Ansgar Scherp, Jennis Meyer-Spradow

Patchwork

Von VGA zu SVGA im Protected Mode

Viele Shareware-Spiele arbeiten auch heute noch mit der Klötzchenauflösung 320 × 200 Pixel. Grund ist inzwischen nicht mehr die fehlende Rechenleistung, sondern mehr die unübersichtliche und komplizierte Programmierung von SuperVGA-Grafik. Doch gerade bei Spielen bietet sich eine einfache und effiziente Alternative an.

Der Grafikmodus 13h riß anno 1987 viele Grafikprogrammierer zu wahren Begeisterungsstürmen hin. Neben der einfachen Programmierbarkeit (ein Pixel entspricht genau einem Byte) eröffnete er erstmals 256 Farben aus einer Palette von 262 144 möglichen. Daß er nur 320 × 200 Pixel bot, störte wenig. Der daraus erwachsende Vorteil, mit 64 000 benötigten Bytes bestens in ein Segment der Intelschen-Speicherarchitektur zu passen, war zu Real-Mode-Zeiten nicht unwichtig. Doch die weitere Entwicklung ging an den meisten Spieleprogrammierern vorbei. Zu langwierig war und ist es, für jede Auflösung neue Routinen zu entwickeln.

Man kann sich aber auch die vorhandenen Routinen zunutze machen, indem man den SVGA-Bildschirm in Blöcke zu 320 × 200 Pixel aufteilt. So erhält man mehrere neben- und untereinanderliegende VGA-Bildschirme, die man eigentlich mit den normalen Routinen bearbeiten könnte – jeweils einzeln, versteht sich. Leider lassen sich die Einzelteile nicht direkt im Bildspeicher bearbeiten, da dieser natürlich nicht im Format 320 × 200 organisiert ist. Lösen läßt sich das Problem dadurch, daß man den Bildschirmspeicher im Hauptspeicher 'simuliert' und dabei die Aufteilung auf VGA-Größe ausrichtet. Die letzte vorbereitende Maßnahme besteht dann darin, die vorhandenen VGA-Routinen so umzuschreiben, daß sie nicht nur in das VGA-Segment A000, sondern in jedes beliebige Segment zeichnen können. Die Anpassung an verschiedene Auflösungen erschöpft sich dadurch in zwei kleinen Routinen: Die eine kopiert die einzelnen Teile in den realen Bildschirm, die andere erledigt die Anpassung an das 'virtuelle' VGA-Koordinatensystem sowie das Setzen der Clippinggrenzen.

Die Kopierroutine greift auf eine vom VESA-Komitee entwickelte Software-Schnittstelle zur Programmierung von SVGA-Grafikkarten zu, die VGA Standard BIOS Extensions (kurz VBE). Moderne Grafikkarten haben heute eigentlich immer ein VESA-VBE-BIOS; für die anderen sind im allgemeinen Treiber vorhanden, die diese Funktionen nachbilden (beispielsweise UniVBE von Scitech Soft). Durch Verwendung von VBE-Funktionen zum Ansprechen der Hardware ist gesichert, daß die Routinen auf einer möglichst großen Palette von Grafikkarten funktionieren.

Ich werde mich im folgenden auf den VESA-Modus 101h mit einer Auflösung von 640 × 480 Pixel und 256 Farben konzentrieren. Alle anderen Modi funktionieren aber analog. Aus der Auslegung für die ursprüngliche Auflösung von 320 × 200 Pixeln ergibt sich für den 640 × 480 Pixel großen Bildschirm die Aufteilung in insgesamt sechs Blöcke (zwei Spalten à drei Zeilen, wobei jedoch von den unten liegenden Blöcken nur 80 Pixel sichtbar sind).

Flüssige und flimmerfreie Animationen erreicht man nur, wenn der Computer ein Bild ohne Änderungen anzeigt, während er ein zweites im Hintergrund berechnet und aufbaut (Double Buffering). Daher ist die Unterstützung von mehreren Bildschirmseiten nötig.

Elegant und sicher

Da jede Bildschirmseite im Modus 101h 300 KByte Speicher belegt, scheidet ein Ablegen der Seiten unter DOS im ohnehin schon knappen Hauptspeicher aus. Eine Möglichkeit, das Problem zu lösen, wäre das Auslagern in den Speicher über 1 MByte mit Hilfe von XMS oder EMS. Diese Methode ist allerdings umständlich und langsam. Schneller und eleganter sind Programme, die im Protected Mode der Intel-Prozessoren laufen, was mit Hilfe einer Extender-Software heutzutage auch unter DOS relativ einfach möglich ist.

Einige Besonderheiten sind bei der VESA-VBE-Programmierung im Protected Mode zu beachten. Der direkte Zugriff auf die VESA-Routinen ist im Protected Mode nicht möglich, da die Interruptverwaltung vollkommen anders funktioniert. Lösen läßt sich das Problem, indem das Programm kurzfristig in den Real Mode umschaltet, die gewünschte Funktion aufruft und wieder in den Protected Mode wechselt.

