

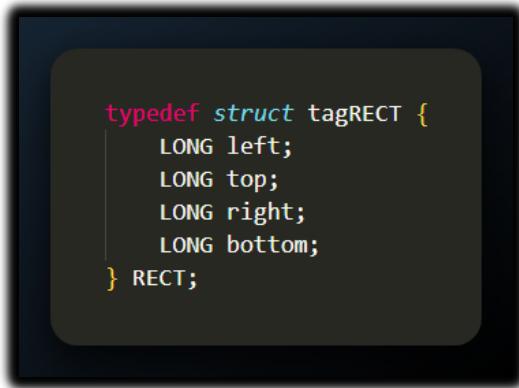
RECTANGLES: THE FOUNDATION

The Concept: The basic building block.

In Windows graphics, almost everything starts with a rectangle.

- **The Structure:** Windows defines the RECT structure to hold four coordinates: left, top, right, and bottom.
- **Usage:** They define window boundaries, control sizes, and areas to be repainted.

The RECT Structure:



Key Rule: A rectangle in Windows typically does **not** include the right and bottom edges. It covers pixels from left up to (but not including) right, and from top up to (but not including) bottom.

1. Drawing Functions for Rectangles

Windows provides specialized functions designed specifically to work with RECT pointers directly, rather than passing four separate coordinates (like the standard Rectangle function does).

FillRect(hdc, &rect, hBrush):

- Fills the rectangular area with a specific brush.
- *Difference from Rectangle:* FillRect does **not** draw a border. It only paints the interior.

FrameRect(hdc, &rect, hBrush):

- Draws a rectangular frame (border) around the rectangle using a **Brush** (not a Pen!).
- *Usage:* Often used for focus rectangles on buttons.

InvertRect(hdc, &rect):

- Inverts every pixel inside the rectangle (White becomes Black, Blue becomes Yellow).
- *Usage:* Used for highlighting text or selections without needing to know the background color.

2. Manipulating Rectangles

You rarely manipulate the left/top/right/bottom members manually. Windows provides a toolkit of math functions to handle the geometry for you.

Function	Description
<code>SetRect</code>	Fills a <code>RECT</code> structure with specific coordinates (Left, Top, Right, Bottom).
<code>CopyRect</code>	Duplicates one <code>RECT</code> structure into another.
<code>SetRectEmpty</code>	Resets all coordinates in the rectangle to zero.
<code>OffsetRect</code>	Moves the rectangle. Adds x to left/right and y to top/bottom without changing its size.
<code>InflateRect</code>	Resizes the rectangle. Positive values expand it; negative values shrink it.
<code>IntersectRect</code>	Calculates the overlapping area between two rectangles. Primary tool for collision detection .
<code>UnionRect</code>	Creates the smallest possible rectangle that contains both source rectangles (a "Bounding Box").
<code>IsRectEmpty</code>	Returns TRUE if the rectangle has no area (width or height is 0).
<code>PtInRect</code>	Returns TRUE if a specific point (e.g., mouse cursor) is inside the rectangle. Primary tool for Hit Testing .

3. Regions and Clipping

The Concept: Beyond the Box.

While a Rectangle is simple, a **Region** is complex. It is a GDI object that describes an area of the screen that can be any shape—an ellipse, a polygon, or a combination of multiple disjoint rectangles.

Why use Regions? Clipping. Clipping is the primary use case for regions. It is a performance optimization technique.

- **The Problem:** If you only need to update the bottom-right corner of a window, redrawing the whole window is wasteful.
- **The Solution:** You define a "Clipping Region" (the dirty area). Windows will then **restrict** all drawing to that area. If you try to draw outside the clipping region, Windows ignores those pixels, saving processing power.

4. Drawing with Rectangles

The Concept: Fast, specialized drawing.

While Rectangle() draws a border *and* fills it using the current Pen and Brush, these functions allow you to do just one or the other, or perform special pixel math.

- **FillRect:** Only fills the interior. It ignores the current Pen (no border). It requires a brush handle as an argument.
- **FrameRect:** Only draws the border. Uniquely, it uses a **Brush** to draw the border, not a Pen. This allows for thick, textured borders.
- **InvertRect:** Flips the bits of the pixels inside. This is computationally fast and is often used for "highlighting" text or menu items because calling it a second time restores the original colors perfectly.

