Eine plattform– und sprachenübergreifende Erweiterung für graphische Benutzerschnittstellen

Inhaltsverzeichnis

1 Einleitung **5**

2 Die Komponenten **11**

2.1 Frame 11
2.2 Menu 13
2.3 Canvas 18
2.4 Button 23
2.5 Label 28
2.6 Checkbox 31
2.7 Radiobutton 33
2.8 Choice 35
2.9 List 37
2.10 Textarea 40
2.11 Textfield 43
2.12 Popupmenu 44
2.13 Scrollbar 46
2.14 Panel 47
2.15 Scrollpane 49
2.16 Dialog 51
2.17 Window 54
2.18 Filedialog 55

3 Die Layoutmanager **59**

3.1 Flowlayout 60
3.2 Gridlayout 64
3.3 Borderlayout 66

4 Die Listener **69**

4.1 Mouse Listener 69
4.2 Key Listener 72
4.3 Focus Listener 73

5 Grafik in JAPI **75**

5.1 Farben 75
5.2 Grafikbefehle 79
5.3 Fonts 86

5.4 Cursor 89
5.5 Bilder 92

6 Nützliches **99**
6.1 Debugmodus 99
6.2 Drucken 99

Kapitel 1

Einleitung

Was ist JAPI?

JAPI (Java Application Programming Interface) ist eine Bibliothek zum Entwickeln grafischer Benutzeroberflächen. Sie ist sowohl für einen Programmieranfänger geeignet, der noch relativ wenig Erfahrungen besitzt, aber dennoch erste graphische Applikationen entwickeln möchte. Aber auch einem Profi bietet sie die Möglichkeit, umfangreiche Projekte schnell und effektiv zu programmieren. Bei der Entwicklung dieser Bibliothek standen drei wesentliche Aspekte im Vordergrund:

1. Die Entwicklung einer Oberfläche soll einfach, schnell und intuitiv erfolgen.
2. Sie läßt sich unter den (nicht objektorientierten) Programmiersprachen C, Fortran und Pascal einbinden (Sprachenunabhängigkeit).
3. Die entstandenen Programme sind plattformunabhängig.

Der erste Punkt wurde erreicht, indem bereits während der Entwicklung der Bibliothek kleine Applikationen geschrieben wurden, um festzustellen, wie das gewünschte Ziel mit möglichst wenig Programmieraufwand zu erreichen ist. Die Bibliothek ist somit direkt der Praxis entwachsen. Alle graphischen Elemente besitzen bespielsweise eine *vernünftige* Voreinstellung, die immer ein gelungenes Aussehen der Applikation garantieren. So ermitteln zum Beispiel die Schaltflächen (Button) ihre minimalen Ausmaße indem die Größe des enthaltenen Schriftzuges ermittelt wird. Dennoch kann einem Button selbstverständlich eine explizite Größe zugewiesen werden. Solche Manipulationsfunktionen sind vielfältig verhanden, werden aber in der Regel kaum gebraucht.

Der zweite Entwicklungsschwerpunkt umfaßt die Sprachenunabhängigkeit. Ein erster Schritt in diese Richtung ist durch die GNU Compilerfamilie vorgegeben, die die drei genannten Sprachen umfaßt. Zwar benötigen der GNU Pascal und der GNU Fortran Compiler auch weiterhin einen C Compiler. Dennoch sind sie keine einfachen Pascal2C oder Fortran2C Converter, sondern eigenständige Entwicklungen, die lediglich auf den GNU-Compiler aufsetzen. Diese Verwandschaft erleichert erheblich die Entwicklung einer gemeinsamen Bibliothek für diese Programmiersprachen.

In der Mircosoft Welt sind es vor allem die DLL (Dynamic Link Libraries), die das Einbinden von fremden Bibliotheksfunktionen erleichtern. Leider gibt es jedoch auch dort

offensichtlich feine Unterschiede zwischen Bibliotheken, die z.B. mit dem Microsoft C Compiler und dem Borland C Compiler erstellt wurden.

Der Kern der JAPI Bibliothek wurde in C und in JAVA geschrieben. Der JAVA Teil übernimmt die eigentlich graphische Ausgabe, und läuft als ein eigenständiges Programm. Der Rest der Bibliothek, der komplett in C verfasst wurde, kommuniziert mit dem JAVA Kern und bietet die Schnittstellen für die Programmiersprachen.

Das JAVA AWT (Abstract Windowing Toolkit) war eine wichtige Voraussetzung für die Entwicklung dieser Bibliothek. Dieses Kit ermöglicht die Entwicklung plattformunabhängiger graphischer Benutzeroberflächen. Die JAPI Bibliothek stellt genau diese Fähigkeit den nicht objektorientierten Sprachen zur Verfügung. Die starke Verbreitung von JAVA läßt darauf hoffen, daß das Ziel einer umfassenden Plattformunabhängigkeit nicht zu hoch angesetzt wurde. Tatsächlich beschränkt sich diese erste Version zunächst nur auf zwei Plattformen, LINUX und die WIN32 Familie (Win95/98 und NT). Solaris sollte eigentlich kein Problem darstellen, und wird sicherlich in Kürze folgen. Weitere UNIX Plattformen sind ebenfalls denkbar, allerdings fehlt zur Zeit die Möglichkeit, um auf diese Plattformen zuzugreifen. Für die alten MAC Plattformen gilt prinzipiel das gleiche. Für das neue MACOS X, das ebenfalls auf den GNU Compilern basiert, stehen die Chancen ebenfalls gut. Bei BeOS hängt eigentlich alles nur von einer gelungenen JAVA Portierung ab.

Was sind nun die Voraussetzungen um JAPI zu benutzen?

Zunächst werden grundlegende Kenntnisse in einer der drei genannten Programmiersprachen gebraucht. Man muss zwar kein Profi zu sein, um graphische Benutzeroberflächen zu entwickeln. Ein Basiswissen über die verwendete Programmiersprache wird in diesem Buch jedoch vorausgesetzt, da dieses Buch kein Lehrbuch zum Erlernen einer Programmiersprache darstellt.

Auch die Voraussetzungen auf der Hardware Ebene sind eher bescheiden. Ein Pentium (P5) sollte es aber schon sein. Da auch die JAVA Laufzeitumgebung einen gewissen Anspruch an Hauptspeicher besitzt, sollte das System mindenstens 32 MByte Hauptspeicher besitzen. Je nach Betriebssystem darf es dann auch schon mal mehr sein. Einige der Beispiele wurden auf einem PC entwickelt, der mit einem Pentium 90 Prozessor und 48 MByte RAM bestückt war. Als Betriebssytem war auf diesem Rechner Win95 installiert.

Auf der Software Seite wird zunächst ein Betriebsystem benötigt, das echtes Multitasking unterstützt. Daher scheidet z.B. Win 3.11 als Basissystem aus. Neben einem Compiler benötigt man weiterhin eine JAVA Laufzeitumgebung (JRE), die man am besten aus dem WWW [1] beziehen kann. Zur Zeit wird noch das alte JRE1.1.x unterstützt, da dieses deutlich kleiner und schneller als das neue JRE1.2 ist. Da sich auch der Aufruf der virtuellen Maschine geändert hat, sollte in jedem Fall dem alten JRE der Vorzug gegeben werden.

An Compilern kann der Entwickler auf eine große Zahl von freien Compilern zugreifen. Unter Linux ist die Wahl eines Compiler eigentlich kein Thema. Nur bei dem Pascal Compiler gibt es zum GNU Pascal eine echte Alternative, den Free Pascal Compiler *ppc386*, der den Distributionen beiliegen sollte

Für die Wintel Welt gibt es ebenfalls diverse freie Compiler. Für die C Proogrammierer gibt es z.B. einen Compiler *LCC*, der eine komplette Entwicklungsumgebung enthält Auch der neue Borland C Compiler wird von Inprise umsonst im Netz angeboten [2].

Für die Pascal Programmierer gibt es zu Turbo Pascal und Delphi mit dem bereits genannten Free Pascal Compiler eine Alternative. Da dieser ja auch unter Linux (und vielen

[1] http://www.javasoft.com
[2] http://www.inprise.com/downloads

anderen Betriebsystemen) läuft, ist hiermit eine plattformunabhängige Entwicklung besonders einfach.

Auch für die Fortran Programmierer gibt es freie Alternativen. Dem LCC Compiler ist ein Fortran2C Umsetzer beigefügt, sodaß unter diesem Compiler auch in Fortran entwickelt werden kann. Ansonsten ist eine Anbindung an den Lahey Fortran Compiler realisiert, der allerdings richtig teuer ist.

Natürlich ist auch die GNU Familie für WIN32 erhältlich. Durch den Einsatz der Firma CYGNUS (heute Red Hat), wurde ein C Compiler fur WIN32 geschaffen, der auch die Socket API enthält . Basierend auf diesem Compiler gibt es inzwischen auch die Portierungen von GPC und G77.

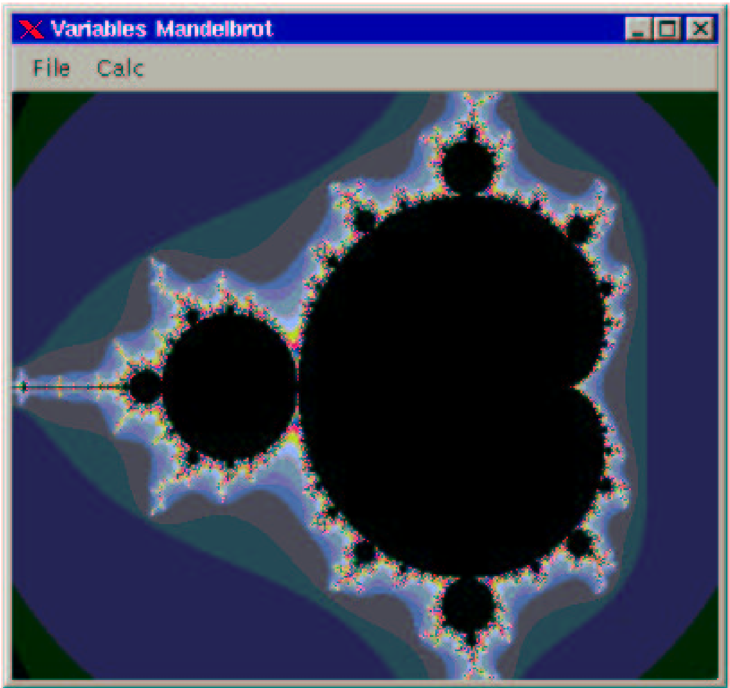


Abbildung 1.1: Ein Beipielprogramm, das Geschmack auf mehr wecken soll.

An dieser Stelle noch ein kurzes Wort zur Plattformunabhängigkeit. Natürlich ist ein Programm, das unter C, Fortran oder Pascal geschrieben wurde, nicht Plattformunabhängig im Sinne von JAVA. Das Konzept *write once, run everywhere* kann nur dann funktionieren, wenn ein interpretativer Anteil vorhanden ist. Unter Java wird der Sourcecode zunächst in einen Zwischencode übersetzt, der dann auf der jeweiligen Plattformen interpretiert wird[3]. Dies hat natürlich zur Folge, daß die Programme wesentlich langsamer ablaufen, als unter einer richtigen Compilersprache. Doch dazu später mehr.

Plattformunabhängigkeit ist hier im Sinne von *write once, compile everywhere* gemeint. Damit ist gemeint, daß es mit JAPI möglich ist, auf einer Plattform (z.B. Linux) eine Applikation zu entwickeln, die dann auf anderen Plattformen durch einen einfachen Com-

[3] Ein übrigens uraltes Konzept, das es bereits unter USD Pascal realisiert wurde.

pileraufruf übersetzt werden können. Dabei kann es allerdings zu Problemen kommen, die durch unterschiedliche Compiler Dialekte entstehen. Einige Funktionen heißen auf anderen Compilern einfach anders. Man findet jedoch eigentlich unter allen Compilern Funktionen, die sich etwa entsprechen. Unter C läßt sich dieses Problem mit geeigneten *#define* Anweisungen schnell in den Griff bekommen. Die strenge Normung in Fortran läßt hingegen kaum Probleme erwarten. Nur unter Pascal kann es durch die extremen Dialektunterschiede zu Problemen kommen. Insbesondere die unterschiedlichen Stringformate bereiten Schwierigkeiten. Lösungsmöglichkeiten sind dabei entweder ebenfalls bedingte Compileranweisungen, oder man beschränkt sich auf Compiler, die auf vielen Plattformen zuhause sind (z.B. der Free Pascal Compiler). Es sind jedoch unter keiner Sprache Probleme bekannt, die sich nicht mit etwas Gehirnschmalz lösen liesen.

Abschließend soll in dieser Einleitung Geschmack auf mehr geweckt werden. Im folgenden wird gezeigt, daß es wirklich möglich ist, mit wenigen Anweisungen eine funktionale und ansprechende Oberfläche aufzubauen. Als Beispiel wurde eine Applikation ausgewählt, auf die in diesem Handbuch immer wieder zurückgegriffen wird, um weitere Funktionen vorzustellen. Es handelt sich um die Mandelbrotmenge. Die Abbildung 1.1 zeigt die Oberfläche dieser Applikation.

Unter der Menuleiste verbergen sich hinter den Menupunkte *File* und *Calc* weiter Untermenüpunkte, die zum Starten und Stoppen der Berechnung, sowie zum Beenden des Programms dienen. Das gesamte Source Listing dieser Applikation ist im folgenden abgedruckt:

```
rem Example mandel.bas

xstart = -1.8
xend   =  0.8
ystart = -1.0
yend   =  1.0

hoehe  = 240
breite = 320

if(j_start() = J_FALSE) then
    print("Fehler beim Starten")
    goto ende
endif

jframe  = j_frame("")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
calc    = j_menu(menubar,"Calc")
quit    = j_menuitem(file,"Quit")
start   = j_menuitem(calc,"Start")
stop    = j_menuitem(calc,"Stop")

canvas  = j_canvas(jframe,breite,hoehe)
j_setpos(canvas,10,60)
j_pack(jframe)
j_show(jframe)

obj = 0
do_work = 0

while((obj <> jframe) and (obj <> quit))

    if(do_work = 1) then
        obj = j_getaction()
    else
        obj = j_nextaction()
    endif
```

```
    if(obj = start) then
        x = -1
        y = -1
        do_work = 1
    endif

    if(obj = stop) do_work = 0

    if(do_work = 1) then
        x = mod(x+1,breite)
        if(x = 0) y = mod(y+1,hoehe)
        if((x = breite-1) and (y = hoehe-1)) then
            do_work = 0
        else
            zre = xstart + x*(xend-xstart)/breite
            zim = ystart + y*(yend-ystart)/hoehe
            it = mandel(zre,zim,512)
            j_setcolor(canvas,it*11,it*13,it*17)
            j_drawpixel(canvas,x,y)
        endif
    endif

wend

label ende
j_quit()



sub mandel(zre,zim,maxiter)
    mx = 0.0
    my = 0.0
    iter=0
    betrag=0.0

    while ((iter < maxiter) and (betrag < 4.0))
        iter = iter+1
        tmp = mx*mx-my*my+zre
        my = 2*mx*my+zim
        mx = tmp
        betrag = (mx*mx + my*my)
    wend
    return iter
end sub
```

Der gesamte Code passt auf etwa 2 DIN A4 Seiten. Dabei entfällt gut die Hälfte des abgedruckten Listings auf die Mandelbrotiteration sowie die Variablendeklaration und deren Initialisierung. Für die eigentliche Oberfläche und die Benutzerinteraktion sind nur weniger als 50 Programmzeilen notwendig.

Sie sollen und können dieses Programm hier noch nicht verstehen. Sie sollen lediglich einen Eindruck gewinnen, wie schön und einfach eine solche Programmierung sein kann. Schon ab Seite 23 werden Sie in der Lage sein, solche und wesentlich umfangreichere Applikation selbst zu erstellen.

In den folgenden Kapiteln werden alle graphischen Elemente vorgestellt, zusammen mit einigen Funktionen, die weitere Einstellungen und Funktionalitäten zur Verfügung stellen. Die geschieht mit vielen Beispielen, die im jeweiligen Sourcecode mit abgedruckt sind. Es wird empfohlen die Listings genau zu studieren, da ein Beipiel oft viel mehr zeigen kann, als eine trockene Funktionsbeschreibung. Alle Beispiele sind im Sourcecode vorhanden, sodaß eigenen Entwicklungen leicht durch Weiterentwicklung geeigneter Beispiele aufgebaut werden können.

Der Refence Teil dieses Handbuchs bietet hingegen eine Nachschlagewerk, in dem ermittelt werden kann, welche Funktionen ein graphisches Element besitzt. Weiterhin sind

dort nochmals alle JAPI Funktionen mit einer ausführlichen Beschreibung in alphabetischer Reihenfolge abgedruckt.

Viel Spass.

Merten Joost

Kapitel 2

Die Komponenten

Die Einführung in die Programmierung von JAPI erfolgt Schritt für Schritt. Zunächst wollen wir uns mit den elementaren Funktionen einer graphischen Benutzeroberfläche vertraut machen.

2.1 Frame

Eine graphische Benutzeroberfläche besteht in erster Linie aus einem Fenster. Das folgende Beispiel zeigt, wie ein einfaches Fenster erzeugt werden kann:

```
rem Example simple.bas

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
end if
f = j_frame("Hello World")
j_show(f)

input a$
j_quit()

run:  yabasic simple.bas
```

Wir wollen nun Schritt für Schritt die benutzten Funktionen besprechen. Um eine graphische Ausgabe zu erzeugen, muss zunächst der JAPI Graphik Kernel gestartet werden:

```
j_start()
```

Die Funktion liefert **J_TRUE** zurück, sofern der Graphik Kernel korrekt gestartet wurde. Ist ein Fehler aufgetreten, so wird **J_FALSE** zurückgeliefert.

Die nächste Funktion:

```
j_frame("Ueberschrift")
```

erzeugt ein Hauptfenster (Frame) für die Applikation. Dieses Graphik Element *Frame* erwartet einen Parameter. Dies ist eine Überschrift, die im Rahmen der Applikation eingeblendet wird. Wird ein leerer String übergeben, so wird eine Standartüberschrift verwendet, wie sie auch in Abbildung 2.1 zu sehen ist.

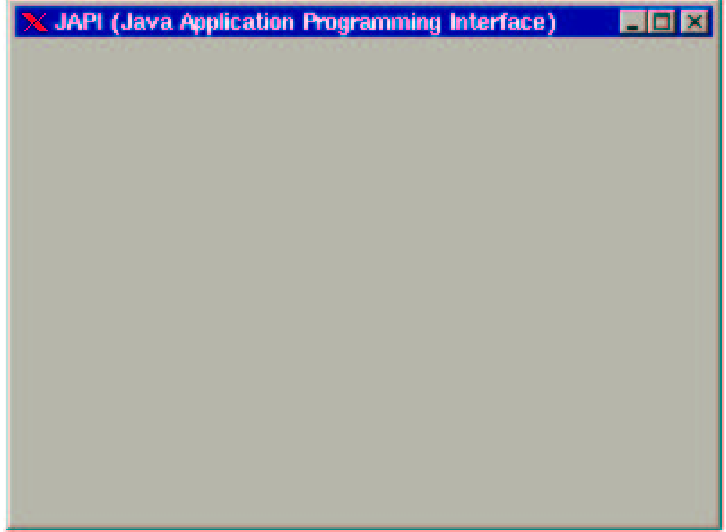


Abbildung 2.1: Ein einfaches Fenter unter JAPI.

Ein Frame besitzt zunächst eine Größe von 400 mal 300 Pixeln. Wie diese Größe verändert werden kann, werden wir später noch kennenlernen.

Die Funktion ***j_frame()*** erzeugt zwar einen Frame, stellt diesen aber noch nicht am Bildschirm dar. Erst nach der folgenden Funktion wird der Frame sichtbar:

```
j_show(jframe)
```

Das Beispiel wartet nun in der Konsole auf die Eingabe eines Zeichens, um die Applikation zu beenden. Dann wird mit der Funktion:

```
j_quit()
```

die Applikation veranlaßt sich vom JAPI Kernel zu trennen. ***j_quit()*** ist somit die letzte Anweisung, die der JAPI Sever entgegennimmt. Weitere japi Funktionen können dann nicht mehr ausgeführt werden.

Das Beispiel erzeugt Fenster, das eine standardisierte Breite und Höhe besitzt, siehe Abbildung 2.1. Es gestattet jedoch noch keine Interaktion mit dem Benutzer. Wir wollen das Beipiel nun so erweitern, daß der Benutzer mit der Maus das Programm beenden kann.

Unter jeder Oberfläche existiert im Fenterrahmen ein Icon, mit dem ein Program beendet werden kann. In der Regel befindet sich dieses Icon in der rechten oberen Ecke, und besizt ein 'X' als Symbol.

Im Beispiel wurde ein Frame erzeugt, und in einer Varaiblen mit dem Namen 'frame' gespeichert. Der gespeicherte Wert repräsentiert einen Index, über den auf das eigentliche graphische Element zugriffen werden kann. So erwarten z.B. auch die Funktion ***j_show()*** einen solchen Indexwert, der ihr zeigt, welches Element anzuzeigen ist. Über diese Indexwerte ist aber noch mehr möglich. Erfolgt eine Benutzeraktion, so wird über diesen Index angezeigt, in welchem graphischen Element die Aktion geschehen ist. Deshalb werden diese Indexwerte auch Eventnummern der Elemente genannt. Die Eventnummer eines Frames ist nun genau

an das Close Icon des Fenterrahmen gebunden. Wird nun also vom Benutzer dieses Icon angeklickt, so wird die Applikation mit der Eventnummer des Frames benachrichtigt, die dann entsprechen reagieren kann. Eine geeignete Reaktion wäre in unserem Beispiel das sofortige Ende der Applikation.

Die Frage ist nun, wie bekommt eine Applikation die Eventnachricht. Unter JAPI gibt es eine Funktion, mit der überprüft werden kann, ob ein Event vorliegt. Die Funktion:

```
j_nextaction()
```

liefert die entsprechende Eventnummer des Objektes in dem der Benutzer eine Aktion durchführte. ***j_nextaction()*** ist blockierend, d.h. die Applikation schlummert in dieser Funktion solange, bis der Benutzer eine Aktion durchführt. Wir werden später auch noch eine nichtblockierende Funktion kennenlernen. Nun können wir unser erstes Beispiel so erweitern, daß der Benutzer die Applikation durch Anklicken des Close Icons im Fensterrahmen beenden kann.

```
rem Example frame.bas

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
end if

jframe = j_frame("Hello World")
j_show(jframe)

while(j_nextaction() <> jframe)
wend

j_quit()

run:  yabasic frame.bas
```

Neu ist die Endlosschleife:

```
while(j_nextaction() <> jframe)
wend
```

in der die Applikation solange verweilt, solage der Japi Server nicht die Identifikationsnummer des Frames zurückliefert. Die Nummer des Frames wird genau dann geliefert, wenn der Benutzer den entsprechenden Close Button des Fensters anklickt. Mit anderen Worten ist die Eventnummer des Frames eindeutig an die Aktion "Schließen des Framewindows" gebunden.

Die Endlosschleife ist ein weiteres elementares Element jeder JAPI Applikation. Diese Schleife findet sich in allen Programmen wieder, die auf Benutzeraktion reagieren. Sie wird auch als "Event Loop" bezeichnet. Wie bereits erwähnt, liefern viele der graphischen Elemente ein Event, wenn der Benutzer eine entsprechende Aktion ausführt. Bei einem Frame ist diese Aktion nun eben das Anklicken des Close Button, der sich im Fenterrahmen befindet.

2.2 Menu

Hauptbestandteil fast aller graphischen Benutzeroberflächen sind die Menüs, die sich in der Regel am oberen Fenterrand befinden. Dieses Kapitel soll einen Überblick schaffen, wie unter

unter Japi solche Menues erstellt und dynamisch verändert werden können. Voraussetzung für die Menüs ist eine Menüleiste, in die die Menüs eingebettet werden koennen. Eine Menuleiste ist mit der Funktion

```
j_menubar(jframe)
```

in einen Frame zu integrieren. Die Funktion liefert einen Identifier zurueck, der für die einzelnen Menupunkte benötigt wird. In diese Menuleiste können nun mit der Funktion

```
j_menu( menubar, title$ )
```

einzelne Menupunkte eingebaut werden. An jeden Menupunkt werden mit der Funktion

```
j_menuitem( menu, titel$ )
```

Untermenüpunkte angehängt, die bei anklicken eines Menupunktes sichtbar werden.

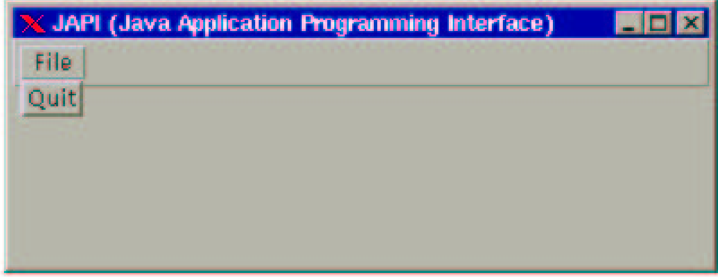


Abbildung 2.2: Ein einfaches Menü

Mit diesen Funktionen läß sich bereit eine funktionale Menüleiste realisieren. Das folgende Beispiel beschränkt sich zunächst auf einen Menüeintrag mit einem Menuitem:

```
rem Example simplemenu.bas

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
end if

jframe  = j_frame("")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
quit    = j_menuitem(file,"Quit")

j_show(jframe)

obj = 0
while((obj <> jframe) and (obj <> quit))

    obj = j_nextaction()

wend
j_quit()
```

Die zeigt die Abbildung 2.2 zeigt die programmierte Oberfläche. An die Menüleiste wurde ein Menü angehängt, das die Beschriftung "File" trägt. Wird dieser Menüpunkt angeklickt, so klappt eine Menüleiste auf, die einen weiteren Menüpunkt enthält, der die Beschriftung "Quit" trägt. Wird dieser Menüpunkt angewählt, so liefert der Japiserver die Identifikationsnummer dieses Menüpunktes zurück. In der modifizierten Endlosschleife

```
while((obj <> jframe) and (obj <> quit))

    obj = j_nextaction()

wend
```

wird nun diese ID ebenfalls berücksichtigt. Sie führt im Beispiel ebenfalls zum Abbruch des Programms, was nicht weiter verwundert.

