COMPUTERBOOKS

VARGA MÁRTON

Játékprogramok készítése

Pascal és Assembly nyelven

Játékprogramok készítése

Pascal és Assembly nyelven

VARGA MÁRTON

Játékprogramok készítése

Pascal és Assembly nyelven

LEKTOR
LÁSZLÓ JÓZSEF

COMPUTERBOOKS
BUDAPEST, 1998

A könyv készítése során a Kiadó és a Szerző a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható. Az ismeretanyag felhasználásának következményeiért sem a Szerző, sem a Kiadó felelősséget nem vállal.

ISBN: 963 618 184 5

©Kiadó: ComputerBooks Kiadói Kft.
1126 Bp., Tartsay Vilmos u. 12.
Tel.: 375-15-64; Tel./fax: 375-35-91
E-mail: info@computerbooks.hu
http://www.computerbooks.hu
Felelős kiadó: ComputerBooks Kft. ügyvezetője
Borítóterv: Székely Edith

ÁFÉSZ Nyomda Vác,
Felelős vezető: dr. Hemela Mihályné

Tartalomjegyzék

Előszó

Sokan vannak, akik a számítógéppel csak játszanak. Sokkal kisebb azoknak a tábora, akik kedvtelésből, időtöltésből saját maguk is írnak játékokat, még akkor is, ha ezek színvonala jócskán alulmarad a kereskedelmi forgalomban lévőknél. Igaz, hogy egy játékprogram elkészítése lényegesen nehezebb, mint egy kész játék használata, viszont sokkal értelmesebb tevékenység annál. A legtöbb játék csak a kézügyességet fejleszti, míg játékprogram-íráshoz szükséges némi fantázia, kreativitás, ötletek, grafikai készség stb. is.

E könyv és lemezmelléklete segítségével készíthető játékok – megjelenésüket tekintve – kétdimenziósak, felbontásuk 320×200, színeik száma 256. A könyv megértéséhez elengedhetetlen a **Pascal** és az **assembly** nyelv ismerete, játékok készítéséhez azonban elegendő a Pascal, mert a lemezmellékleten található egy olyan programkönyvtár (*Game* unit), ami magába foglalja a játékok készítéséhez nélkülözhetetlen gépi rutinokat.

A lemezmellékleten található programok futtatásához és a könyv alapján készített játékokhoz az alábbiakra lesz szükség:

- egy IBM AT típusú számítógépre,
- egy 286-os vagy annál fejlettebb processzorra,
- egy tetszőleges VGA kártyára,
- Turbo Pascal 7.0 fejlesztői környezetre.

A programok szép és egyenletes futásához azonban ez nem elég, ehhez legalább egy 80-90 MHz-es processzor és egy PCI buszos VGA kártya kell.

Játékprogramok készítéséhez sok sikert kíván

A szerző
jatek@elender.hu

HÁROM RÉSZES NYELVOKTATÓ CD-ROM SOROZAT RAJZFILMMEL

COMPFAIR '97
VÁSÁRDÍJ

HunDidac '97
EZÜST DÍJ

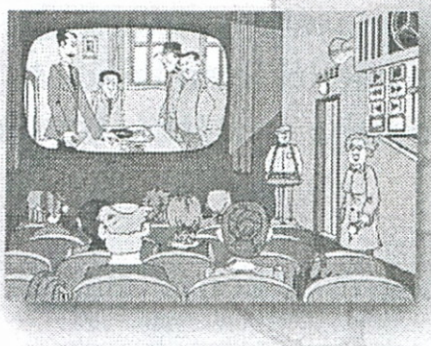
lopva ANGOLUL

— CD-nként ——

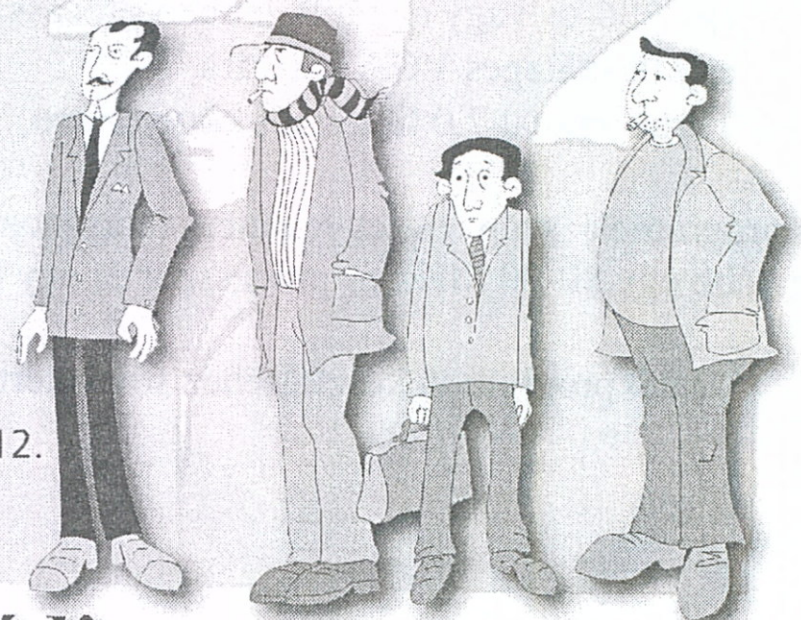
A 60 perces rajzfilm 12 epizódjához 360 feladat,illetve gyakorlat tartozik: szövegértési, olvasási, írási, szókincset bővítő, nyelvtani tudást fejlesztő gyakorlatok, mikrofonos drillek, valamint játékok. INTERNETES feladatjavítási lehetőség!

Mindezeket 1500 szavas hangos szótár, kifejezéstár és 50 oldalas, lényegre törő multimédia nyelvtankönyv egészíti ki.

A CD-k anyagai 80-100 óra alatt dolgozhatók fel.



KULCS AZ ANGOL NYELVHEZ



ÉRDEKLŐDÉSÜKET
AZ ALÁBBI CÍMEKEN VÁRJUK:
ComputerBooks
1126 Budapest, Tartsay V. u. 12.
Tel.: 1/ 175-1564
info@computerbooks.hu

Profi-Média Kft.

Profi-Média, a minőségi oktatóprogramok gyártója
6500 Baja, Déri Frigyes sétány 4. • Tel./fax: (36) 79/325-467
www.profi-media.com • pmed@profi-media.com

1. Alapfogalmak

1.1. A VGA kártya üzemmódjai, az MCGA üzemmód

A VGA kártyák többféle üzemmódban képesek működni, melyek alapvetően kétfélék lehetnek, szöveges és grafikus módok. A szöveges módok karakteres képernyőt használnak, felbontásuk változó, a leggyakrabban használt a 80×25 karakteres képernyőfelbontás. A grafikus képernyő pixeleket tartalmaz, melyek lehetnek kettő, 16 és 256 színűek. Nem érdemes a különböző üzemmódok felbontását, színeinek számát stb. elemezgetni, mert játékaink csak az **MCGA** (vagyis a 320×200/256 színű) üzemmódot alkalmazzák.

MCGA videomódban a képernyő felbontása **320×200**, azaz vízszintesen 320, függőlegesen 200 pixelt tartalmaz. Ez a felbontás elég gyenge a ma használatos SVGA felbontásokhoz képest, viszont előnye, hogy könnyen kezelhető és minden VGA kártyán alkalmazható. Az MCGA az egyetlen a hagyományos szabványos VGA grafikus módjai közül, amelyik **256** színt tud egyszerre megjeleníteni. A memóriában a kezdőcíme **A000:0000**, egy bájt egy pixelnek felel meg, melyek sorfolytonosan vannak tárolva, balról jobbra, fentről le. A képernyő helyigénye 320×200, azaz 64000 bájt, ami majdnem lefoglalja a teljes videomemóriát (az $A000-s szegmenst).

A matematikában használthoz hasonló koordináta-rendszert használunk, hogy a pontokat meg tudjuk különböztetni, csak itt az ordinátatengely (Y tengely) fordított állású, lefelé mutat. A (0;0) pont, az origó, a képernyő bal felső sarkában található képpont, így címe: $A000:0000. Ettől jobbra az első, ettől lefele a második koordináta nő. Például a $A000:0005 című pont koordinátái: (5;0).

A koordináták értelemszerűen csak természetes számok lehetnek. Egy pont abszcisszája (első koordinátája) bele kell hogy essen a [0..319] intervallumba, ordinátájának (második koordinátájának) pedig [0..199] között kell lennie. Ez a képernyő MCGA üzemmódbeli mérete miatt van (320×200).

1.2. Képpont kigyújtása

Azoknál a játékoknál, melyeknél a fő hangsúly a grafikán van, nagyon fontos tudni, hogyan kell a képernyő egy pontját valamilyen színűre befesteni. Minden rajzolás alapja a pontrajzolás, ezért legelőször ezt kell megismerni. Szerencsére az MCGA képszerkezet felépítésének köszönhetően ez nem túl nehéz feladat.

Egy (X;Y) koordinátájú pont kirajzolásához szükséges képlet:

CÍM=320*Y+X

Ez nem egy nagy ördöngösség, ebből is láthatjuk, hogy a memóriában a sorok egymást követően helyezkednek el. Tehát az A000. szegmensen az első 320 bájt az első sornak, a második 320 a második sornak felel meg, és így tovább. A következő rövid példaprogram bemutatja az MCGA üzemmód be- és kikapcsolását, és pixelek kirajzolását.

```
{putpixel.pas}

procedure PutPixel( X,Y: word; C: byte); assembler; asm
                        { (X;Y) helyzetű C színű pont rajzolása   }
   mov     es,sega000   { A videómemória szegmensét a SEGA000 szó }
                        { tartalmazza, értéke $A000               }
   mov     ax,320       { A kép szélessége 320 pixel              }
   mul     Y            { A pont sorának kezdőcíme: 320×Y         }
   add     ax,X         { A pont címe: 320×Y+X                    }
   mov     di,ax        { ES:[DI] a felgyújtandó pont címe        }
   mov     al,C         { C-t kell írni ebbe a bájtba             }
   mov     es:[di],al   { Pixel kigyújtása                        }
   end;

procedure MCGA_On; assembler; asm
                        { MCGA üzemmód bekapcsolása               }
   mov     ax,0013h     { 00h üzemmód, 13h funkció                }
   int     10h          { 10h sorszámú megszakítás                }
   end;

procedure MCGA_Off; assembler; asm
                        { Visszatérés a szöveges üzemmódhoz       }
   mov     ax,0003h     { A 80×25-ös mód a 03h funkció            }
   int     10h          { Megint a 10h megszakítást használjuk    }
   end;
```

```
begin
 MCGA_On;
 PutPixel(   0,   0, 12); { A képernyő négy sarkába négy különbö- }
 PutPixel( 319,   0, 13); { zö színű pont kirajzolása             }
 PutPixel(   0, 199, 14);
 PutPixel( 319, 199, 15);
 readln;                  { Enter megnyomására kilép              }
 MCGA_Off;
 end.
```

Az első eljárás, a *PutPixel*, egy **C** színű és (**X;Y**) koordinátájú pontot rajzol a képernyőre. A memóriába írni Turbo Pascalban a MEM[ ] tömbbel lehet, tehát a *PutPixel* eljárás helyett elég lett volna a MEM[$A000:320*Y+X]:=C értékadó utasítás is. Viszont a későbbiekben főleg assembly eljárásokat alkalmazunk, ezért ismerni kell a pixel kirajzolásához szükséges assembly eljárást is.

A program először bekapcsolja az MCGA üzemmódot, a **10h megszakítás** segítségével. Játékok készítése szempontjából a 10h megszakítás funkciói nem fontosak, csupán ez a kettő, amivel be-, illetve kikapcsoljuk a 320×200/256-os képszerkezetet.

Ezután négy, egy piros, egy lila, egy sárga és egy fehér pontot gyújtunk ki a képernyő négy sarkába, majd kilépés előtt visszaállítjuk a szöveges üzemmódot. Ezt sohase felejtsük programunk befejezése elé beiktatni, mert lehet, hogy csak gépünk újraindításával tudunk később karakteres képernyőt használni.

Pixelek rajzolása nem túl bonyolult, először ki kell számolni a képcímet a koordinátákból (320×Y+X), majd erre a címre ki kell írni azt a bájtot, ami a kívánt színhez tartozik. Az eljárásnak rögtön az első sorában található a *SegA000* változó. Ez a *System* unit egyik tipizált konstansa (inicializált kezdőértékű változója), értéke $A000, vagyis az MCGA kép szegmenscíme. Előnye pedig az, hogy a szegmensregisztereknek egy utasítással érték adható, nem kell valamelyik általános célú regisztert, például az akkumulátor regisztert közbeiktatni. A két alábbi assembly utasítás

```
mov     ax,0a000h
mov     es,ax
```

helyett tehát elég a következő:

```
mov     es,sega000
```

További előnye ennek a konstansnak, hogy védett módban is használható, míg az előző megoldás ott nem működne, hiszen védett módban a szegmensek szervezése egészen máshogy működik, mint valós módban.

1.3. Pixel színének lekérdezése

Csak a teljesség kedvéért foglalkozunk ezzel a problémával, ami valójában egy játék írása során szinte sohasem kerül elő. Ugyanazt a képletet használjuk, amit a pixel kirakásánál, csak nem írjuk, hanem olvassuk a képernyőről az egybájtnyi adatot. A legérdekesebb dolog az egész függvényben talán az, hogy a visszatérési értékét az AL regiszterben kell megadni. Ez jól alkalmazható bármely más assembly függvényben, például a

```
function Fgv: byte; assembler;
 asm
  ...
  end;
```

szubrutin végén ha az AL-be beírunk 3-at, az lesz a *Fgv* értéke, úgy, mint ha Pascalban írtuk volna, és a végén kiadtuk volna az FGV:=3; értékadó utasítást.

```
function GetPixel( X,Y: word): byte; assembler; asm
                        { Az (X;Y) koordinátájú pont színe          }
   mov    es,sega000    { A videómemória szegmensét a SEGA000 szó }
                        { tartalmazza, értéke $A000                 }
   mov    ax,320        { A kép szélessége 320 pixel                }
   mul    Y             { A pont sorának kezdőcíme: 320×Y           }
   add    ax,X          { A pont címe: 320×Y+X                      }
   mov    di,ax         { ES:[DI] a leolvasandó pont címe           }
   mov    al,es:[di]    { A függvény visszatérési értéke AL-ben     }
   end;
```

1.4. A BOB-ok

A **BOB**-ok olyan téglalap alakú grafikai objektumok, melyek tetszőlegesen mozgathatóak, eltüntethetőek és megjeleníthetőek. Szintén 256 színűek, több fázisból (mozdulatokból) állhatnak. A 'BOB' elnevezés az Amiga gépekről származik, a ***Blitter OBject*** rövidítése. Akik jártasak a Commodore 64 programozásában, azok találkoztak már a 'Sprite' fogalommal. Nos, a BOB-ok is hasonló funkciót látnak el, mint a Sprite-ok, azzal a különbséggel, hogy a Sprite-ok előállítása a hardver feladata, a BOB-ok programozásáról viszont nekünk

kell gondoskodni, szoftveres úton. A bobot tehát nevezhetnénk akár szoftver SPRITE-oknak.

Egyszerre több BOB-bal is dolgozhatunk, ezek egymástól és a háttértől is függetlenek. BOB-nak tekinthető például az egér ikonja grafikus képernyőn, ami általában nyíl formájú. Vagy például BOB egy játékban egy futó ember, ráadásul több fázisú. Egy többfázisú BOB olyan, mintha több, azonos méretű egyfázisú BOB-ot jelenítenénk meg egymás után. Egy BOB bizonyos részein áttetsző (**transzparens**) lehet, vagyis ott az látszik, ami mögötte van (háttér, egy másik BOB).

Shape-nek nevezzük a BOB grafikus adatait a memóriában. Egy egyfázisú, W szélességű, H magasságú Shape helyfoglalása W×H bájt. Ha ennek a BOB-nak nem csak egy, hanem P fázisa van (P>1), akkor bájtban mért memóriaigénye P×W×H. A Shape-et dinamikus változóként tároljuk, *GetMem*-mel foglalunk neki P×W×H bájt hosszúságú memóriát. A BOB adatait az adatszegmensben, *Object* típusú változóként, objektumként deklaráljuk. Ide tartozik pl. a BOB szélessége, magassága, fázisainak száma, bájtban vett helyfoglalása, sorszáma, és itt adjuk meg a Shape-re mutató pointert is.

1.5. Az MCGA kép elméleti megjelenítése

MCGA képszerkezet esetében a képernyőn megjelenítendő adatok az $A000 szegmens első 64000 bájtján helyezkednek el, a bal felső sarokban lévő pontot pl. a MEM[$A000:0000] tömbbel érhetjük el. Egy bájt egy képpont színét határozza meg. A VGA kártya ebből a tartományból olvassa ki a képpontokhoz szükséges adatokat, majd ezeket a bájtokat a monitor számára is „érthető" jelekké alakítja. A monitornak másodpercenként legalább 25-ször kell felfrissítenie a képet ahhoz, hogy szemünk a változásokat folyamatosnak érzékelje, ne legyen darabos és villogó.

A képalkotás a monitorban az **elektronsugár** segítségével megy végbe. Ez a sugár minden másodpercben többször végigpásztázza a képcső felénk eső téglalap alakú részét, a képernyőt. Szemből nézve a bal felső sarokból indul, végighalad egy soron, vízszintesen, majd újra a képernyő bal oldalára ugrik, a következő sorra. Ezt a bal oldalra történő visszatérést nevezzük horizontális visszafutásnak (**horizontal retrace**), amely minden sor végigpásztázása után bekövetkezik. Amikor az utolsó soron is végigszaladt az elektronnyaláb, bekö-

vetkezik a vertikális visszafutás (**vertical retrace**), amikor a jobb alsó sarokból a bal felsőbe ugrik. Ezután kezdődhet az egész elölről.

Az elektronsugár tehát kigyújtja a képernyő pontjait. Ezek nem alszanak el rögtön, hanem továbbra is „ingerelt" állapotban vannak, tovább „világítanak". Ezért ha egy képpont hosszú ideig ugyanolyan színű, akkor az folyamatosan látszik, nem alszik el teljesen.

A televíziós készülékekben az elektronsugár először a páratlan, majd a páros sorokon halad végig. Másodpercenként 50 félképet jelenít meg, azaz 25 teljes képet. Ekkor mondjuk, hogy a TV-készülék 50 Hz-es. A monitorok általában 60-70 Hz-esek, de itt nem mindig érvényes ez a félkép-kirakós módszer. MCGA üzemmódban úgy kell az egészet elképzelni, hogy az elektronsugár folyamatosan rajzol, kivéve horizontális és vertikális visszatérésekor. Ilyenkor kialszik a nyaláb, hiszen ha ilyenkor is „felgyújtaná" a képpontokat, az zavaró lenne.

A horizontális és a vertikális visszafutást figyelni tudjuk, meg tudjuk várni amíg egy ilyen visszafutás bekövetkezik. A VGA kártya $03DA állapotregisztere alkalmas erre a célra. A 0. bit a vertikális vagy horizontális, a 3. bit csak a vertikális visszafutást jelzi. Általában ezt a bitet figyeljük, mert ha egy vertikális visszafutás közben kezdünk a képernyőre írni, akkor az egyszerre és egyben fog megjelenni. Ha nem figyeljük az elektronsugarat, és úgy váltogatjuk egymás után a különböző tartalmú képeket, akkor ez darabossághoz vezet.

Vegyünk egy példát: fekete és fehér képernyőket váltogassunk egymás után. Ez egyszerű memóriába írással megvalósítható, 64000 darab 0, utána megint 64000 darab 15 értékű bájtot írunk ki az $A000:0000 címtől kezdve. Az elektronsugár haladása lassabb, mint a memóriába írás. Tegyük fel, hogy a képernyő közepén jár, és a 100. sort kezdi éppen frissíteni. A kép eddig fekete volt, de mi most hirtelen fehérre változtatjuk, azaz a videomemória bájtjait 15-ös értékkel töltjük fel. Ez elég gyorsan végbemegy, tegyük fel, hogy közben az elektronsugár csak a 101. sorig jutott. Eddig csak fekete pontokat rajzolt (azaz nem rajzolt semmit), hiszen a képmemóriában nulla értékű bájtokat talált. A képernyő felső része tehát fekete. Most a 101. sor kirajzolása kezdődik. E sor első képpontjához tartozó bájt értéke viszont már nem nulla, hanem 15, hiszen előzőleg az egész képmemóriát 15-tel telítettük. Tehát innentől kezdve a képernyő fehér

lesz. Egyszóval a kép felső fele fekete, az alsó fehér egészen addig, amíg be nem következik egy újabb frissítés.

A kép tehát feketéről fehérre váltott, de ez a váltás a képernyő közepén ment végbe. Ha sokszor egymás után végzünk el ilyen teljes képernyős színváltásokat, akkor az ezzel a módszerrel darabos lesz, nem lesz szép. De nézzünk egy rövid példaprogramot, mely illusztrálja az ilyen színváltások zavaró hatását. (Érdemes megjegyezni, hogy a teszt sem a szemnek, sem a monitornak nem tesz jót.)

```
{villog1.pas}

uses Crt;

procedure Fill( C: byte); assembler; asm
                          { Képernyő befestése C színűre              }
   mov    es,sega000      { A SEGA000 értéke előre meghatározott,     }
   xor    di,di           { A $A000:0000 címtől indulunk (ES:[DI])    }
   mov    cx,32000        { 320x200 bájt=32000 szó kiírása            }
   mov    al,C
   mov    ah,al
   rep    stosw           { Így a leggyorsabb és a legrövidebb        }
   end;

begin
 asm
   mov    ax,13h
   int    10h             { MCGA üzemmód bekapcsolása                 }
  end;
 repeat
  Fill( 0); Fill(15);     { A képernyő váltakozik, fekete vagy fehér}
  until keypressed;       { Billentyűnyomásra vége                    }
 readkey;                 { A gomb kódja ne maradjon a pufferben      }
 asm
   mov    ax,03h
   int    10h             { Visszatérés a szöveges módhoz             }
  end;
 end.
```

A program – mint ahogy azt szerettük volna – váltogatja a képernyő színét, egyszer feketére, egyszer pedig fehérre. Elvileg. A gyakorlatban ez ennél a példánál nem valósulhat meg, mert a képernyő sohasem lesz teljesen fehér vagy fekete. Mindig lesz benne egy törésvonal, és ráadásul mindig más helyen.

Ha az elektronsugarat figyeljük, és a vertikális visszafutás bekövetkeztekor kezdjük el kirakni a 64000 bájtot, a váltás nem lesz darabos, villogó. Ehhez persze szükséges az is, hogy a bájtok írásának sebessége gyorsabb legyen a visszatérésnél, ellenkező esetben az adatkivitelt beéri, és így hasonlóan darabos lesz a kép (csak a törésvonal körülbelül állandó helyen látható). De egy 80 MHz-es gép sebessége már „legyőzi" az elektronsugarat. Nézzük az eljárást, ami egy vertikális visszafutás bekövetkeztéig várakozik!

```
procedure Retrace; assembler; asm
                          { Vertikális visszafutásra várakozás      }
   mov    dx,03dah
@1:                       { A $3da port 3. bitje 1, ha éppen felfe- }
   in     al,dx           { le halad az elektronsugár               }
   test   al,8
   jz     @1
   end;
```

Ezt a szubrutint mindig közvetlenül a képre írás előtt alkalmazzuk, mert ha korábban tesszük, akkor az elektronsugár már elkezd rajzolni, és a kép darabos lesz. Bővítsük a **VILLOG1.PAS** deklarációs részét a fenti *RetRace* eljárással, és a *repeat ... until* főciklusát a következőre módosítsuk (a bővített program a **VILLOG2.PAS** a lemezmellékleten):

```
repeat
  Retrace;
  Fill( 0);
  Retrace;
  Fill(15);
  until keypressed;
```

Még egyszer az egész elektronsugár-figyelés lényege: ha akkor módosítjuk a képernyő tartalmát, amikor az elektronsugár függőlegesen visszatér, a változás egyszerre következik be, nem lesz darabos. Ez persze feltételezi azt, hogy a képmemória bájtjainak változtatása nem tart több ideig, mint egy vertikális visszafutás, és amint elkezdődik egy ilyen visszafutás, rögtön el kell kezdeni módosítani, hogy elég legyen az idő. Tehát az egyenletes futás érdekében nemcsak alkalmazni kell a fenti (*RetRace* eljárásban elhelyezkedő) sorokat, hanem jó helyen kell alkalmazni.

1.6. A paletta

A színes monitorok képpontjai három, egy vörös, egy zöld és egy kék részből állnak. A képpont színe ennek a három összetevőnek (**alapszín komponensnek**) az intenzitásától függ. A VGA kártyák az MCGA képszerkezetnél 256 színt tudnak egyszerre megjeleníteni. Minden színhez hozzá van rendelve három [0..63] zárt intervallumba eső egész szám, mely a szín összetevőinek erősségét határozza meg. Az MCGA módban a paletta felépítése a következő. Minden színösszetevő egy bájton van tárolva, amely bájtnak csak az alsó 6 (0-5.) bitje használatos, a két legmagasabb helyértékű bit állapota lényegtelen. Minden színnek három, vörös (**red**), zöld (**green**) és kék (**blue**) összetevője van, így a paletta mérete 3×256=768 bájt. A következő táblázatban az alapértelmezett VGA paletta első tizenhat színének **RGB** (vörös-zöld-kék) komponensei olvashatóak. (Alapértelmezett paletta a beépített, az MCGA videómód bekapcsolása után közvetlenül aktív paletta.)

Szín	Neve	Crt-azonosító	R	G	B
0	Fekete	Black	0	0	0
1	Kék	Blue	0	0	42
2	Zöld	Green	0	42	0
3	Türkiz	Cyan	0	42	42
4	Vörös	Red	42	0	0
5	Lila	Magenta	42	0	42
6	Barna	Brown	42	21	0
7	Világosszürke	LightGray	42	42	42
8	Sötétszürke	DarkGray	21	21	21
9	Világoskék	LightBlue	21	21	63
10	Világoszöld	LightGreen	21	63	21
11	Világos türkiz	LightCyan	21	63	63
12	Világos piros	LightRed	63	21	21
13	Világos lila	LightMagenta	63	21	63
14	Sárga	Yellow	63	63	21
15	Fehér	White	63	63	63

Ebből a táblázatból körülbelül látni lehet, hogy melyik komponens milyen arányú keverése milyen színt eredményez. Sejthetjük, hogy a sötétlila komponensei például a 21,0,21 lehetnek, és hogy szürke színeket úgy állíthatunk elő, hogy a komponensek értékei megegyeznek. Érdemes a lemezmellékleten talál-

ható és az 1.6.3. fejezetben leírt **RGBSET.PAS** példaprogrammal kísérletezni, hogy később, a grafikák készítésénél már jártasak legyünk az additív színkeverés sajátosságaiban.

A keret színe megegyezik a 0. színnel, ami alapállapotban fekete. Ezért ezt nem érdemes módosítani, mert ha megváltoztatjuk, nem lesz olyan szép a játékunk, nem fog úgy tűnni például egy beúszó BOB, hogy a képernyő széléből csúszik be. Ettől függetlenül persze éppúgy megváltoztathatjuk, mint bármely másik színt, ha ez a célunk.

A paletta színeit tetszőlegesen módosíthatjuk, így minden szín 64×64×64=262144 féle RGB értéket vehet fel. Összesen tehát ennyi, egyszerre azonban 256 a megjeleníthető színek száma. A változtatás a VGA kártya $03C7-$03C9 regiszterein keresztül valósulhat meg.

1.6.1. Egy szín RGB-komponenseinek olvasása

Nagyon egyszerű feladat. A **$03C7 olvasás címregiszterbe** (*RGB Read Adress)* az olvasni kívánt szín számát írjuk. Ezután a **$03C9 adatregiszter** (*RGB Data*) háromszor egymás után olvasva megkapjuk a három értéket, először a vörös (**R**), majd a zöld (**G**), végül a kék (**B**) összetevőt. A következő eljárás a fejében megadott R,G,B bájt típusú változókban megadja az **N**. szín alapkomponenseinek értékét:

```
procedure GetRGB( N: byte; var R, G, B: byte); assembler; asm
                          { Az N. szín RGB-komponensei                }
   mov     dx,03c7h       { RGB olvasás címregiszter                  }
   mov     al,N           { Jelezzük a kártyának, hogy az N. szín-    }
   out     dx,al          { nel fogunk dolgozni (most: olvasni)       }
   mov     dx,03c9h       { RGB adatregiszter                         }
   in      al,dx          { Első összetevő lekérdezése                }
   les     di,R
   stosb
   in      al,dx          { Második összetevő lekérdezése             }
   les     di,G
   stosb
   in      al,dx          { Harmadik összetevő lekérdezése            }
   les     di,B
   stosb
   end;
```

1.6.2. Egy szín RGB-komponenseinek írása

Ez az előzőhöz hasonlóan szintén nagyon egyszerű feladat. Először a **$03C8 RGB írás címregiszterbe** (*RGB Write Adress*) a kívánt szín számát küldjük, majd a **$03C9** adatregiszter háromszori írásával állíthatjuk be a megfelelő értékeket, a vörös (**R**), a zöld (**G**) és végül a kék (**B**) összetevőt. Erre is nézzünk egy példát, ahol **N** a változtatni kívánt szín száma, R, G, B, pedig az összetevői.

```
procedure SetRGB( N, R, G, B: byte); assembler; asm
                          { Az N. szín összetevőinek változtatása   }
   mov     dx,03c8h       { RGB írás címregiszter                   }
   mov     al,N
   out     dx,al          { N. szín módosítása                      }
   mov     dx,03c9h       { RGB adatregiszter                       }
   mov     al,R
   out     dx,al          { R összetevő írása                       }
   mov     al,G
   out     dx,al          { G összetevő írása                       }
   mov     al,B
   out     dx,al          { B összetevő írása                       }
   end;
```

1.6.3. Példaprogramok a paletta módosításához

Az első program segít gyakorolni az éppen szükséges szín beállítását. Ez a színváltoztató módszer a későbbi grafikáink elkészítésében nagy szerepet fog játszani, mert úgy lehet beállítani egy színt, hogy közben látjuk azt és az alapkomponenseinek értékeit is. Nem lenne ésszerű egy program grafikai részének palettamódosításait számokkal bevinni, azaz nem célszerű egy játék grafikájához úgy kitalálni a színek összetevőit, hogy közben magát a színt nem is látjuk.

A programban a háttér és a keret színét, vagyis a 0. sorszámú színt fogjuk módosítani. Indítás után a bal felső sarokban három fehér (15) számot látunk, ezek a háttér R-G-B alapkomponensei. A következő billentyűk segítségével változtathatjuk ezeket az összetevőket:

Q : R komponens növelése
A : R komponens csökkentése
W : G komponens növelése
S : G komponens csökkentése
E : B komponens növelése
D : B komponens csökkentése

	R	G	B
+	Q	W	E
–	A	S	D

```
{rgbset.pas}

uses Crt;

const
 R: byte = 0;              { Alapállapotban a háttér és a keret,     }
 G: byte = 0;              { vagyis a 0. sorszámú szín fekete, RGB   }
 B: byte = 0;              { összetevőinek értéke nulla              }


var
 C: char;

procedure SetRGB( N, R, G, B: byte); assembler;

 ... { Ez az előző eljárás }

begin
 textbackground( black); { Kivételesen szöveges üzemmódot alkal-  }
 textcolor( white);       { mazunk, a palettát itt is meg lehet   }
 clrscr;                  { változtatni, de csak 16 szín látszik  }
 gotoxy( 1, 3);
 writeln('Billentyűk:  Q/A: R+/-;  W/S: G+/-;  E/D: B+/-;'+
         '  ESC: Kilépés');
 repeat
  gotoxy( 1, 1);
  write( R:3,' ', G:3,' ', B:3);
  C:= readkey;
  case upcase( C) of
   'Q': if R<63 then inc(R); { R összetevő növelése              }
   'A': if R>00 then dec(R); { R összetevő csökkentése           }
   'W': if G<63 then inc(G); { G összetevő növelése              }
   'S': if G>00 then dec(G); { G összetevő csökkentése           }
   'E': if B<63 then inc(B); { B összetevő növelése              }
   'D': if B>00 then dec(B); { B összetevő csökkentése           }
   end;
  SetRGB( 0, R, G, B);   { RGB összetevők változtatása          }
  until c=#27;           { ESC-re kilépés a programból          }
 SetRGB( 0, 0, 0, 0);    { A módosított szín visszaállítása     }
 end.
```

A második példaprogram segítségével megnézhetünk egy háttértárolón tárolt palettafájlt. Általában a 256 színt használó játékok a grafikához szükséges palettát egy **768 bájt** hosszúságú fájlban tárolják, melynek gyakori kiterjesztése a **.PAL**. A színek alapkomponensei egy-egy bájton vannak tárolva, sorban, így jön ki a 768 bájtos hosszúság. Mi is ebben a formátumban mentjük majd el palettáinkat, ezért néha szükséges lehet megtekinteni egy-egy ilyen fájlt. A program paraméterében kell megadni a fájlnevet kiterjesztéssel és útvonallal, ha szükséges, majd futtatás után megjelenik a paletta. Billentyűnyomásra lehet kilépni.

```
{viewpal.pas}

uses Crt;

type Colors = record      { Egy szín komponensei                          }
              R,G,B: byte;
              end;
var
 f: file;                 { Változó a fájlműveletekhez                    }
 x,y: byte;               { A FOR... ciklusokhoz változók                 }
 Palette: array[0..255] of Colors; { Paletta színei                       }

procedure Box8x8( X,Y: word; C: byte);
 var                      { 8×8-as négyzet kirajzolása, bal felső         }
  i,j: byte;              { sarka koordinátái: (X;Y)                      }
 begin
  for i:= 0 to 7 do       { Most nem assembly utasításokat írunk,         }
   for j:= 0 to 7 do      { mert a sebesség annyira nem fontos            }
    mem[$A000:320*(y+i)+(x+j)]:= C;
 end;

procedure SetRGB( N, R, G, B: byte); assembler; asm

 ... { Ez az eljárás már szerepelt feljebb }

 end;

begin
 if paramcount=0 then halt; { Ha a paraméterben nem adunk meg            }
                          { semmit, leáll a program futása               }
 asm
   mov    ax,13h
   int    10h             { MCGA üzemmód bekapcsolása                     }
  end;
 assign( f, paramstr( 1));
 reset( f, 1);
 blockread( f, Palette, 768); { Fájl tartalmának töltése                  }
 close( f);
 for x:= 0 to 255 do with Palette[x] do SetRGB( x, R, G, B);
                          { Paletta beállítása (a 'Palette' változó       }
 for x:= 0 to 15 do       { adatai szerint) ...                           }
  for y:= 0 to 15 do
   Box8x8( x*8, y*8, 16*y+x); { ... és megjelenítése                      }
 readkey;
 asm
   mov    ax,03h
   int    10h             { MCGA üzemmód kikapcsolása                     }
  end;
 end.
```

2. BOB-ok megjelenítése

Ebben a fejezetben megismerkedünk a BOB-ok megjelenítésének különböző lehetőségeivel. Lépésről lépésre haladunk, a legegyszerűbb megjelenítőtől a legösszetettebbig. Minden lehetőséget külön példaprogrammal szemléltetünk, melyekhez igyekszünk részletes magyarázatot adni. A kulcseljárások, azaz a legfontosabb, a megjelenítést végző eljárások mindig assembly sorokból állnak, hiszen – mint azt később látni fogjuk – másodpercenként többször végre kell hajtani ezt a műveletet. Vegyünk egy példát. Egy BOB-ot mozgatni akarunk. A mozgás akkor lesz tökéletes, ha kicsi lépésekből áll, és két lépés között kevés idő telik el. És ha több BOB-bal dolgozunk, már lényegesen számít, hogy ezt a kulcsfontosságú BOB-megjelenítést végző eljárást Pascalban vagy assemblyben írjuk.

A legösszetettebb BOB-megjelenítés a következő lehetőségekkel rendelkezik:

- Egyszerre több BOB és a háttér is látható a képernyőn.
- A háttér független a grafikus objektumoktól, és akár tetszőlegesen változhat is (például a görgetésnél).
- A BOB-ok tetszőleges helyzetűek, méretűek és fázisúak lehetnek.
- A BOB-ok bizonyos részeiken „átlátszóak" (transzparensek) lehetnek, vagyis ezeken a pontokon a mögöttük lévő tartalom látszódik, ami lehet a háttér vagy egy másik BOB része. Ha egy BOB P. fázisának (X;Y) pontja átlátszó, akkor a Shape P×W×H+W×Y+X. bájtja zérus. W a Shape bájtban vett szélességét, H pedig a magasságát jelöli.
- Ha két BOB egymáson helyezkedik el, akkor meghatározott, hogy melyik van „felül", azaz melyik látszik teljes egészében, és melyik van takarásban. (Meghatározott a BOB-ok prioritása.)
- A BOB-ok „kilóghatnak" a képből, azaz néhány esetben csak egy részük látható. Ilyenkor ügyelni kell arra, hogy nem kell és nem is szabad az egész BOB-ot megjeleníteni.
- A villogást és a darabosságot lehetőség szerint legjobban ki kell küszöbölni.

Nehéz lenne elsőre az összes pontot teljesíteni, és megértése is bonyolult lenne. Éppen ezért bontsuk szét a lépéseket, és mindig tegyünk hozzá valamit, hogy a végén megvalósíthassuk a fenti kitételeknek eleget tevő eljárást.

2.1. Egyszerű BOB-megjelenítő

Ez a megjelenítő annyira primitív lesz, hogy csak igen nagy túlzással lehet BOB-megjelenítőnek nevezni. A magunk elé célul kitűzött kritériumok közül egyet sem teljesít. Egyszerre csak egy, 16×16-os méretű, egyfázisú, átlátszatlan BOB-ot tudunk létrehozni. Az egésznek rajta kell lennie a képen, nem tud beúszni. A háttér egyszínű, a megjelenítés pedig meglehetősen villog, élvezhetetlen. Viszont a fő eljárás könnyen érthető, és jó alapot ad a későbbi, bonyolultabb megjelenítőkhöz.

Ez a BOB-ábrázolás nagyon hasonlít a Turbo Pascal *Graph* egységének *Putimage* eljárásához. Lényege az, hogy a memóriának egy bizonyos helyéről (ahol a Shape van) **16×16**, azaz 256 bájtot kell átmásolni a képmemória egy bizonyos bájtjától kezdve (amit a BOB koordinátái alapján határozunk meg), de közben a sorokat szét kell tördelni. Az eljárás vázlata:

1. Tegyük fel, hogy egyszer már lefutott az eljárás, és a BOB látható a képernyőn. A képmemória azon részét, amelyben a BOB helyezkedik el, felül kell írni a háttér színével. Így onnan letakarítjuk a BOB-ot. Most nagyon gyorsnak kell lenni, hiszen ilyenkor nem látszik a BOB, és ha túl sokat várunk, az villogáshoz vezet.
2. A BOB ábrázolása. DS:[SI]-be a Shape, ES:[DI]-be a képernyő kezdőcímét ($A000:0000) töltjük, majd DI-hez hozzáadunk 320×Y+X-et. Tehát kiszámítjuk azt a címet, ahová a BOB kerül, és ezt betesszük a DI regiszterbe. DS:[SI]-ről ES:[DI]-re másoljuk a Shape adatait a *rep movsw* utasítással. Minden sor kirakása után DI-hez hozzá kell adni 304-et (320–16-ot), így a sorok egymás alatt elvágólag helyezkednek el.

A program elején deklaráljuk a *BOB* típust, amely egy rekord. Ebben a koordinátáit és a képmemóriabeli kezdőcímét tároljuk. Ez a változó tárolja a BOB előző helyzetének címét, amit a letörléshez fogunk használni. A BOB típus tartalmaz ezeken kívül egy pointert, ami a Shape kezdőcímére mutat. A deklarációs részben található még a BOB-megjelenítő eljárás, majd a főprogram következik. Itt lefoglaljuk a Shape számára szükséges 16×16=256 bájtot, majd értéket adunk nekik, véletlenszerűen. Az MCGA mód bekapcsolása után feltöltjük a képmemóriát a megadott háttérszínnel, amit a fő ciklus követ, mely a BOB-ot mozgatja és megjeleníti, majd billentyűnyomásra visszalépünk a szöveges üzemmódba, és befejezzük a programot.

```
{showbob1.pas}

uses Crt;                    { A Crt egységre csak a billentyűnyomás   }
                             { figyelésénél lesz szükség               }
type BOB = record            { A BOB nem grafikus adatai:              }
         X,Y: word;          {       szélesség, magasság               }
         CIM: word;          {       előző címe a törléshez            }
         DT:  pointer;       {       a Shape mutatója-grafikus adatok  }
         end;

var    B: BOB;               { B a BOB azonosítója                     }
       i: word;              { A FOR..TO..DO ciklusokhoz               }
const C= 1;                  { A háttér sötétkék                       }

procedure ShowBOB; assembler; asm { A megjelenítő eljárás              }

 { 1. BOB letörlése, azaz felülírása C (háttér) színű bájtokkal }

   mov     es,sega000        { A segA000 értéke a Turbo Pascal 7.0-ban }
                             { előre megadott, $A000                   }
   lea     bx,B              { BX-be tölti a BOB ofszetcímét           }
   mov     di,[bx+4]         { DI-be a BOB előző helyzetének címét     }
   mov     al,C              { AX bájtjaiba a háttérszín kerül, ezzel  }
   mov     ah,al             { írjuk felül a BOB-ot                    }
   cld                       { Növekvő sorrendben írunk                }
   mov     dx,16             { 16 sor van,                             }
@1:mov     cx,8              { és 16 oszlop, aminek a fele 8 (azért a  }
                             { fele, mert szavakat írunk, nem bájtokat)}
   rep     stosw             { Egy sor törlése                         }
   add     di,304            { DI a következő sor kezdőcíme            }
   dec     dx                { A sorszámláló csökkentése              }
   jnz     @1                { Ismétlés, amíg el nem fogynak a sorok   }

 { 2. BOB kirakása az új helyre }

   mov     ax,320            { Új cím számítása a CIM=320×Y+X képlettel}
   mul     word [bx+2]       { A B BOB címéhez (BX) 2-t adva megkaphat-}
                             { juk a BOB ordinátáját                   }
   add     ax,[bx]           { [BX] pedig pont az abszcisszát adja     }
   mov     di,ax             { DI-ben előállt a képcím,                }
   mov     [bx+4],di         { amit előrelátóan elmentünk, hogy a tör- }
                             { léshez ne kelljen ismét számolgatni     }
   push    ds                { Az adatszegmens megváltozik a MOVSW     }
                             { utasítás miatt, ezért el kell menteni   }
   lds     si,[bx+6]         { DS:SI a grafikus adatok kezdőcíme       }
   mov     dx,16             { 16 sor másolása,                        }
@2:mov     cx,8              { és 16 oszlopé                           }
   rep     movsw             { Az aktuális sor megrajzolása            }
   add     di,304            { DI a következő sor kezdőcíme            }
   dec     dx                { Egy sort kirajzoltunk                   }
   jnz     @2                { A többi sor kirajzolása, amíg DX>0      }
   pop     ds                { Vissza az eredeti adatszegmenst         }

   end;
```

```
begin
 with B do begin
  getmem( DT, 256);       { 16x16=256 bájt lefoglalása a Shape-nek  }
  CIM:= 0; X:= 0; Y:= 92; { Kezdőértéket adunk a nem grafikus       }
  randomize;              { adatoknak                               }
  for i:= 0 to 255 do mem[seg(DT^):ofs(DT^)+i]:= random( 256);
                          { A BOB pontjai véletlenszerűek           }
  asm
    mov   ax,13h
    int   10h             { MCGA üzemmód bekapcsolása               }
   end;
  fillchar( mem[$A000:0000], 64000, C);
                          { A képernyő C színű (sötétkék)           }
  repeat
   ShowBOB;               { BOB megjelenítése,                      }
   inc( X);               { mozgatása balról jobbra                 }
   if X=304 then x:=0;    { Ha a képernyő szélére ért, az elejére   }
                          { ugrik                                   }
   delay( 50);            { Kis várakozás, hogy látható legyen      }
   until keypressed;      { Billentyűnyomásra kilép                 }
  readkey;                { Billentyűkód ki a pufferből             }
  asm mov ax,3
      int 10h     end;    { MCGA üzemmód kikapcsolása               }
 end;
end.
```

Futás közben is láthatjuk, hogy ez az ábrázolásmód a gyakorlatban így nem alkalmazható. A BOB mozgása nem egyenletes, darabos és villódzó. Viszont gyors. Ha a fő ciklusból eltávolítjuk a *delay* utasítást, akkor szinte nem is látunk semmit, mert a BOB kirakása gyorsabb, mint a képfrissítés.

Egy előnye mégis van ennek az eljárásnak. A megjelenítés alapjait viszonylag könnyű itt elsajátítani, hiszen nehéz lenne megérteni már elsőre a legfejlettebb BOB-ábrázoló eljárást. Ezért az Olvasó, ha nem érti egészen, ne lépjen tovább, mert a következő megjelenítők mind erre az aránylag primitív eljárásra épülnek.

2.2. Tetszőleges háttér

Ha a háttér nem egyszínű, mint az előző példaprogram futása alatt, és a pixelek színei nem szabályos elrendezésűek, azaz ha a videomemóriában található kép tetszőleges, akkor más módszert kell alkalmaznunk a BOB eltávolítására, amire itt is szükség van, hiszen ha koordinátái megváltoznak, akkor el kell valahogy tüntetni, és csak azután szabad elkezdeni kirakni az új helyre, mert nem lehet a BOB egyszerre két helyen.

Bizonyára érthető, hogy miért nem lehet a színfeltöltéses módszert használni. Ekkor ugyanis módosulna a háttér, és nem ez a célunk. A megoldás, hogy a BOB által letakart területet valamilyen formában tárolni kell. Ezt úgy fogjuk megvalósítani, hogy előre lefoglalunk egy Shape hosszúságú részt a memóriából, amire minden egyes kirakás előtt átmásoljuk a kirakandó BOB alatti területet. Eszerint a megjelenítő eljárás három fő lépésből áll. Itt is feltételezzük, hogy az eljárás már egyszer lefutott, vagyis a BOB már látható, és a BOB-hoz tartozó dinamikus változó a BOB által letakart háttérrészletet tárolja.

1. A képernyőn még látható, előző helyzetű BOB-ot el kell tüntetni, rámásoljuk az előzőleg elmentett háttérrészt.
2. Kiszámoljuk a BOB kezdőcímét, és az ott található 16×16-os téglalapot elmentjük.
3. Kirakjuk a BOB-ot.

Legelőször az első lépés kimarad, mert különben a háttér módosulna. Külön eljárásba kerül az 1. lépés (*ClearBOB*) és a 2-3. lépés (*ShowBOB*), így legelőször csak a *ShowBOB*-ot hívjuk meg.

Az elektronsugár figyelésével, vagyis vertikális visszafutásra várakozással megpróbáljuk kiküszöbölni a darabosságot és a villogást. Ez lelassítja ugyan a futást (elvégre minden megjelenítéskor várni kell), de közben szebbé varázsolja.

Megoldjuk még az átlátszás problémáját is, azaz a BOB bizonyos pontjain áttetsző lehet, mely pontokon a háttér megfelelő pixele látható. Ezeket a pontokat úgy jelöljük, hogy a hozzájuk tartozó Shape-bájtoknak zérus értéket adunk. A feladat megoldásához egy *movsw* utasítást kell néhány másik paranccsal helyettesíteni, melyek megvizsgálják, hogy az éppen aktuális Shape-bájt nem nulla-e, mert csak akkor lehet kiírni ha nem nulla. (Természetesen bármelyik színt választhattuk volna transzparensnek. Itt és a továbbiakban a 0-t használjuk erre a célra, mert ez általában fekete, az üres háttér színe).

A programban először lefoglaljuk a szükséges 2×256 bájt hosszúságú memóriát, majd betöltjük a Shape-et, amit a lemezen tárolunk. A háttérre véletlenszerűen kirakunk 1000 tetszőleges színű pontot a MEM[ ] tömb segítségével. Ezután belépünk a megjelenítő ciklusba, ami billentyűnyomásig működik, és a végén visszaállítjuk a szöveges módot (amit a program elején bekapcsoltunk).

```
{showbob2.pas}

uses Crt;                   { Megint csak a billentyű miatt                 }

type BOB = record           { Bájt  Leírás                                  }
           X: word;         {  0.   A BOB abszcisszája, ez a 0. és az       }
                            {       1. bájt a BOB típusban                  }
           Y: word;         {  2.   Ordináta, 2-3. bájt                     }
           A: word;         {  4.   Előző helyzetének címe (4-5.)           }
           S: pointer;      {  6.   Shape mutatója (6-9. bájt)              }
           H: pointer;      { 10.   Ide kerül a BOB mögötti rész            }
           end;

var
 B: BOB;
 f: file;                   { A grafikus adatokat fájlból töltjük be        }
 i: word;

const
 dx: shortint = 1;          { Vízszintes irány jelzője (1=jobbra)           }
 dy: shortint = 1;          { Függőleges irány jelzője (1=le)               }


procedure ClearBOB; assembler; asm

 { 1. BOB letörlése }

   lea    bx,B             { BX a BOB címe                                 }
   mov    es,sega000       { $A000 az értéke a SEGA000 konstansnak         }
   mov    di,[bx+4]        { Előző helyzetének címe, eltároltuk, így       }
                           { nem kell ismét kiszámolni (gyorsabb)          }
   push   ds               { A DS regiszter kell majd a MOVSW-hez          }
   lds    si,[bx+10]       { A BOB alatti háttérrész mutatója (H)          }
   mov    dx,16            { 16 sor van                                    }
@1:mov    cx,8             { És 8 szónyi oszlop (16)                       }
   rep    movsw            { Egy sor visszaállítása                        }
   add    di,304           { A következő kezdőcíme 304-gyel nagyobb        }
   dec    dx               { Sorszámláló csökkentése,                      }
   jnz    @1               { ismétlés, amíg el nem éri a nullát            }
   pop    ds
   end;


procedure ShowBOB; assembler; asm
                           { BOB megjelenítése                             }

 { 2. Az új koordináták szerinti 16×16-os rész mentése }

   lea    bx,B             { BX a BOB ofszetcíme (szegmenscíme: DS)        }
   mov    ax,320           { Képcím kiszámolása a szokásos módon           }
   mul    word [bx+2]      { Sor címe=320×Y                                }
   add    ax,[bx]          { Képpont címe=sor címe+X                       }
   mov    [bx+4],ax        { Képcím elmentése (B.A szóba)                  }
   mov    si,ax            { A forrásindex a képcím, innen olvasunk        }
```

```
   les     di,[bx+10]      { A célcím a BOB rekord H mezője          }
   push    ds              { Ugyancsak a MOVSW miatt kell a DS       }
   mov     ds,sega000      { A képmemóriából történik az olvasás     }
   mov     dx,16           { 16 sor                                  }
@1:mov     cx,8            { 16 pixel soronként (16 bájt, 8 szó)     }
   rep     movsw           { Egy sor elmentése                       }
   add     si,304          { Forrásindex a következőre mutat         }
   dec     dx              { Egy sort átmásoltunk                    }
   jnz     @1              { A többi sor mentése (ha még van)        }
   pop     ds

 { 3. BOB megjelenítése }

   mov     es,sega000      { Most ES tartalmazza a képszegmenst      }
   mov     di,[bx+4]       { Szerencsére egyszer már kiszámoltuk a   }
                           { képcímet, így időt és helyet spórolunk  }
   push    ds              { Megint kell a DS, de most a LODSB miatt }
   lds     si,[bx+6]       { A BOB grafikus adatainak kezdőcíme      }
   mov     dx,16           { DX ismét a sorokat számlálja            }
@2:mov     cx,16           { CX pedig a pontokat (16 pixel, 16 bájt) }
@3:lodsb                   { DS:[SI] által címzett bájt AL-be töltése}
   cmp     al,0            { Ha ez nulla, nem írunk semmit a képer-  }
   jz      @4              { nyőre                                   }
   mov     es:[di],al      { Egyébként kirajzoljuk                   }
@4:inc     di              { DI a következő képbájtot címzi meg      }
   loop    @3              { Következő pixel megjelenítése           }
   add     di,304          { A következő sor címe                    }
   dec     dx              { Sorszámláló csökkentése                 }
   jnz     @2              { Amíg nagyobb nullánál                   }
   pop     ds
   end;


procedure Retrace; assembler; asm
                           { Várakozás egy vertikális visszafutásra, }
   mov     dx,03dah        { hogy a megjelenítés ne legyen darabos,  }
@1:in      al,dx           { villódzó. A vertikális visszafutás ide- }
   test    al,8            { je alatt a $3DA port 3. bitje 1. Ezt    }
   jz      @1              { kell vizsgálni                          }
   end;


begin
 with B do begin           { A B rekord mezőivel foglalkozunk        }
  getmem( S, 256);         { 16×16=256 bájt a Shape számára,         }
  getmem( H, 256);         { 16×16=256 bájt a háttérrészletnek       }
  assign( f,'rec.dat');    { A grafikus adatok a lemezen vannak, a   }
  reset( f, 1);            { REC.DAT fájlban, sorfolytonosan         }
  blockread( f, s^, 256);
  close( f);
  x:= 0; y:= 0;            { A BOB kiindulópontja a bal felső sarok  }
```

```
  asm
   mov     ax,13h
   int     10h             { MCGA üzemmód bekapcsolása              }
  end;
 randomize;
 for i:= 1 to 1000 do mem[$a000:random( 64000)]:=random( 256);
                         { A háttér tetszőleges beszínezése        }
 ShowBOB;                { BOB megjelenítése                       }
 repeat
  Retrace;               { A szép, egyenletes futás miatt várako-  }
  ClearBOB;              { zás a visszafutásra, majd ez idő alatt  }
  ShowBOB;               { letöröljük és újra kirakjuk a BOB-ot    }
                         { így nem villog, igaz, lassabb lesz      }
  inc( X, dx);           { BOB vízszintes mozgatása                }
  if (X=0) or (X=304) then dx:= -dx; { Kép szélén irányváltás      }
  inc( Y, dy);           { BOB függőleges mozgatása                }
  if (Y=0) or (Y=184) then dy:= -dy; { Kép szélén irányváltás      }
  until keypressed;
 readkey;                { A lenyomott billentyű kiolvasása        }
 asm
   mov     ax,03h
   int     10h             { Visszatérés a szöveges módhoz          }
  end;
 end;
end.
```

(Megjegyzés: a Turbo Pascal 7.0 beépített assemblyjében a rekordok, objektumok mezőire így is lehet hivatkozni: pl. [BX].BOB.X, azonban a könyvbeli programok többsége a 6.0-s verzióban készült, amiben még nincs meg ez a lehetőség.)

