第8章　図形処理

計算機に図形を表示/出力させたいという希望は計算機の初期の頃からの利用者の望みであった。それを最初に実現したのはサザーランドの「スケッチパッド」と呼ばれるシステムだったが、1960 年代前半のことであり、しかもその時点にして対話的グラフィクス (利用者の操作に応じて実時間で表示が変化する) であった。しかしその後も長い間、図形出力装置は特殊で高価なものであり続け、一般の利用者が図形表示の恩恵に預れるのはビットマップディスプレイやレーザープリンタが当り前になったここ数年のことでしかない。本章では、高度なグラフィクスよりは手もとのシステムで普段扱うような図形処理の話題を中心に取り上げる。

8.1　図形出力装置の歴史

8.1.1　ペンプロッタ

計算機が生まれてからずいぶん長い間、計算機と人間のやりとりの手段は「文字」ばっかりであった。これは、最初のころは計算機は文字通り「計算」の道具で、その後も「伝票処理」などがメインだったから、とりあえず計算結果の数表や帳票 (当然、字ばっかりである) が出力できれば十分だったからである。

しかし、技術者などは昔から数値をグラフなどの形にして検討するのに慣れていたから、計算機出力の数表を見て自分で図を描く代わりに、計算機から自動的に図が出てくる時代が来ることを待ち望んでいた。それを実現したのが「計算機で制御するペンにより、紙の上に図形を描く」装置である。こういうのを一般にプロッタ (ペンプロッタ) と呼ぶ。ペンプロッタはペンを交換することにより多色の絵を描くこともできたし、かなり大きな図を細密に描くこともできたが、一方で 1 枚描くのに時間が掛かる (メカだから)、一度描いたものは消せない、などの弱点も持っていた。

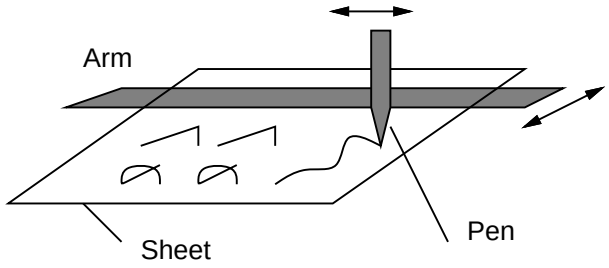

図 8.1: ペンプロッタの原理

8.1.2　グラフィックディスプレイ

プロッタより高価だがこれらの弱点を持たないのが CRT(ブラウン管) を使った図形表示装置である。[1] これは要するにオシロスコープのような原理で、電子ビームを蛍光スクリーンに当てて画面の特定の場所を光らせるのだが、その光る場所は偏向板 (またはコイル) の電圧によって X 方向、 Y 方向の位置をそれぞれ制御できるので、これを計算機によって制御して光る点を移動して図形の「一筆描き」を繰り返し、残像効果 (と蛍光材料の発光時間) を利用して人間の目には図形が見えるようにする、というものである。

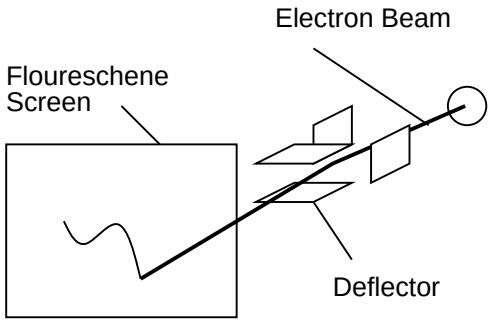

図 8.2: CRT 図形ディスプレイの原理

この装置がプロッタと違うのは、動く部分がないのでプロッタよりはるかに高速に絵が描けることと、描く線を変えればいくらでも図形の形も変化させられることである。このような装置によって始めて、(線ばかりの絵ではあるが) 利用者が「こんな絵を表示せよ」と言えばそれに即応できるような、「対話的」図形処理が実現できるようになった。

しかしこの装置にも弱点は多かった。まず図形が複雑になると「一筆描き」に時間が掛かるのでだんだん絵がちらついて見えるようになるし、[2] しかも、

[1]「スケッチパッド」もこのようなシステムである。

[2]この弱点をなくすため、一度発光させると高電場を掛けるまでずっと発行が続くような画面

表示できる絵は「単色の」「線画」に過ぎない。

なお、ここまでに述べた装置ではどれも文字を表示するには、その文字の線を他の図形と同様になぞって描くことが必要であり、時間も掛かるし見栄もよくない。

8.1.3 CRT 端末とビットマップディスプレイ

そうこうするうちに、上の方式とは別の原理で絵を表示する装置、すなわちテレビ装置が発達してきた。テレビ装置が上の方式と異なるのは、電子ビームは規則的に横の線を繰り返して画面全体を走査し (走査線)、各位置での電子ビームの強弱で発色の強さを変えることによって絵を表示させる点である。テレビの場合にはその強弱の信号はテレビカメラに映っている画像を電波で送ってくることによって制御しているが、計算機の表示装置として使う場合には代わりに画面の各点の明暗を計算機で制御することになる。

最初に作られたテレビを応用した表示装置は、計算機から文字コードを送るとその文字に対応した明暗のパターンを画面に作り出すというもので、要するにタイプライタを CRT に置き換えたようなものであった。これでは当然、文字しか表示できないし、文字の大きさや形も予め端末ハードウェアに組み込まれたものしか使えない。

しかし、1980 年代後半になると前回説明したビットマップディスプレイ、つまり画面上の 1 つの点をフレームバッファと呼ばれるメモリ上の 1 ビットに対応させて表し、これを読み出しながらビームの強弱を制御する表示装置が普及するようになった。(これが可能になったのは、メモリの価格が安くなって 1 画面ぶんのメモリ—1M ビット程度—を計算機に装備するのが負担でなくなったことが大きい)。

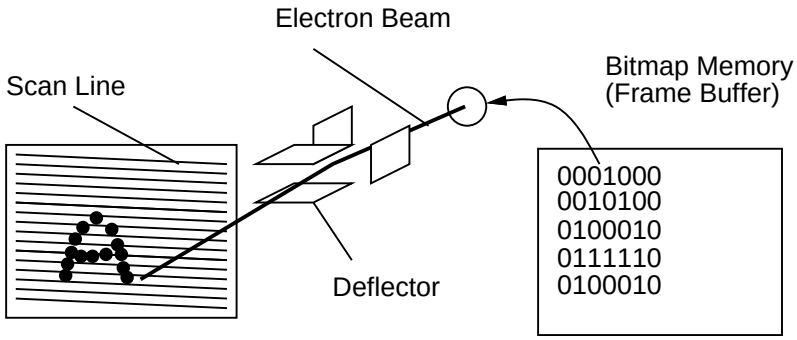

図 8.3: ビットマップディスプレイの原理

(蓄積管) を使用した表示装置もあったが、これはちらつきはない代わりに一度描いてしまうと「全部消す」以外の消し方ができないので、プロッタの画面版のようなものであった。

この方式であればソフトウェアの制御によってどのような形や大きさの文字でも (点の集まりで表せる限りは) 表示させられるし、図形を表示するのも自由である。また、画面の 1 点 (ピクセル、と呼ぶ) に対応する情報が 1 ビットしかなければ「白」と「黒」しか表せないが、このビット数を増やすことで多階調 (つまり複数の灰色の程度) を表示させたり、またカラーブラウン管を使用してカラー表示を行なわせることもできる。[3]

