Graphics in Windows

1. **Coordinate Systems in Windows**
2. **z-Order**
3. **Device Contexts**

Coordinate Systems in Windows

All coordinates on the screen that you need to deal with grow down the y-axis and from left to right along the x-axis.

Most of the graphics manipulation done on standard system windows involves the use of a rectangle structure:

Rectangle Coordinates

An application must use a **RECT** structure to define a rectangle. The structure specifies the coordinates of two points: the upper left and lower right corners of the rectangle. The sides of the rectangle extend from these two points and are parallel to the x- and y-axes.

The coordinate values for a rectangle are expressed as signed integers. The coordinate value of a rectangle's right side must be greater than that of its left side. Likewise, the coordinate value of the bottom must be greater than that of the top.

Because applications can use rectangles for many different purposes, the rectangle functions do not use an explicit unit of measure. Instead, all rectangle coordinates and dimensions are given in signed, logical values.

RECT

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

Members

left
Specifies the x-coordinate of the upper-left corner of the rectangle.

top
Specifies the y-coordinate of the upper-left corner of the rectangle.

right
Specifies the x-coordinate of the lower-right corner of the rectangle.

bottom
Specifies the y-coordinate of the lower-right corner of the rectangle.

Rectangle Operations

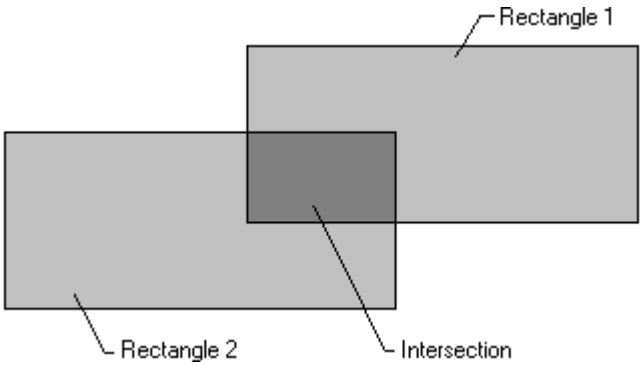
The **SetRect** function creates a rectangle, the **CopyRect** function makes a copy of a given rectangle, and the **SetRectEmpty** function creates an empty rectangle. An empty rectangle is any rectangle that has zero width, zero height, or both. The **IsRectEmpty** function determines whether a given rectangle is empty. The **EqualRect** function determines whether two rectangles are identical — that is, whether they have the same coordinates.

The **InflateRect** function increases or decreases the width or height of a rectangle, or both. It can add or remove width from both ends of the rectangle; it can add or remove height from both the top and bottom of the rectangle.

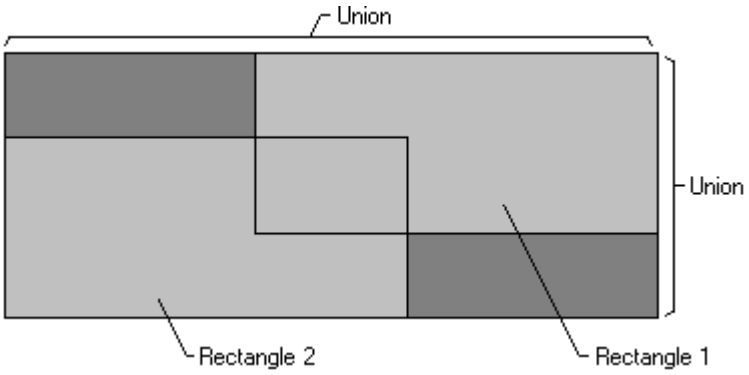
The **OffsetRect** function moves a rectangle by a given amount. It moves the rectangle by adding the given x-amount, y-amount, or x- and y-amounts to the corner coordinates.

The **PtInRect** function determines whether a given point lies within a given rectangle. The point is in the rectangle if it lies on the left or top side or is completely within the rectangle. The point is not in the rectangle if it lies on the right or bottom side.

The **IntersectRect** function creates a new rectangle that is the intersection of two existing rectangles, as shown in the following figure.



The **UnionRect** function creates a new rectangle that is the union of two existing rectangles, as shown in the following figure.


Union
Union
Rectangle 2
Rectangle 1

z-Order

Windows are layered, as if they were stacked on top of the plane defined by the screen, in what is known as a z-order. Typically, the topmost window is the one that is visible, as when we open multiple applications and layer one on top of the other on our desktop.

Windows exist in a hierarchy of *top level windows, owned windows, and child windows.* The windows API provides a number of functions that make it possible to search for a particular window and determine the hierarchy of every window in the system.

Device Contexts

A device context (DC) is a Windows object that allows drawing to a window or device. All graphic and text output to a device takes place through a device context. Some texts refer to device contexts for windows as display contexts.

We most often write output to either the printer or the screen in Windows. Device contexts provide a bridge between the graphic information (text/lines/fills) and the physical device and its associated software drivers.

The classic example of the need for device contexts is word processing. We need to define a region on the screen to which we wish to draw our text as we edit it. We might want to prevent certain areas of the screen from being written to. Maybe we want to offset the text region of our user interface from the absolute top left corner of the screen. As if the calculations to manage such functionality weren't a headache in themselves, consider now the problem of displaying the text on the screen and sending the graphics to the printer so that they look the same...

The printer probably has a different resolution, different color palette, and probably has a different aspect ratio. And all of these conversions would be necessary to support every unique printer and screen setup that you wanted your program to support.

Thankfully, Windows abstracts most of the low level work for us using *clipping* and *coordinate transformation*.

Clipping

A *clipping region* is an area in a window over which your program will allow drawing. Every time a program calls a graphics or text output function, Windows checks to make sure that the area to which the call was made to write falls in a clipping area. Consider again the word processing example: Suppose we want to define an area in our document to contain a graphic. We might want to prevent text from being written to that area of the screen. Clipping regions provide a way to prevent drawing to areas of windows that you (or Windows) wants to keep off-limits.

The clipping region of a form is called the *client area*. This area consists of everything internal or the border of the form and below the title bar. Child windows usually define a clipping region in their parent window's context so

that subsequent drawing to the window's client are will not affect the child window.

Coordinate Transformation

Windows provides the ability to map, or transform, one coordinate system to another. You can specify a logical coordinate system and allow Windows to do the work of mapping that logical coordinate system to the device's coordinate system on which it will be drawn.

Where Device Contexts Come from...

Windows provides device contexts for you. A DC may be associated with a window, a device, or a block of memory that simulates a device. Clipping regions and coordinate systems are associated with each device context.

Think of DCs as a black box: graphics and text instructions go in one end and out the other end come instructions for drawing to the physical device for which the DC exists, with all coordinate systems and clipping regions transformed.