5. Manipulating Rectangles

The Concept: Geometry without drawing.

These functions don't draw anything. They just change the numbers inside the RECT structure variables.

- **OffsetRect:** Moves the box. (e.g., Slide the button 10 pixels right).
- **InflateRect:** Changes the size. (e.g., Add 5 pixels padding around the text).
- **IntersectRect:** Calculates the "overlap" area. Useful for detecting if two objects collided.
- **UnionRect:** Calculates the "bounding box" that covers both shapes.

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

6. Technical Warning: The "Bottom-Right" Trap

In WinAPI, the RECT structure is **exclusive** of the bottom and right edges.

- **Top/Left:** Included inside the rectangle.
- **Bottom/Right:** *Excluded* from the rectangle.

If you have a rectangle {0, 0, 10, 10}, the pixel at 10,10 is **NOT** part of the rectangle. PtInRect will return FALSE for 10,10. This is a classic "off-by-one" bug source in GDI collision logic.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
| | | | | PSTR szCmdLine, int iCmdShow)
{
    // 1. Setup Graphics Context
    HWND hwnd = GetDesktopWindow();
    HDC hdc = GetDC(hwnd);

    // 2. Create Tools
    // Create a Solid Red Brush
    HBRUSH hRedBrush = CreateSolidBrush(RGB(255, 0, 0));
    RECT rect;

    // --- DRAWING EXAMPLES ---
    // A. FillRect: Create a solid red block
    SetRect(&rect, 10, 10, 100, 100);           // Define coords
    FillRect(hdc, &rect, hRedBrush);           // Paint it red
    // B. FrameRect: Draw a red border (hollow inside)
    SetRect(&rect, 20, 20, 120, 120);          // Define coords
    FrameRect(hdc, &rect, hRedBrush);          // Draw frame
    // C. InvertRect: Invert screen colors
    SetRect(&rect, 30, 30, 130, 130);
    InvertRect(hdc, &rect);                   // Invert pixels

    // --- MANIPULATION EXAMPLES (Math only) ---
    // D. Offset: Move the box
    SetRect(&rect, 40, 40, 140, 140);
    OffsetRect(&rect, 20, 20);                // Move right 20, down 20
    // (rect is now left=60, top=60...)
    // E. Inflate: Grow the box
    InflateRect(&rect, 10, 10);              // Add 10px to all sides

    // F. Reset
    SetRectEmpty(&rect);                     // All zeros
    // --- INTERSECTION EXAMPLE ---
    RECT rect1, rect2, destRect;
    | SetRect(&rect1, 10, 10, 50, 50);
    SetRect(&rect2, 20, 20, 60, 60);
    | // Calculate Overlap: destRect will be (20, 20, 50, 50)
    IntersectRect(&destRect, &rect1, &rect2);

    // Calculate Union: destRect will be (10, 10, 60, 60)
    UnionRect(&destRect, &rect1, &rect2);

    // --- CHECKS ---
    BOOL bEmpty = IsRectEmpty(&destRect);    // Check size

    POINT pt = {30, 30};
    BOOL bInRect = PtInRect(&destRect, pt); // Hit testing

    // 3. Cleanup
    ReleaseDC(hwnd, hdc);
    DeleteObject(hRedBrush);

    return 0;
}
```

RANDOM RECTANGLES AND THE ALTERNATIVE MESSAGE LOOP

1. The Problem with the Standard Loop

The Concept: Sleeping vs. Working.

In previous examples, we used the standard Windows message loop:

```
while (GetMessage(&msg, NULL, 0, 0)) { ... }
```

- **How it works:** GetMessage is a blocking function. If there are no messages (no mouse clicks, no key presses), Windows puts your program to **sleep**.
- **The Issue:** If you want to draw thousands of random rectangles rapidly to create an animation, you can't use GetMessage. Your program would stop drawing the moment the user stops moving the mouse.

2. The Solution: PeekMessage

The Concept: checking without waiting.

To utilize the "Dead Time" (the time when the user is doing nothing), we use PeekMessage.

- **Function:** It looks into the message queue to see if anything is there.
- **Result:** It returns immediately, regardless of whether a message exists or not.
 - **If Message:** You process it.
 - **If No Message:** You use this free time to draw your rectangles.

3. The PeekMessage Parameters

The function signature is similar to GetMessage, with one crucial addition:

Parameter	Description
<code>lpMsg</code>	Pointer to the <code>MSG</code> structure that receives message information.
<code>hWnd</code>	Window Handle. Use <code>NULL</code> to check all windows belonging to the current thread.
<code>wMsgFilterMin</code>	The value of the first message in the range to be examined. Use <code>0</code> to check all messages.
<code>wMsgFilterMax</code>	The value of the last message in the range to be examined. Use <code>0</code> to check all messages.
<code>wRemoveMsg</code>	Critical Flag: Determines if the message is removed from the queue. <code>PM_REMOVE</code> : Removes the message after processing. <code>PM_NOREMOVE</code> : Leaves the message in the queue (useful for "peeking" without consuming).

- **PM_REMOVE**: "Read the message and take it off the queue." (Used in the main loop).
- **PM_NOREMOVE**: "Just look at the message but leave it there."

4. The Alternative "Animation" Loop

The Concept: The while(TRUE) Loop.

Instead of looping *while* GetMessage returns true, we loop forever (while(TRUE)) and break manually only when we see a Quit message.

The Structure:

```
while (TRUE)
{
    // 1. Check if a message is waiting
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // 2. Critical: Check for quit manually!
        if (msg.message == WM_QUIT)
            break; // Exit the loop

        // 3. Process normal messages (clicks, keys, etc.)
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        // 4. "Dead Time": No messages waiting? Draw graphics!
        // This runs thousands of times per second.
        DrawRandomRectangle(hwnd);
    }
}
```

5. Drawing the Random Rectangles

Although the text focuses on the loop, the logic for the "Dead Time" drawing usually involves the `rand()` function and the RECT manipulation functions we learned earlier.

```
void DrawRandomRectangle(HWND hwnd) {
    HDC hdc = GetDC(hwnd);
    RECT rect;
    HBRUSH hBrush;

    // Get window size so we don't draw off-screen
    GetClientRect(hwnd, &rect);

    // Create random coordinates
    int x1 = rand() % rect.right;
    int y1 = rand() % rect.bottom;
    int x2 = rand() % rect.right;
    int y2 = rand() % rect.bottom;

    // Create random color
    hBrush = CreateSolidBrush(RGB(rand()%256, rand()%256, rand()%256));

    // Draw
    SetRect(&rect, x1, y1, x2, y2);
    FillRect(hdc, &rect, hBrush);

    // cleanup (Very important in a fast loop!)
    ReleaseDC(hwnd, hdc);
    DeleteObject(hBrush);
}
```

PEEKMESSAGE LIMITATIONS AND REGIONS

1. The Trap of PeekMessage & WM_PAINT

The Concept: The un-killable message.

You might think PeekMessage(..., PM_REMOVE) removes any message it finds. However, WM_PAINT is special.

- **The Issue:** WM_PAINT is not really "in" the queue like a mouse click. It is a flag that Windows raises whenever a part of your window is "invalid" (needs repainting).
- **The Loop:** If you use PeekMessage to remove WM_PAINT but **do not** repaint (or validate) the window, Windows sees the dirty region, realizes it's still dirty, and immediately generates a *new* WM_PAINT message.
- **The Fix:** You cannot just "remove" it. You **must** resolve the invalid region using one of these pairs:
 - BeginPaint() / EndPaint() (Standard method)
 - ValidateRect()
 - ValidateRgn()

Historical Note: In the Windows 98 era, PeekMessage was vital for "cooperative multitasking." Today, with "preemptive multitasking," we typically use **multithreading** (creating a separate thread for background work) rather than modifying the message loop. However, the PeekMessage loop is still the standard for high-performance games (DirectX/OpenGL).

2. What is a Region?

The Concept: The Complex Shape.

So far, we have drawn simple shapes (Rectangles, Ellipses). A **Region** is a GDI Object that describes a **complex area** of the screen.

- It can be a combination of two rectangles, a circle with a hole in it, or a star shape.
- **Primary Use: Clipping.** You can tell Windows, "Only allow drawing inside *this* specific region."

3. Creating Regions

Since Regions are GDI objects, you must create them, use them, and delete them.

WinAPI Function	Description
<code>CreateRectRgn</code>	Creates a simple rectangular region .
<code>CreateEllipticRgn</code>	Creates an elliptical region bounded by specific coordinates.
<code>CreateRoundRectRgn</code>	Creates a rectangle with rounded corners (useful for modern UI window shapes).
<code>CreatePolygonRgn</code>	Creates a region from an array of points, forming a filled polygon .
<code>CreatePolyPolygonRgn</code>	Creates a single region handle consisting of multiple disjoint polygons .

4. Combining Regions (CombineRgn)

The Concept: Boolean Math for Shapes.

The real power of regions comes from combining them. You can take two simple shapes and merge them to create a complex one.

Syntax:

```
// Note: hDest must already exist (usually created as an empty rect region)
CombineRgn(hDest, hSrc1, hSrc2, iMode);
```

The Combination Modes (iMode):

Mode (iMode)	Logic	Visual Result
RGN_AND	Intersection	The area where the two regions overlap. Only the "shared" space remains.
RGN_OR	Union	The total area of both regions combined into one large shape.
RGN_XOR	Exclusive OR	Combines both regions but removes the overlapping part (creates a hole where they met).
RGN_DIFF	Difference	Takes the first region (<code>hsrc1</code>) and subtracts the part covered by the second (<code>hsrc2</code>). Acts like a Cookie Cutter .
RGN_COPY	Copy	Ignores the second region and simply copies the first to the destination.

5. Drawing with Regions

Once you have defined your complex region, you can draw it onto the screen.

- **FillRgn(hdc, hRgn, hBrush)**: Fills the region using a specific brush you provide as an argument.
- **PaintRgn(hdc, hRgn)**: Fills the region using the brush **currently selected** in the Device Context.
- **FrameRgn(hdc, hRgn, hBrush, w, h)**: Draws a border around the region. Unlike FrameRect, you can specify the width and height of the border frame.
- **InvertRgn(hdc, hRgn)**: Inverts the colors of the pixels inside the region.

6. Cleanup

The Concept: Avoid Memory Leaks.

Regions are GDI objects. Like Pens and Brushes, they occupy memory in the GDI Heap. When you are done with a region handle (HRGN), you must delete it:

```
DeleteObject(hRgn);
```

The logic below creates a square, creates a circle, and then merges them into a single shape (a Union) before painting it black.

```

// 1. Create the two source regions
HRGN hRgn1 = CreateRectRgn(10, 10, 50, 50);           // A Square
HRGN hRgn2 = CreateEllipticRgn(20, 20, 60, 60);       // A Circle

// 2. Create an empty region to hold the result
// (CombineRgn does not create a handle; it requires one to write into)
HRGN hDestRgn = CreateRectRgn(0, 0, 0, 0);

// 3. Combine them using OR (Union)
// hDestRgn now contains the shape of both combined
CombineRgn(hDestRgn, hRgn1, hRgn2, RGN_OR);

// 4. Get DC and Paint
HDC hdc = GetDC(hwnd);
HBRUSH hBrush = (HBRUSH)GetStockObject(BLACK_BRUSH);

// Fill the complex shape with black
FillRgn(hdc, hDestRgn, hBrush);