原理からいって、ビットマップディスプレイでは表示されるものは「点の集まり」であり、そのため曲線や斜めの線をよく見るとふちが「ぎざぎざ」になっているし、点の大きさより細かいものは絶対に表現できないという欠点がある。

しかし、その能力の範囲内ではどんな図形を表示してもちらちらしないし、またカラーブラウン管を用いれば色が使えるし、線画ではなく塗りつぶしができるので写真やテレビのようなリアルな絵も表示できるという利点がある。また機器そのものもテレビ技術の応用なので比較的安価に供給される (それでも高解像度のは高いけれど)。というわけで、現在では図形表示装置といえばほとんど全てがビットマップディスプレイになっているわけである。

8.2　図形処理ソフトウェアの構造

8.2.1　プロッタのソフトウェア

図形処理の初期の段階では、図形処理ソフトウェアと図形出力装置のあり方とは密接な関係があった。例えばペンプロッタに備わっている機構というのは「ペンを紙の面に降ろす」「ペンを紙から離す」「モータを動かして、ペンの位置を座標 (X, Y) に直線的に移動する」くらいで (つまり字を書くには毎回「書道」をやる必要がある!)、だからプロッタの使用を前提とした図形処理ソフトウェアには必ず `move(x, y)` だの `penup()` だのの命令が組み込まれていたものである。

もちろん、それだけでは困るので「文字列を描く」とか「円弧を描く」とかの命令もあったが、それらはライブラリの中で最終的には上記の呼び出し列に変換されるようになっていた。

このようなライブラリの代表として、カルコンプルーチンというのがあった。これはカルコンプという有名なプロッタ会社がプロッタの「おまけ」につけていた Fortran 用のサブルーチン群で、最初にこれがあんまり広まったのでカルコンプ以外のプロッタや図形表示装置の会社もこれと互換性のあるサブルーチン群を提供するようになり、「事実上の標準」になってしまった。

[3] プリンタ (というか、ハードコピーを取る装置全般) についてもに対応して、紙の上の各点での色を計算機制御による機械的/電子的/光学的手段でに紙の上に転写する、という原理のものが主流になっている。しかしプロッタもちゃんと生き伸びている、というのは大きな図面を高精度で描くにはこれ以上のものはまだ存在しないからである。

今でもこれの呼び出しで満ち満ちた Fortran による図形処理プログラムが山のようにある。

8.2.2 ビットマップディスプレイのソフトウェア

さて、一方ビットマップディスプレイではどうだろうか。前述のようにビットマップディスプレイの表示内容はフレームバッファと呼ばれるメモリに情報を書き込むことで行なう。マシンによっては OS に頼むと普通のプログラムからフレームバッファをアクセスできるのだが、我々の環境ではちょっと難しいので代りにフレームバッファと同様のイメージをメモリ内に作ってファイルに書き出し、xv と呼ばれるツールで見ることにしよう。

まず、メモリ内にビットイメージを作成するが、ここでは幅 320 ピクセル、高さ 200 ピクセルの白黒の絵にする。従って、1 ピクセルは 1 ビットに対応するので、1 バイト (=8 ビット) に 1 ピクセル入り、合計で $320 \times 200 \div 8 = 8000$ バイト必要であるので、大きさ 8000 の配列 (`buf` という名前をつけた) にこれを格納する。その様子を図**??**に示す。図から分かるように、`buf[0]`〜`buf[39]` までが先頭の 1 行、`buf[40]`〜`buf[79]` までが 2 番目の行、というふうに格納する。それぞれのビットが 0 であれば画面上では白、1 であれば黒の点として見える。

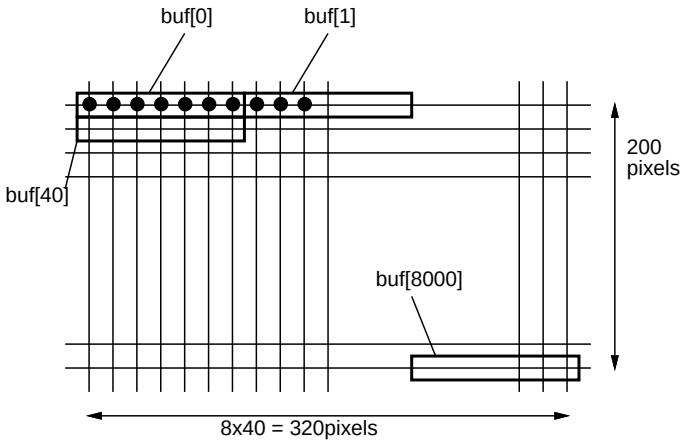

図 8.4: モノクロビットマップの配置

では、このビットマップ上で「斜め線」を作成して書き出すプログラムを示す。

```
/* t81.c --- create image and write in PBM format. */
char buf[8000];
#define black(x,y) buf[y*40+x/8]|=(0x80>>x%8)
```

```
#define white(x,y) buf[y*40+x/8]&=(0x80>>x%8)

main() {
  int i;
  for(i = 0; i < 200; ++i) black(i, i);
  write(1, "P4 320 200\n", 11); write(1, buf, 8000);
}
```

2 行目は配列 buf の宣言。3 行目と 4 行目は、位置 (x, y) の点を黒や白に設定するための操作に読みやすいように名前をつけて定義したもの (なぜこれでいいかは C の参考書を見てよーく考えよう)。そして、main の中ではいきなり斜め線上の各点を黒にしている (配列の初期値はすべて 0 だから最初は「真っ白」になっている)。そして、次の行で配列を書き出すが、まず最初にイメージファイルの種類や幅と高さの情報を書き出し、続いてイメージ本体を書いている (この「P4」で始まるイメージ形式は「PBM 形式」という。また、これのカラー版で PPM という形式もある。他にいろいろな形式のイメージファイルがあるが、プログラムによって扱えるものが決まっていることが多い)。

このプログラムがたとえば t81.c というファイルに入っているものとして、

```
% gcc t81.c
% a.out >test.pbm
% xv test.pbm
```

により、イメージファイルを test.pbm という名前で書き出し xv というイメージ表示コマンドで眺めることができる。その様子を図??に示す。

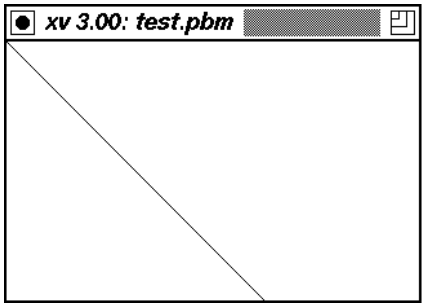

図 8.5: ビットマップ出力プログラムの結果

8.2.3 機器独立性と図形ライブラリ

ここまで見てきたように、出力装置の性質によってプログラムが大幅に異なる (デバイス依存、などという) のは困ったことである。さらに、たとえ同じ種類の装置でも画面の解像度や大きさによってまたプログラムを変更しないといけない。例えば先のプログラムは高解像度の画面 (横のドット数が 1280 ある) では全然動かない。

デバイス依存な図形処理プログラムは特定の出力装置でしか動かず、別の装置で使うには大幅な手直しが必要になり、厄介なものである。この問題を解決する 1 つの方法は、(ちょうどカルコンプルーチンがそうだったように)1 組のサブルーチン群を規定して、そのサブルーチン群は様々な図形出力機器に使用できるようにしておき、プログラムからはそのサブルーチン群の呼び出しのみを使って図形処理部分を記述することである。

