Handy Graphic ユーザーズガイド

for Handy Graphic Version 0.8.0 2019-03-15 荻原剛志

1 Handy Graphic とは

1.1 概要

Handy Graphic とは、簡単な作図を行うための機能を C プログラムから使えるようにまとめたものです。プログラムから使うための機能のまとまりを**ライブラリ**と呼びますが、Handy Graphic は作図を行うためのライブラリですので**グラフィッククライブラリ**と呼ばれます。

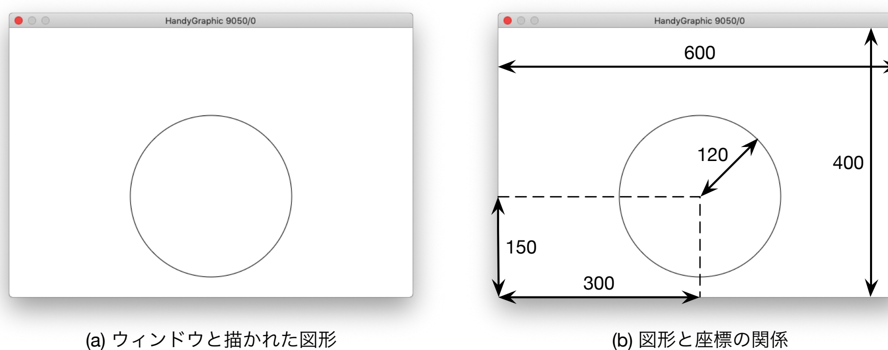
Handy Graphic を使ったプログラムでは、画面上にウィンドウを開いて、自分の思うように直線や円、長方形を描くことができます。単純な機能しか持っていませんが、逆に、覚えることは少なくなっています。これらの機能を組み合わせて幾何学的な模様を描いたり、簡単なグラフを描いたりすることができます。工夫次第ではゲームやパズルなども作成できるでしょう。

このガイドは、Handy Graphic を使うための網羅的な解説になっていますが、ウィンドウを 1 つ表示してシンプルな描画を行う程度のプログラムを作成するためには第 4 章まで読めば十分です。なお、この PDF ファイルには目次のデータが付いています。プレビューなどで参照する際に活用して下さい。

```
#include <stdio.h>
#include <handy.h>                /* Handy Graphic を使うために必要 */

int main(void)
{
    HgOpen(600, 400);             /* 描画用ウィンドウを開く */
    HgCircle(300, 150, 120);      /* 中心 (300, 150)、半径 120 の円を描く */
    getchar();                    /* プログラムを待たせる */
    HgClose();                    /* ウィンドウを閉じる */
    return 0;
}
```

図 1: 円を描くプログラム (prog01.c)



(a) ウィンドウと描かれた図形　(b) 図形と座標の関係

図 2: 円を描くプログラムの実行結果

1.2 Handy Graphic を使ったプログラム例

図 1 は Handy Graphic を使って円を描くプログラムです。

このプログラムはまず、画面に横 600 画素、縦 400 画素の大きさの表示領域を持つウィンドウを開きます。この表示領域の左下を座標の原点と考え、$(300, 150)$ の点を中心とした半径 120 の円を描きます。それだけではプログラムがすぐに終了してしまうため、getchar() を呼び出してターミナルからの入力を待ちます。改行を入力するとプログラムは終了します。描画した結果は図 2 のようになります。

プログラムで図形を描けますので、繰り返したり大きさを変化させたりもできます。図 3 のプログラムは図 4 のように、いくつもの長方形を描きます。HgBox(x, y, w, h) は、左下が (x, y), 幅 w, 高さ h の長方形を描く関数です。

```
#include <stdio.h>
#include <handy.h>

int main(void)
{
    int i;
    double w, h;

    HgOpen(500, 400);
    w = 100;
    h = 40;
    for (i = 1; i <= 15; i++) {
        w -= 4;
        h += 4;
        HgBox(i * 28, i * 20, w, h);
    }
    getchar();
    HgClose();
    return 0;
}
```

図 3: 長方形を描くプログラム (prog02.c)

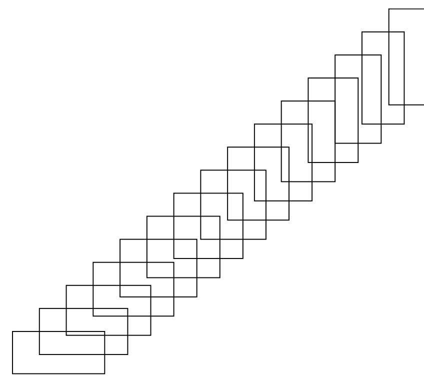


図 4: 長方形を描くプログラムの実行結果

2 プログラムの実行

2.1 Handy Graphic の準備

macOS で Handy Graphic を使ったプログラムで図形を表示するためには、HgDisplayer というアプリケーションを使用する必要があります。Handy Graphic および HgDisplayer のインストールは、パッケージファイルをダブルクリックしてインストーラを起動するだけで行えます。

Handy Graphic をインストールすると、次のものが用意されます。

- HgDisplayer.app:　標準では /Applications に格納されます。
- handy.h:　C 言語のヘッダファイルです。/usr/local/include に作成されます。
- libhg.a:　ライブラリファイルです。/usr/local/lib に作成されます。
- hgcc:　Handy Graphic を使ったプログラムを簡単にコンパイルするためのコマンドです。通常、/usr/local/bin に作成されます。

2.2 コンパイルと実行の方法

図 1 や図 3 のようなプログラムをテキストエディタで作成します。ここでは draw.c という名前のファイルだとして説明します。

Handy Graphic がインストールされていれば hgcc というコマンドでコンパイルできます。動作しない場合、環境変数 PATH の設定を見直す必要があるかもしれません（設定方法はこの文書では触れません）。

次のコマンドでは実行ファイルとして a.out が生成されます（以下の説明で冒頭の$ はターミナルのプロンプトです）。

```
$ hgcc draw.c
```

cc（あるいは gcc）と同様なオプションが指定できます。次の例は、大部分の警告メッセージを表示させ、実行形式のファイル名を draw とするように指定しています。

```
$ hgcc -Wall -o draw draw.c
```

プログラムの実行は通常と同じです。すなわち、実行ファイル名が draw ならばターミナルから次のコマンドを入力します。

```
$ ./draw
```

このプログラムを実行する前に、HgDisplayer を起動しておいた方がよいでしょう。以前に HgDisplayer を使ったことがあれば、ターミナルからプログラムを実行すると自動的に HgDisplayer が起動します。

HgDisplayer は自作のプログラムを終了させるたびに終了させる必要はなく、いったん起動したらそのままにしておいて構いません。ただし、ターミナルでプログラムを動かしたまま HgDisplayer を終了させるとエラーになります。

HgDisplayer は、Handy Graphic を使った複数のプログラムの表示を同時に行うことができます。

2.3 実行の中断

図形の描画をずっと繰り返して終わらない（無限ループ）プログラムを作ることもできます。あるいは、プログラムの誤りによって実行が終了しないことがあるかもしれません。

そのような場合、ターミナルを表示させておいて Control-C （Control キーを押しながら C をタイプ）を入力すると、プログラムを停止させることができます。

もし、Control-C でも終了できない時には、描画ウィンドウの左上にあるクローズボタン（赤いボタン）を押して下さい。

稀に、それでも終了できない場合があります。その場合には、HgDisplayer を終了させます。さらに、ターミナルのプログラムが終了しない場合は、ターミナルのウィンドウをクローズするか、ターミナル自体をいったん終了させて下さい。

2.4 実行中のエラー

コンパイルし、ターミナルから実行ファイルを動作させた後、何らかのエラーが原因でターミナルにメッセージが表示されることがあります。ただし、関数に間違った値を渡した場合、必ずエラーが発生するわけではありません。また、エラーの直接の原因とは一見異なる現象が起きることもあります。

以下に主なエラーメッセージを示します。なお、 Hg___() という部分にはエラーの発生した関数が示されますが、利用者のプログラムに記述されている関数がそのまま表示されるとは限りません。

"ERROR: Can't connect to HgDispalyer."

HgDisplayer に接続して描画などの処理を行うことができません。実行を継続することはできません。インストールが正しく行われているかを調べる必要があります。

"ERROR: internal error."

Handy Graphic の実行処理系の内部でエラーが発生しました。利用者のプログラムの誤りでないと思われる場合、Handy Graphic の開発者に連絡して下さい。

"Illegal image. Hg___()"

ビットマップ画像の処理でエラーが発生しました。イメージ id ではないもの、すでに解放されたイメージを引数に指定しているかもしれません。

"Illegal param. Hg___()"

関数呼び出しの引数が間違っています。ウィンドウ id を渡すべき箇所で別の値を渡しているなど、間違った値の指定が原因です。

"Not connected. Hg___()"

HgDisplayer に接続していません。ウィンドウをオープンせずに描画などの処理を行おうとしているか、Handy Graphic の実行処理系が正しく動作していない可能性があります。

"Terminated: 15"

描画ウィンドウのクローズボタンを押してプログラムを終了した時、このメッセージが表示されます。

"Too long strings. Hg___()"

指定した文字列が長すぎます。

"Interrupted"

ターミナルから Control-C をタイプして実行を中断しました。

3 描画のための設定

3.1 ウィンドウの作成

```
int HgOpen(double w, double h)
```

引数　　w,h:　幅と高さ

返り値　0:　正常、　-1:　異常

描画を行うウィンドウを作成し、画面の中央に表示します。引数 w, h は、ウィンドウ内の描画可能な領域の幅と高さを指定します。

ウィンドウは、自作のプログラムが終了すると自動的にクローズされます。従って、描画結果をしばらく表示させておきたい場合には図 1 や図 3 のプログラムのように、getchar() 関数でターミナルからの入力を待つなどの方法をとる必要があります。

Handy Graphic の関数には、正常に機能した場合に 0、異常が起きた場合に-1 を返すものが多くあります。返り値のチェックでは、数値を直接指定するよりも表 1 に示すマクロを利用した方が分かりやすいでしょう。ただし、関数によって返り値の意味は異なりますので、関数の説明をよく読んで下さい。

なお、ファイルを扱う関数などを除き、ほとんどの関数はめったに異常値を返しません。従って、これ以降の大部分の関数（特に描画関数と色などの属性の設定）について、返り値が正常かどうかを細かくチェックしなくても問題はありません。

表 1: 関数の返り値のマクロ定義

マクロ名	意味	値
HG_SUCCESS	正常	0
HG_ERROR	異常	-1

3.2 ウィンドウを閉じる

標準ウィンドウを画面上から消去します。

```
int HgClose(void)
```

返り値　0:　正常、　-1:　異常

この関数を使わなくても、プログラムが終了する時、ウィンドウは自動的に閉じられます。ただし、実行を正しく終了させることもプログラムの重要な役割ですので、最後に後始末として記述するようにしましょう。

3.3 線の太さ

何も指定しない場合、図形の線の太さは 1 画素分ですが、これを変更することができます。いったん太さを変更すると、別の指定があるまではその値が使われます。

```
int HgSetWidth(double t)
```

引数　　t:　線の太さ

返り値　0:　正常、　-1:　異常

線の太さの指定は、直線、円、長方形などの図形に共通して有効です。

3.4 図形の色の指定

図形を描くのに使われる色を指定することができます。色は hgcolor 型というデータ型で指定しますが、よく使われる色はマクロで定義されています（表 2）。

描かれる線の色を指定するには次の関数を使います。いったん色を指定すると、別の色を指定するまで同じ色が使われます。一度も指定しない場合は黒で描かれます。

```
int HgSetColor(hgcolor clr)
```

引数　clr:色の指定

返り値　0:　正常、　-1:　異常

下で説明する円や長方形では内部を塗りつぶすことができます。このような図形を**塗りつぶし図形**と呼びますが、塗りつぶしに使う色は線を描くための色とは別に、次の関数を使って指定します。いったん色を指定すると、別の色を指定するまで同じ色が使われます。一度も指定しない場合は白で塗りつぶされます。

```
int HgSetFillColor(hgcolor clr)
```

引数　clr:色の指定

返り値　0:　正常、　-1:　異常

3.5 フォントの指定

文字列を描画する場合にフォント（字体）とその大きさを指定できます。フォント名には表 3 のマクロのいずれかを指定します。いったん指定すると、別の指定があるまで同じ字体と大きさが使われます。

```
int HgSetFont(hgfont font, double size)
```

引数　font:　フォントを指定する定数名　size:　文字サイズ

返り値　0:　正常、　-1:　異常

表 2: 色のマクロ定義

色	マクロ名	色	マクロ名	色	マクロ名	色	マクロ名
白	HG_WHITE	赤	HG_RED	シアン	HG_CYAN	空色	HG_SKYBLUE
黒	HG_BLACK	緑	HG_GREEN	オレンジ	HG_ORANGE	濃赤色	HG_DRED
灰色	HG_GRAY	青	HG_BLUE	ピンク	HG_PINK	濃緑色	HG_DGREEN
淡灰色	HG_LGRAY	黄	HG_YELLOW	マゼンタ	HG_MAGENTA	濃青色	HG_DBLUE
濃灰色	HG_DGRAY	紫	HG_PURPLE	茶	HG_BROWN	透明（→節 9.9）	HG_CLEAR

表 3: フォントのマクロ定義

フォント	細字体	斜体	太字体	太字斜体
Times	HG_T	HG_TI	HG_TB	HG_TBI
Helvetica	HG_H	HG_HI	HG_HB	HG_HBI
Courier	HG_C	HG_CI	HG_CB	HG_CBI
明朝	HG_M		HG_MB	
ゴシック	HG_G		HG_GB	

4 主な描画関数

4.1 直線

関数 HgLine() は座標 (x_0, y_0) と (x_1, y_1) を結ぶ線分を描きます。これらの点はウィンドウの外部の点でも構いません。

```
int HgLine(double x0, double y0, double x1, double y1)
```

引数　x0,y0: 線分の始点　x1,y1: 線分の終点
返り値　0: 正常、 -1: 異常

4.2 円

円周だけを描く関数と、塗りつぶした円を描く関数があります。

```
int HgCircle(double x, double y, double r)
```

引数　x,y: 円の中心　r: 半径
返り値　0: 正常、 -1: 異常

```
int HgCircleFill(double x, double y, double r, int stroke)
```

引数　x,y: 円の中心　r: 半径　stroke: 円周を描くかどうか
返り値　0: 正常、 -1: 異常

HgCircle() は、座標 (x, y) を中心とした半径 r の円を描きます。

HgCircleFill() は、座標 (x, y) を中心とした半径 r の塗りつぶされた円を描きます。塗りつぶしには HgSetFillColor() で指定した色が使われます。引数 `stroke` が 0 の場合は円周を描きません。0 以外の値（例えば 1）の場合、他の線と同じ太さ、同じ色の線で円周を描きます。

中心点はウィンドウの外部の点でも構いません。半径は $r \geq 0.0$ の実数値です。

4.3 長方形

```
int HgBox(double x, double y, double w, double h)
```

引数　x,y: 左下隅の座標　w,h: 幅と高さ
返り値　0: 正常、 -1: 異常

```
int HgBoxFill(double x, double y, double w, double h, int stroke)
```