Die eben vorgestellten Funktionen sind das Grundgerüst aller Menüleisten. Neben diesen Funktionen existieren jedoch noch eine Vielzahl weiterer Funktionen, die weitere Menüpunkte und Modifikationen von Menüpunkten erlauben. Das nächste (schon ausführlichere) Beipielprogramm zeigt alle vorhandenen Menüarten, und einige der Manipulationsmöglichkeiten

```
rem Example menu.bas

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
endif

jframe  = j_frame("")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
edit    = j_menu(menubar,"Edit")
options = j_menu(menubar,"Options")
help    = j_helpmenu(menubar,"Help")
submenu = j_menu(options,"Settings")

jopen   = j_menuitem(file,"Open")
save    = j_menuitem(file,"Save")
j_seperator(file)
quit    = j_menuitem(file,"Quit")
j_disable(save)
j_setshortcut(quit,"q")

cut     = j_menuitem(edit,"Cut")
copy    = j_menuitem(edit,"Copy")
paste   = j_menuitem(edit,"Paste")

about   = j_menuitem(help,"About")

enable  = j_checkmenuitem(submenu,"Enable Settings")
settings= j_menuitem(submenu,"Settings")
j_disable(settings)

j_show(jframe)

obj = 0
while((obj <> jframe) and (obj <> quit))

    obj = j_nextaction()

    if(obj = enable) then
        if(j_getstate(enable) = J_TRUE) then
            j_enable(settings)
        else
            j_disable(settings)
        endif
    endif

    if(obj = cut) then
        inhalt$ = j_gettext(cut,inhalt$)
        if(inhalt$ = "Cut") then
            j_settext(cut,"Ausschneiden")
        else
            j_settext(cut,"Cut")
        endif
```

```
        endif

        if(obj = copy) then
            inhalt$ = j_gettext(copy,inhalt$)
            if(inhalt$ = "Copy") then
                j_settext(copy,"Kopieren")
            else
                j_settext(copy,"Copy")
            endif
        endif

        if(obj = paste) then
            inhalt$ = j_gettext(paste,inhalt$)
            if(inhalt$ = "Paste") then
                j_settext(paste,"Einfuegen")
            else
                j_settext(paste,"Paste")
            endif
        endif

    wend

    j_quit()
```

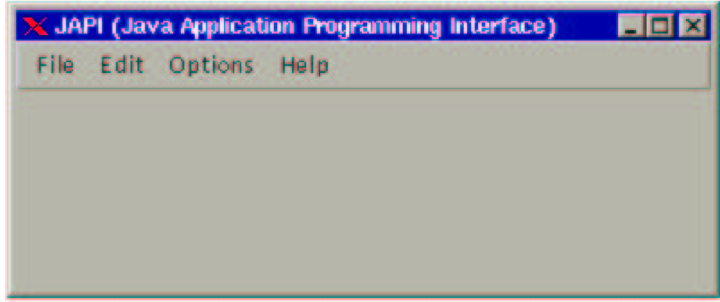


Abbildung 2.3: Alle Menüs

Keine Bange, auch dieses Listing ist mit einigen Erläuterungen leicht zu verstehen. Zunächst einmal zeigt die Abbildung diese Oberfläche mit den Obermenüpunkten. Die ersten Zeilen des Programms sind schon bekannt. Neben der Menübar werden zunächst 3 Menüpunkte an diese Menübar angehängt.

```
menubar  = j_menubar(jframe)
file     = j_menu(menubar,"File")
edit     = j_menu(menubar,"Edit")
options  = j_menu(menubar,"Options")
```

Sie erscheinen in der Reihenfolge, in der sie erzeugt werden von links nach rechts. Mit dem nächsten Befehl

```
submenu  = j_menu(options,"Settings")
```

wird nun ein Menüpunkt nicht an die Menüleiste, sondern an einen Menüeintrag angehängt. Der Menüpunkt "Settings" erscheint somit nicht in der Menüleiste, sondern als Untermenüpunkt innerhalb der Menüleiste "Options".

Mit dem Befehl ***j_helpmenu()*** wird ein Hilfe–Menüpunkt in die Menübar eingefügt, der je nach Oberfläche auch rechtbündig erscheinen kann. Funktionell gibt es keinen Unterschied

zwischen ***j_menu()*** und ***j_helpmenu()***, außer daß ein Hilfemenü kein Untermenüpunkt sein kann. Er muß daher immer an eine Menüleiste gebunden werden (siehe Abbildung 2.3).

```
help     = j_helpmenu(menubar,"Help")
```

Der nächste neue Befehl ist ***j_sepearator()***. Dieser Befehl erzeugt eine Trennlinie innerhalb einer Menüauswahl. Mit diesen Trennlinien lassen sich die Menüpunkte übersichtlicher anordnen (Siehe Abbildung 2.4).

```
j_seperator(file)
```

Mit der Funktion ***j_disable(save)*** wird der Menüpunkt "Save" disabled, sodaß dieser nun nicht mehr angewählt werden kann. Die Schrift dieses Menüpunktes erscheint in hellgrauer Schrift. Mit der Funktion ***j_enable()*** können solche Menüeinträge wieder aktiviert werden. Auf diese Weise lassen sich Menüeintrage temporär als ungültig markieren.

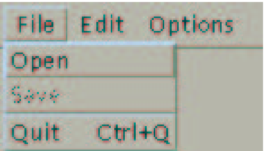


Abbildung 2.4: Der erste Menüpunkt mit Seperator, disableten Menüeintrag und Shortcut

Die nächste Zeile enthält eine Funktion, mir der einzelnen Menüeinträgen sogenannte Shortcuts zugewiesen werden können:

```
japi_setshortcut(quit,"q")
```

Ein Shortcut ist eine Tastaturkombination, die dasselbe bewirkt, wie ein Anklicken den Menüeintrages mit der Maus. Die Kombination besteht immer aus einer Modifier Taste und dem übergebenen Zeichen. Die Modifiertaste ist wiederum vom System abhängig. Unter Unix und Windows ist es die 'Control' (bzw. 'Strg') Taste, unter Macintosh heißt diese Taste 'Command'. Die angegebene Programmzeile bewirkt also, daß das Programm nun zusätzlich durch gleichzeitiges Drücken von Modifiertaste und der Taste 'Q' beendet werden kann.

Die Menüpunkte "cut", "copy" und "paste" werden bei Auswahl ins Englische und bei erneuter Auswahl wieder ins Deutsche übersetzt. Dazu wird in der Eventloop mittels der Funktion ***j_gettext()*** der Inhalt der Menüs ermittelt, und entsprechend mit der Funktion ***j_settext()*** agiert:

```
if(obj = cut) then
    inhalt$ = j_gettext(cut,inhalt$)
    if(inhalt$ = "Cut") then
        j_settext(cut,"Ausschneiden")
    else
        j_settext(cut,"Cut")
    endif
endif
```

Die Funktion

```
enable   = j_checkmenuitem(submenu,"Enable Settings")
```

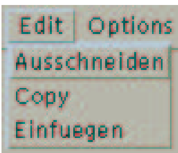


Abbildung 2.5: Der zweite Menüpunkt, bei dem die Menüitems die Sprache wechseln.

erzeugt einen Menüeintrag, der zusätzlich mit einer Checkbox ausgestattet ist. Klick man einen solchen Menüpunkt an, so wechselt der Status der Checkbox. Den Status kann man jederzeit mit der Funktion ***j_getstate()*** abfragen, und mit ***j_setstate()*** setzen. Der Defaultzustand nach den Erzeugen des Menüeintrages ist gesetzt.

Wird das Checkboxmenü "enable" vom Benutzter angewählt, so wird zunäscht mit der Funktion ***j_getstate()*** der Zustand des Menüs abgefragt:

```
if(obj = enable) then
    if(j_getstate(enable) = J_TRUE) then
        j_enable(settings)
    else
        j_disable(settings)
    endif
endif
```

Ist dieser gesetzt (Returnwert = **J_TRUE**), so wird der Menüeintrag "settings" enabled. Ist die Checkbox nicht gesetzt, wird dieser Menüpunkt disabled.

Abbildung 2.6: Der dritte Menüpunkt mit einem Untermenü und einem Checkboxmenü.

Wird der Menüpunkt "quit" angewählt, so wird die Eventloop verlassen.

```
while((obj <> jframe) and (obj <> quit))
   :
   :
wend
```

Das Beispielprogramm stellt die wichtigsten Manipulationsmöglichkeiten der Menüeinträge vor. Eine komplette Liste aller möglichen Funktionen ist im Reference Teil bei den jeweiligen Menüelementen beschrieben.

2.3 Canvas

Ein weiteres elementares Element einer graphischen Benutzeroberfläche ist ein Ausgabefeld, in dem beliebige graphische Aktionen durchgeführt werden können. Unter Japi ein Canvas (Leinwand) eine universelle Zeichenfläche zur Verfügung. Auf dieser Zeichenfläche können unterschiedliche graphische Primitiven ausgeführt werden. Japi stellt neben Punkten, Linien und Rechtecken, noch viele weitere Primitiven zur Verfügung. Eine Beschreibung aller Privitiven befinden sich im Kapitel Graphik.

Das folgende Beispielprogramm nutzt nur das Setzen von Farben und Punkten:

```
rem  Example canvas.bas

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
endif

jframe  = j_frame("")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
calc    = j_menu(menubar,"Calc")
quit    = j_menuitem(file,"Quit")
start   = j_menuitem(calc,"Start")
stop    = j_menuitem(calc,"Stop")

canvas  = j_canvas(jframe,256,256)
j_setpos(canvas,10,60)
j_setnamedcolorbg(canvas,J_YELLOW)

j_pack(jframe)
j_show(jframe)

obj = 0
do_work = 0

while((obj <> jframe) and (obj <> quit))

    if(do_work = 1) then
        obj = j_getaction()
    else
        obj = j_nextaction()
    endif

    if(obj = start) then
        do_work = 1
        j_setnamedcolorbg(canvas,J_YELLOW)
    endif

    if(obj = stop) do_work = 0

    if(do_work = 1) then
        j_setcolor(canvas,mod(j_random(),256),mod(j_random(),256),mod(j_random(),256))
        j_drawpixel(canvas,mod(j_random(),256),mod(j_random(),256))
    endif

wend

j_quit()
```

Zunächst wird ein kleines Menu erstellt, das die aktiven Menupunkte "Quit", "Start" und "Stop" enthält. Im nächsten Schritt wird ein Canvas erzeugt:

```
canvas  = j_canvas(jframe,256,256)
j_setpos(canvas,10,60)
j_setnamedcolorbg(canvas,J_YELLOW)
```

und dem Canvas eine Größe von 256x256 Bildpunkten zugeordnet, und an die Position (10,60) gesetzt. Bezugspunkte zur Positionierung sind jeweils die oberen linken Ecken, der beiden Elemente Frame und Canvas. Somit wird die obere linke Ecke des Canvas 10 Pixel rechts und 60 Pixel unterhalb der oberen linken Ecke des Frames angeordnet.

Da der Canvas zudem dieselbe Hintegrundfarbe besitzt wie der Frame, wäre der Canvas zunächst nicht zu erkennen. Deshalb wird dem Canvas eine andere Hintergrundfarbe zugeordnet:

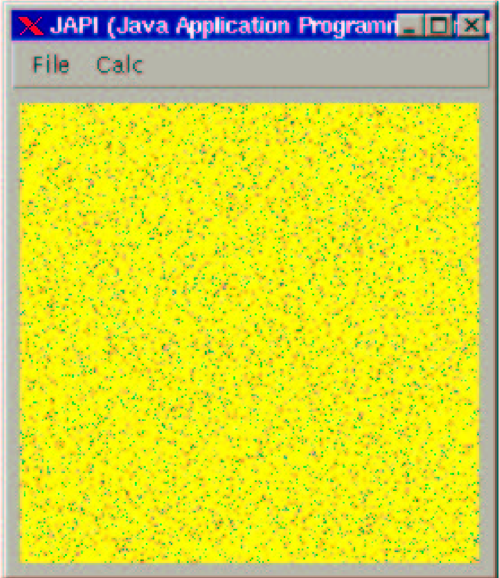


Abbildung 2.7: Ein Canvas

```
j_setnamedcolorbg(canvas,J_YELLOW)
```

Diese Funktion ist eine von vier Funktionen um Farben zu setzen:

```
j_setcolor( obj , red , green , blue )
j_setcolorbg( obj , red , green , blue )
j_setnamedcolor( obj , farbe )
j_setnamedcolorbg( obj , farbe )
```

Die beiden ersten Funktionen setzen die Farben für Vordergrund und Hintergrund durch RGB Werte. Jeder Kanal kann einen Wert von 0 bis 255 annehmen. Die nächsten beiden Funktionen setzen vordefinierte Farben. Die Variable "farbe" kann dabei einen Wert von 0 bis 15 annehmen. Diese 16 Farben entsprechen dem alten CGA Farbmodell und sind wie folgt zugeordnet:

```
J_BLACK             0
J_WHITE             1
J_RED               2
J_GREEN             3
J_BLUE              4
J_CYAN              5
J_MAGENTA           6
J_YELLOW            7
J_ORANGE            8
J_GREEN_YELLOW      9
J_GREEN_CYAN       10
J_BLUE_CYAN        11
J_BLUE_MAGENTA     12
J_RED_MAGENTA      13
J_DARK_GRAY        14
J_LIGHT_GRAY       15
```

Wird bei einem Canvas die Hintergrundfarbe gesetzt, so wird der gesamte Inhalt des Canvas geloescht. Somit können diese Funktionen auch zum Löschen des Fensterinhaltes verwendet werden. Unfangreiche Erläuterungen zu diesen Funktionen und den Farbmodellen finden sich im Kapitel 5.

Nachdem alle graphischen Objekte erzeugt wurden, wird der Frame angezeigt:

```
j_pack(jframe)
j_show(jframe)
```

Der Befehl ***j_pack()*** veranlaßt den Frame seine eigene Größe zu ermitteln. Dabei stellt dieser die Größe aller enthaltener Objekte und deren Position fest. Seine eigene Größe stellt nun die Vereinigungsmenge aller Objektedimensionen dar. Zusätzlich wird eine rechter und unterer Rand von 5 Pixeln hinzugegeben. Durch diese Funktion kann auf eine aufwendiges Berechnungsverfahren verzichtet werden. Alternativ zu ***j_pack()*** kann jedoch auch einem Frame mit der Funktion ***j_setsize()*** eine fixe Größe zugeordnet werden. Es macht keinen Sinn beide Funktionen zu verwenden. In diesem Falle würde immer die zuletzt verwendete Funktion gewinnen.

In der Event Loop findet sich zunächst folgende Abfrage:

```
if(do_work = 1) then
    obj=j_getaction()
else
    obj=j_nextaction()
endif
```

Dabei ist die Variable do_work ein Flag, das gesetzt wird, wenn der Benutzer den Menueintrag "start" angewählt hat. Ist das Flag gesetzt, so wird statt ***j_nextaction()*** die Funktion ***j_getaction()*** aufgerufen. Der Unterschied der beiden Funktionen ist, daß ***j_nextaction()*** solange blockiert, bis eine Aktion vom Benutzer eintritt. ***j_getaction()*** hingegen kehrt sofort zurück, und liefert entweder die Identifikationsnummer eines Objektes, oder wenn keine Aktion vorlag, liefert die Funktion -1 zurück. Prinzipiell kann man in einer Event Loop stets die nicht blockierende Funktion ***j_getaction()*** verwenden. Diese führt jedoch zu einem Busy Waiting in der Application, und verschwendet unnötigerweise Recourcen. ***j_getaction()*** solte daher nur dann benutzt werden, wenn die Application in einer Berechnung steckt, und dennoch auf Benutzereingaben reagieren soll. In unserem Beispiel soll immer auf die Befehle "Quit" und "Stop" reagiert werden.

Wird vom Benutzer der Menupunkt "Start" angewählt

```
if(obj = start) then
    do_work=1
    j_setnamedcolorbg(canvas,J_YELLOW)
endif
```

so wird zunächst das Flag do_work gesetzt. Die folgende Anweisung setzt die Hintergrundfarbe neu. Dadurch wird der Inhalt des Canvas gelöscht.

Die eigentliche Zeichenarbeit wird von folgenden Zeilen erledigt:

```
if(do_work = 1) then
    j_setcolor(canvas,mod(rand(),256),mod(rand(),256),mod(rand(),256))
    j_drawpixel(canvas,mod(rand(),256),mod(rand(),256))
endif
```

Dabei werden zunächst zufällig verteilte Rot-, Grün- und Blauanteile ermittelt, und als entsprechende Vordergrundfarbe gesetzt. mit dieser Farbe wird nun ein Pixel im Canvas gezeichnet. Die X- und Y-Koordinate des Punktes werden dabei ebenfalls wieder zufällig bestimmt. Der Canvas füllt sich langsam mit bunten Punkten.

Während sich der Canvas langsam mit bunten Punkten füllt, wird immer noch mit der Funktion ***j_getaction()*** auf Benutzereingaben reagiert. Ein Anklicken der Menüeintrags "Stop" führt somit zur Beendigung der Zeichenaktivität. Ebenso hat des Anklicken des Menüpunktes "Quit" das Beenden des Programms zur Folge.

An dieser Stelle soll nochmal kurz auf das eingehende Beispiel mit der Mandelbrotmenge zurückgekehrt werden. Obiges Beispielprogramm und das Mandelbrot Beispiel unterscheiden sich nur in der Berechnung der Farbe und Position eines Punktes. Während im jetzigen Beispiel die Punkte und Farben zufällig gewählt werden, werden sie sie dort durch die Mandelbrot Iteration festgelegt.

Auch ein Canvas kann einen Event zurückliefern. Dieser Event ist an die Größe des Canvas gebunden. Wird die Größe des Canvas verändert, so wird die Applikation über die Canvas ID benachrichtigt, und kann entsprechend reagieren. Wir wollen an dieser Stelle das Eingangsbeispiel mit der Mandelbrotmenge aufgreifen, und eine Version erstellen, die es dem Benutzer erlaubt, die Fentergröße zu verändern. Dazu sind nur wenige Änderungen nötig:

```
rem  Example mandel1.bas

xstart = -1.8
xend   =  0.8
ystart = -1.0
yend   =  1.0

hoehe  = 240
breite = 320

if(j_start() = J_FALSE) then
    print("can't connect to JAPI server")
    exit
endif

jframe  = j_frame("Variables Mandelbrot")
j_setborderlayout(jframe)

menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
calc    = j_menu(menubar,"Calc")
quit    = j_menuitem(file,"Quit")
start   = j_menuitem(calc,"Start")
stop    = j_menuitem(calc,"Stop")

canvas  = j_canvas(jframe,breite,hoehe)

j_pack(jframe)
j_show(jframe)

obj = 0
do_work = 0

while((obj <> jframe) and (obj <> quit))

    if(do_work = 1) then
        obj = j_getaction()
    else
        obj = j_nextaction()
    endif

    if(obj = start) then
        x = -1
```

```
        y = -1
        do_work = 1
    endif

    if(obj = stop) do_work = 0

    if(do_work = 1) then
        x = mod(x+1,breite)
        if(x = 0) y = mod(y+1,hoehe)
        if((x = breite-1) and (y = hoehe-1)) then
            do_work = 0
        else
            zre = xstart + x*(xend-xstart)/breite
            zim = ystart + y*(yend-ystart)/hoehe
            it = mandel(zre,zim,512)
            j_setcolor(canvas,it*11,it*13,it*17)
            j_drawpixel(canvas,x,y)
        endif
    endif

    if(obj = canvas) then
        breite = j_getwidth(canvas)
        hoehe = j_getheight(canvas)
        x=-1
        y=-1
    endif
wend
j_quit()
```

Eine wichtige neue Funktion ist

```
j_setborderlayout(jframe)
```

Diese Funktion binden den Canvas so an den Frame, daß bei einer Größenänderung des Frames automatisch der Canvas mit vergrößert wird. Daraufhin erzeugt der Canvas einen Event, der in der Mainloop abgefragt werden kann:

```
if(obj = canvas) then
    breite = j_getwidth(canvas)
    hoehe = j_getheight(canvas)
    x=-1
    y=-1
endif
```

Das Beispielprogramm ermittelt nun die neue Hoehe und Breite des Canvas, und setzt die beiden Laufvariablen so, daß das gesamte Bild neu berechnet wird. Das wars.

Die Abbildung 2.8 und 2.9 zeigt dieses Beispielprogramm mit zwei unterschiedlichen Größen.

Nach diesem Kapitel sollte bereits jeder Programmierer in der Lage sein einfache Programme zu entwerfen und zu programieren. Die Elemente Frame, Canvas und Menüs erlauben bereits eine Vielzahl graphischer Applikationen.

2.4 Button

Ein Button stellt eine Schaltfläche zur Verfügung, die mit der Maus betätigt werden kann. Auf den meisten Systemen wirkt ein Button als hervortretende Fläche, die mit einem Mausklick versenkt werden kann. Ein Button ist kein Schalter, der an– und ausgeschaltet werden kann. Er ist vielmehr als eine Art Taster anzusehen, der nach Lösen der Maustaste wieder sein ursprüngliches Aussehen annimmt. So erzeugt ein Button noch keinen Event,

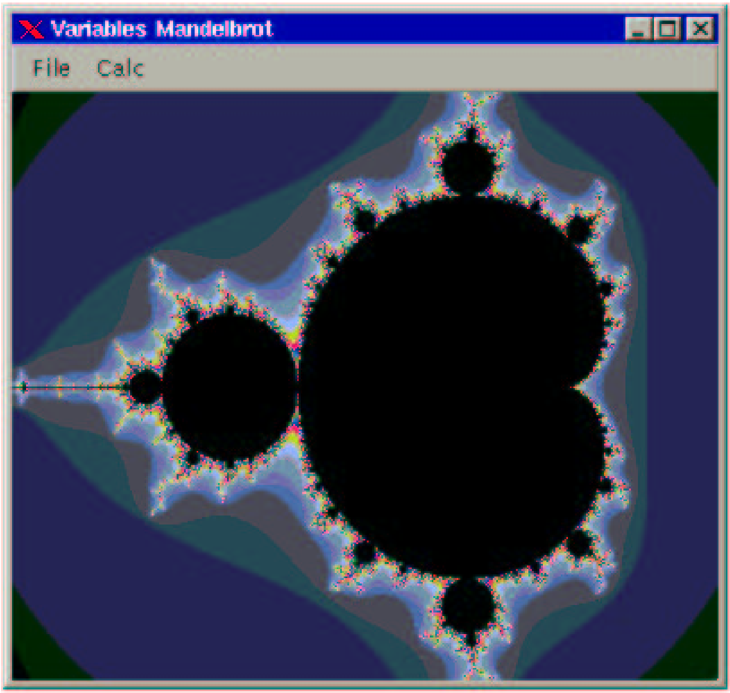


Abbildung 2.8: Das Mandelbrot Programm in seiner initialen Größe.

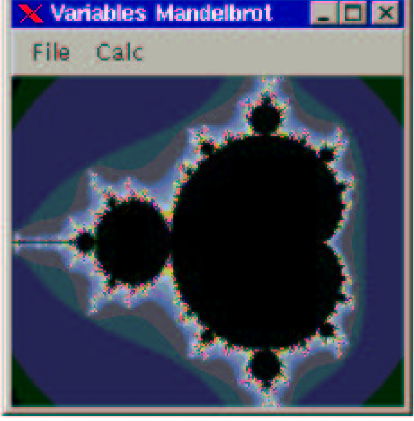


Abbildung 2.9: Das Mandelbrot Programm nach einer Größenänderung und Neuberechnung.

wenn die Maustaste gedrückt wird. Erst das Lösen der Maustaste bewirkt eine entsprechende Nachricht an die Applikation.

Das folgende Beispiel erzeugt einen wachsenden und schrumpfenden Button:

```
rem Example button.bas

if(j_start() = J_FALSE) then
    print "can't connect to JAPI server"
    exit
endif

jframe  = j_frame("")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
quit    = j_menuitem(file,"Quit")


button  = j_button(jframe,"increase")
j_setsize(button,80,20)
j_setpos(button,88,138)

j_setsize(jframe,256,256)
j_show(jframe)

big=0
obj=0
while((obj <> quit) and (obj <> jframe))

   obj=j_nextaction()

   if(obj = button) then
       if(big = 0) then
           for i=0 to 40
              j_setsize(button,80+i*2,20+i*2)
              j_setpos(button,88-i,138-i)
              j_sync()
           next i
           big=1
           j_settext(button,"shrink")
           j_settext(jframe,"Shrinking Button")
       else
           for i=40 to 0 step -1
              j_setsize(button,80+i*2,20+i*2)
              j_setpos(button,88-i,138-i)
              j_sync()
           next i
           big=0
           j_settext(button,"increase")
           j_settext(jframe,"Growing Button")
       endif
   endif
wend

j_quit()
```

Zunächst wird ein Button erzeugt:

```
button = j_button(jframe,"increase")
```

Ein Button benötigt als Parameter einen Titel, und ein Objekt, in dem er erscheinen soll. In diesem Beispiel wird der Button in dem Frame angezeigt. Der Titel bestimmt auch die Größe eines Buttons. Ein Button ist immer so groß, daß er seinen Titel darstellen kann. Daher ist ein in der Regel nicht nötig, einem Button eine spezielle Größe zuzuweisen. Im Beispiel wird dem Button dennoch eine initiale Größe von 80x20 Pixeln zugewiesen:

```
j_setsize(button,80,20)
```

Abbildung 2.10: Einen wachsender und schrumpfender Button.

Zudem wird der Button mit der Funktion:

```
j_setpos(button,88,138)
```

an einen festen Platz innerhalb des umgebenden Objektes zugewiesen. Bezugspunkt ist die linke obere Ecke des Buttons. Durch diese Anweisung soll der Button zentriert im Frame erscheinen. Dazu muß jedoch auch dem Frame eine Größe zugewiesen werden. Da der Button "wachsen" soll, muss der Frame groß genug sein, um auch den "großen" Button noch anzeigen zu können. Daher wird der Frame mit der folgenden Anweisung ausreichend dimensioniert:

```
j_setsize(jframe,256,256)
j_show(jframe)
```

In der Event Loop wird nun bei Anklicken des Buttons zunächst ein Flag 'big' abgefragt, das anzeigt, ob der Button z.Z. klein oder groß dargestellt ist. Hat er eine kleine Form, so wird in der 'for' Schleife:

```
for i=0 to 40
   j_setsize(button,80+i*2,20+i*2)
   j_setpos(button,88-i,138-i)
   j_sync()
next i
```

Größe und Position des Button so verändert, daß er zu wachsen scheint. Mit der anschließenden Anweisung:

```
j_sync()
```

veranlassen wir eine Syncronisierung der beiden Prozesse. Die Applikation wartet an dieser Stelle solange, bis der JAPI Server alle vorherigen Anweisungen vollzogen hat. Ohne diese Syncronisierung würde der Button schlagartig seine Größe ändern. Auf schnellen Systemen kann es zudem sinnvoll sein, eine weitere Bremse einzubauen. Dazu steht eine Funktion ***j_sleep(int msecs)*** zur Verfügung, bei der die Applikation für einige Millisekunden schlafen gelegt werden kann.