Die Arbeit übernimmt die Borland-Implementierung des DOS Protected Mode Interface (DPMI). Nur einen eventuell benötigten Speicherbereich muß man selbst bereitstellen. Dazu dient die Bibliotheksprozedur *GlobalDOSAlloc*. Das Vorbesetzen und den Aufruf der DPMI-Funktion erledigt dann die Prozedur *SimRealModeInt* der Unit DPMI.

Ein weiteres Problem ist die Adressierung der Grafikhardware. IBM sah im Original-PC nur 64 KByte Bildschirmspeicher vor, und in dieses Speicherfenster müssen sich auch heute noch alle Grafikkarten zwängen, sofern sie keinen linearen Framebuffer unterstützen. Sie entledigen sich des

Problems dadurch, daß sie immer nur kleine 64-KByte-Häppchen ihres Speichers einblenden. Diese sogenannten Bänke wählt die Prozedur *SetBank* aus.

Die Banknummer eines Punktes P(x,y) bei einer Granularität von 64 KByte errechnet sich wie folgt:

```
BankNr:=((y × BildschirmBreite+x) ×
        BytesProPixel)DIV 65 535
```

Die Berechnung der Offsetadresse funktioniert nach dieser Formel:

```
Offset:=((y*BildschirmBreite+x)*
        BytesPerPixel) MOD 65 535
```

Ein letztes Problem stellt die Adressierung dar. Sie ist nicht wie im Real Mode über Segment:Offset, sondern nur mit Hilfe von Selektoren möglich. Hier hilft Borland Pascal mit einem Standardselektor namens 'SegA000' weiter.

Arbeitsteilung

Der geschwindigkeitsunrelevante Overhead ist in Pascal geschrieben, die geschwindigkeitsabhängigen Prozeduren und Funktionen (Sprites zeichnen, Bildschirminhalte kopieren …) dagegen in Assembler.

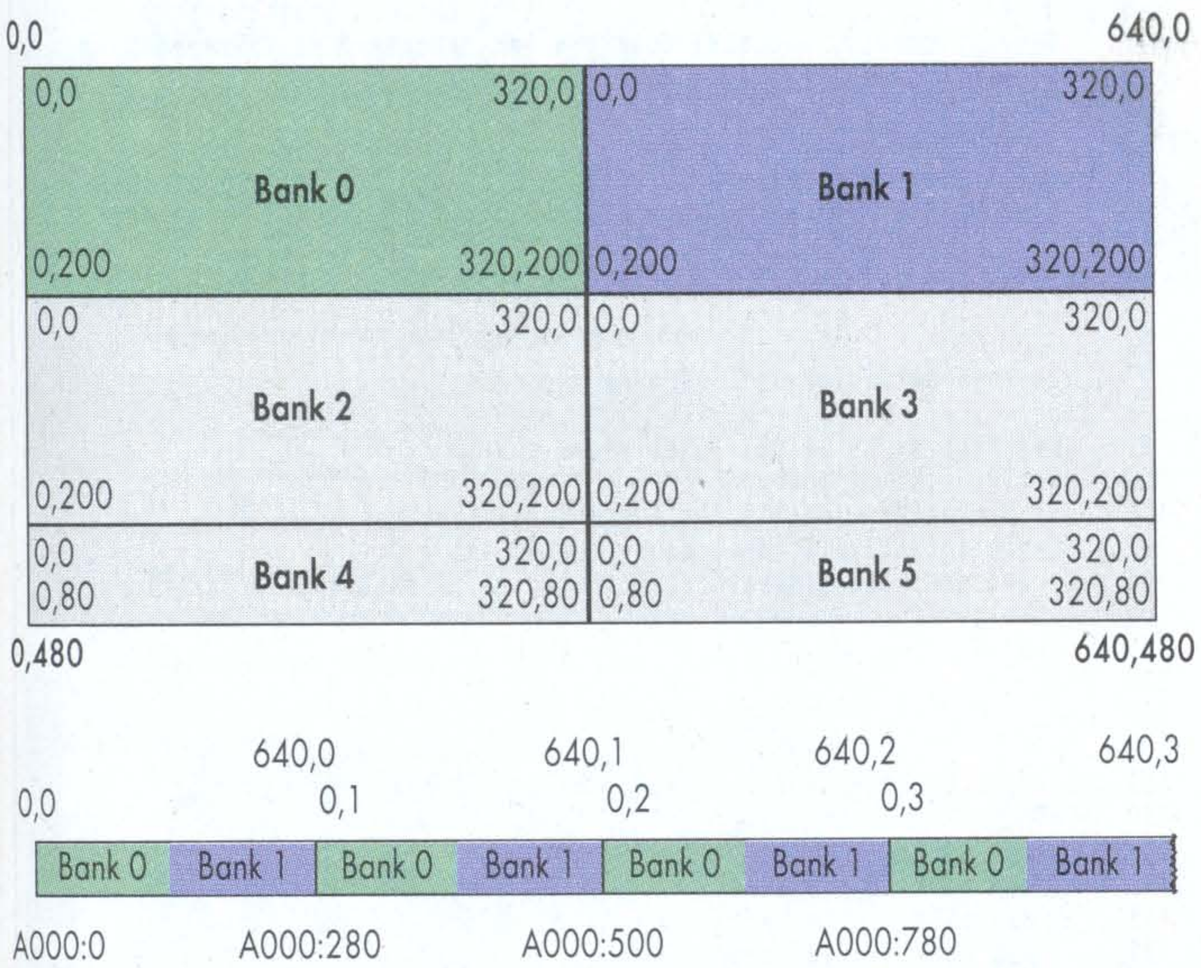
Ähnlich wie bei der objektorientieren Programmierung erlaubt das Verfahren die Wiederverwendung alter Programmteile. Die SVGA-Routinen sind relativ schnell erstellt und normalerweise stabil, da sie auf die erprobten Funktionen für den Modus 13h zugreifen.