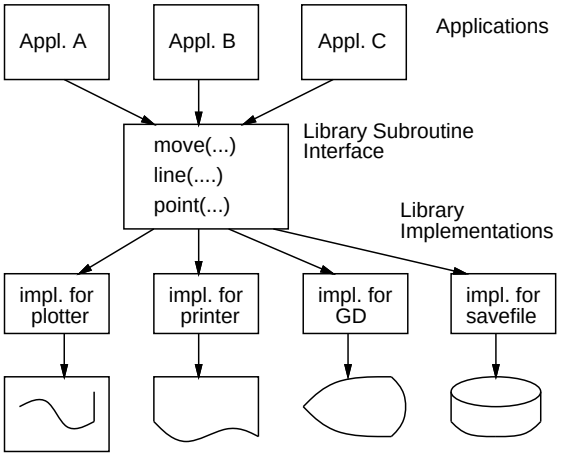

図 8.6: 機器独立なサブルーチンライブラリの概念

ここで注意すべきなのは、共通なのはあくまでもサブルーチン群のインターフェース (名前や呼び出し方の規則) のみであって、サブルーチンの本体は出力装置ごとにそれぞれ別個に 1 セットずつ書いてある、という点である。大変だと思いますか? でも実は、出力装置を売り出すメーカにとっては、その開発コストの一部でサブルーチン群を作ってしまえば、これまでにお客さんが持っているどのアプリケーションもそのままで走りますよ、というのは大きなウリであることを認識して頂きたい。

このようなライブラリとしては 2 次元モデルの GKS、3 次元の PHYGS や PEX などがあるが、ここでは (貧弱だけど)Unix に昔から備わっている plot ライブラリを紹介する。plot ライブラリは次のようなルーチンの集まりである。

```
    openpl();                   -- プロットルーチンの初期設定
    space(x0, y0, x1, y1);      -- 描画領域の設定
    erase();                    -- 描画領域のクリア
    line(x1, y1, x2, y2);       -- 直線を引く
    circle(x, y, r);            -- 円を描く
    arc(x, y, x0, y0, x1, y1);  -- 中心 (x,y) で、(x1,y1) から (x2,y2)
までの円弧
    move(x, y);                 -- 現在位置を (x,y) に設定
    cont(x, y);                 -- 現在位置から (x,y) まで直線を引
く
    label(".....");             -- 現在位置に指定した文字列を描く
    closepl();                  -- プロットの終了を示す
```

ではこれを使った例を示そう。

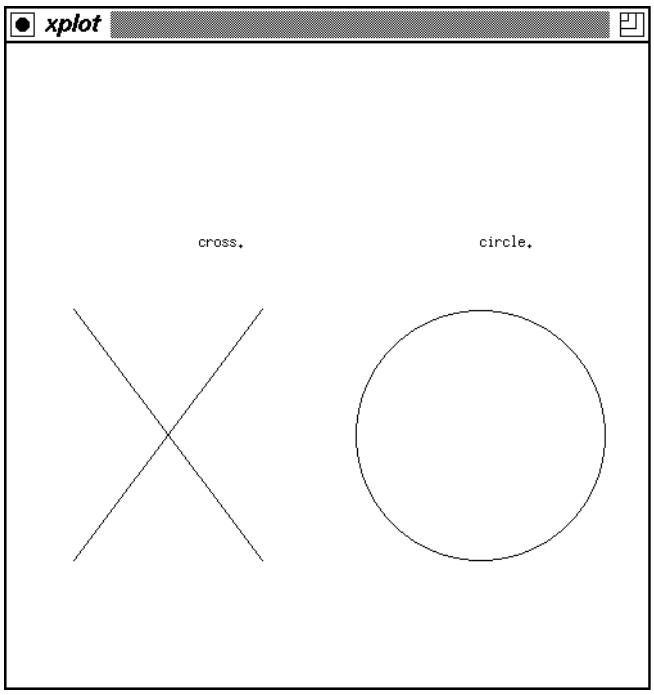

図 8.7: t82.c の出力の xplot による表示

```
/* t82.c -- An example of plot library usage. */

main() {
  openpl();
  space(0, 0, 1024, 1024);
```

```
    erase();
    line(100, 200, 400, 600);
    line(100, 600, 400, 200);
    circle(750, 400, 200);
    move(300, 700); label("cross.");
    move(750, 700); label("circle.");
    closepl();
}
```

これは単に「○と×を描く」だけのものである。これを翻訳するには

```
gcc t82.c -lplot
```

と指定する。これは「直接デバイスに描くのでなく、plot 形式ファイルを作成する」ライブラリを組み込む。[4] これを動かして X-Window 上で図形表示を見るには次のようにすればよい。

```
a.out | xplot
```

その結果を図 8.7 に示す。

plot 形式ファイルというのは要するに「こう描いてこう描いて...」という指示の集まりを直接機器に送る代わりにファイルに保存したものだと言える。こうしておくと、後で思い立った時に好きな機器に送れる、という点はよいのだが、次のような弱点がある。

1. 図の大きさがどれくらいになるかは全く出力機器まかせである。
2. そのくせ、出す前にちょっと大きさを変える、などの処理をするのは困難である。
3. そもそもファイルの中身を直接人が見ることができず、まして修正するのは困難である。
4. 複雑な図になると、ファイルの大きさばかり大きくなってディスクの無駄である。[5]

特に 1 は、このやり方が依然として「機器独立ではない」ことを意味している。これらの弱点を解消して、真に機器独立な図形出力を記述することをめざしたのが次に述べる PostScript である。

[4] その他に色々な図形端末がサポートされているが、うちには残念ながらそのどれもないので...

[5] ほんの短いプログラムでも巨大なプロットファイルを生成してしまうことは大いにあり得る。

8.3　ページ記述言語 PostScript

8.3.1　PS の概観

画面表示やプリンタ出力を行うプログラムを機器独立にするために、標準の「言語」を規定して「出力の内容をその言語で記述」するというアプローチがある。これを「ページ記述言語」という。Adobe 社が開発した PostScript(PS) はその代表的なものである。ごたくを並べるのはあとにして、まずはこれで四角を描いてみよう。

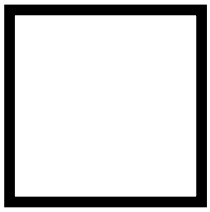

図 8.8: sam1.ps の結果

```
% sam1.ps -- a square  ← % はコメント
newpath               ← 線画はじめ
270 360 moveto        ← 線を引かずに移動
0 72 rlineto          ← 相対移動しながら線を引く
72 0 rlineto          ←  〃
0 -72 rlineto         ←  〃
-72 0 rlineto         ←  〃
closepath             ← 線画おわり
4 setlinewidth        ← 太さを指定
stroke                ← 描く！
showpage              ← 1 画面おわり
```

ちょっと慣れるまで違和感があるかも知れないが、PS は「後置記法」の言語であり、指令のパラメタは指令の**前に**指定する。

さて、これを画面に出すには X-Window 上で動作する PS インタープリタの 1 つである ghostscript により、

```
gs sam1.ps
```

とやると窓ができて表示が見れる。だいたい図 8.8 のような感じのものが見えるはずである。 gs を終るには、gs を起動した窓で^D を打つ。また、プリ

ンタに出すには... 実は、lw や ln などのプリンタは PS インタープリタを内蔵した、「ポストスクリプトプリンタ」なので、ただ

```
lpr -Plw sam1.ps
```