引数　x,y: 左下隅の座標　w,h: 幅と高さ　stroke: 周囲を描くかどうか
返り値　0: 正常、 -1: 異常

HgBox() は、座標 (x, y) を左下隅とする幅 w、高さ h の長方形を描きます。

HgBoxFill() は、座標 (x, y) を左下隅とする幅 w、高さ h の塗りつぶされた長方形を描きます。塗りつぶしには HgSetFillColor() で指定した色が使われます。引数 `stroke` が 0 の場合は周囲に長方形を描きません。0 以外の値（例えば 1）の場合、他の線図形と同じ太さ、同じ色の線で長方形を描きます。

左下隅の座標はウィンドウの外部の点でも構いません。幅と高さは $w \geq 0.0, h \geq 0.0$ の実数値です。

```
#include <stdio.h>
#include <handy.h>

int main(void)
{
    HgOpen(600, 400);                     /* 描画用ウィンドウを開く        */
    HgSetWidth(8.0);                      /* 線の太さを 8 画素分に設定する */
    HgSetColor(HG_DGRAY);                 /* 線の色を薄い灰色にする        */
    HgSetFillColor(HG_YELLOW);            /* 塗りつぶしの色を黄色にする    */
    HgLine(10, 150, 560, 20);             /* 直線（線分）を描く */
    HgCircleFill(200, 220, 175, 1);       /* 円を塗りつぶし、円周も描く    */
    HgSetFont(HG_M, 28);                  /* フォントを明朝体にする        */
    HgText(240, 5, "降水確率は%d%%です。", 30);
    /* 文字列を描く。"%" を表示するためには "%%" とする（printf と同じ） */
    HgSetFillColor(HG_RED);               /* 塗りつぶしの色を赤にする   */
    HgBoxFill(150, 100, 300, 80, 0);      /* 長方形を塗り、辺は描かない */
    HgSetWidth(2.0);                      /* 線の太さを 2 画素分に設定する */
    HgSetColor(HG_BLACK);                 /* 線の色を黒にする              */
    HgCircle(300, 220, 150);              /* 円を描く（塗りつぶしなし）    */
    HgSetFillColor(HG_WHITE);             /* 塗りつぶしの色を白にする      */
    HgBoxFill(200, 160, 320, 80, 1);      /* 長方形を塗りつぶし、辺も描く */
    HgBox(250, 220, 340, 80);             /* 長方形を描く（塗りつぶしなし）*/
    getchar();                            /* プログラムを待たせる          */
    HgClose();                            /* ウィンドウを閉じる   */
    return 0;
}
```

図 5: 円、長方形、文字列を描くプログラム (prog03.c)

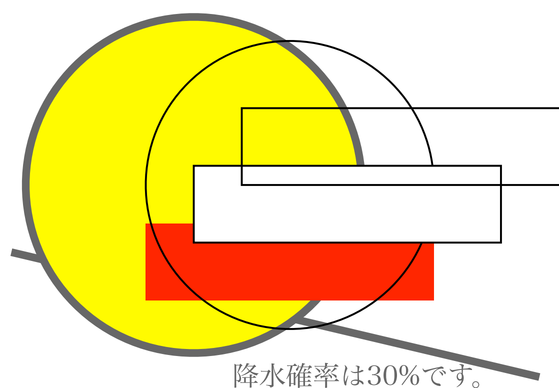


図 6: プログラムの実行結果

◆ 図形を重ね書きする例題

図 5、図 6 は、節 4.4 までで紹介した関数を使った例です（図自体に意味はありません）。

線の太さと色、塗りつぶしの色の指定に注意して、プログラムと実行例を比べてみて下さい。図形は、後から描いたものがそれ以前に描いたものを上書きします。従って、塗りつぶしのない長方形と白で塗りつぶす長方形は結果が異なります。

4.4 文字列

```
int HgText(double x, double y, const char *str, ...)
```

引数　x,y:　左下隅の座標　str:　書式文字列
返り値　0:　正常、　-1:　異常

HgText() は、座標 (x, y) を左下隅として、指定された文字列を描きます。左下隅の座標はウィンドウの外部の点でも構いません。文字列の描画のために、ソースファイルは ASCII コード、または UTF-8（Unicode の符号化のひとつ）で表現されていなければなりません。

文字列 `str` 以降は C の標準関数 printf() と同様に書式を使った記述が可能です（図 5 の例を参照）。

4.5 色を自分で作るには

マクロで定められた色以外に、自分でさまざまな色を詳細に指定することも可能です。hgcolor 型の値の決め方には、グレーの濃度を指定する方法と、赤、緑、青 (RGB) の三原色の濃度で指定する方法があります。

```
hgcolor HgGray(double g)
```

引数　g:　グレーの濃度
返り値　色を表す値

```
hgcolor HgRGB(double r, double g, double b)
```

引数　r:　赤の濃度　g:　緑の濃度　b:　青の濃度
返り値　色を表す値

HgGray() は白から黒までの範囲でグレーの濃度を指定します。引数の `g` は $0.0 \leq g \leq 1.0$ の範囲の実数値で、0.0 が黒、1.0 が白です。

HgRGB() の引数も $0.0 \leq x \leq 1.0$ の範囲の実数値で、`r`、`g`、`b` がそれぞれ赤、緑、青の濃度を表します。例えば、`HgRGB(1.0, 0.0, 0.0)` で赤、`HgRGB(0.8, 0.8, 1.0)` で薄い青になります。

さらに、Web ページの色の表現（色コード）で使われている 16 進数を直接指定することができます。

```
hgcolor HgColorCode(unsigned int code)
```

引数　code:　色コード
返り値　色を表す値

Web サイトの記述でよく利用される色コードは、先頭に "#" を置いた、6 桁（または 3 桁）の 16 進数です。6 桁の色コードは、赤、緑、青の濃度を 2 桁の 16 進数（0 から 255 まで）で表して並べたものです。関数 HgColorCode() は、6 桁からなる色コードを 16 進数の定数として引数に指定できます。

例えば浅葱色を表す [1] 色コード「`#00A3AF`」の色を使うには次のようにします。この引数自体は 16 進数ですので、先頭の 0 はなくても構いません。

```
HgColorCode(0x00A3AF)
```

3 桁からなる色コードは、赤、緑、青の濃度を 1 桁の 16 進数（0 から 15 まで）で表して並べたもので、そのままでは関数 HgColorCode() で利用できません。3 桁からなる色コードは、各桁を 2 回繰り返すことによって同等な色を表す 6 桁の 16 進数に直すことができます。例えばナイルグリーンを「`#3C9`」という 3 桁の表現で表すことがあります。これは、`0x33CC99` という 16 進数に直してから使います。

[1] 色コードと色名の対応に関して、統一された規格は存在しません。

4.6　ウィンドウ内の全消去

```
int HgClear(void)
```

返り値　　0:　正常、　-1:　異常

ウィンドウ内に描かれたすべての図形や文字を消去します。ウィンドウはクローズしません。

なお、関数 HgClear()、HgWClear() は、そのウィンドウのすべてのレイヤの内容も一度に消去します。レイヤ毎の消去には関数 HgLClear() を使います。詳細は節 9.3 を参照して下さい。

5 少し高度な機能と描画関数

5.1 カレントポイントを使って線分を描く

関数 HgLine() では、線分の両端の座標を指定してその間を結んでいました。このような描き方のほかに、**カレントポイント**と呼ばれる点から指定した座標までを結ぶ描き方があります。

カレントポイントとは、紙の上に置いたペン先の位置と考えればよいでしょう。ペンを紙に置いたまま別の点まで動かせば、その間に直線を描くことができます。また、ペン先を紙から放して位置だけ移動させることもできます。この場合、線は描かれません。

カレントポイントだけを設定する（図形は描かない）ためには次の関数を使います。

`int HgMoveTo(double x, double y)`
引数　x,y:　新しいカレントポイントの座標
返り値　0:　正常、　-1:　異常

次の関数は、カレントポイントから指定した点までの線分を描き、その点を新しいカレントポイントにします。従って、この関数を繰り返し呼ぶと指定した点を次々に結ぶ折れ線を描くことができます。

`int HgLineTo(double x, double y)`
引数　x,y:　線分の終点
返り値　0:　正常、　-1:　異常

なお、2 点を指定して直線を描く関数 HgLine() で描画すると、2 つめの点 (x_1, y_1) が新たなカレントポイントになります。

5.2 折れ線と多角形

複数の点の間を結ぶ折れ線を一度に描画することができます。複数の点の座標は配列に用意しておきます。

`int HgLines(int n, const double *xp, const double *yp)`
引数　n:　頂点数　xp,yp:　頂点の x 座標、y 座標を格納した配列へのポインタ
返り値　0:　正常、　-1:　異常

n は折れ線で結ぶ点の個数です。これらの点の座標値は、x 軸と y 軸の値に分かれて、ポインタ xp と yp で参照される配列に入れられているものとします。点の座標はウィンドウの外部でも構いません。最後の座標値が新しいカレントポイントになります。

関数 HgPolygon() では、複数の点を同様な方法で指定し、多角形を描くことができます。折れ線と異なり、最初の点と最後の点の間が結ばれます。

`int HgPolygon(int n, const double *xp, const double *yp)`
引数　n:　頂点数　xp,yp:　頂点の x 座標、y 座標を格納した配列へのポインタ
返り値　0:　正常、　-1:　異常

`int HgPolygonFill(int n, const double *xp, const double *yp, int stroke)`
引数　n:　頂点数　xp,yp:　頂点の x 座標、y 座標を格納した配列へのポインタ
　stroke:　周囲を描くかどうか
返り値　0:　正常、　-1:　異常

関数 HgPolygon() は線を描くだけですが、関数 HgPolygonFill() は描かれた図形を塗りつぶします。引数 stroke が 0 以外の値の場合、周囲の線も描きます。

5.3 円弧と扇型

```
int HgArc(double x, double y, double r, double a0, double a1)
```
引数　x,y: 円の中心　r: 半径　a0: 始点角度　a1: 終点角度
返り値　0: 正常、 -1: 異常

HgArc() は、座標 (x, y) を中心とした半径 r の円について、開始角度 $a0$ と終了角度 $a1$ を指定して円弧を描きます。円弧は始点角度から終点角度までを反時計回りに結びます。中心座標はウィンドウの外部でも構いません。

ここでいう角度とは、弧の端点と中心を結ぶ直線が x 軸となす角度で、弧度法で指定します。従って、例えば 90°ではなく　$\pi/2$ を使います。これは、C 言語の標準の算術関数が弧度法を用いているのに合わせています。

同様な指定で、扇形を描くことができます。

```
int HgFan(double x, double y, double r, double a0, double a1)
```
引数　x,y: 円の中心　r: 半径　a0,a1: 始点、終点角度
返り値　0: 正常、 -1: 異常

```
int HgFanFill(double x, double y, double r, double a0, double a1, int stroke)
```
引数　x,y: 円の中心　r: 半径　a0,a1: 始点、終点角度
　stroke: 周囲を描くかどうか
返り値　0: 正常、 -1: 異常

扇型の弧は円弧と同様、始点角度から終点角度までを反時計回りに結び、円の中心との間に線分を描きます。

HgFanFill() は塗りつぶされた扇型を描きます。引数 `stroke` が 0 以外の値の場合、周囲の線も描きます。

5.4 楕円と長方形

楕円を描くことができます。また、同様な呼び出し方で長方形 (rectangle) の描画ができる関数も用意されています。

```
int HgOval(double x, double y, double r1, double r2, double a)
```
引数　x,y: 楕円の中心　r1, r2: 半径　a: 傾きの角度
返り値　0: 正常、 -1: 異常

```
int HgOvalFill(double x, double y, double r1, double r2, double a, int stroke)
```
引数　x,y: 楕円の中心　r1, r2: 半径　a: 傾きの角度
　stroke: 周囲を描くかどうか
返り値　0: 正常、 -1: 異常

ここで、半径とは楕円の半長径、または半短径のことで、傾きが 0 の場合、r1 が x 軸方向、r2 が y 軸方向の半径を示します。さらに、楕円自体を回転させることができ、この角度を a で表します。角度は弧度法で表し、回転させない時は 0 を指定します。

```
int HgRect(double x, double y, double r1, double r2, double a)
```
引数　　x,y:　長方形の中心　r1, r2:　半径　a:　傾きの角度

返り値　0:　正常、　-1:　異常

```
int HgRectFill(double x, double y, double r1, double r2, double a, int stroke)
```
引数　　x,y:　長方形の中心　r1, r2:　半径　a:　傾きの角度
　　　　stroke:　周囲を描くかどうか

返り値　0:　正常、　-1:　異常

すでに、長方形を描くための関数として、HgBox() と HgBoxFill() を説明しました（節 4.3)。上記の関数 HgRect()、HgRectFill() を使っても、長方形を描くことができます。これらの関数は、楕円と同じ方法で位置と大きさを指定します。つまり、位置は長方形の中心で指定し、大きさは高さと幅の半分の長さで指定します。HgBox() と異なるのは、傾きを指定できる点です。

図 7、図 8 に、楕円、長方形、円弧を描く例を示します。なお、M_PI は算術関数を宣言するヘッダファイル math.h で定義されているマクロで、円周率 π を表します。

```
#include <stdio.h>
#include <math.h>
#include <handy.h>

int main(void)
{
    HgOpen(400, 300);
    HgSetWidth(2.0);
    HgSetFillColor(HgRGB(1.0, 0.7, 0.9));       /* 薄紫色を指定 */
    HgOval(100, 150, 80, 40, 0);                /* 楕円と長方形を描画 */
    HgRect(100, 150, 80, 40, 0);
    HgOvalFill(300, 240, 80, 40, M_PI/6.0, 1);  /* 楕円を傾けて描画 */
    HgRectFill(300, 80, 80, 40, M_PI/6.0, 0);   /* 長方形を傾けて描画 */
    HgSetColor(HgGray(0.8));                    /* 薄い灰色を指定 */
    HgSetWidth(12.0);
    HgArc(100, 150, 100, M_PI*0.2, M_PI*0.8);   /* 円弧を描画 */
    getchar();
    HgClose();
    return 0;
}
```

図 7: 楕円、長方形、円弧を描くプログラム (prog04.c)

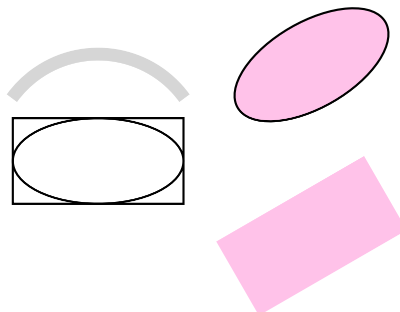


図 8: プログラムの実行結果

5.5　半透明色を作る

描画に使う色については、マクロで指定する方法（節 3.4）、グレーの濃度、RGB の強さを指定する方法（節 4.5）を説明しました。これらの色は不透明で、図形を描くとその位置に描かれていた内容は見えなくなります。

これに対して、色の透明度を指定することによって、色セロファンのような透明度の高い色から不透明に近い色までを作ることができます。

`hgcolor HgGrayA(double g, double a)`

引数　　g:　グレーの濃度　　a:　不透明度

返り値　色を表す値

`hgcolor HgRGBA(double r, double g, double b, double a)`

引数　　r:　赤の濃度　　g:　緑の濃度　　b:　青の濃度　　a:　不透明度

返り値　色を表す値

引数 a は透明度を示す値で「アルファ値」と呼ばれます。$0.0 \leq a \leq 1.0$ の範囲の実数値で、0.0 が透明、1.0 が不透明ですので、「不透明度」を表すと理解するとよいでしょう。その他の引数に関しては、関数 HgGray()、HgRGB() と同様です。

