Ein Uhr Programm

Ziel, Inhalt

- Aufbauend auf dem Programm von letzter Woche, wollen wir ein Programm erstellen, das in einem Fenster eine Uhr mit Zeiger darstellt.

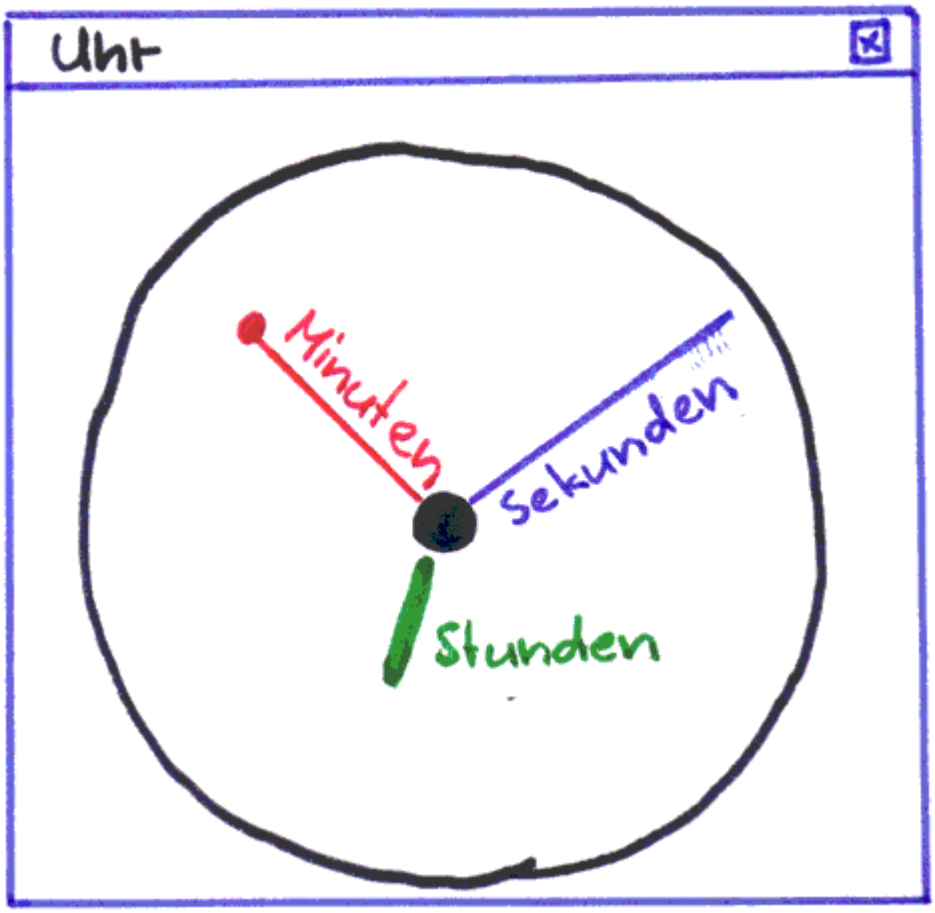
Ein Uhr Programm

Ziel

Wir wollen ein Programm erstellen, das in einem Fenster eine Uhr mit Zeigern zeichnet. Diese Zeiger sollen in regelmässigen Abständen weiterbewegt werden. Vermutlich werden wir ein wenig Trigonometrie brauchen.

Grobe Skizze

Zeichnen wir uns eine grobe Skizze, mit der uns das Design und die Berechnungen leichter fallen.



Bescheidener Anfang

Wir beginnen bescheiden und versuchen zuerst einen Kreis zu malen. Am besten wir gehen vom Programm der letzten Lektion aus, oder nehmen uns das Windows-Programmgerüst, das wir beim vorletzten mal erzeugt haben.