Ist die Schleife beendet, wird das Flag 'big' entsprechend umgesetzt. Anschließend werden mit den Anweisungen

```
j_settext(button,"shrink")
j_settext(jframe,"Shrinking Button")
```

dem Frame eine neue Überschrift gegeben. Weiterhin bekommt der Button einen neuen Titel. Diese Funktion zeigt, daß eine Funktion durchaus auf verschiedene graphische Elemente angewendet werden kann. Welche Funktion auf welche Elemente anwendbar ist, ist im Reference Teil dieser Anleitung in jeder Funktion beschrieben.

Der else Zweig der Flagabfrage bewirkt ein Schrumpfen des Buttons auf die ursprüngliche Größe und benutzt dieselben Funktionen wie zuvor.

Auch graphische Buttons sind unter der Japilib möglich. Die entsprechende JAPI Funktion lautet:

```
j_graphicbutton( obj , filename$ )
```

Im Unterschied zu einem normalen Button, erwartet der graphische Button keinen Titel, sondern einen Filenamen, der ein Bild enthält, das der Button anzeigen soll. Der Filenamen muss dabei auf ein File verweisen, daß eine Graphik Datei in GIF oder JPEG Format enthält.

Auch dieser Button ist in seiner Größe veränderbar. Er besitzt zwar zunächst eine Größe, die durch das Bild vorgegeben ist, kann jedoch jederzeit mit der nun bereits bekannten Funktion ***j_setsize()*** scaliert werden.

Das folgende Beispiel baut mit sechs Button einen Buttonbar auf, die unter den Windows Systemen weit verbreitet ist:

```
rem     Example graphicbutton.bas

:
if( j_start() = J_FALSE ) then
    print "can't connect to JAPI server"
    exit
endif

jframe   = j_frame("Graphic Buttons")
j_setflowlayout(jframe,J_HORIZONTAL)

gbutton = j_graphicbutton(jframe,"images/open.gif")
gbutton = j_graphicbutton(jframe,"images/new.gif")
gbutton = j_graphicbutton(jframe,"images/save.gif")
gbutton = j_graphicbutton(jframe,"images/cut.gif")
gbutton = j_graphicbutton(jframe,"images/copy.gif")
gbutton = j_graphicbutton(jframe,"images/paste.gif")

j_pack(jframe)
j_show(jframe)


while(j_nextaction() <> jframe)
wend
:
```

Auf die noch unbekannte Funktion ***j_flowlayout()*** soll hier nur sehr kurz eingegangen werden. Sie wird im Kapitel 3 ausführlicher erläutert. Sie sorgt in diesem Beispiel dafür, das die Buttons gleichmäßig nebeneinander angeordnet werden, ohne das sich der Programmierer um eine Positionierung kümmern muß.

Da keiner der Button in der Eventloop abgefragt wird, werden allen Elementern dieselbe Variable *gbutton* zugeordnet. Das Beispiel erzeugt eine Oberfläche, die in der Abbildung 2.11 zu sehen ist.

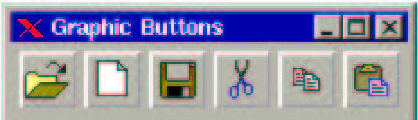


Abbildung 2.11: Eine Demo Applikation fuer Graphicbuttons.

2.5 Label

Ein Label stellt einen einzeiligen Text beliebiger Länge dar. Dieser ist nicht vom Benutzer editierbar, sondern kann nur von der Applikation verändert werden. Ein Label bietet daher keine Interaktionsmöglichkeit, sondern dient nur der Infornationsweitergabe. Folgendes Beispielprogramm läßt ein Label im Applikationsfenster wandern:

```
rem Example label.bas

if( j_start() = J_FALSE ) then
    print "can't connect to JAPI server"
    exit
endif

jframe   = j_frame("Moving Label")

menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
doit    = j_menuitem(file,"Start")
quit    = j_menuitem(file,"Quit")


jlabel   = j_label(jframe,"Hello World")
j_setpos(jlabel,10,120)

j_setsize(jframe,256,256)
j_show(jframe)

width   = j_getwidth(jlabel)
height  = j_getheight(jlabel)

dx=2
dy=1
run=0
obj=0
while((obj <> quit) and (obj <> jframe))

    if(run = 1) then
        obj=j_getaction()
    else
        obj=j_nextaction()
    endif

    if(obj = doit) then
        if(run = 0) then
            run=1
            j_settext(doit,"Stop")
        else
            run=0
            j_settext(doit,"Start")
        endif
    endif

    if(run = 1) then
        x = j_getxpos(jlabel)
        y = j_getypos(jlabel)
        if((x+width  > j_getwidth (jframe)) or (x < 1))  dx = -dx
```

```
        if((y+height > j_getheight(jframe)) or (y < 1))  dy = -dy
        j_setpos(jlabel,x+dx, y+dy)
        j_sync()
        j_sleep(10)
    endif

wend
j_quit()
```

Abbildung 2.12: Moving Label

Diese Programmzeilen:

```
jlabel    = j_label(jframe,"Hello World")
j_setpos(jlabel,10,120)
width  = j_getwidth(jlabel)
height = j_getheight(jlabel)
```

erzeugen ein Label mit der Inschrift "Hello World", setzen seine Position fest, und ermitteln die initiale Höhe und Breite, die durch den Text festgelegt werden. Ein Label verhält sich genauso wie ein Button. Wird dem Element keine spezielle Größe mit ***j_setsize()*** zugeordnet, so wird seine Größe durch den Schriftinhalt festgelegt. Diese Größe kann vom Japi Server jedoch erst dann ermittelt werden, wenn das Element bereits angezeigt ist, da auch der eingestellte Bildsschirmfont die Größe beeinflusst. Aus diesem Grund sollte der umgebende Frame bereits mit ***j_show()*** sichtbar gemacht worden sein, bevor der Aufruf ***j_getwidth()*** oder ***j_getheight()*** erfolgt. Wird eine Größe festgelegt, ist diese Einschränkung nicht zu beachten.

Die Eventloop enthält kaum neue Funktionen. In dem if–Statement:

```
    if(run = 1) then
        x = j_getxpos(jlabel)
        y = j_getypos(jlabel)
        if((x+width  > j_getwidth (jframe)) or (x < 1))  dx = -dx
        if((y+height > j_getheight(jframe)) or (y < 1))  dy = -dy
        j_setpos(jlabel,x+dx, y+dy)
        j_sync()
        j_sleep(10)
    endif
```

wird mit ***j_getpos(label,x,y)*** die aktuelle Position des Labels ermittelt. Die Parameter x bzw. y enthalten nach dem Aufruf die Koordinatenpunkte der linken oberen Ecke des Elements. In den beiden folgenden If Statements wird die Lage des Elements mit der Größe des Fensters verglichen. Erreicht das Label den Rand des Fensters so wird seine Bewegungsrichtungen invertiert. Durch einen erneuten Aufruf von ***j_setpos()*** wird das Label auf die neue Position verschoben. Bei der Lageberechnung werden jedesmal Hoehe und Breite des Frames ermittelt. Dadurch funktioniert die Animation auch dann, wenn das Fenster mit der Maus vergrößert oder verkleinert wurde. Allerdings ist diese permantente Größenermittlung sehr ineffizient. Daher ist die Animation auch sehr zäh in der Bewegung. Wir werden später auf dieses Beispiel noch zurückgreifen, um eine effizientere Implementierung vorzustellen.

Läßt man die Animation eine Weile laufen, so stellt man fest, daß das Label hinter der Menubar verschwindet. Da die Menubar in dem Frame liegt, ist dies eigentlich nicht weiter verwunderlich. Nun könnte man dies recht einfach abfangen, indem man annimmt, daß eine Menübar eine Höhe von ungefähr 50 Pixeln besitzt, und das Label eben umkehren muß, wenn es diese Grenze überschreitet. Problematisch gestaltet sich die Sache dann, wenn dieselbe Applikation auf verschiedenen Plattformen laufen soll. Dort haben Menubars unterschiedliche Höhe, sodaß man den Beginn des sichtbaren Bereich des Frames nicht eindeutig festlegen kann. Auch auf die Lösung dieses Problems wird später noch einmal eingegangen.

Will man mehrere Labels gleicher Größe untereinander darstellen, so stellt man fest, daß der Text innerhalb der Label mittig dargestellt wird. Dies läst sich mit der Anweisung:

```
j_setalign( label, alignment )
```

ändern. Zulässige Werte für alignment sind dabei die vordefinierten Konstanten J_LEFT für linksbündige Ausrichtung, J_CENTER für mittige Ausrichtung und J_RIGHT für rechtsbündige Ausrichtung.

Analog zu den Button gibt es auch grahische Labels. Die Funktion zum Erzeugen dieser Label ist mit denen der Button identisch:

```
j_graphicjlabel( obj , filename$ )
```

Auch hier wird statt einem String zur Labelbeschriftung ein String mit einem Filenamen übergeben, der das Bild enthält, das angezeigt werden soll. Das Bild muß im GIF oder JPEG Format vorliegen. Das folgende Beipiel erzeugt aus einem GIF Bild drei Labels unterschiedlicher Größe (siehe Abbildung 2.13):

```
rem     Example graphicjlabel.bas

if( j_start() = J_FALSE ) then
    print "can't connect to JAPI server"
    exit
endif

jframe   = j_frame("Graphic Label")

jlabel   = j_graphiclabel(jframe,"images/mandel.gif")
j_setpos(jlabel,10,30)
j_setsize(jlabel,50,50)

jlabel   = j_graphiclabel(jframe,"images/mandel.gif")
j_setpos(jlabel,70,30)
j_setsize(jlabel,150,240)

jlabel   = j_graphiclabel(jframe,"images/mandel.gif")
j_setpos(jlabel,230,30)
```

```
j_pack(jframe)
j_show(jframe)

while(j_nextaction()<>jframe)
wend
:
```

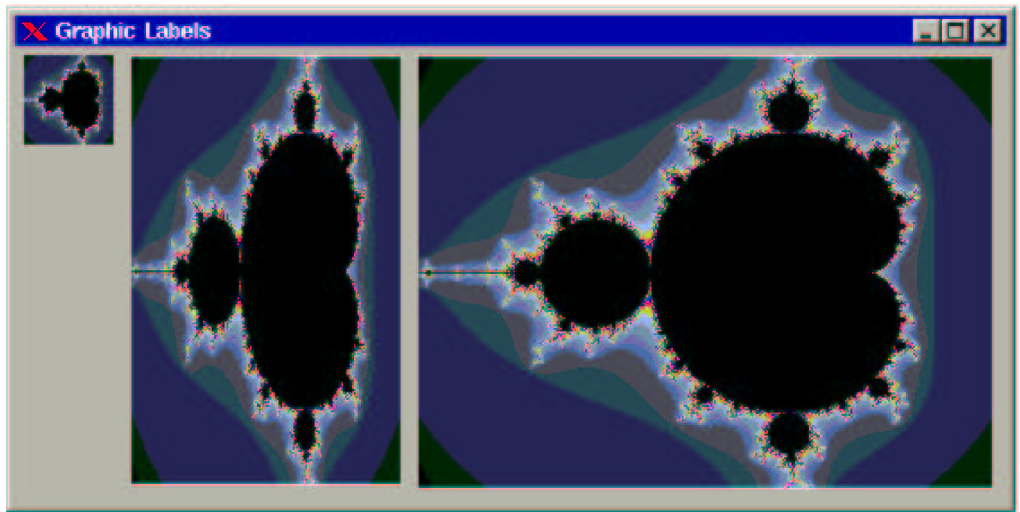


Abbildung 2.13: Demoprogramm zum Erzeugen graphischer Labels.

2.6 Checkbox

Abbildung 2.14: Beispiel für die Programmierung von Checkboxen.

Checkboxen sind graphische Elemente, die zwei Zustände einnehmen können. Sie sind entweder gesetzt oder nicht. Checkboxen bestehen intern aus zwei Elementen, einem Label, und einem kleinen graphischen Element, das den Zustand der Checkbox anzeigt. Unter Win32 wird dies durch ein kleines Feld angezeigt, das entweder leer ist (nicht gesetzt), oder ein Häkchen enthält (gesetzt). Unter dem Motiv GUI wird der Zustand durch kleine Button angezeigt, die entweder hervorgehoben dargestellt sind (nicht gesetzt) oder eingedrückt erscheinen (gesetzt). Das folgende Beispielprogramm erzeugt drei Checkboxen für die Grundfarben Rot, Grün und Blau. Sobald ein Zustand einer Checkbox geändert wird, wird die entstandene Mischfarbe ermittelt, und als Hintergrundfarbe für den Frame gesetzt:

```
rem  Example checkbox.bas

if( j_start() = J_FALSE ) then
    print "can't connect to JAPI server"
    exit
endif

jframe   = j_frame("switch the colors On/Off")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
quit    = j_menuitem(file,"Quit")

blue=j_checkbox(jframe,"Blue")
j_setpos(blue,150,80)
j_setstate(blue, J_TRUE )
b=255

red=j_checkbox(jframe,"Red")
j_setpos(red,150,120)
j_setstate(red, J_FALSE )
r=0

green=j_checkbox(jframe,"Green")
j_setpos(green,150,160)
j_setstate(green, J_FALSE )
g=0

j_setcolorbg(jframe,r,g,b)
j_show(jframe)

while((obj <> jframe) and (obj <> quit))

    obj=j_nextaction()

    if(obj = blue) then
        if(j_getstate(blue) = J_TRUE) then
            b=255
        else
            b=0
        endif
    endif

    if(obj = red) then
        if(j_getstate(red) = J_TRUE) then
            r=255
        else
            r=0
        endif
    endif

    if(obj = green) then
        if(j_getstate(green) = J_TRUE) then
            g=255
        else
            g=0
        endif
    endif
```

```
    j_setcolorbg(jframe,r,g,b)

    if(r+g+b < 256) then
        j_setcolor(jframe,255,255,255)
    else
        j_setcolor(jframe,0,0,0)
    endif

wend

j_quit()
```

Der Zustand einer Checkbox kann mit den Funktionen

```
j_setstate( checkbox )
j_getstate( checkbox )
```

gesetzt bzw. ermittelt werden. Sobald der Benutzer den Zustand einer Checkbox ändert, wird ein Event mit der Identifikationsnummer der entsprechenden Checkbox an die Applikation übermittelt. So wird beispielhaft für die Checkbox der Farbe Blau mit

```
    if(obj = blue) then
        if(j_getstate(blue) = J_TRUE) then
            b=255
        else
            b=0
        endif
    endif
```

der Farbanteil der Variablen b entweder auf 255 oder auf 0 gesetzt. Sind alle drei Farbanteile ermittelt, so kann mit der Funktion

```
j_setcolorbg(jframe,r,g,b)
```

der Hintergrund des Frames auf die entsprechenden rgb (rot, grün, blau) Werte gesetzt werden [1].

Im folgenden if–Statement

```
    if(r+g+b < 256) then
        j_setcolor(jframe,255,255,255)
    else
        j_setcolor(jframe,0,0,0)
    endif
```

wird die Summe der Farbanteile ermittelt, und die Vordergrundfarbe auf Weiss oder Schwarz gesetzt. Dadurch bleiben die Texte der Checkboxen lesbar, da bei dunklen Farben die Schriftfarbe auf Weiss und entsprechend bei hellen Hintergrundfarben die Schriftfarbe auf Schwarz gesetzt wird.

2.7 Radiobutton

Radiobutton unterscheiden sich kaum von den Checkboxen. Sie besitzen ebenfalls ein Label mit einem Textelement, und eine graphische Anzeige, die den Zustand des Buttons anzeigt. Radiobutton unterscheiden sich insofern von den Checkboxen, indem Radiobuttons einer

[1] Die Farbfunktionen sind im Kapitel 5 beschrieben

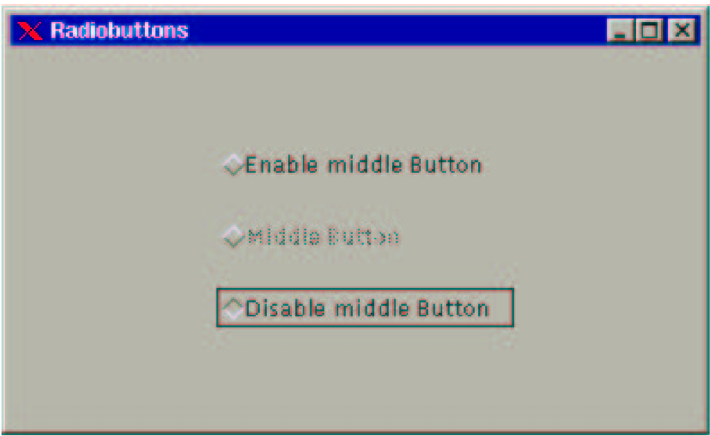


Abbildung 2.15: Beispiel für die Programmierung von Radiobuttons

Radiogroup zugeordnet werden müssen. Innerhalb einer Radiogroup ist immer nur ein Radiobutton aktiv. Daher auch der Name, der sich von den alten Röhrenradios herleitet, bei denen auch immer nur eine Taste gedrückt sein konnte. Das folgende Beispielprogramm erzeugt drei Buttons. Mit den beiden äußeren Button kann der Zugriff auf den inneren Button gesteuert werden:

```
rem     Example radiobutton.bas

if( j_start() = J_FALSE ) then
    print "can't connect to JAPI server"
    exit
endif

jframe   = j_frame("Radiobuttons")

radio   = j_radiogroup(jframe)

enable  = j_radiobutton(radio,"Enable middle Button")
j_setpos(enable,120,80)

middle  = j_radiobutton(radio,"Middle Button")
j_setpos(middle,120,120)

disable = j_radiobutton(radio,"Disable middle Button")
j_setpos(disable,120,160)

j_setstate(enable, J_TRUE )
j_show(jframe)

obj=0
while(obj <> jframe)

    obj=j_nextaction()

    if(obj = enable)  j_enable(middle)
    if(obj = disable) j_disable(middle)

wend
j_quit()
```

Neu sind die Anweisungen :

```
radio   = j_radiogroup(jframe)
enable  = j_radiobutton(radio,"Enable middle Button")
middle  = j_radiobutton(radio,"Middle Button")
disable = j_radiobutton(radio,"Disable middle Button")
```

Znächst wird im Frame eine Radiogroup erzeugt, und anschließend drei Buttons hinzugefügt. In der Main Loop wird mit folgenden If Statements

```
    if(obj = enable)  j_enable(middle)
    if(obj = disable) j_disable(middle)
```

der mittlere Button enabled oder disabled. Der Zustand des mittleren Button ist im Beispiel ohne Bedeutung, und wird auch nicht abgefragt.

2.8 Choice

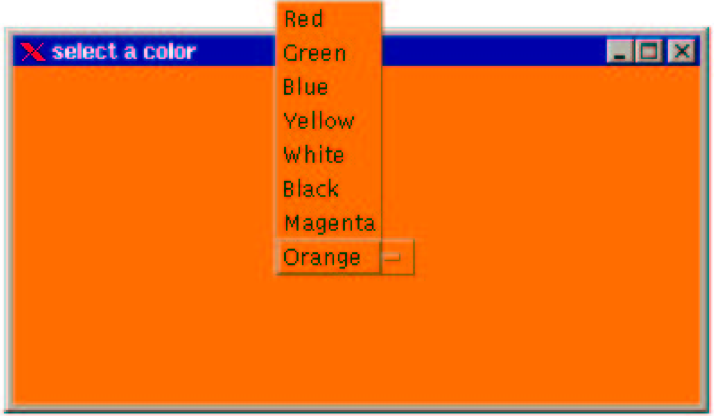


Abbildung 2.16: Beispiel für die Programmierung eines Auswahlmenüs.

Ein weiteres graphisches Element zur Auswahl aus einer festen Liste ist das Choice Element. Dabei befindet sich auf der Oberfläche zunachst ein Element das einem Button ähnlich sieht. Wird es angeklickt, so klappt ein Menu heraus, aus dem man ein Element auswählen kann. Das folgende Beispielprogramm entstammt dem Beispielprogramm der Checkboxen. Es wird eine gewisse Anzahl an Farben angeboten, aus denen eine Farbe ausgewählt werden kann. Mit dieser Farbe wird der Hintergrund des umgebenden Frames gesetzt:

```
rem Example choice.bas

:
choice  = j_choice(jframe)

j_additem(choice,"Red")
j_additem(choice,"Green")
j_additem(choice,"Blue")
j_additem(choice,"Yellow")
j_additem(choice,"White")
j_additem(choice,"Black")
j_additem(choice,"Magenta")
j_additem(choice,"Orange")
```

```
j_setpos(choice,150,120)

j_select(choice,3)
j_setnamedcolorbg(jframe,J_YELLOW)

j_show(jframe)

obj=0
while(obj <> jframe)

    obj=j_nextaction()

    if(obj = choice) then
        jcolor=j_getselect(choice)

        if(jcolor = 0) j_setnamedcolorbg(jframe,J_RED)
        if(jcolor = 1) j_setnamedcolorbg(jframe,J_GREEN)
        if(jcolor = 2) j_setnamedcolorbg(jframe,J_BLUE)
        if(jcolor = 3) j_setnamedcolorbg(jframe,J_YELLOW)
        if(jcolor = 4) j_setnamedcolorbg(jframe,J_WHITE)
        if(jcolor = 5) j_setnamedcolorbg(jframe,J_BLACK)
        if(jcolor = 6) j_setnamedcolorbg(jframe,J_MAGENTA)
        if(jcolor = 7) j_setnamedcolorbg(jframe,J_ORANGE)
    endif

wend
:
```

Mit der Anweisung:

```
choice     = j_choice(jframe)
```

wird zunächst das Grundelement der Choice erstellt. Dieses Element enthält zunächst noch kein Auswahlelement. Die Auswahlelemente werden mit der Anweisung:

```
j_additem(choice,"Red")
```

der Choice hinzugefügt. Diese Funktion liefert ausnahmsweise keine Identifikationsnummer zurück. Intern werden den Elementen laufende Nummern zugewiesen die von null beginnend aufsteigen. Das erste Element bekommt demnach die Nummer 0 zugewiesen, das zweite die 1 usw.

Mit der Funktion:

```
j_select(choice,3)
```

kann ein Element ausgewählt werden. Im Beispiel ist demnach das 4te element ausgewählt, und so erscheint nach dem Start die Farbe 'Yellow' in der Auswahlbox.

Da im Gegensatz zu den Checkboxen die Elemente einer Choice keine eigene Identifikationsnummer erhalten, wird bei einer Auswahl durch den Benutzer die Nummer der Choice an die Applikation übermittelt. Anschließend kann die laufende Nummer des Elements ermittelt werden:

```
selected = j_getselect(choice)
```

Diese Funktion liefert die laufende Nummer des Auswahlelementes zurück, das momentan sichtbar, also ausgewählt ist. Im Beispiel wird durch ein switch Statement eine der Wahl des Benutzers entsprechende Hintergrundfarbe geschaltet.

Mit Hilfe der folgenden (nicht im Beispiel benutzten) Funktionen kann eine dynamische Auswahlliste erstellt werden. Die Funktion

```
j_remove(choice, pos)
```

entfernt das Auswahlelement, das die laufende Nummer pos trägt. Mit der Funktion

```
j_insert(choice, pos, title$)
```

kann ein neues Auswahlelement an der Position pos eingefügt werden. Zu beachten ist allerdings bei diesen Funktionen, daß es stets zu einer Verschiebung der laufenden Nummern der nachfolgenden Auswahlelementen führt.

Die Beschriftung eines Auswahlelement kann man mit der Funktion:

```
str$ = j_getitem(choice, nr)
```

ermitteln. Diese liefert in der übergebenen Stringvariablen die Beschriftung des Eintrages zurück. Die Stringvariable muss dabei groß genug sein, um den String des Auswahlelement aufzunehmen. Die Anzahl aller Einträge liefert die Funktion:

```
nitems = j_getitemcount(choice)
```

zurück. Alle beschriebenen Funktionen erlauben somit eine dynamische Veränderung einer Choice während des Programmlaufs.

2.9 List

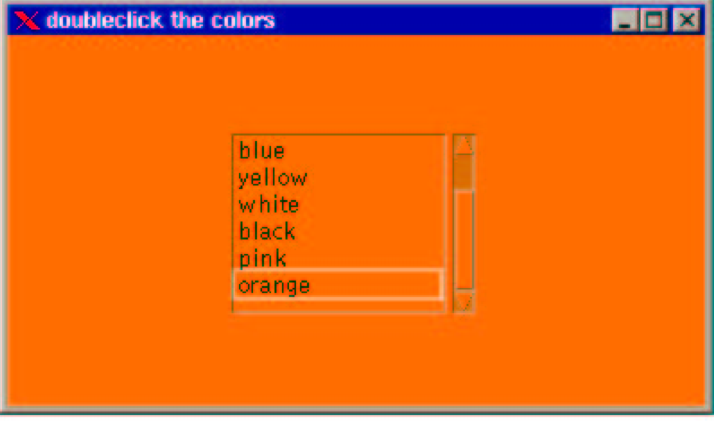


Abbildung 2.17: Beispiel für die Programmierung einer Auswahlliste.

Eine Liste stellt eine Alternative zu den Choices dar. Es bietet dem Benutzer ebenfalls eine vordefinierte Auswahlmöglichkeit an. Diese Auswahlmöglichkeiten werden in Form von anwählbaren Strings dargestellt. Eine bestimmte Anzahl der Strings ist permanent sichtbar, die restlichen sind über einen Scrollbalken erreichbar. Im Gegensatz zur Choice bietet die List die Möglichkeit auch mehrere Einträge zu selectieren.