Az egérkurzor grafikus képernyőn való megjelenítése hasonló, csak az egér nyila már jobboldalt és lent „kicsúszhat". Vegyük át még egyszer, milyen előnyei és hátrányai vannak ennek az eljárásnak!

Előnyök:

- A háttér tetszőleges lehet, nem kell egyszínűnek lennie.
- A BOB átlátszó részeket is tartalmazhat, ahol a háttér látszik.
- A megjelenítés nagyon gyors, próbáljuk csak meg eltávolítani a *RetRace* utasítást a főciklusból! Ha ezután nem iktatunk be egy várakozó parancsot (*delay*), akkor az ábrázolás nagy sebessége miatt szinte semmi sem látható.
- Kevés memóriát igényel, mindössze 256 bájt hosszúságú az a dinamikus változó, ahol a BOB által takart háttérdarabot tároljuk.

Hátrányok:

- Csak egy BOB ábrázolását biztosítja, melynek méretei előre meghatározottak, és csak egyfázisú.
- Nem tudjuk megvalósítani a BOB „beúszását", vagyis nem helyezhetjük el úgy, hogy a képernyő széle keresztbevágja. Próbáljuk meg a főciklust a következő sorral helyettesíteni:

```
B.X:=312; ShowBOB;
```

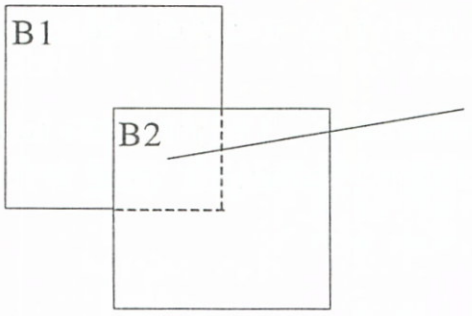
  Azt szeretnénk ezzel elérni, hogy a BOB csak félig látszódjon. Ezzel ellentétben a másik fele is megjelenik, a másik oldalon és egy sorral lejjebb. (A magyarázat, hogy az MCGA üzemmód pixelei sorfolytonos bájtokként jelennek meg a memóriában.)
- A háttér nem változhat, mert a háttér változtatásához a BOB-hoz tartozó puffert is át kellene írni.

Nos, van még mit javítanunk, folytassuk a fejlesztést!

2.3. Több BOB egyszerre

Egyszerre több grafikus objektum megjelenítésénél két probléma merül fel. Az egyik, amelyiknek megoldása egyszerűbb, az, hogy ha két BOB fedi egymást akkor melyik legyen felül, azaz melyik ne legyen takarásban. Ezt egyszerűen a megjelenítés sorrendjével határozhatjuk meg. Amelyik BOB-ot először rakjuk ki, az lesz alul, a következő pedig felül.

A másik nehézség a háttér aktuális részének tárolásában rejlik. Tegyük fel, hogy csak két BOB ábrázolásáról akarunk gondoskodni, melyek legyenek B1 és B2. B1 pozíciója legyen (X;Y), B2-é pedig (X+8;Y+8). Most nézzük meg, mi történik, ha a háttértárolós megjelenítést alkalmazzuk, ahogy azt az előző részben tettük. Az eljárás B1 kirajzolása előtt lemásolja az (X;Y) bal felső sarkú 16×16-os téglalapot, majd kirakja a B1-es azonosítójú BOB-ot. Ezután B2 következik. Itt viszont baj van, mert a tárolásra kerülő háttér-rész már tartalmazza a B1 jobb alsó részét. Tehát ez a B1-részlet is elmentődik, miszerint a B2 háttér pufferében nem a valóságos háttér található. Ez majd csak a BOB-ok letakarításánál jelent problémát, amit kétféleképpen oldhatunk meg.



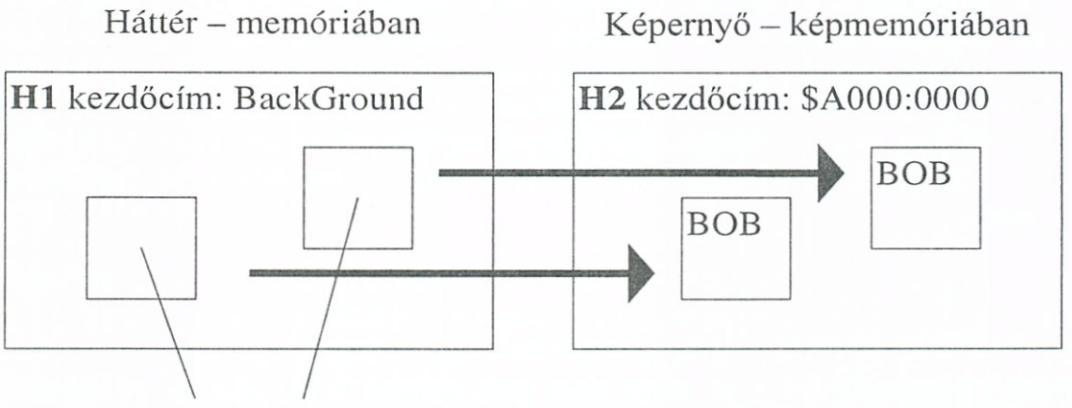
B2 háttérpuffere tartalmazza a B1 egy részletét – tehát nem az eredeti hátteret.

Az egyik megoldás az, hogy a BOB-okat fordított sorrendben tüntetjük el, mint ahogy kiraktuk őket. Az előző példánál maradva a képernyőn látszik B1 és B2, B2 takarja B1 egy részét. Ha most ugyanolyan sorrendben távolítanánk el őket, mint ahogy megjelenítettük, akkor a B1 jobb alsó sarka a képernyőn maradna. De ha fordított sorrendben töröljük őket, akkor lássuk, mi történik. B2 eltávolítása után ott marad B1, teljes egészében, sértetlenül; hiszen B2 háttérpuffere tartalmazta a B1-ből „kiharapott" részt. Végül a B1 eltüntetése már nem okoz gondot.

Ez a fordított letörlés jónak tűnik, mert aránylag kevés memóriát igényel (minden BOB után 256 bájtot, Shape-pel együtt 512-t). Hátránya, hogy mindegyik objektum más-más ideig látszik. A legelőször ábrázolt több időt tölt a képernyőn, mint az utoljára megjelenített, ez villogáshoz vezethet. És ha nem kettő, hanem mondjuk 256 BOB-ot szeretnénk láthatóvá tenni, akkor az összes háttérrészlet eltárolása már tetemes memóriát emészt fel (16×16×256 = 65536). Főleg, ha a BOB-ok nem 16×16-os méretűek, hanem ennél nagyobbak.

A másik megoldás a háttér **pufferelése**. A memóriában is tárolunk egy 320×200-as képet, amelynek tartalma megegyezik a képmemória tartalmával. Így tehát két ugyanolyan képünk van, csak az egyik, amelyik a normál memóriában van, nem látszik. Segítségükkel lényegesen leegyszerűsödik a BOB-ok letörlése, azaz a képernyőn az eredeti háttér visszaállítása. Egyszerűen csak át kell másolni BOB-onként egy-egy 16×16-os tartományt a megfelelő helyre, és már el is tűnt a BOB. A jobb érthetőség kedvéért, nézzük meg röviden, hogyan működik az új, továbbfejlesztett eljárás! Legyen H1 az a háttér, amelyik nem látszik, tartalma változatlan, H2 pedig a valódi kép, amelyik megjelenik, kezdőcíme $A000:0000, és amelyre a grafikus objektumok kerülnek.

1. Minden egyes BOB tárolt kezdőcímétől kezdve átmásolunk H1-ről H2-re egy 16×16-os tartományt.
2. BOB-ok kirakása H2-re, de mindegyik BOB-nál meg kell jegyezni annak a kezdőcímét (320×Y+X), a letörléshez a következő ciklusban.



Ezeket a háttérrészeket kell a képmemóriába másolni, hogy a BOB-ok eltűnjenek.

Itt nem kell első futtatásnál az első lépést átugrani, hiszen eleinte (BOB-ok megjelenítése előtt) H1 és H2 ugyanazt a képet tartalmazza, így – igaz, hogy hiába, de – nyugodtan másolhatunk egyikről a másikra. Viszont az elektronsugárra annál inkább figyelni kell! Sok BOB van, ezért egy-egy megjelenítési ciklusban a képernyőn viszonylag sokáig nem látszanak egyes BOB-ok. (Letörlésnél amelyiket először távolítjuk el, az látszik a legkevesebb ideig.) Ha a képernyő „megtisztítását" nem egy vertikális visszafutás alatt végezzük, az villogáshoz vezet! Amennyiben nagyon sok BOB-ot próbálunk megjeleníteni, akkor egy bizonyos határtól felfelé már úgyse lehet kiküszöbölni ezt a villogást, mert a képernyő „letakarítása" több ideig tart, mint egy vertikális visszafutás. Ez a határ többek között függ a gép sebességétől, a memóriarezidens programoktól, és a BOB-ok méretétől. Egy kb. 133 MHz-es gépen 40 BOB megjelenítése még nem okoz gondot, de 50 darab ábrázolása már villogással jár.

A példában a második, a kettős-háttér technikát valósítjuk meg. A program elején beállítjuk a szükséges adatokat, betöltjük a lemezről a Shape-eket stb. A főciklusban a BOB-ok mozgása átlós irányú, falról lepattanó, ezért a BOB rekordhoz még hozzá kellett venni a *DX* és *DY* elemeket, amelyek a vízszintes illetve függőleges irányt jelzik.

```
{showbob3.pas}

uses Crt;

type BOB = record          { Egy BOB nem grafikus adatai                  }
           X : word;       {  0  Vízszintes koordináta                    }
           Y : word;       {  2  Függőleges koordináta                    }
           A : word;       {  4  Előző helyzetének ofszetcíme             }
           G : pointer;    {  6  Grafikus adatok kezdőcíme                }
           DX: shortint;   {     Vízszintes irány (1: jobb, -1: bal)     }
           DY: shortint;   {     Függőleges irány (1: le  , -1: fel)     }
                           { (A 0,2,4,6 számok a mező távolságát je-      }
                           { lentik, bájtban, a SHOWBOB eljáráshoz)       }
           end;

const
 Bnum = 15;                { BOB-ok   száma: BNUM+1 (itt: 16)             }
 Snum =  3;                { Shape-ek száma: SNUM+1                       }
 Blen = sizeof( BOB);      { A BOB nem grafikus adatainak hossza          }

var
 B: array[0..Bnum] of BOB; { Az összes BOB nem grafikus adatai            }
 S: array[0..Snum] of pointer; { A grafikus adatok mutatói                }
 f: file;                  { Fájl változó a Shape-ek töltéséhez           }
 n: string[1];             { Ugyancsak a töltéshez kell                   }
 i: word;                  { Általános változó a FOR ciklusokhoz          }

 BackGround: pointer;      { A háttér kezdőcíme                           }


procedure ShowBOB; assembler; asm

 { Várakozás, amíg nincs vertikális visszafutás }

   mov    dx,03dah         { Nagy hiba lenne ezt a részt kihagyni,        }
@W:                        { az villogással járhat. Lassú gépeknél        }
   in     al,dx            { ez így is előfordulhat, ilyenkor nem ér      }
   test   al,8             { véget egy elektronsugár-visszatérés          }
   jz     @W               { alatt az eljárás  2. és 3. része             }

 { 1. A BOB-ok helyére az eredeti háttér-részletek visszaírása }

   mov    es,sega000       { A képernyő szegmenscíme                      }
   cld                     { D bit eloltása a sztring-műveletekhez        }
   mov    cx,bnum          { BNUM(=16) db BOB eltávolítása                }
@1:
   push   cx               { A REP utasításhoz is szükség lesz CX-re      }
   lea    bx,b             { A BOB-adatokat tároló tömb címe              }
   mov    ax,blen          { BX-hez annyit kell adni, hogy az az ak-      }
   mul    cx               { tuális (CX.) BOB-ra mutasson                 }
   add    bx,ax            { (BX:=BX+CX×BOB típus hossza)                 }
   mov    di,[bx+4]        { A BOB előző helyzetének címe ('A' mező)      }
   push   ds               { A MOVSW utasításhoz kell                     }
   lds    si,background    { A háttér kezdőcíme DS:[SI]-be                }
```

```
   add     si,di           { SI-hez a képcím hozzáadása                }
   mov     dx,16           { Most is a DX számlálja a sorokat          }
@2:
   mov     cx,8            { És CX az egy sorban lévő szavakat         }
   rep     movsw           { 16 pixel visszaállítása háttér színűre    }
   add     di,304          { Következő sor: DI:=DI+320-16              }
   add     si,304          { A háttér szélessége is 320 bájt           }
   dec     dx              { Eggyel kevesebb a visszaírandó sor        }
   jnz     @2              { Ismétlés, amíg DX el nem éri a nullát     }
   pop     ds
   pop     cx              { CX újra a BOB-okat számlálja              }
   loop    @1              { Többi BOB eltüntetése                     }

 { 2. BOB-ok kirakása }

   mov     cx,bnum         { CX most is BOB-számláló                   }
@3:
   push    cx              { Most is menteni kell, pontszámláló lesz   }
   lea     bx,b            { Megint úgy járunk el, mint a 2. részben   }
   mov     ax,blen         { (ahhoz, hogy a BX regiszter az aktuális   }
   mul     cx              { BOB-ra mutasson)                          }
   add     bx,ax
   mov     ax,320          { Képcím számítása a szokásos módon         }
   mul     word [bx+2]     { A 3. bájttól található az ordináta        }
   add     ax,[bx]         { Az 1. két bájt pedig az abszcissza        }
   mov     di,ax           { DI-ben előállt a képcím (320×Y+X),        }
   mov     [bx+4],ax       { amit elmentünk a BOB eltávolításához      }
   push    ds              { Most a LODSB-hez kell                     }
   lds     si,[bx+6]       { A forráscím a BOB típus 7. bájtjától     }
                           { kezdődik, ez a Shape-re mutat             }
   mov     dx,16           { Még mindig 16 sorból áll a BOB            }
@4:
   mov     cx,16           { És ugyanúgy 16 pontból egy sor            }
@5:
   lodsb                   { DS:[SI]-vel címzett bájt töltése, SI nő   }
   cmp     al,0            { Ha ez a bájt nulla, nem kell kiírni a     }
   jz      @6              { képernyőre, mert itt a BOB átlátszó       }
   mov     es:[di],al      { Egyébként igen                            }
@6:
   inc     di              { DI a következő képpontot címzi            }
   loop    @5              { Egy egész sor megjelenítve, ha CX=0       }
   add     di,304          { ES:[DI] a következő képsor (X-1). bájt-   }
                           { jának címe                                }
   dec     dx              { Sorszámláló csökken                       }
   jnz     @4              { Sorok kirakása, amíg számlálójuk (DX)>0   }
   pop     ds
   pop     cx
   loop    @3              { Többi BOB megjelenítése                   }
   end;
```

```
begin

 { 1. Háttér, képernyő beállítása }

  asm
    mov     ax,13h
    int     10h             { MCGA üzemmód bekapcsolása                }
   end;
  getmem( BackGround, 64000);
  randomize;
  for i:= 1 to 1000 do mem[$a000:random( 64000)]:= random( 256);
  move( ptr($A000, 0)^, BackGround^, 64000);
   { A háttér ugyanazt a képet tartalmazza, mint a képernyő }

 { 2. Grafikus adatok betöltése }

  for i:= 0 to Snum do begin
   getmem( S[i], 256);   { A Shape hossza 16x16=256 bájt             }
   str( i, n);
   assign( f, 'bob'+n+'.dat');
   reset( f, 1);
   seek( f, 7);          { Az első 7 bájt lényegtelen (pl. méret)    }
   blockread( f, s[i]^, 256);
   close( f);
   end;

 { 3. Nem grafikus adatok véletlenszerű beállítása }

  for i:= 0 to Bnum do with B[i] do begin
   G:= S[random( Snum+1)]; { Véletlenszerű minta kiválasztása        }
   X:= random( 303)+1;
   Y:= random( 183)+1;
   DX:= 2*random( 2)-1; { Értéke csak -1 vagy +1 lehet               }
   DY:= 2*random( 2)-1; { Értéke csak -1 vagy +1 lehet               }
   end;

 { 4. Főciklus, megjelenítés és mozgatás billentyűnyomásig }

  repeat
   ShowBOB;
   for i:= 0 to Bnum do with B[i] do begin
    inc( X, DX); if (X=0) or (X=304) then DX:=-DX;
    inc( Y, DY); if (Y=0) or (Y=184) then DY:=-DY;
                         { Falról lepattanó mozgás                    }
    end;
   until keypressed;
  readkey;

 { 5. Visszatérés a szöveges módhoz, vége }

  asm
    mov     ax,03h
    int     10h             { Visszatérés a szöveges módhoz            }
   end;
 end.
```

A lemezen lévő grafikus adatfájlok (pl. **BOB3.DAT**) csak a 8. bájtjuktól kezdve tartalmazzák a Shape adatait, sorfolytonosan, úgy ahogy a memóriában is tárolni szoktuk. Az első 7 bájt a Shape méreteit és fázisait határozzák meg, ami adott (16×16, 1 fázis. Ez okból ugrottunk a fájl 7. bájtjára a *seek* utasítással. (A Shape-ek a BOB-Editorral készültek, melynek fájlformátumáról részletesebben a 6. fejezet elején olvashatunk.)

Ezzel a módszerrel elvileg 5389 BOB-ot ábrázolhatunk. Viszont ha ezt a számot (azaz nála eggyel kevesebbet – 5388-at) beírjuk a *Bnum* konstans után, és úgy futtatjuk a programot, akkor az eredmény: a képernyő tele lesz fejecskékkel, szívekkel, rombuszokkal és almákkal (ezek a BOB-ok), és ütemesen elsötétül mindig, amikor egy BOB-letörlő rész fut. De már kevesebb BOB-nál is feltűnő zavaró hatásokat vehetünk észre, 100 BOB megjelenítésénél a kép felső részében azok nem látszódnak. Ez azért van, mert az elektronsugár már elkezdte frissíteni a képet, mielőtt az összes BOB eltüntetése és újbóli kirakása befejeződött volna, azaz a *ShowBOB* eljárás futási ideje több, mint egy vertikális visszafutásé.

A hátteret nem tudjuk egyszerűen változtatni. Ez majd akkor okoz problémát, ha olyan játékot szeretnénk írni, amelyik színtere nagyobb egy képernyőnél, így azt görgetni akarjuk. A BOB-ok még mindig állandó méretűek és fázisúak (16×16×1), és nem tudnak beúszni. Számuk korlátozott, géptől függ. Folytassuk tehát a fejlesztést!

2.4. Változtatható háttér, animált BOB-ok

Továbbfejlesztjük az eljárást, bővítjük egy lépéssel, hogy tetszőleges számú BOB ábrázolása se járjon villogással. A bővítés lényege, hogy a képalkotás a memóriában történik, és az ott elkészített, már kész képet rakjuk ki a képmemóriába. Ehhez az eljáráshoz szükség van tehát a memóriában még egy 320×200 bájt hosszú tartományra, a **munkaterületre**. Ez a munkaterület létesít kapcsolatot a grafikai adatok (háttér, Shape-ek) és a képmemória között. Használatának előnye: a BOB-ok letörlésének és újbóli kirakásának ideje megnőhet, ami legfeljebb a program futását lassítja, de villogást már nem okoz. Így több BOB-ot tudunk zavaró hatások nélkül ábrázolni, és a hátteret is könnyebben változtathatjuk. A munkaterület lényegének könnyebb megértéséhez nézzük meg eljárásunk lépéseit, mely megmutatja azt is, hogy a háttér egy egyszerű memóriába írással módosítható.

1. Az egész háttér, azaz 32000 szó átmásolása a munkaterületre a *movsw* utasítás segítségével.
2. BOB-ok megrajzolása a munkaterületen.
3. Munkaterület (32000 szó) másolása a képmemóriába, vagyis maga a megjelenítés.

A következő ábra szemléletesebbé teszi az ábrázolásnak ezt a módszerét, és segít megértetni a munkaterület használatát. A nyilak memóriából memóriába másolást, a nyilak feletti számok ezek sorrendjét jelölik.

Ha a hátteret egyszer csak gyökeresen megváltoztatjuk, az sem okoz problémát, úgy fogjuk tapasztalni, mintha az a BOB-ok mögött változott volna meg, tehát elértük célunkat. Viszont mi lenne akkor, ha az első lépésben nem az egész hátteret másolnánk, hanem csak azon részeit, amit a munkaterületen a BOB-ok takarnak? Nos, ez nyílván állandó háttérnél gyorsabb futást eredményez, de azt változtatni nehezebb lenne. Meg kéne változtatni magát a hátteret és a BOB-októl mentes munkaterületet is. Ez görgetésnél **2×64000** bájt mozgatását jelenti, ami semmiben sem gyorsítja a futást, hanem még lassítja is. Változatlan háttérnél beválik, ennek ellenére nem térünk ki külön erre a módszerre.

Miért van szükség arra, hogy a képet a munkaterületen állítsuk elő? Ez főleg a scrollozás, görgetés miatt van így. Ha egy játékban a hátteret görgetjük, akkor minden gördítési fázisban az egész képernyő tartalma megváltozik, hiszen minden egyes pixelnek más helyen kell megjelennie. Tehát mindig felül kell írni az egész képernyőt, azaz 64000 bájttal bizonyos műveleteket kell végrehajtanunk. Amíg a képernyőt „odébb toljuk", vagyis éppen felülírjuk a gördítés miatt, a BOB-ok nem látszanak, és ez a művelet hosszúsága (64k bájt) miatt erős villogáshoz vezet. Ha a memóriában, a munkaterületen végezzük mindezt,

az nem jár villogással, mert a képernyőn a kép korábbi állapota látszódik, egészen a 3. lépés végrehajtásáig.

Lehetőség nyílik az animációra is, a BOB-ok többfázisúak lehetnek. Ezt egy egyszerű szorzás beszúrásával érjük el, aminek eredményét hozzáadjuk a Shape-re mutató forrásindexhez, így az már a kívánt fázisra mutat. A példaprogramban a BOB-ok mérete még mindig 16×16, fázisaik száma viszont 4. Ebből adódóan egy-egy Shape helyfoglalása 16×16×4=1024 bájt.

```
{showbob4.pas}

uses Crt;

type BOB = record          { Bájt Tartalom                              }
         X,Y: word;        { 0;2  Koordináták                           }
         P:   word;        {  4   Aktuális fázis sorszáma (0-tól)       }
         Shp: pointer;     {  6   Grafikus adatok kezdőcíme             }
         V,F: shortint;    {      Vízsz. és függ. irányjelző bájtok     }
         end;

const
 Bnum = 15;                { Megjelenítendő BOB-ok száma -1             }
 Snum =  3;                { A BOB-okhoz tartozó minták száma -1        }
 Blen = sizeof( BOB);      { Egy BOB tömbelem hossza a memóriában       }
 Pspd = 10;                { Fázisváltási késleltetés                   }

 Pcur: word    = 0;        { Számláló a fáziskésleltetéshez             }
 Retr: boolean = true;     { Vertikális visszatérés figyelése           }

var
 B: array[0..Bnum] of BOB; { BOB nem grafikus adatok tömbje             }
 S: array[0..Snum] of pointer; { Grafikus adatok tömbje                 }
 BackGround, Workarea: pointer; { Háttér és munkaterület címe           }
 f: file;
 i: word;                  { Általános célú változók                    }
 n: string[1];

procedure ShowBOB; assembler; asm

 { 1. Háttér másolása a munkaterületre, 32000 szó mozgatása }

   cld                     { D bit eloltása, így az indexregiszte-      }
                           { rek nőnek a karakterlánc-műveleteknél      }
   mov    cx,32000         { 320×200 bájt = 32000 szó                   }
   push   ds               { A DS regisztert megváltoztatjuk            }
   les    di,workarea      { A célcím a munkaterület kezdőcíme          }
   lds    si,background    { A forráscím pedig a háttér kezdőcíme       }
   rep    movsw            { A beállítások után lehet másolni           }
   pop    ds               { Az adatszegmens visszaállítása             }
```

```
 { 2. BOB-ok felrakása a munkaterületre }

   mov     cx,bnum         { BNUM a BOB-ok száma                          }
@1:
   push    cx              { CX majd másra is kell                        }
   lea     bx,B            { A BOB-adatok tömbjének kezdőcíme BX-be       }
   mov     ax,blen         { BX-hez CX×BLEN-t adva megkapjuk az ak-       }
   mul     cx              { tuális (CX-edik) BOB nem grafikus ada-       }
   add .   bx,ax           { tainak kezdőcímét                            }
   les     di,workarea     { A célcím nem a képernyő, hanem a             }
                           { munkaterület kezdőcíme                       }
   mov     ax,320          { Kiszámítjuk annak a bájtnak a címét, a-      }
   mul     word [bx+2]     { mi a BOB bal felső sarka alatt lesz          }
   add     ax,[bx]         { A képlet jól ismert: cím=320×Y+X             }
   add     di,ax           { ES:[DI] már a megfelelő bájt címe            }
   push    ds              { DS lesz a forrásszegmens                     }
   mov     ax,256          { A Shape kezdőcíméhez P×256-ot kell adni,     }
   mul     word [bx+4]     { hogy az aktuális fázis (P) címe legyen       }
   lds     si,[bx+6]       { A forráscím a Shape címe, csak még hoz-      }
   add     si,ax           { záadunk P×256-ot. (Fázishossz: 16×16 b)      }
   mov     dx,16           { Ahogy megszoktuk, DX a sorok számlálója      }
@2:
   mov     cx,16           { CX pedig a soron belüli pixeleké             }
@3:
   lodsb                   { A Shape egy bájtjának betöltése              }
   cmp     al,0            { Ha ez nulla, nem kell a munkaterületre       }
   jz      @4              { írni semmit                                  }
   mov     es:[di],al      { Egy pixel kigyújtása a munkaterületen        }
@4:
   inc     di              { A következő bájt címe eggyel nagyobb         }
   loop    @3              { Egész sor kirakása ciklus alja               }
   add     di,304          { Következő sor első bájtjának címe            }
   dec     dx              { Eggyel kevesebb sort kell még kirakni        }
   jnz     @2              { 16 sort kell kirakni, ismétlés               }
   pop     ds
   pop     cx
   loop    @1              { Többi BOB megrajzolása a munkaterületre      }

 { 3. Munkaterület másolása a képernyőre, maga a megjelenítés }

 { 3.1. Ha a RETR logikai változó igaz, várakozás az elektronsu-
        gár függőleges irányú visszatérésére }

   cmp     retr,0
   jz      @6              { Ha a RETR hamis, nem kell várni              }
   mov     dx,3dah
@5:
   in      al,dx
   test    al,8
   jz      @5              { Csak visszafutás alatt folytatódhat          }
```

```
 { 3.2. Az 1. részhez hasonlóan 32000 szó mozgatása }

@6:
   mov     es,sega000     { MCGA képmemória szegmenscíme $A000      }
   xor     di,di          { A bal felső sarokból kezdünk (DI=0)     }
   mov     cx,32000       { 32000 szó, 64000=320x200 bájt           }
   push    ds
   lds     si,workarea    { A forráscím a munkaterület kezdőcíme    }
   rep     movsw          { Másolás                                 }
   pop     ds             { Soha ne felejtsük a DS regiszter erede- }
                          { ti értékét visszaállítani!              }
   end;


procedure Pixel( A: word; C: byte); assembler; asm
 { C színű pont kirakása a háttérre, A=320×Y+X (X;Y koordináták) }

   les     di,background { Most a háttérre rajzolunk               }
   add     di,A
   mov     al,C
   mov     es:[di],al
   end;


begin

 { 1. Beállítások, inicializálások, helyfoglalások }

  randomize;
  getmem( BackGround, 64000);
  getmem( Workarea,   64000);
  for i:= 0 to Snum do begin
   getmem( S[i], 1024); { A Shape hossza 4×16×16, 4 fázisból áll  }
   str( i, n);
   assign( f, 'anim'+n+'.dat');
   reset( f, 1);
   seek( f, 7);          { Az első 7 bájt számunkra nem fontos,    }
                         { leírásuk a következő részben (2.5.)     }
   blockread( f, s[i]^, 1024);
   close( f);
   end;
  for i:= 0 to Bnum do with B[i] do begin
   Shp:= S[random( Snum+1)];
   X:= random( 303)+1;
   Y:= random( 183)+1;
   P:= random( 4);
   V:= random( 3)-1;     { A BOB egyszerre két, egy vagy nulla i-  }
   F:= random( 3)-1;     { rányba mozoghat                         }
   end;
  asm
   mov     ax,0013h
   int     10h
   end;
  for i:= 0 to 63999 do Pixel( i, 0); { A háttér kezdetben fekete }
```

```
{ 2. Főciklus, megjelenítés és mozgatás, animáció }

 repeat
  ShowBOB;
  for i:= 0 to 3 do Pixel( random( 64000), random( 256));
                          { Háttér változtatása                    }
  for i:= 0 to Bnum do with B[i] do begin
   inc( X, V); if (X=0) or (X=304) then V:=-V;
   inc( Y, F); if (Y=0) or (Y=184) then F:=-F;
   end;
  inc( Pcur);
  if Pcur=Pspd then begin { Fáziskésleltetés, csak minden          }
                          { PSPD-edik ütemben van fázisváltás      }
   Pcur:= 0;
   for i:= 0 to Bnum do with B[i] do begin
    inc( P);
    if P=4 then P:= 0;
    end;
   end;
  until keypressed;
 readkey;

{ 3. Videomód visszaállítása, vége }

 asm
  mov     ax,3
  int     10h
  end;
end.
```

Itt abba is hagyhatnánk a fejlesztést, mert már majdnem minden kritériumot teljesítettük. Két dolgot kell még megvalósítanunk: a BOB-ok legyenek tetszőleges méretűek, és ha úgy helyezzük el, hogy a képernyő széle kettészelje őket, akkor csak az a részük látszódjék, ami a képernyőn rajta van. Sok játékban szükség van a beúszás lehetőségére, mert mondjuk egy autóversenyzős játékban igazán nem lenne szép, ha az autók csak úgy hirtelen megjelennének. A BOB-ok tetszőleges méretének szükségessége pedig egyértelmű, mert például egy lövöldözős programban egy 1×1-es lövedéknek 16×16-os méretű Shape-et adni egyszerűen pazarlás.

Ennek az ábrázolásmódnak két nagy problémája van: lassú és sok memóriát igényel. Lassúsága abból fakad, hogy minden megjelenítésnél legalább 2×64000 bájtot kell mozgatni, és ehhez még hozzájönnek a BOB-ok. Ez azt jelenti, hogy a futás csak egy 80 MHz-es géptől lesz egyenletes és szép, ha a visszatérésre várakozást alkalmazzuk (*RetRace*). Ennél lassabb gépeknél érdemes ennek a változónak hamis értéket adni, ezzel növeljük valamivel a sebességet. Másik gyorsító módszer az lehetne, hogy a BOB-ok letakarításához a

SHOWBOB3.PAS programban megismert módszert alkalmaznánk, ezzel azonban kizárnánk a háttér változtathatóságának lehetőségét, azaz például nem tudnánk azt görgetni, ami sok játékhoz nélkülözhetetlen. A programfutás sebességét a processzor sebessége mellett nagymértékben meghatározza a videokártya sebessége. Az újabb PCI buszos kártyák szerencsére már elég gyorsan tudják kezelni a videomemóriát, így a módszer jól használható. Régebbi ISA buszos kártyákon a futás elég lassú lehet.

A másik gond a nagy memóriaigény, amit a **SHOWBOB2.PAS** példaprogram megjelenítő eljárásában megvalósított háttérrész-tároló módszerrel tudnánk kiküszöbölni, ez azonban lehetetlenné tenné a háttér tetszőleges változtatását. A munkaterület kiiktatásának meg azért nem lenne értelme, mert ha adott számú BOB adott gépen villogásmentesen megjeleníthető, annak ábrázolása lassabb gépen már villogással járhat. És különben is még gyors gépen is nagy esély van a villódzásra, hiszen minden végrehajtáskor az egész képernyőt felülírjuk, azaz elég sokáig látszik csak a háttér.

Ezeket a problémákat tehát nem fogjuk és nem is kell megoldani. Ha az Olvasó olyan programot szeretne írni, amiben a háttér változatlan, nyugodtan kombinálhatja a megjelenítőket a számára optimális memóriafoglalás és sebesség szerint.

2.5. Teljes megjelenítés

Elérkeztünk a BOB megjelenítés utolsó részéhez, amely a fejezet elején megfogalmazott feltételek mindegyikét teljesíti. Egyszerre több transzparens, tetszőleges méretű és fázisú BOB jeleníthető meg, melyek képesek beúszni, és a háttér tetszőlegesen változtatható. A villogás szinte kizárva. A 2.4. fejezet példaprogramjához képest két új dolgot valósítunk meg, a BOB-ok tetszőleges méretűek és helyzetűek lehetnek.

Az ábrázolás fő lépései ugyanazok lesznek, mint az előző példában, csupán a BOB rekordot bővítjük néhány elemmel, és persze a *ShowBOB* eljárást is. A bármekkora méret megvalósítása az egyszerűbb feladat, annyi az egész, hogy az eljárásban két 16-os szám helyett (ennyi volt a szélesség és a magasság) egy-egy változót írunk. A BOB-ok méretének és fázisszámának kiválasztásánál csupán egyvalamire kell ügyelni, hogy a Shape mérete ne haladja meg a 64 Kbájtot.

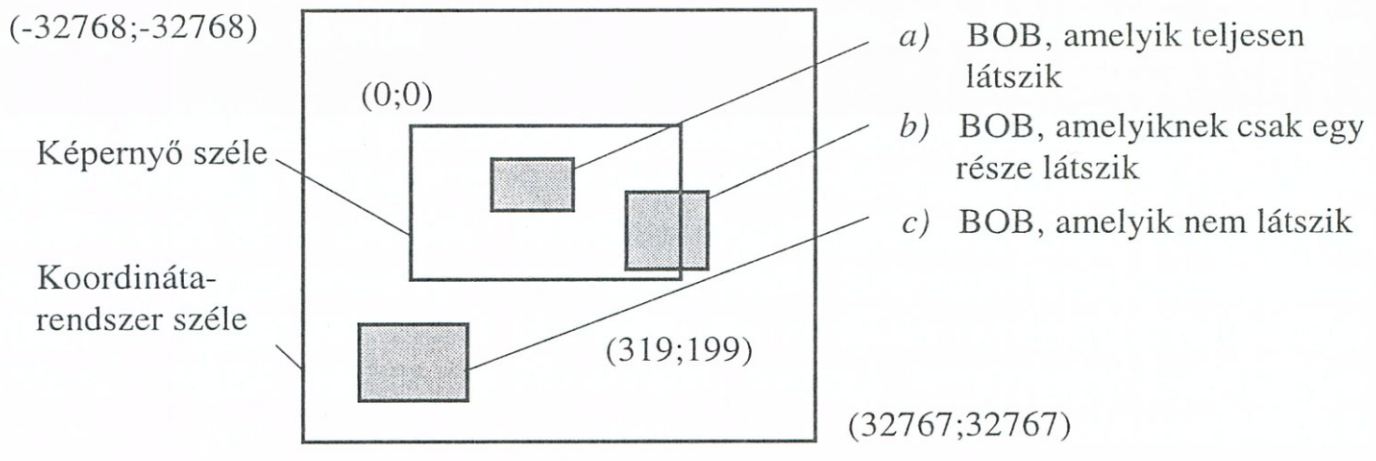
Az, hogy a BOB a képernyő szélén is megjelenhessen, már egy kicsit összetettebb. Meg kell vizsgálni, hogy melyik irányban lóg ki és mennyire. Az is megeshet, bár ritka, hogy a BOB egyszerre mind a négy irányban kilóg a képből. Ez például egy 322×202-es méretű objektumnál fordulhat elő, de ilyen nagy BOB-ot nincs értelme megjeleníteni. Ellenben az gyakori, hogy egyszerre két irányban nyúlik ki a képből, például ha a képernyő sarkán helyezkedik el.

Hogy a képernyő felső vagy bal oldali szélén is megjelenhessen, *integer* típusú koordinátákat fogunk használni. A képernyő bal felső sarka lesz a (0;0) pont, ettől balra és felfelé az adott irányhoz tartozó koordináta negatív. Ha egy BOB bal oldalról úszik be, akkor első koordinátája negatív és egyre nő. Így tehát van egy nagy koordináta-rendszerünk, melynek szélessége és magassága egyaránt 65536 pixel, és egy aránylag kicsi, 320×200-as téglalap alakú tartománya a képernyő.

Egy BOB, koordinátái alapján háromféle helyzetű lehet:

a) Teljes terjedelmében rajta van a képernyőn. Ilyenkor nem okoz nagy problémát a megrajzolása, a korábban kidolgozott módszert kell alkalmazni, egy kis változtatással persze, hogy tetszőleges méretű lehessen.
b) Csak egy része van rajta a képernyőn. Ez a legbonyolultabb, mert meg kell határozni, melyik téglalap alakú tartománya látszik, és ezt és csak ezt a részét kell megjeleníteni.
c) Nincs rajta a képernyőn. Ez a legegyszerűbb, mert nem kell semmit tenni. Ha nem csinálunk semmit, akkor a BOB nem látszik.

Nézzünk egy ábrát, amely szemlélteti a koordináta-rendszert, benne a képernyőt és a BOB-ok különböző elhelyezkedési lehetőségeit.



Eddig 304×184-es méretű koordináta-rendszereket alkalmaztunk (azért nem 320×200-asat, mert akkor a 16×16-os BOB-ok a szélére kerültek volna), ami csak a képernyőt foglalta magába. Most ezt kibővítettük, létrehozván ezzel egy 65536×65536 nagyságú rendszert, aminek része a képernyő. A bővítés miatt felmerült egy probléma, hogy egy BOB lehet olyan helyzetű, hogy a képernyő széle keresztül szelje, és ilyenkor megjelenítése egy fokkal összetettebb. Ezt a problémát igyekszünk az alábbiakban megoldani.

A megjelenítést végző eljárás BOB kirakó része (2.) a következő lépésekkel módosul:

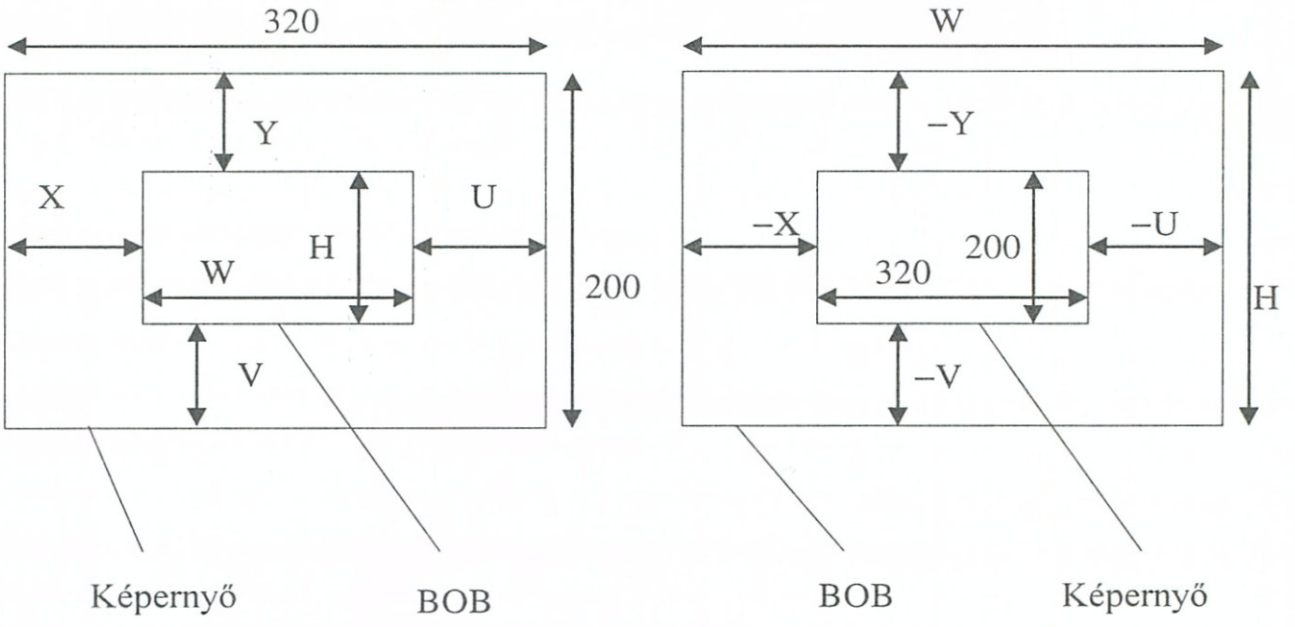
1. Ellenőrizzük, hogy a BOB rajta van-e a képernyőn, egyszerű összeadások és kivonások segítségével. Ha nincs rajta, nem kell ábrázolásával a továbbiakban törődni.
2. Meghatározzuk a BOB-on belül azt a tartományt, ami látható. Ha az egész BOB látható, akkor ennek a tartománynak a méretei megegyeznek a BOB méreteivel.
3. Kirajzoljuk a meghatározott téglalap alakú BOB-részletet a munkaterületre. A korábbiakkal ellentétben nem szabad folytonosan másolni a bájtokat, a részlet minden sora után valamennyivel növelni kell a számlálót, hiszen nem szabad az egészet megjeleníteni.

Azt, hogy egy BOB a képernyő szélén helyezkedik el, négy alapesetre bonthatjuk, majd ezeket kombinálva bármilyen elhelyezkedésű BOB-okat tudunk ábrázolni. Nézzük tehát ezt a négy esetet:

- Csak a képernyő bal széle szeli keresztül. Ekkor a BOB megrajzolandó része keskenyebb lesz, minden sor kiírása előtt a forrásindexet meg kell növelni annyival, amennyivel a BOB balra kilóg, vagyis az abszcisszájának (-1)-szeresével.
- A kép jobb széle vágja ketté. Ez esetben a megrajzolandó rész minden sora után a forrásindexet növelni kell annyival, amennyivel a BOB abszcisszájának és szélességének összege 320-nál nagyobb.
- Ha a képernyő teteje metszi el, akkor a kirakás előtt kell megnövelni a forrásindexet -Y×W-vel, ahol Y a BOB ordinátája, W pedig a szélessége.
- Amikor a kép alsó szélén helyezkedik el, egyszerűen csökkenteni kell a sorkirakás számlálóját a rajzolás megkezdése előtt. Nyilván minden esetben, amikor a BOB mérete csökkent, az aktuális irányhoz tartozó számlálót is csökkenteni kell, hiszen nem szabad többet kirakni belőle, mint amennyi látszik.

Ha egyszerre a képernyő több széle is keresztülmegy a BOB-on, akkor összegezni kell néhányat a négy alapeset közül. Pl. ha a bal felső sarkát tartalmazza, akkor mindenekelőtt növelni kell a forrásindexet annyival, amennyivel felfelé nyúlik ki, majd minden sorrészlet kirakása előtt annyival, amennyivel balra lóg ki a képernyőből, egyszóval kombináljuk az első és a harmadik alapesetet. Ha a kép jobb és bal oldalán is túlnyúlik, akkor az első két esetet kell egyszerre alkalmaznunk.

Az alábbiakban két ábrát láthatunk, két szélsőséges helyzetet arra nézve, hogy a BOB hány képszélt tartalmaz. A bal oldali rajzon egyet sem, tehát teljes egészében látható, a jobb oldali pedig mind a négy szélét tartalmazza a képernyőnek így a sarkait is. A betűk jelentése: **X, Y** – koordináták, **W** – szélesség, **H** – magasság, **U** = 320-X-W, **V** = 200-Y-H.



A megjelenítés továbbra is három lépésből áll, melyek megegyeznek az előző részben leírtakkal (1. háttér másolása a munkaterületre, 2. BOB-ok megjelenítése a munkaterületen, 3. munkaterület másolása a képmemóriába). A második lépés, a BOB-ok rajzolása más, amelyet a fenti ábrák segítségével írunk le. Először megnézzük, hogy az X, Y, U, V változók közül melyik negatív. Ha egyik sem, akkor az egész BOB látható, és nem kell tovább azzal törődni, hogy a BOB mely részét kell kirajzolni, mert az egész BOB-ot meg kell jeleníteni.

Mi a teendő, ha a négy érték valamelyike negatív?

X A BOB a képernyő bal szélén van. DI (a munkaterületre mutat) eltolása 320×Y, a sorok kirakása után 320-(W+X)-szel nő. Minden sor kirajzolása előtt az SI (Shape ofszetjére mutató) regisztert meg kell növelni -X-szel. És CX (oszlopszámláló) értéke minden sor kirakásakor W+X lesz. (X<0)

Y A kép tetején nyúlik túl. A DX (sorszámláló) regiszter H+Y, SI-t -Y×W-vel növeljük. DI kezdeti eltolása X, majd ez minden sor elkészítése után 320-W-vel nő.

U Jobbra lóg ki. DI-t eleinte 320×Y+X-szel növeljük, soronként 320-(W+U)-val nő. CX értéke W+U, SI minden sor befejezése után -U-val nő.

V Olyan, mint a normális megjelenítés, csak a DX regiszter H+V lesz.

Vegyük például a fenti jobb oldali ábrát. SI-hez eleinte hozzáadunk -Y×W-t, majd -X-et, és minden sor kirakása után -U+(-X)-szel növeljük. DI eltolása nulla, vagyis a munkaterület elejére mutat, egy sor kirakása után nem kell növelni semmivel, mert a BOB a képernyő mindkét függőleges szélén is túlnyúlik. CX 320, DX értéke pedig 200.

Ez már így meglehetősen száraz és érthetetlen, ezért nézzük magát a példaprogramot! A fenti ábrák sokat segíthetnek a kérdéses BOB-megrajzoló rész megértésében.

```
{showbob5.pas}

uses Crt;

type BOB = object
 {Eltolás}
 {00}  A : wordbool;     { Aktivitásjelző. Ha TRUE, akkor látható  }
 {02}  X : integer;      { X koordináta (abszcissza)               }
 {04}  Y : integer;      { Y koordináta (ordináta)                 }
 {06}  LX: word;         { Szélesség (lx=0-nál ez 1 képpont)       }
 {08}  LY: word;         { Magasság  (ly=0-nál ez 1 képpont)       }
 {10}  P : word;         { Fázisszámláló (p=0 : első fázis)        }
 {12}  DT: pointer;      { Shape helye a memóriában                }
 {16}  PL: word;         { Egy fázis helyfoglalása                 }
 {18}  W : word;         { Valóságos szélesség (width )            }
 {20}  H : word;         { Valóságos magasság  (height)            }
 {22}  LN: word;         { Shape helyfoglalása bájtban             }
 {26}  PN: word;         { Fázisszám (fázisok 0-val kezdődnek)     }
       procedure Load( FileName: string);
                         { Shape betöltése lemezről                }
       end;
```

```
const Bnum = 1;          { Most csak kettőt jelenítünk meg          }
      Blen = sizeof(BOB); { BOB típus hossza                         }
      Retrace: boolean = true; { Vertikális visszafutás-jelző        }


var B: array[0..Bnum] of BOB; { BOB-ok nemgrafikus adatai            }
    BackGround:pointer; { Háttér mutatója                            }
    WorkArea:  pointer; { Munkaterület mutatója                      }
    u,v: integer;       { A SHOWBOB eljárás használja őket           }
    i: word;            { Általános célú változó (FOR ciklushoz)     }

procedure BOB.Load;
 var f: file;
 begin
  assign( f, FileName);
  reset( f, 1);
  seek( f, 1);          { A fájl első bájtja nem használatos         }
  blockread( f, LX, 2); { A 2-3. bájt a Shape szélességét adja       }
  blockread( f, LY, 2); { A 4-5. pedig a magasságát                  }
  blockread( f, PN, 2); { Ezt követő szó a fázisok száma             }
  W:= LX+1; H:= LY+1;   { Az igazi méretek eggyel nagyobbak          }
  PL:= W*H;             { Egy fázis hossza = szélesség × magasság    }
  LN:= PL*(PN+1);       { Shape hossza = fázishossz × fázisszám      }
  getmem( DT, LN);      { Helyfoglalás a Shape-nek                   }
  blockread( f,DT^,LN); { Grafikus adatok betöltése                  }
  close( f);
  A:= true;             { Bekapcsoljuk, hogy látható legyen          }
  P:= 0;                { A fázismutatót az elsőre állítjuk          }
  end;

procedure ShowBOB; assembler; asm
                        { Ez a fő eljárás, a megjelenítést végzi     }
 { 1. Háttér másolása a munkaterületre }

   push   bp            { BP regisztert másra használjuk,            }
   mov    bp,ds         { az adatszegmenst tároljuk benne            }
   mov    cx,32000      { A háttér hossza 32000 szó (64000 bájt)     }
   cld                  { D jelzőbit törlése a REP utasításhoz       }
   les    di,workarea   { ES:[DI] mutat a munkaterületre             }
   lds    si,background { DS:[SI] pedig a háttérre                   }
   rep    movsw         { Háttér másolása a munkaterületre           }
   mov    ds,bp         { DS újra az eredeti adatszegmens            }

 { 2. BOB-ok rajzolása a munkaterületre }

   mov    cx,bnum       { BNUM BOB kirakásáról kell gondoskodni      }
@putbob;                { Egy BOB kirakása ciklus kezdete            }
   push   cx            { CX-et majd másra használjuk (oszlopszám-}
                        { láló lesz a sorok megrajzolásánál)         }
   lea    bx,B          { BX:= B tömb ofszetcíme                     }
   mov    ax,blen
   mul    cx            { BX-hez hozzáadunk annyit, hogy az éppen    }
   add    bx,ax         { aktuális (CX.) BOB-ra mutasson             }
```

```
{ 2.1. Ellenőrizzük, hogy a BOB, koordinátái alapján, látható-e }

  cmp     word [bx],0     { Az aktivitást jelző szó ellenőrzése     }
  jz      @nextbob        { Ha hamis, akkor nem szabad kirajzolni   }
  cmp     word [bx+2],320 { Ha X>=320, akkor nem látható, ugrás a }
  jge     @nextbob        { következő BOB-ra                        }
  cmp     word [bx+4],200 { Ha Y>=200, akkor a képernyő alja a-   }
  jge     @nextbob        { latt van, ezért nem kell megjeleníteni  }
  mov     ax,[bx+6]       { Amikor balra helyezkedik el a képernyő- }
  add     ax,[bx+2]       { től, LX+X<=0                            }
  jnge    @nextbob        { Ekkor se kell ábrázolni                 }
  mov     u,ax            { LX+X-et tároljuk, mert később még szük- }
  mov     ax,[bx+8]       { ség lesz rá, nem kell újra kiszámolni   }
  add     ax,[bx+4]
  jnge    @nextbob        { Ha LY+Y<=0, akkor ugrás a következőre   }
  mov     v,ax            { LY+Y-t is megjegyezzük                  }
  neg     u
  add     u,319           { U=320-X-W (=319-X-LX)                   }
  neg     v
  add     v,199           { V=200-Y-H (=199-Y-LY)                   }

{ 2.2. Meghatározzuk a BOB látható részét }

  les     si,[bx+12]      { ES:[SI] mutat most a grafikus adatokra  }
  mov     ax,[bx+16]      { [BX+16] = egy fázis hossza (PL)         }
  mul     word [bx+10]    { [BX+10] az aktuális fázis (P)           }
  add     si,ax           { SI a Shape aktuális fázisára mutat      }
  push    es              { ES-t tároljuk, később ez lesz a DS      }
  les     di,workarea     { ES:[DI] a munkaterület mutatója         }
  mov     ax,320          { Kiszámoljuk a képcímet, mint ha normáli-}
  mul     word [bx+4]     { san ábrázolnánk (CIM=320×Y+X)           }
  add     ax,[bx+2]       { Később majd ezt fogjuk módosítani       }
  add     di,ax           { DI=320×Y+X                              }

{ 2.2.1. Ha a kép felső szélén helyezkedik el }
  mov     dx,[bx+20]      { DX a sorszámláló az ábrázolásnál        }
  mov     ax,320          { Egy sor kirajzolása után DI-t @DIPLUS-  }
  sub     ax,[bx+18]      { szal kell növelni, amelynek értéke      }
  mov     word [@diplus],ax { eleinte: 320-W                        }
  mov     word [@siplus],0 { @SIPLUS: amit SI-hez kell adni egy     }
                          { sor kirakása után. Eleinte ez nulla     }
  cmp     word [bx+4],0   { [BX+4] a BOB ordinátája (Y)             }
  jge     @left           { Ha ez nem negatív, lépünk tovább        }
  add     dx,[bx+4]       { Összesen H+Y sort kell kirakni          }
  mov     di,[bx+2]       { DI értéke X, közvetlenül a képernyő     }
                          { tetejétől kezdve rajzoljuk a pontokat   }
  add     di,word [workarea] { DI-hez még hozzáadjuk a munkaterü- }
  mov     ax,[bx+4]       { let ofszetcímét }
  neg     ax
  push    dx
  mul     word [bx+18]    { Csak a -Y. sortól kezdve kell megjele-  }
  pop     dx
  add     si,ax           { níteni, ezért SI-hez hozzáadunk -Y×W-t  }
```

```
 { 2.2.2. Bal oldal }
@left:
   mov     cx,[bx+18]      { CX: hány pontot kell egy sorban kirakni }
   cmp     word [bx+2],0 { [BX+2] a BOB abszcisszája (X)             }
   jge     @right          { Ha nem negatív, a jobb oldalt vizsgálja }
   sub     di,[bx+2]       { Közvetlenül a bal szélén kezdünk        }
   mov     ax,[bx+2]       { @DIPLUS értékét X-szel csökkentjük,     }
   sub     word [@diplus],ax { vagyis @DIPLUS nő                     }
   neg     ax
   add     word [@siplus],ax { @SIPLUS nő (-X-szel)                  }
   add     si,ax           { A -X. oszloptól kezdjük a sorokat       }
   sub     cx,ax           { Oszlopszámláló csökkentése              }

 { 2.2.3. Jobb oldal }
@right:
   cmp     u,0             { U=320-X-W, ha nem negatív, nincs rajta  }
   jge     @down           { a képernyő jobb szélén, tehát ugrás     }
   add     cx,u            { CX csökken, kevesebb oszlop             }
   mov     ax,u
   sub     word [@siplus],ax { @SIPLUS nő, egy sor kirakása után     }
                           { SI-hez -U-val többet kell adni          }
   sub     word [@diplus],ax { @DIPLUS is nő                         }

 { 2.2.4. Vizsgáljuk, hogy a BOB az alsó szélen van-e }
@down:
   cmp     v,0             { V=200-Y-H, ha nulla vagy pozitív, nem   }
   jge     @put            { nyúlik túl a képernyőn lefele           }
   add     dx,v            { Csak a sorszámlálót kell csökkenteni    }

 { 2.3. A BOB kirajzolása a munkaterületre }

@put:
   mov     word [@cxsave],cx { CX-et elmentjük, ne kelljen minden    }
                           { sor kirakásához újra kiszámolni         }
   pop     ds              { A veremben legfelül a Shape szegmense   }
                           { volt, így DS:[SI] a forráscím           }
@putlines:                 { Sorok rajzolása ciklus kezdete          }
   mov     cx,word [@cxsave] { CX a kirakandó oszlopok száma         }
@putline:                  { Egy sor kirakása ciklus kezdete         }
   lodsb                   { AL:=byte ptr DS:[SI], SI:= SI+1         }
   cmp     al,0            { Csak akkor kell kirajzolni ezt a pontot,}
   jz      @notput         { ha nem nulla                            }
   mov     es:[di],al
@notput:
   inc     di
   loop    @putline        { Egy sor kirakása ciklus vége, ha CX=0   }
   add     di,word [@diplus]
   add     si,word [@siplus]
   dec     dx              { Sorszámláló csökkentése                 }
   jnz     @putlines       { Sorok rajzolása ciklus vége, ha DX=0    }
   mov     ds,bp           { DS újra az eredeti adatszegmens         }
@nextbob:                  { Ide ugrik, ha a BOB nem látszik         }
   pop     cx              { CX megint a BOB-okat számolja           }
   dec     cx              { Egyet már kiraktunk                     }
```

```
    cmp     cx,-1
    jnz     @putbob         { De még a többit is ábrázolni kell      }

  { 3. Munkaterület bemásolása a grafikus tárba (megjelenítés) }

  { 3.1. Várakozás egy vertikális visszafutásra }
    cmp     retrace,0       { Ha a RETRACE értéke FALSE,             }
    jz      @show           { egyből a megjelenítés jön, nem várunk  }
    mov     dx,3dah         { A $3DA porton keresztül figyeljük a    }
@wait:                      { vertikális visszafutás bekövetkeztét   }
    in      al,dx
    test    al,8            { 3. bit jelzi a visszatérést            }
    jz      @wait           { Ha nulla, még várni kell               }

  { 3.2. Munkaterület másolása }
@show:
    mov     es,sega000      { $A000 a grafikus tár kezdőcíme         }
    xor     di,di
    mov     cx,32000        { Ugyancsak 32000 szót másolunk          }
    lds     si,workarea     { A munkaterületről másolunk             }
    rep     movsw           { Megjelenítés                           }
    mov     ds,bp
    pop     bp
    jmp     @exit

@diplus: dw 0               { Ez a három változó a kódszegmensben van,}
@siplus: dw 0               { azért nem az adatszegmensben, mert arra }
@cxsave: dw 0               { szükség van a BOB rajzolásánál         }

@exit:
    end;

begin

  asm
    mov     ax,13h
    int     10h             { MCGA üzemmód bekapcsolása              }
   end;
  with B[0] do begin
   Load('plane.bob');       { Az egyik BOB egy repülő                }
   x:=-w;                   { Balról úszik be (éppen nem látható)    }
   y:= 100- h div 2;        { Pont középen megy                      }
   end;
  with B[1] do begin
   Load('rocket.bob');      { A másik meg egy rakéta                 }
   x:= 160- w div 2;
   y:= 200;                 { Lentről jön felfele                    }
   end;
  getmem( background, 64000); { Helyfoglalás a háttérnek és a       }
  getmem( workarea,   64000); { munkaterületnek                     }
  for i:= 0 to 63999 do  { Háttér letakarítása                      }
   mem[seg( background^):ofs( background^)+i]:= 0;
  randomize;
```

```
for i:= 1 to 1000 do    { 1000 színes pixel rajzolása a háttérre  }
 mem[seg( background^):ofs( background^)+random( 64000)]:= i;

repeat                  { Főciklus, a megjelenítést végzi         }
 ShowBOB;
 with B[0] do begin     { Az 1. BOB mozgatása                     }
  inc( x, 4);           { Balról jobbra, amíg ki nem úszik a kép- }
  if x>=320 then begin  { ből, majd az elejére ugrik, csak másik  }
   x:=-w;   y:= random( 200+h)-h; { sorban                        }
   end;
  end;
 with B[1] do begin     { A 2. BOB mozgatása, hasonlóan, csak fel }
  dec( y, 3);
  if y<=-h  then begin
   y:= 200; x:= random( 320+w)-w;
   end;
  if random( 3)=0 then p:= random( 3); { Kb. minden 3.-ra váltás }
  end;
 until keypressed; {*} { Billentyűnyomásig megy az animáció      }
readkey;

asm
  mov    ax,03h
  int    10h            { MCGA üzemmód kikapcsolása              }
 end;
end.
```

A program elején létrehozzuk a BOB típust, ami egy kicsit eltér az eddig megszokottaktól. Ugyanis nem rekord, hanem egy objektum, így egy egységbe tudjuk foglalni az adatokat és a *Load* utasítást. A különböző mezők leírása a mögöttük található {} részben tekinthetőek meg és a 8.23.11. fejezetben, hiszen a *Game* egység *BOB* típusa az itt szereplőnek egy bővített változata.