などとすばよい。

さて、PS を書いたファイルは確かに「読める」形はしているが (先の問題点 3. の解決)、でも plot と変わりないような... と思いますか? 実は、まず単位に注目して欲しい。PS の単位は標準では「ポイント」(つまり 1/72 インチ) であるので、この記述をもとに描かれたものはどの出力機器でも 1 インチ (訳 2.5 センチ) になるはずである (また、線の幅も 4 ポイントのはずである)。これによって問題点 1. が解決されるわけである。

8.3.2 PS と手続き

次は問題点 4. に行こう。例えばこの 4 角を何回も描きたいものとする。当然、場所を変えながら上のものを反復すればいいが、それではひどく大きなファイルになる一方である。そこで...「箱を描く」部分を**サブルーチン**として定義してしまう。見ていただこう。

```
% sam2.ps -- use of procedure in PS
/box
{ 0 72 rlineto
  72 0 rlineto
  0 -72 rlineto
  -72 0 rlineto closepath } def
newpath 270 360 moveto box 4 setlinewidth stroke
newpath 290 380 moveto box 8 setlinewidth stroke
newpath 310 400 moveto box 16 setlinewidth stroke
showpage
```

これで、「box」というとその場所に定義本体が埋め込まれた効果を持つので (実は本当にサブルーチン呼び出しが起きるのだが)、これでちゃんと箱が 4 つ描ける。線の幅はそれぞれ変えてある。(図 8.9)

8.3.3 座標変換

ところで、これでもまだ反復部分が目につく。それは、newpath した後で場所を指定しないといけないからである。そこでやり方を変えて、「box」はいつでも原点 (0,0) から始めて四角を描くことにし、毎回**原点を移動してから**これを呼ぶことにする。

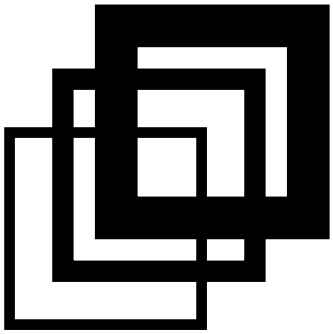

図 8.9: sam2.ps の結果

```
% sam3.ps -- use of translate.
/box
{ newpath
  0 0 moveto
  0 72 rlineto
  72 0 rlineto
  0 -72 rlineto
  -72 0 rlineto closepath
  2 setlinewidth stroke} def
200 200 translate box
40 40 translate box
40 40 translate box
40 40 translate box
40 40 translate box
showpage
```

出力は図 8.9 に示す。ちなみに線の幅はこんどは固定にしたが、手続きへの「パラメタ」の渡し方を覚えればもちろん変更できるようにもなる。さて、今度はまったく同じものの繰り返しになったので、普通のプログラムにならってループにしてしまおう! (出力は当然前と同じである。)

```
% sam4.ps -- use of repeat.
/box
{ newpath
  0 0 moveto
  0 72 rlineto
  72 0 rlineto
```

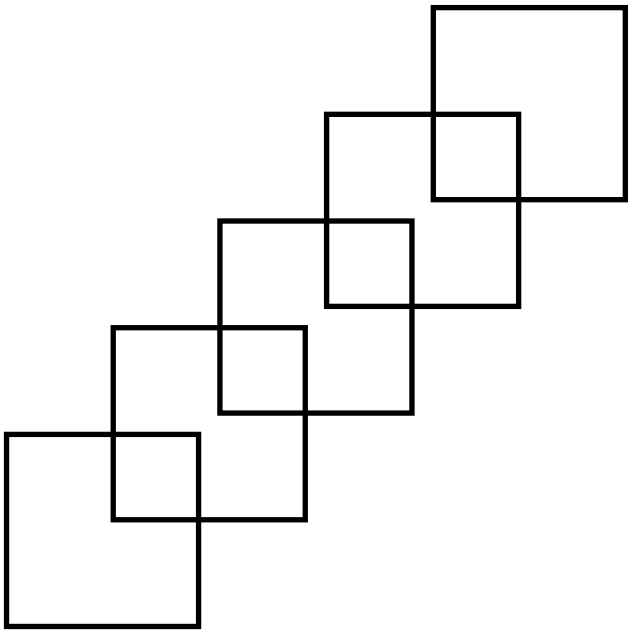

図 8.10: sam3.ps の結果

```
  0 -72 rlineto
  -72 0 rlineto closepath
  2 setlinewidth stroke} def
160 160 translate
5 { 40 40 translate box } repeat
showpage
```

ところで、「原点を移動」があるからには、当然「原点の回りに回転」とか「スケールの拡大/縮小」などもある。

```
% sam5.ps -- use of scale, rotate.
/box
{ newpath
  0 0 moveto
  0 72 rlineto
  72 0 rlineto
  0 -72 rlineto
  -72 0 rlineto closepath
  2 setlinewidth stroke} def
300 400 translate box
11 { 1.1 1.1 scale 22.5 rotate box } repeat
showpage
```

これで問題点 2. の解決方法も分かりましたね? つまり、絵を描く前に原点、スケール、回転をセットしておけばどんな位置/大きさにでも自由自在に直せるわけなのでした。一般に図形処理においては、適当な座標系で絵を描いておき、その位置や大きさは別途座標変換を指定することで任意に制御するというのが定石になっている。

ここまで見ると、PS は「ファイルの形式」というよりは、りっぱな「言語」(プログラムが書ける、という意味での) だということが分かる。特に図形を線の集まりで表す代わりに、線を描く手続き (サブルーチン) で表すことで記述がコンパクトにできる。これも図形処理の世界では一般的な定石である。

8.3.4　PS とフォント

実は機器独立な表示にはもう 1 つ、フォントの問題がある。ビットマップディスプレイが普及してからずいぶん長い間、文字のフォントは文字の大きさぶんのドット数の桝目に黒い点を (例えば bitmap のようなツールで) 配置することでデザインされていた。このようなフォントをビットマップフォントという。

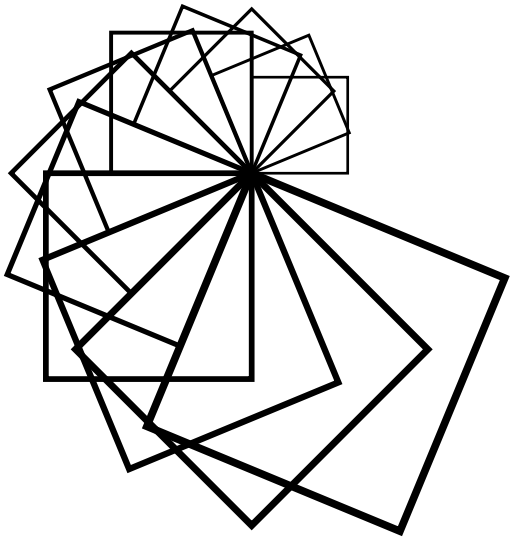

図 8.11: sam5.ps の結果

ビットマップフォントは当然、ある 1 通りの大きさでしか表示されない。(ある程度自動的に拡大・縮小することも試みられたが、その結果はだいたい美しくないのでプレビュー程度にしか使えないような品質である。) さらに困るのは、たとえば 16x16 ドットのフォントは出力機器の解像度によって、72dpi(dot-par-inch) の画面では 16 ポイントの大きさなのに 300dpi のプリンタでは 4 ポという小ささになってしまい、要するに機器依存 (この場合は解像度依存) が避けられないということである。

普通は各種ポイントサイズの文字を各種解像度向けに多数用意することで対処するのだが (xlsfonts で山のようにフォントがあったのはそのためである)、それでもたまたま中途半端な拡大率や解像度を使おうとすると困るし、だいいち扱いが面倒きわまりない。