Das folgende Beispiel demonstriert zunächst die Programmierung einer Liste mit einer einfachen Wahlmöglichkeit.

```
rem Example list.bas

:
list  = j_list(jframe,6)
j_additem(list,"Red")
j_additem(list,"Green")
j_additem(list,"Blue")
j_additem(list,"Yellow")
j_additem(list,"White")
j_additem(list,"Black")
j_additem(list,"Magenta")
j_additem(list,"Orange")

j_setpos(list,150,120)

j_select(list,3)

:
while(obj <> jframe)

    obj=j_nextaction()

    if(obj = list) then
        jcolor=j_getselect(list)

        if(jcolor = 0) j_setnamedcolorbg(jframe,J_RED)
        if(jcolor = 1) j_setnamedcolorbg(jframe,J_GREEN)
        if(jcolor = 2) j_setnamedcolorbg(jframe,J_BLUE)
        if(jcolor = 3) j_setnamedcolorbg(jframe,J_YELLOW)
        if(jcolor = 4) j_setnamedcolorbg(jframe,J_WHITE)
        if(jcolor = 5) j_setnamedcolorbg(jframe,J_BLACK)
        if(jcolor = 6) j_setnamedcolorbg(jframe,J_MAGENTA)
        if(jcolor = 7) j_setnamedcolorbg(jframe,J_ORANGE)

        j_deselect(list,j_getselect(list))
    endif

wend
:
```

Man erkennt, daß die Programmierung ähnlich ist, wie bei den Choice Objekten. Mit der Anweisung:

```
list = j_list(jframe,6)
```

Wird eine Liste generiert, die 6 sichtbare Items enthält. Der erste Parameter bestimmt also die Hoehe des sichtbaren Bereichs. Die Breite ist nicht fest bestimmbar, denn sie wird durch den Inhalt der Strings bestimmt. Eine Liste hat eine Default Einstellung von 4 sichtbaren Zeilen. Dieser Wert wird übernommen, wenn als weiter Parameterder ***j_list()*** Funktion eine 0 übergeben wird.

Wie bei den Choices werden mit den wiederholten Anweisungen:

```
j_additem(list,"red")
```

Auswahlstrings hinzugefügt. Diese Auswahlelemente werden ebenfalls intern mit fortlaufenden Nummern versehen, und zwar in der Reihenfolge, in der diese der Liste zugefügt werden. Die Anweisungen ***j_remove()*** und ***j_insert()*** funktionieren ebenfalls analog zu den Choices.

Die Defaulteinstellung einer Liste ist die 1 aus n Auswahl. Wird vom Benutzer ein String per Doppelklick ausgewählt, so wird die Applikation mit der Identifikationsnummer der Liste benachrichtigt. Analog zu den Choices kann nun mittels

```
j_getselect(list)
```

das ausgewählte Element ermittelt werden.
Mit der Funktion:

```
j_deselect(list,j_getselect(list))
```

kann ein Element deselectiert werde. Im Beispiel wird das ein selectierte Element sofort wieder deselectiert.

Abbildung 2.18: Beispiel für die Programmierung einer Auswahlliste mit mehrfacher Selectierung.

Es wurde bereits erwähnt, das in einer Liste auch mehrfache Selektierungen möglich sind. Um dies zu erreichen, muß der Liste die Anweisung:

```
j_multiplemode(list, J_TRUE )
```

übermittelt werden. Mit dem zweiten Parameter der Funktion kann die Mehrfachselektion ein- und ausgeschaltet werden. Um auf eine mehrfache Selektion reagieren zu können, meldet die Liste nun jeden einfachen Klick auf ein Element. Da nun mehrere Element selektiert sein können, sollte mit der Anweisung:

```
itemsel = j_isselect( choice, element_nummer )
```

der Zustand jedes Element abgefragt werden. Ein Aufruf von ***j_getselect(list)*** liefert nur das erste selectierte Element zurück. Das folgende Beispielprogramm demomstriert diese Arbeitsweise:

```
rem       Example listmultiple.bas

:
list    = j_list(jframe,3)

j_additem(list,"Red")
j_additem(list,"Green")
j_additem(list,"Blue")

j_multiplemode(list, J_TRUE )
```

```
j_select(list,0)
j_select(list,1)

:
while(obj <> jframe)
    obj=j_nextaction()

    if(obj = list) then
        r=0
        g=0
        b=0
        if(j_isselect(list,0) = J_TRUE) r=255
        if(j_isselect(list,1) = J_TRUE) g=255
        if(j_isselect(list,2) = J_TRUE) b=255
        j_setcolorbg(jframe,r,g,b)
    endif
    :
wend
:
```

Sobald der Benutzer durch einfaches Anklicken eines Elements dessen Zustand ändert, wird die Applikation benachrichigt. Die drei Auswahlelemente werden überprüft, und entsprechend des Zustands die Hintergrundfarben gesetzt.

Anstatt direkt die Events der Liste abzufangen, kann alternativ auch ein zusätzlicher OK–Button verwendet werden. In dessen Event Reaktion kann dann überprüft werden, welche Einträge selektiert sind. Dies gibt dem Benutzer die Möglichkeit in Ruhe die geeigneten Einträge auszuwählen. Das Beipiel benutzt einen solchen Button nicht, und reagiert somit immer sofort auf einen Selektionswechsel.

Die im Beispiel nicht benutzten Funktionen zum Manipulieren einer Liste funktionieren analog zum Choice–Element.

```
j_insert( list, pos, title$ )
j_remove( list, pos)
j_getitem( list, nr )
j_getitemcount( list )
```

2.10 Textarea

Ein Textarea Objekt stellt ein Ein- und Ausgabeelement für Texte dar. In dieses Textobjekt kann sowohl von der Applikation als auch vom Benutzer Texte eingetragen werden. Weiterhin bietet ein Textobjekt die Möglichkeit Bereiche mit der Maus zu selektieren. Das folgende Beispiel implementiert einen rudimentären Editor mit den Funktionen Cut, Copy und Paste:

```
rem       Example text.bas

:
(Menupukte New, Save, Quit, Select All,  Cut, Copy und Paste)
:
jtext    = j_textarea(jframe,25,12)

j_settext(jtext,newtext$)

while((obj <> jframe) and (obj <> quit))
    :

    if(obj = jtext) print "text changed", j_getlength(jtext)

    if(obj = jnew) j_settext(jtext,newtext$)

    if(obj = save) then
        inhalt$ = j_gettext(jtext)
```

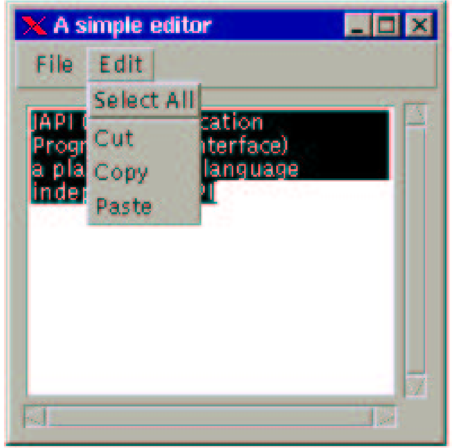


Abbildung 2.19: Beispiel für die Programmierung eines einfachen Editors mit Cut, Copy und Paste Funktionen.

```
            print inhalt$
        endif

        if(obj = selall) j_selectall(jtext)

        if((obj = cut) or (obj = copy) or (obj = paste)) then
            selstart=j_getselstart(jtext)
            selend  =j_getselend(jtext)
        endif

        if(obj = cut) then
            j_delete(jtext,selstart,selend)
            j_selecttext(jtext,selstart,selstart)
        endif

        if(obj = copy) inhalt$ = j_getseltext(jtext)

        if(obj = paste) then
            if(selstart = selend) then
               j_inserttext(jtext,inhalt$,j_getcurpos(jtext))
            else
               j_replacetext(jtext,inhalt$,selstart,selend)
            endif
            j_setcurpos(jtext,selstart)
        endif

    wend
    :
```

Der Editor enthählt weiterhin 7 Menupunkte, die folgende Funktion erfüllen:

1. New
   Der gesamte Text wird durch einen vordefinierten Text ersetzt

2. Save
   Der Inhalt des Textelementes wird in der Konsole ausgegeben

3. Quit

4. Select All
   Der gesamte Text wird selektiert.

5. Cut
   Der selektierte Bereich soll gelöscht werden

6. Copy
   Der selektierte Bereich wird in den Puffer kopiert

7. Paste
   Der Inhalt des Puffers wird an der Cursorposition in das Textobjekt eingefügt

Nach der Definition der Menupunkte wird ein Textelement erzeugt:

```
jtext    = j_textarea(jframe,25,12)
```

Die beiden ersten Argumente definieren die sichtbaren Spalten bzw. Zeilen des Textelements. Werden vom Benutzer über Zeilen bzw. Spaltenende hinausgeschrieben, so erfolgt automatisch eine Anpassung der beiden Scrollbars, die an jedem Textelement angefügt sind. Die folgende Zeile fügt einen vordefinierten Text in das Textelement ein. Der Text befindet sich in der Stringvariablen "newtext", die beliebig lang sein kann[2].

```
j_settext(jtext,newtext$)
```

In der folgenden Eventloop werden die Aktionen des Benutzers abgefragt. Werden Änderungen am Text vorgenommen, so wird die Applikation mit mit der ID des Textelementes benachrichtigt:

```
 if(obj = jtext) print "text changed", j_getlength(jtext)
```

Im Beipiel wird dann die Nachricht "text changed" ausgegeben, und die aktuelle Länge des Textes ermittelt. Dies erfolgt mit der Funktion:

```
j_getlength(jtext)
```

die als Rückgabewert die Länge des Textes enthält. Die Länge eines Textes entspricht der Anzahl der enthaltenen Zeichen.

Wird der Menupunkt "Save" angeklickt, so wird mit der Funktion:

```
inhalt$ = j_gettext(jtext)
```

der Inhalt des Textelement in die Stringvariable "inhalt" kopiert.

Der gesamte Text kann mit der Funktion:

```
j_selectall(jtext)
```

selektiert werden. Soll nur ein Teil des Textes selektiert werden so kann dies mit der Prozedur:

```
j_selecttext(jtext,selstart,selend)
```

[2] Grenzen sind gelegentlich vom verwendeten Compiler vorgegeben.

erreicht werden. Das zweite und dritte Argument bestimmt den Anfang bzw. das Ende) des zu selektierenden Bereichs. Da in der Regel jedoch vom Benutzer ein Textbereich selektiert wird, kann Anfang und Ende eines selektierten Textes mit den Funktionen:

```
selstart = j_getselstart(jtext)
selend   = j_getselend(jtext)
```

ermittelt werden. Die Funktionen liefern die Position des ersten bzw. letzten selektierten Zeichens zurück. Der Inhalt einen markierten Bereichs kann mit der Procedur:

```
inhalt$ = j_getseltext(jtext)
```

in die Stringvariable "inhalt" kopiert werden.

Um Texte einzufügen stehen zwei Funktionen zur Verfügung:

```
j_inserttext(jtext,inhalt$,pos)
j_replacetext(jtext,inhalt$,selstart,selend)
```

Dabei fügt die Funktion ***j_inserttext()*** den Inhalt der Stringvariablen "inhalt" an der Position "pos" im Text ein. Die Funktion ***j_replacetext()*** ersetzt den Textbereich, zwischen des Positionen "start" und "end", durch den Inhalt der Stringvariablen "inhalt". Im Beispiel werden mit diesen Funktionen die Paste Operationen realisiert.

Zum Löschen von Textbereichen steht die Funktion:

```
j_delete(jtext,selstart,selend)
```

zur Verfügung. Sie löscht den Bereich zwischen der angegebenen Start– und Endposition. Im Beispiel sollen dies der selektierte Bereich sein.

Der Textcursor kann mit der Funktion

```
j_setcurpos(jtext,selstart)
```

vor die angegebene Position plaziert werden. Analog wird mit der Funktion

```
j_delete(jtext,selstart,selend)
```

die aktuelle Position des Cursors zurückgegeben. Die Position versteht sich als 'Anzahl der Zeichen' vom Beginn des Textes an gezählt.

Mit Hilfe dieser präsentierten Funktionen realisiert das obige Beispiel einen einfachen Editor mit den wichtigsten Editierfunktionen.

2.11 Textfield

Ein Textfeld dient der einzeiligen Eingabe eines beliebigen Textes durch den Benutzer. Da im wesentlichen alle Funktionen des Textelementes (siehe voriges Kapitel) auch auf das Textfeld anwendbar sind, wird hier nur kurz auf zusätzliche Funktionen eingegangen. Das folgende Beispiel stellt ein login Dialogfenster dar:

```
rem       Example textfield.bas

:
login   = j_textfield(jframe,35)
passwd  = j_textfield(jframe,35)
:
j_setechochar(passwd,"*")
```

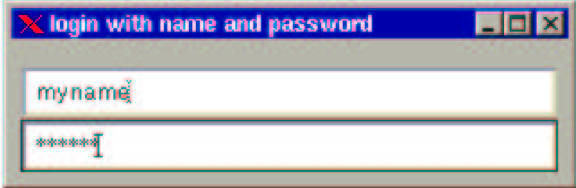


Abbildung 2.20: Beispiel für die Programmierung eines Login Feldes mit der Hilfe von Textfeldern. Das Password wird nicht lesbar dargestellt.

```
while((obj <> jframe) and (inhalt$ <> "exit"))
    :
    if(obj = login) then
        inhalt$ = j_gettext(login)
        print "Your name is ",inhalt$
    endif

    if(obj = passwd) then
        inhalt$ = j_gettext(passwd)
        print "Your password is ",inhalt$
    endif
wend
:
```

Da Textfelder nur einzeilig sind, besitzt der Funktionsaufruf dieses Elements auch nur einen Parameter zur Angabe der sichtbaren Spalten:

```
j_textfield(jframe,n_spalten)
```

Obwohl Textfelder keine Scrollbars besitzen, kann ein überlanger Text im Sichtfenster gescrollt werden, da der Textcursor immer im sichtbaren Bereich bleibt.

Ein Textfield meldet erst dann einen Event, wenn die Eingabe in das Feld durch drücken der Return Taste bestätigt wurde. Dennoch kann jederzeit der Inhalt des Feldes mit der Prozedur j_gettext() ermittelt werden

Soll, z.B. bei Eingabe eines Passwords, kein erkennbarer Text angezeigt werden, so kann mit der Funktion

```
j_setechochar(passwd,"*")
```

ein sogenanntes "echo character" angegeben werden. Im Beispiel wird also in der Password Zeile für jedes eingetippte Zeichen ein Asterix dargestellt. Denoch entspricht der Inhalt des Textfeldes genau der eingetippten Zeichenfolge. Im Beispiel kann im Password Feld die Zeichenfolge "exit" eingetippt werden, um die Applikation zu beenden.

2.12 Popupmenu

Ein Popupmenu bietet ähnliche Funktionalität wie ein normales Menu, das in einer Menubar angebracht ist. Der Unterschied zu einem normalen Menu besteht darin, das es an jeder beliebigen Position innerhalb der Applikation geöffnet werden kann. Das folgende Beispiel öffnet ein Popupmenu in einem Frame, sobald ein Menupunkt "Popup" in der normalen Menuleiste angewählt wird:

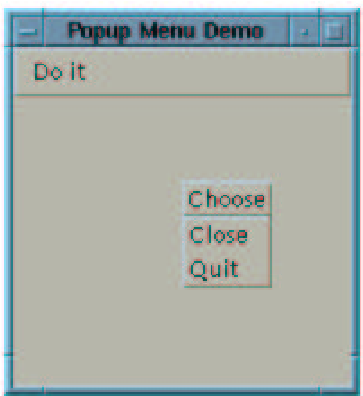


Abbildung 2.21: Beispiel für die Programmierung eines Popupmenüs

```
rem       Example popupmenu.bas

:
jframe  = j_frame("Popup Menu Demo")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"Do it")
pop     = j_menuitem(file,"Popup")

choose  = j_popupmenu(jframe,"Choose")
jclose  = j_menuitem(choose,"Close")
quit    = j_menuitem(choose,"Quit")

j_show(jframe)

while(obj <> quit)

    obj =  j_nextaction()

    if(obj = pop) j_showpopup(choose,100,100)

wend
:
```

Folgende drei Programmzeilen bauen die Elemente des Popupmenüs auf:

```
choose  = j_popupmenu(jframe,"Choose")
close   = j_menuitem(choose,"Close")
quit    = j_menuitem(choose,"Quit")
```

Der Aufbeu eines Popupmenüs erfolgt analog zu einer Standartmenüzeile (siehe Kapitel 2.2). Als Aufnahme–Element ist dabei jedoch keine Menubar erforderlich. Im Beispiel wird der Menueintrag an den Frame der Applikation gebunden. Prinzipiell kann an jedes graphische Element (außer den Menuelementen selbst) ein Popupmenu angebunden werden. Als Menueinträge dienen die normalen Menuitems.

Ein Popupmenu wird durch Aufruf der Funktion:

```
j_showpopup(choose,100,100)
```

sichtbar. Die beiden hinteren Parameter geben die x,y Position an, an der das Popupmenu erscheinen soll. Sobald ein (oder kein) Menuitem ausgewählt wurde, verschwindet

das Popupmenu automatisch. Daher ist keine entsprechende Funktion zum Schließen des Popupmenu vorhanden.

2.13 Scrollbar

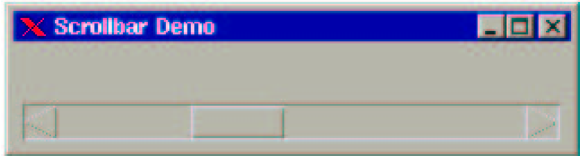


Abbildung 2.22: Beispiel für die Programmierung eines Schiebebalkens.

Scrollbars sind verikale oder horizontale Schieberegler, die in der Regel dazu verwendet werdenFenterinhalte zu verschieben. So sind an einem Textelement automatische zwei Scrollbars angebunden. Mit Hilde der Scrollbars kann so auch Canvas aufgebaut werden, bei dem der Fensterinhalt verschoben werden kann. Da dies jedoch einen nicht unerheblichen Programmieraufwand bedeutet, wird im nächsten Kapitel ein Element (Scrollpane) vorgestellt, mit dem ein spolches Canvas viell leichter erstellt werden kann.

Scrollbar können jedoch auch für andere Zwecke benutzt werden. Das folgende Beispiel verwendet einen Scrollbar, um vom Benutzer einen Zahlenwert zu erhalten, der nur aus einem vorher festgelegten Zahlenbereich stammen darf:

```
rem       Example scrollbar.bas

:
scroll=j_hscrollbar(jframe)
:
j_setmin(scroll,10)
j_setmax(scroll,150)
j_setslidesize(scroll,28)
j_setunitinc(scroll,2)
j_setblockinc(scroll,12)
j_setvalue(scroll,50)

while(obj <> jframe)

    obj =  j_nextaction()

    if(obj = scroll) print"Scrollbar Value : ",j_getvalue(scroll)

wend
:
```

Die Funktion:

```
scroll=j_hscrollbar(jframe)
```

erzeugt zunachst einen Scrollbar, der mit initialen Werten belegt ist [3]. Ein Scrollbar bietet ein Vielzahl von Einstellungsmöglichkeiten, die im folgenden einzeln aufgeführt werden.

Die obere und untere Grenze des wählbaren Bereichs wird mit den Funktionen:

[3] Die initialen Werte können im Reference Teil nachgelesen werden

```
j_setmin(scroll,10)
j_setmax(scroll,150)
```

festgelegt. Dabei legt die obere Grenze den linken Rand des Scrollbars fest. Dieser kann vom Benutzer jedoch nicht erreicht werden, da die Breite des Schiebereglers von diesem Maximalen Wert abzuziehen ist. Werden diese Werte nicht gesetzt, so werden die default Werte von 0 für die untere Grenze und 100 für die obere Grenze übernommen. Die Breite des Schieberegler wird mit der Funktion:

```
j_setslidesize(scroll,28)
```

festgelegt. Daraus ergibt sich für das Beispiel, das mit dem Regler ein Wertebereich von 10 bis $150 - 28 = 122$ angewählt werden kann. Die Standardeinstellung für diesen Wert ist 10. Um den Schieber zu bewegen hat der Benutzer mehrere Möglichkeiten. Die einfachste Art ist den Regler mit der Maus zu greifen, und direkt zu bewegen. Zusätzlich gibt es noc die Möglichkeit die Pfeile am Ende der Scrollbar anzuklicken. In diesem Fall wird der Regel jeweil um eine Einheit in die entsprechende Richtung bewegt. Die Größe dieser Einheit kann mit dem Befehl:

```
j_setunitinc(scroll,2)
```

eingestellt werden. Der Defaulteintrag ist 1. Neben den Pfeilen kann auch der freie Raum neben dem Regler angeklickt werden. In diesem Fall wird der Regler um eine Blocllänge verschoben. Die Blockgröße kann mit dem Befehl:

```
j_setblockinc(scroll,12)
```

eingestellt werden. Der Defaulteintrag ist 10.

Die initiale Position des Reglers wird mit der Funktion:

```
j_setvalue(scroll,50)
```

eingestellt. Der Defaulteintrag ist 0.

2.14 Panel

Ein Panel stellt an sich kein eigenes graphisches Element dar. Es dient vielmehr dazu andrere Elemente aufzunehmen. Ein Panel ist also eine Art Behälter, in dem weitere graphische Elemente positioniert werden können. Somit ähnelt der Panel dem Frame, unterscheidet sich jedoch deutlich von diesem, da er keine Menubar aufnehmen kann, und immer ein Parent Object besitzen muß.

Mit Hilfe eines Panels lassen sich die Probleme des Beispiel aus Kapitel 2.5 beheben. In diesem Beispiel wurde ein Label animiert in einem Frame hin und her bewegt. Dabei tauchte das Label allerdings hinter die Menubar, und war in dieser Zeit nicht sichtbar. Weiterhin mußte permanent die Größe des Frames abgefragt werden, was zu deutlichen Laufzeitverzögerungen führte. Dies kann mit einem Panel vermieden werden, da ein Panel bei einer Größenveränderung einen Event an die Applikation meldet.

Im folgenden nun das optimierte Program:

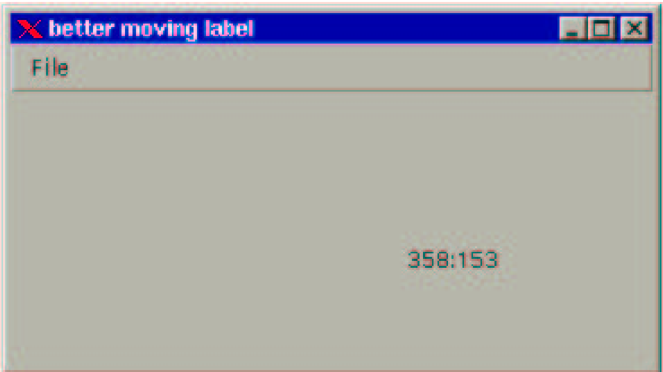


Abbildung 2.23: Eine verbesserte Version des Beispiels aus Kapitel 2.5 mit Hilfe eines Panels.

```
rem       Example panel.bas

:
jframe   = j_frame("Better Moving Label")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
doit    = j_menuitem(file,"Start")
quit    = j_menuitem(file,"Quit")

j_setborderlayout(jframe)
panel   = j_panel(jframe)

j_show(jframe)

jlabel   = j_label(panel,"256:256")
j_setpos(jlabel,120,120)

obj = 0
while((obj <> jframe) and (obj <> quit))

    if(run = 1) then
        obj=j_getaction()
    else
        obj=j_nextaction()
    endif

    if(obj = doit) then
        if(run = 0) then
            run=1
            j_settext(doit,"Stop")
        else
            run=0
            j_settext(doit,"Start")
        endif
    endif

    if(obj = panel) then
        p_width=j_getwidth(panel)
        p_height=j_getheight(panel)
        in$ = str$(p_width)+":"+str$(p_height)
        j_settext(jlabel,in$)
        l_width  = j_getwidth(jlabel)
        l_height = j_getheight(jlabel)
        x=(p_width-l_width)/2
        y=(p_height-l_height)/2
        j_setpos(jlabel,x,y)
    endif
```

```
        if(run = 1) then
            if((x+l_width  > p_width) or (x < 1))  dx = -dx
            if((y+l_height > p_height) or (y < 1)) dy = -dy
            j_setpos(jlabel,x+dx, y+dy)
            j_sync()
            j_sleep(1)
            x=x+dx
            y=y+dy
        endif

    wend
    :
```

Da große Teile des Programms unverändert übernommen wurden soll hier nur der relevante Teil besprochen werden. Neu sind die Zeilen:

```
j_setborderlayout(jframe)
panel  = j_panel(jframe)
```

In einem Objekt, das andere graphische Elemente aufnimmt (Frame, und wie wir jetzt auch wissen, ein Panel) kann neben einer fixen Positionierung auch eine automatische Positionierung vorgenommen werden. Dazu müssen sogenannte Layoutmanager an die Objekte gebunden werden. Genaueres findet man dazu im Kapitel 3. Im Beispiel weist der Borderlayout – Manager nun dem Panel den gesammten Platz zu, der in dem Frame vorhanden ist. Wird nun vom Benutzer die Größe des Frames verändert, so verändert sich auch die Größe des Panels. Damit wird automatisch ein Event des Panels ausgelöst:

```
if(obj = panel) then
    p_width=j_getwidth(panel)
    p_height=j_getheight(panel)
    in$ = str$(p_width)+":"+str$(p_height)
    j_settext(jlabel,in$)
    l_width  = j_getwidth(jlabel)
    l_height = j_getheight(jlabel)
    x=(p_width-l_width)/2
    y=(p_height-l_height)/2
    j_setpos(jlabel,x,y)
endif
```

Dieser Teil nimmt die wesentlichen Änderungen des Programms ein. Zunächst wird die geänderte Größe des Panels ermittelt und als String in den Label eingetragen. Die restlichen Zeilen ermitteln die neue Größe des Labels und positionieren dieses mittig im Panel.

Wichtig ist, daß dem Panel nur der freie Platz unterhalb der Menubar zugewiesen wird, sofern eine Menubar vorhanden ist. Daher kann das Label auch nicht mehr hinter der Menübar verschwinden. Der Einsatz der Prozedur ***j_sync()*** und ***j_sleep()*** wird hier wieder nötig, da das Programm nun so schnell läuft, daß das Label sonst nur wild umher zu springen scheint.

