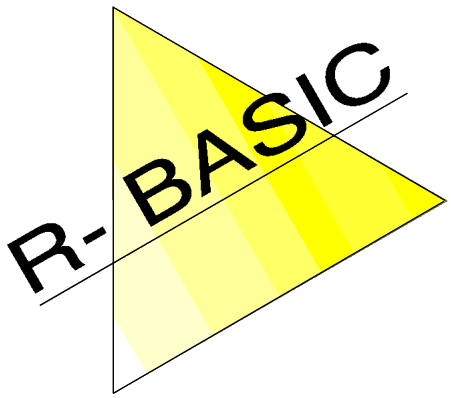
R-BASIC

Einfach unter PC/GEOS programmieren



Objekt-Handbuch

Volume 9
Visual Objekt Klassen

Version 1.0

(Leerseite)

Inhaltsverzeichnis

Volume 9

Volume 10

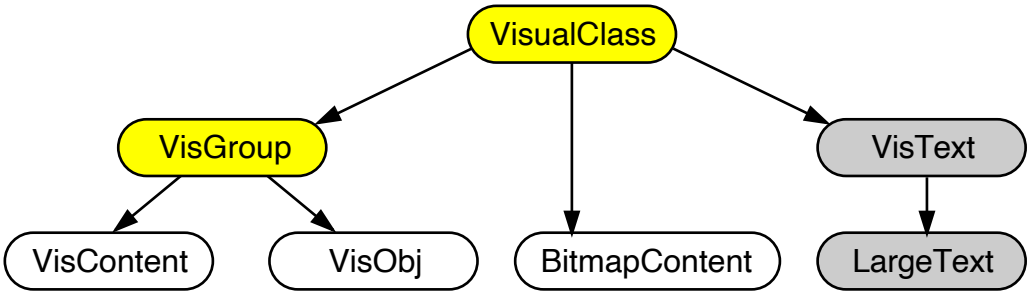
(Leerseite)

5 VisualClass Objekte

5.1 Die VisualClass

Die VisualClass ist die Superclass für alle Visual Objekt Klassen. Visual Objekte werden innerhalb eines View-Objekts dargestellt. Sie dienen der komfortablen Ausgabe von Grafik bzw. Text und können auf Maus- und Tastaturereignisse reagieren. Dieser Abschnitt beschreibt die gemeinsamen Eigenschaften aller VisualClass Objekte. Ausnahmen sind explizit erwähnt.

In R-BASIC gibt es die folgenden VisualClass Objekte:



- BitmapContent: Dieses Objekt verwaltet eine editierbare Bitmap und ist die erste Wahl, wenn es darum geht möglichst einfach Grafik auszugeben.
- VisGroup Die VisGroup Class ist die Superclass für VisContent und VisObj. Sie können in R-BASIC keine Objekte dieser Klasse anlegen.
- VisContent: Objekte dieser Klasse können selbst Grafik ausgeben und sie können Children der Klassen VisObj und VisText haben, die ihrerseits Grafik bzw. Texte anzeigen können.
- VisObj: Objekte dieser Klasse sind die Children eines VisContent Objekts und können selbst Children der Klasse VisObj haben. Sie ermöglichen es zum Beispiel, Grafiken so zu organisieren, dass einzelnen Teile mit der Maus angeklickt und separat bearbeitet werden können.
- VisText: VisText-Objekte erlauben die Anzeige und Bearbeitung von Texten direkt in der Grafikebene. Sie müssen als Children eines VisContent eingebunden werden. VisText-Objekte werden ausführlich nicht hier, sondern im Kapitel 4.10 (Text-Objekte) besprochen.
- LargeText LargeText-Objekte ermöglichen die Anzeige und Bearbeitung von beliebig großen Textmengen (theoretisch bei zu 2 GByte). Sie müssen ebenfalls als Children eines VisContent eingebunden werden. LargeText-Objekte werden ausführlich nicht hier, sondern im Kapitel 4.10 (Text-Objekte) besprochen.

Die VisualClass stellt in R-BASIC keine eigenen Instance-Variablen oder Methoden bereit, erledigt im Hintergrund aber viele Dinge, die unverzichtbar und allen VisualClass Objekten gemeinsam sind. Dazu gehören insbesondere die folgenden Fähigkeiten:

Mausunterstützung

Alle VisualClass-Objekte unterstützten die Behandlung von Mausereignissen. Dazu werden von VisContent, VisObj und BitmapContent die folgende Instancevariablen und Methoden unterstützt. Die Text-Objekte behandeln die Mausereignisse komplett selbständig.

Actionhandler	Instancevariablen	Methoden
OnMouseButton OnMouseMove OnMouseOver	sendMouseEvents	GrabMouse ReleaseMouse TestInside TestInsideAC

Die Maus-Actionhandler müssen als MouseAction deklariert sein. Eine detaillierte Beschreibung der Arbeit mit der Maus finden Sie im Handbuch "Spezielle Themen", Kapitel 17.
Es ist sehr häufig, dass VisualClass-Objekte mit der Maus umgehen müssen. Sie können auch Text und Grafik innerhalb des Maushandlers auf den Schirm ausgeben. Allerdings speichert das VisContent und das VisObj Objekt diese Ausgaben nicht. Nur das BitmapContent Objekt speichert die Grafikausgaben gleichzeitig in der Bitmap.

Tastaturhandling

Sie können sich in das Tastaturhandling aller VisClass-Objekte, auch der Text-Objekte, einklinken, indem Sie einen Tastaturhandler schreiben. Dazu werden die folgenden Instancevariablen und Actionhandler unterstützt:

Actionhandler	Instancevariablen	Methoden
OnKeyPressed	inputFlags	—

Eine ausführliche Beschreibung, wie man einen Tastaturhandler schreibt und was es dabei zu beachten gilt, finden Sie im Handbuch "Spezielle Themen", Kapitel 14.

Focus und Target

Alle VisualClass Objekte interagieren mit der Focus- und Target-Hierarchie. Es ist möglich zu überwachen, ob ein VisualClass-Objekt den Focus oder das Target hat, indem man einen Focus- bzw. Target-Handler schreibt. Dazu werden die folgenden Actionhandler und Systemvariablen unterstützt.

Für VisObj-Objekte ist wichtig, dass die Tastatureingaben an das Focus-Objekt gehen. Das Beispiel "VisObj Keyboard Demo" verwendet die Focus-Hierarchie um dem Nutzer ein visuelles Feedback zu geben, welches Objekt als letztes angeklickt wurde.

Actionhandler	Instancevariablen	Systemvariable
OnFocusChanged OnTargetChanged	—	Focus Target

Die Arbeit mit Focus und Target ist etwas für erfahrene Programmierer und nur in wenigen Fällen notwendig. Die notwendigen Details dazu finden Sie in den Kapiteln 12 (Focus und Target) und 13 (Implementieren von Menüs: Bearbeiten, Textgröße und andere) des Handbuchs "Spezielle Themen".

Arbeit mit dem Clipboard

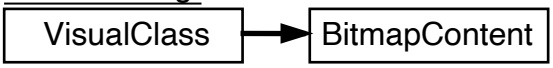
Alle VisualClass Objekte können mit der Zwischenablage (Clipboard) kommunizieren. Die Methoden **ClpTestCopy**, **ClpTestPaste**, **ClpCopy** und **ClpPaste** werden unterstützt. Eine detaillierte Beschreibung dieser Methoden finden Sie im Kapitel "Arbeit mit der Zwischenablage" (Kapitel 5 im Handbuch "Spezielle Themen"). Für BitmapContent-Objekte und die Text-Objekte gibt es dabei keine Einschränkungen, bei VisContent und VisObj-Objekten muss der gepufferte Modus aktiv sein (Instancevariable **buffered** = TRUE). Außerdem müssen Sie die Methode **MarkInvalid** aufrufen, nachdem Sie eine Grafik mit ClpPaste eingefügt haben, damit sich der visual Tree neu darstellt.

5.2 Das BitmapContent

5.2.1 Überblick

Objekte der Klasse BitmapContent verwalten eine editierbare Bitmap. Bitmaps sind digitalisierte Bilder. Sie bestehen aus einer rechteckigen Anordnung von einzelnen Bildpunkten (Picture Element: Pixel). Jedem Pixel kann eine eigene Farbe zugeordnet werden. In die Bitmaps der Klasse BitmapContent kann Text oder Grafik geschrieben werden. Das BitmapContent-Objekt legt die zugehörige Bitmap automatisch selbst an, so dass sie sofort benutzt werden kann.
Einen kompletten Überblick über die weitern Möglichkeiten von R-BASIC, Grafik auszugeben, finden Sie im Kapitel 2.2.

Abstammung:



Da BitmapContent Objekte von der VisualClass abstammen, kommen Sie nicht in den generic Tree des Programms. Stattdessen werden die über die Instance-Variable "Content" eines Views mit dem View verbunden. Das View muss aber in den generic Tree des Programms eingebunden werden.

Arbeit mit dem Clipboard

BitmapContent Objekte können mit der Zwischenablage (Clipboard) kommunizieren. Die Methoden (Objektanweisungen) **ClpTestCopy**, **ClpTestPaste**, **ClpCopy** und **ClpPaste** werden unterstützt. Eine detaillierte Beschreibung dieser Methoden finden Sie im Handbuch "Spezielle Themen", Kapitel 5, "Arbeit mit der Zwischenablage". Für BitmapContent Objekte gelten dabei folgende Besonderheiten:

- Die Methode **ClpCopy** kopiert die Bitmap-Grafik sowohl als reine Bitmap als auch als Graphic String in die Zwischenablage. Damit können sowohl andere BitmapContent Objekte als auch andere GEOS Anwendungen wie GeoWrite oder GeoDraw die Grafik aus der Zwischenablage lesen.
- Die Methode **ClpPaste** akzeptiert sowohl reine Bitmaps als auch als Graphic Strings, wobei Bitmaps bevorzugt werden. Wird ein Graphic String aus der Zwischenablage gelesen, so legt das Objekt eine transparente Bitmap an und kopiert den Graphic String in diese Bitmap.
- Das Objekt passt seine Größe automatisch an das mit **ClpPaste** aus der Zwischenablage gelesene Bild an.
- Die globale Variable **clipboardError** wird auf FALSE oder TRUE gesetzt, je nachdem ob **ClpCopy** bzw. **ClpPaste** erfolgreich waren oder nicht.

Mausunterstützung

Es ist sehr häufig, dass ein BitmapContent-Objekt mit der Maus umgehen muss. Da ein BitmapContent-Objekt in Normalfall keine Children hat muss es die Mausereignisse selbst behandeln. BitmapContent-Objekte erben die Fähigkeiten

im Umgang mit der Maus vom der VisualClass. Eine detaillierte Beschreibung der Arbeit mit der Maus finden Sie im Handbuch "Spezielle Themen", Kapitel 17.

Tastaturhandling

BitmapContent-Objekte erben die Fähigkeiten im Umgang mit der Tastatur vom der VisualClass. Eine ausführliche Beschreibung, wie man einen Tastaturhandler schreibt und was es dabei zu beachten gilt, finden Sie im Handbuch "Spezielle Themen", Kapitel 14.

Focus und Target

BitmapContent Objekte erben die Fähigkeiten im Umgang mit der Focus- und der Traget-Hierarchie von der VisualClass. In den Kapiteln 12 und 13 des Handbuchs "Spezielle Themen" finden Sie eine detaillierte Darstellung des Umgangs mit Focus und Target.

Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
bitmapFormat	bitmapFormat = **x**, **y**, **n** [, **flags**]	lesen, schreiben
defaultColor	defaultColor = **fg**, **bg**	lesen, schreiben
DefaultScreen	DefaultScreen	—
suspendDraw	—	lesen, schreiben
editMask	—	lesen, schreiben

Methoden:

Methode	Aufgabe
Redraw	Bitmap neu zeichnen
GetBitmapHandle	Handle auf die Bitmap des Objekts holen
CopyBitmap	Kopie der Bitmap des Objekts erstellen
NewBitmapFromHandle	Bitmap aus Handle auslesen (ins Objekt kopieren)
GetPaletteEntry	Einzelnen Paletteneintrag holen
SetPaletteEntry	Einzelnen Paletteneintrag setzen
GetFullPalette	Vollständige Palette holen
SetFullPalette	Vollständige Palette setzen
PeekLine	Einzelne Bitmapzeile aus dem RAM holen
PokeLine	Einzelne Bitmapzeile in den RAM schreiben

5.2.2 Grundlegende Funktionen

Beispiel UI-Code:
Das View "MyView" enthält ein BitmapContent, das eine 320x256 Pixel große True-Color Bitmap darstellt. Es kommuniziert automatisch mit dem BitmapContent "MyBitmap" um seine Größe auf 320x256 Pixel zu setzen, so dass die ganze Bitmap sichtbar ist. "DefaultScreen" stellt das BitmapContent als "Standard-Ausgabe-Objekt" für Grafik- und Textausgaben ein.

```
View  MyView
  vControl = HVC_NO_LARGER_THAN_CONTENT + \
                      HVC_NO_SMALLER_THAN_CONTENT
  hControl = HVC NO LARGER THAN CONTENT + \
                      HVC NO SMALLER THAN CONTENT
  Content = MyBitmap
  END Object

BitmapContent    MyBitmap
  bitmapFormat = 320, 256, 24
  DefaultScreen
  defaultColor = BLACK, LIGHT_CYAN
  END Object
```

In vielen Fällen wird der im Code oben verwendete Fall (kein Scrolling der Bitmap, kein Zoom) ausreichend sein. Ein BitmapContent ist jedoch ein vollwertiges Content-Objekt und kann daher z.B. auch in einem scrollbaren View dargestellt werden:

```
View    MyView
  hControl = HVC_SCROLLABLE
  vControl = HVC_SCROLLABLE
  fixedSize = 200, 150
        ! Kleiner als das Content
  Content = MyBitmap
  END Object

BitmapContent  MyBitmap
  bitmapFormat = 320, 256, 24
  DefaultScreen
  defaultColor = BLACK, LIGHT CYAN
  END Object
```

bitmapFormat

Die Instance-Variable bitmapFormat speichert die Größe, die Farbtiefe und weitere Eigenschaften der Bitmap. R-BASIC unterstützt die Farbtiefen 1 (schwarz/weiß), 8 (256 Farben) und 24 (True Color, 16 Mio. Farben). Die Farbtiefe 4 (16 Farben) wird von R-BASIC nicht unterstützt. Verwenden Sie stattdessen 8 Bit Farbtiefe. Über den Parameter flags können Sie einstellen, ob

die Bitmap transparent sein soll ("maskierte" Bitmap) und/oder eine Palette verwenden soll. Masken und Paletten in den nächsten Abschnitten beschrieben.

Syntax UI-Code: **bitmapFormat** = **x**, **y**, **n** [, **flags**]
x: Breite der Bitmap
y: Höhe der Bitmap
n: Farbtiefe (zulässige Werte: 1, 8, 24)
flags: Transparenz und Palette. Siehe Tabelle unten.

```
Lesen:    <numVar> = <obj>.bitmapFormat (0)        ' Breite
          <numVar> = <obj>.bitmapFormat (1)        ' Höhe
          <numVar> = <obj>.bitmapFormat (2)        ' Farbtiefe
          <numVar> = <obj>.bitmapFormat (3)        ' flags
Schreiben: <obj>.bitmapFormat = x, y, n [, flags ]
```

Beispiel UI-Code: siehe oben

Für "flags" sind folgende Werte zugelassen:

Konstante	Wert	Bedeutung
BF_MASK	1	Transparente Bitmap
BF_PALETTE	2	Bitmap mit Palette
BF_MASK + BF_PALETTE		Maske und Palette

Wenn Sie im BASIC-Code die Variable bitmapFormat belegen (schreiben), so wird die Bitmap neu angelegt. Alle vorhandenen Informationen (Grafik, Text..) gehen verloren. Die Bitmap darf dabei weiterhin als Content einen Views gesetzt sein, muss es aber nicht.