全く何の色もない、透明度 100 %の透明色を `HG_CLEAR` というマクロで指定することができます。

透明色、半透明色の使い方については節 9.9「透明色の塗り方」の記述も参照して下さい。

5.6　フォントを名前で指定

標準的に使用できるフォントについては、関数 HgSetFont() を使って指定できることを説明しました（節 3.5）。それ以外のフォントを利用したい場合、フォントの名前を直接指定すれば利用が可能です。

`int HgSetFontByName(const char *fontname, double size)`

引数　　fontname:　フォント名　　size:　文字サイズ

返り値　0:　正常、　-1:　異常（フォント名の誤りなど）

インストールされているフォントを調べるには、フォント管理用のアプリケーションである Font Book.app を使うとよいでしょう。これで表示した場合に「PostScript 名」または「フルネーム」として表示される文字列が、関数 HgSetFontByName() で利用できます（図 16 の例を参照）。

なお、フォントは /Library/Fonts, /System/Library/Fonts, または ~/Library/Fonts というディレクトリにインストールされています。

5.7　描画される文字列の大きさを得る

その時に有効なフォントを使って文字列を描画した時、ウィンドウ上で実際に占める幅と高さを調べることができます。描画自体は行いません。

`int HgTextSize(double *wp, double *hp, const char *str, ...)`

引数　　wp,hp:　幅と高さを書き込むポインタ　　str:　文字列

返り値　0:　正常、　-1:　異常

関数 HgText() と同様に、文字列 `str` には、printf() 関数と同様に書式を指定することができます。書式に対応する引数は `str` の後ろに記述します。

5.8 メッセージパネルを表示

プログラムの実行中に、処理の失敗などを利用者に知らせたり、動作を選択、あるいは確認してもらいたい場合があります。Handy Graphic のプログラムはターミナルから動作させますので、通常の C プログラムと同様にターミナルにメッセージを出すこともできますが、HgDisplayer とターミナルを切り替えるのが少々面倒です。

次の関数を使うと、メッセージパネルを表示させ、利用者にボタンを押してもらうまで実行を停止させることができます。

```
int HgAlert(const char *str,
            const char *btn0, const char *btn1, const char *btn2, ...)
```

引数　str: メッセージ　btn0, btn1, btn2: ボタンに表示する文字列

返り値　0, 1, 2: ボタン 0、1、または 2 が押された、　-1: 異常

btn0 はパネル上で最も右に表示されるボタンで、利用者が選択すると想定される（あるいは選択して欲しい）選択肢をあてるのが普通です。

```
#include <stdio.h>
#include <handy.h>

int main(void)
{
    int r = HgAlert("%s が現れました！",
       "戦う", "逃げる", "ググる",
       "なんか良くわからないモンスター");
    printf("%d\n", r);
    return 0;
}
```

図 9: メッセージパネルを表示する (alert.c)

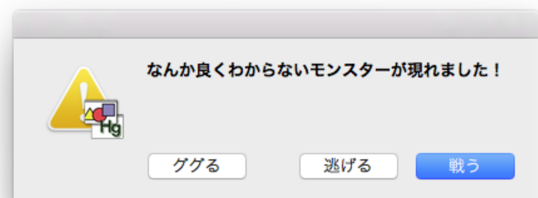


図 10: 表示されるパネル

btn1 は、btn0 のほかの選択肢がある場合に指定します。ない場合は NULL を指定します。さらに、btn2 はもうひとつ別の選択肢がある場合に指定します。なければ NULL を指定します。btn0、btn1、btn2 のすべてに NULL を指定することもでき、その場合、「OK」ボタンが 1 つだけ表示されます。

メッセージを表す文字列 str には、printf() 関数と同様に書式を指定することができます。書式に対応する引数は btn2 の後ろに記述します。

関数の返り値は、パネル上で押したボタンを示す整数値です。

図 9、図 10 に、メッセージパネルを使う例を示します。

5.9 指定時間だけ待つ

短時間だけ描画を停止させる関数を用意しています。ゆっくり描かせたい場合などに、描画関数の間にはさむとよいでしょう。

単位は秒で、例えば引数に 0.5 を指定すると 0.5 秒だけ実行を一時停止します。ただし、時間の厳密な正確さは保障されません。

```
void HgSleep(double sec)
```

引数　sec: 時間間隔（秒）

6　複数のウィンドウを使うプログラム

6.1　複数のウィンドウを作成するには

関数 HgOpen() で作成したウィンドウは**標準ウィンドウ**と呼びます。標準ウィンドウは 1 つしかありません。

Handy Graphic では複数のウィンドウを同時に表示し、それぞれに異なる描画を行うことができます。複数のウィンドウを作成するには次の関数を使います。この関数を呼び出すたびに、新しいウィンドウが作成されます。

```
int HgWOpen(double x, double y, double w, double h)
```

引数　x,y:　左下隅の座標　w,h:　幅と高さ
返り値　0 あるいは正整数:　ウィンドウ id、　-1:　異常

この関数の返す値は**ウィンドウ id** と呼ばれる整数で、ウィンドウごとに異なります。この値を使って、どのウィンドウに描画するかを指定します。標準ウィンドウのウィンドウ id は常に 0 と定められています。なお、ウィンドウが何も表示されていない時に関数 HgWOpen() で新しいウィンドウを作成すると、ウィンドウ id は 0、つまり標準ウィンドウになります。

スクリーンの左下を原点とする座標を**スクリーン座標**と呼びます。一方、個々のウィンドウはそれぞれの左下を原点とする座標を持ち、これに基づいて描画を行います。この座標を**ウィンドウ座標**と呼びます。

関数 HgWOpen() ではウィンドウの表示される位置を指定しなければなりませんが、画面の大きさは使うコンピュータによっては違うこともあります。そこで、プログラムを実行しているコンピュータの画面の大きさを調べる関数が用意されています。

```
void HgScreenSize(double *width, double *height)
```

引数　width, height:　ディスプレイ装置の幅と高さを格納するためのポインタ

ディスプレイ装置の幅と高さを、ポインタ width と height が指す変数へ書き込みます。長さの単位は画素です。

6.2　指定したウィンドウに描画するには

これまでに示した描画用の関数はすべて標準ウィンドウを対象とするものでした。HgWOpen() で作成したウィンドウに描画を行うには、ウィンドウ id を引数とする描画用関数を使います。

例えば、標準ウィンドウに線分を描く関数 HgLine() に対し、指定したウィンドウに描画を行う関数は HgWLine() という名前で定義されています。関数 HgWLine() は第 1 引数にウィンドウ id を指定します。両者を下に示します。

```
int HgLine(double x0, double y0, double x1, double y1)

int HgWLine(int wid, double x0, double y0, double x1, double y1)
```

この例のように、標準ウィンドウを対象とした関数が Hg… という名前なのに対し、ウィンドウを指定できる関数は HgW… という名前で、第 1 引数にウィンドウ id を指定します。

この文書の最後に付録として関数一覧を掲載しています。ここには、標準ウィンドウを対象とする関数と、ウィンドウ id を指定する関数が掲載されています。

ウィンドウ id として 0 を指定すると標準ウィンドウを描画の対象にできます。また、図形の色、線の太さ、フォント、および座標の指定はウィンドウごとに個別の設定が保たれます。

6.3 ウィンドウを閉じる

標準ウィンドウを画面上から消去するには、節 3.2 で説明したように、関数 HgClose() を使います。

ウィンドウ id を指定してウィンドウを個別に閉じるには、次の関数 HgWClose() を使います。さらに、関数 HgCloseAll() は、表示されているウィンドウがあればそれらをすべて閉じます。

```
int HgWClose(int wid)
引数      wid:  ウィンドウ id
返り値    0:  正常、  -1:  異常

void HgCloseAll(void)
```

ウィンドウは実行中にいくつも自由に開いたり、閉じたりできます。プログラムの終了時にはウィンドウは自動的に閉じます。

6.4 ウィンドウの大きさを調べる

ウィンドウの描画部分の大きさを調べることができます。関数 HgGetSize() は標準ウィンドウの大きさを、関数 HgWGetSize() は指定したウィンドウ id を持つウィンドウの大きさを調べます。

```
void HgGetSize(double *width, double *height)
引数      width, height:   ウィンドウの幅と高さを格納するためのポインタ

void HgWGetSize(int wid, double *width, double *height)
引数      wid:   ウィンドウ id
        width, height:   ウィンドウの幅と高さを格納するためのポインタ
```

ウィンドウの幅と高さを、ポインタ width と height が指す変数へ書き込みます。長さの単位は画素です。

6.5 ウィンドウにタイトルを付ける

何も指定しない場合、HgDisplayer は実行しているプログラムのプロセス番号をウィンドウの上部に表示します。この部分に、タイトルとして任意の文字列を表示させることができます。関数 HgSetTitle() は標準ウィンドウ用、関数 HgWSetTitle() はウィンドウを指定して利用します。

文字列 `str` 以降は C の標準関数 printf() と同様に書式を使った記述が可能です。

```
int HgSetTitle(const char *str, ...)
引数      str:   書式文字列
返り値    0:  正常、  -1:  異常

int HgWSetTitle(int wid, const char *str, ...)
引数      wid:  ウィンドウ id    str:  書式文字列
返り値    0:  正常、  -1:  異常
```

7 アプリケーション座標

7.1 座標原点の移動と縮尺の指定

これまでの例題のプログラムでは、ウィンドウの左下を座標原点 (0, 0) として、画素（ピクセル）単位で図形の位置を決めていました（ウィンドウ座標)。ただし、描画の目的によっては原点が左下ではない別の位置にあったり、画素単位ではない縮尺を使ったりしたい場合があります。

次の関数を使うと、座標の原点と縮尺を自由に設定でき、新しく定められた座標を**アプリケーション座標**と呼びます。何も設定していない場合、アプリケーション座標とウィンドウ座標は一致します。

`int HgCoordinate(double sx, double sy, double scale)`
引数　`sx, sy:` 原点の指定　`scale:` 縮尺
返り値　`0:` 正常、　`-1:` 異常

`int HgWCoordinate(int wid, double sx, double sy, double scale)`
引数　`wid:` ウィンドウ id　`sx, sy:` 原点の指定　`scale:` 縮尺
返り値　`0:` 正常、　`-1:` 異常

`sx, sy` は新しい座標で原点とする位置を、ウィンドウ座標で指定します。つまり、ウィンドウ左下を原点として画素単位で位置を表します。新しい原点はウィンドウ上に表示されていない点でもかまいません。

引数 `scale` は縮尺で、正の実数値を指定します。2.0 は 2 倍の拡大で 2 画素が長さの 1 単位に相当し、逆に 0.5 は 1/2 の縮小で 2 単位が 1 画素に相当します。地図の縮尺と同じで、縮尺を 1/10000 にすれば半径が 100000 の円も小さく描画できます。

ただし、縮尺は図形の線の太さ、文字の大きさ、ビットマップ画像の大きさには影響しません。

この関数を用いると、それまでにウィンドウ内に描かれたすべての図形や文字は消されます。ウィンドウにレイヤが追加されている場合、アプリケーション座標はすべてのレイヤで共有されます。

7.2 座標変換

`void HgTransWtoA(double wx, double wy, double *ax, double *ay)`
引数　`wx, wy:` ウィンドウ座標での点の座標
　　　`ax, ay:` アプリケーション座標に変換された値を格納するためのポインタ

`void HgTransAtoW(double ax, double ay, double *wx, double *wy)`
引数　`wx, wy:` アプリケーション座標での点の座標
　　　`ax, ay:` ウィンドウ座標に変換された値を格納するためのポインタ

関数 HgCoordinate() でアプリケーション座標の設定をした場合に、ウィンドウ上の物理的な位置（ウィンドウ座標）とアプリケーション座標の間で、位置の座標変換を行います。

関数 HgTransWtoA() はウィンドウ座標からアプリケーション座標へ変換し、関数 HgTransAtoW() はアプリケーション座標からウィンドウ座標へ変換します。たとえば、マウスクリックのイベントで返される位置情報はウィンドウ座標ですので、必要ならば関数 HgTransWtoA() を使ってアプリケーション座標に変換します。

これらの関数は、下記の関数 HgCoordinateEnable() などでアプリケーション座標が一時的に無効になっている状態でも機能します。

なお、ポインタとして NULL を渡すと、その変数に相当する値は計算しません。例えば関数 HgTransWtoA() を用いて、x 軸の値についてだけ値が必要な場合には、引数 ay には NULL を渡すことができます。

上記の 2 つの関数は標準ウィンドウを対象としていますが、ウィンドウ id でウィンドウを指定して座標変換を行う関数、HgWTransWtoA()、HgWTransAtoW() も用意されています。

7.3 アプリケーション座標の有効／無効

アプリケーション座標を用いて図形を描いている時、ウィンドウの実際の大きさに基づいた操作をしたいことがあります。そのような場合、次の関数を使って、アプリケーション座標の有効／無効を切り替えることができます。

```
void HgCoordinateEnable(int flag)
```

引数　flag: 有効かどうか

```
void HgWCoordinateEnable(int wid, int flag)
```

引数　wid: ウィンドウ id　flag: 有効かどうか

すでにアプリケーション座標が設定されている状態で、関数 HgCoordinateEnable()、または関数 HgWCoordinateEnable() の引数 flag に 0 を指定すると、アプリケーション座標の設定は一時的に無効になり、それ以降の描画はウィンドウ座標に対して行われます。

引数に 0 以外を指定して呼び出すと、無効にされていたアプリケーション座標を再び有効にすることができます。

ただし、アプリケーション座標の有効／無効は座標変換の関数 HgTransWtoA()、HgTransAtoW() には影響しません。

◆ アプリケーション座標でグラフを描画する例題

図 11 は、アプリケーション座標を使って、$-8.0 \leq x \leq 8.0$ 程度の範囲のグラフを拡大して描画するプログラムです。結果を図 12 に示します。グラフは、関数 HgMoveTo()、関数 HgLineTo() を利用し（節 5.1）、短い線分をつなげて描いています。

