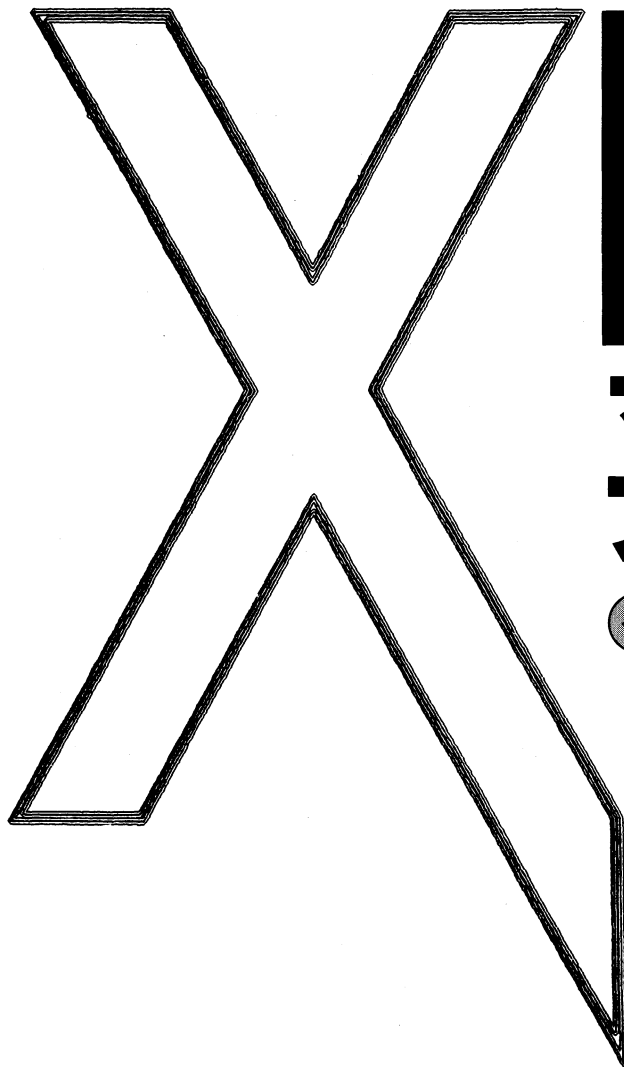




68000

マシン語
プログラミング

グラフィックス編

著・村田敏幸

●本書に掲載したプログラム名，システム名，CPU 名などは一般に各社の登録商標です。本文中では，とくに TM，R マークは明記していません。

©1992 本書並びにディスクのプログラムを含むすべての内容は，著作権法上の保護を受けています。

著者，発行社の許諾を得ず，個人使用の範囲を超えて無断で転載，複製することは禁じられています。

なお，添付ディスク収録のファイルの取り扱いについては APPENDIX を参照下さい。

《入門編》から約1年、お蔭で『X68000マシン語プログラミング』の第2弾、《グラフィックス編》をこうしてお届けすることができた。タイトルどおり、『Oh! X』誌上の連載のうちグラフィック描画を扱った期間の単行本化である。ただし、連載では同期間中に含まれるソーティングを扱った回は、バランス上、割愛した。その代わりといっはなんだが、プログラム例は多少なりとも増えており、連載の読者の方にもいま一度楽しんでもらえると思う。

本書では線分、円の描画から画像の拡大・縮小、回転、変形に至るまで、さまざまなグラフィック描画ルーチンをただひたすらにつくりまくる。その点では、《入門編》以上に実践テキストとしての性格が濃い。たぶん、ぼーっと本文を読んだところで、得られるものは何もないに違いない。読者はむしろプログラムを読み、改造し、作り直すことに集中してほしい。結局のところ、プログラミング技術は自分の頭と体を使わない限り身につくことはないのだ。扱う題材がどう変わろうと、僕の言いたいことはこの1点に尽きる。思い上がりかもしれないが、この主張ゆえに、本書はグラフィックス書の出来損ないではなくプログラミング書でいられるのだと思っている。

各章は比較的独立しているので、Chapter3の前半ぐらいまでを押さえておけば、後は興味のあるテーマだけを拾い読みしてもらってかまわない。Chapter7, 8のあたりは中盤の章にしては難易度が低めだから、早い時期に読んでみてもいいだろう。逆に大物の部類に入る多角形の塗り潰しを扱うChapter4は、ざっと目を通した段階でヤバいと思ったら、さっさと先の章に進んでしまうことを勧める。

なにはともあれ、本書が読者の知的好奇心を満たすのに少しでも役立てれば、著者としてこれほど嬉しいことはない。

1991年初冬

村田 敏幸

まえがき.....3

CHAPTER

1 IOCSコールによるグラフィックス.....9

IOCSコール.....	10
グラフィックス関連IOCSコール.....	12
画面の初期化.....	14
マウスとグラフィックスの組みあわせ.....	16
簡易お絵かきツールの実際.....	20
エントリ〜終了(69行〜).....	28
初期化ルーチン(306行〜).....	28
メニューウィンドウの初期化(340行〜).....	29
メイン処理(106行〜).....	29
後始末(405行〜).....	30
オリジナル描画ルーチン作成を目指して.....	30

CHAPTER

2 直接グラフィック画面を操作する.....31

X68000のグラフィック画面.....	32
矩形の塗り潰し.....	37
矩形のクリッピング.....	40
ボックスフィルルーチンの高速化.....	45
タイリングへの対応.....	51
COLUMN●本書のサブルーチン共通仕様.....	53

CHAPTER

3 必須のラインルーチン.....55

線分描画のアルゴリズム	56
ラインルーチンの原型	61
線分のクリッピング	63
そして高速化	71
ダブルステップBresenhamアルゴリズム	83
もっと速く	90
性能評価	100
COLUMN●4 連結の線分発生アルゴリズム[参考]	106

CHAPTER

4 多角形フィリング

スキャンコンバージョン	110
プログラミングに向けて	113
多角形の内部表現	113
クリッピング	114
多角形とスキャンラインとの交点算出	115
交点のソーティング	117
 実 装	118
クリッピング部(リスト2)	119
辺リストの作成部(リスト3)	122
描画部(リスト4)	124
 動作試験	129

CHAPTER

5 ペイントルーチンの作成

シードフィルアルゴリズム	134
ペイントルーチンの実例	136
アルゴリズムを見直す	142
ペイントルーチンのバリエーション	148

CHAPTER

6 円を描く 151

円描画アルゴリズムを導く	152
円描画ルーチンの作成	156
楕円への対応	162
円の塗り潰し	166

CHAPTER

7 画像に変化を与える処理 171

色コード	172
カラーコード	172
パレットコード	173
HSVによる色指定	177
色の反転	182
色の量子化	188
モノクロ32階調に	189
モザイク処理	194
平滑化	197
鮮鋭化	201
色強調	203
ディザ法による階調変換	205
色の加減算と合成	216
グラデーション	219

CHAPTER

8 グラフィックパターンの扱い 227

ゲット:パターンの取り込み	228
プット:パターンの書き込み	230
プットルーチンいろいろ	233
矩形領域の複写	239

左右上下の反転	242
連続複写	245

CHAPTER

9 画像の拡大・縮小

拡大・縮小のアルゴリズム	248
拡大・縮小ルーチン試作版	251
描画の高速化	253
クリッピングの追加	255
ふたたび高速化	260
バリエーション	262

CHAPTER

A 画像の回転

画像回転のアルゴリズム	274
回転ルーチンの実例	279
サブルーチンrotate (171～246行)	287
サブルーチンline_comp (252～305行)	288
サブルーチンclip (325～424行)	288
サブルーチンexpand_comp (431～453行)	289
サブルーチンgrput (36～105行)	289
とどめの高速化	289

CHAPTER

B 自由変形

自由変形のアルゴリズム	292
プログラムの作成	295
初期化 (74～146行, 296～344行)	302
メインループ (148～187行)	303
1ライン分の描画 (196～291行)	303

禁断の自己書き換え	303
自由変形ルーチンの応用	305

CHAPTER

[C, X-BASICとのリンク	311

C言語における変数の扱い方	312
Cの関数をマシン語で	316
X-BASICの外部関数	320

APPENDIX : 添付ディスクについて	331

あとがき	337

INDEX	339

CHAPTER

1

ーOCSコールによるグラフィックス

IOCSコールによる グラフィックス

ASSEMBLER



肩ならしの意味で、X68000に最初から用意されているサービスルーチン群であるIOCSコールを使ってグラフィック描画を行うことから始めよう。この章では、とくにグラフィック画面の初期化手順をしっかりと押さえてもらいたい。

IOCSコール

X68000ではROMにIOCS (Input/Output Control System) のかたちでさまざまなハードウェア機能の制御ルーチンが用意されている。IOCSは、テキスト画面への文字表示、キー入力に始まって、プリンタ出力、RS-232Cによる入出力、フロッピーディスク/ハードディスクの物理的な読み書き、マウスの制御、グラフィック描画、スプライト、AD PCM、カレンダー時計などの周辺LSIの制御にいたるまで、X68000の（ほとんど）すべての機能を網羅している。位置付けとしてはシステム中、もっともハードウェアに近い部分を担当し、X68000上のプログラムを底辺からささえる低レベルI/Oルーチン集¹⁾であり、OSであるHuman68kもIOCSに乗ったかたちでつくられている。ユーザプログラムがHuman68kにDOSコールを通じて入出力を要求すると、Human68kは必要におうじてIOCSを呼び出し、最終的にIOCSがハードに働きかけて物理的な入出力を行う²⁾。結果は逆のルートを伝って返される。このHuman68kとIOCSとの上下関係（というか依存関係というか階層構造というか）は心に留めておいてもらいたい。

■ 1) 実際にはハードウェアがらみ以外にも、シフトJIS漢字コード ↔ JIS漢字コードの相互変換とか、ユーザモードからスーパーバイザ空間にあるメモリを読み書きするといったユーティリティ的なものもIOCSには用意されている。

■ 2) 論理的には、OSの低レベルI/Oはデバイスドライバに相当することになっているわけだが、現実にはHuman68kのデバイスドライバはさらに下位の存在であるIOCSを下請けに使っている場合が多い。

IOCSを呼び出す手順はいたってかんたんで、d0.lにIOCSコール番号を入れてtrap #15という命令を実行するだけだ。引数がある場合は、d1以降のデータレジスタやa1以降のアドレ

レジスタに入れて渡す。たとえば、IOCSコール番号20_Hに割り当てられている1文字表示機能を使うときには、

```
move.w    #'A',d1
moveq.l   #$20,d0
trap      #15
```

IOCSコール番号21_Hの文字列表示機能を使うのなら、

```
lea.l     mes,a1
moveq.l   #$21,d0
trap      #15
:
mes: .dc.b '文字列',0
```

というぐあいだ。ソースプログラム中にIOCSコール番号を生のまま埋め込むのがいやであれば、インクルードファイルを1つ作成して、その中で、

```
_B_KEYINP      equ    $00
:
_B_PUTC         equ    $20
_B_PRINT        equ    $21
:
```

のようにずらずらとIOCSコール番号をシンボル定義しておけばよい。幸いなことにXCにはこのインクルードファイルがIOCSCALL.MACの名前であらかじめ用意されている。また、IOCSCALL.MAC内では、

```
IOCS      macro    callno
           moveq.l   #callno,d0
           trap      #15
           endm
```

というマクロが定義されていて、このマクロとシンボルを利用すると上の例は、

```
move.w    #'A',d1
IOCS      BPUTC
```

とか、

```
lea.l     mes,a1
IOCS      BPRINT
```

のようにすっきり書けるようになる。今後、本書中では、IOCSコールを利用するときにはIOCSCALL.MACをインクルードし、このスタイルで記述する。

実際にIOCSコールを使ったプログラムの一例をリスト1に示す。こんな機会でもなければ日のあたることもないようなちっぽけなプログラムLEDOFF.Xだ。実行するとキーボード上のすべてのLEDキーをOFF状態にする。AUTOEXEC.BATに潜り込ませるか、Human

68k Ver.2ならCONFIG.SYSのPROGRAM=～行に記述するかして起動時に1回走らせるのが正しい使い方だ。起動直後に“ち”とか“ヂ”と打ち込んで“コマンドまたはファイル名が違います”攻撃を受けたことがある人ならLEDOFF.Xの有用さに気づいてもらえると思う。

見てのとおり、リスト1はLEDキーの状態を操作するIOCSコールLEDMODをループの中から発行するだけという単純さだ。IOCS LEDMODは2つの引数を探り、d1.1でLEDキーの番号(0～6)、d2.bでONにする(1)かOFFにする(0)かを指定する。リスト1ではループ内でd2を0に固定したままd1を順に変化させてすべてのLEDキーをOFFにしていっている。このことから察しがつくと思うが、IOCSコールでは基本的にd0以外のレジスタの値は保存される(例外はある)。d0だけはIOCSコールの終了ステータスないしは適当な戻り値を返すのに使われる。ちなみに、LEDMODは引数の値が範囲外でLEDの設定できなかった場合は-1を、うまく設定ができたときは0をd0.1に返す。

リスト1
LEDOFF.S

```

1: *      全てのLEDキーをOFFにする
2:
3:      .include      iocscall.mac
4:      .include      doscall.mac
5: *
6: ent:
7:      moveq.l #0, d2      *0...OFF/1...ON
8:      moveq.l #7-l, d1    *LEDキー番号兼ループカウンタ
9: loop:  IOCS      LEDMOD  *設定
10:      dbra      d1, loop  *繰り返す
11:
12:      DOS      _EXIT      *終了
13:
14:      .end      ent

```

『プログラマーズマニュアル』をパラパラと眺めてみると、それ単体でプログラムとして成り立つようなIOCSコールがいくつか見つかると思う。例を挙げるなら、コール番号7F_HのONTIME(本体を立ち上げてからの時間を100分の1秒単位で返す)とか8E_HのBOOTINF(前面の電源スイッチにより起動されたのか、タイマにより起動されたのか、また、どのデバイスから起動されたのかといったブート情報を返す)なんかは、IOCSコールからの戻り値を表示するだけでもそれなりに役にたつ(ことがあるかもしれない)プログラムになる。この類のプログラムはあって困るものでもなし、暇を見つけてつくっておくのもよいだろう。

グラフィックス関連IOCSコール

グラフィック描画もIOCSを利用すればほとんどX-BASICと同じ感覚で手軽に行える。IOCSコールのグラフィックス関連のコールには表1のようなものがある。表1には、グラフィックス関連というよりは画面関連といったほうがよいものも交じっているが、これらについてはあとで触れる。描画に直接関わるのはB1_H～BF_Hで、ここにはGRAPH.FNCの各関数と

似たような機能が並んでいる。実際、X-BASICのGRAPH.FNCはIOCSの該当コールを呼び出すことで実現されている。

コール番号	名 前	機 能
10 _H	CRTMOD	画面モードを切り替える
12 _H	HSVTORGB	HSVからカラーコードへ変換する
90 _H	G_CLR_ON	グラフィック画面を消去し表示状態にする
94 _H	GPALET	グラフィックパレットを設定する
B1 _H	APAGE	グラフィック描画ページを設定する
B2 _H	VPAGE	グラフィック表示ページを設定する
B3 _H	HOME	グラフィック画面の表示位置を設定する
B4 _H	WINDOW	グラフィック画面にクリッピングウィンドウを設定する
B5 _H	WIPE	グラフィック画面内容を消去する
B6 _H	PSET	グラフィック画面に点を打つ
B7 _H	POINT	グラフィック画面の点の色を取得する
B8 _H	LINE	グラフィック画面に線分を描く
B9 _H	BOX	グラフィック画面に矩形を描く
BA _H	FILL	グラフィック画面に矩形を描いて塗り潰す
BB _H	CIRCLE	グラフィック画面に円を描く
BC _H	PAINT	グラフィック画面を指定の1点から塗り潰す
BD _H	SYMBOL	グラフィック画面に文字を描く
BE _H	GETGRM	グラフィック画面からパターンを取り込む
BF _H	PUTGRM	グラフィック画面にパターンを描く

表1
グラフィックス関連
IOCSコール

グラフィック描画には座標や描画色など多くの引数が必要なので、表1のうちIOCSコールB6_H～BF_Hへの引数は適当なメモリ領域に所定の形式で並べ、その先頭アドレスをa1レジスタに入れて引き渡すことになっている。たとえば、線分を描くIOCS LINEの呼び出しは次のようになる。

```
lea.l    _arg,a1
IOCS    LINE
実際の引数はa1の指す先、この例でいうとラベルarg以降に、
arg:    .dc.w    100,200,300,400
        .dc.w    15
        .dc.w    $aaaa
のように用意する。この例は、X-BASICの
        line (100,200,300,400,15,&haaaa)
に相当する3)。
```

■3) X-BASICでは第6引数のラインスタイルは省略可能で、その場合はFFFF_Hとみなされる。しかし、これはX-BASICインタプリタより正確にはGRAPH.FNCが足りない引数を補ってくれているのであって、IOCSレベルでは引数の省略は許されない。

画面の初期化

先に描画そのものの方法を示してしまったが、現実には、グラフィック描画を実行する前には、画面モードを切り替えたり、画面を消去したりといった初期設定が必要だ。画面のモード切り替えにはIOCS CRTMODを利用する。どの画面モードに切り替えるかはd1.wで指定する。d1に与えるモード番号の一覧を表2に示しておく⁴⁾。通常CRTMODはテキスト画面のクリア、テキストパレットの初期化などを行うが、表2の値に100_H (=256)を加えた値を指定すると、初期化を行わずに画面モードだけを切り替えることもできる。また、現在の画面モードが知りたい場合は、d1に-1を与えてCRTMODを呼び出すと画面モード番号がd0.1に返ってくる。

■ 4) 実際には、表2に挙げた以外に未公開ながら17、18が存在するが、24kHzモードを備えていないディスプレイでは表示不可能であるため、ここでは触れない。

モード番号	実画面	表示画面	表示色	CRTモード
0	1024×1024	512×512	16	31kHz
1	1024×1024	512×512	16	15kHz
2	1024×1024	256×256	16	31kHz
3	1024×1024	256×256	16	15kHz
4	512× 512	512×512	16	31kHz
5	512× 512	512×512	16	15kHz
6	512× 512	256×256	16	31kHz
7	512× 512	256×256	16	15kHz
8	512× 512	512×512	256	31kHz
9	512× 512	512×512	256	15kHz
10	512× 512	256×256	256	31kHz
11	512× 512	256×256	256	15kHz
12	512× 512	512×512	65536	31kHz
13	512× 512	512×512	65536	15kHz
14	512× 512	256×256	65536	31kHz
15	512× 512	256×256	65536	15kHz
16	1024×1024	768×512	16	31kHz

表2
IOCSコールCRTMODの
画面モード番号

CRTMODを呼び出した時点では、まだグラフィック画面が使える状態にはない。CRTMODに続けてIOCS G_CLR_ONを呼び出してはじめて、グラフィック画面が利用可能になる。G_CLR_ONはグラフィック画面の内容を全消去し (CRTMODは内容を消さない)、グラフィックパレットを初期化し、表示をONにする。引数はない。結局、IOCSによるグラフィック描画時の初期化手順は、

```
move.w    # 画面モード番号, d1
IOCS      _CRTMOD
IOCS      _G_CLR_ON
```

となる。

IOCS CRTMODほど選択肢は多くないが、画面モードの切り替えはHuman68kのDOSコールconctrlのモード16でも行うことができる。

```
move.w    # 画面モード番号, -(sp)
move.w    # 16, -(sp)
DOS       _CONCTRL
addq.l    # 4, sp
```

conctrlでは画面モードを表3の値で指定する。グラフィックなしのモードを指定した場合以外は、グラフィック画面の消去と表示も同時行われる。また、現在の画面モードがd0.1に返る。モード番号に-1を指定すると、画面モードを変えずにモードの取得だけを行うこともできる。

モード番号	実画面	表示画面	表示色	CRTモード
0	—	768×512	なし	31kHz
1	1024×1024	768×512	16	31kHz
2	—	512×512	なし	31kHz
3	512× 512	512×512	16	31kHz
4	512× 512	512×512	256	31kHz
5	512× 512	512×512	65536	31kHz

表3
DOSコールconctrlモード16の
画面モード番号

さて、2種類の方法があると、どちらを使ったらよいのか、それとも、どちらを使っても同じなのか、気にする読者もいるだろう。いちおうの指針はある。先ほどの、Human68kとIOCSとには階層関係があり、Human68kもIOCSに依存するかたちでつくられているという話を思い出してほしい。このことから、conctrlもIOCS CRTMODとG_CLR_ONを呼び出すことで画面モードを設定していることが想像できる。conctrlで画面モードを切り替えた場合、IOCSもHuman68kも、その変更を「把握」するわけだ。だが、その逆は成り立たない。CRTMODで画面モードを切り替えても、そのことはHuman68kには伝わらないから、画面モードが変わっていないつもりで走り、ときに誤動作する（表示が乱れる程度だが）こともある。こう考えると、画面モードを切り替えるときにはDOSコールconctrlを使ったほうが何かと安全だろう。それが不可能な場合、たとえば、表示画面を256×256にしたいといった場合には、画面モードを切り替える前にconctrlで現在の画面モードを取得し、プログラム終了直前に元に戻すのが礼儀といえる。

礼儀といえば、グラフィック画面はRAMディスクに使われる場合もあるから、本来は、使用可能かどうか調べてから使うのが正しい。この検査には、IOCS TGUSEMDを使用する。d1に0、d2に-1を入れてTGUSEMDを呼び出すと現在のグラフィック画面の使用状況が

d0に返る。

- 0……誰も使っていない
- 1……システムが使用中(RAMディスク)
- 2……アプリケーションが使用中
- 3……アプリケーションが使用したあと壊れたまま

0か3ならグラフィック画面を使用(破壊)してもいいと判断する。逆に使用状況を設定(IOCSに通知)するときには、d2に上の0~3を入れてTGUSEMDを呼び出す。d2=-1で呼び出したときに返される値は、この最後の設定値だ。

TGUSEMDはあくまで、アプリケーションなりシステムプログラムなりがグラフィック画面の使用状況を自ら適切に設定し、かつ、ほかのプログラムがそれを参照してくれる場合にのみ機能することに注意したい。X-BASICやWP.Xのように設定を無視してグラフィック画面を使用する悪い例に騙されず、今後、読者が作成するプログラムでは、グラフィック画面を使用する前にはIOCSに使用許可を求め、使い終わったらその旨IOCSに伝えるのを忘れないようにしよう。

マウスとグラフィックスの組みあわせ

応用例として、グラフィックス関連とマウス関連のIOCSコールを組みあわせて、かんたなお絵かきツール(グラフィックエディタなんて呼べるほどの代ものではない)をつくってみたい。当初はマウスボタンが押されたらその位置に点を打つだけのプログラムにしようと思っていたのだが、それではあまりに単純すぎておもしろみに欠けるので、IOCSコールで実現できる範囲で多少彩りを添えてみた。

- 1) 色は右ボタンを押すことでポップアップするウィンドウで選べるようにする
- 2) 同じウィンドウ上にはペンパターンのメニューも並べ、複数の中からペンのパターンを選べるようにする(パターンは最大16×16ドット)

一見複雑な処理が要求されそうだが、X68000のハードの機能とIOCSのおかげで、どちらもかんたんに実現できる。まず、ウィンドウをポップアップする処理だが、256色2画面の画面モードを使用して、1画面をウィンドウ用、残りを描画用と使い分けることで逃げた。ウィンドウはあらかじめ全部描いておき、X-BASICのvpage()関数、home()関数に相当するIOCSコールVPAGEとHOMEで表示のON/OFF、表示位置の変更を行う。2点目のペンパターンについては、“外字をSYMBOLで表示する”という手を使った。ペンのパターンを外字に登録しておき、PSETで点を打つ代わりにSYMBOLで描くわけだ。どちらもかなり安直だが、彩りとしての役目は果たしてくれる。

さて、お絵かきツールのプログラムに進む前に、マウスの扱い方についてもかんたんにまとめておこう。マウス制御機能はIOCSコールの70_H~7D_H番にある。やはりX-BASICの関数

と似たものばかりだから、X-BASICでマウスを扱ったことがあれば、これらを使いこなすのもわけはない。サンプルとして、画面にメニューを1つ表示し（実際には“終了”という文字列を左上隅に書くだけ）、このメニューのうえで左ボタンが押されたら、それに応じた処理をする（終了する）プログラムをリスト2に示す。

リスト2
MSTEST.S

```

1: #      マウスの使用例
2:
3:      .include      iocscall.mac
4:      .include      doscall.mac
5: #
6: CR      equ      13
7: LF      equ      10
8: #
9: ent:
10:      lea, l      inisp(pc), sp
11:
12:      lea, l      menu(pc), a1      *メニューを描く
13:      IOCS      _B_PRINT      *
14:
15:      IOCS      _MS_INIT      *マウス初期化
16:      IOCS      _MS_CURON      *マウスカーソル表示
17:      moveq, l    #0, d1      *ソフトウェアキーボード
18:      IOCS      _SKEY_MOD      * 表示禁止
19:
20: loop:   IOCS      _MS_GETDT      *ボタンの状態を得る
21:      tst, w      d0      *左ボタンは押されているか?
22:      bpl      loop      * 押されていないかった
23:
24:      *左ボタンが押された
25:      IOCS      _MS_CURGT      *マウスカーソル座標を得る
26:      move, w      d0, d1      *d1, w = Y座標
27:      swap, w      d0      *d0, w = X座標
28:
29:      cmpi, w      #32, d0      *X座標のチェック
30:      bcc      loop      * 範囲外
31:      cmpi, w      #16, d1      *Y座標のチェック
32:      bcc      loop      * 範囲外
33:
34:      *終了メニュー上だった
35:      IOCS      _MS_INIT      *マウス再初期化
36:      moveq, l    #-1, d1      *ソフトウェアキーボード
37:      IOCS      _SKEY_MOD      * 表示許可
38:
39:      DOS      _EXIT      *終了
40: #
41:      .data
42:      .even
43: #
44: menu:   .dc, b      26, ' 終了', CR, LF, 0
45: #
46:      .stack
47:      .even
48: #
49: mystack:
50:      .ds, l      256
51: inisp:   .ent
52:      .end      ent

```

15～18行はマウスを使うときには枕詞のように現れる初期化・使用準備の決まりきった手順だ。最初のMS_INITによりマウスカーソルの表示はOFFになり、カーソルパターンは標準の矢印型に、カーソル座標は (0,0) に、カーソルの移動範囲は表示画面の大きさと一致する

ように初期化される。つづくMS_CURONでマウスカーソルを表示状態にし、SKEY_MODでマウスの右ボタンに割り当てられているソフトウェアキーボードとマウスカーソルの表示/非表示切り替え機能を殺して初期化は完了だ。この3つのIOCSコールの組みあわせは、X-BASICの

```
mouse(0)
mouse(1)
mouse(4)
```

にそれぞれ対応している。

20行からメイン処理が始まる。まず、左ボタンが押されるまで待つ (20～22行)。ボタンの状態を得るにはIOCSコールMS_GETDTを利用する。このIOCSコールはX-BASICのmsstat()に相当し、d0.1の上位ワードにマウスカーソルの相対的な移動量を、下位ワードに左右のボタンの状態を返す。相対的なカーソル移動量のほうはあまり利用されることはないからここでは触れない。

ボタンの状態は第0～7ビットに右ボタン、第8～15ビットに左ボタンのON/OFF状態が返り、ボタンが押されているときは8ビットとも1(FF_H)、押されていなければ8ビットとも0(00_H)になる。右ボタンが押されているかどうかチェックしたければ、

```
IOCS      _MS_GETDT
tst.b     d0
beq       押されていない
           押されている～
```

のようにtst.b後のZビットで処理を振り分ければよいのは明らかだろう。左ボタンの場合は、

```
IOCS      _MS_GETDT
tst.w     d0
bpl       押されていない
           押されている～
```

という手が使える。どうせ上位8ビットとも同じ値をとるのだから、第15ビットだけを調べればすむわけだ。

MS_GETDTで左ボタンの押し下げが検出されたら、すかさずMS_CURGTでマウスカーソルの画面上での現在位置を得る (25行)。MS_CURGTはX-BASICのmspos()関数にあたり、d0.1の上位ワードにマウスカーソルのx座標、下位ワードにy座標を返す。得られた座標がメニュー上にあるかどうかを調べているのが29～32行、やっているのは単純な座標の比較だ。

35行以下が忘れてはならない後始末だ。MS_INITでマウスを再初期化して (マウスカーソルを消し)、SKEY_MODでさっき殺したソフトウェアキーボードを使用可能状態に戻している。

マウスについてはだいたいリスト2の応用で片がつく。これ以外では、ダブルクリックの判定方法ぐらいは知っていたほうがいいのかも。そこでリスト3。リスト2の34行以下

と差し換えて使う。ダブルクリックの判定といってもやるべきことは泥臭いといつていいほど直接的だ。ボタンが押されたことがわかったら、

- 1) 一定時間以内に離されるかどうか
- 2) 一定時間以内にまた押されるかどうか

というチェックを続けて行い、両方に通ったらダブルクリックされたと判断する。場合によっては、さらに、

- 3) 一定時間以内にまた離されるかどうか

を確認したほうが堅実かもしれない。

ボタンが押されたり離されたりするのを待つ処理には、IOCSコールのMS_OFTM, MS_ONTMを利用する。d1.wで左右のボタンのどちらか(0なら左, -1なら右), d2.Wで待ち時間を指定すると(とくに0のときは無限と見なされる), 指定時間以内にボタンが離されたり(MS_OFTM) 押されたり(MS_ONTM) したら, それまでの経過時間をd0.wが返る。ただし, ドラッグされた場合(ボタンの状態が変化しないうちにマウスカーソルが動いた場合)にはd0.w=0で即戻ってくる。また, 待ち時間をこえた場合はFFFF_Hが返る。待ち時間の単位はなにやらい加減らしく(ループ回数で計時しているのか?), だいたい40が0.1秒前後に相当する。リスト3では待ち時間を0.2秒ほど度にするために80を指定してある。ダブルクリックの速度は人それぞれ好みがあるものだから, 実用プログラムをつくらうというときには, この値を変更できるようにするのは当然のことだろう(ここではやらないが)。

.....
リスト3
MSTEST2.S
[リスト2に対する
追加・変更部]

```

34:                                     *終了メニュー上だった
35:      moveq.l #0, d1                *左ボタン
36:      moveq.l #80, d2              *待ち時間(約0.2秒)
37:      IOCS      MS_OFFTM           *離されるまで待つ
38:      tst.w     d0                 *0以下なら
39:      ble      loop               * はじく
40:
41:      IOCS      MS_ONTM           *押されるまで待つ
42:      tst.w     d0                 *0以下なら
43:      ble      loop               * はじく
44:
45:                                     *ダブルクリックされた
46:      IOCS      MS_INIT           *マウス再初期化
47:      moveq.l #-1, d1              *ソフトウェアキーボード
48:      IOCS      _SKEY_MOD         * 表示許可
49:
50:      DOS       _EXIT             *終了
51: #
52:      .data
53:      .even
54: #
55: menu:      .dc.b    26, '終了', CR, LF, 0
56: #
57:      .stack
58:      .even
59: #
60: mystack:
61:      .ds.l     256
62: inisp:
63:      .end      ent

```

簡易お絵かきツールの実際

では、手抜きいっぱいのお絵かきプログラム、リスト4のSTAMP.Sを見てもらおう。プログラムはかなり長くなってしまったが、比較的読みやすく書けていると思うので、これまでの話のまとめのつもりで読んでみてもらいたい。各ルーチンごとにポイントとなる部分を拾って軽く解説しておく。

リスト4
STAMP.S

```

1: *      256色モード用簡易簡易お絵書きツール
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: *
6: CFKEYMOD      equ      14      *CONCTRLモード番号
7: CSCREEN      equ      16      *
8: CCURON      equ      17      *
9: CCUROFF      equ      18      *
10: *
11: HIDEKEY      equ      3      *ファンクションキー行非表示
12: DOS_GM3      equ      4      *画面モード512x512, 256色
13: *
14: DISABLEKEY    equ      0      *ソフトウェアキーボード禁止
15: ENABLEKEY     equ     -1      *ソフトウェアキーボード許可
16: *
17: GVRAMUSEMD    equ      0      *TGUSEMDの検査/設定対象はG-VRAM
18: CHECKUSEMD    equ     -1      *検査
19: USING         equ      2      *使用中
20: BROKEN        equ      3      *使用後
21: *
22: WINH          equ     272     *メニューウィンドウ幅
23: WINV          equ     104     *メニューウィンドウ高さ
24: *
25: USERPAGE      equ      1      *描画を行う画面
26: BIT_USERPAGE  equ     %0010   *
27: MENUPAGE      equ      0      *メニューを表示する画面
28: BIT_MENUPAGE  equ     %0001   *
29: *
30:              *メニュー表示
31: SHOWMENU      equ     BIT_USERPAGE|BIT_MENUPAGE
32:              *メニュー非表示
33: HIDEMENU      equ     BIT_USERPAGE
34: *
35: GAIJITOP      equ     $eb9f   *全角外字の先頭文字コード
36: FONT16       equ     $0008   *8x16, 16x16
37: *
38: PATMAX        equ      8      *ペンパターンの最大数
39:
40: *
41: *      グラフィック描画IOCSデータ受け渡し領域の構造
42: *
43:      .offset 0
44: *
45: X0:      .ds.w  1      *POINT *FILL *BOX
46: Y0:      .ds.w  1      *      *      *
47: RETCOL:   *      *      *
48: X1:      .ds.w  1      *      *      *
49: POINTBUFSIZ: *      *      *
50: Y1:      .ds.w  1      *      *      *
51: COL:      .ds.w  1      *      *      *
52: FILLBUFSIZ: *      *      *
53: LS:      .ds.w  1      *      *      *
54: BOXBUFSIZ: *      *      *
55:
56: *

```



```

57: *      フォント読み込み領域の構造
58: *
59: *      .offset 0
60: *
61: XLEN:  .ds.w  1
62: YLEN:  .ds.w  1
63: FPAT:  .ds.w  16      *16x16
64: FNTBUFSIZ:
65: *
66: *      .text
67: *      .even
68: *
69: ent:
70:     lea.l   inisp(pc), sp      *spを初期化する
71:
72:     moveq.l #GVRAMUSEMD, d1    *グラフィック画面は
73:     moveq.l #CHECKUSEMD, d2    * 使用可能か?
74:     lOCS    _TGUSEMD
75:     cmpi.b  #BROKEN, d0        *誰かが使えばなし?
76:     beq     gramok             * そうなら使える
77:     tst.b   d0                *誰も使っていない?
78:     bne     quit              * 誰かが使っていた
79:
80: gramok: moveq.l #USING, d2      *グラフィック画面の使用を宣言する
81:     lOCS    _TGUSEMD
82:
83:     bsr     init              *画面などの初期化
84:
85:     pea.l   break(pc)         *中断時の戻りアドレスを設定
86:     move.w  #_CTRLVC, -(sp)    *
87:     DOS     _INTVCS            *
88:     move.w  #_ERRJVC, (sp)     *
89:     DOS     _INTVCS            *
90:     addq.l  #6, sp            *
91:
92:     bsr     setupmenu         *メニューウィンドウの初期化
93:     bsr     main              *メイン処理
94:
95: break: bsr     windup          *後始末
96:
97:     move.w  #-1, -(sp)         *キーバッファクリア
98:     DOS     _KFLUSH            *
99:     addq.l  #2, sp            *
100:
101: quit:  DOS     _EXIT           *終了
102:
103: *
104: *      メイン処理
105: *
106: main:
107:     lOCS    _MS_GETDT         *ボタンの状態を取得
108:     tst.b   d0                *右ボタンが押されている?
109:     bne     rdown
110:     tst.w   d0                *左ボタンが押されている?
111:     bpl     main
112: *
113: ldown: *左ボタンが押された
114:     lOCS    _MS_CURGT         *マウスカーソル位置を取得
115:     move.w  d0, d1            *d1.w = y
116:     clr.w   d0
117:     swap.w  d0                *d0.w = x
118:     tst.b   menuflag          *ウィンドウは表示中か?
119:     beq     pset
120:
121:     move.w  winx(pc), d2      *d2.w = ウィンドウ表示位置x
122:     move.w  winy(pc), d3      *d3.w = ウィンドウ表示位置y
123:
124:     cmp.w   d2, d0            *ウィンドウ上かどうかチェック
125:     bcs     pset
126:     cmp.w   d3, d1
127:     bcs     pset
128:     addi.w  #WINH, d2
129:     addi.w  #WINV, d3

```

```

130:      cmp.w  d2, d0      *
131:      bcc   pset        *
132:      cmp.w  d3, d1      *
133:      bcc   pset        *
134:
135:
136:      sub.w  winx(pc), d0  *ウィンドウ内でクリックされた
137:      sub.w  winy(pc), d1  *d0.w = ローカルx座標
138:      move.w d0, pntbuf+X0 *d1.w = ローカルy座標
139:      move.w d1, pntbuf+Y0 *x, yそれぞれを待避しておく
140:
141:      subq.w #8, d0
142:      bcs   drag          *ウィンドウの左余白
143:      subq.w #8, d1
144:      bcs   drag          *ウィンドウの上余白
145:      cmp.w  #256, d0
146:      bcc   drag          *ウィンドウの右余白
147:      cmp.w  #16, d1
148:      bcc   ldown1
149:
150:      cmp.w  #224, d0      *上段のメニュー内
151:      bcs   ldown0
152: done:  rts               *終了ボックス内
153:                                     *メインループを抜ける
154:
155: ldown0: subi.w #32, d0
156:      bcs   drag          *ペンメニューより左
157:      divu.w #24, d0
158:      swap.w d0
159:      cmpi.w #16, d0
160:      bcc   drag          *ペンバターンの隙間の余白
161:      swap.w d0
162: *
163: selpen:
164:      move.w d0, d2        *ペンメニュー内
165:      addi.w #GAIJITOP, d0 *d2.w = ペン番号
166:      move.w d0, curpat    *新バターンを設定
167:
168:      moveq.l #MENUPAGE, d1
169:      lOCS   _APAGE        *メニュー用ページに
170:                                     * 切り換える
171:
172:      lea.l  curpen(pc), a1 *以前の枠を消す
173:      move.w #255, COL(a1)
174:      lOCS   _BOX
175:
176:      mulu.w #24, d2        *新たに枠を描く
177:      addi.w #38, d2
178:      move.w d2, X0(a1)
179:      addi.w #19, d2
180:      move.w d2, X1(a1)
181:      move.w #1, COL(a1)
182:      lOCS   _BOX
183:
184:      bsr    lwait         *左ボタンが離されるのを待つ
185:
186:      bra    ldown2        *描画用ページに戻す
187:
188: ldown1: subi.w #24, d1
189:      bcs   drag          *ペンメニューと
190:                                     * 色メニューの隙間の余白
191:      cmpi.w #64, d1
192:      bcc   drag          *ウィンドウの下余白
193: *
194: selcol:
195:      moveq.l #MENUPAGE, d1 *色メニュー内
196:      lOCS   _APAGE        *メニュー用ページに
197:                                     * 切り換える
198:
199:      lea.l  pntbuf(pc), a1 *マウスカーソル位置から
200:      lOCS   _POINT        * 色を拾う
201:      move.w RETCOL(a1), d0
202:      move.w d0, curcol    *カレントカラーにセット

```

```

202:      lea, l    coldat(pc), a1      *カレントカラーで
203:      move, w   d0, COL(a1)         * メニュー左上の枠を
204:      IOCS      _FILL                * 塗り潰す
205:
206:      IOCS      _MS_GETDT            *左ボタンが押されているあいだ
207:      tst, w     d0                  * 繰り返す
208:      bpl       ldown2              *
209:      *
210:      IOCS      _MS_CURGT            *
211:      sub, w     winy(pc), d0        *
212:      move, w    d0, pntbuf+Y0       *
213:      swap, w    d0                  *
214:      sub, w     winx(pc), d0        *
215:      move, w    d0, pntbuf+X0       *
216:      *
217:      bra       scloop              *
218:
219: ldown2: moveq, l #USERPAGE, d1      *描画用ページに戻す
220:      IOCS      _APAGE              *
221:
222:      bra       main                *メインループへ
223: *
224: drag:
225:      bsr       lwait                *ウィンドウの外枠でクリックされた
226:      bra       menuon              *左ボタンが離されるのを待つ
227:      *                               *ウィンドウを描き直す
228: pset:
229:      lea, l     setdat(pc), a1      *ウィンドウ外でクリックされた
230:      subq, w     #8, d0              *マウスカーソルの位置に
231:      move, w     d0, X0(a1)          * パターンを描く
232:      subq, w     #8, d1              *
233:      move, w     d1, Y0(a1)          *
234:      IOCS      _SYMBOL              *
235:
236:      bra       main                *メインループへ
237: *
238: rdown:
239:      move, l     #(WINH/2)<<16, ofst *右ボタンが押された
240:      tst, b      menuflag           *表示位置オフセット
241:      beq        menuon              *メニューは表示中か?
242: *
243: menuoff:
244:      sf, b       menuflag           *メニューウィンドウを消す
245:      moveq, l    #HIDEMENU, d1      *フラグを寝かせる
246:      IOCS      _VPAGE              *メニューページ非表示
247:      *
248:      bsr       rwait                *右ボタンが離されるのを待つ
249:
250:      bra       main                *メインループへ
251: *
252: menuon:
253:      st, b       menuflag           *メニューウィンドウを出す
254:      IOCS      _MS_CURGT            *フラグを立てる
255:      move, w     d0, d1              *マウスカーソル位置を取得
256:      swap, w     d0                  *d1, w = y
257:      *                               *d0, w = x
258:      sub, w     ofst+X0(pc), d0      *ウィンドウが
259:      bcc        mon0                 * 画面からはみ出さないよう
260:      clr, w     d0                  * 調整する
261: mon0:  sub, w     ofst+Y0(pc), d1      *
262:      bcc        mon1                 *
263:      clr, w     d1                  *
264: mon1:  cmp, w     #512-WINH, d0        *
265:      bcs        mon2                 *
266:      move, w     #512-WINH, d0        *
267: mon2:  cmp, w     #512-WINV, d1        *
268:      bcs        mon3                 *
269:      move, w     #512-WINV, d1        *
270:
271: mon3:  move, w     d0, winx           *表示位置を格納
272:      move, w     d1, winy           *
273:

```

```

274:      moveq.l #0, d2          #ウィンドウを目的の位置へ
275:      sub.w   d0, d2          # 移動する
276:      andi.w  #511, d2        #
277:      moveq.l #0, d3          #
278:      sub.w   d1, d3          #
279:      andi.w  #511, d3        #
280:      moveq.l #BIT_MENUPAGE, d1 #
281:      lOCS    _HOME           #
282:
283:      moveq.l #SHOWMENU, d1    #メニューページ表示
284:      lOCS    _VPAGE          #
285:
286:      bsr     rwait            #右ボタンが離されるのを待つ
287:
288:      bra     main             #メインループへ
289:
290: #
291: rwait:      #右ボタンが離されるのを待つ
292:      lOCS    _MS_GETDT
293:      tst.b   d0
294:      bne     rwait
295:      rts
296: #
297: lwait:      #左ボタンが離されるのを待つ
298:      lOCS    _MS_GETDT
299:      tst.w   d0
300:      bmi     lwait
301:      rts
302:
303: #
304: #      初期化
305: #
306: init:
307:      link    a6, #0
308:
309:      #画面
310:      move.w  #DOS_GM3, -(sp)   #画面を512x512, 256色に
311:      move.w  #CSCREEN, -(sp)   # 初期化
312:      DOS     _CONCTRL          #
313:      move.w  d0, scrnmsav      #現在の画面モードを待避
314:
315:      move.w  #HIDEFKEY, -(sp)  #ファンクションキー行を
316:      move.w  #CFKEYMOD, -(sp) # 非表示に設定
317:      DOS     _CONCTRL          #
318:      move.w  d0, fkeymsav      #現在のファンクションキー行
319:                                  # モードを待避
320:
321:      move.w  #CCUROFF, -(sp)   #カーソル非表示モード
322:      DOS     _CONCTRL          #
323:
324:      #外字
325:      bsr     savfont           #待避
326:      bsr     deffont          #定義
327:
328:      #マウス
329:      lOCS    _MS_INIT          #マウス初期化
330:      lOCS    _MS_CURON         #マウスカーソル表示
331:      moveq.l #DISABLESKEY, d1  #ソフトウェアキーボードの
332:      lOCS    _SKEY_MOD         # 使用を禁止
333:
334:      unlk    a6
335:      rts
336: #
337: #      メニューの初期化
338: #      (あらかじめ全部描いておく)
339: #
340: setupmenu:
341:      moveq.l #MENUPAGE, d1      #メニュー用ページに
342:      lOCS    _APAGE            # 切り換える
343:
344:      moveq.l #HIDEMENU, d1      #メニュー用ページ非表示
345:      lOCS    _VPAGE            #

```

```

346:
347:     lea, l    fildat(pc), a1      #ウィンドウ枠を塗り潰す
348:     IOCS     _FILL                #
349:
350:     lea, l    boxes(pc), a1      #BOXを必要なだけ描く
351: boxlp:  tst, w    (a1)            #
352:         bmi    boxed            #
353:         IOCS    _BOX             #
354:         lea, l    BOXBUFSIZ(a1), a1  #
355:         bra     boxlp           #
356:
357: boxed:  lea, l    mendat(pc), a1  #ペンパターンメニューを
358:         IOCS     _SYMBOL        # 描く
359:
360:         bsr     makecoltbl       #カラーテーブル
361:
362:         moveq, l #USERPAGE, d1    #描画用ページに切り換える
363:         IOCS     _APAGE          #
364:
365:         sf, b    menuflag        #フラグを寝かせる
366:
367:         rts
368:
369: #
370: #       256色の色テーブルを描く
371: #
372: makecoltbl:
373:     link     a6, #-FILLBUFSIZ
374:
375:     lea, l    -FILLBUFSIZ(a6), a1  #a1 = FILL用引数領域
376:
377:     moveq, l #0, d1                #d1 = 色
378:
379:     move, w    #32, Y0(a1)         #最初は(8, 32) - (8+7, 32+7) から
380:     move, w    #32+7, Y1(a1)      #
381:
382:     moveq, l #8-1, d6              #縦に8個
383: clp0:  move, w    #8, X0(a1)
384:         move, w    #8+7, X1(a1)
385:
386:     moveq, l #32-1, d7             #横に32個
387: clp1:  move, w    d1, COL(a1)      #四角を描く
388:         IOCS     _FILL            #
389:
390:         addq, w    #8, X0(a1)      #右に8ピクセル移動
391:         addq, w    #8, X1(a1)      #
392:         addq, w    #1, d1          #次の色
393:         dbra      d7, clp1        #横1列分繰り返す
394:
395:         addq, w    #8, Y0(a1)      #下に8ピクセル移動
396:         addq, w    #8, Y1(a1)      #
397:         dbra      d6, clp0        #繰り返す
398:
399:         unlk     a6
400:         rts
401:
402: #
403: #       後始末
404: #
405: windup:
406:     link     a6, #0
407:
408:     move, w    scrnsav(pc), -(sp)  #画面モードを戻す
409:     move, w    #CSCREEN, -(sp)    #
410:     DOS       _CONCTRL           #
411:
412:     move, w    fkeysav(pc), -(sp)  #ファンクションキー行の
413:     move, w    #CFKEYMOD, -(sp)   # モードを戻す
414:     DOS       _CONCTRL           #
415:
416:     move, w    #CCURON, -(sp)      #カーソル表示モード
417:     DOS       _CONCTRL           #

```

```

418:
419:      bsr      rstfont      *外字フォント復帰
420:
421:      IOCS      _MS_INIT      *マウス初期化
422:      moveq.l   #ENABLESKEY, d1 *ソフトウェアキーボードの
423:      IOCS      _SKEY_MOD      * 使用を許可
424:
425:      moveq.l   #GVRAMUSEMD, d1 *G-VRAM内容は破壊した
426:      moveq.l   #BROKEN, d2      *
427:      IOCS      _TGUSEMD      *
428:
429:      unlk     a6
430:      rts
431:
432: *
433: *      外字の先頭8文字のフォントパターンを待避する
434: *
435: savfont:
436:      lea.l     fontbuf(pc), a1
437:      move.l    #FONT16<<161GAIJITOP, d1
438:      moveq.l   #PATMAX-1, d2
439:      savlp:   IOCS      _FNTGET
440:      addq.w    #1, d1
441:      lea.l     FNTBUFSIZ(a1), a1
442:      dbra     d2, savlp
443:      rts
444:
445: *
446: *      外字の先頭8文字にフォントパターンを設定する
447: *
448: deffont:
449:      lea.l     fontdat+FPAT(pc), a1
450:      defnt0:   move.l    #FONT16<<161GAIJITOP, d1
451:      moveq.l   #PATMAX-1, d2
452:      deflp:   IOCS      _DEFCHR
453:      addq.w    #1, d1
454:      lea.l     FNTBUFSIZ(a1), a1
455:      dbra     d2, deflp
456:      rts
457:
458: *
459: *      savfontで待避したフォントパターンを復帰する
460: *
461: rstfont:
462:      lea.l     fontbuf+FPAT(pc), a1
463:      bra      defnt0
464:
465: *
466: *      データ&ワーク
467: *
468:      .data
469:      .even
470: *
471: fontdat:
472:      .dc.w     16, 16      *eb9f
473:      .dc.w     %0000000000000000      *0
474:      .dc.w     %0000000000000000
475:      .dc.w     %0000000000000000
476:      .dc.w     %0011111110111111
477:      .dc.w     %0010000001010001
478:      .dc.w     %0001000001010010
479:      .dc.w     %000100000101100
480:      .dc.w     %0000100000101000
481:      .dc.w     %000010000010000
482:      .dc.w     %0000010000010000
483:      .dc.w     %000001000001000
484:      .dc.w     %0000101000001000
485:      .dc.w     %000110100000100
486:      .dc.w     %0010010100000100
487:      .dc.w     %0100010100000010
488:      .dc.w     %0111110111111110
489:

```

```

490:      .dc.w 16,16      *eba0
491:      .dc.w $0000,$0000,$0000,$0000,$0000,$0000,$0000,$0000
492:      .dc.w $0080,$0000,$0000,$0000,$0000,$0000,$0000,$0000
493:
494:      .dc.w 16,16      *eba1
495:      .dc.w $0000,$0000,$0000,$0000,$0000,$0000,$0000,$0180
496:      .dc.w $0180,$0000,$0000,$0000,$0000,$0000,$0000,$0000
497:
498:      .dc.w 16,16      *eba2
499:      .dc.w $0000,$0000,$0000,$0000,$0000,$0000,$0000,$01c0
500:      .dc.w $01c0,$01c0,$0000,$0000,$0000,$0000,$0000,$0000
501:
502:      .dc.w 16,16      *eba3
503:      .dc.w $0000,$0000,$0000,$0000,$0000,$0410,$0220,$0140
504:      .dc.w $0080,$0140,$0220,$0410,$0000,$0000,$0000,$0000
505:
506:      .dc.w 16,16      *eba4
507:      .dc.w $0000,$0000,$0000,$0000,$0000,$0c00,$0600,$0300
508:      .dc.w $0180,$00c0,$0060,$0030,$0000,$0000,$0000,$0000
509:
510:      .dc.w 16,16      *eba5
511:      .dc.w $0000,$03e0,$07f0,$0ff8,$1ffc,$3ffe,$7fff,$7fff
512:      .dc.w $7fff,$7fff,$7fff,$3ffe,$1ffc,$0ff8,$07f0,$03e0
513:
514:      .dc.w 16,16      *eba6
515:      .dc.w $0000,$7fff,$7fff,$7fff,$7fff,$7fff,$7fff,$7fff
516:      .dc.w $7fff,$7fff,$7fff,$7fff,$7fff,$7fff,$7fff,$7fff
517:
518: fildat: .dc.w 0,0      *ウィンドウ枠塗り潰し用
519:      .dc.w WINH-1,WINV-1
520:      .dc.w 255
521:
522: boxes:      *ウィンドウ内枠描画用
523: box1:      .dc.w 2,2,WINH-2-1,WINV-2-1,1,$fff
524: box2:      .dc.w 6,6,33,25,1,$fff
525: curpen:
526: box3:      .dc.w 38,6,57,25,1,$fff      *カレントペンを示す枠
527: box4:      .dc.w 230,6,265,25,1,$fff
528: box5:      .dc.w 6,30,265,97,1,$fff
529:      .dc.w -1
530:
531: mendat: .dc.w 40,8      *メニュー表示用
532:      .dc.l patstr
533:      .dc.b 1,1
534:      .dc.w 1
535:      .dc.b 1,0
536:
537: coldat: .dc.w 8,8,31,23,255      *カレントカラー表示用
538:
539: setdat: .dc.w 0,0      *点描画用
540:      .dc.l curpat
541:      .dc.b 1,1
542: curcol: .dc.w 255
543:      .dc.b 1,0
544:
545: curpat: .dc.b $eb,$9f,0      *カレントペンパターン
546:
547: patstr: .dc.b $eb,$9f,$20,$eb,$a0,$20      *メニュー文字列
548:      .dc.b $eb,$a1,$20,$eb,$a2,$20
549:      .dc.b $eb,$a3,$20,$eb,$a4,$20
550:      .dc.b $eb,$a5,$20,$eb,$a6,$20
551:      .dc.b '終了',0
552: *
553:      .bss
554:      .even
555: *
556: fontbuf: .ds.b FNTBUFSIZ*8      *フォント待避領域
557: ofst:
558: pntbuf: .ds.b POINTBUFSIZ      *IOCS POINT用
559: winx: .ds.w 1      *メニューウィンドウ表示位置
560: winy: .ds.w 1      *
561: scrnsav: .ds.w 1      *画面モード待避用

```

```

562: fkeymsav:      .ds.w 1          *ファンクションキー行モード待避用
563: menuflag:     .ds.b 1          *メニュー表示/非表示フラグ
564: *
565:               .stack
566:               .even
567: *
568:               .ds.l 1024         *スタック領域
569: inisp:
570:               .end   ent

```

エントリ～終了 (69行～)

まっさきにIOCS TGUSEMDによりグラフィック画面が使用可能かどうかを調べている (72～78行)。使用可能であれば、その使用をIOCSに通知する (80～81行)。後でプログラム終了時にグラフィック画面を破壊した旨、再通知することになるから、ここで「使用中」に設定することにそれほど深い意味はない。ただ、バックグラウンドタスクが勝手にグラフィック画面を使うのを防止する効果はあるだろう。

85～90行では中断時の戻りアドレスを95行のラベルbreakの位置に設定している。これは、Interruptスイッチなどによってプログラムの実行が中断された場合に、TGUSEMDで設定したグラフィック画面の使用状況の設定や画面モードを元に戻さずに親プロセスに戻るのを避けるためだ。break以降では諸々の後始末をするサブルーチンを呼び出してから、キーバッファをクリアし、exitで実行終了する。マウスしか使わないプログラムでキーバッファを気にしているのが変に見えるかもしれないが、“マウスしか使わないからこそ”この処理が必要なのだ。これを怠ると、プログラム走行中に誤って押されたキーがプログラム終了後にまとめて吐き出されることになる。

初期化ルーチン (306行～)

309～317行でDOSコールconctrlによって画面モードを横512×縦512ピクセル、256色モードに切り替えたうえで、邪魔なファンクションキー行とカーソルを消している。画面モードとファンクションキー行についてはあとで元に戻せるように (405行以下の後始末ルーチン参照) 現在の状況をワークエリアにしまっておく。あと、このサブルーチンでは頭でlinkし、リターンする直前でunlkすることによってDOSコール呼び出し時のスタック補正を省略するという姑息なテクニックが使われている。場合によってはプログラムが読みにくくなるのであまり褒められたことではないが、こういうこともできるというのは知っておいてもらいたい。

324～325行は下位のサブルーチンを呼び出して、IOCS DEFCHRによって、ペンパターンとして利用する外字の定義を行っている。ここでも、後で元に戻せるように現在の外字の定義を取得・待避しておくのを忘れない (IOCS FNTGETを利用)。定義する外字のフォントパターンは471行以下に用意しており、16ワードが1文字分のデータにあたる。頭に縦横のピクセル数がつけてあるのはほかとの兼ねあいで、実際には使っていない。フォントパターンは472～

488行の最初の1個だけは見やすく2進数で表記してみた(2個目以降はスペースの都合で詰めて16進数で表記してある)。これを見ればフォントパターンの形式・つくり方は一目瞭然だろう。

328~331行はマウス周りの初期化, リスト2同様の決まり切った手順だ。

メニューウィンドウの初期化 (340行~)

前述したように, メニューはあらかじめ全部描いておく。描画に必要なデータはデータセクションに用意しておき, これを次々にIOCSコールに渡している。IOCS BOX, FILL, SYMBOLの使用例だ。

メイン処理 (106行~)

多少冗長なつくりになっているが, マウスのボタンの状態をチェックし, ボタンが押されていたらその位置におうじてそれなりの処理を行うというパターンの組みあわせであり, リスト2と基本的には大差ない。左ボタンが押された場合は, まずメニューウィンドウ上かどうかを調べ, ウィンドウ外(もしくはウィンドウが非表示状態)であれば228行に飛んでIOCS SYMBOLで現在設定されているペンパターン(に対応する外字1文字)を描く。ウィンドウ上だった場合は, マウスカーソル座標から,

- 1) ペン選択メニュー上
- 2) 色選択メニュー上
- 3) 終了メニュー上
- 4) いずれでもないウィンドウの外枠

を識別し, 対応する処理を行う。1), 2)の場合はさらにメニュー上のどの部分かの判定が加わることになる。また, ウィンドウの外枠で左ボタンが押された場合はウィンドウをドラッグするようにしてみた。本来ならマウスの動きに連動してリアルタイムでウィンドウの位置を変更することもできたのだが, もっと単純に, ボタンが離された位置へいきなりウィンドウを移動するようになっている。ここは読者に手を入れてもらいたい部分の1つだ。

左ボタンの処理に比べれば, 238行以下の右ボタンによるメニューウィンドウのON/OFF切り替え処理はシンプルだ。現在メニューが表示中かどうかを覚えておくワークmenuflagを調べて(240行), もしメニューがすでに表示中であれば243行以下でIOCS VPAGEによりメニューが描かれているページを非表示にする。メニューが表示されていなければ252行以下で現在のマウスカーソルの位置にメニューを表示する。VPAGEの引数はd1.bにX-BASICのvpage()関数と同じ形式, つまり, 表示するページに対応するビットを1にした値で与える。

なお, 253行でmenuflagをセットするのに使っているst.bは, 任意の1バイトをFF_Hにする命令だ(オペランドのサイズはバイト固定)。stの一般形はs??(sはsetの略)であり, ??

の部分には条件分岐命令同様の条件が入る。s??は命令実行の時点でこの条件が成り立っていればオペランドをFF_Hにし、条件が成り立っていなければ00_Hにする命令で、stはこの条件が“t (always True: 常に真)”になったかたちだ。条件がつねに成り立つわけだから、オペランドを00_Hにすることはありえない。逆に、244行で使っているsfという命令は条件が“f (always False: 常に偽)”であり、任意の1バイトを00_Hにする。動作はclr.bと同じだが、ここではstと対にする意味でsfのほうを使ってみた。

後始末 (405行～)

ここでは、さきに初期化時点で取得しておいた情報にしたがって、画面モード、外字フォント、ソフトウェアキーボードの状態を復帰し、グラフィック画面を使い終わったことをIOCSに伝えている。このプログラムでは描いた絵のセーブができなくせに、終了時に画面モードを元に戻してしまっているのは少々間が抜けているが、グラフィックスを利用したプログラムの一般的な後始末の型を示しているものと理解してほしい。

オリジナル描画ルーチン作成を目指して

以上のように、IOCSコールを利用すれば、マシン語レベルでもかんたんにグラフィックスを使ったプログラムが作成できる。あまりにあっけなさすぎて、マシン語プログラミングという実感が沸かないような気さえするほどだ。そこで、強引ながら次章以降では、グラフィック描画ルーチンを0から作成するという方向に話を持っていく。最終的に何か大きなプログラムをつくらうというのではなく、描画ルーチンをつくる過程を楽しんでみたい。もちろん、読者はその後で、応用プログラムをつくってまた楽しめるのだ。

CHAPTER

2

直接グラフィック画面を操作する

直接グラフィック画面を 操作する

ASSEMBLER



この章以降では、IOCSコールに頼らない、自家製グラフィック描画ルーチンを作成していく。ここからが本題だ。最初の題材はボックスフィル、矩形領域の塗り潰しである。

X68000のグラフィック画面

グラフィック画面の実体は“グラフィックVRAM (Video RAM)”とか“G-RAM”と呼ばれるメモリであり、グラフィック描画とは、ある規則にしたがってこのG-RAMにデータを書き込むことにほかならない。

X68000のG-RAMは（正しい用語かどうかは知らないが俗にいう）垂直型で、メモリの1ワード（=2バイト）と画面上の1ピクセルが1対1に対応している。点を打ちたいければ、座標から算出したG-RAM上のアドレスに1ワードのパレットコードを書き込めばよい。これに対して、X1やPC-9801など多くのマシンで採用されている（いわゆる）水平型のG-RAMでは単色の画面（プレーン：plane）を複数重ねあわせてカラー表示を行う構成なので、1つ点を打つものにもプレーンの枚数分の書き込み操作を行わなければならない。しかも横方向の8ピクセルが1バイトにまとまっているためにビット単位の操作も必要になる¹⁾。扱いやすさという点では断然垂直型に軍配が上がるし、速度の点でも多くの場合、垂直型のほうが有利といえる。

■ 1) 逆にいえば、単色にかぎっては1バイトの書き込みで横8ピクセル分が描けるのが水平型の利点だ。

図1～3にX68000のG-RAM構造を示す。図1は画面モード別のメモリマップ、図2は実画面の大きさ別の画面内のワードの並び方、図3は色モード別のワード内部の構成を表している。特徴的なのは、画面モードにおうじてG-RAMのメモリ空間上の大きさが変化する点だろう。最大では2Mバイトもの空間を占める。X68000のG-RAMは512Kバイトしかないからつじつまが合わないように見えるが、図3を見てもらえればわかるように、256色や16色モードのときは各ワードの上位ビットが空いている。結局、256色モードのときは、

8ビット×512ピクセル×512ピクセル×2画面

=512Kバイト
16色モードのときは,
4ビット×512ピクセル×512ピクセル×4画面
=512Kバイト
もしくは,
4ビット×1024ピクセル×1024ピクセル×1画面
=512Kバイト
だから、メモリの総容量が変わるわけではない。

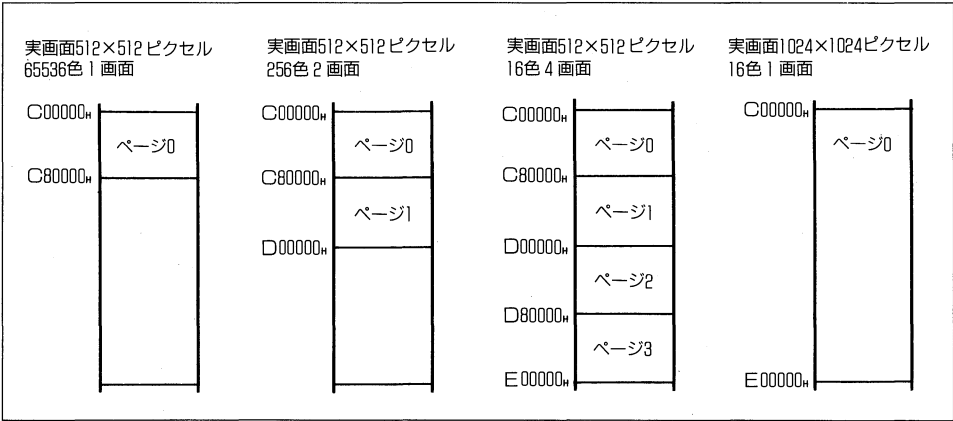


図1
G-RAM
メモリマップ

実画面512×512ピクセル					
	+0	+2	+4	+1020	+1022
+0	(0,0)	(1,0)	(2,0)	(510,0)	(511,0)
+1024	(0,1)	(1,1)	(2,1)	(510,1)	(511,1)
+2048	(0,2)	(1,2)	(2,2)	(510,2)	(511,2)

実画面1024×1024ピクセル					
	+0	+2	+4	+2044	+2046
+0	(0,0)	(1,0)	(2,0)	(1022,0)	(1023,0)
+2048	(0,1)	(1,1)	(2,1)	(1022,1)	(1023,1)
+4096	(0,2)	(1,2)	(2,2)	(1022,2)	(1023,2)

図2
座標と
G-RAM上のアドレス

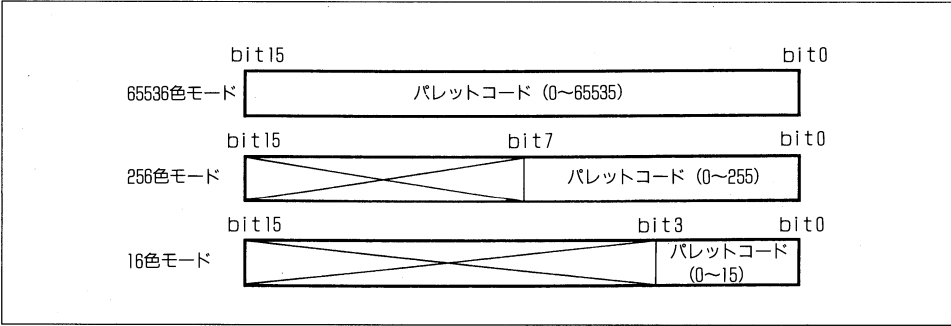


図3
G-RAM上1ワードの
ビット構成

では、図を参考に、xy座標からG-RAM上のアドレスを計算する手順・プログラムを考えてみよう。図2 (p.33) を見ればG-RAM上のメモリは画面の左から右方向にアドレスが増加し、右端で折り返して下のラインの左端につながるといふかたちをしていることがわかる。隣りあったピクセルのアドレスの差は、左右が2バイト、上下が2048 (実画面1024×1024ピクセルモード時) または1024 (512×512ピクセルモード時) バイトという定数だ。これから、ページ0の座標 (x,y) に対応するG-RAM上のアドレスは、実画面1024×1024ピクセルモードのとき、

$$C00000_H + x \times 2 + y \times 2048$$

実画面512×512ピクセルモードのとき、

$$C00000_H + x \times 2 + y \times 1024$$

と導ける。ページ1以降の場合は先頭アドレスが80000_Hバイトずつずれるだけだから、その分を足してやればよい。

リスト1が実際にアドレスを計算するプログラムの1例だ。座標からのアドレス計算はこれからもたびたび必要になるので、独立したサブルーチンとし、関連サブルーチンといっしょにまとめている。

.....
リスト1
GRAMADR.S

```
1: #      G-RAMアドレス関連モジュール
2:
3:      .include      iocscall.mac
4:      .include      gconst.h
5: #
6:      .xdef      gramadr
7:      .xdef      apage
8:      .xdef      gbase
9: #
10:     .text
11: #
12: #      座標からG-RAMアドレスを求める
13: #      (d0.w, d1.w) → a0 = adr
14: #
15: gramadr:
16:     movem.l d0-d1, -(sp)
17:
```

```

18:      movea.w d0, a0      #
19:      adda.w  d0, a0      #a0 = x*2
20:
21:      moveq.l  #GSFTCTR, d0  #
22:      ext.l    d1          #
23:      asl.l    d0, d1      #d1 = y*1024 (or y*2048)
24:      adda.l    d1, a0      #a0 = (x, y) と (0, 0) のG-RAMアドレスの差
25:
26:      adda.l    gbase(pc), a0  #a0 = (x, y) のG-RAMアドレス
27:
28:      movem.l   (sp)+, d0-d1
29:      rts
30:
31:  #
32:  #      描画ページを設定する
33:  #      d1.w = ページ番号 (0~3)
34:  #
35:  apage:
36:      locs      _APAGE      #IOCSにもページ番号を伝える
37:      tst.l     d0          #エラー?
38:      bmi      apage0      # そうなら終了
39:
40:      move.w    d1, d0      #d0.w=ページ番号
41:      lsl.w     #19-16, d0  #ページ番号を
42:      swap.w    d0          # $80000倍する
43:      clr.w     d0          #
44:      addi.l    #GPAGE0, d0  #G-RAMの先頭アドレスを足す
45:      move.l     d0, gbase  #ワークにしまう
46:
47:      moveq.l   #0, d0      #APAGEの戻り値を復帰する
48:  apage0: rts
49:  #
50:  gbase: .dc.l    GPAGE0      # (0, 0) のG-RAMアドレス
51:
52:      .end

```

リスト1は、実画面が1024ピクセルの場合と512ピクセルの場合のどちらでも使えるが、どちらのモード用かは“アセンブル時に決定する”ようになっている。つまり、生成されるオブジェクトはどちらかのモード専用となる。ここでは、AS.Xの条件つきアセンブル機能を利用している。

A>AS GRAMADR

のように普通にアセンブルすると実画面512ピクセルモード用、

A>AS GRAMADR /S_1024

というように“1024”というシンボルを定義すると1024ピクセルモード用のオブジェクトが生成される。サブルーチンの中でモードにおうじて処理を振り分けることもできたのだが、1本のプログラムで複数の画面モードを使うことはあまりないだろうという判断から、こういう形式にしてみた。

条件つきアセンブルされる部分はリスト1本体ではなく、インクルードしているリスト2のGCONST.H中に隠れている。3~16行が条件つきでアセンブルされる部分だ。

```
.ifdef _1024
```

(シンボル_1024が定義されているときアセンブルされる部分)

```
.else
```

(シンボル_1024が定義されていないときアセンブルされる部分)

```
.endif
```

のようなかたちになっているのがわかると思う。

リスト2
GCONST.H

```

1: #      グラフィック関連定数定義
2:
3: .ifdef _1024
4:      #実画面1024x1024
5:
6: GNPPIXEL equ 1024      #1ラインのピクセル数
7: GNBYTE equ 2048      #1ラインのバイト数
8: GSFTCTR equ 11      #GNBYTE倍するためのシフト回数
9: .else
10:      #実画面512x512
11:
12: GNPPIXEL equ 512
13: GNBYTE equ 1024
14: GSFTCTR equ 10
15:
16: .endif
17:
18: GPAGE0 equ $c00000 #グラフィックページ0先頭アドレス
19: GPAGE1 equ $c80000 #
20: GPAGE2 equ $d00000 #
21: GPAGE3 equ $d80000 #
22:
23: GPALET equ $e82000 #グラフィックパレット先頭アドレス

```

サブルーチンgramadrはd0.wにx座標、d1.wにy座標を入れて呼び出すと、対応するG-RAM上のアドレスをa0に返す。y座標を2048倍または1024倍する部分は遅い乗算命令ではなく、ビットシフトに置き換えて計算している。2048は2の11乗だから11ビット左シフトすれば2048倍したことになるし、同様に1024は2の10乗だから1024倍するには10ビット左シフトすればよい。何ビットシフトするかはGCONST.H内で記号定数GSFTCTRに定義している。GSFTCTRの値は、アセンブル時にシンボル“_1024”が定義されているかどうかで適切に切り替わる。

描画ページの先頭アドレスは35行以下の描画ページを設定するサブルーチンapage内であらかじめ計算し、ワークgbaseにしまっておくようにした。これにより、わずかながらアドレス計算時の手間を省くことができる。apageの中ではワークgbaseにアドレスをセットするだけでなく、IOCSコールAPAGEを呼び出すことでIOCS側にも描画ページの変更があったことを伝えている。これは、gramadr（を利用してこれからつくるサブルーチン群）とIOCSのグラフィック描画機能を混在して使用する場合に備えてのことだ。

座標からG-RAM上のアドレスを得るサブルーチンが用意できたところで、グラフィック画面に点を1つ打ってみる。これはもうかんたんで、

```

move.w    x座標,d0
move.w    y座標,d1
jsr       gramadr
move.w    パレットコード,(a0)

```

とやれば、望みの位置に望みの色で点が打てる。ただし、X68000のG-RAMはユーザーモードからはアクセスできないスーパーバイザ空間に割りつけられているから、あらかじめスーパーバイザモードへ移行しておく必要がある。

リスト3にサンプルを示しておこう。65536色モードにし、画面中央の座標 (256, 256) に白で1個点を打つプログラムだ。当然、リスト1とリンクして使う。サブルーチンgramadrが正常に動作しているかどうかテストするつもりで走らせてみてもらいたい。

リスト3
PSETTEST.S

```

1: *      点を打つテストプログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: *
6:      .xref      gramadr
7: *
8: CSCREEN      equ      16
9: DOS_GL3      equ      5
10: *
11:      .text
12:      .even
13: *
14: ent:
15:      lea.l      inisp(pc), sp      *spを初期化する
16:
17:      move.w     #DOS_GL3, -(sp)    *画面を512x512, 65536色に
18:      move.w     #CSCREEN, -(sp)    * 初期化
19:      DOS      _CONCTRL
20:      addq.l     #4, sp
21:
22:      clr.l      -(sp)              *スーパーバイザモードへ
23:      DOS      _SUPER              * 移行
24:      move.l     d0, (sp)           *現在のsspを待避
25:
26:      move.w     #256, d0           *x = 256
27:      move.w     #256, d1           *y = 256
28:      jsr      gramadr              *G-RAM上のアドレスを得る
29:      move.w     #65535, (a0)       *点を打つ
30:
31:      DOS      _SUPER              *
32:      addq.l     #4, sp              *ユーザーモードへ復帰
33:
34:      DOS      _EXIT                *終了
35: *
36:      .stack
37:      .even
38: *
39:      .ds.l      4096
40: inisp:
41:      .end      ent

```

矩形の塗り潰し

点が打てるところまで漕ぎ着けた。理屈のうえでは点が打てればどんな図形でも描けるわけで、後は描画アルゴリズム“だけ”の問題となる。手始めにX-BASICのグラフィック描画関数のなかで (psetを除けば) いちばんかんたんに実現できそうなfill関数の相当品をつくってみることにする。

矩形塗り潰しの描画アルゴリズムについては説明するまでもないだろう。矩形の左端からx座標を1ずつ増やしながら順に点を打っていく処理を矩形領域の高さの回数だけ繰り返せばよ

い。プログラムは単純な2重ループになる。あっという間にできたプログラムがリスト4。サブルーチンだけなので、動作試験にはリスト5のテスト用プログラムを利用してほしい。個別にアセンブルのうえ、

A>LK GFILLTEST GFILL GRAMADR

により必要モジュールをリンクする。なお、リスト4もGCONST.Hをインクルードしているので、アセンブル時にシンボル“_1024”が定義されていれば実画面1024ピクセルモード用、定義されていなければ512ピクセルモード用のオブジェクトが得られる。これ以降、GCONST.Hをインクルードしているプログラムはすべてこのノリだ。

リスト4
GFILL.S

```

1: *      矩形の塗り潰し (第1版)
2:
3:      .include      gconst.h
4: *
5:      .xdef    gfill
6:      .xref    gramadr
7: *
8:      .offset 0      *gfillの引数構造
9: *
10: X0:      .ds.w    1      *描画先座標
11: Y0:      .ds.w    1      *
12: X1:      .ds.w    1      *
13: Y1:      .ds.w    1      *
14: COL:     .ds.w    1      *描画色
15: *
16:      .text
17:      .even
18: *
19: gfill:
20: ARGPTR =      8
21:      link      a6, #0
22:      movem.l   d0-d3/a0-a1, -(sp)
23:
24:      movea.l   ARGPTR(a6), a1      *a1 = 引数列
25:      movem.w   (a1), d0-d3      *d0~d3に座標を取り出す
26:
27:      jsr      gramadr      *G-RAM上のアドレスを得る
28:
29:      sub.w     d0, d2      *d2 = 横ピクセル数-1
30:      sub.w     d1, d3      *d3 = 縦ピクセル数-1
31:
32:      move.w     COL(a1), d0      *d0 = 描画色
33:
34: loop1: movea.l   a0, a1      *a1 = 左端アドレス
35:      move.w     d2, d4      *d4 = 横ピクセル数-1
36: loop2: move.w     d0, (a1)+      *1ピクセル点を打つ
37:      dbra      d4, loop2      *横幅分繰り返す
38:
39:      lea.l      GNBYTE(a0), a0      *すぐ下のラインへ
40:      dbra      d3, loop1      *高さ分繰り返す
41:
42: done: movem.l   (sp)+, d0-d3/a0-a1
43:      unlink    a6
44:      rts
45:
46:      .end

```

リスト5
GFILLTEST.S

```

1: *      gfillの動作試験用プログラム
2:

```

```

3:      .include      doscall.mac
4:      *
5:      .xref      gfill
6:      *
7:      CSCREEN      equ      16
8:      DOS_GL3      equ      5
9:      *
10:     .text
11:     .even
12:     *
13:     ent:
14:     lea.l      inisp, sp      *spを初期化する
15:
16:     move.w      #DOS_GL3, -(sp) *画面を512x512, 65536色に
17:     move.w      #CSCREEN, -(sp) * 初期化
18:     DOS      _CONCTRL      *
19:     addq.l      #4, sp      *
20:
21:     clr.l      -(sp)      *スーパーバイザモードへ
22:     DOS      _SUPER      * 移行
23:     move.l      d0, (sp)    *現在のsspを待避
24:
25:     pea.l      arg      *引数列
26:     bsr      gfill      *描画
27:     addq.l      #4, sp      *
28:
29:     DOS      _SUPER      *
30:     addq.l      #4, sp      *ユーザーモードへ復帰
31:
32:     DOS      _EXIT      *終了
33:     *
34:     .data
35:     .even
36:     *
37:     arg:      .dc.w      100, 100, 200, 200, 12345
38:     *
39:     .stack
40:     .even
41:     *
42:     .ds.l      4096
43:     inisp:
44:     .end      ent

```

リスト4を順に見ていこう。このサブルーチンgfillへの引数受け渡しは、適当なメモリ領域に左上の座標、右下の座標、パレットコード各2バイトを順に格納し（IOCSコールFILLと同じ形式）、その先頭アドレスをスタックに積むことで行う。8～14行ではこの引数の構造を.offset疑似命令で定義している。24～25行で引数が、

d0=左上x座標

d1=左上y座標

d2=右下x座標

d3=右下y座標

に取り出される。27行でサブルーチンgramadrを呼び出し²⁾、左上の座標に対応するG-RAM上のアドレスを得る。それから、29～30行で描くべき長方形の横のピクセル数-1と縦のピクセル数-1を求め、これをループカウンタの初期値とする。32行で描画色のパレットコードをd0.wに入れているのは、たんに、ループの内側でひんぱんに使う定数をレジスタに入れておいて余分なメモリアクセスをしなくてもすむようにしているにすぎない。

■2) gramadrの呼び出しにはjsrではなくbsrを使ったほうが実行速度は上がる。しかし、PC相対のサブルーチンコール命令であるbsrでは分岐できる距離に制限があるため、ある程度以上大きなプログラムに組み込んで使おうとすると、距離が離れすぎていてリンク時にエラーが出る可能性がある。そこで、本書では、別モジュール内のサブルーチンを呼び出すときには、つねにjsr（の絶対ロング形式）を使うよう統一する。

34行以降がG-RAMに書き込みを行う2重ループだ。36～37行の内側のループで水平方向に線分を引く。ここでは、ポストインクリメント付きアドレスレジスタ間接アドレッシングが綺麗に決まっている。1ライン分描画したら、39行で描画先水平ラインの左端を保持するa0に記号定数GNBYTEを加える。GNBYTEはGCONST.H内で定義された、G-RAMの水平1ライン分のバイト数を意味する記号定数で、実画面512ピクセルモード時1024、1024ピクセルモード時2048の値をとる。G-RAMアドレスにGNBYTEを加えることにより、真下のピクセルのG-RAMアドレスが得られる。以上の処理を34～40行の外側のループで必要なライン数分繰り返して描画完了だ。

実際に描画を行うループ中には座標がまったく顔を出さないことに注目してほしい。x座標が1増加するとG-RAMアドレスは2増え、y座標が1増加するとG-RAMアドレスはGNBYTE増えることがわかっているから、基点（いまの場合、左上隅の点）のG-RAMアドレスを一度求めておけば、座標を増減してからG-RAMアドレスを求め直さなくとも、直接G-RAMアドレスを更新することができる。

矩形のクリッピング

リスト4のgfillはエラーチェックを一切していないため、不当な座標値を与えて呼び出すと誤動作したり、バスエラーが発生したりする可能性がある。gfillはスーパーバイザモードで走らようにつくられているプログラムであり、68000のメモリ保護機能もあまりあてにはできない。エラーチェックをして実行速度が遅くなるよりは、呼び出し側でつねに正しいパラメータを渡すよう気をつける、というのも1つの考え方だが、ここでは真面目にエラーチェックする方向で話を進める。どのような場合に誤動作するかを把握し、個別に対応を考えてみよう。

1) 座標の大小関係が狂っているとき

リスト4ではX-BASICでいう、

```
fill (511,511,0,0,15)
```

のような指定がされることを考慮していない。座標をデータレジスタに取り出した時点で、 $d0 \leq d2$, $d1 \leq d3$ であるものと思い込み、29～30行で $d2 - d0$, $d3 - d1$ を求めループカウンタの初期値としている。 $d0 > d2$ や $d1 > d3$ の場合は減算の結果が負となりループカウンタとしては使いものにならないのに、そのまま処理を進めているのだ。つねに正しい動作を保証するた

めには、あらかじめ座標の大小関係を調べ、必要ならd0とd2, d1とd3の値を交換しておかなければならない。ちなみに、レジスタ内容の交換にはexg命令が使える。

2) 座標が実画面の範囲外するとき

これはG-RAMの範囲外の変なメモリへ書き込みを行ってしまうことを意味するから、よくてバスエラー、運が悪ければ暴走する危険がある。対応策としては、四隅のどれか1つでも画面の外だったら描画を一切行わないという消極的な案と、画面外にはみ出す部分を切り取って、画面内に収まる範囲のみを描画する（クリッピングする）という案がある。後者が望ましいのはいうまでもない。

リスト6にgfill用のクリッピングサブルーチンを示す。内部ルーチンということで、引数はレジスタ渡しになっている。座標をd0～d3に入れて渡すと、クリッピングされた座標がやはりd0～d3に戻る。ただし、指定された矩形が完全に実画面の外にある場合はccrのNビットを立てて戻ることにした。X-BASICやIOCS同様に有効な座標の範囲を-32768～+32767にしたかった関係で、座標の比較はすべて符号付き16ビット数として行っている。クリッピングの処理上、 $d0 \leq d2$, $d1 \leq d3$ でない和不都合があるので、どさくさに座標の大小関係の乱れも直している（20～21行）。ここで使っているMINMAXは本書のプログラム作成用に用意したマクロ集GMACRO.H（リスト7）の46～52行で定義してある小さなマクロだ。GMACRO.Hは以降の章でもたびたび利用するので、手元に置いておいてほしい。

リスト6
GCLIPRECT.S

```

1: *      矩形領域のクリッピング
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5:      .include      rect.h
6: *
7:      .xdef      gcliprect
8:      .xref      cliprect
9: *
10:     .text
11:     .even
12: *
13: *      (d0, w, d1, w) - (d2, w, d3, w) を
14: *      cliprectで指定された矩形領域でクリッピングする
15: *
16: gcliprect:
17:     move.l      a0, -(sp)
18:     lea.l       cliprect, a0
19:
20:     MINMAX      d0, d2      #d0 ≤ d2を保証する
21:     MINMAX      d1, d3      #d1 ≤ d3を保証する
22:
23:     cmp.w       MAXX(a0), d0      #d0 > MAXX?
24:     bgt         outofscrn         # そうなら画面外
25:     cmp.w       MAXY(a0), d1      #d1 > MAXY?
26:     bgt         outofscrn         # そうなら画面外
27:
28:     cmp.w       MINX(a0), d2      #d2 < MINX?
29:     blt         outofscrn         # そうなら画面外
30:     cmp.w       MINY(a0), d3      #d3 < MINY?
31:     blt         outofscrn         # そうなら画面外
32:
33:     cmp.w       MINX(a0), d0      #d0 < MINX?

```

```

34:         bge     skip1      #
35:         move.w  MINX(a0), d0 # そうなら修正
36:
37: skip1:    cmp.w   MINY(a0), d1 #d1<MINY?
38:         bge     skip2      #
39:         move.w  MINY(a0), d1 # そうなら修正
40:
41: skip2:    cmp.w   MAXX(a0), d2 #d2>MAXX?
42:         ble     skip3      #
43:         move.w  MAXX(a0), d2 # そうなら修正
44:
45: skip3:    cmp.w   MAXY(a0), d3 #d3>MAXY?
46:         ble     skip4      #
47:         move.w  MAXY(a0), d3 # そうなら修正
48:
49: skip4:    cmp.w   d0, d0     #N=0
50: done:     movea.l (sp)+, a0
51:         rts
52: #
53: outofscrn:
54:         moveq.l  #-1, d0     #N=1
55:         bra     done
56:
57:         .end

```

リスト7
GMACRO.H

```

1: #      グラフィック関連マクロ定義
2:
3: IMASK   equ     %00000_00000_00000_1
4: BMASK   equ     %00000_00000_11111_0
5: RMASK   equ     %00000_11111_00000_0
6: GMASK   equ     %11111_00000_00000_0
7: #
8: RGBMAX  equ     31
9: RGBGRAD equ     RGBMAX+1
10: #
11: ABS      macro   X          #abs(x)
12:         local   done
13:         tst.w   X
14:         bpl     done
15:         neg.w   X
16: done:
17:         .endm
18: #
19: SGN      macro   X          #sgn(x)
20:         local   min, done
21:         tst.w   X
22:         bmi     min
23:         beq     done
24:         move.w  #1, X
25:         bra     done
26: min:     move.w  #-1, X
27: done:
28:         .endm
29: #
30: MIN      macro   Y, X       #min(x, y)
31:         local   done
32:         cmp.w   Y, X
33:         ble     done
34:         move.w  Y, X
35: done:
36:         .endm
37: #
38: MAX      macro   Y, X       #max(x, y)
39:         local   done
40:         cmp.w   Y, X
41:         bge     done
42:         move.w  Y, X
43: done:
44:         .endm
45: #

```

```

46: MINMAX macro Y, X          *x=max(x, y), y=min(x, y)
47:         local skip
48:         cmp.w Y, X
49:         bge skip
50:         exg.l Y, X
51: skip:
52:         .endm
53: *
54: MEAN macro Y, X              *x=(x+y)/2
55:         add.w Y, X
56:         lsr.w #1, X
57:         .endm
58: *
59: DERGB macro COL, _B, _R, _G
60:         move.w #RGBMAX, _B    * B = 0000000000011111
61:         move.w _B, _R        * R = 0000000000011111
62:         lsr.w #1, COL        *COL = 0GGGGGRRRRRBBBBBB
63:         and.w COL, _B        * B = 00000000000BBBBB
64:         lsr.w #5, COL        *COL = 00000GGGGGRRRRRR
65:         and.w COL, _R        * R = 00000000000RRRRR
66:         lsr.w #5, COL        *COL = 00000000000GGGGG
67:         move.w COL, _G        * G = 00000000000GGGGG
68:         .endm
69: *
70: RGB macro B, R, G, COL
71:         move.b G, COL        *COL = xxxxxxxx00GGGGG
72:         lsl.w #5, COL        *COL = xxx00GGGGG00000
73:         or.b R, COL          *COL = 00000GGGGGRRRRRR
74:         lsl.w #5, COL        *COL = 0GGGGGRRRRR00000
75:         or.b B, COL          *COL = 0GGGGGRRRRRBBBBBB
76:         add.w COL, COL        *COL = GGGGGRRRRRBBBBBB0
77:         .endm
78: *
79: RGBtoY macro B, R, G, Y
80:         add.w G, G            *G = 2G
81:         add.w R, G            *G = 2G+R
82:         move.w G, Y          *Y = 2G+R
83:         add.w Y, Y            *Y = 2(2G+R)
84:         add.w G, Y            *Y = 3(2G+R)
85:         add.w B, Y            *Y = 6G+3R+B
86:         ext.l Y
87:         divu.w #10, Y         *Y = (6G+3R+B)/10
88:         .endm

```

リスト6では、クリッピング領域はリスト8のRECT.Hに示す構造で外部ラベルcliprect以降に用意する仕様になっており、そのcliprect本体は、リスト9にクリッピングウィンドウの設定ルーチンといっしょにまとめてある。初期状態におけるクリッピングウィンドウは、実画面と一致するようにしてある（39～42行）。リスト9のサブルーチンsetcliprectは、引数として渡された矩形の座標をcliprect以下のワークに格納すると同時に、IOCS WINDOWを呼び出し、クリッピングウィンドウの変更をIOCSにも伝えている。先ほどのapageと同じ考え方だ。なお、リスト9では、26～31行で少々余計な処理をしているが、この部分については、次章で触れる。

.....
リスト8
RECT.H

```

1: *      矩形指定用構造体
2:
3:         .offset 0
4: *
5: MINX:   .ds.w 1              *左上隅x座標
6: MINY:   .ds.w 1              *左上隅y座標

```

```

7: MAXX: .ds.w 1      #右下隅x座標
8: MAXY: .ds.w 1      #右下隅x座標
9: RECT:
10: #
11: .text

```

.....
リスト9
SETCLIPRECT.S

```

1: #      クリッピングウィンドウの設定
2:
3: .include      iocscall.mac
4: .include      gconst.h
5: #
6: .xdef      setcliprect
7: .xdef      cliprect
8: .xdef      ucliprect
9: #
10: .text
11: .even
12: #
13: setcliprect:
14: ARGPTR =      8
15: link      a6,#0
16: movem.l   d1-d4, -(sp)
17:
18: movea.l   ARGPTR(a6), a1      #a1=引数
19: movem.w   (a1), d1-d4      #d1~d4に座標を取り出し
20: IOCS      _WINDOW      # IOCSに伝える
21: tst.l     d0      #エラーなら
22: bmi      retn      # 抜ける
23:
24: movem.w   d1-d4, cliprect      #クリッピングウィンドウの
25:                                # 座標を覚えておく
26: move.w     #$8000, d0      #$8000のゲタを履かせて
27: add.w      d0, d1      #
28: add.w      d0, d2      #
29: add.w      d0, d3      #
30: add.w      d0, d4      #
31: movem.w   d1-d4, ucliprect      # 別に覚えておく
32:
33: moveq.l    #0, d0      #正常終了
34: retn:      movem.l   (sp)+, d1-d4
35: unlink     a6
36: rts
37: #
38: cliprect:
39: .dc.w      0      #クリッピング領域
40: .dc.w      0      #
41: .dc.w      GNPPIXEL-1      #
42: .dc.w      GNPPIXEL-1      #
43: ucliprect:
44: .dc.w      $8000      #クリッピング領域
45: .dc.w      $8000      # ($8000のゲタ履き)
46: .dc.w      $8000+GNPIXEL-1      #
47: .dc.w      $8000+GNPIXEL-1      #
48:
49: .end

```

では、リスト4にクリッピングサブルーチンの呼び出しをつけ加えておく。25行のうしろに、

```
jsr      gcliprect
```

```
bmi      done
```

の2行を挿入し、また、7行のあたりにでも、

```
.xref      gcliprect
```

の1行を挿入する。修正がすんだら、あらためて動作試験をしてみよう。テストプログラムで

あるリスト5中の座標値をいろいろ変更して実画面の端に掛かるようにし、うまくクリッピングできているかどうか確かめてほしい。モジュールが増えたので、リンク手順は、

```
A>LK GFILLTEST GFILL GRAMADR GCLIPRECT SETCLIPRECT
```

となることに注意。後ろの3つのモジュールは以降の章でも毎回のよう利用するので、ライブラリにまとめておくとよいだろう。

ボックスフィルルーチンの高速化

せっかくつくったサブルーチンだ。もう少し高速化してやろう。といっても、アルゴリズムに関してはもともと単純なのでこれ以上最適化しようがない。コーディングのうえでも、多少は改善の余地があるにせよ、劇的な効果は望めそうもないように見える。ループの中身で無駄なことをしているようならそれをループの外に追い出すことでかなり実行速度が向上するもののだが、リスト4のもっとも内側のループは、

```
loop2: move.w d0, (a1)+
        dbra    d4,loop2
```

というこれ以上はかんたんにならないかたちだし、外側のループにも極端な無駄は見られない。唯一、34行のmoveaは工夫次第で省略できるとはいえ、36～37行のループで食う時間に比べれば転送命令1つの実行時間なんか気にならない程度であり、こだわってみてもあまり意味がない。

どうも真正直にやっていたのではそれほど速くなりそうもないので、ループを展開するという姑息な方法を検討してみる。話をかんたんにするために描く長方形の横幅はつねに512ピクセルということにし、もっとも内側のループだけを取り出して考えよう。いまこのループは、

```
move.w #512-1,d4
loop:  move.w d0, (a0)+
        dbra    d4,loop
```

というかたちになっているわけだが、これを

```
move.w #512/2-1,d4
loop:  move.w d0, (a0)+
        move.w d0, (a0)+
        dbra    d4,loop
```

のように書き直してループの中身を2倍に展開すれば、ループ回数、すなわちdbraの実行回数を半分に減らすことができる。さらには、

```
move.w #512/4-1,d4
loop:  move.w d0, (a0)+
        move.w d0, (a0)+
        move.w d0, (a0)+
        dbra    d4,loop
```

```

move.w d0, (a0) +
move.w d0, (a0) +
dbra    d4, loop

```

と4倍に展開すればdbraの実行回数は4分の1だ。この話をずーっと突き詰めていくと、

```
move.w d0, (a0) +
```

を512個並べて完全に展開してしまえば、dbraを一度も実行しなくてすむ。

当然のようにループを2倍、4倍、……と展開するにつれ、プログラムはだらだらと長くなる。ループ展開とは、メモリ効率を犠牲にして、えげつなく速度を稼ぐ、非常手段なのであった。ループ展開を採用するにあたっては、多少照れたフリをするのを忘れないようにしたい。

で、照れながらも、とにかく512個の、

```
move.w d0, (a0) +
```

とrtsからなるhlineというサブルーチンを用意したとする。a0がG-RAM内を指しているとする、hlineの先頭から実行すればd0.wで指定されたパレットコードで512ピクセルの水平線が描けるわけだ。それでは、hline先頭のmoveを1個飛ばして、その直後から実行すればどうなるだろうか。そう、511ピクセルの線分が描ける。2個飛ばせば510ピクセルだ。

```
move.w d0, (a0) +
```

は1ワード命令だから、

```
hline + (512 - 描きたいピクセル数) × 2
```

のアドレスから実行すれば、希望どおりの長さの線分を描くことができる。サブルーチン末のrtsが置かれたアドレス（仮にhline0とすると）を基点と考えるとこの式はもう少しかんたんに表せ、

```
hline0 - 描きたいピクセル数 × 2
```

となる。

リスト10が水平描画部分のループを展開したgfillの新しい版だ。ソースリストの短さは意外かもしれない。展開したループの中身はどこにいつてしまったかというと、47行に凝縮されている。この.dcbは同じ値の定数が並んだブロックを用意するAS.Xの疑似命令で、47行では30C0_HというワードデータをGNPIXEL個だけ用意している。ここで、30C0_Hは、

```
move.w d0, (a0) +
```

のマシンコードであり、GNPIXELはGCONST.H内で定義された実画面の横幅ピクセル数を表す記号定数だ。結局、アセンブルすると、この1行は、

```
move.w d0, (a0) +
```

を実画面の横幅ピクセル個並べたものに展開される。

.....
リスト10
GFILLS
(ループ展開版)

```

1: *      矩形の塗り潰し (ループ展開版)
2:
3:      .include      gconst.h
4: *

```

```

5:      .xdef    gfill
6:      .xref    gramadr
7:      .xref    gcliprect
8:      #
9:      .offset 0      #gfillの引数構造
10:     #
11:     X0:      .ds.w    1      #描画先座標
12:     Y0:      .ds.w    1      #
13:     X1:      .ds.w    1      #
14:     Y1:      .ds.w    1      #
15:     COL:     .ds.w    1      #描画色
16:     #
17:     .text
18:     .even
19:     #
20:     gfill:
21:     ARGPTR    =        8
22:     link      a6, #0
23:     movem.l   d0-d3/a0-a1, -(sp)
24:
25:     movea.l   ARGPTR(a6), a1    #a1 = 引数列
26:     movem.w   (a1), d0-d3      #d0~d3に座標を取り出す
27:
28:     jsr       gcliprect        #クリッピングする
29:     bmi       done             #N=1なら描画の必要なし
30:
31:     jsr       gramadr          #左上のG-RAM上のアドレスを得る
32:
33:     sub.w     d0, d2            #d2 = 横ピクセル数-1
34:     sub.w     d1, d3            #d3 = 縦ピクセル数-1
35:
36:     move.w    COL(a1), d0      #d0 = 描画色
37:
38:     addq.w    #1, d2            #d2 = 横ピクセル数
39:     add.w     d2, d2            #d2 = 1ライン分のバイト数
40:     lea.l     next(pc), a1     #a1 = 横ピクセル数の
41:     suba.w    d2, a1            # 水平線分描画ルーチン
42:
43:     move.w    #GNBYTE, d1      #d1 = ライン間のアドレスの差
44:     sub.w     d2, d1            #
45:
46:     loop:     jmp      (a1)      #1ライン描画
47:     .dcb.w    GNPPIXEL, $30c0  # move.w d0, (a0) +
48:     next:     adda.w    d1, a0   #すぐ下のラインへ
49:     dbra     d3, loop          #高さ分繰り返す
50:
51:     done:     movem.l   (sp)+, d0-d3/a0-a1
52:     unlk      a6
53:     rts
54:
55:     .end

```

リスト10の改造により、gfillは数段高速になった。だが、gfillの水平線分描画部分は、

```
move.w d0, (a0) +
```

で1ピクセルずつ点を打つ代わりに、

```
move.l d0, (a0) +
```

で2ピクセルまとめて描くことによってさらに高速化できる。しかも、この場合、ループの展開部がいままでの半分の個数のmove命令ですむため、いくぶん、メモリ効率も改善される。ただ、横ピクセル数がつねに偶数とはかぎらないから、奇数の場合は最初か最後の1ピクセルを特別扱いする必要がある。この点に注意し、プログラムにしてみたらリスト11のようになった。リスト10からの変更部分に注目して読んでみてほしい。

リスト11
GFILL.S
(2ピクセル同時書き込み版)

```

1: #      矩形の塗り潰し（2ピクセル同時書き込み版）
2:
3:      .include      gconst.h
4: #
5:      .xdef      gfill
6:      .xref      gramadr
7:      .xref      gcliprect
8: #
9:      .offset 0      *gfillの引数構造
10: #
11: X0:      .ds.w      1      *描画先座標
12: Y0:      .ds.w      1      *
13: X1:      .ds.w      1      *
14: Y1:      .ds.w      1      *
15: COL:     .ds.w      1      *描画色
16: #
17:      .text
18:      .even
19: #
20: gfill:
21: ARGPTR =      8
22:      link      a6, #0
23:      movem.l    d0-d3/a0-a2, -(sp)
24:
25:      movea.l    ARGPTR(a6), a1      *a1 = 引数列
26:      movem.w    (a1), d0-d3      *d0～d3に座標を取り出す
27:
28:      jsr      gcliprect      *クリッピングする
29:      bmi      done      *N=1なら描画の必要なし
30:
31:      jsr      gramadr      *左上のG-RAM上のアドレスを得る
32:
33:      sub.w      d0, d2      *d2 = 横ピクセル数-1
34:      sub.w      d1, d3      *d3 = 縦ピクセル数-1
35:
36:      move.w      COL(a1), d0      *d0 = 描画色
37:      swap.w      d0      *
38:      move.w      COL(a1), d0      *
39:
40:      addq.w      #1, d2      *d2 = 横ピクセル数
41:      lea.l      next(pc), a2      *
42:      bclr.l      #0, d2      *横ピクセル数は奇数か？
43:      beq      skip      *
44:      lea.l      odd(pc), a2      *奇数ピクセルのとき
45:
46: skip:      lea.l      hline0(pc), a1      *
47:      suba.w      d2, a1      *a1 = 横ピクセル数（偶数）の
48:                               * 水平線分描画ルーチン先頭
49:
50:      move.w      #GNBYTE, d1      *d1=ライン間のアドレスの差
51:      add.w      d2, d2      *
52:      sub.w      d2, d1      *
53:
54: loop:      jmp      (a1)      *1ライン描画
55: hline:      .dcb.w      GNPIXEL/2, $20c0      *move.l d0, (a0)+
56: hline0:      jmp      (a2)
57: odd:      move.w      d0, (a0)      *奇数ピクセルの場合
58: next:      adda.w      d1, a0      *すぐ下のラインへ
59:      dbra      d3, loop      *高さ分繰り返す
60:
61: done:      movem.l    (sp)+, d0-d3/a0-a2
62:      unlink      a6
63:      rts
64:
65:      .end