Beispiele BASIC-Code:
Lesen der Werte:

```
DIM  b, h, f  as WORD
  b = MyBitmap.bitmapFomat (0)     ' Breite
  h = MyBitmap.bitmapFomat (1)     ' Höhe
  f = MyBitmap.bitmapFomat (2)     ' Farbtiefe

  Print "Bitmapgröße:" b; "x"; h; "Pixel, "; f; "Bit pro Pixel"
    ! z.B.    320 x 256 Pixel,  24 Bit pro Pixel
```

Neu anlegen der Bitmap: 800 x 600 Pixel, 256 Farben

```
  MyBitmap.bitmapFormat = 800, 600, 8
```

Hinweis: Das Bitmapobjekt informiert sein View automatisch über seine neue Größe, so dass das View ggf. seine eigene Größe anpassen kann.

defaultColor

Die Instance-Variable defaultColor enthält die Farben, die beim Initialisieren der Bitmap (erstmaliges bzw. Neuanlegen der Bitmap) verwendet werden. Außerdem werden sie verwendet, wenn das Objekt zum "Screen" wird. Das tritt auf, wenn das Objekt die Anweisung DefaultScreen im UI-Code enthält oder wenn es der Systemvariablen Screen direkt zugewiesen wird (vergleiche Kapitel 2.3.1 "Die Screen-Variable").

BitmapContent-Objekte ohne die Anweisung defaultColor verwenden die Farben "schwarz auf weiß".

Syntax UI-Code: **defaultColor** = **fg**, **bg**
fg: Vordergrund (foreground)
bg: Hintergrund (background)
fg und bg müssen Indexfarben sein. RGB-Farben sind nicht zulässig.

Lesen: **<numVar>** = **<obj>**.**defaultColor** (0) ' fg
<numVar> = **<obj>**.**defaultColor** (1) ' bg

Schreiben: **<obj>**.**defaultColor** = **fg**, **bg**

Beim Anlegen der Bitmap löscht R-BASIC die Bitmap in der Hintergrundfarbe bg. Wird das zugehörige BitmapContent-Objekt zum Screen setzt R-BASIC die Farben folgendermaßen:

Hintergrundfarbe: bg
Text-, Linien- und Flächenfarbe: fg

Das ist prinzipiell so, als würde automatisch die Anweisung "COLOR fg, bg" ausgeführt.

DefaultScreen

Diese Anweisung im UI-Code bewirkt, dass das entsprechende BitmapContent als "Standard-Ausgabe-Objekt" festgelegt wird. Es wird dazu automatisch in der Systemvariablen Screen gespeichert (vergleiche Kapitel 2.3.1 "Die Screen-Variable"). Alle Grafik- oder Textausgaben gehen damit automatisch auf dieses Objekt.

Syntax UI-Code: **DefaultScreen**

Beispiel UI-Code:

```
BitmapContent  MyBitmap
  bitmapFormat = 320, 256, 24
  DefaultScreen
  END Object
```

Hinweis für Profis

BitmapContent-Objekte sind auch dann voll nutzbar, wenn sie nicht mit einem View verbunden sind, d.h. sie können zum "Screen" gemacht oder als "DefaultScreen" gesetzt werden. Natürlich werden sie dann nicht auf dem Bildschirm erscheinen. Grafik- und Textausgaben gehen dann "im Hintergrund" in die Bitmap und werden sichtbar, sobald das Objekt an ein View gekoppelt wird (z.B. mit der Zuweisung MyView.Content = MyBitmapContent). Insbesondere ist es möglich zwischen zwei BitmapContent Objekten hin- und herzuschalten. Sie können die eine Bitmap im Hintergrund ändern, während die andere sichtbar ist - und dann die Veränderungen mit der Zuweisung MyView.Content = .. auf "einen Schlag" sichtbar machen.

5.2.3 Erweitere Funktionen

suspendDraw

Parallel zur Bitmap gehen die Grafikausgaben gleichzeitig auf den Bildschirm. Dort wird aber weder das **Vorhandensein einer Maske** noch die Information, dass eventuell "nur" die Maske bearbeitet wird, berücksichtigt. Auch eine eventuell **geänderte Farbpalette** (siehe Kapitel 5.2.5) wird nicht berücksichtigt. Das führt zu einem zeitweisen Widerspruch zwischen Darstellung auf dem Bildschirm und der Grafik in der Bitmap. Deswegen sollten Sie, während Sie in eine maskierte Bitmap schreiben (egal ob Maske oder Bitmapdaten), die parallel dazu verlaufende Ausgabe auf den Monitor deaktivieren. Das Gleiche gilt für das Schreiben in eine Bitmap mit geänderter Farbpalette. Für diesen Zweck gibt es die Instancevariable suspendDraw.

Syntax UI-Code: nicht zulässig
Lesen: **<numVar>** = **<Bitmapobj>**.**suspendDraw**
Schreiben: **<Bitmapobj>**.**suspendDraw** = TRUE | FALSE

SuspendDraw = TRUE deaktiviert die die gleichzeitige Ausgabe der Grafikbefehle auf den Bildschirm. Sobald suspendDraw wieder auf FALSE gesetzt wird zeichnet sich die Bitmap neu auf den Schirm, so dass die vorgenommenen Änderungen "auf einen Schlag" sichtbar werden.

SuspendDraw ist - unabhängig von der Existenz einer Maske oder einer Palette - ebenfalls nützlich, wenn Sie eine große Anzahl von Grafikbefehlen haben, die entweder zu "komischen" Zwischenresultaten führen oder sehr lange dauern. Das zeitweise Abschalten der Ausgabe auf den Schirm beschleunigt natürlich die Zeichenoperationen.
Verliert ein Objekt den "Screen" Status (d.h. belegen Sie die Systemvariable Screen neu), so wird die Suspendierung automatisch aufgehoben.

Redraw

Die Methode Redraw bewirkt, dass das Objekt die Bitmap neu auf den Bildschirm zeichnet. Der Aufruf der Methode ist notwendig, wenn Sie einen einzelnen Paletteneintrag geändert haben (Methode SetPaletteEntry, siehe Kapitel 5.2.5.2) oder eine Bitmapzeile manuell verändert haben (Methode PeekLine, siehe Kapitel 5.2.6).

Syntax: **<obj>**.**Redraw** [**drawBackground**]
drawBackground: TRUE | FALSE (Default: FALSE)

Beispiel:

```
DemoBitmap.Redraw
DemoBitmap.Redraw  TRUE
```

DrawBackground = TRUE bewirkt, dass sich das zum Objekt gehörende View neu zeichnet bevor das Objekt die Bitmap neu darstellt. Damit wird der Hintergrund gelöscht. DrawBackground = TRUE ist nur erforderlich, wenn die Bitmap eine Maske (Transparenzebene) hat und Sie mit der Methode PeekLine den von der Maske als durchsichtig markierten Bereich geändert haben.

GetBitmapHandle

Die Methode GetBitmapHandle liefert das Handle auf die vom BitmapContent verwaltete Bitmap. Das Handle kann zum Beispiel verwendet werden, um die Bitmap in ein anderes Objekt oder einen GString zu zeichnen. Im Kapitel 2.8.6.4 (Bitmaps und Bitmap Handles) des R-BASIC Programmierhandbuchs finden Sie eine Übersicht über die Möglichkeiten der Arbeit mit Bitmaphandles.

Syntax BASIC Code: **<han>** = **<obj>**.**GetBitmapHandle**
<han>: Variable vom Typ Handle

Alle Änderungen, die an der Bitmap im BitmapContent-Objekt gemacht werden wirken sich auf das Handle aus. Insbesondere wird das Handle ungültig, wenn das Objekt seine Bitmap neu anlegt (z.B. die Größe oder die Farbtiefe ändert oder wenn einer der Methoden NewBitmapFromHandle oder ClpPaste aufgerufen werden).

Das folgende Codebeispiel zeigt wie man eine Bitmap in eine BMP-Datei schreibt.

```
SUB  WritToBMPFile( fileName$ as String )
DIM han as HANDLE
  han = MyBitmap.GetBitmapHandle
  WriteBitmapToFile(han, fileName$)
End SUB
```

CopyBitmap

Die Methode CopyBitmap fertigt eine Kopie des vom BitmapContent verwalteten Bitmap an und liefert das Handle der Kopie. Das Handle kann verwendet werden, um die kopierte Bitmap mit DrawBitmap() zu zeichnen. Im Kapitel 2.8.6.4 (Bitmaps und Bitmap Handles) des R-BASIC Programmierhandbuchs finden Sie eine Übersicht über die Möglichkeiten der Arbeit mit Bitmaphandles.
Die mit CopyBitmap erstellte Kopie muss mit FreeBitmap wieder freigegeben werden.

Syntax BASIC Code: **<han>** = **<obj>**.**CopyBitmap**
<han>: Variable vom Typ Handle

Im Gegensatz zu GetBitmapHandle wirken sich Änderungen der Bitmap des Objekts nicht mehr auf die kopierte Bitmap und deren Handle aus.

NewBitmapFromHandle

Die Methode NewBitmapFromHandle kopiert eine durch ein Handle referenzierte Bitmap in das Objekt. Die alte vom Objekt gespeicherte Bitmap geht verloren. Das Objekt stellt sich anschließend neu dar. Es informiert auch sein View über die neue Größe der Bitmap.

Syntax BASIC Code: **<obj>.NewBitmapFromHandle <han>**
<han>: Referenz auf die zu kopierende Bitmap

Das folgende Codebeispiel kopiert eine Bitmap von einem Objekt in ein anderes.

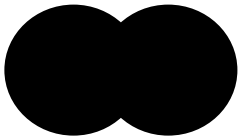
```
SUB  CloneBitmap( )
DIM han as HANDLE
  han = MyBitmap.GetBitmapHandle
  MyOtherBitmap.NewBitmapFromHandle han
End SUB
```

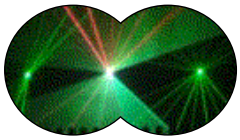
5.2.4 Arbeit mit transparenten Bitmaps

5.2.4.1 Überblick

Der Begriff "transparente Bitmap" oder auch "maskierte Bitmap" beschreibt, dass Teile der Bitmap durchsichtig sind, also den Hintergrund nicht verdecken. Man muss sich das so vorstellen, dass die Bitmap außer der eigentlichen Grafik noch eine schwarz-weiß Bitmap gleicher Größe enthält. Diese heißt "Maske" und bestimmt die Transparenz. Weiße Pixel sind durchsichtig, schwarze nicht.


Bitmapgrafik


Maske


maskierte Bitmap

Um eine transparente Bitmap anzulegen muss das Bit 0 (zugehöriger Wert: 1, Konstante BF_MASK) im Parameter "flags" der Instancevariable bitmapFormat gesetzt sein. Der folgende UI-Code definiert eine transparente 8-Bit Bitmap:

```
BitmapContent DemoBitmap
    bitmapFormat = 300, 100, 8, BF_MASK
    DefaultScreen
    defaultColor = YELLOW, LIGHT_BLUE
END Object
```

Beim Anlegen einer transparenten Bitmap wird die Maske vollständig gefüllt, d.h. die Bitmap ist zunächst nicht durchsichtig.

editMask

Normalerweise gehen Grafik- und Textausgaben direkt in die Bitmap und parallel dazu auf den Bildschirm. Die Maske wird dabei nicht verändert. Um die Maske zu beschreiben müssen Sie die Instancevariable editMask auf TRUE setzen.

Syntax	UI-Code: nicht zulässig	
	Lesen:	**<numVar>** = **<Bitmapobj>**.**editMask**
	Schreiben:	**<Bitmapobj>**.**editMask** = TRUE | FALSE

Danach gehen alle Zeichenoperationen in die Maske und die "normalen" Bilddaten bleiben unberührt.

suspendDraw

Parallel zur Bitmap gehen die Grafikausgaben gleichzeitig auf den Bildschirm. Dort wird aber das Vorhandensein einer Maske nicht berücksichtigt. Deswegen sollten

Sie, während Sie in eine maskierte Bitmap schreiben (egal ob Maske oder Bitmapdaten) die parallel dazu verlaufende Ausgabe auf den Monitor deaktivieren. Für diesen Zweck gibt es die Instancevariable suspendDraw. SuspendDraw = TRUE deaktiviert die die gleichzeitige Ausgabe der Grafikbefehle auf den Bildschirm. Sobald suspendDraw wieder auf FALSE gesetzt wird zeichnet sich die Bitmap neu auf den Schirm, so dass die vorgenommenen Änderungen "auf einen Schlag" sichtbar werden.

Wichtige Hinweise

- Masken werden nicht gescrollt. Werden die Bitmapdaten nach einer Print-Anweisung automatisch nach oben verschoben (Scrolling), so bleibt die Maske davon unberührt. Unter Umständen kann es sinnvoll sein, in diesem Zusammenhang suspendDraw zu verwenden.
- Bei der Ausgabe von Texten (Print-Befehl) wird der Hintergrund im Normalfall mit der Hintergrundfarbe gelöscht. Wenn dies stört setzen Sie die Hintergrundfarbe auf "transparent":

```
Paper BG_TRANSPARENT
```

- **Wichtig!** Das GEOS-System unterstützt transparente Bitmaps auch für 24-Bit Bitmaps, der Versuch, etwas in die Maske zu zeichnen führt jedoch zu einem Crash. Sie können die Methoden PeekLine und PokeLine verwenden, um die Maske von 24-Bit-Bitmap zu bearbeiten.

5.2.4.2 Beschreiben der Maske

Da es sich bei der Maske aus Sicht des Systems um eine schwarz-weiß-Bitmap handelt sollten Sie beim Zeichnen in die Maske (editMask = TRUE) nur die Farben Schwarz (macht den Bereich undurchsichtig) oder Weiß (macht den Bereich durchsichtig) verwenden. Flächen in anderer Farbe (nicht aber Linien und Texte) werden entsprechend der Helligkeit der Farbe gerastet.
Alternativ zu den Farben kann man das Feld "mixMode" der globalen Variablen "graphic" mit einem passenden Wert belegen. Mehr dazu im 2. Beispiel.

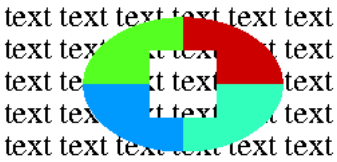
Die folgenden Beispiele verwenden die Kommandos ScreenSaveState (speichern aller Grafikdaten wie Farben, Font, mixMode usw.) und ScreenRestoreState (wiederherstellen der gespeicherten Werte). Außerdem wird die fertige Bitmap ins Clipboard kopiert (DemoBitmap.ClpCopy), von wo aus sie in z.B. GeoWrite für dieses Handbuch verwendet werden kann.

Der Befehl CLS wirkt - wie alle anderen Grafikbefehle - entweder auf die Bitmapdaten (editMask = FALSE) oder auf die Maske (editMask = TRUE). Ist editMask = TRUE löscht er die Maske immer (alles durchsichtig), egal welche Farbe Sie eingestellt haben.