A sebesség nemcsak a gép, hanem a videokártya függvénye is. PCI buszos kártyákon általában jól megy, de ISA buszos kártyák esetében bizony elég lassú a megjelenítés. Ez főképp a két busz sebessége miatt van, de meghatározó a videokártya chipset-je és az alkalmazott memória sebessége is! Szerencsére a PCI buszos kártyák az elterjedtebbek, és az idő előrehaladtával egyre gyorsabb hardverelemek kerülnek forgalomba. A ma legolcsóbban kapható kártyák is olyan gyorsak, hogy ez a módszer jól alkalmazható rajtuk. De természetesen egy elavult, lassú kártyán a futás is lassú, darabos lesz.

A konstansok közül a *RetRace* szorul részletesebb magyarázatra. Ez egy kezdőértékkel rendelkező logikai változó, ha értéke igaz, a megjelenítés 3. lépése előtt várakozik egy vertikális visszafutás bekövetkeztére. Ez lassabb, de szebb

futást eredményez. Ha gépünk sebessége 80 MHz alatti, vagy túl sok a rezidens program, adjunk ennek a változónak hamis értéket.

A változók deklarálását követi a *BOB* típus *Load* eljárásának meghatározása. Ez egy fájl alapján állítja be a BOB grafikus és nem grafikus adatait. A Shape-et a BOB-Editorral (6. fejezet) megszerkesztett fájlban tároljuk, melynek felépítése a következő:

Cím	Hossz (bájtban)	Jelentés
0	1	Fejlesztésre fenntartva, értéke 1. (Verziószám)
1	2	Shape szélessége-1 (LX). A valódi szélesség ennél eggyel nagyobb.
3	2	Magasság-1 (LY).
5	2	Fázisok száma-1 (PN).
7	(LX+1)×(LY+1)×(PN+1)	Grafikus adatok

Ezután jön a megjelenítő eljárás (*ShowBOB*). Lényege megegyezik a 4. példában megfogalmazottéval, tehát továbbra is három fő lépésből áll, háttér és BOB-ok másolása a munkaterületre, munkaterület megjelenítése (képmemóriába másolása). És végül a főprogramban elvégezzük a szükséges beállításokat, ezután belépünk a megjelenítő és mozgató ciklusba, amiből billentyűnyomással lehet kilépni.

A *ShowBOB* eljárás végén, a kódszegmensben található három változó: *@diplus, @siplus, @cxsave*. Ezekre azért van szükség, hogy a megjelenítő cikluson belül ne kelljen mindig visszatölteni az eredeti adatszegmens-regisztert, mert az sok időt vesz igénybe. Viszont emiatt a programot nem lehet védett (*protected)* üzemmódban futtatni, mert ilyenkor a kódszegmenset nem lehet írni.

Ezzel befejeztük a fejlesztés, elértük végső célunkat, ez a megjelenítő a fejezet elején megfogalmazottak mindegyikét teljesíti. Mint mindennek, természetesen vannak hátrányai is, legfőbb hátulütője talán a lassúság, a nagy memóriaigény és a védett mód kizárása. Viszont mindez kell a gördíthető háttér megvalósításához.

A példaprogramoknak nemcsak az a céljuk, hogy lépésenként közelítsük meg a végső eljárást, hanem az is, hogy az Olvasót lehetőleg minél több megjelenítési módszerrel ismertesse meg. Ezért ha egy játékban nincs szükség görgetésre, nyugodtan kombinálhatjuk a 3. és az 5. példaprogramban található BOB-megjelenítő (*ShowBOB*) eljárást. Ilyenkor a háttér állandó, tehát nem kell minden frissítésnél az egészet lemásolni, hanem elég csak a BOB-ok által takart részeit. Ezzel lényegesen megnövelhetjük programunk sebességét, viszont kizárjuk a tetszőlegesen változtatható háttér lehetőségét. A később ismertetésre kerülő *Game* egység megjelenítő eljárásai (*MakeScr* – 8.12., *ShowScr* – 8.20) is ezt, a **SHOWBOB5.PAS** példaprogramban megismert megjelenítési módszert alkalmazzák. (Annak egy kicsit bővített formáját.)

3. Billentyűzet és egér

Ez a két eszköz nem tartozik ugyan szorosan a játékok grafikai részének megvalósításához, viszont nagyon fontos foglalkozni velük, főként a billentyűzettel. Ez ugyanis a leginkább használt bemeneti egység, szinte az összes játék, sőt az összes program kezelhető csak billentyűzettel. A játékok irányítása általában a következő négy berendezés valamelyikével történik, melyeket előfordulásuk gyakoriságának sorrendjében mutatunk be, néhány szóval:

1. Billentyűzet. Minden számítógép elengedhetetlen tartozéka. Tehát a leggyakoribb, ezért külön kitérünk rá.
2. Egér. A programok többségénél nélkülözhető az egér, gyakran ugyanazt a hatást sokkal gyorsabban el tudjuk érni, mint a billentyűkkel. De a felhasználói programok nagy többségénél, a kezelés egyszerűsége és kényelmessége érdekében nélkülözhetetlen. Ezért majdnem minden gép mellett megtalálható.
3. Botkormány. Kifejezetten játékok irányításához alkalmazható.
4. Gamepad. A videojátékok elengedhetetlen tartozéka, de egyre inkább tért hódít a számítógépes változata is.

A négy lehetőség közül csak a két leggyakoribbal, a billentyűzettel és az egérrel foglalkozunk. Ezek sokkal sűrűbben fordulnak elő, mint a másik kettő, emiatt érdemes egy kicsit részletesebben is kitérni rájuk.

3.1. A billentyűzet működése

Ha megnyomunk vagy felengedünk egy billentyűt, aktiválódik a **$09-es megszakítás**. A $09 sorszámú megszakítás beolvassa az őt kiváltó billentyű adatait (*SCAN* kód, állapot) a **$60**-as portról, majd előállítja annak az ASCII kódját. Gépelésre, egyszerű adatbevitelre kiválóan alkalmas, hiszen például automatikusan kiszámolja az ASCII kódot a SCAN kódból. Játékok irányítására viszont alkalmatlan. Ha egy BOB koordinátáit a normális BIOS billentyűzet-megszakítás alkalmazásával akarjuk megváltoztatni, akkor a BOB mozgása meglehetősen darabos lesz, és az első lépés után vár egy ideig.

Az a célunk, hogy egy billentyű lenyomását folyamatosan tudjuk érzékelni. Ez megvalósíthatatlan a bekapcsolás utáni aktív BIOS megszakítással, több okból

is. Folyamatosan nem figyelhető egy billentyű, még ha a következő sort be is gépeljük a DOS prompt után:

```
MODE CON RATE=32 DELAY=1
```

Ha ezzel vezérelnénk egy mozgást, az darabos lenne. Másik probléma, hogy ha minden billentyűnyomás után nagy memóriatartományokat mozgatunk (például a → megnyomására jobbra görgetjük a hátteret), akkor a hangszóró sípol, programunk futásidejében nagy kihagyások keletkeznek. (A sípolás oka, hogy a billentyűpuffer betelik, és a mi programunk nem tudja kellő gyorsasággal kiolvasni az adatokat.)

Tehát például a *Crt* egység billentyűvezérlő eljárásaival (*readkey*, *keypressed*), amik az eredeti megszakítást használják, nem érdemes egy játék vezérlését végeztetni, mert az irányított elem (BOB, háttér) mozgása meglehetősen darabos, egyenetlen lesz. Akár írhatunk egy rövidke kis példaprogramot is, hogy erről meggyőződhessünk. A **SHOWBOB5.PAS** példaprogram {*}-gal jelzett sorát (hátulról az 5. sor) írjuk át a következőre:

```
until readkey=#27;
```

Most a BOB-ok csak akkor mozognak, ha folyamatosan nyomva tartunk egy billentyűt, ESC-re pedig befejeződik a program futása. Nos, szemmel látható, hogy a BIOS megszakítása erre a célra alkalmatlan.

Mit kell akkor tenni? Egyszerű, át kell írni a $09 megszakítást, vagyis a megszakításvektor módosításával utasítani kell a processzort, hogy billentyű állapotának változtatásakor ne az eredeti BIOS rutint hajtsa végre, hanem a saját assembly utasításainkat.

Ez a saját rutin nem túl bonyolult, elmenti a verembe a használt regisztereket, beolvas a $60 portról egy bájtot (0-6. bit: SCAN kód, 7. bit: állapot), tárolja ezt a két adatot, nyugtázza a megszakítási áramköröket, majd a regiszterek visszatöltése után egy *iret* utasítással zárul. A billentyű adatait a következő módon tároljuk. A program elején létrehozunk egy 128 elemű logikai tömböt, majd igaz vagy hamis értéket adunk annak a tagjának, amelyiknek sorszáma megegyezik a lenyomott vagy felengedett billentyű SCAN kódjával. Ebből látszik az is, hogy a SCAN kódok maximális értéke 127, ami jóval több, mint a billentyűk száma (86-102). Minden billentyűhöz egy SCAN kód tartozik.
Nézzük tehát a példaprogramot, mely folyamatosan kiírja a lenyomott billentyű(k) SCAN kódját.

```
{keyb1.pas}

uses DOS;

var  KEY: array [0..127] of boolean;
                          { Ez a tömb tárolja a billentyűk adatait  }
     OLD: procedure;      { A régi megszakításvektor                }
     i  : byte;           { A FOR ... TO ... DO ... ciklushoz       }

procedure NewIRQ; assembler; asm
   push   ds              { Azokat a regisztereket, amelyek a meg-  }
   push   ax              { szakítás végrehajtása közben módosulnak,}
   push   bx              { a veremben tároljuk                     }
   push   cx
   xor    cl,cl           { Néhány regiszter kezdeti értéke         }
   mov    bh,cl
   mov    ax,seg key      { DS a KEY tömb szegmense                 }
   mov    ds,ax
   in     al,60h          { AL-be beolvassuk a billentyű SCAN-kódot }
   mov    bl,al           { BL-be is bevisszük a kódot              }
   shl    al,1            { C jelzőbit (FLAG) = 7. bit              }
   cmc                    { Ezt negáljuk, így ha 0, felengedtük az  }
                          { adott billentyűt, ha 1, akkor lenyomtuk }
   adc    cl,00           { CL most 0 vagy 1 lehet (FALSE, TRUE)    }
   and    bl,127          { Az alsó 7 bit adja a valódi SCAN-kódot  }
   mov    [offset key+bx],cl { A megszakítást kiváltó billentyű-    }
                          { höz tartozó logikai változó beállítása  }
   in     al,61H          { A megszakítás csatorna visszaállítása   }
   mov    ah,al
   or     al,80H
   out    61H,al          { Jelzés a billentyűzetnek                }
   mov    al,ah
   nop                    { Kevés várakozás                         }
   nop                    { (soros adatkiküldés sebessége miatt)    }
   nop
   out    61H,al
   cli
   mov    al,20H          { "Megszakítás vége" jelzés               }
   out    20H,al
   sti
   pop    cx              { Regiszterek visszaolvasása a veremből   }
   pop    bx
   pop    ax
   pop    ds
   iret                   { Vége a megszakításnak                   }
   end;

begin
 getintvec( $09, @OLD); { A régi megszakításvektort tároljuk       }
 setintvec( $09, @NewIRQ); { A $09 megszakítás ezentúl a NEWIRQ    }
                        { eljárást hívja meg                       }
 fillchar(key,sizeof(key),0); { KEY tömb nullázása (FALSE)         }
```

```
  repeat
  for i:= 0 to 127 do if key[i] then writeln(i);
                          { A lenyomott billentyű kódja a képernyőre}
  until key[1];           { ESC megnyomásáig                         }
 setintvec( $09, @OLD);
 end.
```

Mint ahogy az a program fő részéből is látszik, egy billentyű figyeléséhez egyszerűen csak figyelni kell a *KEY* tömb hozzá tartozó elemét. Ha például lenyomjuk az ESC billentyűt, amelynek a SCAN-kódja 1, akkor a *KEY[1]* értéke igaz lesz, ami addig igaz, amíg az ESC-t lenyomott állapotban tartjuk.

Előfordulhat, hogy a programból való kilépés után a billentyűzet nem működik helyesen. Úgy viselkedik ilyenkor, mintha valamelyik *Ctrl* billentyű le lenne nyomva. A hiba megszüntetéséhez nyomjuk meg mindkét *Ctrl* billentyűt egyszerre! Ezt a hibát sajnos nem tudjuk kiküszöbölni.

Adódhat egy másik hiba is. Néhány billentyű lenyomásakor (pl. szürke nyilak) a program azt mutatja, hogy a $2A(42) kódú bal oldali *Shift*-et is megnyomtuk volna, holott nem is tettük ezt. Ez ellen csak annyit tehetünk, hogy nem használjuk egyszerre a bal *Shift* gombot és a nyílbillentyűket. (Egyébként a *NumLock* leütésével megszűnik ez a probléma, majd újbóli megnyomásával újra megjelenik.)

Hasonlítsuk össze az előző programot a következővel, ami a lenyomott billentyű ASCII kódját írja ki, a BIOS megszakítás használatával. Figyeljük meg, az előző mennyire gyorsabb és egyenletesebb!

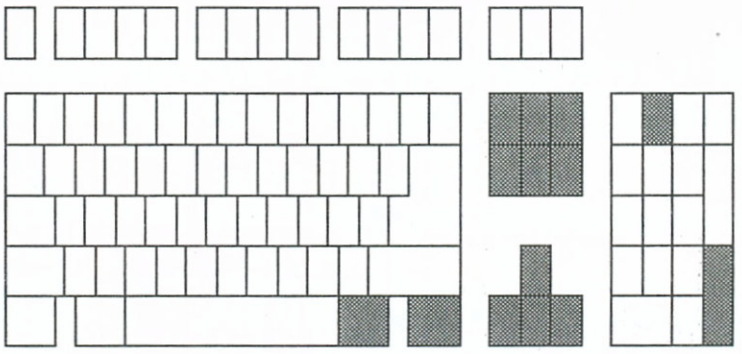
```
{keyb2.pas}

uses Crt;                 { Most a CRT egységet, vagyis a BIOS      }
                          { billentyűmegszakítását használjuk       }
var C: char;              { A lenyomott billentyű ASCII kódja lesz  }

begin
 repeat
  C:= readkey;
  writeln( ord( C));      { A C karakter ASCII kódját kiírjuk       }
  until C=#27;            { ESC-re kilépés a programból             }
 end.
```

A módosított megszakításnak is van egy hátránya, csak az XT-khez gyártott billentyűzeteknél működik tökéletesen, 101 gombos klaviatúrán egyes billentyűpárok között már nem tudunk különbséget tenni. Például mindkét *Ctrl* kódja $1D, pedig gyakran szükség lenne a megkülönböztetésükre.

Nevezzük el bővített billentyűknek azokat, amelyek a 101 gombos tasztatúrán megtalálhatóak, a 86 gomboson viszont nem. Minden ilyen bővített billentyű lenyomásakor vagy felengedésekor a SCAN kód előtt a $60 perifériacímről egy **$E0** értékű bájt érkezik, ezt használjuk majd ki arra, hogy az azonos funkciójú billentyűket megkülönböztessük. A következő billentyűk állapotváltoztatásakor jön létre ez az $E0 érték:

- Szürke nyílbillentyűk,
- Szürke *Ins, Del, Home, End, Page Up* és *Down,*
- Jobb oldali *Alt* és *Ctrl,*
- A numerikus billentyűzet *Enter* és / gombja.

Másként szólva ezek a billentyűk újak a 86 gombos XT billentyűzetekhez képest. Az alábbi ábrán láthatjuk elhelyezkedésüket:



A következő példaprogram annyiban tér el az előzőtől, hogy már az összes billentyűnek saját kódja van. A KEY tömböt 256 eleműre tágítjuk, a bővített billentyűk a [128..255] intervallumba esnek. Eredeti SCAN kódjukat úgy kapjuk meg, hogy az igazra vált elemből kivonunk 128(80h)-t. Mivel az összes billentyű kódja nagyobb nullánál, ezért a nulladik elemében tároljuk átmenetileg az információt a bővített billentyűkről. (Ha KEY[0]=TRUE, akkor a következő billentyű SCAN kódjához 128-at adunk.)

```
{keyb3.pas}

uses DOS;

var  KEY: array [0..255] of boolean;
                         { Most 2×128 elemű a tömb                    }
     OLD: procedure;     { A régi megszakításvektor                   }
     i  : byte;          { A FOR ... TO ... DO ... ciklushoz          }
```

```
procedure NewIRQ; assembler; asm
   push   ds              { Azokat a regisztereket, amelyek a meg-  }
   push   ax              { szakítás végrehajtása közben módosulnak,}
   push   bx              { a veremben tároljuk                      }
   push   cx
   xor    cl,cl           { Néhány regiszter kezdeti értéke          }
   mov    bh,cl
   mov    ax,seg key      { DS a KEY tömb szegmense                  }
   mov    ds,ax
   in     al,60h          { AL-be beolvassuk a billentyű SCAN kódját}
   cmp    al,0e0h         { Ha bővített, előtte E0-t olvashatunk be }
   jnz    @1              { Ha nem az, azt tesszük, amit korábban    }
   mov    byte [offset key],1 { Jelezzük, hogy a következő bil-      }
                          { lentyű kódjához 128-at kell majd adni    }
   jmp    @end            { Most nincs több dolgunk                  }
@1:                       { Ha nem $E0 kódot kaptunk                 }
   cmp    byte [offset key],1 { Megvizsgáljuk, hogy előzőleg nem     }
   jnz    @2              { $E0 kódot kaptunk-e                      }
   mov    cl,128          { Ha igen, 128-cal növeljük a SCAN-kódot   }
   mov    byte [offset key],0 { Nullázzuk a jelző-változót           }
@2:
   mov    bl,al           { BL-be is bevisszük a kódot               }
   and    bl,127          { Az alsó 7 bit adja a valódi SCAN-kódot   }
   add    bl,cl           { Ha bővített a billentyű, BL:= BL+128     }
   xor    cl,cl           { CL regiszter nullázása                   }
   shl    al,1            { C jelzőbit (FLAG) = 7. bit               }
   cmc                    { Ezt negáljuk, így ha 0, felengedtük az   }
                          { adott billentyűt, ha 1, akkor lenyomtuk  }
   adc    cl,00           { CL most 0 vagy 1 lehet (FALSE, TRUE)     }
   mov    [offset key+bx],cl { A megszakítást kiváltó billentyű-     }
                          { höz tartozó logikai változó beállítása   }
@end:
   in     al,61H          { A megszakítás csatorna visszaállítása    }
   mov    ah,al
   or     al,80H
   out    61H,al          { Jelzés a billentyűzetnek                 }
   mov    al,ah
   nop                    { Kevés várakozás                          }
   nop                    { (a soros adatkiküldés sebessége miatt)   }
   nop
   out    61H,al
   cli
   mov    al,20H          { "Megszakítás vége" jelzés                }
   out    20H,al
   sti
   pop    cx              { Regiszterek visszaolvasása a veremből    }
   pop    bx
   pop    ax
   pop    ds
   iret                   { Vége a megszakításnak                    }
   end;
```

```
begin
 getintvec( $09, @OLD); { A régi megszakításvektort tároljuk      }
 setintvec( $09, @NewIRQ); { A $09 megszakítás ezentúl a NEWIRQ   }
                           { eljárást hívja meg                   }
 fillchar(key,sizeof(key),0); { KEY tömb nullázása (FALSE)        }
 repeat
  for i:=   1 to 127 do if key[i] then writeln(i:3);
                          { A lenyomott billentyű kódja a képernyőre}
  for i:= 128 to 255 do if key[i] then { Ha a bővített a billen-  }
   writeln(i-128:3,'+');{ tyű, egy pluszjelet is kiírunk utána    }
  until key[1];          { ESC megnyomásáig                       }
 setintvec( $09, @OLD);
 end.
```

A következő táblázat az egyes billentyűkhöz tartozó kódokat mutatja be, tizenhatos számrendszerben. Sok más könyvben található ugyanilyen táblázat, egy különbséggel, hogy itt néhány billentyű kódja nagyobb 80h-nál. Az eredeti SCAN kódját ezeknek a billentyűknek a kódból 80h-at kivonva kaphatjuk meg.

Ugyanez a táblázat megtalálható a lemezmellékleten, a **SCANCODE.TXT** fájlban. Ezt érdemes kinyomtatni, így nem kell mindig idelapozni.

Az egyes billentyűk SCAN-kódjai, tizenhatos számrendszerben

Ec		F1	F2	F3	F4		F5	F6	F7	F8		F9	10	11	12		PS	SL	Pu
01		3B	3C	3D	3E		3F	40	41	42		43	44	57	58		37	46	45

0	1	2	3	4	5	6	7	8	9	Ö	Ü	Ó	Ű	<-		Is	Hm	PU		NL	/	*	-
29	02	03	04	05	06	07	08	09	0A	0B	0C	0D	2B	0E		D2	C7	C9		45	B5	37	4A
Tab	Q	W	E	R	T	Z	U	I	O	P	Ő	Ú				Dl	En	PD		7	8	9	
0F	10	11	12	13	14	15	16	17	18	19	1A	1B	<┘			D3	CF	D1		47	48	49	+
Caps	A	S	D	F	G	H	J	K	L	É	Á	Enter								4	5	6	
3A	1E	1F	20	21	22	23	24	25	26	27	28	1C								4B	4C	4D	4E
Shift	Y	X	C	V	B	N	M	,	.	-	Shift						Up			1	2	3	
2A	2C	2D	2E	2F	30	31	32	33	34	35	36						C8			4F	50	51	En
Ctrl		Alt	Space							Alt		Ctrl				<-	Dn	->		0		.	tr
1D		38	39							B8		9D				CB	D0	CD		52		53	9C

A módosított megszakítás a *Game* egység *InitKey* eljárásával (8.8. rész) is beállítható. Ez a unit is tartalmaz egy *Key* tömböt. Ha lenyomott állapotban van egy billentyű, akkor ennek a tömbnek a billentyűhöz tartozó eleme igaz. Hogy

melyik billentyűhöz melyik elem tartozik, azt olvashatjuk le a fenti táblázatból (ami a **SCANCODE.TXT** fájlban is megtalálható).

3.2. Az egér programozása, a *Mouse* egység

Az egér adatait legegyszerűbben a **$33 megszakítás** segítségével kérdezhetjük le, illetve állíthatjuk be. A 33h sorszámú megszakítás alapesetben üres, csak az egérkezelő (*mouse driver*) programok betöltése után léteznek a rutinok. Ezt a megszakítást használja a *Mouse* Pascal unit is. Először az *InitMouse* függvénnyel inicializálni kell az egeret, majd ha az egérkurzort meg akarjuk jeleníteni, a *ShowMouse* eljárást is meg kell hívni. Ennyi elegendő ahhoz, hogy látható legyen, az egység azonban nem csak ezt a két szubrutint tartalmazza, ebben a részben a többi eljárással és függvénnyel is megismerkedhetünk, betűrendben.

Az egér koordinátáira ugyanazok érvényesek, mint egy pixel koordinátájára, abszcisszája 0 és 319 közé esik, ordinátájának pedig 0 és 199 között kell lennie. Ez a koordináta-rendszer azonban csak erre az egységre és MCGA képszerkezetre érvényes. Valójában az egérkurzor abszcisszája minden képszerkezetnél beleesik a [0..639] intervallumba, vagyis a képernyő jobb oldali legszélső képpontjainak első koordinátája 639. Hogy ne kelljen mindig kettővel osztani vagy szorozni, a *Mouse* egység már a normál MCGA koordinátákat használja. Ezért ez a unit csak 320×200-as felbontás mellett ad helyes koordinátákat, szöveges módban például már nem, annak ellenére, hogy más videomódoknál is működik.

ButtonPressed függvény

Szintaxis: ButtonPressed: boolean;
Értéke akkor igaz, ha az egér egy gombja le van nyomva. A 33h megszakítás sok egér középső gombjának lenyomását nem tudja érzékelni, ezért legtöbbször csak a bal vagy a jobb gomb kattintása után kapunk igaz értéket.

DisableArea eljárás

Szintaxis: DisableArea(X1, Y1, X2, Y2: word);
Meghatároz egy (X1;Y1) bal felső sarkú, (X2;Y2) jobb alsó sarkú téglalap alakú tartományt, ahol az egér nem mozoghat. Az X1 és X2 változók értékének 0 és 319 közé kell esnie, az Y1 és Y2 pedig 0 és 199 között lehet. Ezenkívül

igaznak kell lenniük a következőknek: X1<X2 és Y1<Y2. Ha például a képernyő jobb alsó sarkában lévő 8×8-as részen nem mozoghat az egérkurzor, a következőt kell beírni:

```
DisableArea( 312,192,319,199);
```

EnableArea eljárás

Szintaxis: EnableArea(X1, Y1, X2, Y2: word);
Az egér mozgásterét változtatja, az utasítás végrehajtása után az egér csak az (X1;Y1) bal felső és (X2;Y2) jobb alsó sarkú téglalapon belül mozoghat. A koordinátákra ugyanazok érvényesek, mint a *DisableArea* eljárásnál, vagyis X1 és X2 0 és 319, Y1 és Y2 pedig 0 és 199 közé esik (a szélső értékeket még felvehetik), és X1<X2, Y1<Y2. Az eljárás a következő paraméterekkel az egész képernyőn engedélyezi a mozgást:

```
EnableArea( 0,0,319,199);
```

Azt, hogy az egérkurzor a képernyő bal felén jelenhessen meg, kétféleképpen oldhatjuk meg:

```
EnableArea( 0,0,159,199);
```

vagy

```
DisableArea( 160,0,319,199);
```

GetMouse eljárás

Szintaxis: GetMouse(var M: MouseType);
Az eljárás az egér adatait adja meg a paraméterében megadott változóba. A *MouseType* típus a *Mouse* egységben így lett deklarálva:

```
type MouseType = record
 X,Y: word;
 Left, Middle, Right: boolean;
 end;
```

X és Y az egér koordinátái, az utolsó három mező (*Left*, *Middle*, *Right*) értéke pedig akkor igaz, ha a bal, középső vagy jobb gombot lenyomtuk. Ezeket az adatokat egyszerűbben megkaphatjuk a *MouseX*, *MouseY*, *LeftButton*, *RightButton* függvények segítségével.

GetSensitivity eljárás

Szintaxis: GetSensitivity(var M: SensType);
Az egér érzékenységét kérdezi le az M változóba, melynek típusa a *Mouse* unit meghatározása szerint:

```
type SensType=record
 X,Y,S: word;
 end;
```

X és Y a vízszintes és függőleges érzékenységet adják, vagyis az adott irányban hány kb. 1/200 hüvelyk hosszúságú egység (**Mickey**) található. Az S tartalmazza a sebességet, a másodpercenkénti *Mickey*-ek számát. Nem túl gyakran van szükség erre az eljárásra, talán inkább a párjára, a *SetSensitivity*-re.

GotoMouse eljárás

Szintaxis: GotoMouse(X, Y: word);
Az egérkurzor helyzetét a paraméterekben megadott értékek szerint állítja be. A koordináták az MCGA képernyő-koordinátákkal kompatibilisek.

HideMouse eljárás

Szintaxis: HideMouse;
Eltünteti az egérkurzort. Ha nem látható, attól még ugyanúgy le tudjuk kérdezni helyzetét, állapotát, ugyanolyan műveletek hajthatók végre, mintha látható lenne. Csak a felhasználó számára lesz egy kicsit nehezebb az egeret a kellő helyre irányítani.

InitMouse függvény

Szintaxis: InitMouse: boolean;
Inicializálja az egeret, bekapcsolja azt, de láthatóvá még nem teszi. Minden egérkezelő program elején le kell futtatni ezt a szubrutint. Ha a visszatérési érték igaz, a művelet sikeresen befejeződött, ha nem, akkor az egér nem installálható.

LeftButton függvény

Szintaxis: LeftButton: boolean;
A függvény értéke akkor, és csak akkor igaz, ha az egér bal oldali gombja le van nyomva. Használata igen egyszerű, a következő rövid program be is mutatja:

```
uses Mouse;
begin
 if not InitMouse then halt; {Megállítás, ha nem installálható}
 write('Kilépés - bal gomb');
 repeat until LeftButton;
 end.
```

MouseMoved függvény

Szintaxis: MouseMoved: boolean;
A függvény akkor igaz, ha az egeret megmozdítottuk. Például a képkímélőknél lehet hasznos ez a rutin, vagy ha az egér koordinátáit állandóan ki szeretnénk írni a képernyőre, így nem kell folyamatosan írni, elég csak akkor, ha azok megváltoztak.

MouseSpeed eljárás

Szintaxis: MouseSpeed(S: word);
Az egér maximális sebességének beállítása. S a sebesség felső korlátja (lépésköz/másodperc). Nem érdemes átállítani, mert az *InitMouse* utáni alapértékek általában megfelelőek, de kísérletezni persze lehet.

MouseX függvény

Szintaxis: MouseX: word;
Az egér abszcisszája (első koordinátája) az MCGA képszerkezethez igazítva, így értéke [0..319] között van.

MouseY függvény

Szintaxis: MouseY: word;
Az egér második koordinátája. Értéke MCGA üzemmódban [0..199] közé esik.

RightButton függvény

Szintaxis: RightButton: boolean;
A függvény igaz, ha a jobb oldali gomb le van nyomva (a *LeftButton*-hoz hasonlóan).

Sensitivity függvény

Szintaxis: Sensitivity: word;
Az egér érzékenységét adja, azt hogy milyen sebességgel mozog. Nem lényeges annyira ez a függvény.

SetSensitivity eljárás

Szintaxis: SetSensitivity(X, Y, S: word);
Az egér érzékenységét változtathatjuk vele, ha a kiindulási állapot nem felel meg. X a vízszintes, Y a függőleges *Mickey*-ek (1/200 inch egységek) száma, S a sebesség (*Mickey*/másodperc). Ha nem jók az alapértékek kevés kísérletezéssel, próbálkozással megfelelően be tudjuk állítani egerünk érzékenységét.

ShowMouse eljárás

Szintaxis: ShowMouse;
Az egérkurzort láthatóvá teszi. Ez grafikus üzemmódban egy fehér nyíl, fekete szegéllyel.

4. Háttér

Legegyszerűbb az egyszínű háttér, azonban a legtöbb játék ennél jobb képet igényel. Vegyük alapul a **SHOWBOB5.PAS** példaprogramot! A háttérnek – mint az MCGA üzemmódnak is – 320×200 felbontásúnak és 256 színűnek kell lennie. A memóriában a *BackGround* pointerrel meghatározott 64000 bájton helyezkedik el. A következő eljárás a hátteret az eljárás paraméterében megadott C színűre festi. Így nagyon könnyen elkészíthetjük az egyszínű hátteret.

```
procedure FillBackGround( C: byte); assembler;
 asm
   mov     al,C
   mov     ah,al
   mov     cx,32000
   cld
   les     di,background
   rep     stosw
  end;
```

Ha tudjuk, hogy játékunk háttere egyszerű, de nem egyszínű, különböző színű pontokat kell kigyújtani. Ezt hasonlóképpen érhetjük el, mint a **PUTPIXEL.PAS** program *PutPixel* eljárásában, csak az ES:[DI] most nem a képernyőre, hanem a *BackGround* pointer által meghatározott tartomány valamely bájtjára mutat. Egyébként az eljárás ugyanaz, először ES:[DI]-be betöltjük a *BackGround* mutatót, majd DI-hez hozzáadunk 320×Y+X-et.

```
procedure PixelBack( X, Y: word; C: byte); assembler; asm
   les     di,background
   mov     ax,320          { Egy sor 320 képpontból áll              }
   mul     y               { Megvan a megfelelő sor első bájtja      }
   add     ax,x            { AX a megfelelő címet tartalmazza,       }
   add     di,ax           { már csak hozzá kell adni DI-hez         }
   mov     al,c
   mov     es:[di],al      { A megfelelő bájt C-re változtatása      }
   end;
```

Ezzel a két eljárással már tudunk egyszerű hátteret készíteni. Először befestjük egyszínűre a *FillBack*-kel, majd a *PixelBack* eljárás és *FOR ... TO ... DO* ciklusok segítségével vízszintes vonalakat rajzolva kész is egy egyszerű ugrálós játék háttere.

4.1. LBM kép betöltése

Egy igényesebb játékhoz nem elég ilyen primitív háttér. Ha olyan programot írunk, melyben a játék mögötti kép változatlan, legjobb rajzolóprogrammal elkészíteni, vagy kézzel, és utána szkenner segítségével lemezre vesszük. Akár programmal, akár kézzel rajzolunk, végül képünknek egy szabványos, sokak által ismert képformátumban kell elhelyezkednie. **Képformátum** az a tárolási módszer, mely a kép egyes bájtjait (színeit, pontjait) tárolja. Ez lehet tömörített és tömörítetlen. A legáltalánosabb képformátumok: **BMP, GIF, JPG, LBM, PCX, TIF**. Az esetek túlnyomó többségében a fájl kiterjesztése megegyezik az előbb felsorolt szavakkal, tehát például a RAJZ.BMP nagy valószínűséggel képfájl (képet tartalmazó fájl), BMP formátumú. Ezeket a tárolási módszereket a legtöbb grafikakészítő vagy -feldolgozó program ismeri, ezért célszerű lenne az egyiket kiválasztani, és az általa tárolt kép megjelenítési módját kicsit részletesebben ismertetni. Elég egy formátummal foglalkozni, mert a rajzoló- és képnézőprogramok legtöbbjével ezek egymásba konvertálhatók.

Mire lesz szükség? Ahhoz, hogy rajzunk a számítógépben megjelenjen, szükség van vagy egy rajzolóprogramra, vagy egy szkennerre (a hozzá tartozó kezelőszoftverrel együtt), vagy egy képlopóra. A képlopó (**capture**) programok lényege a következő: memóriarezidensek, egy tetszőleges programból, akár egy játékból előre megszabott billentyű vagy billentyűkombináció lenyomása után az éppen akkor aktuális kép a lemezre kerül, előre meghatározott képtárolási formában. Persze nem vall valami fantáziadús és igényes munkára, ha játékának háttereit valaki ilyen lopott képekből állítja össze.

A rajz most már a fent említett formátumok egyikével tárolva a lemezre került, feltéve hogy rajzoló- vagy képlopó programunk, vagy szkennerünk szoftvere ismeri a fenti formátumok valamelyikét. Ebben a könyvben csak az LBM képszerkezet ismertetésére kerül sor, így szükség lesz még egy képkonvertáló programra, mely LBM formában is el tudja menteni a képet.

Mielőtt rátérnénk arra, miért pont az LBM-et választjuk, ismerkedjünk meg néhány más típussal is! A **BMP** fájlok a képet tömörítetlen (ritkább esetben RLE kódolással) formában tárolják, egy 320×200/256 kép tárolása így legalább 64000 bájtot vesz igénybe, ehhez még hozzájön a fejléc hossza és a paletta. A fejléc tartalmazza a képre vonatkozó adatokat (szélesség, magasság, színek száma stb.). Igaz, hogy a BMP-ben tárolt képek megjelenítése egyszerű, és a

BMP szerkezet is elterjedt (talán a legelterjedtebb tárolási forma), ennek ellenére nem ezt használjuk, mert túl nagy helyet foglal. Tegyük fel, hogy egy házban játszódó játékot szeretnénk írni, a ház minden szobáját BMP fájlként tároljuk. Hogy játékunkat könnyen másolhassuk egyik gépről a másikra, érdemes rövidnek lennie, mert jó ha ráfér egy $3^1/_2$"-es lemezre, ami általában 1,44 Mbájt méretű. Kevés számolás után nyomban kiderül, hogy a ház legfeljebb 22-23 szobát tartalmazhat.

A **GIF** és a **JPG** formátumok elterjedtek, nagyon jó arányban tömörítnek, azaz a fájl mérete a tömörítetlen kép méreténél sokkal kisebb. A GIF-ben és JPG-ben kódolt képek megjelenítése azonban nagyon nehéz megjeleníteni, és aránylag sok időt vesz igénybe. Ráadásul a GIF fájlokban alkalmazott LZW kódolás jogvédett, ezért elvileg nem is lehet szabadon alkalmazni. (Visszafejteni azért még lehet.)

A középutat az **LBM** jelenti, melynek visszafejtése egyszerű, és a képet sűrítve tárolja. A tömörítés lényege az, hogy az egymás mellett elhelyezkedő azonos bájtok helyére két bájt kerül. Ez pont jó nekünk, mert játékaink többségének háttere sok azonos színű pontot tartalmaz, például a fal vagy az ég. Így elég jó tömörítési arányt érhetünk el.

A lenti *LoadLBM* megjelenítő csak **256 színű**, **320×200**-as felbontású képeket tud megjeleníteni, noha az LBM fájlok tárolhatnak teljesen más méretű és színű grafikákat. Viszont mi csak az MCGA képszerkezettel foglalkozunk, ezért más színű képeket nem is vagy csak igen nehezen tudnánk ábrázolni. A kép mérete ettől függetlenül ugyan lehetne nagyobb, és ilyenkor egy megadott 320×200-as tartományát rajzolnánk ki a képernyőre, de a fájl mérete – a megjelenítés módja miatt – nem haladhatja meg a 65520 bájtot. Egyébként ha az MCGA felbontásától eltérő lenne a kép mérete, az a megjelenítést lassabbá tenné, nem lehetne folyamatos megjelenítést alkalmazni, a memóriaigénye is nagyobb lenne stb. Maradjunk annál, hogy képünk szélessége 320, magassága 200 képpont, a színek száma pedig 256. Erre már az elkészítésénél is ügyeljünk!

A fájl – felépítését tekintve – két főbb részre osztható, elején a kép nemgrafikus adatait találhatjuk (fejléc, paletta stb.), a végén pedig a grafikus adatokat. Az elsővel különösebben nem érdemes foglalkoznunk, hiszen a képméret, színek száma adott, csak a palettát olvassuk ki onnan. A második nagy egységet az elsőtől nyolc bájt választja el egymástól: a **BODY** címke az első négy

bájt, a második négy bájt a kép tömörített mérete. Itt jegyezzük meg, hogy az LBM fájlokban a bájtnál nagyobb méretű adatok (szavak, duplaszavak) nem bájtfordítottan kerülnek tárolásra, egy szó beolvasása után annak alsó és felső bájtját ki kell cserélni (az *xchg* utasítással).

A tömörítési eljárás pedig a következő. A képpontok (sorfolytonosan) változó hosszúságú egységekre vannak felosztva, melyek hossza legfeljebb 128 pixel. Egy ilyen egység lehet tömörített vagy tömörítetlen. Minden egység elején található egy B bájt, amelynek ha a 7. bitje 0, akkor az azt követő B+1 bájt (legfeljebb 128, mert a B 7. bitje zérus) nincs tömörítve, azaz egy bájt egy pixelt határoz meg, melyeket sorfolytonosan kell kiírni. Ha a B legfelső bitje 1, akkor tömörítésről van szó. Ilyenkor az egység hossza (B-vel együtt) 2 bájt, és a másodikat kell **(neg B)+1**-szer kirakni, vagyis a B *kettes komplemenséhez* egyet adunk, és ennyiszer egymás után, sorfolytonosan kirakjuk a B-t követő bájtot. Az egyes egységek végét nem jelzi semmi, így be kell iktatni egy számlálót, aminek kezdeti értéke **B** vagy **(neg B)+1**, ha ez eléri a nullát, új egység kezdődik.

Kis számolás után rájöhetünk, hogy egész jó tömörítési arányt érhetünk el. Ha a kép egyszínű, minden 128 pixelt 2 bájton tárolva az arány 1/64 (1,6%). A fájl mérete így 64 helyett (nem-grafikus adatokkal és a palettával) nem több, mint 2 kB. Viszont ha minden képpont más színű, mint a mellette levő, akkor a fájl mérete nagyobb lesz (64000/128 bájttal), mintha nem lenne tömörítve.

A paletta a fájl **$30** címétől kezdődik, 256 színnél 768 bájt hosszú. Már a méretéből is látszik, hogy R-G-B színösszetevőnként van tárolva, csak egy kicsit másképp, mint ahogy azt gondolnánk (8 bites formában). Itt a 7-2. bit jelzi az adott szín adott összetevőjének az intenzitását, a 1-0. bit értéke nulla. Ezért, hogy a videokártya a megfelelő adatokat kapja, minden P palettabájtot el kell forgatni jobbra kettővel (normális színadat = **P shr 2**).

Most már mindent – ami nekünk fontos – tudunk az LBM fájlról, vázoljuk a megjelenítő eljárást!

1. A lemezről az egész fájl tartalmának beolvasása egy előre lefoglalt változóba. Éppen ezért a fájl mérete nem lehet nagyobb 65520 bájtnál, mert ez a változók maximális mérete. Megtehetnénk azt is, hogy egyből a lemezről

fejtjük vissza az adatokat, így nem lenne szükséges feltétel a 64K szabad memória, de lényegesen lelassítaná a műveletet.

2. A paletta beállítása a $30. bájttól kezdve, minden bájtot 2 bittel jobbra forgatva.
3. BODY címke megkeresése. Először a 'B'-t kutatja a *scansb* karakterlánc-művelet segítségével, majd ha talált egy ilyen bájtot (értéke 66), összehasonlítja az azt követő három bájttal együtt a BODY címkével. Ha végigvizsgálta a megadott tartományt úgy, hogy nem találta meg ezt a négy karaktert, hibát jelez.
4. Megjelenítés.

Az eljárás meghívása előtt be kell állítani a videokártyát MCGA üzemmódba, ha a képet a képernyőre akarjuk kirajzolni. A *FileName* paraméterben a fájl nevét adjuk meg, elérési útvonallal együtt, ha szükséges, a *P* pedig arra a legalább 64000 bájt hosszúságú területre mutasson, ahova a képet meg szeretnénk jeleníteni. Például ha a második paraméter **PTR($A000,0)**, akkor a képernyőn lesz látható.

```
procedure LoadLBM( FileName: string; p: pointer);
                        { FILENAME: forrás, P: cél (ide írjuk)    }
 var f: file;           { Változó a fájlműveletekhez              }
     p1, p2: pointer;   { Mutatók a memóriafoglaló eljárásokhoz   }
     i,j:  byte;        { Számlálók a FOR... ciklusokhoz          }
     Error: boolean;    { Hibát jelző logikai változó             }
     fs: word;          { A fájl méretét fogja tárolni            }
 const
     lab: array[0..3] of char = 'BODY';
                        { Ezt a címkét kell majd megkeresni: BODY }
 begin
  mark( p1);
  assign( f, FileName);
  reset( f, 1);
  if ioresult<>0 then begin
   write('Nincs ilyen fájl');
   halt;
   end;
  fs:= filesize( f);
  if fs>65520 then halt;{ Ennél nagyobb fájlt nem tudunk ábrázolni}
  getmem( p2, fs);      { Helyfoglalás, itt látszik, hogy a fájl  }
                        { mérete nem haladhatja meg a 64 kB-ot    }
  blockread( f, p2^, fs); { Fájl beolvasása a memóriába           }
  close( f);
  for  i:= 0 to 255 do begin
   port[$3c8]:=i;
   for j:= 0 to 2 do
    port[$3c9]:=mem[seg(p2^):ofs(p2^)+$30+3*i+j] shr 2;
   end;                 { Paletta beállítása                      }
```

```
  asm
    mov     error,0         { A hibát jelző változó nullázása          }
    mov     al,byte [lab]   { AL-be 'B'(66) kerül (a címke 1. bájtja)  }
    les     di,p2           { A P2 által meghatározott területen ke-   }
    mov     bx,di           { ressük a 'BODY' címkét.                  }
    mov     cx,fs           { A számláló kezdeti értéke a fájl mérete  }
    cld
@1:repnz    scasb           { Az első bájt keresése                    }
    jnz     @err            { Ha nem talált, hibás a fájl              }
    mov     si,offset lab   { Ha talált egy 'B' karaktert, az még nem  }
    push    di              { biztos, hogy a 'BODY' első karaktere,    }
    push    cx              { ezért ellenőrizni kell a másik hármat is}
    dec     di              { DI most a talált 'B'-re mutat            }
    mov     cx,4            { Összesen 4 karaktert vizsgálunk, ennyi   }
    repz    cmpsb           { a címke hossza (4 bájt)                  }
    pop     cx              { A vizsgálathoz szükséges, előzőleg el-   }
    pop     di              { mentett regiszterek visszaállítása       }
    jnz     @1              { Ha valahol eltérés volt, folytatjuk      }
    mov     dx,es:[di+5]    { A 'BODY'-t követő duplaszó a tömörített  }
    xchg    dl,dh           { méret, nekünk csak az utolsó 2 bájt kell}
                            { A másik kettő nulla, mert a fájl nem le-}
                            { het 65520 bájtnál nagyobb                }
    add     di,7            { Az első adat a 'BODY' utáni 5. bájt      }
    push    ds              { DS kell a MOVSB utastáshoz               }
    mov     ax,di           { DI-t átmenetileg tároljuk                }
    les     di,p            { ES:[DI]-be a célcím kerül (P)            }
    lds     si,p2           { DS:[SI]-be pedig a forráscím             }
    sub     ax,bx           { AX-ből levonjuk a P2^ ofszetjét          }
    add     si,ax           { És ezt adjuk SI-hez, így DS:[SI] az első}
    xor     ch,ch           { grafikus adat helyét határozza meg       }
@2:cmp      dx,0            { Ha a számláló elérte a nullát - vége     }
    jz      @vege
    mov     cl,[si]         { CL az egység első bájtja (B)             }
    inc     si              { Ezután a következő bájttal lesz dolgunk  }
    dec     dx              { Számláló csökkentése                     }
    test    cl,128          { Ellenőrizzük CL legfelső bitjét          }
    jz      @normal         { Ha 0, CL+1 tömörítetlen bájt következik  }
    neg     cl              { Egyébként negálni kell,                  }
    inc     cl              { és hozzá kell adni egyet.                }
    mov     al,[si]         { És ennyiszer kell kirakni a következő    }
    inc     si              { bájtot                                   }
    dec     dx
    rep     stosb
    jmp     @2
@normal:
    inc     cl
    sub     dx,cx           { A számláló csökkentése (ezért CH=0)      }
    rep     movsb           { CL+1 bájt egyszerű kimásolása            }
    jmp     @2
@err:
    mov     error,1
@vege:
    pop     ds
  end;
```

```
if Error then begin     { Ha hiba volt...                    }
 asm
   mov    ax,03h
   int    10h           { Visszatérés a szöveges módhoz      }
  end;
 writeln('Olvasási hiba: '''+FileName+'''+#7);
 halt
 end;
release( p1);           { Lefoglalt memória felszabadítása   }
end;
```

Ha a későbbiekben LBM hátteret választunk játékunkhoz, mindig ügyeljünk arra, hogy a BOB-ok a fájlban tárolt paletta szerint fognak megjelenni. Most pedig nézzünk egy példát egy ilyen háttér elkészítésére! Tegyük fel, hogy hozzájutottunk egy 1024×768-as, 256 színű BMP képhez (például szkennerrel beolvastuk). Ezt egy jó fényképfeldolgozó programmal lekicsinyítjük 320×200-assá, lehetőleg a torzítást (aminek oka: 320/1024≠200/768) is kiküszöböljük. Ha a 256 szín között nincs olyan, amit később a BOB-okhoz fel szeretnénk használni, összemosunk kettőt. Legyen a kép C1 és C2 színe közel azonos (ugyanolyan színű, csak egy kicsit más árnyalatú). Az összes C1 színű pontot cseréljük ki C2 színűre (a legtöbb fényképfestő ezt el tudja végezni), így felszabadult egy szín, C1, amellyel később tetszőlegesen rendelkezhetünk (bárhogy változtathatjuk színösszetevőit).

Tegyük fel, hogy van egy kis probléma: képfeldolgozó programunk nem ismeri az LBM formulát. Ekkor szükségünk van egy képkonvertálóra. Ilyen a **VPIC 6.2** képnéző program is, mellyel nemcsak képeket tudunk megnézni, hanem a képfájlokat más formátumban is elmenthetjük. Hogyan konvertáljuk vele a képeket? Nézzük meg lépésenként:

1. A program elindítása után nyomjuk meg az **F9**-et, majd gépeljük be az átalakítandó kép elérési útvonalát.
2. A kurzormozgató billentyűkkel válasszuk ki a képet, és nyomjunk **Enter**-t.
3. Üssük le a következő billentyűket, ilyen sorrendben: **D, Y, Y, Y**. (Nyugodtan várhatunk két gomb megnyomása közt, így legalább el tudjuk olvasni, mit ír ki a program.)
4. Kevés várakozás után nyomjunk **Enter**-t, majd **ESC**-vel lépjünk ki a programból.