Da die vorliegende Bibliothek ausschließlich in eine virtuelle Seite schreibt, bleibt die Anzahl der (langsamen) Bankumschaltungen pro Bildwechsel konstant. Es sind pro Kopieraktion einer virtuellen Seite zur Grafikkarte im Modus 101h lediglich vier Bankumschaltungen nötig.

Licht …

Das Grundgerüst jeder SVGA-Routine, die auf einer Normal-VGA-Routine basiert, hat folgende Struktur:

```
Schleife: wiederhole für Block=0
          bis Block=n
  aktiviere Block
  setze den für diesen Block festgelegten
  Fensterrahmen
  Berechne die relativen Koordinaten
  bezüglich des gerade aktiven Blocks
  aus den absoluten Koordinaten (etwa
  bei PutSprite). Dies geschieht mit den
  Funktionen CalcPageX und
  CalcPageY.
  führe die Normal-VGA-Prozedur mit
  den relativen Koordinaten aus (die
  relativen Koordinaten beziehen sich
  auf den gerade aktiven Block und sind
  somit die absoluten Koordinaten der
  Normal-VGA-Routine).
Schleifenende
```



Im Hauptspeicher liegt eine Kopie des Grafikspeichers für den SVGA-Modus 101h. Durch die geschickte Aufteilung in Einzelelemente können nun schon fertig programmierte VGA-Routinen genutzt werden. Etwas trickreich ist das Kopieren der Einzelteile vom Haupt- in den Bildspeicher.

Die Funktionen *CalcPageX* und *CalcPageY* erwarten als Parameter eine absolute x- beziehungsweise y-Koordinate und die Angabe eines Blocks. Daraus errechnet sich dann eine relative Koordinate bezüglich der angegebenen. So ist der absolute Punkt P(400,300) der relative Punkt R3(80,80) im dritten Block.

Am problematischsten ist der Kopiervorgang von der virtuellen Bildschirmseite in den Grafikspeicher. Da die virtuelle Seite in Blöcke zu 320 × 200 Pixel aufgeteilt ist, die Grafikkarte in ihrem Speicher aber zusammenhängende Zeilen erwartet, ist die Kopiererei etwas trickreich. Zusätzlich muß sich die Routine um Bankumschaltungen an den Segmentgrenzen kümmern.

Die Prozedur *CopyP2V* besteht hauptsächlich aus zwei Unterprozeduren. *CopyLines* bekommt als Parameter einen Zeiger auf den Speicherbereich des linken Blocks, einen Zeiger auf den Speicher des rechten Blocks, die Offset-Adresse im Bildschirmspeicher der entsprechenden Bank und die Anzahl der zu kopierenden Zeilen. Die Prozedur *CopyLines* kopiert dann die virtuelle Seite Zeile für Zeile in den Grafikspeicher, wobei sie immer abwechselnd eine halbe sichtbare Zeile aus dem linken Block und eine halbe Zeile aus dem rechten Block kopiert. Bei den Zeilen, in denen ein Bankwechsel auftritt, kommt die Prozedur *CopyDWord* zum Einsatz. Sie kopiert eine angegebene Anzahl an Bytes aus einem virtuellen Block in den Grafikspeicher. *CopyDWord* arbeitet aus Geschwindigkeitsgründen mit dem DWord-Kopierbefehl des Prozessors, so daß die Anzahl der zu kopierenden Bytes durch vier teilbar sein muß.

… und Schatten

Ein Nachteil des beschriebenen Verfahrens ist, daß Bezüge zwischen zwei oder mehr Blöcken einer virtuellen Seite nur schwer herzustellen sind, da die Blöcke völlig autark voneinander arbeiten. Des weiteren ist ein direkter Lesezugriff auf den Grafikspeicher nicht möglich. Auf ihn kann nur über eine virtuelle Seite zugegriffen werden. Dies ist jedoch nicht wirklich ein Nachteil, denn flüssige Animationen benötigen zunächst eine virtuelle Seite, die das Programm dann in den sichtbaren Bildschirmspeicher kopiert.

Um die Routinen noch effizienter zu gestalten, wäre eine Implementierung der Routinen in reinem Assembler denkbar. Auch die Unterstützung spezieller Fähigkeiten von Grafikkarten, zum Beispiel direktes Umschalten der Bänke bei ET4000-Grafikkarten, wäre sinnvoll. Dies setzt jedoch eine hundertprozentig funktionierende Hardwareerkennung voraus. Zudem steht der Geschwindigkeitszuwachs in keinem Verhältnis zum Implementierungsaufwand. Dies ist nur für große Unternehmen mit entsprechender Manpower lohnend.