```

リスト11では、ループの中で毎回、矩形の横幅が奇数か偶数か調べて処理を振り分けるのではなく、ループに入る前にひと細工している。横幅が偶数ピクセルのときは、a2レジスタ

が58行を指すよう、奇数のときは57行を指すようにしておき、56行のjmpでa2の指す位置に直接飛んでいる。結果、横幅が偶数ピクセルのときは、57行を飛び越すかたちとなる。

42行のbclrの使い方もポイントといえるかもしれない。普通のやり方では、ここは、

- 1) 横幅を保持するd2が奇数かどうか調べて覚えておく
- 2) d2を2で割る (2ピクセル同時書き込みの分)
- 3) d2を2倍する (move.l d0, (a0) +は2バイト命令)

となるところだが、bclr命令1つで1)～3)の働きをまとめて行っている。bclrは、指定ビットをリセットするのに先立って、そのビットが0か1かをccrのZビットに反映することを思い出す。

さて、1命令で書き込むピクセル数をもっと増やせば描画速度はさらに上がる。68000の扱えるデータは32ビット長までだから、これ以上は無理に見えるかもしれないが、まだ先があるのだ。68000にはメモリに対して最大64バイトを1命令で書き込む手段がある。次の命令がそうだ。

```
movem.l d0-d7/a0-a7, -(a0)
```

これをG-RAMに適用できれば、レジスタの上位/下位ワードにパレットコードを入れておくことで32ピクセルの書き込みが同時に行える計算になる。32個並べれば、1024ピクセルの水平線分が描けるわけだ。もっとも、実際にはスタックポインタに適当な値を入れるわけにはいかないし、アドレスレジスタの1本はG-RAMへのポインタに使うので、同時に書き込めるバイト数はもう少し減る。

リスト12に16ピクセル同時書き込み32倍 (実画面512ピクセルモード時。1024ピクセルモード時は64倍) ループ展開版のボックスフィルルーチンを示す。32個 (または64個) 並べた

```
movem.l d0/d4-d7/a3-a5, -(a1)
```

の途中、適当な位置に飛び込むことで16ピクセル単位の任意の長さの水平線分を描き、16ピクセル未満の端数については、dbraループで処理している。リスト12を読むときにはmovemが4バイト命令であることに気をつけてほしい。

.....
リスト12
GFILL.S
(最終版)

```
1: #      矩形の塗り潰し (最終版: 16ピクセル同時書き込み)
2:
3:      .include      gconst.h
4: #
5:      .xdef      gfill
6:      .xref      gramadr
7:      .xref      gcliprect
8: #
9: #      128ピクセル書き込むマクロ
10: #
11: H128      macro
12:      movem.l d0/d4-d7/a3-a5, -(a1)
13:      movem.l d0/d4-d7/a3-a5, -(a1)
14:      movem.l d0/d4-d7/a3-a5, -(a1)
15:      movem.l d0/d4-d7/a3-a5, -(a1)
16:      movem.l d0/d4-d7/a3-a5, -(a1)
17:      movem.l d0/d4-d7/a3-a5, -(a1)
18:      movem.l d0/d4-d7/a3-a5, -(a1)
19:      movem.l d0/d4-d7/a3-a5, -(a1)
20:      .endm
```

```

21: *
22: .offset 0      *gfillの引数構造
23: *
24: X0: .ds.w 1     *描画先座標
25: Y0: .ds.w 1     *
26: X1: .ds.w 1     *
27: Y1: .ds.w 1     *
28: COL: .ds.w 1    *描画色
29: *
30: .text
31: .even
32: *
33: gfill:
34: ARGPTR = 8
35: link a6, #0
36: movem.l d0-d7/a0-a5, -(sp)
37:
38: movea.l ARGPTR(a6), a1  *a1 = 引数列
39: movem.w (a1), d0-d3     *d0~d3に座標を取り出す
40:
41: jsr gcliprect          *クリッピングする
42: bmi done               *N=1なら描画の必要なし
43:
44: jsr gramadr            *左上のG-RAM上のアドレスを得る
45:
46: sub.w d0, d2           *d2 = 横ピクセル数-1
47: sub.w d1, d3           *d3 = 縦ピクセル数-1
48:
49: move.w COL(a1), d0     *d0 = 描画色
50: move.w d0, d7          *
51: swap.w d0              *
52: move.w d7, d0          *
53:
54: move.l d0, d4
55: move.l d0, d5
56: move.l d0, d6
57: move.l d0, d7
58: move.l d0, a3
59: move.l d0, a4
60: move.l d0, a5
61:
62: addq.w #1, d2           *d2 = 横ピクセル数
63: adda.w d2, a0           *
64: adda.w d2, a0           *a0 = (x1+1, y0) のG-RAMアドレス
65:
66: moveq.l #16-1, d1      *
67: and.w d2, d1           *d1 = 横ピクセル数 % 16
68: sub.w d1, d2           *d2 = (横ピクセル数 / 16) * 16
69:
70: lsr.w #2, d2           *d2 = (横ピクセル数 / 16) * 4
71: lea.l hline0(pc), a2   *
72: suba.w d2, a2          *a2 = (横ピクセル数/16)*16ピクセル分の
73:                        * 水平線分描画ルーチン
74:
75: loop1: movea.l a0, a1
76: jmp (a2)               *1ライン描画
77:
78: .ifdef _1024
79: H128 *1024
80: H128 * :
81: H128 * :
82: H128 * :
83: .endif
84: H128 *512
85: H128 *384
86: H128 *256
87: H128 *128
88:
89: hline0: move.w d1, d2   *16ピクセルに満たない
90: bra next2              * 端数を描画
91: loop2: move.w d0, -(a1) *
92: next2: dbra d2, loop2   *
93:
94: next1: lea.l GNBYTE(a0), a0 *すぐ下のラインへ

```

```

95:      dbra    d3, loop1      #高さ分繰り返す
96:
97: done:  movem.l (sp)+, d0-d7/a0-a5
98:      unlk    a6
99:      rts
100:
101:      .end

```

余談ながら、本書のプログラムでは、アドレスレジスタへの代入時にはかならずmoveaと書くようにしているのだが、リスト12の58～60行では単にmoveと記述している。これは、ちょっとしたおまじないだ。ここでアドレスレジスタに入れているのは「アドレスではない」ことを強調している（どのみちアセンブラは、これをmoveaと解釈する）。

タイリングへの対応

最後にgfillのバリエーションを1つつくって終わりにする。いわゆるタイリングができるようにしてみよう。タイリング (tiling) とは異なる色を交互に並べることで解像度をある程度犠牲にして疑似的に多色表示を行う手段だ。65536色が同時に出せるX68000では無用といえそうなのだが、16色モードや256色モードではまだまだ使い道はあると思う。

あまり複雑にはしたくないので、2色の市松模様専用に限定して考えよう。要は2色の点を交互に並べていけばよいわけだ。リスト12にはレジスタの上位ワードと下位ワードに同一のパレットコードを格納している部分があるが、ここを少し変更し、上位ワードと下位ワードに異なる色のパレットコードを入れるようにすれば、おおむね目的が達せられる。後は細かなつじつまあわせだが、これについては実際のコードを見てもらったほうがよくわかるだろう。リスト13だ。

リスト13
GTILEFILL.S

```

1: #      矩形の塗り潰し (2色タイリング対応版)
2:
3:      .include      gconst.h
4: #
5:      .xdef      gtilefill
6:      .xref      gramadr
7:      .xref      gcliprect
8: #
9: #      128ピクセル書き込むマクロ
10: #
11: H128  macro
12:      movem.l d0/d4-d7/a3-a5, -(a1)
13:      movem.l d0/d4-d7/a3-a5, -(a1)
14:      movem.l d0/d4-d7/a3-a5, -(a1)
15:      movem.l d0/d4-d7/a3-a5, -(a1)
16:      movem.l d0/d4-d7/a3-a5, -(a1)
17:      movem.l d0/d4-d7/a3-a5, -(a1)
18:      movem.l d0/d4-d7/a3-a5, -(a1)
19:      movem.l d0/d4-d7/a3-a5, -(a1)
20:      .endm
21: #
22:      .offset 0      #gtilefillの引数構造
23: #

```

```

24: X0:      .ds.w 1      *描画先座標
25: Y0:      .ds.w 1      *
26: X1:      .ds.w 1      *
27: Y1:      .ds.w 1      *
28: COL1:    .ds.w 1      *描画色1
29: COL2:    .ds.w 1      *描画色2
30: *
31:          .text
32:          .even
33: *
34: gtilefill:
35: ARGPTR    = 8
36:          link    a8, #0
37:          movem.l d0-d7/a0-a5, -(sp)
38:
39:          movea.l ARGPTR(a6), a1      *a1 = 引数列
40:          movem.w (a1), d0-d3        *d0~d3に座標を取り出す
41:
42:          jsr      gcliprect          *クリッピングする
43:          bmi      done                *N=1なら描画の必要なし
44:
45:          jsr      gramadr             *左上のG-RAM上のアドレスを得る
46:
47:          sub.w    d0, d2              *d2 = 横ピクセル数-1
48:          sub.w    d1, d3              *d3 = 縦ピクセル数-1
49:          addq.w   #1, d2              *d2 = 横ピクセル数
50:
51:          eor.w    d0, d1              *
52:          eor.w    d2, d1              *
53:          move.l   COL1(a1), d0        *d0 = 描画色
54:          btst.l   #0, d1              * (0, 0)にCOL1がくるよう
55:          beq      skip                * 必要ならCOL1とCOL2を交換する
56:          swap.w   d0                  *COL1 ↔ COL2
57:
58: skip:      adda.w  d2, a0              *
59:          adda.w   d2, a0              *a0 = (x1+1, y0) のG-RAMアドレス
60:
61:          moveq.l   #16-1, d1          *
62:          and.w     d2, d1              *d1 = 横ピクセル数 % 16
63:          sub.w     d1, d2              *d2 = (横ピクセル数 / 16) * 16
64:
65:          lsr.w     #2, d2              *d2 = (横ピクセル数 / 16) * 4
66:          lea.l     hline0(pc), a2     *
67:          suba.w    d2, a2              *a2 = (横ピクセル数/16)*16ピクセル分の
68:          * 水平線分描画ルーチン
69:
70: loop1:     movea.l a0, a1
71:          move.l    d0, d4
72:          move.l    d0, d5
73:          move.l    d0, d6
74:          move.l    d0, d7
75:          move.l    d0, a3
76:          move.l    d0, a4
77:          move.l    d0, a5
78:          jmp      (a2)                *1ライン描画
79:
80: .ifdef    _1024
81:          H128     *1024
82:          H128     * :
83:          H128     * :
84:          H128     * :
85: .endif
86:          H128     *512
87:          H128     *384
88:          H128     *256
89:          H128     *128
90:
91: hline0:    move.w  d1, d2              *16ピクセルに満たない
92:          bra      next2                * 端数を描画
93: loop2:     move.w  d0, -(a1)          *
94:          swap.w    d0                  *COL1 ↔ COL2
95:          dbra      d2, loop2          *
96:          move.l    d7, d0              *
97:

```



```

98: next1:  swap.w  d0          #COL1 ←→ COL2
99:         lea.l   GNBYTE(a0), a0    #すぐ下のラインへ
100:        dbra    d3, loop1        #高さ分繰り返す
101:
102: done:   movem.l (sp)+, d0-d7/a0-a5
103:        unlk    a6
104:        rts
105:
106:        .end

```

基本的にはリスト12をベースに最小限の修正を加えたかたちになっている。データレジスタの上位ワードと下位ワードを交換するswap命令の使いかたがポイントだ。各swap命令の意味がつかめれば、プログラム全体の動作も見えてくると思う。試しに、swapを1つずつ削除してみて、実行結果がどう変わるか確かめてみるのもいいだろう。

C COLUMN

本書のサブルーチン共通仕様

この章以降で作成するサブルーチンは、基本的に以下の仕様を踏襲する。

- ・各サブルーチンはスーパーバイザモードで走行する
- ・引数はメモリ上に構造体様に用意し、その先頭アドレスをスタックに積んで渡す
- ・有効な座標値の範囲は符号付き16ビット数で表現できる、 $-32768 \sim +32767$ (例外はある)
- ・パレットコードが現在の画面モードで有効な値であるかどうかはチェックしない
- ・画面が初期化されているかどうかはチェックしない
- ・ソースレベルでは実画面512ピクセルモードと1024ピクセルモードのどちらでも動作するよう
に作成し、アセンブル時にどちらのモード用のオブジェクトを作成するか決定する(“/S”ス
イッチによりシンボル “_1024” が定義されていたら1024ピクセルモード用、そうでなければ
512ピクセルモード用)

また、この章で用意した次のモジュール、インクルードファイルは各章で繰り返し使用される。

```

GRAMADR.O
CLIPRECT.O
GCLIPRECT.O
GCONST.H
GMACRO.H
RECT.H

```

これら以外では、Chapter 3で作成する

```
GHLINE.O
```

も比較的ひんぱんに利用する。

C
H
A
P
T
E
R

必須のライブラリー

必須のラインルーチン

ASSEMBLER



この章では、線分描画ルーチンを作成する。線分描画のアルゴリズムは、多角形の塗り潰し (Chapter 4) や、グラデーション (Chapter 7)、グラフィックパターンの拡大・縮小 (Chapter 9)、回転 (Chapter A)、自由変形 (Chapter B) などにも利用される、応用範囲の広いアルゴリズムである。

線分描画のアルゴリズム

リスト1を見てもらいたい。線分 (0,0) - (5,3) をグラフィック画面に描くときに打つべき点の座標を計算し、表示するX-BASICプログラムだ。ラインルーチンの素朴な原型といえる。リスト1では、2点 (x0,y0), (x1,y1) を通る直線が、式

$$y = m(x - x_0) + y_0 \quad (\text{ただし } m = \frac{y_1 - y_0}{x_1 - x_0})$$

で与えられることを利用し、この式のxに0から5を代入して、対応するyの値を求めている。このプログラムをベースに、マシン語にしやすく、また、マシン語にしたときに実行効率がよくなるような線分発生アルゴリズムを導いていこう。

リスト1
LINE0.BAS

```
10 float x0 = 0, y0 = 0
20 float x1 = 5, y1 = 3
30 float m
40 int x, y, dx, sx, i
50 /*
60 dx = abs(x1-x0)
70 sx = sgn(x1-x0)
80 m = (y1-y0)/dx
90 x = x0
100 for i = 0 to dx
110   y = int(m * i + y0 + 0.5#)
120   print x, y
130   x = x + sx
140 next
```

と、その前に、リスト1は任意の座標を受けつけるようにはできていないことを断っておかねばなるまい。始点 (x0,y0) や終点 (x1,y1) をいろいろ変更してリスト1を走らせてみれ

ばすぐわかると思うが、 $x_0=x_1$ の場合は0での除算を生じてプログラムが止まる。また、傾きが急な線分を描かせようとすると、y座標がx座標に対して急激に変化するのでとびとびの座標しか得られず、点線になってしまう。この2つの問題は、傾きが1より小さい場合と大きい場合との処理を分けることで回避できる。つまり、リスト1のループとはxとyの関係を逆にしたループを別に用意し、

$$|x_1 - x_0| < |y_1 - y_0|$$

のときには、「x座標を変化させつつy座標を求めていく」代わりに、「y座標を変化させつつx座標を求めていく」わけだ（図1）。いまはアルゴリズムを導く途中だからこのまま放っておくが、最終的にはループを2つ用意することを頭に入れておいてほしい。

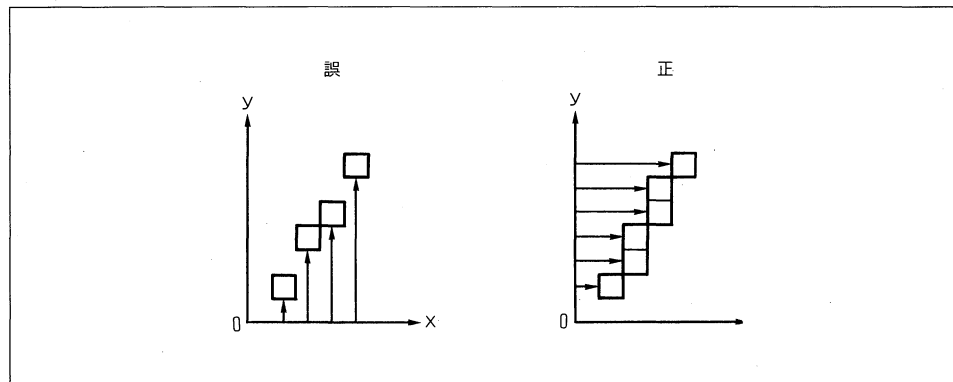


図1
傾きによる処理の振り分け

さて、しばらくのあいだ、

$$|x_1 - x_0| \geq |y_1 - y_0|$$

$$x_0 \leq x_1$$

$$y_0 \leq y_1$$

を仮定しよう。線分は右上がり、傾きは45°以下、これを左から右へと描いていく。ついでに、整数化もあと回しにする。マシン語にするうえでは、処理速度の点からも、実数を使わずに整数だけで計算するようにしたほうが好ましいのはいうまでもないが、整数化はアルゴリズムが固まってからでも遅くはない。

と、いつでもできることは先伸ばしにして思考の範囲を限定し、いま一度リスト1を眺めてみよう。速度低下の最大の元凶になりそうな乗算があるので、これを粛正する。リスト1ではループの中で毎回、式

$$y = m(x - x_0) + y_0$$

にxを代入してy座標を求めている。しかし、xが1増えるとyは傾きmずつ増えるわけだから、

$$y = y_0$$

と初期化しておいて、mを足し込んでいくだけで、yの値は順に求まる。リスト1はリスト2のように簡略化される。リスト2では、 $x_0 \leq x_1$, $y_0 \leq y_1$ の条件をつけたことによる省略も同時に施してあるが、混乱しないよう願いたい。

リスト2
LINE1.BAS

```
10 float x0 = 0, y0 = 0
20 float x1 = 5, y1 = 3
30 float m, fy
40 int x, y, dx, i
50 /#
60 dx = x1-x0
70 m = (y1-y0)/dx
80 x = x0 : fy = y0
90 for i = 0 to dx
100   y = int(fy + 0.5#)
110   print x, y
120   x = x + 1
130   fy = fy + m
140 next
```

ここで、実数で求めたyの小数点以下の値に注目する。最終的にはyは整数に丸められるから、yの小数部は、この丸めの際の四捨五入にのみ影響するわけだ。yの小数部は「2つの可能性のうち、どちらか真の線分に近い点を選択するスイッチ」として働いている。右上がりのなだらかな線分の場合、線分を描いていく途中、ある点 (x, y) に点を打ったあと、次に打つべき点は、

(x+1, y) (すぐ右)

(x+1, y+1) (右上)

のいずれかだ。真の線分がこの2点の中点より下を通るのであれば (x+1, y) を選択し、中点より上を通るのであれば (x+1, y+1) を選択することになる (図2)。いい換えると、実数で求めたyの小数部が $1/2$ より小さいか、大きいかでどちらの点を選ぶべきかが決まる。真の線分がきっかり中点を通る場合はどちらともいえない (どちらでもかまわない) のだが、四捨五入のイメージで、ここでは上の点 (x+1, y+1) を選ぶものとしよう。

少し整理してまとめてみる。実数で求めたyと、それを整数化した値との差を表す「誤差」という概念を導入し、Eとおく。Eは、いままで「yの小数部」と呼んでいたものと実質的には同じものだ。ただし、Eは $-1/2 \sim +1/2$ の値をとり、四捨五入して得られるy座標との差

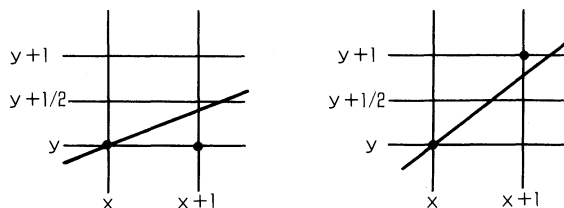


図2
四捨五入の意味

を表す (図3)。すると、右上がりのなだらかな線分は、次のような手順で描ける。

- 1) $x = x_0, y = y_0$ から線分を描き始める。この時点では、 (x, y) は真の線分のうえにあるから誤差はない。 $E = 0$ に初期化する。
- 2) 点 (x, y) に点を打つ。
- 3) 次の描画に備えて x に1を加える。
- 4) 誤差項 E に線分の傾き m (いまの条件下では $0 \leq m \leq 1$) を加える。その結果 E が $1/2$ 以上になったら、 (x, y) よりも $(x, y + 1)$ が真の線分に近いことを意味するので、 y に1を加える。同時に、 E が新たな y 座標との誤差を反映するように E から1を引く。
- 5) x が x_1 にたどり着くまで、2) ~ 4) を繰り返す。

この手順で線分 $(0, 0) - (5, 3)$ を描くときの様子を図4に示す。

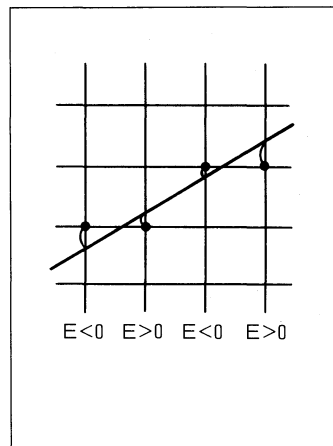


図3
誤差

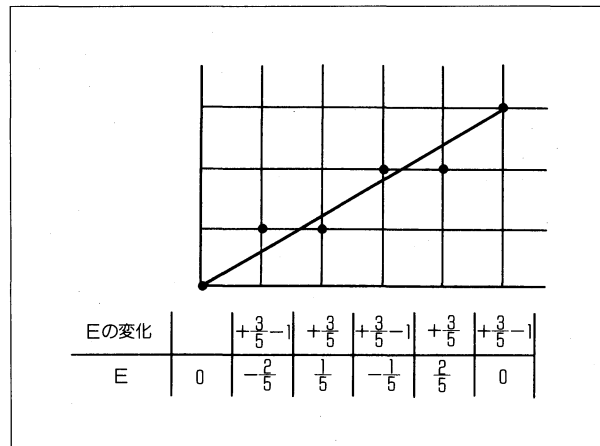


図4
Bresenhamの線分
発生アルゴリズム

リスト3が、このアルゴリズムをプログラムにしたものだ。ただし、 E の初期値を0ではなく $-1/2$ に変更し、 $1/2$ と比較する代わりに0との比較ですむように修正してある。

$$E < 1/2$$

という不等式の両辺から $1/2$ を引いて

$$E - 1/2 < 0$$

とし、この $E - 1/2$ を新たな E とおいたわけだ。X-BASIC ではこうする意味はないが、マシン語では0との比較はほかの値との比較よりも効率よく行える。

リスト3
LINE2.BAS

```
10 float x0 = 0, y0 = 0
20 float x1 = 5, y1 = 3
30 float m, e
40 int x, y, dx, i
```

```

50 /*
60 dx = x1-x0
70 m = (y1-y0)/(x1-x0)
80 e = -0.5#
90 x = x0 : y = y0
100 for i = 0 to dx
110   print x, y
120   x = x + 1
130   e = e + m
140   if (e >= 0) then {
150     y = y + 1
160     e = e - 1
170   }
180 next

```

こうして導いたアルゴリズムはBresenhamの線分発生アルゴリズムとして知られている。実用上は、このアルゴリズムを整数化したものが使われることになる。いまだ実数のまま残っている項はEとmだが、Eは符号だけが意味を持ち、mはそのEの計算に利用されるだけだ。したがって、E（の初期値）とm（線分の傾き=Eの増分）、および、1（y座標を更新したときにEから引く補正值）に適当な正の数をかけてもアルゴリズムは破綻しない。2 (x1-x0) をかければ、

Eの初期値	$-1/2$	$\rightarrow$	$-(x1-x0)$
Eの増分	$(y1-y0)/(x1-x0)$	$\rightarrow$	$2(y1-y0)$
Eの補正值	1	$\rightarrow$	$2(x1-y0)$

と、綺麗に整数化することができる。

リスト3を整数化し、ついでに線分の傾きや向きを考慮して一般化するとリスト4のようになった。整数の加減算と比較のみで（2倍は加算に置き換えて考える）、しかもグラフィック画面上に描きうるもっとも高い精度で線分を描くマシン語向きのアルゴリズムが導き出されたわけだ。

.....
リスト4
LINE3.BAS

```

10 int x0 = 0, y0 = 0
20 int x1 = 5, y1 = 3
30 int e, x, y
40 int dx, dy, sx, sy, i
50 /*
60 dx = abs(x1-x0) : dy = abs(y1-y0)
70 sx = sgn(x1-x0) : sy = sgn(y1-y0)
80 x = x0 : y = y0
90 if (dx >= dy) then {
100   e = -dx
110   for i = 0 to dx
120     print x, y
130     x = x + sx
140     e = e + 2*dy
150     if (e >= 0) then {
160       y = y + sy
170       e = e - 2*dx
180     }
190   next
200 } else {
210   e = -dy

```



```

220      for i = 0 to dy
230          print x, y
240          y = y + sy
250          e = e + 2*dx
260          if (e >= 0) then {
270              x = x + sx
280              e = e - 2*dy
290          }
300      next
310 }

```

ラインルーチンの原型

リスト5にアセンブリ言語版のライン描画サブルーチンを示そう。まだ高速化は考えず、リスト4のX-BASICプログラムを素直にアセンブリ言語に落としてみた。見くらべれば、ほとんどそのまんまだということがわかってもらえると思う。

リスト5
GLINES
(第1版)

```

1: #          線分描画 (整数化Bresenhamアルゴリズム)
2:
3:          .include      gconst. h
4:          .include      gmacro. h
5: #
6:          .xdef      gline
7:          .xref      gramadr
8:          .xref      gclipline
9: #
10:         .offset 0      #glineの引数構造
11: #
12: X0:      .ds. w      1      #始点座標
13: Y0:      .ds. w      1      #
14: X1:      .ds. w      1      #終点座標
15: Y1:      .ds. w      1      #
16: COL:     .ds. w      1      #描画色
17: #
18:         .text
19:         .even
20: #
21: gline:
22: ARGPTR   =          8
23:         link      a6, #0
24:         movem. l   d0-d7/a0-a5, -(sp)
25:
26:         movea. l   ARGPTR(a6), a5      #a5 = 引数列
27:         movem. w   (a5), d0-d3         #d0~d3に座標を取り出す
28:
29: #          jsr      gclipline          #クリッピングする
30: #          bne      done              #Z=0なら描画の必要なし
31:
32:         jsr      gramadr              #始点のG-RAM上アドレスを得る
33:
34:         sub. w     d0, d2              #d2 = x1-x0
35:         move. w    d2, d4              #d4 = d2
36:         ABS        d2                  #d2 = dx = abs(x1-x0)
37:         SGN        d4                  #d4 = sx = sgn(x1-x0)
38:
39:         sub. w     d1, d3              #d3 = y1-y0
40:         move. w    d3, d5              #d5 = d3
41:         ABS        d3                  #d3 = dy = abs(y1-y0)
42:         SGN        d5                  #d5 = sy = sgn(y1-y0)
43:
44:         add. w     d4, d4              #d4 = sx*2

```

```

45:      moveq.l #GSFCTR, d0      *
46:      asl.w   d0, d5           #d5 = sy*1024 (or 2048)
47:
48:      move.w  COL(a5), d0      #d0 = 描画色
49:
50:      cmp.w   d3, d2           #dy>dxならば
51:      bcs     yline           * yについてループ
52:
53:                                #dx≥dyのとき
54: xline:  move.w  d2, d1         #d1 = d2
55:         neg.w   d1            #d1 = e = -dx
56:         move.w  d2, d6         #d6 = n = dx
57:         add.w   d2, d2         #d2 = dx*2
58:         add.w   d3, d3         #d3 = dy*2
59:         do {
60: xline0: move.w  d0, (a0)       # pset(x, y)
61:         adda.w  d4, a0         # x += sx
62:         add.w   d3, d1         # e += 2*dy
63:         bmi    xline1         # if (e >= 0) {
64:         adda.w  d5, a0         #   y += sy
65:         sub.w   d2, d1         #   e -= 2*dx
66:         }
67: xline1: dbra    d6, xline0     #} while (--n >= 0)
68:         bra     done
69:                                #dx<dyのとき
70: yline:  move.w  d3, d1         #d1 = d3
71:         neg.w   d1            #d1 = e = -dy
72:         move.w  d3, d6         #d6 = n = dy
73:         add.w   d2, d2         #d2 = dx*2
74:         add.w   d3, d3         #d3 = dy*2
75:         do {
76: yline0: move.w  d0, (a0)       # pset(x, y)
77:         adda.w  d5, a0         # y += sy
78:         add.w   d2, d1         # e += 2*dx
79:         bmi    yline1         # if (e >= 0) {
80:         adda.w  d4, a0         #   x += sx
81:         sub.w   d3, d1         #   e -= 2*dy
82:         }
83: yline1: dbra    d6, yline0     #} while (--n >= 0)
84:
85: done:   movem.l (sp)+, d0-d7/a0-a5
86:         unlk    a6
87:         rts
88:
89:         .end

```

26～27行で引数受け渡し領域へのポインタをa1で受けて、さらに描くべき線分の座標をd0.w～d3.wに取り出す。実用的なラインルーチンにするためには、実画面外にアクセスしたりしないよう、このタイミングでクリッピングする必要があるが、クリッピングルーチンはまだ用意していない。29～30行はあとでクリッピングルーチンをつけ加えることを想定し、とりあえずコメントにしてある。

32行でサブルーチンgramadr (Chapter 2リスト1) で始点のG-RAMアドレスを計算し、34～42行では、インクルードファイルGMACRO.H (Chapter 2リスト7) で定義してあるマクロABSとSGNを使って、

dx = abs (x1 - x0)

sx = sgn (x1 - x0)

dy = abs (y1 - y0)

sy = sgn (y1 - y0)

を求めている。続く44～46行ではsxとsyを「G-RAM上のアドレスの差」に換算している。

「座標を増減してからG-RAMアドレスを計算し直す」代わりに、「座標の変化をG-RAMアドレスの相対的な変化に反映させる」ためのマシン語流の細工だ。

ここまで準備が整ったら、d0に描画色のパレットコードを取り出し、描く線分の傾き（dxとdyの大小関係）におうじて処理を振り分ける。基本的にどちらもやっていることは同じだから、傾きがなだらかな（ $dx \geq dy$ ）場合を処理する54行以下を見てもらおう。54～56行で誤差項Eとループカウンタを初期化してから、ループ中では定数となる2dx、2dyを先に計算している（57～58行）。60行からが実際に線分を描くループだ。点を1つ打ってから、x座標を進め、Eに2dyを加える（60～62行）。その結果Eが非負になったらy座標を進めて、Eから2dxを引く（64～65行）。この処理をdx+1回繰り返せば目的の線分が描かれる。

線分のクリッピング

線分がいちおう描けるようになったところで、クリッピングの処理をつけ加える。線分を任意の矩形領域でクリッピングする処理の考え方自体はそう難しくはない。線分の端がクリッピングウィンドウの外側にあったら、線分とウィンドウの辺との交点を求めて、ウィンドウ外の部分を切り捨てるというだけのことだ。ここでは、交点の求め方として、中点分割によるアルゴリズムを紹介する。

クリッピングの第1段階では、まず、線分の両端点がウィンドウの中にあるか（以後、「可視」と表現する）、外にあるか（「不可視」と表現する）、外だとすれば線分がウィンドウの辺のいずれと交わる「可能性がある」のかといったおおまかな情報を集める。明らかに、両端点が可視であれば、線分全体が可視であり、クリッピングは必要ない。端点のどちらか一方だけが不可視であれば、線分は部分的に不可視で、不可視の端点はウィンドウの辺でクリップする必要がある。また、両端点がともに不可視であれば、線分は完全に不可視であるか、部分的に不可視であるかのどちらかになる。

端点の位置を図5に示すような4ビットのコードで表すと、このあたりの分類をいくらか効率よく行うことができる（図5の中央の枠がクリッピングウィンドウ）。この4ビットコード（仮に端点の分類コードと呼んでおく）の各ビットはウィンドウの各辺（を延長した直線）に対応しており、その点がウィンドウの辺の内側にあるか外側にあるのかを0と1で示す。図5では、

第0ビット ———— ウィンドウの左辺

第1ビット ———— ウィンドウの右辺

第2ビット ———— ウィンドウの上辺

第3ビット ———— ウィンドウの下辺

のように対応させてある。

0101	0100	0110
0001	0000	0010
1001	1000	1010

図5
端点分類コード

分類コードが0000の端点は可視であり、クリッピングの必要はない。とくに、両端点の分類コードがともに0000の場合、線分は完全に可視になる。また、両端点の分類コードの論理積 (AND) が非0の場合、両端点はともに「同じ辺の不可視側にある」から、その線分は完全不可視になる。

以上の手順で完全可視の線分と、完全不可視の線分の一部をはじいてから、実際にウィンドウの辺と線分と交点を求める処理に入る。まだ完全不可視の線分がすべて除去できているわけではないので、交点が存在しない場合もあることに注意しよう。端点分類コードは同じでも、完全不可視の場合と部分的に不可視の場合がある (図6)。どちらになるかは実際に交点を求めてみる (求めようとしてみる) まではわからない。また、クリップする辺の順序によっては、最大2回 (x座標について1回, y座標について1回) のクリッピングをしないと正しい交点が求まらないこともありうる (図7)。1辺でクリップしたら、その点がウィンドウ内に収まっているかどうか、かならず確認しなければならない。

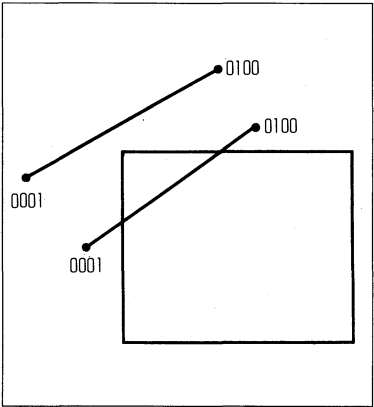


図6
端点分類コードでは区別できない例

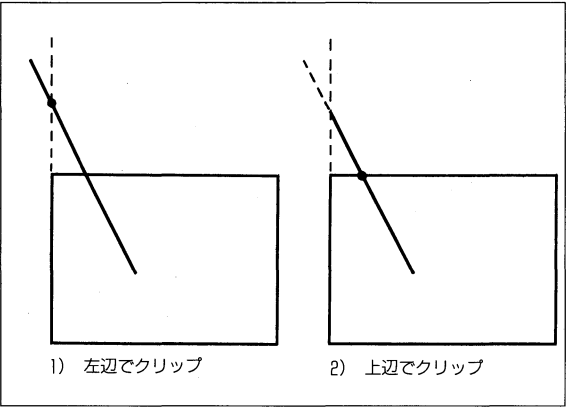


図7
2辺でのクリッピング

具体的な交点の算出に進もう。始点 (x0, y0) の分類コードは0001, 終点 (x1, y1) の分類コードは0000であるような線分 (x0, y0) - (x1, y1) をクリッピングする場合を考える。分類コードは (x0, y0) をウィンドウ左端でクリップする必要があることを示している。

中点分割アルゴリズムでは、線分の中点の座標 (mx, my) を求め、線分を

(x0, y0) - (mx, my)

(mx, my) - (x1, y1)

に2分する。中点の座標 (mx, my) はそれぞれ、

$mx = (x0 + x1) / 2$

$my = (y0 + y1) / 2$

により得られる。加算1回と、1ビットの算術右シフトだ。ウィンドウの左辺でクリップする場合、得られたmxと辺のx座標（以下、minxと呼ぶ）をくらべれば、辺との交点が2分した線分のどちらのうえにあるかがわかる。そうしたらその線分をさらに2等分して、得られた中点のx座標とminxをくらべる。1回分割するごとに探索範囲が半分になっていき、線分がこれ以上分割できなくなったら（つまり点になったら）、その点が求める交点になっている。座標が16ビットの符号付き数であれば、最大でも16回の分割作業で交点が見つかるし、途中で、偶然、中点のx座標がminxと一致すれば以下の分割は不用なので、この方法は見かけ以上に高速に動作する。

ただ、このアルゴリズムは2で割るときの端数の扱いによってはけっこう誤差が大きくなってしまうという欠点がある。最初、この誤差をなくそうと躍起になっていたのだが、IOCSのLINEも誤差には無頓着なようなので、今回は見逃した。後で示すプログラムでは端数をひたすら切り捨てる。したがって、得られる交点はウィンドウの左方向、上方向に何ピクセルかずれ込む傾向にある（IOCSのLINEも同様）。

さて、中点を求めるときには2つの座標値を足してから2で割るわけだが、座標値として任意の16ビットの符号付き数を許そうとすると、オーバーフローの問題が顔を出してくる。たとえば、8000_H (-32768) と8000_H (-32768) の平均を求めるときには、16ビット演算では8000_H どうしを足した時点でオーバーフローしてしまい正しい結果が得られない。もし、扱う数が符号なし数であれば、

add.w d2,d0

roxr.w #1,d0

のように右シフトのかわりにroxrを使うことで、「加算の結果発生したキャリ1ビットを含めた17ビットを右シフトする」のと同じ効果が得られるのだが、この方法も符号付き数には通用しない。考えたあげく、クリッピングルーチン内では座標値に8000_Hのゲタを履かせて、大小関係を保ったまま無符号数として扱うことにした。

8000_H ~ 0000_H ~ 7FFF_H (符号付き)

↓ ↓ ↓

0000_H ~ 8000_H ~ FFFF_H (符号なし)

のように、 $-32768 \sim +32767$ の範囲の座標を $0 \sim +65535$ に一時的にずらしておき、クリッピングがすんだらまた元に戻すわけだ。この考え方が正しいことを各自確かめてもらいたい。

ゲタを履かせてこれでもう安心だと思ったら、気持ちの悪い副作用が出てしまった。クリッピングウィンドウの座標にもゲタを履かせなければつじつまが合わない。といって、座標の比較のたびに、クリッピングウィンドウの座標値にゲタを履かせたり脱がせたりするのは効率が悪そうだ。

結局、しわ寄せをクリッピングウィンドウの設定ルーチンに持っていくことにした。ウィンドウを設定するときに、 8000_H のゲタを履かせた座標を同時にワークにしまっておく。じつは、前章で作成したクリッピングウィンドウ設定サブルーチンsetcliprectは、すでにそうするようになっている (Chapter 2リスト9:26~31行)。

では、リスト6にリスト5用のクリッピングルーチンを示す。同じような処理が何度も出てくる冗長なプログラムになってしまったが、アルゴリズムを理解するうえではかえってわかりやすいだろう。

リスト6
GCLIPLINE.S

```

1: *      線分の矩形領域クリッピング (中点分割)
2:
3:      .include      rect.h
4: *
5:      .xdef      gclipline
6:      .xref      cliprect
7:      .xref      ucliprect
8: *
9:      端点分類コード
10: *
11: LT_MINX_BIT      equ      0
12: GT_MAXX_BIT      equ      1
13: LT_MINY_BIT      equ      2
14: GT_MAXY_BIT      equ      3
15: *
16: LT_MINX          equ      1
17: GT_MAXX          equ      2
18: LT_MINY          equ      4
19: GT_MAXY          equ      8
20:
21: *
22: *      端点分類コードを求めるマクロ
23: *
24: CHKVIS      macro      XX, YY, CODE
25:      local      skip0, skip1, skip2, skip3
26:      movea.l      a0, a1
27:      moveq.l      #0, CODE
28:      cmp.w      (a1)+, XX
29:      bge      skip0
30:      addq.w      #LT_MINX, CODE
31: skip0:      cmp.w      (a1)+, YY
32:      bge      skip1
33:      addq.w      #LT_MINY, CODE
34: skip1:      cmp.w      (a1)+, XX
35:      ble      skip2
36:      addq.w      #GT_MAXX, CODE
37: skip2:      cmp.w      (a1)+, YY
38:      ble      skip3
39:      addq.w      #GT_MAXY, CODE
40: skip3:
41:      .endm
42: *

```

```

43: .text
44: .even
45: #
46: # 線分(d0.w, d1.w) - (d2.w, d3.w)を
47: # cliprectで指定された矩形領域でクリップする
48: # 完全不可視の場合はZ=0で戻る
49: gclipline:
50:     lea.l    cliprect, a0    #クリッピング範囲
51:     CHKVIS   d0, d1, d6      #d6 = 始点の分類コード
52:     CHKVIS   d2, d3, d7      #d7 = 終点の分類コード
53:
54:     add.w    d7, d6          #d6とd7が
55:     # ともに0であれば
56:     beq      done           # 線分は完全に可視
57:     sub.w    d7, d6          #
58:
59:     move.w    d6, d4          #d6とd7のandが
60:     and.w    d7, d4          # 非0であれば
61:     bne      done           # 線分は完全に不可視
62:
63:     lea.l    ucliprect, a0    #クリッピング範囲 (ゲタ履き)
64:     move.w    #$8000, d4      #$8000のゲタを履かせる
65:     add.w    d4, d0          #
66:     add.w    d4, d1          #
67:     add.w    d4, d2          #
68:     add.w    d4, d3          #
69:
70:     tst.w    d6              #始点は可視か?
71:     beq      skip1          # 可視ならクリッピング不要
72:     bsr      gcliplinesub    #始点をクリップ
73:     bne      done           #なおかつ不可視だった
74:
75: skip1: move.w    d7, d6          #終点は可視か?
76:     beq      skip2          # 可視ならクリッピング不要
77:     exg.l    d0, d2          #始点と終点を取り替える
78:     exg.l    d1, d3          #
79:     bsr      gcliplinesub    #終点をクリップ
80:     bne      done           #なおかつ不可視だった
81:     exg.l    d0, d2          #取り替えた始点と終点を
82:     exg.l    d1, d3          #元に戻す
83:
84: skip2: move.w    #$8000, d4      #ゲタを脱がせる
85:     sub.w    d4, d0          #
86:     sub.w    d4, d1          #
87:     sub.w    d4, d2          #
88:     sub.w    d4, d3          #
89:
90:     moveq.l   #0, d4          #Z = 1
91: done:  rts
92:
93: #
94: # 片方の端点(d0.w, d1.w)をクリップする
95: #
96: gcliplinesub:
97:     move.w    d0, a1          #クリッピング前の座標を
98:     move.w    d1, a2          # 覚えておく
99:     move.w    d2, a3          #
100:    move.w    d3, a4          #
101:    moveq.l   #2-1, d4          #クリッピング回数カウンタ
102:    move.w    d6, ccr          #----NZVC
103:    bcs      minxclip          #bit 0 = 1ならば左の辺にクリップ
104:    bvs      maxxclip          #bit 1 = 1ならば右の辺にクリップ
105:    beq      minyclip          #bit 2 = 1ならば上の辺にクリップ
106:    # bmi      maxyclip          #bit 3 = 1ならば下の辺にクリップ
107:
108: #
109: # MAXYでクリップする
110: #
111: maxyclip: # 1 回目のエントリ
112:     move.w    MAXY(a0), d5
113:     cmp.w     d5, d3
114:     bne      maxy1p
115:     move.w    d2, d0
116:     move.w    d3, d1

```

```

117:      bra      yclipn
118: maxylp: move.w d1, d6
119:      add.w    d3, d6
120:      roxr.w   #1, d6
121:      cmp.w    d5, d6
122:      beq      yclipq
123:      bcs      maxy0
124:      move.w   d6, d1
125:      add.w    d2, d0
126:      roxr.w   #1, d0
127:      bra      maxylp
128: maxy0: move.w d6, d3
129:      add.w    d0, d2
130:      roxr.w   #1, d2
131:      bra      maxylp
132: maxyclip2:      *2回目以降のエントリ
133:      subq.w   #1, d4
134:      bcs      outofscrn
135:      cmp.w    MAXY(a0), d3
136:      bhi      outofscrn
137:      move.w   a1, d0
138:      move.w   a2, d1
139:      bra      maxyclip
140: outofscrn:
141:      moveq.l  #-1, d4      *Z = 0
142:      rts
143:
144: *
145: *      MINYでクリップする
146: *
147: minyclip2:      *2回目以降のエントリ
148:      subq.w   #1, d4
149:      bcs      outofscrn
150:      cmp.w    MINY(a0), d3
151:      bcs      outofscrn
152:      move.w   a1, d0
153:      move.w   a2, d1
154: minyclip:      *1回目のエントリ
155:      move.w   MINY(a0), d5
156:      cmp.w    d5, d3
157:      bne      minylp
158:      move.w   d2, d0
159:      move.w   d3, d1
160:      bra      yclipn
161: minylp: move.w d1, d6
162:      add.w    d3, d6
163:      roxr.w   #1, d6
164:      cmp.w    d5, d6
165:      beq      yclipq
166:      bcs      miny0
167:      move.w   d6, d3
168:      add.w    d0, d2
169:      roxr.w   #1, d2
170:      bra      minylp
171: miny0: move.w d6, d1
172:      add.w    d2, d0
173:      roxr.w   #1, d0
174:      bra      minylp
175:
176: yclipq: move.w d6, d1
177:      add.w    d2, d0
178:      roxr.w   #1, d0
179:      move.w   a3, d2
180:      move.w   a4, d3
181: yclipn: cmp.w  MINX(a0), d0
182:      bcs      minxclip2
183:      cmp.w    MAXX(a0), d0
184:      bhi      maxxclip2
185:      moveq.l  #0, d4      *Z = 1
186:      rts
187:
188: *
189: *      MAXXでクリップする
190: *

```



```

191: maxxclip2:                #2回目以降のエントリ
192:     subq.w  #1, d4
193:     bcs     outofscrn
194:     cmp.w   MAXX(a0), d2
195:     bhi     outofscrn
196:     move.w  a1, d0
197:     move.w  a2, d1
198: maxxclip:                  #1回目のエントリ
199:     move.w  MAXX(a0), d5
200:     cmp.w   d5, d2
201:     bne     maxxlp
202:     move.w  d2, d0
203:     move.w  d3, d1
204:     bra     xclipn
205: maxxlp: move.w  d0, d6
206:     add.w   d2, d6
207:     roxr.w  #1, d6
208:     cmp.w   d5, d6
209:     beq     xclipq
210:     bcs     maxx0
211:     move.w  d6, d0
212:     add.w   d3, d1
213:     roxr.w  #1, d1
214:     bra     maxxlp
215: maxx0: move.w  d6, d2
216:     add.w   d1, d3
217:     roxr.w  #1, d3
218:     bra     maxxlp
219:
220: *
221: *      MINXでクリップする
222: *
223: minxclip2:                  #2回目以降のエントリ
224:     subq.w  #1, d4           #2辺でクリッピング済みなら
225:     bcs     outofscrn       # 完全不可視だった
226:     cmp.w   MINX(a0), d2    #x1 < MINXなら
227:     bcs     outofscrn       # 完全不可視
228:     move.w  a1, d0          #クリッピング前の座標から
229:     move.w  a2, d1          # やり直す
230:
231: minxclip:                   #1回目のエントリ
232:     move.w  MINX(a0), d5     #d5 = クリップするx座標
233:     cmp.w   d5, d2          #x1 = MINXなら
234:     bne     minxlp          #
235:     move.w  d2, d0          # その点(x1, y1)が求める点
236:     move.w  d3, d1          #
237:     bra     xclipn
238: minxlp: move.w  d0, d6       #
239:     add.w   d2, d6          #
240:     roxr.w  #1, d6          #d6 = Mx = (x0+x1)/2
241:     cmp.w   d5, d6          #Mx = MINXなら
242:     beq     xclipq          # それを求める点
243:     bcs     minx0
244:
245:     #Mx > MINXなら
246:     move.w  d6, d2          #x1 = Mx
247:     add.w   d1, d3          #
248:     roxr.w  #1, d3          #y1 = My
249:     bra     minxlp          # として繰り返す
250:
251:     #Mx < MINXなら
252:     move.w  d6, d0          #x0 = Mx
253:     add.w   d3, d1          #
254:     roxr.w  #1, d1          #y0 = My
255:     bra     minxlp          # として繰り返す
256:
257: xclipq: move.w  d6, d0       #x0 = Mx
258:     add.w   d3, d1          #
259:     roxr.w  #1, d1          #y0 = My
260:     move.w  a3, d2          #x1を復帰
261:     move.w  a4, d3          #y1を復帰
262:
263: xclipn: cmp.w   MINY(a0), d1
264:     bcs     minyclip2
265:     cmp.w   MAXY(a0), d1
266:     bhi     maxyclip2
267:     moveq.l #0, d4          #Z = 1

```

```

265:      rts
266:
267:      .end

```

50行以下では、両端点の分類コードを求め、54～56行で完全可視の場合をはじく。端点分類コードは24～41行で定義しているマクロを使って求めている。続いて、59～61行で両端点の分類コードのANDをとり、結果が非0の場合をはじいている。なお、このサブルーチンは完全不可視だった場合にはccrのZビットを0にして戻るようにつくってある。64～68行で座標に8000_Hのゲタを履かせてから、始点と終点を下位サブルーチンで順にクリッピングしていく（70～82行）。最終的に両端点が可視になったら、ゲタを脱がせて（84～88行）Z=1で（90行）戻る。

96行からの下位ルーチンがクリッピング処理の本体だ。最初にクリッピング前の座標を待避しておく（97～100行）。レジスタが足りないので、アドレスレジスタを動員している。続く101行では、間違って2辺以上でクリッピングしてしまわないよう、クリッピングした辺のカウンタを用意する（直後に一度クリッピングすることがわかっているから、初期値は最初から1引いてある）。クリップした辺はちゃんと数えておかないと、完全不可視の線分の場合、どの辺でクリップしても可視にならないので、クリッピングルーチン内をうろつき回ることになる。

その次の、102行のmoveの使い方はあまり見慣れないかたちだと思う。d6をccrに代入している¹⁾。d6には端点の分類コードが入っており、分類コードの各ビットが、

第0ビットが1ならば、C=1

第1ビットが1ならば、V=1

第2ビットが1ならば、Z=1

第3ビットが1ならば、N=1

のように、そのままccrの各ビットに反映される。後は103～105行の条件分岐命令により、各辺ごとのクリッピングルーチンに分岐するという寸法だ。ちょっとトリッキーだが、いちいち各ビットが0か1かをbtstやlsrなどで調べるよりはスマートだろう。

■ 1) ccrはバイト長のレジスタだが、

```
move.b d6,ccr
```

という命令はない。ccrへの転送は、つねに

```
move.w d6,ccr
```

のようにワードサイズを使う。この場合、ワードの上位バイトはどこにも送られず、たんに捨てられる。ちなみに、ccrに対して即値との論理演算を行う

```
andi.b #n,ccr
```

```
ori.b #n,ccr
```

```
eori.b #n,ccr
```

という命令もあるが、これらにはバイトサイズの命令しかない。

以下、各辺ごとのクリッピングルーチンが並ぶ。どれもほとんど同じ処理をしているので、代表として、223行からの、ウィンドウ左辺でクリップする場合の処理を見ていこう。

すでにクリッピングした辺の数におうじて、エントリが2つある。一度もクリッピングされていないときは231行に飛び込み、すでにほかの辺でクリッピングされているときは223行に飛んできて、完全不可視と判断できる場合をはじめてから合流する。

232～237行のあたりの処理は不要に見えるかもしれない。いまクリッピングしようとしているのは (d0, d1) で表される点なのに、もう一方の端点である (d2, d3) がクリッピングする辺上にないかどうかを調べている。この処理の必要性は、次の例で納得してもらいたい。

2ピクセルからなる線分 $(-1, 0) - (0, 0)$ を実画面の範囲でクリッピングするとしよう。 $(-1, 0)$ の点を $x=0$ の辺でクリップするために中点分割を試みる。2端点の中点は、

$$mx = (-1+0) / 2$$

$$my = (0+0) / 2$$

だ (実際にはゲタ履きで計算する)。ここで $mx = -1/2$ は整数除算では値が小さくなるように切り捨てが起こって

$$-1/2 = -1$$

になる。 $(-1+0) / 2 = -1$ では、いつまで中点分割を繰り返しても収束してくれない。結果、無限ループに陥ってしまう。232～237行はこの事態を避けるためだけにある。

238～253行が中点分割を繰り返すループだ。ここは見てのとおりのをやっている。交点が見つかったら255行に飛んで、求めた中点の座標を d0, d1 に入れ (255～257行)、演算の途中で値が変わってしまった d2, d3 に処理前の値を復帰する (258～259行)。それから求めた座標が可視かどうかを調べ、可視であればクリッピングは完了したから $Z=1$ で戻り、さもなくば、ウィンドウの上辺か下辺でクリッピングし直す (260～263行)。いまウィンドウの左辺でクリッピングしたばかりだから、ウィンドウの右辺は気にしなくてよい。

そして高速化

なんとかクリッピングもできるようになった。ぼちぼちとラインルーチン本体の高速化にとりかろう。まずは、コーディング上の工夫、細部の最適化でどのくらい変わるか見てもらおうと思う。リスト7だ。リスト7では、比較的ひんぱんに利用されるであろう水平、垂直、ついでに45度の線分を特別扱いして、専用ループで高速に処理している (97～111行)。また、ポストインクリメントがうまく使えるよう、あらかじめ $x_0 \leq x_1$ を保証するようにした (30～33行)、マクロを使っていたあたりを展開してすっきりさせたり (37～42行) といった贅肉取りが施されている。link～unlkも削ってしまった。

リスト7
GLINES
(第2版)

```
1: *      線分描画 (整数化Bresenham+若干の最適化)
2:
3:      .include      gconst.h
4: *
```

```

5:      .xdef    gline
6:      .xref    gramadr
7:      .xref    gclipline
8:      *
9:      .offset 0      *glineの引数構造
10:     *
11: X0:      .ds.w 1      *始点座標
12: Y0:      .ds.w 1      *
13: X1:      .ds.w 1      *終点座標
14: Y1:      .ds.w 1      *
15: COL:     .ds.w 1      *描画色
16:     *
17:      .text
18:      .even
19:     *
20: gline:
21: ARGPTR = 4+8*4+6*4
22: movem.l d0-d7/a0-a5, -(sp)
23:
24: movea.l ARGPTR(sp), a5      *a5 = 引数列
25: movem.w (a5)+, d0-d3      *d0~d3に座標を取り出す
26:
27:      jsr      gclipline      *クリッピングする
28:      bne      done          *Z=0なら完全不可視
29:
30:      cmp.w    d2, d0          *x0 ≤ x1を保証する
31:      ble      gline0        *
32:      exg.l    d0, d2          *
33:      exg.l    d1, d3          *
34:
35: gline0: jsr      gramadr      *始点のG-RAMアドレスを得る
36:
37:      move.w   #GNBYTE, d5     *d5 = 横1ライン分のバイト数
38:      sub.w    d1, d3          *d3 = y1-y0
39:      beq      hor_line       *y0=y1なら水平線
40:      bpl      gline1
41:      neg.w    d3
42:      neg.w    d5
43: gline1: sub.w   d0, d2          *d2 = x1-x0 (>=0)
44:      beq      ver_line       *x0=x1なら垂直線
45: *この時点で
46: *      d2 = dx = abs(x1-x0) (> 0)
47: *      d3 = dy = abs(y1-y0) (> 0)
48: *      d5 = sy = sgn(y1-y0) (-1 or 1)
49: *      (ただしd5はGNBYTE倍済み)
50:
51:      move.w   (a5), d0        *d0 = 描画色
52:
53:      cmp.w    d3, d2          *dy > dxならば
54:      bcs      yline          * yについてループ
55:      beq      xylene         *dy = dxならば45度の線
56:
57:      *dx ≥ dyのとき
58: xline: move.w   d2, d1          *d1 = dx
59:      neg.w    d1              *d1 = e = -dx
60:      move.w   d2, d6          *d6 = n = dx
61:      add.w    d2, d2          *d2 = dx*2
62:      add.w    d3, d3          *d3 = dy*2
63:      *do {
64: xline0: move.w   d0, (a0)+      * pset(x++, y)
65:      add.w    d3, d1          * e += 2*dy
66:      bmi      xline1          * if (e >= 0) {
67:      adda.w   d5, a0          * y += sy
68:      sub.w    d2, d1          * e -= 2*dx
69:      * }
70: xline1: dbra     d6, xline0     *} while (--n >= 0)
71:      bra      done
72:
73:      *dx < dyのとき
74: yline: move.w   d3, d1          *d1 = dy
75:      neg.w    d1              *d1 = e = -dy
76:      move.w   d3, d6          *d6 = n = dy
77:      add.w    d2, d2          *d2 = dx*2
78:      add.w    d3, d3          *d3 = dy*2

```

```

79:                                     *do {
80: yline0: add.w  d2, d1                * e += 2*dx
81:         bpl    yline1                * if (e < 0) {
82:         move.w  d0, (a0)              *   pset(x, y)
83:         adda.w  d5, a0                *   y += sy
84:                                     * }
85:         dbra   d6, yline0
86:         bra    done
87:                                     * else {
88: yline1: move.w  d0, (a0) +            *   pset(x++, y)
89:         adda.w  d5, a0                *   y += sy
90:         sub.w   d3, d1                *   e -= 2*dy
91:                                     * }
92:         dbra   d6, yline0            *} while (--n >= 0)
93:
94: done:  movem.l  (sp)+, d0-d7/a0-a5
95:        rts
96: *
97: hor_line:                *水平線分
98:         sub.w   d0, d2                *d2 = dx = x1-x0
99:         move.w  (a5), d0              *d0 = 描画色
100: hloop: move.w   d0, (a0) +            *pset(x++, y)
101:         dbra   d2, hloop
102:         bra    done
103:
104: xyline:                  *45度の線分
105:         addq.w  #2, d5                *d5 = 2±GNBYTE
106: ver_line:                *垂直線分
107:         move.w  (a5), d0              *d0 = 描画色
108: vloop:  move.w  d0, (a0)              *pset(x, y)
109:         adda.w  d5, a0                *y += sx
110:         dbra   d3, vloop              *dy+1回繰り返す
111:         bra    done
112:
113: .end

```

もうひと工夫しよう。線分が基本的に上下左右対象であることに目をつける。両端から同時に描けば、誤差項の計算回数、ループ回数が半分に抑えられるだろう。実例をリスト8に示す。

リスト8
GLINES
(第3版)

```

1: *      線分描画 (整数化Bresenham+両端描画)
2:
3:        .include      gconst.h
4: *
5:        .xdef    gline
6:        .xref    gclipline
7:        .xref    gbase
8: *
9:        .offset 0      *glineの引数構造
10: *
11: X0:    .ds.w  1      *始点座標
12: Y0:    .ds.w  1      *
13: X1:    .ds.w  1      *終点座標
14: Y1:    .ds.w  1      *
15: COL:   .ds.w  1      *描画色
16: *
17:        .text
18:        .even
19: *
20: gline:
21: ARGPTR =      4+8*4+6*4
22: movem.l  d0-d7/a0-a5, -(sp)
23:
24: movea.l  ARGPTR(sp), a5      *a5 = 引数列
25: movem.w  (a5)+, d0-d3        *d0~d3に座標を取り出す
26:

```

```

27:      jsr      gclipline      *クリッピングする
28:      bne      done           *Z=0なら完全不可視
29:
30:      cmp.w     d2, d0         *x0 ≤ x1を保証する
31:      ble      gline0        *
32:      exg.l     d0, d2         *
33:      exg.l     d1, d3         *
34:
35: gline0: move.w  d1, d6         *始点/終点のG-RAMアドレスを求める
36:      move.w    d3, d7         *
37:      ext.l     d6             *
38:      ext.l     d7             *
39:      *
40:      moveq.l   #GSFCTR, d4    *
41:      asl.l     d4, d6         *
42:      asl.l     d4, d7         *
43:      add.w     d0, d6         *
44:      add.w     d0, d6         *
45:      add.w     d2, d7         *
46:      add.w     d2, d7         *
47:      *
48:      movea.l   gbase, a0      *
49:      movea.l   a0, a2         *
50:      add.l     d6, a0         *a0 = 始点のG-RAMアドレス
51:      add.l     d7, a2         *a2 = 終点のG-RAMアドレス
52:
53:      move.w    #GNBYTE, d5    *d5 = 横1ライン分のバイト数
54:      sub.w     d1, d3         *d3 = y1-y0
55:      beq       hor_line       *y0=y1なら水平線
56:      bpl       gline1         *
57:      neg.w     d3             *
58:      neg.w     d5             *
59: gline1: sub.w   d0, d2         *d2 = x1-x0 (>=0)
60:      beq       ver_line       *x0=x1なら垂直線
61: *この時点で
62: *   d2 = dx = abs(x1-x0) (> 0)
63: *   d3 = dy = abs(y1-y0) (> 0)
64: *   d5 = sy = sgn(y1-y0) (-1 or 1)
65: *   (ただしd5はGNBYTE倍済み)
66:
67:      move.w    (a5), d0        *d0 = 描画色
68:
69:      cmp.w     d3, d2         *dy > dxならば
70:      bcs       yline          * yについてループ
71:      beq       xyline         *dy = dxならば45度の線
72:
73:      *dx ≥ dyのとき
74: xline: move.w  d2, d1         *d1 = dx
75:      neg.w     d1             *d1 = e = -dx
76:      move.w    d2, d6         *d6 = n = dx
77:      add.w     d2, d2         *d2 = 2*dx
78:      add.w     d3, d3         *d3 = 2*dy
79:      subq.w    #1, d6         *dbraの動作を計算に入れて
80:      lsr.w     #1, d6         * ループカウンタを半減
81:      scs.b     d4             *奇数ピクセルのとき非0
82:      addq.l    #2, a2         *プリデクリメントする分補正
83:      *do {
84: xline0: move.w d0, (a0)+      * pset(x++, y)
85:      move.w    d0, -(a2)      * pset(--x, y)
86:      add.w     d3, d1         * e += 2*dy
87:      bmi       xline1         * if (e >= 0) {
88:      adda.w    d5, a0         *   y += sy
89:      suba.w    d5, a2         *   y -= sy
90:      sub.w     d2, d1         *   e -= 2*dx
91:      * }
92: xline1: dbra   d6, xline0     *} while (--n >= 0)
93:
94:      tst.b     d4             *奇数ピクセル?
95:      beq       done          * そうじゃない
96:      bra       odd           *中央のピクセルを点灯
97:
98:      *dx < dyのとき
99: yline: move.w  d3, d1         *d1 = dy
100:      neg.w     d1             *d1 = e = -dy

```

```

101:      move.w  d3, d6      #d6 = n = dy
102:      add.w   d2, d2      #d2 = 2*dx
103:      add.w   d3, d3      #d3 = 2*dy
104:      subq.w  #1, d6      #dbraのことを計算に入れて
105:      lsr.w   #1, d6      # ループカウンタを半減
106:      scs.b   d4          #奇数ピクセルのとき非0
107:      move.w  d5, d7      #d7 = d5 + 2
108:      addq.w  #2, d7      #
109:      do {
110: yline0: add.w  d2, d1      # e += 2*dx
111:      bpl     yline1      # if (e < 0) {
112:      move.w  d0, (a0)      #   pset(x, y)
113:      adda.w  d5, a0        #   y += sy
114:      move.w  d0, (a2)      #   pset(x', y')
115:      suba.w  d5, a2        #   y' -= sy
116:      }
117:      dbra    d6, yline0
118:      bra     done0
119:      else {
120: yline1: move.w  d0, (a0) +  #   pset(x++, y)
121:      adda.w  d5, a0        #   y += sy
122:      move.w  d0, (a2)      #   pset(x', y')
123:      suba.w  d7, a2        #   x' --, y' -= sy
124:      sub.w   d3, d1        #   e -= 2*dy
125:      }
126:      dbra    d6, yline0    #} while (--n >= 0)
127:
128: done0:  tst.b   d4          #奇数ピクセル?
129:      beq     done         # そうじゃない
130: odd:    move.w  d0, (a0)    #中央のピクセルを点灯
131:
132: done:   movem.l (sp)+, d0-d7/a0-a5
133:      rts
134: #
135: hor_line:      #水平線分
136:      sub.w   d0, d2        #d2 = dx = x1-x0
137:      move.w  (a5), d0      #d0 = 描画色
138: hloop:  move.w  d0, (a0) +  #pset(x++, y)
139:      dbra    d2, hloop
140:      bra     done
141:
142: xyline:      #45度の線分
143:      addq.w  #2, d5        #d5 = 2 ±GNBYTE
144: ver_line:      #垂直線分
145:      move.w  (a5), d0      #d0 = 描画色
146: vloop:  move.w  d0, (a0)    #pset(x, y)
147:      adda.w  d5, a0        #y += sx
148:      dbra    d3, vloop     #dy+1回繰り返す
149:      bra     done
150:
151:      .end

```

おもな修正点は、始点と同時に終点のG-RAMアドレスをa2に求めるようになったこと(35～51行)、両端の2点を平行して処理するように84～92行や110～126行のループを修正したこと、それにあわせ、ループカウンタを半減したことだ(79～80行, 104～105行)。あまりやりたくはなかったのだが、G-RAMアドレスの算出部は、効率にこだわって、前章でせっかくなつくたサブルーチンgramadrを使うのをやめ、その場で計算するようにした。

ループカウンタを半減する部分では、線分が奇数ピクセルだった場合に備えて、微妙なつづつまあわせをしている。d4.bは線分の長さが奇数ピクセルかどうかを覚えておくフラグとして使われており、ループから抜けたあと、まだ中央の1ピクセルが残っているかどうかの判断材料になる。

scs.b d4

は条件セット命令で、Cフラグが1ならd4.bをFF_Hにし、Cフラグが0ならd4.bを00_Hにすることを思い出してほしい。

ところで、半端になった中央のピクセルを特別扱いするのをやめれば、プログラムはもう少しすっきりする。79行や104行で1引いている部分を削り、ループカウンタを1つ大きくして、中央のピクセルだけはダブって描画するわけだ。そうすると、d4.bをフラグにして云々というあたりの余計な処理はみんないらなくなるから、多少は実行速度も上がるかもしれない。しかし、その場合、線分の傾きによっては中央だけ線分が太くなってしまうという副作用が見られる（四捨五入の不对称さを考えよう）。また、このラインルーチンを改造してXORモード対応にしたときにも問題が生じる。(a XOR b) XOR b = aだから、XORモードで同位置に2度点を打つと、元に戻ってしまうのだ。

さて、終点側から点を打っていく処理は、始点側から点を打つこれまでの処理とポインタの移動方向を反対にすれば実現できる。この周辺では例によって細部を微調整して処理効率を稼いでいる。傾きが緩やかな線分を描く84行以下のループでは、85行でポインタをプリデクリメントできるよう、あらかじめ82行でa2に2を足してあるのがそれだ。また、急な線分を描く110行以下のループでは、本来a2を「ポストデクリメント」してからd5.wを引くところを、d7.wにd5.w + 2を先に求めておくことで123行のように簡略化している。

さて、リスト8を眺めていると、水平、垂直、45度の線分それぞれの専用ループをついつい展開してみたくなる。展開したのがリスト9だ。リスト8の後半部、134行以下と差し替えて使う。水平線分描画部分はmove.lによる2ピクセル同時書き込みによる露骨なループの完全展開にしてみた。展開部分は以下の章でも利用するので別ファイルにしてある（リスト10）。水平線分を高速に描画する方法としては、前章でmovemを使う方法を紹介してあるわけだが、moveを並べた単純な方法のほうが何かと扱いやすかったのだからこちらを採用した。

.....
リスト9
GLINE.S
(水平/垂直/45° 線分
描画部：リスト8に対
する追加・変更部)

```

134: *
135:         .xref    ghline
136:         .xref    gvline
137: *
138: hor_line:      *水平線分
139:         sub.w    d0, d2          *d2 = dx = x1-x0
140:         move.w   (a5), d1        *d0.l = 描画色 (上位/下位とも)
141:         move.w   d1, d0          *
142:         swap.w   d0              *
143:         move.w   d1, d0          *
144:         addq.w   #1, d2          *d2 = dx+1 = ピクセル数
145:         bclr.l   #0, d2          *
146:         beq      hskip          *
147:         move.w   d0, (a0)+        *奇数ピクセルの分
148: hskip: lea.l    ghline, a1
149:         suba.w   d2, a1
150:         jsr      (a1)
151: hline:  bra      done
152:
153: xyline:      *45度の線分
154:         move.w   (a5), d0        *d0 = color
155:         addq.w   #1, d3          *d3 = dy+1
156:         moveq.l   #8-1, d4        *
157:         and.w    d3, d4          *d4 = 8ピクセル未満の端数

```



```

158:      lsr.w  #3, d3      *d3 = (dy+1)/8
159:      lsl.w  #2, d4      *moveとaddaで4バイト
160:      neg.w  d4          *
161:      jmp    xynext(pc, d4) *ループの途中に飛び込む
162:
163: xyloop: move.w  d0, (a0) +    *pset(x++, y)
164:      adda.w  d5, a0          *y += sy
165:      move.w  d0, (a0) +    *pset(x++, y)
166:      adda.w  d5, a0          *y += sy
167:      move.w  d0, (a0) +    *pset(x++, y)
168:      adda.w  d5, a0          *y += sy
169:      move.w  d0, (a0) +    *pset(x++, y)
170:      adda.w  d5, a0          *y += sy
171:      move.w  d0, (a0) +    *pset(x++, y)
172:      adda.w  d5, a0          *y += sy
173:      move.w  d0, (a0) +    *pset(x++, y)
174:      adda.w  d5, a0          *y += sy
175:      move.w  d0, (a0) +    *pset(x++, y)
176:      adda.w  d5, a0          *y += sy
177:      move.w  d0, (a0) +    *pset(x++, y)
178:      adda.w  d5, a0          *y += sy
179: xynext: dbra    d3, xyloop    *(dy+1)/8回繰り返す
180:      bra     done
181:
182: ver_line:      *垂直線分
183:      tst.w  d5          *d5 = sy
184:      bpl    vskip      *a0 <= a2を
185:      movea.l a2, a0      * 保証する
186: vskip: move.l  #$0000_8000, d5 *long!
187: .ifdef _1024
188:      moveq.l #16-1, d1    *d1 = dy%16
189:      and.w  d3, d1        *
190:      lsr.w  #4, d3        *d3 = dy/16
191: .else
192:      moveq.l #32-1, d1    *d1 = dy%32
193:      and.w  d3, d1        *
194:      lsr.w  #5, d3        *d3 = dy/32
195: .endif
196:      move.w  d1, d2
197:      moveq.l #GSFTCTR, d0
198:      lsl.w  d0, d2
199:      adda.w  d2, a0
200:      move.w  (a5), d0      *d0 = color
201:      addq.w  #1, d1
202:      lsl.w  #2, d1
203:      lea.l  gvline, a1
204:      suba.w  d1, a1
205:      jsr    (a1)
206:      bra     done
207:
208:      .end

```

リスト10
GHLINE.S

```

1: *      水平線分描画ループ展開部
2:
3:      .include      gconst.h
4: *
5:      .xdef      ghline
6: *
7:      .text
8:      .even
9: *
10:     .dcb.w  GNPPIXEL/2, $20c0 *move.l d0, (a0) +
11: ghline: rts
12:
13:     .end

```

45度の線分用のループは8倍展開で、ループ1回につき8ピクセルずつ描画する。8ピクセルに満たない端数のピクセルはループの途中に飛び込むことで描く。垂直の線分を描くループは、ディスプレイメントつきアドレスレジスタ間接形式適用のmove命令をずらっと32個（実画面512ピクセルモード時。1024ピクセルモード時は16個）並べた、32倍（あるいは16倍）展開となっている。ループ1回につき長さ32ピクセル（あるいは16ピクセル）の垂直線分を描くわけだ。展開部は、やはり別ファイル、リスト11に抜き出してある。

リスト11
GVLINE.S

```

1: #      垂直線分描画ループ展開部
2:
3:      .include      gconst.h
4: #
5:      .xdef      gvline
6: #
7:      .text
8:      .even
9: #
10: .ifdef      _1024
11: vloop: move.w    d0, -$7800(a0)    *長さ16の
12:          move.w    d0, -$7000(a0)    * 垂直線分を描く
13:          move.w    d0, -$6800(a0)
14:          move.w    d0, -$6000(a0)
15:          move.w    d0, -$5800(a0)
16:          move.w    d0, -$5000(a0)
17:          move.w    d0, -$4800(a0)
18:          move.w    d0, -$4000(a0)
19:          move.w    d0, -$3800(a0)
20:          move.w    d0, -$3000(a0)
21:          move.w    d0, -$2800(a0)
22:          move.w    d0, -$2000(a0)
23:          move.w    d0, -$1800(a0)
24:          move.w    d0, -$1000(a0)
25:          move.w    d0, -$0800(a0)
26:          move.w    d0, (a0)
27:          adda.l    d5, a0
28: .else
29: vloop: move.w    d0, -$7c00(a0)    *長さ32の
30:          move.w    d0, -$7800(a0)    * 垂直線分を描く
31:          move.w    d0, -$7400(a0)
32:          move.w    d0, -$7000(a0)
33:          move.w    d0, -$6c00(a0)
34:          move.w    d0, -$6800(a0)
35:          move.w    d0, -$6400(a0)
36:          move.w    d0, -$6000(a0)
37:          move.w    d0, -$5c00(a0)
38:          move.w    d0, -$5800(a0)
39:          move.w    d0, -$5400(a0)
40:          move.w    d0, -$5000(a0)
41:          move.w    d0, -$4c00(a0)
42:          move.w    d0, -$4800(a0)
43:          move.w    d0, -$4400(a0)
44:          move.w    d0, -$4000(a0)
45:          move.w    d0, -$3c00(a0)
46:          move.w    d0, -$3800(a0)
47:          move.w    d0, -$3400(a0)
48:          move.w    d0, -$3000(a0)
49:          move.w    d0, -$2c00(a0)
50:          move.w    d0, -$2800(a0)
51:          move.w    d0, -$2400(a0)
52:          move.w    d0, -$2000(a0)
53:          move.w    d0, -$1c00(a0)
54:          move.w    d0, -$1800(a0)
55:          move.w    d0, -$1400(a0)
56:          move.w    d0, -$1000(a0)
57:          move.w    d0, -$0c00(a0)
58:          move.w    d0, -$0800(a0)

```

```

59:      move, w  d0, -$0400 (a0)
60:      move, w  d0, (a0)
61:      adda, l   d5, a0
62: .endif
63: gvline: dbra  d3, vloop
64:      rts
65:
66:      .end

```

高速化するとなれば、クリッピングルーチンにも目を向けなければならないだろう。じつをいうと中点分割アルゴリズムは、高水準言語よりもマシン語、高性能なプロセッサよりは低機能なマイクロプロセッサ、ソフトウェアよりもハードウェアでの実現に適しており、68000クラスのプロセッサになると、乗除算命令を使って数学的な手法で直接交点を算出したほうがより高速になる。ただし、68000の乗除算命令は16ビット×16ビット、32ビット/16ビットの演算しか行えないため、オーバーフローには気を遣わなければならない。

ウィンドウの左辺、 $x = \min x$ で始点 (x_0, y_0) をクリッピングする場合を考えよう。一見、

$$y = m (\min x - x_0) + y_0 \quad \left(\text{ただし } m = \frac{y_1 - y_0}{x_1 - x_0} \right)$$

をそのままプログラムにすればよいように見える。ところが、いま、座標値の範囲は $-32768 \sim +32767$ だから、 $\min x - x_0$, $y_1 - y_0$, $x_1 - x_0$ はすべて、 $-65535 \sim +65535$ の値をとる可能性があり、これは17ビットなければ表現できない。単純に68000の乗除算命令を使うわけにはいかないのだ。そこで、リスト12である。

リスト12では、これまでと同じように、まず、座標値に8000_Hのゲタを履かせて無符号化する。無符号化しても、やはり座標値どうしの差は $-65535 \sim +65535$ の値をとり、16ビット演算ではオーバーフローする可能性はある。だが、無符号数どうしの減算だから、Cフラグが立ったかどうかで結果の正負が判断できる²⁾。リスト12では減算結果が負であれば「それを覚えておいて」、negwにより非負に変換する。この操作は $\min x - x_0$, $y_1 - y_0$, $x_1 - x_0$ のそれぞれについて行う。それから、その結果である $|x - x_0|$, $|y_1 - y_0|$, $|x_1 - x_0|$ を使って、

$$m | \min x - x_0 | \quad \left(\text{ただし } m = \frac{|y_1 - y_0|}{|x_1 - x_0|} \right)$$

を計算する。絶対値をとっているから、各項は $0 \sim 65535$ に収まっており、ここでは無符号の乗除算命令、muluとdivuが利用できる。そして、「何回negを使ったか」におうじて、演算結果の符号を調節する。negを使った回数が奇数回だったなら -1 をかけるわけだ。この時点で、

$$m (\min x - x_0)$$

が求まったから、最後に y_0 を足し込んで $\min x$ に対応する y の値を得る。

■ 2) 8000_Hのゲタを履かせて無符号化しなくとも、VフラグとNフラグの組みあわせで、減算結果の真の符号を知ることができる。ただ、2つのフラグを組みあわせるには2度の条件分岐が必要となり、効率は落ちる。

リスト12
GCLIPLINES
(第2版)

```

1: *      線分の矩形領域クリッピング
2:
3:      .include      rect.h
4: *
5:      .xdef      gclipline
6:      .xref      ucliprect
7: *
8:      .text
9:      .even
10: *
11: *      線分 (d0.w, d1.w) - (d2.w, d3.w) を
12: *      cliprectで指定された矩形領域でクリップする
13: *      完全不可視の場合はZ=0で戻る
14: *
15: gclipline:
16:      movem.w ucliprect, a0-a3  #a0~a3 = ウィンドウ (ゲタ履き)
17:
18:      move.w  #8000, d4          #座標に8000のゲタを履かせる
19:      add.w   d4, d0             #
20:      add.w   d4, d1             #
21:      add.w   d4, d2             #
22:      add.w   d4, d3             #
23:
24:      bsr     gcliplinesub       #始点をクリップ
25:      bne     done               #なおかつ不可視だった
26:
27:      exg.l   d0, d2             #始点と終点を取り替える
28:      exg.l   d1, d3             #
29:      bsr     gcliplinesub       #終点をクリップ
30:      bne     done               #なおかつ不可視だった
31:      exg.l   d0, d2             #取り替えた始点と終点を
32:      exg.l   d1, d3             #元に戻す
33:
34:      move.w  #8000, d4          #ゲタを脱がせる
35:      sub.w   d4, d0             #
36:      sub.w   d4, d1             #
37:      sub.w   d4, d2             #
38:      sub.w   d4, d3             #
39:
40:      moveq.l #0, d4             #Z = 1
41: done:   rts
42:
43: *
44: *      片方の端点 (d0.w, d1.w) をクリップする
45: *
46: gcliplinesub:
47:      move.w  d0, d4             #クリッピング前の座標を
48:      move.w  d1, d5             #覚えておく
49:
50: *
51: *      MINXでクリップする
52: *
53: minxclip:
54:      cmp.w   a0, d0             #MINX ≥ x0ならば
55:      bcc     minyclip           # MINXでのクリッピングは不要
56:      cmp.w   a0, d2             #MINX < x0かつMINX < x1ならば
57:      bcs     outofscrn          # 完全不可視
58:      move.w  a0, d7             #d7 = MINX
59:
60:      moveq.l #0, d6
61:      sub.w   d2, d0             #d0 = abs(x0-x1)
62:      bcc     minx0              #
63:      neg.w   d0                 #
64:      addq.w  #1, d6             #d6 = 符号反転回数
65: minx0:   sub.w  d3, d1          #d1 = abs(y0-y1)
66:      bcc     minx1              #
67:      neg.w   d1                 #
68:      addq.w  #1, d6             #d6 = 符号反転回数
69: minx1:   sub.w  d4, d7          #d7 = abs(x0-MINX)
70:      bcc     minx2              #
71:      neg.w   d7                 #
72:      addq.w  #1, d6             #d6 = 符号反転回数

```

```

73: minx2: mulu.w d7, d1      *d1 = abs(y0-y1) * abs(x0-MINX)
74:         divu.w d0, d1      *d1 /= abs(x0-x1)
75:         btst.l #0, d6      *符号反転回数が奇数回ならば
76:         beq minx3          * 結果の符号を反転する
77:         neg.w d1            *
78: minx3: add.w d5, d1        *d1 = MINXに対応するy
79:         move.w a0, d0      *d0 = MINX
80:
81: *
82: *      MINYでクリップする
83: *
84: minyclip:
85:         cmp.w a1, d1
86:         bcc maxxclip
87:         cmp.w a1, d3
88:         bcs outofscrn
89:         move.w a1, d7
90:
91:         moveq.l #0, d6
92:         move.w d5, d1
93:         sub.w d3, d1
94:         bcc miny0
95:         neg.w d1
96:         addq.w #1, d6
97: miny0: move.w d4, d0
98:         sub.w d2, d0
99:         bcc miny1
100:        neg.w d0
101:        addq.w #1, d6
102: miny1: sub.w d5, d7
103:        bcc miny2
104:        neg.w d7
105:        addq.w #1, d6
106: miny2: mulu.w d7, d0
107:        divu.w d1, d0
108:        btst.l #0, d6
109:        beq miny3
110:        neg.w d0
111: miny3: add.w d4, d0      *d0 = MINYに対応するx
112:        cmp.w a0, d0      *x0 < MINXならば
113:        bcs outofscrn    * 完全不可視
114:        move.w a1, d1      *d1 = MINY
115:
116: *
117: *      MAXXでクリップする
118: *
119: maxxclip:
120:        cmp.w a2, d0
121:        bls maxyclip
122:        cmp.w a2, d2
123:        bhi outofscrn
124:        move.w a2, d7
125:
126:        moveq.l #0, d6
127:        move.w d4, d0
128:        sub.w d2, d0
129:        bcc maxx0
130:        neg.w d0
131:        addq.w #1, d6
132: maxx0: move.w d5, d1
133:        sub.w d3, d1
134:        bcc maxx1
135:        neg.w d1
136:        addq.w #1, d6
137: maxx1: sub.w d4, d7
138:        bcc maxx2
139:        neg.w d7
140:        addq.w #1, d6
141: maxx2: mulu.w d7, d1
142:        divu.w d0, d1
143:        btst.l #0, d6
144:        beq maxx3
145:        neg.w d1
146: maxx3: add.w d5, d1      *d1 = MAXXに対応するy

```

```

147:      cmp.w    a1,d1      *y0<MINYならば
148:      bcs     outofscrn  * 完全不可視
149:      move.w   a2,d0      *d0 = MAXX
150:
151: *
152: *      MAXYでクリップする
153: *
154: maxyclip:
155:      cmp.w    a3,d1
156:      bls     visible
157:      cmp.w    a3,d3
158:      bhi     outofscrn
159:      move.w   a3,d7
160:
161:      moveq.l  #0,d6
162:      move.w   d5,d1
163:      sub.w    d3,d1
164:      bcc     maxy0
165:      neg.w    d1
166:      addq.w   #1,d6
167: maxy0:  move.w   d4,d0
168:      sub.w    d2,d0
169:      bcc     maxy1
170:      neg.w    d0
171:      addq.w   #1,d6
172: maxy1:  sub.w    d5,d7
173:      bcc     maxy2
174:      neg.w    d7
175:      addq.w   #1,d6
176: maxy2:  mulu.w   d7,d0
177:      divu.w   d1,d0
178:      btst.l   #0,d6
179:      beq     maxy3
180:      neg.w    d0
181: maxy3:  add.w    d4,d0      *d0 = MAXYに対応するx
182:      cmp.w    a0,d0      *d0<MINXならば
183:      bcs     outofscrn  * 完全不可視
184:      cmp.w    a2,d0      *d0>MAXXならば
185:      bhi     outofscrn  * 完全不可視
186:      move.w   a3,d1      *d1 = MAXY
187:
188: visible:
189:      moveq.l  #0,d4      *Z=1
190:      rts
191:
192: outofscrn:
193:      moveq.l  #-1,d4     *Z=0
194:      rts
195:
196:      .end

```

リスト13はリスト12のオプションだ。リスト13中、行番号のかわりに“+”で示した行はリスト12に対する追加挿入を意味する。この修正の目的は、線分が完全可視の場合にゲタを履かせたり脱がせたりする部分を飛ばしてすぐ戻るようにすることにある。線分が完全可視ではない場合の効率を落とすことにはなるが、完全可視の確率が高いようであれば、平均の性能向上にわずかなりとも貢献するだろう。

.....
 リスト13
 GCLIPLINE.S
 (完全可視線分特別扱い版: リスト12に対する追加・変更部)

```

5:      .xdef    gclipline
+      .xref    cliprect
6:      .xref    ucliprect

15: gclipline:
+      movem.w  cliprect,a0-a3  *a0~a3 = ウィンドウ
+      cmp.w    a0,d0
+      blt     clip

```

```

+      cmp.w  a0, d2
+      blt    clip
+      cmp.w  a1, d1
+      blt    clip
+      cmp.w  a1, d3
+      blt    clip
+
+      cmp.w  a2, d0
+      bgt    clip
+      cmp.w  a2, d2
+      bgt    clip
+      cmp.w  a3, d1
+      bgt    clip
+      cmp.w  a3, d3
+      bgt    clip
+
+      moveq.l #0, d4      *Z=1
+      rts
+
16: clip:  movem.w ucliprect, a0-a3 *a0~a3 = ウィンドウ (ゲタ履き)

```

ダブルステップBresenhamアルゴリズム

コーディング上の改良はそろそろ限界のようなので、アルゴリズムを変更することを検討してみよう。Bresenhamのアルゴリズムは実に20数年前の古いアルゴリズムだが、つい最近になって（ここ1～2年）その改良版がベル研（だったと思う）の人間によって開発された。そのアルゴリズム、ダブルステップBresenhamを紹介する。その名が示すように、このアルゴリズムではオリジナルのBresenhamアルゴリズムの2ステップ分をまとめて実行し、2ピクセルずつ描画していく。両端から描けば、ループ回数はオリジナルの1/4になる。

図8を見てほしい。傾きが1以下のなだらかな右上がりの線分を2ピクセル単位で切り出してみると、図8のa)～d)の4パターンに分類される（右端は2ピクセル描画した直後、次に点灯すべきピクセルを表す）。ここで、場合分けを推し進め、傾き m が $0 \leq m < 1/2$ の場合と $1/2 \leq m \leq 1$ の場合とに分けて考えると、

$0 \leq m < 1/2$ の線分はa), b), c)

$1/2 \leq m \leq 1$ の線分はb), c), d)

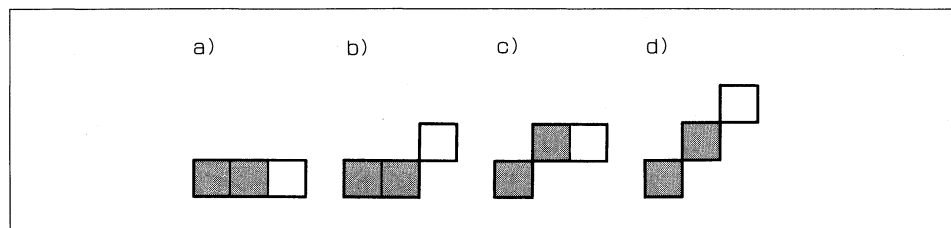


図8
ダブルステップBresenham
アルゴリズムの考え方

の、それぞれ3パターンのみで構成されることがわかる。ダブルステップBresenhamアルゴリズムでは、この3パターンを適切に選択し、線分を2ピクセル単位で描画していく。

傾きが $1/2$ 未満の場合を詳しく見てみよう。まず、いつものように誤差項 E の増分と E が正になったときの補正値を求めておく。

$$dx = |x_1 - x_0|$$

$$dy = |y_1 - y_0|$$

と置けば、増分は $2dy$ 、補正値は $2dx$ だ。で、ループ1回ごとに、 E に増分の2ステップ分、いつもの2倍の $4dy$ を加え、結果の符号を調べる。a) のパターンの場合、 y 座標が変化しないから E の符号は負のままになるはずだ。b), c) の場合は y 座標が一度増加しているので E は正になる。この時点で、a) のケースかどうか判断できる。b) とc) の区別には、これまで使われていなかった誤差項の「値」を使う。かりに、オリジナルどおりに E に増分を2回に分けて加えたとすると、b) のケースは1回目の加算で E が正になるが、c) のケースでは2回目の加算ではじめて正になる。で、c) のケースでは、1回目の加算時はまだ負だったのだから、これに $2dy$ を加えた結果は、 $2dy$ 未満だ。これでb) とc) の区別がつく。

傾きが $1/2 \sim 1$ の場合も似たような考え方をする。ループ1回ごとに、増分の2倍－補正値、 $4dy - 2dx$ を E に加える。b) ~d) のケースでは、 y 座標が最低でも1回変化するからこの分の補正もまとめてしまっているのだ。この加算の結果、 E が正であれば、d) のケースだということがわかる。もう1回補正しなければ、 E が負にならないのは y 座標が2増加するd) のケースだけだからだ。b) とc) は、 E が $2dy - 2dx$ より小さいか大きいかで区別する。この理屈は傾きが $1/2$ 未満の場合と同様だ。

まとめると、こうなる。

- 1) $x = x_0, y = y_0$, 誤差項 $E = 0$ に初期化する。また、 $dx = |x_1 - x_0|$,
 $dy = |y_1 - y_0|$ とおく
- 2) $dy/dx \geq 1/2$ の場合は4) へ
- 3) E に $4dy$ を加える
- 3-1) $E < 0$ ならば、a) のパターンで2ピクセル描画する
- 3-2) $0 \leq E < 2dy$ ならば、b) のパターンで2ピクセル描画して、 E から $2dx$ を引く
- 3-3) さもなくば、c) のパターンで2ピクセル描画して、 E から $2dx$ を引く
- 3-4) 3) に戻って必要なだけ繰り返す
- 4) E に $4dy - 2dx$ を加える
- 4-1) $E \geq 0$ ならば、d) のパターンで2ピクセル描画して、 E から $2dx$ を引く
- 4-2) $2dy - 2dx \leq E < 0$ ならば、b) のパターンで2ピクセル描画する
- 4-3) さもなくば、c) のパターンで2ピクセル描画する
- 4-4) 4) に戻って必要なだけ繰り返す

実際には、これを線分の両端から実行し、最後に、中央に残った1~3ピクセルをオリジナルのBresenhamアルゴリズムで描く。

リスト14にダブルステップBresenhamアルゴリズムによる線分描画ルーチンのコーディング例を示す。3パターンに処理を振り分けたあと合流するために無条件分岐するのを避けて、分岐した先それぞれにdbraを置いてあることに注意してもらえれば、C風の疑似コードのかたちで表した注釈から処理の流れは掴めるだろう。最後に中央に残る端数分はループではなく、ジャンプテーブルですばっと処理を振り分けて個別対応するようにしてみた。

リスト14
GLINE.S
(仮最終版)

```

1: *      線分描画 (整数化ダブルステップBresenham)
2:
3:      .include      gconst.h
4: *
5:      .xdef      gline
6:      .xref      gclipline
7:      .xref      gbase
8: *
9:      .offset 0      *glineの引数構造
10: *
11: X0:      .ds.w      1      *始点座標
12: Y0:      .ds.w      1      *
13: X1:      .ds.w      1      *終点座標
14: Y1:      .ds.w      1      *
15: COL:     .ds.w      1      *描画色
16: *
17:      .text
18:      .even
19: *
20: gline:
21: ARGPTR =      4+8*4+6*4
22: movem.l d0-d7/a0-a5, -(sp)
23:
24: movea.l ARGPTR(sp), a5      *a5 = 引数列
25: movem.w (a5)+, d0-d3      *d0~d3に座標を取り出す
26:
27: jsr      gclipline      *クリッピングする
28: bne      done      *Z=0なら完全不可視
29:
30: cmp.w    d2, d0      *x0 ≤ x1を保証する
31: ble      gline0      *
32: exg.l    d0, d2      *
33: exg.l    d1, d3      *
34:
35: gline0: move.w    d1, d6      *始点/終点のG-RAMアドレスを求める
36: move.w    d3, d7      *
37: ext.l     d6      *
38: ext.l     d7      *
39:      *
40: moveq.l   #GSFCTR, d4      *
41: asl.l     d4, d6      *
42: asl.l     d4, d7      *
43: add.w     d0, d6      *
44: add.w     d0, d6      *
45: add.w     d2, d7      *
46: add.w     d2, d7      *
47:      *
48: movea.l   gbase, a0      *
49: movea.l   a0, a2      *
50: add.l     d6, a0      *a0 = 始点のG-RAMアドレス
51: add.l     d7, a2      *a2 = 終点のG-RAMアドレス
52:
53: move.w    #GNBYTE, d5      *d5 = 横1ライン分のバイト数
54: sub.w     d1, d3      *d3 = y1-y0
55: beq      hor_line      *y0=y1なら水平線
56: bpl      gline1      *
57: neg.w     d3      *
58: neg.w     d5      *
59: gline1: sub.w     d0, d2      *d2 = x1-x0 ( ≥0 )
60: beq      ver_line      *x0=x1なら垂直線
61: *この時点で

```

```

62: *      d2 = dx = abs(x1-x0) (> 0)
63: *      d3 = dy = abs(y1-y0) (> 0)
64: *      d5 = sy = sgn(y1-y0) (-1 or 1)
65: *      (ただしd5はGNBYTE倍済み)
66:
67:      move.w  (a5), d0      *d0.l = 描画色
68:      move.w  d0, d7
69:      swap.w  d0
70:      move.w  d7, d0
71:
72:      cmp.w   d3, d2      *dy > dxならば
73:      bcs     yline      * yについてループ
74:      beq     xylene      *dy = dxならば45度の線
75:
76:      *dx ≥ dyのとき
77: xline: addq.l #2, a2      *プリデクリメントする分補正
78:
79:      move.w  d2, d1      *d1 = dx
80:      neg.w   d1          *d1 = e = -dx
81:      move.w  d2, d6      *d6 = dx
82:      add.w   d2, d2      *d2 = 2*dx
83:      add.w   d3, d3      *d3 = 2*dy
84:      move.w  d3, d4      *d4 = 2*dy
85:      add.w   d3, d3      *d3 = 4*dy
86:
87:      addq.w  #1, d6      *d6 = 線分のピクセル数
88:      moveq.l #3, d7
89:      and.w   d6, d7      *d7 = 線分のピクセル数 mod 4
90:      lsr.w   #2, d6      *ループカウンタを1/4
91:      subq.w  #1, d6      *dbraの動作を考慮
92:      bmi     xlineq      *最初から4ピクセル未満だった
93:
94:      cmp.w   d2, d3      *if ( 4*dy < 2*dx ) {
95:      bcc     xline3
96:
97:      *dy/dx < 1/2のとき
98: xline0: add.w  d3, d1      * e += 4*dy
99:      bpl     xline1      * if ( e < 0 ) {
100:
101: *      ●●○
102:
103:      move.l  d0, (a0)+      * pset(x++, y); pset(x++, y)
104:      move.l  d0, -(a2)      * pset(x'--, y'), pset(x'--, y')
105:
106:      dbra    d6, xline0
107:      bra     xlineq
108:
109: xline1: cmp.w  d4, d1      * } else if ( e < 2*dy ) {
110:      bge     xline2
111:
112: *      ○
113: *      ●●
114:
115:      move.l  d0, (a0)+      * pset(x++, y); pset(x++, y)
116:      move.l  d0, -(a2)      * pset(x'--, y'), pset(x'--, y')
117:      adda.w  d5, a0          * y += sy
118:      suba.w  d5, a2          * y' -= sy
119:
120:      sub.w   d2, d1          * e -= 2*dx
121:
122:      dbra    d6, xline0
123:      bra     xlineq
124:
125:      * } else {
126: *      ●●○
127: *      ●
128:
129: xline2: move.w  d0, (a0)+      * pset(x++, y)
130:      move.w  d0, -(a2)      * pset(--x', y')
131:      adda.w  d5, a0          * y += sy
132:      suba.w  d5, a2          * y' -= sy
133:      move.w  d0, (a0)+      * pset(x++, y)
134:      move.w  d0, -(a2)      * pset(--x', y')
135:

```

```

136:      sub.w  d2,d1      *   e -= 2*dx
137:      *   }
138:      dbra   d6,xline0
139:      bra    xlineq
140:
141:      *} else {
142:
143:      *1/2 ≤ dy/dx ≤ 1 のとき
144: xline3: sub.w  d2,d3      *   d3 = 4*dy-2*dx
145:      sub.w  d2,d4      *   d4 = 2*dy-2*dx
146:
147: xline4: add.w  d3,d1      *   e += 4*dy-2*dx
148:      bmi    xline5      *   if ( e >= 0 ) {
149:
150: *
151: *
152: *
153:
154:      move.w  d0,(a0)+      *   pset(x++,y)
155:      move.w  d0,-(a2)      *   pset(--x',y')
156:      adda.w  d5,a0         *   y += sy
157:      suba.w  d5,a2         *   y' -= sy
158:      move.w  d0,(a0)+      *   pset(x++,y)
159:      move.w  d0,-(a2)      *   pset(--x',y')
160:      adda.w  d5,a0         *   y += sy
161:      suba.w  d5,a2         *   y' -= sy
162:      *
163:      sub.w  d2,d1      *   e -= 2*dx
164:
165:      dbra   d6,xline4
166:      bra    xlineq2
167:
168: xline5: cmp.w  d4,d1      * } else if ( e < 2*dy-2*dx ) {
169:      bge    xline6
170:
171: *
172: *
173:
174:      move.l  d0,(a0)+      *   pset(x++,y), pset(x++,y)
175:      move.l  d0,-(a2)      *   pset(x'--,y'), pset(x'--,y')
176:      adda.w  d5,a0         *   y += sy
177:      suba.w  d5,a2         *   y' -= sy
178:
179:      dbra   d6,xline4
180:      bra    xlineq2
181:
182: *
183: *
184:
185: xline6: move.w  d0,(a0)+      * } else {
186:      move.w  d0,-(a2)      *   pset(x++,y)
187:      adda.w  d5,a0         *   pset(--x',y')
188:      suba.w  d5,a2         *   y += sy
189:      move.w  d0,(a0)+      *   y' -= sy
190:      move.w  d0,-(a2)      *   pset(x++,y)
191:      *   pset(--x',y')
192:      dbra   d6,xline4      * }
193: *   bra    xlineq      *}
194:
195: xlineq2:
196:      add.w  d2,d4      *d4 = 2*dy
197: xlineq: add.w  d7,d7      *4 ピクセル未満の端数分を描写
198:      move.w  xtbl(pc,d7),d7
199:      jmp     xtbl(pc,d7)
200: xtbl:
201: x      =      xtbl
202:      .dc.w  done-x
203:      .dc.w  xleft1-x
204:      .dc.w  xleft2-x
205:      .dc.w  xleft3-x
206:
207: xleft1: move.w  d0,(a0)      *残り 1 ピクセル
208:      bra    done
209:

```

```

210: xleft2: move.w d0, (a0)          *残り2ピクセル
211:         move.w d0, -(a2)
212:         bra     done
213:
214: xleft3: move.w d0, (a0) +        *残り3ピクセル
215:         move.w d0, -(a2)
216:         add.w  d4, d1
217:         bmi    xleft4
218:         adda.w d5, a0
219: xleft4: move.w d0, (a0)
220:         bra     done
221: *
222:
223: yline:  movea.w d5, a3          *dx<dyのとき
224:         addq.l  #2, a3          *a3 = y変化時のG-RAMアドレス増分
225:                                     *a3 = x, y変化時のG-RAMアドレス増分
226:         move.w  d3, d1
227:         neg.w   d1
228:         move.w  d3, d6
229:         add.w   d3, d3
230:         add.w   d2, d2
231:         move.w  d2, d4
232:         add.w   d2, d2
233:
234:         addq.w  #1, d6
235:         moveq.l #3, d7
236:         and.w   d6, d7
237:         lsr.w   #2, d6
238:         subq.w  #1, d6
239:         bmi    ylineq
240:
241:         cmp.w   d3, d2
242:         bcc     yline3
243:
244: yline0: add.w   d2, d1
245:         bpl     yline1
246:
247: *      ○
248: *      ●
249: *      ●
250:
251:         move.w  d0, (a0)
252:         move.w  d0, (a2)
253:         adda.w  d5, a0
254:         suba.w  d5, a2
255:         move.w  d0, (a0)
256:         move.w  d0, (a2)
257:         adda.w  d5, a0
258:         suba.w  d5, a2
259:
260:         dbra    d6, yline0
261:         bra     ylineq
262:
263: yline1: cmp.w   d4, d1
264:         bge     yline2
265:
266: *      ○
267: *      ●
268: *      ●
269:
270:         move.w  d0, (a0)
271:         move.w  d0, (a2)
272:         adda.w  d5, a0
273:         suba.w  d5, a2
274:         move.w  d0, (a0)
275:         move.w  d0, (a2)
276:         adda.w  a3, a0
277:         suba.w  a3, a2
278:
279:         sub.w   d3, d1
280:
281:         dbra    d6, yline0
282:         bra     ylineq
283:

```

```

284: *
285: *
286: *
287:
288: yline2: move.w d0, (a0)
289:         move.w d0, (a2)
290:         adda.w a3, a0
291:         suba.w a3, a2
292:         move.w d0, (a0)
293:         move.w d0, (a2)
294:         adda.w d5, a0
295:         suba.w d5, a2
296:
297:         sub.w d3, d1
298:
299:         dbra d6, yline0
300:         bra ylineq
301:
302:                                     *1/2 ≤ dx/dy ≤ 1 のとき
303: yline3: sub.w d3, d2
304:         sub.w d3, d4
305:
306: yline4: add.w d2, d1
307:         bmi yline5
308:
309: *
310: *
311: *
312:
313:         move.w d0, (a0)
314:         move.w d0, (a2)
315:         adda.w a3, a0
316:         suba.w a3, a2
317:         move.w d0, (a0)
318:         move.w d0, (a2)
319:         adda.w a3, a0
320:         suba.w a3, a2
321:
322:         sub.w d3, d1
323:
324:         dbra d6, yline4
325:         bra ylineq2
326:
327: yline5: cmp.w d4, d1
328:         bge yline6
329:
330: *
331: *
332: *
333:
334:         move.w d0, (a0)
335:         move.w d0, (a2)
336:         adda.w d5, a0
337:         suba.w d5, a2
338:         move.w d0, (a0)
339:         move.w d0, (a2)
340:         adda.w a3, a0
341:         suba.w a3, a2
342:
343:         dbra d6, yline4
344:         bra ylineq2
345:
346: *
347: *
348: *
349:
350: yline6: move.w d0, (a0)
351:         move.w d0, (a2)
352:         adda.w a3, a0
353:         suba.w a3, a2
354:         move.w d0, (a0)
355:         move.w d0, (a2)
356:         adda.w d5, a0
357:         suba.w d5, a2

```

```

358:
359:         dbra     d6, yline4
360:
361: ylineq2:
362:         add.w    d3, d4
363: ylineq:  add.w    d7, d7
364:         move.w   ytbl(pc, d7), d7
365:         jmp      ytbl(pc, d7)
366: ytbl:
367: y        =        ytbl
368:         .dc.w    done-y
369:         .dc.w    yleft1-y
370:         .dc.w    yleft2-y
371:         .dc.w    yleft3-y
372:
373: yleft1:  move.w   d0, (a0)
374:         bra      done
375:
376: yleft2:  move.w   d0, (a0)
377:         move.w   d0, (a2)
378:         bra      done
379:
380: yleft3:  move.w   d0, (a0)
381:         move.w   d0, (a2)
382:         add.w    d4, d1
383:         bmi      yleft4
384:         move.w   d0, 0(a0, a3)
385:         bra      done
386: yleft4:  move.w   d0, 0(a0, d5)
387:         bra      done
388:
389: done:    movem.l  (sp)+, d0-d7/a0-a5
390:         rts
391: * (以下, リスト9の134行~に同じ)

```

もっと速く

ここまでは、グラフィック画面で表現できる限界の精度で線分を描くことにこだわってきた。しかし、精度を犠牲にしてもよければ、まだ実行速度は稼げる。以下の話は、精度よりも速度を追及したい場合向きのオプションである。

整数化Bresenhamでは、E周りの計算に必要な項を2dx倍して整数化したが、そのかわりに65536倍して整数化するという方法がある。

Eの初期値 $-1/2 \rightarrow -32768$

Eの増分 $dy/dx \rightarrow (65536 \times dy) / dx$

Eの補正值 1 $\rightarrow 65536$

ここで、誤差項Eの増分は上の式の小数点以下を切り捨てるものとする（ここで精度が落ちる）。増分を加えた結果0以上になったかどうかの判定にはCフラグを利用し、Cフラグが立ったらy座標を進め、誤差項から補正值分1の65536倍を引く。ところが、16ビット数で表現できるのは65535までだから、65536を引く操作は何もしないことと同じだ。というか、誤差項に増分を加える操作を無符号演算だと思えば、繰り上がった分がCフラグにはみ出した時点で65536を引いたことになる。こうして体よく、Eが正になったときの補正を省略することがで

きる。この手法は精度を犠牲にしても実行速度を優先せざるをえない8ビットマシンではけっこう多用された。

リスト15に実際のプログラムを示す。リスト15自体はリスト8がベースになっているので、リスト8より少し速いという程度でしかない。リスト14のダブルステップBresenhamアルゴリズム版に対抗するためには、こちらもダブルステップ化することになる。だが、同時にひとひねりしてみよう。

リスト15
GLINES
(65536倍整数化版)

```

1: *      線分描画 (65536倍整数化Bresenham)
2:
3:      .include      gconst.h
4: *
5:      .xdef      gline
6:      .xref      gclipline
7:      .xref      gbase
8: *
9:      .offset 0      *glineの引数構造
10: *
11: X0:      .ds.w      1      *始点座標
12: Y0:      .ds.w      1      *
13: X1:      .ds.w      1      *終点座標
14: Y1:      .ds.w      1      *
15: COL:     .ds.w      1      *描画色
16: *
17:      .text
18:      .even
19: *
20: gline:
21: ARGPTR =      4+8*4+6*4
22: movem.l d0-d7/a0-a5, -(sp)
23:
24: movea.l ARGPTR(sp), a5      *a5 = 引数列
25: movem.w (a5)+, d0-d3      *d0~d3に座標を取り出す
26:
27: jsr      gclipline      *クリッピングする
28: bne      done      *Z=0なら完全不可視
29:
30: cmp.w    d2, d0      *x0 ≤ x1を保証する
31: ble      gline0      *
32: exg.l    d0, d2      *
33: exg.l    d1, d3      *
34:
35: gline0: move.w    d1, d6      *始点/終点のG-RAMアドレスを求める
36: move.w    d3, d7      *
37: ext.l     d6      *
38: ext.l     d7      *
39:
40: moveq.l   #GSFCTR, d4      *
41: asl.l     d4, d6      *
42: asl.l     d4, d7      *
43: add.w     d0, d6      *
44: add.w     d0, d6      *
45: add.w     d2, d7      *
46: add.w     d2, d7      *
47:
48: movea.l   gbase, a0      *
49: movea.l   a0, a2      *
50: add.l     d6, a0      *a0 = 始点のG-RAMアドレス
51: add.l     d7, a2      *a2 = 終点のG-RAMアドレス
52:
53: move.w    #GNBYTE, d5      *d5 = 横1ライン分のバイト数
54: sub.w     d1, d3      *d3 = y1-y0
55: beq      hor_line      *y0=y1なら水平線
56: bpl      gline1
57: neg.w     d3
58: neg.w     d5

```

```

59: glinel: sub.w  d0, d2          #d2 = x1-x0 (>=0)
60:          beq    ver_line      #x0=x1なら垂直線
61: #この時点で
62: #    d2 = dx = abs(x1-x0) (> 0)
63: #    d3 = dy = abs(y1-y0) (> 0)
64: #    d5 = sy = sgn(y1-y0) (-1 or 1)
65: #    (ただしd5はGNBYTE倍済み)
66:
67:          move.w  (a5), d0       #d0 = 描画色
68:
69:          move.w  #$8000, d1     #d1 = e = -65536 * 1/2
70:
71:          cmp.w   d3, d2         #dy>dxならば
72:          bcs     yline         # yについてループ
73:          beq     xyline        #dy=dxならば45度の線
74:
75:                                     #dx>=dyのとき
76: xline:   swap.w  d3             #
77:          clr.w   d3             #d3 = 65536*dy
78:          divu.w  d2, d3         #d3 = 65536*dy/dx
79:          subq.w  #1, d2         #dbraの動作を計算に入れて
80:          lsr.w   #1, d2         # ループカウンタを半減
81:          scs.b   d4             #奇数ピクセルのとき非0
82:          addq.l  #2, a2         #プリデクリメントする分補正
83:                                     #do {
84: xline0:  move.w  d0, (a0)+      # pset(x++, y)
85:          move.w  d0, -(a2)      # pset(--x', y')
86:          add.w   d3, d1         # e += 2*dy
87:          bcc     xline1        # if (e >= 0) {
88:          adda.w  d5, a0         #   y += sy
89:          suba.w  d5, a2         #   y' -= sy
90:                                     # }
91: xline1:  dbra    d2, xline0     #} while (--n >= 0)
92:
93:          tst.b   d4             #奇数ピクセル?
94:          beq     done          # そうじゃない
95:          bra     odd           #中央のピクセルを点灯
96:
97:                                     #dx<dyのとき
98: yline:   swap.w  d2             #
99:          clr.w   d2             #d2 = 65536*dx
100:          divu.w  d3, d2         #d2 = 65536*dx/dy
101:          subq.w  #1, d3         #dbraの動作を計算に入れて
102:          lsr.w   #1, d3         # ループカウンタを半減
103:          scs.b   d4             #奇数ピクセルのとき非0
104:          move.w  d5, d6         #d6 = d5 + 2
105:          addq.w  #2, d6         #
106:                                     #do {
107: yline0:  add.w   d2, d1         # e += 2*dx
108:          bcs     yline1        # if (e < 0) {
109:          move.w  d0, (a0)      # pset(x, y)
110:          adda.w  d5, a0         # y += sy
111:          move.w  d0, (a2)      # pset(x', y')
112:          suba.w  d5, a2         # y' -= sy
113:                                     # }
114:          dbra    d3, yline0
115:          bra     done0
116:                                     # else {
117: yline1:  move.w  d0, (a0)      # pset(x++, y)
118:          adda.w  d6, a0         # y += sy
119:          move.w  d0, (a2)      # pset(x', y')
120:          suba.w  d6, a2         # x' --, y' -= sy
121:                                     # }
122:          dbra    d3, yline0     #} while (--n >= 0)
123:
124: done0:   tst.b   d4             #奇数ピクセル?
125:          beq     done          # そうじゃない
126:          odd:    move.w  d0, (a0) #中央のピクセルを点灯
127:
128: done:    movem.l (sp)+, d0-d7/a0-a5
129:          rts
130: # (以下, リスト9の134行~に同じ)

```


さらに精度を犠牲にする。ダブルステップBresenhamアルゴリズムでは、線分の傾きが $1/2$ 以上か未満かで処理を2つに分けた。このうち、傾きが $1/2$ よりなだらかな線分の方は最短2ピクセル以上の細かな水平線分の集まりで描くことができる(図9-a)。実際には、図9-a)の線分はBresenhamのアルゴリズムで描かれる正しい線分からはずれているのだが、そんなことはもう気にしない。ここで、各水平線分の長さはたかだか1ピクセルしか変わらないことに注目する。図9-a)は5本の水平線分からなるが、そのどれもが3ピクセルか4ピクセルの長さだ。ここで、4ピクセルの線分を3ピクセルの線分と1点に分割すると図9-b)となる。ループ1回ごとに無条件に3ピクセルの水平線を描き、Bresenhamのアルゴリズムで残り1ピクセルを描くかどうかを決めるようにすれば、ループ回数を大幅に減らすことができる。この方法は、線分の傾きが $1/2$ 以下か2以上のときにしか有効ではないから、ダブルステップBresenhamアルゴリズムの半分と差し替えて使うことになるだろう。

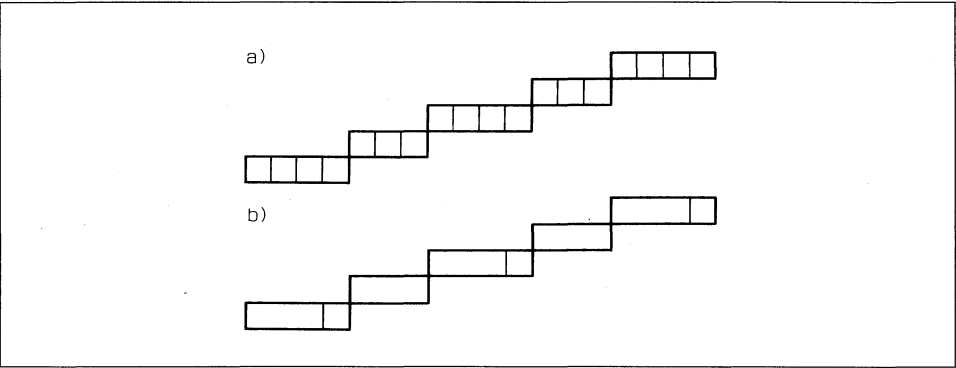


図9
水平線分の集まりとしてとらえた
なだらかな線分

リスト16に実例を示す。骨格はダブルステップBresenhamで、個々のBresenhamアルゴリズム部分は正確な整数化ではなく65536倍版を適用している。また、「短線分+1ピクセル方式」は、ダブルステップBresenhamの半分と置き換えてしまわずに、傾きが $1/4$ 以下か、4以上のときにだけ機能するようにしてある。試してみたところ、傾き $1/4 \sim 1/2$ の範囲ではダブルステップBresenhamそのままのほうが速かったのだ。水平、垂直の短線分を描く部分はその場で派手にループ展開してある。この周辺では少々汚いこともしているので、あえて解説するのはよそう。

リスト16
GLINE.S
(精度犠牲/速度追及
版)

```
1: *      線分描画 (65536倍整数化ダブルステップBresenham,  
2: *      短線分+1ピクセル方式)  
3:      .include      gconst.h  
4: *  
5:      .xdef      gline  
6:      .xref      gclipline  
7:      .xref      gbase  
8: *
```

```

9:      .offset 0      *glineの引数構造
10: *
11: X0:      .ds.w 1      *始点座標
12: Y0:      .ds.w 1      *
13: X1:      .ds.w 1      *終点座標
14: Y1:      .ds.w 1      *
15: COL:     .ds.w 1      *描画色
16: *
17:      .text
18:      .even
19: *
20: gline:
21: ARGPTR    =          4+8*4+6*4
22:      movem.l d0-d7/a0-a5, -(sp)
23:
24:      movea.l ARGPTR(sp), a5      *a5 = 引数列
25:      movem.w (a5)+, d0-d3      *d0~d3に座標を取り出す
26:
27:      jsr     gclicpline      *クリッピングする
28:      bne     done            *Z=0なら完全不可視
29:
30:      cmp.w   d2, d0          *x0 ≤ x1を保証する
31:      ble     gline0          *
32:      exg.l   d0, d2          *
33:      exg.l   d1, d3          *
34:
35: gline0:    move.w d1, d6      *始点/終点のG-RAMアドレスを求める
36:      move.w d3, d7          *
37:      ext.l   d6             *
38:      ext.l   d7             *
39:
40:      moveq.l #GSFCTR, d4     *
41:      asl.l   d4, d6          *
42:      asl.l   d4, d7          *
43:      add.w   d0, d6          *
44:      add.w   d0, d6          *
45:      add.w   d2, d7          *
46:      add.w   d2, d7          *
47:
48:      movea.l gbase, a0       *
49:      movea.l a0, a2          *
50:      add.l   d6, a0          *a0 = 始点のG-RAMアドレス
51:      add.l   d7, a2          *a2 = 終点のG-RAMアドレス
52:
53:      move.w  #GNBYTE, d5     *d5 = 横1ライン分のバイト数
54:      sub.w   d1, d3          *d3 = y1-y0
55:      beq     hor_line        *y0=y1なら水平線
56:      bpl     gline1          *
57:      neg.w   d3              *
58:      neg.w   d5              *
59: gline1:    sub.w d0, d2          *d2 = x1-x0 ( ≥0 )
60:      beq     ver_line        *x0=x1なら垂直線
61: *この時点で
62: *      d2 = dx = abs(x1-x0) ( > 0 )
63: *      d3 = dy = abs(y1-y0) ( > 0 )
64: *      d5 = sy = sgn(y1-y0) ( -1 or 1 )
65: *      (ただしd5はGNBYTE倍済み)
66:
67:      move.w  (a5), d0          *d0 = 描画色
68:      move.w  d0, d7          *
69:      swap.w  d0              *
70:      move.w  d7, d0          *
71:
72:      cmp.w   d3, d2          *dy > dxならば
73:      bcs     yline           * yについてループ
74:      beq     xyline          *dy=dxならば45度の線
75:
76:
77: xline:     addq.l #2, a2      *dx ≥ dyのとき *プリデクリメントする分補正
78:
79:      move.w  #$8000, d1      *d1 = e = -65536 * 1/2
80:
81:      move.w  d3, d7          *
82:      lsl.w   #2, d7          *d7 = dy*4

```

```

83:      cmp.w  d2, d7      *
84:      bcc    xline0      *dy/dx >= 1/4
85:
86:                      *dx/dy < 1/4のとき
87:      addq.w  #1, d2
88:      addq.w  #1, d3
89:      ext.l   d2
90:      divu.w  d3, d2
91:      move.w  d2, d7
92:      move.w  #$20c0, d6      *move.l d0, (a0) +
93:      lea.l   xxlin0(pc), a1
94:      bclr.l  #0, d7
95:      beq     notodd
96:      move.w  #$30c0, d6      *move.w d0, (a0) +
97:      addq.w  #2, d7
98: notodd: neg.w  d7
99:      adda.w  d7, a1
100:     move.w  d6, (a1)
101:     addi.w  #xxlin0-xxlin2-2, d7
102:     move.w  d7, xxlin2+2
103:     clr.w   d2
104:     divu.w  d3, d2
105:     subq.w  #1, d3
106:     jmp     (a1)
107:
108:     .dcb.w  GNPPIXEL/4, $20c0 *move.l d0, (a0) +
109: xline0: add.w  d2, d1
110:     bcc     xxlin1
111:     move.w  d0, (a0) +
112: xxlin1: adda.w  d5, a0
113: xxlin2: dbra   d3, xxlin0
114:
115:     move.w  #$20c0, (a1)      *move.l d0, (a0) +
116:     bra     done
117:
118: xline0: swap.w  d3      *
119:     clr.w   d3      *
120:     divu.w  d2, d3      *d3 = 65536 * dy/dx
121:     move.w  d3, d4      *d4 = 65536 * dy/dx
122:
123:     addq.w  #1, d2      *d7 = 4 ピクセル未満の端数
124:     moveq.l  #3, d7      *
125:     and.w   d2, d7      *
126:     lsr.w   #2, d2      *ループカウンタを1/4
127:     subq.w  #1, d2      *
128:     bmi     xlineq      *
129:
130:     add.w   d3, d3      *d3 = 2 * 65536 * dy/dx
131:     bcs     xline4
132:
133:                      *dy/dx < 1/2のとき
134: xline1: add.w  d3, d1
135:     bcs     xline2
136:
137: *      ●●○
138:
139:     move.l  d0, (a0) +
140:     move.l  d0, -(a2)
141:
142:     dbra   d2, xline1
143:
144:     bra     xlineq
145:
146: xline2: cmp.w  d4, d1
147:     bcc     xline3
148:
149: *      ○
150: *      ●●
151:
152:     move.l  d0, (a0) +
153:     move.l  d0, -(a2)
154:     adda.w  d5, a0
155:     suba.w  d5, a2
156:

```

```

157:      dbra    d2,xline1
158:      bra     xlineq
159:
160: *      ●○
161: *      ●
162:
163: xline3: move.w d0,(a0)+
164:         move.w d0,-(a2)
165:         adda.w d5,a0
166:         suba.w d5,a2
167:         move.w d0,(a0)+
168:         move.w d0,-(a2)
169:
170:      dbra    d2,xline1
171:      bra     xlineq
172:
173:                                     *1/2 ≤ dy/dx ≤ 1 のとき
174: xline4: add.w  d3,d1
175:         bcc   xline5
176:
177: *      ●○
178: *      ●
179: *      ●
180:
181:         move.w d0,(a0)+
182:         move.w d0,-(a2)
183:         adda.w d5,a0
184:         suba.w d5,a2
185:         move.w d0,(a0)+
186:         move.w d0,-(a2)
187:         adda.w d5,a0
188:         suba.w d5,a2
189:
190:      dbra    d2,xline4
191:      bra     xlineq
192:
193: xline5: cmp.w  d4,d1
194:         bcc   xline6
195:
196: *      ●○
197: *      ●●
198:
199:         move.l d0,(a0)+
200:         move.l d0,-(a2)
201:         adda.w d5,a0
202:         suba.w d5,a2
203:
204:      dbra    d2,xline4
205:      bra     xlineq
206:
207: *      ●○
208: *      ●
209: xline6: move.w d0,(a0)+
210:         move.w d0,-(a2)
211:         adda.w d5,a0
212:         suba.w d5,a2
213:         move.w d0,(a0)+
214:         move.w d0,-(a2)
215:
216:      dbra    d2,xline4
217:      bra     xlineq
218: *
219:
220: xlineq: add.w  d7,d7
221:         move.w xtbl(pc,d7),d7
222:         jmp   xtbl(pc,d7)
223: xtbl:
224: x      =      xtbl
225:         .dc.w done-x
226:         .dc.w xleft1-x
227:         .dc.w xleft2-x
228:         .dc.w xleft3-x
229:
230: xleft1: move.w d0,(a0)

```

```

231:         bra     done
232:
233: xleft2: move.w  d0, (a0)
234:         move.w  d0, -(a2)
235:         bra     done
236:
237: xleft3: move.w  d0, (a0) +
238:         move.w  d0, -(a2)
239:         add.w   d4, d1
240:         bcc     xleft4
241:         adda.w  d5, a0
242: xleft4: move.w  d0, (a0)
243:         bra     done
244: *
245:                                     *dx<dyのとき
246: yline:  move.w  d5, d6                *d6 = y変化時のG-RAMアドレス増分
247:         addq.w  #2, d6                *d6 = x, y変化時のG-RAMアドレス増分
248:
249:         move.w  #$8000, d1            *d1 = -65536 * 1/2
250:
251:         move.w  d2, d7
252:         lsl.w   #2, d7
253:         cmp.w   d3, d7
254:         bcc     yline0
255:
256:                                     *dy/dx<1/4のとき
257:         addq.w  #1, d2
258:         addq.w  #1, d3
259:         ext.l   d3
260:         divu.w  d2, d3
261:         move.w  d3, d7
262:         lsl.w   #2, d7
263:         neg.w   d7
264:         lea.l   yylin0(pc), a1
265:         adda.w  d7, a1
266:         addi.w  #yylin0-yylin1-2, d7
267:         move.w  d7, yylin1+2
268:         clr.w   d3
269:         divu.w  d2, d3
270:         subq.w  #1, d2
271:         jmp     (a1)
272:
273:         .dcb.l  GNPPIXEL/2, $3080_d0c5 *move.w d0, (a0) adda.w d5, a0
274:         move.w  d0, (a0)
275:         adda.w  d6, a0
276: yylin0:  add.w   d3, d1
277:         bcc     yylin1
278:         move.w  d0, -2(a0)
279:         adda.w  d5, a0
280: yylin1:  dbra    d2, yylin0
281:
282:         bra     done
283:
284:                                     *dx/dy<1/2のとき
285: yline0:  swap.w  d2
286:         clr.w   d2
287:         divu.w  d3, d2
288:         move.w  d2, d4
289:
290:         addq.w  #1, d3
291:         moveq.l  #3, d7
292:         and.w   d3, d7
293:         lsr.w   #2, d3
294:         subq.w  #1, d3
295:         bmi     ylineq
296:
297:         add.w   d2, d2
298:         bcs     yline4
299:
300: yline1:  add.w   d2, d1
301:         bcs     yline2
302:
303: *       ○

```

```

304: # ●
305: # ●
306:
307:     move.w d0, (a0)
308:     move.w d0, (a2)
309:     adda.w d5, a0
310:     suba.w d5, a2
311:     move.w d0, (a0)
312:     move.w d0, (a2)
313:     adda.w d5, a0
314:     suba.w d5, a2
315:
316:     dbra d3, yline1
317:
318:     bra ylineq
319:
320: yline2: cmp.w d4, d1
321:         bcc yline3
322:
323: # ○
324: # ●
325: # ●
326:
327:     move.w d0, (a0)
328:     move.w d0, (a2)
329:     adda.w d5, a0
330:     suba.w d5, a2
331:     move.w d0, (a0)
332:     move.w d0, (a2)
333:     adda.w d6, a0
334:     suba.w d6, a2
335:
336:     dbra d3, yline1
337:     bra ylineq
338:
339: # ○
340: # ●
341: # ●
342:
343: yline3: move.w d0, (a0)
344:         move.w d0, (a2)
345:         adda.w d6, a0
346:         suba.w d6, a2
347:         move.w d0, (a0)
348:         move.w d0, (a2)
349:         adda.w d5, a0
350:         suba.w d5, a2
351:
352:     dbra d3, yline1
353:     bra ylineq
354:
355:                                     *1/2 ≤ dx/dy ≤ 1 のとき
356: yline4: add.w d2, d1
357:         bcc yline5
358:
359: # ○
360: # ●
361: # ●
362:
363:     move.w d0, (a0)
364:     move.w d0, (a2)
365:     adda.w d6, a0
366:     suba.w d6, a2
367:     move.w d0, (a0)
368:     move.w d0, (a2)
369:     adda.w d6, a0
370:     suba.w d6, a2
371:
372:     dbra d3, yline4
373:     bra ylineq
374:
375: yline5: cmp.w d4, d1
376:         bcc yline6
377:

```

```

378: *      ○
379: *      ●
380: *      ●
381:
382:      move.w d0, (a0)
383:      move.w d0, (a2)
384:      adda.w d5, a0
385:      suba.w d5, a2
386:      move.w d0, (a0)
387:      move.w d0, (a2)
388:      adda.w d6, a0
389:      suba.w d6, a2
390:
391:      dbra   d3, yline4
392:      bra    ylineq
393:
394: *      ○
395: *      ●
396: *      ●
397:
398: yline6: move.w d0, (a0)
399:      move.w d0, (a2)
400:      adda.w d6, a0
401:      suba.w d6, a2
402:      move.w d0, (a0)
403:      move.w d0, (a2)
404:      adda.w d5, a0
405:      suba.w d5, a2
406:
407:      dbra   d3, yline4
408:
409: ylineq: add.w d7, d7
410:      move.w ytbl(pc, d7), d7
411:      jmp     ytbl(pc, d7)
412: ytbl:
413: y      =      ytbl
414:      .dc.w  done-y
415:      .dc.w  yleft1-y
416:      .dc.w  yleft2-y
417:      .dc.w  yleft3-y
418:
419: yleft1: move.w d0, (a0)
420:      bra    done
421:
422: yleft2: move.w d0, (a0)
423:      move.w d0, (a2)
424:      bra    done
425:
426: yleft3: move.w d0, (a0)
427:      move.w d0, (a2)
428:      add.w  d4, d1
429:      bcc    yleft4
430:      move.w d0, 0(a0, d6)
431:      bra    done
432: yleft4: move.w d0, 0(a0, d5)
433:      bra    done
434:
435: done:  movem.l (sp)+, d0-d7/a0-a5
436:      rts
437: * (以下, リスト9の134行~に同じ)

```

ああ、1点補足しておかなければならないのだった。リスト16は、このままではAS.Xでアセンブルできない。AS.Xは、dcb.lのアセンブル動作に不備があり、仕様どおりに使ってもエラーになるのだ。リスト16では傾き4以上のときの垂直線分描画部分、

```

move.w d0,(a0)
adda.w d5,a0

```

の2命令をループ展開するのに、この.dcb.lを使っている。前章のボックスフィルルーチンの最終版でもやったようにマクロを使って展開するようにすればよかったのだが、ここでは、AS.X（ただし、Ver. 2.00）にパッチを当ててバグを回避する方法を示す。AS.Xをカレントディレクトリに置いてリスト17のX-BASICプログラムを実行すると「少なくともリスト16がアセンブルできるように」AS.Xが書き換えられる。ただし、100%の動作保証はできないので、このパッチをあてたAS.Xはリスト16のアセンブルに使った後は「消去する」のが正しい。

.....
リスト17
PATCH.BAS

```
10 /* as.x v2.00 のdcb.lバグ修正
20 dim char q(33)
30 int fp
40 fp=fopen("as.x","rw")
50 fseek(fp,&H5482,0)
60 fread(q,34,fp)
70 q(1)=&H87
80 q(33)=&H7
90 fseek(fp,&H5482,0)
100 fwrite(q,34,fp)
110 fclose(fp)
```

性能評価

最後に、IOCSコールLINEとこの章で作成したサブルーチングlineの各版の性能測定用プログラムをリスト18に示す。DOSコール呼び出し後のスタックポインタの補正を省略してあったり、スーパーバイザモードに移行したまま実行を終了したりしている点で、かなりラフなプログラムになっているが、動作試験用のプログラムとしては十分だろう。リスト18は512×512ピクセルの画面に50000本のランダムな線分を描くのにかかる時間をIOCSコールONTIMEにより計時し、表示する。IOCS ONTIMEは電源投入（あるいはリセット）からの経過時間を1/100秒単位で返すから、描画の前後で1回ずつ呼び出し、その差をとれば描画にかかる時間が計れる。この結果には引数のセットなどの処理も含まれるので、線分描画にかかる正味の時間を知りたければ、44～46行を削除した空ループの時間を測定しておき、その分を引くとよいだろう。リスト18は1回に1種類のラインルーチンの実行時間を測定するだけだから、glineの各版それぞれについて計時するためには、リンク時に適切なモジュールを1つ選んでリンクする必要がある。また、IOCSコールLINEの実行時間を計るときには、44～46行を削り、代わって47行を復活することで対処する。さらに、クリッピングルーチンの性能を加味したければ、7行と33～35行を復活すると、クリッピングウィンドウが(128,128)～(383,383)に設定される。

.....
リスト18
GLINETEST.S

```
1: *      glineのテスト用プログラム
2:
3: .include doscall.mac
```



```

4:      .include      iocscall.mac
5:      *
6:      .xref      gline
7:      *
8:      .xref      setcliprect
9:      *
9:      CR      equ      13
10:     LF      equ      10
11:     *
12:     FPACK    macro    callno
13:     .dc.w    callno
14:     endm
15:     *
16:     _RAND      equ      $fe0e    *乱数 (0~32767)
17:     _IUSING     equ      $fe18    *整数→文字列変換
18:                                     * (桁数指定つき)
19:     *
20:     .text
21:     .even
22:     *
23:     ent:
24:         lea.l    inisp, sp
25:
26:         move.l   #$0010_0005, -(sp)
27:                                     *512x512, 65536色に
28:         DOS      _CONCTRL        * 初期化
29:
30:         clr.l    -(sp)           *スーパーバイザモードへ移行
31:         DOS      _SUPER          *
32:
33:     *
34:     *
35:     *
36:     *
37:     *
38:     *
39:     *
40:     *
41:     *
42:     *
43:     *
44:     *
45:     *
46:     *
47:     *
48:     *
49:     *
50:     *
51:     *
52:     *
53:     *
54:     *
55:     *
56:     *
57:     *
58:     *
59:     *
60:     *
61:     *
62:     *
63:     *
64:     *
65:     *
66:     *
67:     *
68:     *
69:     *
70:     *
71:     *
72:     *
73:     *
74:     *
75:     *
76:     *
77:     *

```

```

78:      dbra    d6, arglp
79:      rts
80:
81: *
82: *      1/100秒単位から 1 秒単位へ変換する (文字列レベル)
83: *
84: conv:
85:      lea.l    temp+7, a1
86:      lea.l    temp+8, a2
87:      moveq.l  #$20, d1
88:      moveq.l  #'0', d2
89:      clr.b    (a2)
90:      move.b   -(a1), -(a2)
91:      move.b   -(a1), d0
92:      cmp.b    d1, d0
93:      bne      skip1
94:      move.b   d2, d0
95: skip1: move.b   d0, -(a2)
96:      move.b   #'.' , -(a2)
97:      move.b   -(a2), d0
98:      cmp.b    d1, d0
99:      bne      skip2
100:     move.b   d2, (a2)
101: skip2: rts
102: *
103:      .data
104:      .even
105: *
106: window: .dc.w 128, 128, 511-128, 511-128
107: argbuf:  .dc.w 0, 0, 0, 0, 63      *引数列
108:      .dc.w    $ffff      *ラインスタイル (10CS用)
109: secmes:  .dc.b ' sec.', CR, LF, 0
110: *
111:      .bss
112:      .even
113: *
114: temp:   .ds.b 10      *数値→文字列変換用
115: *
116:      .stack
117:      .even
118: *
119:      .ds.l 4096
120: inisp:
121:      .end    ent

```

リスト18では、乱数の生成と整数→文字列への変換に数値演算パッケージFLOATn.Xのファンクションコールを利用している。どちらの処理もそう複雑ではないのでFLOATn.Xに頼る必要はなかったのだが、つい楽なほうに走ってしまった。FLOATn.XのファンクションコールはHuman68kのDOSコール同様、F系列の未実装命令を使って呼び出す。ただし、DOSコールがFFxxHを使うのに対し、FLOATn.XのファンクションコールにはFExxHを使う。リスト18ではこの呼び出しを12~14行で定義してあるマクロFPACKで行っている。FPACKの名前や、16~17行のコール名はXC付属のファイルFEFUNC.Hからとっている。RANDは0~32767の整数乱数を返し、IUSINGはd0.lの値をd1桁で10進文字列に変換してa0の指すメモリに収めて返す。より詳しくは、浮動小数点演算ボードのマニュアルか、XC Ver.1.0添付のFEFUNC.H、あるいは、市販の解析本を参照してもらいたい。ここでは、1点、74行で、0~32767の疑似乱数を0~511の座標値の範囲に収めるために、

```
lsl.w  #6, d0
```

を使っている点に触れておく。FLOATn.Xで使用している疑似乱数発生アルゴリズムの性格

で、

```
andi.w #511,d0
```

のように下位ビットを利用するよりは、シフトして上位ビットを利用したほうが概してばらつきのない乱数が得られるのだった。この事情は、X-BASICのrand () 関数などでも変わらない。

表1にラインルーチンの実際の測定結果を示す。空ループの時間はすでに引いてある。クロックは10MHzで、OPMDRV.Xの類の音源ドライバは組み込んでいない。また、クリッピングルーチンはリスト12+リスト13の版を使用している。表にはあわせてサブルーチンの大きさも示してある。ただし、この値には、別ファイルになっているクリッピング部と、水平/垂直線分描画部をループ展開した中身は含まれていない。

	実行時間 (秒)	サイズ (バイト)
IOCS (ROM版)	198.05	—
IOCS (IOCS.X版)	96.82	—
リスト5	61.90	158
リスト7	53.88	152
リスト8	38.53	224
リスト8+9	38.51	336
リスト14+9	32.50	692
リスト15+9	37.86	324
リスト16+9	31.46	2096

表1
ラインルーチンの
性能比較

見てのとおり、この章で作成した線分描画ルーチンは、最初の版ですら、IOCS.Xを組み込んだときのIOCSコールより速いという結果が出た。ただし、この結果を鵜のみにしてはいけない。世の多くのベンチマークテストがそうであるように、このテストは作為的であり、正直な読者を欺くようにできている。第1に、ルーチンの性格の違いがある。IOCSのLINEは毎回の呼び出しごとにグラフィック画面が初期化されているかどうか確認するし、描画色が画面モードにあった範囲に収まっているかどうか調べるし、IOCSコールの呼び出し手順そのもののオーバーヘッドもある。対して、サブルーチングlineは画面モード周りのエラーチェックをしていない。それに加えて、機能の差がある。IOCSのLINEはラインスタイルをサポートしているが、glineは実線専用だ。ラインスタイルをサポートしようとすると、細工の余地がほとんどなくなるので、オーソドックスなつくり (リスト5程度) にせざるをえないし、それに余計な処理を追加すると実線専用ルーチンとくらべて実行時間が60~70%増しになると考えられる。その点を割り引けばIOCS.X版LINEの潜在的な性能はリスト5とトントンだといえる³⁾。参考までに、リスト19はリスト5をラインスタイル対応にしたものだ。要はd7.wに16ビットのビットパターンを入れておき、これをrol.wで回転させて、はみ出したビットの1/0におうじて点を打つか打たないかを決めることでラインスタイルを実現している。実行時間を

測定してみると、IOCS.X版LINEと同程度にまで速度が低下しているのがわかるだろう。

- 3) IOCSもラインスタイルが FFFFH (実線)のときには専用ルーチンで対応するぐらいのことはしてくれなかったってバチは当たらないようにも思うが。

リスト19
GLINE_LS.S

```

1: *      線分描画 (ラインスタイル対応)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6:      .xdef      gline_ls
7:      .xref      gclipline
8:      .xref      gramadr
9: *
10:     .offset 0      *gline_lsの引数構造
11: *
12: X0:     .ds.w      1      *始点座標
13: Y0:     .ds.w      1      *
14: X1:     .ds.w      1      *終点座標
15: Y1:     .ds.w      1      *
16: COL:    .ds.w      1      *描画色
17: BITPAT: .ds.w      1      *ラインスタイル
18: *
19:     .text
20:     .even
21: *
22: gline_ls:
23: ARGPTR =      4+8+4+6*4
24:     movem.l      d0-d7/a0-a5, -(sp)
25:
26:     movea.l      ARGPTR(sp), a5      *a5 = 引数列
27:
28:     movem.w      (a5), d0-d3      *d0~d3に座標を取り出す
29:
30:     jsr      gclipline      *クリッピングする
31:     bne      done      *Z=0なら描画の必要なし
32:
33:     move.w      BITPAT(a5), d7      *d7 = ラインスタイル
34:     beq      done
35:
36:     jsr      gramadr      *始点のG-RAM上アドレスを得る
37:
38:     sub.w      d0, d2      *d2 = x1-x0
39:     move.w      d2, d4      *d4 = d2
40:     ABS      d2      *d2 = dx = abs(x1-x0)
41:     SGN      d4      *d4 = sx = sgn(x1-x0)
42:
43:     sub.w      d1, d3      *d3 = y1-y0
44:     move.w      d3, d5      *d5 = d3
45:     ABS      d3      *d3 = dy = abs(y1-y0)
46:     SGN      d5      *d5 = sy = sgn(y1-y0)
47:
48:     add.w      d4, d4      *d4 = sx*2
49:     moveq.l      #GSFICTR, d0      *
50:     asl.w      d0, d5      *d5 = sy*1024 (or 2048)
51:
52:     move.w      COL(a5), d0      *d0 = 描画色
53:
54:     cmp.w      d3, d2      *dy>dxならば
55:     bcs      yline      * yについてループ
56:
57:     *dx≥dyのとき
58: xline:   move.w      d2, d1      *d1 = d2
59:         neg.w      d1      *d1 = e = -dx
60:         move.w      d2, d6      *d6 = n = dx
61:         add.w      d2, d2      *d2 = dx*2
62:         add.w      d3, d3      *d3 = dy*2
63:         *do {
64: xline0:  rol.w      #1, d7      *
65:         bcc      xskip      *

```

```

66:      move.w  d0, (a0)      # pset(x,y)
67: xskip:  adda.w  d4, a0      # x += sx
68:      add.w   d3, d1      # e += 2*dy
69:      bmi     xline1      # if (e >= 0) {
70:      adda.w  d5, a0      #   y += sy
71:      sub.w   d2, d1      #   e -= 2*dx
72:      # }
73: xline1:  dbra   d6, xline0  #} while (--n >= 0)
74:      bra     done
75:
76:      #dx<dyのとき
77: yline:   move.w  d3, d1      #d1 = d3
78:      neg.w    d1      #d1 = e = -dy
79:      move.w    d3, d6      #d6 = n = dy
80:      add.w     d2, d2      #d2 = dx*2
81:      add.w     d3, d3      #d3 = dy*2
82:      #do {
83: yline0:  rol.w   #1, d7      #
84:      bcc     yskip      #
85:      move.w  d0, (a0)      # pset(x,y)
86:      adda.w  d5, a0      # y += sy
87:      add.w   d2, d1      # e += 2*dx
88:      bmi     yline1      # if (e >= 0) {
89:      adda.w  d4, a0      #   x += sx
90:      sub.w   d3, d1      #   e -= 2*dy
91:      # }
92: yline1:  dbra   d6, yline0  #} while (--n >= 0)
93: done:    movem.l (sp)+, d0-d7/a0-a5
94:      rts
95:      .end

```

glineの個々の版を比較してみよう。最初に施した最適化は有効に働いている。両端からの同時描画もけっこう効果がある。水平/垂直/45°の線分描画部分のループ展開はほとんど効いてはいないが、これは予想できたことだ。表の差は、ランダムな線分が偶然、水平/垂直/45°になる確率を表しているにすぎない。実用上は、もう少し水平/垂直線分を描く機会も増えるだろう。ダブルステップ化はそれなりに有効だが、プログラムの複雑さに見合うかどうかは微妙なところだ。実際、リスト8の版もループをある程度展開すれば、ダブルステップ版と同水準の性能はひねり出せる。それでも、精度を落とさず、ループ展開などの姑息な手も使わずにここまで速くなるのを見ると、よいアルゴリズムを選ぶことの重要性をあらためて感じてしまう。

65536倍整数化版は、リスト8の版と比べ、確かに精度を犠牲にした分だけ速い。ただ、ダブルステップとの相性はよくないようだ。ここでは示さなかったが、単純に65536倍整数化版をダブルステップ化すると、整数化ダブルステップ版との差は事実上なくなる。リスト15とリスト16の差は「短線分+1ピクセル方式」の分だと思ったほうがよい。65536倍整数化Bresenham版はへたにダブルステップにするよりも、ループの中身の軽さを利用してループ展開したほうが効率はいいようだ。

「短線分+1ピクセル方式」は線分が極端になだらかか急な場合にしか有効ではないし、といて、リスト16のように線分の傾きによってアルゴリズムを切り替えると傾きごとに描かれる線分の質が変わってしまうという問題はあるものの、ダブルステップ版まで煮詰めた時点からまだこれだけの速度を稼げるという点で、スピード狂向きではある。もっとも、リスト16では短線分描画部分をまともにループ展開してあるから、ほかの版よりもかなりのメモリを食っ

ていることを指摘しておく必要があるだろう。

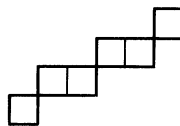
C COLUMN

4 連結の線分発生アルゴリズム [参考]

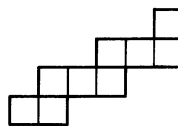
この章では、ピクセルが縦・横だけでなく斜めに接することを許す、いわゆる8連結の線分発生アルゴリズムに絞って話を進めてきたが、ときには、ピクセルがかならず縦か横に並ぶ4連結のアルゴリズムが有用な場合もある(図A)。その4連結の線分発生アルゴリズムは、8連結のアルゴリズムを多少修正することでかんたんに得られる。8連結のアルゴリズムではx, y座標を同時に更新して斜めに移動していた部分を、縦横の2回に分けて移動するようにすればよい。リストAに4連結ラインルーチンのX-BASIC版、ついでに、リストBにマシン語版を示しておく。リスト4, リスト5と見くらべてもらいたい。

図A 8連結の線分と4連結の線分

a) 8連結



b) 4連結



リストA LINE4. BAS

```

10 int x0 = 0, y0 = 0
20 int x1 = 5, y1 = 3
30 int e, x, y
40 int dx, dy, sx, sy, i
50 /*
60 screen 0,0,1,1
70 dx = abs(x1-x0): dy = abs(y1-y0)
80 sx = sgn(x1-x0): sy = sgn(y1-y0)
90 x = x0: y = y0
100 if (dx >= dy) then {
110     e = 2*dy - dx
120     for i = 0 to dx + dy
130         pset(x, y, 15)
140         if (e >= 0) then {
150             y = y + sy
160             e = e - 2*dx
170         } else {
180             x = x + sx
190             e = e + 2*dy
200         }
210     next
220 } else {
230     e = 2*dx - dy
240     for i = 0 to dx + dy
250         pset(x, y, 15)
260         if (e >= 0) then {
270             x = x + sx
280             e = e - 2*dy
290         } else {
300             y = y + sy
310             e = e + 2*dx
320         }
330     next
340 }
```

リストB GLINE4.S

```

1: *      線分描画 (4連結)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6:      .xdef      gline4
7:      .xref      gramadr
8:      .xref      gclipline
9: *
10:     .offset 0      *gline4の引数構造
11: *
12: X0:     .ds.w      1      *始点座標
13: Y0:     .ds.w      1      *
14: X1:     .ds.w      1      *終点座標
15: Y1:     .ds.w      1      *
16: COL:    .ds.w      1      *描画色
17: *
18:     .text
19:     .even
20: *
21: gline4:
22: ARGPTR =      4+8*4+6*4
23:     movem.l      d0-d7/a0-a5, -(sp)
24:
25:     movea.l      ARGPTR(sp), a5      *a5 = 引数列
26:     movem.w      (a5)+, d0-d3      *d0~d3に座標を取り出す
27:
28:     jsr      gclipline      *クリッピングする
29:     bne      done      *Z=0なら完全にウィンドウ外
30:
31:     jsr      gramadr      *a0 = 始点のG-RAMアドレス
32:
33:     sub.w      d0, d2      *d2 = x1-x0
34:     move.w      d2, d4      *d4 = d2
35:     ABS      d2      *d2 = dx = abs(x1-x0)
36:     SGN      d4      *d4 = sx = sgn(x1-x0)
37:
38:     sub.w      d1, d3      *d3 = y1-y0
39:     move.w      d3, d5      *d5 = d3
40:     ABS      d3      *d3 = dy = abs(y1-y0)
41:     SGN      d5      *d5 = sy = sgn(y1-y0)
42:
43:     add.w      d4, d4      *d4 = sx*2
44:     moveq.l      #GSFICTR, d0      *d5 = sy*1024
45:     asl.w      d0, d5      * (or sy*2048)
46:
47:     move.w      (a5), d0      *d0 = 描画色
48:
49:     move.w      d2, d6      *d6 = dx
50:     add.w      d3, d6      *d6 = dx+dy
51:     *      = ループカウンタ
52:
53:     cmp.w      d3, d2      *dy>dxならば
54:     bcs      yline      * yについてループ
55:
56:     *dx≥dyのとき
57: xline:   move.w      d2, d1      *d1 = e = -dx
58:     neg.w      d1      *
59:     add.w      d2, d2      *d2 = 2*dx
60:     add.w      d3, d3      *d3 = 2*dy
61:     add.w      d3, d2      *d2 = 2*dx+2*dy
62:     sub.w      d4, d5      *d5 = yを更新したときの
63:     *      G-RAMアドレス変化量
64:     * (xを先に更新した分の補正を含む)
65:     *do {
66: xline0:  move.w      d0, (a0)      * pset(x, y)
67:     adda.w      d4, a0      * x += sx
68:     add.w      d3, d1      * e += 2*dy
69:     bmi      xline1      * if (e >= 0) {
70:     adda.w      d5, a0      * y += sy, x -= sx
71:     sub.w      d2, d1      * e -= 2*dx+2*dy

```

```

72:                                     * }
73: xline1: dbra    d6, xline0      *} while (--n >= 0)
74:         bra     done
75:
76:                                     * dx<dyのとき
77: yline:  move.w  d3, d1
78:         neg.w   d1
79:         add.w   d2, d2
80:         add.w   d3, d3
81:         add.w   d2, d3
82:         sub.w   d5, d4
83:
84: yline0: move.w  d0, (a0)
85:         adda.w  d5, a0
86:         add.w   d2, d1
87:         bmi     yline1
88:         adda.w  d4, a0
89:         sub.w   d3, d1
90:
91: yline1: dbra    d6, yline0
92: done:   movem.l (sp)+, d0-d7/a0-a5
93:         unlk    a6
94:         rts
95:
96:         .end

```

リストA, Bはともに同じアルゴリズムをプログラムにしたものだが、効率上、マシン語版では誤差項を更新するタイミングをX-BASIC版とは変えてある。X-BASIC版は誤差項の符号におうじてxを更新するか、yを更新するかを選択するという正直なつくりをしているが、マシン語版は毎回x座標をかりに進め、それにあわせて誤差項も更新してから、誤差項の符号を調べるようになっている。その結果、じつはx座標ではなくy座標を更新するタイミングだということがわかったら、進めすぎた分を逆に修正するのだ。リストBでは、y座標を進めたときの誤差項とG-RAMアドレスそれぞれの変化量に、あらかじめ修正分を加味している(61~62行)ことに注目しよう。

C
H
A
P
T
E
R

4

多
角
形
フ
ィ
リ
ン
グ

この章では多角形の塗り潰しを取り上げる。塗り潰しといっても、輪郭を描いてから内側の1点を基点に色を塗っていく、いわゆるペイントではない。スキャンコンバージョン (scan conversion) という手法を使い、最初から中身の詰まった多角形を描くというアプローチを採る。スキャンコンバージョンとは、図形を画面に水平な線分に細分し、その水平線分を並べていくことで描画する方法だ。市販グラフィックエディタの閉曲線ペイントなどに応用例を見ることができる。

スキャンコンバージョン

スキャンコンバージョンのアルゴリズムには驚くような仕かけは何もない。図形の輪郭とスキャンライン (scanline: 走査線、つまり画面の横1ライン) との交点を算出しては求めた点の間を水平線分でつないでいく、という処理を各スキャンラインに対して機械的に繰り返すだけだ (図1)。交点がいくつも存在するときには、x座標の小さい順に2点ずつ選んで、その2点間を結んでいくものとする。結果、ちょうど、スキャンラインを左から見ていって、交点が見れるたびに色を置くか置かないかを反転する感じになる (図2)。このため、図形の内側の閉じた領域は塗り潰されない (図3)。

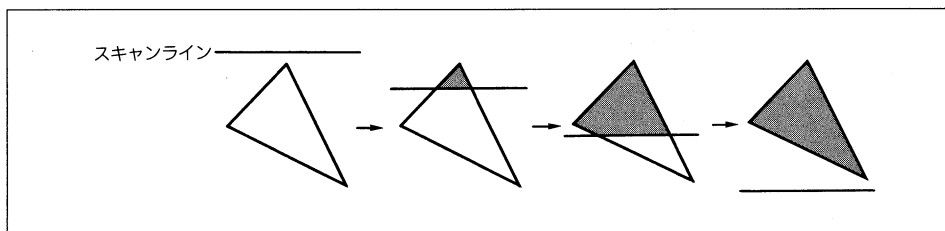


図1
スキャンコンバージョン

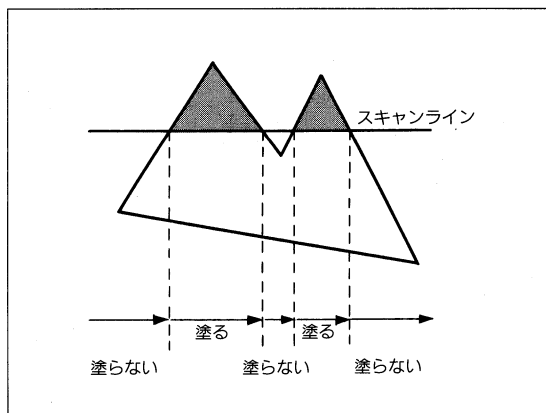


図2
交点間の結び方

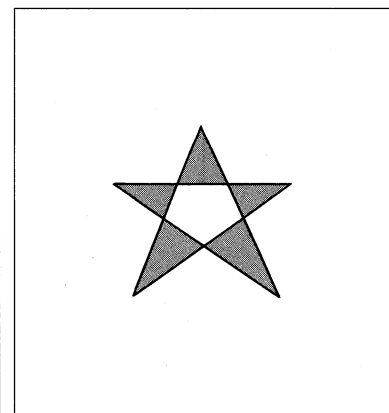


図3
スキャンコンバージョンの特性

スキャンコンバージョンの考え方は円の塗りつぶし描画などにも適用することができるが（円についてはChapter 6で取り扱う）、ここでつくるのはあくまで多角形を描くルーチンであり、輪郭は直線だけからなることが前提になる。この場合、図形とスキャンラインとの交点は、多角形を構成するすべての辺とスキャンラインとの交点を求めることで得られる。交点を求める段階では、多角形であることを忘れ、各辺を独立したものとして扱ってかまわない。ただし、辺を処理する順序によっては、交点がx座標の小さい順に求まるとはかぎらないので、求めた交点は適当なメモリ領域（スキャンラインバッファと呼ばれる）にいったんためておいて、スキャンライン1本分が全部揃ってからx座標の小さい順に並べ換えることが必要になる。

アルゴリズムの大筋は以上のとおりのシンプルなものだ。だが、実現にあたっては2, 3気をつけなければならないことがある。

まず、交点を2個対にして取り扱う都合上、各スキャンラインごとの交点の数は偶数個でなければうまくない。輪郭が閉じていない図形では、交点数が奇数のスキャンラインで図4のような不自然な結果を生じる。いちばん右の交点と対になる点がないため、色を置/置かないのサイクルが尻切れで終わってしまうわけだ。この事態を避けるため、図形の輪郭が閉じていない場合は描画サブルーチン側で最初の点と最後の点をつなぐ辺を補うことになるだろう。

輪郭が閉じていれば丸く収まるかというところでもない。多角形の頂点では微妙な問題が発生する。図5を見てもらいたい。各辺を独立したものとして扱い、個別にスキャンラインとの交点を求めると、頂点において多角形はスキャンラインと2度交わる。図5の頂点Aの場合だと、辺ABとの判定で1回、辺CAとの判定で1回、計2回の判定により、点Aの座標がダブってスキャンラインバッファに格納される。頂点A, Bを通るスキャンラインでは、このダブリのお陰で交点の数が偶数に保たれ、期待どおりの描画結果が得られる。ところが、頂点Cを同じように交点2個にカウントしてしまうと正しい結果にならない。上向きや下向きではない頂点は交点1個と数える必要があるのだ。

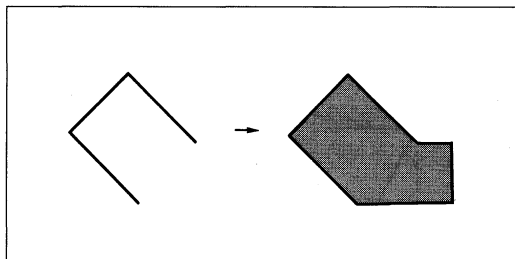


図4
輪郭の閉じていない多角形

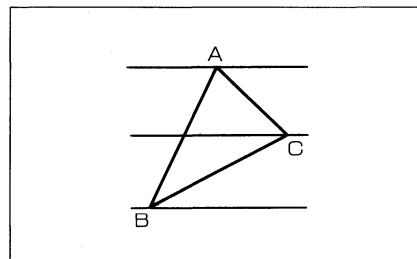


図5
頂点を通るスキャンライン

この問題を真面目に回避しようと思ったら、頂点が上下を向いているのか左右を向いているのかを前後の頂点との関係から判定して、泥臭くつじつまをあわせることになるだろう。しかし、多少不正確になってもかまわなければ、比較的ダメージの少ない手抜き手段がある。辺の長さを1ピクセル分短く見積もって、片方の端点についてはスキャンラインとの交わり判定を省略してしまうという方法だ。かりにy座標の大きなほうの端点を削る（その寸前で辺が終わるものとする）ことにすると、上向きの頂点には辺が2本、下向きの頂点には0本、それ以外の頂点には1本の辺が集まることになる（図6）。線分の長さを誤魔化す分、下向きの頂点が

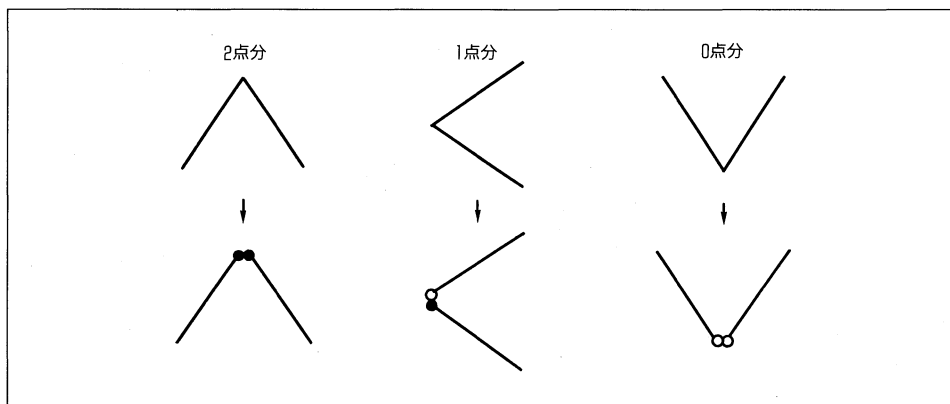


図6
頂点での交点の数え方

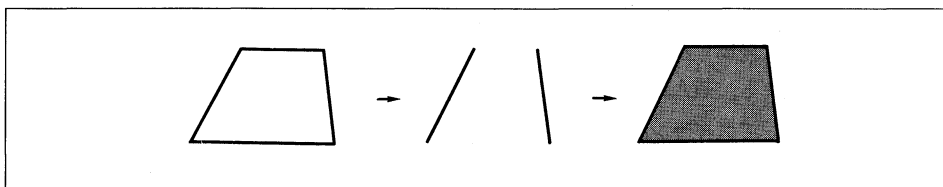


図7
水平な辺の扱い

欠けてしまうのが（言葉どおり）欠点だが、とりあえず、交点の数のバランスだけは保たれる。頂点以外では、水平な辺の扱いにも注意が要る。水平な辺はスキャンラインと重なってしまい、ほかの辺と同じように交点を求めるわけにはいかない。どう対処するのが正しいかというと、無視するのが正解だ。水平な辺は前処理段階で削る。削除しても、その辺を挟む辺どうしが水平線分で結ばれるから、描画結果には影響しない（図7）。

プログラミングに向けて

アルゴリズムを軽く押さえたところで、実際にプログラムにするうえで問題になりそうな細部を煮詰めていこう。

多角形の内部表現

最初に、多角形をメモリ上でどのように表現するか決めておく。描画サブルーチンを呼び出す側から見れば、各頂点の座標をずらずら並べた配列状のデータとして多角形を用意し、サブルーチンに引き渡すというかたちが楽そうに思う。隣りあった2点が辺をかたちづくり、また、最後の点は最初の点と結ばれているものとする。実際にはデータの長さがわからないと困るので、先頭に頂点の個数を格納するとして。今回は、頂点の個数格納用に2バイト、1個の頂点の座標格納用に4バイト（x, y座標に各2バイト）の図8の形式に決める。座標の範囲が-32768～+32767の65535角形までを表現できるというわけだ。

+00	IW	頂点の個数(=n)
+02	IW	頂点1 のx座標
+04	IW	頂点1 のy座標
+06	IW	頂点2 のx座標
+08	IW	頂点2 のy座標
⋮		
+??	IW	頂点nのx座標
+??	IW	頂点nのy座標

図8
多角形の内部表現

さて、いま決めた頂点の列形式の多角形表現は、スキャンコンバージョンの処理過程ではあまり扱いやすいとはいえない。効率よくスキャンラインと辺との交点が求められるよう、処理中は、頂点ではなく辺を単位に扱いたい。そこで、描画に先立って、頂点の列から辺のリストを作成することを考える。辺リストの構造については少しあとで述べよう。

クリッピング

クリッピングすることも考慮しておく。各辺のクリッピングには線分描画のときと同じ手法が使えるが、多角形の場合は、クリッピングしても頂点どうしの関係（どの頂点とどの頂点が辺で結ばれているか）が変わらないよう注意する必要がある。もっとも、多角形が頂点の列のかたちで表現されているなら、並べられたままの順序でクリッピングしていけば、少なくとも頂点を結ぶ順序が乱れることはない。図9に示した三角形を頂点A, B, Cの順にクリッピングすると、点A, a, b, Cが得られ、ごく自然に点aとbをつなぐ辺が補われるかたちとなる。

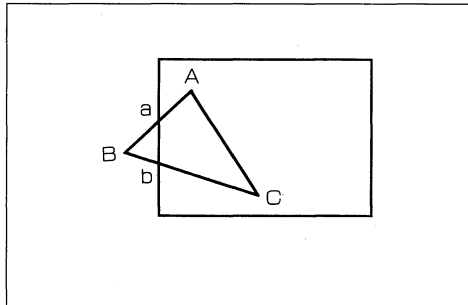


図9
多角形のクリッピング

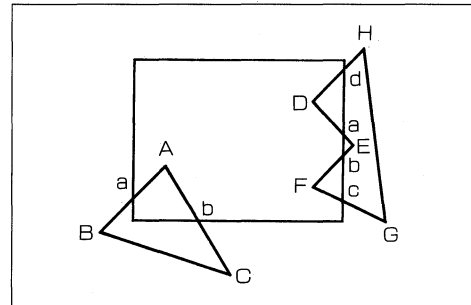


図10
問題のある多角形

だが、安心するのはまだ早い。たとえば、図10左側の三角形の場合、単純にクリッピングしても点A, a, bしか得られず、aとbを結ぶと、結果は妙なことになる。クリッピングウィンドウの隅の点を補えばよいのだが、元の図形の辺上にない点を補うのには工夫が要る。さらに、クリッピングすることによって2つ以上の部分に分割される図形では、分割した図形をつなぐ辺が勝手に生成されることがある。図10右側の凹多角形の場合、クリッピングすると点D, a, b, F, c, dが求まるが、これらの点をぐりりとつなぐと、本当はあってはならない辺abが生まれてしまう。

図10左側の三角形については、段階を踏んでクリッピングすることで正しい結果を求めることができる。まず、クリッピングウィンドウの1辺（の延長線）でクリッピングして中間形をつくり、順次残りの辺でクリッピングしていけばよい。また、一般解ではないが、スキャンコンバージョンのためのクリッピングであれば、x座標とy座標のクリッピングのタイミングをずらすことで、複数に分割される図形にも対処できる。前処理段階ではy座標についてのみクリッピングを行い、描画の時点になってからx座標でクリッピングする。この2回目のクリッピングは、スキャンラインと図形との交点を求めてその交点間を結ぶとき、描画のぎりぎり直前に行く。これにより、“幻の垂直線”は生まれなくなる。幻の水平線はまだ残るが、クリッピング後、水平線は削除するから、やはり問題にはならない。

多角形とスキャンラインとの交点算出

辺とスキャンラインの交点は数学的な手法で1点ずつ計算してもよいが、もっと効率のよい方法がある。Chapter3で紹介したBresenhamの線分発生アルゴリズムを使い、辺に沿って端から順に求めていくのだ。ここで、スキャンラインの本数分のスキャンラインバッファが用意できれば、あらかじめ多角形とスキャンラインとの全交点を求めて記憶しておくことも可能だが、メモリの消費量を考えると、実用上はスキャンラインバッファをあくまで1本分に抑えて使い回したい。その場合、複数の辺に平行してBresenhamのアルゴリズムを適用するかたちになるから、線分発生に必要なパラメータ（誤差項やその増分など）は各辺ごとに構造体様のデータにまとめておく必要が出てくる。

ここではリスト1のEDGE.Hで定義するような構造体に各辺ごとの情報を格納することにした。いい加減につけたラベル名から、以下この構造体をEDGBUF構造体と呼ぶ。EDGBUFの構造はあとのプログラムに都合のよいよう、項目の位置や順序を決めてある。ここでは、とくにx座標を先頭にもってきている点に注目してほしい。通常、構造体の各項目は、構造体の先頭アドレスとメンバーへのオフセットによるディスプレイメントつきアドレスレジスタ間接形式を使ってアクセスする。たとえば、

```
move.w    DY (a0),d0
```

```
move.w    d1,E0 (a0)
```

のようにだ（当然、a0には構造体の先頭アドレスを入れておく）。しかし、オフセットが0のメンバー、つまり、先頭の1項目だけは、単純なアドレスレジスタ間接形式で少しだけ高速にアクセスできる。そこで、もっともよく参照されるx座標を先頭にしているというわけだ。なお、このような小細工はいつまでも覚えておけるものではないので、リスト1では、将来の改良時に誤って項目の位置関係を変えてしまわないよう、注釈をつけてある。

リスト1
EDGE.H

```
1: #      EDGBUF構造体
2:
3:      .offset 0
4: #
5: X:      .ds.w 1      *スキャンラインとの交点のx座標
6:      * (初期値はX0)
7: DEY:    .ds.w 1      *誤差項の補正值 dy*2
8: DEX:    .ds.w 1      *誤差項の増分 dx*2
9: DY:     .ds.w 1      *ループカウンタ
10:      * (初期値はDY0)
11: E:      .ds.w 1      *誤差項 0
12:      * (初期値はE0)
13: SX:    .ds.w 1      *傾きの符号 sgn (x1-x0)
14: X0:    .ds.w 1      *始点のx座標 x0
15: Y0:    .ds.w 1      *始点のy座標 y0
16: DY0:   .ds.w 1      *線分の長さ y1-y0
17: E0:    .ds.w 1      *誤差項の初期値 e0
18: EDGBUFsiz:
19: #
20: *NOTE: genedgelist, gfillpolyはXが先頭にあることを仮定している
21: #      gfillpolyは
22: #      XとDEY DYとE DY0とE0
23: #      が並んでいることを仮定している
```

```

24: #
25: .text

```

さらに具体的な処理に踏み込もう。

すべての辺がつねにスキャンラインと交わるわけではないから、交点の算出は“スキャンラインと交わることがわかっている辺”に対してだけ行う必要がある。この問題は、EDGBUFをアクティブな（活性化した）ものと未アクティブなものに分けて処理することで解決する。前提として、始点がかならず終点より上にくるよう、必要なら始点と終点を交換しておく。初期状態では、各EDGBUFは未アクティブな群にまとめられ、始点のy座標が処理中のスキャンラインと一致した時点でアクティブな群に移す。また、アクティブだったEDGBUFはスキャンラインが終点のy座標に一致した時点で削除する。交点の算出は、このアクティブなEDGBUF群に対してのみ行う。

群にまとめる、という表現を使ったが、処理上は、EDGBUFを実際に移動したりしなくてもフラグを設ければ十分だ。このフラグは初期化時に未アクティブにセットされ、やがてアクティブになり、最後には処理済みの印へと変わる。だが、今回はあえてフラグ方式はとらず、実際に群にまとめる方法をとった。未アクティブなEDGBUF、アクティブなEDGBUFそれぞれにつき、おのこのEDGBUFを指すポインタの配列を用意する。未アクティブな群からアクティブな群への移動は、このポインタの配列のうで行う。図11にイメージを示しておく。

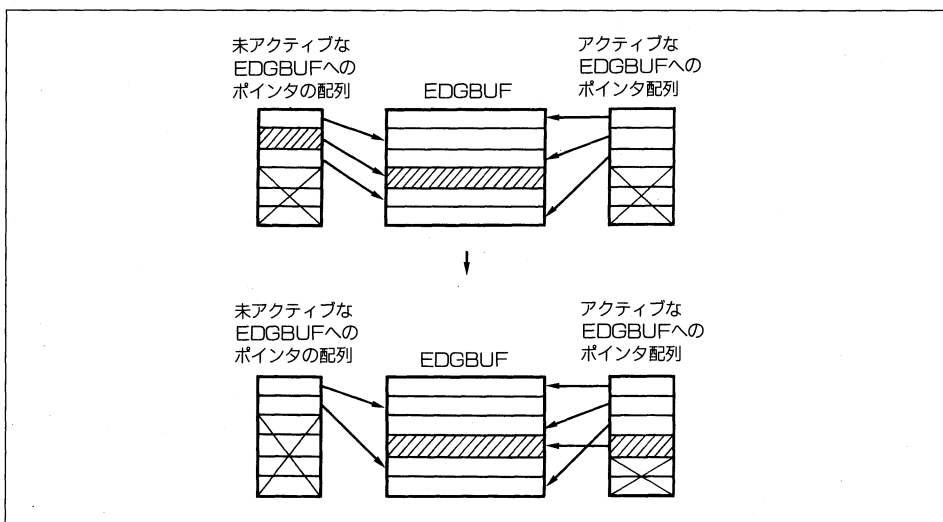


図 11
辺リストのポインタ配列

交点のソーティング

ソーティング (sorting: 整列) のアルゴリズムには単純な方法から凝ったものまでさまざまなものがある。現在、真面目にデータの比較・交換を繰り返すものとしては、クイックソート (quick sort) と呼ばれるアルゴリズムが最高速であることが知られている。だからといって、なんでもかんでもクイックソートでソートすればよいというものではない。クイックソートにかぎらず、高速といわれるソーティングアルゴリズムは、データ数がある程度以上多いときに性能を発揮するよう考えられている。データ数が非常に少ない場合はアルゴリズムの複雑さが災いして、単純な (普通は遅いとされる) アルゴリズムよりも実行速度が劣ることもある。

いま作成しようという多角形塗り潰しルーチンは最大65535角形まで描けるわけだが、その限界に近い図形を描くのはまれで、現実にはもっとずっと辺の数が少ない多角形を描くのに使われるだろう。ということは、データの個数が比較的少ないときに高速に動作する、単純なソートアルゴリズムを利用する場面といえそうだ。ここでは単純挿入法 (straight insertion) と呼ばれるアルゴリズムを採用した。単純挿入法では、データをかりにソートずみの部分と未ソートの部分に分けておき、未ソート要素を1個ずつソートずみの部分の適切な位置に挿入することでソートする。参考までに、図12に適当な8個のデータを単純挿入法でソートする様子を示しておく。

初期状態	51	45	76	23	38	16	83	60
	45	51	76	23	38	16	83	60
	45	51	76	23	38	16	83	60
	23	45	51	76	38	16	83	60
	23	38	45	51	76	16	83	60
	16	23	38	45	51	76	83	60
	16	23	38	45	51	76	83	60
ソート完了	16	23	38	45	51	60	76	83

図12
単純挿入法の動作

ところで、これまでの話だと、スキャンラインと辺の交点はスキャンラインバッファに格納して、ソーティングはそこで行うことになる。しかし、後で示すプログラムにはスキャンラインバッファが表立っては出てこない。求めた交点のx座標はそのままEDGBUF構造体に格納しておいて、EDGBUF構造体ごとソートする。正確には、ソーティングは構造体そのものではなく構造体へのポインタ配列に対して適用する。ソートする過程でデータの位置を交換するときには、データ自体は動かさずにポインタだけをつなぎ替えるわけだ。アクティブかどうかをフラグで表さずにポインタ配列を用意した理由がここにある。では、そういう細工をしてまで、スキャンラインバッファを用意せずにEDGBUF構造体ごとソートするようにしたのはなぜか、説明せねばなるまい。

いま、あるスキャンラインと辺1、辺2が交わり、交点 P_1 、 P_2 がこの順序で得られたとする。 $P_2 < P_1$ の場合は並べ替えて P_2 、 P_1 の順序にしてから描画するわけだ。ここで、この辺1と辺2の位置関係は次のスキャンラインでも変わらない確率が高いということに注目しよう。たぶん、次のスキャンラインでも辺2は辺1よりも左にある。ソートのときにEDGBUF構造体ごと位置を交換してしまえば、次のスキャンラインでは辺1よりも辺2が先に処理されることになり、交点は最初から小さい順に求まる。すでにソートする必要がないわけだ。プログラム上は毎回ソートルーチンを通ることにはなるが、単純挿入法はソートずみのデータに減法強く、1回データを通読した時点ですぐに処理を完了する。多くのソーティングアルゴリズムではソートずみだということを認識するまでに余計な処理が必要で、こううまくはいかない。

理屈のうえでは、この方法はなかなか効率がいはいはずだが、EDGBUFをポインタで間接的に参照する分の手間が処理速度を落とす可能性もある。で、実際に2種類のプログラムをつくらせて速度をくらべてみた。ランダムな多角形を描かせた場合、四～五角形ぐらいまでは、アクティブかどうかをフラグで表しソートはスキャンラインバッファ上で行う方法のほうがいくらか速かった。描く図形の大きさにもよるが、五～六角形ぐらいでとんとんになり、以降はEDGBUFのポインタ配列を用意してソートもポインタ配列上で行うほうが速くなる。ここで選択した方法は、速くするためというよりは、少しぐらい辺の数が増えてもなるべく遅くしないための工夫といえるだろう（三角形もクリッピング結果によってはあつという間に六角形に化けることを考えてみてほしい）。

実装

というあたりでプログラムを見てもらおう。考えがあって、プログラムは大きく3つのサブルーチンに分割した。第1のサブルーチン`gclippoly`では描く多角形の（y座標に関する）クリッピングのみを行う。第2のサブルーチン`genedgelist`はクリッピング後の頂点列から先ほど示した構造の辺リストを作成する。第3のサブルーチン`gfillpoly`はこの辺リストを参照して実際に多角形を描画する。これら3つのサブルーチンを続けて呼び出すことで1つの多角形が描かれる。

このような構成にしたのは、処理単位を明確にするという教育的意味も少しあるが、変則的な図形に対応できるようにするためでもある。メインルーチン側で工夫すれば、輪郭が一筆書きできない（たとえば穴の空いた）図形も描けるようになっている。図形の輪郭を構成する個々の多角形を別データとして用意し、個別に辺リストの作成までを行ってから、作成した辺リストをまとめて`gfillpoly`に渡せばよい。また、同一の図形を色だけ変えて再描画する場合（たとえば、描いた図形を消す場合）、`gclippoly`、`genedgelist`の呼び出しは1回だけで済むというメリットもある。

クリッピング部 (リスト2)

リスト2
GCLIPPOLY.S

```

1: *      多角形をcliprectで指定された矩形領域でクリッピングする
2:
3:      .include      rect.h
4: *
5:      .xdef      gclippoly
6:      .xref      cliprect
7: *
8:      .text
9:      .even
10: *
11: gclippoly:
12: POINTS =      8
13: POINTS2 =     12
14:      link      a6, #0
15:      movem.l    d0-d7/a0-a5, -(sp)
16:
17:      movem.l    POINTS(a6), a1-a2
18:                                     *a1 = クリッピング前の点
19:                                     *a2 = クリッピング後の点
20:      lea.l      cliprect, a0      *a0 = クリッピングウィンドウ
21:
22:      clr.w      (a2)+              *仮に頂点の数を0にしておく
23:
24:      move.w      (a1)+, d7          *d7 = 点の数
25:      beq         retn              *点が与えられていない
26:      subq.w      #1, d7            *
27:      bne         do                *
28:
29:                                     *1点しか与えられていない場合
30:      movem.w      (a1), d0-d1
31:      cmp.w        MINY(a0), d1      *y < MINY なら
32:      blt          done              * 不可視
33:      cmp.w        MAXY(a0), d1      *y > MAXY なら
34:      bgt          done              * 不可視
35:      move.w        d0, (a2)+         * (x, y) を登録する
36:      move.w        d1, (a2)+         *
37:      bra          done              * クリッピング完了
38:
39:                                     *2点以上与えられていた場合
40: do:      moveq.l    #0, d0
41:      move.w        d7, d0
42:      lsl.l        #2, d0
43:
44:      movem.w      0(a1, d0.l), d2-d3
45:                                     * (d2, d3) = 最後の点
46:                                     *              = 最初の点にとっては
47:                                     *              直前の点
48:      movea.w      #$8000, a5        *a5 = ゲタ
49:      bra          lpent
50:                                     *クリッピングする点を (x, y)
51:                                     *直前の点を (x0, y0) とおく
52: loop:      movem.w      (a1)+, d2-d3      * (d2, d3) = (x0, y0)
53: lpent:      movem.w      (a1), d0-d1      * (d0, d1) = (x, y)
54:      moveq.l      #0, d4              *d4 = 直前の点が可視かどうかのフラグ
55:                                     *              (仮に可視と考える)
56:
57: cminy:      move.w        MINY(a0), d5      *d5 = MINY
58:      cmp.w        d5, d1              *y < MINY なら
59:      blt          cminy0              * (x, y) をMINYでクリップ
60:
61:      cmp.w        d5, d3              *y >= MINY かつ y0 >= MINY なら
62:      bge          cmaxy              * MINYでのクリッピングは不要
63:
64:                                     * (x, y) は可視で (x0, y0) は不可視
65:      exg.l        d0, d2              * (x0, y0) をMINYでクリップ
66:      exg.l        d1, d3              *
67:      bsr          minyclip            *
68:      exg.l        d0, d2              *

```

```

69:      exg. l   d1, d3          *
70:      moveq. l #-1, d4        * (x0, y0) をクリッピングした
71:      bra      cmaxy
72:
73: cminy0: cmp. w   d5, d3          * y < MINY かつ y0 < MINY なら
74:      blt      next            * 辺は完全不可視
75:
76:      * (x, y) は不可視で (x0, y0) は可視
77:      bsr      minyclip         * (x, y) をMINYでクリップ
78:
79: cmaxy: move. w   MAXY (a0), d5    * d5 = MAXY
80:      addq. w   #1, d5            * d5 = MAXY+1 = MAXY'
81:      * (gfllpolyの手抜き対応)
82:      cmp. w    d5, d1            * y > MAXY' なら
83:      bgt      cmaxy0           * (x, y) をMAXY' でクリップ
84:
85:      cmp. w    d5, d3            * y ≤ MAXY' かつ y0 ≤ MAXY' なら
86:      ble      setp             * MAXY' でクリッピングは不要
87:
88:      * (x, y) は可視で (x0, y0) は不可視
89:      exg. l    d0, d2            * (x0, y0) をMAXY' でクリップ
90:      exg. l    d1, d3            *
91:      bsr      maxyclip         *
92:      exg. l    d0, d2            *
93:      exg. l    d1, d3            *
94:      moveq. l  #-1, d4          * (x0, y0) をクリッピングした
95:      bra      setp0
96:
97: cmaxy0: cmp. w   d5, d3          * y > MAXY' かつ y0 > MAXY' なら
98:      bgt      next            * 辺は完全不可視
99:
100:      * (x, y) は不可視で (x0, y0) は可視
101:      bsr      maxyclip         * (x, y) をMINYでクリップ
102:
103: setp:  tst. w    d4              * (x0, y0) をクリッピングしたのなら
104:      beq      setp1            *
105: setp0: move. w   d2, (a2) +      * クリッピングした (x0, y0) も
106:      move. w   d3, (a2) +      * 登録する
107: setp1: move. w   d0, (a2) +      * クリッピングした (x, y) を登録する
108:      move. w   d1, (a2) +      *
109:
110: next:  dbra     d7, loop        * すべての点について繰り返す
111:
112: movea. l POINTS2 (a6), a1      * 最初の点を
113: move. w   2 (a1), (a2) +      * バッファ最後に書き込み
114: move. w   4 (a1), (a2) +      * 図形を閉じる
115:
116: done:  movea. l POINTS2 (a6), a1 * クリッピング後の
117:      addq. l   #2, a1          *
118:      suba. l   a1, a2          *
119:      move. l   a2, d0          *
120:      lsr. l    #2, d0          * 点の数を数える
121:
122: swap. w   d0                  * 点の数が65536以上なら
123:      tst. w    d0              *
124:      beq      setn            *
125: moveq. l   #0, d0              * 誤動作防止に0にしてしまう
126:
127: setn:  swap. w   d0            *
128:      move. w   d0, -(a1)        * 点の数を記録する
129:
130: movem. l   (sp) +, d0-d7/a0-a5
131:      unlk     a6
132:      rts
133: *
134: maxyclip: * MAXY' でクリッピングする
135:      cmp. w    d5, d3          * 反対側の端点がMAXY' 上なら
136:      beq      just            * その点が求める点
137:
138:      move. w   d2, a3          * 反対側の端点の
139:      move. w   d3, a4          * 座標を保存
140:      add. w    a5, d0          * 両端点とMAXY' に
141:      add. w    a5, d1          * 8000hのゲタを覆かせて

```

```

142:      add.w  a5, d2      * 無符号化する
143:      add.w  a5, d3      *
144:      add.w  a5, d5      *
145:
146:                                *クリッピングする点を(x1, y1)
147:                                *反対側の点を(x2, y2)
148:                                *両者の中点を(mx, my)とおく
149: maxylp: move.w d1, d6
150:      add.w  d3, d6
151:      roxr.w #1, d6      *d6 = my = 中点のy座標
152:      cmp.w  d5, d6      *my = MAXY' なら
153:      beq    yclipq      * クリッピング完了
154:      bcs    maxy0
155:                                *y1 < my < MAXY' < y2
156:      move.w d6, d1      *y1 = my
157:      add.w  d2, d0
158:      roxr.w #1, d0      *x1 = mx
159:      bra    maxylp      * と更新して繰り返す
160:
161:                                *y1 < MAXY' < my < y2
162: maxy0: move.w d6, d3      *y2 = my
163:      add.w  d0, d2
164:      roxr.w #1, d2      *x1 = mx
165:      bra    maxylp      * と更新して繰り返す
166: *
167: minyclip:                    *MINYでクリッピングする
168:      cmp.w  d5, d3
169:      beq    just
170:
171:      move.w d2, a3
172:      move.w d3, a4
173:      add.w  a5, d0
174:      add.w  a5, d1
175:      add.w  a5, d2
176:      add.w  a5, d3
177:      add.w  a5, d5
178:
179: minylp: move.w d1, d6
180:      add.w  d3, d6
181:      roxr.w #1, d6
182:      cmp.w  d5, d6
183:      beq    yclipq
184:      bcc    miny0
185:
186:      move.w d6, d1
187:      add.w  d2, d0
188:      roxr.w #1, d0
189:      bra    minylp
190:
191: miny0: move.w d6, d3
192:      add.w  d0, d2
193:      roxr.w #1, d2
194:      bra    minylp
195:
196: yclipq: move.w d6, d1      *d1 = 求めたy座標
197:      add.w  d2, d0
198:      roxr.w #1, d0      *d0 = それに対応したx座標
199:
200:      sub.w  a5, d0      *ゲタを脱がせる
201:      sub.w  a5, d1
202:      move.w a3, d2      *反対側の点の
203:      move.w a4, d3      * 座標を復帰
204:      rts
205: *
206: just:  move.w d2, d0      *反対側の点が
207:      move.w d3, d1      * ちょうど求める点
208:      rts
209:
210:      .end

```

多角形のクリッピングルーチン `gclippoly` には、引数として頂点の列データと、クリッピング後の頂点座標を格納するメモリ領域のアドレスを渡す。クリッピングによって角が削ぎ落とされると最大で頂点の数が2倍に増えることになるから、結果を受け取るメモリはそれを見越して大きめに確保しておかなければならない。

クリッピングはまずウィンドウの上端、つづいて下端の2回に分けて行っている。ここでは、処理中の頂点と直前に処理した頂点の関係による処理の振り分け方をよく見てほしい。ある点 P をウィンドウの一端でクリッピングする場合、直前に処理した点 (P_0 とおく) と P との関係には次の4通りが考えられる。

- 1) P も P_0 も不可視側にある
- 2) P は不可視側にあり、 P_0 は可視側にある
- 3) P は可視側にあり、 P_0 は不可視側にある
- 4) P も P_0 も可視側にある

明らかに、両方が不可視側にある1) のケースでは P は単に捨てられる。2) のケースでは P をウィンドウ端にクリップし、得られた点をバッファに登録する。3) のケースではまず P_0 をウィンドウ端にクリップして得られた点を登録してから P も登録する。両方とも可視側にある4) のケースでは P のみを登録する。 P_0 は直前のループで登録ずみのはずだからだ。

なお、リスト2では、直前の点が見えなかったかどうかを覚えておらずに、毎回、可視だったか不可視だったかを調べ直すという手抜きをしている。この部分については改善の余地があるだろう。手抜きといえ、例の“辺の終点を無視する”都合で、ウィンドウ下端でのクリッピングは1ピクセル甘くなっている。きっちりウィンドウの下端でクリッピングしてしまうと、ウィンドウの最下ラインにはまったく描画が行われなくなってしまうのだ。

また、処理単位を明確にするとはいっておきながら、リスト2ではどさくさに最初の頂点と最後の頂点を結ぶ処理を行っている (112~114行)。

辺リストの作成部 (リスト3)

リスト3
GENEDGELIST.S

```

1: *      辺のリストを作成する (gfillpolyの前処理)
2:
3:      .include      edge.h
4: *
5:      .xdef      genedgelist
6: *
7:      .text
8:      .even
9: *
10: genedgelist:
11: POINTS =      8
12: EDGLST =      12
13:      link      a6, #0
14:      movem.l   d0-d4/a1-a2, -(sp)
15:
16:      movem.l   POINTS(a6), a0-a1
17:      exg.l     a0, a1
18:

```

```

19:      move.w  (a1)+, d4      *d4 = 点の数
20:      subq.w  #2, d4        *
21:      bcs     gdone         *最低2点必要
22:
23: gloop: movem.w (a1), d0-d3  *2点の座標を取り出す
24:
25:      cmp.w   d1, d3         *
26:      beq     genext        *水平の辺は無視する
27:
28:      bgt     gnedg1        *y0 < y1を保証する
29:      exg.l   d0, d2        *
30:      exg.l   d1, d3        *
31: gnedg1: move.w d0, X0(a0)   *X0(a0) = x0
32:      move.w  d1, Y0(a0)   *Y0(a0) = y0
33:
34:      sub.w   d3, d1        *d1 = y0 - y1 = -dy
35:      move.w  d1, E0(a0)   *E0 = -dy
36:
37:      neg.w   d1            *
38:      move.w  d1, DY0(a0)  *DY0(a0) = y1 - y0 = dy
39:
40:      add.w   d1, d1        *
41:      move.w  d1, DEY(a0)  *DEY(a0) = dy * 2
42:
43:      moveq.l #0, d1        *sgn(x1-x0) と
44:      sub.w   d0, d2        *abs(x1-x0)を求める
45:      beq     gnedg3        *
46:      bpl     gnedg2        *
47:      moveq.l #-1, d1       *
48:      neg.w   d2            *
49:      bra     gnedg3        *
50: gnedg2: moveq.l #1, d1     *
51:
52: gnedg3: move.w d1, SX(a0)   *SX(a0) = sgn(x1-x0)
53:
54:      add.w   d2, d2        *
55:      move.w  d2, DEX(a0)  *DEX(a0) = dx * 2
56:
57:      lea.l   EDGBUFSIZ(a0), a0
58:
59: genext: addq.l #4, a1       *1点分進める
60:      dbra    d4, gloop
61:
62: gdone: movem.l (sp)+, d0-d4/a1-a2
63:      unlk    a6
64:      rts
65:
66:      .end

```

辺のリストを作成するgenedgelistにはクリッピング後の頂点列と、辺リスト格納用のメモリ領域アドレスを渡す。ここでは、隣りあった頂点を結ぶ辺をBresenhamのアルゴリズムで描くための誤差項そのほかのパラメータの初期値を計算し、EDGBUF構造体に格納する。このとき、水平な辺は見つけしだい削除している(25~26行)。辺リストの作成がすんだら、genedgelistはa0レジスタにその最終アドレスを入れて戻る。作成した辺リストのEDGBUF構造体の個数を返してもよかったのだが、穴のあいた図形を描く場合など、複数の輪郭を一連の辺リストにまとめたいたいというときには、最終アドレスがわかったほうが便利だと判断した結果だ。

描画部 (リスト4)

リスト4
GFillPoly.S

```

1: *      スキャンコンバージョンによる多角形の塗り潰し
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5:      .include      rect.h
6:      .include      edge.h
7: *
8:      .xdef      gfillpoly
9:      .xref      ghline
10:     .xref      gramadr
11:     .xref      cliprect
12: *
13:     .offset 0      *gfillpolyの引数構造
14: *
15: EDGES: .ds.l 1      *辺リスト先頭アドレス
16: EDGEED: .ds.l 1      *辺リスト最終アドレス
17: EDGARY: .ds.l 1      *EDGBUFのポインタ配列用ワーク
18: COL: .ds.l 1      *描画色
19: *
20:     .text
21:     .even
22: *
23: gfillpoly:
24: ARGPTR = 4
25:     movem.l d0-d7/a0-a6, -(sp)
26: SAVREGS = 8+7
27: ARGPTR = ARGPTR+SAVREGS*4
28:     movea.l ARGPTR(sp), a0      *a0 = 引数列
29:     bsr     init      *初期化する
30:     bra     lpent
31:
32: *      d0.l 描画色パレットコード (上位/下位ワードとも)
33: *      d1~d4 作業用
34: *      d5.l ghline
35: *      d6.w 残りEDGBUF数
36: *      d7.w 注目しているスキャンラインのy座標
37: *
38: *      a0~a2 作業用
39: *      a3 未処理EDGBUFのポインタ配列先頭
40: *      a4 未処理EDGBUFのポインタ配列末尾
41: *      (=処理中EDGBUFのポインタ配列先頭)
42: *      a5 処理中EDGBUFのポインタ配列末尾
43: *      a6 注目しているスキャンラインの左端アドレス
44:
45: mainloop:
46:     cmpa.l a4, a3      *未処理のEDGBUFが
47:     beq     nomore     * もうなければスキップ
48:     bsr     activate   *アクティブにすべきEDGBUFを探す
49:
50: nomore: cmpa.l a5, a4      *処理中のEDGBUFが
51:     beq     nextln     *   なければスキップ
52:
53:     bsr     sortlist    *交点のx座標でソートする
54:     bsr     drawline    *1ライン描画する
55:     bsr     calcnextx   *つぎの交点を求める
56:
57: nextln: lea.l GNBYTE(a6), a6      *a6 = つぎのラインの左端アドレス
58:     addq.w #1, d7      *d7 = つぎのラインのy座標
59:
60: lpent: tst.w d6      *EDGBUFが残っているあいだ
61:     bne     mainloop   *   繰り返す
62:
63:     movem.l (sp)+, d0-d7/a0-a6
64:     rts
65:
66: *
67: *      未処理EDGBUF群の中から

```



```

68: *      注目しているスキャンライン上に始点があるものを探し、
69: *      あれば、処理中EDGBUFのポインタ配列に移動する
70: *
71: activate:
72:     movea.l a3, a1          *a1 = 未処理EDGBUFのポインタ配列
73:     actvlp: movea.l (a1)+, a0    *a0 = EDGBUFへのポインタ
74:     cmp.w   Y0(a0), d7        *始点のy座標 = 現在のy座標?
75:     bne     actvnx           * でなければ、まだ
76:
77:     *EDGBUFをアクティブにする
78:     move.w   X0(a0), (a0)      *最初の交点のx座標 (= x0) を設定
79:     move.l   DY0(a0), DY(a0)  *ループカウンタと誤差項を初期化
80:
81:     move.l   -(a4), -(a1)      *未処理EDGBUFのポインタ配列から削除
82:     move.l   a0, (a4)         *処理中EDGBUFのポインタ配列に追加
83:
84:     actvnx: cmpa.l a4, a1      *すべての未処理EDGBUFについて
85:     bcs     actvlp           * 繰り返す
86:     rts
87:
88: *
89: *      交点のx座標の小さい順に
90: *      処理中のEDGBUFのポインタ配列をソートする
91: *
92: sortlist:
93:     move.l   a5, d2          *a5を待避
94:
95:     subq.l   #4, a5          *単純挿入法でソートする
96:     bra      sortnx          *
97:     srtlp2: move.l a1, -8(a2)  *
98:     srtlp1: movea.l (a2)+, a1  *
99:     cmp.w   (a1), d1         *
100:    bgt      srtlp2           *
101:    move.l   a0, -8(a2)       *
102:    sortnx: movea.l a5, a2     *
103:    movea.l   -(a5), a0        *
104:    move.w   (a0), d1         *
105:    cmpa.l   a4, a5           *
106:    bcc      srtlp1          *
107:
108:    movea.l   d2, a5          *a5を復帰
109:    rts
110:
111: *
112: *      1ライン分描画する
113: *
114: drawline:
115:     movea.l   a4, a1          *a1 = 処理中のEDGBUFのポインタ配列先頭
116:     movea.l   d5, a2          *
117:
118:     move.w   cliprect+MINX, d3    *d3 = MINX
119:     move.w   cliprect+MAXX, d4    *d4 = MAXX
120:
121:     drawlp: movea.l (a1)+, a0      *a0 = EDGBUFへのポインタ
122:     move.w   (a0), d1             *d1 = 描く水平線分の始点x座標
123:     cmp.w   d4, d1               *始点 > MAXXならば
124:     bgt     drawq                * 線分は画面の右外だから描画完了
125:
126:     movea.l   (a1)+, a0          *
127:     move.w   (a0), d2            *d2 = 描く水平線分の終点x座標
128:     cmp.w   d3, d2               *終点 < MINXならば
129:     blt     drawlp              * 線分は画面の左外
130:
131:     MAX      d3, d1              *始点を画面左端でクリップ
132:     MIN      d4, d2              *終点を画面右端でクリップ
133:     sub.w   d1, d2               *d2 = 描く水平線分の長さ-1
134:
135:     add.w   d1, d1               *
136:     lea.l   0(a0, d1.w), a0      *a0 = 描く線分の左端アドレス
137:     addq.w   #1, d2              *d2 = 描く線分の長さ
138:     bclr.l   #0, d2              *奇数?
139:     beq     notodd              *
140:     move.w   d0, (a0)+          * 奇数ピクセルの分

```

```

141: notodd: neg. w    d2          *
142:
143:         jsr      0(a2, d2. w)  *水平線分描画
144:
145:         cmpa. l   a5, a1
146:         bcs      drawlp
147:
148: drawq: rts
149:
150: *
151: *      つぎのスキャンラインおける
152: *      交点のx座標を求める
153: *
154: calcnextx:
155:         movea. l   a4, a1        *a1 = 処理中EDGBUFのポインタ配列先頭
156:         calclp: movea. l   (a1)+, a0    *a0 = EDGBUFへのポインタ
157:         subq. w    #1, 0Y(a0)        *カウンタを減らす
158:         beq      deactivate        * 0になったら配列から削除する
159:         *      (最後の1点は処理しない)
160:         move. w    E(a0), d1        *d1 = e
161:         add. w     DEX(a0), d1      *e += 2*dx
162:         bmi      calskp            *e < 0ならx座標は今回のまま
163:
164:         move. w    SX(a0), d4        *sgn(x1-x0) = 0なら
165:         beq      calcnx            * x座標は不変
166:         movem. w   (a0), d2-d3      *d2 = x, d3 = 2*dy
167:         inclp: add. w    d4, d2      *x += sgn(x1-x0)
168:         sub. w     d3, d1          *e -= 2*dy
169:         bpl      inclp            *e ≥ 0のあいだ繰り返し
170:
171:         move. w    d2, (a0)        *求めたx座標
172:         calskp: move. w   d1, E(a0)  *eを更新
173:
174:         calcnx: cmpa. l   a5, a1        *すべての処理中EDGBUFについて
175:         bcs      calclp            * 繰り返し
176:         rts
177: *
178: deactivate:
179:         move. l    -(a5), -(a1)      *用ずみのEDGBUFを削除する
180:         move. l    4(a5), (a5)      *その分番人を詰める
181:         subq. w    #1, d6           *残りEDGBUF数を減らす
182:         bra      calcnx
183:
184: *
185: *      初期化
186: *
187: *      ・すべてのEDGBUFを未処理EDGBUFのポインタ配列にまとめる
188: *      ・処理中EDGBUFのポインタ配列を空にする
189: *      ・最初に注目するスキャンラインのy座標と左端アドレスを求める
190: *      ・EDGBUFの数を数える
191: *      ・描画色をレジスタに設定する
192: *
193: init:
194:         movem. l   (a0)+, a1-a3      *a1 = EDGBUF配列の先頭
195:         *a2 = EDGBUF配列の末尾
196:         movea. l   a3, a4           *a3 = a4
197:         *      = 未処理EDGBUFのポインタ配列先頭
198:         moveq. l   #-1, d1          *d1 = 仮の最小y座標
199:         moveq. l   #0, d6           *d6 = EDGBUFのカウント
200:         move. w    (a0)+, d7        *d7 = 描画色
201:         bra      ilpent
202:
203:         initlp: move. l   a1, (a4)+  *順に未処理EDGBUFのポインタ配列に追加
204:         move. w    Y0(a1), d0      *始点のy座標が
205:         cmp. w     d1, d0          * 仮の最小y座標より小さければ
206:         bcc      notmin            *
207:         move. w    d0, d1          * 仮の最小y座標を更新する
208:
209:         notmin: addq. w   #1, d6      *EDGBUFの数を数え上げる
210:         lea. l     EDGBUFSIZ(a1), a1    *a1 = つぎのEDGBUF
211:         ilpent: cmpa. l   a2, a1        *すべてのEDGBUFについて
212:         bcs      initlp            * 繰り返し
213:
214:         *d1 = 最小のy座標

```

```

215:                                     *d6 = EDGBUFの個数
216:                                     *a4 = 未処理EDGBUFのポインタ配列末尾
217:                                     *   = 処理中EDGBUFのポインタ配列先頭
218:         movea.l a4, a5               *a5 = 処理中EDGBUFのポインタ配列末尾
219:                                     * (a4 = a5だから空)
220:
221:         move.l #dmydat, (a5)        *番人を置く
222:
223:         moveq.l #0, d0               *x = 0
224:         jsr gramadr                 *
225:         movea.l a0, a6               *a6 = 最初に処理する
226:                                     * スキャンラインの左端アドレス
227:         move.l #ghline, d5           *d5 = 水平線分描画ルーチン基準アドレス
228:
229:         move.w d7, d0                *d0 = 描画色
230:         swap.w d0                    *
231:         move.w d7, d0                *
232:
233:         move.w d1, d7                *d7 = 最初に処理する
234:                                     * スキャンラインのy座標
235:         rts
236: *
237: dmydat: .dc.w 32767                 *番人として使うダミーデータ
238:                                     * (XのフィールドしかないEDGBUF)
239:
240:         .end

```

実際に描画を行う gfillpoly には、引数として、

辺リストの先頭アドレス

辺リストの最終アドレス

作業用 (EDGBUFのポインタ配列格納用) のメモリアドレス

描画色のパレットコード

を並べたメモリ領域のアドレスを渡す。辺リストの最終アドレスには genedgelist からの戻り値をセットすることになる。

gfillpoly 自体、処理ごとにさらにいくつかのサブルーチンに分割してある。最初に一度だけ呼び出される初期化ルーチン init (193~235行) では、辺の数を数えながら、すべての辺を未アクティブな群にまとめ (203~212行)、アクティブな辺の群を空にし (218行)、そのほかレジスタの初期化を行う。203~212行のループではついでに y 座標の最小値を求めたりもしている。この y 座標で表されるスキャンラインから処理を開始することにより、どの辺とも交わらないスキャンラインの処理を省くことができる。

ここでは、辺の群 (EDGBUFのポインタの配列) の管理のしかたに注目してもらいたい。ポインタ配列の先頭アドレスは a3 レジスタで示される。a3 の指すアドレス以降に未アクティブな EDGBUF へのポインタが並び、末尾は a4 レジスタで示される。そして、a4 レジスタの指すアドレス以降につづけてアクティブな EDGBUF へのポインタを並べ、末尾を a5 で示す。ある辺がアクティブになったら、適当にポインタをつなぎ替え、境界である a4 をずらしていく。ポインタ配列1本分のスペースで2本分の働きを実現し、使用メモリを節約しているのだ。

初期化ルーチンのもう1つの見どころは、アクティブな EDGBUF のポインタ配列の末尾に立てている番人だ (221行)。番人とは、アルゴリズムを簡潔にするためのダミーデータで、リスト4では頭の x 座標を格納する項目しかない EDGBUF 構造体として、237行で用意している。

この番人は、まず、単純挿入法によるソート過程での挿入位置を探す処理を簡略化するのに一役買っている。通常、挿入位置の検索時には走査が配列の末尾を越えて行われなければならないチェックが必要になるが、末尾に最大値のデータを置いておけば、配列末尾を特別扱いする必要がなくなる。また、この番人は描画時にも、交点が奇数個しかなかった場合 (gfillpolyに故意に輪郭の閉じていない図形を渡した場合など) の予備データとして働く。つまり、余った最後の交点と対になって、水平線分をかたちづくるのだ。

初期化がすんだら45行からのメインループに制御が移る。ループ内では、まず未アクティブなEDGBUF群の中から始点が現在処理中のスキャンライン上にあるものを探して、あればアクティブにする (46~48行)。それから、アクティブなEDGBUFを交点のx座標でソートし、描画して、次のスキャンラインとの交点を求める (50~55行)。交点を求めるときには、同時に処理ずみのEDGBUFの削除も行っている。スキャンラインが画面下に達するまで、この処理を繰り返せば多角形が描かれる。もっとも、リスト4では、スキャンラインのy座標ではなく、EDGBUFの残り個数を終了条件に使っている。EDGBUFがなくなったら、もうそれ以上の処理は必要ないわけだ。

アクティブにすべきEDGBUFを探すサブルーチンactivate (71~86行) ではEDGBUFをアクティブ化するときのポインタのつなぎ替えと境界の移動 (81~82行) のしかたが小さなポイントだが、ほかにはとくに見るべき点はない。ソートするサブルーチンsortlist (92~109行) も、扱うデータが生データではなくポインタになっている点を除けば、単純挿入法のアルゴリズムどおりのつくりだ (ある程度最適化はしてあるが)。なお、sortlistは、番人があらかじめセットされていることを前提にしている。

1ライン描画するサブルーチンdrawline (114~148行) では、得られた交点のx座標を画面の左右端でクリッピングしながら水平線分の描画を行う。始点がウィンドウの右端より外にあったら以降の点も不可視だから処理を打ち切る (122~124行)。また、終点がウィンドウの左端より外にあったら始点も不可視だから、その線分は描かずスキップする (127~129行)。どちらでもなければ、線分は少なくとも部分的に可視だから、GMACRO.Hで定義したマクロMINとMAXを使って両端点がウィンドウ内に収まるのを保証してから実際に描画する。水平線分の描画部分にはループを展開した高速ルーチン (Chapter3) を利用している。

