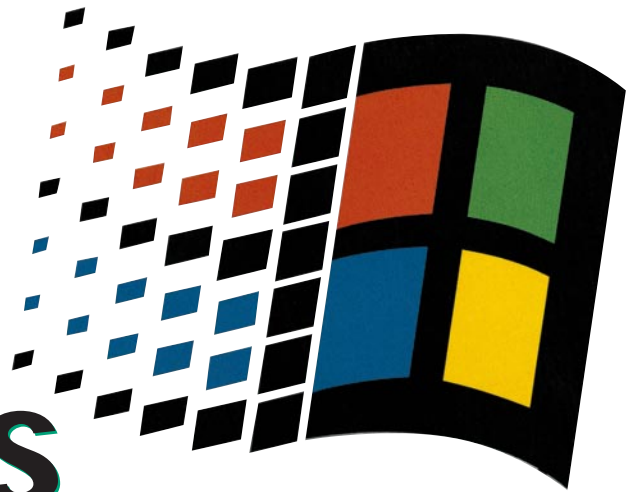




Praxis: Interaktives Malprogramm

FensterIn in Windows



In diesem Sonderheft haben Sie **viel über C++** erfahren – **nun wird es Zeit** für die Praxis: Sie schreiben **Ihr erstes richtiges, interaktives Windows-Programm**.

THOMAS WÖLFER

Wenn Sie sich mit Visual C++ noch nicht so recht auskennen, sollten Sie sich zunächst mit dem Beitrag "Einführung in C++" auf Seite 8 befassen, denn dort werden die Grundlagen für diesen Beitrag gelegt. Die Oberfläche sollten Sie bereits verwenden können, bevor Sie jetzt ein echtes, interaktives Windows-Programm schreiben.

Als Beispiel dient diesmal ein kleines Malprogramm. Am Ende des Artikels

wird das Programm ein ganz normales Windows-Programm sein, einschließlich Ausdruck und einer Druckvorschau. Gedruckt wird der Inhalt der Arbeitsfläche. Was sich darin befindet, bestimmen Sie, denn mit dem fertigen Programm können einfache Zeichnungen erstellt werden. Die Zeichnungen sind zwar simpel und bestehen ausschließlich aus Linien und Kreisen, doch das Programm erlaubt die Eingabe beliebig vieler solcher Elemente, interaktiv und ohne Koordinaten zu tippen.

■ SDI-Anwendung erzeugen

Zunächst muss wie immer ein neues Projekt angelegt werden, diesmal eine SDI-Anwendung. Das SDI steht für "Single Document Interface" und bedeutet, dass das Programm nur ein Dokument auf einmal bearbeiten kann. Neue Versionen von Word funktionieren ebenso, im Gegensatz zu "alten" Word-Versionen, bei denen immer mehrere Dokumente gleichzeitig mit einer Kopie von Word bearbeitet werden konnten.

Abgesehen von der Tatsache, dass Sie das Projekt als SDI-Anwendung anlegen müssen, sollten Sie, damit Sie dem Beispiel besser folgen können, beim Erzeugen des Projektes folgendes beachten: Geben Sie als Namen für die Anwendung den hier im Artikel verwendeten an, also "FensterIn". Der Grund dafür ist relativ einfach: Der VC++ Application Wizard erzeugt eine

ganze Reihe von Klassen und Dateien. Deren Namen resultieren alle aus den angegebenen Namen des Projektes. Nachdem im Beitrag häufig auf diese Dateien und Klassen per Name verwiesen wird, ist es einfacher, wenn diese mit den Ihren übereinstimmen.