Die Listings zeigen einige interessante Aspekte der Programmierung. Vollständige Anwendungen – auch für andere Video-Modi – finden sich in unserer Mailbox und auf dem ftp-Server. Test101.pas ist ein kleines Programm zum Testen der Funktionen. Video13.pas und video101.pas enthalten die Grafikfunktionen, und dpmi.pas stellt Hilfsfunktionen für das DOS Protected Mode Interface zur Verfügung.

Zukunftsausblick

Eine Grafikbibliothek nach dem obigen Vorbild ist auch für HiColor oder TrueColor denkbar. Der Ansatz ist exakt der gleiche. Überlegenswert ist dann allerdings, die Blockgröße zu ändern. Bei 32 K oder 64 K Farben ist die Festlegung der Blockgröße auf 320 × 100 Pixel sinnvoll, da dann jeweils ein Block in ein Segment paßt. Rein theoretisch ist natürlich auch eine Aufteilung des Bildschirms in andere Blockgößen denkbar, etwa 256 × 192 für den Modus 1024 × 768. (jm)

Literatur

[1] Mark Stehr, Pixel-Fenster für DOS, Super-VGA-Grafik programmieren mit VESA-BIOS-Extensions, c't 6/95, S. 266

[2] Jürgen Petsch, Pixel-Pinsel, Programmieren der VESA-Grafikmodi unter DOS, c't 2/97, S. 300

[3] Matthias Withopf, Windo(w)s XL, 32-Bit-Programmierung mit Borland Pascal für DOS und Windows, c't 7/93, S. 204

VESA-Treiber und Linear Frame Buffer

Hat eine Grafikkarte noch kein VESA-VBE-kompatibles BIOS, funktionieren die hier entwickelten Routinen nur in Verbindung mit einem speziellen VESA-Treiber. Normalerweise existiert für jede Grafikkarte ein solcher VESA-Treiber, der die VBE-Funktionen für die jeweilige Grafikkarte zur Verfügung stellt. Außer diesen speziellen VESA-Treibern sind auch noch generische Treiber verfügbar, die mit nahezu allen Grafikkarten zusammenarbeiten. Dazu gehört beispielsweise UniVESA. Dieses Programm erkennt die Grafikhardware selbständig und stellt entsprechende Funktionen zur Verfügung.

Zum Beispiel können Grafikchips ihren Grafikspeicher schon seit einiger Zeit oberhalb des physisch vorhandenen Hauptspeichers einblenden. In diesem 'Linear Frame Buffer' (LFB) kann man den gesamten vorhandenen Bildspeicher in einem Stück ansprechen. UniVESA erkennt diese Chips und schaltet dann die segmentierte Adressierung ab. Falls Programme – wie das hier vorgestellte oder auch viele Spiele – auf die segmentierte Adressierung angewiesen sind, gibt das natürlich Probleme.

Der LFB läßt sich aber auch relativ einfach nutzen. Das VESA-Bios bietet ab der Version 2.0 die Möglichkeit, den LFB einzuschalten. Dazu hat das Komitee die Funktion 2 (Set Super VGA Video Mode) zum Setzen des Videomodus modifiziert. Der gewünschte Videomodus wird im BX-Register übergeben; ist Bit 14 gesetzt, versucht das BIOS den LFB einzuschalten. Über die Funktion 0 (Return Super VGA Mode Information) gibt das BIOS dann in einem Speicherbereich die Startadresse des LFB zurück [1].

Mit Turbo Pascal im Protected Mode hat man allerdings das Problem, keine Segmente erzeugen zu können, die größer als 64 KByte sind. Durch direktes Manipulieren eines Descriptors [3] kann man sich trotzdem Zugang zum LFB verschaffen. Die eigentlichen Kopierroutinen müssen dann allerdings in Assembler verfaßt sein. Sie machen regen Gebrauch von den Präfix-Befehlen (db $66 et cetera), um mit den 32-Bit breiten Registern im 16-Bit-Inline-Assembler zu hantieren.