Die Klassenstile CS_HREDRAW und CS_VREDRAW

Der Kreis soll das Fenster immer ziemlich ausfüllen. Wenn das Fenster die Grösse ändert, soll also auch die Grösse des Kreises ändern. Am besten wäre es, dass jedes Mal wenn die Grösse des Fensters geändert wird, das Fenster neu gezeichnet wird. Durch Angabe zweier Flags beim registrieren der Fensterklasse (nicht C++ Klasse sondern WNDCLASS), wird automatisch beim Ändern der Fenstergrösse eine WM_PAINT Nachricht erzeugt.

```
WNDCLASS wc = {0};                          // Struktur für Window-Information

wc.hInstance = hInstance;                   // Handle der Applikation
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);// Hintergrundfarbe für das Fenster
wc.hCursor = LoadCursor(NULL, IDC_ARROW);   // Maus-Cursor gültig für das
Fenster
wc.lpszClassName = "UhrCls";                // Name für diese Window-Class
wc.lpfnWndProc = WindowProcedure;           // Window-Bearbeitungsfunktion
wc.style = CS_HREDRAW | CS_VREDRAW;         // Bei Aenderung der Fenstergrösse
                                            // wird ein WM_PAINT Nach. erzeugt

if(RegisterClass(&wc))                      // Die Window-Class registrieren
…
```

Fenstergrösse herausfinden

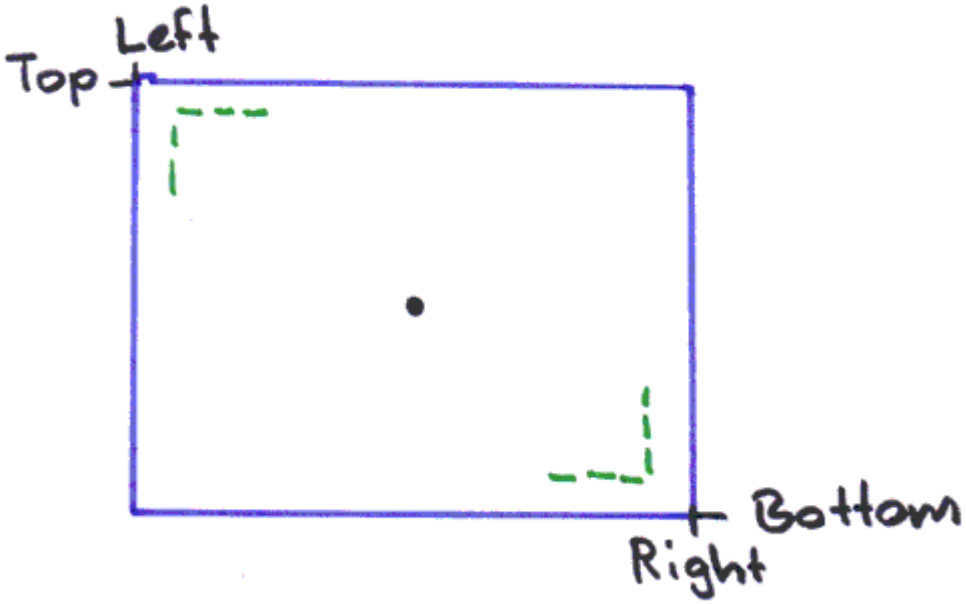
Um einen Kreis zu zeichnen, der das Fenster schön ausfüllt, müssen wir in der Bearbeitungsfunktion für die WM_PAINT Nachricht die Fenstergrösse beim Betriebssystem nachfragen.

```
// Rechteckstruktur erzeugen und
// initialisieren
RECT rect = { 0 };
// Grösse des Fensterinneren
// abfragen
GetClientRect(hWnd, &rect);
```

Mit *GetClientRect* finden wir die Grösse der inneren Fensterfläche, die uns zum Zeichnen zu Verfügung steht. Man spricht auch von der „Client-Area“. Die Funktion braucht hierfür nur das Handle des Fensters, das uns interessiert und füllt das Ergebnis in eine RECT Struktur, deren Adresse wir der Funktion übergeben.
Übrigens: In C++ wäre es nicht unbedingt nötig einen Zeiger zu übergeben, wenn die Funktion eine Referenz als Argument hätte. Die Betriebssystemfunktionen sind jedoch in C definiert, wo es keine Referenzen gibt. Wenn wir einer Betriebssystemfunktion ein Argument übergeben, das von dieser Funktion geändert wird, müssen wir demnach einen Zeiger übergeben.

