm);
   imgRgn = CreateRectRgn(0, 0, bm.bmWidth, bm.bmHeight);
break;
```

## 3.6 Create a timer and display the new mouse cursor when the mouse is over the bitmap region

```
#define TIMER_1 1001
…
case WM_CREATE:
   hBitmap = (HBITMAP)LoadImage(0, TEXT("images/img.bmp"),
                                IMAGE_BITMAP, 0, 0,
                                LR_LOADFROMFILE);
   GetObject(hBitmap, sizeof(BITMAP), &bm);
   imgRgn = CreateRectRgn(0, 0, bm.bmWidth, bm.bmHeight);
   SetTimer(hWnd, TIMER_1, 30, NULL);
break;

case WM_MOUSEMOVE:
   cursorPosX = LOWORD(lParam);
   cursorPosY = HIWORD(lParam);
break;

case WM_SETCURSOR:
   hCursor = LoadCursor(hInst, MAKEINTRESOURCE(IDC_CURSOR1));
   if(PtInRegion(imgRgn, cursorPosX, cursorPosY))
      SetCursor(hCursor);
   else
      SetCursor(LoadCursor(NULL, IDC_ARROW));
break;

case WM_TIMER:
   switch(wParam)
   {
      case TIMER_1:
         if(bitmapStartX < window.right)
            bitmapStartX += 1;
         else
            bitmapStartX = 0;

         imgRgn = CreateRectRgn(bitmapStartX, 0,
                                bitmapStartX + bm.bmWidth,
                                bm.bmHeight);
         InvalidateRect(hWnd, NULL, FALSE);
      break;
   }
```

```
break;

case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   GetClientRect(hWnd, &window);

   hdcDB = CreateCompatibleDC(hdc);
   SelectObject(hdcDB, hBitmap);

   BitBlt(hdc, bitmapStartX, 0, bm.bmWidth, bm.bmHeight,
          hdcDB, 0, 0, SRCCOPY);

   DeleteDC(hdcDB);
   EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
   DeleteObject(hBitmap);
   KillTimer(hWnd, TIMER_1);
   PostQuitMessage(0);
break;
```

# 4 Assignment

- Display 3 different bitmaps on the screen at random positions. Define a timer for each bitmap and after the timer attached to that bitmap expires change the bitmap position (randomly).
- Create a game where the user must click on different bitmaps that are changing regularly the position. Count and display the number of successful hits.

# Laboratory work 5: Keyboard inputs

## 1 Objectives

This laboratory presents the methods required to work with keyboard in a Win32 application. You will learn how to detect usual and special characters and how to apply different actions to each of them. There will be also presented different ways of displaying text into a Win23 application, using different fonts, sizes and colors.

## 2 Theoretical background

In Win32 applications, the keyboard events are handled as messages. The most common message identifiers that can be used to catch keyboard related messages are: **WM_KEYUP**, **WM_SYSKEYUP**, **WM_KEYDOWN**, **WM_SYSKEYDOWN**, **WM_CHAR**. These are sent whenever a keyboard key is pressed or released. All the information, about what key has been pressed and how, is contained in **wParam** and **lParam** parameters of the *WndProc* function.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   PAINTSTRUCT ps;
   HDC hdc;

   switch (message)
   {
      ...

      case WM_KEYDOWN:
         //when any key is pressed
      break;

      case WM_SYSKEYDOWN:
         //when a system key (F10, ALT) is pressed
      break;

      case WM_CHAR:
```

```
            //when a regular key is pressed, called after
              WM_KEYDOWN
        break;
        case WM_KEYUP:
            //when any key is released
        break;

        case WM_SYSKEYUP:
            //when a system key (F10, ALT) is released
        break;

        ...
    }
}
```

**wParam** contains the "virtual-key codes" of the pressed keyboard key. This is a code halfway translated to ASCII that is sufficient for the keyboard control of many applications. For example all the letter keys have a virtual-key code that is just the ASCII code for the capital letters. Also, the number keys above the letters have a virtual-key code that represents the ASCII codes for these digits. But the numeric keypad numbers, for example, have different virtual-key codes (96-105) to distinguish them from the other keys.

**lParam** contains supplemental information about the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag. All these pieces of information can be used to deal with keys pressed for a longer period of time, with extended keyboards or specific OEM encodings of the hardware.

## 2.1 Display text on the screen

As already mentioned in Laboratory 1, in order to display a text message you can use the TextOut function:

```
BOOL TextOut(
  __in  HDC hdc,
  __in  int nXStart,
  __in  int nYStart,
  __in  LPCTSTR lpString,
  __in  int cchString
);
```

An example of usage of the TextOut function:

```
LPCSTR text1 = "Display a sample message";
```

```
…
TextOut(hdc,100,200,TEXT("Text"),strlen("Text"));

//in order to display ANSI encoded chars, you can use:
TextOutA(hdc,100,300,(LPCSTR)(text1),strlen(text1));
```

Through these functions, the text can be displayed with different font, size and color settings.

## 2.2 Change font settings

In order to set the font settings for a specific text display action, you should use the function **CreateFont**, which creates a logical font with the specified characteristics.

```
HFONT CreateFont(
  __in  int nHeight,
  __in  int nWidth,
  __in  int nEscapement,
  __in  int nOrientation,
  __in  int fnWeight,
  __in  DWORD fdwItalic,
  __in  DWORD fdwUnderline,
  __in  DWORD fdwStrikeOut,
  __in  DWORD fdwCharSet,
  __in  DWORD fdwOutputPrecision,
  __in  DWORD fdwClipPrecision,
  __in  DWORD fdwQuality,
  __in  DWORD fdwPitchAndFamily,
  __in  LPCTSTR lpszFace
);
```

An example of usage of the **CreateFont** function:

```
HFONT hFont;
....
hFont = CreateFont(36, 20, 0, 0, FW_DONTCARE, FALSE, TRUE,
                   FALSE, DEFAULT_CHARSET,
                   OUT_OUTLINE_PRECIS, CLIP_DEFAULT_PRECIS,
                   CLEARTYPE_QUALITY, VARIABLE_PITCH,
                   TEXT("Times New Roman"))
```

Through the CreateFont function you can set most of the text display characteristics: the height and width of the characters, font weight, font decoration (italic, underlined, strikeout) etc.

## 2.3 Change text color

The color of the text cannot be changed through the CreateFont function, as it is not a font attribute. In order to modify the color of the displayed text, you need to use the **SetTextColor** function:

```
COLORREF SetTextColor(
  __in  HDC hdc,
  __in  COLORREF crColor
);
```

If the function succeeds, the return value is a color reference for the previous text color.

An example of usage of the **SetTextColor** function:

```
SetTextColor(hdc, textColor);
```

## 2.4 Window focus management messages

In order to be able to process the keyboard messages, your application needs to have the focus (to be the one receiving the keyboard messages). You can check the status of the focus through the **WM_SETFOCUS** and **WM_KILLFOCUS** messages.

```
case WM_SETFOCUS:
   //do what is needed on focus gain (ex. show the carret)
break;

case WM_KILLFOCUS:
   //do what is needed on focus out (ex. hide the carret)
break;
```

# 3 Tutorial

Through this tutorial you will learn how to display characters on the screen using different fonts, colors, sizes and text-decoration settings. As some of the characters are “invisible” we will display for them a specific text message at each key press.

## 3.1 Catch the keyboard related messages

We will use WM_CHAR message to catch regular key presses and WM_KEYDOWN to catch keys like left-arrow, right-arrow, etc.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
   int wmId, wmEvent;
   HDC hdc;

   switch (message)
   {
      ...

      case WM_KEYDOWN:
         //when any key is pressed
      break;

      case WM_CHAR:
         //when a regular key is pressed, called after
           WM_KEYDOWN
      break;

      ...
   }
}
```

## 3.2 Display messages for left and right arrows press

We will display messages corresponding to left and right arrows key presses, at random positions.

```
static LPCSTR specialBuffer[100];
static int specialNr = 0;
int i, posX, posY;
...
case WM_KEYDOWN:
   switch (wParam)
   {
      case VK_LEFT:
         specialBuffer[specialNr] = "left arrow";
         specialNr++;
      break;

      case VK_RIGHT:
         specialBuffer[specialNr] = "right arrow";
         specialNr++;
      break;
   }

   InvalidateRect(hWnd, NULL, TRUE);
break;
```

```
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   for(i = 0; i < specialNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;
      TextOutA(hdc, posX, posY, (LPCSTR)(specialBuffer[i]),
               strlen(specialBuffer[i]));
   }
   EndPaint(hWnd, &ps);
break;
```

## 3.3 Display regular characters pressed

```
static TCHAR charsBuffer[100];
static int charsNr = 0;
...
case WM_CHAR:
   charsBuffer[charsNr] = wParam;
   charsNr++;
   InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);

   for(i = 0; i < specialNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;
      TextOutA(hdc, posX, posY, (LPCSTR)(specialBuffer[i]),
               strlen(specialBuffer[i]));
   }

   for(i = 0; i < charsNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;
      TextOutA(hdc, posX, posY,
              (LPCSTR)(&charsBuffer[i]), 1);
   }

   EndPaint(hWnd, &ps);
break;
```

## 3.4 Display messages for SPACE, ENTER, TAB key presses

```
case WM_CHAR:
   switch(wParam)
   {
      case 0x09:
         specialBuffer[specialNr] = "tab";
         specialNr++;
      break;

      case 0x0D:
         specialBuffer[specialNr] = "enter";
         specialNr++;
      break;

      case 0x20:
         specialBuffer[specialNr] = "space";
         specialNr++;
      break;

      default:
         charsBuffer[charsNr] = wParam;
         charsNr++;
      break;
   }