Beispiel: Zeichenoperationen in die Maske

```
Sub InkDemo()

ScreenSaveState                ' Grafikdaten sichern
DemoBitmap.suspendDraw = TRUE ' Bildschirm tot legen

DemoBitmap.editMask = TRUE     ' Maske editieren
Cls
Ink BLACK
FillEllipse 0, 0, 150, 100
Ink WHITE
FillRect 50, 25, 100, 75
DemoBitmap.editMask = FALSE    ' Maske ist fertig

ScreenRestoreState             ' Grafikeinstellungen
                               ' wiederherstellen
FillRect 0, 0, 75, 50, GREEN
FillRect 75, 0, 150, 50, RED
FillRect 0, 50, 150, 100, BLUE
FillRect 75, 50, 150, 100, CYAN

DemoBitmap.suspendDraw = FALSE     ' alles neu zeichnen
DemoBitmap.ClpCopy

END Sub
```

Die vom Code oben erzeugte Bitmap vor einem Text:



Anmerkung: Da einige Druckertreiber (z.B. Postscript Color) transparente Bitmaps nicht korrekt drucken wurden alle transparenten Bitmaps in diesem Handbuch zuvor in GeoDraw vor einen Text gelegt und diese Kombination in eine (druckbare) unmaskierte Bitmap konvertiert.

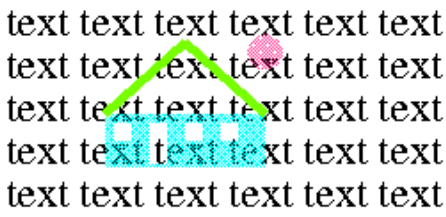
5.2.4.3 Verwendung des MixMode MM_SET

Nicht immer kann man sicherstellen, dass nur die Farben Schwarz und Weiß verwendet werden, z.B. wenn man einen GString (siehe R-BASIC Programmierhandbuch, Kapitel 2.8.5) in eine maskierte Bitmap schreiben will oder wenn die auszugebende Grafik in einer SUB steckt, die selbst Farben einstellt:

```
Sub PaintHouse()

ScreenSaveState
graphic.linewidth = 5
Ink LIGHT BLUE
FillRect 10, 50, 100, 80       ' das Haus
Ink WHITE
FillRect 15, 55, 25, 65        ' ein Fenster
FillRect 35, 55, 45, 78        ' die Tür
```

```
FillRect 55, 55, 65, 65        ' ein Fenster
FillRect 75, 55, 85, 65        ' ein Fenster
Ink LIGHT_GREEN
Line 10, 50, 55, 10            ' das Dach
Line 55, 10, 100, 50
Ink LIGHT_RED
FillEllipse 90, 5, 110, 25     ' die Sonne
ScreenRestoreState

END Sub
```

Um das Haus transparent in eine Bitmap zu zeichnen muss diese Sub sowohl für die Bitmapdaten als auch für die Maske gerufen werden. Dabei würden jedoch die farbigen Flächen gerastert (siehe Bild). Hier hilft das Einstellen des passenden "mixMode".
Dazu belegt man das Feld "mixMode" der globalen Variablen "graphic" mit dem passenden Wert. Für uns sind an dieser Stelle die Modes MM_SET, MM_CLEAR und MM_COPY interessant.



- **graphic.mixMode = MM_COPY** ist der Normalfall.

- **graphic.mixMode = MM_SET** bewirkt, dass die aktuelle Farbe ignoriert wird und alle Ausgaben in schwarz erfolgen. Bereiche der Maske, die in diesem Modus beschrieben werden, werden undurchsichtig.

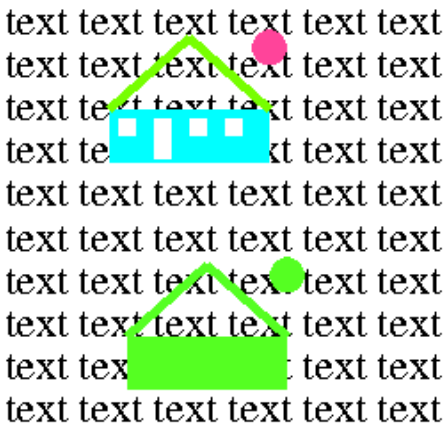
- **graphic.mixMode = MM_CLEAR** bewirkt, dass die aktuelle Farbe ignoriert wird und alle Ausgaben in weiß erfolgen. Bereiche der Maske, die in diesem Modus beschrieben werden, werden transparent. Der Befehl CLS nutzt diesen Modus automatisch um die Maske zu löschen (wenn editMask = TRUE ist).
  Achtung! MM_CLEAR wirkt nicht auf Textausgaben! Für transparente Buchstaben müssen Sie den "normalen" MixMode MM_COPY und die Farbe Weiß verwenden.

Beispiel: Verwendung des MixMode MM_SET. Beachten Sie, dass es egal ist, ob man erst die Maske oder erst die Bitmapdaten zeichnet.

```
Sub MixModeExample()
DIM mmSaved

DemoBitmap.suspendDraw = TRUE ' Schirmausgabe abschalten
Cls                           ' Bitmapdaten löschen
PaintHouse                    ' Farbige Grafik zeichnen

DemoBitmap.editMask = TRUE    ' Maske editieren
mmSaved = graphic.mixMode
graphic.mixMode = MM_SET
Cls                           ' Maske löschen
PaintHouse
graphic.mixMode = mmSaved     ' MixMode restaurieren
```

```
DemoBitmap.editMask = FALSE   ' Maske fertig
DemoBitmap.suspendDraw = FALSE    ' Alles neu zeichnen

DemoBitmap.ClpCopy
END Sub
```

Der Code erzeugt nun wie gewünscht das rechts dargestellte Bild.



Beispiel: Das Resultat vom vorherigen Beispiel soll vollständig einfarbig gefärbt werden. Das ist sehr einfach. Wir löschen die Bitmapdaten ohne die Maske zu verändern.

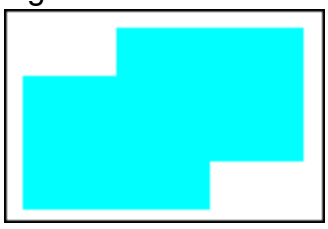
```
DemoBitmap.suspendDraw = TRUE
Paper GREEN
CLS
DemoBitmap.suspendDraw = FALSE
```

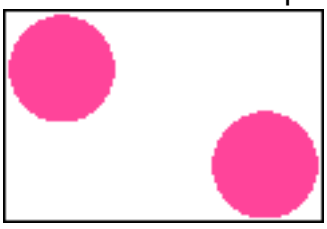
5.2.4.4 Zeichnen einer maskierten Bitmap in eine andere

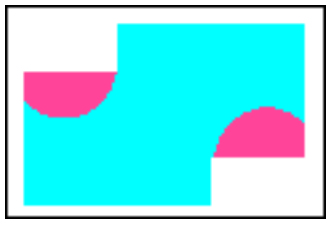
Wird eine maskierte Bitmap in eine andere Bitmap gezeichnet so werden die Masken natürlich berücksichtigt. Der folgende Code zeichnet eine Bitmap (aus dem Objekt DemoBitmap2) in den Screen DemoBitmap. Weil das Zielobjekt (DemoBitmap) ebenfalls eine maskierte Bitmap enthält setzen wir während der eigentlichen Zeichenanweisung die Instancevariable suspendDraw auf TRUE.

```
SUB DoDrawBitmap()
DIM h as Handle

  DemoBitmap.suspendDraw = TRUE
  h = DemoBitmap2.GetBitmapHandle
  DrawBitmap h, 0, 0
  DemoBitmap.suspendDraw = FALSE

End SUB
```

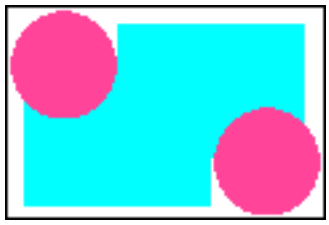
Das Ergebnis sieht so aus. Die Maske der Zielbitmap wurde nicht verändert.


DemoBitmap (Screen)


DemoBitmap2


Nach DoDrawBitmap

Wenn wir die Maske der Zielbitmap anpassen wollen (siehe Bild rechts) müssen wir explizit in die Maske der Zielbitmap schreiben. Dazu setzen wir editMask auf TRUE und stellen den MixMode MM_SET ein, sonst werden die roten Kreise gerastert. MixMode = MM_COPY stellt anschließend den Ausgangszustand wieder her.



Tipp: Wir brauchen in diesem speziellen Fall suspendDraw nicht auf TRUE zu setzen, weil wir den undurchsichtigen Teil der Maske ergänzen und genau diesen Bereich mit Grafik füllen, so dass der Bildschirm automatisch auf dem korrekten Stand ist.

```
SUB DoDrawBitmap()
DIM h as Handle

' hier nicht erforderlich: DemoBitmap.suspendDraw = TRUE
  h = DemoBitmap2.GetBitmapHandle
  Drawbitmap h, 0, 0

  DemoBitmap.editMask = TRUE
  graphic.mixMode = MM_SET
  DrawBitmap h, 0, 0
  graphic.mixMode = MM_COPY
  DemoBitmap.editMask = FALSE

' hier nicht erforderlich: DemoBitmap.suspendDraw = FALSE

End SUB
```

5.2.5 Arbeit mit Paletten

5.2.5.1 Überblick

Unter GEOS bzw. R-BASIC werden Bitmaps mit folgenden Farbtiefen unterstützt:

Bits pro Pixel	Farben	Anmerkung
1	2	Immer Schwarz/Weiß
4	16	Von R-BASIC nicht unterstützt
8	256	Palette möglich
24	True Color	

Eine Bitmap mit 24 Bit pro Pixel enthält für jedes Pixel 3 Byte, je eines für die Farben Rot, Grün und Blau. Da jedes Byte die Werte 0 bis 255 annehmen kann ergeben sich etwa 16,8 Millionen mögliche Farben.
Wenn eine Bitmap weniger als 3 Byte pro Pixel speichert muss das System entscheiden, welche der über 16 Millionen möglichen Farben dargestellt werden sollen. Das wird über eine sogenannte Farbpalette realisiert. Die Palette ist eine Liste von bis zu 256 Einträgen zu je drei Byte - jeweils eins für Rot, Grün und Blau. Der "Farbwert" des Pixels entspricht dann der Nummer (dem sogenannten Index) des Eintrags in der Liste. Die Zählung beginnt dabei immer mit Null.

Zur Verwaltung der Palette sind in R-BASIC die folgenden Strukturen definiert:

```
STRUCT PaletteEntry
  rt, gn, bl as BYTE
  End Struct

STRUCT FullPalette
  item[255] as PaletteEntry
  END Struct
```

PaletteEntry enthält einen einzelnen Paletteneintrag, FullPalette enthält die vollständige Palette einer 256-Farb-Bitmap. Erlaubte Werte für den Index sind 0 bis 255. Die Palettendaten werden in der GEOS-Bitmap selbst gespeichert. R-BASIC erlaubt den Zugriff auf die Palette und deren Änderung. Wenn die Bitmap keine eigene Palette hat nutzt das System die GEOS-Standard-Palette. Dann kann R-BASIC die Farben nicht ändern.

Um eine Bitmap mit Palette anzulegen muss im vierten Parameter (flags) der Instancevariablen bitmapFormat das Bit 1 (zugehöriger Wert: 2, Konstante BF_PALETTE) gesetzt sein. Das System belegt dann die Palettendaten der Bitmap mit der Standardpalette. Diese Daten können später von R-BASIC aus geändert werden. Das passiert individuell für jede Bitmap, so dass Sie in einem Programm mehrere Bitmaps mit verschiedenen Paletten gleichzeitig anzeigen können.

```
' Parameter flags: Transparenz und Palette
  bitmapFormat = 640, 480, 8  ' keine eigene Palette
                              ' nutzt Standardpalette
  bitmapFormat = 640, 480, 8, BF_PALETTE   ' Palette
  bitmapFormat = 640, 480, 8, BF_PALETTE + BF_MASK
        ' Palette + Transparenz
```

Die Verwendung einer Palette ist nur bei 8-Bit-Bitmaps sinnvoll. R-BASIC unterstützt zwar den Zugriff auf die Palette einer schwarz/weiß Bitmap, das System ignoriert die Palettendaten aber. Es zeichnet monochrome Bitmaps immer in schwarz/weiß.

5.2.5.2 Zugriff auf die Farbpalette

GetFullPalette

Die Methode GetFullPalette liest die Palette einer Bitmap aus. Enthält die Palette der Bitmap weniger als 256 Einträge werden die restlichen Einträge mit Null belegt. Enthält die Bitmap keine Palette kommt es zu einem Laufzeitfehler.

Syntax: **<pal>** = **<obj>**.**GetFullPalette**
<pal>: Variable vom Typ FullPalette

SetFullPalette

Die Methode SetFullPalette belegt die Palette einer Bitmap. Das Objekt stellt sich automatisch neu dar. Enthält die Palette der Bitmap weniger als 256 Einträge werden die restlichen Einträge ignoriert. Enthält die Bitmap keine Palette kommt es zu einem Laufzeitfehler.

Syntax: **<obj>**.**SetFullPalette** **<pal>**
<pal>: Variable oder Ausdruck vom Typ FullPalette

GetPaletteEntry

Die Methode GetPaletteEntry liest einen einzelnen Paletteneintrag einer Bitmap aus. Enthält die Bitmap keine Palette kommt es zu einem Laufzeitfehler.

Syntax: **<entry>** = **<obj>**.**GetPaletteEntry** (**index**)
<entry>: Variable vom Typ PaletteEntry
index: Index des auszulesenden Paletteneintrags. Es muss gelten 0 <= index < Anzahl der Paletteneinträge der Bitmap, ansonsten kommt es zu einem Laufzeitfehler.

SetPaletteEntry

Die Methode GetPaletteEntry setzt einen einzelnen Paletteneintrag einer Bitmap aus. Das Objekt stellt sich aber nicht automatisch neu dar. Sie müssen dazu die Methode Redraw aufrufen. Enthält die Bitmap keine Palette kommt es zu einem Laufzeitfehler.

Syntax: **<obj>.SetPaletteEntry <entry>, index**

<entry>: Variable oder Ausdruck vom Typ PaletteEntry

index: Index des auszulesenden Paletteneintrags. Es muss gelten 0 <= index < Anzahl der Paletteneinträge der Bitmap, ansonsten kommt es zu einem Laufzeitfehler.

Tipp: GetPaletteEntry und SetPaletteEntry laufen nur geringfügig schneller als SetFullPalette und GetFullPalette. Wenn Sie mehrere Paletteneinträge ändern wollen ist deshalb häufig effektiver, die komplette Palette zu holen, die zu ändern und sie dann komplett neu zu setzen.

Redraw

Die Methode Redraw (ausführliche Beschreibung siehe vorne) bewirkt, dass das Objekt die Bitmap neu auf den Bildschirm zeichnet. Der Aufruf der Methode ist notwendig, wenn Sie einen einzelnen Paletteneintrag geändert haben (Methode SetPaletteEntry).

5.2.5.3 Beispiele

Die Farbkonstanten von R-BASIC basieren auf der GEOS Standardpalette. Zum Beispiel hat BLACK den Wert Null, BLUE den Wert 1 und WHITE den Wert 15. Wenn Sie beispielsweise dem Palettenwert mit dem Index 1 die RGB-Werte der Farbe Weiß zuweisen, werden alle Pixel, die den Index 1 haben, nicht mehr blau, sondern weiß dargestellt. Auf diese Weise kann man die Farben einer Bitmap sehr schnell ändern.