// 5. Cleanup (Crucial!)
DeleteObject(hRgn1);
DeleteObject(hRgn2);
DeleteObject(hDestRgn);
ReleaseDC(hwnd, hdc);

```

- **Creation:** We created two simple geometric regions (hRgn1 and hRgn2).
- **The Container:** CombineRgn works differently than most math functions. It doesn't return a new object. Instead, you must provide it with an existing region handle (hDestRgn) which it will overwrite with the result. We typically create a dummy empty region for this purpose.
- **The Operation (RGN_OR):** This tells Windows to take every pixel that is in the Square **OR** in the Circle. The result is a single "blob" shape.
- **Drawing:** FillRgn paints the resulting complex blob.
- **Cleanup:** We must delete all three region handles to avoid memory leaks.

This wraps up the section on Creating and Combining Regions.

WHAT IS CLIPPING?

The Concept: The Stencil.

Imagine you are spray-painting a wall, but you hold a piece of cardboard with a star cutout against the wall. Even if you spray paint everywhere, the paint only appears inside the star.

- In GDI, this "cardboard cutout" is called the **Clipping Region**.
 - Any drawing you attempt outside this region is mathematically ignored and does not appear on the screen.
-

1. Automatic Clipping (WM_PAINT)

You have been using clipping all along without knowing it.

When you call BeginPaint(), Windows automatically sets up a default clipping region.

- **Scenario:** You have a calculator app open, and you drag a Notepad window over the bottom half of it.
- **The Invalidation:** When you move Notepad away, Windows marks only that bottom half of the calculator as "Invalid" (Dirty).
- **The Clip:** When the calculator repaints, BeginPaint restricts drawing to **only** that bottom half. Even if your code tries to redraw the whole calculator interface, GDI "clips" the top half to save processor time (since it's already correct).

Functions for this:

- **InvalidateRect(hwnd, &rect, TRUE):** Tells Windows "This rectangle is dirty, please trigger a paint message."
- **InvalidateRgn(hwnd, hRgn, TRUE):** Tells Windows "This complex shape (not just a box) is dirty."

2. Manual Clipping

The Concept: Creating your own Stencil.

You can manually force Windows to restrict drawing to a shape of your choice. This is how you create effects like text appearing only inside a circle, or the "Clover" effect.

The Function:



- **Crucial Note:** When you call this, GDI makes a **copy** of your region. You can (and should) delete your region handle (hRgn) immediately after selecting it to save memory. The copy stays active in the DC until you change it.
-

3. Modifying the Stencil

You don't always have to create a new region. You can modify the current clipping region on the fly using these math functions:

Function	Description	Visual Metaphor
<code>ExcludeClipRect</code>	Removes a specific rectangle from the current drawable area.	<i>Cutting a hole</i> in your canvas so you can't paint there.
<code>IntersectClipRect</code>	Sets the new clipping area to the intersection of the current area and a new rectangle.	"Draw only where the current stencil AND this new box overlap."
<code>OffsetClipRgn</code>	Moves the clipping region by specified x and y offsets.	<i>Sliding the stencil</i> across the canvas.