Kreis zeichnen

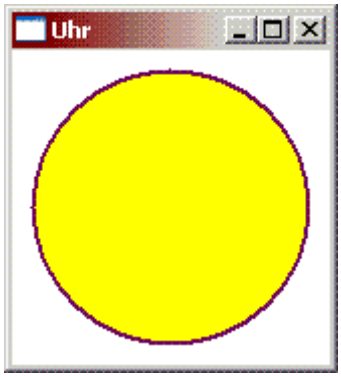
Wie man einen Kreis zeichnet, wisst ihr aus der letzten Lektion. Es ist dir selber überlassen, ob dein Kreis gelb ausgefüllt ist und einen braunen Rand hat. Wir verkleinern das Rechteck auch ein wenig, damit wir nicht bis zum Rand zeichnen.



Hier ist der Code:

```
GetClientRect(hWnd, &rect);

// Das Rechteck ein wenig
// verkleinern
rect.left += 10;
rect.top  += 10;
rect.bottom -= 10;
rect.right  -= 10;

HBRUSH brush = CreateSolidBrush(RGB(255,255,10));
HPEN   pen   = CreatePen(PS_SOLID, 2, RGB(125,0,100));
HBRUSH oldBrush = (HBRUSH)SelectObject(dc, brush);
HPEN   oldPen   = (HPEN)SelectObject(dc, pen);

Ellipse(dc, rect.left, rect.top, rect.right, rect.bottom);

SelectObject(dc, oldBrush);
SelectObject(dc, oldPen);
```

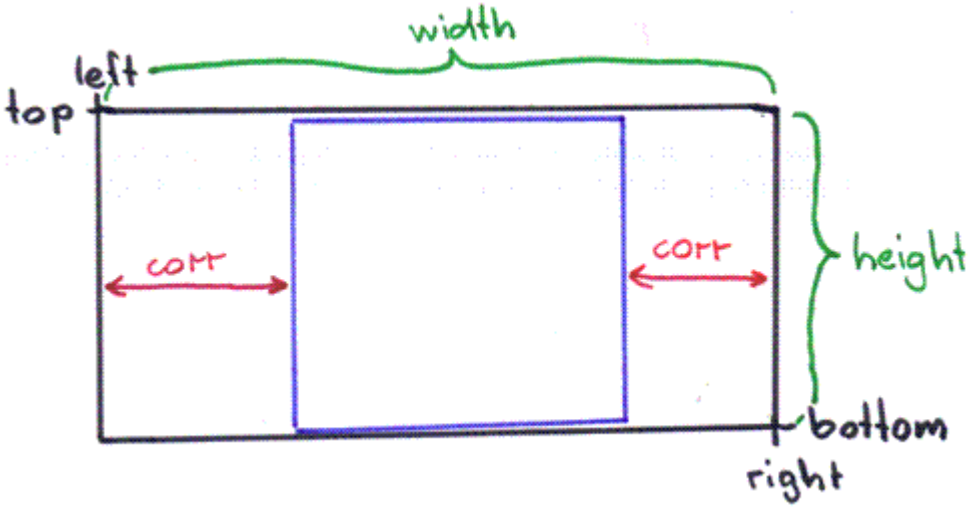
Und das Ergebnis:



Wenn ihr die Grösse des Fensters ändert, ändert sich auch der Kreis, oder besser die Ellipse, denn wir können das Fenster beliebig verziehen.
Mit ein wenig Mathematik sollten wir aber in der Lage sein, den Kreis rund zu behalten. Versuch es!