いったんアプリケーション座標を設定すると、描画関数はその位置と大きさに従って描画を行います。結果から、線の太さや文字の大きさは縮尺に関係しないこと、関数 HgCoordinateEnable() を使って一時的にアプリケーション座標を無効にできることが分かります。縮尺はマクロ Scale で定義していますが、この値を変更してコンパイルしなおすと、グラフの大きさを簡単に変更することができます。

```
#include <stdio.h>
#include <math.h>
#include <handy.h>

#define Width  800.0    /* ウィンドウの幅 */
#define Height 250.0    /* ウィンドウの高さ */
#define Scale  50.0     /* アプリケーション座標の縮尺（50 倍） */
#define Step   0.1      /* グラフを描画する際の刻み幅 */

double func(double x) { /* 描画する関数の定義 */
    return sin(2.0 * x) + cos(x) / 2.0;
}

int main(void)
{
    double x, y, x1, y1, x2, y2;
    HgOpen(Width, Height);
    HgCoordinate(Width / 2, Height / 2, Scale);
      /* ウィンドウ中央を座標原点、縮尺を Scale 倍にする */
    HgTransWtoA(0.0, 0.0, &x1, &y1);
    HgTransWtoA(Width, Height, &x2, &y2); /* ウィンドウの端の座標を得る */
    HgLine(x1, 0.0, x2, 0.0);             /* x 軸を描く */
    HgLine(0.0, y1, 0.0, y2);             /* y 軸を描く */
    HgSetWidth(2.0);
    HgSetColor(HG_BLUE);

    HgMoveTo(x1, sin(x1));         /* sin 関数を描く */
    for (x = x1 + Step; x < x2; x += Step) {
        HgLineTo(x, sin(x));
    }
    HgLineTo(x, sin(x));

    HgSetColor(HG_RED);
    HgSetFont(HG_TI, 20.0);
    HgCoordinateEnable(0);         /* アプリケーション座標を無効にする */
    HgText(50, 10, "y = sin(2x) + cos(x)/2");
    HgCoordinateEnable(1);         /* アプリケーション座標を有効にする */
    HgMoveTo(x1, func(x1));        /* 定義した関数を描く */
    for (x = x1 + Step; x < x2; x += Step) {
        HgLineTo(x, func(x));
    }
    HgLineTo(x, func(x));

    getchar();
    HgClose();
    return 0;
}
```

図 11: アプリケーション座標を使ってグラフを描く (sine.c)

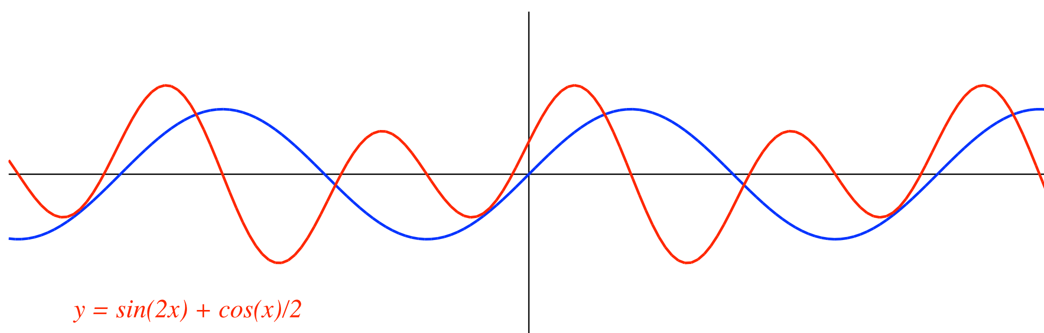


図 12: グラフの描画結果

8 イベントに関する機能

ウィンドウ上をマウスでクリックしたり、キーボードから何かを入力した場合にその情報を取得することができます。このようにプログラム外部からやってくる情報を**イベント**と呼びます。通常の GUI システムは、イベントを受け取ったことをきっかけとしてプログラムが動く仕組み（イベント駆動 (event-driven) と呼びます）になっていますが、Handy Graphic では何かのイベントが来るまで待つという形でイベントを取得します。

なお、短時間に多数のイベントが発生したり、イベント取得や処理に時間がかかったりした場合、発生したイベントのいくつかは自動的に破棄され、すべてを処理しないことがあります。

また、現在の実装では、HgDisplayer で実行中の複数のプログラムが同時にイベントを取得しようとすると、全体の動作が著しく遅くなることがあります。イベントを扱うプログラムは、同時には 1 つだけを実行するようにして下さい。

8.1 キー入力を得る

次の関数を呼び出すことで、標準ウィンドウ、または指定したウィンドウへのキー入力（文字）を得ることができます。

```
int HgGetChar(void)
```
返り値　　0 以上:　入力された文字、　-1:　異常

```
int HgWGetChar(int wid)
```
引数　　wid:　ウィンドウ id

返り値　　0 以上:　入力された文字、　-1:　異常

例えば A のキーを押すと小文字の 'a' の文字コードに相当する 97 が返り、シフトキーを押しながら A を押すと 'A' に相当する 65 が返ります。また、矢印キーを押した場合には、表 4 のマクロで表される整数値が返ります。

表 4: 矢印キーに相当するマクロ名

矢印キー	マクロ名
上向き（↑）	HG_U_ARROW
下向き（↓）	HG_D_ARROW
左向き（←）	HG_L_ARROW
右向き（→）	HG_R_ARROW

ただし、あるウィンドウへのキー入力を待っている間、他のウィンドウに対する入力はすべて捨てられてしまいます。HgGetChar() または HgWGetChar() を使う場合、どれかひとつのウィンドウだけからキー入力を得るようにプログラムする必要があります。

関数 HgGetChar() および HgWGetChar() は、下で述べるイベントの取得のための関数（8.3）を利用して、キー入力だけが簡単に得られるようにしています。複数のウィンドウへのキー入力やマウスのクリックなどのイベントも取得するためには HgSetEventMask() と HgEvent() または HgEventNonBlocking() だけを使い、HgGetChar() または HgWGetChar() はこれらとは同時には使わないようにして下さい。

8.2 イベントマスクの設定

イベントを取得するには、どのようなイベントを扱うのかをウィンドウごとに設定する必要があります。

```
int HgSetEventMask(unsigned int mask)
```

引数　　mask:　イベントマスク
返り値　0:　正常、　-1:　異常

```
int HgWSetEventMask(int wid, unsigned int mask)
```

引数　　wid:　ウィンドウ id　mask:　イベントマスク
返り値　0:　正常、　-1:　異常

イベントマスクは、表 5 のマクロのどれか、あるいはこれらをビット OR で結合したものです。ただし、HG_TIMER_FIRE はマスクの作成には使いません（節 8.5）。

例えばマウスのクリックだけ取得する場合にはイベントマスクとして HG_MOUSE_DOWN を指定し、マウスクリックもキー入力も取得する場合は（HG_MOUSE_DOWN | HG_KEY_DOWN）を指定することになります。

多くの種類のイベントに対応できるように設定することは可能ですが、必要のないイベントに反応していたのではプログラムの処理が重くなります。通常は、マウスクリックとキーボード入力程度で十分です。

なお、表 5 は表 6 と対比させる目的で 16 進数の定義を示していますが、この定義は将来的に変更する可能性もあるため、表の数値をプログラムで直接利用すべきではありません。プログラムではマクロ名を使って下さい。

表 5: イベントを表すマクロ名

イベントの種類	マクロ名	値 (16 進)
マウスボタンが押された	HG_MOUSE_DOWN	0x01
マウスボタンが戻った	HG_MOUSE_UP	0x02
マウスが動いた	HG_MOUSE_MOVE	0x04
マウスがドラッグされた	HG_MOUSE_DRAG	0x08
マウスがウィンドウ内に入った	HG_MOUSE_ENTER	0x10
マウスがウィンドウ外に出た	HG_MOUSE_EXIT	0x20
キーボードが押された	HG_KEY_DOWN	0x40
押されていたキーが戻った	HG_KEY_UP	0x80
タイマが発火した	HG_TIMER_FIRE	0x100

表 6: イベントを扱うためのマスクとマクロ名

説明	マクロ名	値 (16 進)
マウスイベント用マスク	HG_MOUSE_EVENT_MASK	0x3f
キーボードイベント用マスク	HG_KEY_EVENT_MASK	0xc0
ウィンドウイベント用マスク	HG_WINDOW_EVENT_MASK	0xff
（何もイベントがない）	HG_NO_EVENT	0

ウィンドウに対して設定されているイベントマスクの現在の値を得るには、次の関数を使います。

```
unsigned int HgGetEventMask(void)
```
返り値　イベントマスク（0 はイベントの設定がされていないことを表す）

```
unsigned int HgWGetEventMask(int wid)
```
引数　wid: ウィンドウ id
返り値　イベントマスク（0 はイベントの設定がされていないことを表す）

8.3 イベントの取得

イベントを取得するには次の関数を使います。

```
hgevent *HgEvent(void)
```
返り値　NULL: 異常、 NULL 以外: hgevent 構造体へのポインタ

この関数は、HgSetEventMask() で指定したイベントが来るまで待ちます。複数のウィンドウがあっても、すべてこの関数で扱います。ウィンドウがひとつも表示されていないか、どのウィンドウもイベントマスクの設定を行っていないか、あるいは他の何らかのエラーが生じた場合に NULL が返されます。

何かイベントがあると、次のような構造体へのポインタが返されます。

```
typedef struct {
    unsigned long   type;      /* 発生したイベントを表す */
    int             wid;       /* イベントのあったウィンドウの id */
    double          x;         /* (x, y) = マウスイベントが発生した位置 */
    double          y;
    int             count;
    unsigned int    modkey;
    unsigned int    ch;        /* 入力されたキーを示す文字 */
} hgevent;
```

type には、イベントの種類に応じて、表 5 のどれかの値が入れられています。wid はイベントの発生したウィンドウ id を表します。

キーボード上のキーが押された場合、ch にそのキーの文字を表す整数値が入れられます（節 8.1 を参照）。

マウスがクリックされたり、移動したりした場合、(x, y) がその位置を表します。この座標はウィンドウ座標（ウィンドウの左下を原点と考えた時の位置）ですので、アプリケーション座標に変換する必要がある場合は、関数 HgTransWtoA() などを使います（節 7.2）。count には、ダブルクリックなどの場合のクリック回数が入ります。なお、現時点では右クリックやスクロールなどの情報は取得しません。

modkey は、マウスがクリックされた時、またはキーが押された時に同時に押されていた修飾キー（シフト、コントロール、またはオプションキー）の情報が入れられています。情報は表 7 のマクロ名で表される値がビット OR で結合されたものです。

表 7: 修飾キーに相当するマクロ名

修飾キー	マクロ名
シフトキー	HG_SHIFT_KEY
コントロールキー	HG_CONTROL_KEY
オプション（Alt）キー	HG_OPTION_KEY

HgEvent() が返すポインタが指すメモリ領域を解放する必要はありません。ただし、あるイベントによって返された情報は、その後に複数個のイベントが発生すると上書きされることがあります。イベントの情報を保存したい場合は、別の変数を用意して必要な情報をコピーしておいて下さい。

8.4　ブロックしないイベント取得

上で説明した関数 HgEvent() は、指定したイベントが来るまでプログラムを停止させて待ちます。このような動作をブロック待ちと呼びます。

一方、次の関数 HgEventNonBlocking() は、イベントが発生しているかを調べますが、イベントがなければ待つことなく、直ちに呼び出しを終了します。

`hgevent *HgEventNonBlocking(void)`

返り値　`NULL`: イベントがない、　`NULL` 以外: `hgevent` 構造体へのポインタ

イベントが発生していない場合、返り値として NULL が返されます。イベントがあれば、その情報を格納した構造体へのポインタが返されます。

8.5　タイマによるイベント

タイマを設定して、一定の時間後に、または一定の時間間隔でイベントを発生させることができます。タイマにイベントを発生させ始めるには、次の関数で時間間隔を指定します。

`int HgSetIntervalTimer(double t)`

引数　`t`: 時間間隔（秒）

返り値　`0`: 正常、　`-1`: 異常

`int HgSetAlarmTimer(double t)`

引数　`t`: 時間間隔（秒）

返り値　`0`: 正常、　`-1`: 異常

関数 HgSetIntervalTimer() は、指定した時間間隔でイベントを繰り返し発生させます。関数 HgSetAlarmTimer() は、指定した時間の後に 1 回だけイベントを発生させます。

正の値を指定するとタイマが開始し、0.0 以下の値を指定すると、タイマの動作を停止できます。

タイマは、他のイベントの処理をしている間に別の処理を行いたい時に役立ちます。例えば、一定の速度で動作するアニメーションを表示させながら、ユーザのマウスクリックにも対応したいような場合です。あるいは、一定時間内にキー入力がなければ自動的に次の処理を始めるといった動作（タイムアウト）も実現できます。何の処理もせずに指定時間だけ待つのであれば、HgSleep() 関数（節 5.9）を用いた方が簡単でしょう。

タイマによるイベントを取得するためには、関数 HgEvent() または HgEventNonBlocking() を使います。キーボードやマウスからのイベントの取得と同時に使用することができます。タイマは特定のウィンドウとは関係がなく、イベントマスクの指定も必要ありません。これらの関数は hgevent 型の構造体へのポインタを返しますが、タイマによるイベントはメンバ type に `HG_TIMER_FIRE` が設定されています。

イベントの発生間隔は、他のイベントやスレッドの優先順位などに左右されるため、必ずしも正確ではありません。また、発生するイベント数にイベントの処理が追いつかない場合、イベントは自動的に破棄されることがあります。

```
#include <handy.h>

int main(void)
{
    int wid = HgOpen(500, 500);                     /* 標準ウィンドウ         */
    int first = 1;                                  /* 最初のクリックを表す    */
    HgSetWidth(2.0);                                /* マウスクリックと        */
    HgSetEventMask(HG_MOUSE_DOWN | HG_KEY_DOWN);    /*    キー入力を扱う設定   */
    for ( ; ; ) {
        hgevent *event = HgEvent();                 /* イベントを取得 */
        if (event->type == HG_KEY_DOWN) {           /* イベントがキー入力なら */
            switch (event->ch) {
            case 'r': case 'c':
                HgClear(); first = 1; break;        /* 画面をクリア */
            case 'u':
                first = 1; break;                   /* 続けて描くのをやめる    */
            case 'q':
            case 0x1b:
                goto EXIT;    /* q またはエスケープなら終了 */
            }
        }else if (event->type == HG_MOUSE_DOWN) { /* イベントがクリックなら */
            if (first) {                            /* 最初のクリックなら      */
                HgMoveTo(event->x, event->y);       /* カレントポイントを設定 */
                first = 0;
            }else                                   /* 最初でなければクリック */
                HgLineTo(event->x, event->y);       /*      の位置まで線を描く */
        }
    }
EXIT:
    HgClose();
    return 0;
}
```

図 13: マウスクリックとキー入力を使って描画する (plot.c)

◆ イベントを使用する例題

図 13 は、ウィンドウへの操作によって簡単な描画をする例です。プログラムでは、マウスクリックとキーボードの入力のイベントだけを取得するように設定し、関数 HgEvent() を使ってイベントの情報を得ています。

ウィンドウ上のあちこちの位置でクリックを繰り返すと、その間を線分で結ぶことができます。キーボードから r または c を入力すると、画面をクリアします。u を入力すると、続けて描くのをいったんやめます。q またはエスケープを入力するとプログラムを終了できます。

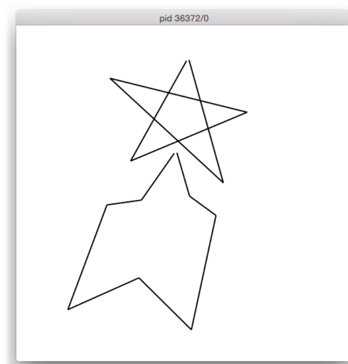


図 14: 描画結果の例

9 レイヤの使い方

9.1 レイヤの概念

Handy Graphic では、1 つのウィンドウに描かれる画像を、**レイヤ**（層）と呼ばれるいくつかの画像を重ね合わせて構成することができます。Handy Graphic のレイヤは、透明なシートのようなものと考えるとよいでしょう。複数のウィンドウにそれぞれ異なる描画ができるように、レイヤにもそれぞれ別の内容を描くことができます。