   InvalidateRect(hWnd, NULL, TRUE);
break;
```

## 3.5 Create multiple fonts to display the text

```
static HFONT hFont[2];
static HFONT defaultFont;
...
case WM_CREATE:
   hFont[0] = CreateFont(36, 20, 0, 0, FW_DONTCARE, FALSE,
                         TRUE, FALSE, DEFAULT_CHARSET,
                         OUT_OUTLINE_PRECIS,
                         CLIP_DEFAULT_PRECIS,
                         CLEARTYPE_QUALITY, VARIABLE_PITCH,
                         TEXT("Times New Roman"));

   hFont[1] = CreateFont(36, 10, 0, 0, FW_DONTCARE, FALSE,
                         FALSE, TRUE, DEFAULT_CHARSET,
                         OUT_OUTLINE_PRECIS,
                         CLIP_DEFAULT_PRECIS,
                         CLEARTYPE_QUALITY, VARIABLE_PITCH,
                         TEXT("Arial"));
```

```
   defaultFont = CreateFont(12, 12, 0, 0, FW_DONTCARE, FALSE,
                            FALSE, FALSE, DEFAULT_CHARSET,
                            OUT_OUTLINE_PRECIS,
                            CLIP_DEFAULT_PRECIS,
                            CLEARTYPE_QUALITY,
                            VARIABLE_PITCH, TEXT("Arial"));
break;

case WM_DESTROY:
   DeleteObject(hFont[0]);
   DeleteObject(hFont[1]);
   DeleteObject(defaultFont);
   PostQuitMessage(0);
break;
```

## 3.6 Display text with random fonts and colors

```
COLORREF textColor;
...
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);

   for(i = 0; i < specialNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;

      textColor = RGB(rand() % 255, rand() % 255,
                      rand() % 255);
      SetTextColor(hdc, textColor);

      fontIndex = rand() % 2;
      SelectObject(hdc,hFont[fontIndex]);

      TextOutA(hdc, posX, posY, (LPCSTR)(specialBuffer[i]),
               strlen(specialBuffer[i]));
   }

   for(i=0; i<charsNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;

      textColor = RGB(rand() % 255, rand() % 255,
                      rand() % 255);
      SetTextColor(hdc, textColor);

      fontIndex = rand() % 2;
      SelectObject(hdc,hFont[fontIndex]);
```

```
      TextOutA(hdc,posX,posY,(LPCSTR)(&charsBuffer[i]),1);
   }

   EndPaint(hWnd, &ps);
break;
```

## 3.7 Detect and display information about Window focus

```
static bool hasFocus = false;
...
case WM_SETFOCUS:
   hasFocus = true;
   InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_KILLFOCUS:
   hasFocus = false;
   InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);

   for(i = 0; i < specialNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;

      textColor = RGB(rand() % 255, rand() % 255,
                      rand() % 255);
      SetTextColor(hdc, textColor);

      fontIndex = rand() % 2;
      SelectObject(hdc,hFont[fontIndex]);

      TextOutA(hdc, posX, posY, (LPCSTR)(specialBuffer[i]),
               strlen(specialBuffer[i]));
   }

   for(i=0; i<charsNr; i++)
   {
      posX = rand() % 500 + 20;
      posY =  rand() % 300 + 20;

      textColor = RGB(rand() % 255, rand() % 255,
                      rand() % 255);
      SetTextColor(hdc, textColor);

      fontIndex = rand() % 2;
```

```
      SelectObject(hdc,hFont[fontIndex]);

      TextOutA(hdc,posX,posY,(LPCSTR)(&charsBuffer[i]),1);
   }

   textColor = RGB(0, 0, 0);
   SetTextColor(hdc, textColor);
   SelectObject(hdc, defaultFont);

   if(hasFocus)
   {
      TextOutA(hdc, 0, 0, (LPCSTR)("Window has focus"),
               strlen("Window has focus"));
   }
   else
   {
      TextOutA(hdc, 0, 0,
              (LPCSTR)("Window doesn't have focus"),
               strlen("Window doesn't have focus"));
   }

   EndPaint(hWnd, &ps);
break;
```

# 4 Assignment

- On lose focus, delete the information displayed on the window and leave the window blank. Only the focus status message remains visible. On gain focus, display again all the information.
- When backspace is pressed, delete the information corresponding to the last key press stored into the buffers. This behavior should apply to both regular and special characters buffer.

# Laboratory work 6: Bresenham algorithm

## 1 Objectives

This laboratory highlights the Bresenham algorithm usage when rendering some of the graphical primitives on the computer display. It begins by presenting some generic information about the algorithm and then exemplifies them for the line and circle raterization.

## 2 Theoretical background

The Bresenham algorithm for drawing lines onto a bi-dimensional space (like a computer display) is the fundamental method used in computer graphics discipline. The algorithm efficiency makes it one of the most required methods for drawing continuous lines, circles or other graphical primitives. This process is called rasterization.

Each line, circle or other graphical primitives could be plotted pixel by pixel. Each pixel is described by a fixed position in the bi-dimensional XOY space. The algorithm approximates the real value of a line by computing each line's pixel positions. Since the pixels are the smallest addressable screen elements in a display device, the algorithm approximation is good enough to "trick" the human eyes and to get the illusion of a real line. Figure 6.1 shows the real line and the approximated line drawn over the pixel grid.

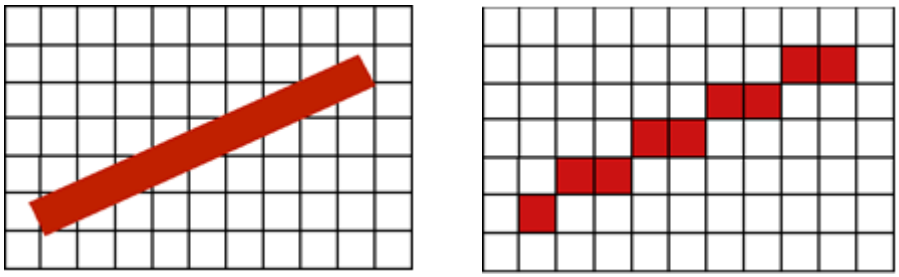

Figure 6.1: Exemplification of line representation in a pixel grid

Before moving on, it is worth to mention that both the line (1) and circle (2) could be mathematically described using the following equations:

$$y = m \cdot x + c \quad (1)$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$(x-a)^2 + (y-b)^2 = R^2 \quad (2)$$

where:

- m is the line slope;

- $(x_1, x_2)$, $(y_1, y_2)$ are the two endpoints of the line segment;

- $(a, b)$ represents the center coordinates of the circle;

## 2.1 Bresenham's algorithm for line

For simplicity, we will take into account a line segment with the slope from 0 to 1. Let set the two endpoints of the line to be $A(x_1, y_1)$ and $B(x_2, y_2)$. At this point we have to choose an initial point to start the algorithm. We can choose this point ($P(x_i, y_i)$) to be either A or B. Based on the starting position, we have eight possible choices to draw the next pixel of the line. This is due to the fact that each pixel is surrounded by 8 adjacent pixels.

Our example will consider only the case where we have two choice alternatives for the next pixel position (in other words this example will work only for the first half of the clockwise circle). For example, for the current point P we have the following drawing possibilities: $T(x_{i+1}, y_i)$ or $S(x_{i+1}, y_{i+1})$.

The decision criterion (Figure 6.2) for the Bresenham algorithm is based on the distance between the current point, P, and the real line segment. So the closest point (T or S) to the line segment will be chosen.

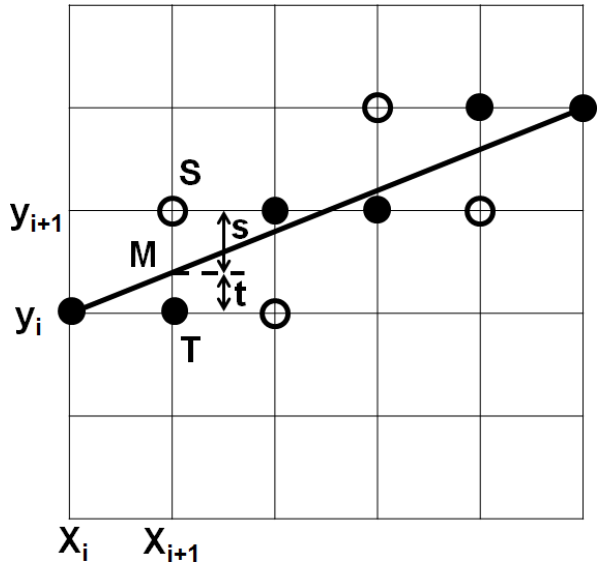


**Figure 6.2: Illustration on the decision making in Bresenham algorithm**

The following paragraphs will describe the general steps of the Bresenham's algorithm in natural language rather than a programmatically one, because it is easier to understand.

a. Let us assume that we have to draw a line segment with the endpoints represented by $A(x_1, y_1)$ and $B(x_2, y_2)$. We translate the line segment with $(-x_1, -y_1)$ to place it on the XOY system origin.

b. Let $dx = x_2 - x_1$, $dy = y_2 - y_1$. The line that needs to be drawn could be described as $y = \frac{dx}{dy} x$ .

c. In this step we intend to compute the next line pixel, using the criterion mentioned above. From Figure 6.2, we can deduce that the closest point to the real line value is $T(x_{i+1}, y_i)$. Based on this observation we could say that $M(x_{i+1}, x_{i+1} \cdot \frac{dy}{dx})$ (3).

In other words $\begin{cases} t = y_m - y_i \\ s = y_{i+1} - y_m \end{cases} \Rightarrow t - s = 2 \cdot y_m - 2 \cdot y_i - 1$ (4).

Taking into account (3) and (4) we obtain $dx \cdot (t-s) = 2 \cdot dy \cdot x_{i+1} - 2 \cdot dx \cdot y_i - dx$. If the $x$ coordinates of the line segment endpoints are in $x_1 < x_2$ relationship, then the $t-s$ sign will coincide with the sign of the $dx \cdot (t-s)$ .

d. We could obtain the following recurrence relationship: $d_{i+1} = d_i + 2 \cdot dy - 2 \cdot dx \cdot (y_i - y_{i+1})$ if we consider that $dx \cdot (t-s) = d_{i+1}$.

The initial value of $d_i = 2 \cdot dy - dx$ is obtained for the $x_0 = 0$ and $y_0 = 0$ . We can conclude that:

- if $d_i \geq 0 \Rightarrow (t-s) \geq 0$ and the closest point to the real line segment is $S(x_{i+1}, y_{i+1})$ . Based on this observations we find that $d_i$ value could be computed as $d_{i+1} = d_i + 2(dy - dx)$ .

- if $d_i < 0 \Rightarrow (t-s) < 0$ and the closest point to the real line segment is $T(x_{i+1}, y_i)$ . Then the recurrence formula to compute $d_i$ is $d_{i+1} = d_i + 2dy$ .

The pseudo code for the Bresenham algorithm is described in the following paragraph, and it is based on the mathematical observations mentioned above.