Végül nézzünk egy rövid kis programot, mely a paraméterében megadott LBM képet jeleníti meg. Ezt DOS-segítő fájlkezelőnkbe akár be is építhetjük, persze

csak ha van rá lehetőség (például az *'Extension file edit'* menüpontnál lehet egyes programoknál beállítani).

```
{wiewlbm.pas}

uses Crt;

procedure LoadLBM( FileName: string; p: pointer);

 { Ez a fenti eljárás }

begin
 if ParamCount=0 then halt;
 asm
   mov ax,13h
   int 10h
  end;
 LoadLBM( ParamStr( 1), PTR($A000,0));
 ReadKey;
 asm
   mov ax,03h
   int 10h
  end;
 end.
```

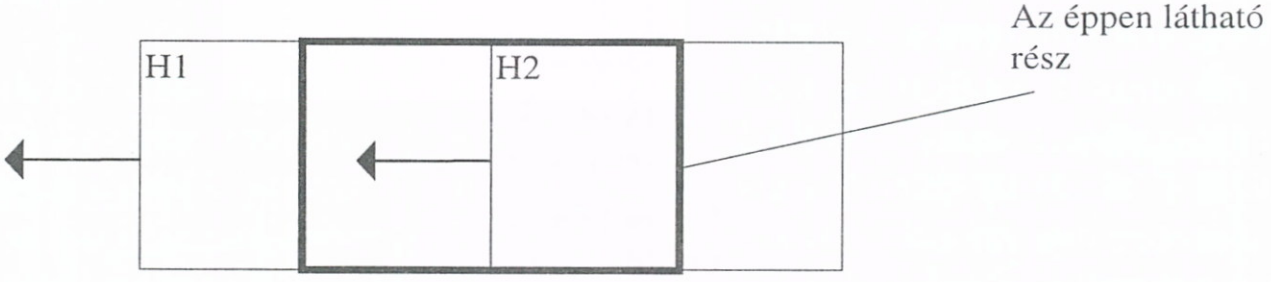
Ez a rövid kis program gyorsabban jeleníti meg a képet, mint a legtöbb képnéző, viszont kevesebbet is tud a legtöbb képnézőnél.

4.2. A MAP képszerkezet

Ahhoz, hogy a háttér gördíthető legyen, méretét növelni kell. Ha a háttér pixeleit bájtonként tárolnánk, akkor csak néhány ezer bájttal lehetne nagyobb 64000-nél, legfeljebb 65520 bájt, ennél nagyobb változókat ugyanis a Pascal nem képes kezelni. Tehát be kell férnie egy szegmensbe (64K), és méretének nagyobbnak kell lennie 320×200-nál. Ezért valahogy tömöríteni kell.

Tömörítés nélkül meg lehetne oldani a következőképpen is. Van két 320×200-as háttér: H1 és H2, H1 látszik, H2 nem. H2-t jobbról görgetjük be úgy, hogy a háttér (ami az elején még H1) minden oszlopát (a 2.-tól kezdve) eggyel balra toljuk, majd az utolsó oszlopba H2 első oszlopát írjuk. Ezt 320-szor megismételve (persze később már nem a H2 első oszlopát, hanem a másodikat, harmadikat stb. kell beírni) a két kép kicserélődik. (Lásd az ábrát lent.) Ennek a görgetési módszernek az az egy előnye van, hogy a két kép tetszőleges lehet. Hátrányai: nagy memóriaigény (128K), egyszerre csak egyirányú görgetés,

darabosság. Ez utóbbi akkor jelenik meg, ha kettőnél több képet görgetünk, ilyenkor 320 léptetés után (az előző példánál maradva) H1 helyére új képet kell betölteni, ami sok időt vesz igénybe.



Legjobb az lenne, ha a memóriában egy hatalmas képet tárolhatnánk, és bármely részlete megjeleníthető lenne. Viszont ezt nagyon nehéz megoldani, a bővített memóriatartományt is igénybe kellene venni, ami miatt játékunk nem futna minden gépen (ami még nem is lenne olyan nagy baj, mert manapság már mindegyik gépben több RAM van 640KB-nál). Egyetlen megoldás, ha kitalálunk egy olyan tömörítési eljárást, ami könnyen visszafejthető, és jó arányban képes tömöríteni.

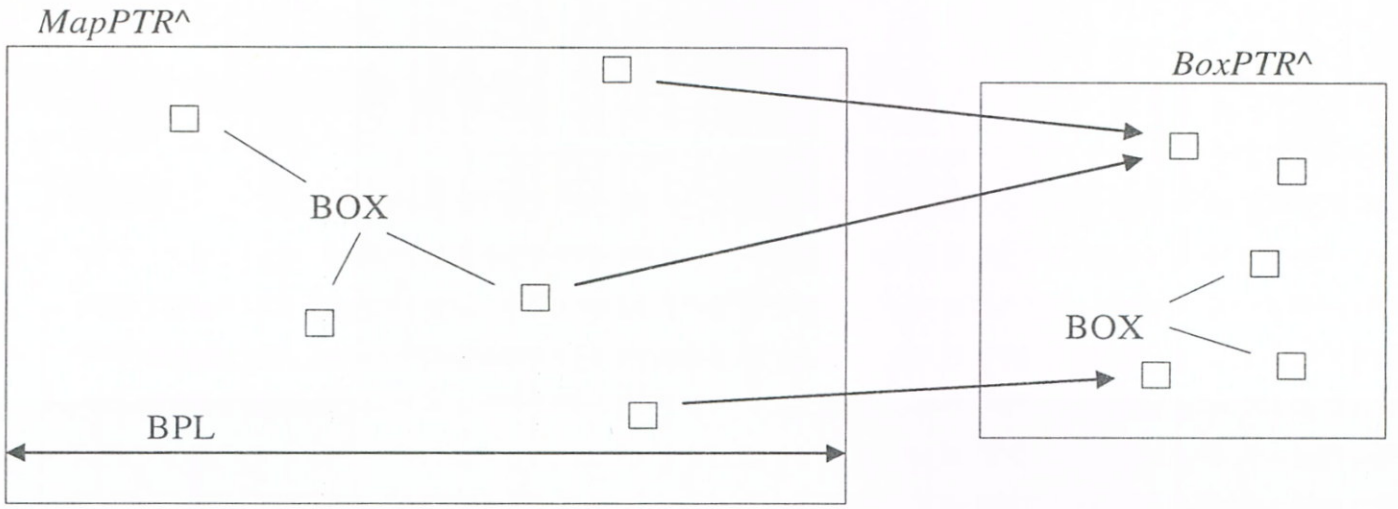
Elérkezett az idő, hogy megismerkedjünk a **MAP** képszerkezettel. Ez nem egy szabványos képtárolási módszer, a fogalmak, definíciók önkényesek. A grafikus adatokat két részletben tároljuk, egyik memóriatartományban van a **térkép** (*MAP*), másikban a **256** darab **8×8** pixel méretű **doboz** (*BOX*). A térkép minden bájtja meghatároz egy-egy dobozt, aminek 64 bájtját a második memóriarészben adjuk meg. Így elég egy dobozt csak egyszer megadni, a térképben csak utalni kell rá.

Olyan ez, mint a mozaik. Tegyük fel, hogy van 256-féle mozaikkövünk, mindegyik típusból korlátlan mennyiség áll rendelkezésünkre. A mozaikkövek jelképezik a BOX-okat, a kép, amit a kövekből rakunk ki, a térképet. Csak a 256 mozaikdarabkát kellett meghatároznunk, és azt, hogy a kép egyes rácspontjaiba milyen típusú mozaik kerüljön.

Legyen például a háttér egy hatalmas, 800×400 pixeles virágmező, 255 különböző virággal. Minden virág belefér egy 8×8-as négyzetbe. A 256. „virág" egy zöld négyzet, ebből lesz a fű ott, ahova nem akarunk virágot rakni. Ha minden pixelt egy bájton tárolnánk, a rét mérete 800×400, vagyis 320000 bájt lenne. A MAP szerkezetben ez sokkal kevesebb helyet igényel. Először is meg kell adnunk a virágok pontjait: 256×8×8=16384 bájt. Másodszor meg kell adnunk a

térképen, hova milyen virág kerüljön. Mivel a térképen, a teljes képhez képest, minden 64 bájtot egy bájt helyettesít, mérete 64-ede a kész képnek: 5000 bájt. Ez összesen 16384+5000=21384 bájt, ami 6,7%-a a tömörítetlen legelőnek.

Nézzünk minderre egy ábrát:



A két nagy téglalap a két memóriatartományt szemlélteti, kezdőcímük *MapPTR* és *BoxPTR*. A rajzon a BOX-okat szemléltető kis négyzetekből csak néhány van berajzolva, természetesen úgy kell elképzelni mind a két változót, hogy az egész fel van osztva kis négyzetekre (nincs üres hely). A *BPL* változó megadja, hogy a térképen soronként hány doboz található. E változó segítségével lesz a térkép kétdimenziós.

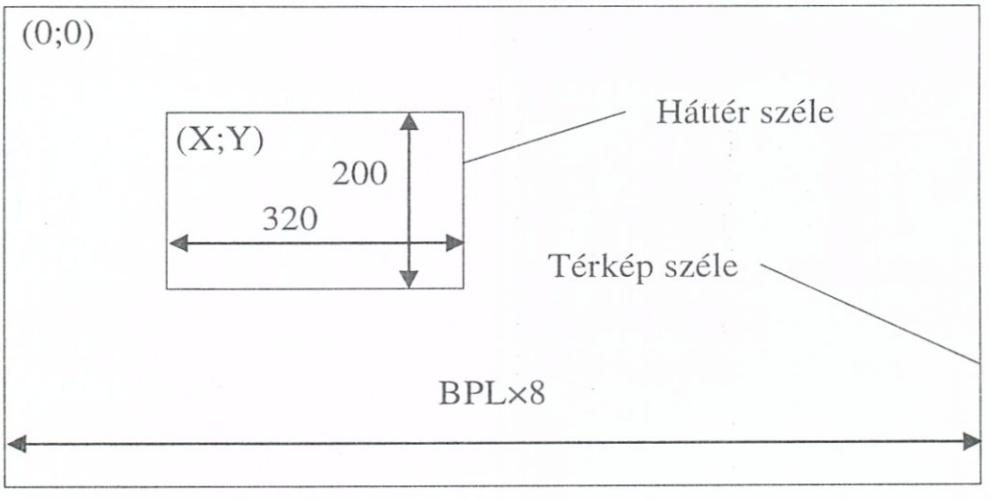
Ez a képszerkezet nagyon hasonlít a karakteres üzemmódra. A térkép, funkcióját tekintve, megegyezik a karakteres képmemóriával ($B800:0000 címtől kezdődő tartomány). Ott két bájt határoz meg egy karaktert, itt egy bájt egy BOX-ot. A karakterkészlet meg lényegében ugyanolyan, mint a BOX-készlet, csak mi máshogy tároljuk az adatokat. Minden bájt egy pixelt határoz meg. Minden dobozra jut 64 egymást követő bájt, a BOX-ok pontjait sorfolytonosan tároljuk. Az **N**. doboz ($0 \le N \le 255$) (**X; Y**) koordinátájú ($0 \le X \le 7$ és $0 \le Y \le 7$) pontjának színe:

```
MEM[seg(BOXPTR^):ofs(BOXPTR^)+N*64+Y*8+X]
```

A térképpel ellentétben a *BoxPTR* változónak csak egy kiterjedése van (egydimenziós), csak a könnyebb szemléltetés érdekében ábrázoltuk téglalapnak.

Érdemes elindítani a MAP-EDITOR programot (**MAPEDIT.EXE** a lemezmellékleten), töltsük be a **PELDA.MAP** képet! Így, kevés ismerkedés után, az a néhány lehetséges homályos folt is kitisztul, ami az eddig olvasottak után talán még maradt.

A térkép koordináta-rendszere nincs összefüggésben a BOB-ok koordináta-rendszerével. Mérete mindkét irányban lehet 65520 BOX, viszont szélességének és magasságának szorzata nem haladhatja meg a 65520 BOX^2-et, ennyi bájt lehet ugyanis legfeljebb egy Pascal-változó hossza. A koordináta-rendszer egységei azonban nem BOX-ok, hanem képpontok, pixelek. Így a szélessége és a magassága [1..524160] pixel között lehet. (65520×8=524160.) A nagy MAP fix, ezen mozog a háttér, és a térkép koordinátái valójában a háttér bal felső sarkát jelentik majd. Az alábbi ábra segít megérteni. X és Y a térkép koordinátái, *BPL* a soronkénti BOX-ok száma (***Boxs Per Line***). Persze nem lényeges az, hogy X és Y osztható legyen 8-cal, azaz a háttér szélei BOX-határra essenek, épp ez fogja a megjelenítést megnehezíteni.



Foglaljuk össze a fogalmakat és azok meghatározását:

MAP Térkép. A memóriában nincs összefüggésben a háttérrel, teljesen külön tartomány. Bájtjai egy-egy BOX-ot határoznak meg.

BOX 8×8-as méretű, 256 színű grafikai objektumok. 256 különböző mintájú BOX van, ezek egymást követve helyezkednek el a memória egy 16384 bájt hosszú tartományán.

Háttér Ez is egy memóriatartomány, 64000 (320×200) bájt hosszú. Itt semmiféle BOX-ot nem találunk, csak pixeleket meghatározó bájtokat.

BPL A térkép szélessége, BOX-ban. Képpontokban mérve a térkép szélében ennek nyolcszorosa. (A BOX szélessége 8.)

Végül vegyük számba ennek a képtárolási módszernek előnyeit és hátrányait!

Előnyök:
- A tömörítési arány nagyon jó, magát a képet 64-ed részére sűrítjük.
- Könnyű és gyors az adatok visszafejtése. Ez utóbbiban nagyrészt segít a 8×8-as BOX-méret (osztani kell, és a 8-cal való osztás bitforgatással is megoldható, ami lényegesen gyorsabb, mind a DIV utasítás).
- A kép mérete meglehetősen terjedelmes is lehet, több mint 65 képernyőnyi.

Hátrányok:
- A BOX mérete előre meghatározott.
- Kevés a 256 BOX, így képünk nem eléggé változatos. Bár ha elég ügyesek vagyunk, akkor ez pont elég. Nyilván ügyelni kell a térkép elkészítésénél arra, hogy sok ismétlődő elemet kell alkalmaznunk.
- Nagy memóriaigény (16384 bájt + a térkép, ami legfeljebb 65520 bájt).

4.3. Megjelenítés

A háttér és a térkép, tartalmát tekintve, tulajdonképpen teljesen elkülönül, közöttük kapcsolatot csak a BOX-ok képeznek, melyek a térkép grafikus adatait tárolják. A megjelenítés alapelve nagyon egyszerű: a térkép bizonyos bájtjától kezdve annak bájtjai által meghatározott BOX-okat elkezdjük kirakosgatni a háttérre, persze ha a háttér egy sorának végére értünk, kis kihagyás következhet, mert a térkép szélesebb lehet a háttérnél. A gyakorlatban túl lassú lenne minden képfrissítésnél vagy görgetés után ennyit számolgatni. A valódi megjelenítés lépései a következők:

1. A térképnek egy részletét, a kezdő képet meg kell jeleníteni. Ez lassú, mert sokat kell számolni (például 8-cal osztani), de a legelején egyszer mindenképp végre kell hajtani, mert ez lesz a kiindulás.
2. Ha a háttér elmozdul a térképen (vagy a térkép mozdul el a háttér alatt), azaz görgetünk, a háttér megváltozik, amit kétféleképpen érhetünk el. Újra végrehajtjuk az 1-es pontot, ez azonban nem szerencsés, mert nagyon lassú. A másik megoldás, hogy eltoljuk a háttér pontjait a *rep movsw* utasítások segítségével, ami már kellően gyors, majd a felszabadult helyre, ami legfeljebb néhány sor lehet a háttér egyik szélén, visszafejtjük a térkép adataiból az oda tartozó BOX-részleteket.

Mi történik, ha a térkép elmozdul egy egységnyit, például felfele? A legfelső sor eltűnik, minden további sor eggyel felugrik, az utolsó sorba pedig az alulról beúszó sor kerül, amit a térkép és a BOX-ok adatai, valamint a koordináták alapján nehéz és időigényes visszafejteni. Viszont ez még mindig gyorsabb, mintha az egész hátteret újrarajzolnánk.

Az eljárásból következik, hogy ez a scrollozás valóban scrollozás, nem jeleníthetjük meg tetszőlegesen egymás után a térkép bármely 320×200-as részletét, azaz megtehetjük, de az mindenképp lassú lesz. A háttér két lépés között csupán néhány pixelnyit mozdulhat el, annak ellenére, hogy valójában egy képernyőnyit is arrébb tolódhat. A szerencse az, hogy ez a néhány képpontnyi elmozdulás bőven elegendő a gördítéshez, mert az nem is lenne scrollozás, ha összevissza ugrálna a térképen a háttér. Persze erre is lesz lehetőség, így kombinálni lehet a kétféle háttértípust: azt, amelyik több, egy képernyőnyi (320×200) képből áll, és azt, amelyik egy globális kép. Meg lehet valósítani például azt, hogy ha a játék főhőse a konyhában tartózkodik, ami egy képernyő nagyságú, kilépve az előszobába már annak csak egy részlete látszik, és ha arrébb megy, akkor az előszoba gördül.

Tehát a két legfontosabb és legnehezebb feladat: egy sor és egy oszlop visszafejtése a térkép- és a BOX-adatok alapján. Az már ehhez képest nagyon egyszerű lesz, hogy a háttér változatlan részét odébb toljuk. Ehhez egy jól megírt ciklus, a ciklusban a *rep movsw* utasítások kellenek.

4.4. Sor megjelenítése

A függőleges görgetés alapja a térkép bármely sorának **320 egymás melletti pontjának visszafejtése**. Ez a gördítés módszeréből következik, ami nagyjából a következő: a háttér minden egyes sorát eggyel feljebb csúsztatjuk (a legfelső sor eltűnik), az utolsó sorába pedig kirakjuk a térképnek az odaillő 320 pontját (ekkor a gördítés iránya: fel). A feladat első része egyszerűbb, sima *movsw* utasítással könnyen megoldható, egyszerűen csak arrébb kell mozgatni a bájtokat a háttéren.

A sorvisszafejtés már egy kicsit nehezebb, éppen ezért három lépésben fogjuk megoldani:

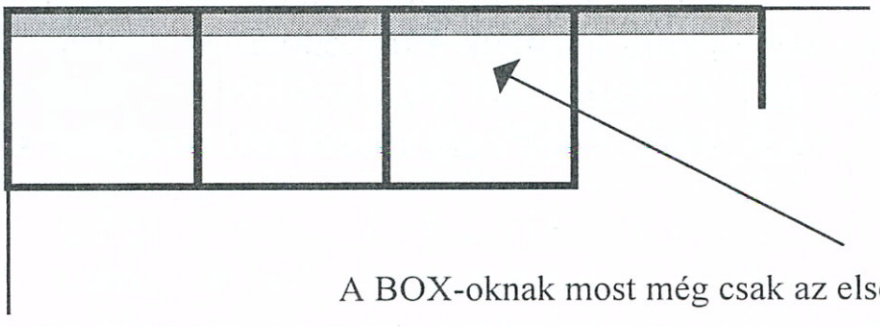
1. Legfelső sor megjelenítése, úgy, hogy a térkép koordinátái: (0;0).
2. Bármely (0-199.) sor megjelenítése, a térkép koordinátái még mindig (0;0), vagyis a háttér a térkép bal felső sarkában van.
3. Végleges megjelenítés, tetszőleges helyzetű háttér, tetszőleges sor.

4.4.1. Legfelső sor

Eleinte tegyük fel, hogy a térkép szélessége 320, ekkor egy sorban található BOX-ok száma 320/8=40 (8 a BOX szélessége). Ekkor a háttér széle BOX-határra esik, ami valamiképp megkönnyíti egy-egy sor megrajzolását. Ezenkívül könnyítésül legyenek a háttér koordinátái (0;0), azaz éppen a térkép bal felső sarkában foglaljon helyet, és a legfelső sor visszafejtéséről kelljen csak gondoskodni. Teendőnk ekkor nagyon egyszerű:

1. Beolvassuk a térkép legelső bájtját, ez megadja a bal felső sarokba kerülő BOX sorszámát. Legyen ez **n**. (ami valójában a **n+1**., mert 0-val kezdődik a számozás).
2. Meghatározzuk az **n**. BOX kezdőcímét: a *BoxPTR* a 0. BOX kezdőcíme, és a 256 BOX egymás után, növekvő sorrendben helyezkedik el. Egy BOX hossza 8×8=64 bájt, így az **n**. BOX kezdőcíme a *BoxPTR*-nél (amit tudunk) n×64-gyel nagyobb.
3. Az **n**. BOX legfelső sorát, vagyis az első 8 bájtját kirajzoljuk a háttérre, a bal felső sarokból indulva, jobbra, a *rep movsw* utasítások segítségével.

Ezek után megismételjük még 39-szer ezt a műveletet, csak az 1. lépésben nem az első, hanem a második, harmadik, ..., negyvenedik térképbájtot olvassuk be. Ez nem olyan nehéz, igaz, hogy az alapelv elsajátításán kívül semmire sem lehet használni.



A BOX-oknak most még csak az első sorát jelenítjük meg.

```
procedure HLine; assembler; asm { H=Horizontal Line, vízsz. sor   }
   push   bp             { BP is a DS-t tárolja, hogy ne kelljen  }
   mov    bp,ds          { a verembe menteni, mert az sok idő     }
   xor    cx,cx          { CX számlálja majd a kirakott BOX-sorokat}
@1:
   push   cx             { Itt a CX a REP utasításhoz kell        }
   les    di,mapptr      { MAPPTR a térkép kezdőcíme              }
   add    di,cx          { DI a térkép CX. bájtjára mutat         }
   mov    al,es:[di]     { AL a térkép aktuális bájtja            }
   xor    ah,ah          { A szorzás miatt az AX felső bájtja 0   }
   mov    bx,64          { Egy BOX mérete 64 bájt                 }
   mul    bx             { AX a kirakandó BOX eltolási címe       }
   les    di,background  { A célcím a háttér egy bájtja           }
   lds    si,boxptr      { A forráscím a BOXPTR egy bájtja        }
   add    si,ax          { DS:[SI] már a kirakandó BOX-ra mutat   }
   mov    ax,8           { DI célindexet is beállítjuk, hogy a    }
   mul    cx             { háttér kellő bájtjára mutasson         }
   add    di,ax          { DI:= DI+CX*8 (CX a számláló)           }
   mov    cx,4           { 8 bájt=4 szó mozgatása következik      }
   cld
   rep    movsw          { Egy BOX egy sorának megjelenítése      }
   mov    ds,bp          { Az eredeti adatszegmens visszaállítása }
   pop    cx             { CX a BOX-okat számlálja                }
   inc    cx             { Számláló növelése                      }
   cmp    cx,40          { Elérte-e a 40-et? (Ennyit kell kirakni) }
   jnz    @1             { Ha még nem, megismételjük az egészet   }
   pop    bp
   end;
```

Az eljárásban szereplő változók (amelyek típusa pointer):

MapPTR	Térkép kezdőcíme.
BoxPTR	BOX grafikus adatok kezdőcíme.
BackGround	Háttér kezdőcíme.

4.4.2. Bármely sor

Az itt szereplő eljárásban osztani fogunk. Ehhez azonban nem a *div*, hanem az *shr* utasítást használjuk. Ezt azért tehetjük meg, mert 8 lesz az osztó. A bittolás sokkal gyorsabb, mint az osztás vagy a szorzás, és rövidebb, igaz, hogy sebességben különbséget nem lehet észrevenni, de a tudat megnyugtathat, hogy programunk logikusabb, gyorsabb és néhány bájttal rövidebb.

8-cal való osztásra akkor lesz szükség, amikor a háttér koordinátarendszeréből, aminek egysége a pixel, áttérünk a térkép koordinátarendszerébe, aminek egysége egy BOX-oldal, 8 pixel. De hogy jobban megért-

sük, nézzük az eljárás lépéseit! A bemeneti paraméter (L) a megjelenítendő sor száma, [0..199] közé esik.

1. L-ből ki kell számolni, a térkép hányadik bájtjától kezdődik a megjelenítés. Az erre szolgáló képlet: első bájt = **(L SHR 3)×40**, ahol 40 az egy sorban lévő BOX-ok száma.
2. Azt, hogy egy BOX-on belül melyik sort kell a háttérre másolni, szintén az L-ből kapjuk meg, csak nem az osztás egész részét, hanem a maradékát vizsgáljuk. Hogy gyorsabb legyen, az osztás utáni maradék kiszámításához az *and* utasítást alkalmazzuk: BOX ábrázolandó sora = **L AND 7**. Ezzel kimaszkoljuk az alsó három bitet.
3. Most már ugyanazt kell tenni, mint az előző eljárásnál, csak a BOX-oknak nem az 1., hanem a 2. lépésben kiszámított sorát kell megjeleníteni.

Mivel a térkép koordinátái még mindig (0;0), a háttér széle megegyezik az első és az utolsó oszlopban elhelyezkedő BOX-ok bal, illetve jobb szélével. Ez megkönnyíti a munkánkat, viszont emiatt ez az eljárás szintén nem használható.

```
procedure HLine( L: word); assembler; asm
                          { A térkép L. sorának megjelenítése       }
   mov    ax,L            { Először kiszámoljuk az első bájt címét  }
   mov    cx,ax           { CX-be is bevisszük                      }
   shr    ax,3            { 8-cal való osztás=3 bit forgatás jobbra }
   mov    bx,40           { 40 BOX-ból áll egy sor                  }
   mul    bx              { Az első téképbájt eltolási címe AX-ben  }
   mov    word [@A],ax    { Eltároljuk, ne kelljen újra kiszámolni  }
   mov    ax,320          { Az első pixel címe: 320×L               }
   mul    cx
   mov    word [@B],ax    { Ezt is tároljuk, időt spórolunk vele    }
   and    cx,7            { Alsó 3 bit marad, többi 0               }
   shl    cx,3            { 8-cal szorzunk, CX: BOX-on belüli elto- }
                          { lási cím                                }
   mov    word [@C],cx    { Ezt is megjegyezzük                     }
   push   bp              { A BP regiszterre nincs szükség, így az  }
   mov    bp,ds           { tárolhatja az adatszegmens regisztert   }
   xor    cx,cx           { CX számlálja a bájtokat, 8-asával       }
@1:
   push   cx              { A sorkirakásnál pixelszámláló lesz      }
   lds    si,mapptr       { MAPPTR a térkép kezdőcíme               }
   add    si,word [@A]    { A megjelenítendő sor első bájtja        }
   add    si,cx           { Most már az aktuális (CX.) bájt         }
   lodsb                  { AL a térkép aktuális bájtja             }
   xor    ah,ah           { AX felső bájtja 0                       }
   shl    ax,6            { AX SHL 6 = AX×64, csak gyorsabb         }
   mov    ds,bp           { DS ismét az eredeti adatszegmens        }
   les    di,background   { A célcím a háttér egy bájtja           }
   add    di,word [@B]    { Növeljük a célcímet, a sor első bájtja  }
```

```
   shl     cx,3            { Még növeljük, hogy a kívánt bájtra mu- }
   add     di,cx           { tasson (DI:=DI+320×L+CX×8)             }
   lds     si,boxptr       { A forráscím a BOXPTR egy bájtja        }
   add     si,ax           { DS:[SI] már a kirakandó BOX-ra mutat   }
   add     si,word [@C]    { És már a megfelelő sorának 1. bájtjára }
   mov     cx,4            { 8 bájt=4 szó mozgatása következik      }
   cld
   rep     movsw           { Egy BOX egy sorának megjelenítése      }
   mov     ds,bp           { Az eredeti adatszegmens visszaállítása }
   pop     cx              { CX a BOX-okat számlálja                }
   inc     cx              { Számláló növelése                      }
   cmp     cx,40           { Elérte-e a 40-et? (Ennyit kell kirakni) }
   jnz     @1              { Ha még nem, megismételjük az egészet   }
   pop     bp
   jmp     @exit
@a:dw      0               { Néhány változót a kódszegmensben helye- }
@b:dw      0               { zünk el, így értékük kiolvasásához nem  }
@c:dw      0               { kell az eredeti adatszegmens           }
@exit:
   end;
```

A *MapPTR*, *BoxPTR*, *BackGround* változók ugyanazt a célt szolgálják, mint az előző eljárásban, talán csak az eljárás végén, a kódszegmensben elhelyezett szó hosszúságú változók szorulnak némi magyarázatra. Ezekre azért van szükség, hogy a főcikluson belül, ami sokszor (negyvenszer) ismétlődik, ne kelljen mindig újra kiszámolni az általuk tárolt értéket.

@A Az **(L SHR 3)×40** értéket tárolja, azt, amit a *MapPTR*-hez kell adni, hogy a kívánt sor kezdőcímét megkapjuk.

@B Értéke **320×L**, ennyit mindenképp hozzá kell adni a célcímhez, hogy ne a háttér első, hanem az L. sorába kerüljön a visszafejtett térképrészlet.

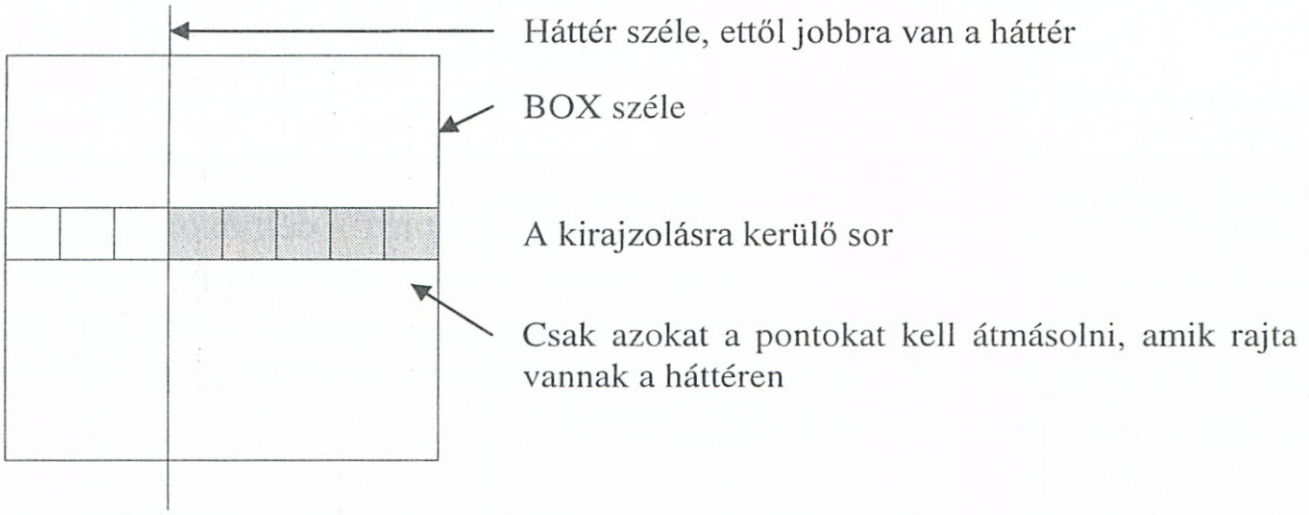
@C **L AND 7**, ezt a sort kell minden egyes BOX-on belül megjeleníteni.

Viszont ezen kódszegmensben lévő változók használata szintén azt eredményezi, hogy a program védett üzemmódban nem futtatható.

Ez az eljárás a gördítéshez még nem használható fel, hiszen a térképnek csak a bal felső sarkában elhelyezkedő 40×25 BOX ábrázolható vele. Továbbá a háttér csak függőleges mozgatásánál sem alkalmazható, mert a háttér második koordinátája nem változtatható, értéke nulla. De a következő részben megismerkedhetünk a végleges *HLine* eljárással, ami a függőleges görgetés alapja lesz.

4.4.3. Tetszőleges helyzetű térkép egy sora

Ha a háttér széle a térképen nem BOX-határra esik, nem lehet az előző két módszert alkalmazni, mert a két szélső BOX-nak nem szabad teljes szélességben egyik sorát megjeleníteni. Legyen a háttér helyzete (3;0), és nézzük meg, mit kell tenni például a legfelső sor kirajzolásánál. Összesen 41 BOX látszik, de az első és az utolsó csak részben, így az első BOX első sorának csak az utolsó 5 pixelét, az utolsó BOX első sorának pedig csak az első 3 pontját kell és szabad megjeleníteni. A következő ábra szemléletesen mutatja be egy háttérhatárra eső BOX egy sorának ábrázolását.



Az első és az utolsó BOX-ot kivéve ugyanúgy kell eljárni, mint az előző rutinnál. Ha a háttér ordinátája a térkép koordináta-rendszerében **Y**, és a háttér **L**. sorát szeretnénk megjeleníteni, akkor a BOX-ok **(Y+L) AND 7**. sorát kell teljes szélességben a háttérre másolni. Azt, hogy az első BOX első hány pontját nem kell kirajzolni, a háttér abszcisszájának 8-cal való osztás utáni maradéka adja, és ennyi pontot kell az utolsó BOX-ból megjeleníteni. Itt osztás és maradékszámítás helyett szintén az *shr,* ill. *and* műveletet alkalmazzuk, ez gyorsabb és rövidebb. Az eljáráshoz szükséges képletek:

A:	Első BOX címe	((Y+L) SHR 3)×BPL+(X SHR 3)
B:	Célcím eltolása a háttér kezdőcíméhez képest	L×320
C:	A BOX-okon belüli eltolási cím	((Y+L) AND 7) SHL 3
D:	Első BOX nem látszó pontjainak száma	X AND 7
E:	Első BOX háttérre kerülő pontjainak száma	8-(X AND 7)

A képletekben szereplő változók:

X,Y	A háttér koordinátái a térképen
L	A megjelenítendő sor, 0≤L≤199
BPL	Egy térképsorban lévő BOX-ok száma

Az eljárás első felében számítjuk ki ezeket az értékeket, amelyeket a végén, a kódszegmensben tárolunk. Ez szintén azt eredményezi, hogy az eljárás védett üzemmódban nem működne. Az eljárás második felében pedig a megjelenítés következik, ami három részre tagolódik, azért, mert a háttér lehet bármilyen helyzetű:

1. Első BOX valamely sorának megjelenítése, arra kell ügyelni, hogy ha a háttér széle nem BOX-határra esik, akkor nem szabad az első néhány pontját kirajzolni.
2. A középső 39 BOX valahányadik sorának átmásolása a háttérre, ugyanúgy, mint az előző eljárásban tettük, teljes szélességben (8 pixel).
3. Utolsó BOX. A kirajzolandó sorból csak annyi pontot szabad ábrázolni, amennyit az első BOX-nál kihagytunk. Ha egyet sem hagytunk ki, vagyis az első BOX-sort teljes szélességében kirajzoltuk, a BOX-ok függőleges szélei háttérhatárra esnek. Ekkor nem jelenik meg az utolsó BOX-nak egyetlen pontja sem, mivel az már nem része a háttérnek. Ezt külön meg kell vizsgálni.

A **HLINE.PAS** példaprogram, ami a szóban forgó eljárást tartalmazza, létrehoz egy 41×26 BOX méretű térképet (ami szélességében és hosszúságában 8 pixellel szélesebb a képernyőnél), majd véletlenszerűen értéket ad bájtjainak. A BOX-ok egyszínűek, így könnyen meg tudjuk különböztetni őket. A háttér koordinátái a térképen (4;4), ami azt jelenti, hogy a kép szélén csak olyan BOX-okat láthatunk, amelyeknek csak a fele látszik (a kép sarkaiban pedig csak a negyede). Ez a példaprogram segíthet megérteni, miért problémás a tetszőleges helyzetű térkép megjelenítése, mert látszik, hogy egy-egy sor szélén a BOX-oknak nem kell az összes pixelét ábrázolni.

```
{hline.pas}

var
 Background: pointer;    { Háttér kezdőcíme                    }
 MAPPTR:     pointer;    { Térkép kezdőcíme                    }
 BOXPTR:     pointer;    { BOX-adatok kezdőcíme                }
 X, Y:       word;       { Háttér koordinátái a térképen       }
 BPL:        word;       { Egy sorban lévő BOX-ok száma        }
 i,j:        word;       { Változók a FOR... ciklusokhoz       }
```

```
procedure HLine( L: word); assembler; asm
                          { A háttér L. sora alatti térkép részlet  }
                          { visszafejtése                           }
 { 1. Címek számítása }

   mov     ax,L           { A = ((Y+L) SHR 3)*BPL+(X SHR 3)         }
   add     ax,Y
   mov     bx,ax          { Az Y+L értéket eltároljuk (helyet és i- }
   shr     ax,3           { dőt takarítunk meg vele)                }
   mul     BPL            { A térkép szélessége már bármennyi lehet }
   mov     cx,X           { (csak 40-nél ne legyen kisebb)          }
   shr     cx,3
   add     ax,cx
   mov     word [@A],ax   { @A: első térképbájt címe                }

   mov     ax,320         { B = L×320                               }
   mul     L
   mov     word [@B],ax   { @B: első háttérbájt címe                }

   and     bx,7           { C = ((Y+L) AND 7) SHL 3                 }
   shl     bx,3           { Y+L értéket a BX-ben tároltuk           }
   mov     word [@C],bx   { @C: első bájt egy BOX-on belül          }

   mov     ax,X           { D = X AND 7                             }
   and     ax,7
   mov     word [@D],ax   { @D: első BOX nem látszó pontjainak száma}

   mov     bx,8           { E = 8-D                                 }
   sub     bx,ax
   mov     word [@E],bx   { @E: első BOX látható pontjainak száma   }

 { 2. Sor visszafejtése, a háttérre }

 { 2.1. Első BOX }
   push    bp             { BP-re nincs szükség,                    }
   mov     bp,ds          { az adatszegmenst tárolja                }
   lds     si,mapptr      { DS:[SI] a térkép kezdőcíme              }
   add     si,word [@A]   { Az első visszafejtendő bájt címe        }
   lodsb                  { AL az első BOX sorszámát tartalmazza    }
   xor     ah,ah          { AH nullázása, a forgatás miatt          }
   shl     ax,6           { 64-gyel szorzunk, AX a BOX eltolási címe}
   mov     ds,bp          { Vissza kell tölteni az eredeti DS-t,    }
   les     di,background  { mert a háttér kezdőcíme abban van       }
   add     di,word [@B]   { ES:[DI] a célcím, a kiválasztott sor    }
                          { első bájtjára mutat                     }
   lds     si,boxptr      { A forráscím a BOXPTR néhány bájtja      }
   add     si,ax          { A kirakandó BOX első bájtja             }
   add     si,word [@C]   { A BOX ábrázolandó sorának első bájtja   }
   add     si,word [@D]   { Az első megjelenítendő pont címe        }
   mov     cx,word [@E]   { 8-D db pontot kell kirajzolni           }
   cld
   rep     movsb          { Rajzolás bájtonként, 'E' lehet páratlan }
   mov     ds,bp
```

```
 { 2.2. Középső 39 BOX }
   mov     bx,1            { BX számlálja a BOX-okat, az 1. BOX [@C].}
@1:                        { sora már ki van rakva a háttérre        }
   lds     si,mapptr       { MAPPTR a térkép kezdőcíme               }
   add     si,word [@A]    { A megjelenítendő sor második bájtja     }
   add     si,bx           { Most már az aktuális (CX.) bájt         }
   lodsb                   { AL a térkép aktuális bájtja             }
   xor     ah,ah           { AX felső bájtja 0                       }
   shl     ax,6            { AX SHL 6 = AX×64, csak rövidebb         }
   mov     ds,bp           { DS ismét az eredeti adatszegmens        }
   lds     si,boxptr       { A forráscím a BOXPTR egy bájtja         }
   add     si,ax           { DS:[SI] már a kirakandó BOX-ra mutat    }
   add     si,word [@C]    { És már a megfelelő sorának 1. bájtjára  }
   mov     cx,4            { 8 bájt=4 szó mozgatása következik       }
   rep     movsw           { Egy BOX egy sorának megjelenitése       }
   mov     ds,bp           { Az eredeti adatszegmens visszaállitása  }
   inc     bx              { Számláló növelése                       }
   cmp     bx,40           { Ha még nem érte el a 40-et,             }
   jnz     @1              { megismételjük az ciklust                }

 { 2.3. utolsó BOX }
   lds     si,mapptr       { MAPPTR a térkép kezdőcíme               }
   add     si,word [@A]    { Ehhez hozzáadunk annyit, hogy a most    }
   add     si,40           { megjelenítendő BOX-ra mutasson          }
   lodsb                   { AL-be tölti a BOX sorszámát             }
   xor     ah,ah
   shl     ax,6            { AX=64×AL, AX a BOXPTR eltolása          }
   mov     ds,bp
   lds     si,boxptr
   add     si,ax
   add     si,word [@C]    { Most annyi pont látszik, amennyi az el- }
                           { ső BOX-nál kilógott, és ennyit kell     }
   mov     cx,word [@D]    { csak kirajzolni                         }
   cmp     cx,0            { Ha ez nulla, nincs semmi dolgunk        }
   jz      @2
   rep     movsb           { Utolsó BOX néhány pontjának kirakása    }
@2:
   mov     ds,bp           { Sohase felejtsük el visszaállítani a DS }
   pop     bp              { értékét, ha megváltoztattuk             }
   jmp     @exit

@A:dw      0               { Ezek a változók azért vannak a kódszeg- }
@B:dw      0               { mensben, hogy elérésükhöz ne kelljen az }
@C:dw      0               { eredeti adatszegmenst visszaírni.       }
@D:dw      0
@E:dw      0               { A változók leírását lásd feljebb        }
@exit:
   end;

begin
 getmem( BackGround, 64000); { A háttér marad 320×200-as             }
 getmem( MAPPTR, 41*26); { A térkép (41×8)×(26×8)=328×208 pixel      }
 getmem( BOXPTR, 16384); { 256 db 8×8-as BOX=256×8×8=16384 bájt      }
 randomize;              { Véletlenszám-generátor inicializálása     }
```

```
for i:= 0 to 1065 do    { Térkép véletlenszerű feltöltése          }
 mem[seg(MAPPTR^):ofs(MAPPTR^)+i]:= random( 256);
for i:= 0 to 255 do     { BOX-adatok feltöltése, minden BOX egy-   }
 for j:= 0 to 63 do     { színű, színe sorszáma=saját sorszáma     }
  mem[seg(BOXPTR^):ofs(BOXPTR^)+i*64+j]:=i;
X:=4; Y:=4; BPL:=41;    { A térkép koordinátái: (4;4), egy sorban  }
                        { 41 BOX található                         }
asm
  mov    ax,13h
  int    10h            { MCGA üzemmód bekapcsolása                }
 end;
for i:= 0 to 199 do     { Sorok megrajzolása, 200 db van           }
 HLine( i);             { Most még csak a háttéren látszik, ezért  }
asm                     { a hátteret a képernyőre kell másolni     }
  push   ds             { DS lesz a forrásszegmens (háttér)        }
  mov    es,sega000     { A célszegmens a képmemória szegmense     }
  xor    di,di          { A célindex nulla                         }
  lds    si,background  { A forráscím a háttér kezdőcíme           }
  cld                   { Növekvő adatirány                        }
  mov    cx,32000       { 32000 szó másolása                       }
  rep    movsw          { Háttér megjelenítése                     }
  pop    ds
  end;
readln;                 { Enter megnyomására kilép                 }
asm
  mov ax,3
  int 10h
 end;                   { Visszatérés a szöveges módhoz            }
end.
```

A függőleges irányú gördítés alapeljárásával tisztában vagyunk, most már csak a térképet az ordinátatengely irányában elmozgató eljárás másik felével kell megismerkedni, ami a háttér pixeleit arrébb mozgatja. Ezt majd a vízszintes görgetéssel együtt, a fejezet végén a *ScrollR, ScrollL* eljárásokban valósítjuk meg (4.6.1-2.), mert előbb a vízszintes mozgatás problémáját is meg kell oldani.

4.5. Oszlop megjelenítése

Mint ahogy a függőleges gördítésnek az alapja a térkép egy sorának visszafejtése, a térkép vízszintes irányú elmozdításához először az oszlop kirajzolásának módjával kell megismerkedni. Az alapelv ugyanaz, ezért itt nem bontjuk fel a megoldást lépésekre, mindjárt a legösszetettebbet írjuk le, amit a *ScrollU, ScrollD* eljárásokban is alkalmazunk (4.6.3-4. fejezet). Oszlop visszafejtése az, amikor a háttér tetszőleges oszlopa alatti térképmintázatot kell a háttérre kiraj-

zolni. Ez összesen 200 bájt átmásolását jelenti, ennyi pixel ugyanis a háttér magassága.

Maga a gördítés két részből áll. Legyen a gördítés iránya: jobb, vagyis a térkép a háttér alatt balra mozdul el. Ekkor először a háttér minden egyes sorának 2-320. bájtját átmásoljuk *rep movsb* utasításokkal az 1-319. bájtba. A sorrendjük közben nem változik, ezért úgy tűnik, mintha az egész háttér egy pixellel balra lenne az előző helyzetéhez képest. Ezután meghívjuk a *VLine* eljárást, ami visszafejti a háttér jobb legszélső oszlopára az oda illő térképrészletet. A gördítés első fele könnyű, a fejezet végén mind a négy irányhoz megtalálható a hozzájuk tartozó memóriamozgató rutin.

Mint a sormegjelenítő eljárás, ez is két részből áll. Először kiszámítjuk a koordináták és egyéb adatok alapján a szükséges címeket, amiket szintén a kódszegmensben tárolunk az eljárás végén, majd az eljárás második felében kirajzoljuk az oszlopot a háttérre. Ez egy kicsit bonyolultabb, mint a sormegjelenítésnél, mert nem lehet sorfolytonosan írni a *movsw* segítségével, minden egyes pixel átírása után növelni kell mind a forrás-, mind a célindexet annyival, hogy a pontok egymás alatt, függőlegesen helyezkedjenek el. Éppen ezért lassabb is lesz, de a különbséget nem lehet majd észrevenni, mert a háttér oszlopai rövidebbek, mint a sorai.

Természetesen az eljárás első fele, a címek kiszámítása is más, a következő táblázat tartalmazza az ehhez szükséges képleteket. Az első oszlopban található betű lesz majd az eljáráson belül az azonosítója, így az eljárást vizsgálva könnyen összepárosíthatjuk az egybetűs változókat (pl. @A) és a képletet, amivel az általuk tárolt értéket meghatározzuk.

A:	Első BOX címe	(Y SHR 3)×BPL+((X+C) SHR 3)
B:	Célcím eltolása a háttér kezdőcíméhez képest	C (ez nem szerepel majd az eljárásban)
C:	A BOX-okon belüli eltolási cím	(X+C) AND 7
D:	Első BOX nem látszó pontjainak száma	Y AND 7
E:	Első BOX háttérre kerülő pontjainak száma	8-(Y AND 7)

A képletekben szereplő változók jelentése:

X,Y	A háttér koordinátái a térképen
C	A megjelenítendő oszlop (column), 0≤C≤319
BPL	Egy térképsorban lévő BOX-ok száma (BOXs per line)

Ha összevetjük a fenti táblázatot a 4.4.3. rész elején találhatóval, észrevehetjük, hogy tulajdonképp ugyanazt tartalmazza, csak a képletekben történt változás, leggyakrabban X helyett Y szerepel (és fordítva).

Az eljárás második fele most is három részből áll: első BOX egy oszlopának részleges kirajzolása, középső 24 BOX egy oszlopának teljes kirajzolása, utolsó BOX-oszlop legfelső néhány pixelének átmásolása. Természetesen itt is átugorjuk a 3. lépést, ha a háttér felső széle két BOX határára esik.

A **VLINE.PAS** program, amiben az oszlopvisszafejtő eljárás helyet kapott, ugyanazt látja el, amit a **HLINE.PAS**, csak a térképrészletet a háttérre nem soronként, hanem oszloponként rajzolja ki. Itt csak a programban felhasznált *VLine* eljárást ismertetjük, ezt kell beszúrni a **HLINE.PAS** program elejére, és át kell írni a `HLine(i);` utasítást `Vline(i);`-re, hogy megkapjuk a **VLINE.PAS** példaprogramot (ami a lemezmellékleten természetesen megtalálható).

```
procedure VLine( C: word); assembler; asm
                          { A háttér C. oszlopa alatti térképrész- }
                          { let visszafejtése a háttérre           }
 { 1. Képletek alkalmazása, @A-E változóknak értékadás }

   mov     ax,Y           { A = (Y SHR 3)×BPL+((X+C) SHR 3)        }
   shr     ax,3
   mul     BPL            { AX-ben az sor első bájtjának címe      }
   mov     bx,X           { A térkép első koordinátája             }
   add     bx,C           { BX tartalma: X+C, ezt az összeget C-nél }
   mov     cx,bx          { még felhasználjuk, ezért tároljuk      }
   shr     bx,3
   add     ax,bx
   mov     word [@A],ax   { @A: első BOX eltolási címe             }

   and     cx,7           { C = (X+C) AND 7, CX-ben volt az X+C    }
   mov     word [@C],cx   { @C: BOX-okon belüli kirajzolandó oszlop }

   mov     ax,Y           { D = Y AND 7                            }
   and     ax,7           { BX tartalma most: Y AND 7              }
   mov     word [@D],ax   { @D: felső BOX nem látszó sorainak száma }

   neg     ax             { E = 8-(Y AND 7)                        }
   add     ax,8           { AX értéke: -(Y AND 7)+8                }
   mov     word [@E],ax   { @E: alsó BOX nem látható sorainak száma }
```

```
{ 2. Oszlop megjelenítése }

{ 2.1. Legfelső BOB [@C]. oszlopa [@E] db pontjának átmásolása }
   les     di,background { A célcímet már most beállítjuk          }
   add     di,C          { az első módosítandó háttér-bájtra       }
   push    bp            { BP ebben az eljárásban csak az          }
   mov     bp,ds         { adatszegmenscímet tárolja               }
   lds     si,mapptr     { MAPPTR a térkép kezdőcíme               }
   add     si,word [@A]  { DS:[SI] az első érintett BOX-ra mutat   }
   lodsb                 { AL az első BOX sorszáma, ebből          }
   xor     ah,ah         { számítjuk ki az eltolási címét,         }
   shl     ax,6          { 64-gyel való szorzással                 }
   mov     ds,bp         { A köv. művelethez az eredeti DS kell    }
   lds     si,boxptr     { BOXPTR a BOX-adatok kezdőcíme            }
   add     si,ax         { A kirajzolandó BOX kezdőcíme            }
   add     si,word [@C]  { Kirajzolandó oszlop                     }
   mov     ax,8          { Egy BOX szélessége 8 bájtnyi            }
   mul     word [@D]     { [@D]×8, ennyit kell adni SI-hez, hogy   }
   add     si,ax         { a felső [@D] sor ne látszódjon          }
   mov     cx,word [@E]  { [@E] db bájt átmásolása következik      }
   cld                   { Az irány: növekvő                       }
@1:                      { Most sajnos nem használható a REP       }
   movsb                 { Egy bájt átírása                        }
   add     si,7          { Forráscím növelése 1+7-tel              }
   add     di,319        { Célcím a következő sor kellő pontja     }
   loop    @1            { CX-szer van ismétlés                    }
   mov     ds,bp         { DS vissza                               }

{ 2.2. Középső 24 db BOX [@C]. oszlopának átrajzolása }
   mov     bx,1          { Az első BOX-szal már nincs dolgunk      }
@2:
   mov     ax,BPL        { A szorzáshoz kell majd                  }
   lds     si,mapptr     { DS:[SI] a térképre mutat                }
   add     si,word [@A]  { Az első megjelenítendő BOX bájtjára     }
   mul     bx            { Szorzás, AX-ben előállt a BX. BOX       }
   add     si,ax         { eltolási címe                           }
   lodsb                 { AL-be töltjük az érintett BOX számát    }
   mov     ds,bp         { A régi DS visszatöltése                 }
   xor     ah,ah         { AH nullázása a forgatáshoz              }
   shl     ax,6          { Szorzás, AX:= AX×64 (egy BOX hossza)    }
   lds     si,boxptr     { A forráscím a BOXPTR egy bájtja         }
   add     si,ax         { Az érintett BOX első bájtja DS:[SI]^    }
   add     si,word [@C]  { Most már az első átrajzolandó pontja    }
   mov     cx,8          { 8 bájt átmásolása, most nem lehet       }
@3:                      { szavanként, mert nem egybefüggők        }
   movsb                 { Egy bájt átmásolása                     }
   add     di,319        { Célindex növelése                       }
   add     si,7          { Forrásindex növelése                    }
   loop    @3            { Amíg CX 0-ra nem csökken                }
   mov     ds,bp         { DS vissza                               }
   inc     bx            { Egy BOX-oszlopot átrajzoltunk           }
   cmp     bx,25         { Összesen 25 középső BOX van             }
   jnz     @2            { Kiírjuk azokat is                       }
```

```
  { 2.3. Az utolsó BOX-oszlop megrajzolása, ha látszik }
    mov     cx,word [@D]   { Alsó BOX látható sorai, akkor nulla,    }
    cmp     cx,0           { ha a háttér széle BOX-határra esik,     }
    jz      @exit          { ekkor nincs több dolgunk                }
    mov     ax,BPL         { Előre beírjuk BPL-t a szorzáshoz, DS    }
                           { módosítása után már nem tudjuk          }
    lds     si,mapptr      { Térkép kezdőcíme                        }
    add     si,word [@A]   { Háttér C. sora alatti felső bájt címe   }
    mov     bx,25          { 26. BOX egy oszlopának átmásolása       }
    mul     bx             { AX-ben előállt az eltolási cím          }
    add     si,ax          { El is toljuk az SI regisztert           }
    lodsb                  { Megvan a BOX sorszáma                   }
    mov     ds,bp          { DS ismét az eredeti adatszegmens        }
    lds     si,boxptr      { A forráscím beállítása                  }
    xor     ah,ah          { AX felső bájtja nulla                   }
    shl     ax,6
    add     ax,word [@C]   { @C a BOX ábrázolandó oszlopa            }
    add     si,ax          { SI:= SI+64×AX+[@C]                      }
@4:
    movsb                  { Egy BOX-pont másolása a háttérre        }
    add     si,7           { Forrás- és célindex növelése,           }
    add     di,319         { a függőleges sorokhoz                   }
    loop    @4             { CX-szer kell ismételni                  }
    mov     ds,bp
    jmp     @exit

@a:dw       0              { A kódszegmensben elhelyezkedő           }
@c:dw       0              { változók                                }
@d:dw       0
@e:dw       0
@exit:
    pop     bp             { BP kivétele a veremből                  }
    end;
```

A 'B' kiszámításának képletét az eljárásban nem alkalmaztuk, mert a C (paraméter) értéke az adatszegmens módosítása után is elérhető, hiszen a vermen keresztül kapja meg az eljárás, és az SS-t nem változtattuk. Vigyázzunk arra, hogy a C és a @C változókat ne keverjük össze, az első a háttér megjelenő sora, a másik pedig egy eljáráson belüli tároló (a BOX-ok kirajzolandó oszlopának száma). Ha a *VLine* eljárást összevetjük „testvérével", a *HLine* eljárással, észrevehetjük a hasonlóságokat, felépítésüknek elve teljesen ugyanaz. Éppen ezért itt nincs értelme részletesebben leírni ezt a szubrutint.