Der folgende Code ersetzt den Paletteneintrag für die Farbe Schwarz (Index Null) durch einen Grauwert. Die erste Variante liest und setzt die vollständige Palette. Die Methode SetFullPalette zeichnet die Bitmap automatisch neu. In der zweiten Variante lesen und schreiben wir genau einen Paletteneintrag. Weil SetPaletteEntry die Bitmap nicht neu zeichnet müssen wir die Methode Redraw aufrufen.

```
SUB ModifyBlack()
DIM pal as FullPalette
  pal = DemoBitmap.GetFullPalette
  pal.item(0).rt = 120
  pal.item(0).gn = 120
  pal.item(0).bl = 120
  DemoBitmap.SetFullPalette pal
End SUB
```

```
SUB ModifyBlack2()
DIM pe as PaletteEntry
  pe = DemoBitmap.GetPaletteEntry (0)
  pe.rt = 120
  pe.gn = 120
  pe.bl = 120
  DemoBitmap.SetPaletteEntry pe, 0
  DemoBitmap.Redraw
End SUB
```

Der folgende Code senkt alle Farbwerte der Palette auf 80% ab. Dadurch wird das Bild deutlich dunkler.

```
SUB MakeDarker()
DIM pal AS FullPalette
DIM n
  pal = DemoBitmap.GetFullPalette
  FOR n = 0 TO 255
    ' Rot, Grün und Blauwert verringern
    pal.item(n).rt = 0.8 * pal.item(n).rt
    pal.item(n).gn = 0.8 * pal.item(n).gn
    pal.item(n).bl = 0.8 * pal.item(n).bl
  NEXT n
  DemoBitmap.SetFullPalette pal
End Sub
```

Das folgende Beispiel tauscht die Palettenwerte für die Farben Blau und Weiß. Alles was bisher weiß war erscheint dann blau und umgekehrt. Die erste Variante verwendet die Methoden GetPaletteEntry und SetPaletteEntry. Deswegen ist der Aufruf der Methode Redraw erforderlich. Variante Zwei liest und schreibt die vollständige Palette. Der Aufruf von SetFullPalette stellt die Bitmap automatisch mit den geänderten Farben dar.

```
SUB SwitchColors()
DIM pe1, pe2 AS PaletteEntry

  ' originale Palettenwerte holen
  pe1 = DemoBitmap.GetPaletteEntry(WHITE)
  pe2 = DemoBitmap.GetPaletteEntry(LIGHT BLUE)

  ' Jeweils dem anderen Farbwert zuweisen
  DemoBitmap.SetPaletteEntry(pe1, LIGHT_BLUE)
  DemoBitmap.SetPaletteEntry(pe2, WHITE)

  ' Objekt neu zeichnen. Das passiert nicht automatisch!
  DemoBitmap.Redraw
End SUB
```

```
SUB SwitchColors2()
DIM pal as FullPalette
DIM pe AS PaletteEntry

  pal = DemoBitmap.GetFullPalette
  pe = pal.item(WHITE)
  pal.item(WHITE) = pal.item(LIGHT BLUE)
  pal.item(LIGHT BLUE) = pe
  DemoBitmap.SetFullPalette pal

End SUB
```

Betrachten wir nun den folgenden Code. Wenn wir ihn ausführen nachdem wir die SUB SwitchColors aufgerufen haben, sollte eine weiße Line erscheinen, weil dem Index der Farbe Blau (LIGHT_BLUE) jetzt die RGB-Werte der Farbe Weiß zugeordnet sind.

```
Line 10, 10, 200, 200, LIGHT_BLUE
```

Wir sehen jedoch eine blaue Linie. Erst wenn wir das Fenster mit der Bitmap auf dem Schirm verschieben (und sich die Bitmap deswegen neu zeichnen muss) wird die Linie weiß. Warum? Jeder Grafikbefehl (auch Textausgaben) gehen nicht nur in die Bitmap, sondern parallel dazu auch direkt auf den Schirm. Bei der Ausgabe auf den Schirm wird die geänderte Palette aber nicht berücksichtigt, sie ist nur der Bitmap bekannt. Deswegen sollten sie während der Ausgabe von Grafik und Text in eine Bitmap mit geänderter Palette die parallele Ausgabe auf den Bildschirm abschalten. Für diesen Zweck gibt es die Instancevariable **suspendDraw**. Der folgende Code erzeugt sofort gewünschte weiße Linie:

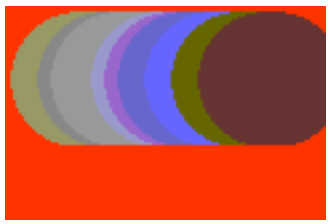
```
DemoBitmap.suspendDraw = TRUE
Line 10, 10, 200, 200, LIGHT_BLUE
' <hier weitere Grafik- und Textausgaben>
DemoBitmap.suspendDraw = FALSE
```

DemoBitmap.suspendDraw = TRUE schaltet die parallele Ausgabe der Grafik auf den Schirm ab. Die Bitmap wird unsichtbar im Hintergrund beschrieben. DemoBitmap.suspendDraw = FALSE hebt die Suspendierung auf und zeichnet die Bitmap neu auf den Schirm, so dass alle Änderungen sichtbar werden.

Eine ähnliche Situation tritt auf, wenn wir eine Bitmap in eine andere Bitmap zeichnen, falls die Paletten nicht übereinstimmen. Oder wir zeichnen eine RGB-Grafik (Bitmap oder Grafikbefehl) in die 8-Bit Bitmap. Das System ersetzt dann die nicht in der Palette befindlichen Farben durch "ähnliche" Farben, die in der Palette der Zielbitmap vorhanden sind. Auf dem Schirm erscheinen jedoch die originalen Farben. Auch hier sollten wir suspendDraw einsetzen. Der folgende Code geht davon aus, dass DemoBitmap1 der Screen ist.

```
SUB CopyBitmap2ToScreen
DIM h as Handle
  DemoBitmap1.suspendDraw = TRUE
  h = DemoBitmap2.GetBitmapHandle
  DrawBitmap h, 10, 20                ' Handle, Koordinaten
  DemoBitmap1.suspendDraw = FALSE
End SUB
```

Bitmap mit geänderter Palette (DemoBitmap2)



Farben, nachdem die Bitmap links in die Bitmap DemoBitmap1 mit Standardpalette gezeichnet wurde.

5.2.6 Direktzugriff auf die Bitmapdaten

Es ist möglich, auf die einzelnen Pixelzeilen einer Bitmapgrafik direkt zuzugreifen. Dazu kann man mit der Methode PokeLine eine komplette Grafikzeile in den virtuelle RAM schreiben, dort die einzelnen Pixel modifizieren und die Zeile dann mit der Methode PeekLine zurück in die Bitmap kopieren. Insbesondere ist es auf diese Weise möglich, die Maskendaten einer 24-bit Bitmap zu ändern.
Man sollte sich jedoch der Tatsache bewusst sein, dass die Manipulation von einigen Tausend Pixeln sehr lange dauern kann.

PokeLine

Die Methode PokeLine kopiert eine komplette Pixelzeile aus der Bitmap in den virtuellen R-BASIC RAM. Falls die Bitmap eine Maske enthält werden die zur Zeile gehörenden Maskendaten ebenfalls kopiert.

Syntax: **<obj>.PokeLine adr, line**
adr: Adresse im virtuellen RAM (0 ... 65535)
Es werden so viele Bytes geschrieben wie die Zeile enthält
line: Zeilennummer der in den RAM zu schreibenden Zeile
Erlaubte Werte: 0 .. Höhe - 1

PeekLine

Die Methode PeekLine kopiert eine komplette Pixelzeile aus dem virtuellen R-BASIC-RAM in die Bitmap des Objekts. Falls die Bitmap eine Maske enthält werden die zur Zeile gehörenden Maskendaten ebenfalls überschrieben. Die Bitmap stellt sich **nicht** neu dar, Sie müssen dazu die Methode Redraw aufrufen.

Syntax: **<obj>.PeekLine adr, line**
adr: Adresse im virtuellen RAM (0 ... 65535)
Es werden so viele Bytes aus dem RAM gelesen, wie die Zeile fasst.
line: Zeilennummer der zu beschreibenden Bitmap-Zeile
Erlaubte Werte: 0 .. Höhe - 1

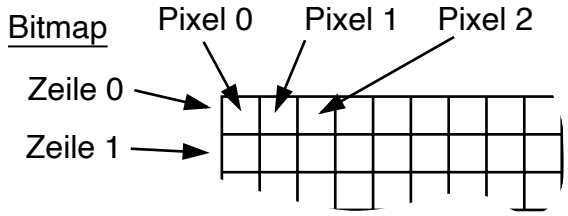
Beispiel: Der folgende Code kopiert die die ersten 50 Zeilen einer Bitmap in die Zeilen 100 bis 149. Die Adresse im virtuellen RAM ist egal, deswegen wählen wir Adresse 0.

```
DIM n
FOR n = 0 TO 49
  DemoBitmap.PokeLine 0, n
  DemoBitmap.PeekLine 0, 100+n
NEXT n
DemoBitmap.Redraw
```

Redraw

Die Methode Redraw (ausführliche Beschreibung siehe vorne) bewirkt, dass das Objekt die Bitmap neu auf den Bildschirm zeichnet. Der Aufruf der Methode ist notwendig, wenn Sie eine Bitmapzeile manuell verändert haben (Methode PeekLine). Falls die Bitmap eine Maske (Transparenzebene) hat und Sie mit der Methode PeekLine den von der Maske als durchsichtig markierten Bereich geändert haben müssen Sie Redraw mit dem Parameter TRUE aufrufen, damit der Hintergrund der Bitmap neu dargestellt wird und die geänderte Maske erkennbar wird.

Aufbau der Bitmapdaten

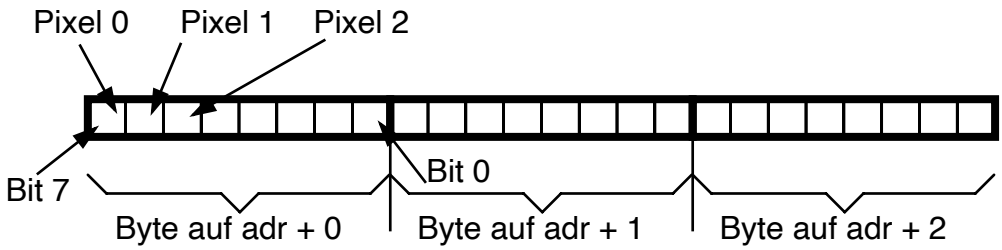


Um die Pixelzeilen bearbeiten zu können müssen Sie die Struktur der Bitmapdaten kennen. Wir nehmen zunächst an, dass die Bitmap keine Maske hat. Je nach Farbtiefe wird eine unterschiedliche Anzahl von Bits für ein Pixel benötigt. Daraus ergibt sich die Anzahl der Bytes für eine Pixelzeile.

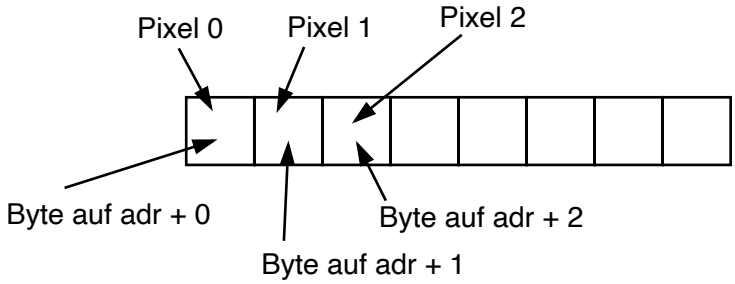
Farbtiefe	Bits pro Pixel	Bytes pro Zeile
Monochrom	1	8 Pixel werden zu einem Byte zusammengefasst. Es wird auf ganze Bytes gerundet. anzahl = INT ((breite+7) / 8)
256 Farben	8	anzahl = breite
True Color	24	anzahl = 3 * breite

Grundsätzlich liegen die in der Zeile links liegenden Pixel auf den niedrigen Adressen im virtuellen RAM. Bei monochromen Bitmaps liegt das ganz linke Pixel auf dem höchstwertigen Bit des Bytes. True-Color Bitmaps speichern die Farbwerte in der Reihenfolge Rot-Grün-Blau. Daraus ergeben sich die folgenden Zusammenhänge. In den Bildern bezeichnet "adr" Adresse "adr", die an die Methoden PeekLine bzw. PokeLine übergeben wurde.

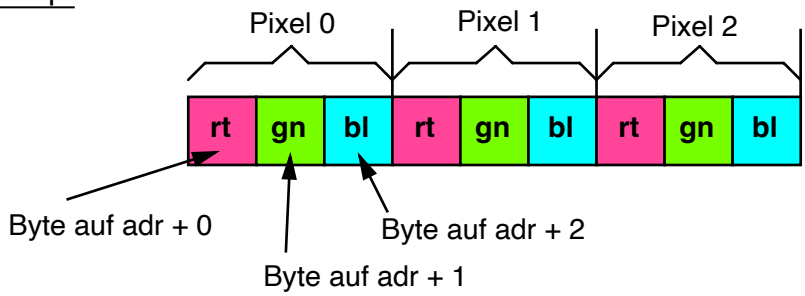
Monochrome Bitmap:
Bit 7 ist das höherwertigste Bit, Bit 0 ist das niederwertigste Bit.



256 Farben Bitmap:



True Color Bitmap:

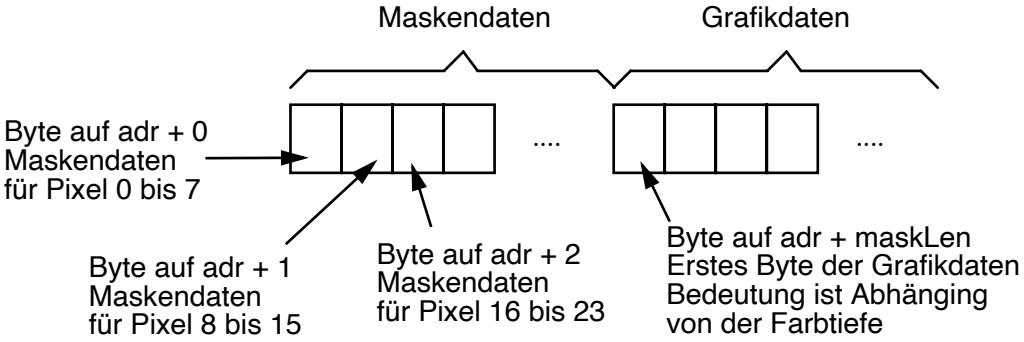


Aufbau der Bitmapdaten mit Maske

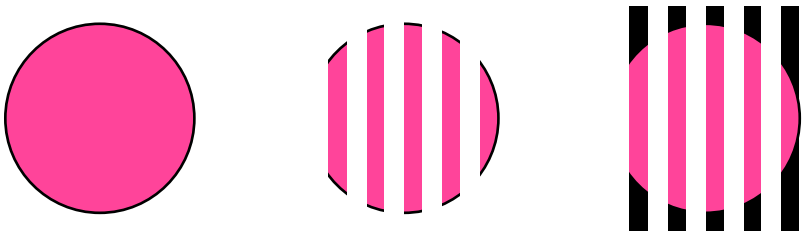
Wenn die Bitmap eine Maske hat sind die Maskendaten für jede Zeile direkt vor den Grafikdaten der Zeile angeordnet. Der Aufbau der Maskendaten entspricht dem einer monochromen Bitmap. Ein gesetztes Bit bedeutet, dass die Grafikdaten des Pixels dargestellt werden sollen. Ist das Bit nicht gesetzt (also Null) ist das Pixel transparent. Die Größe der Maskendaten (Anzahl der Bytes) berechnet sich zu:

maskLen = INT ((breite+7) / 8)

Die Methoden PokeLine und PeekLine kopieren jeweils sowohl die Maskendaten als auch die Grafikdaten.