```
Algorithm draw_line ()
{
   //Initialize the variables
   dx = abs(x2-x1);
   dy = abs(y2-y1);
   d = 2*dy-dx;
   inc1 = 2*dy;
   inc2 = 2*(dy-dx);

   //Set the starting point
   if (x1 > x2) then
   {
      x = x2;
      y = y2;
      xc = x1;
   }
   else{
```

```
        x = x1;
        y = y1;
        xc = x2;
    }

    //Draw each pixel of the line
    while (x < xc) {
        //Draw the current pixel of the line
        DrawPixel(x, y);
        x = x+1;
        if (d < 0) then {
            d = d+inc1;
        }
        else {
            y = y+1;
            d = d+inc2;
        }
    }
}
```

## 2.2 Bresenham's algorithm for circle

Let us say we want to scan-convert a circle centered at $(0,0)$ with an integer radius R (Figure 6.3). We'll see that the ideas we previously used for line scan-conversion can also be used for this task. First of all, notice that the interior of the circle is characterized by the inequality $D(x,y): x^2+y^2-R^2<0$. We'll use $D(x,y)$ to derive our decision variable.

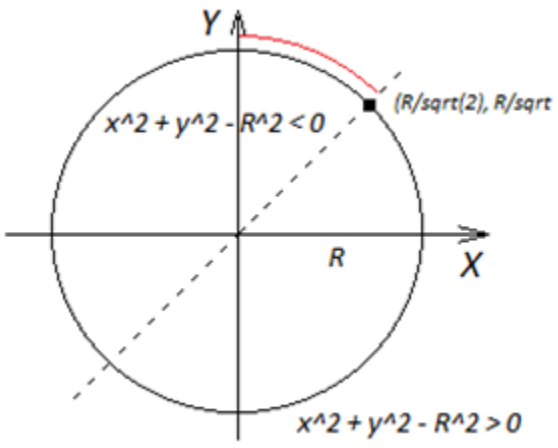


Figure 6.3: Scan conversion of a circle with integer radius R

Following the same approach as for the line segment representation, we'll first present the Bresenham's algorithm for the circle in natural language, describing for each step the general ideas behind it.

a. First, let's think how to plot pixels close to the 1/8 of the circle marked red in Figure 6.3. The range of the coordinate for such pixels is from 0 to $R\sqrt{2}$ . We'll go over vertical scanlines through the centers of the pixels and, for each such scanline, compute the pixel on that line which is the closest to the scanline-circle intersection point (black dots in Figure 6.4). All such pixels will be plotted by our procedure.

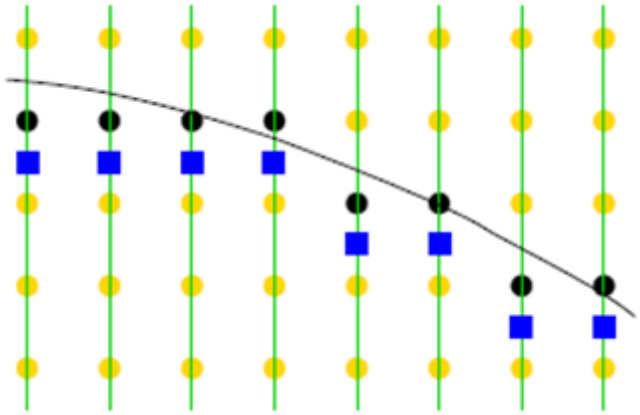

**Figure 6.4: Circle representation on a pixel grid**

b. Notice that each time we move to the next scanline, the y-coordinate of the plotted point either stays the same or decreases by 1 (the slope of the circle is between -1 and 0). To decide what needs to be done, we'll use the decision variable, which will be the value of $D(x, y)$ evaluated at the blue square (e.g. the midpoint between the plotted pixel and the pixel immediately below).

c. The first pixel plotted is $(0, R)$ and therefore the initial value of the decision variable should be

$$D(0, R-0.5)=(R-0.5)^2-R^2=0.25-R$$

The $y$ variable, holding the second coordinates of the plotted pixels, will be initialized to *R*. Let's now think what happens after a point $(x, y)$ is

plotted. First, we'll pretend that we need to move the plotted point to the right (no change in $y$) and check if this keeps the decision variable negative (we don't want any blue squares outside the circle). If $(x, y)$ is the last plotted point, the decision variable is $D(x, y-0.5)$. After we move to the right, it becomes $D(x+1, y-0.5)$. Simple arithmetic shows that it increases by $D(x+1, y-0.5)$ - $D(x, y-0.5) = 2x+1$. If this increase makes it positive, we'd better move down by 1 pixel. This puts the blue square at $(x+1, y-1.5)$ and means that we need to increase the decision value by the previous $2x+1$ plus $D(x+1, y-1.5)-D(x+1, y-0.5)=2-2y$.

d. Clearly, to make the decision variable integer, we need to scale it by a factor of 4. Eight-way symmetry is used to go from 1/8-the of the circle to the full circle.

The pseudo code for the Bresenham's algoritm for circle is described below, based on the earlier made observations.

```
Algorithm draw_circle ()
{
  y = R;
     d = 1/4 - R;

    for x=0 to ceil(R/sqrt(2)) do {
        plot_points(x,y);
        d = d + 2x + 1;
        if (d>0) then
        {
             d = d + 2 - 2y;
             y = y - 1;
        }
    }
}
```

You can find the plot_points function definition below:

```
Function plot_points (x, y)
{
DrawPixel(x,y);
DrawPixel (x,-y);
DrawPixel (-x,y);
DrawPixel (-x,-y);
```