Ist der Wizard abgearbeitet, liegt bereits ein fertig übersetzbares Programm vor. Starten Sie es ruhig einmal, um zu überprüfen, welche Funktionalität bereits im Programm enthalten ist. Es besteht aus einer Arbeitsfläche (mit der allerdings noch nicht viel anzufangen ist), hat bereits ein Menü mit den gängigen Menüpunkten sowie eine Werkzeugleiste, die einen schnelleren Zugriff auf einige der Menübefehle erlaubt. Außerdem gibt es eine "About"-Box im Hilfe-Menü, und auch der Befehl "Datei - Beenden" funktioniert schon. Eine fertige Druckvorschau besteht ebenfalls, auch wenn es noch nicht viel gibt, was man darin sehen könnte, da die zu druckenden Daten noch fehlen.

Das Programm soll nach seiner Fertigstellung zwei Befehle aufweisen: Mit dem einen können Kreise gezeichnet, mit dem anderen Linien gemalt werden. Diese Befehle sollen sowohl im Menü als auch in der Werkzeugleiste zur Verfügung stehen. Zunächst die Menüleiste:

■ Menüs bauen

Für das Editieren der Menüleiste muss zunächst der Menü-Editor gestartet werden. Das funktioniert, indem man die Menü-Ressource im Workspace unter "Ressourcen" per Doppelklick lädt. Durch den Doppelklick landet das Menü im VC++-Arbeitsbereich, der zugehörige Menü-Editor wird automatisch geladen. Zunächst wird ein neuer Top-Level-Eintrag im Menü eingefügt, der die Bezeichnung "Elemente" trägt. Zum Einfügen dieses Eintrags kann der

STEP BY STEP

- ▶ **SDI-Anwendung erzeugen**
Konzentration auf eine Anwendung
- ▶ **Menüs bauen**
Der Menü-Editor hilft strukturieren
- ▶ **Eine Basisklasse wird zum Vererben gebraucht**
Von der Klasse Geo werden Kreis und Kinie abgeleitet
- ▶ **Funktionen für die View-Klasse**
Man will ja etwas sehen
- ▶ **Richtig malen**
Linie und Kreis ersetzen die MessageBox
- ▶ **Der Mausbewegung folgen**
Linie wie Gummi ziehen
- ▶ **Refresh muß sein**
Das Kunstwerk darf nicht einfach verschwinden
- ▶ **Aufräumen!**
Der Destruktor sollte schon ordentlich arbeiten



"leere" Eintrag, der zunächst am Ende des Menüs erscheint, an die gewünscht Stelle verschoben werden. Ein Doppelklick öffnet dann dessen "Eigenschaften"-Fenster, in dem der Text "Elemente" angegeben werden kann. Die beiden darunter liegenden Befehle erhalten die Texte "Kreis" und "Linie". Bei den Befehlen "Kreis" und "Linie" muss noch mehr passieren: Diese Befehle benötigen eine ID. Diese ist das Symbol, das im Programmcode verwendet werden kann, wenn auf die Befehle Bezug genommen werden soll. Auch an einer späteren Stelle im ClassWizard werden diese IDs benötigt. Praktischerweise vergibt der Menü-Editor von sich aus sinnvolle IDs. Die ID für den Befehl Kreis lautet, sofern daran manuell nicht verändert wird, ID_ELEMENTE_KREIS, der für die Linien trägt analog den Namen ID_ELEMENTE_LINIE. Unten auf dem "Eigenschaften"-Dialog für den Menübefehl befindet sich ein Edit-Control, in das Text eingegeben werden kann. Hier gibt man den Text ein, den das Programm in der Statuszeile anzeigen soll, wenn der Anwender mit dem Mauscursor über den Menübefehl fährt, die Schnellhilfe sozusagen. Dieser Text besteht aus zwei Teilen, die durch ein Newline-Zeichen (\n) voneinander getrennt werden. Der erste Teil wird als Schnellhilfe für das Menü verwendet, der zweite ist der Tooltip, der für die noch zu definierenden Buttons in der Werkzeugleiste zuständig ist.