Die Funktion Quad

Diese Funktion dient also dazu aus einem Rechteck, ein Quadrat zu machen, das genügend klein ist um in dem Rechteck Platz zu haben.



Da die Funktion eine C++ Funktion sein darf, verwenden wir hier Referenzen, auch wenn es ein wenig Mischmasch zwischen Parameterübergabe per Zeiger und Parameterübergabe per Referenz gibt.

Hier ist der Code:

```
void Quad(RECT& rect)
{
    int width = rect.right - rect.left;
    int height = rect.bottom - rect.top;

    if(width < height)
    {
        int corr = (height - width) / 2;
        rect.top += corr;
        rect.bottom -= corr;
    }
    else
    {
        int corr = (width - height) / 2;
        rect.left += corr;
        rect.right -= corr;
    }
}
```

Ruf diese Funktion auf, nachdem du das Rechteck verkleinert hast und bevor du den Kreis zeichnest.

Die Uhrzeiger

Hier werden wir ein wenig Trigonometrie verwenden müssen.

Berechnen der Winkel

Aus der Zeit können wir jeweils die Winkel berechnen, die die jeweiligen Uhrzeiger mit der 12-Uhr Position aufspannen.
Der Sekundenzeiger überstreift in 60 Sekunden einen Winkel von 360° oder 2π.

```
Winkel in Grad = 360 / 60 * Sekunden = 6 * Sekunden
Winkel = 2π / 60 * Sekunden
```

Der Minutenzeiger überstreift den gleichen Winkel in 60 Minuten:

```
Winkel in Grad = 6 * Minuten
Winkel = 2π / 60 * Minuten
```

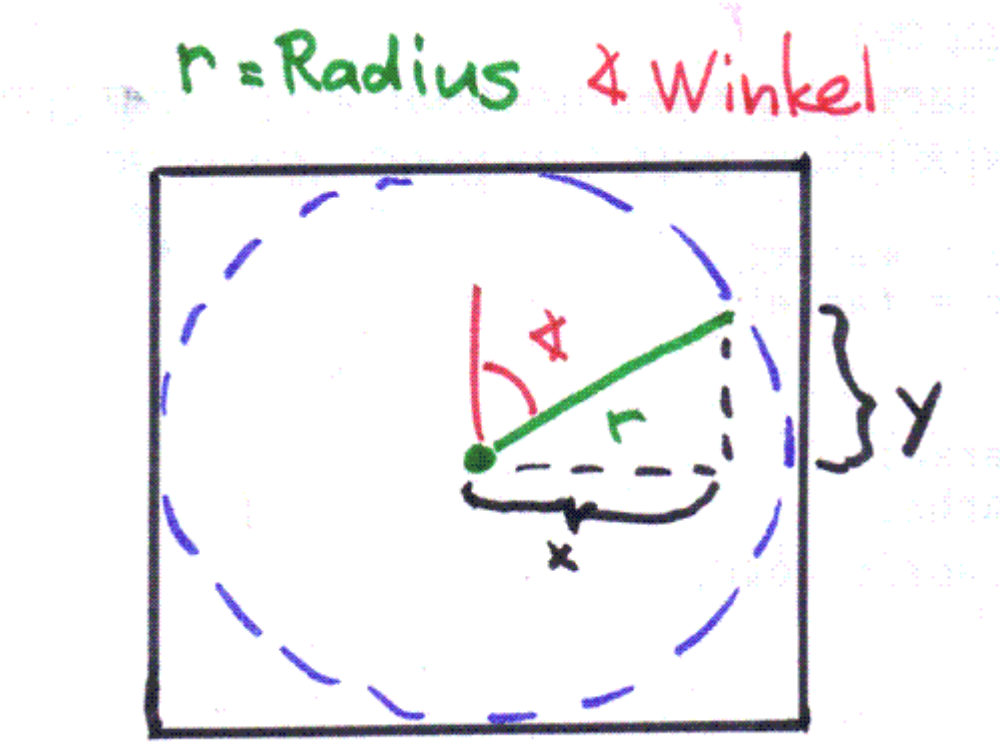
Der Stundenzeiger überstreift diesen Winkel in 12 Stunden:

```
Winkel in Grad = 360 / 12 * Stunden = 30 * Stunden
Winkel = 2π / 12 * Stunden
```

Falls wir die Stunden im 24-Stunden Format haben, rechnen wir das einfach Modulo 12.

