

Die WM_PAINT Nachricht

Ziel, Inhalt

- Bis jetzt können wir nur auf die Fensternachricht WM_DESTROY reagieren, die wir verwenden um die Nachrichtenschleife und damit das Programm zu verlassen.

Die WM_PAINT Nachricht	1
Ziel, Inhalt	1
Die WM_PAINT Nachricht	2
Ein einfaches Windows-Programm	2
Spy++	2
Neuzeichnen des Fensters	3
Gründe	3
Mechanik	3
Funktion zum Neuzeichnen des Fensters	5

Die WM_PAINT Nachricht

Ein einfaches Windows-Programm

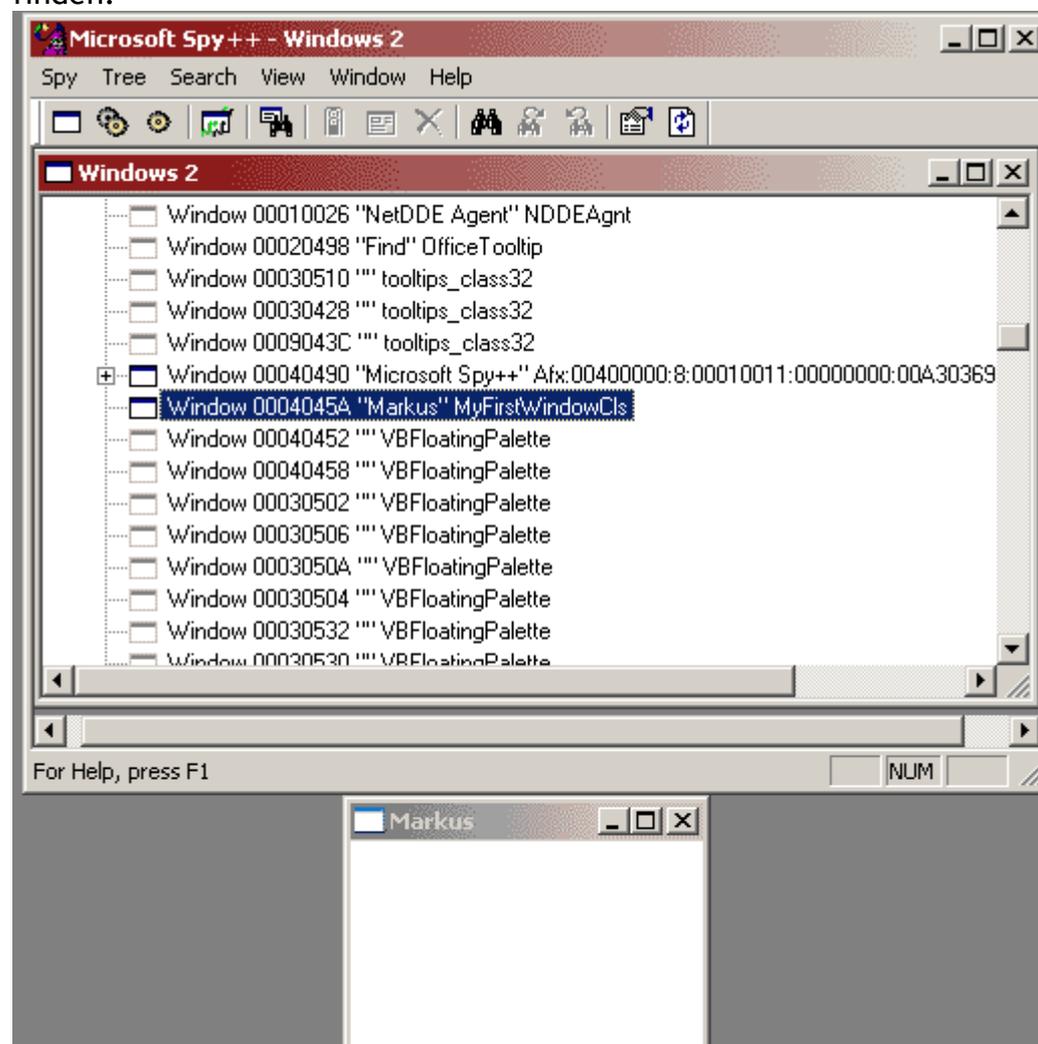
Wir starten mit einem ganz einfachen Windows-Programm das ein Fenster erzeugt und eine Nachrichtenschleife unterhält. Am besten du gehst von der cpp-Datei aus der letzten Lektion aus. Du findest diese hier:

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend12/Code.cpp>

Achte darauf beim Erstellen des Projektes eine WIN32-Anwendung zu erstellen. Danach kopierst du die erwähnte Datei in das Verzeichnis des Projektes und fügst die Datei in das Projekt ein.