Nun sind die Buttons in der Werkzeugleiste an der Reihe. Die Bearbeitung erfolgt ähnlich wie beim Menü: Werkzeugleiste per Doppelklick aus dem Workspace laden, neue Buttons einfügen und IDs vergeben. Die beiden Buttons werden mit einer Grafik ausgestattet. Dazu bietet der Toolbar-Editor die passenden Werkzeuge. Als ID wird für den "Kreis"-Button die gleiche vergeben wie für den Menübefehl "Kreis", entsprechend wird mit dem "Linien"-Button verfahren. Auf dem Eigenschaften-Dialog für die Buttons befindet sich eine Combo-Box, in der nach der passenden ID gesucht werden kann, denn diese wurde bereits im Menü definiert. Sobald die richtige ID ausgewählt ist, erscheint der Hilfstext, der zuvor im Menü-Editor eingegeben wurde. Der Hilfstext ist logisch mit der ID verknüpft. Da Button und Menübefehl die gleiche ID verwenden, bekommen beide den gleichen Text.

Nach diesen Änderungen sollte man das Programm erneut übersetzen und ausprobieren. Nach dem Programmstart sind die neuen Buttons und Menübefehle sofort sichtbar, allerdings grau hinterlegt, sie können nicht angewählt werden. Deshalb nicht, weil noch keine Funktionen implementiert wurden, die aufgerufen werden sollen, wenn die Befehle ausgewählt werden. Erst wenn das der Fall ist, kann man die neuen Befehle tatsächlich anwählen.

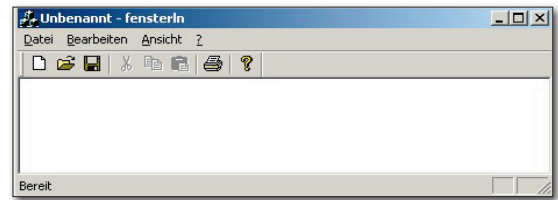
■ Eine Basisklasse wird gebaut

Bevor diese Funktionen geschrieben werden können, muss einiges an Vorarbeit geleistet werden. Damit die Linien und Kreise gespeichert werden können, müssen zunächst einige Klassen entwickelt werden, welche die Daten für diese Elemente aufnehmen können. Der Einfachheit halber werden alle dazu benötigten Klassendefinitionen in einem Header-File Namens Geo.h verpackt und die relativ einfachen Implementierungen gleich inline in der Klassendefinition belassen.

Zunächst wird eine Basisklasse definiert. Es könnte sein, dass zu einem späteren Zeitpunkt noch weitere Klassen mit ähnlichen Eigenschaften hinzukommen. Die Basisklasse bekommt den Namen "Geo" und die folgende Implementierung bzw. Definition:

```
class Geo
{
public:
    Geo() {}
    virtual void Draw( CDC* pDC) = 0;
};
```

Davon wird nun eine Klasse für den Kreis und eine weitere für die Linie abgeleitet. Diese Klassen sehen zunächst



DIREKT NACH DER Erzeugung mit dem AppWizard macht das Programm bereits einen recht vollständigen Eindruck – und das, obwohl es noch keine echte Funktionalität besitzt.

wie folgt aus:

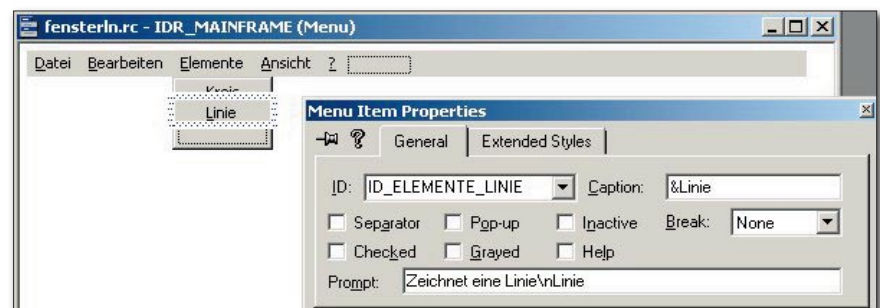
```
class Kreis : public Geo
{
public:
    Kreis();
    Kreis( CPoint center, CPoint rad)
    {
        m_center = center;
        m_rad = rad;
    }

    void Draw( CDC* pDC)
    {
        TRACE("KREIS!");
    }
private:
    CPoint m_center, m_rad;
};

class Linie : public Geo
{
public:
    Linie() {}
    Linie( CPoint a, CPoint b)
    {
        m_a = a;
        m_b = b;
    }

    void Draw( CDC* pDC)
    {
        TRACE("LINIE!");
    }
private:
    CPoint m_a, m_b;
};
```

