RECTANGLES, REGIONS, CLIPPING

Rectangles

A rectangle is a basic geometric shape defined by four coordinates: top, left, bottom, and right. In Windows, rectangles are represented by the RECT structure, which is a type definition for a data structure that holds these four coordinates. Rectangles are often used to define areas on the screen, such as the client area of a window or the boundaries of a control.

Regions

A region is a more complex shape that can be composed of multiple rectangles, polygons, and ellipses. Regions are used to define clipping areas, which are areas of the screen that will be visible when drawing operations are performed. Clipping is important for performance optimization, as it prevents drawing operations from being performed on areas that will not be visible.

Drawing Functions for Rectangles

Windows provides several drawing functions that work with RECT structures. These functions can be used to fill, frame, or invert rectangles.

- FillRect: Fills the specified rectangle with the specified brush.
- FrameRect: Draws a rectangular frame around the specified rectangle using the specified brush.
- InvertRect: Inverts all the pixels in the specified rectangle.

Manipulating RECT Structures

Windows also provides several functions that can be used to manipulate RECT structures. These functions can be used to set the fields of a rectangle, move a rectangle, resize a rectangle, check if a rectangle is empty, and check if a point is inside a rectangle.

- SetRect: Sets the fields of a RECT structure to the specified values.
- OffsetRect: Moves a RECT structure a specified number of units along the x and y axes.
- InflateRect: Inflates or deflates a RECT structure by the specified amounts along the x and y axes.
- SetRectEmpty: Sets all the fields of a RECT structure to 0.
- CopyRect: Copies the contents of one RECT structure to another RECT structure.
- IntersectRect: Calculates the intersection of two RECT structures and stores the result in a third RECT structure.
- UnionRect: Calculates the union of two RECT structures and stores the result in a third RECT structure.
- IsRectEmpty: Checks if a RECT structure is empty.
- PtInRect: Checks if a point is inside a RECT structure.

✅✅✅✅✅✅✅✅✅✅✅✅✅

Example Usage

The following code snippet demonstrates how to use the SetRect function to create a RECT structure and then use the FillRect function to fill the rectangle with a red brush:

```
RECT rect;
HBRUSH brush = CreateSolidBrush(RGB(255, 0, 0));

SetRect(&rect, 10, 10, 100, 100);
FillRect(hdc, &rect, brush);
```

This code will create a red rectangle with a top-left corner at (10, 10) and a bottom-right corner at (100, 100). The rectangle will be filled with the red brush.

The code starts by including the necessary header file, windows.h, which provides access to Windows API functions.

The main() function is the entry point of the program. It initializes the graphics context by obtaining the desktop window handle using GetDesktopWindow() and then retrieving the device context (HDC) using GetDC(hwnd).

A RECT structure called rect is defined to represent rectangles. A solid red brush is created using CreateSolidBrush(RGB(255, 0, 0)).

The code demonstrates the usage of various drawing functions using the defined RECT structure and brush:

FillRect Example:

A rectangle is defined using SetRect(&rect, 10, 10, 100, 100). The rectangle is filled with the red brush using FillRect(hdc, &rect, brush).

FrameRect Example:

Another rectangle is defined using SetRect(&rect, 20, 20, 120, 120). A rectangular frame around the rectangle is drawn using the red brush using FrameRect(hdc, &rect, brush).

InvertRect Example:

A third rectangle is defined using SetRect(&rect, 30, 30, 130, 130). The pixels within the rectangle are inverted using InvertRect(hdc, &rect).

Next, the code demonstrates the usage of functions to manipulate RECT structures:

SetRect Example:

The SetRect(&rect, 40, 40, 140, 140) statement modifies the rect structure to represent a new rectangle.

OffsetRect Example:

The OffsetRect(&rect, 20, 20) statement moves the rectangle defined by rect by 20 units to the right and 20 units down.

InflateRect Example:

The InflateRect(&rect, 10, 10) statement increases the size of the rectangle defined by rect by 10 units in both the horizontal and vertical directions.

SetRectEmpty Example:

The SetRectEmpty(&rect) statement resets the rect structure to represent an empty rectangle.

CopyRect Example:

A new RECT structure, destRect, is defined using RECT destRect;. The CopyRect(&destRect, &rect) statement copies the contents of the rect structure to the destRect structure.