スキャンラインとの交点を求めるサブルーチンcalcnextx (154~182行) では、アクティブな各EDGBUFごとにBresenhamのアルゴリズムの1ステップを施し、x座標と誤差項eを更新している。ただし、傾きがなだらかな場合は、yが変化するまで (誤差項eが負になるまで)、167~169行でBresenhamのアルゴリズムを数ステップ繰り返す。本来、Bresenhamのアルゴリズムでは線分の傾きがなだらかな場合はxについてループを組まなければならないのを、強引にyでループしていることからくるつじつまあわせだ。この結果、図13のような辺の場合は交点として、図中黒で示したとびとびの点が選択される。得られる点は片側に偏ってしまうが、今回は妥協した。

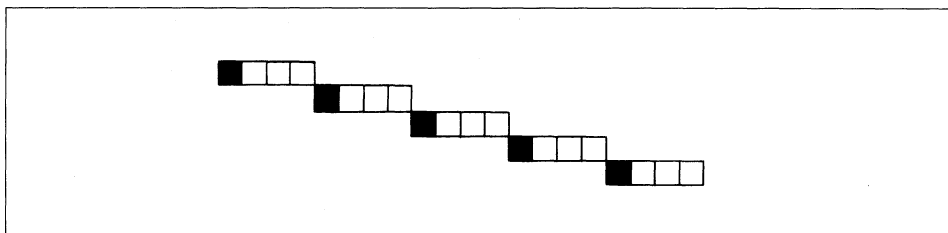


図13
なだらかな線分の扱い

動作試験

最後に動作試験用のプログラムをリスト5に示しておく。リスト5はキーが押されるまで、ランダムな7角形を描画し続ける。座標の範囲を128～383に制限して図形のサイズを限定した“はったりデモモード”にしてあるので、ペイントしか知らない人が見たら驚くぐらいの速度にはなっていると思う。頂点の数は6行のラベルNMAXPOINTで指定しているから適当に変更してみるといい。また、82～83行を殺してかわりに84行を復活すれば座標の範囲が0～511になる。39～41行を復活してクリッピングウィンドウを設定し、クリッピングがうまく働いているかどうか確認しておいてほしい。

リスト5
POLYTEST.S

```

1: *      gfillpolyの動作試験用プログラム
2:
3:      .include      doscall.mac
4:      .include      edge.h
5: *
6: NMAXPOINT      equ      7      #頂点の最大数 (2～32767)
7: NMAXEDGE      equ      NMAXPOINT*2+1  #辺の最大数
8: *
9:      .xref      gfillpoly
10:     .xref      gclippoly
11:     .xref      genedgelist
12:     .xref      setcliprect
13: *
14: FPACK      macro      callno
15:     .dc.w      callno
16:     .endm
17: *
18: _RAND      equ      $fe0e
19: *
20:     .offset 0
21: *
22: EDGES: .ds.l      1      #辺リスト先頭アドレス
23: EDGEED: .ds.l      1      #辺リスト最終アドレス
24: EDGARY: .ds.l      1      #EDGBUFのポインタ配列用ワーク
25: COL: .ds.l      1      #描画色
26: *
27:     .text
28:     .even
29: *
30: ent:
31:     lea.l      inisp, sp
32:
33:     move.l      #$0010_0005, -(sp)      #512x512, 65536

```

```

34:      DOS      _CONCTRL      *画面初期化
35:
36:      clr.l    -(sp)          *スーパーバイザモードへ
37:      DOS      _SUPER        *
38:
39: #      pea.l    window        *クリッピングウィンドウの設定
40: #      jsr     setcliprect    *
41: #      addq.l   #4, sp         *
42:
43:      lea.l     arg, a1        *a1 = 引数受け渡し領域
44: loop: bsr     setarg          *引数をセット
45:
46:      pea.l     edges
47:      pea.l     pnts2
48:      pea.l     pnts
49:      jsr     gclippoly        *クリッピングして
50:      addq.l    #4, sp
51:      jsr     genedgelist      *辺リストを作成し
52:      addq.l    #8, sp
53:      move.l    a0, EDGEED(a1) *辺リスト最終アドレス
54:
55:      pea.l     (a1)           *描画
56:      jsr     gfillpoly        *
57:      addq.l    #4, sp         *
58:
59:      DOS      _KEYSNS        *キーが押されるまで
60:      tst.l     d0             * 繰り返す
61:      beq      loop            *
62:
63:      DOS      _INKEY          *スペースが押されたら
64:      cmpi.b    #$20, d0       * 一時停止する
65:      bne      done            *
66: pause: DOS      _INKEY        *
67:      cmpi.b    #$20, d0       *
68:      beq      loop            *
69:
70: done:  move.w   #-1, -(sp)     *キーを読み捨てる
71:      DOS      _KFLUSH         *
72:
73:      move.l    #$0010_0000, -(sp) *width 96
74:      DOS      _CONCTRL        *画面再初期化
75:      DOS      _EXIT
76: #
77: setarg: lea.l    pnts, a0
78:      move.w    #NMAXPOINT, (a0) +
79:      move.w    #NMAXPOINT*2-1, d1
80:      arglp: FPACK      _RAND
81:      lsr.w     #7, d0          *0 ≤ d0 ≤ 255
82:      addi.w    #128, d0        *128 ≤ d0 ≤ 383
83: #      lsr.w     #6, d0          *0 ≤ d0 ≤ 511
84: #
85:      move.w    d0, (a0) +
86:      dbra     d1, arglp
87:
88:      FPACK      _RAND
89:      move.w    d0, COL(a1)
90:      rts
91: #
92:      .data
93:      .even
94: #
95: arg:   .dc.l    edges          *gfillpolyへの引数
96:      .dc.l    edges            *
97:      .dc.l    edgary          *
98:      .dc.w    0                *描画色
99: #
100: window: .dc.w    64, 64, 511-64, 511-64
101: #
102:      .bss
103:      .even
104: #
105: pnts:   .ds.w    1              *クリッピング前の頂点
106:      .ds.l    NMAXPOINT

```

```
107: pnts2: .ds.w 1          *クリッピング後の頂点
108:      .ds.l NMAXEDGE      *
109: edges: .ds.b NMAXEDGE*EDGBUFSIZ *EDGBUFの配列
110: edgary: .ds.l NMAXEDGE+1 *EDGBUFのポインタ配列
111:      * (番人の分を含む)
112: *
113:      .stack
114:      .even
115: *
116:      .ds.l 2048
117: inisp:
118:      .end ent
```


CHAPTER

5

ペイントルーチンの作成

ペイントルーチンの作成



この章のテーマはシードフィル (seed fill) による閉じた領域の塗り潰し、いわゆるペイントだ。IOCSのPAINT相当のペイントルーチンを作成してみる。

シードフィルアルゴリズム

シードフィルアルゴリズムの基本的な考え方は素朴なものだ。

- 1) 塗り潰しの開始点 (シードと呼ぶ) として指定された1ピクセルを塗る
- 2) いま塗ったシードピクセルの周囲から塗り潰すべきピクセルを探す
- 3) そのピクセルを新たなシードとして1), 2) と同様の処理を繰り返す
- 4) 新たなシードとなるピクセルがなくなった時点で塗り潰しは完了している

ここで、まっさらの画面を中央から塗り潰す場合を例にとるまでもなく、2) のステップで見つかるシードの候補は1点だけとはかぎらない。見つけたシード候補はあとでまな板に乗せるべく、すべてどこかに記憶しておく必要がある。そのあたりを考慮して、アルゴリズムをいくらか具体化すると次のようになる。

- 1') シードピクセルを塗る
- 2') いま塗ったシードピクセルの周囲から塗り潰されるべきピクセルを探し、シードの候補としてその座標をバッファに記憶する
- 3') バッファから1点取り出し、“そのピクセルがまだシードとしての資格を持っている”なら、新たなシードとして1), 2) 同様の処理を繰り返す
- 4') バッファが空になった時点で塗り潰しは完了している

新たなシードを決める際には、バッファ中のどのピクセルを選んでもよいが、新しい順か、古い順かのどちらかに統一するのが普通だろう。バッファをスタック構造にすれば自然に新しい順、キューにすれば古い順になる¹⁾。また、取り出し順序によらず、バッファから取り出したピクセルは無条件にシードとせず、いま一度シードとしての資格審査にかける必要がある。同じピクセルがダブってバッファに格納される場合があり、バッファに格納した段階では未処理のピクセルであったものが取り出したときにはすでに処理済みになっている (もう塗り潰さ

れている) 可能性があるからだ。

- 1) ちなみに、画面中央から塗り潰したときに、色が中央から広がっていきうならそのペイントルーチンはキュー方式、先にいずれかの画面の端に達してから残りに取り掛かるようならスタック方式と判断できる。

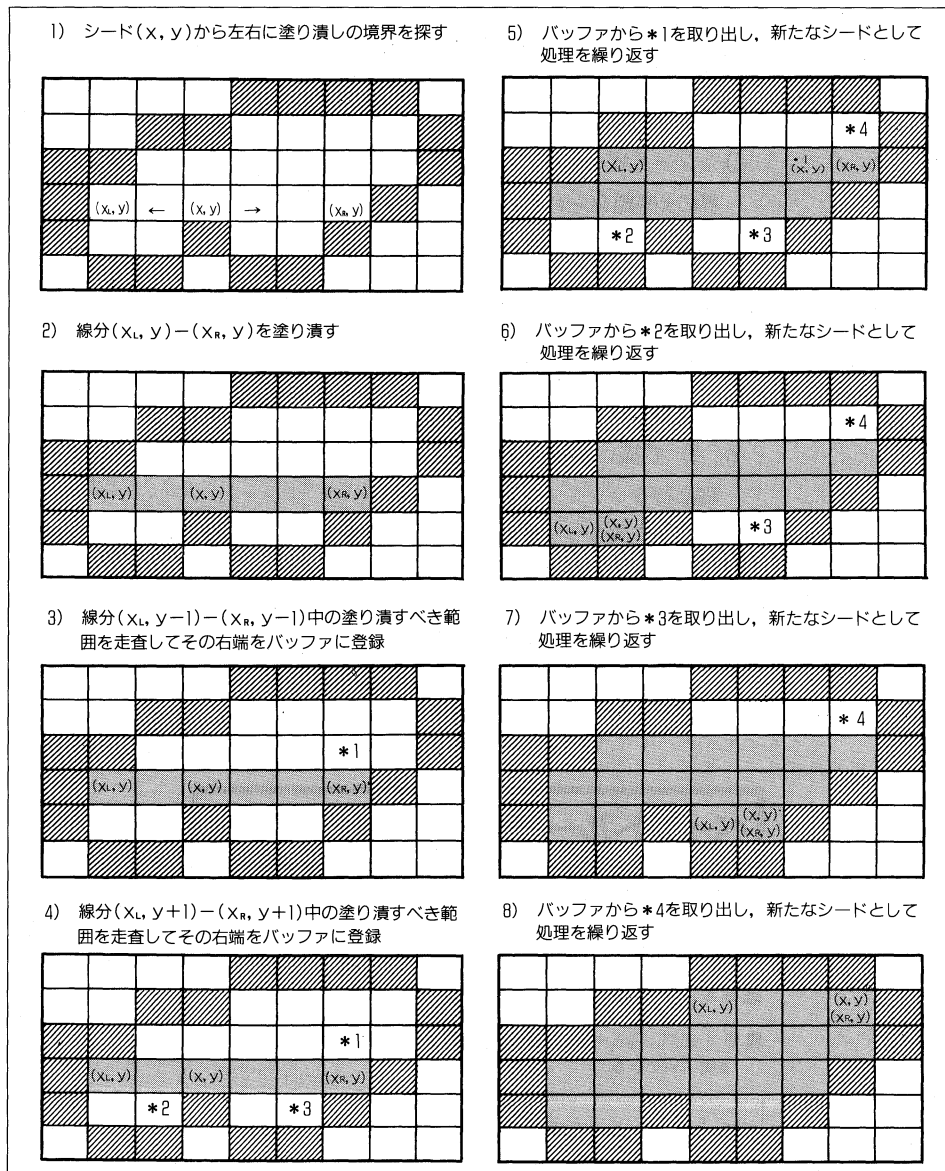


図1
スキャンラインシードフィルアルゴリズム

さて、いま示した単純なシードフィルのアルゴリズムではピクセル単位で処理しているがために、塗り潰す範囲が少し広くなるとバッファに大量のメモリを必要とするようになる。また、同じピクセルを重複してバッファに格納する率も高く、その重複分がさらにバッファを食い潰し、効率を低下させている。そこで、現実のペイントルーチンではもっとまとまった単位、水平線分単位で処理するのが普通だ。塗り潰すべきピクセルが横方向に並んでいたらそれらをひとまとめにして、バッファには代表のピクセル1つだけを格納する。この改良版アルゴリズムは、ときにスキャンラインシードフィルアルゴリズムと呼ばれる。その具体的な手順を次に示す。図1 (p.135) もあわせて見てほしい (図の例ではバッファにキューを採用している)。

- 1) シードピクセル (x, y) から左右方向に走査し、塗り潰しの境界を探す (得られた境界の x 座標をそれぞれ x_L, x_R とおく)
- 2) 線分 $(x_L, y) - (x_R, y)$ を描く
- 3) 2) で描いた線分のすぐ上、線分 $(x_L, y-1) - (x_R, y-1)$ を左から走査して、“塗り潰すべきピクセルが横に連続している部分” をすべて探し、それぞれのもっとも右側の座標をバッファに格納する (ただし、境界が見つからないうちにウィンドウ右端に達した場合は、その右端の座標をバッファに格納する)
- 4) 2) で描いた線分のすぐ下、線分 $(x_L, y+1) - (x_R, y+1)$ に対して3) 同様の処理を行う
- 5) 以下、バッファから有効なシードを取り出しては、1) ~ 4) を繰り返す
- 6) バッファが空になった時点で塗り潰しは完了している

アルゴリズムをこのように変更することにより、バッファの消費は大幅に抑えられる。また、余分なピクセルがバッファに格納されることもほとんどなくなり、とくに、領域に穴が空いていない場合には重複は一切生じない。

ペイントルーチンの実例

とりあえず、ペイントルーチンを作成してみよう。リスト1だ。効率にはこだわらず、素直にアルゴリズムどおりにつくってみた。シードの候補を溜め込むバッファにはキュー (実装上はリングバッファ) を採用している²⁾。

- 2) 一般に、キュー方式のほうがバッファの消費量が少ないことが知られている。

リスト1
GPAINT.S
(第1版)

```
1: #      シードフィルによる領域の塗り潰し (第1版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5:      .include      rect.h
6: #
7:      .xdef      gpaint
8:      .xref      gramadr
9:      .xref      cliprect
```

```

10: *
11:     .offset 0      *gpaintの引数構造
12: *
13: X0:     .ds.w 1     *初期シード座標
14: Y0:     .ds.w 1     *
15: COL:    .ds.w 1     *描画色
16: TEMP:   .ds.l 1     *作業用領域先頭
17: TEMPED: .ds.l 1     *作業用領域末尾+1
18: *
19:     .offset -8      *スタックフレーム
20: WORKSIZ:
21: BUFTOP: .ds.l 1
22: BUFEND: .ds.l 1
23: _a6:    .ds.l 1
24: _pc:    .ds.l 1
25: ARGPTR: .ds.l 1
26: *
27:     .offset 0      *シード候補
28: *
29: ADR:     .ds.l 1
30: X:       .ds.w 1
31: Y:       .ds.w 1
32: SEEDSIZ:
33: *
34:     .text
35:     .even
36: *
37: gpaint:
38:     link    a6, #WORKSIZ
39:     movem.l d0-d7/a0-a5, -(sp)
40:
41:     lea.l   cliprect, a5    *a5=クリッピング領域
42:     movea.l ARGPTR(a6), a1  *a1=引数列
43:
44:     movem.w X0(a1), d0-d1    * (d0, d1)=初期シード
45:     cmp.w   MINX(a5), d0     *初期シードが
46:     blt     done             * ウィンドウ外なら
47:     cmp.w   MINY(a5), d1     * 何もせずに戻る
48:     blt     done             *
49:     cmp.w   MAXX(a5), d0     *
50:     bgt     done             *
51:     cmp.w   MAXY(a5), d1     *
52:     bgt     done             *
53:
54:     jsr     gramadr          *a0=初期シードG-RAMアドレス
55:
56:     move.w  (a0), d6         *d6=領域色
57:
58:     move.w  COL(a1), d7      *d7=描画色
59:     cmp.w   d6, d7           *描画色と領域色が等しいなら
60:     beq     done             * 何もせずに戻る
61:
62:     movea.l TEMP(a1), a3     *a3=キュー内の読み出し位置
63:     movea.l a3, a4           *a4=キュー内の書き込み位置
64:     * (ともにバッファ先頭に初期化)
65:
66:     move.l  TEMPED(a1), d3   *d3=バッファ末尾+1
67:     sub.l   a3, d3           *バッファ先頭<末尾ならば
68:     bcs     done             * 何もせずに戻る
69:     andi.l  #$ffff_ffff, d3 *d3=バッファサイズ(16の倍数)
70:     beq     done             *バッファサイズが0ならば
71:     * 何もせずに戻る
72:     add.l   a3, d3           *d3=バッファ末尾+1
73:
74:     move.l  a3, BUFTOP(a6)   *バッファ先頭と末尾を
75:     move.l  d3, BUFEND(a6)  * ワークに格納しておく
76:
77:     bra     do               *メイン処理開始
78:
79: loop: movea.l (a3)+, a0       *キューからシードを
80:     move.w  (a3)+, d0        * 取り出す
81:     move.w  (a3)+, d1        *
82:
83: do:

```

```

84: *      d0      シードx座標
85: *      d1      シードy座標
86: *      d6      領域色
87: *      d7      描画色
88: *      a0      シードG-RAMアドレス
89: *      a3      キュー内のつぎの読み出し位置
90: *      a4      キュー内のつぎの書き込み位置
91: *      a5      クリッピング領域
92:
93:      movea.l  a0, a1      *a0=a1=シードのG-RAMアドレス
94:      move.w   d0, d2      *d0=d2=シードのx座標
95:
96:      cmp.w    (a1)+, d6    *すでに処理済みのシードならば
97:      bne      next        * 捨てる
98:
99:      *右方向の境界を探す
100: rchk:  move.w  MAXX(a5), d3  *d3=ウィンドウ右端x座標
101:      *d2=x1
102: rloop:  cmp.w   d3, d2      *ウィンドウ右端に達したら
103:      bge      lchk        * その直前の点が右の境界
104:      addq.w   #1, d2      *x1++
105:      cmp.w    (a1)+, d6    * (x1, y) が領域色のあいだ
106:      beq      rloop      * 繰り返す
107:
108:      subq.w   #1, d2      *d2=x1
109:
110:      *左方向の境界を探す
111: lchk:  move.w  MINX(a5), d3  *d3=ウィンドウ左端x座標
112:      *d0=x0
113: lloop:  cmp.w   d3, d0      *ウィンドウ右端に達したら
114:      ble      filspn      * その直前の点が左の境界
115:      subq.w   #1, d0      *x0--
116:      cmp.w    -(a0), d6    * (x0, y) が領域色のあいだ
117:      beq      lloop      * 繰り返す
118:
119:      addq.w   #1, d0      *d0=x0
120:      addq.l   #2, a0      *a0=(x0, y) のG-RAMアドレス
121:
122: filspn: sub.w   d0, d2      *d2=線分のピクセル数-1
123:      movea.l  a0, a2      *a0=a2=線分左端G-RAMアドレス
124:
125:      move.w   d2, d3      *線分 (x0, y) - (x1, y) を描く
126: floop:  move.w  d7, (a0)+   *
127:      dbra     d3, floop    *
128:
129:      move.w   d0, d4      *d0=d4=x0 (線分左端x座標)
130:      subq.w   #1, d4      *d4=x0-1
131:
132:      *真上のスキャンラインを走査する
133: uchk:  subq.w  #1, d1      *y--
134:      cmp.w    MINY(a5), d1  *ウィンドウの上端を越えていたら
135:      blt      dchk        * 走査の必要はない
136:      lea.l    -GNBYTE(a2), a0 *a0=走査するライン左端
137:      bsr      seapix      *走査する
138:
139:      *真下のスキャンラインを走査する
140: dchk:  addq.w  #1+1, d1     *y++
141:      cmp.w    MAXY(a5), d1  *ウィンドウの下端を越えていたら
142:      bgt      next        * 走査の必要はない
143:      lea.l    GNBYTE(a2), a0 *a0=走査するライン左端
144:      bsr      seapix      *走査する
145:
146: next:  cmpa.l  BUFEND(a6), a3 *キューの読み出し位置が
147:      bcs      next0        * バッファ末尾に達していたら
148:      movea.l  BUFTOP(a6), a3 * 先頭を指すよう修正する
149:
150: next0:  cmpa.l  a3, a4      *キューが空になるまで
151:      bne      loop        * 繰り返す
152:
153: done:  movem.l  (sp)+, d0-d7/a0-a5
154:      unlk     a6
155:      rts
156:
157: *

```

```

158: #      上下のラインからシードの候補を探しキューに追加する
159: #
160: seapix:
161:     move.w    d4, d0      #d0=走査開始x座標-1
162:     move.w    d2, d3      #d3=線分のピクセル数-1
163:     sloop1: addq.w    #1, d0      #非領域色部分を飛ばす
164:     cmp.w     (a0)+, d6    #
165:     dbeq      d3, sloop1   #
166:     beq       spix0        #領域色が見つかった
167:     rts          #領域色は見つからなかった
168:
169:     sloop2: addq.w    #1, d0      #領域色部分を飛ばす
170:     cmp.w     (a0)+, d6    #
171:     dbne      d3, sloop2   #
172:     beq       spix1        #
173:     subq.l    #2, a0      #ポインタとx座標は
174:     subq.w    #1, d0      # 進み過ぎている
175:
176:     spix1:  subq.l    #2, a0      #a0=領域色部分の右端
177:     move.l    a0, (a4)+    #見つけたシード候補を
178:     move.w    d0, (a4)+    # バッファに登録する
179:     move.w    d1, (a4)+    #
180:     addq.l    #2, a0      #つぎの走査に備える
181:
182:     cmpa.l    a3, a4      #キューが一杯になったら
183:     beq       drop        # いま登録したシードを捨てる
184:
185:     cmpa.l    BUFEND(a6), a4 #キューの書き込み位置が
186:     bcs       snext        # バッファ末尾に達したら
187:     movea.l    BUFTOP(a6), a4 # 先頭を指すよう修正する
188:
189:     snext:   tst.w    d3      #走査範囲の右端に達するまで
190:     bpl       sloop1       # 繰り返す
191:
192:     sdone:   rts
193:
194:     drop:   subq.l    #SEEDSIZ, a4    #いま登録したシードを捨てる
195:     rts
196:
197:     .end
    
```

リスト1のサブルーチンgpaintは、IOCS コールのPAINTと同じ構造で引数列を用意し、その先頭アドレスをスタックに積んで呼び出す。引数列の具体的な構造はリスト中の11～17行に示してある。シードの候補を溜め込むバッファは、塗り潰す領域のかたちの複雑さにおいて呼び出し側で必要な大きさだけ用意し、gpaintには先頭アドレスと最終アドレス+1のかたちで渡す。リスト2のテストプログラムがバッファサイズの目安となるだろう。リスト2は図2のようなパターンで覆われた画面を中央から塗り潰すだけのプログラムだが、このパター

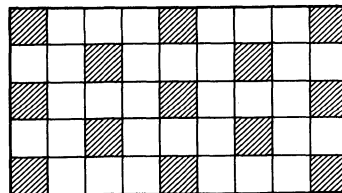


図2
 テスト用パターン

ンはペイントルーチンに負担をかける（バッファを大量に必要とする）ようにつくられている。
リスト1のgpaintの場合、バッファに8Kバイトもあればこの穴だらけの図形を確実に塗り潰
すことができる。

リスト2
GPAINTTEST.S

```

1: #      gpaintのテスト用プログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: #
6:      .xref      gpaint
7: #      .xref      setcliprect
8: #
9:      .offset 0
10: #
11:      .text
12:      .even
13: #
14: ent:
15:      lea, l      inisp, sp
16:
17:      clr, l      -(sp)          #スーパーバイザモードへ
18:      DOS      _SUPER          #
19:
20:
21:      move, l      #$0010_0005, -(sp)    #512x512, 65536
22:      DOS      _CONCTRL          #画面初期化
23:
24: #      pea      window          #クリッピングウィンドウの設定
25: #      jsr      setcliprect      #
26: #      addq, l      #4, sp        #
27:
28:      lea, l      linarg1, a1          #画面をパターンで埋める
29:      lea, l      linarg2, a2          #
30:
31:      moveq, l      #0, d6            #
32:      move, w      #256-1, d7          #
33: loop: move, w      d6, 2(a1)          #
34:      move, w      d6, 6(a1)          #
35:      IOCS      _LINE                #
36:      exg, l      a1, a2              #
37:      addq, w      #1, d6             #
38:
39:      move, w      d6, 2(a1)          #
40:      move, w      d6, 6(a1)          #
41:      IOCS      _LINE                #
42:      exg, l      a1, a2              #
43:      addq, w      #1, d6             #
44:      dbra      d7, loop              #
45:
46:      pea, l      paintarg          #塗り潰す
47:      jsr      gpaint                #
48:      addq, l      #4, sp            #
49:
50: #      lea, l      paintarg, a1
51: #      IOCS      _PAINT
52:
53:      DOS      _EXIT
54: #
55:      .data
56:      .even
57: #
58: linarg1: .dc, w      0, 0, 511, 0, $c000, $8888    #IOCS LINEの引数
59:
60: linarg2: .dc, w      0, 1, 511, 1, $c000, $2222
61:
62: #
63: paintarg: .dc, w      255, 255          #gpaintの引数
64:

```



```

65:      .dc.w  $003f
66:      .dc.l  paintwork
67:      .dc.l  paintworkend
68: *
69: window: .dc.w  64, 64, 511-64, 511-64
70: *
71:      .bss
72:      .even
73: *
74: paintwork:
75:      .ds.b  $4000
76: paintworkend:
77: *
78:      .stack
79:      .even
80: *
81:      .ds.l  4096
82: inisp:
83:      .end    ent

```

gpaintが塗り潰す領域は、上下左右に隣接する（4連結）同じ色のピクセル群として定義されている（内部定義）。ペイントルーチンには、適当な境界色で囲まれた閉じた領域を塗り潰す（境界定義）流儀や、上下左右だけではなく斜め方向に隣接するピクセルも塗り潰す（8連結）流儀もあるが、ここではIOCSの仕様に準じた。

では、プログラム（リスト1）を順に見ていこう。真っ先に45～52行で初期シードがクリッピングウィンドウ内にあるかどうかを調べている。ウィンドウ内であれば、そのG-RAMアドレスをa0に求め（54行）、そのピクセルの色をd6に拾っておく（56行）。この色が塗り潰すべき領域の色となる。続いて、描画色をd7に取り出し、領域色と比較する（59～60行）。両者が一致するようなら、初期シードピクセルがすでに描画色になっているわけだから、何もせずに抜ける。

62～75行はキュー周りの初期化だ。最初の2行で読み出し位置を表すポインタa3と書き込み位置を表すポインタa4が、ともにバッファ先頭を指すようにし、キューを空にしている。話が前後するが、メインの処理をかんたんにするために、このキューにはシード候補の座標と同時にG-RAMアドレスも格納するようにしてある。x, y座標に各2バイト、G-RAMアドレスに4バイトだから、シードひとつあたりのバッファの消費量は8バイトとなる。

ここで、リングバッファに対してデータを読み書きするときには、読み書きごとにポインタがバッファ末尾に達したかどうかを調べ、末尾に達していたら先頭を指すよう修正する必要があることを思い出そう。この都合上、gpaintに渡されるバッファの大きさはきっかり8の倍数であることが望ましい。66～72行では念のため、バッファの大きさが変な値になっていないかどうか調べ、バッファサイズが8の倍数になるようバッファ最終アドレスを調整している。

79行からがメインループだ。最初の1回だけはすでにレジスタにシードの座標とアドレスが格納されているので、ループの途中、83行に飛び込む。2度目以降はその直前に79～81行でキューからシードを取り出す処理が入る。わけがあって、キューからの読み込みポインタがバッファ末尾に達したかどうかのチェックはループの末尾近く（146～148行）で行っている。

100～108行でシードピクセルから右方向に走査して領域色から非領域色への変わり目、あ

るいは、クリッピングウィンドウの右端を探す。その位置が、いま塗り潰す水平線分の右端 (x_R) となる。領域色かどうかの判定に先立って x 座標を増加させている関係で、102～106 行のループを抜けたときの $d2$ は、求める x 座標より 1 大きい。108 行はこの分の補正だ。111～120 行では同様にシードピクセルから左に走査して、水平線分の左端 x 座標 (x_L) を $d0$ に求めている。このとき、同時に対応する水平線分左端の G-RAM アドレスが $a0$ に求まる。

ここまでで得た結果を使い、122～127 行で水平線分を描く。それから、133～137 行で真上、140～144 行で真下の 1 ラインから次のシードの候補となるピクセルを探し、キューに追加する。シードの候補を探す処理は 160 行以下のサブルーチン `seapix` にまとめてある。`seapix` では、ラインの左端から非領域色ピクセルを飛ばして領域色ピクセル列の左端を探し (163～166 行)、そこから非領域色ピクセルにぶつかるまでポインタと x 座標を進める (169～174 行) という手順で、領域色ピクセル列の右端座標と G-RAM アドレスを求めている。得られた座標とアドレスは 176～180 行でキューに追加する。バッファが足りなくなった場合にはいきなりエラー終了せずに、バッファ中の最新の (たったいま求めたばかりの) データを捨てて処理を継続するようにしてある (182～183 行、194 行)。シードの候補を捨ててしまえば塗り残しが生じるから、処理を継続してもしかたないように見えるかもしれないが、バッファが足りなくなるほどの複雑な (穴だらけの) 図形であれば、反対側からの回り込みである程度カバーできるものだ。穴の空いた図形では同一のピクセルが 2 度重複してバッファに登録される場合があることを思い出してほしい。運がよければ、塗り残しなしに処理を完了する可能性だってある。リスト 2 のテストプログラムでバッファサイズをいろいろ変えてみるとおもしろいだろう。

シード候補を探す 163～165 行や 169～171 行のループでは、走査があらかじめ決められた範囲 (x 座標が $x_L \sim x_R$) をこえて行われないようにするために、`dbeq`、`dbne` が微妙な使われ方をしている。`dbeq`、`dbne` のループをさらに 163～190 行のループで括ってある関係で、少し気を抜くと、走査の進みぐあいとループカウンタとのつじつまがあわなくなってしまうのだ。166 行、172 行で、ループを抜けた直後にしている条件判断や、外側のループ最後の 189 行でループカウンタ $d3$ が -1 になっていないかどうか調べ直している意味をよく考えてみてほしい。

上下のラインの走査がすんだ時点で 1 つのシードピクセル (で代表される水平線分) の処理が完了する。以下、同様の処理をキューが空になるまで繰り返す。キューが空かどうかは読み出し位置と書き込み位置が一致しているかどうかで判断している (150～151 行)。

アルゴリズムを見直す

リスト 1 は真正直につくってあるだけあって、実行速度はかなり遅い。平均的な性能は IOCS コールの PAINT よりも数段落ちる。IOCS.X 版はおろか ROM 版にもおよばない。というわけで、ぼちぼちと高速化に取りかかろう。無駄な処理を省くという方針でいく。図 3 のような領域を点 P から塗り潰す場合を例に、アルゴリズムのおさらいついでに、処理の流れを追い、

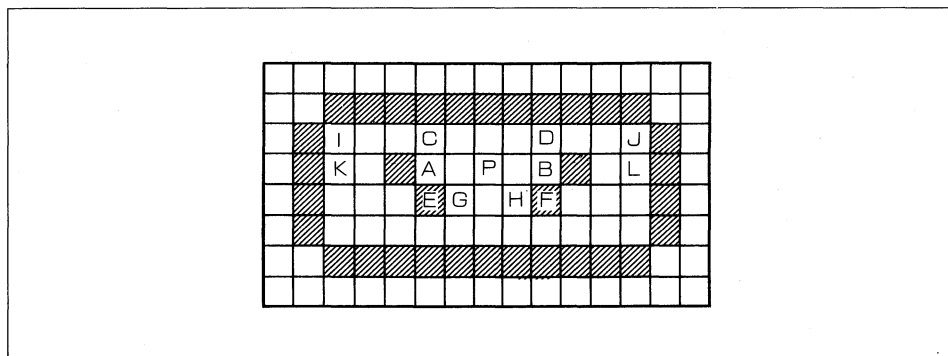


図3
アルゴリズムの再検討

どこで道草を食っているのか検討してみよう。

まず、シードPから左右方向に境界を探すとA、Bが得られる。AB間を塗り潰したら、ABの上下のラインであるCD、EF中から領域色の部分を探す。CDはまるまると領域色、EF中ではGH間が領域色だから、それぞれの代表として右端のD、Hをキューに追加する。これで1ライン分の処理がすんだので、キューから新たなシードを取り出す。リスト1の処理順序にしたがえば、点Dが取り出される。

Dからの左右方向の境界検索時、左方向へはDからIまで、右方向へはDからJまでの範囲を走査することになる。ここで1つ無駄な処理が見つかった。CD間はシードがPのときに一度走査済みであり、すべて領域色だということがわかっている。この情報を利用すればCD間の再走査は不要となり、左方向の走査はCの左のピクセルから始めるよう手抜きができる。もちろん、そのためにはバッファに記憶しておく情報量を増やしておかなければならない。つまり、上下のラインを走査したときに領域色が連続した部分が見つかったら、これまでのように右端だけをバッファに記録するのではなく、左端もセットにして覚えておくのだ。シードを点から線分に拡張するといってもよい。

IJを得、塗り潰した後の上下のラインの走査にも同様の無駄がある。初期シードを除けば、上か下かに“そのシードをバッファに登録した親のライン”があるはずだ。親のラインはすでに一度走査されているから、親ラインとの重複部分は再走査の必要がない。いまの例でいうと、線分IJの下側走査範囲KLのうちCDの真下にあたる部分ABの走査が、ばつさり省略できる。この場合、走査範囲KLは2分されることに注意しよう。また、実際のプログラムでは親が上にあるか下にあるか判断できなければならないから、シードをバッファに登録する際に、親ラインのy座標という情報をつけ加えておくが必要になる。

以上の無駄を省くことにより、G-RAMを読み込む回数はだいたい1/3になる。線分描画時の書き込みとあわせて考えても、G-RAMアクセス回数は半分ですむ計算だ。ある程度以上の面積を塗り潰す場合であれば、これはそのまま実行時間の半減に繋がるだろう。

リスト3が改良版アルゴリズムを使ったペイントルーチンだ。ついでだから、水平線分の描

画にはChapter3のループ展開版を使うようにし、細部も適当に最適化してある。たとえば、左右方向の境界検索を行う126～127行や137～138行のループはリスト1の102～106行、113～117行よりもずっとシンプルになっている。第1にループの中で座標をカウントアップしていた部分がなくなった。座標の変化は、ループに入る前と抜けた後のG-RAMを指すポインタの差から逆算するようにしたのだ。また、ループの中でウィンドウ端に達したかチェックしていた部分もなくなった。ループに入る前にウィンドウ端までのピクセル数を計算してdbneを使うようにしたからだ。

リスト3
GPAINT.S
(高速化版)

```

1: #      シードフィルによる領域の塗り潰し (高速化版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5:      .include      rect.h
6: #
7:      .xdef      gpaint
8:      .xref      ghline
9:      .xref      gramadr
10:     .xref      cliprect
11: #
12:     .offset 0      #gpaintの引数構造
13: #
14:     X0: .ds.w 1      #初期シード座標
15:     Y0: .ds.w 1      #
16:     COL: .ds.w 1      #描画色
17:     TEMP: .ds.l 1      #作業用領域先頭
18:     TEMPED: .ds.l 1      #作業用領域末尾+1
19: #
20:     .offset -14      #スタックフレーム
21:     WORKSIZ:
22:     LXSAV: .ds.w 1      #処理中シードのLX
23:     RXSAV: .ds.w 1      #          RX
24:     MY: .ds.w 1      #          PREVY
25:     BUFTOP: .ds.l 1      #リングバッファ先頭
26:     BUFEND: .ds.l 1      #リングバッファ末尾+1
27:     _a6: .ds.l 1
28:     _pc: .ds.l 1
29:     ARGPTR: .ds.l 1
30: #
31:     .offset 0      #シード候補
32: #
33:     LADR: .ds.l 1      #左端G-RAMアドレス
34:     LX: .ds.w 1      #左端x座標
35:     Y: .ds.w 1      #y座標
36:     RADR: .ds.l 1      #右端G-RAMアドレス
37:     RX: .ds.w 1      #右端x座標
38:     PREVY: .ds.w 1      #親のy座標
39:     SEEDSIZ:
40: #
41:     .text
42:     .even
43: #
44:     gpaint:
45:     link      a6, #WORKSIZ
46:     movem.l   d0-d7/a0-a5, -(sp)
47:
48:     movea.l   ARGPTR(a6), a1      #a1 = 引数列
49:     movem.w   X0(a1), d0-d1      #(d0, d1) = 初期シード
50:
51:     lea.l     cliprect, a5      #a5 = クリッピング領域
52:     cmp.w     (a5)+, d0      #初期シードが
53:     blt       done            # ウィンドウ外なら
54:     cmp.w     (a5)+, d1      # 何もせずに戻る
55:     blt       done            #
56:     cmp.w     (a5)+, d0      #
57:     bgt       done            #

```

```

58:      cmp.w  (a5)+, d1      *
59:      bgt   done           *
60:      subq.l #RECT, a5      *a5 = クリッピング領域
61:
62:      jsr    gramadr        *a0 = 初期シードG-RAMアドレス
63:
64:      move.w (a0), d6       *d6 = 領域色
65:
66:      move.w COL(a1), d7    *d7 = 描画色
67:      cmp.w  d6, d7         *描画色と領域色が等しいなら
68:      beq    done           * 何もせずに戻る
69:      swap.w d7             *
70:      move.w COL(a1), d7    *
71:
72:      movea.l TEMP(a1), a3   *a3 = キュー内の読み出し位置
73:      movea.l a3, a4        *a4 = キュー内の書き込み位置
74:      * (ともにバッファ先頭に初期化)
75:
76:      move.l TEMPED(a1), d3  *d3 = バッファ末尾+1
77:      sub.l   a3, d3         *バッファ先頭<末尾ならば
78:      bcs    done           * 何もせずに戻る
79:      andi.l  #$ffff_fff0, d3 *d3 = バッファサイズ (16の倍数)
80:      beq    done           *バッファサイズが0ならば
81:      * 何もせずに戻る
82:      add.l   a3, d3         *d3 = バッファ末尾+1
83:
84:      move.l  a3, BUFTOP(a6) *バッファ先頭と末尾を
85:      move.l  d3, BUFEND(a6) * ワークに格納しておく
86:
87:      movea.l a0, a1         *初期シードでは
88:      move.w  d0, d2         * LXとRXは一致している
89:
90:      move.w  d1, MY(a6)     *初期シードには親はいない
91:
92:      bra     do
93:
94: loop:  movea.l (a3)+, a0     *a0 = LADR
95:        move.w (a3)+, d0     *d0 = LX
96:        move.w (a3)+, d1     *d1 = Y
97:        movea.l (a3)+, a1    *a1 = RADR
98:        move.w (a3)+, d2     *d2 = RX
99:        move.w (a3)+, MY(a6) *PREVYをワークにセーブ
100:
101: do:
102: *      d0      左側シードx座標
103: *      d1      左側シードy座標
104: *      d2      右側シードx座標
105: *      d6      領域色 (この色の点を塗り潰す)
106: *      d7      描画色
107: *      a0      左側シードG-RAMアドレス
108: *      a1      右側シードG-RAMアドレス
109: *      a3      キュー内のつぎの読み出し位置
110: *      a4      キュー内のつぎの書き込み位置
111: *      a5      クリッピング領域
112:
113:      cmp.w  (a1)+, d6       *すでに処理済みのシードならば
114:      bne    next           * 捨てる
115:
116:      move.w d0, LXSAV(a6)   *LX, RXをワークにセーブ
117:      move.w d2, RXSAV(a6)   *
118:
119:      movea.l a0, a2         *a0 = a2 = シード右端のG-RAMアドレス
120:
121:      *右方向の境界を探す
122: rchk:  move.w MAXX(a5), d3   *
123:        sub.w  d2, d3        *d3 = 走査する最大ピクセル数
124:        subq.w #1, d3        *d3 = 0ならば
125:        bmi    lchk         * 走査は不要
126: rloop: cmp.w  (a1)+, d6     *
127:        dbne   d3, rloop     *
128:        beq    lchk         *
129:        subq.l #2, a1        *
130:
131:      *左方向の境界を探す
    
```

```

132: lchk:  move.w  MINX(a5), d3
133:         sub.w  d0, d3      #d3 = -走査する最大ピクセル数
134:         neg.w  d3          #d3 = 走査する最大ピクセル数
135:         subq.w #1, d3      #d3 = 0ならば
136:         bmi    filspn     # 走査は不要
137: lloop:  cmp.w  -(a0), d6    #
138:         dbne   d3, lloop   #
139:         beq    filspn     #
140:         addq.l #2, a0      #
141:
142: filspn: move.l  a2, d3      #d3 = 走査開始前の位置
143:         sub.l  a0, d3      #d3 = 左へ走査したバイト数
144:         lsr.w  #1, d3      #d3 = 左へ走査したピクセル数
145:         sub.w  d3, d0      #d0 = 線分左端x座標
146:
147:         move.l  a1, d2      #d2 = 走査終了後の右端位置
148:         sub.l  a0, d2      #d2 = 左端から右端のバイト数
149:         lsr.w  #1, d2      #d2 = 線分のピクセル数
150:         subq.w #1, d2      #d2 = 線分のピクセル数-1
151:
152:         movea.l a0, a2      #a0 = a2 = 線分左端G-RAMアドレス
153:
154:         move.w  d2, d3      #d3 = 描く線分の長さ-1
155:         addq.w  #1, d3      #d3 = 描く線分の長さ
156:         bclr.l  #0, d3      #奇数?
157:         beq     notodd      #
158:         move.w  d7, (a0) +  # 奇数ピクセルの分
159:         notodd: neg.w  d3    #
160:
161:         lea.l   ghline, a1  #水平線分描画
162:         exg.l   d0, d7      #
163:         jsr     0(a1, d3.w)  #
164:         exg.l   d0, d7      #
165:
166:                                     #真上のスキャンラインを走査する
167: uchk:   add.w   d0, d2      #d2 = 線分右端x座標
168:         move.w  d1, d4      #d4 = y
169:
170:         subq.w  #1, d1      #y--
171:         cmp.w   MINY(a5), d1 #ウィンドウの上端を越えていたら
172:         blt     dchk        # 走査の必要はない
173:         lea.l   -GNBYTE(a2), a0 #a0 = 走査するライン左端
174:         bsr     seapix      #走査する
175:
176:                                     #真下のスキャンラインを走査する
177: dchk:   addq.w  #1+1, d1     #y++
178:         cmp.w   MAXY(a5), d1 #ウィンドウの下端を越えていたら
179:         bgt     next        # 走査の必要はない
180:         lea.l   GNBYTE(a2), a0 #a0 = 走査するライン左端
181:         bsr     seapix      #走査する
182:
183: next:   cmpa.l  BUFEND(a6), a3 #キューの読み出し位置が
184:         bcs     next0       # バッファ末尾に達していたら
185:         movea.l BUFTOP(a6), a3 # 先頭を指すよう修正する
186:
187: next0:  cmpa.l  a3, a4       #キューが空になるまで
188:         bne     loop        # 繰り返す
189:
190: done:   movem.l (sp)+, d0-d7/a0-a5
191:         unlk   a6
192:         rts
193:
194: #
195: #       上下のラインからシードの候補を探しキューに追加する
196: #
197: seapix:
198:         movea.l a0, a1      #a0 = a1 = 走査開始位置
199:         suba.w  d0, a1      #
200:         suba.w  d0, a1      #a1 = 走査開始ラインの
201:                                     # 物理的な左端 (x=0) のアドレス
202:
203:         cmp.w   MY(a6), d1   #走査するラインは親ライン?
204:         bne     spixl        # 違うのならばふつうに走査する

```

```

205:
206:     move.w  LXSAV(a6), d3      #d3 = 走査範囲のうち
207:     sub.w   d0, d3             # 親ライン左端からのみ出し分
208:     ble     spix0              #はみ出しはなかった
209:     subq.w  #1, d3             #d3 = ループカウンタ
210:
211:     move.l  a0, -(sp)          #走査範囲のうち
212:     bsr     spix2              # 親ライン左端より
213:     movea.l (sp)+, a0          # 左の部分を走査
214:
215: spix0: move.w  d2, d3          #d3 = 走査範囲のうち
216:     move.w  RXSAV(a6), d5      # 親ライン右端からの
217:     sub.w   d5, d3             # はみ出し分
218:     ble     sdone              #はみ出しはなかった
219:     subq.w  #1, d3             #d3 = ループカウンタ
220:
221:     add.w   d5, d5             #a0 = 走査開始位置
222:     lea.l   2(a1, d5), a0      #
223:
224:     bra     spix2              #走査範囲のうち
225:                                     # 親ライン右端より
226:                                     # 右の部分を走査
227:
228: spix1: move.w  d2, d3          #
229:     sub.w   d0, d3             #d3 = 走査範囲のピクセル数-1
230: spix2:
231: sloop1: cmp.w  (a0)+, d6        #非領域色部分を飛ばす
232:     dbeq    d3, sloop1         #
233:     bne     sdone              #
234:
235:     move.l  a0, d5             #d5 = 領域色部分左端アドレス
236:     subq.l  #2, d5
237:
238:     move.l  d5, (a4)+          #LADR
239:     sub.l   a1, d5             #
240:     lsr.w   #1, d5             #
241:     move.w  d5, (a4)+          #LX
242:     move.w  d1, (a4)+          #Y
243:
244:     subq.w  #1, d3             #
245:     bmi     bound              #走査範囲右端に達した
246:
247: sloop2: cmp.w  (a0)+, d6        #領域色部分を飛ばす
248:     dbne    d3, sloop2         #
249:     beq     bound              #
250:     subq.l  #2, a0             #
251:
252: bound: move.l  a0, d5           #d5 = 領域色部分右端アドレス
253:     subq.l  #2, d5
254:
255:     move.l  d5, (a4)+          #RADR
256:     sub.l   a1, d5             #
257:     lsr.w   #1, d5             #
258:     move.w  d5, (a4)+          #RX
259:     move.w  d4, (a4)+          #PREVY
260:
261:     cmpa.l  a3, a4             #キューが一杯になったら
262:     beq     drop               # いま登録したシードを捨てる
263:
264:     cmpa.l  BUFEND(a6), a4     #キューの書き込み位置が
265:     bcs     snext              # バッファ末尾に達したら
266:     movea.l BUFTOP(a6), a4     # 先頭を指すよう修正する
267:
268: snext: tst.w  d3               #走査範囲の右端に達するまで
269:     bpl     sloop1            # 繰り返す
270:
271: sdone: rts
272:
273: drop: lea.l  -SEEDSIZ(a4), a4  #いま登録したシードを捨てる
274:     rts
275:
276:     .end
    
```

アルゴリズムの改良にともなう変更部分をざっと拾っておこう。まず、キューに格納するシード候補についての情報が前述のように増えている (31~39行)。シード候補1つあたり、“シード線分”の両端点のG-RAMアドレスと座標 (ただし、y座標は共用)、それに“親ライン”のy座標の16バイトだ。この変更はキューに対する読み書きを行う部分のほかに、リングバッファの最終アドレスを決める部分 (79行) にも影響している。また、情報量が増えたことにより、キューから取り出したシードの情報をレジスタに留めておくのが難しくなったため、その一部をいったんワークに格納するようにせざるをえなくなった (99, 116~117行)。

シードから左右方向へ塗り潰しの境界を探す122~140行は、走査範囲が重複しないよう修正されている。そして、シードの上下ラインを走査するサブルーチンseapixには、親ラインとの重複部分を飛ばす処理 (203~224行) と、領域色ピクセル列の左端、現在処理中のシードのy座標をキューに登録する処理がつけ加えられた (238~242行)。

速度試験をしてみた結果、リスト3はIOCS.X版のPAINTよりもわずかに速くなったようだ。もっとも、その差はごくわずかでしかない。IOCS呼び出しのオーバーヘッドがない分だけ速いと考えるのが正解かもしれない。なお、先に用意したテストプログラム (リスト2) は、純粋に動作試験用であり、速度試験用ではない。リスト2で速度比較をすると、単純さが幸いしてリスト1の版がIOCSを含むほかのどれよりも高速だという楽しい結果が出る。速度比較がしたかったら、もっと一般的な図形で試してみしてほしい。

ペイントルーチンのバリエーション

いま作成したgpaint (や、IOCSコールのPAINT) は塗り潰す領域があらかじめ単色になっていなければならないため、複雑な絵のうえに適当な輪郭を描いてその内側を塗り潰すといった目的には使えない。そこで、塗り潰す領域を“ある境界色で囲まれた範囲”で指定する、ペイントルーチンの別版を作成してみた (リスト4)。リスト3からの変更点のみを示してある (“+”のついた行は追加を意味する)。基本的には、シードを探す部分の条件判断を“領域色であれば~”から“境界色でなければ~”にしたただけだ。描画色自体は無条件に境界色となるために、パレットコードの比較を2回行わなければならない点にだけ注意しよう。

リスト4
GPAINT_B.S
(リスト3に対する追加
・変更部)

```

1: *      シードフィルによる領域の塗り潰し (境界定義版)
7:      .xdef    gpaint_b

16: COL:      .ds.w    1      *描画色
+ BCOL:      .ds.w    1      *境界色
17: TEMP:     .ds.l    1      *作業用領域先頭

44: gpaint_b:

64:      move.w  BCOL(a1), d6      *d6 = 境界色
65:
```



```

66:      move.w COL(a1), d7      *d7 = 描画色
67:      *
68:      *
69:      swap.w d7              *

113:     cmp.w (a1)+, d7          *すでに処理済みのシードならば
114:     beq next                 * 捨てる

126: rloop: move.w (a1)+, d5      *
+      cmp.w d7, d5            *
+      beq rchkq               *
+      cmp.w d6, d5            *
127:     dbeq d3, rloop           *
128:     bne lchk               *
129: rchkq: subq.l #2, a1         *

137: lloop: move.w -(a0), d5      *
+      cmp.w d7, d5            *
+      beq lchkq               *
+      cmp.w d6, d5            *
138:     dbeq d3, lloop           *
139:     bne filspn              *
140: lchkq: addq.l #2, a0         *

231: sloop1: move.w (a0)+, d5     *境界色部分を飛ばす
+      cmp.w d6, d5            *
+      beq skip0               *
+      cmp.w d7, d5            *
232: skip0: dbne d3, sloop1       *
233:     beq sdone               *

247: sloop2: move.w (a0)+, d5     *非境界色部分を飛ばす
+      cmp.w d6, d5            *
+      beq skip1               *
+      cmp.w d7, d5            *
248:     dbeq d3, sloop2         *
249:     bne bound               *
250: skip1: subq.l #2, a0         *

```

また、リスト5は16色モード専用ながら、境界色を複数指定できるように拡張した版だ。やはり、リスト3からの変更点のかたちで示してある。境界色はパレットコードを並べて指定するのではなく、パレットコードの該当ビットを1にした16ビットデータで指定するようになっている。たとえば、パレットコード1, 4, 13を境界色としたければ、ビット1, 4, 13が1で残りが0であるような値、

0010000000010010₈ (=8210)

を指定すればよい。この形式を採用したことにより、境界色かどうかの判定がbtst命令一発ですんでいることを指摘しておく。

.....
リスト5
GPAINT16.S
(リスト3に対する追加
・変更部)

```

1:      *      シードフィルによる領域の塗り潰し (16色専用境界定義版)
7:      .xdef   gpaint_b16

16: COL:      .ds.w 1          *描画色
+      BCOL:   .ds.w 1          *境界色
17: TEMP:     .ds.l 1          *作業用領域先頭

44: gpaint_b16:

64:      move.w BCQL(a1), d6      *d6 = 境界色
65:

```

```

66:      move.w  COL(a1), d7      *d7 = 描画色
67:      bset.l  d7, d6           *描画色は常に領域色
68:      *
69:      swap.w  d7               *

113:      cmp.w   (a1)+, d7       *すでに処理済みのシードならば
114:      beq      next           * 捨てる

126:  rloop: move.w  (a1)+, d5      *
+      btst.l   d5, d6          *
127:      dbne    d3, rloop       *

137:  lloop: move.w  -(a0), d5      *
+      btst.l   d5, d6          *
138:      dbne    d3, lloop       *

231:  sloop1: move.w  (a0)+, d5      *境界色部分を飛ばす
+      btst.l   d5, d6          *
232:      dbeq    d3, sloop1      *

247:  sloop2: move.w  (a0)+, d5      *非境界色部分を飛ばす
+      btst.l   d5, d6          *
248:      dbeq    d3, sloop2      *

```

CHAPTER

6

円を描く

円を描く



この章では円を取り扱う。

たぶん、円を描く方法といわれて最初に思いつくのは、

$$x^2 + y^2 = R^2$$

あるいは、

$$x = R \cos \theta$$

$$y = R \sin \theta$$

で表される円の方程式から座標を直接算出する方法だろう。だが、どちらの式を使うにしろ実数演算が必要だから、マシン語にするのには難があるし、速度効率もたかがしれている。というわけで、整数演算しか使わずに円を描くアルゴリズムを導くところから始める。

円描画アルゴリズムを導く

話をかんたんにするために、円の8分の1、45°分だけを描くことに集中する。図1に示すような原点 (0,0) を中心とする半径Rの8分円を点 (R, 0) から時計回りに描いていく場合を考えよう (図1では、グラフィック画面にあわせてy軸を下向きにとっている)。この範囲での円弧は45°よりも急な傾きの曲線となるから、描画の途中、ある点P (x, y) を打ったあと、次に打つべき点は、

真下の点P_v (x, y+1)

左下の点P_d (x-1, y+1)

のどちらかに限定される。円を描く問題は、この2点のうち真の円に近い方を選ぶ問題に帰着する。

点P_v、P_dのどちらが真の円に近いかを知るためには、それぞれの点について円周からの距離を求めてみればよい¹⁾。点 (x, y) の原点からの距離は、

$$\sqrt{x^2 + y^2}$$

だから、

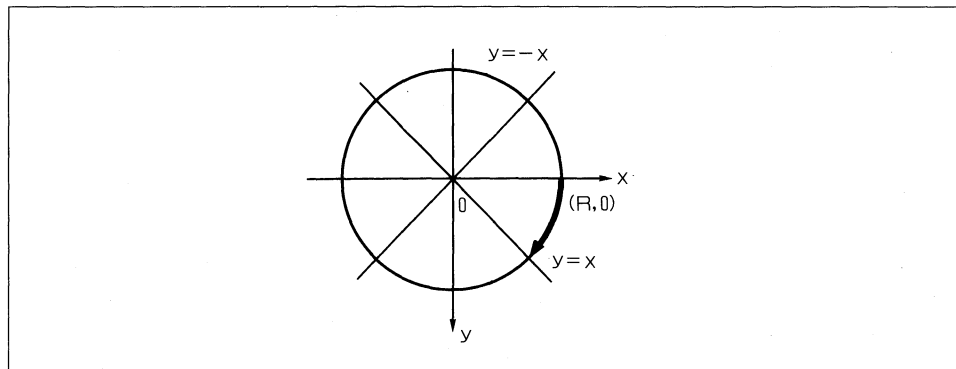


図1
第1八分円

$$D = \sqrt{x^2 + y^2} - R$$

により、円からの距離 D が符号付きで得られる（符号は点が円の内側にあるか外側にあるかを表す）。もっとも、わざわざ平方根をとって正確な距離を求めなくても、点 (x, y) の原点からの距離の2乗と半径 R の2乗との差、

$$E = x^2 + y^2 - R^2$$

が円からの距離の大小を反映する。以下、この E を誤差と呼ぶ。

- 1) もっとかんたんに、 P_V と P_D の中点が円の内側にあるか外側にあるかを調べれば、 P_V と P_D のどちらが円に近いかが判断できるように見えるかもしれないが、それは誤り。

$P_V(x, y+1)$ についての誤差 E_V は、

$$E_V = x^2 + (y+1)^2 - R^2$$

点 $P(x, y)$ についての誤差 E を使って書き直せば、

$$E_V = E + 2y + 1$$

となる。同様に、 $P_D(x-1, y+1)$ についての誤差 E_D は、

$$\begin{aligned} E_D &= (x-1)^2 + (y+1)^2 - R^2 \\ &= E - 2x + 2y + 2 \end{aligned}$$

と表せる。こうして求めた E_V 、 E_D をくらべてみて、

$$|E_V| - |E_D| < 0$$

なら点 P_V 、

$$|E_V| - |E_D| > 0$$

なら点 P_D が真の円に近い。

すでにこの時点で、整数演算だけを使って円を描く用途が立ったわけだが、実用に向けてもう少し整理しておく。点 P_V 、 P_D と真の円との関係を場合分けしてみると、

- 1) 点 P_v , P_D ともに円の外側にある (点 P_D が円周上にある場合も含む)
- 2) 点 P_v が円の外側, 点 P_D が円の内側にある
- 3) 点 P_v , P_D ともに円の内側にある (点 P_v が円周上にある場合も含む)

の3通りが考えられる (図2)。1) の場合は点 P_D , 3) の場合は点 P_v を採用すればよいのは明らかなのでとりあえず後回しにして, いちばん問題になりそうな2) のケースに注目する。この場合,

$$E_v > 0$$

$$E_D \geq 0$$

より,

$$|E_v| - |E_D| = E_v + E_D$$

これに上の E_v , E_D を代入して,

$$2E - 2x + 4y + 3 (=F \text{ とおく})$$

を得る。 E_v , E_D を個別に求め, 絶対値をとってから比較する代わりに, F をスイッチにして,

$$F < 0 \text{ なら } P_v$$

$$F > 0 \text{ なら } P_D$$

を選ぶべきだと直接判断できるわけだ。

F を毎回1から計算する必要はないことに気付いてほしい。 F は, x が1減ると $-4x+4$, y が1増えると $4y+6$ 増えるから,

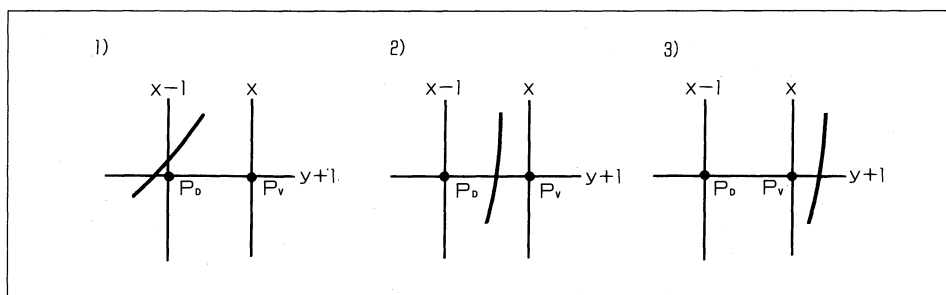


図2
真の円との位置関係

・ $P_v (x, y+1)$ を採用したとき

$$F = F + (4y + 6)$$

・ $P_D (x-1, y+1)$ を採用したとき

$$F = F + (4y + 6) - (4x - 4)$$

というように, 採用した点に応じた変化量を加えていくだけで求まる。 F を真面目に計算するのは描画開始点についてだけでよい。で, その F の初期値 F_0 は上の F の式に $(R, 0)$ を代入すれば,

$$F_0 = -2R + 3$$

と得られる。

さて、試しにFの符号が図2の1), 3) のケースでどうなるか調べてみよう。1) の場合は、

$$E_v > 0$$

$$E_d \geq 0$$

より、

$$F = E_v + E_d > 0$$

また、3) の場合、

$$E_v \leq 0$$

$$E_d < 0$$

$$F = E_v + E_d < 0$$

だ。

$$F < 0 \text{ なら } P_v$$

$$F > 0 \text{ なら } P_d$$

を選ぶという規則はここでも成り立っている。P_v、P_dと真の円との位置関係がどうなっているかと、Fの符号により正しい点を選択できるのだ。

特定の条件をつけて導いた式を一般化するのはなんとなくインチキ臭いので、念のため、裏付けをとっておこう。不等式、

$$|E_v| - |E_d| < 0$$

は、

$$E_d^2 - E_v^2 < 0$$

と等価だ。この不等式の左辺を因数分解すると、

$$(E_v + E_d)(E_v - E_d) < 0$$

$$F(2x - 1) < 0$$

を得る。したがって、

$$x \geq 1/2$$

の条件つきで、 $|E_v| - |E_d|$ と $F = E_v + E_d$ の符号は一致する。ところが、 $x \geq 1/2$ はつねに成り立つ。なぜなら、いま描こうという8分円はy軸とは交わらないから、xは0より大きな整数、つまり、1以上であることが保証されているのだ。

ただし、例外が1つある。半径Rが0の場合だ。このときだけはxが0になるから、上の仮定は通用しない。しかし、R=0なら1点打った段階で描画が完了するわけであり、誤った座標を求めたとしても、その座標が使われることはない。結局、R=0の場合も特別扱いしなくてよいということがわかる。

以上をまとめると、原点を中心とする半径Rの8分円を描くアルゴリズムは次のようになる。

- 1) $x=R$, $y=0$, $F=-2R+3$ と初期化する
- 2) (x, y) に点を打つ

3) $F \geq 0$ であれば, $x = x - 1$, $F = F - 4x$

4) 無条件に $y = y + 1$, $F = F + 4y + 2$

5) $x \geq y$ なら2)に戻る

Fを更新する式が先ほどと異なっているが驚かないでほしい。増減する前のx, yを使って表していた式を, 増減後のx, yを使って書き直ただけだ。

ここまできたらあとは, 点 (x, y) と同時に,

(-x, y)

(x, -y)

(-x, -y)

(y, x)

(-y, x)

(y, -x)

(-y, -x)

にも点を打てば円全体が描けるし, また, これら8点の座標に円の中心座標を加えて平行移動してやれば, 任意の点を中心とする円が描けるようになる。参考までに, リスト1に完全な私たちのX-BASIC版円描画ルーチンを示しておく。

.....
リスト1
CIRCLE.BAS

```
10 int x0,y0,r,x,y,f
20 screen 2,0,1,1
30 /*
40 x0=384:y0=256:r=200
50 circle(x0,y0,r,3)
60 x=r:y=0
70 f=-2*r+3
80 while x>y
90   pset(x0+x,y0+y,15)
100  pset(x0-x,y0+y,15)
110  pset(x0+x,y0-y,15)
120  pset(x0-x,y0-y,15)
130  pset(x0+y,y0+x,15)
140  pset(x0-y,y0+x,15)
150  pset(x0+y,y0-x,15)
160  pset(x0-y,y0-x,15)
170  if f>=0 then {
180    x=x-1
190    f=f-4*x
200  }
210  y=y+1
220  f=f+4*y+2
230 endwhile
```

円描画ルーチンの作成

では, マシン語版の円描画ルーチンを見てもらおう。リスト2のサブルーチンgcircleだ。gcircleには円の中心座標と半径, 描画色を引数として渡す。IOCSコールのCIRCLEになら

い、中心座標は-32768～+32767、半径は0～32767の範囲の値を有効とみなすようになっている。もっとも、IOCSのCIRCLEには、極端な座標を与えると演算過程でオーバーフローして正しい結果が得られないという不具合があり、有効な座標値の範囲はマニュアルに記されているよりも狭い。X-BASICで、

```
circle (-32000,0,32500,65535)
```

を実行してみると、円弧とは似ても似つかないギザギザな線が描かれるのが確認できるだろう。gcircleは、この点、真面目につくってある。

リスト2
GCIRCLE.S

```
1: *      円を描く
2:
3:      .include      gconst.h
4: *
5:      .xdef      gcircle
6:      .xref      gramadr
7:      .xref      cliprect
8: *
9: PSET      macro      X, Y, COL, ADR      *クリッピングしつつ
10:          local      skip      * 点を打つマクロ
11:          movea.l      a4, a5      *
12:          cmp.l      (a5)+, X      *
13:          blt      skip      *
14:          cmp.l      (a5)+, Y      *
15:          blt      skip      *
16:          cmp.l      (a5)+, X      *
17:          bgt      skip      *
18:          cmp.l      (a5)+, Y      *
19:          bgt      skip      *
20:          move.w      COL, ADR      *
21: skip:      *
22:          .endm      *
23: *
24:          .offset 0      *gcircleの引数構造
25: *
26: X0:      .ds.w      1      *中心座標
27: Y0:      .ds.w      1      *
28: R:      .ds.w      1      *半径
29: COL:      .ds.w      1      *描画色
30: *
31:          .offset 0      *クリッピング領域
32: *
33: LMINX:      .ds.l      1
34: LMINY:      .ds.l      1
35: LMAXX:      .ds.l      1
36: LMAXY:      .ds.l      1
37: LRECT:
38: *
39:          .text
40:          .even
41: *
42: gcircle:
43: ARGPTR      =      8
44:          link      a6, #-LRECT
45:          movem.l      d0-d7/a0-a5, -(sp)
46:
47:          movea.l      ARGPTR(a6), a1      *a1 = 引数列
48:
49:          movem.w      (a1), d0-d2/d7      *(d0, d1) = (x0, y0) = 中心座標
50:          *d2 = R = 半径
51:          *d7 = 描画色
52:
53:          tst.w      d2      *R ≤ 0ならば
54:          bmi      done      * エラー終了
55:
```

```

56:      lea.l    -LRECT(a6), a4    * (x0, y0) が原点になるよう
57:      lea.l    cliprect, a5      * クリッピング領域を
58:                                     * 平行移動する
59:      moveq.l   #2-1, d6          *
60: rloop: moveq.l   #0, d5          *
61:      move.w    (a5)+, d5         *
62:      sub.l     d0, d5            *
63:      move.l     d5, (a4)+        *
64:                                     *
65:      moveq.l   #0, d5            *
66:      move.w    (a5)+, d5         *
67:      sub.l     d1, d5            *
68:      move.l     d5, (a4)+        *
69:                                     *
70:      dbra      d6, rloop         *
71:                                     *
72:      lea.l     -LRECT(a4), a4    * a4 = シフト後のクリッピング領域
73:
74:      jsr       gramadr           * a0 = (x0, y0) のG-RAMアドレス
75:
76: *****以下 座標値はすべて (-x0, -y0) のゲタ履き表現*****
77:
78:      move.l     d2, d0            * a2 = (0, R) のG-RAMアドレス
79:      moveq.l    #GSFICTR, d6     *
80:      lsl.l      d6, d0            *
81:      lea.l      0(a0, d0.l), a2   *
82:      neg.l      d0                * a3 = (0, -R) のG-RAMアドレス
83:      lea.l      0(a0, d0.l), a3   *
84:
85:      move.l     d2, d0            * (d0, d1) = (x, y) = (R, 0)
86:      moveq.l    #0, d1            *
87:
88:      add.w      d2, d2            * d2 = 2R
89:      adda.l     d2, a0            * a0 = a1 = (x, y) のG-RAMアドレス
90:      movea.l    a0, a1            *
91:
92:      move.l     d2, d4            * d4 = -2R+1 = F-2
93:      neg.l      d4                *
94:      addq.l     #1, d4            *
95:
96:      add.l      d2, d2            * d2 = 4x = 4R
97:      moveq.l    #2, d3            * d3 = 4y+2 = 2
98:
99:      move.l     d2, d5            * d5 = -4R = (x, y) と (-x, y) との
100:      neg.l      d5                * アドレスの差
101:      moveq.l    #0, d6            * d6 = 0 = (y, x) と (-y, x) との
102:                                     * アドレスの差
103: loop:
104: P0      equ     (a0)              * -x -y y x
105: P1      equ     (a1)              * -x P7 P3
106: P2      equ     (a2)              * -y P5 P1
107: P3      equ     (a3)              *
108: P4      equ     0(a0, d5.l)        *
109: P5      equ     0(a1, d5.l)        * y P4 P0
110: P6      equ     0(a2, d6.l)        * x P6 P2
111: P7      equ     0(a3, d6.l)        *
112:
113: PSET     d0, d1, d7, P0            * P0 (x, y)
114: PSET     d1, d0, d7, P2            * P2 (y, x)
115: neg.l     d1
116: PSET     d0, d1, d7, P1            * P1 (x, -y)
117: PSET     d1, d0, d7, P6            * P3 (-y, x)
118: neg.l     d0
119: PSET     d0, d1, d7, P5            * P5 (-x, -y)
120: PSET     d1, d0, d7, P7            * P7 (-y, -x)
121: neg.l     d1
122: PSET     d0, d1, d7, P4            * P4 (-x, y)
123: PSET     d1, d0, d7, P3            * P6 (y, -x)
124: neg.l     d0
125:
126: add.l     d3, d4                  * F += 4y+2
127: bmi      vmove                    * F < 0 ならば垂直移動
128:
129: dmove:                                     * 斜め移動

```

```

130:      subq.w #1, d0      *x--
131:      subq.l #4, d2      *4x
132:      sub.l  d2, d4      *F -= 4x
133:
134:      subq.l  #2, a0      *P0を左へ移動
135:      subq.l  #2, a1      *P1を左へ移動
136:      lea.l   -GNBYTE(a2), a2 *P2を上へ移動
137:      lea.l   GNBYTE(a3), a3  *P3を下へ移動
138:      addq.l  #4, d5      *P4をP0に近付ける
139:
140: vmove:      *垂直移動
141:      addq.w  #1, d1      *y++
142:      addq.l  #4, d3      *4y+2
143:
144:      lea.l   GNBYTE(a0), a0  *P0を下へ移動
145:      lea.l   -GNBYTE(a1), a1 *P1を上へ移動
146:      addq.l  #2, a2      *P2を右へ移動
147:      addq.l  #2, a3      *P3を右へ移動
148:      subq.l  #4, d6      *P6をP2から遠ざける
149:
150:      cmp.w   d1, d0      *x≥yのあいだ
151:      bge     loop      * 繰り返す
152:
153: done:      movem.l (sp)+, d0-d7/a0-a5
154:      unlk    a6
155:      rts
156:
157:      .end

```

円描画ルーチンではクリッピング周りが泣きどころだ。円のクリッピングは、演算で求めた円周上の各点がクリッピングウィンドウ内に収まっているかどうかを個別に調べるという泥臭い方法で行うよりない。ループ1回につき、8点の座標をウィンドウの4辺とそれぞれ比較するわけだ。円描画アルゴリズムの簡潔さを考えると、クリッピングが実行時間に占める割合は予想以上に大きい。56～72行ではこのクリッピングの手間を多少軽減するために予備工作をしている。

円描画アルゴリズムでは、中心が原点にあるものと仮定して円周上の座標 (x, y) を算出し、それに円の中心座標 (x₀, y₀) を加えて最終的な描画位置を求める。(x, y) は中心からの相対的な座標だ。56～72行の目的は、ウィンドウ自身を相対座標に変換しておくことで、クリッピング時に中心座標を加える手間を省くことにある。

$$x + x_0 \geq \text{MINX}$$

のような座標比較 (ただし、MINXはウィンドウ左端x座標) を、

$$x \geq \text{MINX} - x_0$$

に置き換え、定数となる右辺を先に計算しておくわけだ。

一見、この細工は実行速度の向上にはつながらないようにみえるかもしれない。ループの中で毎回行う加算をループの外に出したとはいえ、上の式の $\text{MINX} - x_0$ は32ビットで求める必要があり (16ビット符号付き数ではオーバーフローする可能性がある)、それにあわせて、クリッピング時の座標比較も32ビットで行うかたちにせざるをえない。通常、座標の大小は16ビット比較で判定できることを考えると、かえって速度を落としそうな気がするだろう。

しかし、真正直に

$$x + x_0 \geq \text{MINX}$$

を16ビットで比較するのには、ひと工夫いる。 $x+x_0$ は16ビット符号付き数に収まらない可能性があるから、加算後にccrのVビットをチェックするためのbvs命令が余計に必要だ²⁾。これで、16ビット比較の優位性は帳消しになる。くわえて、 $x+x_0$ の計算には空きレジスタが必要だから、ほかのところで使えるレジスタの数に制限が生じ、間接的に速度を落とす原因となる。結局、この場面では、16ビット比較にこだわるのは得策ではないのだ。納得できない人は、実際に別版をつくってリスト2と速度比較するなり、卓上でクロック数を計算するなりしてみるとよいだろう。

- 2) add.wでxに x_0 を加えた結果が-32768未満、あるいは、+32768以上であればVビットが立つ。その場合、ウィンドウの辺との比較は無意味であり、また、ウィンドウ外であることは明らかなので、bvsで弾くわけだ。

74行では、サブルーチンgramadrを呼び出して、中心点 (x_0, y_0) のG-RAMアドレスをa0に求めている³⁾。このアドレスを先に求めておくことで、円周上の点のG-RAMアドレスが相対座標から直接（中心の座標を加えて絶対座標に変換せずに）計算できるようになる。もっとも、リスト2では1点ずつG-RAMアドレス計算をするような冗長なことはやっていない。図3を見てほしい。便宜上、平行して描く8点に $P_0 \sim P_7$ の名前をつけた。円描画アルゴリズムで直接求める点は P_0 で、残りは線対称変換により得る点だ。リスト2では、 $P_0 \sim P_3$ のG-RAMアドレスをつねにa0～a3に入れておき（88～90行、78～83行で初期化）、座標と平行して更新するようになっている。また、 $P_0 \sim P_3$ のG-RAMアドレスから $P_4 \sim P_7$ のG-RAMアドレスが求められるよう、d5, d6には、それぞれ、 $P_0 \sim P_4$ 間、 $P_2 \sim P_6$ 間のアドレスの差を入れてある（99～101行で初期化）。74行で求めた中心点のG-RAMアドレスは、 $P_0 \sim P_3$ に対応するG-RAMアドレスの初期値を計算するのにだけ使う。

- 3) この時点では、まだ中心座標をクリッピングしていない。サブルーチンgramadrはこのような使い方に備え、任意の16ビット符号付き数の座標値を受けつけ、筋の通った値を返すようにつくってある。

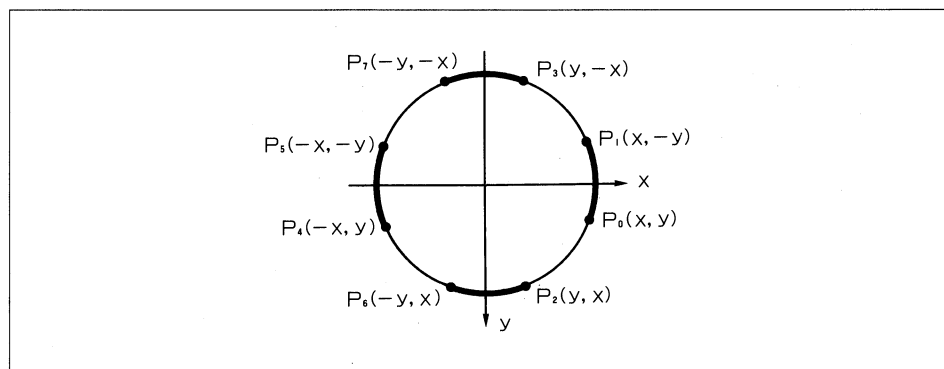


図3
gcircleの描画の様子

これらG-RAMアドレス関係の初期化に交じって、85～86行で座標、92～97行でFとその変化量の初期値をレジスタに求めている。Fの初期値は先ほど導いた値よりも2小さいが、これについてはあとで触れよう。ここでのポイントは、Fを更新するときに加減算する変化量を毎回x, yから計算しなくてもすむように、 $4x$ や $4y+2$ をレジスタに格納しておくようにしてあることだ。

103行からメインループが始まる。113～124行では、9～22行で定義したマクロPSETを使って8か所に点を打っている。マクロPSETは第1, 第2引数で指定した点がクリッピングウィンドウに収まっているかどうかを調べ、収まっていれば第3引数のパレットコードを第4引数で指定したアドレスに書き込む。第4引数として104～111行で定義したシンボルを使った⁴⁾のは、AS.Xのマクロの間抜けな仕様を回避するためだ。

PSET d0,d1,d7,0 (a0,d5.l)

のように直接記述すると、“0 (a0,d5.l)”中の“,”が引数の区切りと見なされてしまうのだ⁵⁾。

■ 4) このように、equではシンボルに値ではなく文字列を定義することもできる。

■ 5) たいてい、マクロアセンブラではこのような場合に備えて引数を括弧することができるようにしているものなのだ。

アルゴリズムどおりだと、点を打ったら、続いてFの符号を調べるわけだが³⁾、リスト2ではこのタイミングでFに $4y+2$ を加えている (126行)。ちょっと考えてもらえればわかると思うが、このように処理の順序を変更することで、tst命令を1個省略できる。もちろん、ただ処理順序を変えたのでは、Fの値が狂ってしまう。その補正のために、Fの初期値を2(= y が0のときの $4y+2$) 小さくしてあったのだ。

Fの符号に応じ、129～148行で次に打つべき点を求める。Fが負のときは真下の点を採用すればよいのだから、140～149行でy座標を1増やす。また、Fが非負の場合は、129～138行でx座標を1減らしてから、140行に合流してy座標も更新する。座標を増減するときには、同時に、

- ・ $4x$, $4y+2$ を保持するd2とd3
- ・ $P_0 \sim P_3$ のG-RAMアドレスを保持するa0～a3
- ・ 点 $P_0 \sim P_4$ 間, 点 $P_2 \sim P_6$ 間のG-RAMアドレスの差を保持するd5とd6

を、それぞれ更新している。

以上の処理を $x \geq y$ のあいだ繰り返せば、円が描ける。そのループの終了判定をするのが150～151行だ。比較を符号付きで行っている点に注意したい。無符号で比較してしまうと、半径が0の場合に処理が止まらなくなる。

楕円への対応

リスト2のgcircleはレジスタを贅沢に使った、比較的高速なルーチンに仕上がった⁶⁾。しかし、“グラフィック画面にとっての真円”しか描けないという点で、実用性には乏しい。表示画面が512×512などの画面モードで見かけ上の真円を描くためには、ピクセルの縦横比を考慮し、計算上は楕円を描く必要がある。

■ 6) まだ細工の余地はある。まず、斜め下の点のG-RAMアドレス算出を水平方向と垂直方向の2回に分けて行っている部分をまとめればいくらか速くなる。また、プログラムがごちゃごちゃしてしまってヤブヘビかもしれないが、クリッピング時の座標比較回数も少なくとも半分には減らせる。さらに、クリッピングがまったく不要の場合（円が完全にウィンドウ内にある場合）を特別扱いし、クリッピングの処理を省いた専用ルーチンで処理するという手もある。

あまり質を追及しないのであれば、リスト2を楕円対応に拡張することはそう難しいことではない。座標の算出自体は真円のつもりで行い、点を打つ直前に、縦長の楕円であればx座標、横長の楕円であればy座標に偏平率をかけて縮小すればよい。この方法では描かれる楕円の輪郭がところどころ太くなってしまうのが難点だが、あらたに楕円を描くアルゴリズムを導くのも面倒なので今回は妥協する（描画結果を見るかぎり、IOCSのCIRCLEもそうしているようだし）。

リスト3がその楕円描画ルーチンだ。サブルーチン名はいちおうgovalに変更しておいた。楕円の潰れぐあいの指定方法はIOCSとは異なり、楕円の横方向の幅と縦方向の幅を別々に指定するようになっている（24～30行）。

リスト3
GOVALS

```

1: *      楕円を描く
2:
3:      .include      gconst.h
4: *
5:      .xdef      goval
6:      .xref      gramadr
7:      .xref      cliprect
8: *
9: PSET   macro X,Y,COL,ADR      *クリッピングしつつ
10:      local skip              * 点を打つマクロ
11:      movea.l a4,a5            *
12:      cmp.l (a5)+,X            *
13:      blt skip                 *
14:      cmp.l (a5)+,Y            *
15:      blt skip                 *
16:      cmp.l (a5)+,X            *
17:      bgt skip                 *
18:      cmp.l (a5)+,Y            *
19:      bgt skip                 *
20:      move.w COL,ADR           *
21: skip:                          *
22:      .endm                    *
23: *
24:      .offset 0                *govalの引数構造
25: *
```

```

26: X0:      .ds.w 1      *中心座標
27: Y0:      .ds.w 1      *
28: XR:      .ds.w 1      *水平方向半径
29: YR:      .ds.w 1      *垂直方向半径
30: COL:     .ds.w 1      *描画色
31: *
32:          .offset 0     *クリッピング領域
33: *
34: LMINX:    .ds.l 1
35: LMINY:    .ds.l 1
36: LMAXX:    .ds.l 1
37: LMAXY:    .ds.l 1
38: LRECT:
39: *
40:          .offset -56    *ワーク
41: *
42: WORKTOP:
43: X1:      .ds.w 1      *第1八分円x座標
44: Y1:      .ds.w 1      *第1八分円y座標
45: X2:      .ds.w 1      *第2八分円x座標
46: Y2:      .ds.w 1      *第2八分円y座標
47: EX1:     .ds.l 1      *x縮小誤差項1
48: EY1:     .ds.l 1      *y縮小誤差項1
49: EX2:     .ds.l 1      *x縮小誤差項2
50: EY2:     .ds.l 1      *y縮小誤差項2
51: EXDX:    .ds.l 1      *x縮小誤差項補正值
52: EXDY:    .ds.l 1      *x縮小誤差項増分
53: EYDX:    .ds.l 1      *y縮小誤差項補正值
54: EYDY:    .ds.l 1      *y縮小誤差項増分
55: CRECT:   .ds.b LRECT  *シフトしたウィンドウ
56: *
57:          .text
58:          .even
59: *
60: goval:
61: ARGPTR =      8
62:      link      a6, #WORKTOP
63:      movem.l d0-d7/a0-a5, -(sp)
64:
65:      movea.l ARGPTR(a6), a1      *a1 = 引数列
66:
67:      movem.w (a1), d0-d3/d7      *(d0, d1) = (x0, y0) = 中心座標
68:                                  *d2 = XR = 水平方向半径
69:                                  *d3 = YR = 垂直方向半径
70:                                  *d7 = 描画色
71:
72:      tst.w d2      *XR<0ならば
73:      bmi done      * エラー終了
74:      tst.w d3      *YR<0ならば
75:      bmi done      * エラー終了
76:
77:      lea.l CRECT(a6), a4      *(x0, y0)が原点になるよう
78:      lea.l cliprect, a5      * クリッピング領域を
79:                                  * 平行移動する
80:      moveq.l #2-1, d6      *
81: rloop: moveq.l #0, d5      *
82:      move.w (a5)+, d5      *
83:      sub.l d0, d5      *
84:      move.l d5, (a4)+      *
85:                                  *
86:      moveq.l #0, d5      *
87:      move.w (a5)+, d5      *
88:      sub.l d1, d5      *
89:      move.l d5, (a4)+      *
90:                                  *
91:      dbra d6, rloop      *
92:                                  *
93:      lea.l -LRECT(a4), a4      *a4 = シフト後のクリッピング領域
94:
95:      jsr gramadr      *a0 = (x0, y0)のG-RAMアドレス
96:
97: *****以下 座標値はすべて(-x0, -y0)のゲタ履き表現*****
98:
99:      move.l d3, d0      *a2 = (0, YR)のG-RAMアドレス

```

```

100:      moveq.l #GSFCTR, d6      *
101:      lsl.l   d6, d0           *
102:      lea.l   0(a0, d0, 1), a2  *
103:      neg.l   d0               *a3 = (0, -YR) のG-RAMアドレス
104:      lea.l   0(a0, d0, 1), a3  *
105:
106:      move.l   d3, X2(a6)       *(x2, y2) = (0, YR)
107:      swap.w   d2
108:      move.l   d2, X1(a6)       *(x1, y1) = (XR, 0)
109:      swap.w   d2
110:
111:      move.l   d2, d0           *x縮小誤差項の
112:      neg.l   d0               * 仮初期化
113:      move.l   d0, EX1(a6)      *
114:
115:      move.l   d3, d0           *y縮小誤差項の
116:      neg.l   d0               * 仮初期化
117:      move.l   d0, EY1(a6)      *
118:
119:      add.w    d2, d2           *d2 = 2XR
120:      add.w    d3, d3           *d3 = 2YR
121:      subq.l   #2, d2           *d2 = 2(XR-1)
122:      subq.l   #2, d3           *d3 = 2(YR-1)
123:      move.l   d3, EXDX(a6)     *x縮小誤差項補正值
124:      move.l   d2, EXDY(a6)     *x縮小誤差項増分
125:      move.l   d2, EYDX(a6)     *y縮小誤差項補正值
126:      move.l   d3, EYDY(a6)     *y縮小誤差項増分
127:      addq.l   #2, d2           *d2 = 2XR
128:      addq.l   #2, d3           *d3 = 2YR
129:
130:      adda.l   d2, a0           *a0 = a1 = (x1, y1) のG-RAMアドレス
131:      movea.l  a0, a1          *
132:
133:      move.l   d2, d5           *d5 = 2XR
134:      add.l    d5, d5           *d5 = -4XR = (x1, y1) と (-x1, y1) との
135:      neg.l    d5               * アドレスの差
136:      moveq.l  #0, d6           *d6 = 0 = (x2, y2) と (-x2, y2) との
137:                               * アドレスの差
138:
139:      cmp.w    d3, d2           *2XRと2YRの大きい方を
140:      bcc      init1            * d2に残す
141:      move.w   d3, d2           *d2 = max(2XR, 2YR) = 2R
142:
143: init0: move.l  d6, EY1(a6)      *縦長楕円だから
144:      move.l  d6, EYDX(a6)      * yは縮小しない
145:      move.l  d6, EYDY(a6)      *
146:      bra     init2
147:
148: init1: move.l  d6, EX1(a6)      *横長楕円だから
149:      move.l  d6, EXDX(a6)      * xは縮小しない
150:      move.l  d6, EXDY(a6)      *
151:
152: init2: move.l  d2, d4           *d4 = -2R+1 = F-2
153:      neg.l   d4               *
154:      addq.l  #1, d4           *
155:
156:      add.l   d2, d2           *d2 = 4x = 4R
157:      moveq.l #0, d3           *d3 = 4y = 0
158:
159:      move.l  EX1(a6), EX2(a6)  *x縮小誤差項2を初期化
160:      move.l  EY1(a6), EY2(a6) *y縮小誤差項2を初期化
161:
162: loop:
163:      reg     (a0)             * -x1-x2 x2 x1
164:      reg     (a1)             * -y2 P7 P3
165:      reg     (a2)             * -y1 P5 P1
166:      reg     (a3)             *
167:      reg     0(a0, d5, 1)      *
168:      reg     0(a1, d5, 1)      * y1 P4 P0
169:      reg     0(a2, d6, 1)      * y2 P6 P2
170:      reg     0(a3, d6, 1)      *
171:
172:      movem.w X1(a6), d0-d1
173:      PSET    d0, d1, d7, P0   *P0(x1, y1)

```



```

174:      neg. l    d1
175:      PSET     d0, d1, d7, P1      #P1 (x1, -y1)
176:      neg. l    d0
177:      PSET     d0, d1, d7, P5      #P5 (-x1, -y1)
178:      neg. l    d1
179:      PSET     d0, d1, d7, P4      #P4 (-x1, y1)
180:
181:      movem. w  X2 (a6), d0-d1
182:      PSET     d0, d1, d7, P2      #P2 (x2, y2)
183:      neg. l    d1
184:      PSET     d0, d1, d7, P3      #P3 (x2, -y2)
185:      neg. l    d0
186:      PSET     d0, d1, d7, P7      #P7 (-x2, -y2)
187:      neg. l    d1
188:      PSET     d0, d1, d7, P6      #P6 (-x2, y2)
189:
190:      add. l    d3, d4              #F+=4y+2
191:      addq. l   #2, d4              #
192:      bmi      vmove              #F<0ならば垂直移動
193:
194: dmove:      #斜め移動
195:      subq. l   #4, d2              #4x
196:      sub. l    d2, d4              #F -= 4x
197:
198: xcalc1: move. l   EX1 (a6), d0      #縮小を考慮して
199:      add. l    EXDY (a6), d0      # x1を更新する
200:      bmi      xskip1              #
201:      #
202:      subq. w   #1, X1 (a6)         #x1--
203:      subq. l   #2, a0              #P0を左へ移動
204:      subq. l   #2, a1              #P1を左へ移動
205:      addq. l   #4, d5              #P4をP0に近付ける
206:      #
207:      sub. l    EXDX (a6), d0       #
208:      #
209: xskip1: move. l   d0, EX1 (a6)      #
210:
211: ycalc1: move. l   EY2 (a6), d0      #縮小を考慮して
212:      add. l    EYDY (a6), d0      # y2を更新する
213:      bmi      yskip1              #
214:      #
215:      subq. w   #1, Y2 (a6)         #y2--
216:      lea. l    -GNBYTE (a2), a2    #P2を上へ移動
217:      lea. l    GNBYTE (a3), a3     #P3を下へ移動
218:      #
219:      sub. l    EYDX (a6), d0       #
220:      #
221: yskip1: move. l   d0, EY2 (a6)      #
222:
223: vmove:      #垂直移動
224:      addq. l   #4, d3              #4y
225:
226: ycalc2: move. l   EY1 (a6), d0      #縮小を考慮して
227:      add. l    EYDY (a6), d0      # y1を更新する
228:      bmi      yskip2              #
229:      #
230:      addq. w   #1, Y1 (a6)         #y1++
231:      lea. l    GNBYTE (a0), a0     #P0を下へ移動
232:      lea. l    -GNBYTE (a1), a1    #P1を上へ移動
233:      #
234:      sub. l    EYDX (a6), d0       #
235:      #
236: yskip2: move. l   d0, EY1 (a6)      #
237:
238: xcalc2: move. l   EX2 (a6), d0      #縮小を考慮して
239:      add. l    EXDY (a6), d0      # x2を更新する
240:      bmi      xskip2              #
241:      #
242:      addq. w   #1, X2 (a6)         #x2++
243:      addq. l   #2, a2              #P2を右へ移動
244:      addq. l   #2, a3              #P3を右へ移動
245:      subq. l   #4, d6              #P6をP2から遠ざける
246:      #
247:      sub. l    EXDX (a6), d0       #

```

```

248:
249: xskip2: move.l  d0, EX2(a6)      *
250:
251:         cmp.l    d3, d2           *4x ≥ 4yのあいだ
252:         bge     loop             * 繰り返す
253:
254: done:   movem.l  (sp)+, d0-d7/a0-a5
255:         unlink  a6
256:         rts
257:
258:         .end

```

座標の縮小には、Bresenhamの線分発生アルゴリズムを応用している(両者の関係についてはChapter 9で取り上げる)。縦長の楕円と横長の楕円を同一ルーチンで描くようにしたかったので、誤差項の初期値、および、誤差項の増分と補正値はかりに横方向と縦方向それぞれについて求めておき(111～128行)、実際に使わないほうをあとで殺すようにしてある(139～150行)。誤差項の増分と補正値を求める際に、唐突に2を引いている部分があるが(121～122行)、こうしないと45°線上での8分円間の繋ぎ目が変わる。真円を描くときの座標の変化回数と半径との関係がどうなっているか調べれば、このつじつまあわせの意味が見えてくるだろう。

ほかには、ループの終了条件が変わったことに気をつけよう(251行)。これにともない、157行や190～191行も修正されている。

円の塗り潰し

Chapter 4の多角形塗り潰しの要領で、円を塗り潰すことも考えてみよう。やるべきことは単純だ。図3でいう $P_0 \sim P_7$ の8点に点を打つ代わりに、 $P_0 - P_4$, $P_1 - P_5$, $P_2 - P_6$, $P_3 - P_7$ を水平線分で結ぶようにすればよい。リスト4に実例を示す。リスト4はリスト3に対する追加・変更のかたちになっているので、最初から楕円にも対応している。

リスト4
GFILLCIRCLES
(リスト3に対する追加・変更部)

```

1: *      円・楕円を塗り潰して描く
5:      .xdef  gfillcircle

60: gfillcircle:

152: init2:  tst.w   d2              *XR = YR = 0ならば
153:         bne     do              *
154:
155:         movem.w  X1(a6), d0-d1    * (x0, y0) に点を打って終了
156:         PSET    d0, d1, d7, (a0)  *
157:         bra      done
158:
159: do:      move.l  d2, d4           *d4 = -2R+1 = F-2
160:         neg.l   d4              *
161:         addq.l  #1, d4           *
162:
163:         add.l   d2, d2           *d2 = 4x = 4R
164:         moveq.l #0, d3          *d3 = 4y = 0
165:

```

```

166:      move.l  EX1(a6), EX2(a6)  *x縮小誤差項2を初期化
167:      move.l  EY1(a6), EY2(a6) *y縮小誤差項2を初期化
168:
169:      move.w  d7, d0            *d7 = 描画色 (上位/下位ワードとも)
170:      swap.w  d7                *
171:      move.w  d0, d7            *
172:
173:      .xref   ghline
174:      lea.l   ghline, a5        *a5 = 水平線分描画ルーチン
175:
176: loop:
177: *P0      reg    (a0)            * -x1-x2  x2 x1
178: *P1      reg    (a1)            * -y2    P7 .P3
179: *P2      reg    (a2)            * -y1    P5      P1
180: *P3      reg    (a3)            *
181: *P4      reg    0(a0, d5, l)    *
182: *P5      reg    0(a1, d5, l)    * y1  P4      P0
183: *P6      reg    0(a2, d6, l)    * y2    P6  P2
184: *P7      reg    0(a3, d6, l)    *
185:
186:      add.l   d3, d4            *F+=4y+2
187:      addq.l  #2, d4            *
188:      bmi     vmove            *F<0ならば垂直移動
189:
190: dmove:   *斜め移動
191:      subq.l  #4, d2            *4x
192:      sub.l   d2, d4            *F -= 4x
193:
194: xcalc1:  move.l  EX1(a6), d0    *縮小を考慮して
195:      add.l   EXDY(a6), d0      * x1を更新する
196:      bmi     xskip1            *
197:
198:      subq.w  #1, X1(a6)        *x1--
199:      subq.l  #2, a0            *P0を左へ移動
200:      subq.l  #2, a1            *P1を左へ移動
201:      addq.l  #4, d5            *P4をP0に近付ける
202:
203:      sub.l   EXDX(a6), d0      *
204:
205: xskip1:  move.l  d0, EX1(a6)    *
206:
207: ycalc1:  move.l  EYDY(a6), d0  *縮小を考慮して
208:      add.l   d0, EY2(a6)      * y2を更新する
209:      bmi     yskip1            *
210:
211:      movem.w X2(a6), d0-d1     *P2-P6, P3-P7を描く
212:      exg.l   d5, d6            *
213:      exg.l   a0, a2            *
214:      exg.l   a1, a3            *
215:      bsr     drawlines         *
216:      exg.l   d5, d6            *
217:      exg.l   a0, a2            *
218:      exg.l   a1, a3            *
219:
220:      subq.w  #1, Y2(a6)        *y2--
221:      lea.l   -GNBYTE(a2), a2   *P2を上へ移動
222:      lea.l   GNBYTE(a3), a3    *P3を下へ移動
223:
224:      move.l  EYDX(a6), d0      *
225:      sub.l   d0, EY2(a6)      *
226:
227: yskip1:
228: vmove:   *垂直移動
229:      addq.l  #4, d3            *4y
230:
231: ycalc2:  move.l  EYDY(a6), d0  *縮小を考慮して
232:      add.l   d0, EY1(a6)      * y1を更新する
233:      bmi     yskip2            *
234:
235:      movem.w X1(a6), d0-d1     *P0-P4, P1-P5を描く
236:      bsr     drawlines         *
237:
238:      addq.w  #1, Y1(a6)        *y1++
239:      lea.l   GNBYTE(a0), a0    *P0を下へ移動

```

```

240:      lea.l    -GNBYTE(a1), a1    #P1を上へ移動
241:      *
242:      move.l   EYDX(a6), d0        #
243:      sub.l    d0, EY1(a6)         #
244:      *
245: yskip2:
246: xcalc2: move.l   EX2(a6), d0      #縮小を考慮して
247:      add.l    EXDY(a6), d0        # x2を更新する
248:      bmi     xskip2              #
249:      *
250:      addq.w   #1, X2(a6)          #x2++
251:      addq.l   #2, a2              #P2を右へ移動
252:      addq.l   #2, a3              #P3を右へ移動
253:      subq.l   #4, d6              #P6をP2から遠ざける
254:      *
255:      sub.l    EXDX(a6), d0        #
256:      *
257: xskip2: move.l   d0, EX2(a6)      #
258:      *
259:      cmp.l    d3, d2              #4x≥4yのあいだ
260:      bge     loop                 # 繰り返す
261:      *
262:      movem.w  X1(a6), d0-d1       #描画のタイミングがずれた分
263:      bsr     drawlines            #
264:      *
265: done:  movem.l  (sp)+, d0-d7/a0-a5
266:      unlk     a6
267:      rts
268:      *
269:      *
270:      *      P0-P4とP1-P5, あるいは, P2-P6とP3-P7の2本の線分を描く
271:      *
272: drawlines:
273:      movem.l  d2/a0, -(sp)
274:      *
275:      lea.l    0(a0, d5, l), a0    #P0-P4 (またはP2-P6) を描く
276:      movem.l  d0-d1, -(sp)        #
277:      bsr     drawline             #
278:      movem.l  (sp)+, d0-d1        #
279:      *
280:      neg.l    d1                  #P1-P5 (またはP3-P7) を描く
281:      lea.l    0(a1, d5, l), a0    #
282:      bsr     drawline             #
283:      *
284:      movem.l  (sp)+, d2/a0
285:      rts
286:      *
287:      *
288:      *      水平線分を1本描く
289:      *
290: drawline:
291:      cmp.l    CRECT+LMINY(a6), d1  #完全にウィンドウ外の場合を弾く
292:      blt     skip                 #
293:      cmp.l    CRECT+LMAXY(a6), d1  #
294:      bgt     skip                 #
295:      cmp.l    CRECT+LMINX(a6), d0  #
296:      blt     skip                 #
297:      *
298:      move.l   d0, d1              #d1 = 描く線分の右端x座標
299:      neg.l    d0                  #d0 = 描く線分の左端x座標
300:      *
301:      cmp.l    CRECT+LMAXX(a6), d0  #完全にウィンドウ外の場合を弾く
302:      bgt     skip                 #
303:      *
304:      move.l   CRECT+LMINX(a6), d2  #ウィンドウ左端でクリップ
305:      cmp.l    d2, d0              #
306:      bge     minskp               #
307:      sub.l    d2, d0              #
308:      suba.l   d0, a0              #←切り取った分だけ
309:      suba.l   d0, a0              # G-RAMアドレスも更新する
310:      move.l   d2, d0              #
311:      *
312: minskp: move.l   CRECT+LMAXX(a6), d2 #ウィンドウ右端でクリップ
313:      cmp.l    d2, d1              #

```

```

314:         ble     maxskp          #
315:         move.l  d2, d1          #
316:
317: maxskp: sub.l   d0, d1           #dl = 描く線分の長さ-1
318:         addq.w  #1, d1           #dl = 描く線分の長さ
319:         bclr.l  #0, d1           #奇数?
320:         beq     notodd          #
321:         move.w  d7, (a0)+        # 奇数ピクセルの分
322: notodd: neg.w   d1              #
323:
324:         move.l  d7, d0
325:         jsr     0(a5, d1.w)      #水平線分描画
326:
327: skip:    rts
328:
329:         .end

```

リスト4では線分描画のタイミングに注目したい。リスト2やリスト3ではループの先頭にあった描画部分が、ループの途中、“y座標を更新する直前”に移動している(211~218行, 235~236行)。これは無駄な描画を避けるためだ(試しに211~218行, 235~236行を176行の直後に移動すると、実行速度が落ちることが確認できるだろう)。ただ、このように描画のタイミングを遅らせると、円を描き終わらないうちにループを抜ける可能性が出てくる。塗り潰しは円の上下、中央から同時に進むわけだが、y方向の半径の値によっては、上下からの描画と中央からの描画のあいだに1ピクセル幅の隙間が残る。この隙間を埋めるため、ループを抜けた後の262~263行で念のためもう1回水平線分描画ルーチン呼び出ししている。そのまた副作用で、半径が(x方向, y方向ともに) 0の場合を特別扱いしなくならなくなった(152~157行)。今回、採用したアルゴリズムでは半径が0だとループを抜けた時点の座標値がでてにならないので、262~263行を実行するわけにはいかないのだ。

最後になったが、リスト5にリスト2~4の動作試験用プログラムを用意しておいた。リスト5は何かキーが押されるまで、ランダムな円(楕円)を描画し続ける。gcircleとgoval, gfillcircleでは引数の数が異なるので、goval, gfillcircleの動作試験時には48~50行を復活して対応してほしい。