Beim Kreis werden ein Zentrumspunkt und ein Radius gespeichert, bei der Linie der Anfangs- und der Endpunkt. Die Draw-Methoden werden noch nicht implementiert, sondern zunächst nur mit den TRACE()-Aufrufen bestückt. Die bei TRACE() angegebenen Texte erscheinen beim Funktionsaufruf im



MIT DEM MENÜ-EDITOR können nicht nur die Texte für das Menü, sondern auch die zugehörigen kurzen Hilfetexte eingegeben werden.

"Output"-Fenster des Debuggers, sofern sich das Programm unter der Kontrolle eines Debuggers befindet.

Schließlich sollen die Linien und Kreise noch in einer gemeinsamen Liste verwaltet werden. Dazu werden die MFC Collections Classes verwendet. Damit dies gelingt, muss die Datei `AfxTempl.h` im Header-File inkludiert werden; und damit die Bezeichnung für die Liste einfacher zu schreiben ist, wird ein `typedef` verwendet:

```
typedef CTypedPtrList<CPtrList,
Geo*> GeoList ;
```

Nun muss die Datei `Geo.h` noch in `FensterInDoc.h` inkludiert und eine Instanz der `GeoList` als Member in der `FensterInDoc`-Klasse eingebettet werden. In der Dokumenten-Klasse werden die Daten der Kreise und Linien dann später aufbewahrt.

Damit neu erzeugte Elemente später an die Liste angehängt werden können, wird die `FensterInDoc`-Klasse um zwei öffentliche Methode erweitert: eine, um die Liste zu erweitern, und eine, um einen Zeiger auf die Liste zu ermitteln. Sauberer wäre es hier, `FensterInDoc` um Methoden zur Iteration über die Liste zu erweitern, aber der gewählte Weg ist für dieses Beispiel sicherlich ausreichend.

```
GeoList* GetGeoList() { return
&m_list; }
```

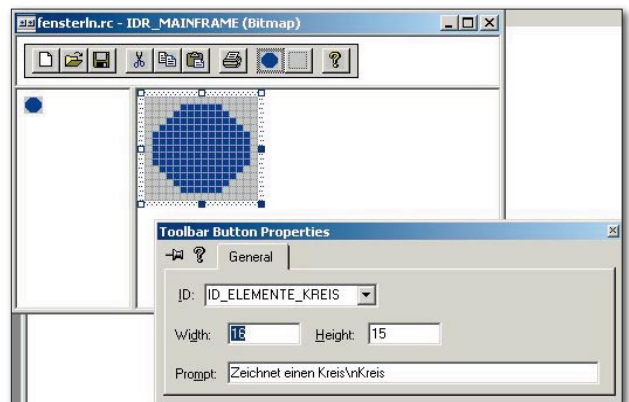
```
void AddGeo( Geo* p) {
m_list.AddTail( p); }
```