図 15 にレイヤの概念を示します。ウィンドウにレイヤを追加しない場合、描画はウィンドウに元々割り当てられている描画用の領域に対して行われます。この部分をベースレイヤと呼びます。ベースレイヤは追加や削除ができないこと、背景が白色であることなど、他のレイヤとは異なります。

ウィンドウに新たに複数のレイヤを追加することができ、新しいレイヤほど手前に置かれます。追加したレイヤの背景色は透明色です。これらのレイヤは削除したり、前後を入れ替えたりできます。

レイヤを用いて描画することの利点はいろいろありますが、図 15 の例では、例えば自動車の位置を描画しなおそうとした場合、背景の建物や前景の樹木を描きなおす必要がありません。このように、別々に描画や消去をしたい図形の集合をレイヤにまとめておくと、不必要な再描画を避けることができます。

また、レイヤは後述のようにアニメーションを滑らかに動かすのにも役立ちます。

ただし、レイヤに描画した内容をウィンドウ上で表示するには、各レイヤを重ね合わせる処理が必要になりますので、あまり多くのレイヤを使うと表示速度が遅くなります。

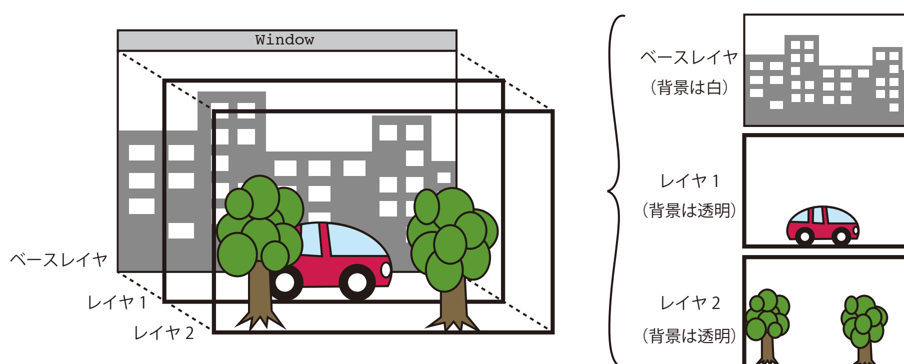


図 15: レイヤの概念図

9.2 レイヤの追加とレイヤ id

関数 HgOpen()、または HgWOpen() を使って新しくウィンドウを開いた時、ウィンドウにはベースレイヤだけが用意されており、ウィンドウ id を指定して描画を行うとベースレイヤに描画が行われます。

新しくレイヤを追加するには、すでに開いているウィンドウを指定し、下の関数 HgWAddLayer() を呼び出します。

```
int HgWAddLayer(int wid)
```

引数　wid:　ウィンドウ id

返り値　正整数:　レイヤ id、　-1:　異常

返される整数値は**レイヤ id** と呼ばれ、描画先を指定するために使用します。描画用関数、色や線の太さ、フォントの設定用関数は、ウィンドウ id の代わりにレイヤ id を指定して使うことができます。

レイヤの性質についてまとめておきます。

- どのレイヤも、大きさはベースレイヤと同じです。
- レイヤの背景色は透明です。一方、ベースレイヤの背景色は白色です。
- レイヤが作成される時、ベースレイヤに対して設定されている線の太さ、描画色、塗りつぶし色、およびフォントの設定がコピーされます。ただしその後で、ベースレイヤおよび各レイヤにはそれぞれ別の設定ができます。
- レイヤに対して描画する場合、あるいは色や線の太さ、フォントの設定を行う場合には、ウィンドウ id の代わりにレイヤ id を指定して関数を呼び出します。
- レイヤは新しく追加されたものが一番手前に配置されますが、順序は入れ替えることができます。
- レイヤは描画内容を表示に反映させない（見せない）ように指定できます。非表示のレイヤにも描画は可能です。表示する／しないという状態は切り替えることができます。
- レイヤ、およびベースレイヤはビットマップ画像として一部、あるいは全体をコピーすることができます（「10 ビットマップ画像」を参照）。
- 表示内容、またはレイヤごとに、ビットマップ画像、または PDF 形式の画像としてファイルに保存することができます（「12 描画結果の保存」を参照）。
- レイヤは個別に削除できます。また、ウィンドウ自体がクローズされる時に、同時にすべてが削除されます。

9.3 レイヤの描画を消去する

レイヤを指定して、そこに描かれた内容を消去するには次の関数を使います。

```
int HgLClear(int lid)
```

引数　lid:　レイヤ id ／ウィンドウ id
返り値　0:　正常、　-1:　異常

関数 HgLClear() は、引数で指定されたレイヤの描画内容を消去します。引数としてウィンドウ id が指定された場合、ベースレイヤの描画を消去します。

ウィンドウの描画を消去する関数として、HgClear()、HgWClear() を紹介しました（節 4.6)。これらの関数はそのウィンドウのすべてのレイヤの内容も一度に消去します。

この関数、および HgClear()、HgWClear() はともに描画領域を背景色で塗りつぶすことで消去を行います。つまり、ベースレイヤは白色で塗りつぶしますが、その他のレイヤは透明色で塗りつぶします。

9.4 表示／非表示を切り替える

```
int HgLShow(int lid, int flag)
```

引数　lid:　レイヤ id　flag:　0 または 1
返り値　0:　正常、　-1:　異常

レイヤ id で指定したレイヤの表示／非表示を変更します。引数にウィンドウ id（ベースレイヤ）を指定することはできません。非表示の状態のレイヤにも描画を行うことができます[2]。

なお、HgWAddLayer() で追加された直後のレイヤは表示状態になっています。

複数のレイヤについて、同時に表示／非表示を変更するためには次の関数を使います。

```
int HgLSetVisible(int lid, int flag, ...)
引数     lid:  レイヤ id   flag:  0 または 1
返り値   0:  正常、  -1:  異常
```

引数には、レイヤ id とそのレイヤに対する指定の組を 1 つ以上指定します。引数の末尾には、引数列の終わりを表すための 0 を必ず置かなければなりません。

例えば、レイヤ space を非表示に、レイヤ buf を表示状態にするには次のようにします。

```
HgLSetVisible(space, 0, buf, 1, 0);
```

9.5 レイヤの順序を変更する

レイヤは新しく追加されたものが一番手前に配置されますが、後から順序を入れ替えることができます。ただし、ベースレイヤは常に一番下で、位置を変更することはできません。

```
int HgLMove(int lid, int pos)
引数     lid:  レイヤ id   pos:  新しい位置
返り値   0:  正常、  -1:  異常
```

ベースレイヤのすぐ上のレイヤを 0 番目として、手前に向かって 1 番目、2 番目、... と数えます。追加されたレイヤが n 枚あれば、最も手前は $n-1$ 番目になります。例えば、追加されたレイヤが 3 枚あるとき、いずれかのレイヤを指定して最も手前になるように移動させるには引数 pos に 2 を指定します。そのレイヤがすでに指定した位置にある場合には何もしません。

その時点で、ウィンドウに何枚のレイヤが追加されているかを知るには次の関数を使います。

```
int HgWLayers(int wid)
引数     lid:  ウィンドウ id
返り値   0 または正整数:  レイヤの数、  -1:  異常
```

9.6 レイヤの内容をコピーする

```
int HgLCopy(int lid, int other)
引数     lid:  コピー先のレイヤ id   other:  コピー元のレイヤ id
返り値   0:  正常、  -1:  異常
```

引数 other が表すレイヤの内容を、引数 lid が表すレイヤにコピーします。このとき、2 つのレイヤは同じウィンドウのレイヤでなければなりません。さらに、どちらもベースレイヤではいけません。

あるレイヤの内容を別のレイヤに描画するには、関数 HgWImagePut()、HgWImageDraw()、HgWImageDrawRect() も使うことができ、拡大／縮小や一部分の描画も指定できます（節 10.5）。一方、この関数 HgLCopy() はレイヤの内容を丸ごとコピーする目的のみに使用します。

[2] このような用途をオフスクリーンバッファなどと呼ぶことがあります。

9.7　レイヤを削除する

```
int HgLRemove(int lid)
```

引数　　lid:　レイヤ id／ウィンドウ id
返り値　0:　正常、　-1:　異常

指定したレイヤを削除します。引数にウィンドウ id を指定すると、ベースレイヤ以外のすべてのレイヤを削除します。
ウィンドウ自体がクローズされる時、そのウィンドウのすべてのレイヤは削除されます。従って、ウィンドウのクローズの前にこの関数を呼び出す必要はありません。

9.8　レイヤを含むウィンドウを知る

```
int HgWindowOfLayer(int lid)
```

引数　　lid:　レイヤ id
返り値　ウィンドウ id

指定したレイヤを含むウィンドウ id を返します。ただし、そのレイヤやウィンドウが利用可能かどうかは分かりません。

9.9　透明色の塗り方

すでに描かれた図形の上に半透明色（節 5.5）で別の図形を描いた場合、通常は下の図形が透けて見えます。これに対し、後から描いた図形を優先し、それ以前に描いた図形が見えなくなるような描き方も用意されています。
透明色を含む図形を重ね合わせて描く方法は、コンポジット・モード、あるいはアルファ・ブレンディングと呼ばれ、次の関数で指定できます。

```
int HgSetComposite(int compo)
```

引数　　compo:　描画方法の指定
返り値　0:　正常、　-1:　異常

```
int HgWSetComposite(int wid, int compo)
```

引数　　wid:　ウィンドウ id／レイヤ id　　compo:　描画方法の指定
返り値　0:　正常、　-1:　異常

引数 compo に指定する値のうち、主なもの 2 つは下記のマクロで指定できます。通常、何も指定をしない場合の設定値（デフォルト値）は HG_BLEND_SOVER です。

HG_BLEND_COPY ：後から描いた図形だけが表示されます。

HG_BLEND_SOVER ：下に描かれていた図形と、後から描いた図形が、色の透明度に応じて混ぜ合わされて描かれます。不透明に近い色ほど、後から描いた図形が強調して描かれます。

ただし、これらの色の重ね合わせに関する指定は、画面上の表示とビットマップ画像に対しては有効ですが、現在の実装では、PDF 形式で保存する場合には機能しません（節 12.3）。

◆ コンポジットモードの効果を見る例題

図 16 は、節 9.9 で説明したコンポジットモードの効果を確認する例です。

このプログラムでは2つのウィンドウを作成し、同じ内容の描画を行いますが、図 17 の左側はレイヤに対して HG_BLEND_SOVER の指定（デフォルト値)、右側は HG_BLEND_COPY の指定を行った結果です。右側では、透明色、半透明色を描いた部分に、先に描画した図形の色が残っていないことが分かります。

描画結果の表示の際、レイヤ同士は HG_BLEND_SOVER のモードで重ね合わせられます。その結果、図の右側で透明色を使って描画した部分からは、下のレイヤが透けて見えるようになります。

```
#include <stdio.h>
#include <handy.h>

void draw(int wid, int lid)
{
    HgWSetFillColor(wid, HG_BLACK);          /* ベースレイヤの下の方を  */
    HgWBoxFill(wid, 0, 0, 400, 50, 0);       /* 帯状に黒く塗る          */
    HgWSetFillColor(lid, HG_ORANGE);         /* 不透明のオレンジ色で    */
    HgWCircleFill(lid, 150, 50, 135, 0);     /* レイヤに円を描く        */
    HgWSetFillColor(lid, HgRGBA(0, 0, 1.0, 0.5));  /* 半透明の青で      */
    HgWBoxFill(lid, 120, 0, 200, 150, 0);    /* 長方形を円に重ねて描く  */
    HgWSetFontByName(lid, "YuGo-Bold", 75.0);      /* 游ゴシック体 */
    HgWSetColor(lid, HG_CLEAR);              /* 100 %の透明色      */
    HgWText(lid, 15, 10, "透明");            /* 透明色で文字を描く */
    HgWSetColor(lid, HgGrayA(0.3, 0.3));     /* 半透明の文字を描く */
    HgWText(lid, 170, 10, "半透明");
}

int main(void)
{
    int wid1 = HgWOpen(200, 300, 400, 200);
    int wid2 = HgWOpen(620, 300, 400, 200);
    int lid1 = HgWAddLayer(wid1);           /* レイヤを追加 */
    int lid2 = HgWAddLayer(wid2);
    HgWSetComposite(lid1, HG_BLEND_SOVER); /* デフォルト値 */
    draw(wid1, lid1);
    HgWSetComposite(lid2, HG_BLEND_COPY);  /* 後からの描画を優先 */
    draw(wid2, lid2);
    (void)getchar();
    HgCloseAll();
    return 0;
}
```

図 16: コンポジットモードの効果を見る (composite.c)

図 17: コンポジットモードの効果

10　ビットマップ画像

10.1　ビットマップ画像の操作

Handy Graphic では、画像ファイルを読み込んで描画に利用できます。扱うことができる主な画像形式は、JPEG、PNG、GIF、TIFF などのビットマップ画像です。特に、PNG 形式は透明色、半透明色を含む画像も扱えるほか、Handy Graphic で描画結果を保存する形式としても使っています（節 12.1）。JPEG は広く利用されていますが、透明色を表現できません。GIF は、特に Web ページで簡易アニメーション機能を利用するためによく使われています。透明色は表現できますが、半透明を表すことはできません。また、Handy Graphic では GIF のアニメーション機能を利用することはできません。

Handy Graphic ではさらに、レイヤ、およびベースレイヤ自体をビットマップ画像として扱うことができます。これにより、いったん描画した内容を複製しておいたり、別な位置に描画しなおしたりすることが容易にできます。

ファイルから読み込んだ画像や、複製して作成した画像は int 型の**イメージ id** で管理します。

10.2　画像ファイルを読み込む

```
int HgImageLoad(const char *path)
```

引数　　path：　画像データを格納したファイルのパス

返り値　　非負整数：　イメージ id　　-1：　異常

ファイルを指定し、画像データを読み込みます。読み込み可能な代表的なデータ形式は、拡張子が jpg、png、gif、tiff のものです。ファイルは、実行時のカレントディレクトリからの相対パス、あるいは絶対パスで指定します。

返り値はイメージ id で、これを用いて描画を行うことができます。

10.3　画像の大きさを調べる

画像の大きさ（縦と横の長さ）を調べることができます。

```
int HgImageSize(int gid, double *width, double *height)
```

引数　　gid：　イメージ id ／レイヤ id

　　　　width，height：　画像の幅と高さを格納するためのポインタ

画像の幅と高さを、ポインタ width と height が指す変数へ書き込みます。長さの単位は画素です。

引数としてレイヤ id を指定することもできます。その場合は関数 HgWGetSize() を使ったのと同様で、ウィンドウの描画部分の大きさが返されます（節 6.4）。

10.4　画像データを複製する

画像を複製 (duplicate) し、新しいイメージ id を返します。

```
int HgImageDup(int gid, double scale, int flip)
```

引数　　gid：　イメージ id ／レイヤ id　　scale：　縮尺　　flip：　反転させるかどうか

返り値　　非負整数：　イメージ id　　-1：　異常

引数にはイメージ id、またはレイヤ id を指定します。複製を作成する際に、拡大／縮尺を指定することができます。1.0 を指定すると同じ大きさです。また、画像を鏡像反転させることができます。反転しない場合は 0（または HG_FLIP_NONE）を指定します。水平方向、垂直方向への反転をするには、表 8 のマクロを指定します。