Das folgende Codefragment belegt die Maskendaten einer Bitmap mit dem Bitmuster 00001111 (= 15 dezimal). Dadurch erscheint die Grafik gestreift. Die Farbtiefe der Grafik spielt dabei keine Rolle, da sie die Größe der Maskendaten nicht beeinflusst.



Hinweis: Je nachdem, welche Grafikdaten die Maske vorher verdeckt hat kann das linke oder das rechte Bild entstehen.

```
DIM width, height, x, y, masklen

  width = DemoBitmap.bitmapformat(0)
  height = DemoBitmap.bitmapformat(1)
  maskLen = INT ( (width+7)/8)

  FOR y = 0 TO height-1
    DemoBitmap.PokeLine 0, y
    FOR x = 0 TO maskLen - 1
        Poke x, 15                ' &B00001111
    NEXT x
    DemoBitmap.PeekLine 0, y
  NEXT y
  DemoBitmap.Redraw
```

Ein etwas komplexeres Beispiel

Die Tatsache, dass das Format der Maskendaten identisch mit dem einer monochromen Bitmap ist ermöglich es, auf relativ einfache Weise die Maske einer 24 Bit Bitmap zu bearbeiten. Das System unterstützt das leider nicht.

Nehmen wir an, wir haben eine 24 Bit Bitmap der Größe 256 x 192 Pixel, die eine Maske enthält (Objekt DemoBitmap). Die Idee hinter dem folgenden Code ist, eine ebenso große monochrome Bitmap ohne (!) Maske (DemoBitmap2) zu verwenden, diese mit Grafikbefehlen zu bearbeiten und dann die Daten der monochromen Bitmap in die Maske der 24 Bit Bitmap zu kopieren. Dieses

Vorgehen setzt voraus, dass beide Bitmaps exakt die gleichen Abmessungen haben.

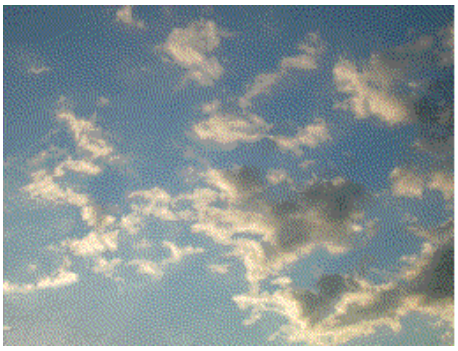
Der folgende Code bearbeitet zunächst die monochromen Bitmap. Weiße Pixel werden später transparent, schwarze Pixel werden undurchsichtig.

```
  Screen = DemoBitmap2
  Paper WHITE
  Cls                          ' komplett transparent
  FillEllipse 64, 32, 192, 160, Black
  FillRect 32, 64, 224, 128, Black
```

Nun holen wir uns jede einzelne Pixelzeile der 24 Bit Bitmap in den virtuellen RAM und kopieren die Daten der monochromen Bitmap an die gleiche Stelle. Weil beide Bitmaps die gleiche Größe haben werden damit nur die Maskendaten der 24 Bit Bitmap überschrieben. Dann kopieren wir die geänderte Pixelzeile zurück in die 24 Bit Bitmap. Abschießend rufen wir die Redraw-Methode mit dem Parameter TRUE auf um die Änderungen sichtbar zu machen.

```
DIM x, y, maskLen

  maskLen = INT ((256+7)/8)

  FOR y = 0 TO 191
    DemoBitmap.PokeLine 0, y
    DemoBitmap2.PokeLine 0, y
    DemoBitmap.PeekLine 0, y
  NEXT y

  DemoBitmap.Redraw TRUE
```

Dieser Code erzeugt aus dem linken das rechte Bild.

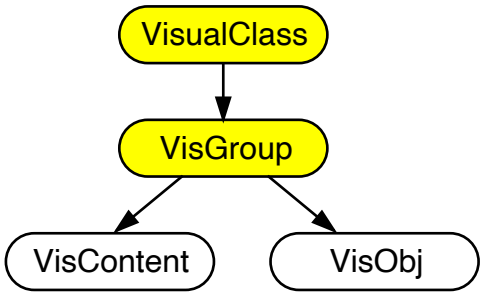


Hinweis: Um Speicherplatz zu sparen wurden die Grafiken für dieses Handbuch auf 8 Bit heruntergerechnet und etwas verkleinert.

5.3 VisGroup

Die VisGroup Class ist die Superclass für die VisContent und VisObj Class. Sie implementiert alle gemeinsamen Fähigkeiten dieser beiden Klassen. Dazu gehören im Wesentlichen die Ausgabe von Grafik und die Verwaltung von Children. Sie können in R-BASIC keine Objekte dieser Klasse anlegen.

Abstammung



Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
OnDraw	OnDraw = **<Handler>**	nur schreiben
defaultColor	defaultColor = **fg**, **bg**	lesen, schreiben
clipDrawing	clipDrawing = TRUE	lesen, schreiben
buffered	buffered = TRUE	lesen, schreiben
bufferedDataSize	bufferedDataSize = **<Wert>**	lesen, schreiben
customManageChildren	customManageChildren = TRUE | FALSE	lesen, schreiben
visPosition	visPosition = **xPos**, **yPos**	lesen, schreiben
visSize	visSize = **width**, **height**	lesen, schreiben
xPosition, yPosition	—	nur lesen
xSize, ySize	—	nur lesen
visSizeOptions	visSizeOptions = **<Wert>**	lesen, schreiben
visSizeFlags	visSizeFlags = **<Wert>**	lesen, schreiben
visMinimumSize	visMinimumSize = **minX, minY**	lesen, schreiben
visOrientVertically	visOrientVertically = TRUE | FALSE	lesen, schreiben
visChildJustification	visChildJustification = **jHor**, **jVert**	lesen, schreiben
visChildSpacing	visChildSpacing = **childSp** , **wrapSp**	lesen, schreiben
visSpacingIncludeEnds	visSpacingIncludeEnds = TRUE | FALSE	lesen, schreiben
visMargins	visMargins = **left** ,**top** ,**right** ,**bottom**	lesen, schreiben
allowChildrenToWrap	allowChildrenToWrap = TRUE | FALSE	lesen, schreiben
visWrapCount	visWrapCount = **numWert**	lesen, schreiben

Methoden:

Methode	Aufgabe
Dirty	Weist das Objekt an, sich neu darzustellen, indem der OnDraw-Handler aufgerufen wird.
Redraw [TRUE]	Weist das Objekt an, sich neu darzustellen.
MarkInvalid	Berechnet die Geometrie neu und löst ein Neuzeichnen aus

Action-Handler-Typen:

Handler-Typ	Parameter
DrawAction	(sender as object, width, height as word)

Instance-Variablen und Methoden für SDK-Programmierer:

Variable/Methode	Syntax im UI-Code	Im BASIC-Code
visClassAttrs	visClassAttrs = **toSet**, **toClear**	lesen, schreiben
visCompGeoAttrs	visCompGeoAttrs = **toSet**, **toClear**	lesen, schreiben
visCompDimensionAttrs	visCompDimensionAttrs = **toSet**, **toClear**	lesen, schreiben
MarkInvalid2	— (Methode)	nur schreiben

5.3.1 Ausgabe von Grafik

Die primäre Aufgabe von VisualClass-Objekten ist die Ausgabe von Grafik auf den Bildschirm. VisContent-Objekte und VisObj-Objekte haben dazu einen OnDraw-Handler, der automatisch gerufen wird, wenn sich das Objekt auf dem Bildschirm neu darstellen muss. Alternativ können Sie in einem "gepufferten" Modus arbeiten. Dabei wird der OnDraw-Handler nur einmalig gerufen und die Grafik intern in einem GString gespeichert. Bei Bedarf wird diese dann ausgegeben. Das ist effizienter als der ständige Aufruf des in BASIC geschrieben OnDraw-Handlers, allerdings ist es weniger flexibel.
Eine ausführliche Beschreibung der dahinter stehenden Konzepte finden Sie beim Canvas-Objekt, im Kapitel 4.16 des Objekthandbuchs. Der einzige Unterschied zum Canvas ist, dass man bei VisContent und bei VisObj-Objekte einstellen kann, ob die Grafik an den eigenen Grenzen abgeschnitten wird (Clipping, Instancevariable clipDrawing), beim Canvas-Objekt jedoch nicht. Außerdem steht die Methode Redraw für Canvas-Objekte nicht zur Verfügung.

Beim Aufruf des OnDraw-Handlers wird das Objekt automatisch zum Screen, das heißt alle Grafikausgaben gehen an die Stelle, an der das Objekt dargestellt wird. Der Koordinatenursprung ist dabei immer die linke obere Ecke des Objekts.
Die dem OnDraw-Handler übergebenen Parameter **width** und **height** enthalten die Breite und die Höhe des Objekts. Da die Koordinaten bei Null anfangen, ist die

Maximale Koordinate, die noch innerhalb des Objekts liegt, jeweils um 1 kleiner. Die globalen Variablen **MaxX** und **MaxY** enthalten diese Werte.

Verfügbare Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
OnDraw	OnDraw = **<Handler>**	nur schreiben
defaultColor	defaultColor = **fg**, **bg**	lesen, schreiben
clipDrawing	clipDrawing = TRUE	lesen, schreiben
buffered	buffered = TRUE	lesen, schreiben
bufferedDataSize	bufferedDataSize = **<Wert>**	lesen, schreiben

Methoden:

Methode	Aufgabe
Dirty	Weist das Objekt an, sich neu darzustellen, indem der OnDraw-Handler aufgerufen wird.
Redraw [TRUE]	Weist das Objekt an, sich neu darzustellen.

Action-Handler-Typen:

Handler-Typ	Parameter
DrawAction	(sender as object, width, height as word)

Kurzbeschreibung der Instancevariablen

Eine ausführliche Beschreibung finden Sie beim Canvas-Objekt, im Kapitel 4.16 des Objekthandbuchs.

OnDraw

Die Instance-Variable **OnDraw** enthält den Namen des Handlers, der die Grafik zeichnen soll. Dieser muss als **DrawAction** vereinbart sein.

Syntax UI- Code: **OnDraw** = **<Handler>**
Schreiben: **<obj>**.**OnDraw** = **<Handler>**

Die Parameter **width** und **height** enthalten die Breite und die Höhe des Objekts. Die globalen Variablen **MaxX** und **MaxY** enthalten die maximale Koordinate, die noch innerhalb des Objekts liegt. Sie ist jeweils um 1 kleiner als width bzw. height.

Weisen Sie zur Laufzeit einen neuen OnDraw-Handler zu, so stellt sich das Objekt automatisch neu dar. Beachten Sie, dass dabei der Hintergrund nicht gelöscht bzw. die bereits vorhandene Grafik nicht vom Schirm genommen wird.

defaultColor

Die Instance-Variable **defaultColor** enthält die Farben, die beim Aufruf des OnDraw Handlers eingestellt werden.

Syntax	UI-Code:	**defaultColor** = **fg**, **bg** fg: Vordergrund (foreground) bg: Hintergrund (background) fg und bg müssen Indexfarben sein. RGB-Farben sind nicht zulässig.	
	Lesen:	**<numVar>** = **<obj>**.**defaultColor** (0)	' fg
		<numVar> = **<obj>**.**defaultColor** (1)	' bg
	Schreiben:	**<obj>**.**defaultColor** = **fg**, **bg**	

buffered

Die Instancevariable **buffered** legt fest, ob das Objekt die anzuzeigende Grafik zwischenspeichert (buffered = TRUE, "gepufferter" Modus) oder nicht (buffered = FALSE, normaler Modus). FALSE ist der Defaultwert.

Syntax	UI- Code:	**buffered** = TRUE
	Schreiben:	**<obj>**.**buffered** = TRUE I FALSE
	Lesen:	**<numVar>** = **<obj>**.**buffered**

Achtung! Im gepufferten Modus zeichnet das Objekt seine Grafik sofort, ohne Umweg über den BASIC-Handler. Das geht deutlich schneller, hat aber Konsequenzen, wenn sich Objekte überlappen.

Solange alle Objekte, die sich überlappen, im gleichen Modus arbeiten, gibt es keine Probleme, die Objekte werden in der richtigen Reihenfolge gezeichnet.
Überlappen sich aber Objekte, von denen einige im gepufferten Modus und andere im normalen Modus arbeiten, so werden immer zuerst alle Objekte gezeichnet, die sich im gepufferten Modus befinden. Objekte im normalen Modus werden danach, also über den anderen Objekten gezeichnet, unabhängig davon, in welcher Reihenfolge sie als Children im UI-Code vereinbart sind. Das liegt daran, dass die BASIC-Handler der Objekte im normalen Modus erst ausgeführt werden, wenn die Objekte im gepufferten Modus fertig sind.

VisText-Objekte haben keine BASIC-OnDraw-Handler. Sie arbeiten intern quasi wie im gepufferten Modus. Wenn Sie also ein VisText-Objekt über ein VisObj-Objekt legen wollen, muss das VisObj im gepufferten Modus arbeiten, sonst wird es über dem VisText gezeichnet. Folglich muss auch ein VisContent, wenn es einen OnDraw-Handler hat, im gepufferten Modus arbeiten, falls es VisText-Objekte als Children hat.

bufferedDataSize

Im gepufferten Modus fordert das Objekt Speicher (in einer Datei) an, um die darzustellende Grafik zu speichern. BufferedDataSize enthält die Information, wie groß der benötigte Speicher ungefähr (!) ist. Der Wert ist nicht kritisch, der Defaultwert ist DS_TINY.

Syntax	UI- Code:	**bufferedDataSize** = **<Wert>**
	Schreiben:	**<obj>**.**bufferedDataSize** = **<Wert>**
	Lesen:	**<numVar>** = **<obj>**.**bufferedDataSize**
	<Wert>:	numerische Konstante, siehe aus der Tabelle unten

Die folgende Tabelle enthält die zulässigen Werte:

Konstante	Wert	Zu erwartende Datenmenge
DS_TINY	0	nicht mehr als 10 .. 20 kByte
DS_SMALL	1	nicht mehr als 50 .. 100 kByte
DS_MEDIUM	2	nicht mehr als 500 kByte ... 1 MB
DS_LARGE	3	nicht mehr als 5 MByte
DS_HUGE	4	möglicherweise mehr als 5 MByte

clipDrawing

Die Instance-Variable clipDrawing enthält die Information ob das Objekt über seine eigenen Grenzen (Bounds) hinausschreiben darf, oder nicht. Der Defaultwert ist FALSE, das heißt, das Objekt kann an beliebige Stellen auf den Schirm schreiben.

Syntax	UI-Code:	**clipDrawing** = TRUE I FALSE
	Lesen:	**<numVar>** = **<obj>**.**clipDrawing**
	Schreiben:	**<obj>**.**clipDrawing** = TRUE I FALSE

Hinweise:

- Wenn Sie clipDrawing zur Laufzeit ändern, löst das kein Neuzeichnen des Objekts aus. Rufen Sie dazu eine der Methoden Redraw, Dirty oder MarkInvalid für das Objekt oder eines seiner Parents auf.
- Weisen Sie clipDrawing= TRUE zur Laufzeit zu, so löscht das nicht die Grafiken, die über den Rand hinausgehen.
- Sollten beim Verschieben des Objekts "Artefakte" zurückbleiben, haben sie über den Rand des Objekts geschrieben. Sie können das vermeiden, indem Sie clipDrawing im UI-Code auf TRUE setzen.

Dirty

Die Methode Dirty (engl: schmutzig) bewirkt, dass sich das Objekt neu darstellt, indem es seinen **OnDraw** Handler ruft. Die Dirty Methode arbeitet auch im gepufferten Modus. Das Objekt gibt die alte gepufferte Grafik automatisch frei und speichert die neue ab.