Funktionen für die View-Klasse

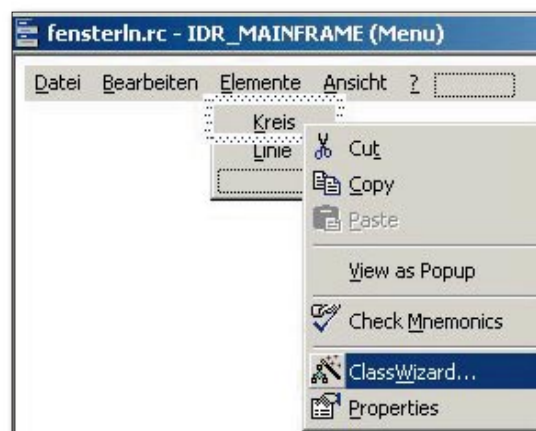
Die Dokumenten-Klasse ist nun für die Aufnahme von Kreis- und Linien-Daten vorbereitet. Es ist daher an der Zeit,

die Funktionen zu entwickeln, mit denen diese Daten erzeugt werden können.

Dafür wird am besten zunächst wieder der Class Wizard bemüht. Die nun zu implementierenden Funktionen sollen die zu Anfang ins Menü aufgenommenen Befehle behandeln. Daher wird nun mit der rechten Maustaste auf den Befehl "Kreis"



IM TOOLBAR-EDITOR werden im Wesentlichen die Bilder für die Buttons gemalt. IDs und Hilfstext kommen aus den zuvor definierten Menübefehlen.



MIT DEM CLASSWIZARD werden die Funktionen an die Menübefehle gebunden.

geklickt und im angezeigten Objekt-Menü der "Class Wizard" ausgewählt. Dieser zeigt sich in Form einer von anderer Stelle her bekannten umfangreichen Dialogbox. Hier ist es nun wichtig, dass man die richtigen Elemente auswählt, damit hinterher alles wie gewünscht funktioniert.

Als "Class Name" muss zunächst "CFensterInView" ausgewählt werden. Das stellt sicher, dass die neuen Funktionen in der View-Klasse landen. Genau in diese Klasse gehören sie hinein, denn die View ist für die optische Repräsentation der Daten, also auch für deren Eingabe, zuständig. Als "ObjectID" wird `ID_ELEMENT_KREIS` ausgewählt. Damit wird die Behandlung dieser ID in `CFensterInView` festgelegt. Abschließend wird als "Command" der Wert `COMMAND` ausge-

wählt. Die Funktion soll das Kommando und nicht `UPDATE_COMMAND_UI` (also das Update des Kommando-Interfaces, vulgo des Buttons) behandeln. Mit dem Button "Add Function" wird die Funktion endgültig der View hinzugefügt. Danach erfolgt das gleiche noch einmal, diesmal jedoch mit `ID_ELEMENT_LINE`. Schließlich wird erneut "Edit Code" betätigt: Der ClassWizard verschwindet und der Cursor befindet sich im Quellcode-Editor, und zwar in der Datei `FensterIn-`

View innerhalb der Funktion, die nun für das Handling der Buttons zuständig sein soll.

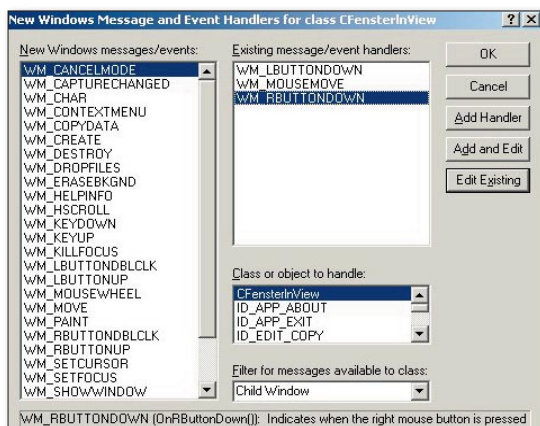
Um diese Funktionalität zu testen, wird die Funktion mit einer einzigen Zeile aufgefüllt:

```
void CFensterInView::OnElemente
Linie()
{
AfxMessageBox("OnElement
Linie");
}
```

Anschließend wird das Programm erneut übersetzt und gestartet – sowohl der Button für den Kreis als auch der für die Linie können nun angeklickt werden. Wird auf den Linien-Button geklickt, erscheint die Dialogbox mit der zuvor programmierten Meldung.