IntersectRect Example:

Two rectangles, rect1 and rect2, are defined using SetRect(&rect1, 10, 10, 50, 50); and SetRect(&rect2, 20, 20, 60, 60);, respectively. The intersection of these two rectangles is calculated and stored in the destRect structure using IntersectRect(&destRect, &rect1, &rect2).

UnionRect Example:

The union of the rectangles rect1 and rect2 is calculated and stored in the destRect structure using UnionRect(&destRect, &rect1, &rect2).

IsRectEmpty Example:

A variable bEmpty is declared to store the result of checking whether the destRect structure represents an empty rectangle. The check is performed using bEmpty = IsRectEmpty(&destRect).

PtInRect Example:

A POINT structure, pt, is defined to represent a point with coordinates (30, 30). A variable bInRect is declared to store the result of checking whether the point pt is inside the destRect rectangle. The check is performed using bInRect = PtInRect(&destRect, pt).

Finally, the resources obtained are released: the device context is released using ReleaseDC(hwnd, hdc), and the brush is deleted using DeleteObject(brush). The program returns 0 to indicate successful execution.

Program code can be found in 7 ... Chapter 5, Rectangle1.c

Random Rectangles

The text discusses how to create a program that continuously draws random rectangles on the screen. It explains that using a traditional message loop, which retrieves messages from the message queue using GetMessage, prevents the program from drawing rectangles as quickly as possible due to the time it takes to process each message.

Alternative Message Loop

To overcome this limitation, the text introduces an alternative message loop that utilizes the PeekMessage function. PeekMessage allows a program to check for messages in the message queue without removing them. This enables the program to draw rectangles during "dead time" when there are no messages in the queue.

PeekMessage Usage

The PeekMessage function takes five parameters:

- A pointer to a MSG structure to receive the message information.
- A window handle, typically set to NULL to check for messages for all windows in the program.
- A message filter minimum, typically set to 0 to allow all messages.
- A message filter maximum, typically set to 0 to allow all messages.
- A flag indicating whether to remove the message from the queue (PM_REMOVE) or not (PM_NOREMOVE).

Alternative Message Loop Structure

The alternative message loop structure looks like this:

```
while (TRUE) {
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
        if (msg.message == WM_QUIT) {
            break;
        }

        TranslateMessage(&msg);
        DispatchMessage(&msg);
    } else {
        // Draw a random rectangle
    }
}
```

In this loop, if a message is present in the queue, it is processed normally. Otherwise, the program draws a random rectangle. This ensures that the program can continuously draw rectangles without being blocked by message processing.

Handling WM_QUIT Message

The WM_QUIT message is explicitly checked in the alternative message loop. When a WM_QUIT message is received, the loop terminates, and the program exits.

💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥

PeekMessage Limitations

PeekMessage cannot be used to remove WM_PAINT messages from the message queue. This is because WM_PAINT messages are triggered by invalid regions in the window's client area. To remove a WM_PAINT message, you need to validate the invalid regions using ValidateRect, ValidateRgn, or a BeginPaint and EndPaint pair.

Preemptive Multitasking

The text explains that PeekMessage was more important in earlier versions of Windows due to non-preemptive multitasking. With preemptive multitasking in Windows 98, programs can create multiple threads of execution, allowing them to handle messages and other tasks simultaneously.

Random Rectangles Program (RANDRECT)

The text concludes by introducing the RANDRECT program, which continuously displays random rectangles using the PeekMessage-based message loop.

The RandRect video is in the chapter 5 folder.

Creating and Painting Regions

Regions are GDI objects that represent areas of the display composed of rectangles, polygons, and ellipses. They are primarily used for clipping, which restricts drawing operations to a specific portion of the client area. Regions can also be used for direct drawing operations.

Creating Regions

Regions are created using various CreateRgn functions, each tailored to specific shapes:

- CreateRectRgn: Creates a rectangular region from specified coordinates.
- CreateEllipticRgn: Creates an elliptical region from specified coordinates.

- CreateRoundRectRgn: Creates a rectangular region with rounded corners.
- CreatePolygonRgn: Creates a polygonal region from an array of points.
- CreatePolyPolygonRgn: Creates multiple polygonal regions from an array of points.