Die Kopierroutinen vereinfachen sich durch die eingesparten Bankumschaltungen erheblich. Ob sich die Optimierungen auch auf die Ausgabegeschwindigkeit auswirkt, ist abhängig von der Grafikkarte. ET6000-Karten beispielsweise profitierten nicht, ein Virge/VX-Board wurde um Faktor vier schneller.

```
1   { VIDEO101.PAS - der Aufsatz auf VIDEO13H.PAS für Mode 101h V2.0
2     Borland Pascal 7.0   Routinen funktionieren NUR im Protected Mode.
3     (c) Ansgar Scherp, Joachim Gelhaus 1996-1997}
4
5   {$IFNDEF DPMI}Nur im Protected Mode lauffähig...{$ENDIF}
6
7   unit VIDEO101;
8
9   interface
10
11  uses VIDEO13;
12
13  type { neuer Datentyp für virtuelle Bildschirmseiten im Modus 101h }
14    TPage101h = array[0..5] of pointer;
15
16  const
17    Mode101h : word = $101; { der Bildschirmmodus 640x480x256 }
18
19
20  { Prozeduren analog wie in Video13.pas }
21  procedure InitVideo101h;{ Initialisieren des VideoModus 101h }
22  procedure ActivePage101h( var page : TPage101h);
23  procedure SetVideoMode101h( mode : word );
24  procedure SetWindow101h( x1, y1, x2, y2 : longint );
25  procedure CopyP2P101h( var DstPage, SrcPage : TPage101h);
26  procedure CopyP2V101h( var page : TPage101h);
27  procedure ClearPage101h( page : TPage101h);
28  procedure ClosePage101h( var page : TPage101h);
29  procedure PutSprite101h( x, y : integer; sprite : TSprite );
30  procedure GetSprite101h( x, y : integer; sprite : TSprite );
31  procedure PutPixel101h( x, y : longint; c : byte );
32  function  GetPixel101h( x, y : longint ) : byte;
33
34  implementation
35
36  uses DPMI;
37
38
39  type
40    Modes   = array[0..255] of word;
41    PModes  = ^Modes;
42    ASCII   = array[0..255] of char;
43    PASCII  = ^PASCII;
44
45    TVESAInfo = { allgemeine VESA-Informationen }
46      record
47        signature    : array[ 0..3 ] of char; {VESA-Signatur: "VESA"}
48        version      : array[ 0 .. 1 ] of byte; { Versionsnummer }
49        OEMName      : PASCII;                  { Herstellername }
50        capabilities : array[ 0 .. 3 ] of byte;
51        vmodes       : PModes;
52        reserved     : array[ 0 .. 237 ] of byte;
53      end;
54
55    TModeInfo = { diverse Infos zu den Videomodus }
56      record
57        attributes       : word;
58        winA             : byte;
59        winB             : byte;
60        granularity      : word;
61        size             : word;
62        segA             : word;
63        segB             : word;
64        eqv4f05          : longint;
65        bytesperscanline : word;
66        width            : word;
67        height           : word;
68        characterwidth   : byte;
69        characterheight  : byte;
70        planes           : byte;
71        bitsperpixel     : byte;
72        banks            : byte;
73        memorymodel      : byte;
74        sizeofbank       : byte;
75        res              : array[ 0 .. 256 - $1E ] of byte;
76      end;
77
78
79  var
80    ActVPage101h   : TPage101h; { aktive virtuelle Seite }
81    RealRegs       : TRealModeRegs; { ein 'Satz' RealMode-Register }
82    LowMemoryBlock : TLowMemoryBlock; { Speicherblock im ersten MB }
83    VesaInfo       : TVesaInfo;       { allgemeine VESA-Informationen }
84    ModeInfo       : TModeInfo;       { Informationen zum Modus 101h }
85    Granny         : byte;            { unterstützte Granularität }
86    BankNr         : byte;            { BankNr der Grafikkarte }
87    GlobalWindowX1, GlobalWindowX2, { Fensterrahmen des SVGA-Modus }
88    GlobalWindowY1, GlobalWindowY2 : longint;
89
90
91  const { die Fensterrahmen für die 6 'kleinen' Blöcke in Mode 101h }
92    Windows101h : array[ 0..5 { Block 0 bis 5 } , 1..4 ] of integer = (
93                  (0,319,0,199), (0,319,0,199), (0,319,0,199),
94                  (0,319,0,199), (0,319,0,199), (0,319,0,199));
95
96
97  procedure SetVideoMode101h( mode : word );
98  begin
99    RealRegs.ax := $4f02; { set video mode }
100   RealRegs.bx := mode;
101   if SimRealModeInt($10,RealRegs)=false then
102   begin { falls ein Fehler aufgetreten ist }
103     writeln( 'GetSimRealModeIntFehler: ', SimRealModeIntErrorCode );
104     halt(1);
105   end;
106 end;
107
108
109 function CalcPageX( x : integer; bank : word ) : integer;
110 begin
111   asm
112     cmp bank, 0; je @ende
113     cmp bank, 2; je @ende
114     cmp bank, 4; je @ende
115     sub x, 320 { Block 1,3,5 }
116     @ende:
117   end;
118   CalcPageX := x;