4.6. Gördítés

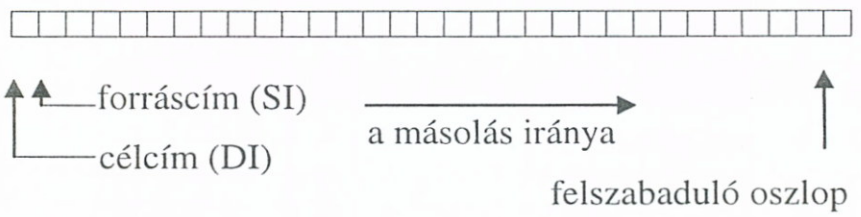
A sor- és oszlopvisszafejtő eljárás önmagában nem sok mindenre alkalmazható, scrollozni velük még nem lehet. Kell még négy olyan eljárás, amely magába foglalja az egyes irányokhoz tartozó háttérmozgatásokat és meghívja a sor- vagy oszlopmegjelenítő eljárást. Természetesen ezeket is assemblyben írjuk, hiszen itt aztán lényeges a sebesség, minden futáskor majdnem 64000 bájt átmozgatásáról van szó, és ennek másodpercenként 25-ször le kell futnia, hogy a térkép mozgását szemünk folyamatosnak érzékelje.

Négy ilyen gördítő eljárást kell tehát írnunk, melyek mindegyike két részből áll:

1. Háttér bájtjainak elmozdítása. A *rep movsw* utasításokkal hajtjuk végre, csak a kezdő cél- és forráscím megadása jelenthet némi nehézséget, egyébként nem bonyolult feladat.
2. A háttér szélén megüresedő részekre kiírjuk az odaillő sort vagy oszlopot. Ha egyszerre több pixelt mozog a háttér, értelemszerűen több sort vagy oszlopot kell visszafejteni.

4.6.1. Jobbra

A háttérbájtok átrakásának alapelve jobban megérthető a következő ábra segítségével. Itt a háttér egy pixelnyit jobbra mozog a térképen, vagyis bájtjai balra gördülnek. Mivel minden sorral ugyanaz történik, elég csak egyet megvizsgálni:



Az indexek kezdőértéke a következő: forrásindex az aktuális sor 2., a célindex az 1. bájtjára mutat. A D irányjelző bitet kioltjuk, így a másolás iránya növekvő lesz. Ez esetben az első bájt átmozgatása a következőképpen történik: [SI]-ből [DI]-be másoljuk, közben mindkét regiszter értéke nő (a D kikapcsolt állapota miatt). Ezt teszi a *movsb* utasítás, a *rep* segítségével pedig mindezt 319-szer végre kell hajtani. (A sebesség növelése érdekében a sorokat nem bájtonként toljuk arrébb a *movsb* segítségével, hanem a *movsw* utasítást alkalmazzuk. Ez

persze összetettebbé teszi a mozgatáshoz szükséges CX regiszter beállítását, mert osztani kell majd kettővel, azaz a biteket eggyel jobbra tolni.)

Az első oszlopba csak írtunk, ezért ami ott volt, eltűnik, az utolsó oszlopból pedig csak olvastunk, így az felszabadul, ide kerülnek majd a *VLine* eljárással az új adatok. Hogy az egész másolás folyamatos legyen, a felszabaduló, utolsó oszlop bájtjaiba a következő sor első bájtját vagy bájtjait írjuk, hiszen teljesen mindegy, hogy ott milyen értékek vannak. Így nem kell minden sor elkészítése után megállni, növelni a forrás- és célindexet, hanem lehet folyamatosan másolni, ami néhány pixeles gördítésnél még gyorsabb, mintha minden sor végén megállnánk. Persze ha például fél háttérnyit mozgatunk, nem jó ez a módszer, mert rengeteg bájtot feleslegesen mozgatunk, viszont a 8 pixeles gördítés is már olyan gyors, hogy nem igazán lesz szükségünk a háttér helyzetének ilyen nagymértékű hirtelen megváltoztatására.

Az eljárás tehát két részből áll, melyekkel már az 4.6. fejezet elején megismerkedhettünk, paraméterében kell megadni a gördítés mértékét (azt, hogy hány darab pixellel mozduljon el).

```
procedure ScrollR( Count: word); assembler; asm
                          { A háttér jobbra mozog, a térkép balra   }
  { 1. Forrás- és célcím beállítása }
    mov     bx,count      { Legelőször megnöveljük a háttér absz-   }
    add     X,bx          { cisszáját                               }
    les     di,background { A cél- és a forrásszegmens egyaránt a   }
    mov     si,di         { háttér szegmense                        }
    add     si,bx         { A forrás a céltól jobbra található      }
    mov     cx,32000      { A háttér 32000 szóból áll               }
    cld                   { Növekvő direkcióban haladunk            }
    rep     seges movsw   { Háttér sorainak eltolása                }

  { 2. Jobboldalt felszabaduló oszlop(ok) kitöltése }
    mov     cx,bx         { BX tartalma még mindig COUNT, ennyi     }
@1:                       { oszlopot kell visszafejteni             }
    push    cx            { Csak a CX értékét kell megjegyezni      }
    mov     ax,320        { Az oszlop sorszámának kiszámítása:      }
    sub     ax,cx         {  oszlop = 320-CX                        }
    push    ax            { Ezt a veremen keresztül adjuk át a      }
    call    vline         { VLINE eljárásnak, aminek futtatása so-  }
    pop     cx            { rán CX megváltozik, ezért visszatöltjük }
    loop    @1            { Ismétlés, amíg CX (oszlopszámláló) > 0  }
    end;
```

Az eljárás paraméteréből látszik, hogy ugyan bármennyit lehet a háttér pozícióján egyszerre változtatni (persze annak méreteinél többet nem), de az elmoz-

dulás csak egész szám lehet. Ez egy olyan játéknál jelenthet problémát, ahol szükséges lenne „igazi", négyzetes gyorsulásra, de a maximális sebesség csak például 2 pixeles gördítés lehet egyszerre. Ekkor, ha a gyorsulás kettőnél több lépésből áll, a hátteret törtszámmal kellene mozdítani. Ez nem megoldhatatlan feladat, írni kell egy új eljárást, ami koordináták alapján görget, és a koordináták lehetnek valós számok, ha kerekít a rutin. Nem megoldhatatlan, de a megvalósítást az Olvasóra bízzuk.

4.6.2. Balra

A balra gördítés majdnem ugyanolyan, mint a jobbra gördítés, csak a memóriamozgatás iránya, és ezért a forrás- és célindexek kezdeti értéke más. Az eljárás második felében a változtatás csak annyi, hogy értelemszerűen nem a háttér jobb, hanem a bal szélére fejtjük vissza az oszlopokat. Ugyanúgy egyszerre mozgatjuk az egész hátteret, így ugyan minden sorban néhány bájtot (annyit, amennyi az eltolás mértéke) fölöslegesen másolunk, viszont maga a művelet folyamatos, ezáltal gyorsabb. Nem is érdemes itt tovább magyarázni, lássuk magát az eljárást!

```
procedure ScrollL( Count: word); assembler; asm
                        { A háttér balra mozgatása                    }
  { 1. Háttér bájtjainak átmozgatása }
    mov     ax,count    { Háttér abszcisszájának csökkentése:         }
    sub     X,ax        {  X:= X-COUNT                                }
    les     di,background { A háttér szegmensén belül dolgozunk       }
    add     di,63999    { DI az utolsó bájtra mutat                   }
    mov     si,di       { Most még SI is, de csökkentjük, mert a      }
    sub     si,ax       { forrás tőle balra található                 }
    mov     cx,32000    { Megint 64000 bájtot mozgatunk,              }
    std                 { de most a másik irányba                     }
    rep     seges movsw { Háttér jobbra csúsztatása                   }

  { 2. Baloldalt felszabadult oszlopok helyére az újakat írjuk }
    mov     cx,ax       { CX számolja a kirakandó oszlopokat          }
@1:                     { (ez AX-ben még benne van)                   }
    push    cx          { A VLINE eljárásban CX megváltozik           }
    dec     cx          { A visszafejtendő oszlop sorszáma CX-1       }
    push    cx          { A paraméterátadás a vermen keresztül        }
    call    vline       { valósul meg                                 }
    pop     cx          { CX ismét az oszlopok számát tartalmazza     }
    loop    @1          { Ha még nem nulla, ismét                     }
    end;
```

4.6.3. Fel

Amikor a térkép lefele mozog, hozzá viszonyítva a háttér pontjai felfele mozdulnak el. A háttér alsó néhány sora eltűnik, a felső pár sorába pedig olyan sorok kerülnek, amik eddig még a háttéren nem voltak jelen. A többi sor pedig lejjebb csúszik.

Az eljárás első része, a memóriamozgatás hasonlóképpen megy végbe, mint az előző két eljárásnál. Az extraszegmens-regiszterbe (ES) betöltjük a háttér szegmensét, az indexregisztereket pedig úgy állítjuk be, hogy a háttér utolsó bájtjára mutassanak. Ezután a forrásindexből kivonunk N×320-at (N a görgetés mértéke), így az feljebb ugrik N sorral. Kezdődhet a másolás, ami most is fordított irányú (ki kell gyújtani az irányjelző bitet az *std* utasítással). Most is szavanként másolunk, de CX értéke nem 32000 lesz, hanem annál N×160-nal kevesebb, mert így nem másolunk fölöslegesen. (A vízszintes gördítésnél nem csökkentettük ezt a 32000-es értéket, így egynél nagyobb mértékű mozgatásnál néhány bájtot szükségtelenül másoltunk át, ez azonban elenyésző. Ha itt nem csökkentenénk ezt az értéket, természetesen semmit sem vennénk észre, nem lenne lassabb, de a tudat megnyugtató, hogy nem dolgoztatjuk gépünk hasztalanul.)

Az eljárás második részében az előzőhöz képest csupán annyi változás lesz, hogy *VLine* helyett *HLine* betűsort írunk.

```
procedure ScrollU( Count: word); assembler; asm
                        { SCROLL-ozás felfelé, a térkép: lefelé   }
  { 1. Minden sor lejjebb csúszik }
    mov     ax,count      { Mindenekelőtt csökkentjük a háttér or-  }
    sub     Y,ax          { dinátáját                               }
    les     di,background { ES:[DI] a háttérre mutat                }
    add     di,63999      { Már annak az utolsó bájtjára            }
    mov     si,di         { Most már SI is                          }
    mov     bx,320        { Kivonás előtt szorzunk, 320×COUNT-tal   }
    mul     bx            { lesz SI értéke kevesebb, így már a kel- }
    sub     si,ax         { lő sorra mutat                          }
    mov     cx,32000      { A háttér 32000 szó, de nem kell ennyit  }
    shr     ax,1          { mozgatni, elég ennél COUNT*160-nal ke-  }
    sub     cx,ax         { sebbet (COUNT×160=COUNT×320 SHR 1)      }
    std                   { Az irány most is csökkenő               }
    rep     seges movsw   { Az ES-en belül történik az adatmásolás  }
    inc     di            { A háttér bal felső sarkában lévő bájtot }
    inc     si            { is lejjebb rakjuk, mert azt eddig ki-   }
    seges   movsb         { hagytuk                                 }
```

```
 { 2. Felső COUNT darab sor megrajzolása }
   mov    cx,count       { AX értéke már megváltozott, ezért itt   }
@1:                       { COUNT-ot írunk helyette                 }
   push   cx
   dec    cx
   push   cx
   call   hline          { Most vízszintesen jelenítjük meg a báj- }
   pop    cx             { tokat                                   }
   loop   @1
   end;
```

Az első részben CX-be 32000,5-et kellene írni ahhoz, hogy a legfelső sort is teljesen lejjebb csúsztassuk. Ha 32000-et írunk, akkor a szavankénti mozgatás miatt a bal felső sarokban lévő bájt helyén a régi adat marad, így az 1. rész végén még egy *movsb*-vel azt is átmásoljuk.

4.6.4. Le

A *ScrollR* (jobbra) és a *ScrollU* (felfelé görgető) eljárás ismeretében a lefelé gördítés annyira egyszerű, hogy magyarázat nélkül, csak magát az eljárást közöljük, azt is csak a teljesség kedvéért.

```
procedure ScrollD( Count: word); assembler; asm
                          { Végül a lefelé gördítés következik      }
 { 1. Sorok feljebb mozgatása a háttéren }
   mov    ax,count       { A háttér (térkép) második koordinátájá- }
   add    Y,ax           { nak növelése                            }
   les    di,background  { A háttéren dolgozunk                    }
   mov    si,di          { SI is felveszi a háttér ofszetcímét     }
   mov    bx,320         { A háttér szélessége 320 pixel           }
   mul    bx             { Ezzel szorozva COUNT-ot megkapjuk,      }
   add    si,ax          { mennyit kell SI-hez adni                }
   mov    cx,32001       { CX a háttér hossza+1 (word-ben)         }
   shr    ax,1           { Ez tulajdonképpen osztás 2-vel          }
   sub    cx,ax          { Csak hogy gyorsabb legyen...            }
   cld                   { Irány: növekvő                          }
   rep    seges movsw    { Sorok felcsúsztatása                    }

 { 2. Az alul megüresedett rész megjelenítése }
   mov    cx,count       { COUNT db. sort kell kiírni              }
@1:
   mov    ax,200         { 200-ból kell kivonni, hogy megkapjuk a  }
   sub    ax,cx          { HLINE-hoz a sor számát                  }
   push   cx             { CX a HLINE során megváltozik, elmentjük }
   push   ax             { AX tartalmazza a rajzolandó sor számát  }
   call   hline          { Egy sor visszafejtése                   }
   pop    cx
   loop   @1             { A többi sor kiírása (ha még van)        }
   end;
```

Az 1. részben a CX regiszter kezdeti értéke 32001, így egy bájttal többet másolunk a kelleténél, viszont most ezt megtehetjük, mert a célindex, ahova írunk, még így is a háttér tartományán belül van. Ezzel megspórolunk egy *movsb* utasítást az első rész végéről.

4.7. Scroll példaprogram, a MAP fájl formátuma

Itt, a háttérrel foglalkozó fejezet végén egy rövid példaprogram által szemlélhetjük meg magát a gördítést. Ez először betölti a lemezről a PELDA.MAP fájlban található térképet és BOX-adatokat. A MAP térképtároló fájlok szerkezete a következő:

Cím	Méret	Tartalom
0	1	Értéke 1, a MAP fájl verzióját jelöli. A továbbfejlesztés miatt szükséges.
1	2	Egy térképsorban elhelyezkedő BOX-szám (BPL), vagyis a térkép pixelben vett szélességének nyolcadrésze.
3	2	A térkép hossza, bájtban (L). Ennyi BOX-ot tartalmazhat. Magasságát ebből és az előző adatból a következőképpen kaphatjuk meg: H=L/BPL. Mivel a térkép téglalap alakú, H csak egész szám lehet.
5	16384	A BOX grafikus adatai. Olyan sorrendben vannak tárolva, ahogy a memóriában, tehát egyszerűen csak be kell olvasni ezeket a bájtokat a *BoxPTR* változóba.
16389	L	Térképadatok. Ezeket is csak egyszerűen be kell olvasni a *MapPTR* címtől kezdve.
ezután	32	A BOX ütközési váza (maszk). A következő fejezetben részletesen megtudhatjuk, mi ez.

Miután az adatok a lemezről a memóriába kerültek, elvégezzük a szükséges beállításokat (pl. MCGA üzemmód bekapcsolása), és belépünk a főciklusba. Itt a nyílbillentyűkkel mozgathatjuk a térképet, és felhasználjuk a 3.1. fejezetben megismert billentyűmegszakítást. Ezt a *Game* egység *InitKey* eljárásával (8.8.) érhetjük el. Ha nem a BIOS megszakítást használjuk, a mozgás sokkal egyenletesebb lesz.

```
{scroll.pas}

uses Game;                  { Most még csak a billentyűzetkezelő ré- }
                            { szét használjuk a GAME egységnek       }
var
 BackGround: pointer;       { A háttér kezdőcíme                     }
 MAPPTR, BOXPTR: pointer;   { Térkép, BOX-adattömb kezdőcíme         }
 X, Y, BPL: word;           { Koordináták, BOX/sor                   }
 i: byte;                   { A FOR... ciklushoz                     }
 f: file;

{$I HLine.inc}              { Ezek az include fájlok a fejezet során }
{$I VLine.inc}              { megismert eljárásokat tartalmazzák,    }
{$I ScrollR.inc}            { mindegyik azt, aminek azonosítója a    }
{$I ScrollL.inc}            { fájlnévvel megegyezik. Például a       }
{$I ScrollU.inc}            { ScrollU.inc fájl a felfelé gördítő el- }
{$I ScrollD.inc}            { járást foglalja magába.                }

begin

 { 1. Adatok beállítása, betöltése }

   getmem( BOXPTR, 16384);     { Helyfoglalások                      }
   getmem( MAPPTR, 80*50);     { A térkép 80×50 BOX                  }
   getmem( BackGround, 64000);
   X:= 0; Y:= 0;               { A háttér kezdőpozíciója             }
   BPL:= 80;                   { A térkép szélessége 80 BOX          }


   assign( f, 'PELDA.MAP');      { Térkép, grafikus adatok be-       }
   reset( f, 1);                 { töltése. A fájl első 5 bájt-      }
   seek( f, 5);                  { ját nyugodtan átugorhatjuk,       }
   blockread( f, BOXPTR^, 16384); { mert az ott tárolt adatokat      }
   blockread( f, MAPPTR^, 80*50); { (pl. BPL) már ismerjük           }
   close( f);

   InitKey;                    { Saját billentyűzetmegszakítás       }
   asm
     mov ax,13h
     int 10h
    end;                       { MCGA képmód bekapcsolása            }
   for i:= 0 to 199 do
    HLine( i);                 { Kezdőkép kiírása                    }

 { 2. Főciklus: mozgatás }

   repeat
    if KEY[$C8] and (y>0  ) then ScrollU(1);
    if KEY[$D0] and (y<199) then ScrollD(1);
    if KEY[$CB] and (x>0  ) then ScrollL(1);
    if KEY[$CD] and (x<319) then ScrollR(1);
```

```
asm
   mov   dx,3dah          { Az egyenletesebb futás érdekében vára- }
@1:in    al,dx            { kozunk egy vertikális visszafutásra    }
   test  al,8
   jz    @1
   cld                    { Megjelenítés: a háttér bájtjait sza-   }
   push   ds              { vanként, növekvő sorrendben átmásoljuk }
   mov    cx,32000        { az $A000-s szegmens első 64000 bájtjára }
   mov    es,SegA000
   xor    di,di
   lds    si,background
   rep    movsw
   pop    ds
   end;

    until KEY[1];         { ESC megnyomására léphetünk ki          }

 asm
   mov ax,3                 { Szöveges mód                         }
   int 10h
  end;
 DoneKey;                   { BIOS-megszakítás visszaállítása      }
 end.
```

Ha két nyilat nyomunk meg egyszerre, a térkép két irányban gördül, ha ez a két nyíl nem ellenkező irányú, a kép átlósan mozog. Ez lassú gépeken darabos lehet, ha azok nem tudják egy elektronsugár-visszatérési idő alatt mindkét irányhoz tartozó memóriamozgatásokat és sorvisszafejtéseket végrehajtani. Ez esetben, de a gyorsabb futás érdekében mindenképp érdemes még négy átlós irányú görgető eljárást írni, melyek ugyanúgy átmozgatják az egész háttér bájtjait, de nemcsak egy sort vagy oszlopot rajzolnak a megüresedő helyre, hanem mindkettőt. Ennek a problémának a megoldását az Olvasóra bízzuk, mert gyors gépeknél (80-100 MHz) nem annyira jelentős.

5. Ütközések

Egy játékprogramban elengedhetetlenül fontos figyelni, hogy egy BOB mikor ütközik egy másikkal, mikor éri el a képernyő szélét, vagy mikor érinti a háttér bizonyos pontjait. Amikor egy lövöldözős játékban a lövedék eltalálja az ellenséget, vagy egy autósban a gépkocsi nekimegy a falnak, a lövedék és az ellenség, az autó és a fal néhány pontja fedi egymást, egymás felett helyezkednek el. Ekkor ezek érintkeznek, ütköztek egymással. Egy BOB mozgása során állandóan figyelni kell, mikor kerül bizonyos elemekkel (BOB-bal, háttéregységgel, bizonyos pixelekkel) fedésbe, mert ettől függ a program további menete: felrobban-e az ellenség, vagy a főhős veszít életeiből.

Azt, hogy a BOB érintkezik-e egyes pontokkal, kétféleképpen vizsgálhatjuk. Minden elemhez rendelhetünk egy ütközési vázat, maszkot, aminek egységei bitek. Két BOB találkozása akkor következik be, ha koordinátáik alapján az ütközési vázukból legalább egy 1-es pixel fedi egymást. De nemcsak BOB-hoz rendelhetünk ilyen maszkot, hanem háttérhez is, például ott 1-re váltjuk a maszk bitjeit, ahol fal van, és így tudjuk figyelni, mikor ütközik falnak a BOB. Ennek az eljárásnak a legnagyobb előnye az, hogy mindegyik grafikus elemhez több ilyen vázat, maszkot is rendelhetünk, így megvalósítható például az, hogy ahol a játék főhőse falnak ütközik, ott egy szellem átjuthat. Hátránya az, hogy minden BOB-hoz, háttérképhez külön meg kell szerkeszteni annak a maszkját, és ezt tárolni kell, ami helyet foglal, igaz, hogy a tárolás miatt csak nyolcadannyit, mint a hozzá tartozó grafikus elem (nyolc bitet lehet egy bájton tárolni).

Egy ilyen bitmaszkot alkalmazó ütközésvizsgáló rutin felépítése a következő lehet: beolvassuk az egymást fedő bájtokat, ezeket el kell forgatni, mivel általában az abszcisszák közti különbség nem osztható nyolccal, majd ha az elforgatott bájtokon végrehajtunk egy ÉS műveletet, és lesz olyan bájtpár, amelyiknél az eredmény nem nulla, akkor a két egység fedi egymást. Mivel mi itt nem ezzel a bitmaszkos találkozásfigyelő módszerrel fogunk foglalkozni, a feladat megoldása az Olvasóra vár, feltéve, hogy szükségesnek találja. Csupán a végrehajtáshoz jól alkalmazható kiinduló ötletet említettük meg, a forgatást és az AND művelet alkalmazását. Természetesen máshogy is megoldható ez a fela-

dat, például úgy, hogy bitenként végignézzük a két ütközési vázat, így nem kell a forgatással vesződni.

Az ütközés vizsgálatára a másik megoldás a **színváz**. Ennek lényege az, hogy kijelölünk egy vagy több színt, és ha két grafikus egység ilyen színű pontjai fedik egymást, akkor azok összeütköztek. Nem kell külön helyet lefoglalni a maszknak, csupán arra kell ügyelni, hogy két külön színt kell alkalmazni azokon a helyeken, ahol az ütközés megengedett, és azokon, ahol nem, még ha ez a két szín teljesen ugyanolyan, akkor is. Például egy autós játékban legyen a járda és az úttest egyaránt szürke, de a járda a kocsi számára „tiltott övezet". Ez esetben már a háttér szerkesztésének kezdeténél figyelni kell arra, hogy más sorszámú színnel kell a járdát és más sorszámúval az utat befesteni.

A színvázas módszer a logikátlanabb, de egyszerűbb. Vele nem tudjuk például megoldani azt, hogy egy ellenség csak akkor haljon meg, ha a fejét eltaláljuk, mert a BOB-okat egységesen kezeli. (Természetesen nincs olyan probléma, ami nem megoldható, itt megoldás lehet például az, hogy az ellenséget két BOB-ként kezeljük, külön a fejét és külön a testét. Ez csak egy példa arra, hogy soha nem szabad feladni, még ha egy kicsit nehezebben megoldható feladat előtt állunk is, mindig tartsuk szem előtt azt, hogy semmi sem megoldhatatlan, csupán türelem és ötlet kérdése.) E színmaszkot alkalmazó módszer után az Olvasó kifejlesztheti a maga bitmaszkos ütközésvizsgálóját.

A fejezetben található szubrutinok mindegyike függvény, visszatérési értékük logikai változó, mely akkor igaz, ha a megadott adatok szerint a vizsgált egységek fedésben vannak. Ezt könnyedén alkalmazhatjuk Pascal programunkban, a következő feltételes mondatok beépítésével: HA BOB1.ütközik (BOB2-vel) AKKOR BOB1.meghal. Persze az utasítások formája nem ilyen, csak a könnyebb érthetőség miatt fordítottuk le magyar nyelvre. De az már látszik, hogy ezentúl minden BOB nemgrafikus adatait *object* változóként fogunk deklarálni, ami a mi szempontunkból csak annyiban tér el a rekordtól, hogy magába foglalja a BOB-hoz tartozó eljárásokat, függvényeket, így az ütközést vizsgáló szubrutinokat is.

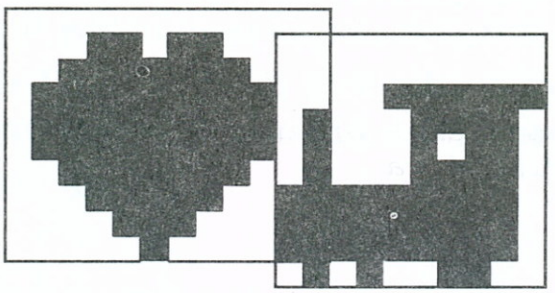
Minden érintkezés egyik alanya BOB lesz, hiszen nem lenne értelme például a térkép egymáshoz viszonyítva változatlan helyzetű egységeinek fedését vizsgálni.

Viszont a BOB már érintkezhet:

1. másik BOB-bal,
2. a háttér előre meghatározott színű pixeleivel
3. a térkép jelzett egységeivel (bizonyos BOX-okkal).

5.1. Egyszerű BOB-BOB ütközés

Ha két BOB alakja téglalap, melyek teljesen kitöltik a rendelkezésre álló területet, azaz a BOB-ok szélén nincsenek üres pixelek, 0 értékű bájtok, akkor ütközésük figyeléséhez elegendő csupán minden mozgásuk után a koordinátájukat és méretüket vizsgálni. Ebből a négy adatból (abszcissza, ordináta, szélesség, magasság) egyértelműen, könnyen és gyorsan meghatározható, hogy két téglalap alakú BOB mikor kerül fedésbe, vagy ha a BOB-ok nem téglalap formájúak, akkor a területük mikor érintkezik. Sok játékban elegendő csupán ezt a fajta érintkezést vizsgálni, melynek előnye, hogy könnyen érthető, röviden megvalósítható, és a vizsgálat gyorsan végrehajtható. Hogy jobban megértsük, figyeljük meg a következő ábrát!



Itt két BOB mintájának fekete-fehér rajza látható. A fekete négyzetekhez tartozó bájtok értéke nullától különböző, a fehér részekhez tartozó memóriaegységek értéke pedig nulla, ezeken a helyeken a BOB átlátszó. Az ábrán jól látszik, hogy a két BOB nem áttetsző pontjai nem fedik egymást, tehát valójában a két BOB nem érintkezik. Mégis, ha az ütközést csak a BOB-ok koordinátái és méretei alapján vizsgáljuk, eredményül IGEN-t kapunk, mert a két BOB alapterülete fedi egymást.

Az alábbi eljárásban tehát a legtöbb esetben nem azt kapjuk, hogy két BOB mikor találkozik, hanem azt, hogy mikor közelíti meg egymást. Biztosan csak

akkor kaphatunk helyes eredményt, ha mindkét BOB téglalap alakú, és nem tartalmaznak átlátszó, 0 színű pontokat.

A feladat megoldása nagyon egyszerű, csak arra kell ügyelni, hogy a koordináták negatív számok is lehetnek, mivel a BOB a háttér felett és attól balra is elhelyezkedhet (meg persze a másik két irányban is). Természetesen két BOB találkozása teljesen független a háttértől, érintkezhetnek úgy is, hogy azt mi nem látjuk, magyarul a BOB koordináta-rendszere látható részén, a háttéren kívül történik az ütközésük.

```
function Collision: boolean; assembler; asm
   mov     ax,X1          { A 2. BOB legalább az első oszlopának    }
   add     ax,W1          { legalább egy pixele (még ha átlát-      }
   cmp     ax,X2          { szó is) belelóg-e az 1. BOB tartományá- }
   jng     @nocoll        { ba? (Ha nem - nem érintkezhetnek)       }

   mov     ax,Y1          { A 2. BOB függőlegesen alulról érinti-e  }
   add     ax,H1          { az 1. BOB-ot?                           }
   cmp     ax,Y2
   jng     @nocoll

   mov     ax,X2          { Az 1. BOB abszcisszája nagyobb-e a 2.   }
   add     ax,W2          { BOB abszcisszájának és szélességének    }
   cmp     ax,X1          { összegénél? (Ha igen - nincs ütközés)   }
   jng     @nocoll

   mov     ax,Y2          { Ugyanez csak ordinátákra és magasságra  }
   add     ax,H2          { megvizsgálva                            }
   cmp     ax,Y1
   jng     @nocoll

   mov     al,1           { A függvény visszatérési értéke 1, azaz  }
   jmp     @exit          { TRUE, ha mind a négy kitétel igaznak    }
@nocoll:                  { bizonyult.                              }
   mov     al,0
@exit:
   end;
```

A függvényben szereplő word típusú változók:

X1	Első BOB abszcisszája	X2	Második BOB abszcisszája
Y1	Első BOB ordinátája	Y2	Második BOB ordinátája
W1	Első BOB szélessége	W2	Második BOB szélessége
H1	Első BOB magassága	H2	Második BOB magassága

Nem azt vizsgáltuk tulajdonképpen, hogy fedésben vannak-e, hanem azt, hogy nem érintkeznek. Csak akkor lehetséges, hogy ne legyen egymást fedő részük, ha a következő négy eset mindegyike teljesül:

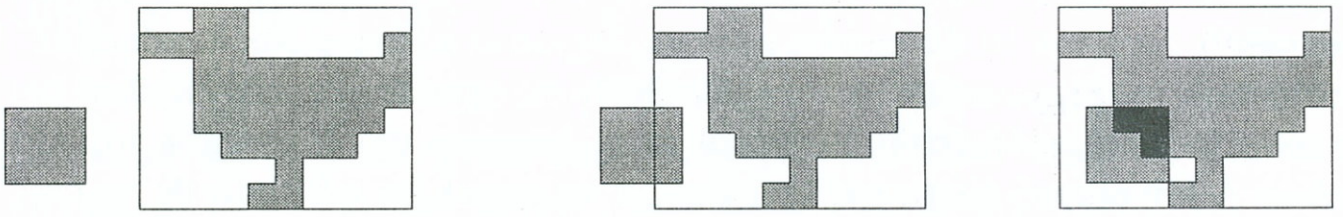
1. `X1+W1≤X2`
2. `Y1+H1≤Y2`
3. `X2+W2≤X1`
4. `Y2+H2≤Y1`

A függvény első négy egysége megvizsgálja ezeket az egyenlőtlenségeket, az utolsó része pedig beállítja a visszatérési értéket. Itt megint azzal a módszerrel találkozunk, amivel a *GetPixel* függvényben (1.2.), a függvényértéket az AL regiszterben adjuk meg.

Ez a módszer – a koordináták alapján végzett vizsgálat – a legegyszerűbb és a leggyorsabb. Ezért általában ezt alkalmazzuk, de van, amikor szükséges a pixelenkénti ellenőrzés is.

5.2. Tényleges BOB-BOB ütközés

Tegyük fel, hogy játékunkban kacsákat kell lődözni. Ha a lövedék a kacsa csőre alatt halad el, de érinti a madár területét, vagyis azt a téglalap alakú részt, amiben a madáralak elhelyezkedik, akkor még nem találtuk el, ellenben az iménti függvény találatot jelez.



A két szélső ábrán az ütközés megállapítása egyértelmű. A baloldalt lévőn sem a lövedék, sem a kacsa, de még téglalap alakú területük egyetlen pontja sem érintkezik, ezek tehát nincsenek fedésben. A jobb szélsőnél valóságos ütközés következett be, van legalább egy-egy olyan pont, amelyek közül az egyik a másikat takarja. Így tehát a területüknek is érintkezniük kell, vagyis az előző függvény visszatérési eredménye: TRUE.

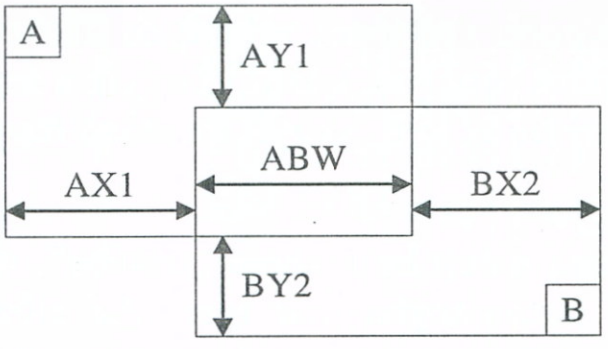
A középső rajzon azonban a „golyó" még nem hatolt be a kacsa „testébe", viszont a mintáját tartalmazó téglalap alakú tartományba már igen. Az előző részben bemutatott *Collision* függvény tehát találatot jelez, holott ha a golyó

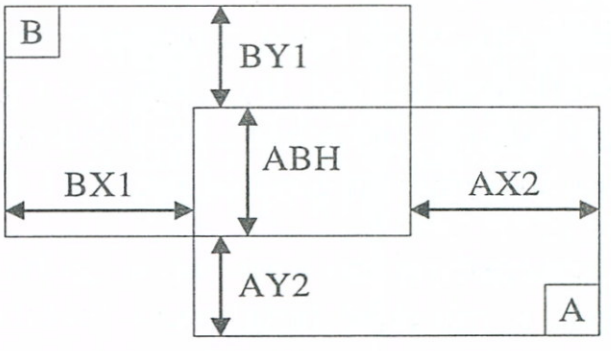
balra halad tovább, akkor a madár még nem fog meghalni. És nem is nézne ki jól, ha a töltények többsége csak megközelítené, de nem találná el a kacsát, és távolról végezne vele. Eszerint a koordináták és a méretek figyelésén kívül még további vizsgálatra lesz szükség.

Hogyan állapítsuk meg, hogy bekövetkezett-e valóságos ütközés? Mindenekelőtt meg kell vizsgálni, hogy a két BOB területének vannak-e egymást fedő pontjai. Ezt az előző részben megismert módon tesszük, a koordináták, a szélességek és a magasságok alapján. Ha a két terület nem érintkezik, a két BOB-nak nem lehet egymást fedő átlátszatlan pontja, ekkor nem is kell mást tenni. Ez az eset fent, az első rajzon látható.

Hogyha egymásba lóg a két BOB mintájának alapterülete, meg kell vizsgálni a közös részt. Ehhez végig kell pásztázni annak pontjait, és ha találunk legalább egy olyan helyet, ahol mindkét BOB átlátszatlan, vagyis ezen a helyen mindkét pixelhez tartozó bájt nullától különböző, akkor ez a két BOB érintkezik, ütköztek. Ez azonban nem olyan egyszerű. Hasonlít a probléma a BOB-megjelenítés utolsó fokozatához (2.5. fejezet), ott is meg kellett határozni egy közös részt, a BOB-nak azt a téglalap alakú területét, ami rajta van a háttéren. Itt is ezt kell tenni, de kétszer, mindkét BOB-ból ki kell vágni egy azonos nagyságú téglalapot, és ezeket kell bájtonként összehasonlítani.

A legnehezebb azonban az összevetendő területek szélességét, magasságát és kezdőcímét meghatározni (ez utóbbi a két BOB-nál különböző, csak akkor lehet egyenlő, ha a két BOB azonos mintájú, ugyanaz a Shape-jük kezdőcíme, és koordinátáik megegyeznek). A vizsgált BOB-ok egymáshoz viszonyított helyzete sokféle lehet, előfordulhat például az is, hogy az egyik magába foglalja a másikat. A következő ábrák két lehetőséget mutatnak be ezekből, de az összes számítás elvégzésének módját megtudhatjuk belőlük.





A változók jelentése az A jelzésű BOB esetében (a B-vel kezdődő ismeretlenek jelentése ezekből értelemszerűen következik):

AX1 A BOB bal oldali nem vizsgálandó oszlopainak száma. A fenti második ábrán ez nulla, mert az összehasonlítandó terület bal széle itt egybeesik az A BOB bal szélével.

AY1 A felső nem ellenőrizendő sorok száma.

AX2 Jobb oldali oszlopok száma, amiket nem kell figyelni.

AY2 Alsó sorok száma. Ezt tulajdonképpen nem használjuk a függvényben, csak a teljesség kedvéért szerepel az ábrán

ABW A vizsgálandó tartomány szélessége. Kiszámításának módja: AW-(AX1+AX2), AW az A jelű BOB szélessége. Ha az A BOB benne van a B BOB-ban, akkor ABW=AW.

ABH A vizsgálandó téglalap magassága.
Meghatározása: AH-(AY1+AY2), AH az A BOB magassága.

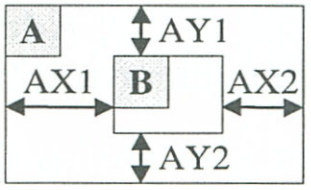
A változók meghatározásához szükséges képletek (megint csak az A BOB-ra vizsgálva):

AX1	=	BX-AX	(B és A abszcisszájának különbsége)
AY1	=	BY-AY	(B és A ordinátájának különbsége)
AX2	=	(AX+AW)-(BX+BW)	(AW, BW a BOB-ok szélessége)
AY2	=	(AY+AH)-(BY+BH)	(AH, BH a BOB-ok magassága)
ABW	=	AW-AX1-AX2 vagy	BW-BX1-BX2
		(ennek a két értéknek	egyeznie kell)
ABH	=	AH-AY1-AY2 vagy	BH-BY1-BY2

Ezek csak nemnegatív egész, azaz természetes számok lehetnek. Erre a kiszámításuknál még nem kell ügyelni, lesz az eljárásban egy olyan rész, mely ellenőrzi az ábrán jelölt összes változót (ABW és ABH kivételével, mert ezek nullánál nagyobbak), és amelyik negatív, annak értékét nullára változtatja. A második esetben például, amikor AX>BX (A a B-től jobbra található), AX1 negatív lesz, de az első nem vizsgálandó oszlopok száma nulla, mert az összehasonlítási terület és az A BOB bal széle megegyezik. Ezért ebben az esetben AX1-et nullára változtatjuk.

Egy másik példát az alábbi ábra szemléltet. Itt az A jelű BOB körülveszi à B-t, mert méretei nagyobbak, és a koordináták úgy vannak beállítva. Ekkor AX1,

AX2, AY1, AY2 mind nagyobb mint 0, viszont ezek B-hez tartozó párja mindegyike zérus.



Az összehasonlítást pontonként végezzük el. Elkezdjük megvizsgálni az A BOB meghatározott tartományának pontjait, majd ahol nullától eltérő színűre bukkanunk, megnézzük, hogy az ezen a helyen lévő B BOB pontja milyen. Ha nulla, nincs érintkező pont, mehet tovább egészen a következő nem nulla A pontig, vagy a rész jobb alsó sarkáig. Ha nem nulla, akkor a két BOB legalább egy pixele fedésben van, tehát a két BOB összeütközött. A fenti ábrák változóira a regiszterek beállításánál lesz szükség:

DI+	= AY1×AW+AX1	Összevetési tartomány eltolási címe az A Shape-jén (ennyit kell DI-hez adni, miután az ES:[DI]-be betöltöttük az A kezdőcímét)
SI+	= BY1×BW+BX1	Tartomány eltolási címe a B Shape-en
CX	= ABW	Egy sor hossza
DX	= ABH	Sorszámláló (a vizsgálódás tágabb ciklusához)

A gyorsaság érdekében a területek vizsgálatához ugyan lehetne a *repnz scasb* utasításpárt alkalmazni, de ekkor csak a DI regiszter mutat a megfelelő pontba, SI nem változik. Ezért olyan módszert használunk, ami minden A-beli pont ellenőrzése után a DI és az SI regisztert is növeli. Az ABW és ABH változókat a kódszegmensben tároljuk, @W és @H a címük.

A függvény ismertetése előtt röviden tekintsük át annak menetét!

1. Koordináták és méretek alapján az ütközés lehetőségének vizsgálata. (Nagyjából megegyezik az előző függvény soraival.)
2. Koordináták és méretek alapján az egymást fedő területek nagyságának és kezdőcímeinek meghatározása.
   2.1. AX1, AX2 stb. kiszámítása, kaphatunk negatív eredményeket is.
   2.2. AX1, AX2 stb. közül a negatívakat nullára változtatjuk.
   2.3. Regiszterek beállítása (DS:[SI], ES:[DI]).

3. Összehasonlítás, megnézzük, hogy akad-e legalább egy olyan hely, ahol mindkét BOB nem átlátszó.

```
{coll2.inc}

function BOB.Collision( var B: BOB): boolean; assembler; asm
   jmp    @start          { Kódszegmensben tárolt változók átugrása }

@W:dw     0               { ABW, a közös rész szélessége             }
@H:dw     0               { ABH, a közös rész magassága              }
@A:dw     0               { AX1+AX2 összeget tároljuk itt            }
@B:dw     0               { BX1+BX2 összeg                           }

@start:
   mov    di,word [Self] { [DI] az A BOB adatainak kezdőcíme         }
   mov    si,word [B]    { B BOB adataira pedig az [SI] mutat        }
   push   ds             { DS is kell                                }

 { 1. Megvizsgáljuk, hogy vannak-e közös pontjaik }

   mov    ax,[di]        { Ha valamelyik BOB nem aktív, nem kell     }
   and    ax,[si]        { tovább vizsgálódni, nem érintkezhetnek,   }
   jz     @nocoll        { mert az egyik nem látszik                 }
   mov    ax,[di+2]      { 'A' BOB abszcisszája                      }
   add    ax,[di+18]     { 'A' BOB szélességét hozzáadjuk            }
   cmp    ax,[si+2]      { 'B' BOB ettől jobbra van-e?               }
   jng    @nocoll        { Ha igen, nem érintkezhetnek               }
   mov    XA,ax          { Az összeget tároljuk                      }
   mov    ax,[di+4]      { Ugyanezt megvizsgáljuk, csak függőlege-   }
   add    ax,[di+20]     { sen                                       }
   cmp    ax,[si+4]
   jng    @nocoll
   mov    YA,ax          { Időspórolás miatt Y+H-t is elmentjük      }
   mov    ax,[si+2]      { Jöhet a másik két vizsgálat:              }
   add    ax,[si+18]     { vízszintesen...                           }
   cmp    ax,[di+2]
   jng    @nocoll
   mov    XB,ax
   mov    ax,[si+4]      { ... és függőlegesen                       }
   add    ax,[si+20]
   cmp    ax,[di+4]
   jng    @nocoll
   mov    YB,ax

 { 2. Most már biztos, hogy van a két BOB-nak közös területe,
      meg kell határozni annak méretét és helyét }

 { 2.1. Értéket adunk az AX1-BY2 változóknak a fenti képletek
        szerint }
   mov    ax,[si+2]
   sub    ax,[di+2]
   mov    AX1,ax
   mov    ax,[si+4]
```

```
   sub     ax,[di+4]
   mov     AY1,ax
   mov     ax,[di+2]
   sub     ax,[si+2]
   mov     BX1,ax
   mov     ax,[di+4]
   sub     ax,[si+4]
   mov     BY1,ax
   mov     ax,XA
   sub     ax,XB
   mov     AX2,ax
   mov     ax,YA
   sub     ax,YB
   mov     AY2,ax
   mov     ax,XB
   sub     ax,XA
   mov     BX2,ax
   mov     ax,YB
   sub     ax,YA
   mov     BY2,ax

 { 2.2. Amelyik ezek közül negatív, nullára változik }
     cmp   ax1,0
     jg    @A1
     mov   ax1,0
@A1: cmp   ax2,0
     jg    @A2
     mov   ax2,0
@A2: cmp   ay1,0
     jg    @A3
     mov   ay1,0
@A3: cmp   ay2,0
     jg    @A4
     mov   ay2,0
@A4: cmp   bx1,0
     jg    @B1
     mov   bx1,0
@B1: cmp   bx2,0
     jg    @B2
     mov   bx2,0
@B2: cmp   by1,0
     jg    @B3
     mov   by1,0
@B3: cmp   by2,0
     jg    @B4
     mov   by2,0
@B4:

   mov     ax,ax1          { Az AX1+AX2 összeg a sorok végén az ug-  }
   add     ax,ax2          { ráshoz szükségesek, ezért tároljuk a    }
   mov     word [@A],ax    { kódszegmensben                          }
   mov     ax,bx1          { Ugyanígy a BX1+BX2 összeget is          }
   add     ax,bx2
   mov     word [@B],ax
   mov     ax,[di+18]      { Már csak az összehasonlítandó terület   }
   sub     ax,word [@A]    { méreteit kell megadni:                  }
```

```
   mov     word [@W],ax  { szélessége: A.W-(AX1+AX2)                    }
   mov     ax,[di+20]
   sub     ax,ay1
   sub     ax,ay2
   mov     word [@H],ax  { magassága:  A.H-AY1-AY2                      }

 { 2.3. Regiszterek beállítása, kezdőcímek kiszámolása }
   mov     ax,[di+10]    { AX: az aktuális fázis távolsága              }
   mul     word [di+16]  {     (fázisszám×fázishossz)                   }
   mov     cx,ax         { AX ideiglenes tárolása                       }
   mov     ax,[di+18]    { Meghatározzuk az A BOB-on belül a vizs-      }
   mul     ay1           { gált tartomány kezdőpontját:                 }
   add     ax,cx
   add     ax,ax1        { A.W×AY1+AX1 (+fázis×fázishossz)              }
   les     di,[di+12]    { ES:[DI] a Shape kezdőcímére mutat            }
   add     di,ax         { Most már az első vizsgálandó pontra          }
   mov     ax,[si+10]    { Ugyanezeket a számításokat B-re is el-       }
   mul     word [si+16]  { végezzük                                     }
   mov     cx,ax
   mov     ax,[si+18]
   mul     by1           { Kezdő pont: B.P×B.PL+B.W×BY1+BX1             }
   add     ax,cx
   add     ax,bx1
   lds     si,[si+12]    { A B Shape-re a DS:[SI] mutat                 }
   add     si,ax         { SI-hez hozzáadjuk a fenti értéket            }

 { 3. Összehasonlítás }

   mov     dx,word [@H]  { DX szokás szerint a sorszámláló              }
@nextline:
   mov     cx,word [@W]  { CX pedig a pontokat számlálja                }
@scan:
   lodsb                 { Egy pont a B-Shape-ből                       }
   cmp     al,0          { Ha nulla, akkor ott a két BOB nem            }
   jz      @nil          { érintkezhet                                  }
   mov     al,es:[di]    { Ha nem nulla, megvizsgáljuk, hogy az A       }
   cmp     al,0          { BOB ezen a helyen átlátszó-e                 }
   jnz     @coll         { Ha nem, ütközés van                          }
@nil:
   inc     di            { DI növelése (SI a LODSB miatt nőtt)          }
   loop    @scan         { Egy sor letapogatása                         }
   add     di,word [@A]  { DI és SI növelése, hogy a következő sor      }
   add     si,word [@B]  { AX1. ill BX1. oszlopára mutasson             }
   dec     dx            { Sorszámláló csökkentése                      }
   jnz     @nextline     { Összes sor letapogatása                      }
@nocoll:                 { Ha nincs ütközés                             }
   mov     al,0          { AL=0, tehát a függvényérték FALSE            }
   jmp     @exit
@coll:
   mov     al,1          { Ellenben AL=1, TRUE                          }
@exit:
   pop     ds            { Eredeti DS                                   }

end;
```

A BOB típust objektumként kell deklarálni majdnem úgy, mint a **SHOWBOB5.PAS** példaprogramban, csak még hozzá kell írni a következő sort:

```
function Collision( B: BOB): boolean;
```

A függvény demonstrációja megtalálható a lemezmelléklet **COLLBOB.PAS** programjában, két BOB közül az egyiket irányíthatjuk, a másik áll, és ha összeütköztek, hangjelzést hallunk. A program csak az eddig megismert eljárásokat tartalmazza, így annak részletei nem szorulnak további magyarázatra.

5.3. BOB-háttér ütközés

Mit is értünk valójában egy BOB háttérrel történő ütközése alatt? Azt, hogy annak bizonyos, általunk előre meghatározott részei felett helyezkedik el, egyes pontjait takarja. Egy labirintusos játékban például szükséges azt figyelni, hogy a BOB, a főhős mikor ütközik falnak, mert arra nem mehet tovább. Ezt a vizsgálódást is el lehetne végezni egy háttérmaszk segítségével, éppúgy, mint a BOB-BOB ütközésnél, mi azonban megint csak a színfigyelős módszerrel foglalkozunk.

A lényeg tehát: ha a BOB legalább egy, nem átlátszó (nem 0 színű) pontja a háttérnek előre megadott színű pontja felett van, azt takarja, akkor a két grafikus elem (BOB és háttér) összeütközött. Fontos, hogy több olyan szín legyen, aminek érintésével az ütközést vizsgáló függvény még hamis, *false* eredménnyel térjen vissza, mert legtöbbször a háttér azon része, amelyben a BOB szabadon mozoghat, nem egyszínű. Már egy egyszerű autós program is sokkal szebb, ha az útra fehér útburkolati jelek vannak festve, így az autó szabad mozgásterülete, az út mindjárt két színű: szürke és fehér.

Maga a függvény nagyon hasonlít a végső *ShowBOB* eljárás középső részére, amikor a BOB a háttérre kerül, a különbség csupán annyi, hogy egy BOB-bal dolgozunk, és háttérre írás helyett onnan olvasunk, éppúgy, mint a BOB-BOB ütközésben a közös rész vizsgálatánál. Egyszóval szinte semmi újat nem készítünk, két eddig ismertetett alprogram, a *ShowBOB* eljárás (2.5. fejezet) és a *Collision* függvény (5.2.) egy-egy részletét használjuk fel újra. A függvény alapelve ugyanaz, mint a BOB-BOB ütközésnél: először meg kell határozni a két grafikus elem közös részét. Ez ugyanaz, mint a *ShowBOB* eljárás 2.1-2.2. jelzésű része, vagyis meghatározzuk, a BOB mennyire lóg ki a háttérből, mek-

kora az összehasonlítandó terület nagysága és kezdőcíme a háttéren, valamint a BOB-on belül. Ha nincs rajta a háttéren, nem ütközhetnek.

A közös területet ellenőrző sorok viszont mások, mint a BOB-BOB ütközésnél, de az alapelv itt is ugyanaz. Csupán annyiban különbözik, hogy a háttérnek több olyan színe van, aminek érintésével még nincs ütközés (ezeket a színeket mi adjuk meg), így a háttér bájtjait másként kell megvizsgálni. A BOB-ot ellenben ugyanúgy ellenőrizzük.

Hogy milyen színű háttérpontok hatására ne jelezzen ütközést a függvény, legegyszerűbb, ha egy tartományként adjuk meg. Két változó tárolja ennek az egybefüggő intervallumnak az alsó és felső határát, az olyan pixelek, amelyek ezek, vagy ezek közé esnek, a háttér szabad részét képezik, ahol a BOB szabadon mozoghat. Erre már a grafika elkészítésénél, a színek meghatározásánál ügyelni kell. Egy repülős játékban tegyük fel, hogy csak akkor robban fel a repülő, ha a földbe csapódik. A hely, ahol mozoghat, legyen többszínű: kék ég, amely világosszürke felhőket tartalmaz. A felhők öt színből állnak, a világosszürke különböző árnyalataiból. Ez esetben, már a háttér megrajzolása kezdetén ki kell jelölni hat egymást követő színt, az ezzel a színnel befestett háttérpixelek adják a BOB szabad mozgásterét, máshol nem szabad őket felhasználni.

A függvénynek csak azt a részét ismertetjük, ami új, vagyis a BOB és a háttér közös területét letapogató sorait. Az alábbi regiszterekbe és változókba a következő értékeket kell tölteni:

BL	A színtartomány alsó határa, ez alatt olyan színek találhatók, aminek érintésére a függvény ütközést jelez.
BH	A színtartomány felső határa. [BL..BH] intervallum adja azon pixelek színét, melyek fölött a BOB szabadon, ütközés nélkül mozoghat.
DX	Az ellenőrizendő terület magassága, ennyi sort kell megnézni.
DS:[SI]	A megvizsgálandó téglalap kezdőcíme a Shape-en belül.
ES:[DI]	A téglalap kezdőcíme a háttéren.
@CXSAVE	A megfigyelt terület szélessége. A kódszegmensben kell tárolni, így DS visszaállítása nélkül is el lehet érni.
@DIPLUS, @SIPLUS	Ennyi kell adni DI-hez, ill. SI-hez egy sor ellenőrzése után.

```
@nextline:
   mov     cx,word [@cxsave] { Egy sorban ennyi pontot ellenőrzümk }
@scan:
   lodsb                     { Egy bájt megvizsgálása a Shape-en be- }
   cmp     al,0              { lül, ha nulla, átlátszó, akkor itt    }
   jz      @nil              { nem érintkezhet a háttérrel           }
   mov     al,es:[di]        { Ha nem nulla, és az ES:[DI] által     }
   cmp     al,bl             { megcímzett bájt nem esik a megadott   }
   jc      @coll             { színintervallumba, ütközés követke-   }
   cmp     al,bh             { zett be, a @coll-ra ugrik, ahol 1-re  }
   ja      @coll             { állítja a visszatérési értéket (AL)   }
@nil:
   inc     di                { Minden pont ellenőrzése után növelni  }
   loop    @scan             { kell a DI-t is                        }
   add     di,word [@diplus] { Egy sor megvizsgálása után a DI és    }
   add     si,word [@siplus] { az SI is növekszik                    }
   dec     dx                { Sorszámláló csökkentése, ha még nem   }
   jnz     @nextline         { nulla, újra végrehajtjuk              }
```

Ez a BOB-háttér ütközést vizsgáló függvény magja, a többi rész elkészítését már az Olvasóra bízzuk. Nem lesz túl nehéz, az eddig ismertetett *ShowBOB* és *Collision* szubrutinokból kell bizonyos részeket kivagdosni. A kész függvény egyébként megtalálható a *Game* egységben a lemezmellékleten (*CollColors* néven), ezt a unitot a 8. fejezetben részletesen megismerhetjük.

A lemezmellékleten található még egy **COLLBACK.PAS** nevű program, ami betölt egy LBM fájlt háttérnek, és bemutatja a BOB-háttér ütközést.