2.15 Scrollpane

Eine Scrollpane ist prinzipiell dasselbe wie ein Panel. Der Unterschied beteht darin, daß eine scrollpane bereits zwei Scrollbars intergriert hat, die es erlauben, den Inhalt des Panels zu scrollen. Somit kann eine Scrollpane viel größere Elemente aufnehmen, als sie eigentlich darstellen kann. Da eine Scrollbar nur aus bereits bekannten Elementen aufgebaut ist, sind alle Funktionen bereits bekannt. Das folgende Beispiel erzeugt eine einfache Graphik, die durch die Scrollbars durch das Sichtfenster verschoben werden kann:

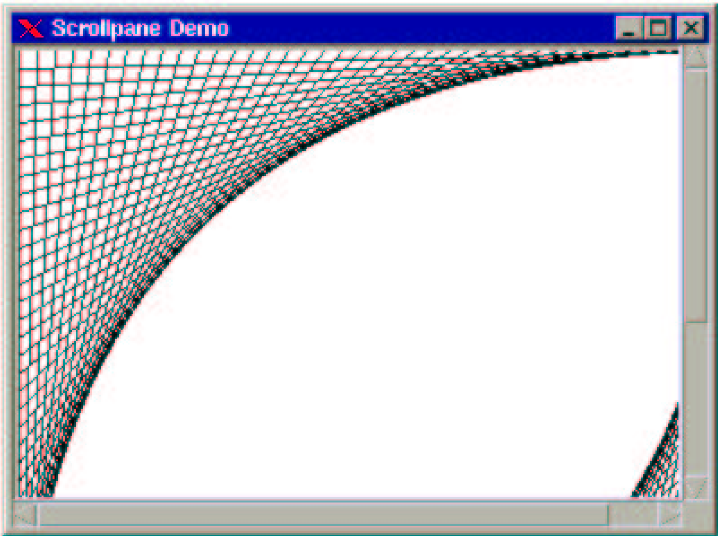


Abbildung 2.24: Beispielprogramm für einen Scrollpane.

```
rem       Example scrollpane.bas

:
jframe      = j_frame("Scrollpane Demo")
j_setborderlayout(jframe)

scrollpane = j_scrollpane(jframe)
vscroll    = j_vscrollbar(scrollpane)
hscroll    = j_hscrollbar(scrollpane)

canvas=j_canvas(scrollpane,400,400)

j_setsize(scrollpane,180,150)

j_pack(jframe)
j_show(jframe)

for x= 0 to 400 step 10
    j_drawline(canvas,x,0,0,400-x)
    j_drawline(canvas,x,400,400,400-x)
next x

obj = 0
while(obj <> jframe)
    obj =  j_nextaction()

    if(obj = jframe) goto 20

    if(obj = scrollpane) then
        width =j_getviewportwidth(scrollpane)
        height=j_getviewportheight(scrollpane)
    endif

    if(obj = hscroll) x=j_getvalue(hscroll)
    if(obj = vscroll) y=j_getvalue(vscroll)

    print "Viewport X = ",x,":",x+width
    print "         Y = ",y,":",y+height
wend
```

```
:
```

Auch in diesem Beispiel wird dem Frame zunächst ein Border Layoutmanager zugeteilt, der sicherstellt, daß bei einer Größenänderung des umgebenden Frames das enthaltene Scrollpane diese Größe zugeordnet bekommt. In den nächsten Anweisungen wird das Scrollpane erzeugt, und die EventID's der beiden Scrollbars ermittelt:

```
scrollpane  = j_scrollpane(jframe)
vscroll     = j_vscrollbar(scrollpane)
hscroll     = j_hscrollbar(scrollpane)
```

Die Scrollbars werden bereits durch die Funktion ***j_scrollpane()*** erzeugt. Der Aufruf der beiden Funktionen ***j_vscrollbar()*** und ***j_hscrollbar()*** weist den scrollbars lediglich die benötigten Event Identifier zu. Da das Scollen des enthaltenen Elements automatisch erfolgt, wird diese Zuweisung von EventID's in der Regel nicht benötig.

Nach dem Erzeugen eines Canvas für die Graphik, wird der Scrollpane eine Fläche zugewiesen, die sich von der des Canvas unterscheidet:

```
canvas=j_canvas(scrollpane,400,400)
j_setsize(scrollpane,180,150)
```

Man erkennt, daß der Canvas eine deutlich größere Fläche einnimmt, als das Scrollpane. Mit diesen Anweisungen ist die eigentliche Programmierung bereits erledigt. Nach dem Anzeigen der Applikation kann der Canvas im Scrollpane verschoben werden. Die folgende Eventloop im Beispiel dient lediglich dem Ermitteln des akteullen Ausschnitts des sichtbaren Bereichs.

Wie ein Panel, so meldet auch ein Scrollpane einen Event, sobald sich seine Größe verändert. Mit den Funktionen:

```
width  = j_getviewportwidth(scrollpane)
height = j_getviewportheight(scrollpane)
```

läßt sich dann Höhe und Breite des sichtbaren Bereichs ermitteln.

Sobald einer der beiden Scrollbars bewegt wird, meldet dieser einen entsprechenden Event. Mit der bereits bekannten Funktion:

```
j_getvalue(scrollbar)
```

läßt sich die atuelle Position eines der Schieberegler ermitteln. Somit sich alle Informationen vorhanden, um den sichtbaren Ausschnitt des Gesamtbildes zu berechnen. Im Beispiel wird diese Bereich jeweils durch eine print Anweisung in der Konsole ausgegeben.

2.16 Dialog

Eine Dialogbox stellt ein weiteres Element dar, das weitere graphische Elemente beinhalten kann. Wie ein Frame, so ist eine Dialogbax ein eigenständiges Fenster, das jedoch keine Menübar besitzen kann. So ist eine Dialogbox zwischen einem Frame und einem Panel anzusiedeln. Eine Dialogbax wird, wie derName schon sagt, dazu benutzt, um temporäre Dialogfenster aufzubauen. Das folgende Beispiel zeigt, wie es gemacht wird:

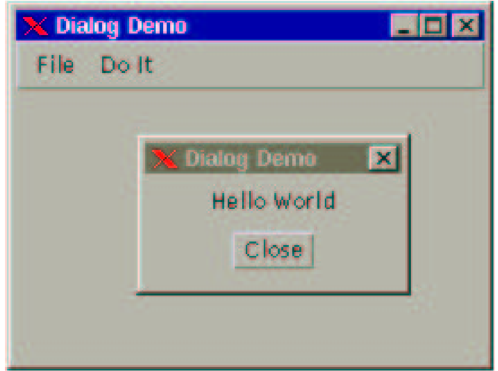


Abbildung 2.25: Beispielprogramm mit einem Dialogfenster.

```
rem       Example dialog.bas

:
dialog  = j_dialog(jframe,"Say Hello!")
j_setflowlayout(dialog,J_VERTICAL)

jlabel   = j_label(dialog,"Hello World")
jclose   = j_button(dialog,"Close")

j_pack(dialog)

obj = 0
while((obj <> jframe) and (obj <> quit))
     obj =  j_nextaction()

     if(obj = jopen) then
         x = j_getxpos(jframe)+j_getwidth(jframe)/2 - j_getwidth(dialog)/2
         y = j_getypos(jframe)+j_getheight(jframe)/2 -  j_getheight(dialog)/2
         j_setpos(dialog,x,y)
         j_show(dialog)
      endif

      if((obj = jclose) or (obj = dialog)) j_hide(dialog)

wend
```

Zunächst wird ein Dialogfenster erzeugt:

```
dialog  = j_dialog(jframe,"Dialog Demo")
```

Die Funktion erwartet eine Überschrift, die im Rahmen der Dialogbox eingeblendet wird, sowie das übergeordnetes Fenster, dem die Dialogbox zugeordnet wird. In diese Dialogbox wird nun im Beispiel ein Label zur Aufnahme einer Nachricht, sowie ein Closebutton eingefügt. Die nächste Funktion:

```
j_pack(dialog)
```

weist den Layoutmanager des Dialogfeldes an, die Elemente so platzspared wie möglich anzuordnen[4].

Über den Befehl:

[4]Mehr zu den Layoutmanagern findet sich im Kapitel 3

```
j_show(dialog)
```

wird die Dialogbox sichtbar, was nicht weiter verwundert. Da die Positionierung einer Dialogbox in absoluten Bildschirmkoordinaten erfolgt, wird zunächst die Position des Frames ermittelt, und aus der Hoehe und Breite beider Fenster eine, zur Frame mittige Psosition errechnet:

```
x = j_getxpos(jframe)
y = j_getypos(jframe)
j_setpos(dialog,x,y)
```

Eine Dialogbox sendet, genau wie ein Frame, einen Event, sobald das Closeicon im Rahmen der Dialogbax angeklickt wurde. Somit sorgt die Abfrage nach dem Closebutton und dem Closeicon für ein Verschwinden der Dialogbox:

```
    if((obj = close) or (obj = dialog)) j_hide(dialog)
```

Während die Dialogbox zu sehen ist, kann weiterhin die Menuzeile des Hauptfensters bedient werden. Häufig ist jedoch erwünscht, das eine Dialogbox *modal* ist, d.h. außer diesem Fenster sollen alle anderen Fenster deaktiviert sein. Dies kann erreicht werden, indem man das Hauptfenster, den Frame, disabled. Wird ein Behälterelement (Frame, Panel, Dialog) disabled, so werden alle in im enthaltenen Elemente deaktiviert. Ein modales Dialogfeld kann somit mit folgender Modifikation erzeugt werden:

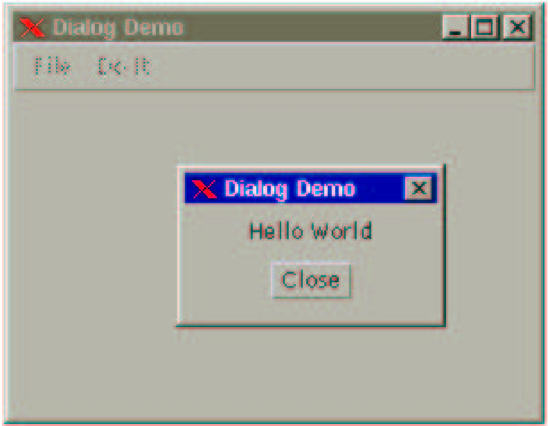


Abbildung 2.26: Beispielprogramm mit einem Dialogfenster. Solange die Dialogbox geöffnet ist, wird das Hauptfenster disabled.

```
if(obj = open) then
    j_disable(jframe)
    :
    :
endif

if((obj = close) or (obj = dialog)) then
    j_hide(dialog)
    j_enable(jframe)
endif
```

Solange nun das Dialogfenster sichtbar ist, ist der Frame (und alle enthaltenen Elemente) disabled (siehe Abbildung 2.26). Natürlich darf man nicht vergessen, nach dem Schließen des Dialogfeldes den Frame wieder zu aktivieren.

2.17 Window

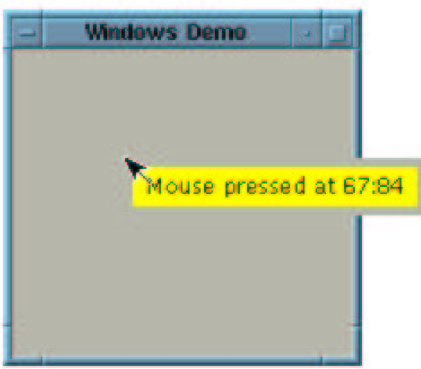


Abbildung 2.27: Beispielprogramm mit einem Window.

Das letzte Element, das dazu dient andere Elemente aufzunehmen ist ein Window. Ein Window ist, im Vergleich zu den anderen Behältern, das einfachste Element. Es besteht lediglich aus einem Rahmen, und kann weder eine Menüzeile noch eine Fensterleiste besitzen. Damit kann ein Window auch nicht vom Benutzer verschoben oder geschlossen werden. Dies muß alles von der Applikation erledigt werden. Ein Window wird häufig als Online–Hilfe verwendet. Das folgende Beispiel zeigt, wie in einem Frame auf Mausklick ein Window geöffnet wird, das die aktuelle Mausposition anzeigt. Wird der Mausbutton wieder losgelassen, so verschwindet auch das Window:

```
rem       Example window.bas

:
jframe   = j_frame("Window Demo")

window  = j_window(jframe)
j_setflowlayout(window)
jlabel   = j_label(window,"")
j_setnamedcolorbg(jlabel,J_YELLOW)

pressed  = j_mouselistener(jframe,J_PRESSED)
released = j_mouselistener(jframe,J_RELEASED)

:
while(obj <> jframe)
    obj =  j_nextaction()

    if(obj = pressed) then
        fx = j_getxpos(jframe)
        fy = j_getypos(jframe)
        x  = j_getmousex(pressed)
        y  = j_getmousey(pressed)
        message$ = "Mouse pressed at "+str$(x)+":"+str$(y)
        j_settext(jlabel,message$)
        j_setpos(jwindow,fx+x,fy+y)
        j_pack(jwindow)
        j_show(jwindow)
    endif

    if(obj = released) j_hide(jwindow)

wend
```

```
:
```

Die Funktion:

```
window  = j_window(jframe)
```

erzeugt zunächst das Window, wiederum ohne es anzuzeigen. In dieses Window wird nun ein Label gepackt, das mittels eines Flowmanagers plaziert wird[5]. Die Funktionen:

```
pressed  = j_mouselistener(jframe,J_PRESSED)
released = j_mouselistener(jframe,J_RELEASED)
```

erzeugen zwei (virtuelle) Objekte, die jeweils einen Event melden, wenn der Benutzer im Frame die Maustaste drückt, bzw. wieder losläßt[6]. Die folgende Programmsequenz reagiert nun auf den Mausdruck:

```
if(obj = pressed) then
    fx = j_getxpos(jframe)
    fy = j_getypos(jframe)
    x  = j_getmousex(pressed)
    y  = j_getmousey(pressed)
    message$ = "Mouse pressed at "+str$(x)+":"+str$(y)
    j_settext(jlabel,message$)
    j_setpos(jwindow,fx+x,fy+y)
    j_pack(jwindow)
    j_show(jwindow)
endif
```

Zunächst wird die aktuelle Position des Frames und der Maus ermittelt. Die Position der Maus wird relativ zum umgebenden Fenster (Frame) angegeben. Aus beiden Positionen errechnet sich nun durch Addition die Position des Windows, die in absoluten Bildschirmkoordinaten angegeben werden muß. Mit der Funktion ***j_setpos()*** kann das Window nun positioniert werden, und mit ***j_show()*** wird es am Bildschirm angezeigt. Zuvor wird die Mausposition als Nachricht in das Labelobjekt eingetragen. Somit öffnet sich das Window genau an der Mausposition, und zeigt dessen Position als Nachricht an.

Mit den Anweisungen

```
if(obj = released) j_hide(window)
```

wird das Window wieder vom Schirm entfernt, sobald die Maustaste wieder losgelassen wird, und ein entsprechendes Event geliefert wird. Dabei wird es jedoch nicht zerstört, und kann daher (wie im Beispiel) jederzeit wieder dargestellt werden.

2.18 Filedialog

Ein Filedialog ist eine vorbereitete Komponente, die auf jedem Rechner mit graphischer Oberfläche existiert. Unter JAPI wird die, zu dem System, vorhandene Fileselectorbox aufgerufen, um auf jedem Rechner das Look and Feel zu erhalten. Das folgende Beispiel bindet an zwei Menupunkte ”Open” und ”Save”, den Aufruf eines Filedialogs:

[5] Näheres dazu findet sich im Kapitel 3

[6] Eine genaue Beschreibung dazu findet sich im Kapitel 4

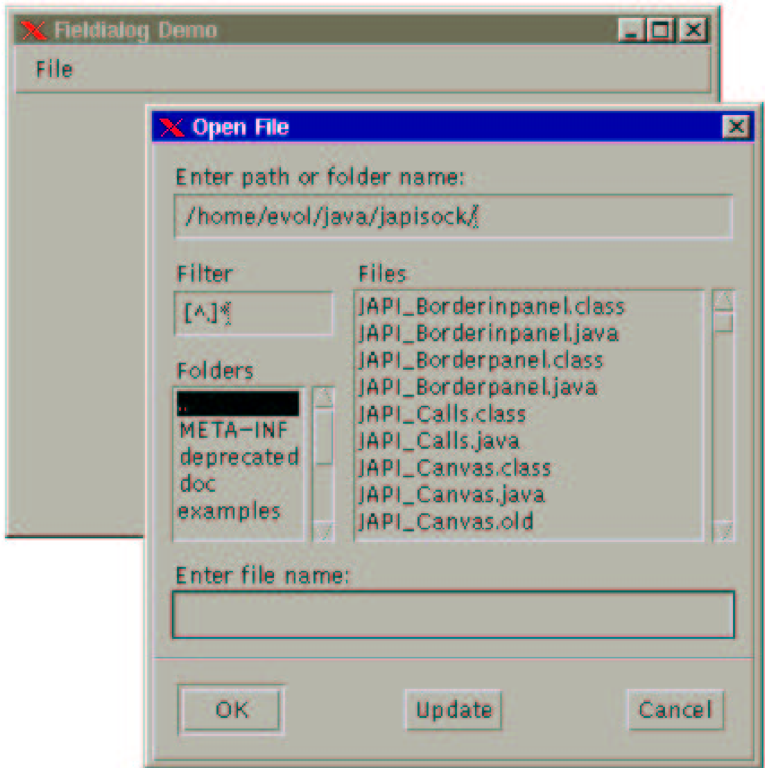


Abbildung 2.28: Beispielprogramm mit einem Filedialog Element.

```
rem       Example filedialog.bas

        :
        jframe    = j_frame("Filedialog Demo")
        menubar  = j_menubar(jframe)
        file     = j_menu(menubar,"File")
        open     = j_menuitem(file,"Open")
        save     = j_menuitem(file,"Save")
        quit     = j_menuitem(file,"Quit")

        :
        while((obj <> jframe) and (obj <> quit))
            obj =  j_nextaction()

            if(obj = jopen) then
                filename$=j_filedialog(jframe,"Open File","..")
                print "Open File ",filename$
            endif

            if(obj = save) then
                filename$=j_filedialog(jframe,"Save File",".")
                print "Save File ",filename$
            endif

        wend
        :
```

Ein Filedialog wird mit der Funktion:

```
j_filedialog(jframe,"Open File","..")
```

erzeugt und dargestellt. Der Programmablauf unterbricht an dieser Stelle, bis von Benutzer eine entsprechende Eingabe gemacht wurde. Wurde ein Filename angewählt oder eingegeben, so wird dieser zurückgegeben. Wird kein File ausgewählt, oder der Cancel Button angewählt, so wird ein leerer String zurückgegeben.

Zu beachten ist, daß der Filedialog auf derselben Maschine erzeugt wird, auf der auch der JAPI–Server läuft. Es können somit nur auf Files zugegriffen werden, auf die diese Maschine auch Zugriff hat. Ist kein gemeinsames Filesystem vorhanden, so kann dies zu Schwierigkeiten führen. Eine Lösung wäre einen eigenen Fileselector zu schreiben, der dann auf der Applikationsebene läuft. Dann ist aber auch das Look and Feel beim Teufel.

Kapitel 3

Die Layoutmanager

Bevor wir die vorhandenen Layoutmanager nächer besprechen, zunächst ein paar Wort, zu den Layoutmanagern an sich, und warum man sie verwenden sollte.

In der Regel werden die Graphischen Objekte einer Applikation in Größe und Position festgelegt. Dies nennt man absolute Positionierung. Ein Layout Manager übernimmt nun diese Aufgaben, d.h. er legt die Größe und die Positionierung der Objekte fest. Damit hat man nun zwar weniger Einflüß auf die Gestaltungsmöglichkeit innerhalb der Applikation, gewinnt aber auch eine Menge nicht zu verachtender Vorteile.

Ein Vorteil ist zunächst, daß man eben keine Layout Fummelei mehr machen muß. Wer schon einmal eine Applikation entworfen hat, und dabei Pixelgenau Ort und Göße eines Objektes festlegen muß, weiß welcher Zeitaufwand damit verbunden sein kann. Bei konsequenter Anwendung von Lauoutmanagern kann auf alle j_setpos() und j_setsize() Funktionen verzichtet werden.

Ein weiterer Punkt, der für die Verwendung von Layoutmanagers spricht, ist die Plattformunabhängigkeit. Es ist praktisch unmöglich, eine Applikation mit absoluter Positionierung zu schreiben, die auf allen Platformen vernünftig aussieht. Insbesondere durch die verschiedenen Fontgrößen und -formen wird ein so unterschiedlicher Platzbedarf gefordert, daß eine feste Größe z.B. für ein Labelobjekt niemals immer vernüftig aussieht.

Zudem sollte eine Applikation immer so geschrieben sein, daß der Benutzer die Fenstergröße verändern kann. Allein schon durch die unterschiedlichen Bildschirmgrößen ist ein solche Forderung einsichtig. Wählt man nun eine absolute Positionierung, so muß bei jeder Fensteränderung ein neues Layout berechnet werden. Abgesehen von dem hohen Programmieraufwand, ist diese Berechnung äußerst lästig, und verführt oft dazu, doch eine unveränderliche Fenstergröße zu wählen.

All diese Probleme verschwinden durch die Verwendung von Layoutmanagern. Allerding muß klar gesagt werden, daß nicht jedes gewünschte Layout mit den vorhandenen Managern realisiert werden kann. Entweder man einigt sich auf einen Kompromiss aus Layout und Machbarkeit, oder man realisiert das Layout eben doch mit der Hand.

Layoutmanager könen in jedem Object angelegt werden, das andere Elemente aufnehmen kann. Die sind unter JAPI die Elemente vom Typ:

- Frame
- Panel

- Dialog
- Window

Für diese Typen wird im folgenden der Begriff ”Container” festgelegt. Alles was sich im folgenden auf einen Container bezieht, ist demnach auf die oben genannten Elemente anwendbar.

3.1 Flowlayout

Der einfachste Layoutmanager ist der Flowlayout Manager. Er kann den Behälterelementen (Frame, Panel, Dialog, Window) mit der Funktion:

```
j_setflowlayout(jframe,J_HORIZONTAL)
```

zugeordnet werden. Das folgende Beispiel ordnet vier Objekte in einen Frame ein, und verwendet dazu den Flowlayoutmanager:

```
rem Example flowsimple.bas

:
jframe     = j_frame("Simple Flowlayout Demo")

j_setflowlayout(jframe,J_HORIZONTAL)

button1    = j_button(jframe,"button1")
button2    = j_button(jframe,"button2")
button3    = j_button(jframe,"button3")
button4    = j_button(jframe,"button4")

j_sethgap(jframe,20)
j_setvgap(jframe,20)

j_setsize(button1,200,200)
j_setsize(button2,200,200)

j_pack(jframe)
j_show(jframe)

while(j_nextaction()<> jframe)
wend
:
```

Die (fast) leere Eventloop soll uns nicht weiter stören, da das Beipiel lediglich die Funktionen des Flowlayoutmanager demonstrieren soll. Die Funktionsweise kann auch nur vollständig erfaßt werden, wenn man das Beipiel ausprobiert, und die Fentergröße verändert. Dabei stellt man fest, das der Layoutmanager die Objekte bei Bedarf umgruppiert.

Ein Flowlayout Manager verfolgt die Stategie, alle Objekte nebeneinander möglichst platzsparend anzuordnen. So erscheint das Beispiel mit der Abbildung 3.3 nach dem Starten der Applikation. Man erkennt daß alle Objekte in Ihere minimalen Größe adrgestellt werden, und nebeneinander angeordnet werden. Die minimale Größe eines Objekt berechnet sich aus einer zugeordneten Größe wie die beiden ersten Button, oder aus dem Inhalt, bei Button 3 und 4 eben aus der Beschriftung.

Die Funktion

```
j_pack(jframe)
```

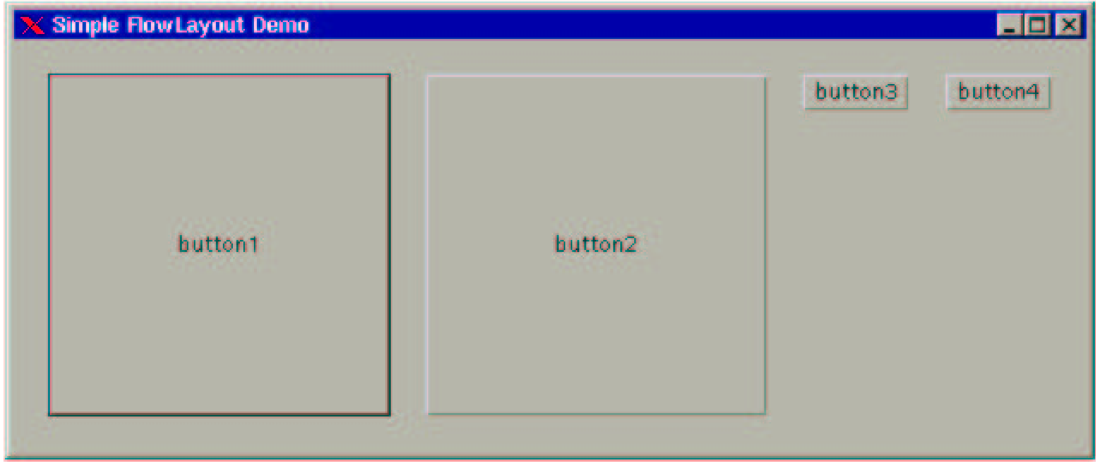


Abbildung 3.1: Beispiel für eine einfache Positionierung mit einem Flowlayoutmanager.

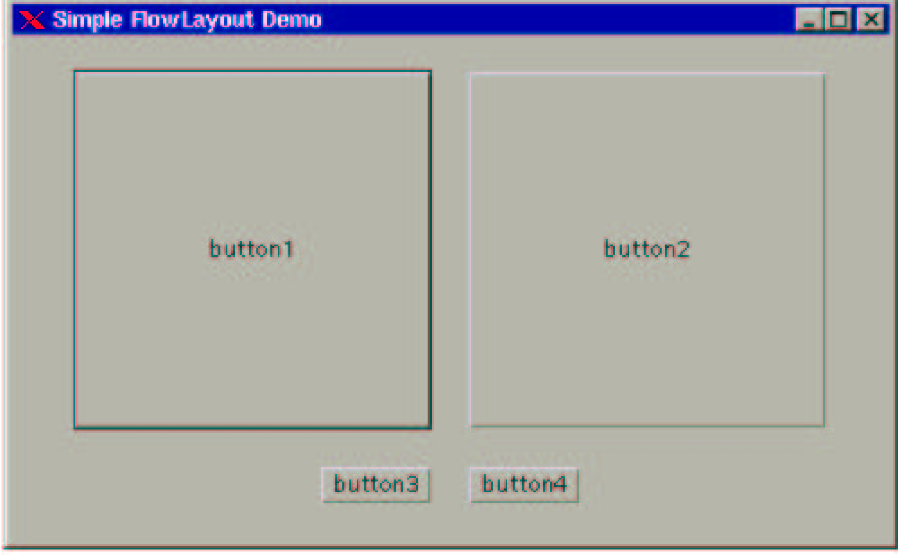


Abbildung 3.2: Die Positionierung der Objekte, nachdem die Fenstergröße verändert wurde. Die Objekte, die nicht mehr in die erste Zele passen, werden darunter in einen neue Zeile eingefügt.

weist einen Layoutmanager an, die Größe der enthaltenen Objekte zu bestimmen, und einen neuen Layoutplan zu entwerfen. Diese Anweisung führt häufig dazu, das das Fenster in seiner Größe verändert wird, und nur noch den minimalen Platzbedarf einnimmt, der durch die enthaltenen Objekte vorgegeben ist.