119 end;
120
121
122 function CalcPageY( y : integer; bank : word) : integer;
```

```
123  begin
124    asm
125      cmp bank, 2
126      jb @ende
127        sub y, 200 { Block 2, 3 }
128        cmp bank, 4
129        jb @ende
130        sub y, 200 { Block 4,5 }
131      @ende:
132    end;
133    CalcPageY := y
134  end;
135
136
137  function GetPage( x, y : integer ) : word;
138  var SubPage : byte;
139  begin
140    if x <= 319 then SubPage := 0 else SubPage := 1;
141    if y >= 400 then inc(SubPage,4)
142      else if y >= 200 then inc(SubPage,2);
143    GetPage := SubPage;
144  end;
145
146
147  procedure SetWindow101h( x1, y1, x2, y2 : longint );
148  var
149    h     : longint;
150    block : byte;
151    rx1, rx2, ry1, ry2 : integer;
152  begin
153    { wenn der Rahmen ausserhalb zu gross, dann begrenzen }
154    if x1 < 0 then x1 := 0; if x1 > 639 then x1 := 639;
155    if y1 < 0 then y1 := 0; if y1 > 479 then y1 := 479;
156    { vertauschen der Grenzen links/rechts bzw. oben/unten falls nötig}
157    if x1 > x2 then begin h := x2; x2 := x1; x1 := h; end;
158    if y1 > y2 then begin h := y2; y2 := y1; y1 := h; end;
159    GlobalWindowX1 := x1;
160    GlobalWindowX2 := x2;
161    GlobalWindowY1 := y1;
162    GlobalWindowY2 := y2;
163    for block:=0 to 5 do {von Block 0-5 den Rahmen berechnen/festlegen}
164      begin
165        rx1 := CalcPageX( x1, block ); ry1 := CalcPageY( y1, block );
166        rx2 := CalcPageX( x2, block ); ry2 := CalcPageY( y2, block );
167        SetWindow( rx1, ry1, rx2, ry2 ); { Rahmen des Blocks setzen }
168        if ( WindowX1 = WindowX2 ) or ( WindowY1 = WindowY2 ) then
169          begin { falls Breite oder Tiefe des Rahmens gleich Null ist}
170            WindowX1 := 0; WindowX2 := 0; WindowY1 := 0; WindowY2 := 0;
171          end;
172      { den Rahmen jedes Blocks für die anderen Routinen 'merken' }
173      Windows101h[block,1]:=WindowX1;Windows101h[block,2]:= WindowX2;
174      Windows101h[block,3]:=WindowY1;Windows101h[block,4]:= WindowY2;
175    end;
176  end;
177
178
179  procedure SetBank( bank : byte );
180  begin
181    RealRegs.ax := $4f05; RealRegs.bx := $0000; { set bank }
182    RealRegs.dx := bank * granny;
183    if SimRealModeInt( $10, RealRegs ) = false then
184      begin { falls ein Fehler aufgetreten ist }
185        writeln( 'GetSimRealModeIntFehler: ', SimRealModeIntErrorCode);
186        halt(1);
187      end;
188  end;
189
190
191  procedure GetVESAInfo;
192  var
193    x : word;
194  begin
195    AllocateLowMem( LowMemoryBlock ,300 );
196    FillChar( RealRegs, SizeOf( RealRegs ), 0);{zunächst mit 0 füllen}
197    RealRegs.es := LowMemoryBlock.RealModeSeg;
198    RealRegs.ax := $4f00; { get VESA info }
199    if SimRealModeInt( $10, RealRegs ) = false then
200      begin { falls ein Fehler aufgetreten ist }
201        writeln( 'GetSimRealModeIntFehler: ', SimRealModeIntErrorCode);
202        halt(1);
203      end;
204    for x:=0 to sizeof( VesaInfo)-1 do { kopieren nach VESAInfo }
205      mem[ seg( VESAInfo ) : ofs( VESAInfo) + x ] :=
206        mem[ LowMemoryBlock.ProtModeSel: x ];
207    FreeLowMem( LowMemoryBlock );
208  end;
```

```
209
210
211  procedure GetModeInfo( mode : word );
212  var x : word;
213  begin
214    AllocateLowMem( LowMemoryBlock,300 );
215    FillChar( RealRegs, SizeOf( RealRegs ), 0 );
216    RealRegs.es := LowMemoryBlock.RealModeSeg;
217    RealRegs.cx := mode;
218    RealRegs.ax := $4f01; { holt sich fnr Mode mode die VESA Info }
219    if SimRealModeInt( $10, RealRegs ) = false then
220      begin
221        writeln( 'GetSimRealModeIntFehler: ', SimRealModeIntErrorCode);
222        halt(1);
223      end;
224    for x := 0 to sizeof( ModeInfo ) - 1 do { kopieren }
225      mem[ seg( ModeInfo ) : ofs( ModeInfo ) + x ] :=
226        mem[ LowMemoryBlock.ProtModeSel : x ];
227    FreeLowMem( LowMemoryBlock );
228    granny := 64 div ModeInfo.granularity;
229  end;
230
231
232  procedure InitVideo101h;
233  begin
234    SetWindow101h( 0, 0, 639, 479 );
235    GetVESAInfo;
236    if VesaInfo.Signature <> 'VESA' then
237      begin
238        writeln('VESA VBE-Treiber nicht gefunden!'); halt(1);
239      end;
240    GetModeInfo( Mode101h );
241    writeln( 'Granularität   : ', ModeInfo.granularity, ' KBytes' );
242    writeln( 'Bytes/Scanline : ', ModeInfo.bytesperscanline );
243    writeln( 'Breite         : ', ModeInfo.width );
244    writeln( 'Höhe           : ', ModeInfo.height );