```
DrawPixel (y,x);
DrawPixel (−y,x);
DrawPixel (y,−x);
DrawPixel (−y,−x);
}
```

# 3 Assignment

- Modify the Bresenham's algorithm for line to work on all quadrants of the clockwise circle. This document describes only the first half of the first quadrant.
- Draw a line made rectangle using the Bresenham's algorithm. Set a random color to each pixel before drawing it on the computer display.

# Laboratory work 7: 2D transformations

## 1 Objectives

This laboratory presents the key notions on 2D transformations (translation, scale, rotation). These transformations can be used to modify the parameters of 2D objects.

## 2 Theoretical background

### 2.1 Defining 2D points

A 2D point is defined in a homogenous coordinate system by (x*w, y*w, w). For simplicity in bi-dimensional systems the w parameter is set to 1. Therefore, the point definition is (x, y, 1). Another representation for the point is the following:

$$P = [x \quad y \quad 1]$$

### 2.2 Translation

The translation transformation is used to move an object (point) by a given amount.

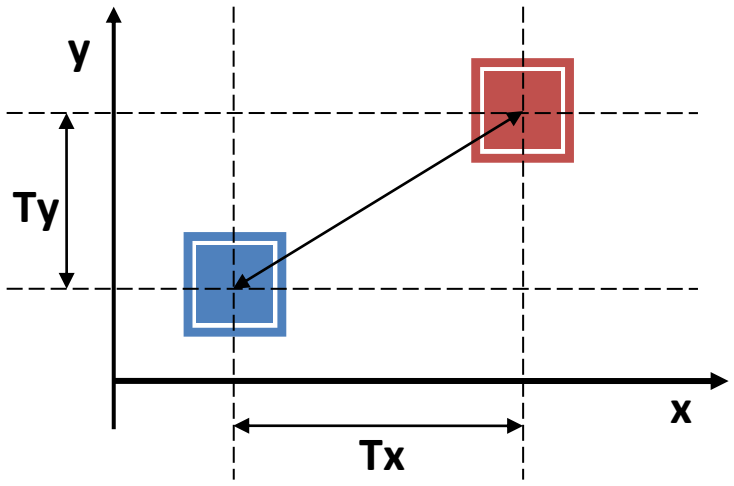


Figure 7.1: Illustration of translation transformation

The matrix for the translation operation is the following:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

where Tx and Ty represent the translation factors on x and y axes. If we apply the transformation to the 2D point, $P' = P * T$, we obtain the new coordinates for that point:

$$x' = x + Tx$$

$$y' = y + Ty$$

The matrix for the inverse transformation is the following:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Tx & -Ty & 1 \end{bmatrix}$$

## 2.3 Scale

The scale transformation enlarges or reduces an object. The transformation is *relative to the origin*.

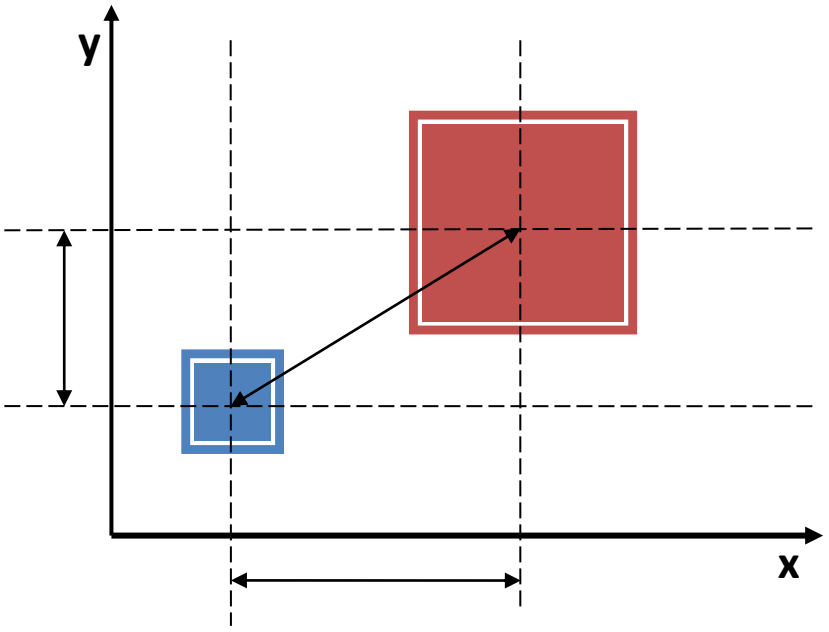


Figure 7.2: Illustration of scale transformation

The matrix for the scale operation is the following:

$$S = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where Sx and Sy represent the scale factors on x and y axes. If the Sx and Sy factors are equals then the scaling transformation is uniform. If the Sx and Sy factors are not equals then the scaling transformation is non-uniform. If we apply the transformation to the 2D point, $\mathrm{P}' = \mathrm{P} * \mathrm{S}$, we obtain the new coordinates for that point:

$$x' = x * Sx, y' = y * Sy$$

If you set the scaling factors to +/- 1 then you can reflect the original shape.

The matrix for the inverse transformation is the following:

$$S = \begin{bmatrix} 1/Sx & 0 & 0 \\ 0 & 1/Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

## 2.4 Rotation

This transformation rotates an object with a given angle. This transformation is also relative to the origin.

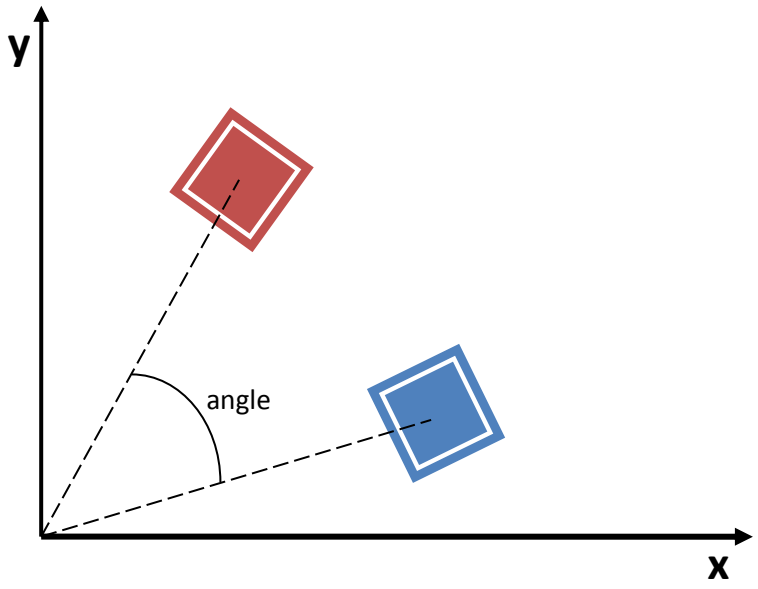


Figure 7.3: Illustration of rotation transformation

The matrix for the rotation operation is the following:

$$R = \begin{bmatrix} \cos \propto & \sin \propto & 0 \\ -\sin \propto & \cos \propto & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\propto$ represent the rotation angle. If we apply the transformation to the 2D point, $\mathrm{P}' = \mathrm{P} * \mathrm{R}$, we obtain the new coordinates for that point:

$$x' = x * \cos \propto \; - y * \sin \propto$$

$$y' = x * \sin \propto \; + y * \cos \propto$$

The matrix for the inverse transformation is the following:

$$R = \begin{bmatrix} \cos \propto & -\sin \propto & 0 \\ \sin \propto & \cos \propto & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

## 2.5 Rotation around an arbitrary point

To rotate a point around an arbitrary point ($x_c$, $y_c$) with an angle $\propto$ we perform the following steps:

1. Change the origin of the coordinates system so that the point ($x_c$, $y_c$) is the new origin
2. Rotate the point with $\propto$ angle
3. Change back the origin of the coordinates system.

## 2.6 Shear

The shear transformation distorts the shape of an object.

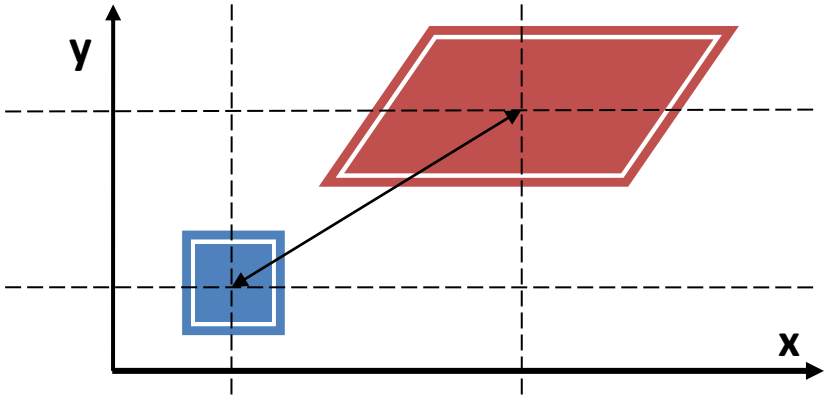


Figure 7.4: Illustration of shear transformation

The matrix for the shear operation is the following:

$$Sh = \begin{bmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where Tx and Ty represent the translation factors on x and y axes. If we apply the transformation to the 2D point, $\mathrm{P}' = \mathrm{P} * \mathrm{Sh}$, we obtain the new coordinates for that point:

$$x' = x + g * y$$

$$y' = y + h * x$$

## 2.7 Reflections

Another transformation that can be applied to object is the reflection transformation.

The matrix for the transformation around the x axis is the following:

$$Ry = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

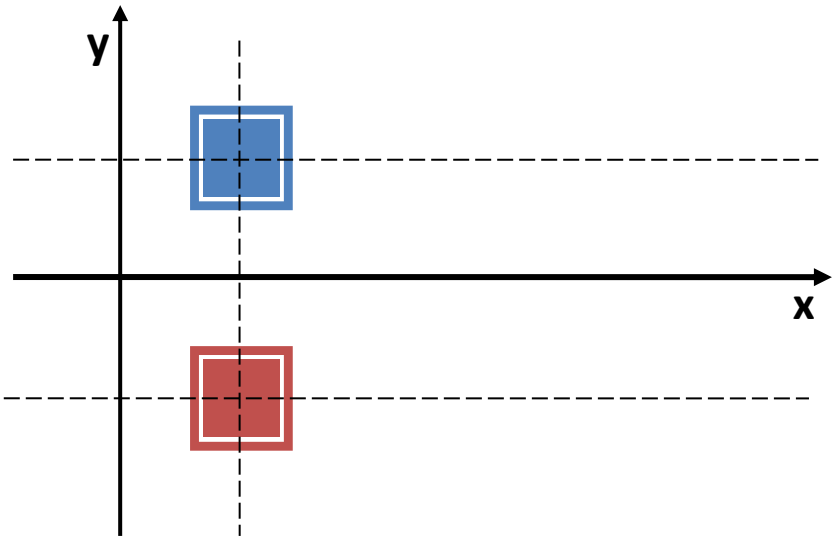


Figure 7.5: Illustration of reflection transformation over X axes

The matrix for the transformation around y axis is the following:

$$Ry = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

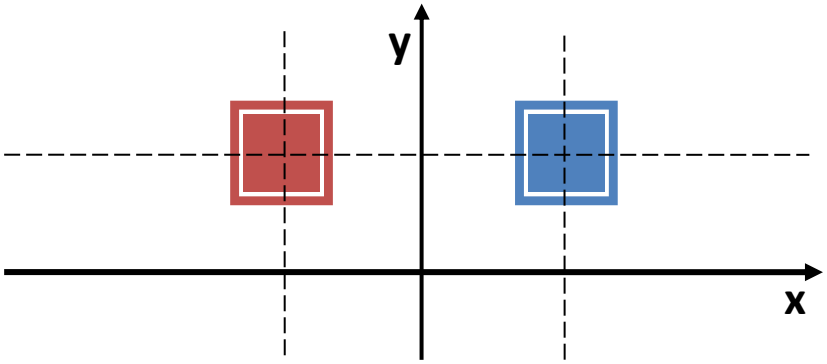


Figure 7.6: Illustration of reflection transformation over Y axes

## 3 Assignment

- Create an application to exemplify the 2D transformations. Define the parameters for each transformation using keyboard inputs.

# Laboratory work 8: 3D transformations

## 1 Objectives

This laboratory presents the key notions on 3D transformations (translation, scale, rotation) using homogenous coordinates. Another topic is related to the projection transformation.

## 2 Theoretical background

### 2.1 Translate, scale and rotate 3D geometrical figures

The following mathematical methods could be used to describe 3D objects and display them onto bi dimensional surfaces (computer monitor). This action requires some visualization transformations, called projections.

First we will extend the 2D translation, scaling and rotation formulas to fit the corresponding 3D transformations. Using homogenous coordinates we obtain the following translation matrix for a 3D point.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

For scaling a 3D point we could use:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

If we choose to make a rotation of a 3D point $P(x, y, z)$ around X, Y and Z axis the following formulas should be used:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & \cos(a) & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where *a* is the rotation angle.

## 2.2 3D rotation around an arbitrary line

In order to make such a rotation three steps must be taken into account:

a. Translate the system origin to a point from the rotation line;
b. Rotate the object around X, Y and Z axes;
c. Change the system origin to its initial position.

This operation could also be described using arithmetical methods. First of all let us denote the segment line this way:

$$\begin{cases} x = Au + x_1 \\ y = Bu + y_1 \\ z = Cu + z_1 \end{cases}$$

where *u* is a real parameter.

Using this notation we could specify a segment point $(x_1, y_1, z_1)$, and the segment line direction as $(A, B, C)$.