Weiterhin erkennt man, daß die Objekte einen gewissen Abstand zueinander besitzen. dieser Abstand kann horizontal und vertikal getrennt eingestellt werden:

```
j_sethgap(jframe,20)
j_setvgap(jframe,20)
```

Der Defaultabstand der Objekte zueinander beträgt 10 Pixel. Wird das Fenster nun breiter gezogen, so werden alle Objekte defaultmäßig mittig zentriert. Wünscht man eine andere Ausrichtung, so kann dies mit der Anweisung:

```
j_setalign(jframe,alignment)
```

eingestellt werden. Gültige Ausrichtungen sind J_RIGHT, J_CENTER, J_LEFT, J_TOP, J_BOTTOM, J_TOPLEFT, J_TOPRIGHT, J_BOTTOMLEFT und J_BOTTOMRIGHT. Wird, wie im Beispiel, keine solche Anweisung gegeben, startet der Flowlayoutmanager mit seiner Defaulteinstellung. Diese ist bei einer horizontalen Ausrichtung J_TOPCENTER, und bei einer vertikalen Ausrichtung J_LEFT;

Wird nun das Fenter verkleinert, so verschwinden zunächst die rechten Buttons. Ein Manager kann nicht zaubern. Ist der Platzbedarf zu groß, so können einige Objekte eben nicht dargestellt werden. Anders verhält es sich jedoch, wenn des Beipielfenster zwar schmaler gemacht wird, jedoch auch höher. Dann werden die verschwundenen Button im freien Bereich wieder dargestellt.

Ein Flowlayoutmanager kennt folgende Optionen:

- Vertikale oder Horizontale Anordnung der Elemente. Die Ausrichtung wird beim Anlegen des Flowlayoutmanagers festgelegt.

  ```
  j_setflowlayout( container , J_VERTICAL)
  j_setflowlayout( container , J_HORIZONTAL)
  ```

- Ausrichtung innerhalb des Containers. Gültige Ausrichtungen sind:
  - J_RIGHT rechtbündig, Hoehe zentriert
  - J_LEFT linksbündig, Hoehe zentriert
  - J_TOP am oberen Rand, in der Beite zentriert
  - J_BOTTOM am unteren Rand, in der Beite zentriert
  - J_CENTER in der Höhe und in der Breite zentriert
  - J_TOPLEFT in der linken oberen Ecke
  - J_TOPRIGHT in der rechten oberen Ecke
  - J_BOTTOMLEFT in der linken unteren Ecke
  - J_BOTTOMRIGHT in der rechten unteren Ecke

- volle Hoehe bei horizontaler Ausrichtun, bzw. volle Breite bei vertikaler Ausrichtung. Eine Ausdehnung auf die volle Breite (Höhe) des umgebenden Containers kann mit der Funktion

```
j_setflowfill( container, J_TRUE )
j_setflowfill(container, J_FALSE )
```

ereicht werden. **J_TRUE** schaltet den Füllmodus ein, mit **J_FALSE** kann die natürliche Größe der Objekte wieder eingestellt werden.

- Eine natürliche Anordnung der Elemente kann mit der Funktion j_pack(container) erreicht werden. Alle Elemente erhalten damit Ihre natürliche Groß e und eine Anordnung die je nach Manager nebeneinander (Horizontale Anordnung) oder untereinander (vertikale Anordnung) erfolgt.

Ein umfangreiches Beispiel für einen Flowlayoutmanager ist im folgenden abgedruckt. Es beinhaltet alle oben genannten Optionen. Es beinhaltet vier Buttons, die die oben genannten Optionen anbieten. Durch Anklicken der Buttons kann umgeschaltet werden zwischen vertikaler und horizontaler Anordnung (Button "orientation"), der Anordnung der Buttons innerhalb des Frames (Button "alignment"), der "Fill" Option, und der "Pack" Option.

```
rem       Example flowsimple.bas

:
align   = j_button(jframe,"alignment")
orient  = j_button(jframe,"orientation")
fill    = j_button(jframe,"fill")
pack    = j_button(jframe,"pack")
:

while(obj<>jframe)
    obj=j_nextaction()

    if(obj = align) then

       if(alignment = J_BOTTOMRIGHT) then
           alignment = J_LEFT
       else
           if(alignment = J_BOTTOMLEFT) alignment = J_BOTTOMRIGHT
           if(alignment = J_TOPRIGHT) alignment = J_BOTTOMLEFT
           if(alignment = J_TOPLEFT) alignment = J_TOPRIGHT
           if(alignment = J_BOTTOM) alignment = J_TOPLEFT
           if(alignment = J_TOP)   alignment = J_BOTTOM
           if(alignment = J_RIGHT)  alignment = J_TOP
           if(alignment = J_CENTER) alignment = J_RIGHT
           if(alignment = J_LEFT)  alignment = J_CENTER
       endif
       j_setalign(jframe,alignment)
    endif

    if(obj = pack) j_pack(jframe)

    if(obj = fill) then
        if (dofill) then
            dofill = J_FALSE
        else
            dofill = J_TRUE
        endif
        j_setflowfill(jframe,dofill)
    endif

    if(obj = orient) then
        if(orientation = J_VERTICAL) then
            orientation=J_HORIZONTAL
        else
            orientation=J_VERTICAL
        endif
        j_setflowlayout(jframe,orientation)
```

```
        j_setalign(jframe,alignment)
        j_setflowfill(jframe,dofill)
    endif

wend
:
```

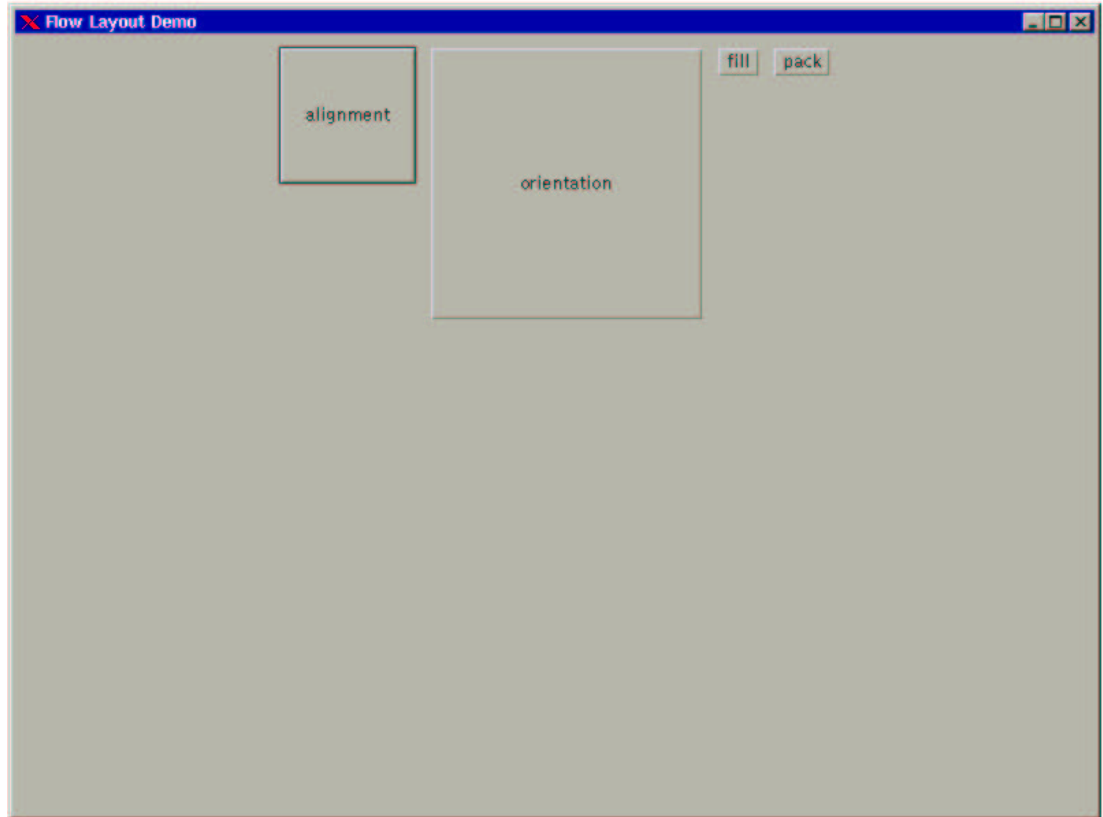


Abbildung 3.3: Demoprogramm zum Flowlayoutmanager. Die wichtigsten Funktionen sind integeriert.

3.2 Gridlayout

Ein Gridlayoutmanager ordnet alle Objekte in einem Gitter an. Die Zahl der Zeilen und Spalten ist einstellbar. Alle Objekte werden von links nach rechts zeilenweise eingefügt, und erhalten alle dieselbe Größe. Diese Größe orientiert sich an dem größten enthaltenen Objekt. Alle anderen Objekte bekommmen dann ebenfalls diese Größe zugeordnet. Das folgende Beispiel demonstriert die Funktionsweise eines Gridlayoutmanagers:

```
rem      Example gridlayout.bas
```

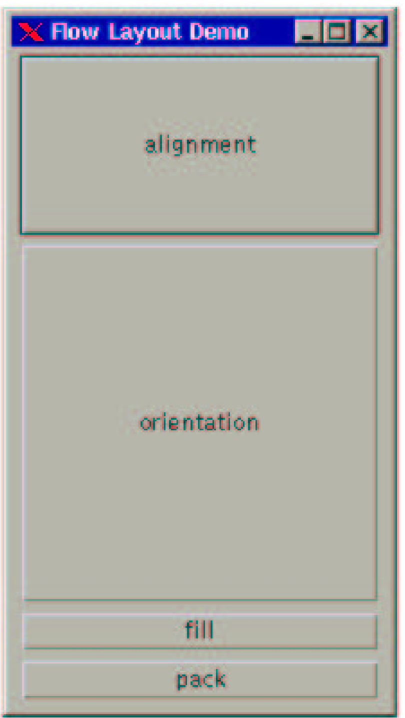


Abbildung 3.4: Die Positionierung der Objekte, nachdem die Orientierung auf Vertikal eingestellt wurde. Weiterhin wurden die Optionen PACK und FILL verwendet.

```
:
jframe    = j_frame("Grid Layout Demo")

j_setgridlayout(jframe,2,2)

button1  = j_button(jframe,"button1")
button2  = j_button(jframe,"button2")
button3  = j_button(jframe,"button3")
button4  = j_button(jframe,"button4")

j_sethgap(jframe,20)
j_setvgap(jframe,5)

j_setsize(button1,200,200)

j_pack(jframe)
j_show(jframe)
:
```

Die Funktionen sind identisch mit denen eines Flowlayoutmanagers. Der Funktionsaufruf:

```
j_setgridlayout(jframe,2,2)
```

weist dem Frame einen Gridlayoutmanager zu, der 2 Zeilen und 2 Spalten verwalten soll. Die Objekte werden anschließend von links nach rechts zeilenweise eingefügt. Werden 0 Spalten oder 0 Zeilen übergeben, so werden automatisch ausreichend Spalten bzw. Zeilen generiert, wie nötig sind, um alle Elemente aufzunehmen.

Die Funktionen:

```
j_sethgap(jframe,20)
j_setvgap(jframe,5)
```

setzen analog zum Flowlayoutmanager die horizontalen und vertikalen Abstände zwischen den Objekten. Die abschließenden Funktionen:

```
j_pack(jframe)
j_show(jframe)
```

sorgen dafür, daß der Frame gepackt und angezeigt wird. Dabei bekommen alle Objekte die Größe des größten Objektes zugeordnet. Im Beispiel gibt der Button 1 die Größe vor, da dieser explizit die Abmessungen 200:100 zugeordnet bekommt, und somit alle anderen Buttons übertrifft.

Da der Gridlayoutmanager die Stategie verfolgt, alle Objekte gleich groß darzustellen, werden bei Fenstergrößenänderungen alle enthaltenen Elemente so modifiziert, daß alle Elemenete wieder dieselbe Größe erhalten.

3.3 Borderlayout

Der Borderlayout Manager kann nur bis zu fuenf Objekte verwalten. Davon wird ein Objekt zentral in der Mitte plaziert. Die restlichen Objekte werden an jeweils einen Rand gelegt. Somit gibnt es fuef plazierungspositionen die netsprechend mit "North","East","South","West" und "Center" bezeichnet werden. Das folgende Beispiel demostriert die Verwendung eines solchen Layoutmanagers.

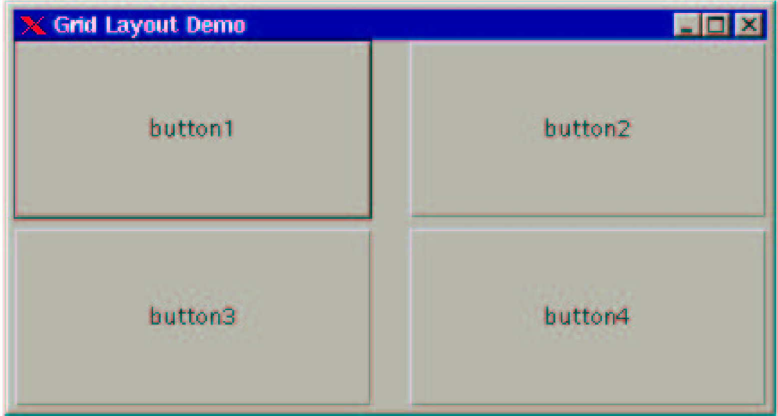


Abbildung 3.5: Demoprogramm zum Gridlayoutmanager. Alle Elemente werden in einem Gitter angeordnet und sind immer gleich groß

```
rem       Example borderlayout.bas

        :
        jframe     = j_frame("Border Layout Demo")

        j_setborderlayout(jframe)

        right  = j_button(jframe,"Right")
        left   = j_button(jframe,"Left")
        bottom = j_button(jframe,"Bottom")
        top    = j_button(jframe,"Top")
        center = j_button(jframe,"Center")

        j_setborderpos(right,J_RIGHT)
        j_setborderpos(left,J_LEFT)
        j_setborderpos(bottom,J_BOTTOM)
        j_setborderpos(top,J_TOP)

        j_show(jframe)
        :
```

Mit der Funktion

```
j_setborderlayout(jframe)
```

wird einem Behälter (Frame, Panel oder Dialog) ein Borderlayoutmanager zugeordnet. Alle weiteren Objekte, die in den Behälter eingefügt werden, werden zunächst im Zentrum plaziert. Sichtbar ist dann allerdings nur das zuletzt eingefügte Objekt. Um die anderen Objekte an ihre zugedachte Psition zu verschieben wird die Funktion:

```
j_setborderpos(right,J_RIGHT)
```

verwendet. Als mögliche Positionen sind die Konstanten: J_RIGHT, J_LEFT, J_BOTTOM, J_TOP und J_CENTER gültig. Im Beispiel wird dem Objekt Center keine weitere Position zugeordnet, da es bereits defaultmäßig im Zentrum plaziert wurde. Es ist darauf zu achten, das das Element, das im Zentrum plaziert werden soll, stets als letztes Element dem Container hinzugefügt wird.

Die (im Beispiel nicht verwendeten) Funktionen:

```
j_sethgap( container , hgap )
j_setvgap( container , vgap )
```

setzen analog zum den anderen Layoutmanager die horizontalen und vertikalen Abstände zwischen den Objekten. Auch Größenzuweisung an die enthaltenen Objekte sind möglich. Werden keine explizieten Größenzuweisungen gemacht, so verfolgt der Borderlayoutmanager die Strategie, allen Randobjekten nur den minimal benötigten Platz zuzuweisen. Der verbleibende Platz wird dem zentralen Objekt zugeordnet.

Mit der Funktion

```
j_pack(jframe)
```

wird der Frame auf den minimalen Platzbedarf zusammengepackt. Da im Beispiel auf ein packen verzichtet wurde, wird die default Größe des Frames übernommen. Wird vom Benutzter die Fentergröße verändert, so ändert sich nur die Größe des zentralen Objektes.

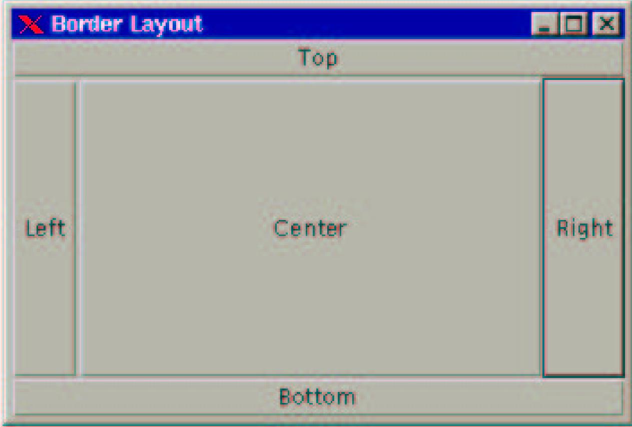


Abbildung 3.6: Demoprogramm zum Borderlayoutmanager. Die Randelemente bekommen Ihre minimale Größe zugeordnet, der Rest bleibt für das zentrale Objekt.

Kapitel 4

Die Listener

Obwohl jedes graphische Objekt über seine Objektnummer ein Event an die Applikation weiterleitet, kann es manchmal wünschenswert sein, auf weitere Events des Benutzers zu reagieren. Insbesondere Mausbewegungen und Tastatureingaben müssen häufig abgefangen werden.

Um auf solche Events zu reagieren, stehen unter japi sogenannte Listener zur Verfügung. Zur Zeit sind drei Listener implementiert. Ein Mouslistener kann unterschiedliche Mouseevents abfangen. Der Keylistener überwacht die Tastatureingaben. Der Focuslistener überwacht die graphischen Elemente und reagiert, wenn ein Element den Focus bekommt oder verliert.

An alle graphischen Objekte können nun ein oder mehrere Listener gebunden werden. So kann zB. eine Tastatureingabe oder ein Mausklick einem Objekt eindeutig zugeordnet werden. Im folgenden werden die Listener und deren Programmierung genauer vorgestellt.

4.1 Mouse Listener

Der Mouslistener dient dazu, Mausbewegungen und Mausklicks innerhalb eines graphischen Objekts der Applikation zu melden. Es können sechs verschiedene Mausoperationen abgefragt werden:

1. pressed Abfrage ob eine Maustaste gedrückt wurde.

2. released Abfrage ob eine Maustaste wieder losgelassen wurde.

3. draggend Abfrage ob die Maus bei gedrückter Maustaste bewegt wurde.

4. entered Abfrage ob der Mauszeiger das Objekt betritt.

5. exited Abfrage ob der Mauszeiger das Objekt verläßt.

6. moved Abfrage ob sich die Maus innerhalb des Objekts bewegt.

Im folgenden Beispiel sind alle sechs Fälle realisiert:

```
rem       Example mouselistener.bas

:
canvas1  = j_canvas(jframe,200,200)
canvas2  = j_canvas(jframe,200,200)

pressed  = j_mouselistener(canvas1,J_PRESSED)
dragged  = j_mouselistener(canvas1,J_DRAGGED)
released = j_mouselistener(canvas1,J_RELEASED)
entered  = j_mouselistener(canvas2,J_ENTERERD)
moved    = j_mouselistener(canvas2,J_MOVED)
exited   = j_mouselistener(canvas2,J_EXITED)
:

while(obj<>jframe)

    obj=j_nextaction()

    if(obj = pressed) then
        startx=j_getmousex(pressed)
        starty=j_getmousey(pressed)
    endif

    if(obj = dragged) then
        x = j_getmousex(dragged)
        y = j_getmousey(dragged)
        j_drawrect(canvas1,startx,starty,x-startx,y-starty)
    endif

    if(obj = released) then
        x = j_getmousex(released)
        y = j_getmousey(released)
        j_drawrect(canvas1,startx,starty,x-startx,y-starty)
    endif

    if(obj = entered) then
        startx=j_getmousex(entered)
        starty=j_getmousey(entered)
    endif

    if(obj = moved) then
        x = j_getmousex(moved)
        y = j_getmousey(moved)
        j_drawline(canvas2,startx,starty,x,y)
        startx=x
        starty=y
    endif

    if(obj = exited) then
        x = j_getmousex(exited)
        y = j_getmousey(exited)
        j_drawline(canvas2,startx,starty,x,y)
    endif

wend
:
```

Im Beispiel werden zunächst zwei Zeichenfelder "canvas1" und "canvas2" erzeugt. An jedes Zeichenfeld werden nun drei Mouselistener gebunden, die jeweils eine eigene Eventnummer erhalten. An "canvas1" wird je ein Listener für "pressed","released" und "dragged" gebunden. In diesem Zeichenfeld sollen bei grdrückter Maustaste Rechtecke aufgezogen werden. Wird nun in diesem Canvas eine Maustaste gedrückt, so wird über die Eventnummer des "pressed" Listener die Applikation benachrichtigt:

```
if(obj = pressed) then
    startx=j_getmousex(pressed)
    starty=j_getmousey(pressed)
endif
```

Mit der Funktionen ***j_getmousex()*** bzw. ***j_getmousey()*** kann die Position der Mouse ermittelt werden, an der das entsprechende Event stattfand. Im Beispiel dient diese Position als Startecke für ein Rechteck.

Wird nun bei gedrückter Taste die Maus bewegt, so liefert die Eventnummer des "dragged" Listener ein entsprechendes Event,

```
if(obj = dragged) then
    x = j_getmousex(dragged)
    y = j_getmousey(dragged)
    j_drawrect(canvas1,startx,starty,x-startx,y-starty)
endif
```

und es wird permanent ein Rechteck von der Startpostion bis zur Aktuellen Postion eingezeichnet. Wird die Maustaste losgelassen, so wird über den "realeased" Listener ein Event weitergereicht, und das abschließende Rechteck wird eingezeichnet. Ein erneutes Drücken der Maustaste wählt einen neuen Startpunkt aus.

```
if(obj = released) then
    x = j_getmousex(released)
    y = j_getmousey(released)
    j_drawrect(canvas1,startx,starty,x-startx,y-starty)
endif
```

Im zweiten Canvas werden alle Mausbewegungen nachgezeichnet. Betritt der Mauszeigen dieses Fenter, so wird vom "entered" Listener ein Event gemeldet:

```
if(obj = entered) then
    startx=j_getmousex(entered)
    starty=j_getmousey(entered)
endif
```

und die Eintrittsposition wird zwischengespeichert. Jede Bewegung der Maus erzeugt nun ein "move" Event:

```
if(obj = moved) then
    x = j_getmousex(moved)
    y = j_getmousey(moved)
    j_drawline(canvas2,startx,starty,x,y)
    startx=x
    starty=y
endif
```

und es wird eine Linie zwischen der letzten Position der Maus und der aktuellen Position eingezeichnet. Abschließend wird die aktuelle Position als letzte Position gespeichert. Verläst die Maus das Fenster, so wird mit Hilfe des "exited" Listener der Austrittsort ermittelt, und eine letzte Linie dorthin gezeichnet:

```
if(obj = exited) then
    x = j_getmousex(exited)
    y = j_getmousey(exited)
    j_drawline(canvas2,startx,starty,x,y)
endif
```

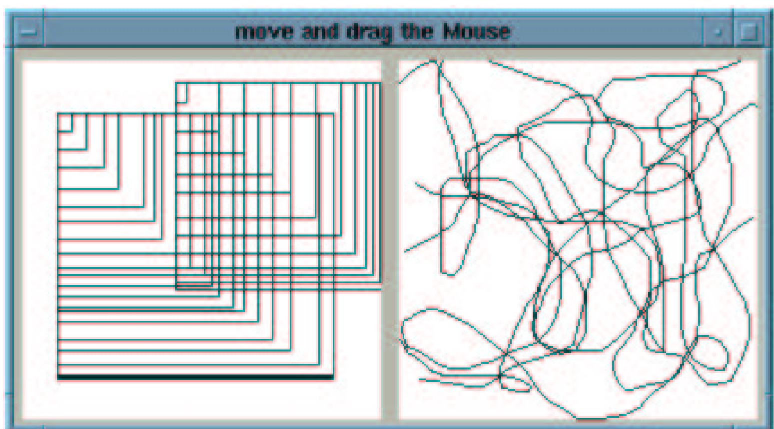


Abbildung 4.1: Der Mouselistener in Action. Im linken Canvas werden Rechtecke mit einer gedrückten Mousetaste aufgezogen. Im rechten Canvas werden alle Mousebewegungen nachgezeichnet.

4.2 Key Listener

Ein Keylistener überwacht die Tastatur auf eine Eingabe durch den Benutzer. Wiederum kann an jedes graphische Objekt ein Keylistener gebunden werden, der dann einen Event meldet, wenn innerhalb dieses Objektes eine Taste gedrückt wird. Über zwei weitere Funktionen kann dann der Keycode der Taste oder der entsprechende ASCII Code abgefragt werden. Das folgende Beispiel demonstriert die Funktionsweise:

```
rem       Example keylistener.bas

:
keylst = j_keylistener(jframe)

:
while(obj<>jframe)
    obj = j_nextaction()

    if(obj = keylst) then
       instr$ = "Keycode:" + str$(j_getkeycode(keylst))
        instr$ = instr$+ "  Ascii:"+str$(j_getkeychar(keylst))
        instr$ = instr$+ "  Char:"+chr$(j_getkeychar(keylst))
        j_settext(jframe,instr$)
    endif

wend
:
```

Zunächst wird mit der Funktion:

```
keylst = j_keylistener(jframe)
```

ein Keylistener erzeugt, und an den Frame gebunden. Wird nun innerhalb des Frames eine Taste gedrückt, so meldet der Keylistener einen Event. Mit den beiden Funktionen:

```
j_getkeycode(keylst)
j_getkeychar(keylst)
```

kann jederzeit die letzte gedrückte Taste abgefragt werden. Während ***j_keychar()*** den ASCII Code zurückliefert, kann mit der Funktion ***j_keycode()*** der Tastaurcode einer Taste abgefragt werden. So können auch ASCII-lose Tasten wie Funktionstasten und Sondertasten ermittelt werden[1].

Abbildung 4.2: Beispielprogramm für einen Keylistener. Jede gedrückte Taste wird in der Kopfzeile der Applikation vermerkt. Es erfolgt die Ausgabe des Tastaturcodes und des ASCII Codes der Taste. Im Beispiel nach dem Drücken der Taste "g".