245  end;
246
247
248  procedure ActivePage101h( var page : TPage101h );
249  begin
250    ActVPage101h := page; { Seite, auf der gearbeitet werden soll }
251  end;
252
253
254  procedure CopyP2P101h( var DstPage, SrcPage : TPage101h );
255  var
256    b : byte;
257    Src, Dst : pointer;
258  begin
259    asm push ds end;
260    for b := 0 to 5 do { kopier Block 0 bis 5 von SrcPage in die }
261      begin           { entsprechenden Blöcke von DstPage }
262        Src := SrcPage[ b ];
263        Dst := DstPage[ b ];
264        asm
265          les di, Src
266          lds si, Dst
267          mov cx, 16000
268          db $66; rep movsw { movsd }
269        end;
270      end;
271    asm pop ds end;
272  END;
273
274
275  { kopiert die virtuelle Seite zeilenweise auf den Bildschirm;
276    dies ist nötig, das sonst das Bild zu flackern beginnt }
277  procedure CopyLines( s1, o1, s2, o2 : word; off : word;
278                       lines : byte ); assembler;
279  asm
280    push ds
281      mov al, lines        { Anzahl der zu kopierenden Zeilen }
282      mov es, SegA000      { nach SegA000:off kopieren }
283      mov di, off          { Offset bezüglich der aktuellen Bank }
284      mov si,o1            { Offset des linken Blocks }
285      mov dx,o2            { Offset des rechten Blocks }
286      mov ds,s1            { Segment des linken Blocks }
287      mov bx,s2            { Segment des rechten Blocks }
288      @loop1:
289        mov cx,80          { linke Seite; 80 DWords=320 Pixel kopieren}
290        db $66; rep movsw  { movsd // und kopieren }
291        xchg si,dx         { Werte von [DS:SI] für rechte Seite holen }
292        push ds            { und [DS:SI] der linken Seite sichern }
293        mov ds,bx
294        pop bx
295        mov cx,80          { rechte Seite; 80 DWords = 320 kopieren }
296        db $66; rep movsw  { movsd }
297        xchg si,dx         { Werte von [DS:SI] für linke Seite holen }
298        push ds            { und [DS:SI] der rechten Seite sichern }
299        mov ds,bx
300        pop bx
301        dec al
302        cmp al, 0          { bereits alle Zeilen kopiert ? }
303      jnz @loop1
```

```
304    pop ds
305  end;
306
307
308  { bei den Zeilen, wo ein Bankwechsel stattfindet, einzeln kopieren;
309    Routine zum Kopieren von length Bytes eines Blocks }
310  procedure CopyDWord(DstOffset,
311                      SrcSegment, SrcOffset, length : word); assembler;
312  asm
313    push ds
314      mov es, SegA000   { [ES:DI] = Zieladresse der aktuellen Bank }
315      mov di, DstOffset
316      mov ds, SrcSegment { [DS:SI] = Quelladresse }
317      mov si, SrcOffset
318      mov cx, length     { Angabe erfolgt in Anzahl an Words }
319      shr cx, 2
320      db $66; rep movsw { movsd }
321    pop ds
322  end;
323
324
325  procedure CopyP2V101h(VAR page:TPage101h);
326  var s0, s1, s2, s3, s4, s5 : word;
327  begin
328    s0 := seg(page[0]^); s1 := seg(page[1]^); s2 := seg(page[2]^);
329    s3 := seg(page[3]^); s4 := seg(page[4]^); s5 := seg(page[5]^);
330    SetBank( 0 );                              { Bank 0                }
331    CopyLines( s0, 0, s1, 0, 0 , 102 );        { Zeile 0 - 101         }
332    CopyDWord( 65280, s0, 32640, 256 );        { Zeile 102             }
333    SetBank( 1 );                              { Bank 1                }
334    CopyDWord( 0, s0, 32896, 64 );             { Zeile 102 linke Seite}
335    CopyDWord( 64, s1, 32640, 320 );           { Zeile 102 rechte Seite }
336    CopyLines( s0, 32960, s1, 32960, 384, 97 );{ Zeile 103 - 199       }
337    CopyLines( s2, 0, s3, 0, 62464, 4 );       { Zeile 200-203         }
338    CopyDWord( 65024, s2, 1280, 320 );         { Zeile 204 linke Seite}
339    CopyDWord( 65344, s3, 1280, 192 );         { Zeile 204 rechte Seite}
340    SetBank( 2 );                              { Bank 2                }
341    CopyDWord( 0, s3, 1472, 128 );             { Zeile 204             }
342    CopyLines( s2, 1600, s3, 1600, 128, 102 ); { Zeile 205-306         }
343    CopyDWord( 65408, s2, 34240, 128 );        { Zeile 307 linke Seite}
344    SetBank( 3 );                              { Bank 3                }
345    CopyDWord( 0 , s2, 34368, 192 );           { Zeile 307 linke Seite}
346    CopyDWord( 192, s3, 34240, 320 );          { Zeile 307 rechte Seite}
347    CopyLines( s2, 34560, s3, 34560, 512, 92 );{ Zeile 308-399         }