To change the system origin to an $(x_1, y_1, z_1)$ point we must first translate all the system points by $(-x_1, -y_1, -z_1)$:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 1 \end{bmatrix}$$

The second step in rotating a 3D geometrical figure around an arbitrary line is the rotation transformation itself. Let us exemplify this procedure by describing the mathematical model used for rotating the object around the X axis. It is important to maintain the rotation process until the segment line is entirely inside XOZ plane. To determine the rotation angle for this operation, we will place the direction vector in the new established system origin and we'll consider its projection in the YOZ plane. The projection length could be computed as $V = \sqrt{B^2 + C^2}$.

Because $\sin(I) = \frac{B}{V}$ and $\cos(I) = \frac{C}{V}$, the object rotation around the X axis could be modeled as:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{C}{V} & \frac{B}{V} & 0 \\ 0 & -\frac{B}{V} & \frac{C}{V} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{C}{V} & -\frac{B}{V} & 0 \\ 0 & \frac{B}{V} & \frac{C}{V} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can consider that the rotation axis is in XOZ plane. The segment length is unmodified by the rotation transformation $L = \sqrt{A^2 + B^2 + C^2}$. The z coordinate has $V = \sqrt{L^2 - A^2} = \sqrt{B^2 + C^2}$. The following step will rotate the object around the Y axis in order to overlap the Z axis over the initial rotation axis. Because $\sin(J) = \frac{A}{L}$ and $\cos(J) = \frac{V}{L}$ the rotation matrix could be defined as:

$$R_y = \begin{bmatrix} \frac{V}{L} & 0 & \frac{A}{L} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{A}{L} & 0 & \frac{V}{L} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y^{-1} = \begin{bmatrix} \frac{V}{L} & 0 & -\frac{A}{L} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{A}{L} & 0 & \frac{V}{L} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When a 3D points rotates around the Z axis the following transformation matrix should be used:

$$R_z = \begin{bmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix around an arbitrary axis is computed using the following formula:

$$R_o = T \cdot R_x \cdot R_y \cdot R_z \cdot R_y^{-1} \cdot R_x^{-1} \cdot T^{-1}$$

## 2.3 Geometrical projections

The following sections of this laboratory detail the most used projections types used in computer graphics. Using different projections techniques an N-dimensional points array is transformed into an M-dimensional one, where M < N. In this case N = 3 (3D points array) and M = 2 (2D points array). Because the computer monitor doesn't support 3D representations the parallel and perspective projections should be used.

### 2.3.1 Parallel projection

Based on the projection direction onto the projection plane there are two parallel projection types, orthogonal parallel projection and oblique parallel projection.

## 1. Orthogonal parallel projection:

Are very common in the technical design, where the projection plane is perpendicular to one of the reference system axes that contain the object. This type of projection keeps the object shapes, dimensions and angles undistorted.

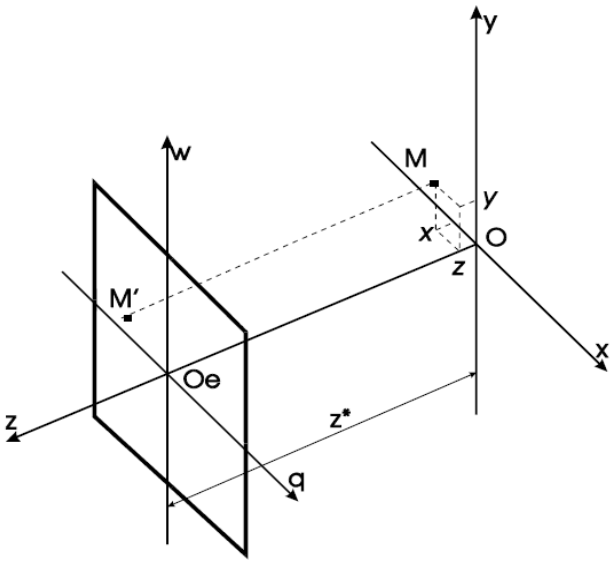


Figure 8.1: Projection plane is parallel with the OZ axis

When the projection plane is parallel with the OZ axis (Figure 8.1) the $M'(q,w)$ point is computed based on the $M(x,y,z)$, using the following coordinates relationship:

$$\begin{cases} q = x \\ w = y \\ z = z^* \end{cases}$$

where $z^*$ is the projection plane height.

The same relationship could be translated into homogenous coordinates:

$$\begin{bmatrix} q \\ w \\ z^{'} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & z^{*} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

## 2. Oblique parallel projection

Figure 8.2 defines an oblique projection through the point $M^{'}$, obtained by $M(0,0,1)$ point projection.

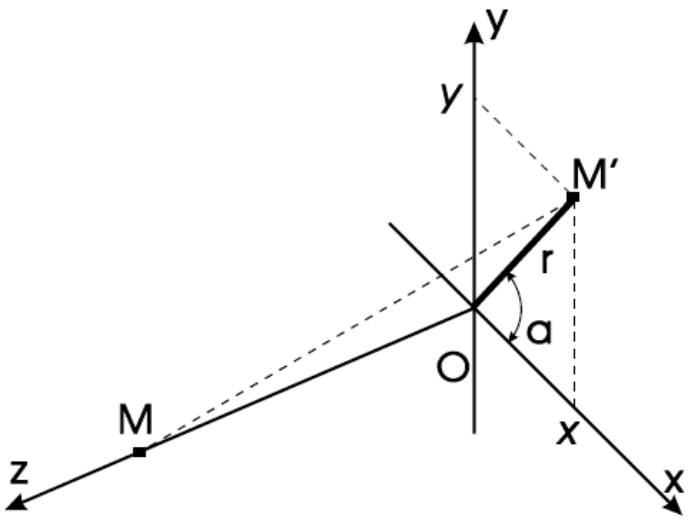


Figure 8.2: Oblique projection through the point M'

The mathematical relationship between $M$ and $M^{'}$ points coordinates could be described as:

$$\begin{cases} x = r \cdot \cos(a) \\ y = r \cdot \sin(a) \\ z = 0 \end{cases}$$

Giving the point $P(x, y, z)$ its projection is the $P'(x', y', z')$ point:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & r \cdot \cos(a) & 0 \\ 0 & 1 & r \cdot \sin(a) & 0 \\ 0 & 0 & 0 & z^* \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Three cases can be distinguished based on the *r* value:

a. $r = 0$: orthogonal projection;
b. $r = 0.5$: cabinet projection. Offers the most photorealistic images from all of the three cases;
c. $r = 1$: cavalier projection.

### 2.3.2 Perspective projection

This type of projection creates images similar to the ones obtained when using the photographic technique (Figure 8.3). Perspective projection gives a more realistic image representation than the ones obtained through the parallel projection. The main disadvantage is that it deforms the objects by distorting their angles and dimensions.

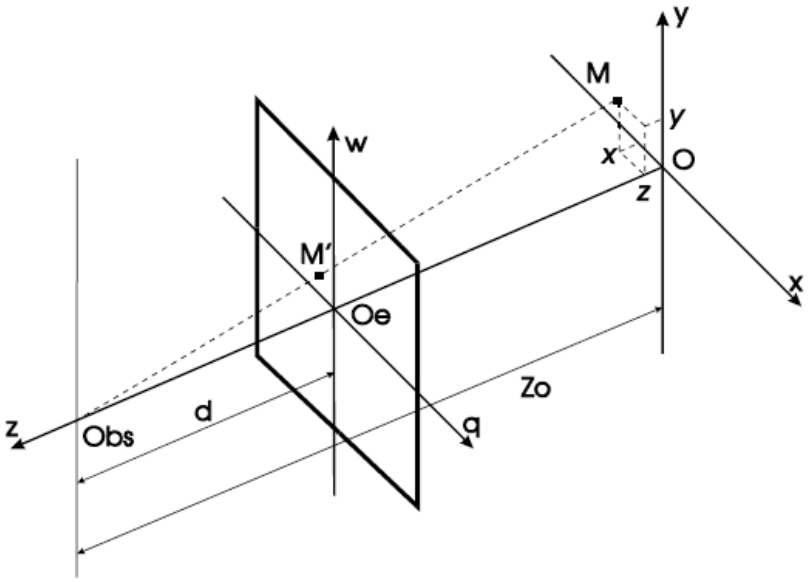


**Figure 8.3: Perspective projection**

Perspective projection accumulates the parallel and unparallel into a "vanishing point". If the visual receptor is fixed than it can see all the objects inside the "frustum". In the graphics applications the frustum is replaced by a "pyramid of view". We can imagine that the observer looks to the world through a rectangular window from an opaque plan located at a distance *d* from the observer.

Let the point $M(x, y, z)$ be in the landmark observation. The projection's coordinates $M^{'}(q, w, z^{'})$ from the projection plan will be:

$$q = \frac{d}{z_0 - z} \cdot x$$

$$w = \frac{d}{z_0 - z} \cdot y$$

$$z' = z_0 - d$$

For reducing the parameters number taken into account, usually $d=z_0$, the transformation written in homogenous coordinate becomes:

$$\begin{bmatrix} q \\ w \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{z_0}{z_0 - z} & 0 & 0 & 0 \\ w & \frac{z_0}{z_0 - z} & 0 & 0 \\ 0 & 0 & 0 & z^* \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we consider the fraction $\frac{z_0}{z_0 - z}$ where we add and subtract *z* to the counter, we will obtain:

$$q' = x \cdot \left(1 + t \cdot \frac{z}{z_0 - z}\right)$$

$$w' = y \cdot \left(1 + t \cdot \frac{z}{z_0 - z}\right)$$

where $\frac{z_0}{z_0 - z}$ it's called *elongation factor with distance* and **t** will be *optical deflection factor* (*or correction*).

There will be the next cases:

a. $t = 0$ : orthogonal parallel projection;
b. $r = 1$ : perspective projection;
c. $t > 1$: the perspective effect is exaggerated, obtaining distortion and deformations of the image like the "fish-eye" large angular aperture objectives.

# 3 Tutorial

## 3.1 Matrix multiplication function

The following pseudo code implements the matrix multiplication function. The function takes three parameters that represent the two multiplicative matrixes and the result. The following example handles only square matrixes with four lines and four columns.