表 8: 画像の反転を指定するマクロ名

操作	マクロ名
反転しない	HG_FLIP_NONE
水平方向に反転	HG_FLIP_HORIZONTAL
垂直方向に反転	HG_FLIP_VERTICAL

画像全体ではなく、一部分を指定して複製を作成することができます。

```
int HgImageDupRect(int gid, double scale, int flip,
                   double x, double y, double w, double h)
```

引数　gid: イメージ id ／レイヤ id　scale: 縮尺　flip: 反転させるかどうか
　　　x,y,w,h: 複製する領域
返り値　非負整数: イメージ id　-1: 異常

引数のうち x, y, w, h で、複製する長方形領域を指定します。引数 gid がレイヤ id の場合、そのレイヤを含むウィンドウのアプリケーション座標に基づいて位置が決められます。引数 gid がイメージ id の場合は画像データ内でピクセル単位で領域を指定します。

作成した画像データは、不要になったら関数 HgImageUnload() を使って解放します（節 10.6)。

10.5 画像を描く

```
int HgImagePut(double x, double y, int gid, double scale, double a)
```

引数　x,y: 描画する位置
　　　gid: イメージ id ／レイヤ id　scale: 縮尺　a: 傾きの角度
返り値　0: 正常、　-1: 異常

```
int HgWImagePut(int wid, double x, double y, int gid, double scale, double a)
```

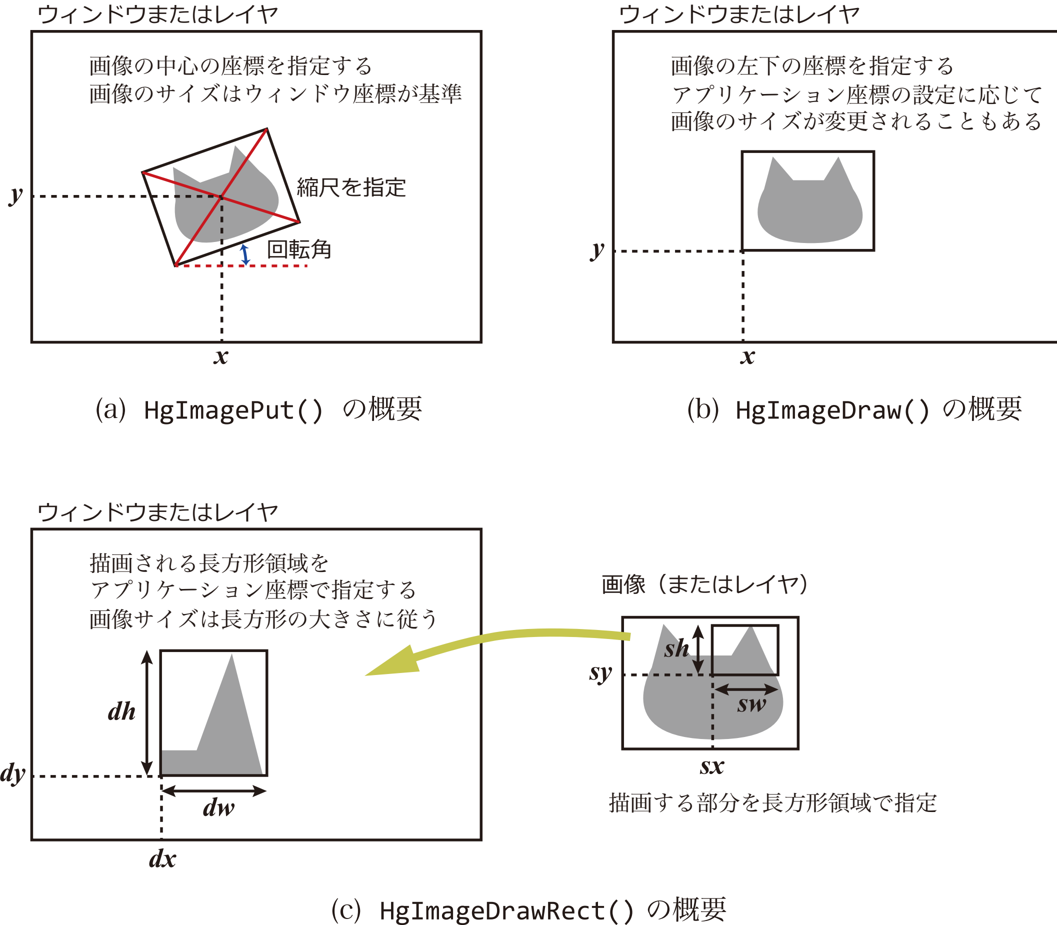
引数　wid: ウィンドウ id ／レイヤ id　x,y: 描画する位置
　　　gid: イメージ id ／レイヤ id　scale: 縮尺　a: 傾きの角度
返り値　0: 正常、　-1: 異常

関数 HgImagePut() および関数 HgWImagePut() は、ウィンドウ（またはレイヤ）の指定された位置に画像を描きます。指定する座標 (x, y) は、画像の中心点になります。

座標 (x, y) はアプリケーション座標で指定しますが、描かれる画像の大きさはアプリケーション座標の縮尺には影響されません。描かれる画像は、縮尺を指定して拡大／縮小が可能です。また、角度（弧度法）を指定して回転させることもできます。この様子を図 18(a) に示します。

```
int HgImageDraw(double x, double y, int gid)
```

引数　x,y: 描画位置　gid: イメージ id ／レイヤ id
返り値　0: 正常、　-1: 異常



(a) `HgImagePut()` の概要　　(b) `HgImageDraw()` の概要

(c) `HgImageDrawRect()` の概要

図 18: ビットマップ画像の描画関数の概要

```
int HgWImageDraw(int wid, double dx, double dy, int gid)
```
引数　wid: ウィンドウ id ／レイヤ id　x,y: 描画位置　gid: イメージ id ／レイヤ id
返り値　0: 正常、 -1: 異常

関数 HgImageDraw() および関数 HgWImageDraw() は、ウィンドウ（またはレイヤ）の指定された位置に画像を描きます。指定する座標 (x, y) は、画像の左下になります。

座標 (x, y) はアプリケーション座標で指定しますが、描かれる画像の大きさはアプリケーション座標の縮尺には影響されません。

この関数では、描画される画像は拡大／縮小したり、回転させることはできません。この様子を図 18(b) に示します。

```
int HgImageDrawRect(double dx, double dy, double dw, double dh,
                    int gid, double sx, double sy, double sw, double sh)
```
引数　dx,dy,dw,dh: 描画先の領域　gid: イメージ id ／レイヤ id
　　　sx,sy,sw,sh: 描画元の領域
返り値　0: 正常、 -1: 異常

```
int HgWImageDrawRect(int wid, double dx, double dy, double dw, double dh,
                     int gid, double sx, double sy, double sw, double sh)
```

<u>引数</u>　wid:　ウィンドウ id／レイヤ id　dx,dy,dw,dh:　描画先の領域
　　　　gid:　イメージ id／レイヤ id　sx,sy,sw,sh:　描画元の領域

<u>返り値</u>　0:　正常、　-1:　異常

関数 HgImageDrawRect() および関数 HgWImageDrawRect() は、ウィンドウ（またはレイヤ）の長方形領域を指定して、その中に画像を描きます。領域の左下の座標と幅、高さを引数 dx, dy, dw, dh で指定します。この領域の指定はアプリケーション座標に基づきます。特別に、引数 dw, dh が 0.0 の場合、長方形領域の大きさは描かれる画像の描画元の領域の大きさに合わせられます。

描かれる画像も、引数 sx, sy, sw, sh で長方形領域を指定します。引数 gid がレイヤ id の場合、そのレイヤを含むウィンドウのアプリケーション座標に基づいて位置が決められます。

引数 gid がイメージ id の場合は画像データ内でピクセル単位で領域を指定します。特別に、引数 sx, sy, sw, sh がすべて 0.0 の場合、画像データのすべての範囲が領域と指定されたとみなします。

描画先の領域と、描画元の領域の大きさが異なる場合、画像は拡大、あるいは縮小されます。縦横の比も変化します。この様子を図 18(c) に示します。

レイヤの内容を丸ごとコピーする目的には、関数 HgLCopy() を利用することもできます（節 9.6）。

10.6　画像データを解放する

```
int HgImageUnload(int gid)
```

<u>引数</u>　gid:　イメージ id

<u>返り値</u>　0:　正常　-1:　異常

ファイルから読み込んだり、関数 HgImageDup()、または HgImageDupRect() を使って作成した画像データをもう使わない場合、この関数を使ってデータをメモリ上から排除できます。引数として指定できるのはイメージ id だけです。レイヤ id は指定できません。

画像データ用に使用されたメモリは、プログラムが終了する時に自動的に解放されます。従って、すべてのデータを完全に解放する必要はありません。ただし、プログラムを実行している間は存在し続けますので、不要になった時点でこまめに解放しないと、メモリ内部が不要なデータだらけになる可能性があります。

◆ 画像データをいくつかの方法で描画する例題

図 19 は、節 10.5 で説明した 3 種類の関数を使って画像データを描画する例です。同じ大きさのウィンドウを 2 つオープンし、一方だけにアプリケーション座標を設定し、同じ関数定義を使って描画を行います。描画結果を図 20 に示します。番号と矢印は、プログラムの該当箇所を実行した結果であることを示しています。

コメント (1) の位置の関数 HgWImagePut() では、画像を 30 °回転して描画します。指定した座標位置は画像の中心になります。一方、コメント (2) の関数 HgWImageDraw() では、指定した座標位置は画像の左下になります。この 2 つの関数では、画像の大きさはアプリケーション座標の影響を受けません。

コメント (3) の位置の関数 HgWImageDrawRect() では、描画元の領域を指定していません。これによって、画像データの全域を指定したと同じことになります。画像は 200 × 100 の長方形の中に描かれます。

コメント (4) の位置の関数 HgWImageDrawRect() では、画像データの一部の領域を指定し、その部分だけを指定された長方形内に描画します。

```
#import <stdio.h>
#import <math.h>
#import <handy.h>

void draw(int wid, int gid)
{
    HgWBox(wid, 0, 100, 300, 100);     /* 座標の目安として長方形を描く */
    HgWBox(wid, 100, 0, 100, 300);
    HgWImagePut(wid, 100, 200, gid, 1.0, M_PI/6.0);  /* (1) */
    HgWImageDraw(wid, 200, 200, gid);                /* (2) */
    HgWImageDrawRect(wid, 0, 0, 200, 100, gid, 0, 0, 0, 0);
        /* (3) 画像を 200 × 100 の長方形の中に描く */
    HgWImageDrawRect(wid, 200, 50, 100, 150, gid, 62, 56, 48, 36);
        /* (4) 画像データの一部分を指定し、100 × 150 の長方形の中に描く */
}

int main(int argc, char **argv)
{
    int wid1, wid2;
    int img = HgImageLoad("cat.png");
    wid1 = HgWOpen(200, 200, 300, 300);
    wid2 = HgWOpen(600, 200, 300, 300);
    HgWCoordinate(wid2, 20, 50, 0.75);  /* アプリケーション座標を設定 */
    draw(wid1, img);                    /* 2つのウィンドウに同じ操作 */
    draw(wid2, img);
    (void)getchar();
    HgCloseAll();
    return 0;
}
```

図 19: 画像データをいくつかの方法で描画する (bitmap.c)

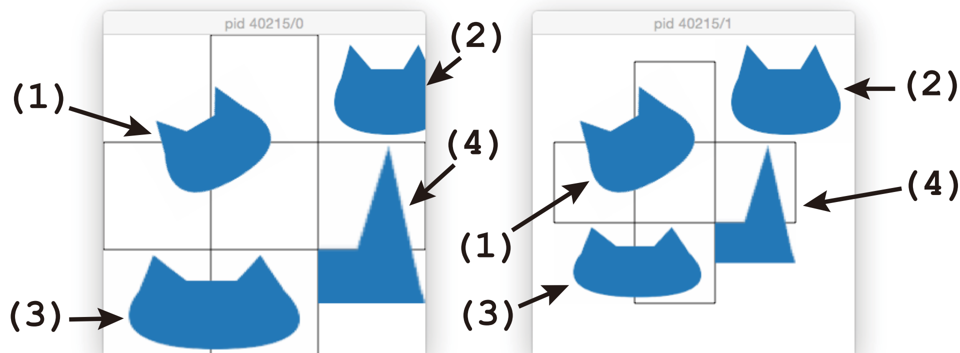


図 20: 画像データを描画した例

11　アニメーション

11.1　シンプルなアニメーション

Handy Graphic の描画機能を使い、図形を少しずつ描いたり消したりすることによって、図形が動いているように見せることができます。つまり、簡単なアニメーションが実現できます。

```
#include <stdio.h>
#include <handy.h>

#define R  30
#define WIDTH   360.0
#define HEIGHT  180.0

int main(void)
{
    double x = R;
    int count;
    HgOpen(WIDTH, HEIGHT);
    for (count = 0; count < 30; count++) {
        HgSetFillColor(HG_WHITE);          /* (1) 塗りつぶし色を白にする */
        HgCircleFill(x, HEIGHT/2.0, R, 0); /* (2) 現在の位置の表示を消す */
        x += R * 2.0;
        if (x > WIDTH) x = R;
        HgSetFillColor(HG_BLUE);
        HgCircleFill(x, HEIGHT/2.0, R-1.0, 0); /* 新しい位置に表示する */
        HgSleep(0.3);              /* (3) 半径がわずかに小さいことに注意 */
    }
    HgClose();
    return 0;
}
```

図 21: 簡単なアニメーション (anime01.c)

画像処理において、描画した物体や図形とその背景を滑らかに見せるため、中間色を混ぜながら境界を描く手法をアンチエイリアスと呼びます。HgDisplayer では、アンチエイリアスを用いて図形や文字の描画を行っていますので、プログラムで指定した図形の大きさよりも、若干外側にまで色の変化が及ぶことがあります。

図 21 では、青い円の上から白い円を描くことで、いったん描いた青い円を消していますが、白い円の方をわずかに大きめにしないと青い輪郭の一部が残されてしまいます（図 21 のコメント (3) に注意）。これはアンチエイリアスによる描画の影響です。

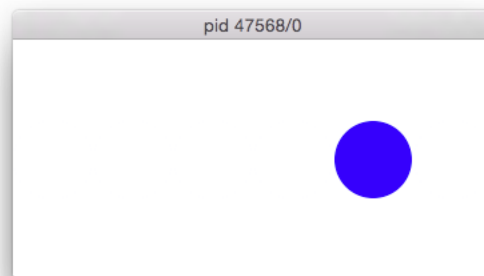


図 22: 簡単なアニメーションの表示例

この例のように、白い色で前の画像を消す方法は、画面上の図形の一部分だけを描きなおす場合などに有効です。別の方法としては、描画をすべて消し、次の場面を最初から描きなおすやり方が考えられます。図 21 の例なら、コメント (1), (2) の行を関数 `HgClear()` で置き換えます。この方法は、画像の一部をわざわざ塗りつぶして消す手間がないために簡単に実現できますが、全体の描きなおしに時間がかかる場合は実用的ではありません。