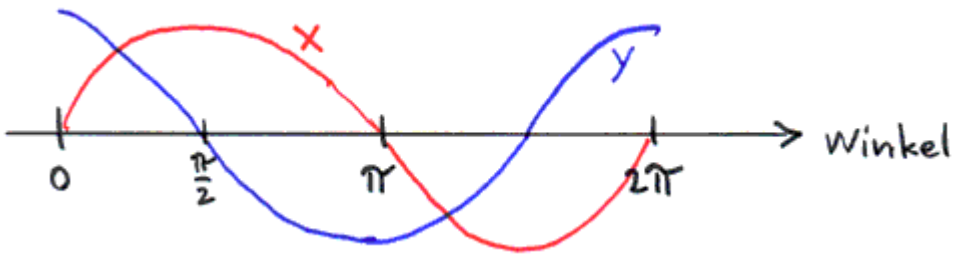
Berechnen der x- und y-Offsets

Um die Linien zu zeichnen brauchen wir einen Start- und einen Endpunkt. Der Startpunkt ist einfach zu finden und für alle drei Linien gleich. Den Endpunkt müssen wir mit Trigonometrie berechnen. Vielleicht hilft diese Zeichnung dabei:



Oder dieser Link:
http://www.mathe-online.at/galerie/wfun/wfun.html

Ich muss zugeben, dass mich die Trigonometrie hier recht gefordert hat. Das Schlimme ist, dass wenn man die Sache pragmatisch angeht und sich nicht sofort auf die Formeln stürzt, das Lösung sehr einfach zu finden ist.
Eine einfache Zeichnung macht es sehr einfach:



Durch einfaches Betrachten der Kurven sehen wir folgenden Zusammenhang:
x = r * sin(Winkel)
y = r * cos(Winkel)

Wir berechnen also x und y für jeden Uhrzeiger gleich indem wir 2π durch die Anzahl Schritte teilen, die ein Uhrzeiger machen kann und diesen Wert mal die erreichten Schritte rechnen. Diesen Winkel verwenden wir als Argument für die Sinus-Funktion oder die Kosinus-Funktion. Diese beiden trigonometrischen Funktionen finden wir in der <math.h> Header-Datei.

```
const double PI = 3.14159;
const double PIDouble = PI * 2.0;

double calculateX(long totalSteps, long actualStep)
{
    double angle = PIDouble * actualStep / totalSteps;
    double x = sin(angle);

    return x;
}

double calculateY(long totalSteps, long actualStep)
{
    double angle = PIDouble * actualStep / totalSteps;
    double y = cos(angle);

    return y;
}
```

Mit diesen zwei Funktionen können wir also berechnen wie weit wir in x- und y-Richtung wir von der Mitte aus gehen.

Die Zeit

Die Funktion time()

Diese Funktion ist in der C-Library bereits vorhanden und ist in der Datei <time.h> definiert. Diese Funktion liefert die Anzahl Sekunden, seit dem 1. Januar 1970. Man muss ihr als Argument einen Zeiger auf eine Variable vom Datentyp time_t geben.

```
// Zeit holen
time_t nowSeconds;
time(&nowSeconds);
```