```
double matrix[4][4];

Algorithm matrix_mltiplication (matrix m1, matrix m2, matrix
res)
{
     int i, j, k;
     matrix restmp;

     //Initialize the result matrix
     for(i = 0; i < 4; i++)
        for(j = 0; j < 4; j++)
            restmp[i][j] = 0;

     for(i = 0; i < 4; i++)
        for(j = 0; j < 4; j++)
           for(k = 0; k < 4; k++)
                 restmp[i][j] += m1[i][k]*m2[k][j];

     for(i = 0; i < 4; i++)
        for(j = 0; j < 4; j++)
           res[i][j] = restmp[i][j];
}
```

## 3.2 Perspective and oblique projection implementation function

The following two functions could be used to implement the perspective or oblique projection when applying a 3D transformation for a geometrical figure.

```
typedef struct pct{
    int x,y,z;
} Pct;

Algoritm pr_oblica(Pct* p, int r, int a)
// a must be defined in degrees
{
   float u;
   u=(3.141592*a)/180; //transform a angle in radians
```

```
    p->x = p->x + p->z * r * cos(u);
    p->y = p->y + p->z * r * sin(u);
    p->z = 0;
}
```

```
typedef struct pct{
    int x,y,z;
} Pct;

Algoritm pr_persp(Pct* p, int zo)
{
    float zz;
    zz = (float)zo / (float)(zo - p->z);
    p->x = p->x * zz ;
    p->y = p->y * zz ;
    p->z = 0 ;
}
```

# 4 Assignment

- Using the formulas contained in this laboratory translate, scale and rotate a cube. Rotate the cube around X, Y, Z and arbitrary axes. The perspective projection should be used.
- Highlight different projections features using the two pseudo code algorithms for the perspective and oblique projection.

# Laboratory work 9: Line clipping algorithm

## 1 Objectives

The study and implementation of clipping algorithms for graphical primitives: line and point. This paper presents the Cohen-Sutherland and Bisection Method algorithms.

## 2 Theoretical background

When computing for display a large image that exceeds the boundaries of the display area, some parts of it will be invisible for the user. In order to speed up the displaying process these parts have to be removed from the computation using 2D-clipping algorithms.

### 2.1 Determine if a point P(x,y) is visible

Considering P1($x_{min}$, $y_{min}$), and P2( $x_{max}$, $Y_{max}$) the defining points of the visible area rectangle, the point P(x,y) is visible only if the following conditions are met:

$$x_{min} \leq x \leq x_{max}$$

$$y_{min} \leq y \leq y_{max}$$

### 2.2 Determine if a segment is visible

In order to determine if a line segment is visible, we need slightly more complex algorithms. One idea would be to test the visibility of each point of the segment, before displaying it on the screen. But this method will require a lot of time and very many computations. The method can be easily improved by testing first the heads of the segment. If both these points are in the visible area, the entire segment will be visible. This case is called “simple acceptance”. On the same logic, if both points are outside and on the same side of the visible area, no part of the segment will be visible. This case is called “simple rejection”. In all the other cases we must use other algorithms to establish which part of the segment (if any) is visible.

## 2.3 Cohen-Sutherland clipping algorithm

One of the most efficient algorithms that can be used in these cases is Cohen-Sutherland (CS clipping). If a line segment cannot be included either in “simple acceptance” or “simple rejection” cases, then we have to compute its intersection points with the following lines:

$$y = y_{max}, x = x_{max}, y = y_{min}, x = x_{min}$$

and to eliminate the segments that are placed outside the visible area. As a result, we will obtain a new line segment. The algorithm is repeated until the resulted segment can be included in one of the “simple acceptance” or “simple rejection” cases.

The Cohen-Sutherland algorithm uses a four digits code to describe each head of the segment. The code has the following structure:

- the first digit is 1 if the point is above the visible area; otherwise is 0
- second digit is 1 if the point is under the visible area; otherwise is 0
- third digit is 1 if the point is on the right of the visible area; otherwise is 0
- fourth digit is 1 if the point is on the left side of the visible area; otherwise is 0

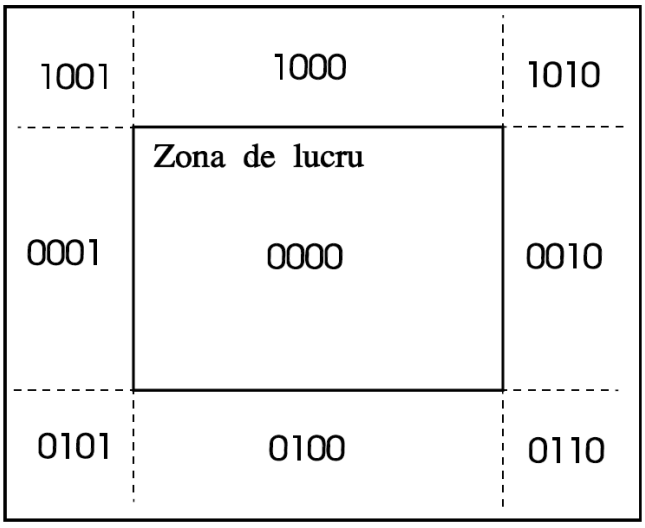


Figure 9.1: Illustration of Cohen-Sutherland algorithm code

# 3 Cohen-Sutherland algorithm implementation

```
read x1, y1, x2, y2, xmin, ymin, xmax, ymax
define the Boolean variables: REJECTED, DISPLAY, FINISHED and
make them all FALSE
repeat until FINISHED = TRUE
{
     //compute the 4 digits code for P(x1, y1)
     COD1 = computeCScode(x1, y1)
     // compute the 4 digits code for P(x2, y2)
     COD2 = computeCScode(x2, y2)
     //test for simple rejection case
     RESPINS = SimpleRejection(COD1, COD2)
     if RESPINS = TRUE
          FINISHED = TRUE
     else
     {
          //test for simple acceptance case
          DISPLAY = SimpleAcceptance(COD1, COD2)
          if DISPLAY = TRUE
               FINISHED = true
          else
          {
               //if P(x1, y1) is inside the display area,
               //invert P(x1, y1) and P(x2, y2) together with
               //their 4 digits CS codes
               invert(x1,y1,x2,y2,COD1,COD2)

               //eliminate the segment above the display area
               if(COD1[1] = 1) and (y2 <> y1)
               {
                    x1 = x1+(x2-x1)*(Ymax-y1)/(y2-y1)
                    y1 = Ymax
               }
               //eliminate the segment under the display area
               elseif(COD1[2] = 1) and (y2 <> y1)
               {
                    x1 = x1+(x2-x1)*(Ymin-y1)/(y2-y1)
                    y1 = Ymin
               }
               //eliminate the segment on the right
               //of the display area
               elseif(COD1[3] = 1) and (x2 <> x1)
               {
                    y1 = y1+(y2-y1)*(Xmax-x1)/(x2-x1)
                    x1 = Xmax
               }
               //eliminate the segment on the left
               //of the display area
```