リスト5
GCIRCLETEST.S

```

1: #      gcircleのテスト用プログラム
2:
3:         .include      doscall.mac
4:         .include      iocscall.mac
5: #
6:         .xref         gcircle
7: #      .xref         goval
8: #      .xref         gfillcircle
9: #      .xref         setcliprect
10: #
11: FPACK  macro    callno
12:         .dc.w      callno
13:         .endm
14: #
15: _RAND      equ     $fe0e
16: #
17:         .text
18:         .even
19: #
20: ent:
21:         lea.l       inisp, sp

```

```

22:
23:      clr.l    -(sp)          #スーパーバイザモードへ
24:      DOS      _SUPER        #
25:
26:                                #512x512, 65536
27:      move.l    #$0010_0005, -(sp)
28:      DOS      _CONCTRL      #画面初期化
29:
30: #      pea      window        #クリッピングウィンドウの設定
31: #      jsr      setcliprect    #
32: #      addq.l    #4, sp        #
33:
34:      lea.l     arg, a1        #a1 = 引数受け渡し領域
35: loop: movea.l  a1, a0          #引数をセット
36:      FPACK     _RAND          # 中心x
37:      lsr.w     #6, d0         #
38:      move.w    d0, (a0) +     #
39:                                #
40:      FPACK     _RAND          # 中心y
41:      lsr.w     #6, d0         #
42:      move.w    d0, (a0) +     #
43:                                #
44:      FPACK     _RAND          # 半径 (x方向半径)
45:      lsr.w     #7, d0         #
46:      move.w    d0, (a0) +     #
47:                                #
48: #      FPACK     _RAND          # y方向半径
49: #      lsr.w     #7, d0         #
50: #      move.w    d0, (a0) +     #
51:                                #
52:      FPACK     _RAND          # 描画色
53:      move.w    d0, (a0) +     #
54:
55:      pea.l     (a1)           #描画
56:      jsr      gcircle         #
57: #      jsr      goval          #
58: #      jsr      gfillcircle    #
59:      addq.l    #4, sp        #
60:
61:      DOS      _KEYSNS        #キーが押されるまで
62:      tst.l     d0            # 繰り返す
63:      beq      loop          #
64:
65:      DOS      _INKEY          #スペースが押されたら
66:      cmpi.b    #$20, d0      # 一時停止する
67:      bne      done          #
68: pause: DOS     _INKEY        #
69:      cmpi.b    #$20, d0      #
70:      beq      loop          #
71:
72: done: move.w    #-1, -(sp)    #キーを読み捨てる
73:      DOS      _KFLUSH        #
74:
75:      move.l    #$0010_0000, -(sp) #width 96
76:      DOS      _CONCTRL      #画面再初期化
77:      DOS      _EXIT
78: #
79: .data
80: .even
81: #
82: arg:                                #gcircle, govalの引数
83: .dc.w         0, 0, 0, 0, 0
84: #
85: window:                                #クリッピングウィンドウ
86: .dc.w         32, 32, 255-32, 255-32
87: #
88: .stack
89: .even
90: #
91: .ds.l         4096
92: inisp:
93: .end          ent

```

画像に変化を与える処理

画像に変化を与える処理



この章のテーマは“色”である。矩形領域中の各点の色をある規則によって置き換える、特殊効果系のサブルーチンを作成してみる。

色コード

最初にX68000における色の扱いをおさらいしておこう。とくに、カラーコードとパレットコードの意味は明確にしておきたい。

カラーコード

カラーコードは色そのものにつけられた番号であり、色とカラーコードは1対1に対応する。X68000のカラーコードはRGB（赤、緑、青の光の3原色）ごとの輝度各5ビット+RGB共通の輝度1ビットからなる計16ビットで表現される（図1）。RGBそれぞれにつき、その色成分をどのくらい含むかが0～31の32段階で表され、組みあわせて $32 \times 32 \times 32 = 32768$ 色、これに輝度ビットが0か1かが加わって65536色となる。輝度ビットはRGB0.5分の重みを持ち、このビットを立てることでRGBそれぞれを同時に0.5だけ明るくすることができる。これは、原色にかざれば64階調の表現が可能なることを意味する。もっとも、複雑になるのを避けるためか、現実のプログラムでは輝度ビットを無視した32768色のみを扱うことが多い。この章で作成するプログラムもその例に漏れない。

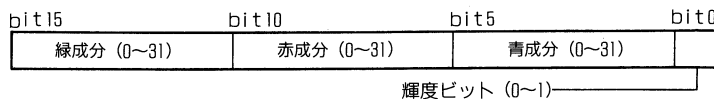


図1
X68000のカラーコード

ここでリスト1を見てもらおう。Chapter 2で用意したマクロ定義ファイルGMACRO.Hのうち、カラーコード関係の部分抜き出してある。ここでは、カラーコードをRGBに分解するマクロDERGBと、RGBごとの色成分からカラーコードを得るマクロRGBを定義している。カラーコード $\leftrightarrow$ RGBの分解・構成はこの章のプログラムではひんぱんに登場する操作なので、図1と見くらべて動作をしっかりとつかんでおいてほしい。ビットシフト、ビットマスク、ビット列のはめ込みといったビット操作のオンパレードだから、このあたりに不安のある人は注釈を参考にデータの変化を追ってみるとよいだろう。

リスト1
GMACRO.H
(一部抜粋)

```

59: DERGB macro COL, _B, _R, _G
60: move.w #RGBMAX, _B      * B = 0000000000011111
61: move.w _B, _R            * R = 0000000000011111
62: lsr.w #1, COL            *COL = 0GGGGRRRRRBBBBB
63: and.w COL, _B            * B = 00000000000BBBBB
64: lsr.w #5, COL            *COL = 00000GGGGRRRRR
65: and.w COL, _R            * R = 00000000000RRRRR
66: lsr.w #5, COL            *COL = 0000000000GGGGG
67: move.w COL, _G          * G = 0000000000GGGGG
68: .endm
69: *
70: RGB macro B, R, G, COL
71: move.b G, COL            *COL = xxxxxxxx00GGGGG
72: lsl.w #5, COL            *COL = xx00GGGGG0000
73: or.b R, COL              *COL = 00000GGGGRRRRR
74: lsl.w #5, COL            *COL = 0GGGGRRRRR0000
75: or.b B, COL              *COL = 0GGGGRRRRRBBBBB
76: add.w COL, COL           *COL = GGGGGRRRRBBBBB0
77: .endm

```

パレットコード

色につけられた絶対的な番号であるカラーコードに対して、パレットコードのほうは、試みるなら画材のパレット（一枚板のじゃなくて小学校の図工の時間に使ったような枠がたくさんあるやつ）の枠についた番号であり、特定のパレットコードがどの色にあたるかは固定されていない。パレットの設定（枠に置いた絵の具の色）によってどうにでも変わる。ただし、画面をIOCSなどによって初期化したときにはあたかもパレットコードの各ビットが、

GGGGGGRRRRRBBBBBI (65536色モード時)

GRRRRBBB (256色モード時)

GRBI (16色モード時)

のような意味をもつかのようにパレットが設定される（ただし、16色モードは若干変則的）。とくに65536色モードにおける初期状態ではパレットコード＝カラーコードとなる。

G-RAMに書き込むデータはパレットコードであり、ディスプレイへの表示の際にはCRTC (CRT Controller: 画面制御を行うLSI) とビデオコントローラがブラウン管の走査とタイミングをあわせてG-RAMの内容を読み出したうえでパレットの設定を参照し、カラーコードに変換してくれている。こういう仕組みだから、パレットを変更すると瞬時に画面の色も変わるのだ。

パレットの設定はビデオコントローラの内部レジスタを書き換えることで行う。ビデオコントローラのパレットレジスタはスーパーバイザ空間のE82000_H番地以降256ワードにマッピング（割りつけ）されており、68000からはこのメモリ領域を窓口としてパレットレジスタを読み書きする¹⁾。要するに、E82000_H～E821FF_Hのメモリ領域の適当なアドレスにカラーコードを書き込むことでパレットが設定できるわけだ。

■ 1) 一般にはCRTCのような（プロセッサから見ての）外部デバイスにはI/O空間に割りつけられるものだが、68000はモトローラの伝統でI/O空間というものを持たず、I/Oをメモリにマッピングする“メモリマップドI/O（memory mapped I/O）”が採用されている（英語の発音にしたがうならメモリマップトと表記すべきだが、ここでは“日本古来”の伝統によりメモリマップドと濁った表記を採用した）。

16色モードや256色モードではE82000_H番地からの1ワードがパレットコード0、E82002_H番地からがパレットコード1…というようにアドレスとパレットコードは素直に対応している。

アドレス = E82000_H + パレットコード × 2

だ。ところが、65536色モードではパレットレジスタの数が全然足りないので、かなり変則的なかたちでパレットが実現されている。65536色モードでパレットを変更する場面はあまりないとは思いますが、かんたんにまとめておこう。

4桁の16進数mmnn_Hで表されるパレットコードには、

$$H = E82002_H + (mm_H \& FE_H) \times 2 + (mm_H \& 1)$$

$$L = E82000_H + (nn_H \& FE_H) \times 2 + (nn_H \& 1)$$

の2つのアドレスが対応し、アドレスHで指定される1バイトにカラーコードの上位バイト、アドレスLにカラーコードの下位バイトが格納される。明らかにパレットコードmmXX_Hで表される256色についてアドレスHは共通となり、パレットコードXXnn_HとアドレスLについても同様の256対1の対応が見られる。このため、パレットコードmmnn_Hのパレット設定を変更すると、パレットコードmmXX_Hで表される256色とXXnn_Hで表される256色の計511色（1色はダブリ）の設定も同時に変わることになる。この様子は、

```
int x,y
screen 1,3,1,1
for y=0 to 255
  for x=0 to 255
    pset (x,y,(y shl 8)+x)
  next
next
```

のようなX-BASICプログラムで画面に65536色を表示してからpalet () 関数を使って適当にパレットを変更してみるとわかると思う。

パレット設定のプログラム例として、パレットの初期化ルーチンを作成してみた。リスト2

のginitpalet_lが65536色モード用、リスト3のginitpalet_mが256色モード用、リスト4のginitpalet_sが16色モード用となっており、それぞれ、色モードに応じたシステム規定値にパレットを初期化する。ginitpalet_lは少々妙なつくりになっているが、上で触れた65536色モード時の変則的なパレットの仕様と見くらべてもらえれば、これで正しく初期化できることが確認できるだろう。それでも納得いかないようなら、COMMAND.X上で、

A>SCREEN 1,3,1

を実行し、画面を65536色モードにしたうえで、DB.XでE82000_H番地以降のパレットレジスタ内容を覗いてみるとよい。

256色モード用のginitpalet_m (リスト3) は、50～56行で青8階調、赤8階調、緑4階調のグラデーションをなすカラーコード列を用意しておき、28～45行の3重ループでこれらを組みあわせて規定のカラーコードを得、順次設定している。50～56行のカラーコード列定義には8～18行のマクロを使っている。16色モード用のginitpalet_s (リスト4) はもっと露骨に、29～44行で16色分のデータをあらかじめ用意しておき、まとめてパレットレジスタに書き込んでいる。

リスト2
GINIPALL.S

```

1: *      パレットを初期化する (65536色モード時)
2:
3:      .include      doscall.mac
4:      .include      gconst.h
5: *
6:      .xdef      ginitpalet_l
7: *
8:      .text
9:      .even
10: *
11: ginitpalet_l:
12:      movem.l d0-d2/a0, -(sp)
13:
14:      lea.l      GPALET, a0
15:      move.l     #$0001_0001, d0
16:      move.l     #$0202_0202, d1
17:
18:      moveq.l    #512/4-1, d2
19: loop:  move.l    d0, (a0)+
20:      add.l      d1, d0
21:      dbra      d2, loop
22:
23:      movem.l    (sp)+, d0-d2/a0
24:      rts
25:
26:      .end

```

リスト3
GINIPALM.S

```

1: *      パレットを初期化する (256色モード時)
2:
3:      .include      doscall.mac
4:      .include      gconst.h
5: *
6:      .xdef      ginitpalet_m
7: *
8: BDAT4  macro      B1, B2, B3, B4
9:      .dc.w        B1*2, B2*2, B3*2, B4*2
10:      .endm

```

```

11: #
12: RDATA macro R1, R2, R3, R4
13: .dc.w R1*64, R2*64, R3*64, R4*64
14: .endm
15: #
16: GDATA macro G1, G2, G3, G4
17: .dc.w G1*2048, G2*2048, G3*2048, G4*2048
18: .endm
19: #
20: .text
21: .even
22: #
23: ginitpalet_m:
24: movem.l d0-d5/a0-a3, -(sp)
25:
26: lea.l GPALET, a0
27:
28: lea.l gdata(pc), a3
29: moveq.l #4-l, d3
30: gloop: move.w (a3)+, d5
31:
32: lea.l rdata(pc), a2
33: move.w #8-l, d2
34: rloop: move.w (a2)+, d4
35: or.w d5, d4
36:
37: lea.l bdata(pc), a1
38: moveq.l #8-l, d1
39: bloop: move.w (a1)+, d0
40: or.w d4, d0
41: move.w d0, (a0)+
42:
43: dbra d1, bloop
44: dbra d2, rloop
45: dbra d3, gloop
46:
47: movem.l (sp)+, d0-d5/a0-a3
48: rts
49: #
50: bdata: BDATA 00, 04, 09, 13
51: BDATA 18, 22, 27, 31
52: #
53: rdata: RDATA 00, 04, 09, 13
54: RDATA 18, 22, 27, 31
55: #
56: gdata: GDATA 00, 10, 21, 31
57:
58: .end

```

リスト4
GINIPALS.S

```

1: # バレットを初期化する (16色モード時)
2:
3: .include doscall.mac
4: .include gconst.h
5: #
6: .xdef ginitpalet_s
7: #
8: RGBDAT macro B, R, G
9: .dc.w B*2+R*64+G*2048
10: .endm
11: #
12: .text
13: .even
14: #
15: ginitpalet_s
16: movem.l d0/a0-a1, -(sp)
17:
18: lea.l GPALET, a0
19: lea.l paletdata_s(pc), a1
20:

```

```

21:      moveq.l #16/2-1,d0
22: loop: move.l (a1)+,(a0)+
23:      dbra    d0,loop
24:
25:      movem.l (sp)+,d0/a0-a1
26:      rts
27: *
28: paletdata_s:
29:      RGBDAT  00,00,00
30:      RGBDAT  10,10,10
31:      RGBDAT  16,00,00
32:      RGBDAT  31,00,00
33:      RGBDAT  00,16,00
34:      RGBDAT  00,31,00
35:      RGBDAT  16,16,00
36:      RGBDAT  31,31,00
37:      RGBDAT  00,00,16
38:      RGBDAT  00,00,31
39:      RGBDAT  16,00,16
40:      RGBDAT  31,00,31
41:      RGBDAT  00,16,16
42:      RGBDAT  00,31,31
43:      RGBDAT  21,21,21
44:      RGBDAT  31,31,31
45:
46:      .end

```

HSVによる色指定

特定の色を指定する方法はRGB表現以外にもたくさんある。X68000では、それらのうちHSVモデルが標準でサポートされており、IOCS HSVTORGBによりHSVからカラーコードを求めることができる。HSVでは、色を色相 (hue)、飽和度 (saturationの程度)、明るさ (value) の3値で表現する。この3つの値は図2のような色立体上の位置を指定するものと考えられる。垂直軸は明るさVを表し、上にいくほど明るい。色相Hはその軸を中心とする回転角度で表され、0°の位置に赤、以下60°ごとに、黄、緑、シアン、青、マゼンタと並び、1

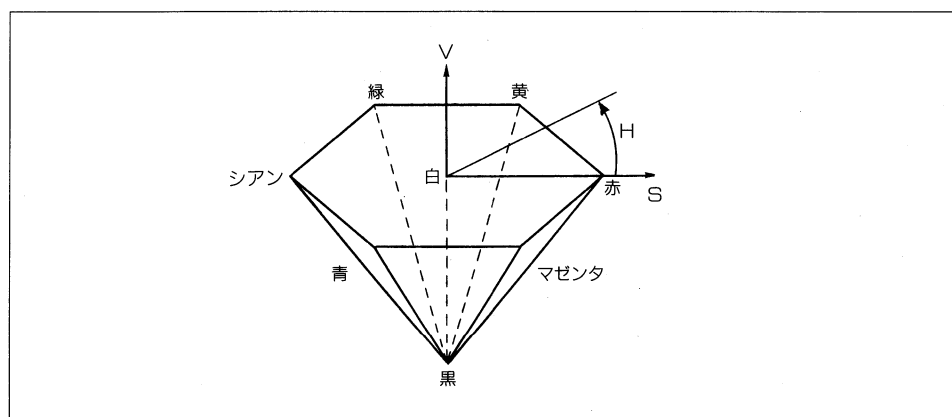


図2
HSV色立体

周して赤に戻る。また、飽和度Sは軸からの距離で表される。軸から離れるにつれ色は鮮やかになり、最外周で原色、逆に内側に近づくにつれ彩度を失っていき、中心で無彩色（黒～灰色～白）になる。IOCS HSVTORGBではS、Vは0～31の32段階で表し、また、Hは360°を192に対応させた値で指定する。

参考までにリスト5にHSV↔RGBの相互変換を行うX-BASICプログラムを示しておく。HSVからRGBへの変換を行う関数hsvtorgbは、かりに原色のつもりで求めた色を、飽和度におうじて白に近づけ、最後に明るさをかけている。真面目に四捨五入するようにしたお陰か、X-BASICに用意されているhsv()関数（ひいてはIOCS HSVTORGB）よりも精度は高いようだ。また、引数のHは0～191の範囲に収まっていなくてもプログラム側で適切な値に変換する。この点でも、hsv()関数より使いやすいかもしれない。

リスト5
HSV.BAS

```

10 str ss
20 int hsvbuf(2)
30 int h,s,v,c0,c1,c2
40 screen 1,3,1,1
50 ss=time$
60 ss=right$(ss,2)+mid$(ss,4,2)+left$(ss,2)
70 randomize (atoi(ss) mod 32768)
80 while (1)
90   h=192*rnd()
100  s=31*rnd()
110  v=31*rnd()
120  c0=hsv(h,s,v)
130  c1=hsvtorgb(h,s,v)
140  rgbtohsv(c1)
150  c2=hsvtorgb(hsvbuf(0),hsvbuf(1),hsvbuf(2))
160  fill(0,50,140,199,c0)
170  fill(150,50,290,199,c1)
180  fill(300,50,440,199,c2)
190  ss=inkey$
200 endwhile
210 end
220 /*
230 func hsvtorgb(h,s,v)
240   int c,i,v1,v2,v3,t
250   /*
260   if s=0 then {
270     c = rgb(v,v,v)
280   } else {
290     i=h/32 /*
300     t=h mod 32
310     v1=(v*(31*32-s*32)+31*16)/(31*32)
320     v2=(v*(31*32-s*t)+31*16)/(31*32)
330     v3=(v*(31*32-s*(32-t))+31*16)/(31*32)
340     switch (i)
350       case 0: c=rgb(v,v3,v1): break
360       case 1: c=rgb(v2,v,v1): break
370       case 2: c=rgb(v1,v,v3): break
380       case 3: c=rgb(v1,v2,v): break
390       case 4: c=rgb(v3,v1,v): break
400       case 5: c=rgb(v,v1,v2): break
410     endswitch
420   }
430   return (c)
440 }
450 /*
460 func rgbtohsv(c)
470   int h,s,v,r,g,b,t,f
480   /*
490   b = (c/2) and 31
500   r = (c/32/2) and 31

```

```

510  g = (c/32/32/2) and 31
520  v = b :f = 0
530  if (r > v) then v = r :f = 1
540  if (g > v) then v = g :f = 2
550  t = b
560  if (r < t) then t = r
570  if (g < t) then t = g
580  if (v = 0) then {
590      s = 0
600  } else {
610      s = (v-t)*31/v
620  }
630  if (s = 0) then {
640      h = -1
650  } else {
660      t = v-t
670      bb = v-b
680      rr = v-r
690      gg = v-g
700      switch ( f )
710          case 0: h = (32*(gg-rr))/t+4*32: break
720          case 1: h = (32*(bb-gg))/t+0*32: break
730          case 2: h = (32*(rr-bb))/t+2*32: break
740      endswitch
760      if (h < 0) then h = h + 192
770  }
780  hsvbuf(0) = h
790  hsvbuf(1) = s
800  hsvbuf(2) = v
810 endfunc

```

RGBからHSVへの変換関数rgbtohsvでは、RGB中最大の値をとるものから明るさと大雑把な色相を求め、2番目に大きなものとの組みあわせで最終的な色相を決め、もっとも小さなものから飽和度を決めている。飽和度が0のときは無彩色であり、色相が定義できないのだが、このプログラムではH=-1を返すようにした。なお、この関数を利用するうえでは、演算誤差などのため、HSV→RGB→HSVと変換を繰り返してもかならずしも元に戻るわけではないことに注意してほしい。

リスト5をアセンブリ言語で書き直すとリスト6,7となる。基本的な仕様はX-BASIC版と変わらない。引数や戻り値の受け渡しはd0レジスタで行う。HSVの指定方法としては、IOCS HSVTORGB同様、上位ワードにH、下位ワードの上位バイトにS、最下位バイトにVを格納するものとした。

リスト6
HSVTORGB.S

```

1: *      hsvからrgbへの変換
2:
3:      .include      gmacro.h
4: *
5:      .xdef      hsvtorgb
6: *
7:      .text
8:      .even
9: *
10: *      d0.l = hhhhhhhhhhhhhh_000sssss_000vvvvv
11: *      → d0.w = gggggrrrrrbbbbbb0
12: *
13: hsvtorgb:
14:      movem.l d1-d5, -(sp)

```

```

15:
16:      moveq.l #0, d2
17:      move.b d0, d2      #d2 = v
18:
19:      moveq.l #0, d3
20:      lsr.w #8, d0
21:      move.b d0, d3      #d3 = s
22:      bne      sskip
23:
24:      *無彩色の場合
25:      move.w d2, d1      #r = g = b = v
26:      move.w d2, d3      #
27:      bra      encode
28:
29:      *有彩色の場合
30: sskip: swap.w d0      #d0 = h
31:      ext.l d0      #hを0～191に正規化する
32:      divs.w #192, d0
33:      swap.w d0
34:      tst.w d0
35:      bpl      hskip
36:      addi.w #192, d0
37:
38: hskip: moveq.l #31, d4
39:      and.w d0, d4      #d4 = t = h%32
40:      mulu.w d3, d4      #d4 = s*t
41:      lsl.w #5, d3      #d3 = s*32
42:      move.w d4, d5      #d5 = s*t
43:      sub.w d3, d5      #d5 = -s*(32-t)
44:      neg.w d4      #d4 = -s*t
45:      neg.w d3      #d3 = -s*32
46:
47:      move.w #31*32, d1
48:      add.w d1, d3      #d3 = 31*32-s*32
49:      add.w d1, d4      #d4 = 31*32-s*t
50:      add.w d1, d5      #d5 = 31*32-s*(32-t)
51:      mulu.w d2, d3      #d3 = v*(31*32-s*32)
52:      mulu.w d2, d4      #d4 = v*(31*32-s*t)
53:      mulu.w d2, d5      #d5 = v*(31*32-s*(32-t))
54:      move.w #31*16, d1
55:      add.w d1, d3      #d3 = v*(31*32-s*32)+31*16
56:      add.w d1, d4      #d4 = v*(31*32-s*t)+31*16
57:      add.w d1, d5      #d5 = v*(31*32-s*(32-t))+31*16
58:      add.w d1, d1
59:      divu.w d1, d3      #d3 = (v*(31*32-s*32)+31*16)/(31*32)
60:      divu.w d1, d4      #d4 = (v*(31*32-s*t)+31*16)/(31*32)
61:      divu.w d1, d5      #d5 = (v*(31*32-s*(32-t))+31*16)/(31*32)
62:
63:      lsr.w #5, d0
64:      add.w d0, d0
65:      move.w jmptbl(pc, d0), d0
66:      jsr      jmptbl(pc, d0)
67:
68: encode: RGB      d1, d2, d3, d0      #RGBからカラーコードに構成する
69:
70:      movem.l (sp)+, d1-d5
71:      rts
72: *
73: jmptbl:
74: x      =      jmptbl
75:      .dc.w RtoY-x
76:      .dc.w YtoG-x
77:      .dc.w GtoC-x
78:      .dc.w CtoB-x
79:      .dc.w BtoM-x
80:      .dc.w MtoR-x
81: *
82: RtoY: move.w d3, d1      #赤～黄
83:      move.w d5, d3
84:      rts
85: YtoG: move.w d3, d1      #黄～緑
86:      move.w d2, d3
87:      move.w d4, d2
88:      rts

```



```

89:  GtoC:   move. w  d5, d1          #緑～シアン
90:         exg. l   d2, d3
91:         rts
92:  CtoB:   move. w  d2, d1          #シアン～青
93:         move. w  d3, d2
94:         move. w  d4, d3
95:         rts
96:  BtoM:   move. w  d2, d1          #青～マゼンタ
97:         move. w  d5, d2
98:         rts
99:  MtoR:   move. w  d4, d1          #マゼンタ～赤
100:         rts
101:
102:         .end

```

リスト7
RGBTOHSV.S

```

1: # rgbからhsvへの変換
2:
3: .include gmacro.h
4: #
5: .xdef rgbtohsv
6: #
7: .text
8: .even
9: #
10: # d0.w = ggggrrrrrrbbbbbb0
11: # → d0.l = hhhhhhhhhhhhhh_000sssss_000vvvvv
12: #
13: rgbtohsv:
14:     movem.l d1-d7, -(sp)
15:
16:     DERGB    d0, d3, d4, d5
17:
18:     moveq.l #0, d7          #仮にbが最大と仮定
19:     move.w d3, d2           #
20:     cmp.w d4, d2            #
21:     bge skip0               #
22:     moveq.l #2, d7          #rが最大と仮定
23:     move.w d4, d2           #
24:     skip0: cmp.w d5, d2      #
25:     bge skip1               #
26:     moveq.l #4, d7          #gが最大
27:     move.w d5, d2           #
28:
29:     skip1: move.w d3, d6     #
30:     MIN     d4, d6           #
31:     MIN     d5, d6           #d6 = t
32:     neg.w d6                 #
33:     add.w d2, d6             #d6 = v-t
34:
35:     # d2 = v = max(r, g, b)
36:     # d6 = v - t = max(r, g, b) - min(r, g, b)
37:     # d7 = 0 ... v = b
38:     #       = 2 ... v = r
39:     #       = 4 ... v = g
40:
41:     moveq.l #0, d1           #s = 0
42:     tst.w d2                 #v=0?
43:     beq sskip                #
44:     move.w d6, d1            #d1 = v-t
45:     mulu.w #31*2, d1         #d1 = 31*2*(v-t)
46:     add.w d2, d1              #d1 = 31*2*(v-t)+v
47:     move.w d2, d0            #d0 = v
48:     add.w d0, d0             #d0 = 2*v
49:     divu.w d0, d1            #d1 = s = (31*2*(v-t)+v)/(2*v)
50:                               #       = (31*(v-t)+0.5)/v
51:
52:     sskip: moveq.l #-1, d0    #h = -1 (無彩色の色相は不定)
53:     tst.w d1                 #s=0?
54:     beq hskip

```

```

55:      neg. w   d3
56:      add. w   d2, d3      *b = v-b
57:      neg. w   d4
58:      add. w   d2, d4      *r = v-r
59:      neg. w   d5
60:      add. w   d2, d5      *g = v-g
61:
62:      move. w   jmptbl (pc, d7), d7
63:      jsr       jmptbl (pc, d7)
64:
65:      asl. w     #6, d7      *d7 = 32*2*(xx-yy)
66:      add. w     d6, d7      *d7 = 32*2*(xx-yy) + (v-t)
67:      ext. l     d7
68:      add. w     d6, d6      *d6 = 2*(v-t)
69:      divs. w    d6, d7      *d7 = (32*2*(xx-yy) + (v-t) / (2*(v-t))
70:                                * = (32*(xx-yy) + 0.5) / (v-t)
71:      add. w     d7, d0      *d0 = h
72:      bpl       hskip
73:      addi. w    #192, d0    *h ≥ 0を保証
74:
75: hskip: swap. w   d0        *hsvをIOCS形式に構成
76:      move. b    d1, d0      *
77:      lsl. w     #8, d0      *
78:      move. b    d2, d0      *
79:
80:      movem. l   (sp)+, d1-d7
81:      rts
82: *
83: jmptbl:
84: x      =      jmptbl
85:      .dc. w     MtoC-x
86:      .dc. w     YtoM-x
87:      .dc. w     CtoY-x
88: *
89: MtoC:  move. w   #4*32, d0
90:      move. w     d5, d7
91:      sub. w      d4, d7      *d7 = gg-rr
92:      rts
93: *
94: YtoM:  moveq. l   #0, d0
95:      move. w      d3, d7
96:      sub. w       d5, d7      *d7 = bb-gg
97:      rts
98: *
99: CtoY:  move. w   #2*32, d0
100:      move. w      d4, d7
101:      sub. w       d3, d7      *d7 = rr-bb
102:      rts
103:
104:      .end

```

*

では、画像エフェクト用のサブルーチンを実際に作成していくとしよう。なお、あらかじめ断っておくが、以下のサブルーチンは基本的に65536色モード専用、かつ、パレットコード＝カラーコードを仮定する。また、65536色モード用とはいっても、実際には輝度ビットを無視した32768色の画像を扱う。

色の反転

まず、色の反転から始める。正しい表現ではないかもしれないが、ここでは色の反転＝補色

への置換と考える。補色 (complementary color) とは、元の色と補いあうことで無彩色となるような色で、加法混色²⁾における補色と減法混色³⁾における補色があるが一般には前者のことを指す。白から引いた残りの色といってもよい。補色を求める操作は、カラーコードレベルではカラーコードの各ビットを反転する (1の補数を求める) 処理に置き換えられる。ボックスフィルルーチンよろしく縦横の2重ループを組み、各ピクセルのカラーコードをnot命令で次々にビット反転していけば色反転が実現できる。notを使う代わりに、eor命令でFFFF_HとXOR (exclusive or: 排他的論理和) をとっても同じことだ。リスト8に実例を示す。リスト8は本体ではなく、実際の処理はリスト9のgxorcolorで行う。gxorcolorは、要はXORモードのボックスフィルルーチンで、矩形領域の各点の色を任意の16ビット数とXORをとった色に置き換えていく。リスト8のgnegateは、描画色としてFFFF_Hを指定してgxorcolorを呼び出し、色の反転を実現している。gxorcolorへ渡す引数はスタック上に構成するようになっている。

- 2) 光は混ぜあわせることによって明るくなり、白に近づいていく。だから加法混色。
- 3) 絵の具は混ぜあわせることによって (吸収される光が多くなるため) 暗くなり、黒に近づいていく。だから減法混色。減法混色系の3原色シアン・マゼンタ・黄色は加法混色系の3原色である赤・青・緑の補色となっている。

リスト8
GNEGATE.S

```

1: *      色反転 (補色への置換)
2:
3:
4:      .xdef    gnegate
5:      .xref    gxorcolor
6: *
7:      .offset 0      *gnegateの引数構造
8: *
9: X0:      .ds.w    1      *矩形座標
10: Y0:      .ds.w    1      *
11: X1:      .ds.w    1      *
12: Y1:      .ds.w    1      *
13: *
14:      .text
15:      .even
16: *
17: gnegate:
18: ARGPTR =      8+4
19:      move.l  a1, -(sp)
20:      link    a6, #0
21:      movea.l ARGPTR(a6), a1      *a1 = 引数列
22:      move.w  #$ffff, -(sp)      *描画色 = ffffh
23:      move.l  4(a1), -(sp)      *(x1, y1)
24:      move.l  (a1), -(sp)      *(x0, y0)
25:      pea.l   (sp)      *gxorcolorへの引数列
26:      jsr     gxorcolor
27:
28:      unlk    a6
29:      movea.l (sp)+, a1
30:      rts
31:
32:      .end

```

リスト9
GXORCOLOR.S

```

1: #      xorモードのボックスフィル
2:
3:      .include      gconst.h
4: #
5:      .xdef      gxorcolor
6:      .xref      gramadr
7:      .xref      gcliprect
8: #
9:      .offset 0      #gxorcolorの引数構造
10: #
11: X0:      .ds.w      1      #描画先座標
12: Y0:      .ds.w      1      #
13: X1:      .ds.w      1      #
14: Y1:      .ds.w      1      #
15: COL:     .ds.w      1      #描画色
16: #
17:      .text
18:      .even
19: #
20: gxorcolor:
21: ARGPTR =      8
22:      link      a6, #0
23:      movem.l   d0-d4/a0-a1, -(sp)
24:
25:      movea.l   ARGPTR(a6), a1      #a1 = 引数列
26:      movem.w   (a1), d0-d3      #d0～d3に座標を取り出す
27:
28:      jsr      gcliprect      #クリッピングする
29:      bmi      done      #N=1なら描画の必要なし
30:
31:      jsr      gramadr      #G-RAM上のアドレスを得る
32:
33:      sub.w     d0, d2      #d2 = 横ピクセル数-1
34:      sub.w     d1, d3      #d3 = 縦ピクセル数-1
35:
36:      move.w    COL(a1), d0      #d0 = 描画色
37:
38:      move.w    #GNBYTE-2, d1      #d1 = ライン間のアドレスの差
39:      sub.w     d2, d1      # (右端から下のラインの左端まで)
40:      sub.w     d3, d1      #
41:
42: loop1:     move.w    d2, d4      #d4 = 横ピクセル数-1
43: loop2:     eor.w     d0, (a0)+      #
44:           dbra      d4, loop2      #横幅分繰り返す
45:
46:           adda.w     d1, a0      #すぐ下のラインへ
47:           dbra      d3, loop1      #高さ分繰り返す
48:
49: done:      movem.l   (sp)+, d0-d4/a0-a1
50:           unlk      a6
51:           rts
52:
53:      .end

```

リスト10に動作試験用のプログラムも用意しておいた。あらかじめ65536色の画像を画面に読み込んでおき起動すると、全画面の色を反転する。反転する範囲は左上隅と右下隅の座標のかたちでコマンドライン引数として与えることもできる。引数は通常の10進指定のほかに、頭に“\$”をつけた16進数での指定も許す。リスト10は動作試験用としては必要以上に真面目につくってあるので、そのまま実用プログラムにしてもよいだろう。なお、適当な画像読み込みプログラムが手元にないようなら、リスト11の簡易GL3ファイルローダもあわせて利用してほしい。

リスト10
FXMAIN.S

```

1: *      エフェクト系サブルーチン群のメインルーチン
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: *
6:      .xref      gnegate
7: *
8: NARG      equ      4          *引数の個数
9: *
10: FPACK      macro      callno
11:             .dc.w      callno
12:             endm
13: *
14: _STOL      equ      $fe10      *10進文字列→数値変換
15: _STOH      equ      $fe12      *16進文字列→数値変換
16: _RAND      equ      $fe0e      *乱数 (0~32767)
17: *
18: CR          equ      13
19: LF          equ      10
20: TAB        equ      9
21: SPACE      equ      32
22: *
23: STDERR      equ      2
24: *
25:             .text
26:             .even
27: *
28: ent:
29:             lea.l      inisp, sp
30:
31:             bsr      getarg      *引数を取得する
32:
33:             moveq.l   #-1, d1      *画面は初期化されているか?
34:             lOCS      _APAGE      *
35:             tst.b      d0          *
36:             bmi      error      *未初期化だった
37:
38:             clr.l      -(sp)        *スーパーバイザモードへ移行する
39:             DOS      _SUPER      *
40:
41:             pea.l      argbuf      *サブルーチンを呼び出す
42:             jsr      gnegate      *
43: *             addq.l      #4, sp      *
44:
45:             DOS      _EXIT
46:
47: *
48: *      NARG個の数値をバッファにセットする
49: *
50: getarg:
51:             tst.b      (a2)+        *空文字列なら
52:             beq      noarg      *デフォルト値を採用
53:
54:             movea.l   a2, a0        *a0 = コマンドライン
55:             lea.l      argbuf, a1    *a1 = 引数格納バッファ
56:             moveq.l   #NARG-1, d1    *d1 = ループカウンタ
57: getpr0:      bsr      skipsp      *先行する空白を飛ばす
58:             move.b      (a0)+, d0    *文字列終端?
59:             beq      usage      *そうなら引数が足りない
60:             cmpi.b      #' ', d0      *
61:             beq      codarg      *
62:             cmpi.b      #'?', d0      *
63:             beq      colarg      *
64:             cmpi.b      #'$', d0      *
65:             beq      hexarg      *
66:             subq.l      #1, a0      *
67: decarg:      FPACK      _STOL      *10進
68:             bra      argchk      *
69: colarg:      FPACK      _RAND      *0~65534の乱数
70:             add.w      d0, d0      *
71:             bra      argset      *
72: codarg:      FPACK      _RAND      *0~511の乱数

```

```

73:      lsr.w  #6, d0      *
74:      bra   argset      *
75: hexarg: FPACK          *16進
76: argchk: bcs   usage    *うまく変換できなかった
77: argset: move.w d0, (a1)+ *引数を格納
78:      dbra  d1, getpr0   *必要なだけ繰り返す
79: noarg: rts
80: *
81: skpsp0: addq.l #1, a0
82: skipsp: cmpi.b #SPACE, (a0)
83:      beq   skpsp0
84:      cmpi.b #TAB, (a0)
85:      beq   skpsp0
86:      rts
87: *
88: usage: lea.l  usgmes, a0
89:      bra   error0
90: *
91: error: lea.l  errmes, a0
92: error0: move.w #STDERR, -(sp) *標準エラー出力へ
93:      pea.l  (a0)           *メッセージを出力する
94:      DOS    _FPUTS
95:      addq.l #6, sp
96:      move.w #1, -(sp)      *終了コード1を持って
97:      DOS    _EXIT2        *エラー終了
98: *
99:      .data
100:     .even
101: *
102: argbuf: .dc.w 0, 0, 511, 511 *デフォルト引数
103: *
104: usgmes: .dc.b '機能: 画像の色を反転する', CR, LF
105:      .dc.b '使用法: GNEGATE [x0 y0 x1 y1]', CR, LF, 0
106: errmes: .dc.b 'グラフィック画面が初期化されていません', CR, LF, 0
107: *
108:      .stack
109:     .even
110: *
111: mystack:
112:      .ds.l 4096           *スタック領域
113: inisp:
114:
115:      .end   ent

```

リストII
GLLOAD.S

```

1: *      簡易GL3ファイルローダ
2:
3:      .include doscall.mac
4:      .include iocscall.mac
5: *
6: CR equ 13
7: LF equ 10
8: *
9: ROPEN equ 0
10: STDERR equ 2
11: *
12: SIZE equ 512*1024
13: GRAMTOP equ $c00000
14: *
15: CSCREEN equ 16
16: DOS_GL3 equ 5
17: *
18: CHKGRAM equ 0
19: CHECK equ -1
20: BROKEN equ 3
21: *
22:      .text
23:     .even
24: *
25: ent:

```

```

26:      lea, l    inisp (pc), sp      *spを初期化する
27:
28:      moveq, l  #CHKGRAM, d1       *G-RAMは
29:      moveq, l  #CHECK, d2         * 使用可能か?
30:      lOCS      _TGUSEMD           *
31:      cmpi, b   #BROKEN, d0        *誰かが使いっぱなし?
32:      beq       gramok             * そうなら使える
33:      tst, b    d0                 *誰も使っていない?
34:      bne       rsrvd              * 誰かが使っていた
35:
36: gramok: tst, b    (a2) +           *ファイル名の指定はある?
37:      beq       usage              * なかった
38:
39:      move, w   #ROPEN, -(sp)       *ファイルを開く
40:      move, l   a2, -(sp)           *
41:      DOS       _OPEN               *
42:      addq, l   #6, sp              *
43:      move, w   d0, d3              *d3 = ファイルハンドル
44:      bmi       nfound              *
45:
46:      move, w   #DOS_GL3, -(sp)     *画面を512x512, 65536色に
47:      move, w   #CSCREEN, -(sp)     * 初期化
48:      DOS       _CONCTRL            *
49:      addq, l   #4, sp              *
50:
51:      moveq, l  #CHKGRAM, d1       *G-RAM内容は
52:      moveq, l  #BROKEN, d2        * 私が壊しました
53:      lOCS      _TGUSEMD           *
54: *
55:      move, l   #SIZE, -(sp)        *ファイルからG-RAMに
56:      move, l   #GRAMTOP, -(sp)     * 直接読み込む
57:      move, w   d3, -(sp)           *
58:      DOS       _READ               *
59:      lea, l    T0 (sp), sp         *
60:      tst, l    d0                  *
61:      bmi       reader              *
62:
63:      move, w   d3, -(sp)           *ファイルを閉じる
64:      DOS       _CLOSE              *
65:      addq, l   #2, sp              *
66:
67:      DOS       _EXIT               *正常終了
68:
69: *
70: *      使用法の表示&エラー終了
71: *
72: usage: lea, l    usgmes (pc), a0
73:      bra       error
74: rsrvd: lea, l    errms1 (pc), a0
75:      bra       error
76: nfound: lea, l   errms2 (pc), a0
77:      bra       error
78: reader: lea, l   errms3 (pc), a0
79: *
80: error: move, w   #STDERR, -(sp)     *標準エラー出力へ
81:      move, l   a0, -(sp)           * メッセージを
82:      DOS       _FPUTS              * 出力する
83:      addq, l   #6, sp              *
84: *
85:      move, w   #1, -(sp)           *終了コード1を持って
86:      DOS       _EXIT2              * エラー終了
87: *
88:      .data
89:      .even
90: *
91: usgmes: .dc, b   '機 能: GL3ファイルをロードする', CR, LF
92:      .dc, b   '  使用法: GLLOAD ファイル名, GL3', CR, LF, 0
93: errms1: .dc, b   '  GLLOAD: G-RAMは他のプログラムが使用中です', CR, LF, 0
94: errms2: .dc, b   '  GLLOAD: 指定の画像ファイルが見つかりません', CR, LF, 0
95: errms3: .dc, b   '  GLLOAD: 画像ファイルが読み込めません', CR, LF, 0
96: *
97:      .stack
98:      .even
99: *

```

```

100:      .ds.l    4096      *スタック領域
101: inisp:
102:
103:      .end    ent

```

ところで、たんに画面上で色を反転した結果を見ただけであれば、G-RAM内容を変えなくてもパレットを操作すれば事足りる。この方法なら、16色モードや256色モードでも色反転が可能だ。リスト12にパレットを補色に置き換えて再設定するサブルーチン例を示しておく。このサブルーチンはすべての色モードで利用できる。

リスト12
GNEGPALETS

```

1: *      パレットレベルで色反転する
2:
3:      .include      doscall.mac
4:      .include      gconst.h
5: *
6:      .xdef      gnegatepalet
7: *
8:      .text
9:      .even
10: *
11: gnegatepalet:
12:      movem.l d0/a0, -(sp)
13:
14:      lea.l      GPALET, a0      *パレットレジスタ
15:      moveq.l    #512/4-1, d0     * 512バイトを4バイト単位で
16: loop:      not.l      (a0)+      * ビット反転する
17:      dbra      d0, loop      *
18:
19:      movem.l    (sp)+, d0/a0
20:      rts
21:
22:      .end

```

色の量子化

つづいては色の量子化。RGB各32階調から16階調や8階調なんかに階調を落とす操作を考える。画像の質をなるべく損わないようにして色数を減らすのではなく、単純に微妙な色の変化を切り捨てる。当然、階調を落とすほど画像は粗くなっていく。色の量子化はカラーコードのレベルで考えると、RGBそれぞれの下位ビットをマスクする処理に相当する。ビットマスクといえばAND、ということはANDモードのボックスフィルルーチンがあればよい。このANDモードのボックスフィルルーチンに描画色として、

```

1111011110111100B  RGB16階調
11100111100111000B  RGB 8 階調
1100011000110000B  RGB 4 階調
1000010000100000B  RGB 2 階調

```

を与えれば色の量子化が実現できる。余談ながら、ANDモードのボックスフィルルーチンは、描画色に、


```
1111111111111110B
```

を指定することで輝度ビットだけをマスクしたり (65536色から32768色への変換),

```
0000000000111110B
```

```
0000011111000000B
```

```
1111100000000000B
```

によってそれぞれ青, 赤, 緑成分だけを残したりといった用途にも使える。

肝心のANDモードのボックスフィルルーチンは, リスト9をちょっと変更すればすぐ得られる。リスト13に変更点を示しておこう。ついでだから, リスト14にORモードのボックスフィルルーチンも用意してみた。

.....
リスト13
GANDCOLOR.S
(リスト9に対する変更部)

```
1: *      andモードのボックスフィル
5:      .xdef  gandcolor
20: gandcolor:
43: loop2:  and.w  d0, (a0)+      *
```

.....
リスト14
GORCOLOR.S
(リスト9に対する変更部)

```
1: *      orモードのボックスフィル
5:      .xdef  gorcolor
20: gorcolor:
43: loop2:  or.w   d0, (a0)+      *
```

動作試験にはリスト10を流用するのがよいだろう。呼び出すサブルーチン名と, 引数の個数, 引数のデフォルト値, ヘルプメッセージを修正すればそのまま使える。具体的には, 6行, 42行のサブルーチン名をgnegateからgandcolorに, 8行の引数の個数を4から5に, 102行のデフォルト引数列兼引数受け渡し用バッファ部分を,

```
.dc.w  0,0,511,511,$f7bc
```

に (第5引数は描画色で, そのデフォルト値, F7BC_Hは16階調変換用のマスクパターン), 104~105行のヘルプメッセージを適当に変更する。以下, この章のプログラムは同じ要領で動作試験 (あるいは実用プログラム化) が行える。

モノクロ32階調に

今度はモノクロ化だ。モノクロといっても白黒の2値化ではなく, グレイスケール (黒~灰色~白に段階的に変化するような色の並び) に変換することを考える。X68000では単色であ

れば輝度ビットを使って64階調を出すこともできるが、ここでは輝度ビットを無視した32色のみを使うことにしよう。アルゴリズムとしては、各点ごとに明るさ（仮にLとおく）を32階調で求め、 $R=G=B=L$ となるような色に置き換えていけばよい。肝心の明るさを求める部分だが、とりあえずはRGBのうち最大のものを使うことにする。プログラムはリスト15のようになった。

43行以下が1ピクセル分の色変換を行う部分だ。43～44行で色コードを取り出しRGBに分解してから、マクロMAXでまず青成分と赤成分の大きいほうをd5に求め（45行）、続いて緑成分とd5の大きいほうをd5に残す（46行）。結果的にRGBのうち最大のものがd5に得られた。これから $R=G=B=d5$ となる色を求めるわけだが、リスト15ではその手間を惜しんで、あらかじめ作成しておいた32色の色テーブルから適切な値をひいてくるという手を使っている（47～48行）。テーブル自体はほかにも使い道がありそうだから別ファイル（リスト16）にしておいた。ちなみに、このテーブルは“無敵の汎用ツール”X-BASICでかんたんなプログラムつくってファイルに落とし、整形したものだ。

リスト15
GMONOTONE.S

```

1: #          32階調モノクロ化
2:
3:          .include      gconst.h
4:          .include      gmacro.h
5: #
6:          .xdef      gmonotone
7:          .xref      gramadr
8:          .xref      gcliprect
9:          .xref      grayscale
10: #
11:          .offset 0      #gmonotoneの引数構造
12: #
13: X0:      .ds.w      1      #矩形座標
14: Y0:      .ds.w      1      #
15: X1:      .ds.w      1      #
16: Y1:      .ds.w      1      #
17: #
18:          .text
19:          .even
20: #
21: gmonotone:
22: ARGPTR   =          8
23:          link      a6, #0
24:          movem.l   d0-d7/a0-a1, -(sp)
25:
26:          movea.l   ARGPTR(a6), a1      #a1 = 引数列
27:          movem.w   (a1), d0-d3      #d0～d3に座標を取り出す
28:
29:          jsr      gcliprect      #クリッピングする
30:          bne      done      #Z=0なら描画の必要なし
31:
32:          jsr      gramadr      #G-RAM上のアドレスを得る
33:
34:          sub.w     d0, d2      #d2 = 横ピクセル数-1
35:          sub.w     d1, d3      #d3 = 縦ピクセル数-1
36:
37:          move.w    #GNBYTE-2, d1      #d1 = ライン間のアドレスの差
38:          sub.w     d2, d1      # (右端から下のラインの左端まで)
39:          sub.w     d2, d1      #
40:
41:          lea.l     grayscale, a1
42: loop1:    move.w    d2, d4      #d4 = 横ピクセル数-1
43: loop2:    move.w    (a0), d0      #カラーコードを取り出し

```

```

44:      DERGB    d0, d5, d6, d7      *RGBに分解する
45:      MAX      d6, d5              *R, G, Bの最大値を
46:      MAX      d7, d5              * d5に求め
47:      add.w     d5, d5              *rgb (d5, d5, d5) の色で
48:      move.w    0(a1, d5), (a0) +  * 点を打つ
49:      dbra      d4, loop2          *横幅分繰り返す
50:
51:      adda.w     d1, a0              *すぐ下のラインへ
52:      dbra      d3, loop1          *高さ分繰り返す
53:
54: done:  movem.l  (sp)+, d0-d7/a0-a1
55:      unlink    a6
56:      rts
57:
58:      .end

```

リスト16 GRAYSCALES

```

1: *      rgb (0, 0, 0) ~ rgb (31, 31, 31) の色テーブル
2:
3:      .xdef     grayscale
4: *
5:      .data
6:      .even
7: *
8: grayscale:
9:      .dc.w     $0000, $0842, $1084, $18c6
10:     .dc.w     $2108, $294a, $318c, $39ce
11:     .dc.w     $4210, $4a52, $5294, $5ad6
12:     .dc.w     $6318, $6b5a, $739c, $7bde
13:     .dc.w     $8420, $8c62, $94a4, $9ce6
14:     .dc.w     $a528, $ad6a, $b5ac, $bdee
15:     .dc.w     $c630, $ce72, $d6b4, $def6
16:     .dc.w     $e738, $ef7a, $f7bc, $fffe
17:
18:      .end

```

さて、リスト15ではとくに根拠もなくRGBのうち最大のものをその点の明るさだとみなした。妥当な線だとも思えるが、実際には何かと不都合がある。端的なところでは、もっとも明るい赤と青で塗り分けられた部分は真っ白に変換されてしまう。また、変換の前後で明るさのバランスが崩れ、ときには明暗が逆になったように見えることもある。これは人間の視覚系が純粋な光の強弱だけで明るさを判断しているのではないため、ということらしい。画像をモノクロに変換するとき、この人間の目の性質を考慮すればより自然な変換が行えるだろう。これには、カラーテレビ放送の仕組みが参考になる。

アメリカや日本などのテレビ放送では画像をYIQの3信号に分解して送る方式を採用している。大雑把にいうとYが輝度、Iが人間の目にとって重要な(色の違いに敏感な)肌色を含むオレンジからシアンにかけての色調、Qはそれ以外の色を表す。テレビの受信機側ではこれをRGBに再構成して画面に映している。3つの信号に分けるのなら最初からRGBに分解しておけばいいように思えるが、そうっていないのには白黒テレビとの互換性を保つ必要があったこと(白黒放送で使われていた電波の帯域にY信号を乗せれば白黒テレビでもカラー放送を受像できる)と、放送に使える電波の帯域に制限がある(人間の目にとって重要でない色は狭い帯域に押し込むことで帯域を節約できる)という2つの理由による。

ここで、Yは標準被視感度曲線なるものにしたがって、人間の主観的な明るさの感じ方にあ

わせて調整されている。ということは、画像の各点の色をRGBからYIQに変換し、Yの値に応じた灰色に置き換えるようにすれば、それらしいモノクロ画像が得られるだろう。タナボタ式に示してしまうと、RGBからYIQへの変換は次の式で行える。

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

いま必要なのはYだけだから、

$$Y = 0.299R + 0.587G + 0.114B$$

となる。

この式をマシン語で計算するにあたっては小数が入っているのがやっかいだ。しかし、どうせYは0~31の整数で求めるのであり、精度を追及することにあまり意味はない。そこで係数を適当にまるめたうえで両辺を10倍し、

$$10Y = 3R + 6G + B$$

として整数演算に帰着させる。右辺を計算した結果はYを10倍したもののだから、最後にこの値を10で割ればYが得られる。

リスト17が画像モノクロ化サブルーチンの新しい版だ(リスト15からの変更部分のみ)。リスト15の版にも使い道はありそうだから、サブルーチン名は変えておいた。Yの算出にはGMACRO.Hで定義されたマクロRGBtoYを利用しているので、ソース上での修正点はごくわずかだ。マクロRGBtoYの定義(リスト18に抜粋)中では例によって乗算を加算に置き換えて計算することで遅い乗算命令を使わずにすまし、さらには式を、

$$10Y = 3(R + 2G) + B$$

と変形して計算することで加算の回数も減らしている。

.....
リスト17
GMONO2.S
(リスト15に対する追加
・変更部)

```
1: *      32階調モノクロ化 (YIQのYを輝度を使う版)
6:      .xdef gmonotone2
21: gmonotone2:
43: loop2: move.w (a0), d0      *カラーコードを取り出し
44:      DERGB d0, d5, d6, d7    *RGBに分解する
45:      RGBtoY d5, d6, d7, d0    *Yをd0に求め
46:      *
47:      add.w d0, d0            *rgb (d0, d0, d0) の色で
48:      move.w 0(a1, d0), (a0) + * 点を打つ
49:      dbra d4, loop2          *横幅分繰り返す
50:
51:      adda.w d1, a0            *すぐ下のラインへ
52:      dbra d3, loop1          *高さ分繰り返す
53:
54: done:  movem.l (sp)+, d0-d7/a0-a1
55:      unlk a6
56:      rts
57:
58:      .end
```

リスト18
GMACRO.H
(抜粋)

```

79: RGBtoY macro B, R, G, Y
80:     add.w G, G          *G = 2G
81:     add.w R, G          *G = 2G+R
82:     move.w G, Y        *Y = 2G+R
83:     add.w Y, Y          *Y = 2 (2G+R)
84:     add.w G, Y          *Y = 3 (2G+R)
85:     add.w B, Y          *Y = 6G+3R+B
86:     ext.l Y
87:     divu.w #10, Y      *Y = (6G+3R+B)/10
88:     .endm

```

バリエーションとして、リスト19に任意の色調（青系統とかセピア調とか）で単色化できる版をつくってみた。色はHSVのHとSで指定する。リスト15やリスト17では色のテーブルをあらかじめ用意していたが、リスト19では与えられたHとSにおうじてその場でテーブルを作成するようになっている（40～47行）。

リスト19
GMONO2HSV.S

```

1: *      単色化（任意色相, Y10のYを輝度に使う版）
2:
3:     .include      gconst.h
4:     .include      gmacro.h
5: *
6:     .xdef         gmonotone2_hsv
7:     .xref         hsvtorgb
8:     .xref         gramadr
9:     .xref         gcliprect
10: *
11:     .offset 0      #gmonotone2_hsvの引数構造
12: *
13: X0:     .ds.w 1      #矩形座標
14: Y0:     .ds.w 1      #
15: X1:     .ds.w 1      #
16: Y1:     .ds.w 1      #
17: H:      .ds.w 1      #色相
18: S:      .ds.w 1      #飽和度
19: *
20:     .text
21:     .even
22: *
23: gmonotone2_hsv:
24: ARGPTR = 8
25: TBLSZ = RGBGRAD*2
26:     link a6, #-TBLSZ
27:     movem.l d0-d7/a0-a1, -(sp)
28:
29:     movea.l ARGPTR(a6), a1    #a1 = 引数列
30:     movem.w (a1), d0-d3      #d0～d3に座標を取り出す
31:
32:     jsr     gcliprect        #クリッピングする
33:     bne     done            #Z=0なら描画の必要なし
34:
35:     jsr     gramadr          #G-RAM上のアドレスを得る
36:
37:     sub.w   d0, d2           #d2 = 横ピクセル数-1
38:     sub.w   d1, d3           #d3 = 縦ピクセル数-1
39:
40:     move.l  H(a1), d1        #hsv(H, S, 0)～hsv(H, S, 31)の
41:     lea.l   -TBLSZ(a6), a1    #色テーブルを作っておく
42:     lsl.w   #8, d1           #
43:     moveq.l #RGBGRAD-1, d4    #
44: loop0:     move.l  d1, d0      #
45:     jsr     hsvtorgb          #
46:     move.w  d0, (a1)+         #
47:     addq.b  #1, d1           #

```

```

48:      dbra    d4, loop0      *
49:
50:      move.w  #GNBYTE-2, d1  *d1 = ライン間のアドレスの差
51:      sub.w   d2, d1          * (右端から下のラインの左端まで)
52:      sub.w   d2, d1          *
53:
54:      lea.l   -TBLSIZE(a6), a1
55: loop1: move.w  d2, d4          *d4 = 横ピクセル数-1
56: loop2: move.w  (a0), d0        *色コードを取り出し
57:      DERGB   d0, d5, d6, d7   *RGBに分解する
58:      RGBtoY  d5, d6, d7, d0   *Yをd0に求め
59:      *
60:      add.w   d0, d0            *hsv (H, S, d0) の色で
61:      move.w  0(a1, d0), (a0)+ * 点を打つ
62:      dbra    d4, loop2        *横幅分繰り返す
63:
64:      adda.w  d1, a0            *すぐ下のラインへ
65:      dbra    d3, loop1        *高さ分繰り返す
66:
67: done:  movem.l (sp)+, d0-d7/a0-a1
68:      unlk    a6
69:      rts
70:
71:      .end

```

モザイク処理

画面のピクセルを粗くする、解像度の量子化、いわゆるモザイク処理は特殊効果としてはなじみの深いものだろう。モザイク処理は、画像を適当な大きさの四角形（以下、大ピクセルと呼ぶ）に分割し、それぞれを代表する色で大ピクセル全体を塗る、という手順で実現される。ここでは、代表となる色をどう決めるかがポイントになる。大ピクセルの左上隅なり中央なりの適当な1ピクセルの色を拾って、その1色で代表させるのがもっとも手軽だが、この方法では残りのピクセルの情報が完全に失われてしまう。失う情報量を抑えるためには、大ピクセルに含まれる全ピクセルの色の平均をとるのがよい。このとき、大ピクセルの中央近くのピクセルを重要視し、適当に重みをつけて平均をとれば、さらに自然な結果が得られる。

リスト20にモザイク処理の1例を示す。大ピクセルの大きさは 2×2 ピクセル～ 32×32 ピクセルまでの範囲で選べるようにしてみた。ただし、色の平均化をとる部分では重みつき平均化は採用せず、単純平均ですましている。また、処理を施す領域は大ピクセルの大きさの倍数でなければならない。端数が出る場合は、領域の大きさを勝手に切り詰めてしまう。たとえば、 512×512 ピクセルの範囲を 3×3 ピクセルでモザイク化する場合、実際に処理されるのは 510×510 の範囲となり、右端と下端の2ピクセル分は元画像のまま残る。

リスト20
GMOAIC.S

```

1: *      モザイク処理
2:
3:      .include    gconst.h
4:      .include    gmacro.h

```

```

5: *
6:     .xdef    gmosaic
7:     .xref    gramadr
8:     .xref    gcliprect
9: *
10:    .offset 0      *gmosaicの引数構造
11: *
12: X0:    .ds.w    1      *矩形座標
13: Y0:    .ds.w    1      *
14: X1:    .ds.w    1      *
15: Y1:    .ds.w    1      *
16: MOSSIZ: .ds.w    1      *モザイクの大きさ (2~32)
17: *
18:     .text
19:     .even
20: *
21: gmosaic:
22: ARGPTR =      8
23:     link     a6, #0
24:     movem.l   d0-d7/a0-a5, -(sp)
25:
26:     movea.l   ARGPTR(a6), a1      *a1 = 引数列
27:     movem.w   (a1), d0-d4        *d0~d3に座標を取り出す
28:
29:     jsr       gcliprect          *クリッピングする
30:     bmi       done              *N=1なら描画の必要なし
31:
32:     jsr       gramadr            *G-RAM上のアドレスを得る
33:
34:     sub.w     d0, d2             *d2 = 横ピクセル数-1
35:     sub.w     d1, d3             *d3 = 縦ピクセル数-1
36:
37:     cmpi.w    #1, d4            *量子化レベルは
38:     ble       done              * 2~32まで
39:     cmpi.w    #32, d4
40:     bgt       done              *
41:
42:     addq.w    #1, d2             *d2 = 横ピクセル数
43:     addq.w    #1, d3             *d3 = 縦ピクセル数
44:     move.w    d2, d6            *d6 = 横ピクセル数
45:     ext.l     d2
46:     ext.l     d3
47:     divu.w    d4, d2             *d2 = 横大ピクセル数
48:     divu.w    d4, d3             *d3 = 縦大ピクセル数
49:
50:     swap.w    d2
51:     sub.w     d2, d6             *d6 = WID = 端数を除いた矩形の横ピクセル数
52:     swap.w    d2
53:
54:     moveq.l   #GSFCTR, d0        *d1 = GNB*SIZ
55:     move.w    d4, d1
56:     lsl.w     d0, d1
57:
58:     movea.w    d1, a4            *a4 = GNB*SIZ-H*2
59:     suba.w     d4, a4
60:     suba.w     d4, a4
61:
62:     sub.w     d6, d1             *d1 = GNB*SIZ-WID*2
63:     sub.w     d6, d1
64:
65:     lea.l     GNBYTE, w, a5      *a5 = GNB-SIZ*2
66:     suba.w     d4, a5
67:     suba.w     d4, a5
68:
69:     subq.w    #1, d2             *d2 = 横大ピクセル数-1
70:     bmi       done              *
71:     subq.w    #1, d3             *d3 = 縦大ピクセル数-1
72:     bmi       done              *
73:
74:     move.w    d4, d0             *a3 = 分母 = SIZ*SIZ
75:     mulu.w    d4, d0
76:     move.w    d0, a3
77:
78:     subq.w    #1, d4             *a2 = SIZ-1

```

```

79:         move.w d4, a2          *
80:
81: loop1:   move.w d2, d4          *d4 = 横大ピクセル数-1
82:         swap.w d1
83:         swap.w d2
84:         swap.w d3
85: loop2:   movea.l a0, a1        *a1 = 描画先ライン左端
86:         swap.w d4
87:
88:         clr.w d4               *d4.w = 青の輝度合計
89:         moveq.l #0, d5         *d5.l = 赤の輝度合計
90:         moveq.l #0, d6         *d6.l = 緑の輝度合計
91:
92:         move.w a2, d7          *d7 = SIZE-1
93: loop3:   swap.w d7
94:         move.w a2, d7          *d7 = SIZE-1
95: loop4:   move.w (a1)+, d0       *1ピクセル取り出す
96:         DERGB d0, d1, d2, d3   *RGBに分解する
97:         add.w d1, d4           *RGBごとに輝度を累積
98:         add.w d2, d5           *
99:         add.w d3, d6           *
100:        dbra d7, loop4         *大ピクセルの横幅分繰り返す
101:
102:        swap.w d7
103:        adda.w a5, a1          *大ピクセルの次ライン左端へ
104:        dbra d7, loop3        *大ピクセルの高さ分繰り返す
105:
106:        moveq.l #0, d7         *d7.l = 青の輝度合計
107:        move.w d4, d7          *
108:        move.w a3, d0          *d0 = 平均化の分母
109:        divu.w d0, d7          *d7 = 青の平均輝度
110:        divu.w d0, d5          *d5 = 赤の平均輝度
111:        divu.w d0, d6          *d6 = 緑の平均輝度
112:        RGB d7, d5, d6, d0     *RGBに再構成
113:
114:        move.w a2, d6          *大ピクセルを塗り潰す
115: loop5:   move.w a2, d7          *
116: loop6:   move.w d0, (a0)+      *
117:         dbra d7, loop6        *
118:         adda.w a5, a0         *
119:         dbra d6, loop5        *
120:
121:        swap.w d4
122:        suba.w a4, a0          *右隣の大ピクセルへ
123:        dbra d4, loop2        *横大ピクセル数分繰り返す
124:
125:        swap.w d1
126:        swap.w d2
127:        swap.w d3
128:        adda.w d1, a0          *1段下左端の大ピクセルへ
129:        dbra d3, loop1        *縦大ピクセル数分繰り返す
130:
131: done:   movem.l (sp)+, d0-d7/a0-a5
132:        unlk a6
133:        rts
134:
135:        .end

```

プログラムは各大ピクセルに含まれるピクセルの色の平均をとる2重ループと、大ピクセルを塗る2重ループを、さらに縦横の2重ループで括った4重ループ構造をしている。扱う変数の数が多いので、レジスタを総動員し、なおかつ、データレジスタの上位ワードと下位ワードを別目的で使ったりもしている。これにより、何とかワークエリアを使わずにすませることができた。プログラムのうえでは、54～67行のあたりに注目してもらおう。ここでは、

大ピクセル内の1ライン処理後の次ライン左端への移動

大ピクセルを1個処理後の右隣の大ピクセル左上隅への移動
 大ピクセルを横1列処理後の次段左端の大ピクセル左上隅への移動
 それぞれに必要な、G-RAMアドレスの差をレジスタに求めている。

平滑化

次は隣接するピクセル間の色の変化をなだらかに均す、いわゆるボカしだ。特殊効果としてだけでなく、画像のノイズを軽減する目的でも使用できる。概念上は色を周囲に少しずつにじませてやればよいわけだが、プログラムでは逆に周囲の点から少しずつ色をもらってくると考えたほうがわかりやすい。ボカしぐあいは色を混ぜあわせる比率と色をにじませる範囲で調節できる。広い範囲ににじませる場合には、中心から遠い点からは色を少しだけもらい、近くの点からはたくさんもらうというように適当な重みをつけて平均化してやると綺麗にボケる。このとき、にじませる範囲は正方形にとるよりも円形にしたほうがより自然な結果が得られる。という話をしておきながら、リスト21に示すサブルーチンはかなり単純な簡略版になってしまった。色をにじませる範囲は上下左右の1ピクセル、混合の比率は中心4：周囲1固定となっているし、ほかにもぼつぼつと手抜きがある。1つの例だと思ってもらえればありがたい。

リスト21
 GDEFOCUS.S

```

1: *      平滑化 (ボカし)
2:
3:      .include      gconst. h
4:      .include      gmacro. h
5: *
6:      .xdef      gdefocus
7:      .xref      gramadr
8:      .xref      gcliprect
9: *
10: DR      equ      2      #右の点とのアドレスの差
11: DD      equ      GNBYTE #下の点とのアドレスの差
12: *
13:      .offset 0      #gdefocusの引数構造
14: *
15: X0:      .ds. w      1      #矩形座標
16: Y0:      .ds. w      1      #
17: X1:      .ds. w      1      #
18: Y1:      .ds. w      1      #
19: *
20:      .offset 0      #5ピクセル分の作業用バッファ
21: *
22: C:      .ds. w      1      #中心
23: L:      .ds. w      1      #左
24: U:      .ds. w      1      #上
25: R:      .ds. w      1      #右
26: D:      .ds. w      1      #下
27: PBUFSIZ:
28: *
29:      .offset -GNBYTE-PBUFSIZ #スタックフレーム
30: *
31: WORKTOP:
32: PBUF:      .ds. b      PBUFSIZ #5ピクセル分の作業用バッファ
33: LBUF:      .ds. b      GNBYTE #1ライン分のバッファ
34: _a6:      .ds. l      1      #±0
35: _pc:      .ds. l      1
    
```

```

36: ARGPTR: .ds, l 1
37: #
38: .text
39: .even
40: #
41: gdefocus:
42:     link    a6, #WORKTOP
43:     movem.l d0-d7/a0-a2, -(sp)
44:
45:     movea.l ARGPTR(a6), a1    *a1 = 引数列
46:     movem.w (a1), d0-d3      *d0~d3に座標を取り出す
47:
48:     jsr     gcliprect        *クリッピングする
49:     bne     done             *Z=0なら描画の必要なし
50:
51:     jsr     gramadr          *G-RAM上のアドレスを得る
52:
53:     sub.w   d0, d2            *d2 = 横ピクセル数-1
54:     sub.w   d1, d3            *d3 = 縦ピクセル数-1
55:
56:     move.w  #GNBYTE-2, d1    *d1 = ライン間のアドレスの差
57:     sub.w   d2, d1            * (右端から下のラインの左端まで)
58:     sub.w   d2, d1            *
59:
60:     movea.l a0, a1            *一番上のラインを
61:     lea.l   LBUF(a6), a2      * バッファにコピーしておく
62:     move.w  d2, d4            *
63: loop0:     move.w  (a1)+, (a2)+ *
64:     dbra    d4, loop0         *
65:
66:     subq.w  #1, d2            *d2 = 横ピクセル数-1-1
67:     bmi     done             *
68:     subq.w  #1, d3            *d3 = 縦ピクセル数-1-1
69:     bmi     done             *
70:
71:     lea.l   PBUF(a6), a1
72: loop1:     move.w  d2, d4      *d4 = 横ピクセル数-1-1
73:     lea.l   LBUF(a6), a2      *a2 = 1ラインのバッファ
74:     move.w  (a0), L(a1)       *左端のみ (x-1, y) の代わりに (x, y)
75: loop2:     move.w  (a0), C(a1) * (x, y)
76:     move.w  (a2), U(a1)       * (x, y-1)
77:     move.w  DR(a0), R(a1)     * (x+1, y)
78:     move.w  DD(a0), D(a1)     * (x, y+1)
79:     bsr     gsub              *色を混ぜ合わせる
80:     move.w  (a0), (a2)+       *下のライン用に覚えておく
81:     move.w  (a0), L(a1)       * (x-1, y)
82:     move.w  d0, (a0)+         * 1 ピクセル書き込む
83:     dbra    d4, loop2         *x1-x0-1回繰り返す
84:
85:                                     *右端の処理
86:     move.w  (a0), C(a1)       * (x, y)
87:     move.w  (a2), U(a1)       * (x, y-1)
88:     move.w  (a0), R(a1)       * (x+1, y) の代わりに (x, y)
89:     move.w  DD(a0), D(a1)     * (x, y+1)
90:     bsr     gsub              *色を混ぜ合わせる
91:     move.w  (a0), (a2)+       *下のライン用に覚えておく
92:     move.w  d0, (a0)+         * 1 ピクセル書き込む
93:
94:     adda.w  d1, a0            *すぐ下のラインへ
95:     dbra    d3, loop1         *y1-y0-1回繰り返す
96:
97:                                     *最下ラインの処理
98:     move.w  d2, d4            *d4 = 横ピクセル数-2
99:     lea.l   LBUF(a6), a2      *a2 = 1ラインのバッファ
100:    move.w  (a0), L(a1)        *左端のみ
101:                                     * (x-1, y) の代わりに (x, y)
102: loop3:    move.w  (a0), C(a1)  * (x, y)
103:    move.w  (a2), U(a1)        * (x, y-1)
104:    move.w  DR(a0), R(a1)      * (x+1, y)
105:    move.w  (a0), D(a1)        * (x, y+1) の代わりに (x, y)
106:    bsr     gsub              *色を混ぜ合わせる
107:    move.w  (a0), L(a1)        * (x-1, y)
108:    move.w  d0, (a0)+          * 1 ピクセル書き込む
109:    dbra    d4, loop3         *x1-x0-1回繰り返す

```

```

110:
111:                *右下隅の処理
112:    move.w    (a0), C(a1)    * (x, y)
113:    move.w    (a2), U(a1)    * (x, y-1)
114:    move.w    (a0), R(a1)    * (x+1, y) の代わりに (x, y)
115:    move.w    (a0), D(a1)    * (x, y+1) の代わりに (x, y)
116:    bsr       gsub           *色を混ぜ合わせる
117:    move.w    d0, (a0)       *1ピクセル書き込む
118:
119: done:    movem.l    (sp)+, d0-d7/a0-a2
120:    unlk      a6
121:    rts
122: *
123: gsub:    movem.l    d1-d4/a1, -(sp)
124:    move.w    (a1)+, d0      *中心の点
125:    DERGB     d0, d1, d2, d3 *RGBごとに
126:    lsl.w     #2, d1         * 4倍
127:    lsl.w     #2, d2         *
128:    lsl.w     #2, d3         *
129:
130:    moveq.l    #4-1, d0      *周囲の4点の色を
131:    sublp:    move.w    (a1)+, d4    * RGBごとに加える
132:    DERGB     d4, d5, d6, d7    *
133:    add.w     d5, d1         *
134:    add.w     d6, d2         *
135:    add.w     d7, d3         *
136:    dbra      d0, sublp      *
137:
138:    lsr.w     #3, d1         *8で割る
139:    lsr.w     #3, d2         *
140:    lsr.w     #3, d3         *
141:
142:    RGB       d1, d2, d3, d0    *カラーコードに再構成
143:
144:    movem.l    (sp)+, d1-d4/a1
145:    rts
146:
147:    .end

```

メインルーチンの前に、先に123行以下のサブルーチンを見てもらおう。ここでは中心の色と周囲の色のRGBごとの混ぜあわせを行っている。ボカし処理の本体だ。サブルーチンが呼び出された時点で、いま処理する点とその周囲の点の色が、a3レジスタでポイントされるメモリ領域に22～26行のような順序・構造で格納されている。中心の点の色をRGBに分解してから4倍し、これに周囲の色をRGBごとに足し込んで、8で割れば色を4:1:1:1:1で混ぜあわせたことになる。

72行からメインループが始まる。色の変換自体はサブルーチンにまかせているので、メインループでは処理対象となる5ピクセルの色をバッファにセットしてサブルーチンに渡す以上の仕事はしていない。いつものように、処理は左上の点から右方向、そして下方向に進む。ここで、処理途中におけるある1点に注目しよう。すぐ左の点と真上の点はすでにボカし処理済みであり、この点の色を拾ってしまうと正しい色の平均化が行えない。そこで、リスト21では真上の1ライン分はスタック上に確保したバッファに保存しておき、左の点に関しては画面の書き換えを1拍遅らせる（80～82行）ことで対処している。

また、リスト21では、指定された領域のいちばん上といちばん下の1ライン、および各ラインの左端と右端を特別扱いし、本来であれば指定された矩形領域の1ピクセル外側の点の色を拾うべきところを1ピクセル内側の点の色で代用している。これは指定の矩形領域が実画面の

端に接しているときに変なメモリをアクセスしないようにするための処置だ……が、領域が実画面の端に触れていない場合でも同じ処理をするという手抜きをしてしまった。このため、つねに領域の外枠部分だけボケぐあいがある。隣接する領域を別々にボカすようなことをしないかぎりには目立たないということで目をつむってもらおう。

もう1つ、別の方法を使った平滑化処理ルーチンを示そう。リスト22（リスト21に対する変更点のみ）はいわゆるメディアン（median：中央値）フィルタのもっとも単純な例で、各ピクセルについてRGBごとに周囲4ピクセルと自分自身の輝度の中央値を求め、カラーコードに再構成して、得られた色に置き換えていく。取り込み画像には、よく点状や線状のノイズが入ることがあるが、メディアンフィルタはそのような突発的なノイズを除去・軽減するのに有効だ。色の混ぜあわせを行うわけではないので、基本的に画像の輪郭は保たれる。

リスト22
GMEDIAN.S

```

1: *      平滑化（中央値フィルタ）
6:      .xdef    gmedian
41: gmedian:
123: gsub:   movem.l d1-d4/a0-a1, -(sp)
124:      lea.l   -PBUFSIZ*3(sp), sp
125:
126:      movea.l  sp, a0
127:      moveq.l  #5-l, d4
128: gloop:   move.w  (a1)+, d0      #中心の点
129:      DERGB    d0, d5, d6, d7    #RGBに分解
130:      move.w   d5, PBUFSIZ*2(a0)  #g
131:      move.w   d6, PBUFSIZ(a0)    #r
132:      move.w   d7, (a0)+         #b
133:      dbra     d4, gloop
134:
135:      movea.l  sp, a0
136:      moveq.l  #0, d0
137:      bsr      median
138:      lsl.w    #5, d0
139:      bsr      median
140:      lsl.w    #5, d0
141:      bsr      median
142:      add.w    d0, d0
143:
144:      lea.l    PBUFSIZ*3(sp), sp
145:      movem.l  (sp)+, d1-d4/a0-a1
146:      rts
147: *
148: median:
149:      movem.w  (a0)+, d1-d5
150:      MINMAX   d1, d2
151:      MINMAX   d1, d3
152:      MINMAX   d1, d4
153:      MINMAX   d1, d5
154:      MINMAX   d2, d3
155:      MINMAX   d2, d4
156:      MINMAX   d2, d5
157:      MINMAX   d3, d4
158:      MINMAX   d3, d5
159:      or.w     d3, d0
160:      rts
161:
162:      .end

```

プログラムのうえでは、リスト22には見るべき点は何もない。とくに、中央値を求める148

行以下は力づくもいいところだ。150～153行で最小値、154～156行で2番目に小さい値、157～158行で中央値である3番目に小さい値を直接的な値の比較により求めている。

鮮鋭化

ここていう鮮鋭化とは、ピクセル間の輝度変化を増幅し、画像（主として自然画像）の細部をくっきりさせる処理をいう。ちょうどボカしと対になる操作だ。前段階として、輝度の変化を視覚化することを考えよう。処理中のピクセルの輝度と右隣のピクセルの輝度との差から水平方向の輝度変化量が、同様に真下のピクセルの輝度との差から垂直方向の輝度変化量が得られる。その和を、注目しているピクセルの輝度変化量と考える。ちょっとインチキ気味だが、これは数学的にはx方向とy方向に画像を微分することだったりする。リスト23はその微分処理を行うプログラムだ。リスト21からの変更点のかたちで示してある。周囲よりも暗い点は暗い灰色、周囲よりも明るい点は明るい灰色に置き換えることで輝度変化を目に見えるかたちで表示する。ちょうど、画像の輝度を立体的な凹凸に置き換え、左斜め上から光を当てたような結果が得られる。32階調の場合、輝度変化量は-62～+62の値で得られるわけだが、リスト23ではこれに62を足して（144行）非負にしたうえで4で割って（145行）0～31に収め、最終的な灰色の輝度として使っている。

.....
リスト23
GDIFFERS
(リスト21に対する追加
・変更部)

```

1: *      微分処理
6:      .xdef  gdifferntial

41: gdifferential:

123:      .xref  grayscale
124: *
125: gsub:  movem, l  d1/a1, -(sp)
126:      move, w  (a1), d0      *中心の点
127:      DERGB    d0, d5, d6, d7 *RGBに分解し
128:      RGBtoY    d5, d6, d7, d0 * 輝度lcを得る
129:      add, w     d0, d0        *
130:      neg, w     d0            *d0 = -2*lc
131:
132:      addq, l    #R-C, a1      *左の点と上の点を飛ばす
133:
134:      move, w    (a1)+, d1      *右の点
135:      DERGB      d1, d5, d6, d7 *RGBに分解し
136:      RGBtoY      d5, d6, d7, d1 * 輝度lrを得る
137:      add, w      d1, d0        *d0 = lr-2lc
138:
139:      move, w    (a1), d1      *下の点
140:      DERGB      d1, d5, d6, d7 *RGBに分解し
141:      RGBtoY      d5, d6, d7, d1 * 輝度ldを得る
142:      add, w      d1, d0        *d0 = (lr-lc)+(ld-lc)
143:
144:      addi, w     #RGBMAX*2, d0 * -31*2～+31*2を
145:      lsr, w      #2, d0        * 0～31に変換
146:      add, w      d0, d0        *対応する灰色の
147:      lea, l      grayscale, a1 * カラーコードに
148:      move, w     0(a1, d0), d0 * 置き換える

```

```
149:
150:      movem.l (sp)+, d1/a1
151:      rts
152:
153:      .end
```

また、リスト24は、右と下だけではなく上下左右のピクセルとの輝度の差の総和を使って同様の処理を行うプログラム例だ。これは画像を2次元2次微分（ラプラシアン）することに相当するのだが、直感的には、周囲4ピクセルの輝度の平均との差を求めていると思えばよい。鮮鋭化は、こうして得られたラプラシアン像と元画像を重ねあわせることで実現できる。リスト25にプログラム例を示す。鮮鋭化はその性格上、画像のノイズまでも増幅してしまう傾向にあるので、場合によってはあらかじめリスト22などによりノイズを軽減しておくのがよいだろう。

.....
リスト24
GLAPLACIANS
(リスト21に対する追加
・変更部)

```
1: #      微分処理（ラプラシアン）
6:      .xdef   glaplacian

41: glaplacian:

123:      .xref   grayscale
124: #
125: gsub:  movem.l d1-d2/a1, -(sp)
126:      move.w   (a1)+, d0      *中心の点
127:      DERGB    d0, d5, d6, d7  *RGBに分解し
128:      RGBtoY   d5, d6, d7, d0  * 輝度1cを得る
129:
130:      lsl.w    #2, d0          *その4倍から
131:      moveq.l  #4-1, d2        * 周囲の4点の輝度を減じる
132: sublp:  move.w   (a1)+, d1      *
133:      DERGB    d1, d5, d6, d7  *
134:      RGBtoY   d5, d6, d7, d1  *
135:      sub.w    d1, d0          *
136:      dbra     d2, sublp       *
137:
138: #
139:      addi.w   #RGBMAX*4, d0    *~31*4~+31*4を
140:      lsr.w    #3, d0          * 0~31に変換
141:      add.w    d0, d0          *対応する灰色の
142:      lea.l    grayscale, a1   * カラーコードに
143:      move.w   0(a1, d0), d0    * 置き換える
144:
145:      movem.l  (sp)+, d1-d2/a1
146:      rts
147:
148:      .end
```

.....
リスト25
GSHARP.S
(リスト21に対する追加
・変更部)

```
1: #      鮮鋭化（ラプラシアン像との合成）
6:      .xdef   gsharp

41: gsharp:
```

```

123: gsub:   movem.l d1-d4/a1, -(sp)
124:         move.w (a1)+, d0      *中心の点
125:         DERGB d0, d1, d2, d3  *RGBごとに8倍する
126:         lsl.w #3, d1          *
127:         lsl.w #3, d2          *
128:         lsl.w #3, d3          *
129:
130:         moveq.l #4-1, d0      *周囲の4点の色を
131: sublp:   move.w (a1)+, d4      * RGBごとに減じる
132:         DERGB d4, d5, d6, d7  *
133:         sub.w d5, d1          *
134:         sub.w d6, d2          *
135:         sub.w d7, d3          *
136:         dbra d0, sublp        *
137:
138:         asr.w #2, d1          *4で割る
139:         asr.w #2, d2          *
140:         asr.w #2, d3          *
141:
142:         moveq.l #0, d0        *最小輝度以上を保証
143:         MAX d0, d1            *
144:         MAX d0, d2            *
145:         MAX d0, d3            *
146:
147:         moveq.l #RGBMAX, d0   *最大輝度以下を保証
148:         MIN d0, d1            *
149:         MIN d0, d2            *
150:         MIN d0, d3            *
151:
152:         RGB d1, d2, d3, d0    *カラーコードに再構成
153:
154:         movem.l (sp)+, d1-d4/a1
155:         rts
156:
157:         .end

```

色強調

これは各ピクセルの色を原色に近づけることで画像のコントラストを上げる操作だ。色をRGBに分解したうえで次式の変換を行うと、RGBのうち最大のものが強調され、原色に近づいていく。

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} a & -b & -b \\ -b & a & -b \\ -b & -b & a \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

ここで、

$a = \text{強調率} (\geq 1)$

$b = (a-1) / 2$

である。

リスト26に実例を示す。本来、強調率 a は1より少し大きい程度の実数値をとるのが、リスト26では64倍して整数化した値を使う。プログラムでは、 a 倍、 b 倍に乗算命令を使わなくてもすむようにするために、初期化時点で、0~31の値を a 倍、 b 倍するといくつになるのか

をテーブルにしておく (51~61行) という姑息な技が使われている。

リスト26
GACCENT.S

```

1: #      色強調
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gaccent
7:      .xref      gramadr
8:      .xref      gcliprect
9: #
10:     .offset 0      *gaccentの引数構造
11: #
12: X0:   .ds.w 1      *矩形座標
13: Y0:   .ds.w 1      *
14: X1:   .ds.w 1      *
15: Y1:   .ds.w 1      *
16: A:    .ds.w 1      *強調率 (基準値64)
17: #
18:     .offset -RGBGRAD*2*2  *スタックフレーム
19: #
20: WORKTOP:
21: ATBL: .ds.w RGBGRAD *0~31をa倍したテーブル
22: BTBL: .ds.w RGBGRAD *0~31をb=(a-1)/2倍したテーブル
23: _a6:  .ds.l 1      *±0
24: _pc:  .ds.l 1
25: ARGPTR: .ds.l 1
26: #
27:     .text
28:     .even
29: #
30: gaccent:
31:     link    a6, #WORKTOP
32:     movem.l d0-d7/a0-a1, -(sp)
33:
34:     movea.l ARGPTR(a6), a1      *a1 = 引数列
35:     movem.w (a1), d0-d3        *d0~d3に座標を取り出す
36:
37:     jsr     gcliprect          *クリッピングする
38:     bne     done              *Z=0なら描画の必要なし
39:
40:     jsr     gramadr            *G-RAM上のアドレスを得る
41:
42:     sub.w   d0, d2             *d2 = 横ピクセル数-1
43:     sub.w   d1, d3             *d3 = 縦ピクセル数-1
44:
45:     move.w  A(a1), d7          *d7 = a (ただし64倍)
46:     move.w  d7, d6             *d6 = a-1 (ただし64倍)
47:     subi.w  #64, d6            *
48:
49: BOFST =      BTBL-ATBL        *2つのテーブルのアドレスの差
50:
51:     lea.l   ATBL(a6), a1      *0~31の値をa倍, (a-1)/2倍すると
52:     moveq.l #0, d0            *いくつになるか計算して
53:     moveq.l #0, d1            *テーブルにしておく
54:     moveq.l #32-1, d4         *
55: loop0: move.w d1, d5           *
56:     lsr.w   #1, d5            *
57:     move.w  d5, BOFST(a1)     *
58:     move.w  d0, (a1)+         *
59:     add.w   d7, d0            *
60:     add.w   d6, d1            *
61:     dbra   d4, loop0          *
62:
63:     move.w  #GNBYTE-2, d1     *d1=ライン間のアドレスの差
64:     sub.w   d2, d1            * (右端から下のラインの左端まで)
65:     sub.w   d2, d1            *
66:
67:     lea.l   ATBL(a6), a1
68: loop1: move.w d2, d4          *d4=横ピクセル数-1
69:     swap.w  d1

```



```

70:      swap.w d2
71:      swap.w d3
72: loop2: move.w (a0), d0      *カラーコードを取り出し
73:      DERGB d0, d5, d6, d7 *RGBに分解する
74:      add.w d5, d5          *
75:      add.w d6, d6          *
76:      add.w d7, d7          *
77:      move.w 0(a1, d5), d1  *d1 = Ab
78:      move.w 0(a1, d6), d2  *d2 = Ar
79:      move.w 0(a1, d7), d3  *d3 = Ag
80:      move.w BOFST(a1, d5), d5 *d5 = Bb
81:      move.w BOFST(a1, d6), d6 *d6 = Br
82:      move.w BOFST(a1, d7), d7 *d7 = Bg
83:
84:      sub.w d6, d1          *d1 = b' = Ab-Br-Bg
85:      sub.w d7, d1          *
86:      sub.w d7, d2          *d2 = r' = Ar-Bg-Bb
87:      sub.w d5, d2          *
88:      sub.w d5, d3          *d3 = g' = Ag-Bb-Br
89:      sub.w d6, d3          *
90:
91:      asr.w #6, d1          *1/64にする
92:      asr.w #6, d2          *
93:      asr.w #6, d3          *
94:
95:      moveq.l #0, d0        *最小輝度以上を保証
96:      MAX d0, d1            *
97:      MAX d0, d2            *
98:      MAX d0, d3            *
99:
100:     moveq.l #RGBMAX, d0    *最大輝度以上を保証
101:     MIN d0, d1            *
102:     MIN d0, d2            *
103:     MIN d0, d3            *
104:
105:     RGB d1, d2, d3, d0     *カラーコードに再構成して
106:     move.w d0, (a0)+       * 点を打つ
107:
108:     dbra d4, loop2         *横幅分繰り返す
109:
110:     swap.w d1
111:     swap.w d2
112:     swap.w d3
113:     adda.w d1, a0          *すぐ下のラインへ
114:     dbra d3, loop1        *高さ分繰り返す
115:
116: done: movem.l (sp)+, d0-d7/a0-a1
117:      unlk a6
118:      rts
119:
120:      .end

```

ディザ法による階調変換

ディザ (dither) 法は、多階調で表現された画像をより少ない階調の画像に変換する方法の1つだ。すでに紹介した単純な色の量子化と異なり、ディザ法では色数を減らしても、画質の劣化はある程度抑えられる。

単純な色の量子化により階調を落とすと画質が劣化するのは、下位ビットの端数を一律切り捨てることで細部の情報が失われるからだ。この端数も生かそうというのが、ディザ法やそれに類するアルゴリズムの基本方針となる。少々脱線するが、ディザ法と目的を同じくする別の

方法に、誤差分散（拡散）法と呼ばれるアルゴリズムがある。このアルゴリズムでは、ビット数を詰めるときに出る端数を周囲のピクセルに散らして加算していく。すると、端数がある程度累積したところで切り上げが起り、そのピクセルは少し明るくなる。この明るいピクセルは適当な頻度で生まれて、いわば不規則なタイルパターンをかたちづくる。こうしてできる中間色により、階調の不足分が補われるというわけだ。

誤差分散法が処理過程で発生する内部的な誤差を利用するのに対し、ディザ法では誤差を外部から持ち込む。その一種、ランダムディザと呼ばれる方法では、この誤差として文字どおり乱数を使う。各ピクセルごとに下位ビットを切り捨てる前に切り捨てられるビット数に応じた大きさの乱数を加えてやることで、端数が切り捨てられたり、切り上げられたりして、中間色のタイルパターンを生み出すのだ。

乱数を使わずに、持ち込む誤差のパターンをあらかじめ決めておく方式はオーダーディザと呼ばれる。誤差のパターンはマトリクス（行列）のかたちで用意する。最小のディザマトリクスは次に示す 2×2 のものだ。

$$\begin{bmatrix} D_2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$$

ディザマトリクスは画像に対して繰り返し適用される。つまり、 2×2 の場合、x座標を2で割ったあまりを列位置、y座標を2で割ったあまりを行位置とする要素を該当位置のピクセル輝度に加算するわけだ。

各要素が0～3の値をとることから、 2×2 のディザマトリクスは4階調の輝度レベルを生み出す。これは、このディザマトリクスが2ビットの階調不足を補えることを意味する。もっと多くの階調不足を補うときには、この 2×2 のマトリクスと、次の漸化式からより大きなディザマトリクスを導いて使う（添え字は行列の大きさを表す）。

$$\begin{bmatrix} D_n \end{bmatrix} = \begin{bmatrix} 4D_{n/2} & 4D_{n/2} + 2U_{n/2} \\ 4D_{n/2} + 3U_{n/2} & 4D_{n/2} + U_{n/2} \end{bmatrix} \quad (n=4, 8, 16, \dots)$$

ここで、 U は

$$\begin{bmatrix} U_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ & 1 & 1 & 1 & \dots & 1 \\ & & 1 & 1 & 1 & \dots & 1 \\ & & & \vdots & & \vdots \\ & & & & 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

のような全要素が1である $n \times n$ 行列だ⁴⁾。この漸化式により、たとえば、 4×4 のディザマトリクスは、

$$\begin{bmatrix} & & & \\ & & & \\ D_4 & = & \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \\ & & & \end{bmatrix}$$

のように導かれる。4×4になっても、行列中には要素の重複がないことに注目してほしい。4×4のディザマトリクスは16階調、4ビット相当の輝度レベルをつくり出す。以下、行列のサイズを2倍、また、2倍と大きくしていっても、この性質は保たれる。

■4) 老婆心ながら、行列どうしの加算は同じ行・列位置の要素間の加算、行列のスカラー（ベクトルなどではない“普通の数”）倍はその値を行列の全要素に個別にかけることを表す。

リスト27はオーダードディザによりRGB32階調の画像を16、8、4、2階調に変換するプログラムの1例だ。何階調に落とすかは、残すビット数で指定する。16階調にする場合は4、8階調にする場合は3というぐあいだ。ディザマトリクスは8×8の大きさに、別ファイル、リスト28に用意してある。リスト28自体はリスト29のX-BASICプログラムで生成した。8×8のディザマトリクスは64階調、6ビット分の階調不足を補うわけだが、このプログラムで落とすビット数は1〜4ビットであり、このままではディザマトリクスが強く影響しすぎる。そこで、リスト27では、ディザマトリクス中の各要素が落とすビット数に見合った大きさの値になるよう、数分の1に小さくして使うようになっている。どれだけ小さくするかは固定でもよかったのだが、引数で指定できるようにしてみた。リスト27では、実際にはディザマトリクスの要素を小さくする代わりに、元画像側の輝度を増幅するので、与える引数はディザマトリクスの重みの逆数のかたちで指定する。基準値は、

16階調 → 32
8階調 → 16
4階調 → 8
2階調 → 4

だ。故意にディザマトリクスの影響力を強めて特殊効果を狙いたい場合は、基準値よりも少し小さな値を指定すればよい。

リスト27
GDITHER.S

```
1: #      オーダードディザにより階調を落とす
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gdither
7:      .xref      gramadr
8:      .xref      gcliprect
9:      .xref      dithermatrix
10: #
11: MATSIZ equ      8      *ディザマトリクスの大きさ
12: #
13:      .offset 0      *gditherの引数構造
14: #
15: X0:      .ds.w      1      *矩形座標
```

```

16: Y0: .ds.w 1 *
17: X1: .ds.w 1 *
18: Y1: .ds.w 1 *
19: NBIT: .ds.w 1 *変換後のRGBごとのビット数 (1~5)
20: SCL: .ds.w 1 *ディザマトリクスの重みの逆数 (基準値2`NBIT)
21: *
22: .text
23: .even
24: *
25: gdither:
26: ARGPTR = 8
27: link a6, #0
28: movem.l d0-d7/a0-a4, -(sp)
29:
30: movea.l ARGPTR(a6), a1 *a1 = 引数列
31: movem.w (a1), d0-d3 *d0~d3に座標を取り出す
32:
33: jsr gcliprect *クリッピングする
34: bmi done *N=1なら描画の必要なし
35:
36: jsr gramadr *G-RAM上のアドレスを得る
37:
38: sub.w d0, d2 *d2 = 横ピクセル数-1
39: sub.w d1, d3 *d3 = 縦ピクセル数-1
40: lsl.w #3, d1
41:
42: move.w NBIT(a1), d4 *落とす階調に応じた
43: subq.w #1, d4 * スケーリング用のテーブルを得る
44: bmi done *
45: cmpi.w #5+1, d4 *
46: bcc done *
47: lsl.w #5, d4 *
48: lea.l ctbl(pc), a4 *
49: adda.w d4, a4 *
50:
51: swap.w d3 *d3 = マトリクスの重みの逆数
52: move.w SCL(a1), d3 *
53: beq done *
54: swap.w d3 *
55:
56: lea.l dithermatrix, a2
57: move.w d0, d4 *d4 = 左端x座標
58: loop1: movea.l a0, a1 *a1 = ライン左端
59: andi.w # (MATSIZ-1)*MATSIZ, d1
60: *d1 = ディザマトリクスの行インデックス
61: lea.l 0(a2, d1), a3 *a3 = ディザマトリクスの行
62:
63: swap.w d4
64: move.w d2, d4 *d4 = 横ピクセル数-1
65: swap.w d1
66: swap.w d2
67: swap.w d3
68: loop2: move.w (a1), d1 *1ピクセル取り出す
69: DERGB d1, d5, d6, d7 *RGBに分解する
70: mulu.w d3, d5 *適当な重みを掛ける
71: mulu.w d3, d6 *
72: mulu.w d3, d7 *
73:
74: andi.w #MATSIZ-1, d0 *d0 = ディザマトリクスの列インデックス
75: move.b 0(a3, d0), d2 *d2 = 加算する震え成分
76: ext.w d2 *
77:
78: add.w d2, d5 *RGBごとに震え成分を加える
79: add.w d2, d6 *
80: add.w d2, d7 *
81:
82: clr.w d1 *最小輝度以上を保証
83: MAX d1, d5 *
84: MAX d1, d6 *
85: MAX d1, d7 *
86:
87: divu.w d3, d5 *RGB32階調に再変換
88: divu.w d3, d6 *
89: divu.w d3, d7 *

```

```

90:
91:     move.w  $RGBMAX, d1    *最大輝度以下を保証
92:     MIN     d1, d5         *
93:     MIN     d1, d6         *
94:     MIN     d1, d7         *
95:
96:     move.b  0(a4, d5), d5   *階調に応じてスケーリング
97:     move.b  0(a4, d6), d6   *
98:     move.b  0(a4, d7), d7   *
99:
100:    RGB      d5, d6, d7, d1  *カラーコードに再構成
101:    move.w   d1, (a1)+       *書き込む
102:
103:    addq.w   #1, d0          *ディザマトリクスの列を進める
104:    dbra     d4, loop2       *横幅分繰り返す
105:
106:    swap.w   d1
107:    swap.w   d2
108:    swap.w   d3
109:    swap.w   d4
110:    move.w   d4, d0          *ディザマトリクス列インデックスをリセット
111:    addq.w   #MATSIZE, d1    *ディザマトリクスの行を進める
112:    lea.l    GNBYTE(a0), a0  *すぐ下のラインへ
113:    dbra     d3, loop1       *高さ分繰り返す
114:
115: done:      movem.l  (sp)+, d0-d7/a0-a4
116:            unlk     a6
117:            rts
118: *
119: ctbl:      .dc.b    00, 00, 00, 00, 00, 00, 00, 00 *RGB各2階調
120:            .dc.b    00, 00, 00, 00, 00, 00, 00, 00
121:            .dc.b    31, 31, 31, 31, 31, 31, 31, 31
122:            .dc.b    31, 31, 31, 31, 31, 31, 31, 31
123: *
124:            .dc.b    00, 00, 00, 00, 00, 00, 00, 00 *RGB各4階調
125:            .dc.b    10, 10, 10, 10, 10, 10, 10, 10
126:            .dc.b    21, 21, 21, 21, 21, 21, 21, 21
127:            .dc.b    31, 31, 31, 31, 31, 31, 31, 31
128: *
129:            .dc.b    00, 00, 00, 00, 04, 04, 04, 04 *RGB各8階調
130:            .dc.b    09, 09, 09, 09, 13, 13, 13, 13
131:            .dc.b    18, 18, 18, 18, 22, 22, 22, 22
132:            .dc.b    27, 27, 27, 27, 31, 31, 31, 31
133: *
134:            .dc.b    00, 00, 02, 02, 04, 04, 06, 06 *RGB各16階調
135:            .dc.b    08, 08, 10, 10, 12, 12, 14, 14
136:            .dc.b    17, 17, 19, 19, 21, 21, 23, 23
137:            .dc.b    25, 25, 27, 27, 29, 29, 31, 31
138: *
139:            .dc.b    00, 01, 02, 03, 04, 05, 06, 07 *RGB各32階調
140:            .dc.b    08, 09, 10, 11, 12, 13, 14, 15
141:            .dc.b    16, 17, 18, 19, 20, 21, 22, 23
142:            .dc.b    24, 25, 26, 27, 28, 29, 30, 31
143:
144:            .end

```

リスト28
DITHERMAT.S

```

1: *      8x8ディザマトリクス
2:
3:      .xdef  dithermatrix
4: *
5:      .data
6:      .even
7: *
8: dithermatrix:
9:      .dc.b  -32, +00, -20, +12, -29, +03, -17, +15
10:     .dc.b  +16, -16, +28, -04, +19, -13, +31, -01
11:     .dc.b  -24, +08, -28, +04, -21, +11, -25, +07
12:     .dc.b  +24, -08, +20, -12, +27, -05, +23, -09

```

```

13:      .dc.b   -30,+02,-18,+14,-31,+01,-19,+13
14:      .dc.b   +18,-14,+30,-02,+17,-15,+29,-03
15:      .dc.b   -22,+10,-26,+06,-23,+09,-27,+05
16:      .dc.b   +26,-06,+22,-10,+25,-07,+21,-11
17:
18:      .end