そこで考えられたのがアウトラインフォントで、これは文字のビットマップをデザインするのではなく、輪郭の曲線を (曲線を描くような数式のパラメタを用いて) デザインする。数式だから、どの大きさにするのも定数を掛ければ自由であり、特定解像度の特定サイズの文字を表示するときにはそれ用の大きさに拡大した後でその輪郭内側に相当する点はどれとどれかを計算して画面やプリンタのピクセルを制御する。

これだけ見ても計算が相当大変そうだが、最近では CPU がとても高速なので、まあ実用に耐えるようになっている。現在では Macintosh、Win3、X のいずれもビットマップフォントに加えてアウトラインフォントを利用できるが、PS は「アウトラインフォントを最初に採用し、しかもアウトライン

フォントしか (普通は) 存在しない」という意味でとても先進的かつ徹底したシステムである。

ごたくはそれくらいにして、例を見ていただこう。PS ではフォントを使うには、まずフォント名を指定して `findfont` でアウトラインフォントを持って来て、次に `scalefont` で必要なポイント数のデータを作り出す。1 文字打つたびにそれをやるのではさすがに大変なので、作り出した結果は適当な名前で保存しておく。

```
% sam6.ps -- use of outline fonts.
/hv24 /Helvetica findfont 24 scalefont def
/tr36 /Times-Roman findfont 36 scalefont def
/cb12 /Courier-Bold findfont 12 scalefont def

hv24 setfont 100 100 moveto (This is Helvetica 24pt.) show
tr36 setfont 100 140 moveto (This is Times-Roman 36pt.) show
cb12 setfont 100 180 moveto (This is Courier-Bold 12pt.) show
showpage
```

実際に文字を書くには、このフォントデータを `setfont` により指定して、文字列をオペランドとして `show` 命令を実行する。文字列は「`(`」と「`)`」で囲む。(文字列のなかにかっこをいれたければ`\`を前に置けばよい。) 実行例を図**??**に示す。

図 8.12: sam6.ps の結果

8.4　より進んだグラフィクス

8.4.1　3 次元グラフィクス

さて、PS が機器独立でよいというのは分かったが、それでもその表示は何となく物足りないと思われませんでしたか? つまり

- 色がないし、
- 絵が立体的でない

ですよね? 色の方は単に我々のところはモノクロ画面が大部分だしこの資料も単色コピーだから略しただけで、PS では

赤 緑 青 `setrgbcolor`

という命令を使えば字や塗りつぶしを任意の色にできる。(3 原色の各色ごとの明るさを 0.0〜1.0 の数値で指定する。)

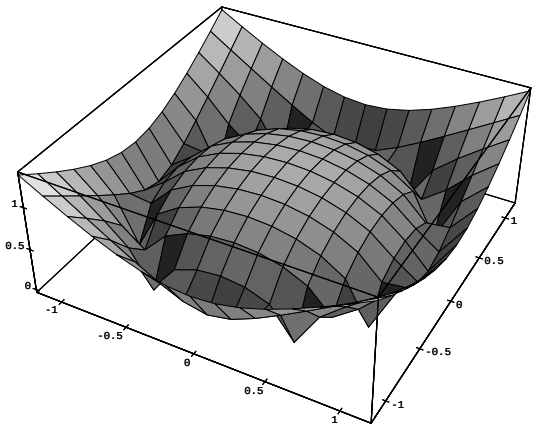

図 8.13: Mathematica の立体グラフィクス

一方、立体的の方はどうだろう? PS そのものは 2 次元グラフィクスモデルだけれど、Mathematica などのソフトはその上で簡単な 3 次元の絵を描かせるようになっている (図 8.13)。これはよく見ると図形を多面体で近似していることが分かる。

しかし最近の CG を駆使した映画などではもっとずっと高度 (リアル) な 3 次元の絵が使われていますね? それには次の 2 つの技術が使われる。

- レンダリング: 3 次元の物体の曲面を先の図 8.14 のように多面体で一担近似し、その各面ごとに「物体の色」「光源 (太陽やランプなど) と面の角度に応じた反射」などを計算し、その面を画面上に投射した範囲を計算した色で塗る。ただしその後、各面のつながりをスムーズにするよう平均などを取って調整する。
- 光線追跡 (lay tracing): 目の位置から画面上の各ピクセルを結んだ線を延長し、3 次元モデルの物体にぶつかる位置を求める。その位置での物体の色合いを求め、さらに物体が「つるつる」なら線を「反射」させて

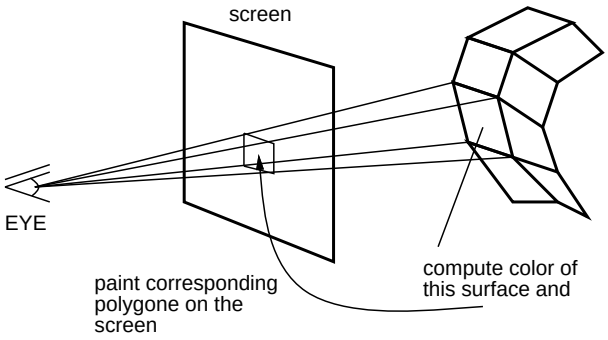

図 8.14: レンダリングの原理

その先へ進む。物体がレンズみたいに「(半) 透明」なら反射光と屈折光の両方を処理する。その結果求めた「色」でそのピクセルを塗る。これを画面上の全ピクセルについて行う (図 8.15)。

これらを比べると光線追跡の方が (反射や屈折まで扱うから) ぐっとリアルな絵が作れるが、計算量は膨大になる。一方、レンダリングは多面体近似の荒さによって計算の節約がコントロールできるので、最近のグラフィクスワークステーションでこの計算を行うための専用ハードウェアを備えているものでは、かなりリアルな動く立体画像が作れる。

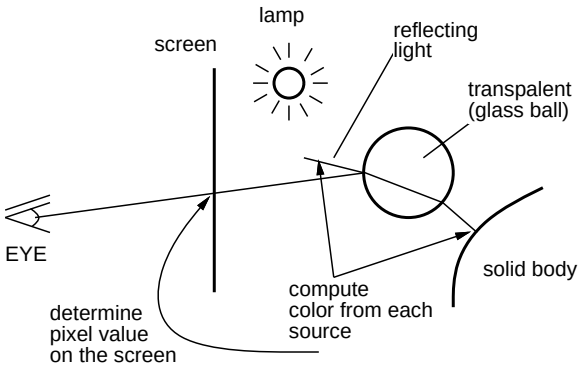

図 8.15: レイトレーシングの原理

8.4.2　対話的グラフィクス

ここまでに述べてきた図形処理では、絵は基本的に「プログラムが一方的に作り出してくる」形であった。しかし実際には、プログラムと利用者が対

話しながら行う図形処理も多く使われる (CAD — computer aided design、計算機を使った製図のようなもの — が代表的)。このようなものを対話的グラフィクスという。

対話的グラフィクスでは、単なる絵の表示に加えて次のような処理が必要となる。

- 利用者がマウスなどで「ここ」と指したときに、それが計算機の中でのモデルで「どこ」かをつきとめる機能 (ピッキング)。
- 利用者が指定した箇所を「こう動かせ」と指令した時にそれに応じて実時間で図形を変化させる機能。