```
                elseif(COD1[4] = 1) and (x2 <> x1)
                {
                      y1 = y1+(y2-y1)*(Xmin-x1)/(x2-x1)
                      x1 = Xmin
                }
           }
      }
}

if DISPLAY = TRUE
{
      round(x1)
      round(x2)
      round(y1)
      round(y2)
      DrawLine(x1, y1, x2, y2)
}
```

# 4 Assignment

- Define a display area and mark it with a rectangle. Display a segment that can be included in “simple acceptance” case, one for “simple rejection” case and other 5 segments for more complex cases. Display with thicker line the removed parts as a result of Cohen-Sutherland algorithm.
- Extend the above request by allowing the user to define the display area using the mouse.

# Laboratory work 10: Polygon clipping algorithms

## 1 Objectives

Study, implement and evaluate the Sutherland-Hodgman and Weiler-Atherton clipping algorithms for polygons, in 2D object coordinate system.

## 2 Theoretical background

There are two main types of clipping algorithms against the margins of a display window, according to the coordinate space where we compute the operations:

a. Raster algorithms, which operate in video memory. With these algorithms the clipping is computed for each pixel. Even if the algorithms themselves are pretty simple their implementation requires a large number of accesses to the video memory.
b. Vectorial algorithms, which operate with the nodes describing the polygon. These clipping algorithms work directly with the data structure describing the polygons and will result in one or more new polygons described through a list of nodes.

The second types of algorithms involve more complex computations which are executed in the main system memory but their results is compatible with any graphical system as is described generally through a list of points. Two of the vectorial algorithms are very often used: Sutherland-Hodgman and Weiler-Atherton.

## 3 Sutherland-Hodgman clipping algorithm

Sutherland-Hodgman algorithm considers that the initial polygon is defined through a list of nodes inv[] = {v1, v2, …, vn}. A conventional direction of nodes inspection is determined, for example: v1v2, v2v3, …, vnv1. The clipping of the polygon is finalized in four steps. At each step, all the edges of the polygon are clipped against one side of the working area. In the end, we will obtain a list of points outv[]={v1', v2' ,…, vp'} that describe the clipped polygon.

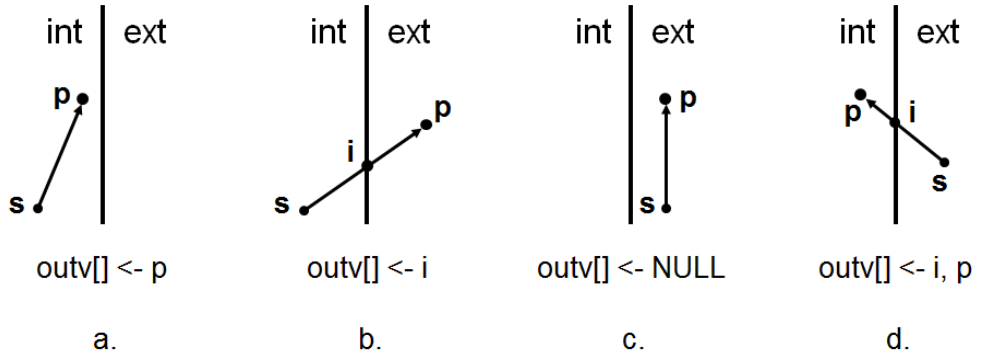


**Figure 10.1: Relationships between the margins of the display area and any edge of a polygon**

We can identify four different relationships between the margins of the display area and any edge of a polygon (see Figure 10.1). We will consider **s** to be the initial node and **p** the final node of the edge. The four cases are:

a. Both nodes **s** and **p** are inside the display area. Node **p** will be added to the list of clipped nodes: outv[] <- p.
b. Node **s** is inside the display area while **p** is outside. We have to compute the intersection point **i** between the margin of the display area and the edge described by **s** and **p**. Node **i** will be added to the list of clipped nodes: outv[] <- i.
c. Both nodes **s** and **p** are outside the display area. Nothing will be added to the list of clipped nodes.
d. Node **s** is outside the display area while **p** is inside. We have to compute the intersection point between the margin of the display area and the edge described by **s** and **p**. We will add both **p** and **i** to the list of clipped nodes: outv[] <- p, outv[] <- i.

## 3.1 Sutherland-Hodgman algorithm pseudo-code description

```
Clipping SH(nodesList: inv, outv;
            displayAreaMargins: margine_dec[4])
{
     for j=0,4
     {
          ClipMarginSH(inv, outv, margine_dec[j]);
          //update the current nodes list
```

```
          inv = outv;
     }
}

ClipMarginSH(nodesList: inv, outv;
             displayAreaMargin: margine_dec)
{
     node i, p, s;
     s = last node from inv;
     for p=each node in inv
     {
         //cases A and D
          if(InDisplayArea(p, margine_dec))
          {
               //case A
               if(InDisplayArea(s, margine_dec))
               {
                    add p to outv
               }
               //case D
               else
               {
                    i = IntersectionPoint(s, p, margine_dec);
                    add i to outv
                    add p to outv
               }
          }
          //case B
          else if(InDisplayArea(s, margine_dec))
          {
               i = IntersectionPoint(s, p, margine_dec);
               add i to outv
          }
          //we do nothing for case C
          //update the starting point
          s = p;
     }
}
```

As you can see, the algorithm eliminates from the display list the parts of the polygons that are placed into the exterior half-plane determined by the display area margin.

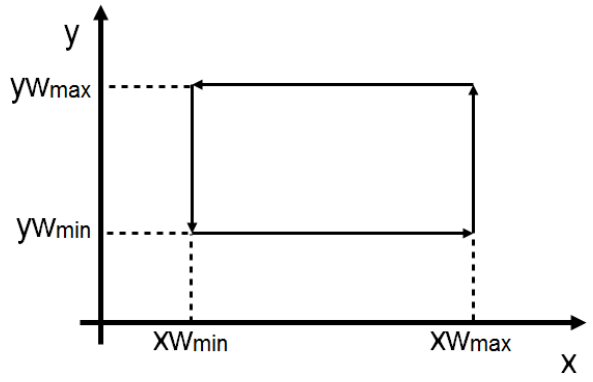


**Figure 10.2: Representation of the display area and coordinates of the half planes**

The function **InDisplayArea(node: p; displayAreaMargin: margine_dec)** returns **true** if point **p** is in the same half-plane with the display area related to the line described by **margine_dec**. For simplicity we will consider a conventional direction for testing margins of the display area. Let this be the trigonometric direction. While we keep the same order into the display area margins list we can check the position of **p** according to Figure 10.2. This way, the function can be described:

```
InDisplayArea(node: p;  displayAreaMargin: margine_dec)
{
      switch(margine_dec)
      {
            case right_margin:
               if(xp < xmargine_dec) return true; break;
            case top_margin:
               if(yp < ymargine_dec) return true; break;
            case left_margin:
               if(xp > xmargine_dec) return true; break;
            case bottom_margin:
               if(yp > ymargine_dec) return true; break;
      }
}
```

# 4 Weiler-Atherton clipping algorithm

Weiler-Atherton algorithm considers that the initial polygon is defined through a list of nodes inv[] = {v1, v2, …, vn}. After the clipping algorithm is applied we will

obtain zero, one or more polygons, each defined through a list of nodes outvk[] = {$v_{k1}'$, $v_{k2}'$, ..., $v_{kp}'$}.

If the initial polygon intersects the margins of the display area, the result polygon will contain at least a portion of an edge of the initial polygon and portions from the display area margins. If the polygon is entirely outside of the display area, the result will be empty.

## 4.1 General description of the algorithm

The algorithm starts from one node of the polygon. Let us consider the polygon from Figure 10.3 and v1 as the starting node. We will use **i** to count the resulting polygons. For the beginning, i = 1.

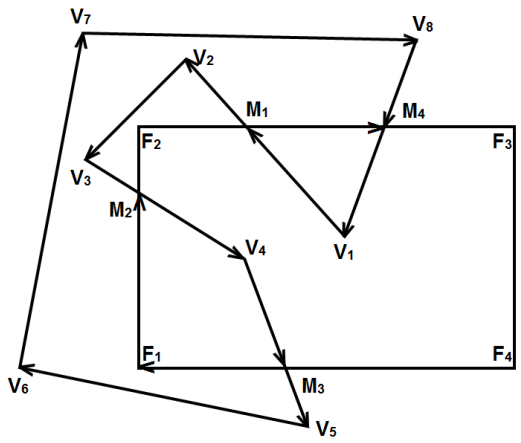


**Figure 10.3: Representation of all the nodes and intersection points**

1. We go through all the polygon's nodes in a conventional order, for example v1v2, v2v3…. As v1 in inside the display area we add it to the results nodes list for the first polygon outv[1][] <- v1.
2. We continue to check all the nodes of the initial polygon, in the previously established conventional order, until we get out of the display area. We add all these nodes to the outv[1][] list.
3. If we consider M1 to be the exit point on the v1v2 edge, we add M1 to outv[1][].
4. We will continue to check the margin of the display area which is inside the polygon, until we meet the first intersection with the initial polygon.

We add this intersection point (M4 in Figure 10.3Figure) to the outv[1][] list, as it represents the entry point of the polygon into the display area.

5. We then continue to check the nodes of the initial polygon which are inside the display area until we get back again to v1 or we get out again, in which case we go back to step 3 of the algorithm.
6. If v1 has been reached, we will obtain the first result polygon. For our example: outv[1][] = v1, M1, M4 (,v1).
7. We can go further to the next polygon: i = i + 1. Our new starting point will be M1.
8. We go through all the polygon nodes, in the conventional order, until we discover the first entry point (M2 in our example).
9. Starting with M2 we begin to construct a new polygon outv[2][]. In our example outv[2][] = M2, v4, M3, F1 (, M2).

## 4.2 Weiler-Atherton algorithm implementation example

One possible implementation of the Weiler-Atherton algorithm could be:

1. Create a list (**Ipp**) with the nodes of the initial polygon.
2. Create another list (**Ipf**) that contains the corners of the display area.
3. Compute the intersection point of each polygon edge with the margins of the display area and add the resulting nodes to the **Ipp** and **Ipf** lists.
4. Add to the **Ipf** the nodes of the polygon which reside on the margins of the display area.
5. Create the list of polygon's edges (**ILp**) which will keep for each edge a reference to two consecutive nodes of the polygon.
6. We check each edge from **ILp** to determine if it is:
    a. Inside the display area
    b. Outside the display area
    c. On one of the margins of the display area
7. For each point in **Ipp** we determine if it is placed:
    a. Inside the display area
    b. Outside the display area
    c. On one of the margins of the display area
8. For each point in **Ipf** we determine if it is placed:
    a. Inside the polygon
    b. Outside the polygon
    c. On one of the edges of the polygon

9. We determine the computation order of the points in **Ipf** list, keeping the result in var **sens**.
    a. sens = LEFT, the next element is ->urm
    b. sens = RIGHT, the next element is -> pred
    c. sens = NEDEF
10. if sens = NEDEF then
        //q0 is the list of result polygons
        if(*polygon inside the window*)
            //each polygon is represented by a list of edges
            q0 = ILp
        else
            q0 = null

    else
        for(each element of **ILp**) do
            Atherton(current_element of ILp);
            current_element = current_element -> urm;
11. Stop.