Spy++

Starte nun das Programm Spy++ mit dem du Fensternachrichten von beliebigen Fenstern betrachten kannst. Starte auch deine kleine Applikation. Mit dem Spy++ solltest du das Fenster unseres Programms finden.



Durch Rechtsklick auf den Eintrag kannst du dir alle Nachrichten ansehen, die unser Programm für unser Fenster erhält. Du kannst jetzt verschiedene Aktionen durchführen wie mit der Maus über das Fenster fahren, das Fenster in der Grösse ändern oder das Fenster mit einem anderen Fenster teilweise bedecken.

Neuzeichnen des Fensters

Gründe

Für das Neuzeichnen des Fensters gibt es verschiedene Gründe. Die Frage, die wir dabei beantworten müssen ist, wodurch wird der Inhalt des Fensters ungültig und muss neu gezeichnet werden.

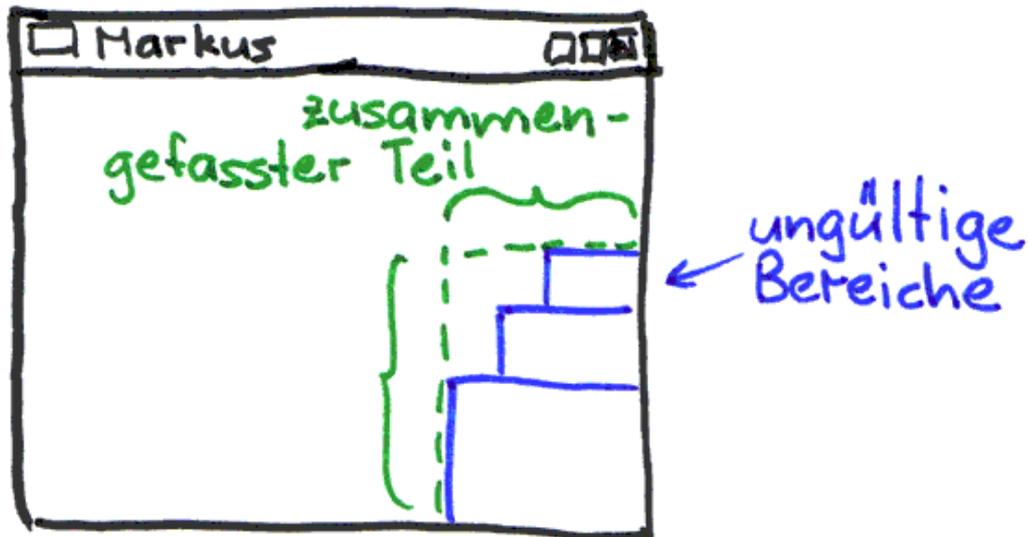
1. Das Fenster wird sichtbar
Das Fenster wird erstellt und sichtbar gemacht. Bei meinem Beispiel dadurch, dass das `WS_VISIBLE` flag bei `CreateWindow` gesetzt war. Das Fenster wird auch sichtbar, wenn es ganz oder teilweise durch ein anderes Fenster verdeckt war und wieder sichtbar wird.
2. Die Fenstergrösse ändert sich
Wenn sich die Fenstergrösse ändern werden je nach Darstellung neue Bereiche sichtbar, die neu gezeichnet werden müssen. Einige Fenster passen auch die Grössenverhältnisse an, nimm als Beispiel ein Uhrenprogramm, das eine runde Uhr darstellt, die immer die ganze Fenstergrösse einnimmt.
3. Die Daten, die dargestellt werden ändern sich
Ein weiterer wichtiger Grund dafür, dass ein Fenster neu gezeichnet werden muss ist, dass sich Daten ändern. Nehmen wir wieder die Uhr als Beispiel. Je nach Darstellung kann sich der Fensterinhalt häufig ändern (z.B. jede Sekunde), was zu einem Neuzeichnen des Fensters führt.

Mechanik

Wenn das Betriebssystem feststellt, dass unser Fenster neu gezeichnet werden muss, erhalten wir die `WM_PAINT` Nachricht. Du solltest diese Nachricht auch mit dem Spy++ bereits gesehen haben. Je nach Installation, solltest du in der Hilfe zum Visual Studio einfach `WM_PAINT` im Index finden. Falls mehrere Hilfethemen gefunden werden, wählst du den Eintrag, der zum Thema „Windows GDI: Platform SDK“ gehört. Diese Hilfe wird von Microsoft nur auf Englisch ausgeliefert!