Richtig malen

Nun muss noch gemalt werden, denn die Dialogbox ist auf Dauer natürlich keine zufriedenstellende Lösung. Auch fürs Malen ist zunächst noch Vorarbeit zu leisten. In der Datei `fensterInView.h`



DER CLASSWIZARD macht es einfach, die benötigten Windows-Meldungen innerhalb der gewünschten Klasse zu behandeln.



MIT EINER EINFACHEN
`MessageBox()`-
Aufruf kann man
leicht überprüfen,
ob die richtige Funktion aufgerufen wird.

werden einige Konstanten definiert, die später feststellen sollen, in welchem "Mal-Modus" sich das Programm befindet:

```
const int STATE_NONE = 0;
const int STATE_BEGIN_LINE = 1;
const int STATE_END_LINE = 2;
const int STATE_BEGIN_CIRCLE = 3;
const int STATE_END_CIRCLE = 4;
```

In der gleichen Datei, aber als Member der `CFensterlnView`-Klasse, werden ferner einige Variablen definiert, die zum "Merken" des Zustands und einiger Punkte gedacht sind:

```
private:
    int m_state;
    CPoint m_ptBegin;
    CPoint m_ptLast;
```

Die Variable "m_state" ist dabei die, die über den aktuellen Zustand des Programms Aufschluss geben soll. Daher nicht vergessen, diese Variable im Konstruktor zu `FensterlnView` (zu finden in `FensterlnView.cpp`) mit `STATE_NONE` zu initialisieren.

Schließlich müssen noch mit Hilfe des Class Wizards einige Funktionen zu `FensterlnView` hinzugefügt werden. Dazu wird der Class Wizard über das "View"-Menü geöffnet. Nachdem sichergestellt ist, dass als "ClassName" `CFensterlnView` ausgewählt ist, werden unter Messages `WM_MOUSEMOVE`, `WM_RBUTTONDOWN` und `WM_LBUTTONDOWN` mit Hilfe von "Add Function" mit einem

Message-Handler versorgt. Bei diesen `WM_*`-Symbolen handelt es sich um die Nachrichten, die die View-Klasse erhält, wenn sich die Maus über ihr befindet bzw. der linke oder der rechte Mausknopf gedrückt werden. Inhaltlich soll das Ganze im fertigen Zustand folgendermaßen aussehen: Der Anwender drückt auf den "Linie"-Button und kann dann mit der linken Maustaste beliebig in die Arbeitsfläche klicken. Ab diesem ersten Klick wird eine Linie gezeichnet, die bei der angeklickten Stelle beginnt und ansonsten dem Mauszeiger folgt. Beim zweiten Klick

LISTING 1

```
void CFensterlnView::OnElemente
Linie()
{
    GetCapture();
    m_state = STATE_BEGIN_LINE;
}
```

LISTING 2

```
void CFensterlnView::OnMouse
Move(UINT nFlags, CPoint point)
{
    if(STATE_END_LINE == m_state)
    {
        CDC* pdc = GetDC();
        int nOld = pdc->SetROP2( R2_NOT);
        pdc->MoveTo( m_ptBegin);
        pdc->LineTo( m_ptLast);
        pdc->MoveTo( m_ptBegin);
        pdc->LineTo( point);
        m_ptLast = point;
        pdc->SetROP2( nOld);
    }
    CView::OnMouseMove(nFlags, point);
}
```