```
Atherton (latura: elem)
{
     if (elem is outside the window)
     {
          if(newp = true)  secventa();
     }
     else if(elem is inside the window)
     {
          if(newp = false)
          {
               newp = true;
               //q0 is the list of result polygons
               add elem to q0;
          }
          else
          {
               add elem to q0;
               if(elem->urm is inside the window) secventa();
          }
     }
     //the point is on one margin of the display area
     else
     {
```

```
            if(newp = true) secventa();
      }
}
```

The function **secventa()** should:

1. Locate in **lpf** the node that represents the beginning of the segment that is being analyzed.
2. Go through **lpf** in the conventional chosen direction until identifies a common point with **lpp**, adding to q0 each new edge from two consecutive points from **lpf**
3. if(the last point is the same with the starting point in q0)

   *//newp is true while we are building on the same polygon*

   then newp = false;

# 5 Assignment

- Define a display area and mark it with a rectangle. Display a polygon and clip it against the display area previously defined using:
  - Sutherland-Hodgman algorithm
  - Weiler-Atherton algorithm
- Extend the above request by allowing the user to define the display area using the mouse.

# Laboratory work 11: Bezier curves

## 1 Objectives

This laboratory presents the key notions on Bezier curves.

## 2 Theoretical background

### 2.1 Bezier curves

A Bezier curve is a parametric curve, used in computer graphics and other related fields, used to model smooth curves that can be scaled indefinitely. Another applicability of the Bezier curves is in animations where an object movement can be defined by using a Bezier curve, modifying in this way the velocity of the object.

### 2.2 Cubic Bezier curves

In order to define a cubic Bezier curve we need 4 points. The curve will pass through P0 and P3, which are the starting point and the ending point of the curve. The curve will not pass through P1 and P3 which are control points and are used to provide directional information.

$$\boldsymbol{B}(t) = (1-t)^3\boldsymbol{P}_0 + 3(1-t)^2t\boldsymbol{P}_1 + 3(1-t)t^2\boldsymbol{P}_2 + t^3\boldsymbol{P}_3, t \in [0,1]$$

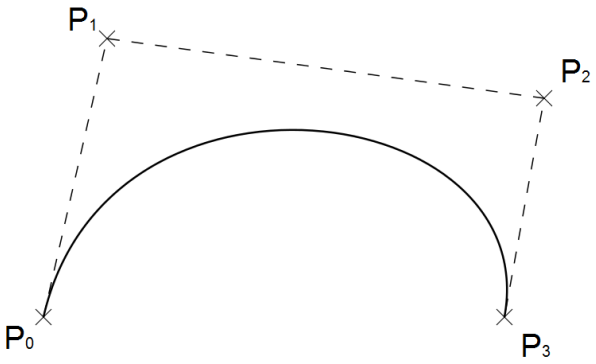


Figure 11.1: Bezier curve with four points

## 2.3 Generalization

The Bezier curve of degree n can be generalized in this way:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i$$

For example, for n = 5:

$$B(t) = (1-t)^5 P_0 + 5t(1-t)^4 P_1 + 10t^2(1-t)^3 P_2 + 10t^3(1-t)^2 P_3 + 5t^4(1-t)P_4 + t^5 P_5 ,$$

$$t \in [0,1]$$

Recursively the formula can be expressed as follows:

$$B(t) = B_{P0P1\ldots Pn}(t) = (1-t)B_{P0P1\ldots Pn-1}(t) + tB_{P1P2\ldots Pn}(t)$$

## 2.4 Bezier curves in Win32 GDI API

The PolyBezier function draws one or more cubic Bezier curves.

```
BOOL PolyBezier(
  __in  HDC hdc,
  __in  const POINT *lppt,
  __in  DWORD cPoints
);
```

The first Bezier curve is drawn from the first point to the fourth point. The second and the third points are control points.

# 3 Assignment

- Create an application to exemplify the Bezier curves.

# Quiz

1. Which of the following differences between the functions TextOut and TextOutA are real:

A. TextOutA is a new and better implementation of the TextOut function
B. TextOutA enables the customization of more attributes than TextOut (for example font size, font type, text italic/bold etc.)
C. TextOut displays the text at the current position while TextOutA requires to specify X and Y coordinates
D. TextOut displays UNICODE strings while TextOutA displays ANSI encoded strings

2. Is it possible to use the Rectangle function with 3D points (ex.: P(X, Y, Z)) without applying the Perspective/Orthogonal projection first?

A. True
B. False

3. What is the default color of pixels belonging to a bitmap created using the CreateCompatibleBitmap function:

A. RGB(255, 255, 255);
B. RGB(0, 0, 0);
C. The color of each pixel is undefined until the first call to the FloodFill function;
D. The color of each pixel depends on the values previously stored in the memory region that will be assigned to the bitmap;

4. The LoadImage function can be used to:

A. load icons
B. load bitmaps from JPG files
C. draw mouse cursors
D. load animated mouse cursors

5. Having the following code:

```
SendMessage(comboBoxHWND, CB_RESETCONTENT, 0, 0);
SendMessage(comboBoxHWND, CB_ADDSTRING, 0,
(LPARAM)TEXT("Polyline"));
SendMessage(comboBoxHWND, CB_ADDSTRING, 0,
(LPARAM)TEXT("Circle"));
SendMessage(comboBoxHWND, CB_ADDSTRING, 0,
(LPARAM)TEXT("Rectangle"));
SendMessage(comboBoxHWND, CB_ADDSTRING, 0,
(LPARAM)TEXT("Line"));
```

and the Sort attribute of comboBoxHWND is false, which of the following statements will select Rectangle:

A. SendMessage(comboBoxHWND, CB_SELECTED, 2, 0);
B. SendMessage(comboBoxHWND, CB_SELECTED, 3, 0);
C. SendMessage(comboBoxHWND, CB_SETCURSEL, 2, 0);
D. SendMessage(comboBoxHWND, CB_SETCURSEL, 3, 0);

6. Choose the correct answers related to the Bresenham's algorithm for line drawing:

A. At step K, the algorithm searches for the pixel that is closest to the current drawn point
B. The algorithm takes into account the octant position of the line end points
C. This algorithm is not used for line drawing
D. It uses the LineTo() function to draw the line
E. The line is drawn pixel by pixel

7. What is the border color of the rectangle and ellipse after executing this piece of code:

```
HPEN hPen;
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
      myPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
      SelectObject(hdc, myPen);
      Rectangle(hdc, 100, 100, 200, 200);
      myPen = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));
      SelectObject(hdc, myPen);
      Ellipse(hdc, 300, 300, 500, 500);
   EndPaint(hWnd, &ps);
break;
```

A. Black color for rectangle and ellipse
B. The "myPen" object is not used for specifying the border color of the primitives. There are other objects that are used for this purpose.
C. Green color - for rectangle, Red color - for ellipse
D. Red color - for rectangle, Green color - for ellipse
E. Red color - for rectangle, Red color - for ellipse

8. The Cohen-Sutherland algorithm is useful for clipping a polygon shape against a rectangular window (input: polygon edges, output: edges of the clipped polygon).

A. True
B. False

9. Given two points, P1 and P2 with their codes computed according to the Cohen-Sutherland algorithm (code1 and code2), which of the following conditions would trigger the acceptance of the line defined by P1 and P2 ?

A. The two codes must differ by at least one digit.
B. All digits of all codes are '0'.
C. Each code should have at least one '1' digit.
D. One of the codes has the form "0000".
E. The two codes are equal.

10. Which of the following is true?

A. Sutherland-Hodgman is a vectorial clipping algorithm.
B. Raster algorithms generally require more complex computations than vectorial algorithms.
C. Weiler-Atherton is a raster clipping algorithm.

# References


1. Windows GDI - https://msdn.microsoft.com/en-us/library/windows/desktop/dd145203%28v=vs.85%29.aspx
2. Hughes J.F., van Dam A., McGuire M., Sklar D.F., Foley J.D., Feiner, S.K., Akeley K., ”Computer Graphics. Principles and Practice". 3rd Edition, Addison-Wesley Pub. Comp., 2013.
3. Shirley P., Ashikhmin M., Marschner S., “Fundamentals of Computer Graphics”, A K Peters/CRC Press, 2009.
4. Foley J.D., van Dam A., Feiner S.K., Hughes J.F., “Computer Graphics: Principles and Practice in C”, 2nd Edition, Addison-Wesley Professional, 1995.
5. Angel E., “Interactive computer graphics: A top-down approach with opengl”, Addison-Wesley, 2002.
6. Blinn J., “Jim Blinn’s Corner”, Morgan Kaufmann, 1996.
7. Watt A., “3D Computer Graphics”, Addison-Wesley, 1993.
8. Bresenham J.E., “Algorithm for computer control of a digital plotter”, IBM Syst. J., 4(1):25–30, 1965.
9. Sutherland I.E., Hodgman G.W., “Reentrant polygon clipping”, Commun. ACM, 17(1):32–42, 1974.
10. Weiler K., Atherton P. “Hidden surface removal using polygon area sorting”, SIGGRAPH Comput. Graph., 11:214–222, 977.
11. Akenine-Möller T., Haines E., Hoffman N., “Real-Time Rendering”, A K Peters, 2008.