Wenn ein Programm diese Nachricht bearbeitet gilt es einige API's aufzurufen, damit das Betriebssystem merkt, dass wir ungültige Fensterbereiche aufgefrischt haben.

Wenn Teilbereiche unseres Fensters ungültig werden, merkt sich das OS diese Teilbereiche und sendet `WM_PAINT` Nachrichten an unsere Nachrichtenschleife. Dabei kann das Betriebssystem zur Optimierung mehrere solche Teilbereiche zusammenfassen und nur eine `WM_PAINT` Nachricht erzeugen.



Wir erhalten diese Nachricht und können unsere *WindowProcedure* erweitern mit einem case für *WM_PAINT*:

```

LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT message,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
    switch(message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_PAINT:
            break;
        default:
            // Diese Funktion verarbeitet die Nachricht auf Standard-Art
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

```

Diesen Code aber besser nicht ausführen, denn das könnte zu ungewollten Reaktionen des Betriebssystems führen. Wir müssen nämlich die Funktion *BeginPaint* und *EndPaint* aufrufen. Die Funktion *BeginPaint* erzeugt einen so genannten Geräte-Kontext, der auch über ein Handle (HDC) identifiziert wird. Diesen können wir verwenden um Grafik auszugeben.

Nachdem wir gezeichnet haben was immer wir wollen, können wir das dem Betriebssystem mit *EndPaint* mitteilen. Die ungültigen Bereiche werden dadurch wieder als gültig markiert. Vergessen wir dies, glaubt das Betriebssystem die Bereiche seien noch ungültig und erzeugt endlos *WM_PAINT* Nachrichten. Bevor die *WM_PAINT* Nachricht erzeugt wird, erhalten wir auch die Nachricht *WM_ERASEBKGD*. Die *DefWindowProc* füllt dabei das gesamte Fenster mit dem BRUSH (Pinsel), den wir beim Registrieren der Window-Class definiert haben. Wir können auch selber auf

diese Nachricht reagieren. Beachte dabei, dass dabei unsere *WindowProcedure* nicht 0 zurückgeben darf:

```
LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT message,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
    long result = 0;
    switch(message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_ERASEBKGD:
            result = 1;
            break;
        default:
            result = DefWindowProc(hWnd,
                                   message,
                                   wParam,
                                   lParam);
    }

    return result;
}
```

Probier das einmal aus. Verstehst du was geschieht?

Funktion zum Neuzeichnen des Fensters

Definieren wir nun eine zusätzliche Funktion *MyPaint*, in der wir versuchsshalber etwas Text ausgeben:

```
// Funktionsprototyp
void MyPaint(HWND hwnd);

LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT message,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
    long result = 0;
    switch(message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_PAINT:
            MyPaint(hWnd);
            break;
        default:
            result = DefWindowProc(hWnd, message, wParam, lParam);
    }
    return result;
}
```

Hier ist sie nun die Funktion, die endlich etwas auf unser eigenes Fenster ausgibt:

```
const char* text = "Hallo Welt";

void MyPaint(HWND hWnd)
{
    // Die Struktur wird bei BeginPaint
    // mit einigen zusätzlichen Infos
    // abgefüllt
    PAINTSTRUCT ps = { 0 };
    HDC dc = BeginPaint(hWnd, &ps);
    // Das HDC ist der Gerätekontext,
    // es steht sizusagen für die
    // physikalische Bildschirmoberfläche

    // Alle Funktionen, die etwas ausgeben
    // brauchen einen Gerätekontext (HDC)
    TextOut(dc,
            30, // x
            30, // y
            text, // Zeiger auf const char
            strlen(text)); // Anzahl Zeichen

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

Nun kennst du die erste Ausgabefunktion *TextOut*. An dieser Stelle solltest du in der Hilfe nachsehen. Mit ein wenig Geschick findest du weitere Funktionen wie *Ellipse*, *FillRect* etc. Bei diesen Figuren wird die Linie mit dem aktuellen PEN gezeichnet. Die Flächen werden mit dem aktuellen BRUSH ausgefüllt. Das sind Objekte die zuerst erzeugt werden müssen und danach mit *SelectObject* in den Gerätekontext geladen werden müssen. Findest du mit der Dokumentation heraus wie das funktioniert? Vielleicht findest du sogar Beispiele. Versuche etwas anderes als Text auszugeben.