11.2 レイヤを用いて描きなおしを減らす

表示内容の一部のみを描きなおす場合で、特に、図形が重なり合う場合にはレイヤを利用するとプログラムが簡単になります。図 3 に例を示します。

```
#include <stdio.h>
#include <handy.h>

#define R  30
#define WIDTH   360.0
#define HEIGHT  180.0

int main(void)
{
    double x, v;
    int count;
    int layer1, layer2;
    HgOpen(WIDTH, HEIGHT);
    layer1 = HgWAddLayer(0);          /* アニメーション用レイヤ */
    layer2 = HgWAddLayer(0);          /* 前景用レイヤ */
    HgWSetFillColor(0, HG_DGRAY);     /* 背景をベースレイヤに描く */
    for (v = 20.0; v < WIDTH; v += WIDTH * 0.2)
        HgWBoxFill(0, v, HEIGHT*0.4, R*1.2, HEIGHT*0.6, 0);
    HgWSetFillColor(layer2, HG_ORANGE);      /* 前景を描く */
    for (v = 50.0; v < WIDTH; v += WIDTH * 0.15)
        HgWBoxFill(layer2, v, 0, R*0.5, HEIGHT*0.55, 0);
    HgWSetComposite(layer1, HG_BLEND_COPY);  /* 透明色で塗りつぶすため */
    x = R;
    for (count = 0; count < 30; count++) {   /* layer1 でアニメーション */
        HgWSetFillColor(layer1, HG_CLEAR);   /* 透明色を指定 */
        HgWCircleFill(layer1, x, HEIGHT/2.0, R, 0);
        x += R*2.0;
        if (x > WIDTH) x = R;
        HgWSetFillColor(layer1, HG_BLUE);
        HgWCircleFill(layer1, x, HEIGHT/2.0, R-1.0, 0);
        HgSleep(0.3);
    }
    HgClose();
    return 0;
}
```

図 23: レイヤを使って描きなおしを減らす例 (anime02.c)

図 23 の例は、アニメーションを行うレイヤに加え、ベースレイヤに背景、もう 1 つのレイヤに前景を描いています。この例の場合、前景と背景は動作中に変更されませんので、いったん描画すれば後から描きなおす必要がありません。

レイヤを使って描画と塗りつぶしのアニメーションを行う場合、塗りつぶしの色は白ではなく、透明色 HG_CLEAR を使い、コンポジットに HG_BLEND_COPY を指定する必要があります。または、レイヤに対して関数 HgLClear() を使用し、全体を消去する方法もあります。

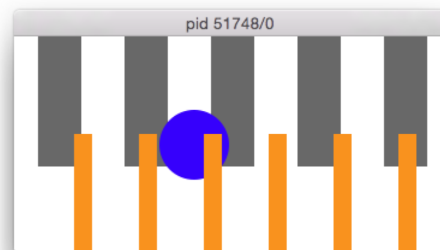


図 24: レイヤを使った描画の例

11.3 レイヤを利用したダブルバッファリング

前節で述べたアニメーションの方法では、動きを連続的に見せることができません。滑らかな移動を実現しようとして、画面の描きなおしのスピードを上げると、画面がちらついてしまいます。これは、移動前と移動後の図形だけではなく、描きなおしをするための塗りつぶしなどの過程も見えてしまうことに原因があります。

アニメーションを滑らかに見せるための基本的な技法に**ダブルバッファリング**があります。ダブルバッファリングでは、画面に表示させるための画像を保持しているメモリと、次の画面を作成するために描画を行う作業用メモリを別々にして、ひとつの画面が完成するたびに表示用メモリと作業用メモリを切り替えます。描画作業の間、そのメモリは表示されませんので、ちらつきを抑えることができます。

Handy Graphic では、2つのレイヤを用いてダブルバッファリングを実現できます。

```
#include <stdio.h>
#include <handy.h>

#define R  30
#define WIDTH   360.0
#define HEIGHT  180.0

int main(void)
{
    double x, y;
    int lid;
    HgOpen(WIDTH, HEIGHT);
    lid = HgWAddLayer(0);
    HgWSetFillColor(0, HG_DGRAY);  /* ベースレイヤに背景を描く */
    for (x = 20.0, y = 0.0; x < WIDTH; x += WIDTH * 0.2, y += 30.0)
        HgWBoxFill(0, x, y, R*1.2, HEIGHT*0.4, 0);

    double obj[2] = { 0.0, 0.0 };  /* 2つの円の x 軸方向の位置 */
    double dx[2] = { 6.0, 8.0 };   /* 2つの円の速度 */
    int i;
    for ( ; ; ) {
        HgLClear(lid);                       /* レイヤを消去 */
        HgWSetFillColor(lid, HG_BLUE);       /* 1つめの円を描く */
        HgWCircleFill(lid, obj[0], HEIGHT * 0.3, R, 0);
        HgWSetFillColor(lid, HG_RED);        /* 2つめの円を描く */
        HgWCircleFill(lid, obj[1], HEIGHT * 0.7, R, 0);
        for (i = 0; i < 2; i++) {
            double w = obj[i] + dx[i];       /* 円の次の位置を計算 */
            obj[i] = (w > WIDTH) ? 0.0 : w;
        }
        HgSleep(0.02);
    }                       /* このプログラムは Control-C で終了させる */
    HgClose();
    return 0;
}
```

図 25: ダブルバッファリングを使用しない例 (anime03.c)

まず、ダブルバッファリングを用いず、レイヤの消去と描画を繰り返す方式のプログラムを示します（図 25）。このプログラムでは、2つの円が左から右への移動を繰り返しますが、円は非常にちらついて見えます。

一方。図 26 はダブルバッファリングを使った例です。このプログラムを実行させると、円がちらつくことはなく、円の後ろの背景が透けて見えるようなことがありません。

図 26 のプログラムが図 25 と異なるのは、コメント (1), (2), (3) を付けた行だけです。次節でこの機能について説明します。

```c
#include <stdio.h>
#include <handy.h>

#define R  30
#define WIDTH   360.0
#define HEIGHT  180.0

int main(void)
{
    double x, y;
    doubleLayer layers;            /* (1) ダブルレイヤの変数を宣言する */
    HgOpen(WIDTH, HEIGHT);
    layers = HgWAddDoubleLayer(0); /* (2) ダブルレイヤを作成する */
    HgWSetFillColor(0, HG_DGRAY);  /* ベースレイヤに背景を描く */
    for (x = 20.0, y = 0.0; x < WIDTH; x += WIDTH * 0.2, y += 30.0)
        HgWBoxFill(0, x, y, R*1.2, HEIGHT*0.4, 0);

    double obj[2] = { 0.0, 0.0 };  /* 2つの円のx軸方向の位置 */
    double dx[2] = { 6.0, 8.0 };   /* 2つの円の速度 */
    int i;
    for ( ; ; ) {
        int lid = HgLSwitch(&layers);          /* (3) レイヤを切り替える */
        HgLClear(lid);                         /* レイヤを消去 */
        HgWSetFillColor(lid, HG_BLUE);         /* 1つめの円を描く */
        HgWCircleFill(lid, obj[0], HEIGHT * 0.3, R, 0);
        HgWSetFillColor(lid, HG_RED);          /* 2つめの円を描く */
        HgWCircleFill(lid, obj[1], HEIGHT * 0.7, R, 0);
        for (i = 0; i < 2; i++) {
            double w = obj[i] + dx[i];         /* 円の次の位置を計算 */
            obj[i] = (w > WIDTH) ? 0.0 : w;
        }
        HgSleep(0.02);
    }                      /* このプログラムは Control-C で終了させる */
    HgClose();
    return 0;
}
```

図 26: ダブルバッファリングを使用する例 (anime04.c)

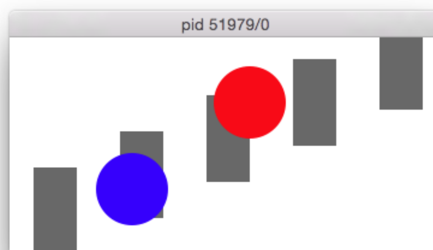


図 27: 2つの円のアニメーションの表示例

11.4 ダブルレイヤ

Handy Graphic では、ダブルバッファリングの機能を使ってアニメーションを簡単に実現できるように、ダブルレイヤと呼ぶ機能を提供します（この呼び名は Handy Graphic に固有のもので、一般的な用語ではありません）。

ダブルレイヤは2つのレイヤをペアにして扱うものです。ダブルレイヤを表現するためのデータ型とし

て doubleLayer 型を用意しています。これは次のように定義された構造体です。

```
typedef struct {
    int display;
    int hidden;
} doubleLayer;
```

使いやすさを考えて、２つのレイヤを同時に作成する関数 HgWAddDoubleLayer() を用意しています。

```
doubleLayer HgWAddDoubleLayer(int wid)
```
<u>引数</u>　　wid:　ウィンドウ id
<u>返り値</u>　　ダブルレイヤ

この関数 HgWAddDoubleLayer() は次のように定義されています。構造体のメンバ display は表示に利用されるレイヤ、メンバ hidden は表示には利用していないレイヤになります。

```
doubleLayer HgWAddDoubleLayer(int wid) {
    doubleLayer dl;
    dl.display = HgWAddLayer(wid);
    dl.hidden = HgWAddLayer(wid);
    HgLShow(dl.display, 1);
    HgLShow(dl.hidden, 0);
    return dl;
}
```

ダブルバッファリングを行うには、２つのレイヤの役割を切り替える必要があります。そのために関数 HgLSwitch() を用意しています。

```
int HgLSwitch(doubleLayer *dp)
```
<u>引数</u>　　dp:　ダブルレイヤ構造体へのポインタ
<u>返り値</u>　　描画に使うレイヤのレイヤ id

関数 HgLSwitch() は次のように定義されています。

```
int HgLSwitch(doubleLayer *dp)
{
    int disp = dp->display;
    int hidd = dp->hidden;
    HgLSetVisible(hidd, 1, disp, 0, 0);
    dp->display = hidd;
    dp->hidden = disp;
    return disp;
}
```

関数 HgLSwitch() を呼び出すたび、２つのレイヤの役割が入れ替わり、表示されない方のレイヤ id が関数の返り値になります。返り値のレイヤに対して描画を行い、アニメーションの画面をひとつ描き終わったら、再び関数 HgLSwitch() を呼び出して次の画面の描画に移ります。このようにして構造体と関数を使うことによって、図 26 のプログラムはダブルバッファリングを比較的簡単に記述できています。

なお、ダブルレイヤをウィンドウから削除するための関数も用意されています。

```
int HgLRemoveDoubleLayer(doubleLayer dl)
```

引数　　dl:　ダブルレイヤ

返り値　0:　正常　-1:　異常

図 26 のようなアニメーションを行うプログラムを作成しようとすると、プログラムの概要は次のようになるでしょう。

```
doubleLayer layers = HgWAddDoubleLayer(0); /* ダブルレイヤを作成する */
// ...
while ( 条件 ) {
    int lid = HgLSwitch(&layers);          /* レイヤを切り替える */
    HgLClear(lid);                         /* レイヤを消去 */
    //
    // ここで、レイヤ lid に対して次の画面を描画する。
    //
    HgSleep(...);  /* 必要なら少々待つ（必須ではない） */
}
```

アニメーションの実現には、doubleLayer 型や関数 HgLSwitch() を必ず使わなければならないわけではありません。同様な、あるいは拡張された機能を自分で定義することもできるでしょう。

12　描画結果の保存

12.1　ビットマップ画像として保存する

描画した結果を PNG 形式のファイルとして保存できます。

HgDisplayer のメニューから「描画 → 画像を保存」を選択すると、ファイル名と保存場所を指定して、その時に描かれている内容を保存できます。

プログラムから保存を指定することもできます。これには次の関数を使います。

```
int HgSave(const char *str)
```
引数　　str:　ファイルのパス
返り値　0:　正常、　-1:　異常

```
int HgWSave(int win, const char *str)
```
引数　　win:　ウィンドウ id　str:　ファイルのパス
返り値　0:　正常、　-1:　異常

```
int HgLSave(int gin, const char *str)
```
引数　　win:　レイヤ id　str:　ファイルのパス
返り値　0:　正常、　-1:　異常

関数 HgSave() は標準ウィンドウ、関数 HgWSave() は指定したウィンドウについて、呼び出された時に描かれている内容を画像ファイルに保存します。ファイルは実行時のカレントディレクトリからの相対パス、あるいは絶対パスで指定します。必要なら拡張子（.png）が付けられます。書き出しが正常に行えなかった場合はエラーになります。

関数 HgLSave() には、レイヤ id を指定します。この関数は、指定されたレイヤに描画されている内容だけをファイルに保存します。関数 HgSave() および関数 HgWSave() で保存した画像の背景は白色になりますが、関数 HgLSave() の場合、背景は透明色です。

12.2　描画履歴を記録する

Handy Graphic では描画した結果を PDF 形式のファイルとして保存できますが、そのためには、保存すべき描画内容を記録しておく必要があります。

描画履歴は、レイヤごとに記録します。関数 HgLPDFRecord() はレイヤを指定して履歴の記録を開始します。関数 HgLPDFCancel() は描画履歴の記録をやめて、それまでの情報を破棄します。

```
int HgLPDFRecord(int lid)
```
引数　　lid:　レイヤ id
返り値　0:　正常、　-1:　異常

```
int HgLPDFCancel(int lid)
```
引数　　lid:　レイヤ id
返り値　0:　正常、　-1:　異常

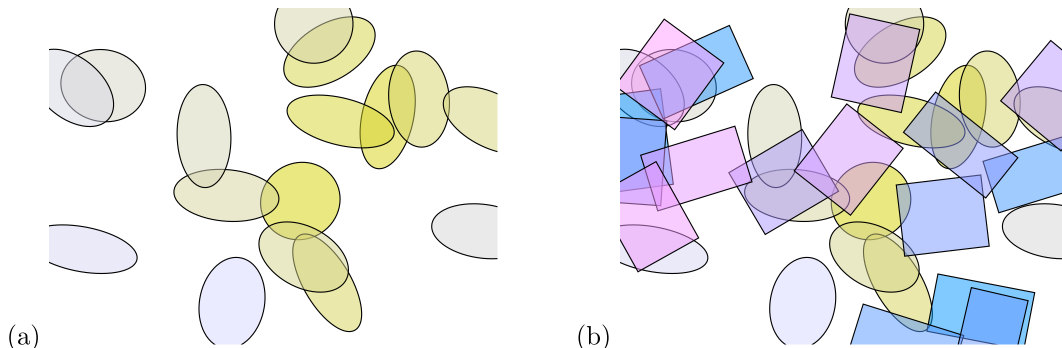
描画履歴は、関数 HgLPDFCancel() を明示的に使用しなくても、関数 HgClear()、HgWClear()、または HgLClear() を使って描画内容を消去すると削除されます。また、そのレイヤを削除した時、ウィンドウをクローズした時、あるいはプログラムが終了した時にも削除されます。

```
#include <stdlib.h>
#include <handy.h>

void draw(int lid, int w, int h, int flag)
{
    int n;
    for (n = 0; n < 16; n++) {
        double x = (double)(random() % w);    /* 位置を乱数で決める */
        double y = (double)(random() % h);
        double r = (double)(random() % 30) + 20.0; /* 大きさ */
        double a = (random() % 31415) / 10000.0;   /* 回転角 */
        if (flag) { // 長方形を半透明で塗る
            HgWSetFillColor(lid, HgRGBA(n/15.0, 0.5, 1.0, 0.5));
            HgWRectFill(lid, x, y, r, 70.0 - r, a, 1);
        }else { // 楕円を半透明で塗る
            HgWSetFillColor(lid, HgRGBA(0.8, 0.8, n/15.0, 0.5));
            HgWOvalFill(lid, x, y, r, 70.0 - r, a, 1);
        }
    }
}

int main(void)
{
    srandomdev();                       /* 乱数の初期設定 */
    int wid = HgWOpen(200, 200, 400, 300);
    int layer1 = HgWAddLayer(wid);  /* レイヤを追加 */
    int layer2 = HgWAddLayer(wid);
    for ( ; ; ) {
        int ans = 0;
        HgLPDFRecord(layer1);       /* 描画履歴を記録 */
        HgLPDFRecord(layer2);
        draw(layer1, 400, 300, 0);  /* 描画する */
        draw(layer2, 400, 300, 1);
        ans = HgAlert("保存しますか？", "保存", "再描画", NULL);
        if (ans == 0) {
            HgLPDFSave(layer1, "pdf-1.pdf");  /* レイヤについて保存*/
            HgWPDFSave(wid, "pdf-all.pdf");   /* 全体を保存 */
            break;
        }
        HgWClear(wid);       /* 描画履歴はここで破棄される */
    }
    return 0;
}
```