Syntax im BASIC Code: **<obj>.Dirty**

Redraw

Die Methode Redraw bewirkt, dass das Objekt sich neu auf den Bildschirm zeichnet. Im gepufferten Modus wird der gespeicherte GString neu ausgegeben, im ungepufferten Modus wird der OnDraw-Handler aufgerufen.

Syntax: **<obj>.Redraw** [**drawBackground**]
drawBackground: TRUE | FALSE (Default: FALSE)

Beispiel:

```
MyVisObj1.Redraw
MyVisObj1.Redraw  TRUE
```

DrawBackground = TRUE bewirkt, dass der Redraw-Befehl an das zugehörige View weitergeleitet wird. Damit wird zuerst der Hintergrund gelöscht und dann **alle** im View dargestellten Objekte neu gezeichnet. Je nach Komplexität der Darstellung und der Anzahl der Objekte kann das einen Moment dauern.

5.3.2 Manuelle Anordnung der Children

5.3.2.1 Größe und Position

Per Default verwaltet das GEOS-System die Anordnung und Größe der Objekte in einem visual Tree automatisch. Um die Children manuell zu positionieren, müssen Sie im VisContent-Objekt die Instancevariable **customManageChildren** auf TRUE setzen. Außerdem müssen Sie die Instancevariablen **visPosition** und **visSize** der VisObj-Objekte im visual Tree belegen.
Um die Größe des VisContent-Objekts festzulegen haben Sie neben dem Belegen der Instancevariablen visSize weitere Möglichkeiten, die im Kapitel 5.4.2 (View-Content Konfiguration) beschrieben sind.

Tipp:
Es ist meist komfortabler, zum Lesen der Werte von visPosition und visSize die für alle Objekte verfügbaren read-only Instancevariablen xPosition, yPosition, xSize bzw. ySize zu benutzen.

Im Ordner "Visual Class" finden Sie mehrere Beispiele zur Verwendung von customManageChildren, z.B. "VisObj privData Demo", "VisObj Keyboard Demo" und "Create Custom Managed VisObj".

Zugehörige Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
customManageChildren	customManageChildren = TRUE I FALSE	lesen, schreiben
visPosition	visPosition = **xPos**, **yPos**	lesen, schreiben
visSize	visSize = **width**, **height**	lesen, schreiben
xPosition, yPosition	—	nur lesen
xSize, ySize	—	nur lesen

customManageChildren

Die Instance-Variable customManageChildren legt fest, ob der Programmierer oder der Geometriemanager des GEOS-Systems die Anordnung der Children des Objekts steuert. Der Defaultwert für customManageChildren ist FALSE, d.h. der Geometriemanager des GEOS-Systems übernimmt die Anordnung der Children.
Sie müssen customManageChildren auf TRUE setzen, um die Anordnung der Children selbst zu kontrollieren.

Syntax	UI-Code:	**customManageChildren** = TRUE I FALSE
	Lesen:	**<numVar>** = **<obj>**.**customManageChildren**
	Schreiben:	**<obj>**.**customManageChildren** = TRUE I FALSE

Es ist möglich und zulässig, den Wert für customManageChildren in einem VisObj des visual Trees auf FALSE zu setzen, wenn im VisContent der Wert auf TRUE gesetzt ist - und umgekehrt. Das GEOS System versucht dann, Ihre Wünsche "so gut wie möglich" zu erfüllen. Ob die Ergebnisse dann Ihren Wünschen entsprechen, müssen Sie ausprobieren.

visPosition

Die Instance-Variable visPosition enthält die aktuelle Position des Objekts, relativ zu seinem VisContent.

Syntax UI-Code: **visPosition** = **xPos**, **yPos**
xPos: x-Position
yPos: y-Position
Lesen: **<numVar>** = **<obj>**.**visPosition**(0) ' xPos
<numVar> = **<obj>**.**visPosition**(1) ' yPos
Schreiben: **<obj>**.**visPosition** = **xPos**, **yPos** [, autoRedraw]
autoRedraw:
FALSE (Default): keine sofortige Neudarstellung
TRUE: sofortige Neudarstellung (Move-To-Funktion)

Per Default führt ein manuelles Verändern der visPosition nicht automatisch zum Neuzeichnen des Objekts an der neuen Position. Dazu müssen Sie die Methode MarkInvalid aufrufen.
Geben Sie als zusätzlichen Parameter TRUE an, so stellt sich das Objekt sofort neu dar, wobei Bereiche, die jetzt nicht mehr vor Objekt überdeckt sind, ebenfalls geupdatet werden. Sie verschieben das Objekt also sofort an seine neue Position.

Hinweise:

- Im Allgemeinen werden auch andere Objekte neu gezeichnet, wenn Sie autoRedraw = TRUE angeben. Häufig flackert es jedoch weniger, als wenn Sie MarkInvalid für das zugehörige VisContent aufrufen. Im Zweifel hilft hier nur Probieren.
- Sie sollten die visPosition-Werte für VisContent-Objekte nicht ändern. Das kann zu unerwarteten Ergebnissen, insbesondere einer Verschiebung der Position aller beteiligten Objekte führen.

visSize

Die Instance-Variable visSize enthält die aktuelle Größe des Objekts.

```
Syntax  UI-Code: visSize =  width, height
                 width:  Breite
                 height: Höhe
        Lesen:   <numVar> = <obj>.visSize(0)        ' Breite
                 <numVar> = <obj>.visSize(1)        ' Höhe
        Schreiben:    <obj>.visSize =  width, height   [, autoRedraw ]
                 autoRedraw:
                      FALSE (Default): keine sofortige Neudarstellung
                      TRUE: sofortige Neudarstellung
```

Per Default führt ein manuelles Verändern von visSize nicht automatisch zum Neuzeichnen des Objekts in der neuen Größe. Dazu müssen Sie die Methode MarkInvalid aufrufen.
Geben Sie als zusätzlichen Parameter TRUE an, so stellt sich das Objekt sofort neu dar, wobei Bereiche, die jetzt nicht mehr vor Objekt überdeckt sind, ebenfalls geupdatet werden. Beachten Sie, dass dabei im Allgemeinen auch andere Objekte neu gezeichnet werden.

xPosition, yPosition

Diese Werte liefern die aktuelle Position des Objekts.

```
Syntax  Lesen:   <numVar> = <obj>.xPosition
                 <numVar> = <obj>.yPosition
```

xSize, ySize

Diese Werte liefern die aktuelle Größe des Objekts in Pixeln.

```
Syntax  Lesen:   <numVar> = <obj>.xSize
                 <numVar> = <obj>.ySize
```

5.3.2.2 Wenn sich die Children überlappen

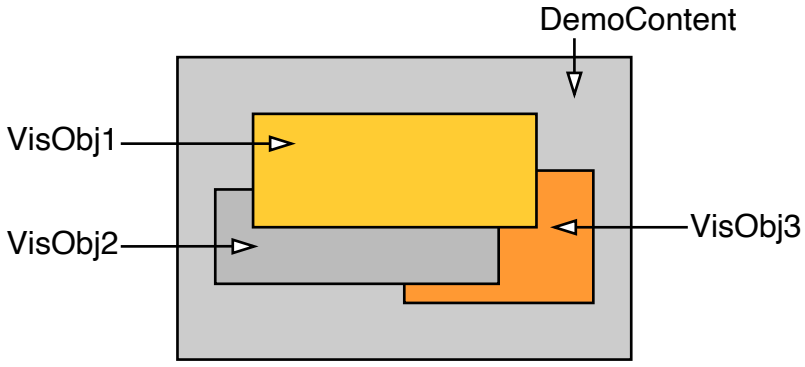
Im customManageChildren-Modus kommt es häufig vor, dass sich Objekte gegenseitig überlappen. Natürlich sollen in diesen Fällen die "oben" liegenden Objekt auch die Maus-Ereignisse erhalten. Deshalb zeichnet R-BASIC die Objekte in der entgegengesetzten Reihenfolge zu der, in der sie im UI-Code definiert wurden. GEOS gibt nämlich die Mausereignisse von Objekt zu Objekt weitergibt, und zwar in der Reihenfolge, in der sie im UI-Code vereinbart sind. Das zuerst vereinbarte Objekt bekommt das Mausereignis zuerst zu sehen. Damit es "oben" liegt, muss es also zuletzt gezeichnet werden.

Achtung! Wenn sich Objekte überlappen, von denen einige im gepufferten Modus und andere im normalen (ungepufferten) Modus arbeiten, ändert sich die Zeichenreihenfolge, nicht aber die Reihenfolge, in der sie Mausereignisse erhalten. Eine genauere Erläuterung dazu finden Sie weiter oben im Kapitel 5.3.1 (Ausgabe von Grafik) bei der Beschreibung der Instancevariablen buffered. Wir setzen daher im Folgenden voraus, dass alle Objekte im gleichen Modus arbeiten.
VisText-Objekte zählen dabei als Objekte, die im gepufferten Modus arbeiten!

Nehmen wir an, wir haben ein VisContent mit 3 Children.

```
VisContent DemoContent
  Children = VisObj1, VisObj2, VisObj3
  customManageChildren = TRUE
  ...
End OBJECT
```

Größe und Position der Children seien so, dass sie sich überlappen. Da sie von R-BASIC in der "umgekehrten" Reihenfolge gezeichnet werden, kann sich z.B. folgendes Bild ergeben.

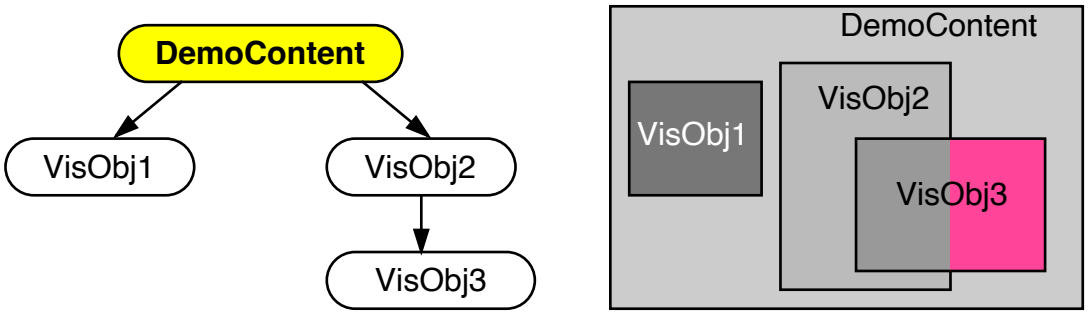


Damit bekommt jedes Objekt genau dann die Mausklicks zu sehen, wenn der Nutzer in den sichtbaren Bereich des Objekts klickt.

Natürlich können Sie auch hier Objekt-Trees verwenden. Allerdings müssen Sie wissen, dass Mausklicks immer vom den Parents an die Children weitergeleitet werden. Children, die ganz oder teilweise außerhalb ihrer Parents gezeichnet werden, erhalten in den außerhalb liegenden Bereichen keine Mausklicks. **Sie müssen selbst dafür sorgen, das jedes Objekt vollständig innerhalb der**

Grenzen seines Parents dargestellt wird, sonst erhält es möglicherweise Mausklicks nicht oder nur teilweise.

Im folgenden Beispiel ist VisObj3 Child von VisObj 2. Im rot markierten Bereich leitet das Content die Mausklicks nicht an VisObj2 weiter. Deswegen erhält VisObj3 diesem Bereich auch keine Mausklicks.



Ein schlechtes Beispiel. VisObj3 erhält als Child von VisObj2 im roten Bereich keine Mausklicks.

5.3.3 Automatische Anordnung der Children

5.3.3.1 Überblick

Per Default verwaltet das GEOS-System die Anordnung und Größe der Objekte in einem visual Tree automatisch. Dabei werden die Children nebeneinander (oder untereinander) mit einem vorgegebenen Abstand angeordnet. Sie kennen das von den Dateien im GeoManager oder von den Vorschaubildern im GrafikViewer Gonzo.

Solange der Geometriemanager die Größe und Position der Objekte im visual Tree verwaltet, müssen Sie keine Werte für die im letzten Kapitel besprochenen Instancevariablen **visPosition** und **visSize** angeben. Tun Sie es doch, wird der Geometriemanager die Werte überschreiben. Objekte, die **keine Children** haben, müssen allerdings ihre Größe kennen, so dass Sie hier **visSize belegen** müssen.

Ansonsten steuern Sie das Verhalten des Geometriemanagers, indem Sie die folgenden Instancevariablen belegen.

Mit Hilfe der Instancevariablen **visSizeOptions** und **visSizeFlags** können Sie, getrennt nach x- und y-Richtung, festlegen, ob das Objekt eine feste Größe hat, seine Größe dem Platzbedarf seiner Children anpasst oder sich an der Größe des parent-Objekts orientiert. Mit **visMinimumSize** können Sie für Objekte variabler Größe festlegen, dass das Objekt nicht beliebig klein werden kann, auch wenn die Children weniger Platz erfordern.

Mit den Instancevariablen **visChildJustification** und **visOrientVertically** können Sie die Ausrichtung der Children-Objekte festlegen. Sie können z.B. festlegen ob Sie nebeneinander oder übereinander angeordnet werden sollen, ob sie zentriert, linksbündig oder über die verfügbare Breite verteilt werden sollen, oder ob sie sich den Platz gleichberechtigt aufteilen sollen.

Die Instancevariablen **visChildSpacing**, **visSpacingIncludeEnds** und **visMargins** bestimmen den Platz zwischen benachbarten Objekten.

Schließlich erlauben **AllowChildrenToWrap** und **visWrapCount**, dass die Children einer neuen Reihe (oder Spalte) angeordnet werden, wenn der verfügbare Platz in die entsprechende Richtung nicht ausreicht.

Ändern Sie die in diesem Kapitel besprochenen Instancevariablen zur Laufzeit, so werden die Objekte und ihre Children nicht sofort an ihrer neuen Position bzw. in der neuen Größe gezeichnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen. Der Vorteil dieser Vorgehensweise ist, dass Sie mehrere Änderungen an der Geometrie Ihres visual Trees vornehmen können, ohne dass der Bildschirm mehrfach aktualisiert wird und so unnötig flackert.

Hinweis zur Fehlersuche

Prinzipiell sind die durch die genannten Instancevariablen gebotenen Möglichkeiten beliebig miteinander kombinierbar. Dabei kann es aber schnell passieren, dass Sie Forderungen stellen, die nicht gleichzeitig erfüllbar sind. Typische Probleme sind z.B., dass ein Children-Wrapping erfordert, dass das Objekt seine Größe selbst bestimmen kann oder dass eine horizontale Zentrierung der Children erfordert, dass das Objekt größer ist, als der von den Children selbst eingenommene Platz. Children-Wrapping und gleichzeitige Zentrierung der Objekte ist folglich zunächst auch nicht möglich.
Der Geometriemanager muss in diesen Fällen eine Entscheidung treffen - oftmals wird das Ergebnis nicht Ihren Vorstellungen entsprechen. Es übersteigt die Möglichkeiten dieses Handbuchs bei Weitem, alle denkbaren Fälle zu besprechen. Auf typische Fallen oder Besonderheiten wird an den entsprechenden Stellen eingegangen. Lassen Sie sich dadurch bitte nicht abschrecken, sondern versuchen Sie es einfach.

In vielen Fällen können Sie das Problem lösen, indem Sie ein oder mehrere weitere VisObj-Objekte zum Gruppieren der eigentlichen Objekte einsetzen. Im Kapitel Children Wrapping finden Sie ein Beispiel, wie das oben angesprochene Problem (Children-Wrapping + zentrieren) auf diese Weise gelöst werden kann.
Ansonsten hilft nur systematisches Probieren. Und sehen Sie sich die Beispiele an.