5.4. BOB-BOX ütközés

Ha a 4. fejezetben leírt térképtechnikát alkalmazzuk, gyakran felmerülhet az a probléma, hogy a háttéren kívül is vizsgálhassunk összeütközést, a BOB és a térképnek a háttértől különálló részei között. Ugyanis a BOB-ok, helyzetüket tekintve, teljesen függetlenek a háttértől, mozoghatnak nem látható részeken is, és néhány játékban ezt jól ki lehet használni: előre beállítjuk a BOB-ok koordinátáit, nem baj, ha némelyik nem látszik, és görgetésnél változtatjuk őket. Így a játék területe az egész térképre kiterjedhet. Éppen ezért, mivel a háttéren kívül is folyik a játék, ott is kell ütközéseket vizsgálni, ott sem mehet neki például egy autó a falnak.

Most egy ütközési maszkos módszert alkalmazunk. Mindegyik BOX-hoz hozzárendelünk egy bitet, ami ha 1, akkor azzal a vizsgált BOB-nak nem szabad érintkeznie, vagyis a függvény igaz eredménnyel tér vissza. Nem vizsgáljuk

pixelenként, csak azt, hogy az adott BOB területe érint-e olyan BOX-ot, amihez korábban 1-et rendeltünk. A függvény részletesebb magyarázata a *Game* egység leírásánál található (8.23.3. fejezet, *CollBOX* függvény).

6. BOB-EDITOR

A lemezmellékleten található **BOBEDIT** segédprogrammal BOB-okat lehet rajzolni. A BOB-adatok a lemezen úgy kerülnek tárolásra, hogy az a *Game* egység által felhasználható legyen, a következő sorrendben:

Bájt	Tartalom
0.	Ennek a bájtnak az értéke 1, a BOB verziószáma.
1-2.	Shape szélessége (a valódi szélesség ennél 1-gyel nagyobb).
3-4.	Shape magasság (ugyanez itt is érvényes).
5-6.	Fázisok száma. Ha ez nulla, akkor a BOB egyfázisú.
7-	Grafikus adatok olyan sorrendben, ahogy a memóriába kerülnek.

Az editor forrásszövege a lemezen megtalálható, a program által használt unitok forráskódjával együtt. Ezeket az egységeket valószínűleg az Olvasó is tudja hasznosítani, ezért először ezekkel ismerkedhetünk meg. Majd a program kezelése következik, amit a menürendszer és a funkciók leírása követ.
Mindezek előtt néhány fontos tudnivaló.

1. A program használatához nélkülözhetetlen az egér, csak ezzel lehet ugyanis szerkeszteni, színt választani, és a menük is csak egérrel érhetők el, igaz, hogy a hozzájuk kapcsolódó funkciók legtöbbje ún. forrókulcsokkal is elérhető.
2. Angol nyelvű.
3. A programhoz tartozik egy **HOTKEYS.TXT** fájl, ami nélkül a *Help* menü *Hot Keys* funkciója nem működik.

6.1. A program által használt unitok

A BOB-szerkesztő segédprogram a következő unitokat alkalmazza: *Crt, DOS, Game, MCGA, Mouse, _System.* Ezek közül a *Crt* és a *DOS* beépített Pascal-unitok, a *Mouse* egységgel már foglalkoztunk (3.2.) és a *Game* egység ismertetésére majd a 8. fejezetben kerül sor. Ezek szerint most az *MCGA* grafikai rutinokat tartalmazó, és a *_System* unit leírása következik, amely a beépített *Sytem* bővítése.

6.1.1. Az MCGA egység

Az MCGA unit segítségével egyszerű műveleteket végezhetünk el a képernyőn 320×200/256-os üzemmódban. Először eljárásait, függvényeit, majd beépített konstansait ismerhetjük meg.

```
procedure SetMode( Mode: word);
```

A videokártyát a MODE változóban megadott üzemmódba állítja. Szöveges képmódot például a SetMode(3); utasítással érhetünk el, az MCGA üzemmód bekapcsolására pedig $13-at kell paraméterként megadni.

```
procedure PutPixel( X, Y: word; C: byte);
```

Pixel kigyújtása. A balról X. oszlopban, fentről Y. sorban lévő képpontot C színűre festi. (A sorok és az oszlopok számozása 0-val kezdődik.)

```
procedure Point( X, Y: word);
```

Az egység *Interface* részében deklarált *Pointcolor* tipizált konstans által meghatározott színűre festi az (X;Y) pontot.

```
procedure Rectangle( X1, Y1, X2, Y2: word);
```

Keret rajzolása, melynek bal felső sarka (X1;Y1), a jobb alsó pedig (X2;Y2). Színét a *PointColor* változóban adjuk meg, az eljárás meghívása előtt.

```
procedure CBar( Color: byte; X1, Y1, X2, Y2: word);
```

Color színű (X1;Y1) bal felső sarkú és (X2;Y2) jobb alsó sarkú kitöltött téglalap rajzolása.

```
procedure Bar( X1, Y1, X2, Y2: word);
```

Téglalap rajzolása, csak színét nem az eljárás fejében, hanem a *PointColor* változóban kell megadni.

```
procedure OutText( X, Y: word; TXT: string);
```

Szöveg kiíratása. A karakterláncot a TXT változó tartalmazza, helyét pedig az X;Y számpárral határozhatjuk meg. A szöveg a 8×8-as karakterekből áll, a szabvány VGA karakterkészletet alkalmazza, ezért egyes ékezetes betűk nem jeleníthetők meg (pl. Ő, Ű). Színét a *PointColor*, a szöveg alapszínét (hátterét) a *TextBackColor* változó tartalmazza.

```
procedure OutTransparentText( X, Y: word; TXT: string);
```

Átlátszó" szöveg kiírása. Annyi különbség van közte és az előző eljárás között, hogy az *OutTransparentText* által kiírt szöveg átlátszó, mögötte látszik, ami előtte a képernyőn volt.

```
procedure SaveBar( P: pointer; X1, Y1, X2, Y2: word);
```

Téglalap alakú tartomány elmentése a P mutató által megcímzett területre. A téglalap bal felső sarka: (X1;Y1), jobb alsó: (X2;Y2). A képernyőrészlet tartalmát bájtonként, sorfolytonosan tárolja.

```
procedure LoadBar( P: pointer; X1, Y1, X2, Y2: word);
```

Előzőleg elmentett terület újra megjelenítése. P a forráscím, X1, Y1, X2, Y2 adják a téglalap koordinátáit.

```
procedure VerticalBlink;
```

Várakozás addig, amíg az elektronsugár el nem éri a képernyő jobb alsó sarkát.

```
function  GetPixel( X, Y: word): byte;
```

Pixel színének lekérdezése.

```
const PointColor: byte = 15;
```

Ezzel a bájttal határozhatjuk meg a rajzolási színt a következő eljárásokban: *Point, Rectangle, Bar, OutText, OutTransparentText*. Mindig az eljárás meghívása előtt kell értéket adni neki.

```
const TextBackColor: byte = 0;
```

Az OutText eljárás által kiírt szöveg hátterének színe. Ennek is az eljárás meghívása előtt kell értéket adni.

6.1.2. A _SYSTEM egység

```
function  _VAL( S: string): longint;
```

Az S sztringet számmá konvertálja. Ha S a 0...9 számjegyeken kívül még egyéb karaktereket is tartalmaz, a függvény visszatérési értéke 0.

```
function  _STR( X: longint): string;
```

A paraméterben megadott egész számot szöveggé konvertálja (akkor használjuk majd, amikor az egér koordinátáit szeretnénk az *OutText* eljárással megjeleníteni).

```
function  _COPY( S: string; B: byte): char;
```

Visszatérési értéke az S karakterlánc B. karaktere.

```
procedure sw_byte( var A, B: byte);
procedure sw_word( var A, B: word);
```

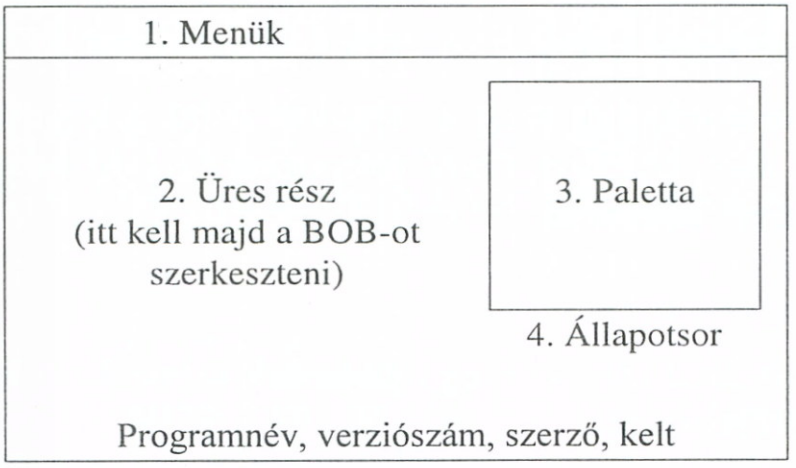
Ezek az eljárások felcserélik A és B tartalmát.

```
function  IOError: word;
```

Ellenőrzi az IOResult változó értékét. Ha nem nulla, hibát jelez és megállítja a program futását.

6.2. A BOB-Editor megjelenése és használata

A BOB-Editor indítása után a következő felépítésű képet láthatjuk:



6.2.1. Menük

Négy főmenü van: *File, Edit, Tools* és *Help*. Az egérrel, kattintással érhetjük el őket, aminek hatására legördülnek a kiválasztott menü funkciói. Részletesebben később foglalkozunk a menürendszerrel (6.3. fejezet).

6.2.2. Szerkesztési terület

A képernyő bal-középső része a BOB rajzolás területe. Ha nem adunk meg a programnak paramétert, indítás után ezen a részen még semmi sem található, csak fájl megnyitása vagy új fájl létrehozása után lehet szerkeszteni. Hozzunk létre egy új fájlt: kattintsunk a *File* menüre, majd a legördülő *New* funkcióra. Itt meg kell adni a Shape méreteit (először a szélességét, majd a magasságát), legyen ez kerek szám, pl. 32×32. Ezután a szerkesztési terület bal felső sarkában egy fehér keret található, most már lehet rá rajzolni: egyszerűen csak ki kell választani (bal oldali egérgombbal) a paletta egy színét, majd az egérkurzort a szerkesztési területre irányítva a bal oldali egérgomb lenyomásával befesthetjük a BOB pontjait. Ahol a BOB 0 színű, ott átlátszó lesz.

Nagyítani a '+', kicsinyíteni a '–' gombbal lehet (érdemes a numerikus billentyűzetet használni), gördíteni pedig a nyílbillentyűkkel, vagy egérrel a képernyőn a paletta alatt található nyíl gombokkal lehet. Ha ez utóbbit használjuk, mindig egy pixelt mozdul el a BOB, viszont a billentyűk segítségével gyorsabban, 10 pontonként választhatjuk ki a BOB látszó részét (ha a nagyítási arány, a ZOOM 10-nél nagyobb, akkor ezekkel is 1-1 pixelt mozdíthatunk). A maximális nagyítási arány 174-szeres, a *Home* billentyű hatására ez 1-re változik, és a BOB is hirtelen az alaphelyzetbe gördül, vagyis látszik a bal felső sarka. Ezzel a gombbal gyorsan meg tudjuk nézni, milyen a BOB eredeti, 1:1-es nagyságban, majd az *End* billentyű hatására gyorsan visszaugorhatunk az eredeti nagyításhoz és helyzethez.

Ha 1:1 arányú a nagyítás (ZOOM=1), akkor az egér jobb oldali gombjával egy téglalap alakú rész jelölhető ki, ehhez folyamatosan kell nyomni a jobb gombot, amíg a kijelölés el nem éri a kívánt nagyságot. Ha ezután a bal gombbal kattintunk a kijelölt részre (amihez hozzátartozik annak szegélye is), azt arrébb mozgathatjuk. Ezenkívül még sok más művelet is elvégezhető, ha van érvényes kijelölés (pl. kivágás, másolás, tükrözés, forgatás), ezekre majd a menürendszer tárgyalásánál térünk ki. A kijelölt rész szélességét és magasságát a paletta alatt a W és a H betűk után leolvashatjuk. Ez akkor fontos, ha a kijelölést forgatni akarjuk (*Tools\Rotate* menü), mert ilyenkor érdemes négyzet alakra törekedni, csak ekkor lesz tökéletes a forgatás.

Új fázist az **Ins** vagy az **I** billentyűvel szúrhatunk be az éppen aktuális fázis mögé. Az első hatására az új fázis az előzőnek a mintáját tartalmazza (így

könnyen tudunk olyan animációt készíteni, amikor csak kis változás van két fázis között), míg az **I** billentyűvel üres, 0 színű fázist iktathatunk be. A fázisokat a *Page Up/Down* billentyűkkel változtathatjuk, de az első tizet a 0-9 gombokkal is elérhetjük. Az aktuális fázis sorszámát a képernyő jobb alsó sarkában, a *Phase* felirat alatt láthatjuk.

6.2.3. Paletta

A jobboldalt található paletta mind a 256 színt tartalmazza, a program indítása után ezek a VGA-alappaletta színei. Az egérmutató alatti szín hexadecimális sorszáma könnyen leolvasható a palettától balra és felfele elhelyezkedő számjegyek segítségével, tízes számrendszerbeli sorszámát pedig az alatta levő információs részből tudhatjuk meg. Ha a bal gombbal kattintunk egy színre, az lesz a rajzolószín, a jobb oldali egérgombbal pedig módosíthatjuk az egérmutató bal felső sarka alatti szín összetevőit.

Színszerkesztésnél eltűnik minden, csak a paletta marad, és a képernyő bal oldalán a következőket láthatjuk (fentről lefelé haladva): billentyűk használata, színösszetevők intenzitása, egy, a kiválasztott színnel beszínezett téglalap, amely a módosított színt szemlélteti.

A színösszetevőket billentyűkkel módosíthatjuk, az alábbi táblázat és ábra szerint:

Q	Vörös (*Red*) növelése.
A	Vörös csökkentése.
W	Zöld (*Green*) összetevő növelése.
S	Zöld csökkentése.
E	Kék (*Blue*) komponens növelése
D	Kék csökkentése

	R	G	B
+	Q	W	E
–	A	S	D

Ha a bal egérgombbal egy másik színre kattintunk, átmásolhatjuk annak összetevőit a módosítandó színre, tehát ugyanolyan lesz a két szín. Ez akkor lehet előnyös, ha egy szín fokozatos árnyalatait készítjük el: mindig lemásoljuk az előző színt, és csak kevéssel változtatjuk összetevőit attól függően, hogy milyen finom árnyalatot kívánunk létrehozni.

A palettaszerkesztésből visszalépni a jobb gombbal vagy az **Enter** billentyűvel lehet. Ezután ugyan megváltoznak az adott szín összetevői, de ez még önmagától nem kerül sehol tárolásra, a BOB mentésével csak azt jegyezzük meg, hogy melyik helyen milyen sorszámú szín szerepel, magát a színt nem. Ezért, ha módosítjuk az alappalettát (amire gyakran lesz szükség), minden kilépés előtt mentsük el az **F4** billentyű vagy a *File\Save Pal* funkció segítségével, mert ha ezt nem tesszük, a színek adatai elvesznek. Egyébként a program kilépés után megkérdezi, hogy el akarjuk-e menteni a palettát, nyomjunk **Enter**-t vagy **Y**-t, ha igen, más gombot, ha nem.

A 15-ös színt ($0F, első sor utolsó szín, fehér) nem érdemes módosítani, lévén ez az előtér színe, ha például feketére változtatjuk, előfordulhat, hogy nem látunk semmit. Ekkor nyomjuk meg az **F8**-at, és válasszunk új színt az előtérnek (bal gomb), majd a jobb gombbal vagy bármely billentyűvel léphetünk vissza. Ez főleg nem általunk készített, például LBM-ből importálás után kiszedett paletták esetében szükséges, mert ezeknél előfordulhat, hogy a 15-ös szín halvány vagy nem látszik. Ezenkívül természetesen akkor is módosíthatjuk az előtérszín sorszámát, ha fehér helyett például zöldet szeretnénk látni, ami egy sokkal kellemesebb szín.

6.2.4. Állapotsor

A paletta alatt található területen egyrészt információkat kaphatunk a koordinátákról, színről stb., másrészt az itt elhelyezkedő nyilak segítségével a BOB-ot pixelesen tudjuk görgetni. Az állapotot jelző elemek jelentése (balról jobbra, fentről le):

X,Y — Ha az egérkurzor a szerkesztési területen van, akkor az általa mutatott pont koordinátái, egyébként a görgetés mértéke, a szerkesztési hely bal felső sarkában található pont koordinátái.

□ — (változó színű négyzet) Ha az egérkurzor a szerkesztési területen vagy a palettán van, akkor az alatta lévő pont színe és sorszáma, egyébként pedig a kiválasztott szín és annak sorszáma.

Nyilak — A BOB-ból látszó részt állíthatjuk, ha az egyik egérgombbal rájuk kattintunk. Ezenkívül még a billentyűzet nyíl gombjaival is lehet görgetni, ráadásul gyorsabban.

ZOOM — A nagyítás arányát mutatja. 1 esetén nincs nagyítás, értékét a +/- billentyűkkel változtathatjuk, szélsőértékei: 1 és 174 (csak egész szám lehet).

PHASE Az aktuális, éppen látható fázis sorszáma (nullával kezdődik a számozás). Változtatni a 0-9, valamint a *Pg Up/Down* gombokkal lehet.

W,H Kijelölés közben láthatók, a kijelölt rész szélességét (W) és magasságát (H) mutatják, ha egy BOB-részletből új BOB-ot kívánunk létrehozni, ezeket adjuk meg az új BOB méreteinek.

6.3. Menük és funkciók

A BOB-Editor menürendszere két szintből áll, az indítás után is látható főmenükből (pl. *File*) és a kattintás hatására legördülő funkciókból. Ez utóbbiak legtöbbjéhez tartozik egy-egy funkcióbillentyű, amivel gyorsabban, kényelmesebben érhetjük el az adott funkciót. Ezeket a billentyűket a következő részben külön is felsoroljuk, ismeretük hasznos lehet. Ellenben a menüket csak egérrel, gombnyomással érhetjük el (és néhány funkciót is).

6.3.1. A *File* menü

Mint a felhasználói programok általában, fájlműveletek kezelésére szolgál.

Funkció	**Billentyű**	**Rövid leírás**
New		Új fájl létrehozása.
Open	F3	Meglévő fájl megnyitása.
Save	F2	Fájl mentése.
Save As	Shift+F2	Fájl mentése más néven.
Import	Shift+F3	Grafika importálása LBM fájlból.
Load Pal	F5	Paletta betöltése lemezről, alappaletta visszaállítása.
Save Pal	F4	Paletta mentése.
DIR		Aktuális könyvtár tartalma, könyvtárváltás.
Exit	Alt-X	Kilépés (mentési szándék kérdezése nélkül!).

New

Új fájlt hozhatunk létre vele. Ha esetleg már volt szerkesztés alatt egy fájl, az kérdezés nélkül kitörlődik a memóriából, igaz, hogy üres **Enter** megnyomására még visszaléphetünk. A funkció meghívása után először gépeljük be a Shape szélességét, majd magasságát (mindkettő után **Enter**-t kell nyomni), és már lehet is dolgozni rajta. Arra kell csupán ügyelni, hogy a szélesség és a magasság szorzata ne haladja meg a 65534-et.

Eleinte a BOB egyfázisú, amit az **I** vagy **INS** billentyűkkel lehet bővíteni. Ha valamelyik mérethez nem adunk meg adatot, csak leütjük az **Enter**-t, változás nélkül visszaléphetünk, mintha egy *Cancel* gombra kattintottunk volna, így nem vész el a korábban szerkesztett fájl.

Open (F3)

Egy 40×25-ös felbontású kép jelenik meg, középen kék részben az aktuális könyvtár állományai találhatók, baloldalt pedig egy kis segítség az *Open* művelet használatához. A fájlok fizikai sorrendben követik egymást (ahogy a lemezen megtalálhatóak), **Enter** leütésére a legfelül látható fájl nyílik meg. A 'BOB' kiterjesztésű fájlok sárga színnel ki vannak jelölve, így könnyebben észrevehetők. A fájlkiválasztást és -megnyitást a következő billentyűk segítik:

Enter	A képernyőn a legfelső, a fehér nyilakkal megjelölt fájl megnyitása. Ha ez két pont, vagy egy könyvtár, akkor directory-váltás következik.
ESC	Művelet visszavonása, ha esetleg meggondoltuk magunkat (például nem mentettük el az előző fájlt).
A-Z	Lemezmeghajtó-váltás. A meghajtóhoz tartozó betűvel megjelölt billentyűt kell leütni a kívánt meghajtó aktuális könyvtárának listázásához.
Szóköz	Gyakran egyszerűbb a fájl nevét és útvonalát begépelni, mint kikeresni, ezt tehetjük meg a Space billentyű lenyomásával. (Például ha túl sok fájl található az aktuális könyvtárban.)
↑, ↓	Fájllista gördítése fel és le, egyesével.
Home	Ugrás a fájllista elejére (az első fájlra).
End	Ugrás a fájllista végére (az utolsó fájlra).
Page Up	Egy oldal (22 fájl) ugrás felfele.
Page Down	Egy oldal ugrás lefele.

Itt még érdemes megjegyezni, hogy az *Open* művelet magját a programban a *FileList* függvény alkotja, paraméterében kell megadni a sárgán kiemelt fájlok kiterjesztését, visszatérési értéke pedig a kiválasztott fájl neve. Ezt a szubrutint talán az Olvasó is fel tudja használni egy saját editor készítéséhez.

Save (F2)

Ha a fájl még nem volt elmentve (vagyis nem egy meglévő, hanem új fájlt kezdtünk módosítani), akkor a *Save As*-nél leírt funkció lép érvénybe, egyébként elmenti a fájlt a régi nevén. Fontos, hogy a *Save* művelettel a paletta még nem kerül elmentésre, erről külön gondoskodni kell a *Save Pal* funkcióval!

Save As (Shift+F2)

Úgy menthetjük el a fájlt, hogy új nevet adunk neki. A funkció hatására megjelenik egy kék csík, ez a beviteli mező, nekünk csupán be kell gépelni a fájl nevét, kiterjesztését, és elérési útvonalát, ha más könyvtárba kívánjuk menteni. Ha már létezik ilyen fájl, a program visszakérdez, hogy valóban felül akarjuk-e írni („Overwrite?"). Ha igen, nyomjunk **Y**-t vagy **Enter**-t; más billentyűt, ha nem. Ha nem adunk meg fájlnevet, csupán egy **Enter**-t nyomunk, visszavonhatjuk a megkezdett műveletet (a mentést).

A funkció után, a szerkesztéshez visszatérés előtt a program figyelmeztet, hogy ne felejtsük elmenteni a palettát, amiről most is külön kell gondoskodni (*Save Pal* vagy **F4**).

Import (Shift+F3)

Először meg kell adni a fájl nevét kiterjesztéssel (és elérési útvonallal együtt, ha szükséges), ahonnan a BOB grafikáját importálni szeretnénk, itt még üres **Enter**-re visszaléphetünk. Fontos, hogy jól adjuk meg a fájlnevet, és tényleg LBM formátumú legyen a kép, mert különben a program futása megszakad (más funkcióknál nem szakad meg, csak hibát jelez), és fontos az is, hogy előzőleg elmentsük a palettát és a szerkesztett BOB-ot, mert azok tartalma megváltozik.

Ha nincs semmi hiba, megjelenik a megadott LBM-kép, rajta jó esetben látszik egy egérrel mozgatható képpont. Ez előtér színű, és ha sehogy sem akar megjelenni, adjunk az előtérnek más színt (**F8**). A jobb oldali gomb folyamatos nyomása és az egér mozgatása mellett jelölhetjük ki az importálandó részlet nagyságát, majd a gomb elengedése után a szükséges helyre irányíthatjuk a kijelölést. Ezután a bal gomb megnyomása után másolhatjuk ki a megjelölt részt, ami a BOB grafikus adata lesz.

Nemcsak grafikát, palettát is importálunk ezzel a funkcióval, így tudjuk például „lelopni" egy LBM kép színskáláját, viszont ha magát a képet is felhasználjuk játékunk hátteréhez, a paletta színeinek összetevőit nem szabad módosítani, mert megváltozhat a háttér színe. Csak palettaimportáláshoz nem a bal egérgombot, hanem egy billentyűt kell megnyomni.

Akkor előnyös ez a funkció, ha egy másik rajzolóprogrammal szerkesztjük a BOB-ok grafikáját, így konvertálni tudjuk az ismertebb LBM formátumú képet a kevésbé ismert BOB formátumba.

Load Pal (F5)

Egy 768 bájt hosszúságú fájl nevét és kiterjesztését, esetleg elérési útvonalát kell megadni, üres Enter hatására innen is visszaléphetünk. Az alappaletta visszaállításához csupán a következőt kell begépelni: **def pal**, kisbetűkkel.

Save Pal (F4)

Adjuk meg a fájl nevét, kiterjesztését, ahova a paletta adatait menteni kívánjuk. Csak Enter leütésére természetesen innen is visszaléphetünk. Ha már létezik a megadott fájl, a program megkérdezi, hogy szándékunkban áll-e azt felülírni. Nyomjunk **Enter**-t vagy **Y**-t, ha igen, más billentyűt, ha nem. A paletta mentése nagyon fontos, ha egyszer elfelejtjük, csak nagyon nehezen tudjuk újra beállítani a módosított színeket, és ez sok bosszúságot okozhat.

A paletta színösszetevőinek a tárolása is olyan módon történik, hogy az a *Game* unit *LoadPal* eljárásával (8.11.) egyből betölthető.

DIR

A BOB-Editor hőskorából maradt meg ez a funkció, ezt műveletet az *Open*-nel is végrehajthatjuk. Listázza az aktuális könyvtár tartalmát, majd megkérdezi, hogy kívánunk-e könyvtárat váltani („Change directory?"). Ha **Y**-t vagy **Enter**-t ütünk le, meg kell adni az új könyvtárat, két pont a feljebblépés.

Exit (Alt-X)

Kilépés előtt mindig mentsük el a BOB-adatokat és a palettát, kilépés után már csak a paletta mentésére van lehetőségünk (ha netán elfelejtettük volna).

6.3.2. Az *Edit* (szerkesztési) menü

Négy funkciójával a kijelölt rész tárolása, eltüntetése, visszahelyezése valósítható meg. Indítás után közvetlenül még egyik funkció sem használható, ehhez érvényes kijelölés szükséges. Először tekintsük át a funkciókat és a hozzájuk tartozó billentyűket!

Funkció	Billentyű	Rövid leírás
Delete	Del	Kijelölt rész törlése.
Cut		Kijelölt rész kivágása.
Copy	Ins	Kijelölt rész memóriába másolása.
Paste		Előzőleg memóriába másolt rész beillesztése.

Delete (Del)

Ha 1:1 arányú nagyítás (ZOOM=1) mellett a jobb egérgombbal kijelöltünk egy téglalap alakú részt, ezzel a funkcióval törölhetjük úgy, hogy azt visszaírni később nem tudjuk, a törölt részt a program nem jegyzi meg. A kijelölést 0 értékű bájtokkal tölti fel.

Cut

Ugyanúgy eltüntethetjük vele a kijelölt részt (0 színre festés), csak azt a program megjegyzi, s később a *Paste* funkcióval beszúrhatjuk. Ezzel lehet egy BOB bizonyos részeit egy másik BOB-ba átvinni, mert új BOB létrehozása vagy megnyitása esetén a memóriából nem törlődik ki a *Cut* funkcióval elmentett rész.

Copy (Ins)

Ez is a memóriába másolja a kijelölt rész tartalmát, viszont nem törli azt, és a kijelölést sem szünteti meg. A memóriában mindig csak egy, a legutolsó *Cut* vagy *Copy* művelettel bevitt részlet tárolódik, két *Copy* vagy *Cut* művelet után az első által elmentett rész törlődik a memóriából.

Paste

Ha előzőleg a *Cut* vagy a *Copy* funkcióval elmentettünk egy részletet, azt a *Paste* funkcióval rajzolhatjuk vissza, ha nincs aktív kijelölés. A *Paste* funkció

meghívása után az egér mozgatásával irányíthatjuk a beillesztendő részletet, majd ha a kellő helyre ért, kattintással ragaszthatjuk be.

6.3.3. A *Tools* (eszközök) menü

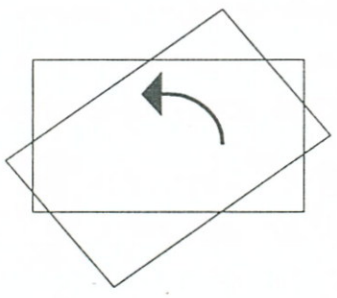
Funkciói főként a kijelölt rész módosítására (tükrözésére, forgatására stb.) szolgálnak, de innen érhető el a színcsere és az animáció művelet is.
:

Funkció	**Billentyű**	**Rövid leírás**
Turn		Aktuális fázis tetszőleges forgatása.
Inverse	I	A kijelölt részt inverzére változtatja.
Mirror ↔	X, H	Kijelölt rész horizontális tükrözése.
Mirror ↕	Y, V	Kijelölt rész vertikális tükrözése.
Rotate	R	Kijelölt rész derékszögű forgatása.
Bar		Kijelölt rész feltöltése egy színnel.
Change	F7	Színcsere.
Animate	F6	Animáció.

Turn

Tetszőleges szöggel forgathatjuk el a BOB aktuális fázisát. A funkció meghívása után meg kell adni a forgatás szögét, melynek egész számnak kell lennie, továbbá -32768 és 32767 közé kell esnie (*Integer* típus). Ha csak **Enter**-t nyomunk, még visszaléphetünk, egyébként a műveletet nem lehet visszavonni.

A *Turn* funkció a BOB-ot középpontja körül forgatja el, az alábbi ábrán látható módon:



Az elforgatott téglalap sarkai kilóghatnak a Shape területéből, az itt lévő adatok elvesznek. Ezért forgatásnál mindig nagyobb méretű BOB-bal dolgozzunk, aminek csak a közepére rajzolunk. Maga a forgatás természetesen nem tökéle-

tes, hiszen túl nagyok a pixelek, így némi utómunkára, igazításra mindig szükség van.

Inverse (I)

Csak az alappaletta alkalmazása mellett eredményes ez a funkció. A kijelölés pontjainak színe úgy változik meg, mintha a palettát függőlegesen tükröznénk, egy soron belül az első színnel beszínezett pontok a soron belüli utolsó színt kapják (egy sor 16 színből áll). Így például a 16-os fekete színű pontokból 31-es fehér lesz, és fordítva. Ezért kell az alappaletta, mert ott – legalábbis az első 32 szín esetében – a színek megfelelően vannak sorrendbe rakva.

Mirror ↔ (X,H)

Csak érvényes kijelölés mellett alkalmazható; a kijelölt rész pontjait annak függőleges, a téglalapot megfelező tengelyére tükrözi, tehát a tükrözés iránya vízszintes lesz.

Mirror ↕ (Y,V)

A kijelölt részt függőlegesen tükrözi (annak középső, vízszintes tengelyére). Ha az *Inverse* vagy a *Mirror* funkciók valamelyikét kétszer egymás után alkalmazzuk, a módosított rész visszaáll az eredeti állapotába, így ezekkel bátran próbálkozhatunk, az eredeti grafika visszaállítható.

Rotate (R)

Hogy a *Rotate* funkció ne módosítsa a kijelölt részen kívüli pontokat, érdemes a kijelölést négyzet alakúra szabni, a jobb gomb nyomása közben figyelni kell, hogy a paletta alatti W és H betűk után egyenlő számok álljanak. Ha ez sikerült, a *Rotate* hatására a megjelölt négyzetet óramutató járásával ellentétesen elforgathatjuk 90°-kal (négyszer megismételve a műveletet az eredeti grafikát kapjuk).

Ha a kijelölt téglalap nem négyzet, a BOB bizonyos pontjai elveszhetnek, vagy egyáltalán nem is működik a funkció (ekkor hangjelzést hallhatunk).

Bar

A kijelölt téglalapot feltölti az aktuális színnel. Ha a palettára kattintunk, az nem szünteti meg a kijelölést (ha máshova, akkor igen), így új színt választhatunk.

tunk anélkül, hogy újra meg kellene határozni a feltöltendő területet. A *Bar* funkcióval azonban óvatosan bánjunk, a kijelölt rész eredeti tartalma visszaállíthatatlan.

Change (F7)

A BOB-on belül bizonyos színű pontokat más színűre fest, nem csak egy fázison belül. A funkció meghívása után a szerkesztési terület kivételével minden eltűnik, de ez változatlan marad (például a nagyítás nem szűnik meg). Először ki kell választani a megváltoztatandó pontok eredeti színét, rá kell kattintani az egyik ilyen pontra a bal egérgombbal. A képernyő jobb alsó sarkából leolvashatjuk a kiválasztott pont helyzetét, színét. Jobb gombbal kattintva vagy billentyűnyomásra visszaléphetünk.

Ha kiválasztottuk az átfestendő színt, helyébe a palettából új színt kell kijelölni, a bal gombbal. Billentyűnyomással vagy a jobb gombbal még mindig visszaléphetünk. Átfestés után még más színekre is megismételhetjük a műveletet, a ciklusból a jobb gombbal (vagy egy billentyűvel) lehet kilépni.

A funkció az összes fázis kiválasztott színű pontjait átfesti!

Animate (F6)

Úgy jeleníti meg a BOB-ot, ahogy az egy játékban majd felhasználásra kerül. Az *Animate* funkció meghívása után a képernyő két részre oszlik: felül látható maga az animáció, alul pedig a közben használható billentyűk leírása, valamint az animáció sebessége. A BOB fázisai növekvő sorrendben követik egymást, a legutolsó után a legelső jön. A nyilakkal mozgatható, míg a sebességet a +/- gombokkal állíthatjuk (numerikus billentyűzet). A funkció közben használható billentyűk:

+, –	Sebesség növelése, csökkentése; késleltetési idő (*Delay*) csökkentése, növelése.
Nyilak	BOB mozgatása.
V	Beállíthatjuk, hogy minden egyes megjelenítés után várakozzon-e az elektronsugár vertikális visszafutására (ON = várakozás bekapcsolva, OFF = kikapcsolva). Itt megfigyelhető, hogy a mozgás szebb, de lassabb, ha figyeljük az elektronsugarat.
Ctrl-X	Visszalépés a szerkesztéshez.

6.4. Funkcióbillentyűk

A *Help* menü *Hot Keys* funkciója is megjeleníti az itt leírtak egy részét, ezt a funkciót az **F1** billentyűvel érhetjük el közvetlenül.

Szerkesztés közben használható billentyűk

F1	HOTKEYS.TXT fájl megjelenítése (*Help* menü).
F2	Fájl mentése.
Shift+F2	Fájl mentése új néven.
F3	Fájl megnyitása.
Shift+F3	Importálás (fájl menü).
F4	Paletta mentése.
F5	Paletta töltése, alappaletta visszaállítása.
F6	Animáció (*Tools\Animate*).
F7	Színcsere (*Tools\Change*).
F8	Új szín kiválasztása az előtérnek (ha a fehér nem tetszik). Ha palettabetöltés vagy importálás után nem látunk semmit, nyomjuk meg az **F8**-at!
Nyilak	BOB görgetése tíz pixelenként; ha a nagyítási arány 9-nél nagyobb, csak 1 pixelenként.
+, –	Nagyítási arány változtatása.
Home	Váltás 1:1-es nézetbe. A nagyítási és eltolási értékeket tárolja.
End	A tárolt nagyítási és eltolási értékek szerint a *Home* billentyű megnyomása előtti állapot visszaállítása.
INS, I	Új fázis beszúrása, az előző lemásolásával, ill. anélkül.
0..9	Gyors fázisváltás.
Page Up/Down	Következő/előző fázis.
Alt-X	Kilépés. Előtte ne felejtsünk menteni!

Kijelölés alatt használható billentyűk

X, H	Vízszintes irányú tükrözés.
Y, V	Függőleges tükrözés.
R	90°-os forgatás (óramutató járásával ellentétes irányba).
I	Kijelölt rész inverzére váltása. (Alappaletta mellett a fehérből fekete lesz.)
INS	Memóriába másolás (*Edit\Copy*).
DEL	Kijelölt rész törlése (*Edit\Del*).

Megnyitás funkció közben alkalmazható billentyűk

Enter	Megnyitás.
ESC	Visszavonás.
A-Z	Meghajtóváltás.
Szóköz	Fájlnév begépelése.
↑, ↓	Fájllista gördítése fel, le.
Home	Első fájl.
End	Utolsó fájl.
Page Up/Down	Egy oldal ugrás fel/le.

Animáció közben használható billentyűk

+, –	Sebesség változtatása.
Nyilak	BOB mozgatása.
V	Vertikális visszafutás figyelésének ki-, ill. bekapcsolása.
Ctrl-X	Visszalépés.

6.5. Egyéb lehetőségek

A BOB-Editort a DOS-ból paraméteresen is meghívhatjuk, egy vagy két paraméterrel, pl.:

```
BOBEDIT.EXE BOBFILE.BOB PALFILE.PAL
```

Az első paraméter annak a BOB-fájlnak a neve, amivel a program bejelentkezése után rögtön dolgozni szeretnénk, a második paraméter pedig egy palettafájl, ami szintén automatikusan betöltődik (ha adunk a programnak paramétert). Elérési útvonalat is írhatunk a fájlnevek elé, ha szükséges, viszont a kiterjesztést ne felejtsük el.

Az editor által elkészített grafikus objektum közvetlenül, átalakítás nélkül felhasználható játékainkban a *Game* egység segítségével. A betöltés módját a unit ismertetésénél, a 8.23.7. fejezetben ismerhetjük meg. A példaprogramok által használt BOB-okat is átalakíthatjuk, próbáljuk meg betölteni például a **ROCKET.BOB** fájlt!

7. MAP-EDITOR

A térképszerkesztő segédprogrammal **MAP** formátumú hátteret készíthetünk játékainkhoz, amelyet a *Game* unit is tud kezelni. Felépítése:

Cím	Méret	Tartalom
0.	1	Verziószám, értéke 1.
1.	2	Egy sorban található dobozok száma (térkép szélessége).
3.	2	Térkép hossza (bájtban vett helyfoglalása).
5.	16384	BOX adatok, sorrendben, sorfolytonosan.
16389.	(3.)	Térképadatok, sorfolytonosan.
ezután	32	Maszk, ütközési váz. Minden bit egy BOX-hoz tartozik.

A program által használt unitok közül eddig egy kivételével mindet ismertettük, most már csak a *Menu* egység működésének leírása van hátra, ezzel kezdjük a fejezetet. Ezután a program megjelenése, használata, majd a menük és funkciók ismertetése következik, mint a BOB-Editor leírásánál. A MAP-Editor forrásprogramja a **MAPEDIT.PAS** fájlban található, működéséhez szükséges még a **MECR.DAT** fájl, ha ez nincs az editor könyvtárában, nem indul. A program szintén angol nyelvű, és kezeléséhez elengedhetetlenül fontos egy működőképes egér, aminek legalább két gombja van.

7.1. A Menu egység

A korábban keletkezett BOB-Editor menükezelése még a programon belüli eljárásokkal van megoldva, amik rugalmatlanok, bővíthetetlenek (vagy csak nehezen). Ezzel szemben a MAP editor olyan eljárásokat használ, amelyek külön egységben találhatók (**MENU.PAS**), a menük feliratát, funkciójuk számát és nevét a felhasználó határozza meg, tetszőlegesen, így egy saját szerkesztőprogramhoz az Olvasó is hasznosítani tudja. A *Menu* unit eljárásai csak MCGA képszerkezet mellett működnek, az MCGA egység (MCGA.PAS – 6.1.1. fejezet) szubrutinjait használják. A menük és funkciók csak egérrel érhetők el, nincsenek gyorsbillentyűk, *Hot Key*-ek. A unithoz ezért szükséges még a **MOUSE.PAS** egység is.

A unit által elkészíthető menük kétszintesek, egy főmenüből és egy almenüből állnak. A főmenük a képernyő legfelső sorában helyezkednek el, ha egyikükre rákattintunk, legördülnek a főmenüből nyíló almenük, funkciók, de előtte a funkciólista által eltakart rész tárolódik, a visszaállítással tehát nem kell törődnünk. A menüsor vastagsága nyolc pixel, szélessége lehet teljes képernyőnyi (320) vagy annyi, amennyi szükséges (ezt a *Refresh* eljárásnál szabhatjuk meg).

7.1.1. Eljárások

```
MainMenu( s: string);
```

A főmenü meghatározása. Úgy kell megadni az elemeit, ahogy azok megjelennek, szóközökkel elválasztva. Például:

```
MainMenu('FILE EDIT SEARCH TOOLS HELP');
```

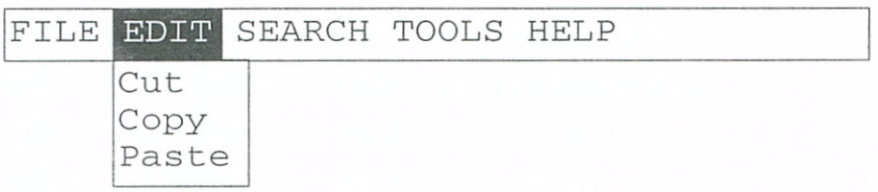
Semmi egyebet nem kell tenni, a program magától szétszedi az egyes menüket, és elraktározza. A *MainMenu* eljárás hatására a menük még nem jelennek meg.

```
SubMenu( count: word; s: string);
```

Almenük, funkciók létrehozása. A *Count* változóban adjuk meg a főmenü azon elemének számát, amelyből az S által meghatározott lista legördül. A funkciólistát az S paraméterben a következőképp kell megadni: egymás után írjuk a funkcióneveket, pontosvesszővel elválasztva, szóköz nélkül. Az előző példánál maradva a

```
SubMenu(2,'Cut;Copy;Paste');
```

végrehajtása után így fog kinézni a menürendszer:



A legördülő lista sorrendjét mi szabjuk meg, szélessége automatikus. A *SubMenu* eljárás önmagában még nem jelenít meg semmit, ehhez a *Refresh* eljárás szükséges.

```
GetMenu;
```

Kiválasztott menü, kiválasztott funkció lekérdezése. A programban, ahol ezt a unitot felhasználjuk, figyelni kell, a felhasználó mikor kattint a felső, 8 pixel

vastagságú részre, ezután kell meghívni ezt az eljárást, ami a *MainRes* változóba a kiválasztott főmenü sorszámát, a *MenuRes*-be pedig a funkciósorszámot tölti (1, ha a legfelső funkció lett kiválasztva). Nekünk nincs más dolgunk, mint meghívni ezt az eljárást, ami ezután elvégzi a funkciólista legördítését, figyeli az egeret stb. A legördülő funkciólista alatti rész nem vész el, elraktározódik, és az eljárásból való kilépés előtt visszaíródik.

```
Refresh( BarToEnd: boolean);
```

Megjeleníti a főmenüt (a funkciólista legördítését a *GetMenu* végzi). Ha IGAZ (*true*) paramétert adunk, a főmenü jobb széle egybeesik a képernyő jobb szélével, ellenkező esetben (*false*) hossza annyi, amennyi minimálisan szükséges. Fontos, hogy az eljárás meghívása előtt a videokártya MCGA ($13-as) üzemmódban legyen.

7.1.2. Változók, konstansok

Azonosító	Típus	Érték	Leírás
MainRes	word		A kiválasztott főmenü és funkció sorszámát tartalmazzák a *GetMenu* meghívása után.
MenuRes	word		

A következő négy konstanst csak úgy tudjuk módosítani, hogy átírjuk értéküket a forrásállományban (a **MENU.PAS** fájlban). A unit e négy érték alapján határozza meg az implementációs részben deklarált változók méreteit, ezért nem lehet őket futás közben módosítani.

Azonosító	Típus	Érték	Leírás
MaxLength		8	Főmenü egy-egy elemének maximális hossza (karakterben).
MaxMains		8	Főmenük maximális száma.
MenuLen		12	Funkciónevek maximális hossza (karakterben).
MaxMenus		7	Egy menüből legfeljebb ennyi funkció gördülhet le.

Az alábbi változók értékét bármikor módosíthatjuk, ha a fekete-fehér összeállítás nem tetszik, vagy ha nem az alappalettát használjuk. Minden változtatás után, hogy az eredmény látható legyen, meg kell hívni a *Refresh* eljárást.

Azonosító	Típus	Érték	Leírás
MFColor	byte	0	Menük és funkciók előterének színe.
MBColor	byte	15	Szövegek alap-, háttérszíne.
SFColor	byte	15	Kiválasztott menü, ill. funkció előtérszíne.
SBColor	byte	0	Kiválasztott elem háttérszíne.

Menük, funkciók betűsorát tároló változók, azok típusa:

```
MainStr: array [1..MaxMains] of string [MaxLength];
SubStr:  array [1..MaxMains,1..MaxMenus] of string [MenuLen];
```

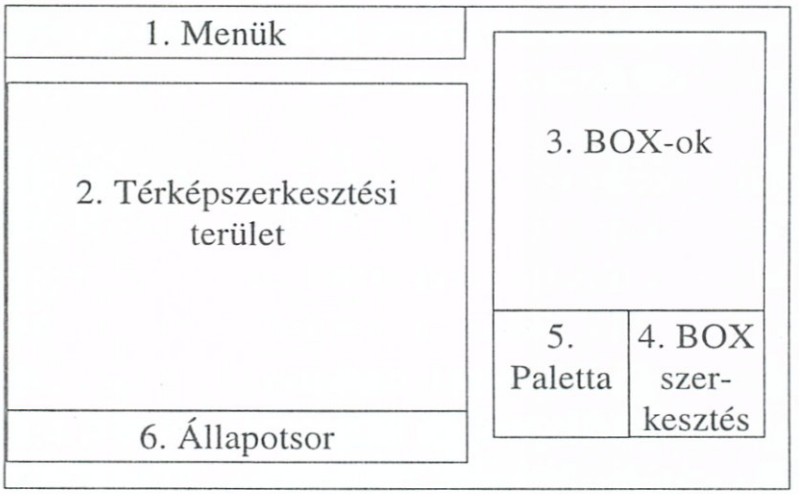
7.1.3. Példaprogram

Az alábbi példa szemlélteti a *Menu* egység használatát, de felhasználjuk benne a *Mouse* és az *MCGA* egység részeit is. A programból úgy lehet kilépni, ha nem a menüsoron kattintunk. Egér nélkül nem indul.

```
{menudemo.pas}

uses Menu, Mouse, MCGA; { Ezeket a unitokat már ismertettük        }
begin
 SetMode($13);          { MCGA üzemmód bekapcsolása                }
 if not InitMouse then begin
  SetMode( 3);
  writeln('Az egér nincs installálva.'+#7 { Hangjelzés } );
  halt
  end;
 MainMenu('Zöldség Gyümölcs');      { A menüsor két egységből áll  }
 SubMenu( 1, 'Répa;Retek;Dinnye'); { A 'Zöldség' menü funkciói     }
 SubMenu( 2, 'Alma;Körte;Meggy');  { A 'Gyümölcs' menü funkciói    }
 MFColor:=  1;                     { Előtérszín: kék               }
 SBColor:= 12;                     { Választott elem háttérszíne   }
 Refresh( true);                   { Menüsor megjelenítése         }
 ShowMouse;                        { Egérkurzor megjelenítése      }
 PointColor:= 12;
 TextBackColor:= 0;

 repeat
  repeat until ButtonPressed;    { Gombnyomásra várakozás          }
  GetMenu;                       { Kiválasztott elem lekérdezése   }
  PointColor:= 15;               { Ezt a két változót a GetMenu    }
  TextBackColor:= 0;             { eljárás megváltoztatja          }
  if MenuRes>0 then
   OutText( 0, 100,              { Eredmény kiírása                }
    SubStr[MainRes,MenuRes]+' '+MainStr[MainRes]+'      ');
 until MainRes=0;                { Kilépés, ha máshol kattintunk   }

 SetMode( 3);           { Szöveges mód                             }
 end.
```

7.2. A képernyő felépítése, az editor használata

Indítás után a következő kép ötlik szemünkbe:



Ez egy kicsit bonyolultabb, mint a BOB-Editor megjelenése, első indítás után talán egy kicsit nehéz lenne megérteni, mi hol van, melyik fekete téglalapon mit kell szerkeszteni (mert eleinte a BOX-bájtok mindegyike nulla, így a képernyő nagy része fekete). Ha rögtön az elején betöltjük a **PELDA.MAP** fájlt, könnyebb lesz megérteni a szerkesztő felépítését.

A MAP-Editor használatához szükséges a térképszerkezet ismerete (4.2. fejezet), ami összetettebb, mint egy egyszerű pixelenkénti képtárolási módszer, ezért most röviden átismételjük. Két részből áll: a térképből, és 256 BOX-ból. Minden BOX 8×8-as, és a memóriában egymás után helyezkednek el. A térkép minden egyes bájtja helyén egy 8×8-as négyzet, BOX jelenik meg. Ez akkor jó, ha egy grafikus ábrát többször fel akarunk használni, ezzel a módszerrel lényeges memóriát tudunk spórolni.

7.2.1. Menük

A főmenük száma négy, ezek sorrendben: *File, Edit, Options, Help*. Ha valamelyikre rákattintunk, legördül egy funkciólista, melyekkel a 7.3. fejezetben foglalkozunk részletesebben. Itt elég annyit megjegyezni, hogy a menük a *Menu* unittal készültek, és hogy a *Help* menü funkciói nem működnek, tetszés szerint kiegészíthetőek.

7.2.2. Térképszerkesztési terület

A menük alatt található részen szerkeszthetjük a térképet. Ez nem túl bonyolult, a bal egérgombbal szúrhatjuk be az aktuális dobozt, amit a képernyő jobb felső részében (3.) választhatunk ki, szintén a bal gombbal. Ha az egérkurzort a szerkesztési területen mozgatjuk, a státussor jobb oldalán a 'BOX' felirat után olvashatjuk le a kurzor alatti doboz sorszámát.

Már indítás után lehet szerkeszteni, akkor is, ha nem adunk paramétert, a futtatás után beköszönő térkép mérete 40×25 BOX, ami éppen egy képernyőnyi (320×200 pixel). Mivel a térképszerkesztési terület (2.) ennél sokkal kisebb, a MAP-et gördíteni kell. Ha ehhez a nyílbillentyűket használjuk, akkor az *Options* menü *Tab size* (7.3.3. rész) funkciójával beállított értékkel mozgathatjuk a térképet, ami alapállapotban egy, tehát eleinte a nyilak hatására egy BOX-nyit mozdul el a térkép. Tízesével gördíthetünk a következő billentyűkkel:

Crtl-→	Jobbra
Crtl-←	Balra
Page Up	Fel
Page Down	Le

A jobb oldali egérgomb folyamatos nyomása mellett egy téglalap jelölhető ki, ezzel különböző szerkesztési műveletek végezhetők el (*Edit* menü 1-3. funkció – 7.3.2. fejezet). Ha a bal gombbal rákattintunk a kijelölt részre, átmásolhatjuk egy másik helyre. Eközben a kiválasztott részt egérrel mozgathatjuk. A térképet beillesztés alatt is lehet gördíteni, a nyilakkal és a fenti billentyűkkel. Ha a megfelelő helyre irányítottuk a kijelölt téglalapot, a bal oldali gomb megnyomásával szúrhatjuk be.

7.2.3. BOX-kiválasztó rész

A képernyő jobb oldalán, felül vannak felsorakoztatva a dobozok, egymás után, növekvő sorrendben. Ha az egérkurzort valamelyikre ráirányítjuk, megjelenik annak mintája és sorszáma az állapotsoron. A bal gomb megnyomásával választhatjuk ki a szerkeszteni kívánt BOX-ot, és ez lesz egyben az aktuális doboz, amit a térképszerkesztési területen (2.) a bal gombbal illeszthetünk be. Ha kiválasztunk egy dobozt, annak mintája a képernyő jobb alsó részében, a BOX-szerkesztési területen (3.) nyolcszoros nagyítással jelenik meg.

Kiválasztáson kívül még egy egyszerű másolási művelet is elvégezhető a képernyőnek ezen a részén: ha valamelyik BOX fölött a jobb oldali gombot nyomjuk meg, mintáját átmásoljuk az aktuális dobozra. Tehát ha azt akarjuk hogy például az 1. BOX mintája olyan legyen, mint a 2.-é, akkor először a bal gombbal kattintsunk az első dobozra, majd a jobb oldalival a másodikra. Nem szabad összekeverni, mert az sok kellemetlenséget okozhat, ugyanis az 1. BOX mintáját csak úgy tudjuk visszaállítani, ha újrarajzoljuk.

Az aktuális, bal gombbal kiválasztott BOX sorszáma az állapotsorról olvasható le, ha az egérkurzort a képernyő tetejére vagy bal szélére húzzuk.

7.2.4. BOX-szerkesztési terület

A képernyő jobb alsó sarkában található. A jobb vagy bal oldali gombbal színezhetjük át az aktuális BOX pixeleit, a rajzolószínt a palettából a bal gombbal lehet kiválasztani. Az egér kurzora alatti szín sorszáma az állapotsorból olvasható le. Ha átfestjük egy BOX pixeleit, a változás megjelenik a dobozkiválasztó (3.) részen is, ahol megnézhetjük, hogyan néz ki lekicsinyítve, 1:1-es arányban.

7.2.5. Paletta

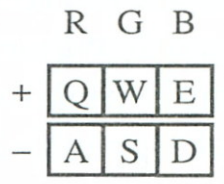
A dobozkiválasztótól lefele, a dobozszerkesztőtől balra található, színes négyzeteiről már első indítás után is könnyű felismerni. A bal oldali gombbal lehet a rajzolószínt kiválasztani, a jobb egérgombbal pedig szerkeszteni, összetevőit változtatni. Az egérkurzor által mutatott szín sorszáma az állapotsoron megjelenik.

Az egész paletta nem fért volna el úgy, hogy ilyen nagy részen (8×8 pixel) jelenik meg egy-egy színe, ezért vagy kicsinyíteni kellett, vagy úgy megoldani, hogy mindig csak egy részét látjuk. Ha egy palettaszínhez tartozó területet negyedére csökkentenénk, a paletta ugyan elférne, viszont nagyon nehéz lenne egy színét kiválasztani. Nem maradt más hátra, görgetéses módszert kell alkalmazni. Ha a paletta felett található **UP** feliratú részre kattintunk, felfelé, ha a **DOWN**-ra, lefelé gördül egy sorral.

Az egyik színre a jobb egérgombbal kattintva változtathatjuk annak összetevőit. Ekkor a palettát kivéve az egész képernyő megváltozik, feliratok, számok jelennek meg annak bal oldalán, és egy téglalap, amelynek színe megegyezik az

általunk kiválasztottal, itt ellenőrizhetjük, hogy jól állítottuk-e be a színösszetevőket. A komponensek változtatása a következő gombokkal történik:

Q Vörös (*Red*) növelése.
A Vörös csökkentése.
W Zöld (*Green*) összetevő növelése.
S Zöld csökkentése.
E Kék (*Blue*) komponens növelése.
D Kék csökkentése.