特にピッキングした時そのままマウスを使って掴んだ位置を動かす場合にはこれを「直接操作」という。図形があまり複雑でなければ図形全体をマウスの移動につれて描き直すが、それが無理な場合には枠線や骨格線など簡単に描き直せるような形 (マウスに追従してゴムみたいに線が伸び縮みするので、「ラバーバンド」などとも呼ばれる) のみが追従し、最終位置が確定したところで改めて全体を描き直すなどの工夫が必要になる。

ここでちょっとでも対話的グラフィクスの気分を味わっていただくために、tcl/tk で記述したデモをお見せしよう。これは、黒い 4 角形の 4 頂点を直接操作で動かして変形させられるというものである。(図 8.16)。

```
#!/usr/local/bin/wish -f
proc mkbox {n x y} {
  global id px py
  set x1 [expr $x-5]; set x2 [expr $x+5]
  set y1 [expr $y-5]; set y2 [expr $y+5]
  set id($n) [.can create rectangle $x1 $y1 $x2 $y2 -fill white]
  set px($n) $x; set py($n) $y
  .can bind $id($n) <B1-Motion> [list mvbox $n %x %y]
}
proc mvbox {n x y} {
  global id px py tr
  .can move $id($n) [expr $x-$px($n)] [expr $y-$py($n)]
  set px($n) $x; set py($n) $y
  .can coord $tr $px(1) $py(1) $px(2) $py(2) $px(3) $py(3) $px(4) $py(4)
}
. configure -geom 480x220
canvas .can  -relief sunken
place .can -x 10 -y 10 -width 400 -height 200
mkbox 1 50 50; mkbox 2 50 150; mkbox 3 150 150; mkbox 4 150 50
set tr [.can create polygon 50 50 50 150 150 150 150 50 -fill black]
```

```
button .b1 -relief raised -text "出力" -command { .can postscript -file out.ps }
button .b2 -relief raised -text "終了" -command { exit 0 }
place .b1 -x 420 -y 20; place .b2 -x 420 -y 50
```

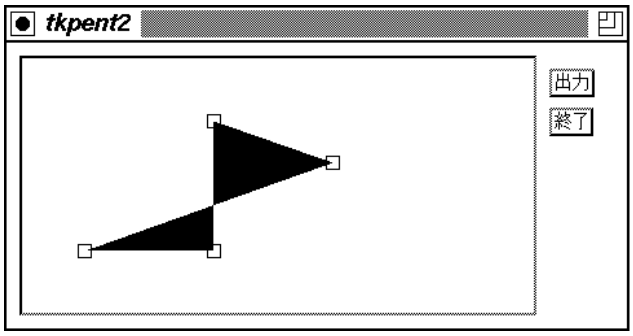

図 8.16: tkpent の実行の様子

このコードの概要は次の通り (都合上順番は上からではない)。

- 16-18: 最外側の窓の大きさを指定し、canvas つまり図形を描くウィジェットを作り、位置を設定。
- 19: 4 つの箱を作る。
- 20: 4 つの箱の位置を頂点とする 4 角形を作り、その図形 ID を変数 tr にいれる。
- 21-23: 出力ボタン (押されると canvas にその内容を PS 形式でファイル `out.ps` に書き出すように指令する) と終了ボタン (押されると実行を終わる) を作って、配置する。

のこりは 2 つのサブルーチンであるが:

- 2-3: サブルーチンの名前と引数を規定し、広域変数のアクセスを指定。
- 4-7: 指定された座標を中心とする正方形を作り、その ID を配列 id に格納する。また x、y 座標を配列 px、py に覚えておく (n は正方形の番号)。
- 8: この正方形の上でマウスボタンが押されたまま動いたら mvbox 命令を、正方形番号、マウスが押された位置の x、y 座標を引数として呼び出す。
- 10-11: サブルーチンの名前、引数、広域変数アクセス指定。
- 12-13: この正方形の位置を、マウス移動量に対応してずらす。また新しい x と y の位置を覚え直す。
- 14: 4 角形の 4 頂点の位置を設定し直す。

このように、対話的グラフィクスの場合にはマウスなどの操作に従って、表示中の図形に対応した変形を施す必要があり、その処理は結構面倒である。また図形が複雑になってくるとマウスの動きに追従するのが CPU 能力の面から苦しくなってくるなどの難しさもある。(そのためラバーバンドを使ったりする。)

8.5 描画ソフト

8.5.1 ペイント系とドロー系

PBM/PPM ファイルや PostScript をプログラムやエディタで出力させることは可能だが、ものが絵だけにやはりどんな絵になるかを画面で見ながらでないと複雑なものは描きにくい。このためのソフトが描画ソフトで、読んで字の通り「絵を描く」ためのソフトである。描画ソフトは大きく分けて

- ペイント系 — PBM/PPM のように、描画面は様々な色の点 (ピクセル) の集まりで、個々のピクセルの値を設定することで絵を描く。
- ドロー系 — 幾何学図形 (直線、円、長方形、多角形など) を描画面に配置していくことで絵を描く。

の 2 種類に分けられる。(これらの名前はそれぞれに対応する代表的な/ 最初のソフトである MacPaint と MacDraw から来ている。)

ペイント系では、絵を描くということはそれぞれのピクセルの値を書き換えてしまう (そこに前にあった絵は上描きされてしまう) ので、一度描いてしまった絵の位置を移動したり大きさを変えたりするのは困難だが、その代わり個々の点に任意の色がつけられるのでビットマップディスプレイの表示能力を最大限に活かした絵が作れるという利点を持つ。

これに対し、ドロー系の場合は予め決められた図形の部品を組み合わせた絵しか描けないが、無限に伸縮可能な針金でできた枠に無限に伸縮可能なスクリーントーンを貼ったものを「置いていく」ようなものなので、置いてみて場所や大きさや重ね順が気にいらなければいくらでも直すことができる。

これらの特性から、一般にペイント系は「写真」や「絵」に向いているのに対し、ドロー系は「図」や「グラフ」などに向いているといえる。以下では実際に両者の具体例を見てみよう。

8.5.2 ドロー系ソフト k2d

ドロー系ソフトとしては、古典的名作かつ定番でもある Macintosh 用の MacDraw の「そっくりさん」である、フリーソフト kdraw を筆者が改良して使っている k2d を取り上げる。その起動方法は単に

```
k2d -m &
```

というだけである。しばし待つと、図 8.17 のような窓が現れる。その上の方に「File Edit Structure ... Option」というのが見えるが、ここがプルダウンメニューになっている。

これで基本的な図形を描くには、左端の図形一覧の中から描きたい図形を選んで黒く表示させておき、その状態で左のマウスボタンを使って描く。直線、円、四角など 2 点を指定すると定まるものは、左ボタンを押し下げて始点を示し、押し下げたままマウスを移動して終点でボタンを離す。

一方、折れ線、カーブ、多角形、閉じたカーブでは各接点ごとに 1 回ずつ左ボタンを押しては離し、一番最後の点だけは中央ボタンを押して離す。

文字を入れるときは「Text」の所を黒く反転させてから左ボタンで入れたい場所を指定し、あとはキーボードで打ち込む。打ち込み終わったら [ESC] を打てばよい。

漢字を入れたい場合には、まず前回やった kinput2 を smb の窓で「kinput2 &」により起動しておく必要がある。そして、入力は「KText」を反転させてから英字テキストと同様に打ち込むが、例によって「^\」でローマ字モードに切り替えると kinput2 の窓が開いてかな漢字変換が使える。終わりが [ESC] なのは同様である。