4. The "Clover" Program (The Example)

The Concept: The Magic of Mathematics + Clipping.

The "Clover" program mentioned in your text is a classic Windows programming example (often found in Charles Petzold's *Programming Windows*). It demonstrates the power of the RGN_XOR and RGN_AND combination.

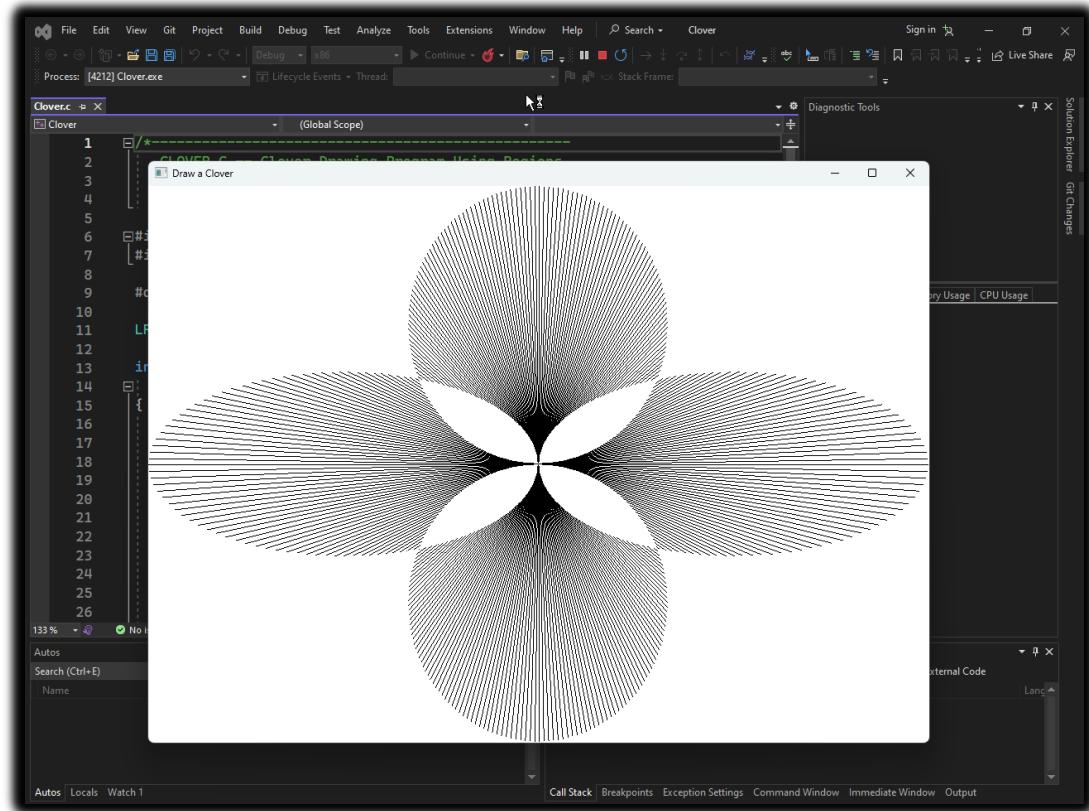
How it works:

1. **Create 4 Circular Regions:** It creates four ellipses that overlap to form a 4-leaf clover shape.
2. **Combine Them:** It uses CombineRgn to merge them into a single complex "Clover" region.
3. **Select Clip:** It calls SelectClipRgn to force all drawing to stay inside this clover.
4. **Draw Lines:** It draws straight lines radiating from the center of the screen to the edges (360 degrees).

The Result: Because the Clipping Region is active, the straight lines are essentially "cut" into the shape of the clover. You see a beautiful, complex geometric pattern that looks like a fabric texture, perfectly bounded by the clover curves.

Why is it "Freaking Amazing"? It shows that you don't need to do complex trigonometry to calculate where the lines should stop at the curved edge of a leaf. You just tell Windows "Clip here," and you draw simple long lines. Windows handles the hard intersection math for you.

Clover program Chapter 5, clover folder: This is just freaking amazing!



CLOVER DRAWING PROGRAM

This example brings together everything you have learned in this chapter: Device Contexts, Drawing Tools, Region Mathematics (RGN_XOR), and Clipping. It produces a stunning visual result with very few lines of code by letting the GDI math do the heavy lifting.

The Logic: "String Art" via Mathematics

The visual effect relies on two distinct steps:

The Invisible Stencil (Clipping Region): We create separate elliptical regions (the "leaves") and combine them using **XOR** (Exclusive OR). *Why XOR?* XOR removes the pixels where the regions overlap. The center of the clover, where the leaves overlap, becomes hollow (or re-inverted depending on the overlap count). This creates the complex internal geometry.

The Drawing (The Strings): We don't draw the clover outline directly. Instead, we draw hundreds of straight lines radiating from the center of the screen to the edges. Because the **Clipping Region** is active, these lines are visible *only* when they pass inside the "Clover" shape. This creates a "string art" texture automatically.

The Complete Code (Clover.c)

Here is the implementation. Note how the complexity is hidden inside the WM_SIZE (where the math happens) and WM_PAINT (where the drawing happens).

Tool	Purpose	Key Functions
Device Context (DC)	The Canvas	<code>GetDC</code> , <code>BeginPaint</code> , <code>ReleaseDC</code>
Pen	Line Drawing	<code>CreatePen</code> , <code>SelectObject</code>
Brush	Area Filling	<code>CreateSolidBrush</code> , <code>CreateHatchBrush</code>
Shape Functions	The Geometry	<code>Rectangle</code> , <code>Ellipse</code> , <code>Polygon</code>
Mapping Modes	Coordinate Conversion	<code>SetMapMode</code> (Pixels vs Inches vs MM)
Regions	Complex Shapes & Clipping	<code>CreateEllipticRgn</code> , <code>CombineRgn</code> , <code>SelectClipRgn</code>