4.3 Focus Listener

Ein Focuslistener überwacht ein Objekt, ob es den Focus verliert oder bekommt. Mit entsprechenden Funktionen kann bei Bedarf einem Objekt der Focus zurückgegeben werden. Das Objekt, das den Focus besitzt, bekommt die Eingaben der Tastatur zugeordnet. Daher ist ein Focuslistener oft in Verbindung mit einem Keylistener zu verwenden. Das folgende Beispiel sorgt dafür, das der Dritte Button niemals den Focus bekommt, und somit auch nie durch die Lerrtaste aktiviert werden kann:

```
rem       Example focuslistener.bas

:
button1  = j_button(jframe,"Button 1")
button2  = j_button(jframe,"Button 2")
button3  = j_button(jframe,"Button 3")

focuslst = j_focuslistener(button3)

:
while(obj<>jframe)
    obj =  j_nextaction()

    if(obj = focuslst) then
        if(j_hasfocus(focuslst) = J_TRUE) then
            j_setfocus(button2)
        endif
    endif
wend
:
```

Nach dem Erzeugen der drei Buttons wird zunächst ein Focislistener an den dritten Button gebunden:

```
focuslst=j_focuslistener(button3)
```

[1] Offen ist zur Zeit die Frage, ob der Keycode genormt ist. Liefern alle Tastaturen denselben Keycode für z.B. F1 zurück?

Die Focus liefern sowohl bei Verlust, als auch bei Erlangen des Focus mit einem Event. Ob ein Objekt nun den Focus hat oder nicht, ist mit der Funktion

```
j_hasfocus(focuslst)
```

zu erfragen. Die Funktion liefert J_TRUE zurück, wenn das Objekt den Focus hat, und bei Verlust entsprechend J_FALSE. Mit der Prozedur

```
j_setfocus(button2)
```

kann einem Objekt der Focus aufgezwungen werden. Im Beispielprogramm verhalten sich die beiden ersten Button zunächst ganz normal. Wird einer dieser Button angeklickt, so erhält er auch den Focus, und kann dann auch mit der Leertaste "gedrückt" werden. Wird jedoch der dritte Button angeklickt, so wird diesem sofort der Focus entzogen, und dem mittleren Button zugeordnet. Somit kann der dritte Button niemals mit der Tastatur bedient werden.

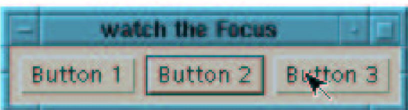


Abbildung 4.3: Beispielprogramm für den Einsatz eines Focuslistener. Sobald der dritte Button gedrückt wurde, wird der Focus sofort den mittleren Button zugeordnet.

Kapitel 5

Grafik in JAPI

Die graphischen Funktionen der JAPILIB umfassen mehrere Funktionsgruppen, die in den folgenden Kapiteln ausführlich vorgestellt werden. Bei diesen Funktionsgruppen handelt es sich um Funktionen zur:

- Farbauswahl
- Darstellung elementarer graphischer Primitiven
- Fonteinstelungen
- Cursorwahl
- Bildbearbeitung

5.1 Farben

Die Funktionen zur Farbauswahl wurden bereits in den Beispielen vorgestellt, und dort auch benutzt. Es exitieren vier Funktionen zur Farbwahl, je zwei zum Einstellen der Hintergrund- und Vordergrundfarbe. Zur Wahl steht jeweils eine Funktion zur Auswahl eines vordefinierten Farbwertes, bzw. die direkte Angabe eines RGB Wertes:

```
j_setcolor( obj , red , green , blue )
j_setcolorbg( obj , red , green , blue )
j_setnamedcolor( obj , farbe )
j_setnamedcolorbg( obj , farbe )
```

Die beiden ersten Funktionen setzen die Farben für Vordergrund und Hintergrund durch RGB Werte. Jeder Kanal kann einen Wert von 0 bis 255 annehmen. Die nächsten beiden Funktionen setzen vordefinierte Farben. Die Variable "farbe" kann dabei einen Wert von 0 bis 15 annehmen. Diese 16 Farben entsprechen dem alten CGA Farbmodell und sind wie folgt zugeordnet:

```
J_BLACK         0
J_WHITE         1
J_RED           2
J_GREEN         3
J_BLUE          4
J_CYAN          5
```

```
J_MAGENTA              6
J_YELLOW               7
J_ORANGE               8
J_GREEN_YELLOW         9
J_GREEN_CYAN          10
J_BLUE_CYAN           11
J_BLUE_MAGENTA        12
J_RED_MAGENTA         13
J_DARK_GRAY           14
J_LIGHT_GRAY          15
```

Die Angabe eines RGB Wertes ist unabhängig von der tatsächlichen Farbtiefe des Bildschirms. Die Umrechnung von der übergebenen 24 Bit Farbtiefe auf die vorhandene Farbtiefe erfolgt automatisch. Der JAPI Kernel wählt dann die jeweils "ähnlichste" Farbe aus.

Da diese Funktionen in den vorangegangenen Beispielen bereits ausführlich benutzt und beschrieben wurden, erübrigt sich an dieser Stelle eigentlich ein Beispielprogramm. Wir wollen stattdessen die Gelegenheit nutzen, hier die Japilib durch ein selbsgeschriebene Farbdialog Element zu erweitern.

Eine Farbauswahl Dalogbox (auch Colorpicker genannt) soll die Möglichkeit bieten über drei Farbregler eine Farbe einzustellen, und dessen RGB Werte an das aufrufende Programm zurückliefern. Der Aufruf der Dialogbox sollte demnach wie folgt erfolgen:

```
j_colorpicker(jframe,rgb())
```

Neben der Farbeinstellung sollte die Dialogbox folgende Eigenschaften haben:

- Sie sollte einen "OK" Button und einen "Cancel" Button besitzen.
- Die Rückgabe der Funktion sollte **J_TRUE** sein, falls der "OK" Button gedrückt wurde, bei "Cancel" soll **J_FALSE** zurückgegeben werden.
- Die Werte die in den übergebenen Variablen r,g und b stehen, sollten beim Funktionsaufruf dargestellt werden.
- In der Dialogbox sollte ein Element zur Darstellung der gewählten Farbe vorhanden sein. Dieses Element sollte mit der Dialogbox größenänderlich sein.
- Die RGB Werte sollen zudem als Zahlenwerte dargestellt werden.
- Die Colordialogbox soll ihre eigenen Events verwalten.
- Sie sollte mittig im Frame erscheinen.

Aus diesen Forderungen ergibt sich folgendes Layout:

Die Dialogbox erhält einen Borderlayoutmanager, der im Zentrum einen Canvas zu Farbdarstellung enthählt. Die RGB Scollbars werden in den linken Bereich des Borderlayouts positioniert. Dieser Bereich erhält nun ebenfalls einen Borderlayoutmanager. In dessen Kopfteil werden drei Labels, die die RGB Zahlenwerte darstellen, in einem Gridlayoutmanager positioniert. In den zentralen Bereich dieses zweiten Borderlayoutmanager werden die Scrollbars positioniert. Auch hierbei wird wiederum ein Gridlayout verwendet. Die beiden Button "OK" und "Cancel" werden in einem Flowlayoutmanager im Fussbereich der äußeren Borderlayout angeordnet.

Durch diese Anordnung hat das Labelfeld immer die gleiche Größe, und alle Label sind gleich groß. Das Scrollbarfeld hat eine konstante Breite aber immer die volle Höhe des

Dialograhmens. Die Button liegen immer zentriert im Fußbereich. Der verbleibende Rest wird komplett dem Canvas zugeordnet.

Hier zunächst das komplette Listing der Funktion ***j_colordialog()***

```
rem
rem
rem     Funktion Colorpicker( container , rgb() )
rem
rem

sub j_colorpicker(jframe,rgb())

j_disable(jframe)

dialog = j_dialog(jframe,"Colorpicker")
j_setnamedcolorbg(dialog,J_WHITE)
j_setborderlayout(dialog)
panel1=j_panel(dialog)
j_setborderpos(panel1,J_LEFT)
j_setborderlayout(panel1)

panel2=j_panel(panel1)
j_setborderpos(panel2,J_TOP)
j_setgridlayout(panel2,0,3)
rjlabel=j_label(panel2,"255")
gjlabel=j_label(panel2,"255")
bjlabel=j_label(panel2,"255")

panel2=j_panel(panel1)
j_setgridlayout(panel2,0,3)
j_sethgap(panel2,20)
rscroll=j_vscrollbar(panel2)
gscroll=j_vscrollbar(panel2)
bscroll=j_vscrollbar(panel2)
j_setmax(rscroll,265)
j_setmax(gscroll,265)
j_setmax(bscroll,265)
j_setnamedcolorbg(rscroll,J_RED)
j_setnamedcolorbg(gscroll,J_GREEN)
j_setnamedcolorbg(bscroll,J_BLUE)
j_setvalue(rscroll,rgb(0))
j_setvalue(gscroll,rgb(1))
j_setvalue(bscroll,rgb(2))

panel1=j_panel(dialog)
j_setborderpos(panel1,J_BOTTOM)
j_setflowlayout(panel1,J_HORIZONTAL)
ok = j_button(panel1,".   OK   .")
cancel = j_button(panel1,"Cancel")

canvas=j_canvas(dialog,200,200)

j_pack(dialog)
xpos = j_getxpos(jframe) + j_getwidth(jframe)/2-j_getwidth(dialog)/2
ypos = j_getypos(jframe) + j_getheight(jframe)/2-j_getheight(dialog)/2
j_setpos(dialog,xpos,ypos)
j_show(dialog)

obj = 0
retval = J_FALSE
while ((obj <> cancel) and (obj <> dialog) and (obj <> ok))
    lr = j_getvalue(rscroll)
    lg = j_getvalue(gscroll)
    lb = j_getvalue(bscroll)

    j_setcolorbg(canvas,lr,lg,lb)
    instr$ = str$(lr)
    j_settext(rjlabel,instr$)
    instr$ = str$(lg)
    j_settext(gjlabel,instr$)
```

```
        instr$ = str$(lb)
        j_settext(bjlabel,instr$)

        obj = j_nextaction()

        if(obj=ok) then
            rgb(0) = lr
            rgb(1) = lg
            rgb(2) = lb
            retval = J_TRUE
        endif
    wend

    j_dispose(dialog)
    j_enable(jframe)
    return retval

    end sub
```

Der Aufruf der Funktion:

```
j_disable(jframe)
```

sorgt dafür, das vom aufrufenden Programm keine Events mehr kommen können. Dies ermöglicht uns, in der Funktion eine eigene Eventloop zu programmieren. Der Aufbau des Layout dürfte selbsterklärend sein. Die Maximalwerte der Scrollbars werden auf den wert 265 gesetzt. Dadurch ergibt sich, abzüglich der Hoehe des Schiebereglers von 10, ein maximal erreichbarer Wert von 255. Damit die Dialogbox immer mittig im aufrufenden Frame erscheint, wird die Position der Dialogbox mit folgenden Funktionsaufrufen gesetzt:

```
xpos = j_getxpos(jframe) + j_getwidth(jframe)/2-j_getwidth(dialog)/2
ypos = j_getypos(jframe) + j_getheight(jframe)/2-j_getheight(dialog)/2
j_setpos(dialog,xpos,ypos)
```

Dabei wird zunächst die Schirmposition des Frames ermittelt. Als der Hälfte der Breite von Frame und Colordialogbox kann die horizontale Position ermittelt werden. Die vertikale Position errechnet suich analog.

In der Eventloop wird bei jedem Event die Werte der drei Scrollbars ermittelt und die Farbe des Canvas sowie die Inhalte der Labels neu gesetzt. Diese Programmierung ist sicherlich ineffizient un könnte durch das Abfangen der Scrollbar Events verbessert werden. Da der Overhead jedoch nicht so groß ist, soll dies hier genügen.

Abschließend wird der Colordialog entfernt, und der aufrufende Frame wieder enabled:

```
j_dispose(dialog)
j_enable(jframe)
```

Das folgende Beispiel nutzt den Colordialog, um seine Hintergrundfarbe einzustellen (Abbildung 5.1):

```
rem       Example text.bas

dim rgb(2)
:
jframe   = j_frame("Colorpicker Demo")
menubar = j_menubar(jframe)
file    = j_menu(menubar,"File")
jcolor   = j_menuitem(file,"Color")
quit    = j_menuitem(file,"Quit")

j_setcolorbg(jframe,rgb(0),rgb(1),rgb(2))
j_show(jframe)
```

```
obj=0
while((obj <> jframe) and (obj <> quit))
    obj =  j_nextaction()

    if(obj = jcolor) then
        if(j_colorpicker(jframe,rgb())=J_TRUE) then
           j_setcolorbg(jframe,rgb(0),rgb(1),rgb(2))
        endif
    endif
wend
:
```

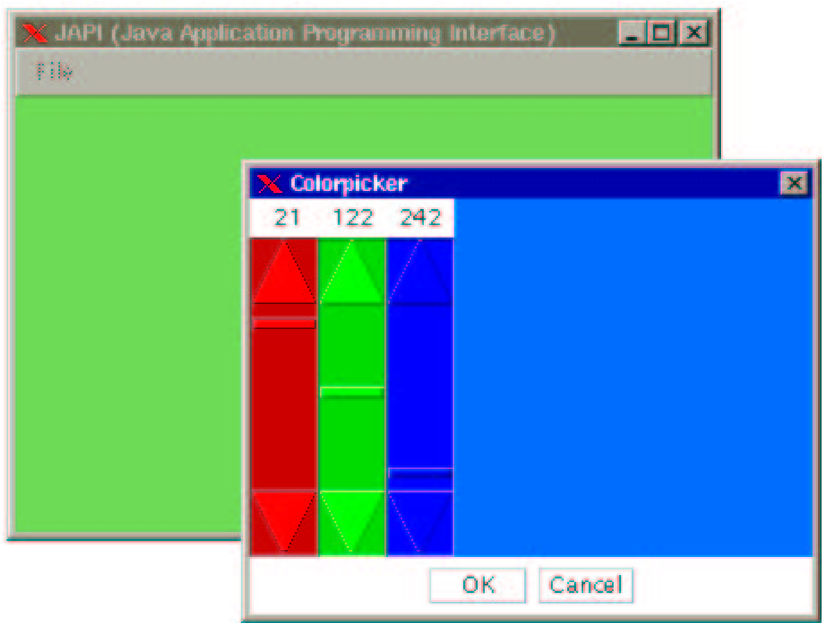


Abbildung 5.1: Der Colordialog im Einsatz

5.2 Grafikbefehle

Die Japilib bietet folgende Graphic Primitiven, die alle in der momentan gesetzten Vordergrundfarbe ausgeführt werden. Als Zielobjekte sind prinzipiel alle Componenten verwendbar. Da jedoch nur der Canvas ein sg. Dubble Buffering besitzt, ist der Einsatz dieser Funktionen nur innerhalb eines Canvas empfehlenswert.

`j_drawpixel(component,x,y)`

zeichnet einen Pixel in der Componente an der Position (x,y).

`j_drawline(component,x1,y1,x2,y2)`

zeichnet eine Linie von Punkt (x1,y1) nach Punkt (x2,y2).

```
j_drawpolyline(component,nval,x(),y())
```

Zeichnet eine Polyline die aus den Punkten der Arrays *x* und *y* definiert wird. Die Arrays müssen mindestens *nval* Elemente besitzen.

```
j_drawpolygon(component,nval,x(),y())
j_fillpolygon(component,nval,x(),y())
```

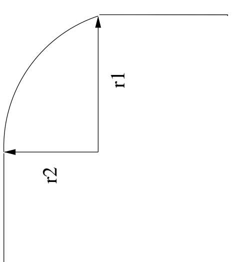
Zeichnet ein (gefülltes) Polygon die aus den Punkten der Arrays x und y definiert wird. Die Arrays müssen mindestens *nval* Elemente besitzen. Im Gegensatz zur Polyline werden hier die Endpunkte zusätzlich miteinander verbunden.

```
j_drawrect(component,x,y,width,height)
j_fillrect(component,x,y,width,height)
```

Zeichnet ein (gefülltes) Rechteck mit der oberen linken Ecke an der Position (x,y). Da Rechteck besitzt die Breite *width* und die Höhe *height*.

```
j_drawroundrect(component,x,y,width,height,r1,r2)
j_fillroundrect(component,x,y,width,height,r1,r2)
```

Zeichnet ein (gefülltes) Rechteck mit der oberen linken Ecke an der Position (x,y). Da Rechteck besitzt die Breite *width* und die Höhe *height*. Das Recheck besitzt abgerundete Ecken, die durch die Parameter *r1* und *r2* definiert werden. *r1* bestimmt dabei den horizontalen Radius der Bögen, *r2* den vertikalen Radius.

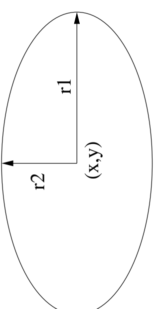



```
j_drawcircle(omponent,x,y,r)
j_fillcircle(omponent,x,y,r)
```

Zeichnet einen (gefüllten) Kreis um den Punkt (x,y) mit dem Radius *r*.

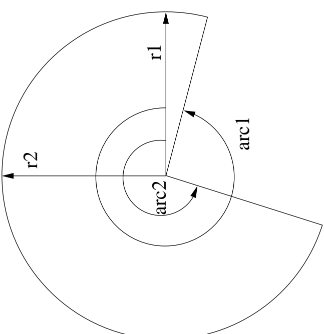
```
j_drawoval(component,x,y,r1,r2)
j_filloval(component,x,y,r1,r2)
```

Zeichnet ein (gefülltes) Oval, um den Punkt (x,y) mit den Hauptradien *r1* und *r2*.




```
j_drawarc(component,x,y,r1,r2,arc1,arc2)
j_fillarc(component,x,y,r1,r2,arc1,arc2)
```

Zeichnet einen (gefüllen) Kreisbogen um den Punkt (x,y) mit den Hauptradien *r1* und *r2*. Der Kreisbogen begint im Startwinkel *arc1* und endet am Stopwinkel *arc2*. Die Winkel sind im Bogenmaß anzugeben.



`j_drawstring(component,x,y,str)`

Zeichnet den String *str* an die Position (x,y). Die Position bezeichnet den Anfang derGrundlinie des Strings (linke untere Ecke).

Der folgende Auszug aus einem Beipielprogramm demonstriert diese Graphikprimitiven (siehe Abbildung 5.2):

```
rem       Example graphic.bas

:
j_translate(canvas,10,10)
j_drawline(canvas,10,10,90,90)
j_drawstring(canvas,0,105,"Line")
```

```
j_translate(canvas,100,0)
j_drawpolygon(canvas,10,x(),y())
j_drawstring(canvas,0,105,"Polygon")

j_translate(canvas,100,0)
j_drawrect(canvas,10,10,80,80)
j_drawstring(canvas,0,105,"Rectangle")

j_translate(canvas,100,0)
j_drawroundrect(canvas,10,10,80,80,20,20)
j_drawstring(canvas,0,105,"RoundRect")

j_translate(canvas,100,0)
j_drawcircle(canvas,50,50,40)
j_drawstring(canvas,0,105,"Circle")

j_translate(canvas,100,0)
j_drawoval(canvas,50,50,40,20)
j_drawstring(canvas,0,105,"Oval")

j_translate(canvas,100,0)
j_drawarc(canvas,50,50,40,30,113,210)
j_drawstring(canvas,0,105,"Arc")


rem     Filled

j_translate(canvas,-600,100)
j_drawpolyline(canvas,10,x(),y())
j_drawstring(canvas,0,105,"Polyline")

j_translate(canvas,100,0)
j_fillpolygon(canvas,10,x(),y())
j_drawstring(canvas,0,105,"FillPolygon")

j_translate(canvas,100,0)
j_fillrect(canvas,10,10,80,80)
j_drawstring(canvas,0,105,"FillRectangle")

j_translate(canvas,100,0)
j_fillroundrect(canvas,10,10,80,80,20,20)
j_drawstring(canvas,0,105,"FillRoundRect")

j_translate(canvas,100,0)
j_fillcircle(canvas,50,50,40)
j_drawstring(canvas,0,105,"FillCircle")

j_translate(canvas,100,0)
j_filloval(canvas,50,50,40,20)
j_drawstring(canvas,0,105,"FillOval")

j_translate(canvas,100,0)
j_fillarc(canvas,50,50,40,30,113,210)
j_drawstring(canvas,0,105,"FillArc")
:
```

Das Beispiel benutzt zudem eine weitere Funktion ***j_translate()*** mit der sich der Koordinatenursprung verschieben läßt. Diese Funktion gehört zu einer weiteren Gruppe von Funktionen zu einer erweiterten Graphikausgabe:

`j_translate(component,x,y)`

> Diese Funktion verschiebt den Koordinatenursprung um x Pixel in der Horizontalen und y Pixel in der Vertikalen. Alle folgenden Graphikbefehle beziehen sich nun auf den neuen Koordinatenursprung.

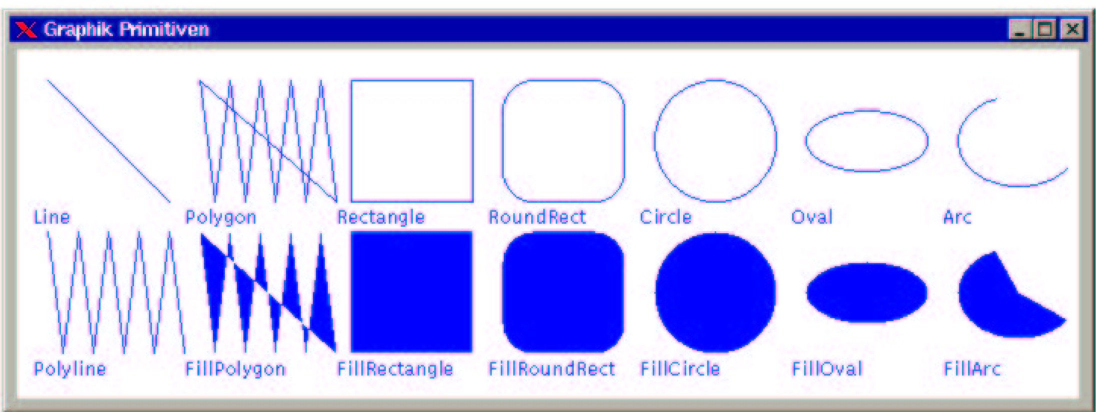


Abbildung 5.2: Alle Graphic Primitiven der Japilib

`j_cliprect(component,x,y,width,height)`

> Diese Funktion setzt ein Clipping Rechteck an die Position (x,y) mit der Breite *width* und der Hoehe *height*. Das Clipping Rechteck bewirkt, daß im folgenden nur noch in diesem Rechteck graphische Ausgaben erfolgen. Alles was über die Grenzen des Rechecks hinausragt, wird geclipt (abgeschnitten).

`j_setxor(component, bool)`

> Diese Funktion schalten den XOR Modus beim Zeichen ein (bool = J_TRUE) und aus (bool = J_FALSE). Ist der XOR Modus gesetzt, so heben sich zwei identische Zeichenbefehle wieder auf.

Der folgende Auszug aus einer Eventloop demonstriert, wie man mit dem XOR Modus ein Rubberband realisieren kann. Ein Rubberband ist ein variables Rechteck, das man mit gerdrückter Maustaste aufziehen kann. Das Beispiel enthält einen Canvas mit einem Mousepressed– und einem Mousedragged– Listener.

```
if(obj = pressed) then
    j_setxor(canvas, J_TRUE )
    j_drawrect(canvas,startx,starty,x-startx,y-starty)
    j_getmousepos(pressed,x,y)
    startx = j_getmousex(pressed)
    starty = j_getmousey(pressed)
    x = startx
    y = starty
    j_setxor(canvas, J_FALSE )
endif

if(obj == dragged) then
    j_setxor(canvas, J_TRUE )
    j_drawrect(canvas,startx,starty,x-startx,y-starty)
    x = j_getmousex(dragged)
    y = j_getmousey(dragged)
    j_drawrect(canvas,startx,starty,x-startx,y-starty)
    j_setxor(canvas, J_FALSE )
endif
```

Wird die Maus gedrückt (pressed), so wird der XOR Modus eingeschaltet. Dadurch kann ein evtl. bereits vorhandenes altes Rechteck durch Überzeichnen gelöscht werden Die Position der Maus wird gespeichert. Wird die Maus bewegt (dragged), so wird ebenfalls zunächst der XOR Modus gesetzt. Ebenso wird als nächstes ein evtl. vorhandenes Rechteck gelöscht. Danach wird die neue Position der Maus ermittelt und ein neues Rechteck gezeichnet. Durch Löschen des XOR Modus kann nun normal weitergezeichnen werden.

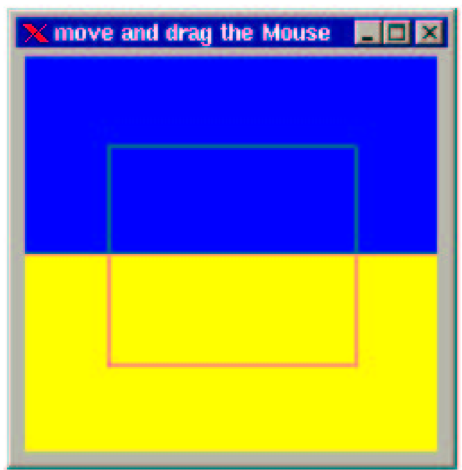


Abbildung 5.3: Mit dem XOR Modus ein Rubberband realisieren.

5.3 Fonts

Es gibt 5 verschiedene Fonttypen innerhalb der Japilib [1]. Dies sind:

1. Courier (Monospaced)
2. Helvetia (Sans Serif)
3. Times (Serif)
4. Dialoginput
5. Dialogoutput

Neben dem Fonttyp sind weitere Attribute vorhanden. Ein Font kann Normal, Fett, Kursiv und Fett-Kursiv dargestellt werden. Auch die Fontgröße muss angegeben werden. Somit besitzt die Funktion zur Auswahl eines Fonts 4 Parameter:

```
j_setfont(component,typ,style,height)
```

Der Parameter *typ* kann dabei folgende Werte annehmen:

[1] Es kann vorkommen, daß auf einer Plattform nicht alle Fonttypen vorhanden sind. Ineinem sochen Fall wird automatisch ein ähnlicher Font ausgewählt

- J_COURIER
- J_HELVETIA
- J_TIMES
- J_DIALOGIN
- J_DIALOGOUT

Für den Parameter *style* sind folgende Werte möglich:

- J_PLAIN
- J_BOLD
- J_ITALIC
- J_BOLD + J_ITALIC

Die Größe *height* wird als Integerzahl übergeben. Sie kann prinzipiel beliebige positive Zahlen annehmen.