Itt is, mint a BOB-Editorban, ha a bal oldali gombbal kattintunk egy színre, átmásolhatjuk annak összetevőit az éppen szerkesztés alatt álló színre. Kilépni a színszerkesztőből **Enter**-rel vagy az egér jobb oldali gombjának megnyomásával lehet. Ha a 15-ös, fehér színt módosítjuk, a szerkesztő előtérelemeinek új színt kell adni a 7.5. fejezetben leírtak szerint.

7.2.6. Állapotsor

A térképszerkesztő rész alatt található állapotsor két részre osztható. Bal oldalán kaphatunk információkat a koordinátákról, ha az egérkurzor a térképen van, akkor az általa megjelölt BOX koordinátáját láthatjuk itt, a következő sorrendben és formában: abszcissza : ordináta. A koordináta-rendszer alapegysége 1 BOX. Ha az egér mutatója nem a térképszerkesztő részen van, akkor az állapotsor bal oldalán a térkép látható részének bal felső sarkában elhelyezkedő BOX koordinátáit láthatjuk, más szóval a térkép eltolási mértékét vízszintesen és függőlegesen.

A státussor jobb oldalán pedig az aktuális vagy az egér által mutatott dobozról vagy színről kaphatunk információkat. Az aktuális, kiválasztott BOX mintáját és sorszámát akkor láthatjuk ezen a részen, ha az egeret a képernyő bal oldalára vagy tetejére mozgatjuk, például a menüsorra. A rajzolószín sorszámát pedig akkor tudhatjuk meg, ha a képernyő jobb alsó részére visszük az egérkurzort.

7.3. Menük

A MAP-Editor menürendszere is kétszintes, egyrészt áll a főmenükből, amit a képernyő tetején, baloldalt láthatunk, másrészt a funkciókból, melyek egy főmenüre való kattintás után gördülnek le. Ugyanilyen menükezelést az Olvasó is alkalmazhat saját programjában, a 7.1. részben kifejtett *Menu* egység segítségével.

7.3.1. A *File* menü

A fájlműveletek (pl. mentés, töltés) elvégzésére szolgál, funkciói:

Funkció	**Billentyű**	**Rövid leírás**
New		Új fájl létrehozása.
Open	F3	Meglévő fájl megnyitása.
Save	F2	Fájl mentése.
Save as	Shift+F2	Fájl mentése más néven.
Save Pal	F4	Paletta mentése.
Load Pal	F5	Paletta betöltése lemezről.
Exit	Alt-X	Kilépés.

New

Az aktuális fájlt alaphelyzetbe állítja: a térképen lévő BOX-ok sorszáma 0 lesz, mindegyik dobozt 0 sorszámú színnel tölti fel. Előtte azonban a program megkérdezi, hogy valóban új térképet akarunk-e szerkeszteni. Ha igen, nyomjunk **Enter**-t vagy **Y**-t, ha nem, bármely más billentyűt vagy egérgombot. Az új fájl méreteit nem itt, hanem az *Options* menü *Map size* funkciójával lehet beállítani, a *New* funkcióval létrehozott térkép mérete megegyezik az azelőttiével.

Open (F3)

Egy gördíthető listából kell kiválasztani a megnyitni kívánt fájlt. Ez a funkció teljes egészében megegyezik a BOB-Editor *File\Open* funkciójával (6.3.1. fejezet), ezért itt már csak a fájl kiválasztását segítő billentyűket írjuk le, röviden.

Enter	Fájlkiválasztás vagy könyvtárváltás.
ESC	Visszavonás, megnyitás nélküli visszalépés.
A-Z	Lemezmeghajtó-váltás.

Szóköz	Fájl megadása nevének (és útvonalának) begépelésével.
↑,↓	Fájllista gördítése fel és le, egyesével.
Home	Ugrás a fájllista elejére (az első fájlra).
End	Ugrás a fájllista végére (az utolsó fájlra).
Page Up	Egy oldal (22 fájl) ugrás felfele.
Page Down	Egy oldal ugrás lefele.

Save (F2)

Fájl gyorsmentése azonos néven. Ha a térképet eddig még nem mentettük el, a *Save as* funkció lép érvénybe, ahol meg kell adni a nevét. A *Save* csak a térkép adatait menti el, a palettát nem!

Save as (Shift+F2)

Fájl mentése más néven. A műveletből egy üres **Enter** megnyomásával visszaléphetünk. Ha a fájlnak nem adunk kiterjesztést, az automatikusan **'.MAP'** lesz. Ha szükséges, a fájl neve elé gépeljük be az elérési útvonalat is. Ha már létezik a megadott fájl, a program megkérdezi, hogy felül kívánjuk-e írni. Üssünk **Enter**-t vagy **Y**-t, ha igen, mást, ha nem.

A paletta elmentéséről most is külön kell gondoskodni (*Save pal* vagy **F4**).

Save pal (F4)

A paletta mentésénél az előbbiek érvényesek, csak az alapkiterjesztés **'.PAL'**, és természetesen a paletta mentéséről ezután már nem kell külön gondoskodni. A színösszetevők tárolása: 0. szín R, G, B; 1. szín R, G, B; ... (és így tovább egészen 256-ig). Minden színösszetevő egy bájtot foglal, így a PAL-fájl mérete: 3×256=768 bájt.

Load pal (F5)

Ugyanolyan módszerrel választhatjuk ki a betölteni kívánt palettafájlt, mint a két editor *Open* funkciójánál (6.3.1. és 7.3.1. fejezet), csak a sárgán kijelölt fájlok kiterjesztése most **'.PAL'**. Az alappalettát visszaállítani nem itt, hanem az *Options* menü *Default pal* funkciójával lehet.

Exit (Alt-X)

Kilépés előtt a program megkérdezi, hogy valóban ki akarunk-e lépni. Ekkor érdemes végiggondolni, hogy mindent elmentettünk-e (térképet, palettát), és ha igen, nyomjunk **Enter**-t, vagy az **Y** billentyűt.

7.3.2. Az *Edit* (szerkesztési) menü

A felső három funkciójához (*Delete, Cut, Copy*) szükséges egy érvényes kijelölés a térképszerkesztési területen (a jobb oldali egérgomb folyamatos nyomása mellet keríthetünk el egy ilyen téglalap alakú területet). Az *Edit* menü funkciói:

Funkció	**Billentyű**	**Rövid leírás**
Delete	Del	Kijelölt rész törlése.
Cut		Kijelölt rész kivágása.
Copy	Ins	Kijelölt rész memóriába másolása.
Paste		Előzőleg memóriába másolt rész beillesztése.
Clear map		Térkép törlése, 0 sorszámú BOX-okkal történő feltöltése.
Go to	F6, Enter	Ugrás a megadott pozícióba.
Mask		BOX ütközési váz szerkesztése.

Delete (Del)

A kijelölt rész törlése (0-sorszámú dobozokkal felülírása), úgy, hogy azt nem lehet visszaállítani, csak ha újrarajzoljuk. Ha nincs érvényes kijelölés, a funkció hatástalan.

Cut

Kijelölt rész törlése, de annak tartalma a memóriában megmarad egészen addig, amíg ezt a memóriában egy újabb *Cut* vagy *Copy* művelettel felül nem írjuk (mindig a legutolsó *Cut* vagy *Copy* által elraktározott rész tárolódik csak a memóriában). Az elmentett tartomány a *Paste* funkcióval illeszthető be.

Copy (Ins)

Ugyanazt a hatást érhetjük el vele, mint az előző funkcióval, csak a kijelölt rész nem törlődik. Ha a kijelölt téglalapra a bal oldali egérgombbal kattintunk, a program először végrehajt egy *Copy* műveletet, majd egy *Paste* eljárást.

Paste

Ha előzőleg a *Copy* vagy a *Cut* funkció valamelyikével elmentettünk egy téglalapot a térképből, azt a *Paste* művelettel illeszthetjük be egy új helyre, vagy akár egy másik fájlba. Ha előzőleg a memóriába nem mentettünk el ilyen részt, a *Paste* funkció meghívása után hangjelzést hallhatunk. Beillesztés közben **ESC**-vel visszavonhatjuk a végrehajtást, a bal gombbal pedig nyugtázhatjuk azt.

Clear map

Ezzel a funkcióval letörölhetjük a térképet, úgy, hogy a BOX-ok változatlanok maradnak. Az eljárás végrehajtása előtt azonban még rákérdez, hogy biztosak vagyunk-e a dolgunkban (**Enter** vagy **Y**, ha igen).

Go to (F6, Enter)

Meghívása után kiürül a képernyő, majd az első sorból kiolvashatjuk az aktuális pozíciót. Ezek után a program megkérdezi, hova szeretnénk ugrani, először az abszcisszát, majd az ordinátát, és beállítja a térképet, hogy a bal felső sarkában az általunk megadott koordinátájú BOX legyen látható. A koordináták egysége: 1 BOX. Ha valamelyik irányhoz nem írunk be semmit, csak egy **Enter**-t ütünk, annak az értékét a program 0-nak veszi.

Mask

Az ütközési vázat a *Game* egység BOB típusának *CollBOX* függvénye használja (8.23.3. fejezet). A funkció kiválasztása után a bal gombbal jelölhetők ki azok a dobozok, amelyekkel a vizsgált BOB nem érintkezhet, azaz ha ezekkel érintkezik, a *CollBOX* függvény visszatérési értéke igaz (*true*). A jobb oldali gombbal kapcsolható ki a maszk adott BOX-hoz tartozó bitje. Visszalépni bármely billentyű leütésével lehet.

7.3.3. *Options* (opciók) menü

Különféle beállítások végezhetők el az *Options* menü négy funkciójával, melyek röviden összefoglalva a következő táblázatban láthatók:

Funkció	Billentyű	Rövid leírás
Tab size		Elmozdulási egység méretének változtatása.
Map size		Térkép méretének átállítása.
Full screen	Szóköz	Teljes képernyős nézet.
Default pal		Alappaletta visszaállítása

Tab size

A nyílbillentyűkkel történő térképgörgetés nagyságát adhatjuk meg itt, 1 és 23 közé kell esnie, az alapérték: 1. Ha rossz értéket adunk meg, sípolást hallhatunk, majd újra kell próbálkoznunk.

Map size

A térkép méreteit változtathatjuk ezzel a funkcióval. A legfelső sorban olvashatjuk az aktuális méreteket, miközben be kell gépelni az új szélességet, majd a magasságot. Csupán egy **Enter** megnyomására a művelet visszavonható. Ha növeljük vagy nem változtatjuk valamelyik irányhoz tartozó méretet, a térkép tartalma nem változik, az új részre 0 sorszámú dobozok kerülnek. Viszont ha csökkentjük szélességét vagy magasságát, az így levágott részek elvesznek, ezért legyünk óvatosak!

Full screen (szóköz)

A MAP megjelenítése az egész képernyőn. A szerkesztési területen a bal felső sarokban található BOX kerül a képernyő bal felső sarkába. Billentyű- vagy gombnyomással lehet visszalépni.

Default pal

VGA-alappaletta visszaállítása. A funkció kiválasztása után megjelenő kérdésre igennel kell válaszolni (**Enter** vagy **Y** billentyű), ha az alappalettát valóban vissza szeretnénk állítani, bármely más billentyű vagy egérgomb hatására visszavonhatjuk a műveletet.

7.4. Funkcióbillentyűk

Mivel a *Help* menü egyik funkciója sem működik, az egyes műveletek gyors elérésére szolgáló billentyűket az alábbi táblázatban foglaljuk össze. Néhány

billentyű csak bizonyos feltételek mellett használható (pl. érvényes kijelölés, megnyitás közben), ezeket külön jelezzük.

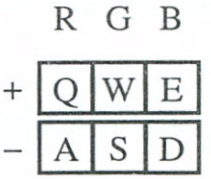
F2	Fájl mentése.
Shift+F2	Fájl mentése új néven.
F3	Fájl megnyitása.
F4	Paletta mentése.
F5	Paletta betöltése.
F6, Enter	Ugrás.
Szóköz	Teljes képernyős nézet.
Alt-X	Kilépés.
Nyilak	Térkép mozgatása, az *Optoins* menü *Tab size* funkciójánál megadott egységgel. Eleinte ez 1.
Ctrl-→/Ctrl-←	Térkép mozgatása tízesével jobbra/balra.
Page Up/Down	Térkép mozgatása tízesével fel/le.
Del	Kijelölt rész törlése (érvényes kijelölés szükséges).
Ins	Kijelölt rész memóriába másolása (érvényes kijelölés szükséges).

Megnyitás vagy palettabetöltés közben alkalmazható billentyűk

Enter	Megnyitás.
ESC	Visszavonás.
A-Z	Meghajtóváltás.
Szóköz	Fájlnév begépelése.
↑,↓	Fájllista gördítése fel, le.
Home	Első fájl.
End	Utolsó fájl.
Page Up/ Down	Egy oldal ugrás fel/le.

Színösszetevők változtatására szolgáló billentyűk

Q	Vörös (*Red*) növelése.
A	Vörös csökkentése.
W	Zöld (*Green*) összetevő növelése.
S	Zöld csökkentése.
E	Kék (*Blue*) komponens növelése
D	Kék csökkentése



7.5. További lehetőségek

A MAP-Editort is paraméterezhetjük, éppúgy, mint a BOB-Editort. Ha csak egy paramétert adunk meg, az annak a térkép-fájlnak a neve, ami indítás után automatikusan betöltődik. Két paraméter esetében az első ugyanez, míg a második az automatikusan betöltődő palettafájl neve.

Egy **MAPEDIT.COL** nevű fájl létrehozásával megváltoztathatók az editor előtérszínei. A program futtatás után megvizsgálja, hogy van-e ilyen nevű fájl az aktuális könyvtárban, ha van, annak megfelelően állítja be a színeket, ha nincs, akkor az előtér fehér lesz, ami meglehetősen egyhangú. A szerkesztő különböző egységei (keretek, menük stb.) más-más színűek lehetnek, a fájl soraiban ezeket kell megadni. Szöveges legyen az adatállomány, minden sorába kerüljön egy 0 és 255, vagy ezek közé eső egész szám, és semmi más. Összesen 11 sorból áll, ezek a következő színeket tartalmazzák:

1. Szövegek, információk színe (pl. koordináták az állapotsorban).
2. Keretek, például a szerkesztési területet körülölelő ponthalmaz színe.
3. Paletta Scroll gombjainak (UP, ill. DOWN felirattal) betűszíne.
4. Palettagörgető gombok alapszíne.
5. Menük, funkciók betűinek színe.
6. Menük, funkciók alapszíne.
7. Kiválasztott menü vagy funkció előtérszíne.
8. Kiválasztott menü vagy funkció háttérszíne.
9. Kérdező ablakok alapszíne (ilyen ablak jelenik meg például a kilépés előtt).
10. Kérdező ablakok betűszíne.
11. Kurzor színe, a kurzor egy négyzet alakú keret, ami körülveszi az egérmutató által kijelölt BOX-ot.

A lemezmellékleten található **COLORS.PLD** fájlt nevezzük át **MAPEDIT.COL**-ra, és ezek után indítva az editornak sokkal kellemesebb színei lesznek.

A MAP-Editor a térképet olyan formában menti el, hogy az a *Game* egység *LoadMAP* (8.10.) eljárásával saját játékban is felhasználható.

8. A GAME unit

E fejezetben leírt *Game* egység egy olyan programkönyvtár, melynek eljárásai, függvényei nagy segítséget nyújtanak egy játék elkészítéséhez. Maga a forrásszöveg nagy terjedelme miatt itt, a könyvben nem található meg, csak a könyvhöz mellékelt lemezen egy **GAME.PAS** nevű fájlban. A unit nem használ más, általunk készített unitot, csak a *Crt* és a *DOS* egységeket, melyek a Turbo Pascal rendszer részét képezik. Így a GAME.PAS lefordításához nem szükséges semmilyen más fájl, feltéve persze, hogy a *Crt* és *DOS* unitok a rendszer számára elérhetők. (A Turbo Pascal telepítése után ez a két unit a TURBO.TPL fájlban található meg, és ha nem töröljük őket onnan a *tpumover* programmal, akkor a *Crt* és a *DOS* mindig hozzáférhető.)

A *Game* egység főként a kétdimenziós játékhoz szükséges grafikai megoldásokat biztosítja, de megtalálhatók benne például ütközéseket vizsgáló függvények is. A unit által készíthető játékok a VGA kártya MCGA üzemmódját alkalmazzák, ezért grafikájuk felbontása nem túl jó (320 pixel vízszintesen, 200 pixel függőlegesen), viszont az egyszerre megjeleníthető színek száma (256) már kielégítő.

Az egységben eddigi tapasztalatainkat foglaljuk össze és bővítjük. Sok olyan szubrutint tartalmaz, aminek forráskódja a könyvben is megtalálható, ezek helyére mindig utalni fogunk. Viszont akadnak olyan eljárások, függvények is, amelyek teljesen újak vagy csak érintőlegesen foglalkoztunk velük. Például a palettáról csak az 1. fejezetben volt szó, ezzel szemben a unitban található egy elsötétítő (8.6.) és egy kivilágosító (8.19.) eljárás, melyek már sokkal bonyolultabbak egy egyszerű színbeállításnál.

A *Game* unit eljárásainak, függvényeinek és változóinak egy része a *BOB* objektumtípusban található meg, ennek a típusnak leírását és használatát a többi eljárástól külön, a 8.23. részben találhatjuk meg. Az alábbiakban először azokat a szubrutinokat ismertetjük, amelyek közvetlenül elérhetők, majd a *BOB* típus leírása következik, végül pedig az egység *Interface* részében megtalálható globális változók és konstansok bemutatására kerül sor.

8.1. DoneGame eljárás: az egység lezárása

Szintaxis: DoneGame;

Lezárja a *Game* egység használatát, felszabadítja a lefoglalt memóriatartományokat és visszaállítja a szöveges üzemmódot. Működése ellentétes az *InitGame* eljárással. A hasonlóság köztük annyi, hogy mindkettőt csak egyszer szabad használni egy programban, mivel a Turbo Pascal memóriakezelése hagy némi kívánnivalót maga után. Ha tehát egy játék közben valamiért szöveges képernyőt szeretnénk használni (pl. toplista kiírása), semmiképp sem ajánlatos a text üzemmódot a *DoneGame* eljárással beállítani, inkább használjuk a 10h megszakítást a következőképpen:

```
asm
  mov ax,3
  int 10h
end;
```

vagy az *MCGA* unit *SetMode* eljárását (6.1.1. fejezet). A *DoneGame* ugyanis felszabadítja a megjelenítéshez szükséges memóriát (háttér és munkaterület), ráadásul szabaddá teszi a Shape-ek tárolásához felhasznált bájtokat is. És a tapasztalatok azt mutatják, hogy egy Turbo Pascalban írt program memóriakezelés szempontjából akkor lesz jó, ha az elején lefoglaljuk az összes szükséges memóriát, és csak a végén szabadítjuk fel, vagy a szabaddá tételével egyáltalán nem is törődünk.

8.2. DoneKey eljárás: BIOS billentyűzetmegszakítás vissza

Szintaxis: DoneKey;

Az InitKey eljárással módosított billentyűzetmegszakítás vektort visszaállítja az eredeti címére, vagyis ezek után ismét a BIOS-megszakítás lesz aktív. Csak akkor hívjuk meg a *DoneKey* eljárást, ha előtte párját, az *InitKey*-t már lefuttattuk, ellenkező esetben a rendszer lefagyhat. A módosított billentyűzetmegszakítás lényege, hogy egyszerre több billentyű állapota figyelhető, folyamatosan. Részletesebben a 3.1. részben olvashatunk róla. Minden olyan program végén meg kell hívni ezt az eljárást, amiben az *InitKey*-t is meghívtuk, különben a rendszer lefagyhat, mert nem állítottuk vissza az általa használt megszakításvektort. Előfordulhat, hogy utána a billentyűzet nem működik helyesen, ekkor nyomjuk meg mindkét *Ctrl* billentyűt egyszerre.

Az *InitKey* és *DoneKey* eljáráspárt akárhányszor meghívhatjuk egy program során, arra azonban ügyelni kell, hogy egymás után kétszer ne futtassuk le csak az egyiket. A *DoneKey* kikapcsolja az *InitKey* által beállított megszakítást, amire akkor lehet például szükség, amikor valamilyen adatot kell bevinni (név, szám stb.). Ekkor sokkal egyszerűbb a BIOS megszakítását alkalmazni, ami gépelésre, adatbevitelre sokkal megfelelőbb.

8.3. DrawBox eljárás: egy BOX megjelenítése

Szintaxis: DrawBox(P: pointer; B, X, Y: word);

Egy BOX-ot rajzol ki a megadott helyre. P annak a memóriatartománynak a kezdőcíme, ahová a BOX kerül. Ez lehet például a háttér (*Background*), a munkaterület (*WorkArea*) vagy a képmemória ($A000:0). B-ben adjuk meg a BOX sorszámát, X-ben és Y-ban pedig a koordinátáit. Ez utóbbiak maximális értéke 311, illetve 192, vagyis a koordináta-rendszer alapegysége 1 pixel. Ellenőrzés nincs, ezért rossz paraméterbeállítással a rendszer akár le is állítható.

A *DrawBox* eljárással megvalósítható, hogy legyen a háttérnek olyan része, ami valójában előtér, mert minden más grafikus elem fölött helyezkedik el. Ezt úgy kell megvalósítani, hogy először összeállítjuk a képet a munkaterületen a *MakeScr* eljárással (8.12. rész), de még nem jelenítjük meg, hanem meghívjuk a *DrawBox* szubrutint, aminek első paraméterében a munkaterület kezdőcímét adjuk meg. Például így:

```
MakeScr;
DrawBox( WorkArea, 2, 220, 150);
ShowScr;
```

Ekkor a megjelenő képen lesz egy BOX, melynek a sorszáma 2, bal felső sarkának képernyő-koordinátái pedig (220;150). Ez a doboz minden alatta levő BOB-ot eltakar, így úgy tűnik, mintha legfelül lenne. Ez a módszer alkalmazható például egy hajós játékban, amikor a hajó a híd alatt úszik át. De vigyázzunk, minél nagyobb az a rész, ami legfelül van, azaz minél több BOX-ot rakunk ki egyszerre egy ciklusban, annál lassabb lesz a program futása.

8.4. FillBack eljárás: háttér beszínezése

Szintaxis: FillBack(Color:byte);

Beszínezi a hátteret az eljárás paraméterében megadott színűre. Az eljárás nagyjából megegyezik a 4. fejezet elején ismertetett eljárással. Az *InitGame* eljárás is meghívja a *FillBack*-et, paraméterként a *BackColor* változót adja meg, aminek kezdőértéke 0.

8.5. GrayPal eljárás: szürkeskálává konvertálás

Szintaxis: GrayPal(First, Last: word);

Szürkeskálát állít elő a paletta bizonyos tartományából. A tartomány első színét az eljárás első, míg az utolsót a második paraméterében kell megadni. Akkor lehet az eljárásra szükség, ha színes monitorra írt játékot monokróm monitorral rendelkező gépen akarunk futtatni. Azok a színek ugyanis, amelyek nem tartalmaznak zöld színösszetevőt, ezeken a monitorokon nem látszanak. Az eljárás egyszerűen meghívja a BIOS 10h sorszámú megszakítását, a megfelelő paraméterekkel.

8.6. HidePal eljárás: színek egybemosása

Szintaxis: HidePal(Speed: word; First, Last, R, G, B: byte);

A *First* és a *Last* paraméterek által meghatározott színtartomány színeit fokozatos átmenettel azonos színűvé „varázsolja". Ennek a színnek az összetevőit adjuk meg az utolsó három paraméterben (R, G, B). Ha ezek mindegyike 0, a kép fokozatosan, de egyenletesen elsötétül, fekete lesz. A folyamat sebességét az első paraméterrel szabályozhatjuk, ami valójában nem a sebesség nagyságát, hanem a késleltetési időt tartalmazza. Ha ez nulla, akkor lesz a leggyorsabb, és ha növeljük ezt a számot, a sebesség csökken, lassabban tűnik el a kép. Fontos azt megjegyezni, hogy az eljárás végrehajtása alatt a program áll, nem lehet irányítani vagy más eljárásokat meghívni. A vezérlés csak a színek egybemosása után kerül vissza a főprogramhoz. Az aktuális színösszetevőket az eljárás eltárolja a *Colors* globális változóba, hogy a *ShowPal* eljárás vissza tudja őket állítani.

A *HidePal* eljárást például a játék végén vagy a főhős halálánál alkalmazzuk. Párjával, a *ShowPal* eljárással szép képváltásokat valósíthatunk meg: például

ha a játék főszereplője átmegy az egyik szobából a másikba, kilépéskor elsötétül a kép, ezután átrajzoljuk a hátteret (ha térképet használunk, akkor a *NewPos* eljárással), és újra kivilágosítjuk a *ShowPal* eljárással, csak most már a következő szoba lesz látható.

Megjegyzés: a *HidePal* nem módosítja a 0. színt, mert az nem szép. Ha 0 színű, de fekete képpontokat tartalmaz a képernyő, akkor a *HidePal* eljárás utolsó három paramétere legyen mindig 0, mert ha ettől eltérő, akkor a keletkezett egyszínű kép fekete pontokat tartalmaz. Legegyszerűbb a probléma megoldása akkor, ha a 0. szín fekete, azaz összetevői: **0-0-0**, és ezt a színt nem használjuk a grafika pixeleihez.

8.7. InitGame eljárás: az egység inicializálása

Szintaxis: InitGame;

Inicializálja a *Game* unitot, lefoglalja a háttérhez és a munkaterülethez szükséges 2×64000 bájtot, bekapcsolja az MCGA üzemmódot, a *BackColor* változóban megadott színűre festi a hátteret. Mielőtt elkezdjük használni a *Game* egység eljárásait, függvényeit, meg kell hívni az *InitGame* eljárást. Azonban ügyeljünk arra, hogy egy programban csak egyszer szerepeljen, a Pascal memóriakezelésbeli tökéletlensége miatt. Ha egy játék közben valamiért átváltunk egy másik VGA üzemmódba, semmiképp se használjuk az *InitGame*-et az MCGA módba való visszatéréshez, alkalmazzuk helyette inkább a következő assembly-sorokat:

```
asm
  mov ax,13h
  int 10h
end;
```

vagy az *MCGA* egység (6.1.) üzemmód-beállító eljárását: `SetMode($13);`

A *Game* egység használatát az *InitGame* eljárás nyitja, és a *DoneGame* eljárás zárja, működésük tehát hasonlít a *Graph* unit *InitGraph* és *CloseGraph* eljárásaihoz. Ha úgy indítjuk el például a *MakeScr* eljárást, hogy előtte nem hívtuk meg az *InitGame*-et, az lefagyáshoz vezet, tehát nagyon fontos minden *Game* unitot használó játékprogram elejére beírni egy

```
InitGame;
```

sort. Ajánlatos mindjárt ezzel kezdeni a programot.

8.8. InitKey eljárás: módosított billentyűzetmegszakítás

Szintaxis: InitKey;

Bekapcsolja a módosított billentyűzetmegszakítást, vagyis a $09 megszakításvektort egy saját eljárásra irányítja, a *NewIRQ*-ra. Ez a programrész megegyezik a 3.1. fejezetben leírt és a lemezmellékleten megtalálható **KEYB3.PAS** példaprogram *NewIRQ* elnevezésű eljárásával. Közvetlenül nem érhető el (a unit implementációs részében van), de nincs is rá szükség.

Ha meghívjuk az *InitKey*-t, azontúl a billentyűk állapotáról a *Key* logikai tömbből kaphatunk információt. Ez a tömb 256 elemű, a unit *Interface* részében lett deklarálva, a következőképpen:

```
var
    Key: array [0..255] of boolean; .
```

Mindegyik billentyűhöz tartozik egy szám (egy SCAN-kód), ha lenyomunk egy billentyűt a hozzá tartozó *Key* elem igazra (*true*) vált, ha nincs lenyomva, a változó értéke hamis (*false*). Hogy melyik billentyűhöz melyik *Key*-elem tartozik, a 3.1. rész végén találhatjuk meg (közvetlenül a 3.2. fejezet kezdete előtt), és a lemezmelléklet **SCANCODE.TXT** fájljában.

Az eredeti megszakítást a *DoneKey* eljárással állíthatjuk vissza. Ha ezt programunk végén elmulasztjuk, az óhatatlanul lefagyáshoz vezet. A módosított megszakítás mellett ne használjuk a *Crt* unit billentyűzetkezelő függvényeit (*ReadKey* és *KeyPressed*), mert ez szintén a rendszer azonnali munkabeszüntetését vonja maga után. Helyettük inkább e unit *_ReadKey* ill. *_KeyPressed* függvényeit alkalmazzuk.

8.9. LoadLBM eljárás: kép betöltése

Szintaxis: LoadLBM(FileName: String; P: pointer);

Egy **LBM** kép betöltésére szolgál. A *FileName* paraméterben adjuk meg az LBM-fájl nevét, elérési útvonallal, ha szükséges, a P paraméterben pedig a célcímet, annak a tartománynak a kezdőcímét, ahová az LBM kép kerül. Ha ez a háttér kezdőcíme (*Background*), akkor a lemezen található LBM formátumban tárolt képet játékunk hátttereként jeleníthetjük meg, de megadhatjuk második paraméternek a képernyőt is: *PTR($A000,0)*, ekkor a kép rögtön megjelenik,

ami például kezdőképek kirajzolására alkalmas. A *LoadLBM* eljárás egyben betölti a paraméterében megadott fájlban tartalmazó palettát is. A forráskódja megegyezik a 4.1. fejezetben található programszöveggel. Ügyelni kell arra, hogy a fájlban tárolt kép 256 színű és 320 pixel szélességű legyen, mert az eljárás nem ellenőrzi ezeket az értékeket

8.10. LoadMap eljárás: térkép betöltése

Szintaxis: LoadMap(FileName: string);

Paraméterként azt a fájlt kell megadni, ami a térképet tárolja, kiterjesztése általában **.MAP**. A térképszerkezetről részletesebben a 4.2. fejezetben olvashatunk, a térképszerkesztő segédprogrammal (MAP-Editor) pedig a 7. fejezet foglalkozik. A MAP-Editorral készített és elmentett térképek betöltésére alkalmas ez az eljárás, a fájl bájtjainak helyet foglal, és elhelyezi őket a memóriában, hogy azok a *DrawBox*, *NewPos* és *Scroll* eljárások számára hozzáférhetők legyenek. Az eljárás hatására az alábbi változók kapnak értéket:

Változó	Típus	Tartalom
BoxPtr	pointer	BOX grafikus adatainak kezdőcíme.
MapPtr	pointer	Térkép kezdőcíme.
BPL	word	Egy térképsorban lévő dobozok száma (*Boxs Per Line*).
MapLength	word	Térkép hossza bájtban, dobozainak száma (1 bájt–1 doboz).
Mask	tömb, 16×2 bájt	BOB-BOX ütközési váz.

Egy programban általában csak egyszer alkalmazható a *LoadMap* eljárás, mivel a lefoglalt memóriát nem szabadítja föl, így hamar betelik a rendelkezésre álló szabad heap. Ha mégis új térképet szeretnénk a régi helyébe tölteni, azt megtehetjük elvileg az alábbi módon. Ez azonban a gyakorlatban nem alkalmazható, mert a Turbo Pascal memóriakezelése nem tökéletes.

```
freemem( MapPtr, MapLength);
freemem( BoxPtr, 16384); { 256 db 8×8-as BOX helyigénye 16384 bájt }
LoadMap( 'UJFILE.MAP');
```

Végezetül nézzük meg még egyszer, hogyan épül fel egy MAP fájl. A másik megoldás arra, hogy egy programon belül több térképet is megjeleníthessünk,

az lehet, hogy írunk egy saját térképbetöltő eljárást. Ehhez nyújt segítséget az alábbi táblázat.

Cím	Hossz	Tartalom
0.	1	Verziószám. Ez ellenőrzésre is felhasználható, ha értéke nem 1, biztos, hogy nem MAP-Editorral készített térképpel van dolgunk.
1.	2	Egy sorban található dobozok száma (térkép szélessége), a *BPL* változóba kell tölteni.
3.	2	Térkép hossza (bájtban vett helyfoglalása), MapLength változó.
5.	16384	BOX adatok, sorrendben, sorfolytonosan (*BoxPTR*).
16389.	[3.]	Térképadatok, sorfolytonosan (*MapPTR*).
végül	32	Maszk, ütközési váz. Minden bit egy BOX-hoz tartozik (*Mask*).

8.11. LoadPal eljárás: paletta betöltése

Szintaxis: LoadPal(FileName: string);

Betölt és megjelenít egy lemezen tárolt palettát. Egyetlen paraméterében annak a fájlnak a nevét (elérési útvonallal, ha kell) adjuk meg, ami a paletta színösszetevőinek adatait tartalmazza. Általában **.PAL** a kiterjesztése. Legalább 768 bájt hosszúságúnak kell lennie, mert 256 szín van, és minden szín három színösszetevőjét 1-1 bájt határozza meg. A BOB-Editorral (6. fejezet) és a MAP-Editorral (7. fejezet) készíthetünk ilyen palettafájlokat. Felépítésének elve a következő:

Cím	Tartalom
1.	0. szín vörös (*Red*) összetevője.
2.	0. szín zöld (*Green*) komponense.
3.	0. szín kék (*Blue*) alkotóeleme.
4.	1. szín vörös összetevője.
...	...
768.	255. szín kék összetevő.

8.12. MakeScr eljárás: megjelenítés előkészítése

Szintaxis: MakeScr;

A munkaterületen (*WorkArea*) összeállítja a megjelenítendő képet, ezt már csak át kell másolni a képmemóriába, a *ShowScr* eljárással. A megjelenítés elve nagyban hasonlít a 2.5. rész *ShowBOB* eljárásának elvéhez. A *MakeScr* ugyanazt végzi el, mint a *ShowBOB* az első két lépésben, azaz letakarítja a munkaterületről a BOB-okat (felülírja azt a háttérrel), és kirakja őket újra, de most már más helyre, ha koordinátáik megváltoztak. Megjegyzés: az első lépésben az egész hátteret átírja a *rep movsw* segítségével. Ez alapállapotban 32000 szó mozgatását jelenti, ami meglehetősen időigényes. Azonban mindenképp szükség van rá a tetszőlegesen változó háttér miatt, bármikor kirajzolhatunk egy pontot a háttérre a *PixelBack* eljárással, és a görgetéshez is elengedhetetlen a szabad háttér. Ha túl lassú a gépünk (80 MHz alatt), vagy túl sok a memóriarezidens program, melyek sűrűn szakítják meg a futást, valahogy gyorsítani kell az eljáráson. Például csökkenthetjük a kép méretét a *SetArea* eljárással (8.17. rész), így a ciklusonkénti átírandó területek nagysága is csökken. Nehezebb megvalósítani, hogy egy új *MakeScr* eljárást írunk. Alapját a 2.3. fejezet *ShowBOB* eljárás szolgáltathatja, de természetesen ki kell egészíteni a 2.4-5. részekben megismert lehetőségekkel (tetszőleges méret, animáció stb.). A változtatható háttérrel szakítanunk kell, hiszen a háttérnek csak a BOB-ok által takart részét írjuk vissza a munkaterületre, ez a sebességnövekedés lényege.

Joggal kérdezhetjük, hogy miért kellett a megjelenítő eljárást két részre (*MakeScr* és *ShowScr*) bontani. A válasz egyértelmű: mert így több előnye van, mint hátránya. Hátránya csupán annyi, hogy kétszer kell eljárást meghívni a művelet végrehajtásához. Előnye pedig, hogy a kész munkaterületre írhatunk, így az összes grafikai elem „fölé" helyezhetünk háttérelemeket, adatokat, bekeretezhetjük az egész képet, és a görgetés a kereten belül folyik stb. Kiírhatjuk akár középre is, hogy hány pontot ért el a játékos, és hogy hány élete van még. Csak sajnos egy ilyen munkaterületre író eljárás elkészítéséről az Olvasónak kell gondoskodnia. Alapul az *MCGA* egység *OutText* eljárása szolgálhat, melynek leírása a 6.1.1. fejezetben található meg. Nem is kell benne sokat változtatni csak annyit, hogy a célcím ne a képernyő, hanem a munkaterület legyen.

8.13. NewPos eljárás: ugrás a térképen

Szintaxis: NewPos(X, Y: word);

A háttér bal felső sarkát a térkép (X;Y) koordinátájú pontjába állítja. Meg is jeleníti a háttéren a térképrészletet, így az eljárás meghívása után a háttér már a térkép visszafejtett pixeleit tartalmazza. Az eljárás paramétereinek megfelelően állítja be a *PosX* és *PosY* térképkoordináta-jelző változókat. Ha csupán ennek a két változónak az értékét változtatjuk, a háttér még változatlan marad. Ezért minden térképet használó program elején, a térkép betöltése után hívjuk meg ezt az eljárást, ha azt szeretnénk, hogy a térkép rögtön látható legyen.

Önmagában görgetésre nem alkalmas, mert túl lassú, a térkép kis lépésenkénti mozgatására a *Scroll* eljárást használjuk (8.16. rész). Jól használható például akkor, ha a térkép szobákra van felosztva, és a főhős átmegy az egyik szobából a másikba. Ez gyorsabb és kevesebb helyet igényel, mint az LBM-es módszer.

8.14. PixelBack eljárás: háttérpont megváltoztatása

Szintaxis: PixelBack(X, Y: word; C: byte);

A háttér egy bájtját C-re változtatja. A *BackGround* változóban tárolt cím utáni 320×Y+X. bájtot állítja át. Hasonlít a lemezmelléklet **PUTPIXEL.PAS** programjában található *PutPixel* eljáráshoz (1.2. fejezet), csak a célcím nem a képernyő ($A000:0), hanem a háttér (*BackGround*). Tartományvizsgálat nincs, ezért vigyázzunk, hogy az X és Y paraméterek által meghatározott bájt ne essen a háttéren kívülre (biztos nem lesz baj, ha 320×Y+X<64000).

Folyamatosan változó háttér megvalósításához használható fel ez az eljárás, de ügyeljünk arra, hogy nagy változások jelentősen lassíthatják a program futását. Erre is van megoldás: úgy kell összeállítani a háttér pontjait, hogy palettamódosítással animációt lehessen előidézni.

8.15. RetRace eljárás: várakozás vertikális visszafutásra

Szintaxis: RetRace;

Felfüggeszti a program futását mindaddig, amíg az elektronsugár a kép alsó sarkába nem ér. Ha közvetlenül a *ShowScr* eljárás előtt hívjuk meg, progra-

munk szép és egyenletes lesz, igaz, egy kicsit lassabb is. Ha minden megjelenítés előtt meghívjuk, kiküszöböljük az abból eredő problémát, hogy a gépek különböző sebességűek. Ha nagyon gyors egy gép, elvileg nagyon gyors lenne a játék is, viszont minden ciklusban várakoznia kell a *RetRace* eljárás miatt. Ezért a program sebessége nem a géptől, hanem a videokártyától függ, a képváltási frekvenciától. Szerencsére ez nagyjából állandó (60-70 Hz).

Ha túl lassú a gépünk (kb. 80 MHz alatt), előfordulhat, hogy az elektronsugár beéri a memóriába írást, mivel egy ciklusban nagyon sok adatot kell átmozgatni (ált. 2×32000 szó). Ez szintén darabossághoz vezet, ráadásul sokkal zavaróbb, mint ha a *RetRace* eljárást nem alkalmaznánk, mivel a törésvonal nagyjából állandó helyen lesz.

Az elektronsugárról részletesebben az 1.4. fejezetben olvashatunk. Az ott található *RetRace* eljárás megegyezik a *Game* unit elektronsugár-figyelő eljárásával. A **$3da** regiszterről beolvasott bájtnak jobbról a negyedik bitje **1**, ha az elektronsugár vertikálisan visszafut, egyébként **0**. Ezt a bitet figyeli az eljárás.

8.16. Scroll eljárás: térkép gördítése

Szintaxis: Scroll(Dir: byte; Count: word);

A térképet kis lépéssel mozdítja el, ha többször meghívjuk egymás után (mondjuk, egy cikluson belül), folyamatosan gördíti a hátteret. Első paraméterében adjuk meg az irányt, másodikban pedig az elmozdulás mértékét, pixelben. Az irány megadását a következő konstansok segítik:

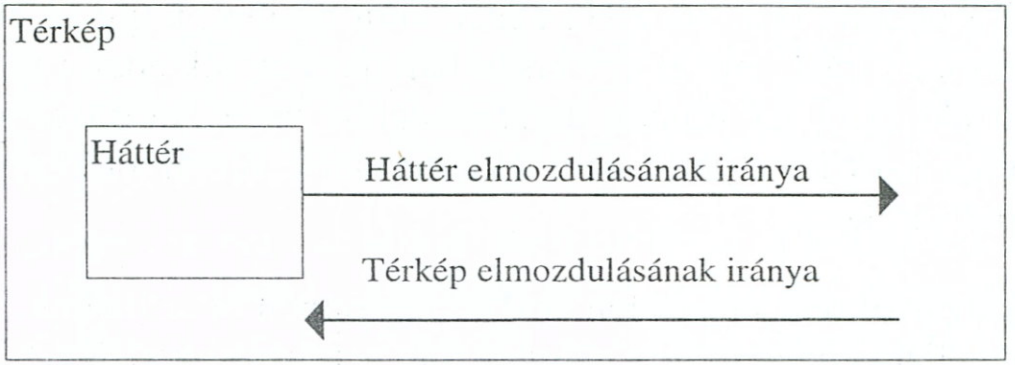
```
Up    = 8;  { Fel    }
Down  = 2;  { Le     }
Left  = 4;  { Balra  }
Right = 6;  { Jobbra }
```

(Az egyes számokhoz tartozó irányt a numerikus billentyűzetről is leolvashatjuk. Ha a nyíllal megjelölt billentyűkön található számot írjuk az eljárás első paraméterébe, akkor a térkép a nyíl irányában mozdul el.)

A *Scroll* eljárás egyesíti a 4.6. fejezetben tárgyalt négy eljárást. A *PosX* és *PosY* térkép-koordinátákat jelző változókat megváltoztatja, azonban a két változó értékének módosítása nem elég ahhoz, hogy a háttéren is megjelenjenek a változások. Az eljárás meghívása jelentősen lassítja a program futását, mivel a

háttér tartalmát, annak az összes bájtját átmozgatja. Szép, egyenletes gördítéshez legalább 80-90 MHz-es gép kell, ez alatt ugyanis a mozgás darabos.

Hogy valójában mit jelent a görgetés, az ábra segít megérteni. Itt éppen jobbra gördül a háttér, tehát a térkép hozzá viszonyítva balra mozdul el, ezért első paraméterként a *Left* konstanst kell megadni.



Egy megjelenítés során a térképnek a kisebbik téglalapba (háttér) eső része lesz látható. A gyakorlatban ez ebben az esetben úgy oldható meg a leggyorsabban, hogy a háttér minden pixelét balra toljuk eggyel (az első oszlopot természetesen nem), és a jobboldalt felszabaduló oszlopba visszafejtjük a tömörített térképadatokat.

Ez az eljárás csak négy irányban tud görgetni. Átlós mozgatás az eljárás egymás után kétszeri meghívásával valósítható meg, de ez túl lassú, a program sebessége ebben az esetben a felére csökken. Gyors gépeknél természetesen ez nem probléma, viszont lassú gépekre az átlós irányú gördítés kifejlesztése már az Olvasó feladata. A MAP-képszerkezetről és a gördítésről részletesebben a 4.2-7. részekben olvashatunk.

8.17. SetArea eljárás: kép magassága

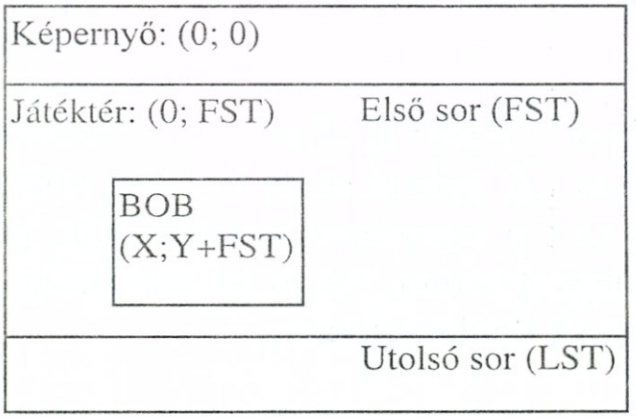
Szintaxis: SetArea(Fst, Lst: word);

A megjelenő kép függőleges méretét, vagyis annak kezdő és záró sorát állíthatjuk ezzel az eljárással. Ebből következik, hogy a háttér és a munkaterület mérete is változhat, így csökkenthetjük a ciklusonkénti átmozdítandó bájtok számát. Az eljárás első paraméterében adjuk meg az első olyan képernyősort, amire a *ShowScr* eljárás ír, a másodikban pedig az utolsót. Értelemszerűen az első paraméter nem nagyobb a másodiknál. Értéküknek a képernyőkoordi-

nátáknak megfelelően a [0..199] intervallumba kell esniük. Teljes képernyős módhoz használjuk a (0, 199) számpárt.

Sok előnye van annak, ha játékunk nem az egész képernyőt használja, hanem annak csak egy vízszintes sávját. A játékkép felett és alatt elhelyezkedő területet a *ShowScr* eljárás nem módosítja, ide bármit írhatunk, az változatlan marad. Tehát felhasználhatjuk ezt a részt például dekorációra, de az állapot megjelenítésére is alkalmas (pontszám, energia, idő stb. – ehhez ajánljuk az *MCGA* unit *OutText* eljárását, 6.1.1. rész). Másrészt ha csökkentjük a kép méretét, növekszik a játék sebessége, mivel egy cikluson belül egy sor csökkentése után 2×160 szóval kevesebb adat kerül átmozgatásra.

A kép magasságának csökkentése semmilyen következményt nem okoz, a BOB-ok nem úsznak ki a játéktéren kívüli részre, teljesen olyan mintha magát a képernyőt szűkítenénk. Ha az első paraméter nem nulla, a koordináta-rendszer egy kicsit megváltozik. Kezdőpontja nem egyezik meg a képernyő kezdőpontjával, hanem annál lejjebb lesz. De ez csak annyiban jelenthet problémát, hogy nehezebb a képernyő koordináta-rendszeréhez viszonyítani. A következő ábra bemutatja egy BOB képernyő-koordinátáinak meghatározását:



A *Game* unit és a könyv első felében található eljárások közti különbségének oka ez a változtatható játékképmagasság. Például a *ShowScr* eljárásban nem szabad mindig 32000 szót megjeleníteni, mert ez mindjárt a felére csökken például egy *SetArea*(0, 99); utasítással.

A paraméterként megadott változók később visszaolvashatók a *First* és a *Last* globális változókból. Ezek változtatásával a játékkép magassága nem állítható, ezeket a változókat inkább csak olvasásra használjuk.

8.18. SetRGB eljárás: színösszetevők változtatása

Szintaxis: SetRGB(ColNum, R, G, B: byte);

Az eljárás – működését tekintve – teljes egészében megegyezik a *Graph* egység *SetRGBPalette* eljárásával. Az első paraméterben megadott szín összetevőit állíthatjuk be vele. Az **R, G, B** paraméterek határozzák meg a vörös, zöld és kék színösszetevőket. Ezeknek értéke legfeljebb 63 lehet, ha ennél nagyobb, akkor csak az alsó 6 bit számít. Az eljárás felhasználható pl. egyéni palettabetöltésre vagy elsötétítésre. A palettáról részletesebben az 1.6. fejezetben olvashatunk.

8.19. ShowPal eljárás: kivilágosítás

Szintaxis: ShowPal(Speed: word; First, Last: byte);

Az eljárás első megközelítésben a *HidePal* (8.6.) eljárással eltüntetett palettát jeleníti meg ismét. A megjelenítés sebessége kerül az első paraméterbe, ami valójában nem sebesség, hanem késleltetési idő (ha ez 0, akkor a leggyorsabb). Az utolsó két paraméter azt a színtartományt határozza meg, amelynek színei megváltoznak, ezek értéke általában 1 és 255. A második paraméter nem lehet nagyobb a harmadiknál.

Ez az eljárás azonban nemcsak a *HidePal* eljárással egybemosott paletta lassú visszaállítására szolgál, hanem két paletta szép, lineáris váltása is megoldható vele. A régi palettát a VGA kártya regiszterei tartalmazzák, például a *LoadPal* (8.11.) eljárással lehet ezt megvalósítani. Az új paletta pedig a *Colors* változó bájtjaiban foglal helyet, melynek deklarációja:

```
type
 RGBType = record
  Red, Green, Blue: byte;
  end;

var
 Colors: array [0..255] of RGBType;
```

Ha az új palettát elhelyeztük a *Colors* változóba az alábbi utasítással tudjuk lineárisan megjeleníteni. (Lassítás végett írjunk az első paraméterbe nagyobb számot!)

```
ShowPal( 0, 1, 255);
```

A középső paraméter lehet nulla is, de ez hatástalan, mert sem a *HidePal*, sem a *ShowPal* nem változtatja a keret színét és a 0 színű képpontokat, mert az nem lenne szép. Éppen ezért a 0. szín legyen mindig fekete, és lehetőleg ne használjuk fel a grafikához.

Lassan eltüntethető bármelyik paletta a *HidePal* eljárással, lassú megjelenítésre mutat be egy példát az alábbi program.

```
{palette.pas}

uses Crt, Game;
var
 f: file;                  { Változó a paletta betöltéséhez          }
 i: word;                  { A FOR ciklusokhoz                       }

begin
 assign( f, 'default.pal');
 reset( f, 1);
 blockread( f, Colors, 768);
                           { Paletta beolvasása - még nem látható    }
 close( f);
 asm
  mov ax,13h
  int 10h                  { MCGA üzemmód bekapcsolása               }
  end;
 for i:= 0 to 255 do SetRGB( i, 0, 0, 0); { Minden szín fekete      }
 randomize;
 for i:= 0 to 64000 do mem[$a000:i]:= random( 256);
                           { Képernyő véletlenszerű feltöltése       }
 ShowPal( 100, 1, 255); { Paletta lassú megjelenítése               }
 ReadKey;
 asm
  mov ax,3
  int 10h                  { Szöveges mód                            }
  end;
 end.
```

8.20. ShowScr eljárás: megjelenítés

Szintaxis: ShowScr;

Megjeleníti a munkaterületet, amit előzőleg a *MakeScr* (8.12. rész) eljárással állítottunk össze, vagyis a munkaterület bájtjait a képernyőre másolja. A unitban ez az egyetlen képernyőre író eljárás. A *MakeScr; RetRace; ShowScr;* eljárások lényegében ugyanazt hajtják végre, mint a 2.5. fejezetben és a **SHOWBOB5.PAS** példaprogramban megtalálható *ShowBOB* eljárás. Arról,

hogy miért előnyösebb így három részletben, a 8.12. részben olvashatunk. A szép, egyenletes megjelenítés érdekében minden *ShowScr* előtt közvetlenül hívjuk meg a *RetRace* eljárást!

8.21. _KeyPressed függvény: billentyűzet figyelése

Szintaxis: _KeyPressed: Boolean;

Értéke akkor igaz, ha van legalább egy lenyomott billentyű. A *Crt* unit *KeyPressed* függvényét helyettesíti a módosított billentyűzet-megszakítás működése alatt. Csak az *InitKey* eljárás meghívása után és a *DoneKey* előtt alkalmazható. Ha ekkor a *Crt*-féle *KeyPressed* függvénnyel vizsgálnánk a billentyűzetet, gépünk lefagyna.

8.22. _ReadKey függvény: SCAN-kód beolvasása

Szintaxis: _ReadKey: byte;

Várakozik addig, amíg le nem nyomunk egy billentyűt. Ha ezt megtettük, a lenyomott billentyű SCAN-kódjával tér vissza. A *Crt* unit *ReadKey* függvényét helyettesíti a módosított megszakítás mellett. A programot **{$X+}** direktívával fordítva használható egyszerű várakozásra, ha eljárásként hívjuk meg a függvényt, a program addig vár, amíg le nem nyomunk egy billentyűt, és futása csak akkor folytatódik, ha újra felengedtük.

8.23. A BOB objektumtípus

A BOB-okhoz kapcsolódó változókat és szubrutinokat a *Game* uniton belül egy külön egységbe foglaltuk, a BOB típusba, amely egy objektum. Ez sok szempontból előnyös. Minden BOB-nak külön nevet adhatunk, nem szükséges őket egy tömbben tárolni, szabadon kezelhetők, és az objektumok örökítő lehetősége révén tetszőlegesen bővíthetők. Ezenkívül áttekinthetőbbé teszi a programot. Mielőtt elkezdenénk a BOB típus mezőit és metódusait tárgyalni, nézzünk egy rövid példaprogramot, amely betölti és megjeleníti az egyik lemezmellékleten található fájlban tárolt BOB-ot. Már ebből is láthatjuk, milyen egyszerű a unit kezelése.

```
{loadbob.pas}

uses Crt, Game;

var
 Nap: BOB;                  { A BOB azonosítója 'nap' lesz              }

begin
 InitGame;                  { GAME egység inicializálása                }
 with Nap do begin          { A BOB-bal kapcsolatos műveletek           }
  Load('napocska.bob');     { BOB inicializálása, Shape betöltése       }
  X:= 10;                   { Koordináták beállítása                    }
  Y:= 10;
  end;
 MakeScr;                   { Megjelenítés előkésztése                  }
 ShowScr;                   { Megjelenítés                              }
 readkey;                   { Várakozás egy billentyűre                 }
 DoneGame;                  { Egység lezárása                           }
 end.
```

A BOB objektumban találhatók meg többek között az ütközéseket vizsgáló függvények, a Shape betöltése eljárás, a méreteket, koordinátákat jelző változók stb. Ezeket foglaljuk össze ebben a részben, először a BOB metódusait, majd mezőit, tehát fordítva, mint ahogy egy objektumot deklarálunk. A BOB típus nem tartalmaz privát mezőket vagy metódusokat, így az összes eleme tetszőlegesen hozzáférhető, bármikor olvasható vagy megváltoztatható.

8.23.1. Collision függvény: ütközés BOB-bal

Szintaxis: Collision(var B: BOB): boolean;

Teljes egészében megegyezik az 5.2. fejezetben található *CollBob* függvénnyel. Értéke akkor igaz, ha a BOB ütközik a függvény paraméterében megadott BOB-bal. Ez az ütközés valóságos ütközés, tehát csak akkor kapunk *true* értéket eredményül, ha a vizsgált két BOB-nak legalább egy-egy pixele érintkezik. Ha csak területüknek van közös része, de nem nulla értékű pixeleik nem fedik egymást, a függvényérték *false* lesz. Erről részletesebben az 5.1-2. részekben olvashatunk.