図 28: レイヤに描いた内容を PDF に保存するプログラム (pdftest.c)



(a) (b)

図 29: 保存された PDF ファイル

12.3 PDF ファイルに保存する

関数 HgLPDFRecord() で指定して記録した描画内容を、PDF 形式のファイルとして書き出します。記録されている描画履歴に変更を加えることはありません。

```
int HgLPDFSave(int lid, const char *str)
```

引数　lid:　レイヤ id　str:　ファイルのパス
返り値　0:　正常、　-1:　異常

```
int HgWPDFSave(int wid, const char *str)
```

引数　wid:　ウィンドウ id　str:　ファイルのパス
返り値　0:　正常、　-1:　異常

関数 HgLPDFSave() は、指定したレイヤに記録されている描画履歴を PDF ファイルに書き出します。ファイルは実行時のカレントディレクトリからの相対パス、あるいは絶対パスで指定します。

一方、関数 HgWPDFSave() はウィンドウ id を指定し、そのウィンドウにあるレイヤで、描画履歴を記録しているものがあれば、複数のレイヤをまとめて（重ね合わせて）PDF ファイルに書き出します。

なお、現在の実装では、PDF ファイルに書き出される際、重ね塗り（コンポジットモード）の設定（節 9.9）は無効になり、すべて `HG_BLEND_SOVER` の状態で表示されます。

◆ PDF 形式のファイルに描画結果を保存する例題

図 28 は、節 12.2、節 12.3 で説明した関数を使って描画結果を PDF 形式のファイルに書き出す例です。

このプログラムでは 2 つのレイヤを作成し、それぞれに長方形と楕円を描きます。塗りつぶしの色として半透明色を使っていますので、重なった場合にも下になった図形が透けて見えます。

図 29(a) は一方のレイヤに描かれた内容を関数 HgLPDFSave() で保存したもの、図 29(b) は関数 HgW-PDFSave() を使って 2 つのレイヤに描かれた内容を保存したものです。

13 サウンド

13.1 サウンド機能の概要

効果音や BGM のサウンドファイルを再生する程度の簡単な機能が利用できます。

音源として、効果音や音楽が保存されたファイルが必要です。システムに用意されている効果音を利用することもできます。これらのファイルからデータを読み込み、読み込んだデータは hgsound 型の**サウンド id** で管理します。サウンド id を指定して再生や中断、停止を行います。

13.2 サウンドデータの読み込み

```
hgsound HgSoundLoad(const char *path)
```
引数　path: サウンドデータを格納したファイルのパス
返り値　NoSound（マクロ）: 異常　それ以外: サウンド id

ファイルを指定し、サウンドデータを読み込みます。ファイルは実行時のカレントディレクトリからの相対パス、あるいは絶対パスで指定します。読み込み可能な代表的なデータ形式は、拡張子が mp3、m4a, aiff、wav のものです。

返り値はサウンド id で、これを用いて再生などを行うことができます。

次の関数を使うことで、システムが用意している効果音を読み込むことができます。

```
hgsound HgSoundWithName(const char *name)
```
引数　name: 効果音の名前
返り値　NoSound（マクロ）: 異常　それ以外: サウンド id

効果音の名前は、「システム環境設定」の「サウンド」の中にある「サウンドエフェクト」で確認することができます。例えば、“Funk” を引数に指定できます（拡張子は不要）。

なお、これらに対応するファイルは /System/Library/Sounds にあります。また、~/Library/Sounds/ に追加することもできます。

13.3 音量と繰り返しの設定

```
void HgSoundVolume(hgsound sid, double volume, int loop)
```
引数　sid: サウンド id　volume: 音量　loop: 繰り返し再生するかどうか

サウンドデータを再生する際の音量を、$0.0 \leq v \leq 1.0$ の範囲で指定します。サウンドが大音量で再生されるのを避けるため、サウンドデータを読み込んだ時点で音量は 0.2 に設定されています。必要に応じて再設定して下さい。

引数 loop は、サウンドを繰り返し何度も再生するかどうかを指定します。

13.4 再生と停止

```
void HgSoundPlay(hgsound sid)
```
引数　sid: サウンド id

```
void HgSoundStop(hgsound sid)
```
引数　sid: サウンド id

関数 HgSoundPlay() は、サウンド id を指定して再生を開始します。繰り返し再生を指定していなければ、サウンドは最後まで再生して停止します。

サウンドの再生を途中で停止させるためには、関数 HgSoundStop() を使います。

13.5 一時停止と再開

```
void HgSoundPause(hgsound sid)
```
引数　　sid:　サウンド id

```
void HgSoundResume(hgsound sid)
```
引数　　sid:　サウンド id

関数 HgSoundPause() は、サウンドの再生を一時停止させます。関数 HgSoundResume() を使うことで、停止した位置から再生を再開できます。

13.6 サウンドデータの解放

```
int HgSoundUnload(hgsound sid)
```
引数　　sid:　サウンド id

返り値　0:　正常　-1:　異常

いったん読み込んだサウンドデータをもう使わない場合、この関数を使ってデータをメモリ上から排除できます。例えば、ゲームの開始時にしか使わないサウンドは解放しておいた方がよいでしょう。

13.7 ビープ音

```
void HgBeep(void)
```

警告、あるいは注意喚起のためにシステムの効果音を鳴らします。自分でサウンドデータを管理しなくてよいので簡単です。

付録　関数の一覧

ウィンドウ

関数の役割	関数	指定ウィンドウ用の関数	参照
ウィンドウを作成	HgOpen(wd,hg)	HgWOpen(x,y,wd,hg)	3.1,6.1
ウィンドウを閉じる	HgClose()	HgWClose(w)	3.2,6.3
全ウィンドウを閉じる	HgCloseAll()		6.3
ウィンドウタイトル	HgSetTitle(str,...)	HgWSetTitle(w,str,...)	6.5
ウィンドウの大きさを得る	HgGetSize(wp,hp)	HgWGetSize(w,wp,hp)	6.4
画面の大きさを得る	HgScreenSize(wp,hp)		6.1

図形の描画

関数の役割	関数	指定ウィンドウ用の関数	参照
２点間に線分を描く	HgLine(x0,y0,x1,y1)	HgWLine(w,x0,y0,x1,y1)	4.1
カレントポイントを移動	HgMoveTo(x,y)	HgWMoveTo(w,x,y)	5.1
指定点まで線分を描く	HgLineTo(x,y)	HgWLineTo(w,x,y)	5.1
円を描く	HgCircle(x,y,r)	HgWCircle(w,x,y,r)	4.2
円を塗りつぶす	HgCircleFill(x,y,r,s)	HgWCircleFill(w,x,y,r,s)	4.2
楕円を描く	HgOval(x,y,r1,r2,a)	HgWOval(w,x,y,r1,r2,a)	5.4
楕円を塗りつぶす	HgOvalFill(x,y,r1,r2,a,s)	HgWOvalFill(w,x,y,r1,r2,a,s)	5.4
円弧を描く	HgArc(x,y,r,a0,a1)	HgWArc(w,x,y,r,a0,a1)	5.3
扇型を描く	HgFan(x,y,r,a0,a1)	HgWFan(w,x,y,r,a0,a1)	5.3
扇型を塗りつぶす	HgFanFill(x,y,r,a0,a1,s)	HgWFanFill(w,x,y,r,a0,a1,s)	5.3
長方形を描く	HgBox(x,y,wd,hg)	HgWBox(w,x,y,wd,hg)	4.3
	HgRect(x,y,r1,r2,a)	HgWRect(w,x,y,r1,r2,a)	5.4
長方形を塗りつぶす	HgBoxFill(x,y,w,h,s)	HgWBoxFill(w,x,y,wd,hg,s)	4.3
	HgRectFill(x,y,r1,r2,a,s)	HgWRectFill(w,x,y,r1,r2,a,s)	5.4
文字列を描く	HgText(x,y,str,...)	HgWText(w,x,y,str,...)	4.4
文字列のサイズ	HgTextSize(xp,yp,str,...)	HgWTextSize(w,xp,yp,str,...)	5.7
折れ線を描く	HgLines(n,x,y)	HgWLines(w,n,x,y)	5.2
多角形を描く	HgPolygon(n,x,y)	HgWPolygon(w,n,x,y)	5.2
多角形を塗りつぶす	HgPolygonFill(n,x,y,s)	HgWPolygonFill(w,n,x,y,s)	5.2
全消去	HgClear()	HgWClear(w)	4.6

線の太さ、色、フォントの指定

関数の役割	関数	指定ウィンドウ用の関数	参照
線の太さを指定	HgSetWidth(t)	HgWSetWidth(w,t)	3.3
灰色を作る	HgGray(g)		4.5
	HgGrayA(g,al)		5.5
色を RGB で作る	HgRGB(r,g,b)		4.5
	HgRGBA(r,g,b,al)		5.5
カラーコード	HgColorCode(code)		4.5
線の色を指定	HgSetColor(c)	HgWSetColor(w,c)	3.4
塗りつぶし色を指定	HgSetFillColor(c)	HgWSetFillColor(w,c)	3.4
フォントを指定	HgSetFont(f,sz)	HgWSetFont(w,f,sz)	3.5
	HgSetFontByName(nm,sz)	HgWSetFontByName(w,nm,sz)	5.6
透明色の塗り方	HgSetComposite(m)	HgWSetComposite(w,m)	9.9

アプリケーション座標

関数の役割	関数	指定ウィンドウ用の関数	参照
アプリケーション座標	HgCoordinate(x,y,sc)	HgWCoordinate(w,x,y,sc)	7.1
座標変換	HgTransWtoA(wx,wy,ax,ay)	HgWTransWtoA(w,wx,wy,ax,ay)	7.2
座標変換	HgTransAtoW(ax,ay,wx,wy)	HgWTransAtoW(w,ax,ay,wx,wy)	7.2
有効／無効の切替	HgCoordinateEnable(f)	HgWCoordinateEnable(w,f)	7.3

イベント

関数の役割	関数	指定ウィンドウ用の関数	参照
キー入力を得る	HgGetChar()	HgWGetChar(w)	8.1
イベントマスクを設定	HgSetEventMask(m)	HgWSetEventMask(w,m)	8.2
イベントマスクを得る	HgGetEventMask()	HgWGetEventMask(w)	8.2
イベントを取得	HgEvent()		8.3
	HgEventNonBlocking()		8.4
タイマを設定	HgSetIntervalTimer(t)		8.5
	HgSetAlarmTimer(t)		8.5

レイヤ

関数の役割	関数	参照
レイヤを追加	HgWAddLayer(wid)	9.2
レイヤの描画を消去	HgLClear(lid)	9.3
表示／非表示を切り替える	HgLShow(lid, f)	9.4
	HgLSetVisible(l1, f1, ...)	9.4
レイヤの順序を変更	HgLMove(lid, pos)	9.5
レイヤをコピーする	HgLCopy(lid, othr)	9.6
レイヤを削除	HgLRemove(lid)	9.7
レイヤを含むウィンドウ	HgWindowOfLayer(lid)	9.8
レイヤの枚数	HgWLayers(wid)	9.5
ダブルレイヤを作る	HgWAddDoubleLayer(wid)	11.4
ダブルレイヤを切り替える	HgLSwitch(wl)	11.4

ビットマップ画像

関数の役割	関数	指定ウィンドウ用の関数	参照
画像を読み込む	HgImageLoad(path)		10.2
画像を解放する	HgImageUnload(gid)		10.6
画像サイズ	HgImageSize(g,xp,yp)		10.3
画像を複製する	HgImageDup(g,s,f)		10.4
	HgImageDupRect(g,s,f,x,y,w,h)		10.4
画像を描く	HgImagePut(x,y,g,s,a)	HgWImagePut(w,x,y,g,s,a)	10.5
	HgImageDraw(x,y,g)	HgWImageDraw(w,x,y,g)	10.5
	HgImageDrawRect(dx,dy,dw,dh,g,sx,sy,sw,sh)	HgWImageDrawRect(w,dx,dy,dw,dh,g,sx,sy,sw,sh)	10.5

描画結果の保存

関数の役割	関数	指定ウィンドウ用の関数	参照
ビットマップ画像を保存	HgSave(fn)	HgWSave(wid, fn)	12.1
描画履歴を記録	HgLPDFRecord(lid)		12.2
描画記録を破棄	HgLPDFCancel(lid)		12.2
レイヤを PDF で保存	HgLPDFSave(lid, fn)		12.3
複数レイヤを PDF で保存	HgWPDFSave(wid, fn)		12.3

サウンド

関数の役割	関数	参照
サウンドデータを読む	HgSoundLoad(fn)	13.2
	HgSoundWithName(nm)	13.2
サウンドデータの解放	HgSoundUnload(sid)	13.6
音量と繰り返しの設定	HgSoundVolume(sid,vol,loop)	13.3
再生	HgSoundPlay(sid)	13.4
停止	HgSoundStop(sid)	13.4
一時停止	HgSoundPause(sid)	13.5
再開	HgSoundResume(sid)	13.5
ビープ音	HgBeep()	13.7

その他

関数の役割	関数	参照
指定した時間だけ待つ	HgSleep(s)	5.9
パネルを表示	HgAlert(s,b0,b1,b2,...)	5.8

互換性に関するコメント

バージョン 0.6 以前の関数

Handy Graphic 0.6 までは、ソースコードが利用している文字コードを指定する関数 HgEncoding() を提供していましたが、バージョン 0.7 以降では廃止しました。文字コードは UTF-8 でエンコードされた

Unicode を使って下さい。関数 HgEncoding() はプログラムに記述しても効果がありません。今後のバージョンアップに伴って廃止する予定です。

バージョン 0.5 以前の関数

Handy Graphic 0.7 までは、バージョン 0.5 との互換性のため、以下の関数が利用できましたが、バージョン 0.8 以降では廃止しました。

```
HgSetPaintColor()
HgWSetPaintColor()
HgEventMask()
HgWEventMask()
msleep()
```