```

リスト29
GENDITHMAT.BAS

```

10 dim char m22(1,1) = { 0, 2, 3, 1}
20 dim char m44(3,3),m88(7,7)
30 int i,j,fp
40 /*
50 /* 2x2 → 4x4
60 for i=0 to 1:for j=0 to 1
70   m44(i+0,j+0) = 4*m22(i,j)
80   m44(i+2,j+0) = 4*m22(i,j)+2
90   m44(i+0,j+2) = 4*m22(i,j)+3
100  m44(i+2,j+2) = 4*m22(i,j)+1
110 next:next
120 /*
130 /* 4x4 → 8x8
140 for i=0 to 3:for j=0 to 3
150   m88(i+0,j+0) = 4*m44(i,j)
160   m88(i+4,j+0) = 4*m44(i,j)+2
170   m88(i+0,j+4) = 4*m44(i,j)+3
180   m88(i+4,j+4) = 4*m44(i,j)+1
190 next:next
200 /*
210 fp = fopen("$","c")
220 for i=0 to 7:for j=0 to 7
230   fwrites(itoa(m88(i,j)-32)+",",fp)
240 next:fwrites(chr$(13)+chr$(10),fp):next
250 fclose(fp)

```

リスト28のディザマトリクスは先に示した漸化式から導かれる行列そのままではなく、行列中の中央値である32を全要素から引いたかたちになっている。これは、リスト27では、階調を落とすときに四捨五入的な操作を導入したことによる。たとえば、32階調を4階調に落とす場合、普通に11000_Bでマスクすると、

0,8,16,24

の4段階の輝度が得られるわけだが、リスト27ではこれを、

0,10,21,31

に引き伸ばし、画面上の最大輝度を31に保つようになっている。この操作は元画像の輝度の総和をほぼ保つ効果があり、そこへ全要素が正の数であるようなディザマトリクスを加えたのでは、画像全体が明るいほうにシフトしてしまう。リスト28のディザマトリクスはこの分の補正を含んでいるのだ。正直なところ、この対応が正しいかどうか自信がないのだが、実行結果を見るかぎり問題はないように見える。

さて、上では65536色モードの画像を単に画面上で階調を落として遊んでみたわけだが、ディザ法のありがたみがわかるのは、実際に少ない階調しか表現できない256色モードや16色モードへ画像をコンバートする場合だ。リスト30に65536色（実際は32768色）から256色モードへの画像変換ルーチンを示す。すでに触れたように、X68000の256色モードでの8ビットパレットコードは、

GGRRRBBB

の意味を持つかのように初期化されるから、これにあわせて、赤、青を8階調、緑だけを4階調に落としている。このプログラムでは、画面モード自体は65536色モードのまま256色モード時のパレットコードをG-RAMに書き込んでいることに注意しよう。このままでは画面モードもパレットの設定もあっていないので正しい色では表示されない。変換後の画像はいったんファイルに落とすなり、メインメモリに転送するなりしてから、画面モードを256色モードに切り替えて再表示する必要がある。もともと、うまくしたもので、G-RAM内容を消去せずに65536色モードから256色モードに切り替えると、65536色モード時の各ピクセルは座標位置は変わらないまま、ワードの下位バイトがページ0、上位バイトがページ1に対応するよう分割される。つまり、初期化しないよう指定してIOCS CRTMODで256色モードに切り替えれば、いま変換した画像をそのまま256色モードで表示することができる。パレットは自前で初期化しなければならないが、初期化ルーチンはすでに用意してあったから、こいつを呼び出すだけで。結局、リスト30のgltomで変換した画像は、gltomの呼び出しの直後に、

```
move.w  #$100+8      *512×512,256色モード (初期化なし)
IOCS     _CRTMOD      *
jsr      ginitpalet_m *256色モード用パレット初期化
```

を実行すれば、その場で見られる。

リスト30
GLTOM.S

```
1: *      32768色→256色変換
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6: MATSIZ equ      8      *ディザマトリクス of の大きさ
7: *
8:      .xdef      gltom
9:      .xref      gramadr
10:     .xref      gcliprect
11:     .xref      dithermatrix
12: *
13:     .offset 0      *gltomの引数構造
14: *
15: X0:     .ds.w      1      *矩形座標
16: Y0:     .ds.w      1      *
17: X1:     .ds.w      1      *
18: Y1:     .ds.w      1      *
19: *
20:     .text
21:     .even
22: *
23: gltom: =
24: ARGPTR 8
25:     link      a6, #0
26:     movem.l   d0-d7/a0-a3, -(sp)
27:
28:     movea.l   ARGPTR(a6), a1      *a1 = 引数列
29:     movem.w   (a1), d0-d3      *d0~d3に座標を取り出す
30:
31:     jsr      gcliprect      *クリッピングする
32:     bmi      done      *N=1なら描画の必要なし
33:
34:     jsr      gramadr      *G-RAM上のアドレスを得る
35:
```

```

36:      sub.w  d0, d2      *d2 = 横ピクセル数-1
37:      sub.w  d1, d3      *d3 = 縦ピクセル数-1
38:      lsl.w  #3, d1
39:
40:      lea.l   dithermatrix, a2
41:      move.w  d0, d4      *d4 = 左端x座標
42: loop1: movea.l a0, a1      *a1 = ライン左端
43:      andi.w  # (MATSIZ-1)*MATSIZ, d1
44:      *d1 = ディザマトリクスの行インデックス
45:      lea.l   0(a2, d1), a3 *a3 = ディザマトリクスの行
46:
47:      swap.w  d4
48:      move.w  d2, d4      *d4 = 横ピクセル数-1
49:      swap.w  d1
50:      swap.w  d2
51: loop2: move.w  (a1), d1    *1ピクセル取り出す
52:      DERGB   d1, d5, d6, d7 *RGBに分解する
53:      lsl.w  #4, d5        *RGB512階調に変換する
54:      lsl.w  #4, d6
55:      lsl.w  #4, d7
56:
57:      andi.w  #MATSIZ-1, d0 *d0 = ディザマトリクスの列インデックス
58:      move.b  0(a3, d0), d2 *d2 = 加算する震え成分
59:      ext.w   d2
60:
61:      add.w   d2, d5
62:      add.w   d2, d6
63:      add.w   d2, d7
64:
65:      clr.w   d1          *最小輝度以上を保証
66:      MAX     d1, d5
67:      MAX     d1, d6
68:      MAX     d1, d7
69:
70:      move.w  #RGBMAX*16, d1 *最大輝度以下を保証
71:      MIN     d1, d5
72:      MIN     d1, d6
73:      MIN     d1, d7
74:
75:      lsr.w   #6, d5      *下位ビットを切り捨てる
76:      lsr.w   #6, d6
77:      lsr.w   #7, d7
78:
79:      move.w  d7, d1      *256色モードのパレットコード
80:      lsl.w   #3, d1      * に構成する
81:      or.w    d6, d1
82:      lsl.w   #3, d1
83:      or.w    d5, d1
84:
85:      move.w  d1, (a1)+    *書き込む
86:
87:      addq.w  #1, d0      *ディザマトリクスの列を進める
88:      dbra    d4, loop2   *横幅分繰り返す
89:
90:      swap.w  d1
91:      swap.w  d2
92:      swap.w  d4
93:      move.w  d4, d0      *ディザマトリクス列インデックスをリセット
94:      addq.w  #MATSIZ, d1 *ディザマトリクスの行を進める
95:      lea.l   GN8YTE(a0), a0 *すぐ下のラインへ
96:      dbra    d3, loop1   *高さ分繰り返す
97:
98: done: movem.l (sp)+, d0-d7/a0-a3
99:      unlk    a6
100:      rts
101:
102:      .end

```

ところで、先ほど示したYIQのYを求める式、

$$Y = 0.299R + 0.587G + 0.114B$$

は、人間の目にとっての明るさには緑がもっとも強く影響することを示している。256色モードにおけるパレットの割り当てでは、このいちばん重要視される緑の階調が2ビットに押し込まれているが、これを3ビットに拡張し、代わりにもっとも影響力の弱い青にしわ寄せを持っていけば、256色モードの表現力が増すと考えられる。そこで、パレット設定を変更する前提で、赤、緑を8階調、青を4階調に落とすようにリスト30を修正してみた。リスト31に変更点、あわせて、リスト32にこの独自形式のパレット初期化ルーチンを示す。実行結果を見くらべてみると、リスト30の版で見られた少々ぎすぎすした感じが薄れ、より自然な変換結果が得られることがわかんと思う。

.....
リスト31
GLTOM2.S
(リスト30に対する変更部)

```
1: #      32768色→256色 (gggrrrbb) 変換
8:      .xdef    gltom2
23: gltom2:
75:      lsr.w    #7, d5      *下位ビットを切り捨てる
76:      lsr.w    #6, d6      *
77:      lsr.w    #6, d7      *
78:
79:      move.w    d7, d1      *256色モードのパレットコード
80:      lsl.w     #3, d1      * (ただしgggrrrbb) に構成する
81:      or.w      d6, d1      *
82:      lsl.w     #2, d1      *
83:      or.w      d5, d1      *
```

.....
リスト32
GINIPLM2.S

```
1: #      パレットを初期化する (256色モード時: gggrrrbb)
2:
3:      .include    doscall.mac
4:      .include    gconst.h
5: #
6:      .xdef    ginitpalet_m2
7: #
8: BDAT4 macro    B1, B2, B3, B4
9:      .dc.w      B1*2, B2*2, B3*2, B4*2
10:     .endm
11: #
12: RDATA4 macro    R1, R2, R3, R4
13:      .dc.w      R1*64, R2*64, R3*64, R4*64
14:     .endm
15: #
16: GDATA4 macro    G1, G2, G3, G4
17:      .dc.w      G1*2048, G2*2048, G3*2048, G4*2048
18:     .endm
19: #
20:      .text
21:      .even
22: #
23: ginitpalet_m2:
24:      movem.l    d0-d5/a0-a3, -(sp)
25:
26:      lea.l      GPALET, a0
27:
28:      lea.l      gdata(pc), a3
29:      moveq.l    #8-l, d3
30: gloop: move.w    (a3)+, d5
31:
32:      lea.l      rdata(pc), a2
```

```

33:      move.w  #8-1, d2
34: rloop: move.w  (a2)+, d4
35:      or.w    d5, d4
36:
37:      lea.l    bdata(pc), a1
38:      moveq.l  #4-1, d1
39: bloop: move.w  (a1)+, d0
40:      or.w    d4, d0
41:      move.w  d0, (a0)+
42:
43:      dbra    d1, bloop
44:      dbra    d2, rloop
45:      dbra    d3, gloop
46:
47:      movem.l  (sp)+, d0-d5/a0-a3
48:      rts
49: *
50: bdata: BDAT4   00, 10, 21, 31
51: *
52: rdata: RDAT4   00, 04, 09, 13
53:      RDAT4   18, 22, 27, 31
54: *
55: gdata: GDAT4   00, 04, 09, 13
56:      GDAT4   18, 22, 27, 31
57:
58:      .end

```

ディザ法の応用例としてもう1つ、2値化を取り上げよう。リスト33だ。ディザ法を使って画像を白と黒の2色に置き換える。ディザ法自体、もともとは多階調のモノクロ画像を2値化する目的で考案されたものだ。これまでの例ではRGBの色成分ごとにディザ法を適用するかたちだったが、リスト33ではYIQに変換した後のYだけを扱っている。誤差を加えた結果、輝度がいくつ以上になったら白の点とするかは、引数で指定する。中央値は64だ（16の4倍。内部では輝度を4倍した値を使っている）。実行結果が白くなりすぎるようなら、この値をもう少し大きくしてみるとよい。

リスト33
GDITH_BIN.S

```

1: *      オーダードディザによる2値化
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6: MATSIZ equ      8      *ディザマトリクスの大きさ
7: *
8:      .xdef      gdither_bin
9:      .xref      gramadr
10:     .xref      gcliprect
11:     .xref      dithermatrix
12: *
13:     .offset 0      *gdither_binの引数構造
14: *
15: X0:   .ds.w 1      *矩形座標
16: Y0:   .ds.w 1      *
17: X1:   .ds.w 1      *
18: Y1:   .ds.w 1      *
19: T:    .ds.w 1      *しきい値（基準値64）
20: *
21:     .text
22:     .even
23: *
24: gdither_bin:
25: ARGPTR =      8

```

```

26:      link    a6, #0
27:      movem.l d0-d7/a0-a4, -(sp)
28:
29:      movea.l ARGPTR(a6), a1    *a1 = 引数列
30:      movem.w (a1), d0-d3      *d0~d3に座標を取り出す
31:
32:      jsr      gcliprect        *クリッピングする
33:      bmi      done            *N=1なら描画の必要なし
34:
35:      jsr      gramadr          *G-RAM上のアドレスを得る
36:
37:      sub.w    d0, d2           *d2 = 横ピクセル数-1
38:      sub.w    d1, d3           *d3 = 縦ピクセル数-1
39:      lsl.w    #3, d1
40:
41:      swap.w   d3               *d3 = マトリクスの重み
42:      move.w   T(a1), d3        *
43:      swap.w   d3               *
44:
45:      lea.l    dithermatrix, a2
46:      move.w   d0, d4           *d4 = 左端x座標
47: loop1: movea.l a0, a1           *a1 = ライン左端
48:      andi.w   # (MATSIZE-1) * MATSIZE, d1
49:      *d1 = ディザマトリクスの行インデックス
50:      lea.l    0(a2, d1), a3    *a3 = ディザマトリクスの行
51:
52:      swap.w   d4
53:      move.w   d2, d4           *d4 = 横ピクセル数-1
54:      swap.w   d1
55:      swap.w   d2
56:      swap.w   d3
57: loop2: move.w   (a1), d7        *1ピクセル取り出す
58:      DERGB    d7, d1, d5, d6    *RGBに分解する
59:      RGBtoY    d1, d5, d6, d7    *明るさを求める
60:      lsl.w    #2, d7           *128階調に変換
61:
62:      andi.w   #MATSIZE-1, d0    *d0 = ディザマトリクスの列インデックス
63:      move.b    0(a3, d0), d2     *d2 = 加算する震え成分
64:      ext.w     d2               *
65:
66:      add.w     d2, d7           *震え成分を加える
67:
68:      clr.w     d1               *rgb(0, 0, 0)
69:      cmp.w     d3, d7           *しきい値未満?
70:      blt       skip            *
71:      move.w     $$fffe, d1      *rgb(1, 1, 1)
72:
73: skip:  move.w   d1, (a1)+        *書き込む
74:
75:      addq.w    #1, d0           *ディザマトリクスの列を進める
76:      dbra      d4, loop2        *横幅分繰り返す
77:
78:      swap.w   d1
79:      swap.w   d2
80:      swap.w   d3
81:      swap.w   d4
82:      move.w   d4, d0           *ディザマトリクス列インデックスをリセット
83:      addq.w    #MATSIZE, d1     *ディザマトリクスの行を進める
84:      lea.l     GNBYTE(a0), a0   *すぐ下のラインへ
85:      dbra      d3, loop1        *高さ分繰り返す
86:
87: done:  movem.l (sp)+, d0-d7/a0-a4
88:      unlink   a6
89:      rts
90:
91:      .end

```

色の加減算と合成

リスト34～36に示す3本のプログラムはボックスフィルルーチンのバリエーションのようなものだ。指定色をRGBごとに加えたり（リスト34）、引いたり（リスト35）、平均をとったり（リスト36）して、得られる色に置き換えていく。色を加えるgaddcolorは、ちょうど色のついた光を当てることに相当し、減じるgsubcolorはその逆の働きをする。リスト34単体では無理だが、円形の領域に対してだけgaddcolor相当の処理を加えれば、スポットライトを当てたような表現も可能になると思う。gaddcolorにかぎらず、この章で作成したプログラムはChapter 4の多角形塗り潰しルーチンや、Chapter 5の円塗り潰しルーチンと合体させれば用途が広がる。さすがにそれは面倒だというのであれば、次章で作成するパターンのはめ込みプットルーチンを使った画像合成という道もある。つまり、とりあえずは矩形領域に対してエフェクトをかけ、保存しておいた元画像に穴を空けて合成するわけだ。

リスト34
GADDCOLOR.S

```

1: #      色の加算
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gaddcolor
7:      .xref      gramadr
8:      .xref      gcliprect
9: #
10:     .offset 0      *gaddcolorの引数構造
11: #
12: X0:   .ds.w 1      *矩形座標
13: Y0:   .ds.w 1      *
14: X1:   .ds.w 1      *
15: Y1:   .ds.w 1      *
16: COL:  .ds.w 1      *加える色
17: #
18:      .text
19:      .even
20: #
21: gaddcolor:
22: ARGPTR = 8
23:      link a6, #0
24:      movem.l d0-d7/a0-a1, -(sp)
25:
26:      movea.l ARGPTR(a6), a1      *a1 = 引数列
27:      movem.w (a1), d0-d3      *d0～d3に座標を取り出す
28:
29:      jsr      gcliprect      *クリッピングする
30:      bmi      done      *N=1なら描画の必要なし
31:
32:      jsr      gramadr      *左上のG-RAM上のアドレスを得る
33:
34:      sub.w d0, d2      *d2 = 横ピクセル数-1
35:      sub.w d1, d3      *d3 = 縦ピクセル数-1
36:
37:      move.w #GNBYTE-2, d1      *d1 = ライン間のアドレスの差
38:      sub.w d2, d1      * (右端から下のラインの左端まで)
39:      sub.w d2, d1      *
40:
41:      move.w COL(a1), d4      *d0 = 加える色
42:      DERGB d4, d5, d6, d7      *
43:
44: loop1: move.w d2, d4      *d4 = 横ピクセル数-1
45:      swap.w d1      *レジスタが足りないから

```

```
46:      swap.w d2      # データレジスタの
47:      swap.w d3      # 上位ワードも使う
48: loop2: move.w (a0), d0 # 色コードを取り出し
49:      DERGB d0, d1, d2, d3 # RGBごとに
50:      add.w d5, d1      # 色成分を加える
51:      add.w d6, d2      #
52:      add.w d7, d3      #
53:
54:      moveq.l #RGBMAX, d0 # 最大輝度以下を保証
55:      MIN d0, d1      #
56:      MIN d0, d2      #
57:      MIN d0, d3      #
58:
59:      RGB d1, d2, d3, d0 # カラーコードに再構成
60:      move.w d0, (a0) + # 1ピクセル書き込む
61:      dbra d4, loop2    # 横幅分繰り返す
62:
63:      swap.w d1
64:      swap.w d2
65:      swap.w d3
66:      adda.w d1, a0      # すぐ下のラインへ
67:      dbra d3, loop1    # 高さ分繰り返す
68:
69: done: movem.l (sp)+, d0-d7/a0-a1
70:      unlk a6
71:      rts
72:
73:      .end
```

.....
リスト35
GSUBCOLOR.S
(リスト34に対する追加
・変更部)

```
1: #      色の減算
6:      .xdef gsubcolor
21: gsubcolor:
48: loop2: move.w (a0), d0      # 色コードを取り出し
49:      DERGB d0, d1, d2, d3    # RGBごとに
50:      sub.w d5, d1            # 色成分を減じる
51:      sub.w d6, d2            #
52:      sub.w d7, d3            #
53:
54:      moveq.l #0, d0          # 最小輝度以上を保証
55:      MAX d0, d1              #
56:      MAX d0, d2              #
57:      MAX d0, d3              #
58:
59:      RGB d1, d2, d3, d0      # カラーコードに再構成
60:      move.w d0, (a0) +      # 1ピクセル書き込む
61:      dbra d4, loop2          # 横幅分繰り返す
62:
63:      swap.w d1
64:      swap.w d2
65:      swap.w d3
66:      adda.w d1, a0          # すぐ下のラインへ
67:      dbra d3, loop1          # 高さ分繰り返す
68:
69: done: movem.l (sp)+, d0-d7/a0-a1
70:      unlk a6
71:      rts
72:
73:      .end
```

リスト36
GMIXCOLOR.S
(リスト34に対する追加
・変更部)

```

1: *      色の混ぜ合わせ
6:      .xdef    gmixcolor

21: gmixcolor:
48: loop2:  move.w  (a0), d0      *色コードを取り出し
49:          DERGB  d0, d1, d2, d3 * RGBごとに
50:          MEAN   d5, d1      * 色成分の平均を取る
51:          MEAN   d6, d2      *
52:          MEAN   d7, d3      *
53:
54:          RGB     d1, d2, d3, d0 *カラーコードに再構成
55:          move.w  d0, (a0) +    * 1ピクセル書き込む
56:          dbra    d4, loop2     * 横幅分繰り返す
57:
58:          swap.w  d1
59:          swap.w  d2
60:          swap.w  d3
61:          adda.w  d1, a0        * すぐ下のラインへ
62:          dbra    d3, loop1     * 高さ分繰り返す
63:
64: done:   movem.l  (sp) +, d0-d7/a0-a1
65:          unlk    a6
66:          rts
67:
68:      .end

```

色の平均をとる gmixcolor は、画像のコントラストを下げる目的で利用できるだろう。とくに、0 と平均を取れば、画像全体の輝度を半減することができる。もっとも、そのくらいの操作であれば、専用ルーチンを使ったほうが効率がよい。というわけで、リスト37に画像の輝度を半減する ghalftone も用意しておく。ghalftone では RGB に分解してから輝度を半減するのではなく、カラーコードのまま2で割って、切り捨てられるはずの分をマスクしているのがポイントだ (44～45行)。

リスト37
GHALFTONE.S

```

1: *      輝度半減
2:
3:      .include    gconst.h
4:      .include    gmacro.h
5: *
6:      .xdef    ghalftone
7:      .xref    gramadr
8:      .xref    gcliprect
9: *
10:     .offset 0      *ghalftoneの引数構造
11: *
12: X0:   .ds.w  1      *矩形座標
13: Y0:   .ds.w  1      *
14: X1:   .ds.w  1      *
15: Y1:   .ds.w  1      *
16: *
17:     .text
18:     .even
19: *
20: ghalftone:
21: ARGPTR = 8
22:     link    a6, #0
23:     movem.l  d0-d5/a0-a1, -(sp)
24:
25:     movea.l  ARGPTR(a6), a1      *a1 = 引数列

```

```

26:      movem.w (a1), d0-d3      *d0~d3に座標を取り出す
27:
28:      jsr      gcliprect      *クリッピングする
29:      bmi      done          *N=1なら描画の必要なし
30:
31:      jsr      gramadr        *左上のG-RAM上のアドレスを得る
32:
33:      sub.w    d0, d2          *d2 = 横ピクセル数-1
34:      sub.w    d1, d3          *d3 = 縦ピクセル数-1
35:
36:      move.w   #GNBYTE-2, d1  *d1 = ライン間のアドレスの差
37:      sub.w    d2, d1          * (右端から下のラインの左端まで)
38:      sub.w    d2, d1          *
39:
40:      move.w   #%01111_01111_01111_0, d5
41:      *マスクデータ
42: loop1: move.w   d2, d4          *d4 = 横ピクセル数-1
43: loop2: move.w   (a0), d0        *色コードを取り出し
44:      lsr.w    #1, d0          * 全体を2で割り
45:      and.w    d5, d0          * つじつまを合わせる
46:      move.w   d0, (a0)+        *1ピクセル書き込む
47:      dbra     d4, loop2        *横幅分繰り返す
48:
49:      adda.w   d1, a0           *すぐ下のラインへ
50:      dbra     d3, loop1        *高さ分繰り返す
51:
52: done:  movem.l (sp)+, d0-d5/a0-a1
53:      unlk     a6
54:      rts
55:
56:      .end

```

グラデーション

色をテーマとするからにはグラデーション (gradation) を取り上げないわけにはいかないだろう。この章最後のプログラムとして、矩形領域中の指定の1色の部分を任意のグラデーションパターンに変換するサブルーチンを作成してみる。

かんたんのため、黒 (輝度0) から白 (輝度31) へと明るさを変化させながら、無彩色のグラデーションで水平線分を描くことを考える。この処理は、描画を進める過程で、輝度を線分の長さに応じたステップで0から31へとなだらかに変化させることで実現される。これは、横軸にx座標、縦軸に輝度をとった座標系上に線分を描く操作に置き換えられる。つまり、輝度の補間には線分描画アルゴリズムが応用できるということだ。いま話を無彩色にかぎったが、同じ操作をRGB個別に平行して実行すればカラーにも容易に対応できる。

ただ、人間の目は輝度変化に必要な以上に敏感で、明るさの変化を実際よりも強調して知覚してしまうという性質がある。32階調程度のグラデーションでは、どんなになだらかに輝度を変化させたつもりでも、人間の目は誤魔化し切れず、マッハバンド (Mach band) と呼ばれる縞模様が発生する。高品質のグラデーションを実現するためには、このマッハバンドを消すための対策が必要になる。その目的では、ディザ法、あるいは、誤差分散法が利用できる。内部では実際よりも高精度でグラデーションの輝度計算をし、描画時にこれらアルゴリズムを使って32階調に落とすわけだ。

リスト38にグラデーションによる矩形塗り潰しルーチンを示す。リスト38は2色ではなく、矩形の4頂点に置かれた4色を補間するようにつくってある。リスト38の56～59行にも示してあるように、4色は、左上、左下、右上、右下の順に指定する。左側2色、右側2色を同じ色にすれば横方向にだけ色が変化するグラデーションになるし、上側2色、下側2色を共通にすれば縦のグラデーションになる。うまく4色を調節すれば、斜めのグラデーションも実現できるだろう。

リスト38
GGRADFILL.S

```

1: *      グラデーションによる矩形の塗り潰し
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6:      .xdef      ggradfill
7:      .xref      gramadr
8:      .xref      gcliprect
9:
10: *
11: *      誤差の累積を考慮して色を決定するマクロ (1色分)
12: *
13: PROC1 macro COL
14:      move.w COL, d1      *d1 = 現在の8192階調の色
15:      add.w (a4), d1      * + 周囲からの誤差
16:      ror.w #8, d1      *d1.b = 32階調の色
17:      or.b d1, d0      *カラーコードに構成していく
18:      clr.b d1      *d1.w = 8192階調から32階調に
19:      rol.w #8, d1      * 落とすときの切り捨て分
20:      move.w #8-1, d2
21:      and.w d1, d2      *d2 = 誤差 mod 8
22:      lsr.w #3, d1      *d1 = 誤差/8
23:      add.w d1, d2      *
24:      add.w d2, (a5)+      *真下に誤差の1/8+端数
25:      move.w d1, 6-2(a5)      *右下に誤差の1/8
26:      add.w d1, d1      *
27:      add.w d1, 6-2(a5)      *左下に誤差の2/8
28:      add.w d1, d1      *
29:      add.w d1, 6-2(a4)      *右に誤差の4/8
30:      .endm
31:
32: *
33: *      Bresenhamのアルゴリズムによりつぎの色を求めるマクロ
34: *
35: PROC2 macro COL, P
36:      local loop, skip
37:      lea.l P(a6), a0      *a0 = パラメータ列 (PARBUF)
38:      move.w (a0), d0      *d0 = 誤差項
39:      add.w 0(a0), d0      *誤差項に増分を加える
40:      ble skip      *
41:      move.w S(a0), d1      *d1 = 色の増加方向
42:      move.w HC(a6), d2      *d2 = 誤差項の補正值
43: loop: add.w d1, COL      *色を更新する
44:      sub.w d2, d0      *誤差項を補正する
45:      bgt loop      *誤差項が0以下になるまで繰り返す
46: skip: move.w d0, (a0)      *誤差項を更新する
47:      .endm
48: *
49:      .offset 0      *ggradfillの引数構造
50: *
51: X0:      .ds.w 1      *描画先座標
52: Y0:      .ds.w 1      *
53: X1:      .ds.w 1      *
54: Y1:      .ds.w 1      *
55: *COL:      .ds.w 1      *置換対象の色
56: COL1:      .ds.w 1      *左上隅の色
57: COL2:      .ds.w 1      *左下隅の色

```



```

58: COL3: .ds.w 1      *右上隅の色
59: COL4: .ds.w 1      *右下隅の色
60: BUFF: .ds.l 1      *作業用メモリ (最大12Kバイト強)
61: *
62: .offset 0          *2色のrgb展開結果
63: *
64: BBO: .ds.b 1       *色1 (3)
65: RRO: .ds.b 1       *
66: GGO: .ds.b 1       *
67: BBI: .ds.b 1       *色2 (4)
68: RRI: .ds.b 1       *
69: GGI: .ds.b 1       *
70: RGB2BUF:
71: *
72: .offset 0          *水平方向補間時のBresenhamパラメータ
73: *
74: E: .ds.w 1          *誤差項
75: D: .ds.w 1          *増分
76: S: .ds.w 1          *符号
77: PARBUF:
78: *
79: .offset -RGB2BUF*2-2*4-PARBUF*3-2*2
80: *スタックフレーム
81: *
82: WORKTOP:
83: C12: .ds.b RGB2BUF *色1, 2のrgb分解結果
84: C34: .ds.b RGB2BUF *色3, 4のrgb分解結果
85: HE: .ds.w 1          *水平方向補間時の誤差項初期値
86: HC: .ds.w 1          *水平方向補間時の誤差項補正值
87: VE: .ds.w 1          *垂直方向補間時の誤差項初期値
88: VC: .ds.w 1          *垂直方向補間時の誤差項補正值
89: BBUF: .ds.b PARBUF *青水平方向補間用Bresenhamパラメータ
90: RBUF: .ds.b PARBUF *赤水平方向補間用Bresenhamパラメータ
91: GBUF: .ds.b PARBUF *緑水平方向補間用Bresenhamパラメータ
92: ROFS: .ds.w 1        *赤の垂直方向補間データへのオフセット
93: GOFS: .ds.w 1        *赤の垂直方向補間データへのオフセット
94: A6: .ds.l 1          *±0
95: PC: .ds.l 1          *
96: ARGPTR: .ds.l 1      *引数列
97: *
98: .text
99: .even
100: *
101: ggradfill:
102: link a6, #WORKTOP
103: movem.l d0-d7/a0-a5, -(sp)
104:
105: movea.l ARGPTR(a6), a1 *a1 = 引数列
106: movem.w (a1)+, d0-d3 *d0~d3に座標を取り出す
107:
108: jsr gcliprect          *クリッピングする
109: bmi done               *N=1なら描画の必要なし
110:
111: jsr gramadr            *左上のG-RAM上のアドレスを得る
112:
113: sub.w d0, d2            *d2 = 横ピクセル数-1
114: sub.w d1, d3            *d3 = 縦ピクセル数-1
115:
116: * swap.w d3            *d3 = 置換対象の色
117: * move.w (a1)+, d3    *
118: * swap.w d3          *
119:
120: lea.l C12(a6), a3      *4色をrgbに分解しておく
121: moveq.l #4-l, d7       *
122: iloop: move.w (a1)+, d0 *
123: DERGB d0, d4, d5, d6   *
124: move.b d4, (a3)+       *
125: move.b d5, (a3)+       *
126: move.b d6, (a3)+       *
127: dbra d7, iloop         *
128:
129: move.w d2, d4          *水平/垂直の色補間の

```

```

130:      move.w d2, d5      * 誤差項/補正値を求める
131:      move.w d3, d6      *
132:      move.w d3, d7      *
133:      neg.w d4            *
134:      add.w d5, d5        *
135:      neg.w d6            *
136:      add.w d7, d7        *
137:      move.w d4, HE(a6)   *
138:      move.w d5, HC(a6)   *
139:      move.w d6, VE(a6)   *
140:      move.w d7, VC(a6)   *
141:
142:      addq.w #2, d7        *d7 = 垂直補間したテーブル1色分サイズ
143:      move.w d7, ROFS(a6) *青のテーブルと赤のテーブルの差
144:      add.w d7, d7        *
145:      move.w d7, GOFS(a6) *青のテーブルと緑のテーブルの差
146:
147:      movea.l (a1), a1     *a1 = 作業用領域
148:
149:      lea.l C12(a6), a4    *COL1~COL2を補間
150:      movea.l a1, a2       *
151:      bsr genvtbl         *
152:
153:      lea.l C34(a6), a4    *COL3~COL4を補間
154:      movea.l a1, a3       *
155:      bsr genvtbl         *
156:
157:      lea.l 6(a1), a4      *a4 = 誤差格納用バッファ1
158:
159:      moveq.l #0, d0       *誤差格納用バッファをクリア
160:      move.w d2, d4        *
161:      addq.w #2, d4        *
162: loop:  move.l d0, (a1)+    *
163:      move.w d0, (a1)+    *
164:      dbra d4, cloop      *
165:
166:      movea.l a1, a5       *a5 = 誤差格納用バッファ2
167: loop1: moveq.l #0, d0     *誤差格納用バッファの先頭をクリア
168:      move.l d0, (a5)      *
169:      move.w d0, 4(a5)     *
170:
171:      move.w HE(a6), d1     *d1 = 水平方向色補間時の誤差項初期値
172:
173:      move.w GOFS(a6), d0   *
174:      move.w 0(a2, d0), d4  *d4 = g0
175:      move.w 0(a3, d0), d5  *d5 = g1
176:      lea.l GBUF(a6), a1    *
177:      bsr hinit            *
178:      move.w d4, d7        *d7 = g0
179:
180:      move.w ROFS(a6), d0   *
181:      move.w 0(a2, d0), d4  *d4 = r0
182:      move.w 0(a3, d0), d5  *d5 = r1
183:      lea.l RBUF(a6), a1    *
184:      bsr hinit            *
185:      move.w d4, d6        *d6 = r0
186:
187:      move.w (a2)+, d4      *d4 = b0
188:      move.w (a3)+, d5      *d5 = b1
189:      lea.l BBUF(a6), a1    *
190:      bsr hinit            *
191:      move.w d4, d5        *d5 = b0
192:
193:      movea.l a0, a1        *a1 = 描画先ライン左端
194:
195:      move.w d2, d4        *d4 = 矩形の幅-1 = ループカウンタ
196:      swap.w d1
197:      swap.w d2
198:      swap.w d3
199:      movem.l a0/a4-a5, -(sp)
200: loop2: moveq.l #0, d0      *誤差の累積を考慮して描画色を決める
201:      PROC1 d7             *g

```

```

202:      lsl.w  #5, d0      *
203:      PROC1 d6          *r
204:      lsl.w  #5, d0      *
205:      PROC1 d5          *b
206:      add.w  d0, d0      *
207:
208: *      cmp.w  (a1), d3    *置換対象色?
209: *      beq   skip        *
210: *      move.w (a1), d0    *ダミー
211: skip: move.w d0, (a1)+   *1ピクセル描画
212:
213:      PROC2 d5, BBUF      *色を更新する
214:      PROC2 d6, RBUF      *
215:      PROC2 d7, GBUF      *
216:
217:      dbra   d4, loop2     *横幅分繰り返す
218:      movem.l (sp)+, a0/a4-a5
219:      swap.w d1
220:      swap.w d2
221: *      swap.w d3
222:
223:      exg.l  a4, a5        *誤差格納用バッファを交換する
224:      lea.l  GNBYTE(a0), a0 *つぎのラインへ
225:      dbra   d3, loop1     *高さ分繰り返す
226:
227: done: movem.l (sp)+, d0-d7/a0-a5
228:      unlk   a6
229:      rts
230:
231: *
232: *      垂直方向に色を補間してテーブルを作成する
233: *
234: genvtbl:
235:      move.b B80(a4), d0    *青成分を補間
236:      move.b B81(a4), d1    *
237:      bsr    gsub           *
238:
239:      move.b R80(a4), d0    *赤成分を補間
240:      move.b R81(a4), d1    *
241:      bsr    gsub           *
242:
243:      move.b G80(a4), d0    *緑成分を補間
244:      move.b G81(a4), d1    *
245: *      bsr    gsub
246: *
247: gsub:  lsl.w  #8, d0      *精度確保のため256倍スケーリング
248:      lsl.w  #8, d1      * (32階調→8192階調)
249:
250:      sub.w  d0, d1        *Bresenhamのアルゴリズムのための初期化
251:      move.w d1, d4        *
252:      ABS    d1            *
253:      SGN    d4            *
254:      add.w  d1, d1        *
255:      move.w VE(a6), d5    *
256:      move.w VC(a6), d6    *
257:      bne    fix0         *矩形の高さが1ピクセルしかなかった場合の
258:      moveq.l #0, d1       * つじつま合わせ
259:
260: fix0:  move.w d3, d7      *d7 = 矩形の高さ-1 = ループカウンタ
261: gloop1: move.w d0, (a1)+   *Bresenhamのアルゴリズムで色を補間する
262:      add.w  d1, d5        *
263:      ble    gskip        *
264: gloop2: add.w d4, d0      *
265:      sub.w  d6, d5        *
266:      bgt    gloop2       *
267: gskip: dbra   d7, gloop1  *
268:      rts
269:
270: *
271: *      水平方向の色補間のための初期化
272: *
273: hinit: sub.w  d4, d5

```

```

274:      move.w  d5, d0
275:      ABS     d5
276:      SGN     d0
277:      add.w   d5, d5
278:      move.w  d1, (a1)+      *誤差項初期値
279:      bne     fix1
280:      moveq.l  #0, d5
281: fix1:      move.w  d5, (a1)+      *誤差項増分
282:      move.w  d0, (a1)+      *符号
283:      rts
284:
285:      .end

```

リスト38では、まず、左上と左下の色、右上と右下の色を補間して中間形の1次元のグラデーションパターンを2つ作成し、両者間を補間して2次元のグラデーションを実現している。2度の補間を行うためには十分な精度が必要なため、内部では32階調ではなく256倍した8192階調で計算するようにした。なお、中間形を作成するメモリは呼び出し側が確保し、引数で渡すようになっている。このメモリ領域は512×512ピクセルの範囲を塗り潰す場合で12Kバイト強必要だ。

内部8192階調で計算した輝度レベルを32階調に落とす処理には誤差分散法を使った。ディザ法でもよかったのだが、ディザ法の実例はすでにいくつか示したし、比較的規則正しいパターンになりがちなオーダードディザよりも、不規則さがある誤差分散法のほうがグラデーションには適しているように思い、こちらを用いた。誤差(階調を落とす際に切り捨てられる端数)の散らし方としては、栗野雅彦氏の方法を採用している。この方法では、誤差の1/2を右隣、1/4を右下、1/8を左下、残り(1/8と8で割ったときのあまり)を真下のピクセルに散らす。処理が左上から右下に進むことを考慮し、2度手間にならないよう、未処理のピクセルにだけ誤差を散らしていることに注目しよう。

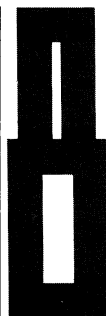
リスト38では誤差を覚えておくバッファを2本用意している(先の12Kバイト強という数字にはこのバッファの分も含まれている)。1本は現在処理中の水平ラインに実際に足し込む誤差、もう1本は真下のラインに送る誤差を格納する目的で使われる。2本のバッファは1ライン処理がすむごとに交換される。バッファには、描く矩形の横幅分よりも2ピクセル分余計に使う。これは、斜め下に誤差を送る都合だ。きっかり横幅分のバッファしか確保しないと、各ラインの1番左のピクセルから左下、あるいは、1番右のピクセルから右下に誤差を送るときに、バッファ外のメモリに書き込んでしまうことになる。左右端のピクセルを特別扱いする手もあるが、バッファの前後に1ピクセル分ずつの予備があれば余計な判定は不要だ。なお、リスト38では画面外に送られた誤差は捨ててしまっているが、1ラインの処理完了後、1ピクセル内側のピクセルに足し込むよう改造してみるのもよいだろう。

リスト38中、メインループ内でRGBそれぞれに対して同じような処理を行う部分はサブルーチンではなくマクロにして抜き出してある。13～30行で定義しているPROC1は8192階調で求めた輝度に誤差分散法により32階調に変換し、同時にいま出た誤差を周囲のピクセルに散らしている。誤差を散らす部分(20～29行)では、毎回バッファをクリアしなくてもすむ

ように、微妙なことをやっている。右下へ送る誤差だけはaddで加算せずに、moveしているのがそれだ。処理順序をよく考えてみてほしい。ちなみに、PROC1中に出てくる魔法の数字6は、誤差を溜め込むバッファ1ピクセル分（RGB各2バイト）の大きさを、2を足したり引いたりしているのはポストインクリメントしている分の補正だ。記号定数に定義してないのは、きっと僕が疲れていたからなんだろう。

35～47行のPROC2のほうは、Bresenhamのアルゴリズムで次のピクセルの輝度を算出する。本来、Bresenhamのアルゴリズムでは線分の傾きにおうじて2つの変数のどちらについてループするか決めるわけだが、ここでは無条件に座標から輝度を求めるかたちになければならないので、それなりのつじつまあわせを行っている。この点については、Chapter9の拡大・縮小ルーチンを参照してもらいたい。

さて、こうしてグラデーションによるボックスフィルルーチンができてしまえば、矩形領域内の1色をグラデーションパターンに置き換える処理が実現されたも同然だ。リスト38では無条件に点を打っている部分を、そのピクセルが指定色であるときにだけ点を打つように修正すればよい。後は引数周りの微調整だ。リスト38にはあらかじめ、修正点をコメントのかたちで入れてある。55, 116～118, 198, 208～210, 221行の先頭の“*”を外したうえで、6行, 101行のサブルーチン名を適当に (ggradreplaceとか) 変更してもらいたい。



グラフィックパターンの扱い

リスト1
GGET.S

```

1: *      G-RAMからグラフィックパターンを取り込む
2:
3:      .include      gconst.h
4: *
5:      .xdef      gget
6:      .xref      gramadr
7:      .xref      gcliprect
8:      .xref      gcopyline_L
9: *
10:     .offset 0      *ggetの引数構造
11: *
12: X0:     .ds.w      1      *取り込む矩形座標
13: Y0:     .ds.w      1      *
14: X1:     .ds.w      1      *
15: Y1:     .ds.w      1      *
16: BUF:    .ds.l      1      *パターン取り込み用バッファ
17: *
18:     .text
19:     .even
20: *
21: gget:
22: ARGPTR = 8
23:     link      a6, #0
24:     movem.l   d0-d3/a0-a3, -(sp)
25:
26:     movea.l   ARGPTR(a6), a1      *a1 = 引数列
27:     movem.w   (a1), d0-d3        *d0~d3に座標を取り出す
28:
29:     jsr       gcliprect          *クリッピングする
30:     bmi       done              *N=1なら転送の必要なし
31:
32:     jsr       gramadr            *左上のG-RAM上のアドレスを得る
33:
34:     sub.w     d0, d2              *d2 = 横ピクセル数-1
35:     sub.w     d1, d3              *d3 = 縦ピクセル数-1
36:
37:     movea.l   BUF(a1), a1        *a1 = 転送先
38:     exg.l     a0, a1             *a0 = 転送先
39:                                     *a1 = G-RAMアドレス
40:
41:     addq.w    #1, d2              *d2 = 横ピクセル数
42:     lea.l     next(pc), a2        *
43:     bclr.l    #0, d2              *横ピクセル数は奇数か?
44:     beq       skip              *
45:     lea.l     odd(pc), a2         *奇数ピクセルのとき
46:
47: skip:    lea.l   gcopyline_L, a3   *a3 = 1ライン分の転送ルーチン
48:     suba.w    d2, a3              *
49:
50:     move.w    #GNBYTE, d1         *d1 = ライン間のアドレスの差
51:     add.w     d2, d2              *
52:     sub.w     d2, d1              *
53:
54: loop:    jsr      (a3)              *1ライン転送
55:     jmp       (a2)                  *
56: odd:     move.w   (a1), (a0) +      *奇数ピクセルの場合
57: next:    adda.w   d1, a1             *すぐ下のラインへ
58:     dbra      d3, loop             *高さ分繰り返す
59:
60: done:    movem.l   (sp)+, d0-d3/a0-a3
61:     unlk      a6
62:     rts
63:
64:     .end

```

がちよつとあって、54～58行のループの中で左から右、上から下に向かって領域内の各点の色を拾ってはバッファにずらずらと書き並べている。横1ライン分をバッファに転送する処理は通常ならx座標に関するループを組むところだが、リスト1ではループを露骨に展開して速

度を稼いでみた。展開したループの中身はリスト2に分離してあり、その実体は、

```
move.l (a1)+, (a0)+
```

が実画面のピクセル数÷2個並んだものだ。ロングワードでmoveすることで、一度に2ピクセル分ずつ G-RAM からバッファに転送している。取り込む領域の横幅は偶数ピクセルとはかぎらないから、奇数ピクセルの場合は56行で端数の1ピクセルを処理する。

.....
リスト2
GCOPYL_LS

```
1:      .include      gconst.h
2:  #
3:      .xdef      gcopyline_L
4:  #
5:      .dcb.w      GNPIXEL/2,$20d9 +move.l (a1)+, (a0)+
6: gcopyline_L:
7:      rts
8:
9:      .end
```

ところで、リスト1は、256色モードや16色モードで使うにはメモリ効率が悪い。本来、X-BASICのget () 関数 (結局はIOCSコールGRMGET) のように、256色モード時は1ピクセル/1バイト、16色モード時は2ピクセル/1バイトとなるようにデータをパックする (詰める) べきなのだが、それを怠ってつねに1ピクセル/1ワードのまま取り込んでいるのだった。余計な処理をしない分、実行速度は速いとはいえ、あまり褒められた仕様とはいえないだろう。ここではこのまま押し通すが、ゆとりのある読者は、256色モードや16色モード用の別版をつくることを検討してみしてほしい。

プット:パターンの書き込み

続いてプット、メインメモリからグラフィック画面へのパターン転送だ。基本的にはゲットルーチンの転送方向を逆にすればよいわけで、その線でいくと、リスト1の38行を削除、56～57行を、

```
odd:   move.w (a1)+, (a0)
```

```
next:  adda.w d1,a0
```

に修正すればプットルーチンのできあがりとなる。だが、気付いていると思うが、リスト1は必要最小限のクリッピングしかしておらず、取り込む領域がウィンドウの端にかかる場合の動作に不自然さが残っている。ウィンドウの端を跨ぐ領域を取り込むことはあまりないだろうと勝手に決めて手抜きをしたのだが、プットルーチンではそうもいってられない。安直な流用はあきらめて、真面目にクリッピングする方向で考えてみよう。図2を見てほしい。4×4ピクセルのパターンを画面のいろいろなところに書き込む例を8通りあげてある。右側はパターンデータのメモリ上の姿で、網が掛かっている部分はクリッピングウィンドウの外にあたる

部分だ。この網の部分飛ばし、それ以外の必要な部分のみをG-RAMに書き込むようにしたい。

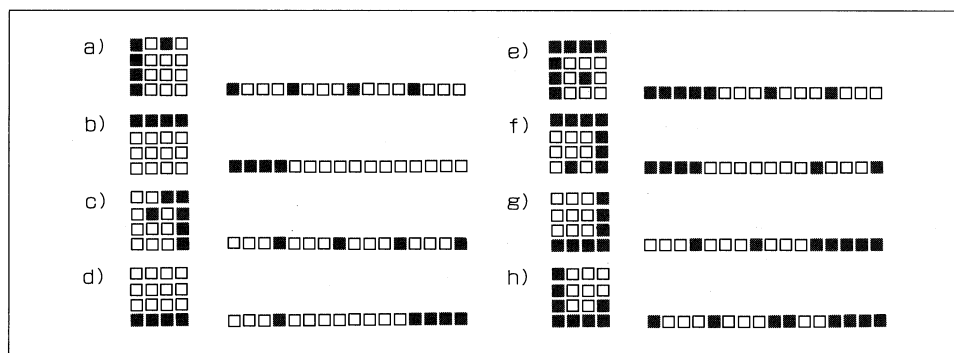


図2
パターン描画時のクリッピング

比較的単純なのは、転送先領域がウィンドウの上や下にはみ出している場合だろう。とくに、下にはみ出している場合はパターンデータの後半を無視するだけで済む。y座標についてのループカウンタを適当に小さくするだけだ。上にはみ出している場合もパターンデータの前半をスキップすれば、後は普通に転送できる。スキップする量は、

パターンの横幅 × 上にはみ出したピクセル数

となる（単位はワード）。

左右にはみ出している場合は、パターンデータを飛びとびに使うことになるので少々ややこしそうだが、落ち着いて、横1ライン分だけを取り出して考えればそう難しい話ではない。右にはみ出ているのなら、1ライン分のパターンデータ（のうち実画面の収まる範囲）を転送した“直後に”

右にはみ出したピクセル数

分パターンをスキップし、左にはみ出した場合は、1ライン分の転送に“先立って”

左にはみ出したピクセル数

分パターンを飛ばせばよい。ここで、左にはみ出る場合は図3のように変形して考えると、最初に1回余分にスキップしておけば、後は右にはみ出た場合と同じ処理ですむことがわかる。

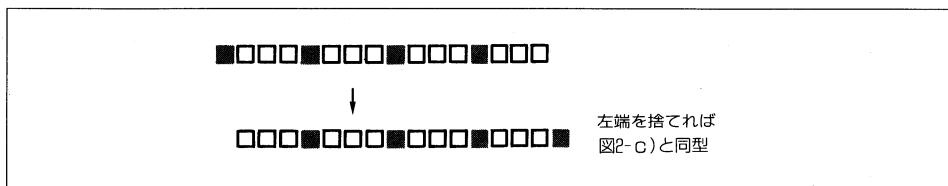


図3
図2-a)の変形

また、転送先領域がウィンドウの角にかかる場合や、パターンの横幅がウィンドウの幅より

も広くて両端がはみ出るような場合には、これまでの話の組みあわせて対処できる。

まとめると、次のようになる。

- 1) とにかく転送先領域をクリッピングする
- 2) クリッピングの結果、ウィンドウの上にはみ出したピクセル数（はみ出さなかったら0ピクセルと考える）分パターンデータをスキップする
- 3) クリッピングの結果、ウィンドウの左にはみ出したピクセル数（はみ出さなかったら0ピクセルと考える）分パターンデータをスキップする
- 4) ここからメインループ
- 5) G-RAMに1ライン分書き込む
- 6) 左右にはみ出したピクセル数分だけパターンデータをスキップする（どれだけ飛ばすかはループに入る前に計算しておくことができる）
- 7) 5),6) を必要なだけ繰り返す

できたプログラムがリスト3だ。48～53行がクリッピングの結果、上や左にはみ出した分だけパターンデータをスキップする処理、55～57行は1ライン処理するごとにスキップするパターンデータのバイト数を求める処理、となっている。データは1ピクセルにつき2バイトという点に注意して処理を追ってみたい。

リスト3
GPUT.S

```

1: #      G-RAMへグラフィックパターンを書き込む
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gput
7:      .xref      gramadr
8:      .xref      gcliprect
9:      .xref      gcopyline_L
10: #
11:      .offset 0      #gputの引数構造
12: #
13: X0:      .ds.w      1      #描画先座標
14: Y0:      .ds.w      1      #
15: X1:      .ds.w      1      #
16: Y1:      .ds.w      1      #
17: PAT:      .ds.l      1      #パターン
18: #
19:      .text
20:      .even
21: #
22: gput:
23: ARGPTR = 8
24:      link      a6, #0
25:      movem.l    d0-d7/a0-a3, -(sp)
26:
27:      movea.l    ARGPTR(a6), a1      #a1 = 引数列
28:      movem.w    (a1), d0-d3      #d0～d3に座標を取り出す
29:
30:      jsr      gcliprect      #クリッピングする
31:      bmi      done      #N=1なら描画の必要なし
32:
33:      jsr      gramadr      #左上のG-RAM上のアドレスを得る
34:
35:      sub.w      d0, d2      #d2 = 横ピクセル数-1
36:      sub.w      d1, d3      #d3 = 縦ピクセル数-1
37:

```

```

38:      movem.w (a1), d4-d7      *d4~d7にも座標を取り出す
39:      MINMAX d4, d6           *d4≤d6を保証
40:      MINMAX d5, d7           *d5≤d7を保証
41:
42:      sub.w d4, d0             *d0 = 切り取られた左端ピクセル数
43:      sub.w d5, d1             *d1 = 切り取られた上端ピクセル数
44:
45:      sub.w d4, d6             *d6 = 元のパターンの横ピクセル数-1
46:      addq.w #1, d6            *d6 = 元のパターンの横ピクセル数
47:
48:      movea.l PAT(a1), a1      *a1 = パターンデータ
49:      add.w d0, d0             *クリッピングした分だけ
50:      adda.w d0, a1            *
51:      add.w d1, d1             *
52:      mulu.w d6, d1            *
53:      adda.l d1, a1            * パターンデータを飛ばす
54:
55:      addq.w #1, d2            *d2 = 横ピクセル数
56:      sub.w d2, d6             *d6 = スキップするピクセル数
57:      add.w d6, d6             *d6 = スキップするバイト数
58:
59:      lea.l next(pc), a2       *
60:      bclr.l #0, d2            *横ピクセル数は奇数か?
61:      beq skip                 *
62:      lea.l odd(pc), a2        *奇数ピクセルのとき
63:
64: skip: lea.l gcopyline_L, a3    *a3 = 1ライン分の転送ルーチン
65:      suba.w d2, a3            *
66:
67:      move.w #GNBYTE, d1       *d1 = ライン間のアドレスの差
68:      add.w d2, d2             *
69:      sub.w d2, d1             *
70:
71: loop: jsr (a3)                *1ライン転送
72:      jmp (a2)                 *
73: odd:  move.w (a1)+, (a0)       *奇数ピクセルの場合
74: next: adda.w d1, a0            *すぐ下のラインへ
75:      adda.w d6, a1            *クリッピングした分
76:      * パターンを飛ばす
77:      dbra d3, loop            *高さ分繰り返す
78:
79: done: movem.l (sp)+, d0-d7/a0-a3
80:      unlk a6
81:      rts
82:
83:      .end

```

プットルーチンいろいろ

ここで、プットルーチンのバリエーションを3つばかりつくってみたのでリスト4以下に示しておく。1つ目は、パターン中の特定の1色を透明色と見なすgputon (リスト4)。透明色の部分はG-RAM上の元データをそのまま残し、それ以外の部分にのみパターンを書き込む。背景のうえに人物を重ねるといった画像の重ねあわせに使えるだろう。2つ目はその逆に、G-RAM上の特定の1色を透明色とみなし、その部分にパターンをはめ込むgputin。リスト5にはリスト4からの変更点のみが示されている。

3つ目はG-RAM上の画像の色とパターンデータの色を1対1の比率で混ぜあわせ、得られた平均の色をG-RAMに書き込んでいくというもの。下地が透けてみえるような効果が得ら

れる。これもリスト6にリスト4からの変更点のみを示しておく。注釈を見てもえれば、何をやっているかはすぐわかるだろう。各点につき、パターンデータとG-RAM上のパレットコードをそれぞれRGBに分解し、RGBごとに足して2で割ってからパレットコードに再構成するという処理を繰り返しているだけだ。足して2で割る部分はGMACRO.H内で定義してあるマクロを使っている。色を混合する比率を変えた版や、比率を可変にした版をつくってみるのもおもしろいかもしれない。

.....
リスト4
GPUTON.S

```

1: #      G-RAMへグラフィックパターンを書き込む (重ね合わせ)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gputon
7:      .xref      gramadr
8:      .xref      gcliprect
9: #
10:     .offset 0      *gputonの引数構造
11: #
12: X0:    .ds.w      1      *描画先座標
13: Y0:    .ds.w      1      *
14: X1:    .ds.w      1      *
15: Y1:    .ds.w      1      *
16: PAT:   .ds.l      1      *パターン
17: COL:   .ds.w      1      *透明色
18: #
19:     .text
20:     .even
21: #
22: gputon:
23: ARGPTR = 8
24:     link      a6, #0
25:     movem.l    d0-d7/a0-a1, -(sp)
26:
27:     movea.l    ARGPTR(a6), a1      *a1 = 引数列
28:     movem.w    (a1), d0-d3        *d0~d3に座標を取り出す
29:
30:     jsr      gcliprect      *クリッピングする
31:     bmi      done          *N=1なら描画の必要なし
32:
33:     jsr      gramadr        *左上のG-RAM上のアドレスを得る
34:
35:     sub.w     d0, d2        *d2 = 横ピクセル数-1
36:     sub.w     d1, d3        *d3 = 縦ピクセル数-1
37:
38:     movem.w    (a1), d4-d7      *d4~d7にも座標を取り出す
39:     MINMAX     d4, d6          *d4≤d6を保証
40:     MINMAX     d5, d7          *d5≤d7を保証
41:
42:     sub.w     d4, d0        *d0 = 切り取られた左端ピクセル数
43:     sub.w     d5, d1        *d1 = 切り取られた上端ピクセル数
44:
45:     sub.w     d4, d6        *d6 = 元のパターンの横ピクセル数-1
46:     addq.w    #1, d6        *d6 = 元のパターンの横ピクセル数
47:
48:     move.w     COL(a1), d5     *d5 = 透明色
49:     movea.l    PAT(a1), a1     *a1 = パターン先頭アドレス
50:     add.w     d0, d0          *クリッピングした分だけ
51:     adda.w     d0, a1          *
52:     add.w     d1, d1          *
53:     mulu.w     d6, d1          *
54:     adda.l     d1, a1          * パターンを飛ばす
55:
56:     sub.w     d2, d6          *
57:     subq.w     #1, d6          *

```

```

58:      add.w  d6, d6      *d6 = スキップするバイト数
59:
60:      move.w  #GNBYTE-2, d1  *d1 = ライン間のアドレスの差
61:      sub.w   d2, d1      *
62:      sub.w   d2, d1      *
63:
64: loop1: move.w  d2, d4      *d4 = 横ピクセル数-1
65: loop2: move.w  (a1)+, d0    *パターンを1ピクセル取り出す
66:      cmp.w   d0, d5      *透明色?
67:      bne     skip        * そうでなければ
68:      move.w  (a0), d0     *ダミー
69:
70: skip:  move.w  d0, (a0)+    *1ピクセル書き込む
71:      dbra    d4, loop2    *横幅分繰り返す
72:
73:      adda.w  d1, a0        *すぐ下のラインへ
74:      adda.w  d6, a1        *クリッピングした分
75:      dbra    d3, loop1    *パターンを飛ばす
76:      dbra    d3, loop1    *高さ分繰り返す
77:
78: done:  movem.l (sp)+, d0-d7/a0-a1
79:      unlk    a6
80:      rts
81:
82:      .end

```

リスト5
GPUTIN.S

```

1: *      G-RAMへグラフィックパターンを書き込む (はめ込み)
6:      .xdef  gputin
22: gputin:
65: loop2: move.w  (a1)+, d0    *パターンを1ピクセル取り出す
66:      move.w  (a0), d7      *d7 = 描画位置の色
67:      cmp.w   d7, d5      *透明色?
68:      bne     skip        * そうでなければそのまま
69:      move.w  d0, d7      *d7 = 描画色
70: skip:  move.w  d7, (a0)+    *1ピクセル書き込む
71:      dbra    d4, loop2    *横幅分だけ繰り返す

```

リスト6
GHPUT.S

```

1: *      G-RAMへグラフィックパターンを書き込む (半透明)
6:      .xdef  ghalftoneput
22: ghalftoneput:
48: *move.w  COL (a1), d5      *d5 = 透明色
64:      move.w  d6, d0      *d0 = スキップするバイト数
65:
66: loop1: move.w  d2, d4      *d4 = 横ピクセル数-1
67:      swap.w  d0
68:      swap.w  d1
69:      swap.w  d2
70:      swap.w  d3
71: loop2: move.w  (a1)+, d0    *パターン側から1ピクセル
72:      DERGB  d0, d1, d2, d3  * RGBに分解する
73:      move.w  (a0), d0      *画面からも1ピクセル
74:      DERGB  d0, d5, d6, d7  * RGBに分解する
75:      MEAN   d5, d1        *RGBごとに平均を求め
76:      MEAN   d6, d2        *
77:      MEAN   d7, d3        *
78:      RGB    d1, d2, d3, d0  * カラーコードに再構成して
79:      move.w  d0, (a0)+    * 書き込む
80:      dbra    d4, loop2    *横幅分繰り返す

```

```

81:
82:      swap.w d0
83:      swap.w d1
84:      swap.w d2
85:      swap.w d3
86:      adda.w d1, a0      #つぎのラインへ
87:      adda.w d0, a1      #クリッピングした分パターンを飛ばす
88:      dbra d3, loop1     #ライン数分繰り返す
89:
90: done:  movem.l (sp)+, d0-d7/a0-a1
91:      unlk a6
92:      rts
93:
94:      .end

```

ここでgputonの応用例として、GL3ファイルの重ねあわせロードプログラムGLOADON.Xを示そう。リスト7だ。前章のGLLOAD.Xなどであらかじめ画面に65536色の画像データを表示しておき、

GLOADON ファイル名.GL3

のようにして使う。

リスト7ではスタックポインタの初期化、コマンドライン引数の有無のチェック(33～34行)に続いて、DOSコールsetblockにより自分の後ろの余計なメモリを切り離している(36～40行)。これはあとでファイルを読み込むメモリをDOSコールmallocを使って確保する都合だ(→『入門編』Chapter9)。以下、IOCSコールAPAGEによりグラフィック画面が初期化されているかどうかチェックし(42～45行)、スーパーバイザモードに切り替えてから(47～48行)、ファイル読み込み用のメモリを確保する(50～63行)。

リスト7
GLOADON.S

```

1: #      画像を重ね合わせてロードする
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5:      .include      gconst.h
6: #
7:      .xref      gputon
8: #
9: CR      equ      13
10: LF      equ      10
11: #
12: ROPEN      equ      0
13: STDERR      equ      2
14: #
15: MAXMEM      equ      GNBYTE*GNPIXEL
16: MINMEM      equ      GNBYTE
17: #
18:      .offset 0      #gputonの引数構造
19: #
20: X0:      .ds.w      1      #矩形領域
21: Y0:      .ds.w      1      #
22: X1:      .ds.w      1      #
23: Y1:      .ds.w      1      #
24: PAT:      .ds.l      1      #パターン
25: COL:      .ds.l      1      #透明色
26: #
27:      .text
28:      .even
29: #
30: ent:
31:      lea.l      inisp(pc), sp

```



```

32:
33:     tst, b    (a2) +      *コマンド行引数はあるか?
34:     beq      usage      * なければ使用法を表示して終了
35:
36:     lea, l    16 (a0), a0  *プログラムの後ろの余分なメモリを切り放す
37:     suba, l   a0, a1      *
38:     movem, l  a0-a1, -(sp) *
39:     DOS      _SETBLOCK    *
40:     addq, l   #8, sp      *
41:
42:     moveq, l  #-1, d1     *グラフィック画面は初期化されているか?
43:     lOCS     _APAGE      *
44:     tst, b    d0         *
45:     bmi      ninit       *未初期化ならエラー終了
46:
47:     clr, l    -(sp)       *スーパーバイザモードへ移行
48:     DOS      _SUPER      *
49:
50:     move, l   #MAXMEM, d1  *ファイル読み込み用のメモリを確保する
51:     move, w   #GNPIXEL, d2 *
52:     moveq, l  #1, d3      *
53:     move, l   #MINMEM, d4 *
54: memlp: move, l  d1, -(sp)  *
55:     DOS      _MALLOC      *
56:     addq, l   #4, sp      *
57:     tst, l    d0         *
58:     bpl      memok       *
59:     lsr, l    #1, d1      *
60:     lsr, w    #1, d2      *
61:     add, w    d3, d3      *
62:     cmp, l    d4, d1      *
63:     bcc      memlp       *
64:     bra      nomem       *メモリ不足だった
65: *
66: memok: movea, l  d0, a0
67:     move, w   d2, d4
68:     subq, w   #1, d2
69:     subq, w   #1, d3
70:
71: *   a0 = ファイル読み込み用バッファ
72: *   a2 = ファイル名
73: *   d1 = 一度に処理するバイト数
74: *   d2 = 一度に処理するライン数-1
75: *   d3 = 分割回数-1 (=ループカウンタ)
76: *   d4 = 一度に処理するライン数
77:
78:     lea, l    argbuf(pc), a1 *gputonへの引数の初期化
79:     clr, l    X0(a1)
80:     move, w   #GNPIXEL-1, X1(a1)
81:     move, w   d2, Y1(a1)
82:     move, l   a0, PAT(a1)
83: *
84:     move, w   #ROPEN, -(sp)  *ファイルオープン
85:     pea, l    (a2)          *
86:     DOS      _OPEN          *
87:     addq, l   #6, sp        *
88:     move, w   d0, d2        *d2 = ファイルハンドル
89:     bmi      nfound        *
90: *
91:     pea, l    2, w          *ファイル先頭の2バイトを読み込み
92:     pea, l    COL(a1)       * 透明色とする
93:     move, w   d2, -(sp)     *
94:     DOS      _READ          *
95:     lea, l    T0(sp), sp    *
96:     subq, l   #2, d0        *
97:     bne      rerror        *
98: *
99:     move, w   d0, -(sp)     *ファイルを巻き戻す
100:     move, l   d0, -(sp)     *
101:     move, w   d2, -(sp)     *
102:     DOS      _SEEK          *
103:     addq, l   #8, sp        *
104: *
105: putlp: move, l  d1, -(sp)   *ファイルから読み込む

```

```

106:      pea.l    (a0)          *
107:      move.w   d2, -(sp)     *
108:      DOS      _READ         *
109:      lea.l    10(sp), sp    *
110:      tst.l    d0            *
111:      bmi      rerror        *
112:
113:      pea.l    (a1)          *描画
114:      jsr      gputon        *
115:      addq.l    #4, sp        *
116:
117:      add.w     d4, Y0(a1)    *つぎの描画位置
118:      add.w     d4, Y1(a1)    *
119:
120:      dbra     d3, putlp      *繰り返す
121:
122:      DOS      _EXIT
123: *
124: usage: lea.l    usgmes(pc), a0
125:        bra     error
126: ninit: lea.l    errms1(pc), a0
127:        bra     error
128: nomem: lea.l    errms2(pc), a0
129:        bra     error
130: nfound: lea.l   errms3(pc), a0
131:        bra     error
132: rerror: lea.l    errms4(pc), a0
133: *
134: error: move.w   #STDERR, -(sp)
135:        pea.l    (a0)
136:        DOS      _FPUTS
137:
138:        move.w   #1, -(sp)
139:        DOS      _EXIT2
140: *
141: .data
142: .even
143: *
144: usgmes: .dc.b   '機能: GL3ファイルを重ね合わせロードする', CR, LF
145:        .dc.b   '  使用法: GLOADON   ファイル名', CR, LF, 0
146: errms1: .dc.b   'GLOADON: G-RAMが初期化されていません', CR, LF, 0
147: errms2: .dc.b   'GLOADON: メモリ不足です', CR, LF, 0
148: errms3: .dc.b   'GLOADON: 指定のファイルが見つかりません', CR, LF, 0
149: errms4: .dc.b   'GLOADON: ファイルがうまく読み込めません', CR, LF, 0
150: *
151: .bss
152: .even
153: *
154: argbuf: .ds.w   4           *矩形領域
155:        .ds.l   1           *パターン
156:        .ds.w   1           *透明色
157: *
158: .stack
159: .even
160: *
161:        .ds.l   4096
162: inisp:
163:
164: .end     ent

```

さて、いま扱うファイルは512Kバイトにおよぶ画像ファイルだ。これをそっくりそのまま読み込めるだけのメモリがつねに確保できるとはかぎらない。そこで、このプログラムでは、512Kバイト確保できなかったら256Kバイト、それも無理なら128Kバイト、というように、確保できるまで要求メモリサイズを半減していくようにしてある。全部をまとめて読み込めなかった場合はファイルを等分し、ファイルの読み込みと画像の重ねあわせ処理を必要回数繰り返すわけだ。

メモリさえ確保できたら78～103行でgputonへ引き渡す引数の用意、ファイルのオープンなど、ループの前処理を行う。ファイルの先頭1ワード（画像の左上隅のパレットコード）を透明色とみなすことにしてあるので、91～97行で2バイトだけ先読みし、99～103行でファイルを巻戻している。後は、105～120行が見てのと通りのメインループだ。終了前にはスーパーバイザモードからユーザモードに戻り、確保したメモリを解放して、開いたファイルを閉じるのが礼儀だが、このプログラムは不作法で、いきなりexitしてHuman68kに片付けさせている。

矩形領域の複写

今度はG-RAM上の矩形領域を別の領域にコピーすることを考える。すでに作成したggetとgputを使えば目的を達することはできるが、あえてG-RAM上で転送することにこだわってみたい。ここでは転送元領域と転送先領域が重なっている場合の処理が重要なポイントとなる。対策を練っておこう。

図4に領域の重なり方を9通り示してある（上になったほうが転送先）。試しに、それぞれの場合についていつものように領域の左上隅から右、そして下方向にコピー処理を進めてみよう。すると、f)～i)は正しくコピーされないことがわかる。転送元データが参照前に上書きされてしまうのだ。では、f)～i)はどのような順序で処理していったらいいかというと、g)～i)に関しては領域の下ラインから転送を始めて、そして上方向に処理を進めればよい。残るf)は右端から処理を始め、左方向に転送していく。結局、重なり方におうじて、3通りの場合分けが必要ということだ。

というわけでリスト8が矩形領域コピールーチンだ。領域が重なっているかどうかは気にせ

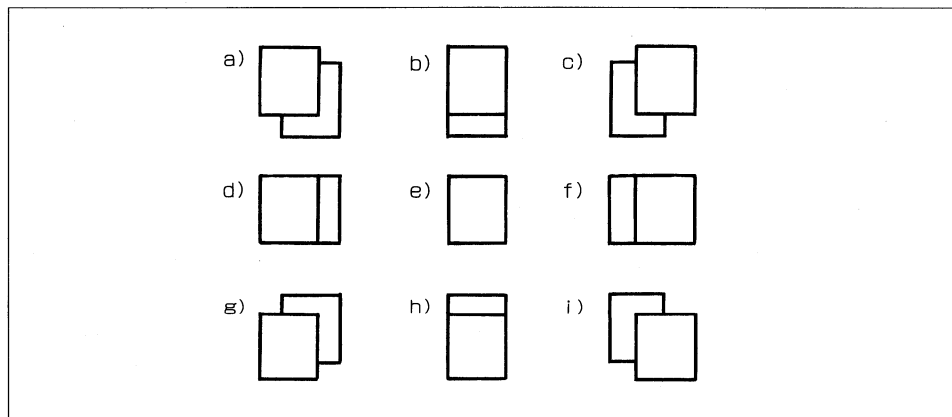


図4
転送元領域と転送先領域の重なり方

ずに、転送元領域と転送先領域の位置関係だけで処理を振り分けている（ちょっと手抜き）。図4のf) の場合は完全に特別扱いで、リスト9には専用の横1ライン転送ルーチンまでもが用意されている。そのほかのケースはうまいぐあいにまとめることができた。102～112行のあたりがそのつじつまあわせをしている部分だ。

あと、つくってみて気づいたのだが、このプログラムでは転送元領域と転送先領域の両方に気を配らなければならない関係で、クリッピング周りが少々ややこしくなっている。注釈を参考に読み切してほしい。

リスト8
GCOPY.S

```

1: #      矩形領域の複写
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gcopy
7:      .xref      gramadr
8:      .xref      gcliprect
9:      .xref      gcopyline_L
10:     .xref      gcopyline_R
11: #
12:     .offset 0      *gcopyの引数構造
13: #
14: X0:   .ds.w      1      *転送元座標
15: Y0:   .ds.w      1      *
16: X1:   .ds.w      1      *
17: Y1:   .ds.w      1      *
18: X2:   .ds.w      1      *転送先左上隅座標
19: Y2:   .ds.w      1      *
20: #
21:     .text
22:     .even
23: #
24: gcopy:
25: ARGPTR = 8
26:     link      a6, #0
27:     movem.l   d0-d7/a0-a3, -(sp)
28:
29:     movea.l   ARGPTR(a6), a1      *a1 = 引数列
30:     movem.w   (a1), d0-d3      *d0～d3に座標を取り出す
31:
32:     jsr      gcliprect      *転送元領域をクリッピングする
33:     bmi     done            *N=1なら転送の必要なし
34:
35:     jsr      gramadr        *左上のG-RAM上のアドレスを得る
36:
37:     sub.w    d0, d2          *d2 = 横ピクセル数-1
38:     sub.w    d1, d3          *d3 = 縦ピクセル数-1
39:
40:     movem.w   (a1)+, d4-d7      *(x0, y0) - (x1, y1)
41:     MINMAX    d4, d6          *d4 ≤ d6を保証
42:     MINMAX    d5, d7          *d5 ≤ d7を保証
43:
44:     *転送元領域をクリッピングしたときに
45:     sub.w     d4, d0          * d0 = 切り取られた左端ピクセル数
46:     sub.w     d5, d1          * d1 = 切り取られた上端ピクセル数
47:
48:     movem.w   (a1)+, d4-d5      *(x2, y2)
49:     add.w     d0, d4          *切り取られた分だけx2, y2もずらす
50:     add.w     d1, d5          *
51:     move.w    d4, d0          *x2'
52:     move.w    d5, d1          *y2'
53:     add.w     d4, d2          *x3'
54:     add.w     d5, d3          *y3'
55:

```

```

56:      sub.w  d2, d6      *d6 = x1' - x3' = x0' - x2'
57:      sub.w  d3, d7      *d7 = y1' - y3' = y0' - y2'
58:
59:      jsr    gcliprect    *転送先領域をクリッピングする
60:      bmi    done          *N=1なら転送の必要なし
61:
62:      movea.l a0, a1      *a1 = 転送元アドレス
63:      jsr    gramadr       *a0 = 転送先アドレス
64:
65:      sub.w  d0, d2      *d2 = 最終的な転送領域の横ピクセル数-1
66:      sub.w  d1, d3      *d3 = 最終的な転送領域の縦ピクセル数-1
67:
68:                                *転送先領域をクリッピングしたときに
69:                                * d0 = 切り取られた左端ピクセル数
70:      sub.w  d4, d0      * d1 = 切り取られた上端ピクセル数
71:      sub.w  d5, d1
72:
73:      add.w  d0, d0      *d0, d1の分だけ
74:      adda.w d0, a1      * 転送元アドレスもずらす
75:      ext.l  d1          *
76:      moveq.l #GSFCTR, d0 *
77:      lsl.l  d0, d1      *
78:      adda.l d1, a1      *
79:
79:      addq.w #1, d2      *d2 = 最終的な転送領域の横ピクセル数
80:      move.w #GNBYTE, d1 *d1 = ライン間のアドレスの差
81:
82:      tst.w  d6          *1) x0' < x2' かつ
83:      bpl    nright      *
84:      tst.w  d7          *2) y0' = y2' ならば
85:      beq    right       * 真右へのコピー
86:
87: nright:      *真右以外
88:      lea.l  next(pc), a2 *
89:      bclr.l #0, d2      *横ピクセル数は奇数か?
90:      beq    skip        *
91:      lea.l  odd(pc), a2  *奇数ピクセルのとき
92:
93: skip:      lea.l  gcopyline_L, a3 *正方向の転送
94:      suba.w  d2, a3      *
95:
96:      add.w  d2, d2      *
97:      sub.w  d2, d1      *
98:
99:      tst.w  d7          *y0' < y2'
100:     bge     down        * ならば
101:                                *下から上への転送
102: up:      add.w  d2, d1
103:      add.w  d2, d1
104:      neg.w  d1
105:
106:      moveq.l #0, d0      *上位ワードをクリア
107:      move.w  d3, d0      *d0, 1 = ライン数-1
108:      moveq.l #GSFCTR, d2 *1024 (または2048) 倍
109:      lsl.l  d2, d0      *
110:      adda.l d0, a0      *a0 = 転送先領域の
111:                                * 一番下のラインの左端アドレス
112:      adda.l d0, a1      *a1 = 転送元領域の
113:                                * 一番下のラインの左端アドレス
114:                                *つじつまは合わせたから合流
115:
116: down:      *上から下への転送
117: loop:      jsr    (a3)    *1ライン転送
118:      jmp     (a2)        *
119: odd:      move.w  (a1), (a0) *奇数ピクセルの場合
120: next:      adda.w  d1, a0    *転送先をすぐ下のラインへ
121:      adda.w  d1, a1    *転送元をすぐ下のラインへ
122:      dbra     d3, loop    *高さ分繰り返す
123:
124: done:      movem.l (sp)+, d0-d7/a0-a3
125:      unlk    a6
126:      rts
127: *
128: right:      *真右へのコピーは特別扱い
129:      move.w  d2, d0      *d0 = 横ピクセル数

```

```

130:      add.w   d0, d0          #d0 = 1ライン分のバイト数
131:      adda.w  d0, a0          #a0 = 複写先ライン右端+2
132:      adda.w  d0, a1          #a1 = 複写元ライン右端+2
133:      add.w   d0, d1
134:
135:      lea.l    rnext(pc), a2   #
136:      bclr.l   #0, d2          #横ピクセル数は奇数か?
137:      beq      rskip          #
138:      lea.l    rodd(pc), a2    #奇数ピクセルのとき
139:
140: rskip: lea.l    gcopyline_R, a3 #逆方向の転送
141:      suba.w   d2, a3          #
142:
143: rloop: jsr     (a3)           #1ライン転送
144:      jmp      (a2)           #
145: rodd:   move.w -(a1), -(a0)    #奇数ピクセルの場合
146: rnext:  adda.w d1, a0          #転送先をすぐ下のラインへ
147:      adda.w   d1, a1          #転送元をすぐ下のラインへ
148:      dbra     d3, rloop       #高さ分繰り返す
149:
150:      bra      done
151:
152:      .end

```

.....
リスト9
GCOPYL_R.S

```

1:      .include      gconst.h
2: #
3:      .xdef          gcopyline_R
4: #
5:      .dcb.w         GNPIXEL/2, $2121 #move.l -(a1), -(a0)
6: gcopyline_R:
7:      rts
8:
9:      .end

```

左右上下の反転

グラフィックパターンの扱いという話題からは少しはずれるが、次は画面上の矩形領域の左右反転、上下反転を考えてみよう。処理自体は単純なので、試しに各自つくってみてもらいたい。ほんの参考までに、リスト10に左右反転ルーチン、リスト11に上下反転ルーチンの例を示す。以下、この参考リストに軽い解説を加えておく。

左右の反転は領域の各水平ラインに対し、

- 1) 左端と右端の色を交換する
- 2) 左端から1ピクセル右によった点と右端から1ピクセル左によった点の色を交換する
- 3) 左端から2ピクセル右によった点と右端から2ピクセル左によった点の色を交換する

⋮

という処理を“すれ違いそうになるまで”繰り返せば実現できる。同様に、上下の反転は各垂直ラインに対して、

- 1) 上端と下端の色を交換する

2) 上端から1ピクセル下によった点と下端から1ピクセル上によった点の色を交換する

⋮

処理の繰り返しとなるわけだが³, G-RAMの構造上,

1) いちばん上のラインといちばん下のラインを交換する

2) 上から2番目のラインと下から2番目のラインを交換する

⋮

というように、横1ライン単位で処理したほうが楽だ。

リスト10, 11ではいつものように左上隅のG-RAM上のアドレスを求めたあと、ついでに右上隅のアドレス, または, 左下隅のアドレスを計算している部分が小さなポイントだろう。

リスト10の35~36行ではd2 (領域の横ピクセル数-1) を2倍し, それをa0 (領域左上隅のアドレス) と足すことで右上隅のアドレスを計算している。36行の,

```
lea.l 0(a0,d2.w), a2
```

の意味に注意してほしい。a0+d2.w+0で表されるアドレスをa2に代入している。リスト11では35~39行が左下隅のアドレスを求める処理だ。ここでは36行でextを使ってd0の上位ワードをクリアしているのに注目。ext.lはデータレジスタの下位ワードを符号拡張してロングワードデータにする命令だから, もとのd0.wが非負であることが保証されていれば, 結果の上位ワードはつねに0となる。

リスト10
GHREV.S

```

1: #      矩形領域の左右反転
2:
3:      .include      gconst.h
4: #
5:      .xdef      ghreverse
6:      .xref      gramadr
7:      .xref      gcliprect
8: #
9:      .offset 0      #ghreverseの引数構造
10: #
11: X0:      .ds.w      1      #矩形座標
12: Y0:      .ds.w      1      #
13: X1:      .ds.w      1      #
14: Y1:      .ds.w      1      #
15: #
16:      .text
17:      .even
18: #
19: ghreverse:
20: ARGPTR = 8
21:      link      a6, #0
22:      movem.l    d0-d3/a0-a3, -(sp)
23:
24:      movea.l    ARGPTR(a6), a1      #a1 = 引数列
25:      movem.w    (a1), d0-d3      #d0~d3に座標を取り出す
26:
27:      jsr      gcliprect      #クリッピングする
28:      bmi      done      #N=1なら描画の必要なし
29:
30:      jsr      gramadr      #左上のG-RAM上のアドレスを得る
31:
32:      sub.w      d0, d2      #d2 = 横ピクセル数-1
33:      sub.w      d1, d3      #d3 = 縦ピクセル数-1
34:
35:      add.w      d2, d2
36:      lea.l      0(a0, d2.w), a2      #a2 = ライン右端アドレス
37:      move.w      #GNBYTE, d1      #d1 = ライン間のアドレスの差

```

```

38:
39: loop1: movea.l a0, a1      *a1 = ライン左端
40:         movea.l a2, a3      *a3 = ライン右端
41:
42: loop2: move.w (a1), d0      *交換
43:         move.w (a3), (a1)+  *
44:         move.w d0, (a3)     *
45:         subq.l #2, a3       *
46:
47:         cmpa.l a3, a1        *すれ違うまで
48:         bcs    loop2         * 繰り返す
49:
50:         adda.w d1, a0         *つぎのラインへ
51:         adda.w d1, a2         *つぎのラインへ
52:         dbra   d3, loop1      *高さ分繰り返す
53:
54: done:   movem.l (sp)+, d0-d3/a0-a3
55:         unlk   a6
56:         rts
57:
58:         .end

```

リストII GVREV.S

```

1: #      矩形領域の上下反転
2:
3:         .include      gconst.h
4: #
5:         .xdef    gvreverse
6:         .xref    gramadr
7:         .xref    gcliprect
8: #
9:         .offset 0      *gvreverseの引数構造
10: #
11: X0:     .ds.w 1        *矩形領域
12: Y0:     .ds.w 1        *
13: X1:     .ds.w 1        *
14: Y1:     .ds.w 1        *
15: #
16:         .text
17:         .even
18: #
19: gvreverse:
20: ARGPTR = 8
21:         link    a6, #0
22:         movem.l d0-d4/a0-a3, -(sp)
23:
24:         movea.l ARGPTR(a6), a1      *a1 = 引数列
25:         movem.w (a1), d0-d3        *d0-d3に座標を取り出す
26:
27:         jsr     gcliprect          *クリッピングする
28:         bmi     done              *N=1なら描画の必要なし
29:
30:         jsr     gramadr            *左上のG-RAM上のアドレスを得る
31:
32:         sub.w   d0, d2              *d2 = 横ピクセル数-1
33:         sub.w   d1, d3              *d3 = 縦ピクセル数-1
34:
35:         move.w  d3, d0              *縦ピクセル数-1をGNBYTE倍する
36:         ext.l   d0                  *
37:         moveq.l #GSFCTR, d1        *
38:         asl.l   d1, d0              *
39:         lea.l   0(a0, d0, 1), a2    *a2 = 領域の左下隅アドレス
40:         lsr.w   #1, d3              *
41:
42:         move.w  #GNBYTE, d1         *d1 = ライン間のアドレスの差
43:
44: loop1:   movea.l a0, a1              *a1 = 左上
45:         movea.l a2, a3              *a3 = 左下
46:
47:         move.w  d2, d4              *d4 = 横ピクセル数-1
48: loop2:   move.w  (a1), d0            *交換

```



```

49:      move.w (a3), (a1)+      *
50:      move.w d0, (a3)+      *
51:      dbra   d4, loop2      *横幅分繰り返す
52:
53:      adda.w d1, a0          *すぐ下のラインへ
54:      suba.w d1, a2          *すぐ上のラインへ
55:      dbra   d3, loop1      *高さ/2回繰り返す
56:
57: done:  movem.l (sp)+, d0-d4/a0-a3
58:      unlk   a6
59:      rts
60:
61:      .end

```

連続複写

最後に、パターンの連続複写を取り上げる。矩形領域の上端数ピクセル分を複写して、矩形領域全体を覆う処理を考えよう。たとえば、(0, 0) - (511, 15) に何かパターンを描いておき、そのパターンで画面全体を覆う、といった使い方を想定している。疑似的に“パターンによるボックスフィル”が実現できるわけだ。これまた読者に自分でつくってみてもらいたいと思う。先ほどのgcopyで領域の重なりあいに関して検討したときの図4-f) のケースがヒントになるだろう。リスト12も参考までに。

それができたら、横方向のパターン連続複写ルーチンもつくってみよう。領域の左端数ピクセルを連続複写するわけだ。リスト13に1例を示しておく。リスト12の部分修正ですませてみた。ただし、リスト13にはバグ（というよりは実行速度をかせいだため、および、リスト12からの修正を最小限にしたための弊害）が残っている。ある特定の条件のときに、正常動作しないのだ（バスエラーが出たりはしないが）。暇なときにでも、バグ探しをしてみるとおもしろいかもしれない。

リスト12
GVDUP.S

```

1: *      パターンの連続垂直複写
2:
3:      .include      gconst.h
4: *
5:      .xdef      gvdup
6:      .xref      gramadr
7:      .xref      gcliprect
8:      .xref      gcopyline_L
9: *
10:      .offset 0      *gvdupの引数構造
11: *
12: X0:      .ds.w 1      *矩形領域
13: Y0:      .ds.w 1      *
14: X1:      .ds.w 1      *
15: Y1:      .ds.w 1      *
16: DY:      .ds.w 1      *複写する元パターンの高さ
17: *
18:      .text
19:      .even
20: *
21: gvdup:
22: ARGPTR = 8

```

```

23:      link    a6, #0
24:      movem.l d0-d4/a0-a3, -(sp)
25:
26:      movea.l ARGPTR(a6), a1    #a1 = 引数列
27:      movem.w (a1), d0-d4      #d0~d3に座標
28:                                # d4に複写するライン数を取り出す
29:
30:      jsr      gcliprect        #クリッピングする
31:      bmi     done              #N=1なら描画の必要なし
32:
33:      jsr      gramadr          #左上のG-RAM上のアドレスを得る
34:
35:      sub.w    d0, d2            #d2 = 横ピクセル数-1
36:      sub.w    d1, d3            #d3 = 縦ピクセル数-1
37:      sub.w    d4, d3            #複写するライン数より小さければ
38:      bmi     done              # なにもしくなくていい
39:
40:      movea.l  a0, a1            #a1 = 転送元
41:      ext.l    d4                #
42:      moveq.l  #GSFCTR, d0      #
43:      lsl.l    d0, d4            #
44:      adda.l   d4, a0            #a0 = 転送先
45:
46:      addq.w    #1, d2            #d2 = 横ピクセル数
47:      lea.l     next(pc), a2      #
48:      bclr.l    #0, d2            #横ピクセル数は奇数か?
49:      beq      skip              #
50:      lea.l     odd(pc), a2        #奇数ピクセルのとき
51:
52: skip:  lea.l     gcopyline_L, a3  #a3 = 1ライン分の転送ルーチン
53:      suba.w    d2, a3            #
54:
55:      move.w    #GNBYTE, d1       #d1 = ライン間のアドレスの差
56:      add.w     d2, d2            #
57:      sub.w     d2, d1            #
58:
59: loop:  jsr      (a3)              #1ライン描画
60:      jmp      (a2)
61: odd:   move.w    (a1), (a0)        #奇数ピクセルの場合
62: next:  adda.w    d1, a0            #すぐ下のラインへ
63:      adda.w    d1, a1            #すぐ下のラインへ
64:      dbra     d3, loop            #高さ分繰り返す
65:
66: done:  movem.l  (sp)+, d0-d4/a0-a3
67:      unlk     a6
68:      rts
69:
70:      .end

```

.....
リスト13
GHDUP.S

```

1: #      パターンの連続水平複写
5:      .xdef    ghdup
16: DX:  .ds.w    1                #複写する元パターンの幅
21: ghdup:
35:      sub.w    d0, d2            #d2 = 横ピクセル数-1
36:      sub.w    d4, d2            #それが複写幅より小さければ
37:      bmi     done              # なにもしくなくていい
38:      sub.w    d1, d3            #d3 = 縦ピクセル数-1
39:
40:      movea.l  a0, a1            #a1 = 転送元
41:      add.w     d4, d4            #
42:      adda.w    d4, a0            #a0 = 転送先
43: #
44: #

```

C
H
A
P
T
E
R

9

画像の拡大・縮小

画像の拡大・縮小



この章ではグラフィックパターンの拡大・縮小を取り上げる。ひとくちに拡大・縮小といっても目的におうじてプログラムのつくり方は変わってくるものだ。ゲームに使うのであれば多少画質が落ちてでも実行速度が速いほうがいいし、グラフィックエディタに組み込むのなら速度よりも変換後の画質を重視したい。本稿では質より描画速度を優先した拡大・縮小プットルーチンの作成を中心に話を進めるが、最後におまけとして、画質優先の画像縮小ルーチンの例も示すことにしよう。

拡大・縮小のアルゴリズム

グラフィックパターンの拡大・縮小は要は単純な比の問題だ。矩形領域 $(x_0, y_0) - (x_1, y_1)$ を領域 $(x_2, y_2) - (x_3, y_3)$ に写像する場合、元領域内のある点 (x, y) は、

$$x' = \frac{x_3 - x_2}{x_1 - x_0} (x - x_0) + x_2$$

$$y' = \frac{y_3 - y_2}{y_1 - y_0} (y - y_0) + y_2$$

で求められる点 (x', y') に対応する (図1)。ここで、

$$\frac{x_3 - x_2}{x_1 - x_0} = \text{横方向の拡大・縮小率}$$

$$\frac{y_3 - y_2}{y_1 - y_0} = \text{縦方向の拡大・縮小率}$$

だ。各点につき、上の式で対応する写像先の点を求めれば、任意の比率での拡大・縮小が行えるだろう。試しにこのとおりX-BASICの関数にしてみるとリスト1のようになった。

リスト1は縮小時にはうまく動いているように見える。しかし、拡大時には描画結果が“すかすか”になるという症状を示す。考えてみれば当たり前のことで、リスト1では拡大元の矩形の大きさでループしているために、拡大先の領域を埋めるだけの点の数がないのだ。そのつ

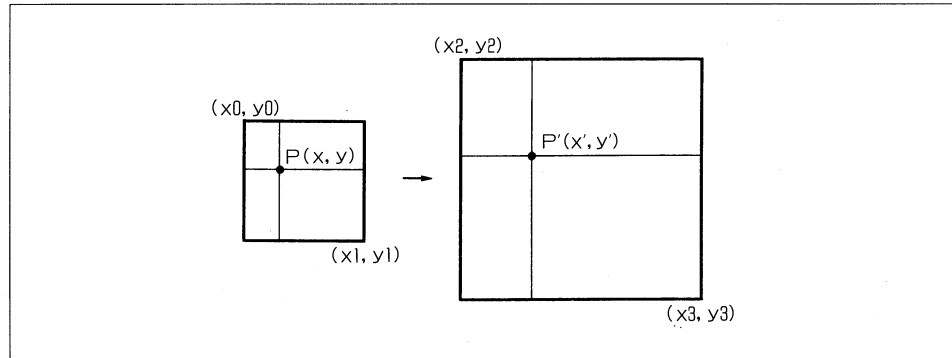


図1
拡大・縮小時の座標変換

リスト1
SCALING1.BAS

```
10 func test(x0, y0, x1, y1, x2, y2, x3, y3)
20   int x, y, xx, yy, c
30   for y=y0 to y1
40     yy=(y-y0)*(y3-y2)/(y1-y0)+y2
50     for x=x0 to x1
60       xx=(x-x0)*(x3-x2)/(x1-x0)+x2
70       c=point(x, y)
80       pset(xx, yy, c)
90     next
100  next
110 endfunc
```

もりで見直してみると、一見問題なさそうだった縮小時には逆に描く点が多すぎ、重複して点を打っていることに気づく。過不足なく描画するためには、拡大・縮小先の矩形の大きさでループを組み、“拡大・縮小元の各点がどこに写るか”ではなく“拡大・縮小先の各点がどこから写ってくるか”を求めるのが正しい。拡大・縮小先の点 (x', y') から逆に (x, y) を求め、 (x, y) の色を拾って (x', y') に打つわけだ。見かけ上、拡大時は縮小の計算を、縮小時には拡大の計算をすることになる。

逆変換の式は上の式をちょっとひっくり返せば、

$$x = \frac{x1-x0}{x3-x2} (x'-x2) + x0$$

$$y = \frac{y1-y0}{y3-y2} (y'-y2) + y0$$

と得られ、プログラムのほうもリスト2となる。より正確な結果を得るためには、40行や60行の割算の商を四捨五入すべきだが、その点さえ除けばリスト2は期待どおりの動作をする。

さて、マシン語にするうへでは、例によって実行速度を上げるために実数も乗除算も使わずにすませたい。じつは、その方法はすでに前章で示してある。次の式を見てもらいたい。

$$y = \frac{y1-y0}{x1-x0} (x-x0) + y0$$

リスト2
SCALING2.BAS

```

10 func test (x0, y0, x1, y1, x2, y2, x3, y3)
20   int x, y, xx, yy, c
30   for yy=y2 to y3
40     y=(yy-y2)*(y1-y0)/(y3-y2)+y0
50     for xx=x2 to x3
60       x=(xx-x2)*(x1-x0)/(x3-x2)+x0
70       c=point(x, y)
80       pset (xx, yy, c)
90     next
100  next
110 endfunc

```

これは2点 (x_0, y_0) , (x_1, y_1) を通る直線の方程式だが、上に示した拡大・縮小時の座標計算の式とまったく同じかたちをしている。すなわち、拡大・縮小の座標計算には線分発生アルゴリズムがそっくりそのまま使えるのだ。そうとわかれば話はかんたん、座標計算部分はBresenhamの線分発生アルゴリズムで x' から x を求めるループを組み、同様に y' から y を求めるループで括ればでき上がる。Bresenhamのアルゴリズムの二重ループだ。

拡大・縮小に線分発生アルゴリズムが応用できることに納得できない人のために若干補足しておく。話をかんたんにするために、 x 方向についての拡大・縮小にかぎって考える。図2-a)のように3ピクセル長の細長い画像を、5ピクセルに引き伸ばしたいとしよう。描画すべき点 P' が左から右に移動するにつれ、対応する元画像上の点 P もある一定の速度で移動していく。点 P' が左端から右端まで4ピクセル分移動する間に、点 P は2ピクセル移動する。この点 P の移動速度（というか移動のタイミング）は、線分 $(0, 0) - (4, 2)$ を描くときの y 座標の変化のしかたと一致するだろう。図2-b)のような線分を描くつもりになって、 x 座標を増加するタイミングで点 P' を、 y 座標を増加するタイミングで点 P を移動してやれば、点 P' に対応する元画像上の点 P が順次求まるというわけだ。

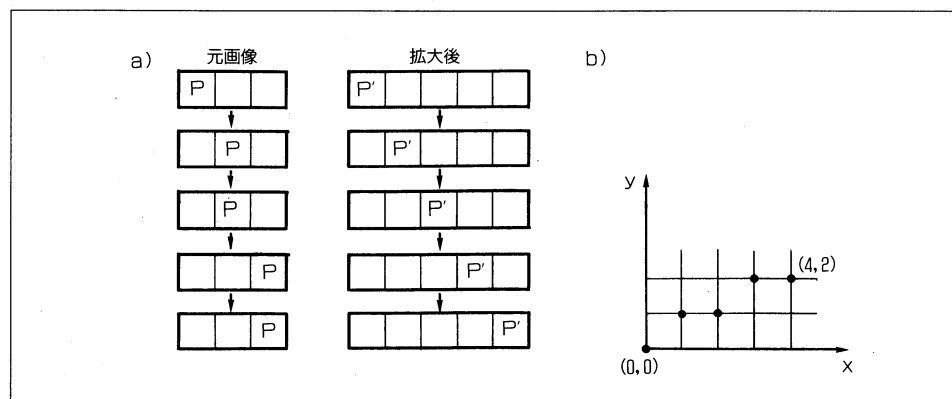


図2
拡大・縮小と線分発生アルゴリズムの関係

拡大・縮小ルーチン試作版

リスト3にBresenhamのアルゴリズムを応用したマシン語版の拡大・縮小ルーチンの第1版を示そう。画面上の矩形領域を拡大・縮小すると領域の重なりにおうじて処理を振り分ける必要が出てきて面倒なので、メインメモリ上に用意したグラフィックパターンを拡大・縮小して画面に描くプットルーチン仕様にした。なお、いまのところクリッピングはしていない。

リスト3
GXPUT.S
(第1版)

```

1: #      グラフィックパターンを拡大/縮小してプットする (第1版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gxput
7:      .xref      gramadr
8: #
9:      .offset 0      *gxputの引数構造
10: #
11: X0:      .ds.w      1      *描画先座標
12: Y0:      .ds.w      1      *
13: X1:      .ds.w      1      *
14: Y1:      .ds.w      1      *
15: PAT:      .ds.l      1      *パターンアドレス
16: XL:      .ds.w      1      *パターン横長さ-1
17: YL:      .ds.w      1      *パターン縦長さ-1
18: #
19:      .text
20:      .even
21: #
22: gxput:
23: ARGPTR = 4+8*4+7*4
24:      movem.l d0-d7/a0-a6, -(sp)
25:
26:      movea.l ARGPTR(sp), a6      *a6 = 引数列
27:
28:      movem.w (a6)+, d0-d3      *d0~d3 = 描画範囲の座標
29:      MINMAX d0, d2      *x0 ≤ x1を保証する
30:      MINMAX d1, d3      *y0 ≤ y1を保証する
31:
32:      movea.l (a6)+, a1      *a1 = パターン先頭アドレス
33:      movem.w (a6)+, a2-a3      *a2 = パターン横ピクセル数-1
34:      *a3 = パターン縦ピクセル数-1
35:
36:      sub.w d0, d2      *d2 = 描画範囲の横ピクセル数-1
37:      sub.w d1, d3      *d3 = 描画範囲の縦ピクセル数-1
38:
39:      jsr      gramadr      *a0 = 描画先左上G-RAMアドレス
40:
41:      lea.l 1(a2), a4      *a4 = パターン1ライン分ピクセル数
42:      adda.l a4, a4      *a4 = パターン1ライン分バイト数
43:
44:      move.w d2, d0      *
45:      neg.w d0      *d0 = xについての誤差項初期値
46:      bne fix0      *描画先幅が1ピクセルしかない場合の
47:      move.w d0, a2      * つじつま合わせ
48:      fix0:      move.w d2, d4      *d4 = xループカウンタ初期値
49:      add.w d2, d2      *d2 = xについての誤差項補正值
50:      add.w a2, a2      *a2 = xについての誤差項増分
51:
52:      move.w d3, d1      *
53:      neg.w d1      *d1 = yについての誤差項初期値
54:      bne fix1      *描画先高が1ピクセルしかない場合の
55:      move.w d1, a3      * つじつま合わせ
56:      fix1:      move.w d3, d5      *d5 = yループカウンタ初期値
57:      add.w d3, d3      *d3 = yについての誤差項補正值

```

```

58:      add.w  a3, a3          #a3 = yについての誤差項増分
59:
60: yloop: movea.l a0, a5      #a5 = 描画先
61:      movea.l a1, a6        #a6 = 参照元
62:
63:      move.w  d0, d6         #d6 = x誤差項
64:      move.w  d4, d7         #d7 = xループカウンタ
65: xloop: move.w  (a6), (a5)+   #1ピクセル書き込む
66:      add.w    a2, d6         #xについての誤差を累積させる
67:      ble     xnext          #
68: xincpl: addq.l #2, a6        #参照元x座標を進める
69:      sub.w    d2, d6         #x誤差項を補正する
70:      bgt     xincpl         #誤差項が0以下になるまで繰り返す
71: xnext: dbra   d7, xloop     #横幅分繰り返す
72:
73:      add.w    a3, d1         #yについての誤差を累積させる
74:      ble     ynext          #
75: yincpl: adda.l a4, a1        #参照元y座標を進める
76:      sub.w    d3, d1         #y誤差項を補正する
77:      bgt     yincpl         #誤差項が0以下になるまで繰り返す
78:
79: ynext: lea.l   GNBYTE(a0), a0 #描画先y座標を進める
80:      dbra   d5, yloop       #高さ分繰り返す
81:
82: done: movem.l (sp)+, d0-d7/a0-a6
83:      rts
84:
85:      .end

```

リスト3のサブルーチンgxputには9～17行のような形式で用意した引数列の先頭アドレスをスタックに積んで渡す。グラフィックパターンは前章で単純なプットルーチンをつくったときと同じ、1ピクセル/1ワードでパレットコードを並べた形式で用意する。前章でも触れたように、この形式は扱いが楽でプログラムの実行速度が稼げる反面、65536色モード以外で使うにはメモリ効率が悪い。必要であれば、256色モードや16色モード専用に改造することも考えてみてほしい。

では、プログラムを順に見ていこう。22行からサブルーチン本体が始まる。33行までで引数をレジスタに取り出す。d0～d3に描画先の矩形の座標、a1に描くグラフィックパターンの先頭アドレス、a2、a3にパターンの大きさだ。データレジスタが足りないものだから、a2、a3もデータ格納用に駆り出されている。そういえば、普通はフレームポインタに使うa6もすでに動員されている。