文字のフォント、線の形、図形の塗りつぶしパターンなどはすべて描いた直後にメニューで選ぶことができる。

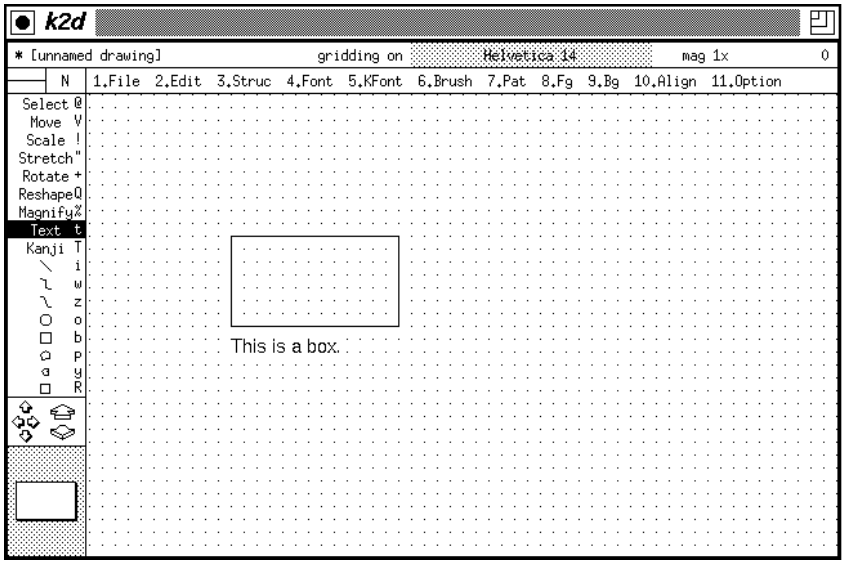

図 8.17: k2d の窓

ところで、図形の回り 8 箇所に小さい四角が表示されている時、その図形は

「選ばれている」という。いつでも右ボタンを図形の上で押すこと出図形を選ぶことができる。後からパターンなどを変更したい時は図形を選んだ状態でメニューを使えば良い。同様に、図形を消したり (Cut)、複製したり (Duplicate) するには選んだ状態で Edit メニューからそれらの機能を選べば良い。一方、中央ボタンは上で述べた場合以外は常に移動機能になっていて、図形の上で押して、そのままマウスを動かすことで図形を好きな位置に持っていける。

最後に、描いた図形をファイルに保存するには File メニューの中の「Save」を使用する。別の窓が現れてファイル名を聞いてくるので (「なんとか`.ps`」という名前にすること)、ファイル名を入れて [ret] を押すと保存される。k2d を終るには同じく File メニューから「Quit」を選ぶ。一度描いた図形を再度編集したい時は File メニューの「Open」を選ぶとファイル名の窓がでてくるので、ここでファイルを選ぶかファイル名を打ち込む。

`k2d` で描いた図形は PS プログラムそのものである。だからそれを `gs` で見ることもできるし、打ち出すには `lpr -Plw` を使えばよい。

8.5.3 ペイント系ソフト xpaint

ペイント系ソフトで Unix 上で動くフリーのものはあまり多くないが、とりあえず xpaint が代表的なのでこれを使う。xpaint を起動するには

```
% xpaint &
```

と打ち込む。しばらくすると、図 8.18 のような窓が現われる。

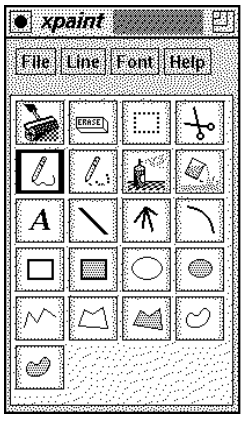

図 8.18: xpaint のツールパレット窓の様子

ちょっと、k2d のパレットに似ているでしょう? ただし、xpaint では後で出てくる色パレットと区別するため「ツールパレット」と呼ぶことにしよう。xpaint の場合、絵の大きさを指定してから描き始めるので、まずツールパレッ

トの窓が出て、ここからキャンバスの窓を開くようになっている。なお、ツールパレットの上の方にある「File」などは k2d と同じくプルダウンメニューになっている。

さて、以下では当面、特に述べない限りマウスの左ボタンを使用すること。キャンバスを開くには、「File」メニューの中から「New With Size」を選ぶ。すると、幅と高さと拡大率を打ち込む窓が出て来るので、ここでは練習用に幅と高さは 64、拡大率は 8 として (どれも既に値が入っているところをマウスでクリックして [DEL] キーでもとの値を消して新しい値を打ち込む)、最後に OK のボタンをつつくと図 8.19 のような四角い窓が現われる。

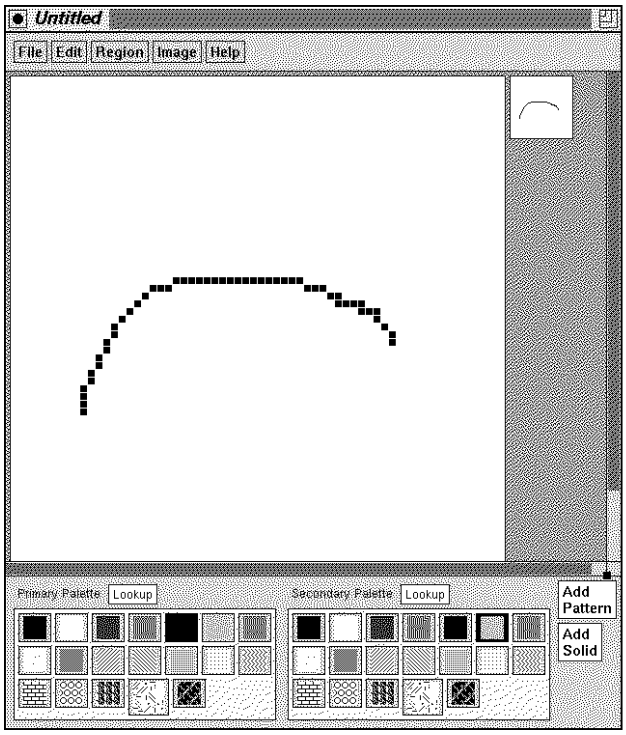

図 8.19: xpaint のキャンバス窓の様子

キャンバス窓の下には 2 つの「色パレット」があるがが、これは左側が鉛筆や筆の色、右側がスクリーントーンの色と思えばよい。とりあえず両方のパレットとも模様ではない色 (で互いに違うもの) をクリックして選んでおくこと。

つぎに、ツールパレットから「鉛筆」マークを選び、キャンバスの上でどれかのボタンを押して動かす。すると、鉛筆の動いた後の点に色がついて残る。鉛筆の太さを変えたい時はパレットの「Line」メニューを使えばできる。

もっと太い線を描きたい場合にはパレットから「ローラ」を選んで使う。

ローラの形はいく通りかに変えられるが、変えたい場合にはローラのパレットを右ボタンでつつくとメニューが出て、その中から「Select Brush」を選べばよい。ローラの隣にあるのは「消しゴム」で、これは間違って書いてしまったところを白に戻すのに使う。消しゴムもローラと同様形が変えられる。「スプレー」はやってみれば分る通りスプレーペイントのような効果がある。「バケツ」は、閉じた形を作ってその中でバケツをこぼすと色が行き渡る。やってみて「失敗した!」と思ったときは、キャンバスの「Edit」メニューの「Undo」を選ぶと 1 回ぶんだけは操作を元に戻すことができる。