LISTING 3

```
void CFensterlnView::OnLButtonDown(UINT nFlags, CPoint point)
{
    if( STATE_BEGIN_LINE == m_state)
    {
        m_ptBegin = point;
        m_state = STATE_END_LINE;
    }
    else if( STATE_END_LINE == m_state)
    {
        ReleaseCapture();
        m_state = STATE_NONE;
        Linie* pLinie = new Linie( m_ptBegin, point);
        GetDocument()->AddGeo( pLinie);
    }

    CView::OnLButtonDown(nFlags, point);
}
```



```

Loaded 'C:\WINNT\system32\RPCRT4.dll', no matching symbolic information found.
Loaded 'C:\WINNT\system32\COMCTL32.dll', no matching symbolic information found.
KRIS detected memory leaks!
Dumping objects ->
C:\Artikel\c++sonderheft\fensterln\fensterln\fensterlnView.cpp(231) : {446} normal block at 0x002F5AD0,
Data: < pa ? > 1C 70 41 00 1B 01 00 00 3F 02 00 00 A1 00 00 00
C:\Artikel\c++sonderheft\fensterln\fensterln\fensterlnView.cpp(231) : {341} normal block at 0x002F5A78,
Data: < pa R J > 1C 70 41 00 52 02 00 00 A3 01 00 00 4A 03 00 00
C:\Artikel\c++sonderheft\fensterln\fensterln\fensterlnView.cpp(239) : {235} normal block at 0x002F5968,
Data: < xa > C4 78 41 00 EF 00 00 00 7D 00 00 00 7B 01 00 00
Object dump complete.
The thread 0x730 has exited with code 0 (0x0).
The program 'C:\ARTIKEL\C++SONDERHEFT\FENSTERLN\Fensterln\Debug\Fensterln.exe' has exited with code 0 (

```

MEMORY-LEAKS! So etwas muss man auf jeden Fall vermeiden.

wird die Linie abgeschlossen und gespeichert. Ferner soll das Ganze mit der rechten Maustaste abgebrochen werden können. Dieser Vorgang wird im folgenden nur für die Linie exemplarisch beschrieben. Funktional unterscheidet sich der Programmcode, der sich um den Kreis kümmert, davon nicht. Abgesehen natürlich von der Tatsache, dass dort ein Kreis ausgegeben wird.

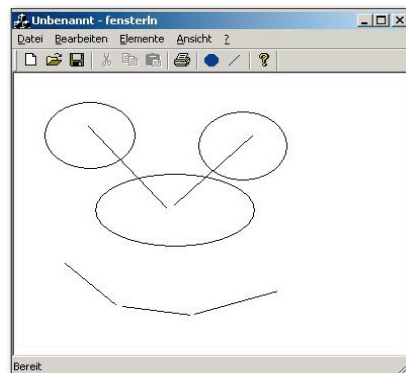
Zunächst muss die `MessageBox()` aus der Funktion `OnElementLinie()` in `FensterlnView.cpp` durch folgenden Code ersetzt werden: (Listing 1, S. 57)

Hier wird die View in einen Status versetzt, dieser ist nun `STATE_BEGIN_LINE`. Weitere Funktionen können jetzt auf diesen Status reagieren. Von Seiten des Anwenders ist noch nichts passiert. Er hat nur auf den "Linien"-Knopf gedrückt. Im nächsten Schritt klickt der Anwender in die Arbeitsfläche. Das führt dazu, dass `OnLButtonDown()` aufgerufen wird. Diese Funktion muss nun reagieren, und das tut sie mit folgendem Code: (Listing 2, S. 57). Hier finden gleich zwei Schritte statt: Befindet sich das Programm im Modus `STATE_BEGIN_LINE`, so wird der beim Funktionsaufruf übergebene Punkt gemerkt und in einen neuen State – `STATE_END_LINE` – gewechselt. Befindet sich das Programm hingegen schon im Zustand `STATE_END_LINE`, so wird der State auf `STATE_NONE` zurückgesetzt, ein neues Objekt vom Typ Linie konstruiert und per `AddGeo()` in der Dokumenten-Klasse abgelegt.