Zugehörige Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
visSizeOptions	visSizeOptions = **<Wert>**	lesen, schreiben
visSizeFlags	visSizeFlags = **<Wert>**	lesen, schreiben
visMinimumSize	visMinimumSize = **minX, minY**	lesen, schreiben
visOrientVertically	visOrientVertically = TRUE | FALSE	lesen, schreiben
visChildJustification	visChildJustification = **jHor**, **jVert**	lesen, schreiben
visChildSpacing	visChildSpacing = **childSp** , **wrapSp**	lesen, schreiben
visSpacingIncludeEnds	visSpacingIncludeEnds = TRUE | FALSE	lesen, schreiben
visMargins	visMargins = **left** ,**top** ,**right** ,**bottom**	lesen, schreiben
allowChildrenToWrap	allowChildrenToWrap = TRUE | FALSE	lesen, schreiben
visWrapCount	visWrapCount = **numWert**	lesen, schreiben

MarkInvalid

Die Methode MarkInvalid bewirkt ein Neuzeichnen des Objekts, wobei - im Gegensatz zu Redraw und Dirty - die Geometrie des visual Trees neu berechnet wird. Dadurch werden unter Umständen auch andere Objekte neu (z.B. an anderer Position) gezeichnet.

Syntax: **<obj>.MarkInvalid**

Sie müssen MarkInvalid aufrufen, wenn Sie die Geometrie des visual Tree geändert haben, z.B. nach dem Ändern der Instancevariablen visChildJustification oder visMargins. Wenn Sie die Geometrie von mehreren Objekten geändert haben ist es im Allgemeinen ausreichend, MarkInvalid für eins der betroffenen Objekte zu rufen.

Eine Neuberechnung der Geometrie des visual Tree erfolgt auch, wenn der Nutzer das zugehörige View zoomt oder scrollt. Falls Sie vergessen haben, MarkInvalid zu rufen, kann das zu scheinbar seltsamen Effekten führen.

5.3.3.2 Festlegen der Größe

Die Bestimmung der Größe eines Objekts ist ein elementares Problem bei der Berechnung des visual Trees. Mittels der Instancevariablen **visSizeOptions** und **visSizeFlags** steuern Sie, wie ein Objekt seine Größe berechnen soll. Außerdem können Sie mit **visMinimumSize** eine Mindestgröße für Objekte mit nicht fest vorgegebener Größe festlegen.
Um die Größe des VisContent-Objekts festzulegen haben Sie weitere Möglichkeiten, die im Kapitel zum VisContent-Objekt beschrieben sind.

Zugehörige Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
visSizeOptions	visSizeOptions = **<Wert>**	lesen, schreiben
visSizeFlags	visSizeFlags = **<Wert>**	lesen, schreiben
visMinimumSize	visMinimumSize = **minX, minY**	lesen, schreiben

visSizeOptions

Die Instancevariable visSizeOptions bestimmt, wie das Objekt seine Größe berechnet. Dieser Wert wird benötigt, wenn sich das Objekt auf dem Schirm darstellt oder wenn das Parent-Objekt seine Größe anhand der Größe seiner Children berechnet.

Syntax UI-Code: **visSizeOptions** = **numWert**
Lesen: **<numVar>** = **<obj>.visSizeOptions**
Schreiben: **<obj>.visSizeOptions** = **numWert**

Für visSizeOptions stehen die folgenden Werte zur Verfügung. Der Defaultwert ist VSO_AUTO_SIZE.

Konstante	Wert	Kurzbeschreibung
VSO_AUTO_SIZE	0	Berechnungsmethode der Größe hängt von den Umständen ab
VSO_VARIABLE_SIZE	1	Objekt passt seine Größe selbständig an
VSO_FIXED_SIZE	2	Objekt hat eine feste Größe
VSO_FIXED_WIDTH	3	Objekt hat eine feste Breite
VSO_FIXED_HEIGHT	4	Objekt hat eine feste Höhe

Im Detail haben die Werte die folgende Wirkung:

VSO_AUTO_SIZE

Das Objekt berechnet seine Größe in Abhängigkeit davon, ob es Children hat, oder nicht. Dieses Verhalten ist für die meisten Situationen sinnvoll und deswegen der Defaultwert.

1. Fall: Es existieren Children
   Das Objekt berechnet seine Größe so, dass alle Children umfasst werden. Außerdem werden eventuelle Randbedingungen berücksichtigt wie z.B.
   * Ein VisContent soll seine Größe an die Größe des View-Objekts anpassen (siehe Instancevariable contentAttrs)
   * Ein Objekt soll sich den Platz mit seinen Geschwistern gleichmäßig aufteilen oder seine Breite bzw. Höhe maximieren (siehe Instancevariable visSizeFlags)
   * Eventuell gesetzte visMargins
   * ...

   Sie müssen keinen Wert für visSize festlegen, das macht der Geometriemanager.
2. Fall: Das Objekt hat keine Children
   In diesem Fall wird der in visSize festgelegte Wert benutzt. Sie müssen diesen Wert selbst festlegen.

VSO_VARIABLE_SIZE

Diesen Wert sollten Sie nutzen, wenn ein Objekt keine Children hat, seine Größe aber trotzdem entsprechend den bei VSO_AUTO_SIZE erwähnten Randbedingungen anpassen soll. Ein eventuell in visSize festgelegter Wert wird nicht benutzt.

VSO_FIXED_SIZE

Das Objekt hat eine feste Größe, die durch den in visSize eingestellten Wert bestimmt wird. Dieser Wert kann größer oder kleiner sein, als der von den Children des Objekts benötigte Platz. Die Children werden dann möglicherweise über den Rand des Objekts hinaus gezeichnet.
Sie müssen einen Wert für visSize festlegen.

Hinweis: Wenn Sie diesen Wert verwenden, funktioniert die automatische Anordnung der Children des Objekts in einigen Fällen nur eingeschränkt, da Sie dem Geometriemanager die Kontrolle entziehen. Versuchen Sie im Problemfall einen anderen Wert für visSizeOptions oder verwenden Sie den Modus mit customManageChildren = TRUE.

VSO_FIXED_WIDTH
Das Objekt hat eine feste Breite. Die Höhe wird entsprechend den bei VSO_AUTO_SIZE beschriebenen Bedingungen berechnet. Sie müssen einen Wert für visSize festlegen.

VSO_FIXED_HEIGHT
Das Objekt hat eine feste Höhe. Die Breite wird entsprechend den bei VSO_AUTO_SIZE beschriebenen Bedingungen berechnet. Sie müssen einen Wert für visSize festlegen.

Hinweis:
Ein Ändern des Wertes für visSizeOptions führt nicht automatisch zur Neudarstellung der Objekte. Dazu müssen Sie die Methode MarkInvalid aufrufen.

visSizeFlags

Die Instancevariable visSizeFlags enthält einzelne Bits, welche die Berechnung der Größe des Objekts beeinflussen.

Syntax	UI- Code:	visSizeFlags = **Wert**
	Lesen:	**<numVar>** = **<Obj>**.visSizeFlags
	Schreiben:	**<Obj>**.visSizeFlags = **Wert**

Für visSizeFlags stehen die folgenden Bits zur Verfügung. Bits, die nicht in der Tabelle aufgeführt sind, werden ignoriert.

Konstante	Wert (hex)	Dezimalwert
VSF_EXPAND_WIDTH	&H20	32
VSF_DIVIDE_WIDTH_EQUALLY	&H10	16
VSF_EXPAND_HEIGHT	&H02	2
VSF_DIVIDE_HEIGHT_EQUALLY	&H01	1

Per Default ist keins dieser Bits gesetzt.
Ein Ändern des Wertes zur Laufzeit löst noch kein Neuzeichnen der Objekte an ihrer neuen Position aus. Dazu müssen Sie die Methode **MarkInvalid** aufrufen.

Bedeutung der einzelnen Bits:

VSF_EXPAND_WIDTH
Bewirkt, dass sich das Objekt selbst so breit wie möglich macht. Damit kann es breiter werden, als es die Children erfordern, womit z.B. eine horizontale Zentrierung der Children (siehe Instancevariable visChildJustification) möglich wird.

VSF_DIVIDE_WIDTH_EQUALLY
Bewirkt, dass sich die Children des Objekts den verfügbaren Platz in der Breite gleichmäßig untereinander aufteilen. Dazu müssen alle Children

dieses Bit ebenfalls gesetzt haben. Das Objekt selbst hat oftmals zusätzlich das Bit VSF_EXPAND_WIDTH gesetzt.

VSF_EXPAND_HEIGHT
Bewirkt, dass sich das Objekt selbst so hoch wie möglich macht. Damit kann es höher werden, als es die Children erfordern, womit z.B. eine vertikale Zentrierung der Children (siehe Instancevariable visChildJustification) möglich wird.

VSF_DIVIDE_HEIGHT_EQUALLY
Bewirkt, dass sich die Children des Objekts den verfügbaren Platz in der Höhe gleichmäßig unterinander aufteilen. Dazu müssen alle Children dieses Bit ebenfalls gesetzt haben. Das Objekt selbst hat oftmals zusätzlich das Bit VSF_EXPAND_HEIGHT gesetzt.

Hinweis:
Die visSizeFlags-Bits sind wirkungslos, wenn das Objekt eine feste Größe hat, z.B. wenn das Objekt keine Children hat.
Setzen Sie dann **visSizeOptions** auf VSO_VARIABLE_SIZE, um eine variable Größe zu erzwingen

Beispiel
Drei Vis-Objekte sollen sich die Breite in einem View bzw. VisContent gleichmäßig aufteilen. Damit das funktioniert, müssen sowohl das VisContent als auch die VisObj-Objekte die Bits VSF_EXPAND_WIDTH und VSF_DIVIDE_ WIDTH_EQUALLY gesetzt haben.
Die VisObj-Objekte haben keine Children und würden deshalb eine feste Größe haben. Das muss in x-Richtung, aber nicht in y-Richtung geändert werden. Deswegen bekommen Sie einen Wert für visSize und visSizeOptions wird auf den Wert VSO_FIXED_HEIGHT gesetzt, wodurch die Breite variabel wird.

Den kompletten Code finden Sie im Beispiel "ExpandWidth Demo".

```
VisContent DemoContent
  Children = VisObj1, VisObj2, VisObj3

  visSizeFlags = \
  VSF_EXPAND_WIDTH +VSF_EXPAND_HEIGHT + VSF_DIVIDE_WIDTH_EQUALLY
  ...
End OBJECT

VisObj VisObj1
  visSize = 60, 40
  OnDraw = VisObjDraw
  visSizeFlags = VSF EXPAND WIDTH + VSF DIVIDE WIDTH EQUALLY
  visSizeOptions = VSO FIXED HEIGHT
End OBJECT
```

visMinimumSize

Die Instancevariable visMinimumSize enthält die minimale Größe des Objekts, getrennt nach x- und y-Richtung. Objekte, die ihre Größe verändern können, können nicht kleiner werden, als in visMinimumSize angegeben. Für Objekte mit einer festen Größe wird visMinimumSize ignoriert. Die Default-Werte für beide Werte sind 0, d.h. das Objekt hat keine minimale Größe. Beide Werte werden in Pixeln angegeben.

Syntax	UI-Code:	**visMinimumSize** = **minWidth** , **minHeight**
		mindWidth: minimale Breite
		minHeight: minimale Höhe
	Lesen:	**<numVar>** = **<obj>**.**visMinimumSize** (0) ' minWidth
		<numVar> = **<obj>**.**visMinimumSize** (1) ' minHeight
	Schreiben:	**<obj>**.**visMinimumSize** = **minWidth** , **minHeight**

Ändern Sie visMinimumSize zur Laufzeit, so wird das Objekt und seine Children nicht sofort an ihrer neuen Position bzw. in der neuen Größe gezeichnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

5.3.3.3 Ausrichtung und Abstand der Children

Per Default werden die Children eines VisObj oder VisContent-Objekts linksbündig, oben und mit 3 Pixeln Abstand voneinander angeordnet. Mit den in diesem Kapitel besprochenen Instancevariablen können Sie dieses Verhalten in weiten Grenzen ändern.

Zugehörige Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
visOrientVertically	visOrientVertically = TRUE I FALSE	lesen, schreiben
visChildJustification	visChildJustification = **jHor, jVert**	lesen, schreiben
visChildSpacing	visChildSpacing = **childSp , wrapSp**	lesen, schreiben
visSpacingIncludeEnds	visSpacingIncludeEnds = TRUE I FALSE	lesen, schreiben
visMargins	visMargins = **left ,top ,right ,bottom**	lesen, schreiben

visOrientVertically

Per Default werden die Children eines VisObj bzw. VisContent nebeneinander angeordnet. Setzen Sie die Instancevariable visOrientVertically auf TRUE, wenn Sie die Children untereinander anordnen möchten.

Syntax	UI- Code:	**visOrientVertically** = TRUE I FALSE
	Lesen:	**<numVar>** = **<Obj>**.**visOrientVertically**
	Schreiben:	**<Obj>**.**visOrientVertically** = TRUE I FALSE

visChildJustification

Die Instancevariable visChildJustification enthält die Information, wie die Children des Objekts in horizontaler und vertikaler Richtung ausgerichtet sind.

Syntax	UI- Code:	**visChildJustification** = **jHor, jVert**
	Lesen:	**<numVar>** = **<Obj>**.**visChildJustification** (0) ' jHor
		<numVar> = **<Obj>**.**visChildJustification** (1) ' jVert
	Schreiben:	**<Obj>**.**visChildJustification** = **jHor, jVert**
		jHor: horizontale Ausrichtung, siehe Tabelle
		jVert: vertikale Ausrichtung, siehe Tabelle

Prinzipiell können Sie die Werte für die horizontale Ausrichtung jHor und die vertikale Ausrichtung jVert beliebig kombinieren. Beachten Sie aber, dass die Randbedingungen für die gewählte Ausrichtung auch stimmen müssen. So muss

z.B. für eine horizontale Zentrierung der Children das Objekt selbst breiter sein, als der Platz, den die Children sowieso benötigen. Es ergibt auch keinen Sinn jVert auf J_FULL zu setzen (gleichmäßig über die ganze Höhe verteilt), wenn alle Children nebeneinander angeordnet sind.

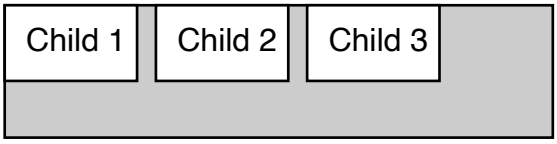
Für die horizontale Ausrichtung jHor stehen die folgenden Werte zur Verfügung. Andere Werte führen zu einem Fehler. Der Defaultwert für jHor ist J_LEFT.

Konstante	Wert	Anordnung der Children
J_LEFT	2	linksbündig
J_RIGHT	4	rechtsbündig
J_CENTER	1	horizontal zentriert
J_FULL	32	horizontal über die ganze Breite verteilt

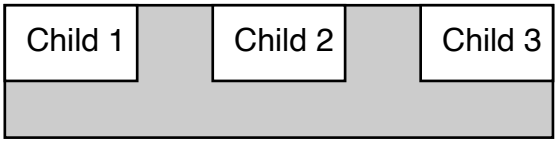
Für die vertikale Ausrichtung jVert stehen die folgenden Werte zur Verfügung. Andere Werte führen zu einem Fehler. Der Defaultwert für jVert ist J_TOP.