348    CopyLines( s4, 0, s5, 0, 59392, 9 );       { Zeile 400-408         }
349    CopyDWord( 65152, s4, 2880, 320 );         { Zeile 409 linke Seite}
350    CopyDWord( 65472, s5, 2880, 64 );          { Zeile 409 rechte Seite}
351    SetBank( 4 );                              { Bank 4                }
352    CopyDWord( 0, s5, 2944, 256);              { Zeile 409 rechte Seite}
353    CopyLines( s4, 3200, s5, 3200, 256, 70 );  { Zeile 410-479         }
354  end;
355
356
357  procedure ClearPage101h( page : TPage101h );
358  var
359    block : byte;
360    ppage : pointer;
361  begin
362    for block :=0 to 3 do          { Block 0 bis 3 }
363      begin
364        ppage := page[ block ];
365        asm
366          les di, ppage
367          db $66; xor ax,ax { xor eax, eax }
368          mov cx,64000 / 4  { volle 64000 Byte löschen, = 200 Zeilen }
369          db $66; rep stosw { stosd }
370        end;
371      end;
372    for block := 4 to 5 do       { Block 4 bis 5 }
373      begin
374        ppage := page[ block ];
375        asm
376          les di, ppage
377          db $66; xor ax,ax { xor eax, eax }
378          mov cx,25600 / 4  { 25600 Bytes löschen, d.h. 80 Zeilen }
379          db $66; rep stosw { stosd }
380        end;
381      end;
382  end;
383
384
385  procedure InitPage101h( var page : TPage101h );
386  var
387    block : byte;
388  begin
389    for block := 0 to 5 do InitPage( Page[ block ] );
390    ActivePage101h( page );
391  end;
392
393
394  procedure ClosePage101h( var page : TPage101h );
395  var
396    block : byte;
397  begin
398    for block := 0 to 5 do ClosePage( Page[ block ] );
```

```
399  end;
400
401
402  procedure PutSprite101h( x, y : integer; sprite : TSprite );
403  var
404    block      : word;
405    posx, posy : integer;
406  begin
407    block := 0;
408    repeat
409      { aktiven Block der 640x480 großen Seite auswählen }
410      ActVPage := ActVPage101h[ block ];
411      { Rahmen für den Block entsprechend des großen Rahmens festlegen}
412      SetWindow( Windows101h[ block, 1 ], Windows101h[ block, 3 ],
413                 Windows101h[ block, 2 ], Windows101h[ block, 4 ] );
414      { aus absoluten Koordinaten der 640x480 Seite die Koordinaten des
415        Blocks berechnen, in dem gerade geschrieben werden soll }
416      posx :=  CalcPageX( x, block ); posy :=  CalcPageY( y, block );
417      { Sprite setzen; mit Hilfe der Routine für die 320x200-Auflösung}
418      PutSprite( posx, posy, sprite );
419      inc( block ); { nächsten Block nehmen }
420    until block > 5;
421  end;
422
423
424  procedure GetSprite101h( x, y : integer; sprite : TSprite );
425  var
426    b       : byte;
427    rx1,ry1 : longint;
428    block   : word;
429  begin
430    for block := 0 to 5 do
431      begin
432        rx1 := calcpagex( x, block );
433        ry1 := calcpagey( y, block );
434        ActVPage := ActVPage101h[ block ];
435        SetWindow( Windows101h[ block, 1 ], Windows101h[ block, 3 ],
436                   Windows101h[ block, 2 ], Windows101h[ block, 4 ] );
437        GetSprite( rx1, ry1, sprite );
438      end;
439  end;
440
441
442  procedure PutPixelHelp101h( x, y : integer; c : byte );
443  var
444    block : word;
445  begin
446    block := GetPage(x,y);
447    ActVPage := ActVPage101h[ block ];
448    PutPixel( x mod 320, y mod 200, c );
449  end;
450
451
452  function GetPixelHelp101h(x,y:integer ) : byte;
453  var
454    block : word;
455  begin
456    block := GetPage( x, y );
457    ActVPage := ActVPage101h[ block ];
458    GetPixelHelp101h := GetPixel( x mod 320, y mod 200 );
459  end;
460
461
462  procedure PutPixel101h( x, y : longint; c : byte );
463  begin
464    if ( x >= GlobalWindowX1 ) and ( x <= GlobalWindowX2 ) and
465       ( y >= GlobalWindowY1 ) and ( y <= GlobalWindowY2 ) then
466      PutPixelHelp101h( x, y, c );
467  end;
468
469
470  function GetPixel101h( x, y : longint ) : byte;
471  begin
472    GetPixel101h := 0; { außerhalb des Fensters, dann Farbwert null }
473    if ( x >= GlobalWindowX1 ) and ( x <= GlobalWindowX2 ) and
474       ( y >= GlobalWindowY1 ) and ( y <= GlobalWindowY2 ) then
475      GetPixel101h := GetPixelHelp101h( x, y );
476  end;
477
478  end.
```

Video101.pas greift auf die Datei Video13.pas (c't-Mailbox) zurück, um mit relativ wenig Aufwand Routinen für den SVGA-Modus 101h zu implementieren. Mit kleinen Modifikationen funktioniert das auch für alle anderen 256-Farben-Modi.