Der Mausbewegung folgen

Auf diese Weise sind zwei Punkte für die Linie gespeichert, und zwar beide in der gleichen Funktion. Während der Anwender die Maus vom zuerst angeklickten Punkt zum zweiten fährt, soll die Linie natürlich als Gummibandcursor am Bildschirm zu sehen sein. Dafür ist die Funktion `OnMouseMove()` zuständig: (Listing 3, S. 57).

Diese Funktion zeichnet im Zustand `STATE_END_LINE`, während auf den abschließenden Klick für die Linie gewartet wird, ein Gummiband zwischen dem Start und dem aktuellen Punkt. Damit die Linien wieder entfernt werden können, also beim nächsten Aufruf von `OnMouseMove()`; wird der momentan aktuelle Punkt gespeichert. Als Mal-Modus wird `R2_NOT` verwendet, das macht die Ausgabe des Cursors am einfachsten. Beim ersten Zeichnen der Linie in diesem Modus wird sie



DAS FERTIGE PROGRAMM ist ein einfaches Malprogramm. Erweiterungen steht nichts im Wege.

angezeigt, beim zweiten Zeichnen der gleichen Linie verschwindet sie wieder. Schließlich soll das Ganze noch per rechter Maustaste abgebrochen werden können. Dafür ist die Funktion `OnRButtonDown()` zuständig:

```

void CFensterlnView::OnRButtonDown(
UINT nFlags, CPoint point)
{
    ReleaseCapture();
    m_state = STATE_NONE;
    CView::OnRButtonDown(nFlags,
point);
}

```

Refresh muß sein

Nun kann das Programm erneut übersetzt und gestartet werden: Es überrascht nicht, dass man nun Linien per Mausklick malen kann. Wird das Anwendungsfenster allerdings in der

Größe verändert, verschwinden die Linien und erscheinen nicht wieder. Dies liegt daran, dass die Daten für die Linien zwar gemerkt werden, ansonsten passiert damit nichts. Die Linien werden nur ein einziges Mal ausgegeben, und zwar beim Erzeugen, danach nicht mehr. Dies ist ein unhaltbarer Zustand, der allerdings leicht ausgebessert werden kann. Benötigt wird hierzu zunächst die Funktion `CFensterlnView::Draw()` aus der Datei `fensterlnView.cpp`. In dieser Funktion muss die Liste der bisher gemerkten Elemente erfragt werden (dafür wurde am Anfang die `GetGeoList()`-Funktion implementiert) und die enthaltenen Elemente müssen ausgegeben werden:

```

void CFensterlnView::OnDraw(CDC*
pDC)
{
    CFensterlnDoc* pDoc =
GetDocument();
    GeoList* pList = Get
Document()->GetGeoList();
    for( POSITION pos =
pList->GetHeadPosition(); pos;)
    {
        Geo* pG =
pList->GetNext( pos);
        pG->Draw( pDC);
    }
}

```

Das endgültige Zeichnen der Elemente muss ebenfalls geregelt sein, denn bisher enthält `Linie::Draw()` nur einen `TRACE()`-Aufruf. Die fertige Funktion hat folgenden Aufbau:

```

void Draw( CDC* pDC)
{
    pDC->MoveTo( m_a);
    pDC->LineTo( m_b);
}

```

Aufräumen!

Erneut kann das Programm übersetzt und ausprobiert werden, und – siehe da – die Linien bleiben erhalten. Ein letzter Wermutstropfen bleibt. Dies sind Speicherbereiche, die nicht freigegeben werden. Der Grund dafür: In dem Mouse-Message Handler (`OnLButtonDown()`) werden neue Objekte an keiner Stelle freigegeben. Das besorgt zum Schluss eine kleine Erweiterung im Destruktor der `FensterlnDoc`-Klasse:

```

CFensterlnDoc::~CFensterlnDoc()
{
    while( ! m_list.IsEmpty())
        delete m_list.

    RemoveHead();
}

```

Mit diesem Programm haben Sie alle wesentlichen Elemente eines "echten" Windows-Programms im Griff.

UR