Es ist jedoch nicht immer nötig einen Font komplett zu spezifiezieren. Die Attribute eines Fonts können auch getrennt manipuliert werden. Dazu stellt die Japilib folgende Funktionen zur Verfügung:

```
j_setfontname(component, typ)
j_setfontstyle(component, style)
j_setfontsize(component, height)
```

Die Parameter entsprechnen dabei genau den drei Fontparameter der ***j_setfont()*** Funktion.

Das folgende Beipiel realisiert eine dynamische Fontauswahl mit Preview–Funktion:

```
rem       Example font.bas

:
jframe     = j_frame("Font Demo")
menubar   = j_menubar(jframe)
file      = j_menu(menubar,"File")
quit      = j_menuitem(file,"Quit")

font      = j_menu(menubar,"Font")
courier   = j_menuitem(font,"Courier")
helvetia  = j_menuitem(font,"Helvetia")
times     = j_menuitem(font,"Times")
dialogin  = j_menuitem(font,"DialogIn")
dialogout = j_menuitem(font,"DialogOut")

style     = j_menu(menubar,"Style")
normal    = j_menuitem(style,"Plain")
bold      = j_checkmenuitem(style,"Bold")
italic    = j_checkmenuitem(style,"Italic")

size      = j_menu(menubar,"Size")
f10       = j_menuitem(size,"10 pt")
f12       = j_menuitem(size,"12 pt")
f14       = j_menuitem(size,"14 pt")
f18       = j_menuitem(size,"18 pt")

jlabel     = j_label(jframe,"abcdefghijklmnopqrstuvwxyz")
j_setsize(jlabel,400,120)
```

```
j_setpos(jlabel,5,60)

fontstyle = J_PLAIN

j_setfont(jframe,J_HELVETIA,J_PLAIN,12)

j_pack(jframe)
j_show(jframe)

obj=0
while((obj <> jframe) and (obj <> quit))
   obj=j_nextaction()

    if(obj = courier)   j_setfontname(jlabel,J_COURIER)
    if(obj = helvetia)  j_setfontname(jlabel,J_HELVETIA)
    if(obj = times)     j_setfontname(jlabel,J_TIMES)
    if(obj = dialogin)  j_setfontname(jlabel,J_DIALOGIN)
    if(obj = dialogout) j_setfontname(jlabel,J_DIALOGOUT)

    if(obj = normal) then
        fontstyle=J_PLAIN
        j_setstate(bold, J_FALSE )
        j_setstate(italic, J_FALSE )
        j_setfontstyle(jlabel,fontstyle)
    endif

    if(obj = bold) then
       if(j_getstate(bold)=J_TRUE)  then
          fontstyle = fontstyle + J_BOLD
       else
          fontstyle = fontstyle - J_BOLD
       endif
       j_setfontstyle(jlabel,fontstyle)
    endif

    if(obj = italic) then
       if(j_getstate(italic)=J_TRUE)  then
          fontstyle = fontstyle + J_ITALIC
       else
          fontstyle = fontstyle - J_ITALIC
       endif
       j_setfontstyle(jlabel,fontstyle)
    endif

    if(obj = f10)   j_setfontsize(jlabel,10)
    if(obj = f12)   j_setfontsize(jlabel,11)
    if(obj = f14)   j_setfontsize(jlabel,13)
    if(obj = f18)   j_setfontsize(jlabel,18)

    instr$ = " StringWidth = "
    instr$ = instr$ + str$(j_getstringwidth(jlabel,"abcdefghijklmnopqrstuvwxyz"))
    instr$ = instr$ + " FontHeight = "+str$(j_getfontheight(jlabel))
    instr$ = instr$ + " FontAscent = "+str$(j_getfontascent(jlabel))
    j_settext(jframe,instr$)

wend
:
```

Das Beispiel bietet in der Menuleiste die Manipulationen des Fonttyps, des Fontstils, und der Fontgröße. Sobald ein Parameter verändert wurde, wird im Fenter ein String mit den entsprechenden Attributen dargestellt (siehe Abbildung 5.4).

Gleichzeitig wird in der Kopfzeile des Frames Größeninformationen über den String ausgegeben. Die beiden ersten Ausgabewerte definieren die Größe des Rechtecks, das den String umschließt. Der Ascent ist die Höhe des Fonts, gemessen von der Grundlinie bis zur Oberlängenlinie (Siehe Abbildung 5.5).

Die vorhandenen Funktionen zur Bestimmung von Stingabmessungen sind:

Abbildung 5.4: Eine dynamische Fontauswahl mit Previewfunktion. Dargestellt wird der String in Times, fett und kursiv mit einer Fontgröße von 18 Pixeln.

```
j_getstringwidth(component,instr$)
j_getfontheight(component)
j_getfontascent(component)
```

Die Funktion ***j_stringwidth(component,str)*** liefert die Breite des Strings *str* zurück, die dieser haben würde, wenn er in der Komponente *component* dargestellt werden würde. Die Funktion ***j_fontheight(component)*** liefert die Gesamthöhe des Fonts zurück, die der Font der Komponente *component* hat. Analog liefert ***j_fontascent(component)*** den Ascent des Fonts zurück.

5.4 Cursor

Die Japilib bietet 14 verschiedene Cursortypen. Mit der Funktion

```
j_setcursor(component, type)
```

kann an eine Komponente ein bestimmter Cursortyp gebunden werden. Es ist allerdings zu beachten, daß auf den unterschiedlichen Plattformen die Cursor auch verschiedenes Aussehen besitzen. Die folgende Abbildung zeigt die vorhandenen Cursortypen mit dem jeweiligen JAPI Namen:

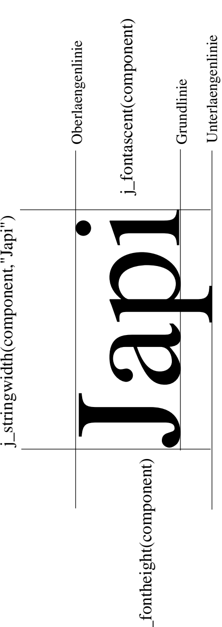


Abbildung 5.5: Die Metric eines Fonts.

JAPI TYP	Motif	Windows
J_DEFAULT_CURSOR		
J_CROSSHAIR_CURSOR		
J_TEXT_CURSOR		
J_WAIT_CURSOR		
J_HAND_CURSOR		
J_MOVE_CURSOR		
J_N_RESIZE_CURSOR		
J_NE_RESIZE_CURSOR		
J_E_RESIZE_CURSOR		
J_SE_RESIZE_CURSOR		
J_S_RESIZE_CURSOR		
J_SW_RESIZE_CURSOR		
J_W_RESIZE_CURSOR		
J_NW_RESIZE_CURSOR		

Der Afolgende Auszug aus einer Eventloop zeigtein Demoprogramm zum Anzeigen der vorhandenen Cursortypen. Es ist für jeden Cursor ein Menüeintrag vorhanden, der in der Eventloop abgefragt wird. Entsprechend der Auswahl wird dann für den Frame der Cursor gesetzt:

```
rem       Example cursor.bas

:
while((obj <> jframe) and (obj <> quit))
   obj=j_nextaction()

   if(obj=def)   j_setcursor(jframe,J_DEFAULT_CURSOR)
   if(obj=cross) j_setcursor(jframe,J_CROSSHAIR_CURSOR)
   if(obj=hand)  j_setcursor(jframe,J_HAND_CURSOR)
   if(obj=move)  j_setcursor(jframe,J_MOVE_CURSOR)
   if(obj=jtext) j_setcursor(jframe,J_TEXT_CURSOR)
   if(obj=jwait) j_setcursor(jframe,J_WAIT_CURSOR)

   if(obj=nr)    j_setcursor(jframe,J_N_RESIZE_CURSOR)
   if(obj=ner)   j_setcursor(jframe,J_NE_RESIZE_CURSOR)
   if(obj=er)    j_setcursor(jframe,J_E_RESIZE_CURSOR)
   if(obj=ser)   j_setcursor(jframe,J_SE_RESIZE_CURSOR)
```

```
    if(obj=sr)    j_setcursor(jframe,J_S_RESIZE_CURSOR)
    if(obj=swr)   j_setcursor(jframe,J_SW_RESIZE_CURSOR)
    if(obj=wr)    j_setcursor(jframe,J_W_RESIZE_CURSOR)
    if(obj=nwr)   j_setcursor(jframe,J_NW_RESIZE_CURSOR)

wend
:
```

Abbildung 5.6 zeigt diese kleine Applikation.

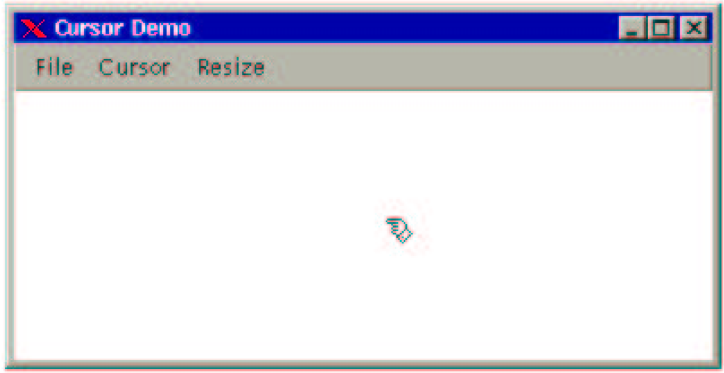


Abbildung 5.6: Eine Applikation zur Auswahl aller Cursortypen.

5.5 Bilder

Die Japilib bietet verschiedene Funktionen zum Laden und Manipulieren von Bildern. Die beiden Bildformate, die Zur Zeit unterstützt werden sind GIF und JPEG. Ein Bild wird unter JAPI ähnlich behandelt wie die anderen Graphischen Objekte, d.h. für jedes Bild wird ein Eventcounter zurückgeliefert. Im gegensatz zu den interactiven Elementen, kann jedoch ein Bild niemals einen Event liefern. Der Eventcounter eines Bildes ist vielmehr als eine Art Index zu verstehen, über den man auf ein Bild zugreifen kann.

Ein Bild wird mit der Funktion:

```
image = j_loadimage(filename$)
```

in den JAPI Kernel geladen. Die Abmessungen eines geladenen Bildes können mit den bereits bekannten Funktionen:

```
breite = j_getwidth(image)
hoehe  = j_getheight(image)
```

erfragt werden. Über die Funktion:

```
j_drawimage(component, image, x, y)
```

kann ein Bild in einer Component angezeigt werden. Die beiden Parameter x und y legen den linken oberen Punkt des Bildes innerhalb der Component fest.

Bei der Darstellung eines Bildes in einer Component, gelten die gleichen Einschränkungen wie bei den Graphic Primitiven. Prinzipiell kann ein Bild in jeder Component angezeigt

werde, aber nur der Canvas verfügt über die Double Buffer Technik, die sicherstellt, das die Komponente nach einer Verdeckung wieder restauriert wird.

Mit diesen Funktionen wird im folgenden Beispiel ein einfacher Videoplayer erstellt:

```
rem       Example video.bas

:
for i=0 to 17
   filename$ = "managerspiel/ms"+str$(i+1)+".gif"
   print "Loading ",filename$
   image(i)  = j_loadimage(filename$)
next i
:
breite  = j_getwidth(image(0))
hoehe   = j_getheight(image(0))
canvas  = j_canvas(jframe,breite,hoehe)

while((obj <> jframe) and (obj <> quit))

    :
    if(do_work=J_TRUE) then
        j_drawimage(canvas,image(i),0,0)
        j_sync()
        j_sleep(50)
        i = mod((i+1),18)
    endif

wend
:
```

Znächst werden in einer Schleife alle Bilder geladen. Da alle Bilder gleich groß sind wird nur vom ersten Bild die Abmessungen ermittelt, und der Canvas entsprechend dimensioniert. In der Eventloop werden die Bilder nun hintereinander dargestellt, was zu einer flüssigen Animation führt (siehe Abbildung 5.7).

Abbildung 5.7: Der Videobetrachter in Aktion.

Auch von dem Inhalt eines Canvas läßt sich ein Bild erstellen, das eine Momentaufname des aktuellen Inhalts des Canvas enthält. Die Funktion dazu ist:

```
image = j_getimage(canvas)
```

Soll das Bild jedoch nur einen Teil des Canvas beinhalten, oder eine andere Größe als der Canvas besitzen, so kann mit der Funktion

```
image = japi_getscaledimage(canvas, x, y, sw, sh, tw, th)
```

ein Bild erstellt werden, das aus dem Canvas von der Position (x,y) einen Auschnitt der Breite *sw* und der Höhe *sh* in ein Bild speichert mit den Ausmaßen: Breite = *tw* und Höhe = *th*

In ähnlicher Weise kann ein bereit bestehendes Bild scaliert werden, wenn es mit der Funktion

```
j_drawscaledimage(component,image,sx,sy,sw,sh,tx,ty,tw,th)
```

dargestellt wird. Diese Funktion bewirkt, das ein Ausschnitt vom Punkt (sx,sy) mit der Breite *sw* und der Hoehe *sy* in der Component dargestellt wird. Dabei wird das Bild an der Position (tx,ty) der Component dargestellt und auf eine Breite von *tw* und eine Höhe vom *th* transformiert.

Das folgende Beispiel zeigt diese Funktionen im Einsatz:

```
j_fillcircle(canvas,150,150,100)
image = j_getscaledimage(canvas, 50, 50, 250, 250, 50, 50 )
j_drawscaledimage(canvas,image,0,0,25,25,0,0,100,100)
```

Zunächst wird in einem Canvas mittig ein gefüllter Kreis erzeugt, der einen Radius von 100 Pixeln hat. Mittels ***j_getscaledimage()*** wird nun ein Bild erzeugt, das nur den Ausschnitt des Keises enthält. Dabei wird das Bild gleichzeitig auf eine Kantenlänge von 50:50 reduziert. Mit der Funktion ***j_drawscaledimage()*** wird nun aus diesem Bild das linke obere Viertel entnommen und im Canvas ebenfalls links oben angezeigt. Dabei wird der Ausschnitt nun auf ein Größe von 100x100 Pixeln vergrößert. Die Abbildung 5.8 verdeutlicht diesen Vorgang, und zeigt das resultierte Bild.

Die beiden letzten Funktionen erlauben eine direkte Manipulation von Bildinhalten. Dabei wird das Bild in Form von RGB Werten an die Applikation geliefert, bzw. an den JAPI Kernel überliefert. Durch Änderung der RGB Werte kann nun das Bild verändert werden. Diese beiden Funktionen sind nur auf den Canvas anwendbar.

```
j_getimagesource( canvas, r(), g(), b() )
j_drawimagesource( canvas, r(), g(), b() )
```

Die Funktion ***j_getimagesource()*** liefert einen Ausschnitt vom Punkt (x,y) der Breite *w* und der Höhe *h* zurück. Der Bildinhalt wird in den Arrays *r,g,b* abgelegt. Diese müssen daher mindestens die Länge *w*h* besitzen. Analog wird mit der Funktion ***j_drawimagesource()*** der Inhalt der Arrays *r,g,b* in der übergebenen Komponente dargestellt. Die Bilddaten werden ab der Position (x,y) mit einer Breite von *w* und einer Höhe von *h* dargestellt. Der folgende Auszug aus einer Eventloop realisiert mit diesen beiden Funktionen einen Inverter des Bildinhaltes eines Canvas (siehe Abbildung 5.9):

```
rem Example image.bas

:
if(obj = invert) then
```

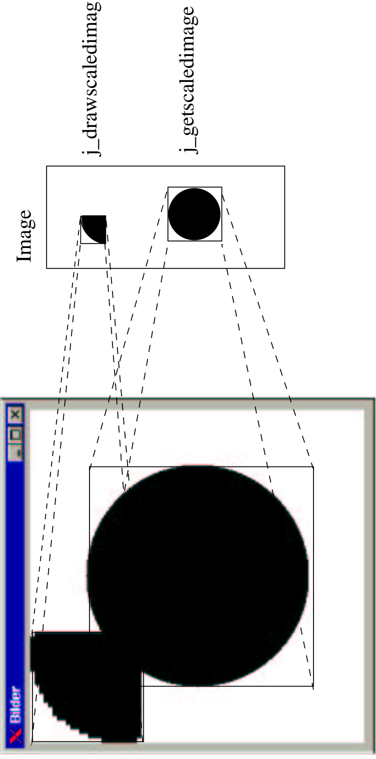


Abbildung 5.8: Beispiel zur Benutzung der *scaledimage* Funktionen

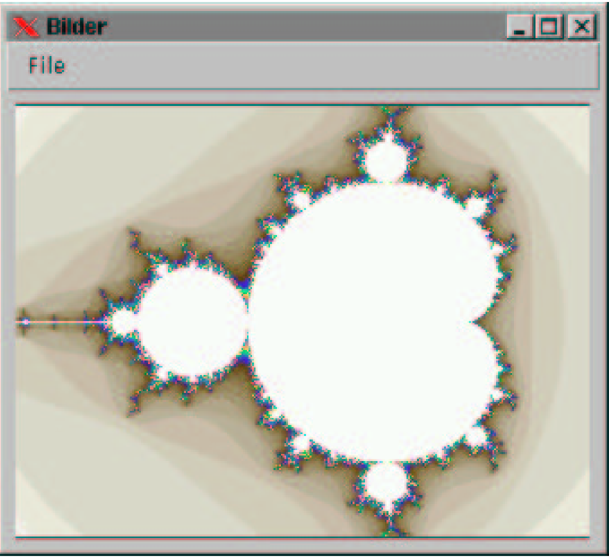


Abbildung 5.9: Beispiel zur Benutzung der ***j_getimagesource()*** und ***j_drawimagesource()*** Funktionen. Das Image des Canvas wurde invertiert

```
        j_getimagesource(canvas,0,0,breite,hoehe,r(),g(),b())
        for i=0 to breite*hoehe
            r(i) = 255-r(i)
            g(i) = 255-g(i)
            b(i) = 255-b(i)
        next i
        j_drawimagesource(canvas,0,0,breite,hoehe,r(),g(),b())
    endif
    :
```

Abschließend soll auch an dieser Stelle nochmal das einleitende Beispiel der Mandelbrot Applikation aufgegriffen werden. Mit Hilfe der ***j_drawimagesource()*** Funktion läßt sich die Applikation deutlich beschleunigen, sodaß sich gegenüber einer plattformnahen Implementierung kaum noch Geschwindigkeitseinbußen festzustellen sind. Anstatt jeden berechneten Punkt nun einzeln zu setzen, wird im nachfolgenden Beispiel immer eine komplette Zeiel des Bildes berechnet und mit der ***j_drawimagesource()*** Funktion dargestellt. Das zeilenweise Darstellen ermöglich es, auch während der Berechnung auf weitere Events (wie Abbruch) zu reagieren.

```
rem       Example mandel2.bas

:
do x=0,breite
   zre = xstart + x*(xend-xstart)/breite
   zim = ystart + y*(yend-ystart)/hoehe
   it = mandel(zre,zim,512)
   r(x)=it*11
   g(x)=it*13
   b(x)=it*17
enddo
j_drawimagesource(canvas,0,y,breite,1,r,g,b)
:
```

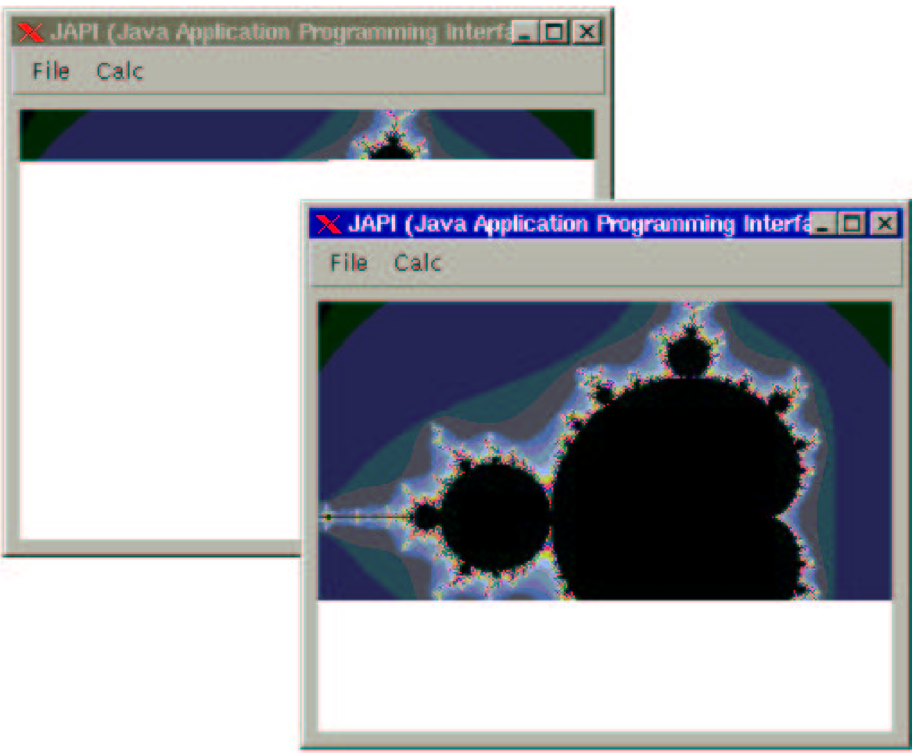


Abbildung 5.10: Ein Vergleich der alten und neuen Mandelbrot Applikation. Beide Applikationen wurden nach ca. 5 sec gestoppt (Intel P5 166MHz MMX).

Kapitel 6

Nützliches

6.1 Debugmodus

mit der Anweisung:

```
j_setdebug(debuglevel)
```

kann ein Debugmodus des Japiservers eingeschaltet werden. Sobald ein debuglevel > 0 übergeben wird, öffnet sich ein weiteres Textfenster, in dem die Aktionen des JAPI Kernels protokoliert werden (siehe Abbildung 6.1). So kann in der Entwicklungsphase einer Applikation die korrekte Parameterübergabe an den Kernel überwacht werden. Wird ein Debuglevel von 0 angegeben, so verschwindet das Debugfenster.

Die Menge der Protokollausgaben wird über den Degublevel gesteuert. Momentan sind 5 Level implementiert:

0: keine Ausgabe (default Wert)

1: Rückmeldung der konstruktiven Funktionen. Nur das Erzeugen der graphischen Objekte wird protokolliert.

2: Wie 1, zusätzliche Ausgabe aller Aktionen, die vom Benutzer ausgeführt werden.

3: Wie 2, zusätzlich werden alle weiteren Funktionen (außer den graphischen Befehlen) protokoliert.

4: Wie 3, zusätzlich mit allen graphischen Befehlen.

Die Funktion ***j_***setdebug() ist die einzige Funktion die bereits vor der ***j_***start() Funktion aufgerufen werden kann. Durch mehrmaliges Aufrufen der Funktion ***j_***setdebug() kann der Debuglevel während des Programmlaufs dynamisch angepasst werden. Abbildung 6.1 zeigt ein beispielhaftes Ausgabefenster.

6.2 Drucken

Auch ein rudimentäres Drucken ist unter der Japilib möglich. Es erlaubt zwar (noch) kein Positionieren und Skalieren, dafür ist der Aufruf aber auch denkbar einfach:

Abbildung 6.1: Das Debugfenster mit einem Protokoll einer Japi Sitzung.

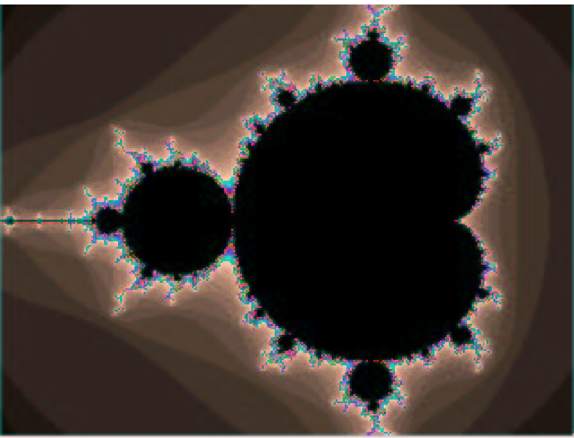
```
j_print(component)
```

Ist das übergebene Objekt ein Container (Frame, Dialog, Panel oder Window) so wird es samt seinem Inahlt ausgedruckt. Nach dem Absetzen eines ***j_print()*** Kommandos, erscheint, je nach Plattform, eine spezifische Print–Dialogbox, in der man in der Regel auch die Möglichkeit besizt, den Inhalt des Druckjobs abzuspeichern. Die folgenden Beispiele sind so entstanden:

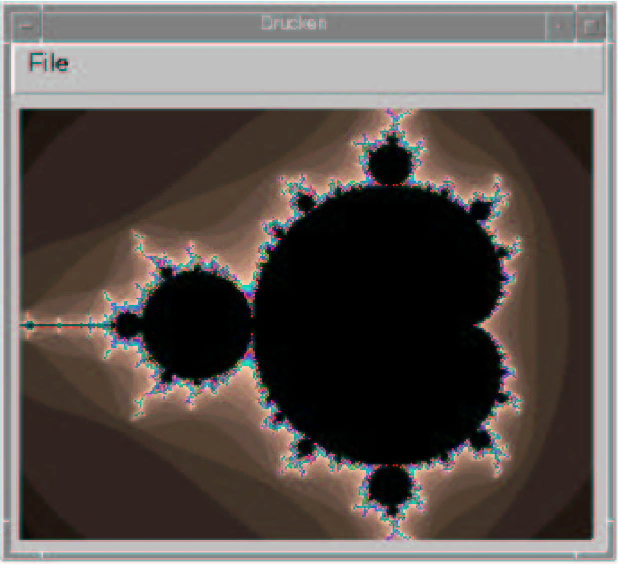
```
rem       Example print.bas

:
if(obj = printcanvas)  j_print(canvas)
if(obj = printjframe)  j_print(jframe)
:
```

erzeugt somit einen Ausdruck mit folgendem Inhalt,



wenn der Menupunkt "printcanvas" gewählt wurde. Nach Anklicken des Menupunktes "printframe" wird folgendes Bild ausgedruckt:



Unter X-Windows wird eine Applikation immer im Motif–Look gedruckt (siehe Abbildung). Dies geschieht auch dann, wenn die Applikation unter einem anderen Windowmanager läuft, und somit ein anderes Look and Feel besitzt. Der obige Ausdruck entstand zB. unter dem TWM Windowmanager, unter dem diese Applikation deutlich anders aussieht.