41～42行でグラフィックパターンの横1ライン分のバイト数をa4に求め、44～50行でx座標についての拡大・縮小計算にBresenhamのアルゴリズムを適用するのに必要なパラメータを計算する。誤差項の初期値をd0、誤差項の増分をa2、補正値をd2、ついでにループカウンタの初期値をd4に求めている。52～53行では同様にy座標についてのパラメータを計算している。46～47行、54～55行では少し変なことをやっているが、これについてはあとで触れる。

この時点で前処理が完了し、60行からy座標についてのループに突入する。60～61行でつぎに描画する横1ラインの左端のアドレスをa5に、対応するパターン横1ライン分の左端アドレスをa6に入れておき、その位置から右に向かって65～71行のx座標についてのループで1ライン分を描画する。

1ライン描画するループでは、まず65行で無条件に1ピクセル描く。a5はポストインクリメントされて2バイト進む。座標で考えると、これはx座標に1を足すことに相当する。それから66行で誤差項に増分を加える。その結果、誤差項が正になったら“誤差が十分溜まった”ので68～69行でパターンを指すポインタa6を1ピクセル分（2バイト）進め、同時に誤差項を補正する。ここで、縮小時にはポインタa6を数ピクセル分まとめて進める（パターンを数ピクセルとばす）場合があることに注意しよう。そのため、68～70行では誤差項が0以下になるまで、ポインタを進める処理と誤差項の補正をループで繰り返している。似たようなことはChapter4の多角形塗り潰しのときにもやったが、誤差項が0のときの扱いが異なっていることに注目してほしい。描画先の幅や高さが1ピクセルしかないときにBresenhamのアルゴリズムを使って拡大・縮小の座標計算をしようすると、ちょうど、“垂直な線分をx座標についてのループで描く”のと同様の矛盾した状態になる。誤差項の初期値は0、増分が正の値、補正値が0だから、誤差項に増分を加えると、もう負の値に戻ることがない。リスト3ではこの問題を避けるため、描画先の幅が1ピクセルしかない場合には、誤差項の増分を強制的に0にしておき（先ほどの46～47行）、誤差項が0のときは68～70行のループを擦り抜けるよう、70行の条件判断部分を微調整してあるというわけだ。

1ラインの描画がすんだら、73～77行で今度はy座標についてBresenhamのアルゴリズムによりパターンの参照位置を次のラインに進めるかどうか、進めるとしたらどれだけ進めるかを判定する。やっていることはx座標の場合の66～70行とほぼ同様だ。あわせて、79行で次の描画に備えて描画先を1ライン下に進める。

ここまでの処理を描画範囲の高さ分繰り返して描画完了だ。

描画の高速化

一応動く版ができたところで質の向上を目指す。クリッピング処理の追加の前に、無駄を省いて軽く高速化してやろう。多重ループのプログラムでは、もっとも内側のループに注目し、ループの外でできることをどんどん追いつけよう。リスト3なら、x座標に関してBresenhamのアルゴリズムを適用する65～71行のループが狙い目だ。リストをじっくり見て、余計な処理を行っていないかどうか検討してみよう。

読者はこのループ中に無駄を見つけることができただろうか。じつは、このループの中でBresenhamのアルゴリズムを適用していること自体が無駄なのだ。パターンへのポインタa6の進み方は毎回同じであり、このループを1回実行してみればどういうステップで進んでいくかわかる。何も、y座標が変わるごとに再計算する必要はない。そこで、前処理段階であらかじめポインタの増分をテーブルにしておくことを考える。たとえば、2倍に拡大する場合は同じ点を2度参照してとりに進むので、ポインタの増分（バイト単位）は0, 2, 0, 2……の繰り返しになる。また、2/3に縮小する場合であれば、パターンを2ピクセル拾ってから1ピク

セル分とばすわけだから、ポインタの増分は2, 4, 2, 2, 4, 2……の繰り返した。このような増分を並べたテーブルを先につくっておけば、横1ライン分を描くループが簡略化でき、実行速度もかなり違って来るだろう。

その方針でリスト3をつくり直すとリスト4のようになった。レジスタの使い方が細かく変わってたりするので変更点のみではなく完全なかたちのリストで示す。テーブルを置くメモリはサブルーチン呼び出す側が用意し、引数で渡すようにした。20行のように、引数列の末尾にこのワーク用メモリへのポインタが追加されている。ワークはスタック上に確保してもよかったのだが、描画する範囲の横幅×2バイト、1024ドットモード時で最大2Kバイトの大きさともなると、メインルーチン側でそれを見越してスタックを確保しなければならない。それなら公に引数で渡すようにしても、メインにかかる負担は同じようなものだと考えた。

リスト4
GXPUT.S
(高速化版)

```

1: #      グラフィックパターンを拡大/縮小してブットする (高速化版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gxput
7:      .xref      gramadr
8: #
9:      .offset 0      #gxputの引数構造
10: #
11: X0:      .ds.w      1      #描画先座標
12: Y0:      .ds.w      1      #
13: X1:      .ds.w      1      #
14: Y1:      .ds.w      1      #
15: PAT:      .ds.l      1      #パターンアドレス
16: XL:      .ds.w      1      #パターン横長さ-1
17: YL:      .ds.w      1      #パターン縦長さ-1
18: TEMP:      .ds.l      1      #テーブル用
19: #
20:      .text
21:      .even
22: #
23: gxput:
24: ARGPTR = 4+6*4+7*4
25:      movem.l d0-d5/a0-a6, -(sp)
26:
27:      movea.l ARGPTR(sp), a6      #a6 = 引数列
28:
29:      movem.w (a6)+, d0-d3      #描画範囲の座標を取り出す
30:      MINMAX d0, d2      #x0 ≤ x1を保証する
31:      MINMAX d1, d3      #y0 ≤ y1を保証する
32:
33:      movea.l (a6)+, a1      #a1 = パターン先頭アドレス
34:      movem.w (a6)+, a2-a3      #a2 = パターン横ピクセル数-1
35:                                   #a3 = パターン縦ピクセル数-1
36:      movea.l (a6), a6      #a6 = テーブル用ワーク
37:
38:      sub.w d0, d2      #d2 = 描画範囲の横ピクセル数-1
39:      sub.w d1, d3      #d3 = 描画範囲の縦ピクセル数-1
40:
41:      jsr      gramadr      #a0 = 描画先左上G-RAMアドレス
42:
43:      lea.l 1(a2), a4      #a4 = パターン1ライン分ピクセル数
44:      adda.l a4, a4      #a4 = パターン1ライン分バイト数
45:
46:      move.w d2, d0      #
47:      neg.w d0      #d0 = xについての誤差項初期値
48:      bne fix0      #描画先幅が1ピクセルしかない場合の
49:      move.w d0, a2      # つじつま合わせ
50: fix0:      move.w d2, d4      #d4 = xループカウンタ初期値

```

```

51:      add.w  d2, d2      *d2 = xについての誤差項増分
52:      add.w  a2, a2      *a2 = xについての誤差項補正值
53:
54:      movea.l a6, a5      *a5 = テーブル先頭アドレス
55:      move.w  d4, d5      *d5 = xループカウンタ
56:  iloop: moveq.l #0, d1    *d1 = xの増分(0に初期化)
57:      add.w  a2, d0      *xについての誤差を累積させる
58:      ble    inext       *
59:  iincp: addq.w #2, d1     *xの増分を増す
60:      sub.w  d2, d0      *x誤差項を補正する
61:      bgt    iincp       *誤差項が0以下になるまで繰り返す
62:  inext: move.w d1, (a5)+  *テーブルに登録
63:      dbra   d5, iloop    *横幅分繰り返す
64:
65:      move.w  d3, d1      *
66:      neg.w   d1          *d1 = yについての誤差項初期値
67:      bne     fix1        *描画先高が1ピクセルしかない場合の
68:      move.w  d1, a3      * つじつま合わせ
69:  fix1: move.w  d3, d5      *d5 = yループカウンタ初期値
70:      add.w  d3, d3      *d3 = yについての誤差項増分
71:      add.w  a3, a3      *a3 = yについての誤差項補正值
72:
73:      move.l  a6, d2      *d2 = テーブル先頭アドレス
74:  yloop: movea.l a0, a5    *a5 = 描画先
75:      movea.l a1, a6      *a6 = 参照元
76:
77:      movea.l d2, a2      *a2 = テーブル先頭アドレス
78:      move.w  d4, d0      *d0 = xループカウンタ
79:  xloop: move.w (a6), (a5)+ *1ピクセル書き込む
80:      adda.w  (a2)+, a6    *参照元x座標を進める
81:      dbra   d0, xloop    *横幅分繰り返す
82:
83:      add.w  a3, d1      *yについての誤差を累積させる
84:      ble    ynext       *
85:  yincp: adda.l a4, a1     *参照元y座標を進める
86:      sub.w  d3, d1      *y誤差項を補正する
87:      bgt    yincp       *誤差項が負になるまで繰り返す
88:
89:  ynext: lea.l  GNBYTE(a0), a0 *描画先y座標を進める
90:      dbra   d5, yloop    *高さ分繰り返す
91:
92: done: movem.l (sp)+, d0-d5/a0-a6
93:      rts
94:
95:      .end

```

54～63行がテーブルを作成するループだ。やっていることはリスト3の内側のループと同様だが、ポインタを実際に進める代わりに、ポインタの増分をカウントするよう変更されている。飛んで79～81行が作成したテーブルを使って1ライン分を描くループだ。ループの中身は点を打ち、テーブルから引いた増分をポインタに足すだけのわずかな2命令に簡略化されている。ここまでシンプルになると、ついつい展開してもっと速くしてやろうという気になるが、それはあとの楽しみにとっておく。

クリッピングの追加

では、クリッピングの処理を追加しよう。前章でもやったように、プットルーチンのクリッピングでは描画範囲を切り詰めたら、同時にその分パターンの方も切り捨てなければならない。ところが、いまつくっているのは拡大・縮小機能付きのプットルーチンであり、どれだけ

パターンを切り捨てるかがかんたんには求まらないという点で若干難易度が高い。たとえば、クリッピングの結果30%描画範囲が狭まったら、パターンも30%切り捨てるわけだが、その30%が何ピクセルに相当するかはパターンの大きさや拡大・縮小率におうじて変わってくるのだ。

と書くとなにやら複雑そうだが、似たような処理はもう紹介済みだ。拡大・縮小ルーチンとラインルーチンの対比からピンときたと思う。Chapter3の線分のクリッピングアルゴリズムが利用できる。線分のクリッピング時には、x座標を切り詰めたら、同時にy座標も同じ比率で切り詰めた。同じ考え方で、描画範囲を狭めるのと同じ比率で元パターンを切り捨てようというわけだ。

で、リスト5がクリッピングの処理を追加した新版の拡大・縮小プットルーチンだ。

リスト5
GXPUT.S
(クリッピング対応版)

```

1: #      グラフィックパターンを拡大/縮小してプットする (クリッピング対応版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gxput
7:      .xref      gramadr
8:      .xref      cliprect
9:      .xref      ucliprect
10: #
11:      .offset 0      *gxputの引数構造
12: #
13: X0:      .ds.w      1      *描画先座標
14: Y0:      .ds.w      1      *
15: X1:      .ds.w      1      *
16: Y1:      .ds.w      1      *
17: PAT:      .ds.l      1      *パターンアドレス
18: XL:      .ds.w      1      *パターン横長さ-1
19: YL:      .ds.w      1      *パターン縦長さ-1
20: TEMP:      .ds.l      1      *テーブル用
21: #
22:      .text
23:      .even
24: #
25: gxput:
26: ARGPTR = 4+8+4+7*4
27:      movem.l d0-d7/a0-a6, -(sp)
28:
29:      movea.l ARGPTR(sp), a6      *a6 = 引数列
30:
31:      movem.w (a6)+, d0-d3      *描画範囲の座標を取り出す
32:      MINMAX d0, d2      *x0 ≤ x1を保証する
33:      MINMAX d1, d3      *y0 ≤ y1を保証する
34:
35:      movea.l (a6)+, a1      *a1 = パターン先頭アドレス
36:      movem.w (a6)+, a2-a3      *a2 = パターン横ピクセル数-1
37:      *a3 = パターン縦ピクセル数-1
38:      movea.l (a6), a6      *a6 = テーブル用ワーク
39:
40:      lea.l cliprect, a0      *a0 = クリッピング範囲
41:      cmp.w (a0)+, d2      *x1 < minxならウィンドウ外
42:      blt done
43:      cmp.w (a0)+, d3      *y1 < minyならウィンドウ外
44:      blt done
45:      cmp.w (a0)+, d0      *x0 > maxxならウィンドウ外
46:      bgt done
47:      cmp.w (a0)+, d1      *y0 > maxyならウィンドウ外
48:      bgt done
49:
50:      lea.l ucliprect, a0      *a0 = クリッピング範囲 (ゲタ履き)

```

```

51:      move.w  #$8000, d5      *x0, y0, x1, y1に8000Hのゲタを戻させる
52:      add.w   d5, d0          *
53:      add.w   d5, d1          *
54:      add.w   d5, d2          *
55:      add.w   d5, d3          *
56:
57:      move.w  a2, d4          *d4 = パターン横ピクセル数-1
58:      move.w  a3, d5          *d5 = パターン縦ピクセル数-1
59:      move.w  d2, a4          *d2をa4に待避
60:      move.w  d3, a5          *d3をa5に待避
61:
62: minxclip:      *MINXでクリッピングする
63:      move.w  (a0)+, d6      *d6 = minx
64:      cmp.w   d6, d0          *x0>minxなら
65:      bcc     minyclip      * クリッピング不要
66:
67:      cmp.w   d6, d2
68:      bne     minx0
69:
70:      move.w  d2, d0
71:      move.w  d4, d3
72:      bra     minx3
73:
74: minx0:  moveq.l #0, d3
75: minxlp:  move.w  d0, d7
76:      add.w   d2, d7
77:      roxr.w  #1, d7
78:      cmp.w   d6, d7
79:      beq     minx2
80:      bcs     minx1
81:
82:      move.w  d7, d2
83:      add.w   d3, d4
84:      roxr.w  #1, d4
85:      bra     minxlp
86:
87: minx1:  move.w  d7, d0
88:      add.w   d4, d3
89:      roxr.w  #1, d3
90:      bra     minxlp
91:
92: minx2:  move.w  d7, d0      *d0 = x0 = minx
93:      add.w   d4, d3
94:      roxr.w  #1, d3      *d3 = パターンが左端にはみ出る分
95: minx3:  move.w  a2, d4      *d4 = パターン横ピクセル数-1
96:      sub.w   d3, a2      *はみ出た分パターンの横幅を詰める
97:      add.w   d3, d3      *
98:      adda.w  d3, a1      *はみ出た分パターンを切り捨てる
99:      move.w  a4, d2      *d2を復帰
100:     move.w  a5, d3      *d3を復帰
101:
102: minyclip:      *MINYでクリッピングする
103:      addq.w  #1, d4          *d4 = パターン1ライン分のピクセル数
104:      add.w   d4, d4          *d4 = パターン1ライン分のバイト数
105:
106:      move.w  (a0)+, d6
107:      cmp.w   d6, d1
108:      bcc     maxxclip
109:
110:      cmp.w   d6, d3
111:      bne     miny0
112:
113:      move.w  d3, d1
114:      move.w  d5, d2
115:      bra     miny3
116:
117: miny0:  moveq.l #0, d2
118: miny1p:  move.w  d1, d7
119:      add.w   d3, d7
120:      roxr.w  #1, d7
121:      cmp.w   d6, d7
122:      beq     miny2
123:      bcs     miny1
124:

```

```

125:      move.w d7, d3
126:      add.w d2, d5
127:      roxr.w #1, d5
128:      bra miny1p
129:
130: miny1: move.w d7, d1
131:      add.w d5, d2
132:      roxr.w #1, d2
133:      bra miny1p
134:
135: miny2: move.w d7, d1
136:      add.w d5, d2
137:      roxr.w #1, d2
138: miny3: sub.w d2, a3
139:      mulu.w d4, d2
140:      adda.l d2, a1
141:      move.w a4, d2
142:      move.w a5, d3
143:
144: maxxclip:      *MAXXでクリッピングする
145:      move.w d4, d6      *d6 = パターン1ライン分バイト数
146:
147:      move.w (a0)+, d4      *d4 = maxx
148:      sub.w d2, d4      *d4 = maxx-x1
149:      bmi maxyclip      *
150:      moveq.l #0, d4      *右端にははみ出ていなかった
151:
152: maxyclip:
153:      move.w (a0)+, d5      *d5 = maxy
154:      sub.w d3, d5      *d5 = maxy-y1
155:      bmi clipped      *
156:      moveq.l #0, d5      *下端にははみ出ていなかった
157:
158: clipped:
159:      sub.w d0, d2      *d2 = 描画範囲の横ピクセル数-1
160:      sub.w d1, d3      *d3 = 描画範囲の縦ピクセル数-1
161:
162:      move.w #$8000, d7      *ゲタを脱がせる
163:      sub.w d7, d0      *
164:      sub.w d7, d1      *
165:
166:      jsr gramadr      *a0 = 描画先左上G-RAMアドレス
167:
168:      move.w d2, d0      *
169:      neg.w d0      *d0 = xについての誤差項初期値
170:      bne fix0      *描画先幅が1ピクセルしかない場合の
171:      move.w d0, a2      * つじつま合わせ
172:      add.w d2, d4      *d4 = xループカウンタ初期値
173:      add.w d2, d2      *d2 = xについての誤差項増分
174:      add.w a2, a2      *a2 = xについての誤差項補正値
175:
176:      movea.l a6, a4      *a4 = テーブル先頭アドレス
177:      move.w d4, d7      *d7 = xループカウンタ
178:      moveq.l #0, d1      *d1 = x方向のアドレス増分(0に初期化)
179:      add.w a2, d0      *xについての誤差を累積させる
180:      ble inext      *
181:      iinclp: addq.w #2, d1      *アドレスの増分を累積する
182:      sub.w d2, d0      *x誤差項を補正する
183:      bgt iinclp      *誤差項が0以下になるまで繰り返す
184:      inext: move.w d1, (a4)+      *テーブルに登録
185:      dbra d7, iloop      *横幅分繰り返す
186:
187:      move.w d3, d1      *
188:      neg.w d1      *d1 = yについての誤差項初期値
189:      bne fix1      *描画先高が1ピクセルしかない場合の
190:      move.w d1, a3      * つじつま合わせ
191:      add.w d3, d5      *d5 = yループカウンタ初期値
192:      add.w d3, d3      *d3 = yについての誤差項増分
193:      add.w a3, a3      *a3 = yについての誤差項補正値
194:
195:      move.l a6, d2      *d2 = テーブル先頭アドレス
196:      yloop: movea.l a0, a5      *a5 = 描画先
197:      movea.l a1, a6      *a6 = 参照元
198:

```

```

199:      movea.l d2, a4      #a4 = テーブル先頭アドレス
200:      move.w d4, d0      #d0 = xループカウンタ
201: xloop: move.w (a6), (a5)+ #1ピクセル書き込む
202:      adda.w (a4)+, a6    #参照元x座標を進める
203:      dbra d0; xloop     #横幅分繰り返す
204:
205:      add.w a3, d1        #yについての誤差を累積させる
206:      ble ynext          #
207: yinc1p: adda.w d6, a1    #参照元y座標を進める
208:      sub.w d3, d1        #y誤差項を補正する
209:      bgt yinc1p         #誤差項が0以下になるまで繰り返す
210:
211: ynext: lea.l GNBYTE(a0), a0 #描画先y座標を進める
212:      dbra d5; yloop     #高さ分繰り返す
213:
214: done: movem.l (sp)+, d0-d7/a0-a6
215:      rts
216:
217:      .end

```

40～152行あたりがごっそり追加された部分だ。ここでは、完全にクリッピングウィンドウ外のケースを真っ先に弾いている。すると50行に到達した時点で描画範囲は（最低限、角は）クリッピングウィンドウの中にあるので、ここから本格的なクリッピングに取りかかる。

51～55行で線分のクリッピングのときにもやったように座標に8000_Hのバイアスをかけて（ゲタを履かせて）大小関係を保ったまま無符号数に変換している。この変換は、中点の座標を求める過程で平均を取るときのオーバーフローを防ぐのが目的だということを思い出そう。

続いて、パターンの大きさがアドレスレジスタに格納されたままでは演算がやりにくいので、57～58行でa2, a3を空いているデータレジスタd4, d5に転送する。また、59～60行では描画範囲の右下隅の座標を保持するd2, d3をa4, a5に待避している。この時点で、データレジスタには次のような値が収まっている。

d0 = 描画範囲の左上隅x座標（ゲタ履き）
d1 = 描画範囲の左上隅y座標（ゲタ履き）
d2 = 描画範囲の右下隅x座標（ゲタ履き）
d3 = 描画範囲の右下隅y座標（ゲタ履き）
d4 = パターンの横幅 - 1
d5 = パターンの高さ - 1

62～100行で、ウィンドウの左端について描画範囲、グラフィックパターンともどもクリッピングする。これは、線分 (d0, 0) - (d2, d4) をウィンドウ左端のx座標minxでクリッピングする処理になぞらえることができる。ただし、d0, d2はゲタ履きなのにパターンの横幅d4にはゲタを履かせていないという点で少々変則的だ。パターンの大きさは負になりえないから、これでよいのである。その点さえ注意して見てもらえれば、Chapter3の線分クリッピングルーチンとまったく同じようなことをしているのがわかるだろう。処理後、この疑似線分は (minx, ?) - (d2, d4) のように切り詰められ、“?”の部分に“グラフィックパターンが何ピクセル分左側にはみ出すか”が求まる。リスト5でいうと95行に達した時点でd3にこの値が収まっている。96行ではその分だけパターンの横幅を減じ、97～98行でパターンを実際に

切り捨てている。102～142行のウィンドウ上端でのクリッピングもレジスタの割り当てが異なるだけでやっていることは同じだ。

ウィンドウの左端、上端でのクリッピングがすんだら制御は144行にくる。ラベルが示しているようにここでウィンドウ右端についてクリッピングする。といっても、実際にはx座標についてのループカウンタのつじつまをあわせているだけだ。描画時の処理は左から右へと進むわけであり、処理開始位置である左端はきちんとクリッピングして正確な位置を求める必要があるが、処理終了位置である右端についてはループカウンタを適当に減らしてウィンドウの外まで描画しないよう調整すればクリッピングしたのと同じ効果がある。なまじクリッピングの計算をしない分、誤差の入り込む余地もない。152～156行のウィンドウ下端でのクリッピングも同様だ。

ところが、残念ながらこの簡略クリッピングには副作用がある。与えられた座標値によっては、後でBresenhamのアルゴリズムを適用する際にオーバーフローを生じるのだ。具体的には描画範囲の右下隅のx座標、y座標のどちらか少なくとも一方が4000_Hよりも大きいと演算過程で座標（の差）を2倍した段階で値が16ビットの負の数になり、アルゴリズムに混乱をきたす可能性が出てくる。バグも仕様書に書けば仕様が変わるというから、作者である僕は次の1文を追加することによって責任を回避するとしてよう。

「gxputが受けつける座標の範囲は－32768～＋16383である」

もちろん、読者はこんなみっともない仕様を甘受せず、プログラムを正しく修正することを考えてみてほしい。対処としてはクリッピングの処理を追加する正攻法と、Bresenhamのアルゴリズムを適用している部分をロングワードで演算するように修正する泥縄法の2通りがある。

ふたたび高速化

クリッピングの処理もついたことだし、速度追及の最終段階に入ろう。宿題になっていた水平描画部分のループ展開を片付ける。ただ展開するだけならかんたんだ。

```
move.w  (a6), (a5) +
adda.w  (a4) +, a6
move.w  (a6), (a5) +
adda.w  (a4) +, a6
      :
```

のようにループの中身を必要な数だけ並べればよい。それでは芸がないのもう1段工夫しよう。もし、拡大・縮小率が固定ならば、この命令列は、

```
move.w  (a6), (a5) +
lea.l    n1 (a6), a6
move.w  (a6), (a5) +
```



```
lea.l    n2 (a6),a6
```

```
:
```

という、より高速なルーチンに置き換えることができる (n1, n2は本来はテーブルに並べてあったポインタの移動量)。理屈では、拡大・縮小率ごとにこのような専用ルーチンを用意できれば、最高速のプログラムになる。だが、さすがにそれではメモリがいくらあっても足りない。そこで、gxputの呼び出しごとに専用ルーチンを“つくる”ことにする。ポインタの移動量のテーブルをつくる代わりに、描画ルーチン自体を生成するのだ。

リスト6にリスト5からの変更点を示す。178～186行のループではこれまでどおりの方法でポインタの増分を求めては、

```
move.w (a4), (a5) +
```

```
lea.l    XXX(a4),a4
```

のXXXの部分にその値を埋め込んだ3ワードの命令列を作業用メモリに書き込んでいる (パターンを指すポインタはa6からa4に変更されている)。最後に、

```
rts
```

の命令コードを書き込んで、1ライン分描画するサブルーチンができ上がる (188行)。201行ではこのサブルーチンを呼び出すことで1ライン分の描画を行っている。この変更により、ワークエリアの使用量が3倍強に増えたことに注意してほしい。

リスト6
GXPUT.S
(最終版)

```
1: #      グラフィックパターンを拡大/縮小してブットする (最終版)

176:      movea.l a6, a4      #a4 = ルーチン作成先先頭アドレス
177:      move.w d4, d7      #d7 = xループカウンタ
178: iloop: moveq.l #0, d1    #d1 = x方向のアドレス増分(0に初期化)
179:      add.w a2, d0        #xについての誤差を累積させる
180:      ble inext          #
181: iincpl: addq.w #2, d1    #アドレスの増分を累積する
182:      sub.w d2, d0        #x誤差項を補正する
183:      bgt iincpl         #誤差項が0以下になるまで繰り返す
184: inext: move.l pset(pc), (a4) + #move, leaの2命令を書き込む
185:      move.w d1, (a4) +   #アドレスの増分を書き込む
186:      dbra d7, iloop      #横幅分繰り返す
187:
188:      move.w term(pc), -4(a4) #rts命令を書き込む
189:
190:      move.w d3, d1        #
191:      neg.w d1             #d1 = yについての誤差項初期値
192:      bne fixl            #描画先高が1ピクセルしかない場合の
193:      move.w d1, a3        # つじつま合わせ
194: fixl:  add.w d3, d5        #d5 = yループカウンタ初期値
195:      add.w d3, d3        #d3 = yについての誤差項増分
196:      add.w a3, a3        #a3 = yについての誤差項補正值
197:
198: yloop: movea.l a0, a5      #a5 = 描画先
199:      movea.l a1, a4      #a4 = 参照元
200:
201:      jsr (a6)            #1ライン描画する
202:
203:      add.w a3, d1        #yについての誤差を累積させる
204:      ble ynext          #
205: yincpl: adda.w d6, a1      #参照元y座標を進める
206:      sub.w d3, d1        #y誤差項を補正する
207:      bgt yincpl         #誤差項が0以下になるまで繰り返す
208:
```

```

209: ynext: lea.l   GNBYTE(a0), a0    #描画先y座標を進める
210:       dbra   d5, yloop           #高さ分繰り返す
211:
212: done:  movem.l (sp)+, d0-d7/a0-a6
213: term:  rts
214:
215: pset:  move.w  (a4), (a5)+        #1ピクセル書き込む命令
216:       lea.l   0(a4), a4          #参照位置を移動する命令
217:
218:       .end

```

バリエーション

ここから先はオプションだ。拡大・縮小ルーチンのバリエーションを4本ほど示す。

リスト7はリスト6の縦方向の拡大時の処理をもう一段高速化した版だ（変更点のみ）。そのかわり、縮小時には不要な処理を行うので、わずかに速度が低下する。高速化の理屈は単純で、“パターンの参照位置が直前に描いたラインと変わらなければ、真面目に描く代わりに1ラインコピーする”というものだ。

.....
 リスト7
 GXP.UT.S
 (拡大時処理高速化オプション)

```

1: #      グラフィックパターンを拡大/縮小してプットする（拡大時処理高速化版）

196:      add.w   a3, a3              #a3 = yについての誤差項補正值
197:      move.w   a3, d7              #d7 = yについての誤差項補正值
198:
199:      move.w   d4, d0
200:      addq.w   #1, d0
201:      lea.l    cnext(pc), a2       #a2 = 戻りアドレス
202:      bclr.l   #0, d0              #横ピクセル数は奇数か?
203:      beq      skip
204:      subq.l   #2, a2
205: skip:   lea.l  alincopy(pc), a3
206:      suba.w   d0, a3
207:
208: yloop:  movea.l a0, a5             #a5 = 描画先
209:      movea.l  a1, a4             #a4 = 参照元
210:
211:      jsr      (a6)               #1ライン描画する
212:
213:      add.w    d7, d1              #yについての誤差を累積させる
214:      ble      lincpy
215: yinc1p: adda.w  d6, a1             #参照元y座標を進める
216:      sub.w    d3, d1              #y誤差項を補正する
217:      bgt      yinc1p             #誤差項が0以下になるまで繰り返す
218:
219: ynext:  lea.l   GNBYTE(a0), a0    #描画先y座標を進める
220:       dbra   d5, yloop           #高さ分繰り返す
221:
222: done:  movem.l (sp)+, d0-d7/a0-a6
223: term:  rts
224:
225: pset:  move.w  (a4), (a5)+        #1ピクセル書き込む命令
226:       lea.l   0(a4), a4          #参照位置を移動する命令
227:
228: cpy1p1: movea.l a0, a4
229:       lea.l   GNBYTE(a0), a0
230:       movea.l a0, a5
231:       jmp     (a3)
232:       move.w  (a4)+, (a5)+
233: cnext:  add.w   d7, d1

```

```

234:      bgt      yinc1p
235: lincpy: dbra   d5, cpy1pl
236:      bra      done
237:
238:      .dcb, w   GNP1XEL/2, $2adc #move, l (a4)+, (a5)+
239: alincopy:
240:      jmp      (a2)
241:
242:      .end

```

リスト8はリスト5（リスト6ではない）を透明色に対応させた版だ（変更点のみ）。パターン中、パレットコードが0の部分は描画せずに下地をそのまま残す。ゲームのキャラクター描画なんかに使えるかもしれない。もっとも、ゲームに使うのならより高速化すべきだろう。リスト6を参考に1ライン分描画するループを展開するのがずっと早いと思う。パターンの大きさや拡大・縮小率が決まりきった値しかとらないようなら、それぞれの場合に応じた専用ルーチンをあらかじめ用意しておくことも検討したいところだ。もちろん、先に拡大・縮小したパターンを作成してメモリ上に持ってしまうのがいちばん速いが、その場合メモリ容量の問題もあって、あまり大きく拡大するのはあきらめなければならない。メモリ容量といえば、最初にも触れたように、ゲーム向きの256色モードなどで使うにはメモリ効率が悪いから、その点もどうにかすることになるだろう。

リスト8
GXPUTON.S

```

1: #      グラフィックパターンを拡大/縮小してプットする（透明色対応版）
6:      .xdef    gxputon
25: gxputon:
201: xloop: move, w  (a6), d7      #d7=描画色
202:      beq      xskip          #透明色部分は飛ばす
203:      move, w  d7, (a5)+      #1ピクセル書き込む
204:      adda, w  (a4)+, a6      #参照元y座標を進める
205:      dbra    d0, xloop      #横幅分繰り返す
206:      bra      xdone
207:
208: xskip:  addq, l  #2, a5      #描画先x座標を進める
209:      adda, w  (a4)+, a6      #参照元x座標を進める
210:      dbra    d0, xloop      #横幅分繰り返す
211:
212: xdone:  add, w  a3, d1      #yについての誤差を累積させる
213:      ble     ynext          #
214: yinc1p: adda, w  d6, a1      #参照元y座標を進める
215:      sub, w  d3, d1          #y誤差項を補正する
216:      bgt     yinc1p          #誤差項が0以下になるまで繰り返す
217:
218: ynext:  lea, l  GNP1XEL(a0), a0 #描画先y座標を進める
219:      dbra    d5, yloop      #高さ分繰り返す
220:
221: done:   movem, l (sp)+, d0-d7/a0-a6
222:      rts
223:
224:      .end

```

一転してリスト9のg1to2は、質優先の画像1/2変換サブルーチンだ。ただし、65536色モード専用となっている。絵を描く人ならこの種のルーチンは自前で用意しているかもしれない。引

数としては画面上の矩形範囲の座標（を格納したメモリへのポインタ）を与える。縮小後の画像は元絵の左上隅の部分に重ねて描かれる。gxputのように点を間引いて縮小するのではなく、元絵の2×2の範囲をRGBごとに足して平均をとる。

リスト9
G1TO2.S

```

1: #      矩形領域を縦横1/2に縮小する
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      glto2
7:      .xref      gramadr
8:      .xref      gcliprect
9: #
10:     .offset 0      *glto2の引数構造
11: #
12: X0:   .ds.w      1
13: Y0:   .ds.w      1
14: X1:   .ds.w      1
15: Y1:   .ds.w      1
16: #
17:     .text
18:     .even
19: #
20: glto2:
21: ARGPTR = 8
22:     link      a6, #0
23:     movem.l   d0-d7/a0-a3, -(sp)
24:
25:     movea.l   ARGPTR(a6), a1      *a1 = 引数数列
26:     movem.w   (a1), d0-d3        *d0~d3=座標
27:
28:     jsr      gcliprect          *クリッピングする
29:     bne      done              *Z=0なら描画の必要なし
30:
31:     jsr      gramadr            *G-RAM上のアドレスを得る
32:
33:     sub.w    d0, d2             *d2=横ピクセル数-1
34:     sub.w    d1, d3             *d3=縦ピクセル数-1
35:
36:     addq.w   #1, d2             *横ピクセル数を半減
37:     lsr.w    #1, d2             *
38:     subq.w   #1, d2             *
39:     bmi      done              *
40:
41:     addq.w   #1, d3             *縦ピクセル数を半減
42:     lsr.w    #1, d3             *
43:     subq.w   #1, d3             *
44:     bmi      done              *
45:
46:     move.w   #GNBYTE, d1        *d1=ライン間のアドレスの差
47:
48:     movea.l   a0, a2
49: loop1:  movea.l   a0, a1          *a1=参照元ライン左端
50:         movea.l   a2, a3          *a3=描画先ライン左端
51:         move.w    d2, d4          *d4=横ピクセル数-1
52:
53:         swap.w    d1
54:         swap.w    d2
55:         swap.w    d3
56:
57: loop2:  move.w    (a1)+, d0        * (x, y) の色を
58:         DERGB     d0, d1, d2, d3  *   rgbに分解
59:
60:         move.w    (a1)+, d0        * (x+1, y) の色を
61:         DERGB     d0, d5, d6, d7  *   rgbに分解して
62:         add.w     d5, d1            *   加算
63:         add.w     d6, d2            *
64:         add.w     d7, d3            *
65:

```

```

66:      move.w  GNBYTE-4(a1), d0  # (x, y+1)の色を
67:      DERGB   d0, d5, d6, d7    #  rgbに分解して
68:      add.w    d5, d1             #  加算
69:      add.w    d6, d2             #
70:      add.w    d7, d3             #
71:
72:      move.w  GNBYTE-2(a1), d0  # (x+1, y+1)の色を
73:      DERGB   d0, d5, d6, d7    #  rgbに分解して
74:      add.w    d5, d1             #  加算
75:      add.w    d6, d2             #
76:      add.w    d7, d3             #
77:
78:      lsr.w    #2, d1             # b/4
79:      lsr.w    #2, d2             # r/4
80:      lsr.w    #2, d3             # g/4
81:
82:      RGB      d1, d2, d3, d0    # カラーコードに再構成して
83:      move.w   d0, (a3)+         # 書き込む
84:
85:      dbra     d4, loop2         # 横幅分繰り返す
86:
87:      swap.w   d1
88:      swap.w   d2
89:      swap.w   d3
90:
91:      adda.w   d1, a0             # 参照元は2ライン下へ
92:      adda.w   d1, a0             #
93:      adda.w   d1, a2             # 描画先は1ライン下へ
94:
95:      dbra     d3, loop1         # 高さ分繰り返す
96:
97: done:  movem.l  (sp)+, d0-d7/a0-a3
98:      unlink   a6
99:      rts
100:
101:      .end

```

リスト10のgshrinkはやはり質優先の画像縮小ルーチン（参考）で、こっちは縮小率を縦横個別に自由に指定できる。リスト8同様、縮小後の絵は元絵の左上隅に置かれる。半分以上冗談でつくった工夫のかけらもないプログラムだが“注意してつかえば”実用になるかもしれない。引数は30～36行のような構造で用意して、その先頭アドレスをスタックに積んで渡す。TEMPは作業用に使うメモリで、最大6Kバイト必要だ。

リスト10
GSHRINK.S

```

1: #      グラフィック画像を縮小する
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef         gshrink
7:      .xref         gramadr
8:      .xref         gcliprect
9: #
10: FPACK macro      callno
11:      .dc.w        callno
12:      .endm
13:
14: __UDIV equ        $fe05
15: #
16: GCM      macro    M, N      #最大公約数を求めるマクロ
17:      local      loop
18:      move.w     N, d0
19: loop:     moveq.l  #0, N
20:      move.w     d0, N
21:      divu.w     M, N

```

```

22:      move.w M, d0
23:      swap.w N
24:      move.w N, M
25:      bne     loop
26:      .endm
27:  *
28:      .offset 0      *gshrinkの引数構造
29:  *
30:  X0:      .ds.w 1      *描画先座標
31:  Y0:      .ds.w 1      *
32:  X1:      .ds.w 1      *
33:  Y1:      .ds.w 1      *
34:  XL:      .ds.w 1      *縮小後の横ピクセル数-1
35:  YL:      .ds.w 1      *縮小後の縦ピクセル数-1
36:  TEMP:    .ds.l 1      *作業用ワーク (最大6 Kバイト)
37:  *
38:      .offset -42
39:  *
40:  WORK:
41:  EX:      .ds.l 1
42:  DEX:     .ds.l 1
43:  CEX:     .ds.l 1
44:  EY:      .ds.l 1
45:  DEY:     .ds.l 1
46:  CEY:     .ds.l 1
47:  XSTEP:   .ds.w 1
48:  YSTEP:   .ds.w 1
49:  DENOM:   .ds.l 1
50:  DXL:     .ds.w 1
51:  SXL:     .ds.w 1
52:  PATSIZ:  .ds.w 1
53:  BUFF:    .ds.l 1
54:  _a6:     .ds.l 1      *±0
55:  _a7:     .ds.l 1
56:  ARGPTR:  .ds.l 1
57:  *
58:      .text
59:      .even
60:  *
61:  gshrink:
62:      link    a6, #WORK
63:      movem.l d0-d7/a0-a5, -(sp)
64:
65:      movea.l ARGPTR(a6), a1      *a1 = 引数列
66:
67:      movem.w (a1)+, d0-d3/a2-a3
68:                                     *d0~d3=描画範囲の座標
69:                                     *a2=縮小後横ピクセル数-1
70:                                     *a3=縮小後縦ピクセル数-1
71:      movea.l (a1), a4      *a4=1ラインバッファ
72:      move.l  a4, BUFF(a6)  *
73:
74:      jsr     gcliprect      *クリッピングする
75:      bne     done           *Z=0なら描画の必要なし
76:
77:      jsr     gramadr        *a0=描画先左上G-RAMアドレス
78:      movea.l a0, a1
79:
80:      sub.w   d0, d2          *d2=描画範囲の横ピクセル数-1
81:      beq     done
82:      sub.w   d1, d3          *d3=描画範囲の縦ピクセル数-1
83:      beq     done
84:
85:      cmp.w   a2, d2          *拡大はできない
86:      bcs     done           *
87:      cmp.w   a3, d3          *
88:      bcs     done           *
89:
90:      move.w  a2, d7          *
91:      addq.w  #1, d7          *
92:      add.w   d7, d7          *
93:      move.w  d7, PATSIZ(a6) *パターン1ラインのバイト数

```

```

94:
95:      move.w d2, SXL(a6)      *縮小元横幅-1
96:      move.w a2, DXL(a6)      *xループカウンタ初期値
97:      move.w a3, d7           *d7=yループカウンタ初期値
98:
99:      swap.w d7               *d7の下位ワードを待避
100:     move.w d2, d6
101:     move.w d3, d7
102:     addq.w #1, a2            *a2=パターン
103:     addq.w #1, a3
104:
105:     addq.w #1, d2            *x方向縮小率を約分する
106:     move.w d2, d1           *
107:     move.w a2, d4           *
108:     GCM d1, d4              *
109:     divu.w d0, d2           *分子
110:     move.l a2, d4           *
111:     divu.w d0, d4           *分母
112:
113:     addq.w #1, d3            *y方向縮小率を約分する
114:     move.w d3, d1           *
115:     move.w a3, d5           *
116:     GCM d1, d5              *
117:     divu.w d0, d3           *分子
118:     move.l a3, d5           *
119:     divu.w d0, d5           *分母
120:
121:     move.w d2, d0           *
122:     mulu.w d3, d0           *
123:     move.l d0, DENOM(a6)    *平均するサブピクセル数
124:
125:     move.w d2, d4
126:     subq.w #1, d2           *平均するサブピクセルの
127:     move.w d2, XSTEP(a6)    * 横方向の数-1
128:     move.w d3, d5
129:     subq.w #1, d3           *平均するサブピクセルの
130:     move.w d3, YSTEP(a6)    * 縦方向の数-1
131:
132:     move.w a2, d0           *xに関するパラメータ計算
133:     mulu.w d4, d0           *←疑似拡大
134:     subq.l #1, d0           *
135:     neg.l d0                *
136:     move.l d0, EX(a6)       *
137:     neg.l d0                *
138:     add.l d0, d0            *
139:     move.l d0, CEX(a6)      *
140:     move.w d6, d2           *
141:     add.l d2, d2            *
142:     move.l d2, DEX(a6)      *
143:
144:     move.w a3, d0           *yに関するパラメータ計算
145:     mulu.w d5, d0           *←疑似拡大
146:     subq.l #1, d0           *
147:     neg.l d0                *
148:     move.l d0, EY(a6)       *
149:     neg.l d0                *
150:     add.l d0, d0            *
151:     move.l d0, CEY(a6)      *
152:     move.w d7, d3           *
153:     add.l d3, d3            *
154:     move.l d2, DEY(a6)      *
155:
156:     bsr dergb               *最初のラインをrgbに分解
157:     movea.l a5, a2          *a2=rgbごとの累算用バッファ
158:
159:     move.l EY(a6), d5
160:     swap.w d7               *d7を復帰
161:
162: yloop: move.l a0, -(sp)
163:
164:     moveq.l #0, d0          *rgbごとの累算用バッファを
165:     movea.l a2, a5          * 0で初期化

```

```

166:      move.w DXL(a6), d4      *
167: cirlp: move.l d0, (a5)+      *
168:      move.l d0, (a5)+      *
169:      move.l d0, (a5)+      *
170:      dbra    d4, cirlp      *
171:
172:      move.w YSTEP(a6), d6
173: yloop2: movea.l a2, a4
174:      movea.l BUFF(a6), a5
175:
176:      swap.w d6
177:      swap.w d7
178:      movem.l d5/a1-a3, -(sp)
179:
180:      movem.w (a5)+, a0-a2    *a0~a2=参照する点のbrg
181:
182:      move.l DEX(a6), d3
183:      move.l CEX(a6), a3
184:      move.w XSTEP(a6), d5
185:
186:      move.l EX(a6), d4
187:      move.w DXL(a6), d7
188: xloop: moveq.l #0, d0        *B
189:      moveq.l #0, d1        *R
190:      moveq.l #0, d2        *G
191:      move.w d5, d6
192: xloop2: add.l a0, d0        *B
193:      add.l a1, d1        *R
194:      add.l a2, d2        *G
195:
196:      add.l d3, d4        *EX += DEX
197:      bmi     xnext2
198:      sub.l a3, d4        *EX -= CEX
199:      movem.w (a5)+, a0-a2  *参照元x座標を進める
200: xnext2: dbra    d6, xloop2
201:
202: xnext: add.l d0, (a4)+      *R
203:      add.l d1, (a4)+      *G
204:      add.l d2, (a4)+      *B
205:      dbra    d7, xloop
206:
207:      movem.l (sp)+, d5/a1-a3
208:      swap.w d7
209:      swap.w d6
210:
211:      add.l DEY(a6), d5    *EY += DEY
212:      bmi     ynext2
213:      sub.l CEY(a6), d5    *EY -= CEY
214:      lea.l GNBYTE(a1), a1 *参照元y座標を進める
215:
216:      move.w d6, d0        *処理完了寸前であれば
217:      or.w d7, d0          * この下のラインを
218:      beq     ynext        * rgbに分解する必要はない
219:
220:      bsr     dergb        *1ライン分rgbに分解する
221:
222: ynext2: dbra    d6, yloop2
223:
224: ynext: movea.l (sp)+, a0
225:
226:      movea.l a0, a3
227:      movea.l a2, a5
228:      move.l DENOM(a6), d1  *d1=平均の分母
229:
230:      move.w DXL(a6), d6
231: drawlp: move.l (a5)+, d0
232:      FPACK    _UDIV
233:      move.w d0, d2        *B
234:      move.l (a5)+, d0
235:      FPACK    _UDIV
236:      move.w d0, d3        *R
237:      move.l (a5)+, d0

```



```

238:      FPACK    _UDIV      #G
239:      move.w   d0, d4
240:      RGB      d2, d3, d4, d0    #カラーコードに構成して
241:      move.w   d0, (a3)+        #書き込む
242:      dbra     d6, drawlp       #横幅分繰り返す
243:
244:      lea.l    GNBYTE(a0), a0    #描画先y座標を進める
245:      dbra     d7, yloop        #高さ分繰り返す
246:
247: done:  movem.l (sp)+, d0-d6/a0-a5
248:      unlk     a6
249:      rts
250:
251: #
252: #      1ライン分rgbに分解して
253: #      バッファに展開する
254: dergb:
255:      movea.l  a1, a3
256:      movea.l  BUFF(a6), a5
257:      move.w   SXL(a6), d4
258: dergb0: move.w (a3)+, d0
259:      DERGB    d0, d1, d2, d3
260:      movem.w  d1-d3, (a5)
261:      addq.l   #6, a5
262:      dbra     d4, dergb0
263:      rts
264:
265:      .end

```

このプログラムでは任意の縮小率を実現するために、各ピクセルを縮小率の分子におうじて等分して細かなサブピクセルに分割したうえで、得られたサブピクセルを縦横分母の数だけ集めて平均をとっている。内部でいったん元画像を縮小率の分子倍し、それから分母で割ると思えばいい。これを力ずくの4重ループで処理している。

このプログラムにおける縮小率とは、縮小先の画像の大きさと縮小元の画像の大きさの比ということになるが、安直に縮小先の大きさを分子、縮小元の大きさを分母とみなしてそのまま処理するととんでもなく遅くなるのは想像がつくだろう。たとえば、512×512の範囲を256×256に縮小するときに、各ピクセルを縦横256等分し、1ピクセル描くごとにサブピクセルを512×512集めて平均をとっていたのでは日が暮れるどころの騒ぎではない。というわけで、冗談プログラムとはいえリスト9でも約分だけはちゃんとやっている。本筋ではないので細かくは触れないが、約分するのに必要な分子と分母の最大公約数はユークリッドの互除法と呼ばれるアルゴリズムで求めている。

で、ここまでいえば、このプログラムの使用上の注意は明らかだろう。“約分して分母が十分小さくなるような縮小率を選ばないと死ぬほど遅い”のだ。512×512の範囲を511×512に縮小するのに13分かかった。この場合、y方向については約分により縮小率1になるが、x方向の縮小率は511/512のままで、1点ごとに512個のサブピクセルの平均をとっていることになる。間違っても512×512の範囲を511×511に縮小するようなことはしないように。もしどうしてもそんなことをしたいのであれば（誰もしないって）、多少の誤差は出るかもしれないがx方向とy方向を個別に511/512に縮小して回避しよう。

ちなみに、リスト10は少しの修正で拡大にも使える。85～88行のチェックを外し、161行の

空行を

```
lea.l    $c00000,a0
```

に変更する。この変更により拡大後の画像の描画位置は画面の左上に固定される。後は、拡大後の画像が拡大元の画像をへたに上書きしてしまわないよう、元データをなるべく画面の右隅においてサブルーチンを呼び出せばよい。整数倍するときにはただのモザイクになるのでおもしろくもなんともないが、3/2倍などの非整数の倍率を指定した場合は（モザイクっぽさは残るものの）ある程度、色を補間する。

では、最後になったが、gxputの動作試験用ルーチンをリスト11に示す。あらかじめ65536色の画像を画面にロードしてから実行すると、画面中央の128×128の範囲を切り出し、256×256ピクセルモードでランダムな縮小・拡大率で描き続ける。ついでにリスト12にglto2とgshrinkのテストプログラムを示しておこう。画像をロードしてから走らせると、まず、gshrinkで3/4に縮小し、それをさらにglto2で1/2にする。

リスト11
GXPUTTEST.S

```
1: #      gxputの動作試験用プログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: #
6:      .xref      gxput
7: #      .xref      gxputon
8: #      .xref      setcliprect
9: #
10: FPACK macro      callno
11:      .dc.w      callno
12:      .endm
13: #
14: __RAND      equ      $fe0e
15: #
16:      .text
17:      .even
18: #
19: ent:
20:      lea.l      inisp, sp
21:
22:      clr.l      -(sp)          #スーパーバイザモードへ
23:      DOS      _SUPER          #
24:
25:      lea.l      arg0, a1      #画面からパターンを取り込む
26:      IOCS      _GETGRM        #
27:
28: #      pea      window          #クリッピングウィンドウの設定
29: #      jsr      setcliprect      #
30: #      addq.l      #4, sp          #
31:
32:      moveq.l      #14, d1      #256x256, 65536
33:      IOCS      _CRTMOD          #画面初期化
34:      IOCS      _G_CLR_ON        #
35:
36:      lea.l      arg, a1          #a1 = 引数受け渡し領域
37: loop:      movea.l      a1, a0      #乱数で描画範囲を決める
38:      moveq.l      #4-1, d1      #
39: loop2:      FPACK      __RAND      #
40:      lsr.w      #7, d0          #0 ≤ d0 ≤ 255
41:      move.w      d0, (a0) +      #
42:      dbra      d1, loop2        #
43:
44:      pea.l      (a1)          #描画
45:      jsr      gxput          #
46:      addq.l      #4, sp          #
```

```

47:
48:      DOS      _KEYSNS      #キーが押されるまで
49:      tst.l    d0           # 繰り返す
50:      beq      loop        #
51:
52:      DOS      _INKEY       #スペースが押されたら
53:      cmpi.b   #$20,d0      # 一時停止する
54:      bne      done        #
55: pause:  DOS      _INKEY       #
56:      cmpi.b   #$20,d0      #
57:      beq      loop        #
58:
59: done:   move.w  #-1,-(sp)    #キーを読み捨てる
60:      DOS      _KFLUSH      #
61:
62:      move.l    #$0010_0000,-(sp) #width 96
63:      DOS      _CONCTRL     #画面再初期化
64:      DOS      _EXIT
65: #
66:      .data
67:      .even
68: #
69: HSIZE equ 128
70: VSIZE equ 128
71: #
72: arg:   #gxputの引数
73:      .dc.w    0,0,511,511
74:      .dc.l    pat
75:      .dc.w    HSIZE-1,VSIZE-1
76:      .dc.l    temp
77: #
78: arg0:  #LOCS GETGRMの引数
79:      .dc.w    256-HSIZE/2,256-VSIZE/2
80:      .dc.w    255+HSIZE/2,255+VSIZE/2
81:      .dc.l    pat
82:      .dc.l    pate
83: #
84: window: #クリッピングウィンドウ
85:      .dc.w    32,32,255-32,255-32
86: #
87:      .bss
88:      .even
89: #
90: pat:   .ds.w    HSIZE*VSIZE #描画パターン
91: pate:
92: temp:  .ds.w    512*3       #gxputのワーク
93: #
94:      .stack
95:      .even
96: #
97:      .ds.l    4096
98: inisp:
99:      .end     ent

```

.....
リスト12
G1TO2TEST.S

```

1: #      glto2, gshrinkの動作試験用プログラム
2:
3:      .include      doscall.mac
4: #
5:      .xref      glto2
6:      .xref      gshrink
7: #
8:      .text
9:      .even
10: #
11: ent:
12:      lea.l      inisp,sp
13:
14:      clr.l      -(sp)      #スーパーバイザモードへ
15:      DOS      _SUPER      #

```

```

16:
17:     pea.l   arg          *gshrinkで3/4に縮小
18:     jsr     gshrink      *
19:     addq.l  #4, sp       *
20:
21:     pea.l   argl         *glto2でさらに1/2に縮小
22:     jsr     glto2        *
23:     addq.l  #4, sp       *
24:
25:     DOS     _EXIT
26: *
27:     .data
28:     .even
29: *
30: arg:
31:     .dc.w   0, 0, 511, 511    *gshrinkの引数
32:     .dc.w   383, 383         *3/4
33:     .dc.l   temp
34: *
35: argl:
36:     .dc.w   0, 0, 511, 511    *glto2の引数
37: *
38:     .bss
39:     .even
40: *
41: temp: .ds.w  1024*3          *gshrinkのワーク
42: *
43:     .stack
44:     .even
45: *
46:     .ds.l   4096
47: inisp:
48:     .end    ent

```

C
H
A
P
T
E
R

A

画像の回転

画像の回転

前章の拡大・縮小に引き続き、この章ではグラフィックパターンの回転を取り上げる。回転機能つきのプットルーチンを作成してみよう。

画像回転のアルゴリズム

理屈の面から攻める。図1を見てもらいたい。この図から、点 (x, y) を原点について角度 θ だけ回転する座標変換式が得られる。移動先の点 (x', y') は次の式で与えられることがわかる。

$$x' = x \times \cos \theta - y \times \sin \theta$$

$$y' = x \times \sin \theta + y \times \cos \theta$$

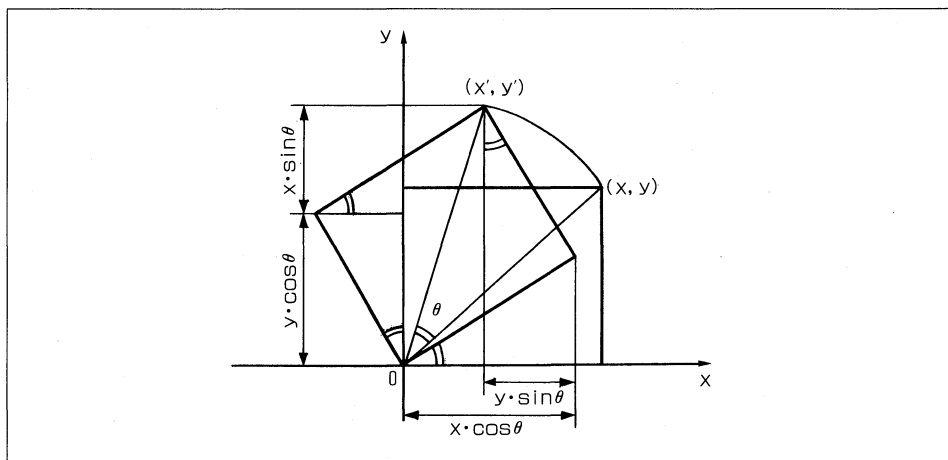


図1
回転時の座標変換

さっそくこの式を使ってX-BASICで回転ルーチンを書いてみた。リスト1だ。リスト1は画面左上に描いた64ピクセル四方の適当な絵を30°回転して画面中央に描き直す。回転に先立って座標から32を引いている点と、回転角度が30°ではなく-30°になっている点をチェック

しておいてほしい。前者は回転の中心を画像の中央にとるためで、後者はグラフィック画面と数学の座標系とではy座標軸の向きが逆になっていることを反映している。

.....
リスト1
ROTATE1.BAS

```
10 int x,y
20 float c,s
30 /*
40 screen 0,3,1,1
50 fill (0,0,99,99,rgb(8,8,8))
60 for i=0 to 31
70   fill(i,i,63-i,63-i,hsv(i*6,31,31))
80 next
90 /*
100 c=cos(-30*pi()/180)
110 s=sin(-30*pi()/180)
120 /*
130 for y=0 to 63
140   for x=0 to 63
150     x0=c*(x-32)-s*(y-32)+32
160     y0=s*(x-32)+c*(y-32)+32
170     pset(x0+128-32,y0+128-32,point(x,y))
180   next
190 next
```

リスト1は大筋ではうまく動く。だが、回転後の画像の質はあまりよくない。いびつに歪むし、黒い点がポツポツと残る。

歪みの原因はピクセルの縦横比にある。表示画面が512×512や256×256のとき、ピクセルのかたちは縦横比約2:3の横長であり、見た目に自然な回転結果を得るにはこの点を考慮する必要がある。具体的には、x座標を1.5倍してから回転の計算をし、回転後、得られたx座標を2/3に再修正する。

黒い点が残るのは、グラフィック画面上では斜めの線が正確には表現できず、階段状になるためだ。グラフィック画面上に斜線を並べて面にするときには、その段がきれいに噛みあわない部分に隙間が生まれる。この穴は前章の拡大・縮小のときと同じ逆転の発想で埋めることができる。(x, y) を回転して (x', y') を求める代わりに、(x', y') を逆回転することで対応する変換元の位置 (x, y) を求めるという考え方だ。元画像を傾けて張りつけるのではなく、回転後水平になるよう画像を切り出してくると思えばよい (図2)。この場合、回転によってはみ出す角の部分は描画されず、そのかわり元画像外周の余白もいっしょに回転して描かれること

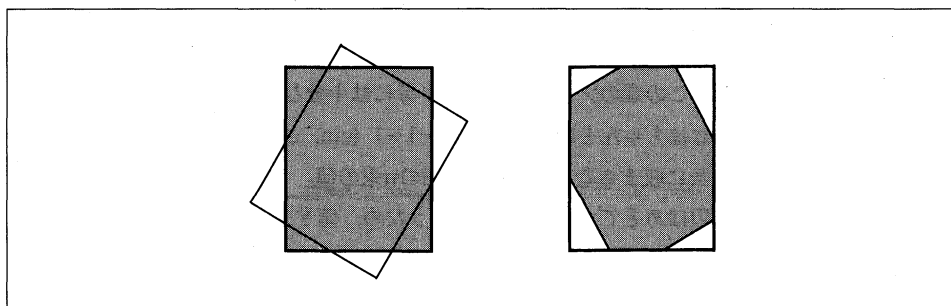


図2
動作結果イメージ

になる。プットルーチンにおいて余白はグラフィックパターンを格納したメモリの外側を意味するから、実際のプログラムでは余白ができないよう、表示するサイズよりもグラフィックパターンをひと回り大きく用意することになるだろう。

以上2点に注意してつくり直したのがリスト2だ。隙間は埋まり、歪みもだいぶなくなっている。ピクセルの縦横比が正確に2:3ではないため補正が不十分（というかやりすぎ）でいびつさが消えてはいないが、なんとか許容範囲だと思う。気になるなら、より自然な結果が得られるよう補正値を微調整してみてもらいたい。

リスト2
ROTATE2.BAS

```
10 int x, y, rx=3, ry=2
20 float c, s
30 /*
40 screen 0, 3, 1, 1
50 fill (0, 0, 99, 99, rgb(8, 8, 8))
60 for i=0 to 31
70   fill (i, i, 63-i, 63-i, hsv (i*6, 31, 31))
80 next
90 /*
100 c=cos (30*pi ()/180)
110 s=sin (30*pi ()/180)
120 /*
130 for y=0 to 63
140   for x=0 to 63
150     x0=ry*(rx*c*(x-32)/ry-s*(y-32))/rx+32
160     y0=rx*s*(x-32)/ry+c*(y-32)+32
170     pset (x+128-32, y+128-32, point (x0, y0))
180   next
190 next
```

さて、リスト2は、実数、とくに三角関数を使っている部分をどうにかしないことには、まだマシン語プログラムに書き直せそうもない。三角関数の値を求めるのにFLOATn.Xのファンクションコールを利用するという逃げ道もあるが、それではマシン語で書くメリットがなくなる。どうせやるなら、三角関数周りから何からすべて整数演算ですませたいところだ。そこで、今回は整数化した三角関数のテーブルを先につくっておくという方法を採用。

テーブル部分だけを先に示しておく。リスト3だ。リスト4のX-BASICプログラムの出力を整形して作成した。0°～90°まで1°きざみで $\sin \theta$ を4000_H倍した値を並べてある。 $\sin$ の値を小数部14ビットの固定小数点で表現しているといういい方もできる。 $\sin \theta$ 倍するときには θ に対応する値をこのテーブルから取り出して、乗じたのち、4000_Hで割る。あくまで近似計算だが、この章のプログラムで使う分には十分な精度が出る。ここで、4000_Hという数字を選んだのにはちゃんと理由がある。-1～1 ($\sin$, $\cos$ の値域)の値とかけあわせても16ビット符号付き数に収まるなるべく大きな2の n 乗の値、ということで決めた。16ビット符号付き数に収めるのはあとで muls を有効に使うため、値を大きくするのは精度を上げるため、そして、2の n 乗にするのは除算がシフトですむようにするためだ¹⁾。

■ 1) 4000_Hではなく8000_H倍を採用してさらに精度を稼ぐという選択肢もある。この場合、 $\sin 90^\circ$ だけは8000_H倍するとオーバーフローして負の数に化けてしまうが、値が1だというのがわかっているのだから、特別扱いすればよい。

リスト3
SINTABLE.S

```

1: #      sin 0° ~ 90° の表 (16384倍)
2:
3:      .xdef  sintable
4: #
5:      .text
6:      .even
7: #
8: sintable:
9:      .dc.w  00000,00286,00572,00857,01143  #0
10:     .dc.w  01428,01713,01997,02280,02563
11:     .dc.w  02845,03126,03406,03686,03964  #10
12:     .dc.w  04240,04516,04790,05063,05334
13:     .dc.w  05604,05872,06138,06402,06664  #20
14:     .dc.w  06924,07182,07438,07692,07943
15:     .dc.w  08192,08438,08682,08923,09162  #30
16:     .dc.w  09397,09630,09860,10087,10311
17:     .dc.w  10531,10749,10963,11174,11381  #40
18:     .dc.w  11585,11786,11982,12176,12365
19:     .dc.w  12551,12733,12911,13085,13255  #50
20:     .dc.w  13421,13583,13741,13894,14044
21:     .dc.w  14189,14330,14466,14598,14726  #60
22:     .dc.w  14849,14968,15082,15191,15296
23:     .dc.w  15396,15491,15582,15668,15749  #70
24:     .dc.w  15826,15897,15964,16026,16083
25:     .dc.w  16135,16182,16225,16262,16294  #80
26:     .dc.w  16322,16344,16362,16374,16382
27:     .dc.w  16384  #90
28:
29:     .end

```

リスト4
SIN.BAS

```

10 int fp, i, s
20 /*
30 fp = fopen("sintable.s", "c")
40 d = 0
50 for i = 0 to 90
60     s = int(sin(pi()*i/180)*16384+0.5#)
70     fwrites(itoa(s)+chr$(13)+chr$(10), fp)
80 next
90 fclose(fp)

```

リスト3にはsinだけ、それも90°までの狭い範囲の値しか用意していない。cosを求める場合や角度が0°~90°の範囲外有的时候には、三角関数の性質を利用して対応する。cosはsinの位相を90°ずらしたものだから、

$$\cos\theta = \sin(90^\circ - \theta)$$

により得られる。また、角度 θ は、

$$\sin(-\theta) = -\sin\theta$$

$$\sin(180^\circ - \theta) = \sin\theta$$

の関係から0~90°に収めることができる。たとえば、

$$\sin(-150^\circ) = -\sin(150^\circ)$$

$$= -\sin(180^\circ - 30^\circ)$$

$$= -\sin 30^\circ$$

となる。もっとも、ケチらずに360°をカバーしたsinとcosのテーブルを別々に用意したほう

がプログラムのつくりはかんたんになるし、実行速度もいくらか上がる。ただ、このテーブルは1回の回転ごとにわずか2回 (sin と cos1回ずつ) しか参照されないから、こんなところの実行速度を上げたところでプログラム全体の性能にはほとんど貢献しない。ここは、実行速度よりはメモリ効率を優先する場面だろう。

sin, cosが求められるようになったことで、マシン語で回転ルーチンを書く目途が立った。さらにマシン語化に向けて、アルゴリズムの効率向上を目指す。リスト2のX-BASICプログラムでは1ピクセルごとに回転の計算をしているが、この計算回数をなるべく減らす方向で考えてみる。

斜めに点を拾って水平に並べることで回転を実現していることに注目する。水平1ラインを描く過程で拾う点は (グラフィック画面で表現できる精度の) 直線上に位置しているわけだ。ということは、始めの点と最後の点が決まれば、Bresenhamの線分発生アルゴリズムの応用で、そのあいだのどの点を拾ったらよいか²⁾が求まる。しかも、各水平ラインに対応する元画像上の参照開始位置もまた直線上に並ぶから、やはり線分発生アルゴリズムにより順次求めることができる。結局、回転の計算は矩形領域の角3点に対してだけ行えば十分だということがわかる。この様子を図3に示した。ここで、図3の点Bからbへの回転による移動距離とDからdへの移動距離は符号が違っただけで“ほぼ”等しい²⁾ことを利用すると、一方の回転計算を手抜きできる点も指摘しておこう。

■2) 数学上は等しいのだが、グラフィック画面上では座標は整数値しかとりえないため、回転するパターンの大きさが偶数ピクセルの場合、回転の中心となるピクセルを決める段階で誤差が出る。

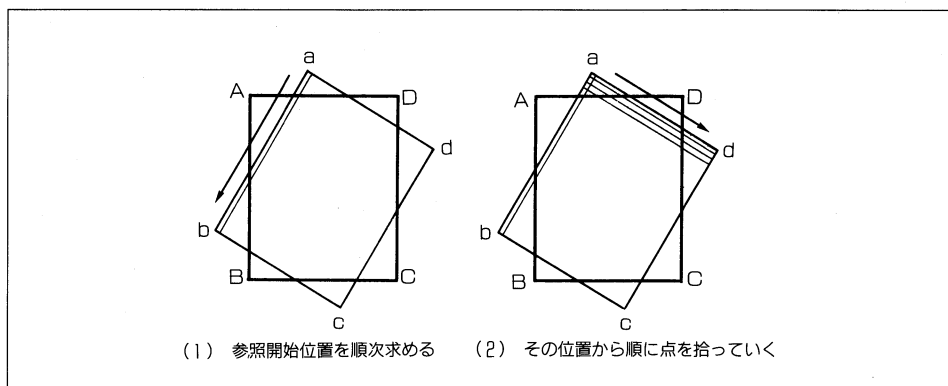


図3
線分発生アルゴリズムの応用

さて、全ピクセルについて回転の計算をしていたのが、2.5点分の計算ですむところまで簡略化できた。しかし、この考え方を実用に供するには最後にもうひと工夫いる。グラフィック画面上では数学的な意味での線分の長さ²⁾とピクセル数が一致せず、斜めの線分は同じ長さの水平/垂直線分よりもピクセル数が少ないのだ。元画像を斜めに拾っても、描画先の水平ラインを埋めるだけのピクセル数がない。したがって、斜めに拾った点の集まりを細長いパターンに

見立て、拡大しつつ描画する必要がある。

最終的に、回転プログラムは、斜めに点を拾うのに Bresenham のアルゴリズムを利用し、それを拡大するのにも Bresenham のアルゴリズムを使い、これを縦方向と横方向に独立して行うという構成で実現される。

回転ルーチンの実例

では、回転機能つきプットルーチンをリスト5に示そう。回転処理過程でのパターン引き伸ばし部分に少々融通を効かせた結果、回転だけではなく拡大・縮小も同時に行うことができる贅沢なサブルーチンになった。どのみち拡大の処理は必要なのだから、このように拡張してもプログラムの複雑さは変わらない。ただ、拡大・縮小はあくまでおまけにすぎず、演算過程で比較的可以たんにオーバーフローを起こすので、あまり極端に大きくは拡大できない（画面の数倍までならゆとりだが）。

リスト5
GRPUT.S
(第1版)

```

1: #      パターンを回転/拡大/縮小してプットする
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      grput
7:      .xref      gramadr
8:      .xref      ucliprect
9:      .xref      sintable
10: #
11:      .offset 0      *grputの引数構造
12: #
13: X0:      .ds.w      1      *描画先座標
14: Y0:      .ds.w      1      *
15: X1:      .ds.w      1      *
16: Y1:      .ds.w      1      *
17: DEG:      .ds.w      1      *回転角度 (°)
18: PAT:      .ds.l      1      *パターンアドレス (※)
19: XL:      .ds.w      1      *パターンの横の長さ-1 (※)
20: YL:      .ds.w      1      *パターンの縦の長さ-1 (※)
21: TEMP:      .ds.l      1      *作業用メモリへのポインタ
22: #
23: # (※)
24: #
25: #
26: #
27: #
28: #
29: #
30: #
31: #
32: #
33:      .text
34:      .even
35: #
36: grput:
37: ARGPTR = 4+8*4+7*4
38:      movem.l d0-d7/a0-a6, -(sp)
39:
40:      movea.l ARGPTR(sp), a6      *a6 = 引数列
41:
42:      bsr      rotate      *回転の計算をする

```

```

43:
44:      movea.l TEMP(a6), a5
45:      movea.l a5, a4
46:      movem.w d0-d1/d4-d5, -(sp)
47:      bsr     line_comp      *水平1ライン分の
48:                               * 点の拾い方を決める
49:
50:      lea.l    ucliprect, a0  *左右端の
51:      move.w   X0(a6), d1     * クリッピングをする
52:      move.w   X1(a6), d2     *
53:      bsr     clip            *
54:      bmi     done2          *
55:
56:      movea.l  a5, a2
57:      bsr     expand_comp     *水平1ライン分の
58:                               * 拡大/縮小を考慮した
59:                               * 点の拾い方を決める
60:
61:      movea.l  a5, a4
62:      movem.w  (sp)+, d0-d3
63:      bsr     line_comp      *垂直1ライン分の
64:                               * 点の拾い方を決める
65:
66:      lea.l    ucliprect+2, a0 *上下端の
67:      move.w   Y0(a6), d1     * クリッピングをする
68:      move.w   Y1(a6), d2     *
69:      bsr     clip            *
70:      bmi     done           *
71:
72:      movea.l  a5, a3
73:      bsr     expand_comp     *垂直1ライン分の
74:                               * 拡大/縮小を考慮した
75:                               * 点の拾い方を決める
76:
77:      move.w   (a2)+, d0      * (d0, d1) = 描画開始位置
78:      move.w   (a3)+, d1      *
79:      jsr     gramadr        *a0 = そのG-RAM上アドレス
80:
81:      move.w   (a2)+, d7      *d7 = 描画範囲横方向の長さ-1
82:      move.w   (a3)+, d5      *d5 = 描画範囲縦方向の長さ-1
83:
84:      move.w   d7, d0         *d0 =
85:      addq.w   #1, d0         * ライン右端と
86:      add.w    d0, d0         * 次のライン左端との
87:      neg.w    d0             * G-RAM上のアドレスの差
88:      addi.w   #GNBYTE, d0    *
89:
90: yloop: movea.l a1, a4         *a4 = 参照するパターン上位演
91:      movea.l  a2, a5         *a5 = 1ライン分の参照位置の
92:                               * 移動量のテーブル
93:
94: xloop: move.w  (a4), (a0)+    *1点描画する
95:      adda.w    (a5)+, a4      *参照位置を移動する
96:      dbra     d6, xloop      *横幅分繰り返す
97:
98:      adda.w    (a3)+, a1      *参照位置を縦に移動する
99:      adda.w    d0, a0         *a0 = 次のライン左端
100:     dbra     d5, yloop      *高さ分繰り返す
101:
102: done:  movem.l (sp)+, d0-d7/a0-a6
103:      rts
104: done2: addq.l  #8, sp
105:      bra     done
106:
107: *
108: *      sin(DEG)をテーブルから引いてくるマクロ
109: *
110: ISIN  macro TABLE, DEG
111:      local skip1, skip2, skip3
112:
113:      ext.l    DEG            *sin(-θ) = -sin(θ)
114:      bpl     skip1           *
115:      neg.w    DEG            *
116:
117: skip1: subi.w  #90, DEG      *sin(180°-θ) = sin(θ)

```

```

118:      bcs      skip2      *
119:      neg.w    DEG        *
120: skip2: addi.w  #90, DEG   *0 ≤ deg ≤ 90°
121:
122:      add.w    DEG, DEG
123:      move.w   0(TABLE, DEG), DEG
124:
125:      tst.l    DEG         *sin(-θ) = -sin(θ)
126:      bpl      skip3      *
127:      neg.w    DEG        *
128: skip3:
129:      .endm
130:
131: *
132: *      点(X, Y)を回転するマクロ
133: *
134: ROTP macro X, Y
135:      local    skip1, skip2
136:
137:      move.w   X, d6
138:      move.w   Y, d7
139:
140:      muls.w   d5, X        *x*cos(θ)
141:      muls.w   d5, Y        *y*cos(θ)
142:      muls.w   d4, d6       *x*sin(θ)
143:      muls.w   d4, d7       *y*sin(θ)
144:
145:      sub.l    d7, X        *x*cos(θ)-y*sin(θ)
146:      bpl      skip1
147:      addi.l   #0x3fff, X
148: skip1: asl.l  #2, X
149:      swap.w   X
150:
151:      add.l    d6, Y        *x*sin(θ)+y*cos(θ)
152:      bpl      skip2
153:      addi.l   #0x3fff, Y
154: skip2: asl.l  #2, Y
155:      swap.w   Y
156:
157:      add.w    a2, X
158:      add.w    a3, Y
159:
160:      add.w    X, X        *x座標を2/3にする
161:      add.w    X, X        *
162:      addq.w   #3, X        *
163:      ext.l    X           *
164:      divs.w   #6, X        *
165:
166:      .endm
167:
168: *
169: *      パターンの角3点を回転する
170: *
171: rotate:
172:      move.w   DEG(a6), d4   *d4 = 角度
173:
174:      move.w   #360, d6      *角度を
175:      move.w   #180, d7      * -180° ~ 179° に
176:      add.w    d7, d4        * 正規化する
177:      bpl      norm2
178:      add.w    d6, d4
179:      bmi      norm1
180:      cmp.w    d6, d4
181:      bcs      norm3
182:      sub.w    d6, d4
183:      bra      norm2
184:      norm3: sub.w    d7, d4
185:
186:      moveq.l  #90, d5       *cos θ = sin(90° - θ)
187:      sub.w    d4, d5
188:      cmp.w    d7, d5
189:      ble      rot
190:      sub.w    d6, d5
191:
192: rot:   lea.l  sintable, a0

```

```

193:      ISIN    a0, d4      *d4 = sin(θ)
194:      ISIN    a0, d5      *d5 = cos(θ)
195:
196:      movem.w XL(a6), d0-d1  *d0, d1 = パターンの縦横の大きさ
197:
198:      move.w   d0, a5
199:
200:      add.w    d0, d0      *x方向の長さを1.5倍する
201:      add.w    a5, d0      *
202:      lsr.w    #1, d0      *
203:
204:      move.w   d0, d2
205:      move.w   d1, d3
206:
207:      lsr.w    #1, d0      *(d0, d1) = 回転の中心
208:      lsr.w    #1, d1      *
209:
210:      move.w   d0, a2      *(a2, a3) = 回転の中心
211:      move.w   d1, a3      *
212:
213:      sub.w    d0, d2      *(d0, d1) = パターン中心を
214:      neg.w    d0          * 原点とする左上隅の座標
215:      neg.w    d1          *(d2, d1) = パターン中心を
216:                        * 原点とする右上隅の座標
217:
218:      move.w   d1, a4
219:      ROTP     d0, d1      *左上の点を回転する
220:
221:      exg.l    d1, a4      *(d2, d1) = パターン中心を
222:                        * 原点とする右上隅の座標
223:      ROTP     d2, d1      *右上の点を回転する
224:
225:      move.w   a5, d4      *左下の点を回転する
226:      sub.w    d2, d4      *
227:      move.w   d3, d5      *
228:      sub.w    d1, d5      *
229:
230:      move.w   d1, d3
231:      move.w   a4, d1
232:
233: *      左上      (d0, d1)
234: *      右上      (d2, d3)
235: *      左下      (d4, d5)
236:
237:      movea.l  PAT(a6), a1  *(d0, d1)に対応する
238:      adda.w   d0, a1      * パターン上の位置を
239:      adda.w   d0, a1      * 求める
240:      move.w   a5, d6      *
241:      addq.w   #1, d6      *
242:      add.w    d6, d6      *
243:      add.w    d6, d6      *
244:      muls.w   d1, d6      *
245:      adda.l   d6, a1      *
246:      rts
247:
248: *
249: *      パターン上の線分 (d0, d1) - (d2, d3) に沿って
250: *      ポインタを移動する相対アドレスの表を作成する
251: *
252: line_comp:
253:      sub.w    d0, d2
254:      move.w   d2, d4
255:      ABS      d2
256:      SGN      d4
257:
258:      sub.w    d1, d3
259:      move.w   d3, d5
260:      ABS      d3
261:      SGN      d5
262:
263:      add.w    d4, d4
264:      move.w   XL(a6), d0
265:      addq.w   #1, d0
266:      add.w    d0, d0
267:      add.w    d0, d0

```

```

268:      muls.w  d0, d5
269:
270:      cmp.w   d3, d2
271:      bcs     yline
272:
273: xline: move.w  d2, d0      *傾きが緩やかな線分の場合
274:
275:      move.w  d2, d1      *Bresenhamのアルゴリズムに
276:      neg.w   d1          * 必要なパラメータの計算
277:      move.w  d2, d6      *
278:      add.w   d2, d2      *
279:      add.w   d3, d3      *
280:
281: xline0: move.w  d4, d7      *アドレスの相対的な移動量を
282:      add.w   d3, d1      *
283:      bmi     xline1      *
284:      add.w   d5, d7      *
285:      sub.w   d2, d1      *
286: xline1: move.w  d7, (a5)+   * テーブルに登録していく
287:      dbra    d6, xline0
288:      rts
289:
290: yline: move.w  d3, d0      *傾きが急な線分の場合
291:
292:      move.w  d3, d1      *Bresenhamのアルゴリズムに
293:      neg.w   d1          * 必要なパラメータの計算
294:      move.w  d3, d6      *
295:      add.w   d2, d2      *
296:      add.w   d3, d3      *
297:
298: yline0: move.w  d5, d7      *アドレスの相対的な移動量を
299:      add.w   d2, d1      *
300:      bmi     yline1      *
301:      add.w   d4, d7      *
302:      sub.w   d3, d1      *
303: yline1: move.w  d7, (a5)+   * テーブルに登録していく
304:      dbra    d6, yline0
305:      rts
306:
307: #
308: #      クリッピングする
309: #
310:      .offset 0
311: #
312: Min:  .ds.w   1          #MINX  MINX
313:      .ds.w   1          #MINY  MINY
314: Max:  .ds.w   1          #MAXX  MAXX
315:      .ds.w   1          #MAXY  MAXY
316: #
317:      .offset 0
318: #
319: DOSAV: .ds.w   1
320: D2SAV: .ds.w   1
321: NPIX:  .ds.w   1
322: #
323:      .text
324: #
325: clip: MINMAX  d1, d2      *d1 ≤ d2を保証する
326:
327:
328:      addi.w  #$8000, d1    *ゲタを履かせる
329:      addi.w  #$8000, d2    *
330:      moveq.l #0, d3
331:      movem.w d0/d2-d3, -(sp)
332: minclip:
333:      move.w  Min(a0), d6
334:      cmp.w   d6, d1
335:      bcc     maxclip      *ウィンドウ内だった
336:
337:      cmp.w   d6, d2
338:      bcc     min0
339:      bne     outofscrn    *完全にウィンドウ外
340:
341:      move.w  d2, d1
342:      bra     maxclip

```

```

343:
344: min0: moveq.l #0, d3
345: minlp: move.w d1, d7
346:        add.w d2, d7
347:        roxr.w #1, d7
348:        cmp.w d6, d7
349:        beq min2
350:        bcs min1
351:
352:        move.w d7, d2
353:        add.w d3, d0
354:        roxr.w #1, d0
355:        bra minlp
356:
357: min1: move.w d7, d1
358:        add.w d0, d3
359:        roxr.w #1, d3
360:        bra minlp
361:
362: min2: move.w d7, d1
363:        add.w d3, d0
364:        roxr.w #1, d0      *d0 = ウィンドウ外にはみ出す
365:                                * パターンのピクセル数
366:        sub.w d0, NPIX(sp)
367:
368:        subq.w #1, d0
369:        bcs min3
370: skiplp: adda.w (a4)+, a1    *切り捨てた分
371:        dbra d0, skiplp     * 参照開始位置を
372:                                * ずらす
373: min3: move.w DOSAV(sp), d0
374:        move.w D2SAV(sp), d2
375:
376: maxclip:      *右端/下端でクリップ
377:        move.w Max(a0), d6
378:        cmp.w d6, d2
379:        bls clipped      *ウィンドウ内だった
380:
381:        cmp.w d6, d1
382:        bls max0
383:        bne outofscrn     *完全にウィンドウ外
384:
385:        moveq.l #0, d0
386:        move.w d1, d2
387:        bra clipped
388:
389: max0: move.w d1, d4
390:        moveq.l #0, d3
391: maxlp: move.w d4, d7
392:        add.w d2, d7
393:        roxr.w #1, d7
394:        cmp.w d6, d7
395:        beq max2
396:        bcs max1
397:
398:        move.w d7, d2
399:        add.w d3, d0
400:        roxr.w #1, d0
401:        bra maxlp
402:
403: max1: move.w d7, d4
404:        add.w d0, d3
405:        roxr.w #1, d3
406:        bra maxlp
407:
408: max2: move.w d7, d2
409:        add.w d3, d0
410:        roxr.w #1, d0
411:
412: clipped:      *クリッピング完了
413:        subi.w #$8000, d1    *ゲタを脱がせる
414:        subi.w #$8000, d2    *
415:
416:        addq.l #4, sp

```



```

417:      add.w    (sp)+, d0      *d0 = パターン上の
418:      *        点を拾う斜め線分長
419:      rts              *N = 0
420:
421: outofscrn:
422:      addq.l    #6, sp
423:      moveq.l   #-1, d0      *N = 1
424:      rts
425:
426: *
427: *      斜めに切り出したパターンを
428: *      描画先の大きさに合わせて拡大/縮小するための
429: *      テーブルを作成する
430: *
431: expand_comp:
432:      move.w    d1, (a5)+      *クリッピング後の
433:      *        描画先左上隅x/y座標
434:      sub.w     d1, d2
435:      move.w    d2, (a5)+      *描画幅/高さ
436:      beq       exdone
437:
438:      move.w    d2, d1      *Bresenhamのアルゴリズムに
439:      neg.w     d1          * 必要なパラメータの計算
440:      move.w    d2, d3
441:      add.w     d2, d2
442:      add.w     d0, d0
443:
444: explp1: moveq.l #0, d4      *アドレスの相対的な移動量を
445:      add.w     d0, d1
446:      ble      expnxt
447: explp2: add.w   (a4)+, d4
448:      sub.w     d2, d1
449:      bgt      explp2
450: expnxt: move.w  d4, (a5)+    * テーブルに登録していく
451:      dbra     d3, explp1
452:
453: exdone: rts
454:
455:      .end

```

リスト5のサブルーチンgrputには13～21行のような構造で用意した引数列へのポインタをスタックに積んで渡す。パターンとその大きさの指定のしかたにはちょっと癖がある。パターンは余白を計算に入れて、実際に描く範囲の縦横2倍の大きさに用意するのだが、引数中のXL, YLは2倍しない値で表す。また、PATはパターンの先頭アドレスではなく、“実際に描く範囲の左上隅”を指す仕様になっている。この関係はリスト中の23～31行に示した。あと、少々手抜きがあるので、パターンの縦横の大きさはなるべく近い値でなければならない。このプログラムはパターンが細長いと回転の過程でパターン外のメモリを平気でアクセスしてしまうのだった。

末尾の引数TEMPは作業用のメモリへのポインタだ。grputは処理速度向上のため、前章の拡大・縮小ルーチン同様、描画に先立ってテーブルを作成する。作業用に使うメモリ量は描画範囲によっても変わるが、後で施す高速化を計算に入れても、256×256の範囲に描画する場合で、4Kバイトもあればよい。

grputの引数については、リスト6に用意した動作試験用のプログラムも参考にしてもらいたい。リスト6は適当な65536色モードの画像を画面に読み込んでから実行すると、中央の256ピクセル四方を取り込み、1°単位で回転させる（仕様上、実際に描かれるのは128ピクセル四方）。リスト6中、66行の表示範囲を適当に変更すれば、拡大・縮小の動作が確認できる。

リスト6
GRPUTTEST.S

```

1: #      grputの動作試験用プログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: #
6:      .xref      grput
7: #      .xref      setcliprect
8: #
9:      .text
10:     .even
11: #
12: ent:
13:     lea, l      inisp, sp
14:
15:     clr, l      -(sp)          #スーパーバイザモードへ
16:     DOS         _SUPER        #
17:
18:     lea, l      arg0, a1       #画面からパターンを取り込む
19:     IOCS        _GETGRM       #
20:
21: #      pea      window         #クリッピングウィンドウの設定
22: #      jsr      setcliprect    #
23: #      addq, l   #4, sp        #
24:
25:     moveq, l    #14, d1        #256x256.65536
26:     IOCS        _CRTMOD        #画面初期化
27:     IOCS        _G_CLR_ON     #
28:
29:     moveq, l    #1, d1         #dl = 角度増分
30:
31:     lea, l      arg, a1        #a1 = 引数受け渡し領域
32:     pea, l      (a1)          #描画
33:     jsr         grput         #
34:     addq, l     #4, sp        #
35:
36:     add, w      d1, 8(a1)      #角度を増減する
37:
38:     DOS         _KEYSNS       #キーが押されるまで
39:     tst, l      d0            # 繰り返す
40:     beq         loop         #
41:
42:     neg, w      d1            #回転方向反転
43:
44:     DOS         _INKEY        #スペースが押されたら
45:     cmpi, b     $$20, d0      # 一時停止する
46:     bne         done         #
47:     pause:     DOS         _INKEY
48:     cmpi, b     $$20, d0
49:     beq         loop         #
50:
51:     done:     move, w      #-1, -(sp)    #キーを読み捨てる
52:     DOS         _KFLUSH       #
53:
54:     move, l     $$0010_0000, -(sp)      #width 96
55:     DOS         _CONCTL       #画面再初期化
56:     DOS         _EXIT
57: #
58:     .data
59:     .even
60: #
61:     HSIZE      equ      128
62:     VSIZE      equ      128
63: #
64:     arg:
65: #      .dc, w    96, 96, 159, 159    #縮小
66: #      .dc, w    64, 64, 191, 191    #等倍
67: #      .dc, w    32, 32, 223, 223    #拡大
68: #      .dc, w    0
69: #      .dc, l    patt+(HSIZE*4)*(VSIZE/2)+(HSIZE/2)*2
70: #      .dc, w    HSIZE-1, VSIZE-1
71: #      .dc, l    temp
72: #
73:     arg0:                                #IOCS GETGRMの引数

```

```

74:      .dc.w 256-HSIZE, 256-VSIZE
75:      .dc.w 255+HSIZE, 255+VSIZE
76:      .dc.l pat
77:      .dc.l pate
78: #
79: window: .dc.w 80, 80, 175, 175
80: #
81:      .bss
82:      .even
83: #
84: pat:    .ds.w HSIZE*VSIZE*4  #描画パターン
85: pate:   .ds.w 512*4          #grputのワーク
86: temp:   .ds.w 512*4
87: #
88:      .stack
89:      .even
90: #
91:      .ds.l 4096
92: inisp:   .end  ent
93:

```

サブルーチンgrputはいくつかの下請けサブルーチンからなる。以下、順に見ていこう。

サブルーチンrotate (171～246行)

rotateは座標回転の計算をする。172行で引数列から回転角度をd4に取り出し、174～184行でこの角度が $-180^{\circ} \sim +179^{\circ}$ に収まるように360を加減算する。186～190行では、後でcosを求めるためのために $90 - d4$ をd5に計算している。角度の正規化がすんだら192行でsinのテーブルへのポインタをa0に入れ、193～194行でマクロISINを使って $\sin(d4)$, $\cos(d4) = \sin(d5)$ をテーブル参照により得る。

マクロISINは110～129行で定義してある。マクロ中では113～115行で、

$$\sin(-\theta) = -\sin\theta$$

の関係から角度を正に修正し、さらに、117～120行で、

$$\sin(180^{\circ} - \theta) = \sin\theta$$

を利用して $0^{\circ} \sim 90^{\circ}$ に収めている。後は122～123行でテーブルからsinの値を取り出し、125～127行で符号のつじつまをあわせているだけだ。ここでは、角度が正か負か判定し、同時にその結果を覚えておくためにextを利用するという小技に注目してほしい(113行)。

回転角度に応じたsin, cosの値が求まったら、描くパターンの縦横の大きさをd0, d1に取り出す(196行)。縦横比の補正のため横方向の長さをすかさず15倍している(198～202行)。すでに触れたように、この2:3という比率は近似値だから、こだわりたい人は微調整を。

207～211行で回転の中心を求め、213～219行でパターンの左上隅の点、221～223行で右上隅の点、225～228行で左下隅の点を回転する。左下の点については、右上の点の回転結果を使った簡略計算で誤魔化し、残り2点の回転計算は134～166行のマクロROTPで実現している。ROTPは汎用性のまったくないその場逃れのマクロだが、とりあえず、リストの行数を減らすのには貢献してくれた。

ROTP中では、140～143行で、

```

x × cos θ
y × cos θ
x × sin θ
y × sin θ

```

をレジスタに求め、この値を使って145～149行で回転後のx座標、151～155行でy座標を計算する。sin, cosの値はテーブル上では4000_H倍してあるため、得られた座標は4000_Hで割る(14ビット右シフトする)必要があることを思い出そう。リストではこのビットシフトを、2ビットの左シフトとswapで実現した。ここで、座標が負の場合のみシフトに先立って3FFF_Hを足しているのは、小数点以下の切り捨てが0に近づく方向に行われるようにするための処置だ。ROTPの最後、160～164行ではピクセルの縦横比による再補正のためにx座標を2/3にする。四捨五入したいがために、4倍してから3を足して6で割るという、少々冗長な手順を踏んでいる。

237行に達した時点で、パターンの右下を除く3つの角の回転結果が、

```

左上 (d0,d1)
右上 (d2,d3)
左下 (d4,d5)

```

に求まっている。サブルーチンrotateは最後に237～245行で、回転後の左上の点に対応するパターン上の位置をa1に計算し、d0～d5,a1をサブルーチンからの戻り値として返す。

サブルーチンline_comp (252～305行)

line_compは線分 (d0, d1) - (d2, d3) に沿ってグラフィックパターン上の点を拾うときのポイントの相対的な移動量をBresenhamのアルゴリズムによって求め、a5以下のメモリに並べてテーブルを作成する。ついでに、その線分上に位置するピクセル数をd0に返す。描画時にパターン上の点を斜めに拾う、その拾い方を決めるわけだ。

テーブルにはBresenhamのアルゴリズムを実行したときの、ポイントの変化のしかた(たとえば、始点から、右、右、右下、右、右、右下、……)をパターン上のアドレスの差のかたちで記録する。一度このテーブルをつくってしまえば、始点を指すポイントにテーブルの値を順次加えていくだけで、同じ傾き/長さの線分をなぞることができる。

サブルーチンclip (325～424行)

clipはクリッピングを担当する。一度の呼び出してx方向かy方向かのどちらか片方のクリッピングのみを行う。d1, d2で渡されるパターンの左右端のx座標(あるいは上下端のy座標)をクリッピングウィンドウの大きさをクリップし、そのクリッピング量におうじてパターン側も同じ比率で切り捨てる。例によって、このクリッピングには線分のクリッピングアルゴリズム

ムを応用している。

パターン側のクリッピングはけっこう面倒くさい。パターンを斜めに切り捨てなければなら
ないのだ。右端（あるいは下端）でのクリッピング時には何ピクセルはみ出したかがわかれば
十分だが、左端（あるいは上端）でクリッピングする場合は、a1がポイントしているパター
ン上の参照開始位置も再計算する必要がある。ここでは何ピクセル切り捨てるか求めたあと、
line_compが作成したテーブルをそのピクセル数だけ頭から参照して移動量を合計し、クリッ
ピング後の位置までポインタa1を進めることで対処した（370～371行）。

サブルーチンexpand_comp (431～453行)

expand_compはパターンを斜めに拾ってできるピクセル列を描画先領域の大きさにあわせて
拡大・縮小するためのテーブルを作成する。考え方は前章の拡大・縮小ルーチンと同様だ。
やはり、ポインタの相対的な移動量のテーブルをつくる。

すでにline_compがパターンを斜めに拾うときのポインタの移動量をテーブルにしている
ので、expand_compの仕事はそのテーブルを適当に引き伸ばしたり圧縮したりするだけだ。
拡大する場合は、同じピクセルが複数回参照されるようにすればよいわけだから、テーブルの
適当なところに0を入れ引き伸ばす。縮小するときには、ポインタが斜め線上を飛び飛びに移
動するよう、適当に隣りあった要素を足してまとめてテーブルを圧縮する。もちろん、引き伸
ばしたり、圧縮したりするテーブル上の位置はBresenhamのアルゴリズムを使ってバランス
よく決める。

サブルーチンgrput (98～105行)

描画ルーチン本体であるgrputは、サブルーチンrotate、clip、line_comp、expand_comp
を順次呼び出し、最終的にexpand_compが作成したテーブルだけを使って描画を行う。clip、
line_comp、expand_compはx方向とy方向それぞれについて2回呼び出し、テーブルも2本つ
くる。実際に描画を行う90～100行のループでは、y方向のテーブルを使って始点を決め、x
方向のテーブルを使ってそこから斜めに点を拾い描画する。

ところで、y方向のテーブルはただ一度しか使われないから、本来はテーブルをつくらずに
描画と平行してBresenhamのアルゴリズムを適用すればすむ。今回のプログラムではプログ
ラムの簡潔さを優先し、y方向についてもテーブルを作成するかたちにした。

とどめの高速化

最後に、リスト5をもう1段高速化して終わろう。回転角度や拡大・縮小率に応じ、前処理

段階で1ライン分の高速描画ルーチンを作成するよう改良する。考え方は前章で示してある。
リスト7がその1ライン分の描画ルーチンを作成するサブルーチンだ。リスト5の最後に追加し、
さらにリスト5の57行を、

```
bsr    expand_comp2
```

に、90~100行を、

```
yloop: movea.l  a1,a4
        jsr      (a2)
        adda.w   (a3)+,a1
        adda.w   d0,a0
        dbra     d5,yloop
```

に変更して組み込む。

.....
リスト7
GRPUT.S
(高速化版)

```
1: *
2: *      水平1ライン分描画する専用ルーチンを生成する
3: *
4: expand_comp2:
5:     move.w  d1, (a5)+      *クリッピング後の
6:                               * 描画先左上隅x/y座標
7:     sub.w   d1, d2
8:     move.w  d2, (a5)+      *描画幅/高さ
9:
10:    move.w  d2, d1          *Bresenhamのアルゴリズムに
11:    neg.w   d1              * 必要なパラメータの計算
12:    move.w  d2, d3
13:    add.w   d2, d2
14:    add.w   d0, d0
15:
16:    move.l   pset(pc), d5    *d5.l = 1ピクセル書き込んでから
17:                               * 参照位置を移動する命令コード
18:
19:    Explp1: moveq.l #0, d4    *アドレスの相対的な移動量を
20:    add.w    d0, d1          * 求める
21:    ble      Expnxt
22:    Explp2: add.w    (a4)+, d4
23:    sub.w    d2, d1
24:    bgt      Exp1p2
25:    Expnxt: move.l   d5, (a5)+ *move, leaの2命令を書き込む
26:    move.w    d4, (a5)+      *アドレスの変位を書き込む
27:                               * (leaのディスプレースメント)
28:    Expskp: dbra     d3, Explp1
29:    move.w    term(pc), -4(a5) *rts命令を書き込む
30:    term:     rts
31:
32:    pset:     move.w  (a4), (a0)+ *1ピクセル書き込む命令
33:    lea.l     0(a4), a4          *参照位置を移動する命令
34:
35:    .end
```

CHAPTER



自由變形

自由変形



この章では画像の自由変形を取り上げる。ここでいう自由変形とは、矩形の画像を引き伸ばしたり、押し潰したり、歪めたり、回転したり、ねじったり、裏返したりして、任意の四角形にはめ込んで描く操作のことを指す。そのような自由変形機能を備えた、多目的のツールルーチンを作成してみよう。

自由変形のアルゴリズム

図1に自由変形の漠然としたイメージを示す。元画像上の1点P'は、変形後の四角形の向かいあった辺どうしを同じ比で分割して、結んだ交点Pに変換される。もっとも、実際のプログラムでは1点ずつ座標変換するのは効率が悪いので、点単位ではなく線分単位で考える。図2のように描画先を斜めの線分に分割し、水平に1ライン分ずつ切り出した元画像を適当に引き伸ばしたり、縮めたりして、貼りつける感じだ。ここで、隙間なく描画するために、描画先の四角形はAB、DCのどちらか長いほうのピクセル数本の線分に分割する必要がある。副作用で、短いほうの辺近くでは線分どうしが重なり、処理過程で重複して描画されるピクセルが発生するが、本稿では気にしないことにする。

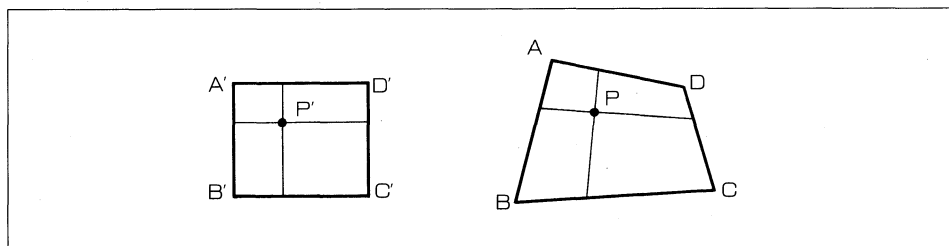


図1
自由変形の考え方

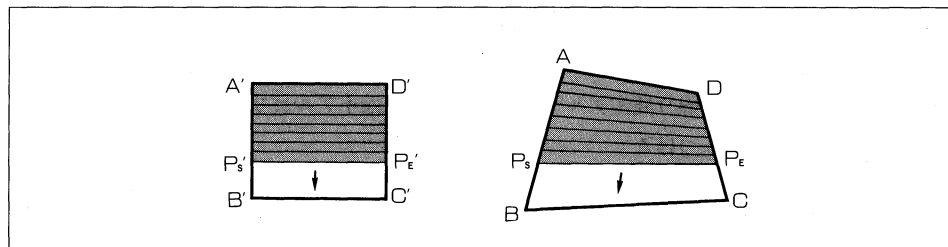


図2
線分単位の自由変形

では、アルゴリズムを次の3ステップに分けて考え、個別に具体化していこう。

- 1) 描画先となる線分 P_S-P_E を決める
- 2) P_S-P_E に対応する元画像上の水平線分 $P'_S-P'_E$ を求める
- 3) $P'_S-P'_E$ を P_S-P_E に貼りつける

1) 描画先線分の決定

描画先 P_S-P_E はADからBCへ傾きを変えながら移動していく。両端点 P_S と P_E に注目するなら、 P_S はAB上を、 P_E はDC上を移動していくことになり、その移動経路はBresenhamの線分発生アルゴリズムで順次求まる。当然、両端点が決まれば、それらを結ぶ線分もまた決定する。

ここで、ABとDCの長さはかならずしも等しいわけではないから、AB、DC上の各点を1対1に結んだのでは長いほうが余る。正しくは、 P_S と P_E が同時にB、Cに達するよう、短いほうの線分をなぞる速度を適当に遅らせなければならない。この速度調節には、拡大・縮小アルゴリズム（その実体は線分発生アルゴリズムと同じもの）が利用できる。

2) 描画先線分に対応する元画像上の水平線分の決定

$P'_S-P'_E$ は、 P_S-P_E がADからBCに移動するのと連動してA'D'からB'C'へと動いていく。したがって、適当な移動速度で $P'_S-P'_E$ を動かしてやれば、 P_S-P_E からわざわざ逆算しなくても、対応する $P'_S-P'_E$ が順次求まる。移動速度の調節には、上と同様、拡大・縮小アルゴリズムが応用できる。

3) 元画像の水平線分を傾けて貼りつける

$P'_S-P'_E$ を P_S-P_E に貼りつける処理は、線分発生アルゴリズムと拡大・縮小アルゴリズムの組みあわせで実現できる。 P_S-P_E 上の点を順次求めつつ、拡大・縮小の手順で対応する $P'_S-P'_E$ 上の位置を求め、そこから色を拾って点を打っていけばよい。