Nézzünk egy rövid példát két BOB ütközésének vizsgálatára! Itt láthatjuk, hogyan kell BOB-ot létrehozni, és hogyan kell kezelni a BOB típus metódusait. Ha a függvényben felcseréljük az A és B betűt, ugyanazt az eredményt kapjuk.

```
uses Game;

var
 A, B: BOB;                    { A két BOB azonosítója: A, ill. B  }

begin
 if A.Collision( B) then       { Ha A ütközik B-vel, akkor ...     }
  ...
 end.
```

Ha X egy aktív BOB azonosítója, akkor `X.Collision( X)` mindig igaz (önmagával mindig érintkezik). Ha a vizsgált BOB-ok valamelyike nem látható, vagyis az A aktivitásjelző mezőjének értéke *false*, akkor a függvény visszatérési értéke is mindenképp *false*.

8.23.2. CollBack függvény: ütközés háttérrel

Szintaxis: CollBack: boolean;

Ha a BOB a háttéren van, és legalább egy pontja olyan háttérpontot fed, amelynek színe különbözik a *BackColor* változóban megadott színtől, a függvény visszatérési értéke igaz, *true*, vagyis a BOB érintkezik a háttérrel. Ez esetben a *BackColor* globális, 0 kezdőértékkel rendelkező bájt típusú változóban adjuk meg a háttér alapszínét. Ha a BOB nem nulla pontjai csak ilyen pixeleket érintenek, nincs ütközés. A függvény nagyon hasonlít a *CollColors* függvényhez (8.23.4.), vele szemben az a hátránya, hogy a háttér „szabad" része csak egyszínű lehet (ez az a rész, ahol a BOB „szabadon" mozoghat, ütközés nélkül).

8.23.3. CollBox függvény: ütközés térképegységgel

Szintaxis: CollBox: boolean;

A *Mask* globális tömb 32 bájt hosszú, a *LoadMap* eljárással töltődik fel, de értékét bármikor megváltoztathatjuk. Mindegyik bitjéhez egy BOX tartozik, egymásnak sorrendben megfeleltetve. Ha a BOB egy olyan BOX-szal érintkezik, amihez 1-es értékű bit tartozik, a függvény *true* értékkel tér vissza, egyébként az eredmény hamis, *false* lesz. Azt, hogy melyik BOX-hoz milyen bit tartozik, a térkép szerkesztésénél lehet megadni (*Edit\Mask* menüpont, 7.3.2. rész vége).

A függvény alkalmazásával a játékteret megnövelhetjük. Ha csak a *CollBack* és a *CollColors* függvényekkel vizsgáljuk a BOB-háttér ütközést, a játéktér a képernyőn is látható legfeljebb 320×200 pixel nagyságú terület. Ezzel szemben a

CollBox-szal a játék az egész térkép területére kiterjedhet. Annyit kell csupán tenni, hogy az olyan BOX-okhoz, aminek nem szabad „nekimenni" (pl. fa, fal, ház), 1-et rendelünk a maszkban, a „szabad" BOX-okhoz (pl. fű, út) pedig 0-t.

8.23.4. CollColors függvény: ütközés háttérrel

Szintaxis: *CollColors*: boolean;

Akkor igaz, ha a háttér egy meghatározott színtartományon kívül eső színű pontjaival ütközik a BOB. Ezt a színtartományt a *CFirst* és *CLast* globális változók határozzák meg, kezdőértéke mindkettőnek nulla, típusuk pedig bájt. Ha tehát a BOB minden nem átlátszó pontja csak olyan háttérpontokkal érintkezik, melyeknek színe beleesik ebbe a tartományba, a *CollColors* függvény értéke *false*, ekkor nincs ütközés.

Ez a *CollBack* függvény továbbfejlesztett változata. Ott csak egyszínű lehetett az a rész, ahol a BOB szabadon mozoghat, itt viszont bármennyi lehet e terület színeinek száma. Csak az a fontos, hogy ezeknek a színeknek egymás után kell következniük, és erre már a grafika megszerkesztésénél is ügyelni kell. Minderről részletesebben az 5.3. fejezetben olvashatunk.

8.23.5. Copy eljárás: BOB másolása

Szintaxis: Copy(var B: BOB);

Az eljárás a paraméterében megadott BOB változó mezőértékeit változtatás nélkül átmásolja a saját mezőibe. Ezelőtt inicializálja a BOB-ot, vagyis meghívja az *Init* metódust.

A *Copy* eljárást akkor használjuk, ha sok azonos mintájú BOB-ot szeretnénk létrehozni. Helyfoglalás szempontjából nem lenne ugyanis előnyös, hogy mindegyik Shape-hez külön helyet foglalunk le, hiszen ugyanolyanok. Ekkor csak egynek foglalunk helyet (a *Load*, betöltő metódussal), és a többinek adatait ehhez igazítjuk a *Copy* segítségével. Ezek után már teljesen külön kezelhető valahány BOB. A következő sorok bemutatják használatát:

```
uses Game;

var
 B: array [00..99] of BOB; { Itt vannak a BOB-ok, most tömbben    }
 i: byte;                  { Változó a FOR ciklushoz              }
```

```
begin
 B[00].Load('valami.bob'); { Az első BOB-hoz meghatározzuk a      }
                           { Shape-et,                            }
 for i:= 01 to 99 do       { a többit pedig erről                 }
  B[i].Copy( B[00]);       { lemásoljuk                           }
 ...
 end.
```

Ezek után a B tömb 100 db egyforma BOB-ot tartalmaz, és nem kellett mind a 100-szor memóriát lefoglalni, csupán egyszer, a `B[00].Load...` metódussal. Ezenkívül az összes BOB-ot inicializáltuk, mert a *Copy* meghívja ezt a metódust is, nekünk nem kell ezzel törődni. (Hogy az inicializálás, az *Init* meghívása miért fontos, lásd lejjebb, az eljárás leírásánál, a 8.23.6. részben.)

8.23.6. Init eljárás: inicializálás

Szintaxis: Init;

Ha BOB típusú változókat deklarálunk, azok a memóriában szanaszét helyezkednek el. A megjelenítendő képet összeállító *MakeScr* eljárásnak (8.12.) nehéz dolga lenne. Össze kellene szednie az összes BOB-ot, ami lehetetlen feladat, ha nincsenek rendszerezve. Ezért a *Game* unit implementációs részében található egy olyan tömb, ami rendszerezi a BOB-okat, miután meghívjuk az Init metódust, a tömb egy eleme a BOB-ra mutat, így a *MakeScr*-ben könnyen kikereshető, melyik BOB hol található.

Ebből az is következik, hogy nem hozhatunk létre akárhány BOB-ot, mivel a tömbnek kell hogy legyen egy felső határa. Ez a határ 512. Ennél több BOB-ot nem deklarálhatunk, csak akkor, ha a unit implementációs részében található *MaxBOB* konstans értékét növeljük.

A *Load* és *Copy* eljárások automatikusan meghívják az *Init* metódust, tehát ha általános módszerrel adjuk meg a BOB-adatokat (betöltjük vagy lemásoljuk), akkor az inicializálással nem kell törődnünk. Még annyit jegyezzünk meg, hogy nem érdemes kétszer meghívni ezt az eljárást, mert akkor a BOB kétszer szerepel a „listán", és grafikus adatai kétszer lesznek kimásolva egy-egy megjelenítés során, ami lassítja a program futását. Egyébként más következménye nem lesz.

8.23.7. Load eljárás: Shape betöltése

Szintaxis: Load(FileName: string);

Inicializálja a BOB-ot, és betölti a paraméterben megadott fájlt. Az eljárás meghívása után majdnem minden mező értéket kap (a koordináták nem). A memóriába kerül a Shape is, aminek kezdőcímét a DT mező tárolja. Ha az eljárást meghívjuk és a koordinátákat beállítjuk, a *MakeScr; ShowScr;* eljáráspárral vagy a *Put* metódussal (8.23.9.) a BOB már meg is jeleníthető.

Ezzel az eljárással a BOB-Editorral (6. fejezet) készített BOB-okat tölthetjük be. Az editorral előállított fájlok felépítése a következő:

Cím	Hossz	Tartalom
00	1	A BOB verziószáma, értéke 1.
01	2	Shape szélessége (a valódi szélesség ennél 1-gyel nagyobb).
03	2	Shape magassága (a valódi magasság ennél 1-gyel nagyobb).
05	2	Fázisok száma. Ha ez nulla, akkor a BOB egyfázisú.
07		Grafikus adatok.

8.23.8. OnScreen függvény: láthatóság vizsgálata

Szintaxis: OnScreen: boolean;

Ha a BOB aktív, és legalább egy pontja rajta van a háttéren – így megjelenítés után a képernyőn is – akkor a függvény visszatérési értéke *true*, egyébként *false*. Mindez a BOB *A* (aktivitásjelző) mezője és koordinátái alapján dől el.

8.23.9. Put eljárás: képernyőre rajzolás

Szintaxis: Put(XX, YY, PP: word; ShowBack: Boolean);

Az eljárás egyből a képernyőre rajzolja a BOB-ot, minden közbülső lépés nélkül. Első két paraméterében a koordinátáit kell megadni a képernyő koordinátarendszeréhez igazítva (bal felső sarok – 0;0). **PP** adja a megjelenítendő fázis számát (0, ha az 1. fázis), az utolsó paraméterben pedig azt adhatjuk meg, látszódjon-e az, ami a BOB mögött van, vagyis a BOB 0 színű pontjai átlátszóak legyenek-e.

Akkor szükséges például ez az eljárás, ha a *Setarea* eljárással csökkentettük a játékkép méretét, és az így létrejött szabad területen a főhős életeinek számát

magával a főhős grafikájával szeretnénk szemléltetni. Természetesen ezenkívül még sok helyen felhasználható, például a BOB-Editorral megrajzolt ábrákat dekorációként alkalmazhatjuk.

8.23.10. ShowUpon eljárás: prioritásváltoztatás

Szintaxis: ShowUpon(var B: BOB);

Meghívása után a BOB biztos, hogy a paraméterében megadott másik BOB „felett" lesz. Ez azt jelenti, hogy ha érintkeznek, akkor a B lesz takarásban. Mindig az a BOB van legfelül, amelyet utoljára inicializáltunk az *Init*, *Load* vagy *Copy* metódusok valamelyikével. Ezen a sorrenden lehet változtatni a *Showupon* eljárással.

8.23.11. A BOB típus mezői

Itt találhatók meg a BOB nem grafikus adatai (koordináták, fázisszám stb.). A grafikus adatok, a Shape bájtjai a heap-ben foglalnak helyet, mert dinamikus változóként tároljuk őket. Kezdőcímét a DT mező adja. Az alábbiakban olyan sorrendben ismertetjük a BOB típus mezőit, ahogy deklarálva lettek, így könnyen kiszámítható az elemek eltolási címe.

Mezőkód	Típus	Eltolás	Tartalom
A	wordbool	0	Aktivitásjelző. Ha *false*, akkor a BOB nem látható, akárhol van, és ilyenkor minden ütközésvizsgálat eredménye is *false*.
X	integer	2	Abszcissza. Ha 0, akkor a BOB a háttér bal szélén helyezkedik el. (Lehet 0-nál kisebb is.)
Y	integer	4	Ordináta. Ha 0, akkor a BOB a háttér tetejével egy vonalban van.
LX	word	6	Értéke a BOB szélességénél eggyel kevesebb (W-1).
LY	word	8	Értéke a BOB magasságánál eggyel kevesebb (H-1).
P	word	10	Fázisszámláló, a következő megjelenítéskor a kirajzolandó fázis. Ha értéke 0, akkor az 1. fázis kerül kirajzolásra.
DT	pointer	12	Shape kezdőcíme. A *Load* metódussal kap értéket.

Mezőkód	Típus	Eltolás	Tartalom
PL	word	16	Egy fázis hossza, bájtban. (A *MakeScr* eljárás használja.)
W	word	18	Szélesség pixelben.
H	word	20	Magasság.
LN	word	22	Shape hossza bájtban.
PN	word	24	Értéke az összes fázisnál eggyel kevesebb.
NR	word	26	BOB sorszáma.

Ha a fázisokat folyamatosan szeretnénk görgetni, úgy, hogy az utolsó után az első jöjjön, építsük be a megjelenítő ciklusba a következő sort (B egy BOB típusú változó):

```
inc( B.P); if B.P>B.PN then B.P:=0;
```

Mivel az első mező *wordbool* és nem *boolean* típusú, az **{$A+}** direktívával a mezők szóhatárra kerülnek. Ez valamennyire növelheti a program sebességét.

8.24. A *Game* unit konstansai és változói

A konstansok, változók működését már leírtuk, elszórva, mindig egy-egy szubrutin tárgyalásánál. Most csak összefoglaljuk és rendszerezzük őket. Három csoportba sorolhatók: konstansok, tipizált konstansok, változók.

Összesen négy nem tipizált konstans van, melyek a *Scroll* eljárás (8.16. rész) paraméterezését segítik. Ennek az eljárásnak az első paraméterében adjuk meg ezeket az értékeket, és a térkép az adott irányban fog elmozdulni.

```
const

 Up    = 8;   { Fel    }
 Down  = 2;   { Le     }
 Left  = 4;   { Balra  }
 Right = 6;   { Jobbra }
```

Tipizált konstansból, azaz kezdőértékkel rendelkező változóból csak három van. Ezek a BOB-háttér ütközésvizsgálathoz kellenek (8.23.2, 8.23.4.), meghatározzák azokat a színeket, ahol a BOB-ok szabadon mozoghatnak, ütközés nélkül.

```
const

 BackColor:  byte     = 0;     { Háttér alapszíne a CollBack-hez     }
 CFirst:     byte     = 0;     { Színtartomány első és utolsó színe  }
 CLast:      byte     = 0;     { a CollColors metódushoz             }
```

Változókból már jóval több van, mint konstansokból. Ezekhez részletesebb magyarázatot is adunk, és ábécérendben írjuk le őket.

```
var BackGround: pointer;
```

A háttér kezdőcíme. Az *InitGame* meghívása után az eljárás által lefoglalt 64002 bájt nagyságú területre mutat. Ez az érték akkor sem változik, ha a *SetArea* eljárással csökkentjük a háttér méretét.

```
var BoxPtr: pointer;
```

BOX-adatok kezdőcíme. A *LoadMap* eljárás meghívása után a BOX-ok grafikus adataira mutat. Ennek a dinamikus változónak a hossza 16384 bájt. 256 db 64 bájtos egységre bontható, melyek egy-egy 8×8-as doboz pontjait tárolják.

```
var BPL: word;
```

Egy térképsorban található dobozok száma, azaz a térkép szélessége. A *NewPos*, a *Scroll* és néhány belső eljárás használja.

```
var Colors: array [0..255] of RGBType;
```

Az RGBType típus deklarációja a következő:

```
type
 RGBType = record
  Red, Green, Blue: byte;
  end;
```

A *Colors* tömb tárolja a *ShowPal* eljárás (8.19) számára a megjelenítendő paletta színeinek összetevőit. Értéket a *HidePal* eljárás ad elemeinek.

```
var First: word;
```

A képernyőn megjelenő játékkép első sora, a képernyő bal felső sarkához viszonyítva. Csak olvasásra ajánljuk, változtatni a *SetArea* eljárással lehet (8.17.).

```
var key: array [0..255] of boolean;
```

A billentyűk információit tárolja, ha korábban meghívtuk az *InitKey* eljárást, de annak a *DoneKey* párját még nem futtattuk le. Mindegyik billentyűhöz hozzá van rendelve egy szám, ha lenyomtuk, a billentyűhöz tartózó sorszámú *Key-*

elem igazra vált. Tehát ez a tömb azt mutatja, hogy az adott sorszámú billentyűt lenyomtuk-e vagy sem.

A hozzárendelési táblázat a 3.1. fejezet végén és a lemezmelléklet **SCANCODE.TXT** fájljában található. Ezt érdemes kinyomtatni.

```
var Last: word;
```

A képernyőn a játéktér utolsó sora. A *First* változóhoz hasonlóan szintén nem érdemes módosítani, csak a *Setarea* eljárással (8.17.).

```
var MapLength: word;
```

A térkép hossza bájtban, dobozainak száma. A LoadMap eljárás meghívásával kap értéket. A térkép magasságát nem tároljuk külön változóban, ennek kiszámítására a következő egyszerű képlet szolgál:

térkép magassága = MapLenght/BPL

```
var MapPtr: pointer;
```

Térkép kezdőcíme. A *LoadMap* eljárással a térképnek lefoglalt terület első bájtjára mutat.

```
var Mask: array [0..31]  of byte;
```

BOB-BOX ütközési váz. A BOB típus *CollBOX* metódusa (8.23.3. rész) használja. Összesen 256 bitet tartalmaz, mindegyik bithez egy BOX tartozik. Ha a BOB 1-es jelzésű BOX-szal érintkezik, a *CollBOX* függvény visszatérési értéke igaz lesz.

Ütközési vázat a térképhez a MAP-Editorral szerkeszthetünk, az Edit menü *Mask* funkciójának kiválasztásával. A *LoadMap* eljárással a térképhez tartozó maszk is betöltődik.

```
var PosX, PosY: word;
```

A térkép koordinátái, pontosabban a háttér pixelekben mért vízszintes és függőleges eltolása a térkép bal felső sarkához viszonyítva. A *Scroll* és a *NewPos* eljárás változtatja meg értéküket. Ha mi írjuk át őket, megoldható például a végtelenített térkép: figyeljük, mikor ér véget a térkép mondjuk balra gördítésnél, s ha ez bekövetkezett, a *PosX* változót 0-ra állítjuk. Ezután már a térkép eleje görög be, de a vége még nem tűnik el, hanem szép lassan kicsúszik balra.

```
var WorkArea: pointer;
```

Munkaterület kezdőcíme. A munkaterületnek az *InitGame* eljárással foglalunk helyet. Ha ezt nem tettük meg, és úgy indítjuk el a *MakeScr* eljárást, gépünk nagy valószínűséggel lefagy, mivel ez az eljárás a munkaterületre ír. Ezért is nagyon fontos, hogy minden játékban lehetőleg minél előbb hívjuk meg az *InitGame* eljárást.

9. Példajáték

Ebben a fejezetben a *Game* egység használatára láthatunk példát. Annak elemeit az előző részben már megismertük, most átültetjük mindezt a gyakorlatba. A játékot a lemezmelléklet **EXAMPLE** könyvtárában találhatjuk meg, lefordított és eredeti állapotban egyaránt (**EXAMPLE.EXE**, **EXAMPLE.PAS**). Az **EXE** kiterjesztésű fájlt rögtön le is futtathatjuk.

A játék meglehetősen primitív, egy ház falán kell felmásznia a főhősnek, közben ki kell kerülnie a nyitott ablakokat, erkélyeket és a potyogó tárgyakat (virágcserép, akvárium stb.). Ha egy ilyen zuhanó tárgy eltalálja a főhőst, az holtan esik alá, és veszít egyet életeiből. A játék lényege: felmászni a ház tetejére. Talán elég egyszerű ahhoz, hogy ne legyen megterhelő működését megismerni.

```
{example.pas}

uses Crt, Game, MCGA, _System;
```

A *Game* uniton kívül az *MCGA* és a *_System* unit eljárásait is használja a program. Ezek a 6.1.1-2. részekben lelhetők meg. (A *_System* unitból csak az *_Str* függvényre van szükség, amit a magasság és az életek számának kiírásához használunk.)

```
const

 MaxFalls        =   3;
 PalSpd          =  10;
 Skill           =  50;
```

A *MaxFalls* konstans értéke a lefelé eső tárgyak számánál eggyel kevesebb. Könnyen bővíthetjük a program grafikáját, ha a BOB-Editorral tervezünk egy zuhanó objektumot, amit FALL4.BOB néven elmentünk, és növeljük a *MaxFalls* konstans értékét. Ezután a lefelé eső dolgok közt ott lesz az általunk tervezett BOB is.

A *PalSpd* konstans az elsötétítések és kivilágosítások sebességét adja, a *HidePal* és a *ShowPal* eljárások első paraméterének mindig ezt az értéket adjuk meg. Ha csökkentjük, nő a palettaváltoztatás sebessége.

Végül a *Skill* konstans a zuhanó tárgyak gyakoriságát szabályozza. Ennyi két szomszédos tárgy ordinátájának átlagos különbsége.

```
PDly:    byte  =   0;
Lives:   byte  =   9;
Height:  word  =   0;
```

A *PDly* 0 kezdőértékű változó a fázisváltás késleltetéséhez szükséges változó. A főhős négyfázisú, az első kettő a függőleges, a második kettő a vízszintes irányú mozgás közben váltakozik. Fáziskésleltetés nélkül vagy nagyon sok fázist kellene létrehozni, vagy pedig túl gyors lenne a mozgás.

A *Lives* az életek számát tartalmazza, a *Height* pedig a magasságot képpontban.

```
var

 Hero:       BOB;
 Fall:       array [0..MaxFalls] of BOB;
 POld:       word;
 Killed:     boolean;
 Won:        boolean;
 GameOver:   boolean;
 i:          word;
 f:          file;
```

A *Hero* a főhős azonosítója, a *Fall* tömb pedig az „ellenségek", a zuhanó tárgyak nemgrafikus adatait tartalmazza. A *POld* változóra a fázisváltoztatásnál lesz szükség – az előző fázis sorszámát tárolja. A *Killed*, *Won*, *GameOver* logikai változók a *repeat...until* ciklusok végét jelzik, a *Killed* akkor lesz igaz, ha a főhős meghalt, a *Won*, ha nyert, a *GameOver* pedig akkor kap *true* értéket, ha elfogytak az életek, vagy ESC-t nyomtunk. Az utolsó két változó általános célú, *i*-re a *for...* ciklusoknál, *f*-re a palettabetöltésnél lesz szükség.

A deklarációs rész végén található két eljárás megjeleníti az életek számát, illetve az aktuális BOX-ban mért magasságot. Alapelvük a következő: az *MCGA* unit (6.1.1) *CBar* eljárásával letörlik a régi számokat, egy fekete téglalappal borítják be, majd kiírják az újat, ugyanennek az egységnek az *OutTransparentText* eljárásával. Kiírásnál szép térbeli hatást érhetünk el, ha kétszer jelenítjük meg a szöveget, csak másodszor egy pixellel balra és fel elcsúsztatva és más, lehetőleg világosabb színnel.

A számok szöveggé konvertálásához a *_System* unit (6.1.2.) *_Str* függvényét használjuk.

```
procedure ShowLives;
 begin
  CBar( 19, 220, 190, 300, 199);
  PointColor:=  7;
  OutTransparentText( 221, 192, _str( Lives));
  PointColor:= 15;
  OutTransparentText( 220, 191, _str( Lives));
  end;
procedure ShowHeight;
 begin
  CBar( 19, 80, 190, 120, 199);
  PointColor:= 7;
  OutTransparentText(  81, 192, _str( Height shr 3));
  PointColor:= 15;
  OutTransparentText(  80, 191, _str( Height shr 3));
  end;
```

Most következik a főprogram. Elején el kell végezni a szükséges inicializálásokat, betöltéseket, majd következhetnek az egybeágyazott ciklusok, melyek a játék mozgatását végzik. Végül egy rövid lezáró rész található, melyben visszaállítjuk az eredeti billentyűmegszakítást és a szöveges képernyő üzemmódot.

```
begin
 InitGame;
 LoadLBM( 'example.lbm', ptr($a000,0));
 readkey;
 HidePal( PalSpd, 1, 255, 0, 0, 0);
```

Rögtön első eljárásként az *InitGame*-et hívtuk meg. Nagyon fontos, hogy a *Game* egység részeit csak az után használjuk, miután inicializáltuk a unitot ezzel az eljárással. Ennek itt eleget tettünk. Az *MCGA* üzemmódot az *InitGame* bekapcsolta, így a *LoadLBM* eljárással a kezdőkép megjeleníthető. Ez addig látszik, amíg meg nem nyomunk egy billentyűt, utána lassan eltűnik, a *HidePal* eljárásnak köszönhetően.

A következő két sor betölti a térképet, ami a házat tartalmazza, és beállítja, hogy a legalja látszódjon. Ezenkívül a NewPos eljárásnak köszönhetően a háttéren is megjelenik a kép, ha utána közvetlenül kiadnánk a MakeScr; ShowScr; utasításpárt, a térkép alja látható lenne.

```
 LoadMap( 'example.map');
 NewPos( 0, (MapLength div BPL) shl 3 - 190);

 with Hero do begin
  Load( 'hero.bob');
  X:= 160 - w shr 1;
  Y:= 180 - h;
  end;
```

A főhős inicializálását elvégzi a *Load* metódus. Ez egyben betölti a Shape-et is, viszont a koordinátákat nem változtatja. A betöltést követő két sorban történik az elhelyezés: vízszintesen pont középen van (W SHR 1=W DIV 2), függőlegesen pedig úgy foglal helyet, hogy talpa pont a 180. sort érinti.

Az alábbi ciklus a hulló tárgyakat inicializálja. A *Fall* tömb elemeibe sorszámuk alapján betölti a lemezen lévő **FALLx.BOB** fájlokat. Ezután véletlenszerűen elhelyezi ezeket a BOB-okat. A háttér függőleges széleinek vonalán belül helyezkednek el, viszont függőlegesen a háttér fölött, így eleinte nem láthatók. A függőleges elhelyezésnél kap szerepet a *Skill* konstans. Minél nagyobb ez az érték, annál ritkábban jönnek a BOB-ok. Tehát most, a véletlenszám-generátor inicializálása után következik az „ellenfelek" inicializálása.

```
randomize;
for i:= 0 to MaxFalls do with Fall[i] do begin
 Load( 'fall'+_str(i)+'.bob');
 X:= random( 320-w);
 Y:= -h-random( Skill*MaxFalls);
 end;

SetArea( 0, 189);
MakeScr;
ShowScr;
```

A *SetArea* beállítja a játékteret úgy, hogy a képernyőn az alsó 10 sor szabadon maradjon, ezt a sávot a *ShowScr* eljárás nem módosítja. Ide kerül majd az állapotsor, mely jelzi a magasságot és az életek számát. A *MakeScr–ShowScr* eljáráspár megjeleníti a kezdő játékképet. Viszont ez még nem látható, hiszen a kezdőkép eltüntetése után a paletta minden színe fekete lett, és ez azóta nem változott.

```
CBar( 19, 0, 190, 319, 199);
TextBackColor:= 19;
PointColor:=  3;
 OutText(   1, 192, 'Magasság:');
 OutText( 161, 192, 'Életek:' );
PointColor:= 14;
 OutTransparentText(   0, 191, 'Magasság:');
 OutTransparentText( 160, 191, 'Életek:' );
ShowLives;
ShowHeight;
```

A fenti sorok az állapotsort jelenítik meg. Először befedjük ezt a részt egy fekete téglalappal, a *CBar* eljárással. Bár a játék által használt (alább betöltendő) palettában a 0. szín is fekete, mégsem ezzel töltjük fel a képernyő alsó 10 sorát. Ugyanis palettagörgetésnél a 0. szín változatlan marad, és nem lenne szép, ha

fekete részek maradnának, miután a kép elpiroslik (halál után) vagy elfehéredik (győzelem után). Tehát nagyon fontos, hogy a *HidePal* és a *ShowPal* eljárásokat használó programok grafikájához sehol ne használjuk a 0. színt.

Ezután megjelennek a feliratok és a számok, kétszer, a térbeli hatás érdekében. Természetesen még mindig nem látható semmi, mert a paletta fekete. Azt később állítjuk be, a most következő négy sor csupán betölti a *Colors* tömbbe (*Game* egység). Ezután egy *ShowPal* eljárással fokozatosan megjeleníthető. Persze sokkal egyszerűbb lenne a palettát a *LoadPal* eljárással elővarázsolni, viszont az nem lenne ilyen szép, mert hirtelen jelenne meg.

```
assign( f, 'example.pal');
reset( f, 1);
blockread( f, Colors, 768);
close( f);

CFirst:= 64;
CLast := 79;
```

A *CFirst* és a *CLast* változók beállításával meghatározzuk azt a színtartományt, aminek színeivel a főhős ütközhet. Ebbe az intervallumba esik a házfal, a cserép és a zárt ablakok színe. Viszont ezen kívül kell lennie az égszínnek, az erkélyek és a nyitott ablakok színének, mivel ezekkel nem érintkezhet a főhős, ki kell kerülnie. Minden játék grafikája elkészítésének már a kezdetén vegyük figyelembe, hogy meg kell határozni egy ilyen színintervallumot, és aszerint kell a pontokat beszínezni.

```
InitKey;
```

A főciklus megkezdése előtt inicializáljuk a saját billentyűmegszakítást. Ezt lehetőleg minél később tegyük, így csökken az esély arra, hogy a gép „befagy". Ha az *InitKey* eljárást már a program elején meghívtuk volna, egy-egy lemezműveletnél könnyen leállhat a rendszer. Ha például az adott fájl nem érhető el, a program futása megszakad, és ha nem állítjuk vissza a megszakításvektort, akkor újra kell indítani a gépet. Tehát: vagy a fájlműveletek elvégzése után hívjuk meg az *InitKey* eljárást, vagy pedig tüzetesen vizsgáljunk meg minden fájlműveletet.

Elérkeztünk a főciklus kezdetéhez. Ez tulajdonképpen két egybeágyazott ciklus. A belső kap nagyobb szerepet: ez végzi a játék mozgatását. Ebből csak halál, győzelem vagy az ESC billentyű megnyomása esetén lépünk ki. (Ha a *Killed*, *Won* vagy *GameOver* logikai változók valamelyike igazra vált.)

```
repeat

 Killed:= false;
 ShowPal( PalSpd, 1, 255);
```

A belső ciklus megkezdése előtt a *Killed* változó hamisra vált, és a paletta kivilágosodik. Erre szükség lesz egy-egy élet elvesztése után is, amikor az elpirosodott képből kell az eredeti palettát visszaállítani. Ezért nem hívtuk meg a *ShowPal* eljárást az inicializáló részben a főciklus előtt.

```
 repeat

  if Key[$C8] then with Hero do begin

   Scroll( Down, 1);
   if CollColors then Scroll( Up, 1) else begin
```

A belső ciklusban legelőször a felfelé mutató nyílbillentyű megnyomását vizsgáljuk. Ha ez bekövetkezik, a térkép lefelé mozdul egy pixellel. Ha most olyan helyzetbe került, hogy a BOB érintkezik a *CFirst–Clast* változókban megadott színtartományon kívül eső színnel, akkor visszagördül a térkép, egy pixelt felfele. (Magyarul: nyitott ablakon, erkélyen nem lehet felmászni.) Ekkor tehát a térkép megmozdult: lefele és felfele gördült egy sort, mindebből azonban semmit sem veszünk észre, mert csak a háttér változott, a képernyő tartalma nem.

Ha sikerült a főhősnek felfelé mozdulni (nem ütközött pl. nyitott ablakba), be kell állítani még pár dolgot. Először növeljük a magasságot jelző változót, majd kiírjuk a képernyőre. (A *ShowHeight* eljárás a játék deklarációs részében található.) Ezután ellenőrizzük, nem ért-e föl a hős a ház tetejére, ennek megfelelően állítjuk be a *Won* győzelmet jelző változót, amit a ciklus alján, az *until* után ellenőrzünk.

```
inc( Height);
ShowHeight;
Won:= PosY=4;
```

A következő részben a fázisváltás következik. Legelőször tároljuk az aktuális fázist a *POld* változóba, hogy vissza lehessen állítani, ha a BOB a fázisváltoztatás után „tiltott" színekkel érintkezik (nyitott ablakkal, éggel stb.). A felfelé mozgást az első két fázis tartalmazza, így ha a BOB a 3. vagy a 4. szakaszban van, az alábbi rész második sora 0-ra állítja a fázisszámlálót (az 1. fázisra).

```
POld:= P;
if P>1 then P:= 0;
inc( PDly);
```

```
    if PDly=10 then begin
     PDly:= 0;
     inc( P);
     if P>1 then P:= 0;
     sound( 200);
     end;
    if CollColors then P:= POld;
```

A fázisváltás lényege a következő: van egy számláló, mely folyamatosan növekszik, és ha elért egy bizonyos határt (itt: 10), lenullázódik, és eközben megváltozik az aktuális fázis. Így ugyan darabosabb a mozgás, de legalább nem kell minden pixel elmozduláshoz külön periódust rajzolni. Ezt a fáziskésleltető módszer saját játékokban is felhasználható. Ezenkívül minden fázisváltást hangadás kísér, majd az utolsó sor visszaállítja az eredeti fázist, ha a *CollColors* metódus visszatérési értéke *true*.

A BOB felfelé mozgásához tartozik még az „ellenségek", a hulló tárgyak lefelé mozgatása:

```
    for i:= 0 to MaxFalls do inc( Fall[i].y);

    end;

  end
```

Ezután következik a balra, illetve jobbra mozgás. Alapelvük ugyanaz, mint a felfelé mozgásnak. Csak apró különbségek vannak: nem a térkép mozog, hanem a BOB, a 3. és a 4. fázis váltakozik, mélyebb hang szól egy-egy fázisváltásnál. Csupán annyit érdemes még megjegyezni, hogy a három vizsgált nyílbillentyű közül egyszerre csak egynek a lenyomását érzékeli a program (a következő két rész *else* kezdete miatt), tehát nem lehet átlós irányban haladni.

```
  else if Key[$CB] then with Hero do begin
   dec( x);
   if CollColors then inc( x) else begin
    POld:= P;
    if P<2 then P:= 2;
    inc( PDly);
    if PDly=10 then begin
     PDly:= 0;
     inc( P);
     if P>PN then P:= 2;
     sound( 100);
     end;
    if CollColors then P:= POld;
    end;
  end
```

```
else if Key[$CD] then with Hero do begin
 inc( x);
 if CollColors then dec( x) else begin
  POld:= P;
  if P<2 then P:= 2;
  inc( PDly);
  if PDly=10 then begin
   PDly:= 0;
   inc( P);
   if P>PN then P:= 2;
   sound( 100);
   end;
   if CollColors then P:= POld;
  end;
 end;
```

Az alábbi részre azért van szükség, hogy minden billentyű lenyomásakor legyen fázisváltás:

```
if not( Key[$C8] or Key[$CB] or Key[$CD])
 then PDLy:= 9;
```

ESC nyomásra vége a játéknak:

```
GameOver:= Key[1];
```

A belső ciklus végén már csak a tárgyak folyamatos eséséről és a megjelenítésről kell gondoskodni. Ha egy tárgy elhagyta alul a képet, visszaugrik föléje, új koordinátákkal. Átlagosan minden harmadik pont a főhős feje fölött esik. Ha valamelyik ütközik a főhőssel, a *Killed* változó igaz értéket kap, aminek hatására a program alul, az *until* szónál kilép a belső ciklusból.

A megjelenítés a *MakeScr; RetRace; ShowScr*; eljáráshármassal történik. Ezután hívjuk meg a *Nosound* eljárást. Ha a megjelenítés előtt halkítanánk el a hangszórót, sokkal rövidebb ideig szólna a fázisváltásoknál kiadott hang, mivel a megjelenítés meglehetősen sok időt vesz igénybe.

```
for i:= 0 to MaxFalls do with Fall[i] do begin
 inc( y);
 Killed:= Killed or Collision( Hero);
 if y>190 then begin
  y:= -h-random( Skill*MaxFalls);
  if random( 3)>0 then x:= random( 320-w)
   else x:= Hero.X;
  end;
 end;

MakeScr;
RetRace;
ShowScr;
```

```
  nosound;

  until Killed or Won or GameOver;
```

A program kilép a belső ciklusból, ha a főhős meghalt vagy nyert; vagy a játékos megnyomta az ESC billentyűt, aminek hatására a *GameOver* igazra változott.

A külső ciklus még két részből áll: az első akkor kerül végrehajtásra, ha a főhős meghalt, a második akkor fut le, ha nyert.

```
if Killed then begin
```

Ha a főhőst eltalálta egy tárgy, lezuhan. Eközben gondoskodni kell a tárgyak eséséről és a megjelenítésről is. Ezután elvöröslik a kép. A vörösnek egy véletlenszerűen kiválasztott árnyalata telíti be a képernyőt. Végül csökkentjük és megjelenítjük az életek számát, majd visszaállítjuk a BOB-ok koordinátáit. A színek összetevői a külső ciklus elején állnak majd vissza eredeti állapotukba, a *ShowPal* eljárás segítségével.

```
  inc( Hero.y);
  repeat
   for i:= 0 to MaxFalls do with Fall[i] do inc( y);
   inc( Hero.y);
   MakeScr;
   RetRace;
   ShowScr;
   until Hero.y>220;
  HidePal( PalSpd, 1, 255, random( 64), 0, 0);
  dec( Lives);
  ShowLives;
  for i:= 0 to MaxFalls do with Fall[i] do begin
   y:= -h-random( Skill*MaxFalls);
   x:= random( 320-w);
   end;
  Hero.y:= 180-Hero.h;
  GameOver:= Lives=0;
  MakeScr;
  ShowScr;

  end;
```

Ha a főhős feljutott a ház tetejére, nyert. Ekkor a képernyő lassan elfehéredik, majd elsötétül.

```
if Won then begin
 HidePal( PalSpd, 1, 255, 63, 63, 63);
 HidePal( PalSpd, 1, 255,  0,  0,  0);
 end;
until GameOver or Won;
```

A program befejeződik, ha ESC-t nyomott a játékos, ha elfogytak az életek, vagy ha nyert. A BIOS billentyűmegszakítás és a szöveges mód visszaállításán kívül már nem is kell mást tenni, viszont ezeket nem szabad elfelejteni!

```
DoneKey;
DoneGame;
end.
```

Megjegyzés: a *DoneGame* eljárás helyett írhattuk volna a következőt:

```
SetMode( 3);
```

Egy konkrét példán keresztül mutattuk be a *Game* unit működésének főbb elveit, de persze általánosságokat is le lehetett szűrni belőle. Az egység minden részét csak egy sokkal terjedelmesebb példa által lehetett volna megismertetni, aminek megértése is sokkal nehezebb lenne. Bízunk abban, hogy az Olvasó azokat a lehetőségeket is jól tudja majd hasznosítani, melyek ebben a programban nem szerepelnek.

A lemezmellékleten található még egy játék, a **TANK** könyvtárban, ami szintén főként a *Game* unit elemeit használja. Érdemes elindítani.

10. A lemezmelléklet ismertetése

A lemezmellékleten megtalálható minden program forráskódja, grafikai adatfájlok és más, a programok futtatásához szükséges adatállományok. Némelyik program lefordított, rögtön indítható állapotban is meglelhető, ilyenek pl. az editorok. A könyvben található önálló eljárások, függvények pedig *include* (**.INC**) fájlokban kaptak helyet.

A programok könnyen megtalálhatók, fejezetenként vannak csoportosítva a **FEJEZET1, FEJEZET2, ...** könyvtárakban. Ezeken kívül még más könyvtárak is vannak a lemezen, melyekbe kiemeltünk egyes programokat. A lemezmelléklet könyvtárai tehát:

BOBEDIT	BOB-szerkesztő segédprogram. Rögtön futtatható.
EXAMPLE	Példajáték, ami a *Game* unit egyszerű demonstrációja. Fordítás nélkül indítható.
FEJEZET1	Az **Alapfogalmak** fejezet programjai.
FEJEZET2	A **BOB-ok megjelenítése** fejezet programjai.
FEJEZET3	A **Billentyűzet és egér** fejezet programjai.
FEJEZET4	A **Háttér** fejezet programjai.
FEJEZET5	Az **Ütközések** fejezet programjai.
FEJEZET6	A **BOB-Editor** fejezet programjai.
FEJEZET7	A **MAP-Editor** fejezet programjai.
FEJEZET8	**A Game unit** fejezet programjai.
FEJEZET9	A **Példajáték** fejezet programjai.
MAPEDIT	MAP-szerkesztő segédprogram. Lefordított állapotban is megtalálható.
TANK	Egy másik példajáték. Szintén a *Game* unit felhasználásával készült. Rögtön futtatható.
UNIT	A lemezen szereplő legtöbb program az itt található unitokat használja (*_System, Game, MCGA, Menu, Mouse*), ezért érdemes ezeket lefordítani, és a keletkezett TPU fájlokat egy saját unit-könyvtárba másolni, hogy a Turbo Pascal számára hozzáférhetőek legyenek.

A könyvben szereplő önálló eljárások/függvények a lemezmellékleten is megtalálhatók. A következő táblázat megmutatja, hogy melyik szubrutin melyik fájlban lelhető meg.

Név	Típus	Fejezet	Fájl
Collision	függvény	5.1.	COLL1.INC
Collision	függvény	5.2.	COLL2.INC
FillBack	eljárás	4.	FILLBACK.INC
GetPixel	függvény	1.3.	GETPIXEL.INC
GetRGB	eljárás	1.6.1.	GETRGB.INC
HLine	eljárás	4.4.1.	HLINE1.INC
HLine	eljárás	4.4.2.	HLINE2.INC
LoadLBM	eljárás	4.1.	LOADLBM.INC
ScrollD	eljárás	4.6.4.	SCROLLD.INC
ScrollL	eljárás	4.6.2.	SCROLLL.INC
ScrollR	eljárás	4.6.1.	SCROLLR.INC
ScrollU	eljárás	4.6.3.	SCROLLU.INC
SetRGB	eljárás	1.6.2.	SETRGB.INC
VLine	eljárás	4.5.	VLINE.INC

Ha a lemezmellékleten található programokat futtatni kívánjuk, a következő fordító direktívákat kapcsoljuk be:

- G – 286-os kód generálása
- I – I/O ellenőrzés
- X – a kiterjesztett szintaxis alkalmazása

Irodalomjegyzék

Benkő Tiborné – Benkő László – Tóth Bertalan – Varga Balázs:
Programozzunk Turbo Pascal nyelven
ComputerBooks, Budapest, 1998.

Benkő Tiborné – Benkő László – Dr. Meszéna Zsolt – Dr. Gyenes Károly:
Programozási feladatok és algoritmusok Turbo Pascal nyelven
ComputerBooks, Budapest, 1996.

Benkő Tiborné – Benkő László – Dr. Gyenes Károly – Dr. Komócsin Zoltán:
Objektum-orientált programozás Turbo Pascal nyelven 7.0
ComputerBooks, Budapest, 1997.

Benkő Tiborné – Kiss Zoltán – Dr. Tamás Péter – Tóth Bertalan:
Programozás Borland Pascal 7.0 rendszerben
ComputerBooks, Budapest, 1994.

László József:
A VGA kártya programozása
ComputerBooks, Budapest, 1995.

Marton László:
Bevezetés a Pascal nyelvű programozásba
Novadat, Budapest, 1996.

Pethő Ádám:
Variációk assembly-re
PSS Kiadó, 1996.

Agárdi Gábor:
IBM PC Gyakorlati assembly
LSI oktatóközpont, Budapest, 1993.

Pirkó József:
Turbo Pascal 6.0 for Windows
LSI oktatóközpont, Budapest, 1992.

Tárgymutató

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Megrendelőlap

Megrendelem az alábbi kiadványokat postai utánvéttel. Tudomásul veszem, hogy a postaköltség felszámításra kerül, és a szállítási idő 2-3 hét.

Utánnyomásoknál árváltozás lehetséges.

... *pld. Füzi János:* **3 dimenziós grafika és animáció IBM PC-n** - 🖫	**1.480.-**
... *pld. Füzi János:* **Interaktív grafika** - 🖫	**1.993.-**
... *pld. Jakab Zs.:* **Adobe PHOTOSHOP 4.0** - magyar változat	**2.464.-**
... *pld. Dr.Kovács T. -Dr.Kovácsné C.J -Ozsváth M.:* **Adatkezelés az MS ACCESS 2.0 alkalmazásával**	**1.890.-**
... *pld. Dr.Kovács T. -Dr.Kovácsné C.J.-Ozsváth M.:* **Adatkezelés az MS ACCESS 7.0 alkalmazásával**	**2.688.-**
...*pld Nagy Gábor:* **Adattömörítési kézikönyv**	**1.498.-**
... *pld. Pintér M.:* **AutoCAD tankönyv** - DOS & WINDOWS; AutoCAD LT; AutoCAD R12 angol & magyar változathoz	**1.200.-**
... *pld. Pintér M.:* Szilárdtestek modellezése **AutoCAD R12-vel**	**1.200.-**
... *pld. Pintér M.:* **AutoCAD Designer**	**980.-**
... *pld. Pintér M.:* **AutoVision**	**1.961.-**
... *pld Pintér M.:* **Új AutoCAD tankönyv 1. - R14** - síkbeli rajzok készítése	**1.680.-**
... *pld Pintér M.:* **Új AutoCAD tankönyv 2. - R14** - térbeli ábrázolás	**1.680.-**
... *pld. Benkő L.-Benkő T.né-Tóth:* **Programozzunk C nyelven** - 🖫	**1.499.-**
...*pld. Benkő L. -Benkő T.-né:* **Programozási feladatok és algoritmusok TURBO C és C++ nyelven** - 🖫	**1.998.-**
...*pld Benkő T.né.- Poppe A.:* **Objektum-orientált programozás C++ nyelven**- 🖫	**2.464.-**
.. *pld. Dr.Kondorosi K.-Dr.László Z.-Dr.Szirmay-Kalos László:* **Objektum-orientált szoftverfejlesztés (C és C++)**- 🖫	**2.616.-**
... *pld Benkő T.-né- Móré G.:* **ObjectWindows** - Windows alkalmazások fejlesztése Borland C++ rendszerben 🖫	**3.472.-**
... *pld. Dr.Dedinszky F.:* **CLIPPER 5 - 5.0, 5.01 és segédprogramjai**	**1.200.-**
... *pld. Nagy Z.-Spányik B.-Weisz T.:* **CorelDRAW! 5**	**1.568.-**
... *pld Bognár Júlia:* **dBASE III PLUS**	**1.232.-**
... *pld. Gazsó Z.:* **"VISUAL" Adatbázis-kezelők objektum-orientált programozása - *Visual dBASE; Visual FoxPro; Visual Object*** - 🖫	**1.493.-**
... *pld. Kovalcsik G.:* **EXCEL 5.0 for Windows kezdőknek * haladóknak** - magyar és angol változathoz	**1.449.-**
...*pld Kovácsné C. Judit -Ozsváth M* **EXCEL 5 függvényei magyar változat**	**1.200.-**
...*pld. Kóczy Judit:* **EXCEL 7 for Windows 95 felhasználóknak**	**1.994.-**
... *pld. Krizsák László:* **EXCEL 7.0 programozása** - 🖫	**1.456.-**
... *pld Kovalcsik Géza:* **EXCEL 97** - 🖫	**2.490.-**

A 375-35-91 telefonszámon tájékoztatjuk Önt a lakó- vagy munkahelyéhez legközelebbi szaküzletről, ahol kiadványainkat megvásárolhatja.

Ha a postai utat választja, kérjük a Megrendelőlapot levélcímünkre, **COMPUTERBOOKS Kft - 1253 Bp., Pf.: 71**. visszaküldeni.

... *pld Dedinszky F.-Balogh J.* **FoxPro 2.0** változatlan utánnyomás	**1.650.-**
... *pld Gazsó Zoltán:* **Adatbáziskezelés FoxProban** - 2.5, 2.6 - Windows/DOS - 🖫	**1.475.-**
... *pld. Abonyi Zs.*: **PC hardver kézikönyv** (bővített, átdolgozott kiadás)	**1.300.-**
...*pld Ozsváth Miklós.* **QuarkXPress4** - 🖹	**2.340.-**
... *pld. László J.*: **Hangkártya programozása Pascal és Assembly nyelven** - 🖫	**1.568.-**
... *pld. Lengyel Veronika:* **Az INTERNET világa**	**1.456.-**
... *pld. Nagy Sándor*: **Internet és Intranet IntraNetware hálózaton**	**1.988.-**
... *pld Bócz P.-Szász P.*: **A Világháló lehetőségei** - Interaktiv weblapok készitése HTML4, JAVA 1.2	**2.990.-**
...*pld Séra Tamás:* **LOTUS NOTES 4.5** - 4.6 kiegészitéssel	**2.464.-**
... *pld. Tamás-Kiss-Tóth:* **MS-DOS 6 - 6.2; 6.22 kiegészítéssel**	**1.344.-**
... *pld Rudnai Péterné:* **NetWare 4.11** - felhasználóknak és rendszeradminisztrátoroknak	**2.576.-**
... *pld. Juhász-Kiss-Kuzmina-Sölétormos-Dr.Tamás-Tóth:* **DELPHI - Út a jövőbe** - 🖫	**1.999.-**
...*pld Benkő T.né - Tamás P.-Benkő L:* Windows alkalmazások **DELPHI 3** rendszerben CD melléklettel	**2.912.-**
... *pld Varga Márton:* **Játékprogramok készítése Pascal és Assembly nyelven** - 🖫	**1.456.-**
... *pld. Benkő-Tóth-Varga*: **Programozzunk TURBO PASCAL nyelven!** - 🖫	**1.490.-**
... *pld. Benkő T.-né és társai:* **Programozási feladatok és algoritmusok TURBO PASCAL nyelven (OOP példák is)** - 🖫	**1.488.-**
... *pld Benkő T.né-Benkő L.-Dr.Gyenes K.-Dr.Komócsin Z.:* **Objektum-orientált programozás Turbo Pascal nyelven - 7.0** 🖫	**1.978.-**
... *pld. Benkő T.né-Kiss Z.-Tamás P.-Tóth B.*: **Programozás Borland Pascal 7.0 rendszerben /DPMI, WINDOWS** - 🖫	**1.586.-**
... *pld. Kovácsné C. J.-Pergelné B. I.-Benkő L.*: **Mindenkinek! a PC-ről**	**799.-**
... *pld Váradi Zsolt:* **Hatékonyabb PC használat**	**990.-**
....*pld. Tóth Dezső:* **OS/2 Warp felhasználói ismeretek**	**1.680.-**
... *pld. László J.*: **Perifériák programozása Pascal és Assembly nyelven-** 🖫	**1.895.-**
... *pld. Stolnicki Gy.*: **SQL kézikönyv** -🖫	**1.799.-**
... *pld. László József:* **VGA kártya programozása** Pascal és Assembly nyelven - 🖫	**1.375.-**
... *pld. dr.Kovácsné Cohner Judit:* **Magyar WINDOWS 3.1**	**1.400.-**
... *pld. Benkő T.né-Kuzmina J.-Kiss Z.-dr.Tamás P.-Tóth B.*: **Könnyű a WINDOWS-t programozni!?** - 🖫	**2.400.-**
... *pld. dr. Kovácsné C. J - Ozsváth M. - G. Nagy J.*: **OFFICE 95**	**1.568.-**
... *pld. dr. Kovácsné C. J - Ozsváth M. - G. Nagy J.*: **OFFICE 97**	**1.985.-**
... *pld. Tóth B.-Tamás P és trsai:* **WINDOWS 95 & Microsoft Plus felhasználóknak - angol nyelvű változathoz**	**1.995.-**
... *pld. Tóth B.-Tamás P és trsai:* **WINDOWS 95 - magyar nyelvű változathoz - & Microsoft Plus felhasználóknak**	**1.960.-**
... *pld. Gerő Judit-Reich Gábor*: **WORD for WINDOWS 6.0** - magyar & angol	**1.499.-**
... *pld. Gerő Judit:* **WORD for WINDOWS 95 - 7 verzió-** magyar & angol	**1.960.-**

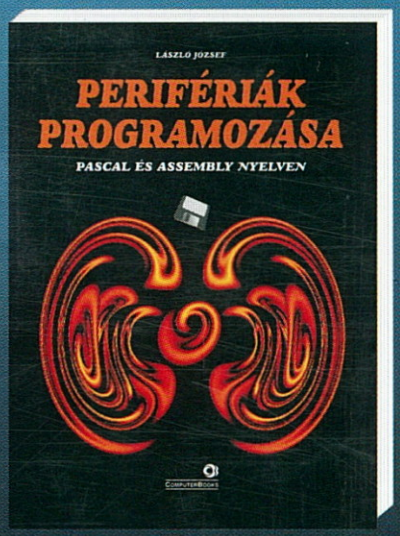
Cég, Megrendelő neve:_____

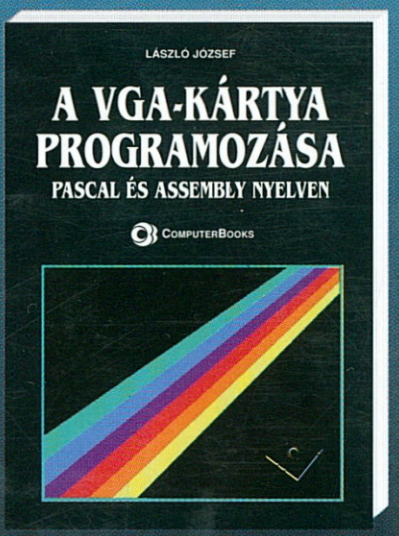
irányítószám, város:_____

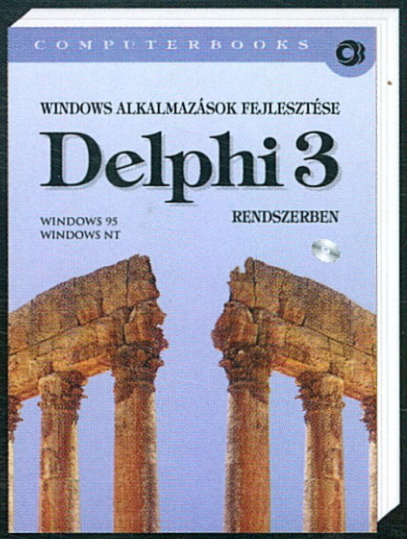
utca:_____

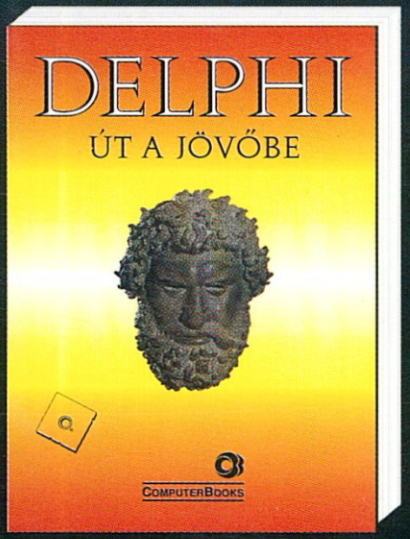
COMPUTERBOOKS

! Biztosan érdekel !









ISBN 963-618-184-5
9 789636 181840

Ára: 1.456,-Ft (Áfával)