Konstante	Wert	Anordnung der Children
J_TOP	8	oben bündig
J_BOTTOM	16	unten bündig
J_CENTER	1	vertikal zentriert
J_FULL	32	vertikal über die ganze Höhe verteilt

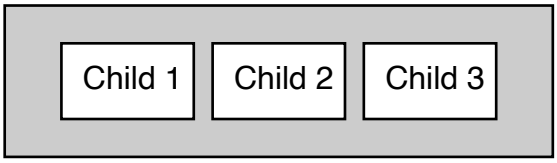
Beispiele für visChildJustification. Beachten Sie, dass vorausgesetzt ist, dass das grau gezeichnete Parent-Objekt größer ist, als die Children erfordern.



visChildJustification = J_JEFT, J_TOP (Defaultwert)



visChildJustification = J_FULL, J_TOP



visChildJustification = J_CENTER, J_CENTER

Hinweise:
- Um den Abstand zwischen den Children zu verändern, verwenden Sie bitte visChildSpacing.
- Um Platz zwischen den Children und dem Rand zu lassen, verwenden Sie bitte visMargins.
- Um die Children untereinander anzuordnen, setzen Sie die Instancevariable visOrientVertically auf den Wert TRUE.

- Wenn Sie den Wert von visChildJustification zur Laufzeit ändern, führt das nicht zum sofortigen neu Ausrichten der Objekte. Rufen Sie dazu die Methode MarkInvalid auf.

Beispiele mit **`visOrientVertically = TRUE`**:

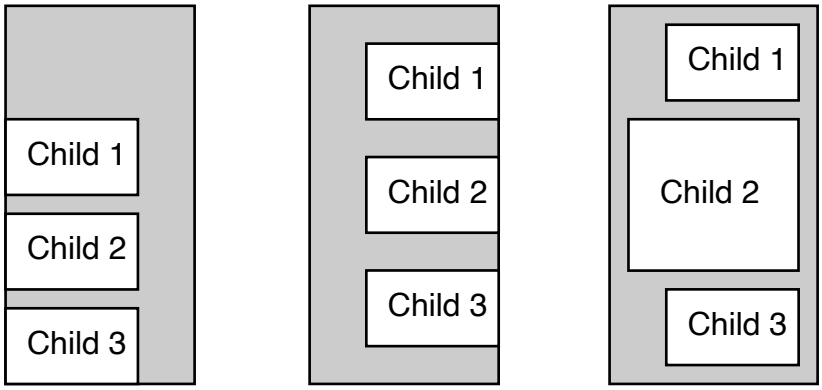


Bild 1 Bild 2 Bild 3

Bild 1: `visChildJustification = J_LEFT, J_BOTTOM`

Bild 2:
```
visChildJustification = J_RIGHT, J_FULL
visSpacingIncludeEnds = TRUE
```

Bild 3:
```
visChildJustification = J_RIGHT, J_FULL
visMargins = 5, 5, 5, 5
```

Child 2 hat zusätzlich folgende Instancevariablen gesetzt:
```
visSizeFlags = VSF_EXPAND_WIDTH + VSF_EXPAND_HEIGHT
visSizeOptions = VSO_VARIABLE_SIZE
```

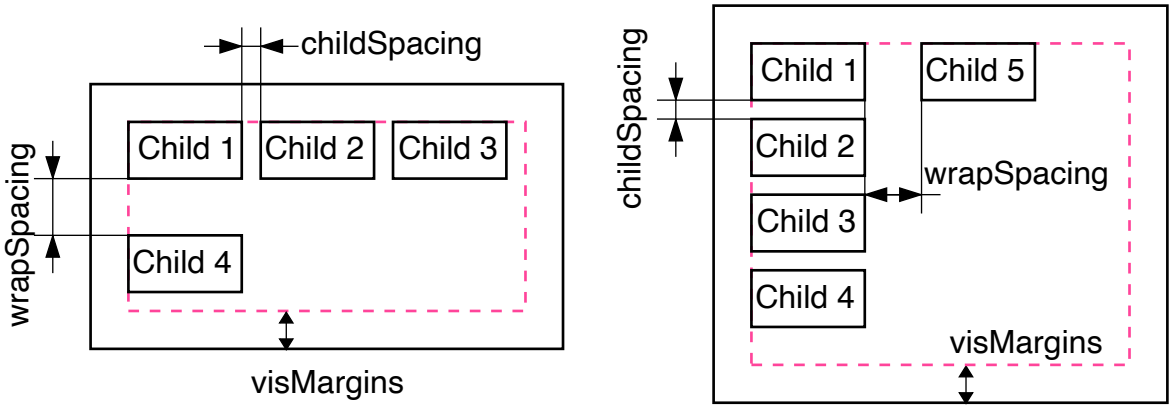
visChildSpacing

Die Instancevariable visChildSpacing bestimmt den Abstand zwischen benachbarten Children des Objekts. Der erste Wert (childSpacing) enthält den Abstand zwischen aufeinanderfolgenden (i.a. nebeneinander liegenden) Children, der zweite Wert (wrapSpacing) enthält den vertikalen Abstand zwischen aufeinanderfolgenden "Zeilen" von Children. Dazu muss die automatische Anordnung in mehreren Zeilen (Wrapping, Instancevariable allowChildrenToWrap, Siehe Kapitel 5.3.3.4) aktiv sein.
Die Default-Werte für childSpacing und wrapSpacing sind 3. Beide Werte werden in Pixeln angegeben.

Syntax	UI-Code:	**visChildSpacing** = **childSpacing** , **wrapSpacing** childSpacing: Abstand "benachbarter" Children wrapSpacing: Abstand "umgebrochener" Children
	Lesen:	**<numVar>** = **<obj>**.**visChildSpacing** (0) **<numVar>** = **<obj>**.**visChildSpacing** (1)
	Schreiben:	**<obj>**.**visChildSpacing** = **childSpacing** , **wrapSpacing**

Ändern Sie visChildSpacing zur Laufzeit, so werden die Children-Objekte nicht sofort an ihrer neuen Position gezeichnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

Achtung! Sind die Children untereinander angeordnet (siehe Instancevariable visOrientVertically), so enthält childSpacing immer noch den Abstand aufeinander folgender Children, also den vertikalen Abstand. WrapSpacing beschreibt wiederum den Wrapping-Abstand, jetzt also den horizontalen Abstand.



Unterschied zwischen childSpacing und wrapSpacing bei horizontaler (links) und bei vertikaler Anordnung der Children (rechts).

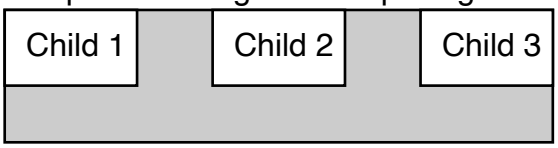
visSpacingIncludeEnds

Diese Instancevariable bewirkt, wenn sie auf TRUE gesetzt ist, dass bei der Berechnung der Geometrie zusätzlicher Platz neben (bzw. bei vertikaler Anordnung über oder unter) den Children berücksichtigt wird.
Der Defaultwert für visSpacingIncludeEnds ist FALSE.

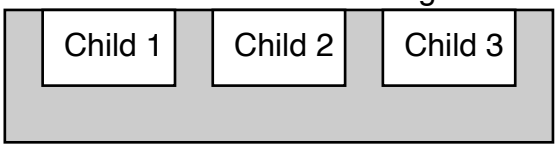
Syntax	UI-Code:	**visSpacingIncludeEnds** = TRUE I FALSE
	Lesen:	**<numVar>** = **<obj>**.**visSpacingIncludeEnds**
	Schreiben:	**<obj>**.**visSpacingIncludeEnds** = TRUE I FALSE

Ändern Sie visSpacingIncludeEnds zur Laufzeit, so werden die Objekte nicht sofort neu angeordnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

Beispiel: Wirkung von visSpacingIncludeEnds bei horizontaler Anordnung



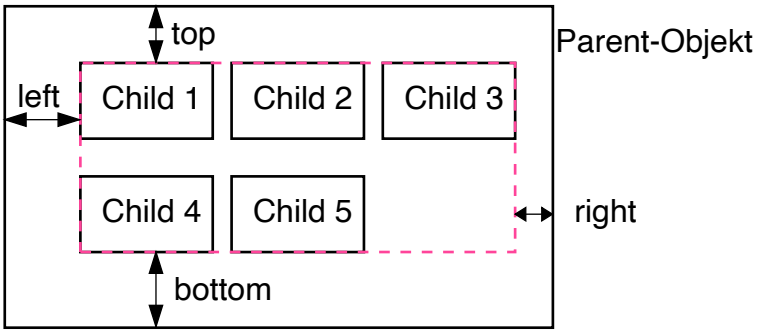
visChildJustification = J_FULL, J_TOP
visSpacingIncludeEnds = FALSE



visChildJustification = J_FULL, J_TOP
visSpacingIncludeEnds = **TRUE**

visMargins

Mit visMargins können Sie einen zusätzlichen Rand um die Children des Objekts reservieren.



Die Default-Werte für alle visMargins-Werte sind 0. Alle Werte werden in Pixeln angegeben.

Syntax	UI-Code:	**visMargins** = **left** , **top , right , bottom**	
		left: linker Rand	
		top: oberer Rand	
		right: rechter Rand	
		bottom: unterer Rand	
	Lesen:	**<numVar>** = **<obj>**.**visMargins** (0)	' left
		<numVar> = **<obj>**.**visMargins** (1)	' top
		<numVar> = **<obj>**.**visMargins** (2)	' right
		<numVar> = **<obj>**.**visMargins** (3)	' bottom
	Schreiben:	**<obj>**.**visMargins** = **left** , **top , right , bottom**	

Ändern Sie visMargins zur Laufzeit, so werden die Children-Objekte nicht sofort an ihrer neuen Position gezeichnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

5.3.3.4 Children Wrapping

Unter Wrapping versteht man, dass Objekte automatisch ein einer neuen Zeile oder Spalte angeordnet werden, wenn der Platz nicht mehr ausreicht.
Ist ein Wrapping nicht möglich bzw. nicht erlaubt, so werden die Child-Objekte über die Grenzen ihres Parent-Objekts hinaus angeordnet.

Zugehörige Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
allowChildrenToWrap	allowChildrenToWrap = TRUE | FALSE	lesen, schreiben
visWrapCount	visWrapCount = **numWert**	lesen, schreiben

Im Zusammenhang mit dem Children Wrapping kann es allerdings schnell zu widersprüchlichen Geometrie-Anweisungen kommen. Einige Beispiele:

- Children-Wrapping erfordert, dass das Parent-Objekt in seiner Größe begrenzt ist. Für VisContent-Objekte ist es deswegen meist erforderlich, die folgenden Instancevariablen zu setzen:

```
contentAttrs = \
          CA_SAME_WIDTH_AS_VIEW + CA_SAME_HEIGHT_AS_VIEW , 0
visSizeFlags = VSF_EXPAND_WIDTH + VSF_EXPAND_HEIGHT
```

- Wenn das Wrapping erlaubt ist (allowChildrenToWrap = TRUE) werden horizontal angeordnete Children immer am oberen Rand ausgerichtet - sonst ist unten kein Platz für die nächste Zeile. Anderslautende Einstellungen in der Instancevariablen visChildJustification werden ignoriert. Analog werden vertikal ausgerichtete Children immer links ausgerichtet. Das bedeutet zum Beispiel, dass mit
  visChildJustification = J_CENTER, J_CENTER
  die Children nur in eine Richtung zentriert werden.
  Unten finden Sie ein Beispiel, wie man dieses Problem umgehen kann.

- Hat das Objekt (VisContent oder VisObj) in eine Richtung eine feste Größe (Instancevariable visSizeOptions = VSO_FIXED_WIDTH bzw. ..._HEIGHT), so wird die Größe in die andere Richtung oft nicht korrekt berechnet. Das muss kein Problem sein, da es erlaubt ist, dass sich Children außerhalb der Grenzen ihres Parent-Objekts befinden.

- Hat das Objekt (VisContent oder VisObj) in beiden Richtungen eine feste Größe (Instancevariable visSizeOptions = VSO_FIXED_SIZE), so wird der Parameter wrapSpacing der Instancevariablen visChildSpacing anders interpretiert. Er beschreibt nicht mehr den Platz zwischen den Objekten sondern bezieht die Größe des darüber bzw. links liegende Objekts mit ein.

Es übersteigt die Möglichkeiten dieses Handbuchs bei Weitem, alle denkbaren Fälle zu besprechen. Für den Fall, dass die Children nicht so angeordnet werden, wie Sie sich das vorstellen, hilft meist nur systematisches Probieren.
Bitte beachten Sie die Beispiele. Dort finden Sie auch Tipps, wie man das Children Wrapping in verschiedenen Fällen konfiguriert.

allowChildrenToWrap

Diese Instancevariable erlaubt, wenn sie auf TRUE gesetzt ist, dass die Children des Objekts "umgebrochen", d.h. in einer neuen Zeile bzw. Spalte dargestellt werden, wenn das Objekt zu klein ist, um alle Children in einer Reihe darzustellen. Wenn die Children horizontal dargestellt werden wird eine neue Zeile eröffnet, werden sie vertikal dargestellt, so wird eine neue Spalte eröffnet.
Der Defaultwert für allowChildrenToWrap ist FALSE.

Syntax	UI-Code:	**allowChildrenToWrap** = TRUE I FALSE
	Lesen:	**<numVar>** = **<obj>**.**allowChildrenToWrap**
	Schreiben:	**<obj>**.**allowChildrenToWrap** = TRUE I FALSE

Ändern Sie allowChildrenToWrap zur Laufzeit, so werden die Objekte nicht sofort neu angeordnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

visWrapCount

Mit der Instancevariablen visWrapCount legen Sie fest, dass der Children-Umbruch (Wrapping) nach einer bestimmten Anzahl von Children erzwungen wird. Der Defaultwert für visWrapCount ist Null, d.h. es erfolgt kein erzwungenes Wrapping.

Syntax	UI-Code:	**visWrapCount** = **value**
		value: Anzahl Children, nach denen umgebrochen wird
	Lesen:	**<numVar>** = **<obj>**.**visWrapCount**
	Schreiben:	**<obj>**.**visWrapCount** = **value**

Es gibt zwei Voraussetzungen, dass visWrapCount funktioniert.
1. Das Wrapping muss erlaubt sein, d.h. die Instancevariable allow-ChildrenToWrap muss auf TRUE gesetzt sein. Das ist per Default **nicht** der Fall!
2. Das Objekt muss in die entsprechende Richtung eine variable Größe haben. Das ist per Default der Fall.
   Setzen Sie jedoch bei horizontal angeordneten Children z.B. visSizeOptions = VSO_FIXED_WIDTH, so hat das Objekt eine vorgegebene Größe und visWrapCount kann nicht funktionieren.

Ändern Sie visWrapCount zur Laufzeit, so werden die Objekte nicht sofort neu angeordnet. Dazu müssen Sie erst die Methode MarkInvalid aufrufen.

Beispiel: Sie möchten 4 Vis-Objekte im Quadrat zentriert in einem View darstellen. VisWrapCount und visChildJustification = J_CENTER funktionieren jedoch nicht gemeinsam, so dass Sie ein weiteres Objekt als Grouping-Objekt für die VisObj-Objekte verwenden müssen. Den kompletten Code finden Sie im Beispiel "Wrap und Center"

```
VisContent DemoContent
  Children = VisGroupObj
  visChildJustification = J CENTER, J CENTER
    ...
End OBJECT

VisObj VisGroupObj
  Children = VisObj1, VisObj2, VisObj3, VisObj4
  allowChildrenToWrap = TRUE
  visWrapCount = 2
  ...
End OBJECT
```

(Leerseite)