Combining Regions

The CombineRgn function combines two source regions and stores the resulting combined region in a destination region. The iCombine parameter specifies the combination mode:

- RGN_AND: Intersect the two regions.
- RGN_OR: Union the two regions.
- RGN_XOR: Combine the regions excluding the overlapping area.
- RGN_DIFF: Combine the parts of hSrcRgn1 that are not in hSrcRgn2.
- RGN_COPY: Copy hSrcRgn1 to the destination region.

Using Regions for Drawing

Regions can be used for drawing operations using the following functions:

- FillRgn: Fills the specified region with the current brush.
- FrameRgn: Draws a frame around the specified region using the current brush.
- InvertRgn: Inverts the pixels within the specified region.
- PaintRgn: Fills the specified region with the current brush.

These functions assume the region is defined in logical coordinates.

Deleting Regions

When you're finished with a region, you should delete it using the DeleteObject function to release the associated resources.

💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥

Example Usage

Here's an example of how to create and use regions:

```
HRGN hRgn1 = CreateRectRgn(10, 10, 50, 50);
HRGN hRgn2 = CreateEllipticRgn(20, 20, 60, 60);

HRGN hDestRgn = CombineRgn(NULL, hRgn1, hRgn2, RGN_OR);

HDC hdc = GetDC(hwnd);
FillRgn(hdc, hDestRgn, GetStockObject(BLACK_BRUSH));
DeleteObject(hRgn1);
DeleteObject(hRgn2);
DeleteObject(hDestRgn);
ReleaseDC(hwnd, hdc);
```

This code creates a rectangular region (hRgn1), an elliptical region (hRgn2), and then combines them using RGN_OR to create a destination region (hDestRgn).

It then fills the destination region with a black brush. Finally, it deletes all the regions and releases the device context. Regions provide a powerful tool for clipping and drawing operations in Windows graphics programming.

CLIPPING WITH RECTANGLES AND REGIONS

Clipping is a fundamental concept in graphics programming, restricting drawing operations to a specific area of the display. Regions, along with rectangles, play a crucial role in clipping operations.

InvalidateRect and ValidateRect

The InvalidateRect function invalidates a rectangular portion of the display, triggering a WM_PAINT message. This function can be used to erase the client area and generate a WM_PAINT message:

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

The GetUpdateRect function retrieves the coordinates of the invalid rectangle. Alternatively, the ValidateRect function validates a specific rectangle within the client area.

Upon receiving a WM_PAINT message, the invalid rectangle's coordinates are accessible through the PAINTSTRUCT structure populated by the BeginPaint function.

This invalid rectangle defines the "clipping region," restricting drawing operations to its boundaries.

InvalidateRgn and ValidateRgn

Windows provides InvalidateRgn and ValidateRgn functions, similar to their rectangular counterparts, but operate on regions instead of rectangles:

```
InvalidateRgn(hwnd, hRgn, bErase);
```

```
ValidateRgn(hwnd, hRgn);
```

When a WM_PAINT message is triggered due to an invalid region, the clipping region may not necessarily be rectangular.

Selecting Clipping Regions

You can create a custom clipping region by selecting a region into the device context using either of the following functions:

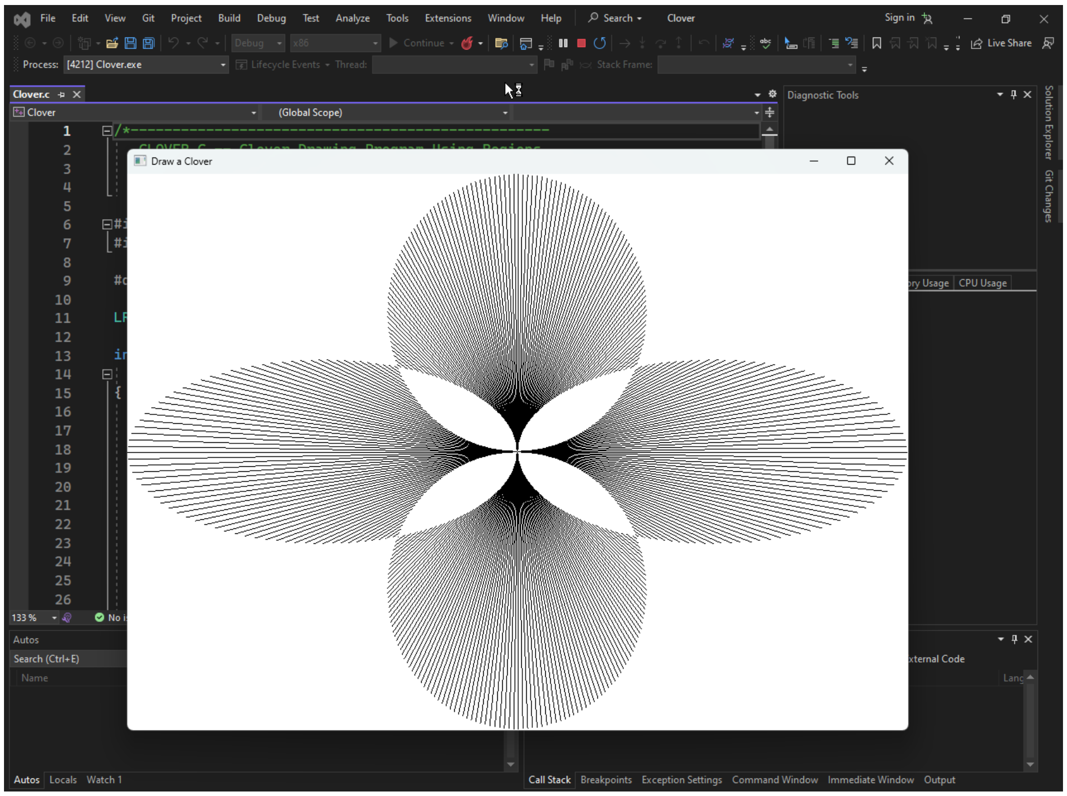
```
SelectObject(hdc, hRgn);
```

```
SelectClipRgn(hdc, hRgn);
```

The clipping region is assumed to be measured in device coordinates.

- GDI creates a copy of the selected clipping region, allowing you to delete the original region object. Windows provides several functions to manipulate this clipping region, including:
- ExcludeClipRect: Excludes a rectangle from the clipping region.
- IntersectClipRect: Creates a new clipping region as the intersection of the existing clipping region and a specified rectangle.
- OffsetClipRgn: Moves the clipping region to a different location within the client area.

Clover program Chapter 5, clover folder: That's freaking amazing!



Clover Drawing Program Using Regions

The code defines a Windows application that draws a clover using regions. It creates three elliptical regions, one for each oval of the clover, and then combines them using the XOR (exclusive OR) operation to create the final clipping region. This clipping region is then used to restrict the drawing of a series of lines that form the clover's outline.

Regions: The program utilizes regions to define the clipping area for drawing the clover. Regions provide a powerful mechanism for clipping operations, allowing for non-rectangular clipping boundaries.

Clipping: Clipping restricts drawing operations to a specific area of the display. In this case, the clipping region ensures that the clover is drawn within the window bounds.

Combining Regions: The XOR (exclusive OR) operation is used to combine the three elliptical regions into a single clipping region. This operation ensures that the overlapping portions of the ellipses are not drawn, resulting in the distinct clover shape.

Line Drawing: The clover's outline is drawn using a series of lines. The MoveToEx and LineTo functions are used to position the starting point and endpoint of each line segment.

Code Breakdown

Define Window Class: The WNDCLASS structure is initialized with the necessary attributes for the window class, including the window procedure, background brush, and class name.

Register Window Class: The RegisterClass function registers the window class with Windows, making it available for window creation.

Create Window: The CreateWindow function creates the main window for the application, specifying its parent window, title, style, position, and initial size.

Message Loop: The main message loop retrieves messages from the system's message queue and dispatches them to the window procedure for processing.

Window Procedure: The WndProc function handles messages sent to the window, including WM_SIZE, WM_PAINT, and WM_DESTROY.

- WM_SIZE: Upon receiving a WM_SIZE message, the window's client area dimensions are retrieved and used to calculate the radius of the clover and create the temporary regions.
- WM_PAINT: In response to a WM_PAINT message, a device context is obtained, and the viewport origin is set to the center of the client area. The clipping region is selected into the device context, and the clover's outline is drawn using a series of lines.
- WM_DESTROY: When the window is destroyed, the clipping region is deleted, and a WM_QUIT message is posted to terminate the application.

The clover drawing program demonstrates the use of regions for clipping operations and line drawing to create a visually appealing graphic. It highlights the concept of combining regions to achieve complex clipping shapes and the efficiency of using regions for non-rectangular clipping scenarios.

End of Chapter 5….