Die Funktion localtime

Um die Zeit in Jahren, Monaten, Tagen, Stunden, Minuten und Sekunden zu erhalten verwenden wir die Funktion *localtime*. Die Funktion *localtime* liefert als Rückgabewert einen Zeiger auf eine tm Struktur. Das Besondere dabei ist, dass die Struktur von der Funktion erzeugt wird und nicht eine Struktur füllt, die wir anlegen müssen. Diese Struktur wird von der C-Library statisch angelegt und wird bei jedem Aufruf von *localtime* wieder überschrieben.

```
// Zeit holen
time_t nowSeconds;
time(&nowSeconds);
tm* now = localtime(&nowSeconds);
long seconds = now->tm_sec;
long minutes = now->tm_min;
long hours   = now->tm_hour % 12;
```

Zeichnen der Uhrzeiger

Berechnungen

Am besten wir fassen, das Zeichnen der drei Uhrzeiger in einer Funktion zusammen. Der Uhrzeiger heisst auf Englisch *Hand*. Ich nenne meine Funktion also *DrawHands*. Diese Funktion braucht als Argument sicher einen HDC um Linien zeichnen zu können. Zur korrekten Berechnung der Winkel und Uhrzeigerlänge brauchen wir das Rechteck in dem sich der Kreis befindet.

```
void DrawHands(HDC& dc, const RECT& rect);
```

In dieser Funktion bestimmen wir aus dem übergebenen Rechteck zuerst die halbe Seitenlänge des Quadrats, die auch dem Radius des Kreises entspricht.

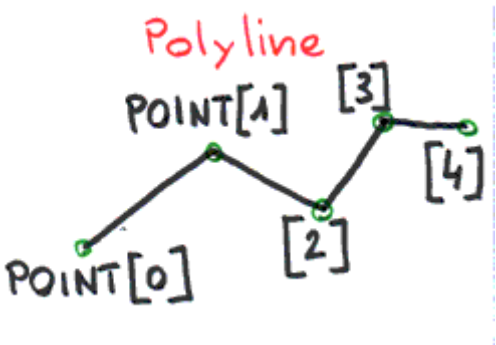
```
void DrawHands(HDC& dc, const RECT& rect)
{
    // mitte
    long middle = (rect.right - rect.left) / 2;
```

Darunter folgt bei mir die Abfrage der Zeit, wie oben beschrieben und gleich darunter die Festlegung der Uhrzeigerlängen. Hier ist es dir überlassen wie lange du die Uhrzeiger machst. Ich habe hier den Sekunden- und den Minutenzeiger gleich lang gemacht (80% des Kreisradius). Die Länge des Stundenzeigers hat entspricht dem halben Radius des Kreises.
Ihr seht darunter auch gleich die Berechnung der x- und y- Koordinaten für die Uhrzeiger.

```
// Radien
double rSeconds = 4 * middle / 5;
double rMinutes = 4 * middle / 5;
double rHours   = middle / 2;

long xSeconds = (long)(rSeconds * calculateX(60, seconds));
long ySeconds = (long)(rSeconds * calculateY(60, seconds));
long xMinutes = (long)(rMinutes * calculateX(60, minutes));
long yMinutes = (long)(rMinutes * calculateY(60, minutes));
long xHours   = (long)(rHours   * calculateX(12, hours));
long yHours   = (long)(rHours   * calculateY(12, hours));
```

Zeichnen einer Linie

Zum Zeichnen einer Linie gibt es verschiedene Möglichkeiten. Es gibt die Möglichkeit einen Stift zu simulieren mit *MoveToEx* (Stift bewegen) und *LineTo* (Mit Stift zeichnen). Wir verwenden aber die Funktion *Polyline*. Diese Funktion nimmt als Argument einen HDC (Gerätekontext zum Zeichnen) und ein Array von Elementen des Typs POINT. Das dritte Argument gibt an wie viele Elemente sich im Array befinden.