あと、直線や四角などの図形のパレットは k2d と同じように使えるが、ただしペイント系なので一度描いてしまったものは動かしたり消したりできない。なお、図形で中が灰色のものは、描いた後で中を色やパターンで塗る。その色やパターンはキャンバス窓の下右側のパレットで選ぶ。文字は文字が描けるが、ただし xpaint では漢字は使えない (どのみち文字はあまり重要でない)。

さて、キャンバス窓の色パレットについてだが、もっと別な色を使いたい場合には右下の「Add Solid」と書かれたボタンを押す。すると図 8.20 のような窓が現われるので、ここで円形のうち望む色のところを左ボタンでクリックするか、またはスライドレバーのところでマウスの中ボタンを押して動かし、望む色が出た所で「OK」をクリックすればその色がパレットに追加されて利用可能になる。その他の機能については、「Help」ボタンなどを利用して調べて欲しい。

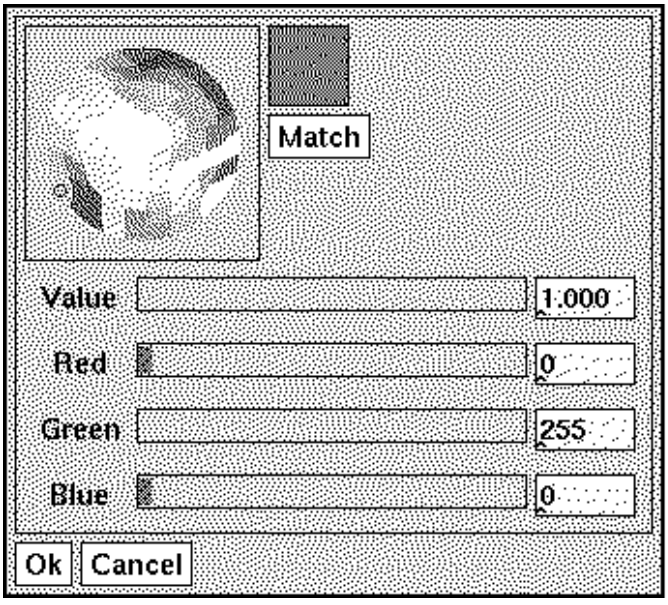

図 8.20: xpaint の色パレット窓の様子

最後に、絵が完成して保存する時は例によって「File」メニューから「Save As」を撰び、ファイル名を入力し [RET] する。ところで、絵を保存するとき様々なファイル形式が選べるが、ここでは「GIF」を選択する。もし別の形式にしたければファイル名を入れる前に使いたい形式のボタンをクリックして選んでおく。なお、GIF のファイルは「test.gif」のように最後を「.gif」で終わらせておくとよい。

さて、これを何に使おうか? もちろん、これで WWW のページに自作の絵が入れられる。そのためにはまず、

```
% mv test.gif WWW/test.gif
% chmod go+r WWW/test.gif
```

のようにファイルの位置をサブディレクトリ WWW の下に移動し、保護設定を変更しておく。あとは、自分の WWW ページの HTML ファイルに

```
<IMG SRC="test.gif">
```

のようにタグを書けば、そこに今描いた絵が埋め込まれる。(多少ソフト的に問題があって色化けするかも知れません。原因は探求中です。)

8.5.4　おまけ: k2d の秘密

k2d は kdraw の「改良である」と先に書いたが、その話を簡単にしておこう。実は「-m」オプションを指定しなかった場合は、k2d は「キーボードによる図操作」ができるようになっている。例えばマウスでいくつか長方形を描いた後、次のコマンドを実行してみて欲しい:

```
^P --- 選択図形の 1 つ前に描いた図形を選択
^N --- 選択図形の 1 つ後に描いた図形を選択
```

順番だけでなく、次の方法でも選択できる:

```
^H --- 選択図形の左側の図形を選択
^J --- 選択図形の下側の図形を選択
^K --- 選択図形の上側の図形を選択
^L --- 選択図形の右側の図形を選択
```

次に、選択した図形を移動することができる。

```
h --- 選択図形を左に 1 目盛り移動
j --- 選択図形を下に 1 目盛り移動
k --- 選択図形を上に 1 目盛り移動
l --- 選択図形を右に 1 目盛り移動
```

これらの前に「10」などと数値を入れておくとその目盛り数だけ移動できる。また、Meta キーを押しながらやれば 1 ドットずつ移動するので、位置の微調整に使える。

「d」を打つと選択図形が複製されるので、例えば「d10jd10j」と打つと選択図形を 3 個、10 目盛りピッチで並べることができる。

「r」を打つと選択図形の大きさ変更モードになり、h/j/k/l のキーで大きさ枠の右下隅を任意に移動し、[RET] を打つとその時点での枠に対応した大きさに変更される。

最後になったが、「n」を打つと現在選択されているパレットの図形を新規に作ることができる。この説明を始めると長くなるので省略するが、例えば「tn.....[ESC]」で選択図形の近辺に説明の文字列などを入れることができる。より詳しい説明が知りたければマニュアルがあるので久野までどうぞ。

さて、なんでこんなことをやっているのだろう? 図形なのだからマウスで扱うのが自然だし分かりやすくないか? もちろん、そうなのだが、しかし「分かりやすい」のと「す速く操作できる」のは同じでない、という話を前にもしましたね? つまり、キーボードによる図形操作は最初はとっつきにくいが、慣れるとマウス操作よりずっと速くできるという考えのもとにこれをやっているわけである。よかったら体験してみて頂きたい。

8.6 演習

11-1. プログラム t81.c を動かしてみよ。また、うまく動いたらこれを直して次のいずれかをやってみよ。l

1. 白地に黒線でなく、黒地に白線にしてみよ。
2. 45 度以外の角度の斜め線をいくつか描け。
3. 円を描いてみよ。

11-2. 同じくプログラム t82.c を動かしてみよ。次に、四角形の中を斜め線 (難しければ水平線) で塗ったものを描いて見よ。両者のファイルの大きさはそれぞれ何バイトか較べて、できたらその結果からファイル形式を推定してみよ。

11-3. ポストスクリプトの例題 sam1.ps〜sam6.ps を動かした後、次のことからどれか 1 つやってみよ。

- 図 8.21 のような図 (難しければそれに近いもの) を PS で書け。名前はもちろん自分の名前でどうぞ。
- さらに、それを複数個、大きさや位置や角度を変えて 1 画面に出してみよ。

- k2d や tkpent で出力された PS ファイルをエディタで編集して、大きさや位置を変更して紙に出力してみよ。
- k2d の出力と tkpent の出力を 1 枚の紙に一緒に出してみよ。

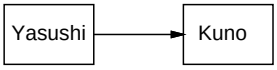

図 8.21: 例題用の図

11-4. k2d を動かし、次のいずれか 1 つ以上やってみよ。

- 図 8.21 のような図を描け。
- 前にやったネットワーク地図をそれらしく清書してみよ。
- k2d のキーボードコマンドで図 8.21 のようなものを描くには次のように打鍵する。

```
bn[RET]ll[RET]11dtnKuno[ESC]^HtnYasushi[ESC]^P^P^Pi1n[RET][RET]3/2.
```

この各操作がどういう意味を持っているのか、調べてみて解説せよ。

11-5. xpaint を動かし、何か好きな絵 (できれば自分のトレードマーク) を描いてみよ。できれば自分の WWW ページにその絵を入れられるとなおよい。

今回の課題は上記から 3 問選択してください (問題の中にも選択肢がありますが、それもどれか 1 つを選べば結構です)。