*

以上をまとめると、自由変形のアルゴリズムは次のようになる。

- 1) ABとCDの長いほうを選び出す。そのピクセル数がループ回数となる。以下、かり

にABのほうが長いものとする

- 2) $P_S - P_E$ をAD, $P_S' - P_E'$ をA'D'に初期化する
- 3) 線分発生アルゴリズムと拡大・縮小アルゴリズムを利用して, $P_S' - P_E'$ を $P_S - P_E$ に貼りつける
- 4) 線分発生アルゴリズムを使って P_S をABに沿って移動する
- 5) 拡大・縮小アルゴリズムを使って P_E を移動すべきかどうか調べる。移動すべきであれば, 線分発生アルゴリズムを使って P_E をDCに沿って移動する
- 6) 拡大・縮小アルゴリズムを使って $P_S' - P_E'$ を移動すべきかどうか調べる。移動すべきであれば, $P_S' - P_E'$ を1ライン下に移動する
- 7) 2) ~6) を1) で求めた回数だけ繰り返す

さて, 上で導いたアルゴリズムを実際にプログラムにするうえでは, 若干の小細工が必要だ。グラフィック画面上に斜めの線分を並べて面にしようとする, 階段状になった線分の段の部分に隙間を生じる場合がある (図3)。この問題は P_S , P_E の移動経路に冗長性を持たせ, 線分の数を増やすことで解決する。具体的には, P_S , P_E の移動経路を求めるのに, 通常使われる8連結 (ピクセルが斜めに並ぶことを許す) の線分発生アルゴリズムではなく4連結 (ピクセル

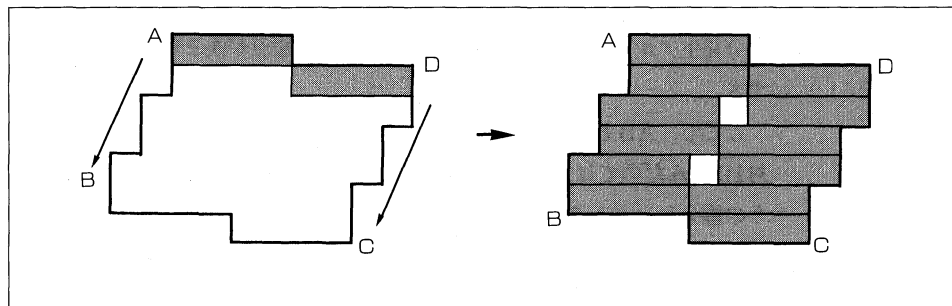


図3
線分間の隙間

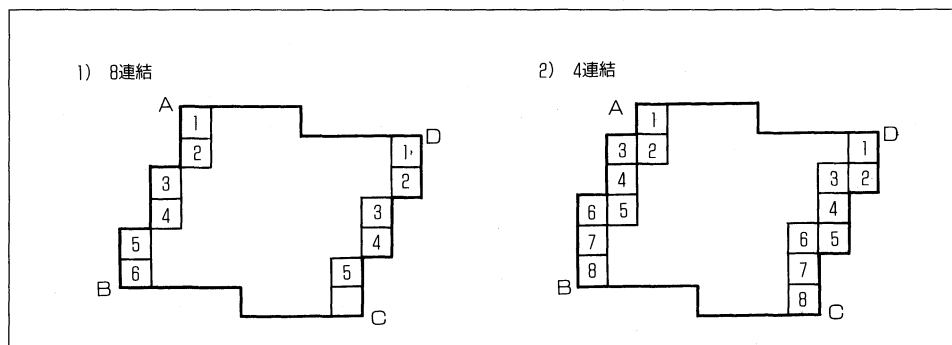


図4
8連結と4連結の移動経路の違い

がかならず縦か横に並ぶ) のアルゴリズムを利用する (図4)。こうすると、ちょうど隙間を埋める (隙間の上を通る) 新たな線分が生まれることを確認しておいてほしい。副作用で、同一ピクセルに重複して描画する場面が増えることにはなるが、背に腹は代えられない。

プログラムの作成

リスト1に自由変形機能付きのグラフィックパターンプットルーチン、あわせてリスト2にその動作試験用のプログラムを示す。リスト2は画像を画面に読み込んでおいてから起動すると、その中央128ピクセル四方を切り出し、何かキーが押されるまで、ランダムな四角形に変形して描画し続ける。

サブルーチンgtputには28~38行のような構造で引数を渡す。描画先は4頂点を反時計回りに並べることになっている。並べる順序にも意味があり、頂点を逆順に並べれば、画像を裏返して描くことができる。また、これまでとくに触れなかったが、描画先の四角形はへこんでいたり、ねじれていたりしてもかまわない。ただ、アルゴリズムの性格上、へこんだ角やねじれた部分は丸みを帯びる。その様子は、実際に自分の目で確かめてもらいたい。

リスト1
GTPUT.S
(第1版)

```

1: #      パターンを任意四角形にはめ込んでプットする (第1版)
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: #
6:      .xdef      gtput
7:      .xref      gramadr
8:      .xref      cliprect
9: #
10: PSET    macro    X,Y,COL,ADR      *クリッピングしつつ
11:         local    skip              * 点を打つマクロ
12:         lea.l     cliprect,a5      *
13:         cmp.w     (a5)+,X           *
14:         blt       skip              *
15:         cmp.w     (a5)+,Y           *
16:         blt       skip              *
17:         cmp.w     (a5)+,X           *
18:         bgt       skip              *
19:         cmp.w     (a5)+,Y           *
20:         bgt       skip              *
21:         move.w    COL,ADR           *
22:     skip:
23:         .endm                      *
24:
25: #
26:      .offset 0      *gtputの引数構造
27: #
28: X0:      .ds.w      1      *描画先座標      A
29: Y0:      .ds.w      1      *
30: X1:      .ds.w      1      *
31: Y1:      .ds.w      1      *
32: X2:      .ds.w      1      *
33: Y2:      .ds.w      1      *
34: X3:      .ds.w      1      *
35: Y3:      .ds.w      1      *
36: PAT:     .ds.l      1      *パターンアドレス
37: XL:      .ds.w      1      *パターンの横の長さ-1
38: YL:      .ds.w      1      *パターンの縦の長さ-1

```

```

39: #
40: .offset 0      #線分発生用ワーク
41: #
42: X: .ds.w 1      #x座標
43: Y: .ds.w 1      #y座標
44: DX: .ds.w 1     #y更新時にeから引く定数
45: DY: .ds.w 1     #x更新時にeに加える定数
46: SX: .ds.w 1     #x増加方向
47: SY: .ds.w 1     #y増加方向
48: E: .ds.w 1      #誤差項e
49: LEN: .ds.w 1     #線分のピクセル数-1
50: SDX: .ds.w 1     #速度調節用e補正值
51: SDY: .ds.w 1     #速度調節用e増分
52: EDG:
53: #
54: .offset -EDG*2-2*4      #スタックフレーム
55: #
56: WORKTOP:
57: ST: .ds.b EDG      #始点関連ワーク
58: ED: .ds.b EDG      #終点関連ワーク
59: PHE: .ds.w 1       #水平スケーリング用e初期値
60: PHDY: .ds.w 1      #水平スケーリング用e増分
61: PVDX: .ds.w 1      #垂直スケーリング用e補正值
62: PVDY: .ds.w 1      #垂直スケーリング用e増分
63: _A6: .ds.l 1       #0
64: _SP: .ds.l 1       #4
65: ARGPTR: .ds.l 1    #8
66: #
67: .text
68: .even
69: #
70: gtput:
71: link a6, #WORKTOP
72: movem.l d0-d7/a0-a5, -(sp)
73:
74: movea.l ARGPTR(a6), a5      #a5=引数列
75:
76: movem.w X0(a5), d0-d3      #ABを発生するための
77: lea.l ST(a6), a0           # (=始点の経路を決めるための)
78: bsr init                   # 誤差項周りの初期化
79: movea.l a1, a3              #a3=始点を移動するサブルーチン
80:
81: movem.w X3(a5), d0-d1      #DCを発生するための
82: movem.w X2(a5), d2-d3      # (=終点の経路を決めるための)
83: lea.l ED(a6), a0           # 誤差項周りの初期化
84: bsr init                   #
85: movea.l a1, a4              #a4=終点を移動するサブルーチン
86:
87: move.w ST+LEN(a6), d7      #d7=ABの長さ-1
88: move.w ED+LEN(a6), d0      #d0=DCの長さ-1
89: lea.l xline-xlinex(a3), a3
90: #ABの方が長いと仮定し
91: # 仮に始点の速度調節を殺す
92:
93: cmp.w d0, d7               #ABとDCの長さが等しければ
94: beq equskp                 # 始点、終点ともに速度調節不要
95: bcc maxskp                 #ABの方が長ければ
96: # 始点については速度調節不要
97: #DCの方が長ければ
98: # 終点については速度調節不要
99: move.w d0, d7              #d7=DCの長さ-1
100: lea.l xlinex-xline(a3), a3      #始点の速度調節を復活する
101:
102: equskp: lea.l xline-xlinex(a4), a4      #終点の速度調節を殺す
103:
104: maxskp: move.w d7, d5          #d7=d5=ABとDCの長さの長い方
105: neg.w d5                      #d5=始点 (または終点) の
106: # 移動速度調節用e
107:
108: move.w d7, d0                #
109: add.w d0, d0                  #
110: move.w d0, ST+SDX(a6)         #始点移動速度調節用e補正值
111: move.w d0, ED+SDX(a6)         #終点移動速度調節用e補正值
112: move.w d0, PVDX(a6)           #パターンを垂直になぞる速度の

```

```

113:                                     * 調節用e補正值
114:
115: move.w YL(a5), d0                *d0=パターン高-1
116: move.w d0, d6                    *d6=パターンを垂直になぞる速度の
117: neg.w d6                         * 調節用e
118: tst.w d7                         *
119: bne fix0                          *描画先高が1ピクセルしかない場合の
120: move.w d7, d0                    * つじつま合わせ
121: fix0: add.w d0, d0                *パターンを垂直になぞる速度の
122:                                     * 調節用e増分
123: move.w d0, PVDY(a6)              *
124:
125: movea.l PAT(a5), a1              *a1=パターン先頭アドレス
126:
127: move.w XL(a5), d0                *d0=パターン幅-1
128: movea.w d0, a5                   *a5=パターン1ライン分バイト数
129: addq.l #1, a5                     *
130: adda.l a5, a5                     *
131:
132: move.w d0, d1                    *パターンを水平になぞる速度の
133: add.w d1, d1                     * 調節用e増分
134: move.w d1, PHDY(a6)              *
135:
136: neg.w d0                          *パターンを水平になぞる速度の
137: move.w d0, PHE(a6)               * 調節用e
138:
139: movem.w ED+X(a6), d2-d3          * (d2.w, d3.w)=終点座標
140: move.w ED+E(a6), d4              *d4.w=終点移動経路決定用e
141: swap.w d2                        *それぞれレジスタの
142: swap.w d3                        * 上位ワードに保存
143: swap.w d4                        *
144: move.w ST+X(a6), d2              * (d2.w, d3.w)=始点座標
145: move.w ST+Y(a6), d3              *
146: move.w ST+E(a6), d4              *d4.w=始点移動経路決定用e
147:
148: loop:
149: * d2.l 始点.終点のx座標
150: * d3.l 始点.終点のy座標
151: * d4.l 始点.終点移動経路決定用e
152: * (上位ワード:終点/下位ワード:始点)
153: * d5.w 始点(または終点)移動速度調節用e
154: * d6.w パターンを垂直になぞる速度の調節用e
155: * d7.w ループカウンタ
156: * a1 参照中のパターン上水平線分左端アドレス
157: * a3 始点を移動するサブルーチン
158: * a4 終点を移動するサブルーチン
159: * a5 パターン1ラインバイト数
160:
161: move.w d2, d0                    *d0=始点のx座標
162: move.w d3, d1                    *d1=始点のy座標
163:
164: swap.w d2                        *d2.w=終点のx座標
165: swap.w d3                        *d3.w=終点のy座標
166: swap.w d4                        *
167:
168: bsr putline                      *1ライン分描画する
169:
170: lea.l ED(a6), a0                 *終点を更新する
171: jsr (a4)                         *
172:
173: swap.w d2                        *
174: swap.w d3                        *
175: swap.w d4                        *
176:
177: lea.l ST(a6), a0                 *始点を更新する
178: jsr (a3)                         *
179:
180: add.w PVDY(a6), d6               *移動速度を考慮して
181: ble nextln                       * パターンの垂直方向参照位置を
182:                                     * 移動する
183: skppat: adda.l a5, a1             *
184: sub.w PVDX(a6), d6               *
185: bgt skppat                       *
186:

```

```

187: nextln: dbra    d7, loop      *必要なだけ繰り返す
188:
189:         movem.l  (sp)+, d0-d7/a0-a5
190:         unlk     a6
191:         rts
192:
193: *
194: *       1ライン分描画する
195: *
196: putline:
197:         movem.l  d2-d7/a1/a3-a5, -(sp)
198:
199:         jsr      gramadr        *a0=始点のG-RAMアドレス
200:
201:         sub.w    d0, d2        *始点と終点を結ぶ線分を
202:         move.w   d2, d4        * 発生するための
203:         ABS      d2            * パラメータの初期化
204:         move.w   d2, a2        *
205:         add.w    a2, a2        *
206:         SGN      d4            *
207:         *
208:         sub.w    d1, d3        *
209:         move.w   d3, d5        *
210:         ABS      d3            *
211:         move.w   d3, a3        *
212:         add.w    a3, a3        *
213:         SGN      d5            *
214:
215:         move.w   d5, d6        *d6=y座標を更新したときの
216:         moveq.l  #GSFCTR, d7   * G-RAMアドレスの変化量
217:         asl.w    d7, d6        *
218:
219:         move.w   PHDY(a6), a4   *パターンを水平になぞる速度の
220:         * 調節用e増分
221:
222:         cmp.w    d3, d2        *線分の傾きに応じて
223:         bcs      yput          * 処理を振り分ける
224:
225: xput:    *なだらかな傾きの線分へ張り付ける場合
226:         move.w   d2, d7        *d7=ループカウンタ
227:         neg.w    d2            *d2=誤差項e
228:         bne      fix1          *描画先幅が1ピクセルしかない場合の
229:         move.w   d2, a4        * つじつま合わせ
230:         fix1:    move.w   PHE(a6), d3 *パターンを水平になぞる速度の
231:         * 調節用e
232:
233: xloop:   PSET     d0, d1, (a1), (a0) *1ピクセル描画
234:
235:         add.w    a4, d3        *移動速度を考慮して
236:         ble      xput1        * パターンの水平方向参照位置を
237:         * 更新する
238: xput0:    addq.l  #2, a1        *
239:         sub.w    a2, d3        *
240:         bgt      xput0        *
241:
242: xput1:    add.w   d4, d0        *x+=sx
243:         adda.w   d4, a0        *
244:         adda.w   d4, a0        *
245:
246:         add.w    a3, d2        *e+=dy
247:         bmi      xnext        *
248:
249:         add.w    d5, d1        *y+=sy
250:         adda.w   d6, a0        *
251:
252:         sub.w    a2, d2        *e-=dx
253:
254: xnext:    dbra    d7, xloop
255:
256:         movem.l  (sp)+, d2-d7/a1/a3-a5
257:         rts
258:
259: yput:    *急な傾きの線分へ張り付ける場合
260:         move.w   d3, d2

```

```

261:      move.w  d2, d7
262:      neg.w   d2
263:      bne     fix2
264:      move.w  d2, a4
265: fix2:  move.w  PHE(a6), d3
266:
267: yloop: PSET    d0, d1, (a1), (a0)
268:
269:      add.w   a4, d3
270:      ble     yput1
271:
272: yput0:  sub.w   a3, d3
273:      addq.l  #2, a1
274:      bgt     yput0
275:
276: yput1:  add.w   d5, d1
277:      adda.w  d6, a0
278:
279:      add.w   a2, d2
280:      bmi     ynext
281:
282:      add.w   d4, d0
283:      adda.w  d4, a0
284:      adda.w  d4, a0
285:
286:      sub.w   a3, d2
287:
288: ynext:  dbra    d7, yloop
289:
290:      movem.l (sp)+, d2-d7/a1/a3-a5
291:      rts
292:
293: *
294: *      AB, DC発生のための初期化
295: *
296: init:
297:      sub.w   d0, d2      *d2=d4=x1-x0
298:      move.w  d2, d4      *
299:      ABS     d2          *d2=dx=abs(x1-x0)
300:      SGN     d4          *d4=sx=sgn(x1-x0)
301:
302:      sub.w   d1, d3      *d3=d5=y1-y0
303:      move.w  d3, d5      *
304:      ABS     d3          *d3=dy=abs(y1-y0)
305:      SGN     d5          *d5=sy=sgn(y1-y0)
306:
307:      cmp.w   d3, d2      *傾きに応じて
308:      bcs     yinit       * 処理を振り分ける
309:
310:      *dx ≥ dyの場合
311: xinit:  move.w  d2, d6      *d6=dx
312:      neg.w   d6          *d6=e=-dx
313:      add.w   d3, d2      *d2=dx+dy
314:      move.w  d2, d7      *d7=dx+dy
315:      *      =線分のピクセル数-1
316:      add.w   d2, d2      *d2=2dx+2dy
317:      *      =y座標更新時のe補正值
318:      add.w   d3, d3      *d3=2dy
319:      *      =x座標更新時のe増分
320:      movem.w d0-d7, X(a0) *ワークに格納する
321:
322:      move.w  d2, SDY(a0)  *速度調節用e増分
323:      lea.l   xlinex(pc), a1 *a1=傾きがなだらかな線分を
324:      *      = 1ピクセルずつ発生する
325:      *      サブルーチン
326:      rts
327:
328:      *dx < dyの場合
329: yinit:  move.w  d3, d6      *d6=dy
330:      neg.w   d6          *d6=e=-dy
331:      add.w   d2, d3      *d3=dx+dy
332:      move.w  d3, d7      *d7=dx+dy
333:      *      =線分のピクセル数-1
334:      add.w   d2, d2      *d2=2dx

```

```

335:                                     # =x座標更新時のe増分
336:      add.w    d3,d3                 #d3=2dx+2dy
337:                                     # =y座標更新時のe補正值
338:      movem.w  d0-d7,X(a0)          #ワークに格納する
339:
340:      move.w   d3,SDY(a0)           #速度調節用e増分
341:      lea.l    ylinex(pc),a1        #a1=傾きが急な線分を
342:                                     # 1ピクセルずつ発生する
343:                                     # サブルーチン
344:      rts
345:
346: #
347: #      移動速度を考慮して始点,終点を移動する
348: #      (なだらかな傾きの線分用)
349: #
350: xlinex: add.w   SDY(a0),d5          #速度の調節
351:      bmi.s    xline1              #
352:      #
353:      sub.w    SDX(a0),d5          #
354:
355: xline:  add.w   DY(a0),d4           #e+=2dy
356:      bmi      xline0
357:
358:      add.w    SY(a0),d3            #y+=sy
359:      sub.w    DX(a0),d4            #e-=2(dx+dy)
360:      rts
361:
362: xline0: add.w   SX(a0),d2           #x+=sx
363: xline1: rts
364: #
365: #
366: #      移動速度を考慮して始点,終点を移動する
367: #      (急な傾きの線分用)
368: #
369: ylinex: add.w   SDY(a0),d5          #速度の調節
370:      bmi.s    yline1              #
371:      #
372:      sub.w    SDX(a0),d5          #
373:
374: yline:  add.w   DX(a0),d4           #e+=2dx
375:      bmi      yline0
376:
377:      add.w    SX(a0),d2            #x+=sx
378:      sub.w    DY(a0),d4            #e-=2(dx+dy)
379:      rts
380:
381: yline0: add.w   SY(a0),d3            #y+=sy
382: yline1: rts
383:
384:      .end

```

.....
リスト2
GTPUTTEST.S

```

1: #      gtputの動作試験用プログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: #
6:      .xref         gtput
7: #      .xref         setcliprect
8: #
9: FPACK macro      callno
10:      .dc.w         callno
11:      .endm
12: #
13: __RAND equ        $fe0e
14: #
15:      .text
16:      .even
17: #
18: ent:
19:      lea.l         inisp,sp
20:

```



```

21:      clr.l    -(sp)          *スーパーバイザモードへ
22:      DOS     _SUPER         *
23:
24:      lea.l    arg0, a1       *画面からパターンを取り込む
25:      lOCS     _GETGRM       *
26:
27:                                     *512x512, 65536
28:      move.l   #$0010_0005, -(sp)
29:      DOS     _CONCTRL        *画面初期化
30:
31:      *      pea    window      *クリッピングウィンドウの設定
32:      *      jsr    setcliprect *
33:      *      addq.l  #4, sp      *
34:
35:      lea.l    arg, a1         *a1 = 引数受け渡し領域
36: loop:   pea.l  (a1)           *描画
37:      bsr     gtput            *
38:      addq.l   #4, sp          *
39:
40:      movea.l  a1, a0          *引数をセット
41:      moveq.l  #8-1, d1        *4点分の座標
42: loop2:  FPACK    _RAND        *
43:      lsr.w    #6, d0          *0 ≤ d0 ≤ 511
44:      *      lsr.w    #5, d0      *0 ≤ d0 ≤ 1023
45:      *      subi.w   #256, d0    *-256 ≤ d0 ≤ 767
46:      move.w   d0, (a0)+       *
47:      dbra     d1, loop2       *
48:
49:      DOS     _KEYSNS         *キーが押されるまで
50:      tst.l    d0             * 繰り返す
51:      beq      loop           *
52:
53:      DOS     _INKEY          *スペースが押されたら
54:      cmpi.b   #$20, d0       * 一時停止する
55:      bne      done           *
56: pause:  DOS     _INKEY          *
57:      cmpi.b   #$20, d0       *
58:      beq      loop           *
59:
60: done:   move.w  #-1, -(sp)     *キーを読み捨てる
61:      DOS     _KFLUSH         *
62:
63:      move.l   #$0010_0000, -(sp) *width 96
64:      DOS     _CONCTRL        *画面再初期化
65:      DOS     _EXIT
66: *
67: .data
68: .even
69: *
70: HSIZE equ 128
71: VSIZE equ 128
72: *
73: arg:                                     *gtputの引数
74:      .dc.w    50, 0, 0, 480, 511, 350, 400, 50
75:      .dc.l    pat
76:      .dc.w    HSIZE-1, VSIZE-1
77: *
78: arg0:                                     *IOCS GETGRMの引数
79:      .dc.w    256-HSIZE/2, 256-VSIZE/2
80:      .dc.w    255+HSIZE/2, 255+VSIZE/2
81:      .dc.l    pat
82:      .dc.l    pate
83: *
84: window: .dc.w  64, 64, 511-64, 511-64
85: *
86:      .bss
87:      .even
88: *
89: pat:     .ds.w  HSIZE*VSIZE      *描画パターン
90: pate:
91: *
92:      .stack
93:      .even
94: *

```

```

95:      .ds.l   4096
96: inisp:
97:
98:      .end    ent

```

リスト1は比較的長いプログラムだが、ただひたすらに、Bresenhamの線分発生アルゴリズムを組みあわせているだけだ。あまり見るべき点はないが、ざっと全体の処理の流れを追いつながら、ポイントになる部分を拾っていくとしよう。

初期化 (74～146行, 296～344行)

初期化部分では、

P_s , P_E の移動経路決定

P_s , P_E どちらかの移動速度調節

$P_s' - P_E'$ の移動速度調節

のそれぞれに対して、Bresenhamの線分発生アルゴリズムを適用するための誤差項関連の初期化をしている。

76～85行が P_s , P_E の移動経路関係の初期化だ。実際の誤差項周りの計算は296～344行のサブルーチンinitにまとめてある。 P_s , P_E の移動経路は4連結の線分発生アルゴリズムで求めることに注意してほしい。さすがに扱う変数の数が多いので、求めたパラメータは40～52行のような構造のワークにしまっておくようになっている。

ここで、線分発生時には、線分の傾きにおうじて処理を振り分けなければならないことを思い出そう。一気に線分全体を発生する場合は、ループに入る直前に1回処理を振り分けるだけで済むが、いまの場合、ループの中で2本の線分を平行して発生しなければならない。常識的なやり方では、線分の傾きが 45° より急かなだらかかを覚えておき、それをスイッチにしてループ中で毎回処理を振り分けることになる。が、リスト1では350～363行、369～382行に線分を1ピクセルずつ発生するサブルーチンを傾き別に2つ用意しておき、初期化時点でその先頭アドレスをアドレスレジスタに入れておくことにより、ループ内部での条件分岐を省略できるようにしている。

87～102行ではAB, DCのどちらが長いかを調べ、長いほうのピクセル数をループカウンタとして使うためにd7に格納している。ここで、 P_s , P_E の移動速度は、移動経路の長いほうを基準にして、短いほうを適当に遅らせることで調節するのだから、ABのほうが高い場合は P_s , DCのほうが高いときは P_E 、等しいときは両方の速度調節が不要になる。87～102行ではついでにそのあたりのつじつまあわせを行っている。350～363行、369～382行の線分を1ピクセルずつ発生するサブルーチンの先頭についている速度調節処理の部分が実行されないよう、アドレスレジスタに入れたサブルーチンのアドレスを必要だけ進めているのだ。

初期化部分の残りはあまりおもしろくない。104～123行は、 P_s , P_E (のどちらか)、 $P_s' -$

P_E' の移動速度調節関係の初期化で、見てのとおり。127～130行もパターンの横幅からパターン1ライン分のバイト数を計算しているだけだ。132～137行ではパターンを画面に貼りつけるときの拡大・縮小に使う値のうち、定数となるものを先に計算している。最後に139～146行で P_S 、 P_E の座標と、移動経路決定用の誤差項をレジスタに取り出して初期化は終わりだ。

メインループ (148～187行)

プログラムの実行に必要なパラメータはほとんど初期化時点で計算してあるし、パターン1ライン分を貼りつける処理はサブルーチンにして抜き出してあるので、メインループはかたがたかんたんになっている。161～168行で1ライン分描画し、170～171行で P_E 、173～178行で P_S 、180～185行で $P_S'-P_E'$ を速度を考慮して移動するという、アルゴリズムどおりのつくりだ。

1ライン分の描画 (186～291行)

サブルーチンputtlineでは、線分 P_S-P_E を発生する線分発生アルゴリズムと、 $P_S'-P_E'$ を P_S-P_E の長さに揃える拡大・縮小アルゴリズムを組みあわせて、1ライン分のパターンを画面に貼りつける。

このサブルーチンではとくにクリッピング部分に触れておきたい。クリッピングは、10～23行で定義した1点分描画するマクロPSETの中で1点ごとに行っている。線分のクリッピングはかんたんに実現できるのに、あえて1点ずつクリッピングしているのは、誤差を避けるためだ。線分の端点のクリッピングは整数化して計算する以上、どんなに正確を期しても小数点以下の誤差が生じる。斜めの線分を並べて面にするうでは、そのわずかな誤差が隙間の原因となりうる。しかたなく、1点ごとにクリッピングウィンドウに収まっているかどうか調べるという原始的な方法をとっているのだ。

しかし、この方法にもまた問題がある。クリッピングされていない（-32768～+32767の座標値をとる）線分に対してBresenhamの線分発生アルゴリズムを適用すると、誤差項の計算が16ビットでは収まらない。リスト1はこの点には目をつむって16ビット計算で誤魔化しているため、ある程度以上に大きく拡大するとオーバーフローを起こす。気になるようなら、誤差項の計算を32ビットで行うように修正してみてほしい。

禁断の自己書き換え

前述のように、リスト1では精度を保つ目的でクリッピングを点単位で行っているため、実行速度はあまり速くない。そこで、クリッピングしなくてもすむ場合を特別扱いし、平均的な性能を向上させることを考えてみよう。1ライン分描画する時点で、描画先の線分が完全にク

リッピングウィンドウに収まっているなら、座標比較を省略した専用ループで処理するという発想だ。リスト1からの変更点のかたちで実例をリスト3に示す。

リスト3では、実際にはループを2つ用意せずに、俗に自己書き換えと呼ばれる方法を使い、1つのループで間にあわせている。自己書き換えとは、言葉通りプログラムによってプログラム自身を書き換えてしまうという技だ。マシン語プログラムはメモリ上ではただのビット列であり、データとの区別はないから、プログラム自身を書き換えることもできるというわけだ。自己書き換えは、うまく使えばプログラムサイズを小さくするのに役立つ。もっとも、多用するとあつという間にわけのわからない下品なプログラムになる。そのため、本書では自己書き換えを使うのを避けてきたのだが、理屈だけは知っておいてもらおうと、終わり間際のこの章であえてやってみせた次第だ。

.....
リスト3
GTPUT.S
(高速化版:リスト1
に対する追加・変更部)

```

1: #          パターンを任意四角形にはめ込んでプットする (高速化版)
193: #
194: #          1ライン分描画する
195: #
196: putline:
197:     movem.l d2-d7/a1/a3-a5, -(sp)
198:
199:     move.w #xloop-xnext-2, d4
200:
201:     lea.l cliprect, a5
202:     cmp.w (a5), d0
203:     blt skip
204:     cmp.w (a5)+, d2
205:     blt skip
206:     cmp.w (a5), d1
207:     blt skip
208:     cmp.w (a5)+, d3
209:     blt skip
210:     cmp.w (a5), d0
211:     bgt skip
212:     cmp.w (a5)+, d2
213:     bgt skip
214:     cmp.w (a5), d1
215:     bgt skip
216:     cmp.w (a5)+, d3
217:     bgt skip
218:
219:     addi.w #22, d4
220:
221: skip:     move.w d4, xnext+2
222:           move.w d4, ynext+2
223:
224:     jsr gramadr          #a0=始点のG-RAMアドレス
225:
(以下リスト1の201行以降と同様)

```

新版のgtputでは、リスト1の233～254行や267～288行のループを構成するdbraのディスプレイスメント部分に自己書き換えを適用している。マクロPSETの中に埋もれているのでわかりにくいかもしれないが、このループのちょうど先頭に点のクリッピング処理がある。したがって、クリッピングをしたくない場合は、ループ末尾のdbraの分岐先がクリッピング部の直後を指すよう書き換えればよい。この分岐距離の計算をしているのがリスト3の199～219行だ。199行でクリッピングするときの分岐距離をd4に求めておき、201～217行で線分の両端

点がウィンドウ内に収まっているかどうか調べ、収まっているようなら219行でクリッピング部分を飛ばすように、分岐距離を調節している。199行の“-2”は、68000のPC相対アドレッシングにおけるディスプレイメントは命令の置かれたアドレス+2を基点に表されることを反映している。求めたディスプレイメントは221~222行で実際に書き込む。dbraは2ワード命令で、第2ワードにディスプレイメントが埋め込まれているから、書き込むアドレスはdbraが置かれたアドレス+2となる。

自由変形ルーチンの応用

自由変形ルーチンはけっこう応用範囲が広い。拡大・縮小ルーチンとしても、回転ルーチンとしても使えるし（当然、頂点の回転計算はべつにすることになるが）、もっと単純に画像の上下左右反転にも利用できる。欠点は実行速度で、さすがに専用ルーチンにはおよばない。ただ、回転ルーチンの代用品としては積極的に使ってみてもよいだろう。前章でつくった回転ルーチンgrputは元画像を斜めに切り出して水平に貼りつけるという方法を採用したために、描画範囲はあくまで画面に平行な矩形だった。しかし、いま作成した自由変形ルーチンgtputを使えば、描画範囲自体を回転した結果を得ることができる。また、拡大をとまなう回転の場合、gtputのほうがより自然な回転結果を得られる。grputでは回転してから拡大するかたちになるので、描画先画面の水平・垂直方向にモザイク模様ができるが、gtputではモザイクごと回転する格好になる。

参考までに、リスト4にgtputを応用した画像回転ルーチンgrput2を示しておこう。描画先矩形の回転計算をしてgtputを呼び出すだけのプログラムだ。仕様は微妙にgrputとは異なっているので、呼び出し手順についてはリスト5のテストプログラムも参考にしてほしい。

リスト4
GRPUT2.S

```

1: *      パターンを回転/拡大/縮小してプットする（自由変形利用版）
2:
3:      .include      gconst.h
4:      .include      gmacro.h
5: *
6:      .xdef      grput2
7:      .xref      gtput
8:      .xref      sintable
9: *
10: *      sin(DEG)をテーブルから引いてくるマクロ
11: *
12: !SIN      macro      TABLE, DEG
13:      local      skip1, skip2, skip3
14:
15:      ext.l      DEG          *sin(-θ) = -sin(θ)
16:      bpl      skip1          *
17:      neg.w      DEG          *
18:
19: skip1:      subi.w      #90, DEG          *sin(180°-θ) = sin(θ)
20:      bcs      skip2          *
21:      neg.w      DEG          *
22: skip2:      addi.w      #90, DEG          *0 ≤ deg ≤ 90°
23:
24:      add.w      DEG, DEG

```

```

25:      move.w 0(TABLE, DEG), DEG
26:
27:      tst.l   DEG          #sin(-θ) = -sin(θ)
28:      bpl     skip3        #
29:      neg.w   DEG          #
30: skip3:
31:      .endm
32:
33: #
34: #      点(X, Y)を回転するマクロ
35: #
36: ROTP macro X, Y
37:      local skip1, skip2
38:
39:      move.w X, d6
40:      move.w Y, d7
41:
42:      muls.w d5, X          #x*cos(θ)
43:      muls.w d5, Y          #y*cos(θ)
44:      muls.w d4, d6         #x*sin(θ)
45:      muls.w d4, d7         #y*sin(θ)
46:
47:      sub.l   d7, X          #x*cos(θ)-y*sin(θ)
48:      bpl     skip1
49:      addi.l  #$3fff, X
50: skip1:  asl.l  #2, X
51:      swap.w X
52:
53:      add.l   d6, Y          #x*sin(θ)+y*cos(θ)
54:      bpl     skip2
55:      addi.l  #$3fff, Y
56: skip2:  asl.l  #2, Y
57:      swap.w Y
58:
59:      add.w   a2, X
60:      add.w   a3, Y
61:
62:      add.w   X, X           #x座標を2/3にする
63:      add.w   X, X           #
64:      addq.w  #3, X          #
65:      ext.l   X              #
66:      divs.w  #6, X          #
67:
68:      .endm
69: #
70:      .offset 0              #grput2の引数構造
71: #
72: X0:      .ds.w 1             #描画先座標
73: Y0:      .ds.w 1             #
74: X1:      .ds.w 1             #
75: Y1:      .ds.w 1             #
76: DEG:      .ds.w 1            #回転角度(°)
77: PAT:      .ds.l 1            #パターンアドレス
78: XL:      .ds.w 1             #パターンの横の長さ-1
79: YL:      .ds.w 1             #パターンの縦の長さ-1
80: #
81:      .offset -48            #スタックフレーム
82: #
83: WORKTOP:
84: ARGBUF:  .ds.b 48           #gtputへの引数
85: _A6:      .ds.l 1            #0
86: _SP:      .ds.l 1            #4
87: ARGPTR:   .ds.l 1            #8
88: #
89:      .text
90:      .even
91: #
92: grput2:
93:      link    a6, #WORKTOP
94:      movem.l d0-d7/a0-a3, -(sp)
95:
96:      movea.l ARGPTR(a6), a1    #a1 = 引数列
97:      move.w  DEG(a1), d4       #d4 = 角度
98:

```

```

99:      move.w #360, d6      *角度を
100:     move.w #180, d7      * -180° ~ 179° に
101:     add.w d7, d4          * 正規化する
102:     bpl norm2             *
103: norm1: add.w d6, d4        *
104:     bmi norm1             *
105: norm2: cmp.w d6, d4        *
106:     bcs norm3             *
107:     sub.w d6, d4           *
108:     bra norm2              *
109: norm3: sub.w d7, d4        *
110:
111:     moveq.l #90, d5        *cos θ = sin(90° - θ)
112:     sub.w d4, d5           *
113:     cmp.w d7, d5           *
114:     ble rot                *
115:     sub.w d6, d5           *
116:
117: rot:   lea.l sintable, a0
118:     lsin a0, d4            *d4 = sin(θ)
119:     lsin a0, d5            *d5 = cos(θ)
120:
121:     movem.w X0(a1), d0-d3  *d0~d3に座標を取り出す
122:
123:     lea.l ARGBUF(a6), a0   *a0 = gtputへ渡す引数列
124:
125:     move.w d0, d6           *x0を1.5倍する
126:     add.w d0, d0            *
127:     add.w d6, d0            *
128:     asr.w #1, d0            *
129:
130:     move.w d2, d6           *x1を1.5倍する
131:     add.w d2, d2            *
132:     add.w d6, d2            *
133:     asr.w #1, d2            *
134:
135:     move.w d2, d6           *a2 = 回転の中心x座標
136:     add.w d0, d6            *
137:     asr.w #1, d6            *
138:     move.w d6, a2           *
139:
140:     move.w d3, d6           *a3 = 回転の中心y座標
141:     add.w d1, d6            *
142:     asr.w #1, d6            *
143:     move.w d6, a3           *
144:
145:     sub.w a2, d0            * (d0, d1) = パターン中心を
146:     sub.w a3, d1            * 原点とする左上隅の座標
147:     sub.w a2, d2            * (d2, d1) = パターン中心を
148:                                * 原点とする右上隅の座標
149:
150:     move.w d1, d3           *d1を待避
151:     ROTP d0, d1             *左上の点を回転する
152:     move.w d0, (a0) +       *結果を記録
153:     move.w d1, (a0) +       *
154:
155:     sub.w X0(a1), d0         *右下の点を回転する
156:     sub.w Y0(a1), d1         * (簡略計算)
157:     sub.w X1(a1), d0         *
158:     sub.w Y1(a1), d1         *
159:     neg.w d0                *
160:     neg.w d1                *
161:     move.w d0, 4(a0)         *結果を記録
162:     move.w d1, 6(a0)         *
163:
164:     ROTP d2, d3             *右上の点を回転する
165:     move.w d2, 8(a0)         *結果を記録
166:     move.w d3, 10(a0)        *
167:
168:     sub.w X1(a1), d2         *左下の点を回転する
169:     sub.w Y0(a1), d3         * (簡略計算)
170:     sub.w X0(a1), d2         *
171:     sub.w Y1(a1), d3         *
172:     neg.w d2                *

```

```

173:      neg.w    d3          *
174:      move.w   d2, (a0) +  *結果を記録
175:      move.w   d3, (a0) +  *
176:
177:      addq.l    #8, a0
178:
179:      move.l    PAT(a1), (a0) + *パターンアドレスをセット
180:      move.l    XL(a1), (a0) +  *パターンサイズをセット
181:
182:      pea.l     ARGBUF(a6)      *自由変形ルーチンを
183:      jsr       grput          * 呼び出す
184:      addq.l    #4, sp          *
185:
186:      movem.l   (sp)+, d0-d7/a0-a3
187:      unlk     a6
188:      rts

```

.....
リスト5
GRPUT2 TEST.S

```

1: *      grput2のテスト用プログラム
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5: *
6:      .xref         grput2
7: *      .xref         setcliprect
8: *
9: FPACK macro      callno
10:      .dc.w        callno
11:      .endm
12: *
13: _RAND             equ      $fe0e
14: *
15:      .text
16:      .even
17: *
18: ent:
19:      lea.l        inisp, sp
20:
21:      clr.l        -(sp)          *スーパーバイザモードへ
22:      DOS          _SUPER        *
23:
24:      lea.l        arg0, a1       *画面からパターンを取り込む
25:      IOCS         _GETGRM       *
26:
27: *      pea         window        *クリッピングウィンドウの設定
28: *      jsr         setcliprect   *
29: *      addq.l      #4, sp        *
30:
31:      move.l       #512x512, 65536 *12x512, 65536
32:      DOS          _CONCTRL      *画面初期化
33:
34:
35:      lea.l        arg, a1        *a1 = 引数受け渡し領域
36: loop: movea.l     a1, a0          *引数をセット
37:      moveq.l      #4-1, d1       *
38:      loop2: FPACK _RAND          *
39:      lsr.w        #6, d0         *
40:      move.w       d0, (a0) +     *
41:      dbra        d1, loop2      *
42:      *
43:      FPACK _RAND                *
44:      divu.w       #360, d0       *
45:      swap.w       d0             *
46:      move.w       d0, (a0) +     *回転角度
47:
48:      pea.l        (a1)           *描画
49:      jsr          grput2         *
50:      addq.l       #4, sp         *
51:
52:      DOS          _KEYSNS        *キーが押されるまで
53:      tst.l        d0             * 繰り返す

```



```

54:      beq    loop      #
55:
56:      DOS    _INKEY     #スペースが押されたら
57:      cmpi.b #$20,d0    # 一時停止する
58:      bne    done       #
59: pause:  DOS    _INKEY     #
60:      cmpi.b #$20,d0    #
61:      beq    loop       #
62:
63: done:   move.w #-1,-(sp) #キーを読み捨てる
64:      DOS    _KFLUSH     #
65:
66:      move.l #$0010_0000,-(sp) #width 96
67:      DOS    _CONCTRL    #画面再初期化
68:      DOS    _EXIT
69: #
70:      .data
71:      .even
72: #
73: HSIZE equ 128
74: VSIZE equ 128
75: #
76: arg:   #grput2の引数
77:      .dc.w  0,0,0,0
78:      .dc.w  0
79:      .dc.l  pat
80:      .dc.w  HSIZE-1,VSIZE-1
81: #
82: arg0:  #IOCS GETGRMの引数
83:      .dc.w  256-HSIZE/2,256-VSIZE/2
84:      .dc.w  255+HSIZE/2,255+VSIZE/2
85:      .dc.l  pat
86:      .dc.l  pate
87: #
88: window: .dc.w 80,80,175,175
89: #
90:      .bss
91:      .even
92: #
93: pat:   .ds.w HSIZE*VSIZE #描画パターン
94: pate:
95: #
96:      .stack
97:      .even
98: #
99:      .ds.l  4096
100: inisp:
101:      .end   ent

```


C
H
A
P
T
E
R

C, C, X-BASS-CHUNK

C, X-BASICとのリンク

ASSEMBLER



最後のこの章では、これまでに作成したサブルーチン群をC言語、X-BASICから利用する方法を見ていく。もちろん、以下の話は、一般的なCの関数、X-BASICの外部関数をマシン語で作成する場合にも通用する。

C言語における変数の扱い方

予備知識として、Cにおける変数の扱いをマシン語の立場から眺めてみる。

まず、Cのデータ型と記憶領域の大きさの関係を示そう。XCにおける基本データ型とその占める領域は、

char	1バイト
short	2バイト
int	4バイト
long	4バイト
float	4バイト
double	8バイト

である。また、ポインタは、それが何を指していようと常に4バイトだ。さらに、配列は同じ型のデータが連続したメモリに要素数だけ並んだもの¹⁾であり、要素のサイズ×要素数分の領域を占める。構造体はいくつかのデータ型を寄せ集めてまとめて扱えるようにしたもの²⁾だから、基本的には構造体の各メンバーのサイズを合計しただけの大きさを持つ。ただし、68000ではアドレスの偶数境界をまたいだワード/ロングワードのアクセスが禁止されている関係で、char型のメンバーの直後に1バイトの詰め物が入る場合がある。

- 1) 配列は .dc がずらずら並んだもの、もしくは、.ds で確保された一連のメモリ領域をイメージすればよい。
- 2) 構造体 (のテンプレート) は .offset によるオフセット表のイメージ。

Cの変数はその記憶場所がどこにとられるかによって次のように分類される。

- 1) メモリ上の固定領域にとられるもの (長命: プログラムの実行中ずっと値を保持し続ける)
- 2) スタック上にとられるもの (短命)
- 3) レジスタ上にとられるもの (短命)

関数の外部で定義された変数と、関数の内部でとくに静的 (static) であると明示された変数は1), 関数内の自動的 (auto) な変数は2), そして、いわゆるレジスタ変数として宣言されたもの (register) は空いているレジスタがあれば3), そうでなければ2) のパターンとなる。

リスト1に示した例では、

- a,b,c,d,g,h は 1)
 e,x,y は 2)
 f は 3)

にそれぞれ該当する。Cでは関数への引数の引き渡しにスタックを利用するので、関数fooの仮引数であるx, yも2) のパターンに含まれることに注目してもらいたい。

リスト1
 CVARTEST.C

```

1: int a = 1;
2: int b;
3:
4: static void foo( x, y )
5: int x, y;
6: {
7:     int e;
8:     register int f;
9:     static int g = 1;
10:    static int h;
11:
12:    e = x;
13:    h = y;
14:    f = g;
15: }
16:
17: static int c = 1;
18: static int d;
19:
20: int main()
21: {
22:     foo( c, 50000 );
23:
24:     return 0;
25: }
```

マシン語に当てはめて考えると、1)はデータセクションやブロックストレージセクション(以下、bss)に確保したワークエリア、2)はスタック上に確保したワークエリアと、サブルーチン呼び出しときにスタックに積まれた引数、3)はレジスタそのものに相当するわけだ。

通用範囲 (scope: スコープ) について分類するなら、次のようになる。

- ア) プログラム中のどこからでも参照できるもの
- イ) 1本のソースファイル中でのみ参照できるもの
- ウ) 特定のブロック内 (“{” ~ “}”) でのみ参照できるもの

関数の外部でstaticをつけずに定義するとア), 関数外部でstaticに定義するとイ), 関数

内で定義するとウ) になる。リスト1では、

a, b は ア)
c, d は イ)
e, f, g, h, x, y は ウ)

にそれぞれ該当する。

AS.X仕様のアセンブリ言語に当てはめてみると、ア) は.xdef (ないしは.globl) で外部宣言されたワーク、イ), ウ) は.xdefで宣言されていない、もしくはスタック上の一時的なワークと考えることができる。また、リスト1には出てきていないが、C言語のextern宣言は、“プログラムのどこか (別ファイルかもしれない) で同名の変数が定義されている”ということコンパイラに知らせるものだから、.xref (ないしは.globl) に相当すると思えばよい。

実際にリスト1をXC Ver.2.00でコンパイルしてみた結果がリスト2。XCは馬鹿正直なコードを出してくれるので、このような例にはうってつけた。リスト1のa~hの変数定義がどのようにコンパイルされているか見てもらおう。

リスト2
CVARTEST.S

```

1:      INCLUDE fefunc. h
2:      .GLOBL  _a
3:      .GLOBL  _main
4:      .XREF   _main
5:      .TEXT
6:      _foo:
7:      LINK    A6, #-4
8:      MOVE.L  D7, -(SP)
9:      MOVE.L  8(A6), -4(A6)    *e = x;
10:     MOVE.L  12(A6), L5       *h = y;
11:     MOVE.L  L4, D7           *f = g;
12:     MOVE.L  (SP)+, D7
13:     UNLK    A6
14:     RTS
15:     _main:
16:     LINK    A6, #0
17:     MOVE.L  #50000, -(SP)    *
18:     MOVE.L  _c, -(SP)        *
19:     JSR     _foo             *foo( c, 50000 );
20:     ADDQ.L  #8, SP           *
21:     MOVEQ.L #0, D0           *return 0;
22:     UNLK    A6              *
23:     RTS                    *
24:     .DATA
25:     _a:
26:     .DC.L   $00000001
27:     _c:
28:     .DC.L   $00000001
29:     L4:
30:     .DC.L   $00000001
31:     .EVEN
32:     .BSS
33:     _d:
34:     .DS.B   4
35:     .COMM   _b, 4
36:     L5:
37:     .DS.B   4
38:     .END

```

・変数 a

“外部からも参照可能なメモリ上のintの変数で初期値は1” だから、データセクション中に

.dc.lで領域が確保される (26行) と同時に.globlによる外部定義が行われている (2行)。

見てのとおり, XCでは, 変数に割り当てられたメモリ領域の先頭アドレスを, “Cにおける識別名の頭にアンダーバー (_) をつけたラベル” で表す³⁾。

- 3) “ _ ” をつけるのは, コンパイラが自分で勝手に使うラベル (ライブラリ中の下請けサブルーチンなども含む) とCソース中の識別名がかちあわないようにするため。

・変数 b

“外部からも参照可能なメモリ上のintの変数で初期値は指定されていない” というパターンだ。Cでは静的な変数は初期値が指定されなければ0で初期化されることになっており, リスト2では, comm疑似命令によってコモン (common: 共通) エリア⁴⁾に4バイト分の0で初期化された領域を確保している (35行)。コモンエリアは, bssの一部で, 複数ファイルで同名のコモンエリアが定義されている場合, リンカはこれを1つにまとめる (大きさがまちまちだった場合は最大のものをとる)。

- 4) ここでいうコモンエリアはHuman68k Ver. 2.0のconfig.sysのcommon行で指定するものとはまったく別のもの。念のため。

・変数 c

“このソース内部でのみ通用するメモリ上のintの変数で初期値は1” だ。28行で.dc.lにより領域は確保されているが, 外部定義はない。

・変数 d

“このソース内部でのみ通用するメモリ上のintの変数で初期値の指定なし” なので, bssに.ds.bにより4バイトの領域が確保されている (34行)。外部定義もない。

・変数 e

関数内で記憶クラスの指定なしに宣言された変数はautoと見なされ, 関数呼び出しのたびにスタックフレーム上に領域が一時的に確保される。リスト2では, 7行で4バイト分のローカルエリアが確保され, 変数eは “-4 (a6)” で参照されているのがわかるだろう。

・変数 f

レジスタ変数だ。XCではレジスタ変数はd3~d7, a3~a5のいずれかに割り当てられる。通常の変数はデータレジスタに, ポインタはアドレスレジスタに, しかも, レジスタ番号の大きいほうから割りつけることに決まっているようだ。リスト2ではd7に変数fが割り当てられている。

・変数 g

関数内でのみ通用するローカル変数だが, staticがついているから領域はメモリ上の固定領域に取られる。リスト2では37行でbssに4バイトの領域が確保されている。ただ, その領域にふられたラベルが “_g” ではなく, “L5” になっている。これは, もし関数の外側でgという別の変数が定義されていても, ラベル定義がぶつからないようにするためだ。この変数gはあ

くまでローカル変数だから、ほかの関数から参照されることはない。

・変数 h

ローカルでstaticな変数に初期値を与えた例だ。データセクションに.dc.lで領域が確保されている (30行)。

・変数 x, y

関数の仮引数だ。auto変数同様、スタックフレーム上の位置で示されている。リスト2では“8 (a6)”が変数x, “12 (a6)”が変数yにあたる。

・関数 foo, main

ついでに、関数にも目を向けてもらいたい。関数fooはstaticに宣言してある。したがって、.xdefによる外部定義は行われず、ほかのソースから参照することはできない。対して、mainは記憶クラスの指定がないので、3行で外部定義が行われている。

余談ながら、4行で外部参照定義された__mainというラベルは、Cのスタートアップルーチンの先頭アドレス、つまりはこのCプログラムの実行開始アドレスを意味している⁵⁾。この定義があることで、リンク時にリンカは__mainというシンボルをライブラリから探し、結果としてスタートアップルーチンをリンクすることになるわけだ。

■ 5) Cでは一部の関数の初期化のために、関数mainの実行に先立って、このスタートアップルーチンが実行される。関数mainはそこからサブルーチンコールされることになる。

Cの関数をマシン語で

そろそろCの関数をアセンブリ言語で書く方法を考えてみよう。上で示した情報から、いくつかの約束ごとが見えてくる。

- 1) 関数は、Cで使う関数名の頭に “_” をつけたラベル名で表されるサブルーチンとして記述する
- 2) そのサブルーチン名は.xdefまたは.globlで外部定義しておく
- 3) 引数の受け渡しはスタックを介して行う
- 4) 呼び出し元の関数でレジスタ変数を使っているかもしれないから、サブルーチン内でd3～d7, a3～a5を使うときには値を保存しておく
- 5) 当然、フレームポインタであるa6とスタックポインタであるa7 (sp) の値も保存する必要がある
また、リスト1と2を注意深く見くらべると、次のようなこともわかる
- 6) 関数へ渡す引数がスタックを積む順序は、Cにおける引数の並び順の逆 (リスト2の17～18行)
- 7) 関数の戻り値はd0レジスタに入れて返す (リスト2の21行)⁶⁾。ただし、foo () のようなvoid型の (値を返さない) 関数では戻り値の心配をする必要はない

- 6) doubleや構造体に戻り値とする関数ではまた違ってくるのだが、
ここでは触れない。

では、試しにChapter8で作成したサブルーチンgcopyをCプログラムから呼び出せるようにしてみよう。たんに呼び出せるようにするだけであれば、非常にかんたんだ。Chapter8のリスト8 (GCOPY.S) 中、6行の

```
.xdef gcopy
```

の直後に、

```
.xdef _gcopy
```

を、また、24行の、

```
gcopy:
```

の直後に、

```
_gcopy:
```

を挿入し、アセンブルし直せば、サブルーチン_gcopy (実体はサブルーチンgcopyと同じもの) がgcopy () という名前の関数としてCから呼び出せる形式になる。

ただ、サブルーチンgcopyは転送元/転送先の領域の座標を任意のメモリ領域に書き込み、その先頭アドレスをスタックに積んで呼び出すようにできており、これをCで実現するにはリスト3のように構造体に引数を代入し、その構造体の先頭アドレスを関数に渡すしかない。また、gcopy内ではスーパーバイザ空間にアクセスすることになるので、gcopy () を呼び出す前にあらかじめスーパーバイザモードへ移行しておく必要もある。仕様です、と突っぱねることもできるが、どうせCの関数にするのなら、

```
gcopy (0, 0, 50, 50, 100, 0) ;
```

のような素直な形式で、ユーザモードからも呼び出せるようにしたいところだ。

.....
リスト3
CALLGCOPY.S

```
1: main()
2: {
3:     struct GCOPYBUF {
4:         short  x0, y0;
5:         short  x1, y1;
6:         short  x2, y2;
7:     } gcopybuf;
8:
9:     :
10:
11:     gcopybuf.x0 = 0;
12:     gcopybuf.y0 = 0;
13:     gcopybuf.x1 = 50;
14:     gcopybuf.y1 = 50;
15:     gcopybuf.x2 = 100;
16:     gcopybuf.y2 = 0;
17:     gcopy( &gcopybuf );
18:
19:     :
20: }
```

そこで、また別の方法を検討する。サブルーチンgcopyとCプログラムとの間にワンクッシ

ョン入れて、つじつまをあわせることにしよう。リスト4だ。サブルーチンの頭でスーパーバイザモードへ移行し、スタック上に積まれた座標値をサブルーチンgcopyに渡せるかたちに再構成してからgcopyをコールし、そののちふたたびB_SUPERでユーザーモードに戻している。

リスト4
_GCOPY.S

```

1: #      gcopy ()
2:
3:      .include      iocscall.mac
4: #
5:      .xdef      _gcopy
6:      .xref      gcopy
7: #
8:      .offset 8      *gcopy () の引数構造
9: #
10: X0:      .ds. l      1
11: Y0:      .ds. l      1
12: X1:      .ds. l      1
13: Y1:      .ds. l      1
14: X2:      .ds. l      1
15: Y2:      .ds. l      1
16: #
17:      .text
18:      .even
19: #
20: _gcopy:
21:      link      a6, #0
22:
23:      suba. l    a1, a1
24:      IOCS      _B_SUPER
25:
26:      move. w    Y2+2 (a6), -(sp)
27:      move. w    X2+2 (a6), -(sp)
28:      move. w    Y1+2 (a6), -(sp)
29:      move. w    X1+2 (a6), -(sp)
30:      move. w    Y0+2 (a6), -(sp)
31:      move. w    X0+2 (a6), -(sp)
32:
33:      pea. l     (sp)
34:      jsr      gcopy
35:
36:      tst. l     d0
37:      bmi      done
38:
39:      movea. l   d0, a1
40:      IOCS      _B_SUPER
41:
42: done:      unlk      a6
43:      rts
44:
45:      .end

```

スーパーバイザモードへの移行にはいつものDOSコールsuperではなく、IOCSコールのB_SUPERを使ってみた。機能に違いはない。ただ、superではスタックを介して引数の受け渡しを行うのに対し、B_SUPERではスタックのかわりにa1レジスタが使われる。なお、B_SUPERはすでにスーパーバイザモードなのにさらにスーパーバイザモードにしようとした場合にはd0に-1を返すことになっているので、36～37行でそのチェックをしている。関数が呼び出された時点ですでにスーパーバイザモードであれば、そのまま呼び出し元に戻るわけだ。

引数の再構成に関しては、とくに説明するまでもないだろう。スタック上にロングワード単

位で積まれた6つの値を、ワード単位でもう一度スタックに積み直している。その時点で、スタック上にサブルーチンgcopyの引数受け渡し領域と同じ構造のデータができ上がるから、その先頭アドレス(=sp)をgcopyに引き渡せばよい。

こうして作成した関数gcopy()をCから使うときの注意点を挙げておこう。まず、当然、必要なモジュールをすべてリンクしてやらなければならない。_GCOPY.O(リスト4をアセンブルしたもの)とGCOPY.O(Chapter8リスト8をアセンブルしたもの)、さらにはGCOPY.S内で外部参照されているいくつかのサブルーチン、データも忘れずにリンクする。実用上は、これら(何なら本書の全サブルーチン)をひとまとめのライブラリにしておくのが便利だろう。

また、gcopy()内ではグラフィック画面が初期化されているかどうか、画面モードがどうなっているのかのチェックは一切行っていない。画面の初期化は呼び出し元のCプログラム側で管理すること。これは「仕様です」。

さらに、描画ページの切り替えの問題もある。本書で作成したプログラムは描画ページを自前で管理しているから、Cの関数にした場合でも、XCのapage()関数の影響を受けない。この問題は、リスト5のような関数(サブルーチン)を元のapage()のかわりに使うことで回避できる。その際には、リンク順序に注意しよう。コマンドラインで、XCのライブラリよりもリスト5(をアセンブルすることで得られるオブジェクト)を先に指定しないと意味がない。

リスト5
_APAGE.S

```

1: #      apage()
2:
3:      .include      iocscall.mac
4: #
5:      .xdef      _apage
6:      .xref      apage
7: #
8:      .text
9:      .even
10: #
11: _apage:
12: PAGENO =      4+3
13:      move.b      PAGENO(sp), d1
14:      bmi      getapage
15:      ext.w      d1
16:      jmp      apage
17: #
18: getapage:
19:      lOCS      _APAGE
20:      rts
21:
22:      .end

```

ああ、まだあった。gcopyはアセンブル時に実画面1024×1024ピクセルモード用か、512×512ピクセルモード用かを決定するようにつくられていたのだった。1024ピクセルモード用と、512ピクセルモード用の2つのライブラリをつくっておき、プログラムで使う画面モードにおうじてリンクするライブラリを変えるなどの方法で逃げてもらいたい。

X-BASICの外部関数

X-BASICの外部関数の作成方法についてはXC添付の『プログラマーズマニュアル』に詳しい。あえて補足することは何もないのだが、とりあえず実例を1つ示しておきたい。ここでは、Chapter7で作成したエフェクト関連サブルーチンをひとまとめにしたGFX.FNCをつくってみる。ただし、この外部関数は実画面モードが512×512ピクセル、基本的には65536色モードでのみ使用可能とする。

X-BASICの外部関数ファイルはXファイルと同じ形式をとる。作成手順も普通のマシン語プログラムと変わらない。ただし、プログラムの頭の部分にインフォメーションテーブルと呼ばれるヘッダを置くことになっている (図1)。実例を見ないとピンとこないかもしれないが、かんたんに解説しておこう。

```

+00H  1L X-BASIC起動時に呼び出されるサブルーチン先頭アドレス
+04H  1L RUN命令実行時に呼び出されるサブルーチン先頭アドレス
+08H  1L END命令実行時に呼び出されるサブルーチン先頭アドレス
+0CH  1L SYSTEM命令, EXIT () 関数実行時に呼び出されるサブルーチン先頭アドレス
+10H  1L BREAK, ^Cによりプログラムが中断されたときに呼び出されるサブルーチン先頭アドレス
+14H  1L ^D入力時に呼び出されるサブルーチン先頭アドレス
+18H  1L 予備
+1CH  1L 予備
+20H  1L トークンテーブル先頭アドレス
+24H  1L パラメータテーブル先頭アドレス
+28H  1L 実行アドレステーブル先頭アドレス
+2CH  5L 予備 (0で埋めておく)
      ⋮
+3FH

```

図1
X-BASIC外部関数
インフォメーションテーブル

インフォメーションテーブルの頭の部分には何か特別なイベントが発生したときに呼び出される6つのルーチンの先頭アドレスが並ぶ。これらのすべてをサポートする必要はないが、サポートしないものについてはrts命令が置かれたアドレスを格納しておかなければならない⁷⁾。続く2ロングワードは将来の拡張用で、いまのところはrtsが置かれたアドレスを入れておけばよい。

■ 7) 外部関数とそのイベントをサポートしているかどうかに関わらず、X-BASICインタプリタはイベント発生時に無条件にこのテーブルを参照し、該当アドレスをサブルーチンコールする。

その後ろに、トークンテーブル、パラメータテーブル、関数の実行アドレステーブルの各先頭アドレスが続く。トークンテーブルは外部関数ファイル内の全関数名を0で終わる文字列のかたちで並べたものだ。テーブルの最後を示す意味でトークンテーブルの末尾にはもう1個余分な0を置く。

パラメータテーブルにはポインタが並び、“そのポインタの指す先”に、パラメータIDとか呼ばれているコードで各関数の引数/戻り値の型情報を置く。パラメータIDは1つの引数/戻り値あたり16ビットで、各ビットにはいちおう図2に示すような意味が割り振られている。ただし、X-BASICはすべての組みあわせをサポートしているわけではないようだ。なお、XC添付のFDEF.Hというファイルには、よく使われるパラメータIDがシンボル定義されている。

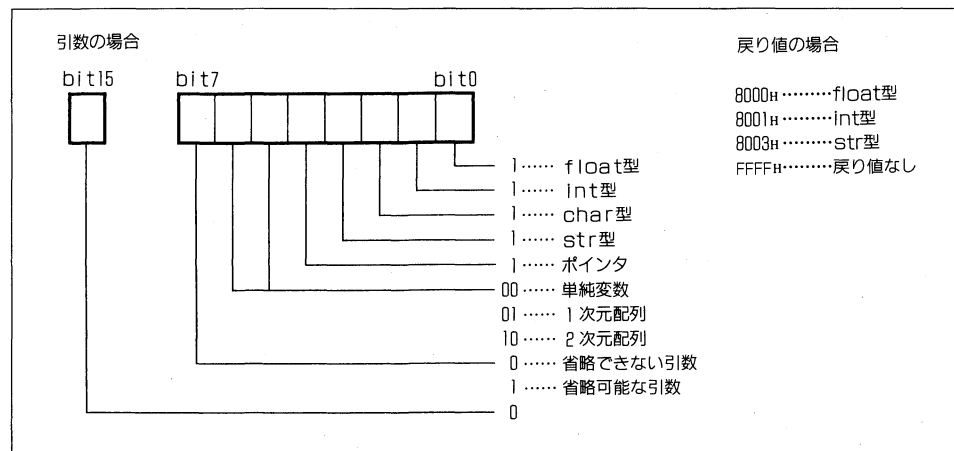


図2

X-BASIC外部関数パラメータID

最後の実行アドレステーブルは言葉通り各関数の処理ルーチン本体の先頭アドレスを並べたものだ。

ここで、トークンテーブル、パラメータテーブル、実行アドレステーブル内の関数の並びはすべて同じ順序でなければならない。

さて、X-BASICの外部関数への引数受け渡しにはやはりスタックが用いられ、C同様、引数の並びの後ろから順に積まれる。ただし、スタックには引数の値に加えて、各引数の型、引数の総個数が積まれる（図3）。1つの引数あたり10バイト、うち、値の格納用に8バイトが割り当てられているわけだが、実際に8バイトすべてが使われるのはfloat型の場合だけで、int型の場合は下位4バイトのみ、char型の場合は下位の1バイトのみが使われ、残りは0で埋められる。また、str型/配列の場合は下位4バイトにデータへのポインタが格納されている。このポインタはstr型の場合は文字列の先頭を指すが、配列の場合は図4に示すような構造を持った

メモリ領域を指す。なお、図4は1次元配列の場合で、多次元配列の場合はもう少し情報量が増える。

SP+00 _H	1 L	リターンアドレス
SP+04 _H	1 W	引数の総個数
SP+06 _H	1 W	第1引数の型
SP+08 _H	1 L	第1引数の値 (上位4バイト)
SP+0C _H	1 L	第1引数の値 (下位4バイト)
SP+10 _H	1 W	第2引数の型
SP+12 _H	1 L	第2引数の値 (上位4バイト)
SP+16 _H	1 L	第2引数の値 (下位4バイト)
SP+1A _H	1 W	第3引数の型

図3

X-BASIC外部関数へ渡される引数

+00 _H	1 L	X-BASIC内部で使用
+04 _H	1 W	配列の次元数-1
+06 _H	1 W	配列要素1個のバイト数
+08 _H	1 W	配列の要素数-1 (添え字の最大値)
+0A _H		配列の内容

図4

X-BASICの配列

関数からの戻り値は引数と同じような構造の10バイトの領域 (ただし、型の部分は0にする) に値を収め、その先頭アドレスをa0に入れて返す。くわえて、エラーなく処理が終了した場合にはd0.1に0を、エラーの場合はd0.1に0以外のエラーコードを、a1にエラーメッセージへのポインタを返すことになっている。ただし、負のエラーコードにはHuman68kのDOSコールと同等の意味が割り当てられているようで、とくにa1で指定しなくても適当なメッセージが表示される。外部関数特有のエラーメッセージを表示したい場合は、正のエラーコードを使えばよい。なお、正常終了時はa1は無視され、エラー時にはa0が無視される。もっとも、error offによりエラーを無視する設定になっているときには、エラー時にもa0が参照されるので、とりえずa0が変なアドレスを指すことがないようにしなければならない。

では、リスト6を見てもらおう。各関数の実行ルーチン本体は含まれていない。引数や戻り値のつじつまあわせだけを行い、処理自体はすでにつくってあるサブルーチンにまかせるようになっている。

リスト6
GFX.S

```

1: #      X-BASIC用エフェクト系グラフィック関数
2:
3:      .include      doscall.mac
4:      .include      iocscall.mac
5:      .include      fdef.h
6:      .include      gmacro.h
7: #
8: #int_val      equ      $0002      #int型の引数
9: #aryl_c      equ      $0034      #char型1次元配列
10: #aryl_fic    equ      $0037      #数値型1次元配列
11: #int_ret     equ      $8001      #int型の戻り値
12: #
13:      .xref      gandcolor
14:      .xref      gxorcolor
15:      .xref      gorcolor
16:      .xref      gaddcolor
17:      .xref      gsubcolor
18:      .xref      gmixcolor
19:      .xref      gnegate
20:      .xref      ghalftone
21:      .xref      gmonotone
22:      .xref      gmonotone2
23:      .xref      gmonotone2_hsv
24:      .xref      gmosaic
25:      .xref      gdefocus
26:      .xref      gmedian
27:      .xref      gaccent
28:      .xref      gdifferential
29:      .xref      glaplacian
30:      .xref      gsharp
31:      .xref      gdither
32:      .xref      gdither_bin
33:      .xref      gltom
34:      .xref      gltom2
35:      .xref      ggradfill
36:      .xref      ggradreplace
37: #
38:      .xref      hsvtorgb
39:      .xref      rgbtohsv
40:      .xref      ginitpalet_l
41:      .xref      ginitpalet_m
42:      .xref      ginitpalet_m2
43:      .xref      ginitpalet_s
44:      .xref      gnegatepalet
45: #
46:      .offset 0      #スタックフレーム
47: #
48: A6BUF: .ds.l      1
49: RETADR: .ds.l      1
50: ARG0: .ds.w      1
51: ARG1: .ds.b      10
52: ARG2: .ds.b      10
53: ARG3: .ds.b      10
54: ARG4: .ds.b      10
55: ARG5: .ds.b      10
56: ARG6: .ds.b      10
57: ARG7: .ds.b      10
58: ARG8: .ds.b      10
59: ARG9: .ds.b      10
60: #
61:      .offset 0      #引数バッファ
62: #
63: TYPE: .ds.w      1      #型
64: FVAL: .ds.l      1      #実数
65: PVAL:      #ポインタ (配列, 文字列)
66: LVAL: .ds.w      1      #32ビット数
67: WVAL: .ds.b      1      #16ビット数
68: BVAL: .ds.b      1      #8ビット数
69: #
70:      .offset 0      #X-BASICの配列
71: #
72: ADUMMY: .ds.l      1
73: ADIM: .ds.w      1      #次元数-1

```

```

74: ASIZ: .ds.w 1 #1要素のバイト数
75: ALEN: .ds.w 1 #要素数-1
76: ADAT: #データ本体
77: #
78: .text
79: .even
80: #
81: information_table:
82: .dc.l dummy #X-BASIC起動時
83: .dc.l dummy #run
84: .dc.l dummy #end
85: .dc.l dummy #system,exit
86: .dc.l dummy #BREAK,^C
87: .dc.l dummy #^D
88: .dc.l dummy #予備
89: .dc.l dummy #予備
90: .dc.l token_table
91: .dc.l param_table
92: .dc.l exec_table
93: .dc.l 0,0,0,0,0 #予備
94: #
95: token_table:
96: .dc.b 'gandcolor',0
97: .dc.b 'gxorcolor',0
98: .dc.b 'gorcolor',0
99: .dc.b 'gaddcolor',0
100: .dc.b 'gsubcolor',0
101: .dc.b 'gmixcolor',0
102: .dc.b 'gnegate',0
103: .dc.b 'ghalftone',0
104: .dc.b 'gmonotone',0
105: .dc.b 'gmonotone2',0
106: .dc.b 'gmonotone2_hsv',0
107: .dc.b 'gmosaic',0
108: .dc.b 'gdefocus',0
109: .dc.b 'gmedian',0
110: .dc.b 'gaccent',0
111: .dc.b 'gdifferential',0
112: .dc.b 'glaplacian',0
113: .dc.b 'gsharp',0
114: .dc.b 'gdither',0
115: .dc.b 'gdither_bin',0
116: .dc.b 'gltom',0
117: .dc.b 'gltom2',0
118: .dc.b 'ggradfill',0
119: .dc.b 'ggradreplace',0
120: .dc.b 'ginitpalet_1',0
121: .dc.b 'ginitpalet_m',0
122: .dc.b 'ginitpalet_m2',0
123: .dc.b 'ginitpalet_s',0
124: .dc.b 'gnegatepalet',0
125: .dc.b 'hsvtorgb',0
126: .dc.b 'rgbttohsv',0
127: .dc.b 0 #テーブル終端
128: .even
129: #
130: param_table:
131: .dc.l par_5 #gandcolor
132: .dc.l par_5 #gxorcolor
133: .dc.l par_5 #gorcolor
134: .dc.l par_5 #gaddcolor
135: .dc.l par_5 #gsubcolor
136: .dc.l par_5 #gmixcolor
137: .dc.l par_4 #gnegate
138: .dc.l par_4 #ghalftone
139: .dc.l par_4 #gmonotone
140: .dc.l par_4 #gmonotone2
141: .dc.l par_6 #gmonotone2_hsv
142: .dc.l par_5 #gmosaic
143: .dc.l par_4 #gdefocus
144: .dc.l par_4 #gmedian
145: .dc.l par_5 #gaccent
146: .dc.l par_4 #gdifferential
147: .dc.l par_4 #glaplacian

```



```

148: .dc.l par_4 #gsharp
149: .dc.l par_6 #gdither
150: .dc.l par_5 #gdither_bin
151: .dc.l par_4 #gltom
152: .dc.l par_4 #gltom2
153: .dc.l par_8 #ggradfill
154: .dc.l par_9 #ggradreplace
155: .dc.l par_0 #ginitpalet_l
156: .dc.l par_0 #ginitpalet_m
157: .dc.l par_0 #ginitpalet_m2
158: .dc.l par_0 #ginitpalet_s
159: .dc.l par_0 #gnegatepalet
160: .dc.l par_3 #hsvtorgb
161: .dc.l par_1a #rbttohsv
162: #
163: par_9: .dc.w int_val #引数はint 9 個
164: par_8: .dc.w int_val #引数はint 8 個
165: par_7: .dc.w int_val #引数はint 7 個
166: par_6: .dc.w int_val #引数はint 6 個
167: par_5: .dc.w int_val #引数はint 5 個
168: par_4: .dc.w int_val #引数はint 4 個
169: par_3: .dc.w int_val #引数はint 3 個
170: par_2: .dc.w int_val #引数はint 2 個
171: par_1: .dc.w int_val #引数はint 1 個
172: par_0: .dc.w int_ret #戻り値はint
173: #
174: par_1a: .dc.w int_val #引数はint 1 個
175: .dc.w aryl_c #引数はchar配列
176: .dc.w int_ret #戻り値はint
177: #
178: exec_table:
179: .dc.l x_gandcolor
180: .dc.l x_gxorcolor
181: .dc.l x_gorcolor
182: .dc.l x_gaddcolor
183: .dc.l x_gsubcolor
184: .dc.l x_gmixcolor
185: .dc.l x_gnegate
186: .dc.l x_ghalftone
187: .dc.l x_gmonotone
188: .dc.l x_gmonotone2
189: .dc.l x_gmonotone2_hsv
190: .dc.l x_gmosaic
191: .dc.l x_gdefocus
192: .dc.l x_gmedian
193: .dc.l x_gaccent
194: .dc.l x_gdifferential
195: .dc.l x_glaplacian
196: .dc.l x_gsharp
197: .dc.l x_gdither
198: .dc.l x_gdither_bin
199: .dc.l x_gltom
200: .dc.l x_gltom2
201: .dc.l x_ggradfill
202: .dc.l x_ggradreplace
203: .dc.l x_ginitpalet_l
204: .dc.l x_ginitpalet_m
205: .dc.l x_ginitpalet_m2
206: .dc.l x_ginitpalet_s
207: .dc.l x_gnegatepalet
208: .dc.l x_hsvtorgb
209: .dc.l x_rbttohsv
210: #
211: # 引数がint 4 個の関数
212: #
213: x_gnegate:
214: lea.l gnegate, a5
215: bra exec_4
216: x_ghalftone:
217: lea.l ghalftone, a5
218: bra exec_4
219: x_gmonotone:
220: lea.l gmonotone, a5
221: bra exec_4

```

```

222: x_gmonotone2:
223:     lea.l    gmonotone2, a5
224:     bra      exec_4
225: x_gdefocus:
226:     lea.l    gdefocus, a5
227:     bra      exec_4
228: x_gmedian:
229:     lea.l    gmedian, a5
230:     bra      exec_4
231: x_gdifferential:
232:     lea.l    gdifferential, a5
233:     bra      exec_4
234: x_glaplacian:
235:     lea.l    glaplacian, a5
236:     bra      exec_4
237: x_gsharp:
238:     lea.l    gsharp, a5
239:     bra      exec_4
240: x_gltom:
241:     lea.l    gltom, a5
242:     bra      exec_4
243: x_gltom2:
244:     lea.l    gltom2, a5
245:     bra      exec_4
246: *
247: *       引数がint 5 個の関数
248: *
249: x_gandcolor:
250:     lea.l    gandcolor, a5
251:     bra      exec_5
252: x_gxorcolor:
253:     lea.l    gxorcolor, a5
254:     bra      exec_5
255: x_gorcolor:
256:     lea.l    gorcolor, a5
257:     bra      exec_5
258: x_gaddcolor:
259:     lea.l    gaddcolor, a5
260:     bra      exec_5
261: x_gsubcolor:
262:     lea.l    gsubcolor, a5
263:     bra      exec_5
264: x_gmixcolor:
265:     lea.l    gmixcolor, a5
266:     bra      exec_5
267: x_gmosaic:
268:     lea.l    gmosaic, a5
269:     bra      exec_5
270: x_gaccent:
271:     lea.l    gaccent, a5
272:     bra      exec_5
273: x_gdither_bin:
274:     lea.l    gdither_bin, a5
275:     bra      exec_5
276: *
277: *       引数がint 6 個の関数
278: *
279: x_gdither:
280:     lea.l    gdither, a5
281:     bra      exec_6
282: x_gmonotone2_hsv:
283:     lea.l    gmonotone2_hsv, a5
284:     bra      exec_6
285: *
286: *       グラデーション関係
287: *
288: x_ggradfill:
289:     link     a6, #0
290:     lea.l    ggradfill, a5
291:     pea.l    temp
292:     move.w   ARG8+WVAL(a6), -(sp)
293:     move.w   ARG7+WVAL(a6), -(sp)
294:     bra      exec6
295: *

```

```

296: x_ggradreplace:
297:     link    a6, #0
298:     lea.l   ggradreplace, a5
299:     pea.l   temp
300:     move.w  ARG9+WVAL(a6), -(sp)
301:     move.w  ARG8+WVAL(a6), -(sp)
302:     move.w  ARG7+WVAL(a6), -(sp)
303:     bra     exec6
304: #
305: #       バレット関係
306: #
307: x_ginitpalet_l:
308:     link    a6, #0
309:     lea.l   ginitpalet_l, a5
310:     bra     exec
311: x_ginitpalet_m:
312:     link    a6, #0
313:     lea.l   ginitpalet_m, a5
314:     bra     exec
315: x_ginitpalet_m2:
316:     link    a6, #0
317:     lea.l   ginitpalet_m2, a5
318:     bra     exec
319: x_ginitpalet_s:
320:     link    a6, #0
321:     lea.l   ginitpalet_s, a5
322:     bra     exec
323: x_gnegatepalet:
324:     link    a6, #0
325:     lea.l   gnegatepalet, a5
326:     bra     exec
327: #
328: #       色コード関係
329: #
330: x_hsvtorgb:
331:     link    a6, #0
332:     move.w  ARG1+WVAL(a6), d0
333:     swap.w  d0
334:     move.b  ARG2+BVAL(a6), d0
335:     lsl.w   #8, d0
336:     move.b  ARG3+BVAL(a6), d0
337:     jsr     hsvtorgb
338:     andi.l  #$0000ffff, d0
339:     lea.l   retval(pc), a0
340:     move.l  d0, LVAL(a0)      *戻り値
341:     moveq.l #0, d0
342:     unlk    a6
343:     rts
344: #
345: x_rgbtohsv:
346:     link    a6, #0
347:     move.w  ARG1+WVAL(a6), d0
348:     movea.l ARG2+PVAL(a6), a0
349:
350:     cmpi.w  #3-1, ALEN(a0)    *バッファは3バイト以上必要
351:     bcs     error3            #
352:     jsr     rgbtohsv
353:     lea.l   ADAT+3(a0), a0
354:     move.b  d0, -(a0)          #v
355:     lsr.w   #8, d0
356:     move.b  d0, -(a0)          #s
357:     swap.w  d0
358:     move.b  d0, -(a0)          #h
359:     bra     okretn
360: #
361: exec_6:
362:     link    a6, #0
363:     move.w  ARG6+WVAL(a6), -(sp)
364:     bra     exec5
365: #
366: exec_5:
367:     link    a6, #0
368:     move.w  ARG5+WVAL(a6), -(sp)
369:     bra     exec4

```

```

370: #
371: exec_4:
372:     link    a6, #0
373: exec4:    move.w  ARG4+WVAL(a6), -(sp)
374:     move.w  ARG3+WVAL(a6), -(sp)
375:     move.w  ARG2+WVAL(a6), -(sp)
376:     move.w  ARG1+WVAL(a6), -(sp)
377:
378:     moveq.l  #-1, d1      #グラフィック画面は
379:     lOCS     _APAGE      #
380:     tst.b    d0          # 初期化されているか?
381:     bmi     error1
382:
383:     moveq.l  #-1, d1      #画面モードを取得
384:     lOCS     _CRTMOD     # (d0 = 12~15... 65536色モード)
385:     cmpi.b   #12, d0     #
386:     bcs     error2       #
387:     cmpi.b   #15+1, d0   #
388:     bcc     error2       #
389:
390: exec:     pea.l    (sp)    #引数受け渡し
391:
392: do:       suba.l   a1, a1  #スーパーバイザモードへ
393:     lOCS     _B_SUPER    #
394:
395:     jsr      (a5)        #実行ルーチン本体
396:     #d0.lは保存される前提
397:     tst.l    d0
398:     bmi     done        #最初からスーパーバイザモードだった
399:
400:     movea.l  d0, a1      #ユーザーモードへ復帰
401:     lOCS     _B_SUPER    #
402:
403: okretn:    moveq.l  #0, d0    #正常終了
404:
405: done:     lea.l    retval(pc), a0 #
406:     move.l    d0, LVAL(a0)    #戻り値
407:
408:     unlk     a6
409: dummy:    rts
410: #
411: error1:   lea.l    errms1(pc), a1 #画面が初期化されていない
412:     bra      error
413: error2:   lea.l    errms2(pc), a1 #65536色モードではない
414:     bra      error
415: error3:   lea.l    errms3(pc), a1 #配列の大きさが足りない
416: error:    moveq.l  #1, d0
417:     bra      done
418: #
419: failsafe:
420:     DOS      _EXIT
421: #
422: retval:   .dc.w    0        #戻り値格納用
423:     .dc.l    0, 0         #
424: #
425: errms1:   .dc.b    '画面が初期化されていません', 0
426: errms2:   .dc.b    'この関数は65536色モードでのみ動作します', 0
427: errms3:   .dc.b    '配列の大きさが不十分です', 0
428:     .even
429: #
430:     .bss
431:     .even
432: #
433: temp:     .ds.b    12*1024+100 #gggradfill, gradreplaceのワーク
434: #
435:     .end    failsafe

```

まず、インクルードファイルを取り込んでいる (3~6行)。FDEF.HはXCのものを流用しているが、不幸にしてFDEF.Hが手元にない人のために8~11行に注釈のかたちで必要部

分を抜粋してある。続いて、外部参照定義（13～44行）、スタックフレームなどのデータ構造のオフセット表定義（46～76行）が並び、81行からがインフォメーションテーブルだ。イベントに対応した処理ルーチンは使っていないので、どれもrtsを指している。95行からのトークンテーブルは偶数境界の問題を避ける意味で最後にeven疑似命令が置いてある⁸⁾。

■ 8) 実際に文字列が何文字かは数えていない。もし、もともとトークンテーブルが偶数バイトだった場合にはevenは単に無視されるだけだからだ。

その後ろ、130行以降がパラメータテーブル。先に触れたとおり、パラメータテーブルには“関数の引数/戻り値の個数/型を表す情報へのポインタ”を並べる。この外部関数ファイル中では引数の数や型が同一の関数が多いので、パラメータID列もある程度共通化してメモリを節約している。ちなみに、どの関数も戻り値はintで、RGBとHSVの変換をする2関数以外は、正常終了時0、エラー時1を関数の戻り値とする（と決めた）。なお、ここでいっている関数の戻り値はd0に返すエラーコードではなく、X-BASICプログラム側から見た関数の戻り値だが、結局はd0にも同じ値を返すようにつくってある。

178行からの実行アドレステーブルはいいとして、213行から始まる各関数の処理ルーチンを見てもらおう。関数の多くは“引数の数”と“どの既存サブルーチン呼び出すか”が異なるだけなので、処理の多くの部分は（紙面の無駄遣いを避ける意味で）やはり共通化してあり、なんだかんだでほとんどの関数は373行で合流している（その分braだらけになってしまったが⁹⁾）。

378～388行がエラーチェックを行っている部分だ。先ほど、Cの関数を作成した際にはエラーチェックを省略したが、さすがに、X-BASICの外部関数の場合は最低限のチェックは必要だろう。ここでは、IOCSコールAPAGEでグラフィック画面が初期化されているかどうかと、IOCSコールCRTMODにより画面モードが65536色モードかどうかを確認し、違っていたらその場でエラー処理ルーチンに飛んでエラー終了している。ちなみに、終了時点のスタックにはいくつもの値が積み放しになっているが、408行のunlkでご破算になるから問題はない。

A
P
P
E
N
D
I
X

添付ディスクについて

添付ディスクについて

本書の添付ディスクには《入門編》,《グラフィックス編》の全ソースファイル+αが収録されている。詳細については,関連各章,添付ディスクのルートディレクトリにあるファイル README, および, DOCディレクトリ中のドキュメントファイルを参照してもらうとして,この場ではごくかんたんに内容を紹介しておく。

なお,頼みもしないのに本書のプログラムを含む全内容は日本の著作権法の保護下にあるが,添付ディスクに収めたプログラムとドキュメントは,読者各自の責任において,形態・目的を問わず,使用・改造・配付してもらってかまわない。有効に活用していただければむしろ幸いである。ただし,添付ディスク中には一部,『C Compiler PRO-68K』からの引用が含まれるので,著作権法上,添付ディスクまるごとそのままの配付は許されない。

ディレクトリ構成

¥		
—	DOC	簡易マニュアル
—	BIN	実行ファイル
—	SYS	デバイスドライバ
—	INCLUDE	インクルードファイル
—	LIB	ライブラリ
—	BASIC	X-BASIC外部関数
—	VOL1	《入門編》 各章ごとのソースプログラム
	—CHAP_1	
	—CHAP_2	
	⋮	
—	VOL2	《グラフィックス編》 各章ごとのソースプログラム
	—CHAP_1	
	—CHAP_2	
	⋮	
—	SUPPLEMENT	おまけ

ファイル

●DOC

DOCディレクトリには添付ディスク中の実行ファイル, サブルーチン, CおよびX-BASIC用関数の簡易マニュアルが収められている。

●BIN

BINディレクトリには、《入門編》、《グラフィックス編》で作成したプログラムのうち、そのまま実用に使えるような（使えなくもない）プログラムの実行ファイルが入っている。

《入門編》からは、

UPPER.X	半角英小文字→大文字フィルタ
RDERR.X	標準エラー出力をリダイレクトする
FILELIST.X	ファイルの一覧を表示する
ATR.X	ファイル属性を表示する
REVATR.X	不可視属性を反転する
FNOON.X	タイムスタンプを12:00:00に揃える
DEVICE.X	デバイスドライバの一覧を表示する
PCMMODE.X	PCMデバイスのモードを切り替える

を収めた（こうして並べると《入門編》では本当に実用的なプログラムは何ひとつ作成しなかったことがばれる）。《グラフィックス編》からは、数少ない実行ファイル、

GLLOAD.X	簡易GL3ファイルローダ
GLOADON.X	GL3ファイル重ね合わせローダ

にくわえ、以下の小物が収録されている。

MSRECT.X	矩形座標をマウスで指定するための補助プログラム
CRTMOD.X	画面モードを切り替える
POINT.X	マウスカーソル位置の色コードを表示する
GON.X	グラフィック画面を表示状態にする
GPINIT.X	グラフィックパレットを初期化する
T.X	プログラムの実行時間を計る

また、ディスク中には《グラフィックス編》で作成した各描画サブルーチンをコマンドラインから直接呼び出すプログラムを

GFILL.X	矩形を塗り潰す
GLINE.X	線分を描く

⋮

のノリで自動生成するバッチファイルを用意してある。これらの“グラフィック描画コマンド”群の使用法は、Chapter 7のFXMAIN.Sに準じるが、コマンドライン引数として、

10進数

“\$” から始まる16進数

以外に、

“_” から始まる2進数

“[B R G]”形式のRGBによる色指定

“(H S V)”形式のHSVによる色指定

も受け付けるよう拡張されている。たとえば、

```
A>GFILL 0 0 511 511 62
A>GFILL 0 0 511 511 $3E
A>GFILL 0 0 511 511 _111110
A>GFILL 0 0 511 511 [31 0 0]
A>GFILL 0 0 511 511 (128 31 31)
```

はどれももっとも明るい青（ただし、輝度ビット0）による全画面塗り潰しを意味する。

また、MSRECT.Xを使うとこれら描画コマンドの座標指定をマウスで行うことができる。

MSRECT.Xは、

```
A>MSRECT GFILL 62
```

のようにして起動し、マウスで矩形を指定すると、内部で

```
A>GFILL X0 Y0 X1 Y1 62
```

のようなコマンドラインを組み立て、実行する。

●SYS

SYSディレクトリには、《入門編》、《グラフィックス編》を通じて作成した唯一の実用デバイスドライバ、

```
TAPDRV.SYS      コンソールへの出力と平行して、データをバッファに溜め込み、
                  必要ならあとから取り出せるキャラクタデバイス
```

を収めてある。TAPDRV.SYSについてはDOCディレクトリ中のドキュメント、および、《入門編》Chapter B参照のこと。

●INCLUDE

INCLUDEディレクトリには《入門編》、《グラフィックス編》の各プログラムをアセンブルするのに必要なインクルードファイルが1セット収めてある。これらすべてのファイルをカレントディレクトリ（AS.X Ver.2.0などを使う場合は環境変数“include”の指すディレクトリ）に置いておけば、添付ディスク中の全ソースが間違いなくアセンブルできる。

INCLUDE中のファイルのうち、

```
DOSCALL.MAC      Human68k DOSコール名定義
IOCSCALL.MAC     IOCSコール名定義
FEFUNC.H         FLOATn.Xファンクションコール名定義
FDEF.H           X-BASIC外部関数作成用定数定義
```

は、『C Compiler PRO-68K』からの引用である。FEFUNC.H以外はVer.2.0、FEFUNC.HのみはVer.1.01相当の版を収録している。FEFUNC.Hについては、Ver.2.0の版と区別するために、FEFUNC.MACにリネームして使うことを推奨したい。

●LIB

LIBディレクトリには、《入門編》、《グラフィックス編》で作成したサブルーチンをそれぞれまとめたライブラリ、

MLIB.A	《入門編》
G512LIB.A	《グラフィックス編》(実画面512ピクセル用)
G1024LIB.A	《グラフィックス編》(実画面1024ピクセル用)

および、C言語用のライブラリ、

GCLIB.A

を収めてある。ただし、ディスク容量の都合でG512LIB.A、G1024LIB.A、GCLIB.AはLHA.X(吉崎栄泰, 岡田紀雄作)による自己解凍書庫の形で収録した。LIBディレクトリ中のGLIB.Xを実行すれば、これら3本のライブラリがカレントディレクトリに展開される。各ライブラリファイルとも、.L形式のライブラリに対応したリンクを使用する場合は、

A > LIB MLIB MLIB.A

のようにして、LIB.X(『C Compiler PRO-68K Ver.2.0』に付属)で.L形式に変換しておくのがよいだろう。

MLIB.Aには《入門編》で作成したいくつかのサブルーチンにくわえ、ソーティングサブルーチン群と、C言語ライクなメモリ管理サブルーチン群が含まれる。

G512LIB.Aは《グラフィックス編》で作成した全サブルーチンと若干の追加分、G1024LIB.Aにはそのうち実画面1024ピクセルモードで使えるサブルーチンからなる。ただし、パレット関係など、実画面に依存しないサブルーチンはG512LIB.Aにのみ収録してあるので、G1024LIB.Aを使用するときには、

A > LK FOO G1024LIB.A G512LIB.A

のように、G512LIB.Aも一緒にリンクする必要がある。このとき、かならずG1024LIB.Aを先に指定すること。

GCLIB.Aは《グラフィックス編》で作成したサブルーチンをCの関数として使えるようにするためのインタフェース関数群(未完)である。基本的に描画処理ルーチン本体は含まないので、使用時にはG512LIB.A(必要ならG1024LIB.A)もまとめてリンクする。なお、GCLIB.A内の一部関数はXC標準ライブラリ内の関数と差し替わるかたちになるため、GCLIB.AはXC標準ライブラリよりも先にリンクしなければならない。

●BASIC

BASICディレクトリには、《グラフィックス編》で作成したサブルーチン+ α を詰め込んだX-BASIC用外部関数、

GRAPH2.FNC

を収めてある。GRAPH2.FNCはChapterCで作成したGFX.FNCを包含し、かつ、X-BASIC標準のGRAPH.FNCの上位互換となっている。やはり、自己解凍書庫になっており、

GRAPH2FNC.Xを実行すれば展開される。あとはBASIC.Xと同じディレクトリにコピーしたうえで、BASIC.CNFの

```
FUNC = GRAPH
```

の行を

```
FUNC = GRAPH2
```

に変更して組みこむ。GRAPH.FNCとGRAPH2.FNCの両方を同時に組みこむと、関数名がダブる部分については“先に組みこんだほう”が有効になる（X-BASICの仕様）ので注意してほしい。なお、GRAPH2.FNC内のうち、GRAPH.FNCと重複する関数は実画面512ピクセル、1024ピクセル両方に対応しているが、そのほかの関数は実画面512ピクセルモード専用となっている。

●VOL1, VOL2

VOL1ディレクトリには《入門編》、VOL2ディレクトリには《グラフィックス編》の全ソースファイルが各章ごとに収められている。ただし、インクルードファイル類はINCLUDEディレクトリのほうに収録した。同名サブルーチンの複数の版は、ファイル名の末尾に“_1”、“_2”のような番号を付けて区別しているが、長さの制限でファイル名が短く詰められている場合もある。

●SUPPLEMENT

SUPPLEMENTディレクトリには、BINディレクトリに収めた実行ファイル、BASICディレクトリのGRAPH2.FNC、および、描画サブルーチンの追加分の各ソースが収録されている。また、『Oh! X』連載のうち本書では割愛した1991年2、3月号分のプログラム（ソーティングサブルーチン）とC言語風のメモリ管理サブルーチン群をあわせて収録している。

いまさらこんなことを言っではいけないのだろうが、正直なところ、グラフィックスは僕の得意な分野ではない。少なくとも専門に勉強したことはないし、それ自体を目的にしてプログラムを作ったこともほとんどない。ただ、好きなことは好きらしい。X68000を手に入れて最初にやったのは、ポリゴンで3D物体を表示するプログラムをX-BASICで作ることだった。読者が想像するとおり、そのプログラムは眩暈がするほど遅くてすぐに捨てられたっけ。

グラフィックスの分野は、アルゴリズムが洗練されている、実行結果を目で見て楽しめる、という2点で、とくにパソコン上のプログラミングの教材としては最適といえるだろう。結局はメモリに、ある規則でデータを格納したり、メモリ上のデータに算術的・論理的な演算をくわえたりするだけのことなのだが、結果が見えるというだけで口あたりが違う。だからこそ、それを隠れ蓑にして体よく“データ転送と演算”かなんかを裏テーマにした、本書のような本が生まれる余地もあるわけだが。

末筆ながら、本書を著すにあたっては、多くの方々の直接・間接的な支援をいただいた。この場を借りて、感謝の意を表したい。

Chapter3で紹介したダブルステップBresenhamアルゴリズムは『Oh! X』の模範的な読者にして投稿者である古籬一浩氏に『Oh! X』編集部を通じて教えていただいた。また、なかなか線分を短い線分と1ピクセルに分割して考えるというアイデアは氏のものである。

本文中でも触れたが、Chapter7での誤差分散アルゴリズム部分における誤差の散らし方は栗野雅彦氏が試行錯誤のうえ考案した方法である。くわえて、氏の記事には昔から多大な影響を受けており、いまの僕があるのも栗野氏に負うところが大きい。よいしょする気はないが、パソコン少年あがりとおじさん技術者しかいないこの業界にあって、氏は数少ない本物だ。

『Oh! X』でグラフィックス関係の特集を一手に引き受ける丹明彦氏の記事もなにかと参考にさせていただいた。とくに、何か所かで“処理を簡略化させるか、泥臭くても真正直にやるか”の決断に迫られた際には氏の記事を指針とした。

また、本書のプログラム作成には、アセンブラHAS.X (中村祐一氏作) とリンカHLK.X (SALT氏作) を使用した。もし、読者が不幸にしてAS.X Ver.2.0やLKX Ver.2.0を使

用しているのであれば、そんなもんはさっさと捨ててこれらの優秀なツールを入手すべく手を
尽くすべきである。

何かと迷惑をかけている『Oh! X』ならびに書籍各編集部の編集の方々に対しても（半分が
ところは仕事のうちだと諦めてもらうとしてもそれなりに）申し訳なく思っていることを伝え
ておきたい。

もちろん、読者に対しては（陳腐な言い回しだが）言葉では言い表せないほど深く感謝して
いる。今後とも、僕が怠けたり道を踏み外したりしないよう、叱咤激励のほどをお願いする。

1991年12月

村田 敏幸

参考文献

- 1) 『実践コンピュータグラフィックス～基礎手続と応用』, David F. Rogers, 山口富士男監修, セ
イコー電子工業(株)電子機器事業部訳, 日刊工業新聞社
 - 2) 『M68000マイクロプロセッサ・ユーザーズ・マニュアル』, CQ出版社
 - 3) 『プログラマーズマニュアル』(C Compiler PRO-68K Ver.2.0同梱), シヤープ株式会社
-

I N D E X ● 欧文

B・C・D

Bresenhamの線分発生アルゴリズム

▶59, 60, 115, 166, 250, 278, 293

Cの関数▶316

DOSコールconctrl▶15

F・G・H

FLOATn.Xのファンクションコール▶102

G-RAM▶32

G-RAMアドレス計算▶34

HSV▶177

I

IOCSコール▶10

— APAGE▶36

— BOX▶29

— CIRCLE▶157

— CRTMOD▶14, 211

— DEFCHR▶28

— FILL▶29, 39

— FNTGET▶28

— G_CLR_ON▶14

— HSVTORGB▶177

— LEDMOD▶12

— LINE▶13, 65, 100

— MS_CURGT▶18

— MS_CURON▶18

— MS_GETDT▶18

— MS_INIT▶17

— MS_OFMT▶19

— MS_ONTM▶19

— ONTIME▶100

— PAINT▶139

— SKEY_MOD▶17

— SYMBOL▶29

— TGUSEMD▶15, 28

— VPAGE▶29

— WINDOW▶43

X・Y

X-BASICの外部関数▶320

YIQ▶191, 212

あ行

色強調▶203
 色の反転▶182
 色の量子化▶188, 205
 円のクリッピング▶159
 円の塗り潰し▶166
 円描画アルゴリズム▶152
 オーダードディザ▶206

か行

解像度の量子化▶194
 回転▶274, 305
 拡大・縮小▶248
 重ねあわせ▶233
 画像合成▶233
 加法混色▶183
 画面モード切り替え▶14
 カラーコード▶172
 キュー▶134, 141
 矩形のクリッピング▶40
 矩形の塗り潰し▶37
 グラデーション▶219
 グラフィックVRAM▶32
 グラフィックパターン▶228
 — の取り込み▶228
 — の描画▶230
 クリッピング
 — (円)▶159
 — (回転)▶289
 — (拡大・縮小)▶255
 — (矩形)▶40
 — (グラフィックパターン)▶230
 — (自由変形)▶303
 — (線分)▶63
 — (多角形)▶114
 グレイスケール▶189
 ゲット▶228
 減法混色▶183

誤差分散(拡散)法▶206, 224
 コピー(矩形領域)▶239

さ行

3 原色▶172
 自己書き換え▶303
 シード▶134
 シードフィルアルゴリズム▶134
 自由変形▶292
 条件つきアセンブル▶35
 スキャンコンバージョン▶110
 スキャンライン▶110
 スキャンラインシードフィルアルゴリズム
 ▶136
 整数→文字列変換▶102
 鮮鋭化▶201
 線分描画アルゴリズム▶56
 線分のクリッピング▶63
 ソーティング▶117

た行

タイリング▶51
 多角形のクリッピング▶114, 122
 多角形の塗り潰し▶110
 ダブルクリック▶18
 ダブルステップBresenhamアルゴリズム
 ▶83
 単純挿入法▶117
 中断時の戻りアドレス設定▶28
 中点分割アルゴリズム▶63
 デイザ法▶205

な行

2 値化▶214
 ノイズ軽減▶197

は行

8 連結▶106
 はめ込み▶233

パレットコード▶173

パレット設定▶173

反転

——（色）▶182

——（左右上下）▶242

微分処理▶201

複写（矩形領域）▶239

プット▶230

——（回転）▶274

——（拡大・縮小）▶251

——（自由変形）▶292

平滑化▶197

ペイント▶134

ベンチマークテスト▶103

ボカシ▶197

補色▶182

ま行

マウス▶16

マッハバンド▶219

メディアンフィルタ▶200

モザイク処理▶194

モノクロ化▶189

や行

4 連結▶106

ら行

ラインスタイル▶104

ラブラシアン像▶202

乱数▶102

ランダムディザ▶206

量子化

——（色）▶188

——（解像度）▶194

リングバッファ▶136, 141

連続複写▶245

X68000マシン語プログラミング グラフィックス編

1992年1月22日 初版第1刷印刷

1994年6月15日 初版第5刷発行

著者……^{むらたとしゆき}村田敏幸

発行者……橋本五郎

発行所……ソフトバンク株式会社 出版事業部

〒103 東京都中央区日本橋浜町3-42-3

販売 03(5642)8101

編集 03(5642)8140

印刷所……株式会社厚德社

組版 株式会社オービット

©T.MURATA

ISBN4-89052-288-3 C0055

落丁、乱丁はお取り替え致します。定価は表紙に表示してあります。