Wir werden in diesem Array nur zwei Elemente haben, nämlich den Mittelpunkt als Startpunkt und den Endpunkt jeweils mit den berechneten Koordinaten. Im Code seht ihr auch wie ich den Pen austausche um für die Uhrzeiger verschiedene Dicken und Farben zu erzeugen.
Auf der nächsten Seite findest du den gesamten Code der Funktion *DrawHands*.

```
void DrawHands(HDC& dc, const RECT& rect)
{
    // mitte
    long middle = (rect.right - rect.left) / 2;
    // Zeit holen
    time_t nowSeconds;
    time(&nowSeconds);
    tm* now = localtime(&nowSeconds);
    long seconds = now->tm_sec;
    long minutes = now->tm_min;
    long hours   = now->tm_hour % 12;

    // Radien
    double rSeconds = 4 * middle / 5;
    double rMinutes = 4 * middle / 5;
    double rHours   = middle / 2;

    long xSeconds = (long)(rSeconds * calculateX(60, seconds));
    long ySeconds = (long)(rSeconds * calculateY(60, seconds));
    long xMinutes = (long)(rMinutes * calculateX(60, minutes));
    long yMinutes = (long)(rMinutes * calculateY(60, minutes));
    long xHours   = (long)(rHours   * calculateX(12, hours));
    long yHours   = (long)(rHours   * calculateY(12, hours));

    // Mittelpunkt
    long middleX = rect.left + middle;
    long middleY = rect.top + middle;

    POINT points[2];
    points[0].x = middleX;
    points[0].y = middleY;

    // Stundenzeiger
    HPEN hourPen = CreatePen(PS_SOLID, 6, RGB(0,10,200));
    HPEN oldPen = (HPEN)(SelectObject(dc, hourPen));
    points[1].x = middleX + xHours;
    points[1].y = middleY - yHours;

    Polyline(dc, points, 2);

    // Minutenzeiger
    HPEN minutePen = CreatePen(PS_SOLID, 3, RGB(0,50,240));
    SelectObject(dc, minutePen);
    points[1].x = middleX + xMinutes;
    points[1].y = middleY - yMinutes;

    Polyline(dc, points, 2);

    // Sekundenzeiger
    HPEN secondPen = CreatePen(PS_SOLID, 1, RGB(0,0,0));
    SelectObject(dc, secondPen);
    points[1].x = middleX + xSeconds;
    points[1].y = middleY - ySeconds;

    Polyline(dc, points, 2);

    SelectObject(dc, oldPen);
}
```

Der Timer

Jetzt müssen wir nur noch dafür sorgen, dass unser Fenster regelmässig zum Beispiel jede Sekunden frisch gezeichnet wird.

Aufsetzen eines Timers

Das Betriebssystem bietet so genannte Timer an, die wir dazu verwenden können in einem bestimmten Intervall unser Fenster zu benachrichtigen. Natürlich geschieht dies über eine Windows-Message. Die Nachricht heisst WM_TIMER. Das Starten des Timers übernimmt die Funktion *SetTimer*.

```
SetTimer(hWnd,// Fenster das WM_TIMER Nachrichten erhalten soll
         0,    // darf 0 sein
         1000, // Intervall in Millisekunden
         0);   // muss 0 sein
```

Mit *KillTimer* wird der Timer wieder zerstört.

Die WM_CREATE Nachricht

Diese Nachricht wird an ein Fenster gesendet, wenn es erzeugt wird. Wir reagieren auf diese Nachricht und setzen dort den Timer auf.

Die WM_TIMER Nachricht und Invalidate

Wir erhalten mit dem Intervall, das wir in SetTimer gewählt haben ein WM_TIMER Nachricht. Als Reaktion sagen wir dem Betriebssystem einfach, dass unser Fenster ungültig. Das Betriebssystem seinerseits erzeugt als Reaktion eine WM_PAINT Nachricht.

```
case WM_CREATE:
    SetTimer(hWnd, // Fenster für WM_TIMER
             0,    // darf 0 sein
             1000, // Intervall in Millisekunden
             0);   // muss 0 sein
break;
case WM_TIMER:
    InvalidateRect(hWnd, 0, TRUE); // alles ungültig
break;
case WM_DESTROY:
    KillTimer(hWnd, 0);
    PostQuitMessage(0);
break;
```

Hier das Ergebnis: