Handy Graphic ユーザーズガイド 【C++編】

for Handy Graphic Version 0.9.6　2025-09-24 荻原剛志

1　Handy Graphic とは

1.1　概要

Handy Graphic は、簡単な作図を行うための機能を C/C++プログラムから使えるようにまとめた**グラフィックライブラリ**です。

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、C++のプログラムで Handy Graphic を使うための網羅的な解説です。全体の構成は、先行する C 言語のための「Handy Graphic ユーザーズガイド」（以下、「C 言語編」と呼称）とおおよそ対応しています。ただし、C 言語編がプログラミングの初学者の利用を想定しているのに対し、この C++編は C 言語によるプログラミングを習得済みで、C++プログラミングにおける基本的な概念（クラスの定義や継承の概念など）は把握している利用者を想定して記述してあります。

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。C 言語で Handy Graphic を利用した経験があれば、C++でプログラミングするのも比較的容易と思われます。

なお、この PDF ファイルの最後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。

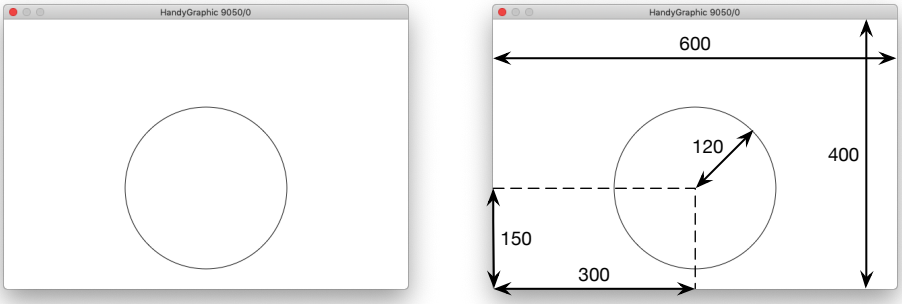
```
#include <iostream>
#include <handy++>

int main()
{
    char ch;
    { // 変数 win の有効範囲
        hg::Window win = hg::Window(600.0, 400.0); // 描画用ウィンドウを開く
        win.circle(300.0, 150.0, 120.0);  // 中心 (300, 150)、半径 120 の円を描く
        std::cin.get(ch);                 // 改行の入力でウィンドウがクローズする
    }
    std::cin.get(ch);                     // 改行の入力でプログラムが終了する
    return 0;
}
```

図 1: 円を描くプログラム (prog01.cpp)

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。



(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

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、このプログラムでは C++言語の標準入出力機能を使ってターミナルからの入力を待ちます。ターミナルに改行を 1 つ入力するとウィンドウがクローズし、もう一度入力するとプログラムは終了します。

“handy++” というヘッダファイルを読み込むことで、**名前空間 hg** で宣言されたクラス Window を使うことができます。変数 win にクラス Window のインスタンスが作られますが、このインスタンスが破棄される時に自動的にウィンドウもクローズされます。図 1 のプログラムは、変数 win の有効範囲から出ることによってウィンドウがクローズされることを確認する例になっています。

描画用ウィンドウに円を描くには、クラス Window のメンバ関数 circle() を使います。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。メンバ関数 box(x, y, w, h) は、左下が (x, y), 幅 w, 高さ h の長方形を描きます。

```
#include <iostream>
#include <handy++>

int main()
{
    auto win = hg::Window( 500.0, 400.0 );
    double w = 100.0, h = 40.0;
    for (int i = 1; i <= 15; i++) {
        w -= 4.0;
        h += 4.0;
        win.box(i * 28.0, i * 20.0, w, h);
    }
    char ch;
    std::cin.get(ch);
    return 0;
}
```

図 3: 長方形を描くプログラム (prog02.cpp)

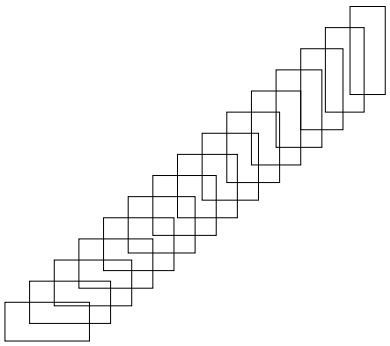


図 4: 長方形を描くプログラムの実行結果

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。
- handy++:　C++言語のヘッダファイルです。/usr/local/include に作成されます。
- libhg.dylib:　C 言語のライブラリファイルです。/usr/local/lib に作成されます。
- libhg++.dylib:　C++言語の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 C 言語の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。/usr/local/bin に作成されます。
- hgc++:　Handy Graphic を使った C++言語の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。/usr/local/bin に作成されます。
- hgcc-uninstall:　インストール済みの Handy Graphic をすべて抹消するためのコマンドです。実行すると Handy Graphic は消去されてしまいます。通常は使う必要はありません。

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.cpp 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 hgc++ 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）。

```
$ hgc++ draw.cpp
```

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。

```
$ hgc++ -Wall -o draw draw.cpp
```

なお、オブジェクトファイルだけを生成したい場合には -c オプションを指定できます。この場合、ヘッダファイルのみが参照され、ライブラリの指定は行いません。

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ではなく、C++ライブラリの実装に用いている C 言語のライブラリの名前です。

“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。いったん HgDisplayer と利用者のプログラムを終了させ、HgDisplayer を起動してから利用者のプログラムを動かしてみて下さい。それでも描画できない場合には、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。データ型のチェックはコンパイラで行いますので、それ以外の原因（例えば、可変長引数の個数や型が適切ではない、指定できる値の範囲を超えている、など）の場合が多いと思われます。

“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。

2.5 C++言語のバージョンを設定する

C++には新旧いくつかの言語規格が存在します。例えば、 c++11 や c++17 あるいは c++23 などが指定できます。また、GNU 独自の言語機能を使うことができる設定も可能です。C++のコンパイラ `c++` で利用可能な言語規格は、オンラインマニュアルで `c++` を参照し、`-std=<standard>` というオプションの解説を読むと分かります。

通常、C++のコンパイラを使う場合、コマンドラインのオプションに `-std=c++20` のような指定を追加します。

Handy Graphic のコンパイルには hgc++ コマンドを利用するのが便利ですが、hgc++ では `-std=c++17` を標準の設定としています。これを変更するには、 CPPSTD という環境変数を設定してから hgc++ コマンドを使用して下さい。例えば、c++20 を指定するには以下のようにすれば環境変数が設定できます。

```
$ CPPSTD="c++20"; export CPPSTD
```

あるいは、その hgc++ コマンドについてだけ一時的に環境変数を設定してコンパイルすることもできます。

```
$ CPPSTD="c++20" hgc++ draw.cpp
```

3　描画のための設定

3.1　ウィンドウの作成

Handy Graphic では、**クラス Window** のインスタンスを生成してウィンドウを表示し、そこに図形を描画します。クラス Window のコンストラクタのうち、一番簡単なものを示します。

クラス Window **のコンストラクタ**

```
explicit hg::Window(double w, double h)
```

引数　　w,h:　幅と高さ

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、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。

この文書でクラス名などの説明を行う場合、基本的には名前空間の修飾を行いますが、コンストラクタやメンバ関数の説明を行う場合、わかりやすさを損なわない範囲で省略した形式を利用することがあります。プログラムの厳密な宣言形式としては必ずしも正しくない場合があることに注意して下さい。

例えば、上記のコンストラクタでインスタンスを生成する場合、

```
hg::Window newwin = hg::Window(400, 300)
```

あるいは

```
hg::Window newwin(400, 300)
```

もしくは

```
auto newwin = hg::Window(400, 300)
```

のように記述できます。なお、explicit が付いているため、代入形式でのインスタンス生成はできません。一方、クラス Window のコンストラクタの新しい定義を記述する場合、

```
hg::Window::Window(int w, int h)
```

のような定義が必要ですが、必ずしも（特に初学者には）わかりやすいものではありません。

メンバ関数などの説明の場合も同様です。例えば「クラス Canvas のメンバ関数」であることを明示して説明している場合、その関数名をさらに hg::Canvas で修飾することはしません。

3.2　線の太さ

何も指定しない場合、図形の線の太さは 1 ピクセル（画素）分ですが、これを変更することができます。メンバ関数 setWidth() はクラス Window の基底クラスである **Canvas** というクラスで定義されており、Window のインスタンスに対して使うことができます。

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。太さの指定は、対象となるウィンドウに対してのみ有効です。

クラス Canvas（Window **の基底クラス）のメンバ関数**

```
void setWidth(double t)
```

引数　　t:　線の太さ

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。

3.3 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**クラス Color** のインスタンスで指定します。クラス Color の詳しい利用方法は節 4.5 を参照して下さい。ここでは次のコンストラクタについてのみ説明します。

クラス Color のコンストラクタ

```
hg::Color(unsigned long hex)
```

引数　　hex:　色を指定する数値

引数には色を表す整数値を指定しますが、よく使われる色はマクロで定義されています（表 1）。この節ではこのコンストラクタとマクロを用いた例を示します。

円や長方形を描くための線の色を指定するには、クラス Canvas の次のメンバ関数を使います。いったん色を指定すると、別の色を指定するまでそのウィンドウでは同じ色が使われます。なお、一度も指定しない場合は黒で描かれます。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void setColor(Color c)
```

引数　　c:　色の指定

表 1: 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 9.10）	HG_CLEAR

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、次のメンバ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そのウィンドウでは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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void setFillColor(Color c)
```

引数　　c:　色の指定

3.4 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 2 のマクロのいずれかを指定します（現在の実装では hg::font 型は int 型と同じです）。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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void setFont(hg::font font, double size)
```

引数　　font:　フォントを指定する定数名　　size:　文字サイズ

```
#include <iostream>
#include <handy++>

int main()
{
    char ch;
    hg::Window win = hg::Window(600.0, 400.0); // 描画用ウィンドウを開く
    win.setWidth(8.0);                  // 線の太さを 8 ピクセルにする
    win.setColor(hg::Color(HG_RED));    // 線の色を赤にする
    win.circle(250.0, 150.0, 120.0);    // 中心 (250, 150)、半径 120 の円を描く
    win.setWidth(5.0);                  // 線の太さを 5 ピクセルにする
    win.setColor(HG_BLUE);              // 線の色を青にする（Color を省略した例）
    win.box(150.0, 120.0, 400.0, 200.0);  // 長方形を描く
    std::cin.get(ch);                   // 改行の入力を待つ
    return 0;
}
```

図 5: 円、長方形、文字列を描くプログラム (prog03.cpp)

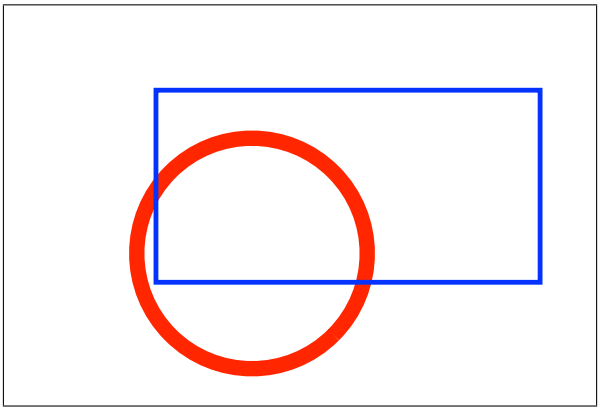


図 6: プログラムの実行結果

◆ 線の太さと色の指定の例

図 5、図 6 は、線の太さと色の指定の例です。ここでマクロ HG_BLUE で青色を指定している部分は、メンバ関数 setColor() の引数が Color 型であるため、コンパイル時に自動的に Color のコンストラクタが補われています。

表 2: 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

クラス Canvas のメンバ関数 line() 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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void line(double x0, double y0, double x1, double y1)
```

引数　　x0,y0:　線分の始点　　x1,y1:　線分の終点

Handy Graphic では点の座標 (x, y) を表す次のような構造体 hg::point をあらかじめ定義しています[1]。

```
struct point { double x, y; };
```

この構造体で線分の両端を指定するメンバ関数 line() の定義も用意してあります。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void line(hg::point p0, hg::point p1)
```

引数　　p0:　線分の始点　　p1:　線分の終点

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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void circle(double x, double y, double r)
```

引数　　x,y:　円の中心　　r:　半径

```
void circle(hg::point p, double r)
```

引数　　p:　円の中心　　r:　半径

```
void circleFill(double x, double y, double r, bool stroke = true)
```

引数　　x,y:　円の中心　　r:　半径　　stroke:　円周を描くかどうか

```
void circleFill(hg::point p, double r, bool stroke = true)
```

引数　　p:　円の中心　　r:　半径　　stroke:　円周を描くかどうか

circle() は、座標 (x, y) を中心とした半径 r の円を描きます。中心を hg::point 型で指定することもできます。

circleFill() 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、やはり中心を hg::point 型で指定することもできます。

塗りつぶしには setFillColor() で指定した色が使われます。引数 `stroke` が false の場合は円周を描きません。true を指定した場合、または引数を省略した場合、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。

[1] ヘッダファイル handy++ における実際の宣言は、C 言語との互換性のために必ずしもこの記述とは一致しません。

4.3 長方形

Handy Graphic では長方形領域の幅 w と高さ h を表す次のような構造体 hg::size をあらかじめ定義しています。

```
struct size { double w, h; };
```

また、左下隅の座標 (x, y) と幅 w と高さ h を指定して長方形領域を表現する構造体 hg::rect も用意しています。以下のように、コンストラクタが 2 種類定義されています。

```
struct rect {
    double x, y, w, h;
    rect(double x0, double y0, double w0, double h0) {
        x = x0; y = y0; w = w0; h = h0;
    }
    rect(hg::point p, hg::size sz) {
        x = p.x; y = p.y; w = sz.w; h = sz.h;
    }
};
```

長方形を描く場合、構造体 hg::point, hg::size, および hg::rect を使うこともできます。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void box(double x, double y, double w, double h)
```
引数　x,y: 左下隅の座標　w,h: 幅と高さ

```
void box(hg::point p, hg::size sz)
```
引数　p: 左下隅の座標　sz: 幅と高さ

```
void box(hg::rect r)
```
引数　r: 長方形領域

```
void boxFill(double x, double y, double w, double h, bool stroke = true)
```
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

```
void boxFill(hg::point p, hg::size sz, bool stroke = true)
```
引数　p: 左下隅の座標　sz: 幅と高さ　stroke: 周囲を描くかどうか

```
void boxFill(hg::rect r, bool stroke = true)
```
引数　r: 長方形領域　stroke: 周囲を描くかどうか

box() 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。左下隅の座標を hg::point 型で、幅と高さを hg::size 型で指定する方法と、hg::rect 型で長方形領域自体を指定する方法もあります。

boxFill() 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。hg::point 型、hg::size 型、hg::rect 型を使って指定する方法は box() と同様です。

塗りつぶしには setFillColor() で指定した色が使われます。引数 `stroke` が false の場合は周囲に長方形を描きません。true を指定した場合、または引数を省略した場合、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。

4.4 文字列

クラス Canvas（Window の基底クラス）のメンバ関数

```
void text(double x, double y, const std::string& str)
```

引数　　x,y:　左下隅の座標　　str:　C++文字列

```
void text(hg::point p, const std::string& str)
```

引数　　p:　左下隅の座標　　str:　C++文字列

text() は、座標 (x, y)（または hg::point 型）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。文字列は UTF-8 であることを前提としています。

C 言語の文字列（NUL 文字が終端の char 型の列）を使って、文字列を描画することもできます。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void cText(double x, double y, const char *str, ...)
```

引数　　x,y:　左下隅の座標　　str:　書式文字列

```
void cText(hg::point p, const char *str, ...)
```

引数　　p:　左下隅の座標　　str:　書式文字列

cText() も左下隅の座標を指定して文字列を描きます。文字列は UTF-8 であることを前提としています。

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。str に指定した文字列に `%` が含まれていると書式の記述と見なされますので、注意が必要です（逆に、`%` が含まれる文字列は `%%` と記述する必要があります）。

4.5 色を自分で作るには

クラス Color のコンストラクタには、既に述べた unsigned long 型を引数とするものの他に、赤、緑、青 (RGB) の三原色の濃度で色を指定するものと、グレーの濃度を指定するものがあります。

クラス Color のコンストラクタ

```
hg::Color(double r, double g, double b)
```

引数　　r:　赤の濃度　　g:　緑の濃度　　b:　青の濃度

引数の `r`、`g`、`b` は $0.0 \leq x \leq 1.0$ の範囲の実数値で、それぞれ赤、緑、青の濃度を表します。

例えば、`Color(1.0, 0.0, 0.0)` で赤、`Color(0.8, 0.8, 1.0)` で薄い青になります。

クラス Color のコンストラクタ

```
hg::Color(double g)
```

引数　　g:　グレーの濃度

```
#include <iostream>
#include <sstream>
#include <handy++>

int main()
{
    auto win = hg::Window(600.0, 400.0); // 描画用ウィンドウを開く
    win.setWidth(8.0);                   // 線の太さを 8 ピクセル分に設定する
    win.setColor(HG_DGRAY);              // 線の色を薄い灰色にする
    win.setFillColor(HG_YELLOW);         // 塗りつぶしの色を黄色にする
    win.line(10.0, 150.0, 560.0, 20.0);  // 直線（線分）を描く
    win.circleFill(200.0, 220.0, 175.0); // 円を塗りつぶし、円周も描く
    win.setFont(HG_M, 28);               // フォントを明朝体にする
    std::ostringstream text;             // 文字列を合成可能なクラス
    text << "降水確率は" << 30 << "%です。";
    win.text(240.0, 5.0, text.str());

    win.setFillColor(hg::Color(0xA3AFUL));   // 浅葱色で塗りつぶす
    win.boxFill(150.0, 100.0, 300.0, 80.0, false); // 長方形を塗り、辺は描かない
    win.setWidth(2.0);                   // 線の太さを 2 ピクセル分に設定する
    win.setColor(HG_BLACK);              // 線の色を黒にする
    win.circle(300.0, 220.0, 150.0);     // 円を描く（塗りつぶしなし）
    win.setFillColor(HG_WHITE);          // 塗りつぶしの色を白にする
    win.boxFill(200.0, 160.0, 320.0, 80.0, true);  // 長方形を塗り、辺も描く
    win.box(250.0, 220.0, 340.0, 80.0);  // 長方形を描く（塗りつぶしなし）

    char ch;
    std::cin.get(ch);                    // 改行の入力を待つ
    return 0;
}
```

図 7: 円、長方形、文字列を描くプログラム (prog04.cpp)

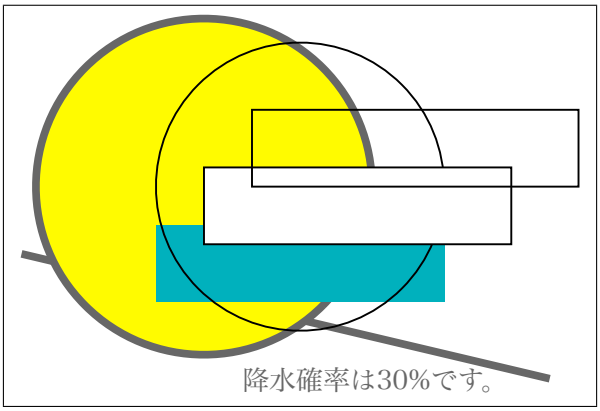


図 8: プログラムの実行結果

◆ 図形を重ね書きする例題

図 7、図 8 は、節 4.5 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。

文字列を描画するには、引数に string 型を指定します。ここでは書式指定などを使って文字列を合成できる ostringstream 型で文字列を構成しています。

引数の g は $0.0 \leq g \leq 1.0$ の範囲の実数値でグレーの濃度を指定し、0.0 が黒、1.0 が白です。

次のコンストラクタは既に説明したものですが、引数にマクロを指定するだけではなく、自分で整数値を指定することによってさまざまな色を表すことができます。

クラス Color のコンストラクタ

```
hg::Color(unsigned long hex)
```

引数　　hex:　色を指定する数値

Web サイトの記述でよく利用される色の表現（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（00 から FF まで）で表して並べたものです。上記のコンストラクタでは、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。ただし、通常の整数定数は int 型とみなされ、コンパイラが double 型に変換してしまいます。従って、整数定数の指定には型を表す UL または ul を付加する必要があります。

例えば浅葱色を表す[2]色コード「#00A3AF」の色を使うには Color(0x00A3AFUL) のようにします（引数は 0xA3AFUL と書いても構いません）。UL を付加している点に注意して下さい。

なお、Web の色コードで 3 桁のものは赤、緑、青の濃度を 1 桁の 16 進数（0 から F まで）で表して並べたもので、そのままでは上記のコンストラクタの引数にはできません。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、Color(0x33CC99UL) のようにします。

4.6　ウィンドウ内の全消去

クラス Window のメンバ関数

```
void clear()
```

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、このメンバ関数 clear() 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クラス Canvas のメンバ関数 clear() を使います。詳細は節 9.4 を参照して下さい。

[2]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。

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

クラス Canvas のメンバ関数 line() 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（描き始めの点だけ指定して図形は描かない）には次のメンバ関数を使います。引数は (x, y) の 2 つの値を指定する方法と、hg::point 型の値を指定する方法があります。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void moveTo(double x, double y)
```

引数　　x,y:　新しいカレントポイントの座標

```
void moveTo(hg::point p)
```

引数　　p:　新しいカレントポイントの座標

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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void lineTo(double x, double y)
```

引数　　x,y:　線分の終点の座標

```
void lineTo(hg::point p)
```

引数　　p:　線分の終点の座標

なお、2 点を指定して直線を描くメンバ関数 line() 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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void lines(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

```
void lines(int n, const hg::point *p)
```

引数　　n:　頂点数　p:　頂点の座標を格納した配列へのポインタ

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 軸の値を別の配列に格納してポインタ xp と yp で指定する方法と、hg::point 型の配列に格納して参照する方法があります。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。

メンバ関数 plygon() 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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void polygon(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

```
void polygon(int n, hg::point *p)
```
引数　　n:　頂点数　　p:　頂点の座標を格納した配列へのポインタ

```
void polygonFill(int n, const double *xp, const double *yp, bool stroke = true)
```
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

```
void polygonFill(int n, hg::point *p, bool stroke = true)
```
引数　　n:　頂点数　　p:　頂点の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

メンバ関数 polygon() は線を描くだけですが、メンバ関数 polygonFill() は描かれた図形を塗りつぶします。引数 stroke が false の場合は周囲に線を描きません。true を指定した場合、または引数を省略した場合、他の線図形と同じ太さ、同じ色の線で線を描きます。

5.3　円弧と扇型

クラス Canvas（Window の基底クラス）のメンバ関数

```
void arc(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

```
void arc(hg::point p, double r, double a0, double a1)
```
引数　　p:　円の中心　　r:　半径　　a0:　始点角度　　a1:　終点角度

メンバ関数 arc() 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 x と y を与える方法と、hg::point 型で指定する方法があります。この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（ラジアン）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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void fan(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

```
void fan(hg::point p, double r, double a0, double a1)
```
引数　　p:　円の中心　　r:　半径　　a0:　始点角度　　a1:　終点角度

```
void fanFill(double x, double y, double r,
        double a0, double a1, bool stroke = true)
```

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

```
void fanFill(hg::point p, double r, double a0, double a1, bool stroke = true)
```
引数　　p:　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

メンバ関数 fan() は扇型の周囲の線を描きます。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。

メンバ関数 fanFill() は塗りつぶされた扇型を描きます。引数 stroke が false の場合は周囲に線を描きません。true を指定した場合、または引数を省略した場合、他の線図形と同じ太さ、同じ色の線で線を描きます。

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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void oval(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

```
void oval(hg::point p, double r1, double r2, double a)
```
引数　　p:　楕円の中心　　r1, r2:　半径　　a:　傾きの角度

```
void ovalFill(double x, double y, double r1, double r2,
        double a, bool stroke = true)
```
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

```
void ovalFill(hg::point p, double r1, double r2, double a, bool stroke = true)
```
引数　　p:　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

中心座標は x と y を与える方法と、hg::point 型で指定する方法があります。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（ラジアン）で表し、回転させない時は 0 を指定します。

クラス Canvas（Window の基底クラス）のメンバ関数

```
void rect(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

```
void rect(hg::point p, double r1, double r2, double a)
```
引数　　p:　長方形の中心　　r1, r2:　半径　　a:　傾きの角度

```
void rectFill(double x, double y, double r1, double r2,
        double a, bool stroke = true)
```
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

```
void rectFill(hg::point p, double r1, double r2, double a, bool stroke = true)
```
引数　p: 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

すでに、長方形を描くためのメンバ関数として、box() と boxFill() を説明しました（節 4.3）。上記のメンバ関数 rect()、rectFill() 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。box() と異なるのは、傾きを指定できる点です。

図 9、図 10 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 cmath で定義されているマクロで、円周率 π を表します。

```
#include <iostream>
#include <cmath>
#include <handy++>

int main()
{
    hg::Window win = hg::Window(400.0, 300.0);
    win.setWidth(2.0);
    win.setFillColor(hg::Color(1.0, 0.7, 0.9)); // 薄紫色
    hg::point p = { 100.0, 150.0 };
    win.oval(p, 80, 40, 0);          // 楕円と長方形を描画
    win.rect(p, 80, 40, 0);
    win.ovalFill(300, 240, 80, 40, M_PI/6.0);        // 楕円と長方形を
    win.rectFill(300, 80, 80, 40, M_PI/6.0, false); // 傾けて描画
    win.setColor(hg::Color(0.8));    // 薄い灰色
    win.setWidth(12.0);
    win.arc(p, 100, M_PI*0.2, M_PI*0.8); // 円弧を描画
    char ch;
    std::cin.get(ch);
    return 0;
}
```

図 9: 楕円、長方形、円弧を描くプログラム (prog05.cpp)

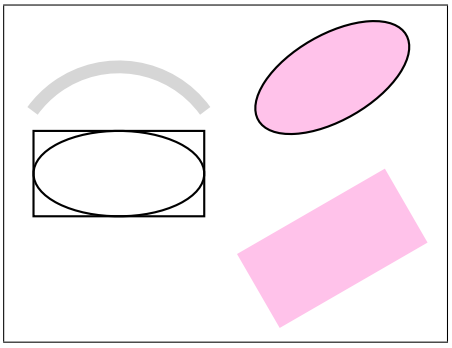


図 10: プログラムの実行結果

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 3.3）、RGB 各要素の強さ、グレーの濃度を指定する方法（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。

クラス Color のコンストラクタ

```
hg::Color(double r, double g, double b, double alpha)
```

引数　　r:　赤の濃度　　g:　緑の濃度　　b:　青の濃度　　alpha:　アルファ値

```
hg::Color(double g, double alpha)
```

引数　　g:　グレーの濃度　　alpha:　アルファ値

引数 alpha は透明度を示す値で**アルファ値**と呼ばれます。透明度という名前で呼ばれることも多いですが、$0.0 \leq alpha \leq 1.0$ の範囲の実数値で、0.0 が透明、1.0 が不透明ですので、むしろ「不透明の度合い」を表す数値になっています。その他の引数に関しては、既に説明したコンストラクタと同様です。

クラス Color のコンストラクタ

```
hg::Color(hg::Color clr, double alpha)
```

引数　　clr:　元の色　　alpha:　新たに指定するアルファ値

このコンストラクタは、既に存在している色にアルファ値を指定した新しい色を作ります。元の色が何らかのアルファ値を持っていた場合、それは無視されて新しく指定されたアルファ値が使われます。

例えば、`hg::Color(hg::Color(HG_RED), 0.2))` とすると、透明度の高い赤を作ることができます。

なお、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。`Color(0.0, 0,0)` のように記述しても、`Color(HG_CLEAR)` と同じ意味になります。

透明色、半透明色の使い方については図 11 のプログラム、節 9.10「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、メンバ関数 setFont() を使って指定できることを説明しました（節 3.4）。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。

クラス Canvas（Window の基底クラス）のメンバ関数

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

引数　　fontname:　フォント名　　size:　文字サイズ

戻り値　　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 22 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。

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 を返すものがあります。戻り値のチェックでは、数値を直接指定するよりも表 3 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、関数によって戻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。

表 3: 関数の戻り値のマクロ定義

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

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。

クラス Canvas（Window の基底クラス）のメンバ関数

```
hg::size getTextSize(const std::string& str)
```

引数　　str:　C++文字列
戻り値　　幅と高さ

```
hg::size getCTextSize(const char *str, ...)
```

引数　　str:　書式文字列
戻り値　　幅と高さ

メンバ関数 getCTextSize() では、文字列を描画する cText() と同様に、文字列 str に printf() 関数と同様の書式を指定できます。書式に対応する引数は str の後ろに記述します。

戻り値は hg::size 型ですが、描画できない何らかのエラーがある場合、幅も高さも 0.0 の値が返されます。

図 11、図 12 に、アルファ値のある図形を重ねて描く例を示します。この例では、フォント名でフォントを指定し、文字列の描かれる大きさを調べてその範囲を長方形で塗りつぶしています。後から描かれた図形が半透明なため、下の図形が透けて見えることに注意して下さい。

5.8 メッセージパネルを表示

プログラムの実行中に、利用者に動作を選択、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、通常のプログラムと同様にターミナルでの入出力で対応することもできますが、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。

名前空間 hg のグローバル関数

```
int hg::alert(const std::string& str, const std::string& btn0 = "",
          const std::string& btn1 = "", const std::string& btn2 = "")
```

引数　　str:　メッセージ　　btn0, btn1, btn2:　ボタンに表示する文字列
戻り値　　0, 1, 2:　ボタン 0、1、または 2 が押された、　-1:　異常

```
#include <iostream>
#include <string>
#include <handy++>

const double WW = 540.0, WH = 360.0; // ウィンドウの幅と高さ

void textBox(hg::Window& wn, double y, const std::string& str) {
    hg::size sz = wn.getTextSize(str);  // テキストの描画サイズを得る
    double x = (WW - sz.w) / 2.0;
    wn.boxFill(x, y, sz.w, sz.h, false); // テキストと長方形を
    wn.text(x, y, str);                  // 中央にそろえて描く
}

int main() {
    auto win = hg::Window(WW, WH);
    win.setColor(HG_WHITE);                              // 文字は白
    win.setFontByName("HiraMaruProN-W4", 42.0);          // フォントを指定
    win.setFillColor(hg::Color(0.7, 0.3, 1.0, 0.3)); // 薄い紫
    textBox(win, WH * 0.6, "月のころはさらなり");
    win.setFillColor(hg::Color(1.0, 1.0, 0.0, 0.5)); // 透明な黄色
    win.circleFill(WW * 0.6, WH * 0.5, WH * 0.4, 0); // 円を描く
    win.setFontByName("Meiryo-Bold", 32.0);              // フォントを指定
    win.setFillColor(hg::Color(1.0, 0.0, 0.0, 0.5)); // 透明な赤
    textBox(win, WH * 0.3, "雪の降りたるはいふべきにもあらず");
    char ch;
    std::cin.get(ch);
    return 0;
}
```

図 11: アルファ値のある図形を重ねて描く例 (bound.cpp)

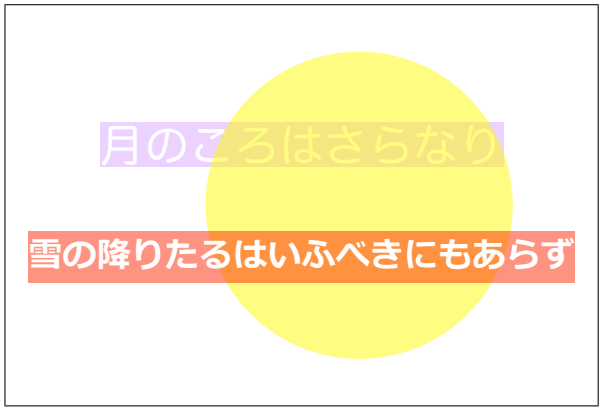


図 12: プログラムの実行結果

btn0 はパネル上で最も上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。btn1 は、btn0 のほかの選択肢がある場合に指定します。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。

不要なボタンの分は引数を省略できますが、btn0、btn1、btn2 のすべてを省略した場合、「OK」ボタンが 1 つだけ表示されます。

関数の戻り値は、パネル上で押したボタンを示す整数値です。

名前空間 hg のグローバル関数

```
int hg::alertCText(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

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

関数 alertCText() では引数は省略できません。必要のないボタンには NULL を指定します。

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

関数の戻り値は、パネル上で押したボタンを示す整数値です。

図 13、図 14 に、メッセージパネルを使う例を示します。

```
#include <iostream>
#include <handy++>

int main()
{
    int r = hg::alert(
        "なんか良くわからないモンスターが現れました！",
        "戦う", "逃げる", "ググる");
    std::cout << r << std::endl;  // ボタンの値を表示
    return 0;
}
```

図 13: メッセージパネルを表示する (alert.cpp)

図 14: 表示されるパネル

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。

名前空間 hg のグローバル関数

```
void hg::sleep(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

クラス Window のインスタンスは画面上に表示されたウィンドウの 1 つに対応しています。インスタンスを新しく生成するたびに、新しいウィンドウが作成されます。

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。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。

節 3.1 で説明したものも含め、クラス Window のコンストラクタをまとめて説明します。これらはいずれも explicit が指定されているため、例えば代入の形式でインスタンスを作ることはできません。

クラス Window のコンストラクタ

```
explicit hg::Window::Window(double w, double h)
```

引数　　`w,h:`　幅と高さ

ウィンドウを画面の中央に表示します。次の 2 つのコンストラクタは、ウィンドウの左下隅が画面上のどの位置かを指定してウィンドウを表示できます。

クラス Window のコンストラクタ

```
explicit hg::Window::Window(double x, double y, double w, double h)
```

引数　　`x,y:`　左下隅の座標　　`w,h:`　幅と高さ

```
explicit hg::Window::Window(hg::rect rect)
```

引数　　`rect:`　画面上の左下隅の座標、ウィンドウの幅と高さを指定

位置を指定することで、ウィンドウ上に複数のウィンドウを同時に表示させ、それぞれの内部に描画を行うことが可能です。ウィンドウの位置は、描画後にユーザが移動させることができます。

画面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。

ただし、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。

名前空間 hg のグローバル関数

```
hg::size hg::screenSize()
```

戻り値　　ディスプレイ装置の幅と高さ（単位は画素）

6.2　ウィンドウを閉じる

クラス Window のインスタンスが破棄される時、デストラクタが呼び出され、同時にウィンドウは自動的にクローズ（画面上から消去）されます。プログラムが終了する時、Window のインスタンスは全て破棄されますので、ウィンドウを逐一クローズする必要はありません。

クラス Window のデストラクタ

```
hg::Window::~Window()
```

ウィンドウを明示的にクローズするためのメンバ関数 close() も用意されています。さらに、static メンバ関数 closeAll() 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。

ただし、ウィンドウをクローズした後、その Window クラスのインスタンスは存在してはいるものの、利用できなくなります。従って、以下の関数よりも、デストラクタを使ってクローズする方法を推奨します。

クラス Window のメンバ関数

```
void close()

static void closeAll()
```

6.3 ウィンドウの大きさを調べる

クラス Window のメンバ関数 getSize() を使って、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。

クラス Window のメンバ関数

```
hg::size getSize() const
```

戻り値　ウィンドウの幅と高さ（単位は画素）

6.4 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。

クラス Window のメンバ関数 setTitle()、または setTitleCText() が利用できますが、setTitleCText() の文字列 `str` 以降は C の標準関数 printf() と同様の書式を使った記述が可能です。

クラス Window のメンバ関数

```
void setTitle(const std::string& str)
```

引数　`str:`　文字列

```
void setTitleCText(const char *str, ...)
```

引数　`str:`　書式文字列

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。

そのような目的のため、ウィンドウごとに座標の原点と縮尺を自由に設定できます。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。

アプリケーション座標の設定には、クラス Window の次のメンバ関数を使います。

クラス Window **のメンバ関数**

```
void setCoordinate(double x, double y, double scale)
```
引数　　x, y:　原点の指定　scale:　縮尺

```
void setCoordinate(hg::point p, double scale)
```
引数　　p:　原点の指定　scale:　縮尺

引数 sx, sy または p で、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

メンバ関数 setCoordinate() 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。

クラス Window **のメンバ関数**

```
void transWtoA(double wx, double wy, double *ax, double *ay) const
```
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

```
hg::point transWtoA(hg::point w) const
```
引数　　w:　ウィンドウ座標での点の座標
戻り値　アプリケーション座標に変換された値

```
void transAtoW(double ax, double ay, double *wx, double *wy) const
```
引数　　ax, ay:　アプリケーション座標での点の座標

wx, wy:　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

```
hg::point transAtoW(hg::point a) const
```
引数　　a:　アプリケーション座標での点の座標
戻り値　　ウィンドウ座標に変換された値

メンバ関数 transWtoA() はウィンドウ座標からアプリケーション座標へ変換し、transAtoW() 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。例えば、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、必要ならばメンバ関数 transWtoA() 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、下記のメンバ関数 setCoordinateEnable() 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、引数のポインタとして NULL を渡すと、その変数に相当する値は計算しません。例えばメンバ関数 transWtoA() 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。

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、次のメンバ関数を使って、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。

クラス Window のメンバ関数

```
void setCoordinateEnable(bool flag)
```
引数　　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、メンバ関数 setCoordinateEnable() の引数 flag に false 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 true 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／無効は座標変換のメンバ関数 transWtoA()、transAtoW() 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 15 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 16 に示します。グラフは、メンバ関数 moveTo()、lineTo() 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、メンバ関数 setCoordinateEnable() 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。

なお、この例題ではラムダ式を利用しています。ある関数の中で設定した値をキャプチャして繰り返し使うような用途の例にもなっています。

```
#include <iostream>
#include <cmath>
#include <handy++>

#define Width  800.0
#define Height 250.0
#define Scale  50.0
#define Step   0.1

int main()
{
    hg::Window win = hg::Window(Width, Height);
    win.setCoordinate(Width / 2, Height / 2, Scale);

    double x1, y1, x2, y2;
    win.transWtoA(0.0, 0.0, &x1, &y1);
    win.transWtoA(Width, Height, &x2, &y2);
    win.line(x1, 0.0, x2, 0.0);
    win.line(0.0, y1, 0.0, y2);

    win.setWidth(2.0);
    win.setColor(HG_BLUE);
    // 与えられた関数の曲線を描くラムダ式
    auto plot = [&win, x1, x2](double (*f)(double)) {
        double x;
        win.moveTo(x1, f(x1));
        for (x = x1 + Step; x < x2; x += Step) {
            win.lineTo(x, f(x));
        }
        win.lineTo(x, f(x));
    };
    plot(sin); // sin(x) の曲線を描く

    win.setColor(HG_RED);
    win.setFont(HG_TI, 20.0);
    win.setCoordinateEnable(false);              // 座標変換を OFF
    win.text(50, 10, "y = sin(2x) + cos(x)/2"); // 文字列を描く
    win.setCoordinateEnable(true);               // 座標変換を ON

    // 引数にラムダ式を使ってみる
    auto func = [](double x) { return sin(2.0 * x) + cos(x) / 2.0; };
    plot(func);

    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 15: アプリケーション座標を使ってグラフを描く (sine.cpp)

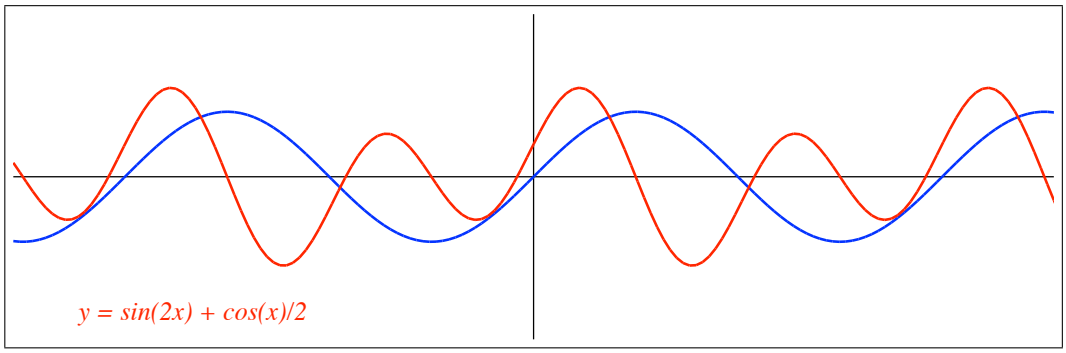


図 16: グラフの描画結果

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、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。

クラス Window のメンバ関数

`int getKeyIn()`

戻り値　`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。getKeyIn() 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。

メンバ関数 getKeyIn() 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クラス Window のメンバ関数 setEventMask() と節 8.3 で示す hg::event のメンバ関数だけを使い、メンバ関数 getKeyIn() 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。

クラス Window のメンバ関数

```
void setEventMask(unsigned int mask)
```

引数　　mask:　イベントマスク

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。

クラス Window のメンバ関数

```
unsigned int getEventMask() const
```

戻り値　　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構造体 hg::event のメンバ関数を使います。

構造体 hg::event のメンバ関数

`int receiveEvent()`

戻り値　`0:` 正常、　`-1:` イベントがない、または引数が不正

この関数は、ウィンドウに対して節 8.2 で示したイベントマスクが設定されていれば、そのイベントが発生するまで待ちます。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 -1 が返されます。

何かイベントがあると 0 が返され、構造体に情報が書き込まれます。

構造体 hg::event は次のように定義されています[3]。

```
struct event {
    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
    int receiveEvent();
    int receiveEventNonBlocking();
};
```

`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`

構造体 hg::event が返す情報の中にウィンドウ id があります。これは表示されているウィンドウを識別するための値で、クラス Window のインスタンスに対して次のメンバ関数を適用すると取得できます。

ウィンドウが複数個表示されている場合、この値を調べることでどのウィンドウに関係するイベントなのかを知ることができます。

[3]ヘッダファイル handy++ における実際の宣言は、C 言語との互換性のために必ずしもこの記述とは一致しません。

クラス Window のメンバ関数

```
int getWindowID() const
```

戻り値　　ウィンドウ id

8.4　ブロックしないイベント取得

上で説明したメンバ関数 receiveEvent() 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、次のメンバ関数 receiveEventNonBlocking() 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。

構造体 hg::event のメンバ関数

```
int receiveEventNonBlocking()
```

戻り値　　0:　イベントがない　　1:　イベントが取得できた　　-1:　エラー、または引数が不正

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て戻り値として 0 が返されます。イベントが取得できた場合は 1 が返され、構造体にイベントの情報が書き込まれます。上記の receiveEvent() とは戻り値の意味が異なりますので注意して下さい。

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。

名前空間 hg のグローバル関数

```
void hg::setIntervalTimer(double t)
```

引数　　t:　時間間隔（秒）

```
void hg::setAlarmTimer(double t)
```

引数　　t:　時間間隔（秒）

関数 setIntervalTimer() 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 setAlarmTimer() 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、sleep() 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、hg::event 構造体のメンバ関数 receiveEvent() または receiveEventNonBlocking() 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。タイマによるイベントの場合、hg::event 型のメンバ type には HG_TIMER_FIRE が設定されます。

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。

◆ イベントを使用する例題

図 17、図 18 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。同時にタイマを利用して、1 秒毎に数字を増やして行く処理もしています。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ていますが、タイマのイベントはこの設定と関係なく取得できます。

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。

```
#include <handy++>

int main()
{
    auto win = hg::Window(500, 500);                // ウィンドウを開く
    int first = 1;      // 最初のクリックを表す
    int sec = 0;        // 秒をカウント
    win.setWidth(2.0);
    win.setFillColor(HG_WHITE);
    win.setFont(HG_CB, 64.0);                       // マウスクリックと
    win.setEventMask(HG_MOUSE_DOWN | HG_KEY_DOWN); //   キー入力を扱う設定
    hg::setIntervalTimer(1.0);                      // タイマをセット
    for ( ; ; ) {
        hg::event event;
        if (event.receiveEvent() != 0) continue; // 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':
                win.clear(); 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:        // q またはエスケープなら終了
                goto EXIT;
            }
        }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
                win.moveTo(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の位置まで線を描く
                win.lineTo(event.x, event.y);
        }else if (event.type == HG_TIMER_FIRE) { // イベントがタイマなら
            win.boxFill(20.0, 0.0, 140.0, 80.0, false);
            win.cText(25.0, 5.0, "%03d", sec++);  // 左下に秒数を描く
        }
    }
EXIT:
    hg::setIntervalTimer(0.0); // タイマをキャンセル
    return 0;
}
```

図 17: マウスクリックとキー入力を使って描画する (plot.cpp)

図 18: 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 19 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 19 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。

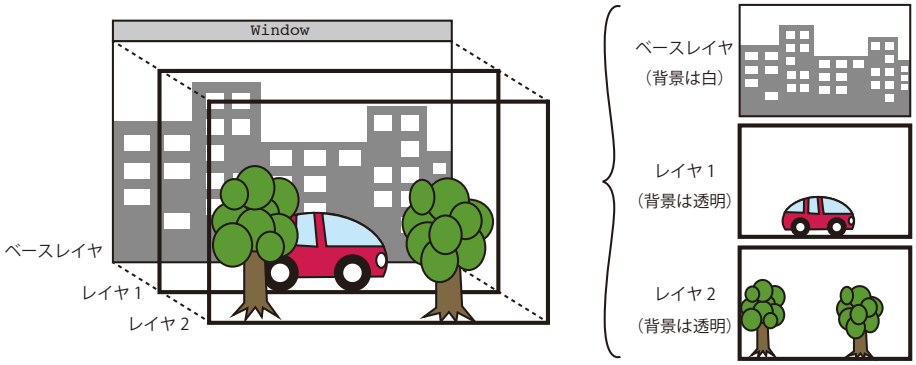


図 19: レイヤの概念図

9.2　レイヤの追加とレイヤ id

クラス Window のインスタンスを生成し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、描画はベースレイヤに対して行われます。

新しくレイヤを追加するには、Window のメンバ関数 addLayer() を使います。

クラス Window のメンバ関数

`hg::Layer addLayer()`

戻り値　　追加されたレイヤ

戻り値は、レイヤを表す**クラス Layer** の新しいインスタンスです。クラス Layer はクラス Window と同様にクラス Canvas を基底クラスとしており、ここまでで説明した様々な描画用、設定用のメンバ関数を利

用することができます。具体的には、色や線の太さ、フォントの設定用を各レイヤに対して行うことができ、折れ線、円、四角形などの描画が可能です。

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、クラス Layer のメンバ関数を利用します。
- 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。ただし、ベースレイヤの順番を入れ替えることはできません。
- 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。ただし、ベースレイヤは非表示にはできません。
- 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。ただし、ベースレイヤは削除できません。また、ウィンドウ自体がクローズされる時、同時にすべてのレイヤが削除されますので個別に削除する必要はありません。

9.3 ベースレイヤの取得

通常、ウィンドウに対する描画はクラス Window のインスタンスに対して行えば十分ですが、ウィンドウのベースレイヤを指定して描画や何らかの操作を行いたい場合があります。そのような場合、Window のインスタンスに対して次のメンバ関数を適用すると、ベースレイヤを表す Layer クラスのインスタンスを取得することができます。

クラス Window のメンバ関数

`hg::Layer baseLayer()`

戻り値　　ベースレイヤ

ベースレイヤは、後から追加されるレイヤとは異なり、削除したり、非表示にしたりできないなど、扱いが異なる点がありますので注意して下さい。

9.4 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。

クラス Canvas（Window, Layer の基底クラス）のメンバ関数

```
void clear()
```

このメンバ関数 clear() は、クラス Layer のインスタンスに対して適用した場合、そのレイヤの描画内容を消去し、透明色で塗りつぶします。

ただし、ウィンドウのベースレイヤに対して適用した場合、ベースレイヤの描画内容を消去し、背景色（白）で塗りつぶします。

ウィンドウの描画を消去するための機能として、クラス Window のメンバ関数 clear() を紹介しました（節 4.6）。クラス Window はクラス Canvas を継承していますが、メンバ関数 clear() の定義は上記のメンバ関数をオーバーライドし、そのウィンドウのすべてのレイヤの内容を一度に消去するという機能を提供します。消去は描画領域を背景色で塗りつぶすことで行います。つまり、ベースレイヤは白色で塗りつぶし、その他のレイヤは透明色で塗りつぶします。

ベースレイヤの内容だけを消去し、他のレイヤを消去したくない場合には、ベースレイヤのインスタンスを取得し、それに対してメンバ関数 clear() を適用します。

9.5 表示／非表示を切り替える

クラス Layer のメンバ関数

```
void show(bool flag)
```

引数　　`flag:`　表示する場合 `true`、しない場合 `false`

クラス Layer のインスタンスの表示／非表示を変更します。ただし、ベースレイヤを非表示に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[4]。

なお、addLayer() 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、クラス Window の次のメンバ関数を使います。

クラス Window のメンバ関数

```
int setVisible(hg::Layer *la, int flag, ...)
```

引数　　`la:`　Layer のインスタンスへのポインタ　`flag:`　`0`（非表示）　または `1`（表示）

戻り値　`0:`　正常、　`-1:`　異常

引数には、Layer のインスタンスへのポインタと、そのレイヤを表示するかどうかを示す整数の組を 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[5]。

例えば、ウィンドウ win のレイヤ 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。

```
win.setVisible(&space, 0, &buf, 1, 0); // 末尾の 0 に注意
```

図 20 にレイヤへの描画と、表示、非表示を切り替える例を示します。前面に描かれているレイヤを非表示にすることで、後ろのレイヤの描画内容が見えるようになります。

[4] 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。
[5] ポインタや整数値を用いる指定（C 言語風）になっているのは、引数を不定個指定できるようにするためです。

```
#include <iostream>
#include <handy++>

int main()
{
    hg::Window win(500, 500, 300, 240);
    hg::Layer lay1 = win.addLayer();  // レイヤを追加
    hg::Layer lay2 = win.addLayer();

    win.setFillColor(HG_BLUE);
    win.circleFill(150, 80, 100);  // ベースレイヤに描画
    lay1.setFillColor(HG_SKYBLUE);
    lay1.circleFill(100, 200, 80); // レイヤ 1 に描画
    lay2.setFillColor(HG_ORANGE);
    lay2.circleFill(200, 150, 60); // レイヤ 2 に描画
    hg::sleep(1);     // 1 秒待つ
    lay1.show(false);              // レイヤ 1 を非表示に
    hg::sleep(1);
    win.setVisible(&lay1, 1, &lay2, 0, 0);
    hg::sleep(1);                  // レイヤ 1 を表示、レイヤ 2 を非表示
    win.baseLayer().clear();       // ベースレイヤを消去
    lay2.show(true);               // レイヤ 2 を表示
    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 20: レイヤへの描画と表示／非表示の例 (lay01.cpp)

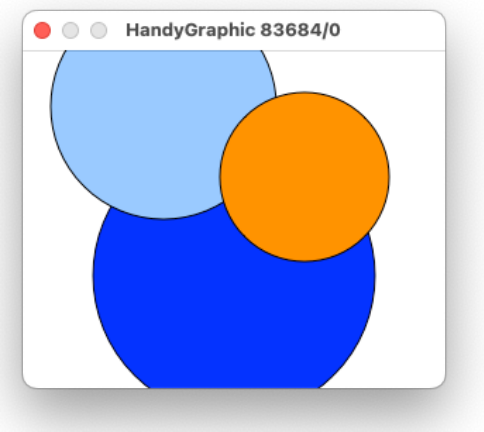


図 21: レイヤへの描画例

9.6 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。このためには、クラス Window のメンバ関数 moveLayer() を使います。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。

クラス Window のメンバ関数

`void moveLayer(const hg::Layer& la, int pos)`

引数　　`la:`　指定したいレイヤ　　`pos:`　新しい位置

引数 pos として、マクロ `HG_LAYER_TOP` または `HG_LAYER_BOTTOM` を指定できます。`HG_LAYER_TOP` を

指定するとそのレイヤが最も手前になり、HG_LAYER_BOTTOM を指定すると最も奥、つまりベースレイヤのすぐ上のレイヤになります。

引数 pos として整数値を指定することもできます。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。

クラス Window のメンバ関数

```
int layers() const
```

戻り値　0 または正整数: レイヤの数、 -1: 異常

9.7 レイヤの内容をコピーする

クラス Layer のメンバ関数

```
void copyTo(hg::Layer& to)
```

引数　to: コピー先のレイヤ

あるレイヤに描画された内容を別のレイヤにコピーするには、コピー先のレイヤを引数に指定して、コピー元のレイヤに対してこのメンバ関数を呼び出します。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。

あるレイヤの内容を別のレイヤに描画するには、クラス Canvas のメンバ関数 imagePut()、imageDraw()、imageDrawRect() 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 copyTo() 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。

ベースレイヤの内容もコピーできますが、レイヤの背景色が問題になることがあります。ベースレイヤとそれ以外のレイヤ間では、コピーではなく、上記の描画関数を使う方法も検討して下さい。

9.8 レイヤを削除する

クラス Window のメンバ関数

```
void removeLayer(hg::Layer& la)
```

引数　la: 対象となるレイヤ

```
void removeLayers()
```

メンバ関数 removeLayer() を使って、指定したレイヤを削除できます。ベースレイヤは削除できません。

メンバ関数 removeLayers() は、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れらの関数を呼び出す必要はありません。

9.9 レイヤを含むウィンドウを知る

クラス Layer のメンバ関数

```
hg::Window window()
```

戻り値　ウィンドウのインスタンス

指定したレイヤを含む Window のインスタンスを返します。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。

9.10 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、次のメンバ関数で指定できます。

クラス Canvas（Window, Layer の基底クラス）のメンバ関数

```
void setComposite(int compo)
```

引数　　compo:　描画方法の指定

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 22 は、節 9.10 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 23 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 <iostream>
#include <handy++>

void draw(hg::Window& win, hg::Layer& lay)
{
    win.setFillColor(HG_BLACK);           // ベースレイヤの下の方を
    win.boxFill(0, 0, 400, 50, 0);        // 帯状に黒く塗る
    lay.setFillColor(HG_ORANGE);          // 不透明のオレンジ色で
    lay.circleFill(150, 50, 135, 0);      // レイヤに円を描く
    lay.setFillColor(HgRGBA(0, 0, 1.0, 0.5));  // 半透明の青で
    lay.boxFill(120, 0, 200, 150, 0);     // 長方形を円に重ねて描く
    lay.setFontByName("YuGo-Bold", 75.0);      // 游ゴシック体
    lay.setColor(HG_CLEAR);               // 100 %の透明色
    lay.text(15, 10, "透明");             // 透明色で文字を描く
    lay.setColor(HgGrayA(0.3, 0.3));      // 半透明の文字を描く
    lay.text(170, 10, "半透明");
}

int main()
{
    hg::Window win1 = hg::Window(200, 300, 400, 200);
    hg::Window win2 = hg::Window(620, 300, 400, 200);
    hg::Layer lay1 = win1.addLayer();  // レイヤを追加
    hg::Layer lay2 = win2.addLayer();
    lay1.setComposite(HG_BLEND_SOVER); // デフォルト値
    draw(win1, lay1);
    lay2.setComposite(HG_BLEND_COPY);  // 後からの描画を優先
    draw(win2, lay2);
    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 22: コンポジットモードの効果を見る (composite.cpp)

図 23: 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 などのビットマップ画像[6]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**クラス Image** のインスタンスとして管理します。クラス Image とクラス Layer は、**Drawable** というクラスを基底クラスとしており、レイヤの描画内容をビットマップ画像として扱うこともできます。具体的には、「クラス Drawable のメンバ関数」と記載されている関数は、ビットマップ画像、レイヤのどちらにも適用可能です。

10.2　画像ファイルを読み込む

クラス Image のコンストラクタ

```
explicit hg::Image(const char *path)
```

引数　　`path:`　画像データを格納したファイルのパス

例外　　ファイルを読めなかった場合（`std::exception` の派生クラス）

指定した画像ファイルからデータを読み込み、クラス Image のインスタンスを生成します。読み込み可能な代表的なデータ形式は、拡張子が jpg、png、gif、heic、webp、tiff、bmp 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。

ファイルが読めなかった場合は例外が発生します。エラーとして処理したい場合には try-catch 構文で捕捉する必要があります（図 24 を参照）。

作成した画像データは、不要になったらデストラクタを適用して解放できます。

クラス Image のデストラクタ

```
hg::Image::~Image()
```

画像データ用に使用されたメモリは、対応するクラス Image のインスタンスが破棄される際に自動的に解放されます。画像データのサイズが大きい場合や数が多い場合には、不要になった時点でこまめに解放した方が良いでしょう。

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。

[6] 主要なデータ形式として、JPEG、PNG、GIF 以外に HEIC, WebP, TIFF, BMP などが利用可能です。PDF は利用できません。

クラス Drawable（Image の基底クラス）のメンバ関数

```
hg::size getSize() const
```

戻り値　　ウィンドウの幅と高さ（単位は画素）

クラス Layer もクラス Drawable を基底クラスとしていますので、クラス Layer に対して getSize() を適用することもできます。その場合はウィンドウの描画部分の大きさが返されます（節 6.3）。

10.4　画像データを複製する

画像を複製 (duplicate) し、新しい Image のインスタンスを返します。

クラス Drawable（Image の基底クラス）のメンバ関数

```
hg::Image dupImage(double scale, int flip = 0)
```

引数　　`scale:`　縮尺　`flip:`　反転の指定

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、第 2 引数に true を指定すると画像を鏡像反転できます。引数を省略するか、0（または `HG_FLIP_NONE`）を指定すると反転しません。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。

クラス Drawable（Image の基底クラス）のメンバ関数

```
hg::Image dupRect(double scale,
                  double x, double y, double w, double h, int flip = 0)
```

引数　　`scale:`　縮尺　`x,y,w,h:`　複製する領域　`flip:`　反転の指定

```
hg::Image dupRect(double scale, hg::rect rect, int flip = 0)
```

引数　　`scale:`　縮尺　`rect:`　複製する領域　`flip:`　反転の指定

引数のうち x, y, w, h または rect で、複製する長方形領域を指定します。対象が画像の場合もレイヤの場合も、左下を原点とし、ピクセル単位で領域を指定します。

◆ 画像の複製を作って表示する例題

図 24 に、読み込んだ画像を指定した大きさに拡大、縮小して表示するプログラムの例を示します。

画像ファイルはコマンドラインから指定できますが、ファイルが読めずに例外が発生した場合、例外を捕捉してメッセージを表示します。メンバ関数 dupImage() を用いて、横または縦の長い方が FixSize になるような複製を作成し、ウィンドウに表示します。画像を表示する関数 imageDraw() は下の節 10.5 を参照して下さい。

```
#include <iostream>
#include <handy++>

const double FixSize = 960.0;        // 画像をこのサイズに合わせる

int main(int argc, char *argv[])
{
    const char *filename = "cat.png";
    hg::Image *src;

    if (argc > 1)
        filename = argv[1];
    try {
        src = new hg::Image(filename);
    }
    catch (const std::exception& e) {        // 例外が発生したら捕捉する
        std::cerr << "ファイルが読めません: " << filename << std::endl;
        return 1;
    }
    hg::size size = src->getSize();
    double scale = FixSize / (size.w > size.h ? size.w : size.h);
    hg::Image img = src->dupImage(scale);    // 大きさを変えて複製
    delete src;                              // 元のデータは不要
    size = img.getSize();
    hg::Window win(size.w, size.h);          // ウィンドウを開く
    win.imageDraw(img, 0.0, 0.0);            // 画像を表示

    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 24: 画像のサイズを変更して表示するプログラム (showimage.cpp)

10.5 画像を描く

クラス Canvas （Window または Layer の基底クラス）には指定された位置に画像を描くメンバ関数があります。

クラス Canvas（Window, Layer の基底クラス）のメンバ関数

```
void imagePut(const Drawable& draw, double x, double y, double scale, double a)
```
引数　　draw:　イメージ　x,y:　描画する位置　scale:　縮尺　a:　傾きの角度

```
void imagePut(const Drawable& draw, hg::point p, double scale, double a)
```
引数　　draw:　イメージ　p:　描画する位置　scale:　縮尺　a:　傾きの角度

指定する座標 (x, y) （または p）は画像の中心点です。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（ラジアン）を指定して回転させることもできます。この様子を図 25(a) に示します。

クラス Canvas（Window, Layer の基底クラス）のメンバ関数

```
void imageDraw(const Drawable& draw, double x, double y)
```
引数　　draw:　イメージ　x,y:　描画位置

```
void imageDraw(const Drawable& draw, hg::point p)
```
引数　　draw:　イメージ　p:　描画位置

メンバ関数 imageDraw() での描画位置の座標 (x, y) （または p）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。また、画像を拡大／縮小、または回転させて描画することはできません（図 25(b)）。

クラス Canvas（Window, Layer の基底クラス）のメンバ関数

```
void imageDrawRect(const Drawable& draw, hg::rect dst, hg::rect src)
```
引数　　draw:　イメージ　dst:　描画先の領域　src:　描画元の領域

```
void imageDrawRect(const Drawable& draw, hg::point p, hg::rect src)
```
引数　　draw:　イメージ　p:　描画位置　src:　描画元の領域

```
void imageDrawRect(const Drawable& draw, hg::rect dst)
```
引数　　draw:　イメージ　dst:　描画先の領域

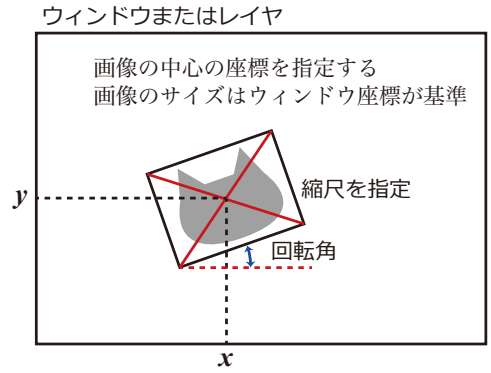
メンバ関数 imageDrawRect() はウィンドウ（またはレイヤ）の長方形領域を指定して、その中に画像を描きます。領域の左下の座標と幅、高さを引数 dst で指定します。この領域の指定はアプリケーション座標に基づきます。

引数 dst の幅と高さが 0.0 の場合、または第 2 引数に hg::point 型の値が与えられた場合、長方形領域の左下の座標のみが有効となり、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。

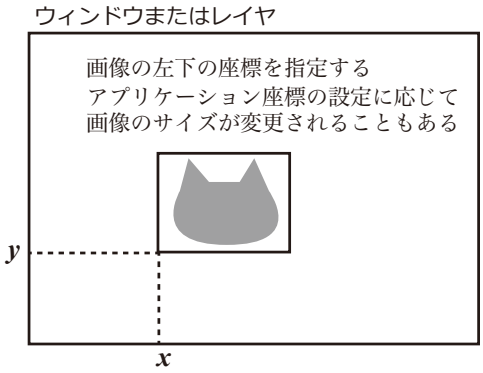
描かれる画像も、引数 src で長方形領域を指定します。引数 src を指定しない場合（または hg::rect のメンバ x, y, w, h がすべて 0.0 の場合）、画像データのすべての範囲が領域と指定されたとみなします。

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 25(c) に示します。

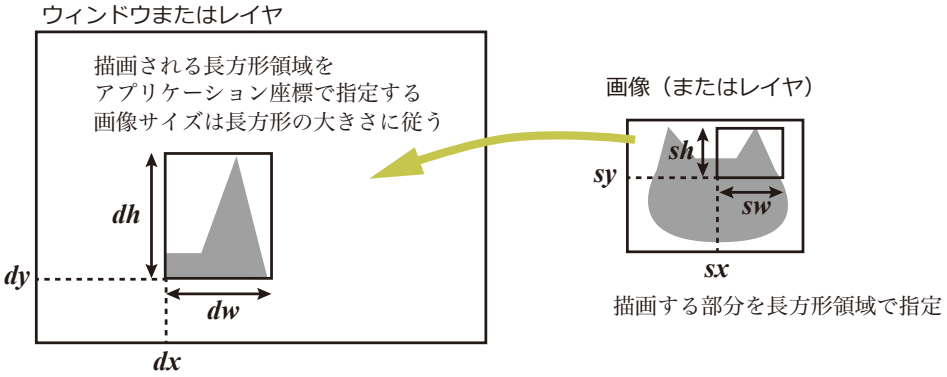
レイヤの内容を丸ごと別のレイヤにコピーする目的には、メンバ関数 copyTo() を利用することもできます（節 9.7）。



(a) HgImagePut() の概要



(b) HgImageDraw() の概要



(c) HgImageDrawRect() の概要

図 25: ビットマップ画像の描画関数の概要

◆ 画像データをいくつかの方法で描画する例題

図 26 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（描画結果は図 27）。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。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) の位置のメンバ関数 imagePut() 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) のメンバ関数 imageDraw() 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) のメンバ関数 imageDrawRect() 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) 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。

```
#include <iostream>
#include <cmath>
#include <handy++>

void draw(hg::Window& win, hg::Image& img)
{
    win.box(0, 100, 300, 100);  // 座標の目安として長方形を描く
    win.box(100, 0, 100, 300);
    win.imagePut(img, 100, 200, 1.0, M_PI/6.0); // (1)
    win.imageDraw(img, 200, 200);                // (2)
    win.imageDrawRect(img, hg::rect{0, 0, 200, 100});
        // (3) 画像を 200 × 100 の長方形の中に描く
    win.imageDrawRect(img, hg::rect{200, 50, 100, 150},
                           hg::rect{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()
{
    hg::Window win1(200, 200, 300, 300);
    hg::Window win2(600, 200, 300, 300);
    auto img = hg::Image("cat.png");
    win2.setCoordinate(20.0, 50.0, 0.75); // アプリケーション座標を設定
    draw(win1, img);  // 2つのウィンドウに同じ操作
    draw(win2, img);

    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 26: 画像データをいくつかの方法で描画する (bitmap.cpp)

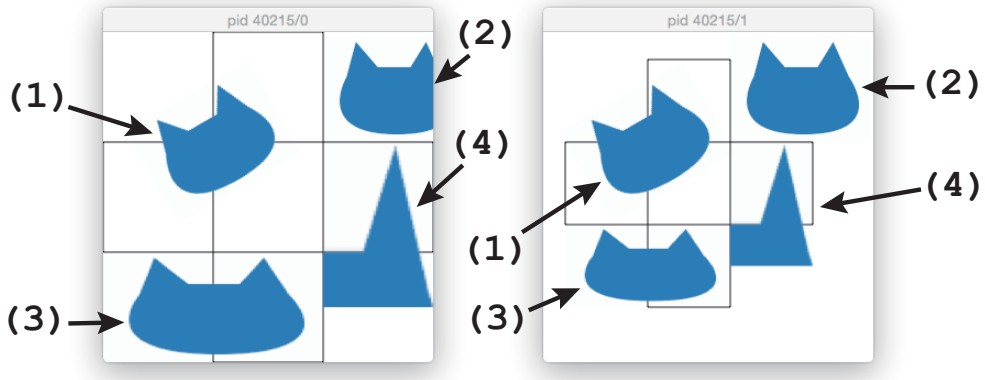


図 27: 画像データを描画した例

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 <iostream>
#include <handy++>

const double radius = 30.0;
const double width = 360.0;
const double height = 180.0;

int main()
{
    hg::Window win(width, height);
    double x = radius;
    for (int count = 0; count < 30; count++) {
        win.setFillColor(HG_WHITE);                    // (1) 塗りつぶし色を白にする
        win.circleFill(x, height/2.0, radius, false); // (2) 現在の位置の表示を消す
        x += radius * 2.0;
        if (x > width) x = radius;
        win.setFillColor(HG_BLUE);
        win.circleFill(x, height/2.0, radius - 1.0, false);
            // (3) 新しい位置に表示する。半径がわずかに小さいことに注意
        hg::sleep(0.3);
    }
    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 28: 簡単なアニメーション (anime01.cpp)

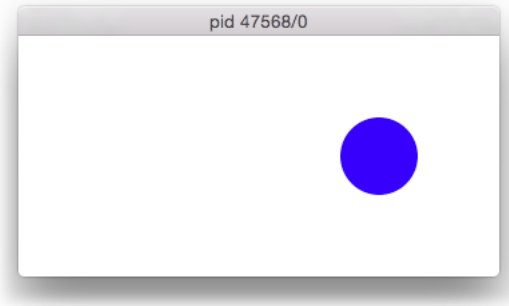


図 29: 簡単なアニメーションの表示例

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 28 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ています（図 28 のコメント (3) に注意）。そのようにしないと、アンチエイリアスによる描画の影響で青い輪郭が残されてしまいます。

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 28 の例なら、コメント (1), (2) の行を関数 `win.clear()` 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 30 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、レイヤに対してメンバ関数 clear() を使用し、全体を消去する方法もあります。

```
#include <iostream>
#include <handy++>

const double radius = 30.0;
const double width = 360.0;
const double height = 180.0;

int main()
{
    hg::Window win(width, height);
    hg::Layer layer1 = win.addLayer();      // アニメーション用レイヤ
    hg::Layer layer2 = win.addLayer();      // 前景用レイヤ

    win.setFillColor(HG_DGRAY);             // 背景をベースレイヤに描く
    for (double v = 20.0; v < width; v += width * 0.2)
        win.boxFill(v, height*0.4, radius*1.2, height*0.6, false);
    layer2.setFillColor(HG_ORANGE);         // 前景を描く
    for (double v = 50.0; v < width; v += width * 0.15)
        layer2.boxFill(v, 0, radius*0.5, height*0.55, false);
    layer1.setComposite(HG_BLEND_COPY);     // 透明色で塗りつぶすため

    double x = radius;
    for (int count = 0; count < 30; count++) { // layer1 でアニメーション
        layer1.setFillColor(HG_CLEAR);          // 透明色を指定
        layer1.circleFill(x, height/2.0, radius, false);     // 消す
        x += radius*2.0;
        if (x > width) x = radius;
        layer1.setFillColor(HG_BLUE);
        layer1.circleFill(x, height/2.0, radius-1.0, false); // 描く
        HgSleep(0.3);
    }

    char ch;
    std::cin.get(ch); // 改行の入力を待つ
    return 0;
}
```

図 30: レイヤを使って描きなおしを減らす例 (anime02.cpp)

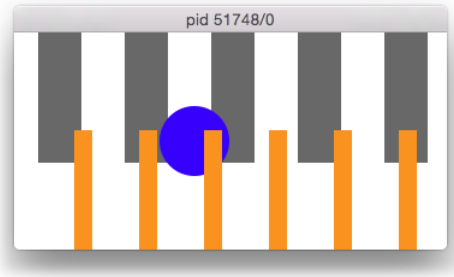


図 31: レイヤを使った描画の例

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、描き直しの過程なども見えてしまうことに原因があります。

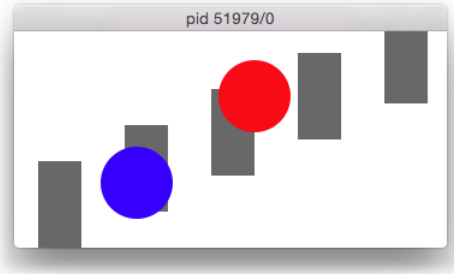


図 32: ２つの円のアニメーションの表示例

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。

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 33、実行例は図 32）。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 34 はダブルバッファリングを使って同様なアニメーションを記述した例です。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 34 のプログラムが図 33 と異なるのは、コメント (1), (2) 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。

11.4 ダブルレイヤのためのクラス

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

```
#include <iostream>
#include <handy++>

const double radius = 30.0;
const double width = 360.0;
const double height = 180.0;

struct Ball {
    double x, y;  // 描画する円の位置
    double dx;    // 円の x 軸方向への速度
    void move() { // １つ分動かす
        x += dx;
        if (x > width) x = 0.0;
    };
};

int main()
{
    double x, y;
    hg::Window win(width, height);
    win.setFillColor(HG_DGRAY);        // 背景をベースレイヤに描く
    for (double v = 20.0; v < width; v += width * 0.2)
    for (x = 20.0, y = 0.0; x < width; x += width * 0.2, y += 30.0)
        win.boxFill(x, y, radius*1.2, height*0.4, false);

    Ball b1 = { 0.0, height * 0.3, 6.0 }; // ２つの円の初期設定
    Ball b2 = { 0.0, height * 0.7, 8.0 };
    hg::Layer layer = win.addLayer();  // アニメーション用レイヤ
    for ( ; ; ) {
        layer.clear();
        layer.setFillColor(HG_BLUE);
        layer.circleFill(b1.x, b1.y, radius, false);
        b1.move();
        layer.setFillColor(HG_RED);
        layer.circleFill(b2.x, b2.y, radius, false);
        b2.move();
        HgSleep(0.02);
    }
    // このプログラムは Control-C で終了させる
    return 0;
}
```

図 33: ダブルバッファリングを使用しない例 (anime03.cpp)

ありません)。

ダブルレイヤを表現するため、**クラス DoubleLayer** を用意しています。ダブルレイヤはペアになった２つのレイヤを簡便に扱うためのものです。DoubleLayer はクラス Layer の派生クラスとして定義されていますが、内部的には２つのレイヤを持ち、描画を行うレイヤと表示に用いるレイヤを切り替えるようになっています。

通常はクラス DoubleLayer の生成にコンストラクタを使う必要はなく、クラス Window の次のメンバ関数を使います。この関数は、新たなレイヤを２つ作成すると同時にクラス DoubleLayer のインスタンスを生成して返します。２つのレイヤのうち、表示されていない描画用レイヤに対して設定や描画が行われます。もう一方のレイヤはウィンドウに表示されていますが、こちらに対しての描画は行われません。

クラス Window **のメンバ関数**

`hg::DoubleLayer addDoubleLayer()`

戻り値　　ダブルレイヤ

DoubleLayer はクラス Layer の派生クラスであり、DoubleLayer のインスタンスに対して色の設定や描

```
#include <iostream>
#include <handy++>

const double radius = 30.0;
const double width = 360.0;
const double height = 180.0;

struct Ball {
    double x, y;  // 描画する円の位置
    double dx;    // 円の x 軸方向への速度
    void move() { // １つ分動かす
        x += dx;
        if (x > width) x = 0.0;
    };
};

int main()
{
    double x, y;
    hg::Window win(width, height);
    win.setFillColor(HG_DGRAY);         // 背景をベースレイヤに描く
    for (x = 20.0, y = 0.0; x < width; x += width * 0.2, y += 30.0)
        win.boxFill(x, y, radius*1.2, height*0.4, false);

    Ball b1 = { 0.0, height * 0.3, 6.0 }; // ２つの円の初期設定
    Ball b2 = { 0.0, height * 0.7, 8.0 };
    hg::DoubleLayer layer = win.addDoubleLayer(); // (1) ダブルレイヤ
    for ( ; ; ) {
        layer.switchLayers();  // (2) 内部の描画レイヤを切り替える
        layer.clear();
        layer.setFillColor(HG_BLUE);
        layer.circleFill(b1.x, b1.y, radius, false);
        b1.move();
        layer.setFillColor(HG_RED);
        layer.circleFill(b2.x, b2.y, radius, false);
        b2.move();
        HgSleep(0.02);
    }
    // このプログラムは Control-C で終了させる
    return 0;
}
```

図 34: ダブルバッファリングを使用する例 (anime04.c)

画などを行うことができます。これらの設定は、その時に表示されている片方のレイヤに対して有効です。

ただし、節 9.10「透明色の塗り方」で述べたコンポジット・モードは２つのレイヤで共通していなければなりません。そこで、基底クラス Canvas のメンバ関数 setComposite() についてはクラス DoubleLayer で再定義をし、２つのレイヤが両方とも同じ設定になるようにしています。

クラス DoubleLayer のメンバ関数

`void setComposite(int compo)`

引数　　`compo:`　描画方法の指定（２つのレイヤに同じ設定）

11.5　ダブルレイヤによるアニメーション

ダブルバッファリングを利用したアニメーションのプログラムを記述するには、２つのレイヤの役割を切り替える操作が必要です。そのためにクラス DoubleLayer の次のメンバ関数を使います。

クラス DoubleLayer のメンバ関数

```
void switchLayers()
```

メンバ関数 switchLayers() を呼び出すたび、２つのレイヤの役割が入れ替わり、それまで表示されずに描画対象だったレイヤが表示され、逆にそれまで表示されていたレイヤが表示されなくなって描画対象になります。

描画対象のレイヤに対してアニメーションの画面をひとつ描き終わったら、メンバ関数 switchLayers() を呼び出して次の画面の描画に移ります。これを繰り返すことで、２つのレイヤの管理を手作業で行うことなく、ダブルバッファリングを利用したアニメーションのプログラムが記述できます。

図 34 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。

```
hg::DoubleLayer layer = win.addDoubleLayer(); // ダブルレイヤを作成する
// ...
while ( 条件 ) {
    layer.switchLayers();  // レイヤを切り替える
    layer.clear();         // レイヤを消去
    //
    // ここでダブルレイヤ layer に対し、通常のレイヤと同様に次の画面を描画する。
    //
    HgSleep(...);  // 必要なら少々待つ（必須ではない）
}
```

ここでは、レイヤを切り替えた後でメンバ関数 clear() を使ってレイヤの内容をすべて消去しています。一方、レイヤの切り替え前まで描画した結果をそのまま残したい場合、レイヤ間で内容を複製する必要があります。そのためにはクラス DoubleLayer の次のメンバ関数を使います。

クラス DoubleLayer のメンバ関数

```
void duplicate()
```

上記の説明において、消去でなく、継続して描画を行うのであれば clear() の代わりに duplicate() を使うだけです。

このような場合の例題プログラムを図 35 に示します（実行例は図 36）。このプログラムでは、任意の位置に円を描きますが、それぞれの円は異なる透明色で複数回重ね書きをして構成しています。

11.6　ダブルレイヤの管理

ダブルレイヤも通常のレイヤと同様に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。このためには、クラス Window のメンバ関数 moveLayer() を使います。

クラス Window のメンバ関数

```
void moveLayer(const hg::DoubleLayer& dla, int pos)
```

引数　　`dla`:　指定したいダブルレイヤ　　`pos`:　新しい位置

引数 pos に マクロ `HG_LAYER_TOP` を指定するとそのダブルレイヤが最も手前になり、`HG_LAYER_BOTTOM` を指定すると最も奥、つまりベースレイヤのすぐ上に移動します。

引数 pos として整数値を指定することもできます（節 9.6 を参照）。ただし、ダブルレイヤは通常のレイヤを２つ利用しているため、整数値で位置を指定すると予期せぬ描画結果になる可能性があります。

ダブルレイヤを使うプログラムでは、ダブルレイヤも通常のレイヤも、上記のマクロで位置を指定することを推奨します。

ダブルレイヤをウィンドウから削除するためのメンバ関数も用意されています。

クラス Window **のメンバ関数**

```
void removeLayer(const hg::DoubleLayer &dla)
```

引数　　`dla`:　対象となるダブルレイヤ

アニメーションの実現には、クラス DoubleLayer を必ず使わなければならないわけではありません。レイヤを利用して、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。

```
#include <unistd.h>
#include <cstdlib>
#include <handy++>

const double width = 400.0;
const double height = 400.0;
const double rad = 20.0;  // 円の最小半径
int steps = 20;           // 円を描くステップ数

int main()
{
    double r = height * 0.3;
    hg::Window win(width, height);
    win.setFillColor(HG_LGRAY);  // 背景をベースレイヤに描く
    win.rectFill(width/2.0, height/2.0, r, r, M_PI/4.0, false);
    srandom(getpid()); // 乱数の初期設定

    hg::DoubleLayer wlay = win.addDoubleLayer(); // ダブルレイヤ
    wlay.setComposite(HG_BLEND_COPY); // この設定は２つのレイヤに共通

    for ( ; ; ) {
        wlay.switchLayers();  // 内部の描画レイヤを切り替える
        wlay.duplicate();     // 直前までの描画結果を引き継ぐ

        double x = random() % (int)width;  // 描画位置と半径を乱数で決める
        double y = random() % (int)height;
        double circle = random() % (int)(rad * 3.0) + rad;
        hg::Color rndcolor((random() & 0xffffffUL) | 0x404040UL);
            // 乱数を使って比較的明るい色を作る（random() は 32 ビットの乱数）
        for (int i = steps; i > 0; i--) {    // 円の外側から内側へ
            double rate = (double)i / steps; // 次第に透明になるように描画
            wlay.setFillColor(hg::Color(rndcolor, rate));
            wlay.circleFill(x, y, circle * rate, false);
        }
        HgSleep(0.25);
    }
    return 0;    // このプログラムは Control-C で終了させる
}
```

図 35: ダブルバッファリングを使用する例 (anime05.cpp)

図 36: プログラム anime05.cpp の表示例

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。

プログラムからウィンドウを指定して、描画内容を保存することもできます。これにはクラス Window のメンバ関数を使います。複数のレイヤを使用している場合、そのレイヤの表示も含め、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。

クラス Window のメンバ関数

```
int save(const char *str)
```

引数	`str:` ファイルのパス
戻り値	`0:` 正常、 `-1:` 異常

また、指定したレイヤについてだけ、描画内容を保存することもできます。

クラス Layer のメンバ関数

```
int save(const char *str)
```

引数	`str:` ファイルのパス
戻り値	`0:` 正常、 `-1:` 異常

ファイルを保存する前にセーブパネルが開き、指定した保存場所とファイル名が表示されます。この時、必要なら拡張子（.png）が付けられます。セーブパネルを使って、指定した以外の保存場所、ファイル名に変更することもできます[7]。

引数 `str` に NULL または空文字列を指定した場合、実行時のカレントディレクトリを保存場所としてセーブパネルが開きますので、ここで保存場所を変更したり、ファイル名を指定し直したりすることができます。

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。

クラス Window に対して保存操作を行った場合、画像の背景は白色になりますが、クラス Layer の内容を保存した場合、背景は透明色（塗りつぶしていない限りは）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。クラス Layer のメンバ関数 PDFRecord() はそのレイヤに対する描画履歴の記録を開始します。メンバ関数 PDFCancel() は履歴の記録をやめて、それまでの情報を破棄します。

クラス Layer のメンバ関数

```
void PDFRecord()


void PDFCancel()
```

[7] これは最近の macOS のセキュリティ機能（サンドボックス）に対応した動作です。

描画履歴はメンバ関数 PDFCancel() を明示的に使用しなくても、メンバ関数 clear() など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。

12.3 PDF ファイルに保存する

クラス Layer のメンバ関数 PDFSave() は、そのレイヤについて記録した描画履歴の内容を 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。

クラス Layer のメンバ関数

```
int PDFSave(const char *str)
```

引数	`str`: ファイルのパス
戻り値	`0`: 正常、 `-1`: 異常

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。

ファイルを保存する前にセーブパネルが開き、指定した保存場所とファイル名が表示されます。この時、必要なら拡張子（.pdf）が付けられます。

引数 `str` に NULL または空文字列を指定した場合、実行時のカレントディレクトリを保存場所としてセーブパネルが開きますので、ここで保存場所を変更したり、ファイル名を指定し直したりすることができます。

一方、クラス Window のメンバ関数 PDFSave() は、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。

クラス Window のメンバ関数

```
int PDFSave(const char *str)
```

引数	`str`: ファイルのパス
戻り値	`0`: 正常、 `-1`: 異常

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 9.10）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 39 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 37(a) は一方のレイヤに描かれた内容だけを保存したもの、図 37(b) はクラス Window のメンバ関数 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。図 38 は図 37(b) の一部を拡大したもので、拡大しても滑らかに表示されていることが分かります。

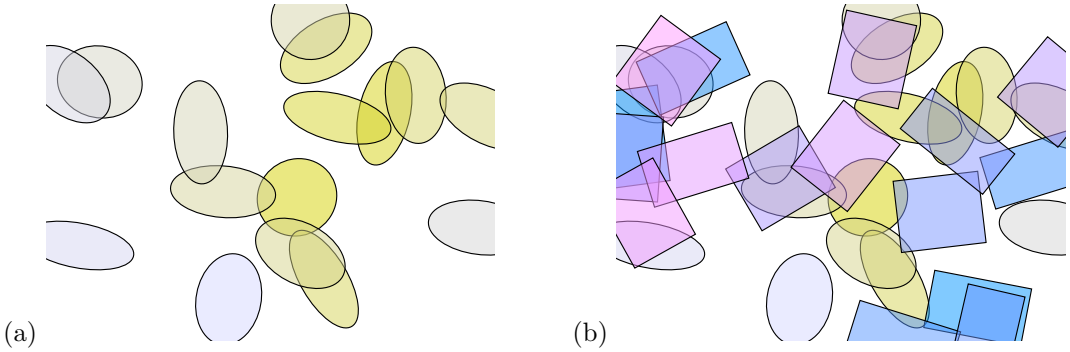


図 37: 保存された PDF ファイル

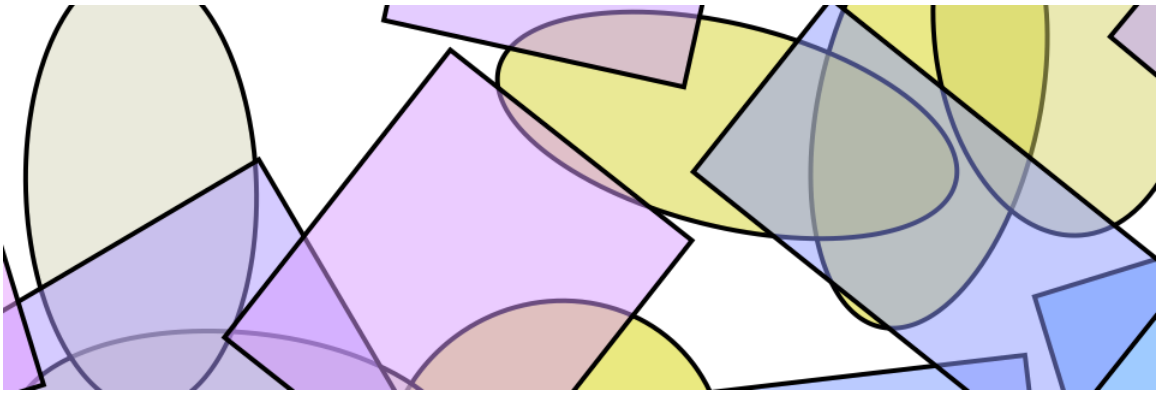


図 38: 保存された PDF ファイルの一部を拡大

なお、メンバ関数 PDFSave() の代わりに save() を使用すると、その時表示されているウィンドウの描画内容、レイヤの描画内容を PNG 形式で保存できます。この場合、描画履歴を記録しておく必要はありません。

```
#include <cstdlib>
#include <cmath>
#include <handy++>

double drnd() { // 0 ≦ x < 1.0 の乱数
    return static_cast<double>(random() % 10000) / 10000.0;
}

void draw(hg::Layer& lay, bool oval)
{
    const int count = 16;
    hg::size sz = lay.getSize();
    for (int i = 0; i < count; i++) {
        hg::point p { drnd() * sz.w, drnd() * sz.h };
        double r = drnd() * 30 + 20.0; // 大きさ
        double a = drnd() * M_PI;      // 回転角
        double t = static_cast<double>(i) / (count - 1);
        if (oval) {
            lay.setFillColor(hg::Color(0.8, 0.8, t, 0.5));
            lay.ovalFill(p, r, 70.0 - r, a); // 楕円を描く
        } else {
            lay.setFillColor(hg::Color(t, 0.5, 1.0, 0.5));
            lay.rectFill(p, r, 70.0 - r, a); // 長方形を描く
        }
    }
}

int main()
{
    srandomdev(); // 乱数の初期化
    hg::Window win(200.0, 200.0, 400.0, 300.0);
    hg::Layer layer1 = win.addLayer(); // レイヤを追加
    hg::Layer layer2 = win.addLayer();
    for ( ; ; ) {
        layer1.PDFRecord();      // 描画履歴の記録を開始
        layer2.PDFRecord();
        draw(layer1, false);     // 長方形を 16 個描く
        draw(layer2, true);      // 楕円を 16 個描く
        int ans = hg::alert("保存しますか？", "保存", "再描画");
        if (ans == 0) {          // 保存
            win.PDFSave("pdf-all.pdf");
            layer1.PDFSave("pdf-1.pdf");
            break;
        }
        win.clear();        // 描画履歴はリセットされる
    }
    return 0;
}
```

図 39: レイヤに描いた内容を PDF に保存するプログラム (pdftest.cpp)

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、**クラス Sound** のインスタンスで管理します。このインスタンス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

クラス Sound のコンストラクタ

```
explicit hg::Sound(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

例外　　ファイルを読めなかった場合 (`std::exception` の派生クラス)

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。

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。

クラス Sound の静的メンバ関数

```
static hg::Sound name(const char *name)
```

引数　　`name:`　効果音の名前

例外　　ファイルを読めなかった場合 (`std::exception` の派生クラス)

効果音に対応するファイルは /System/Library/Sounds にあります[8]。拡張子は不要で、例えば “Funk” を引数に指定できます。

自分で作成した効果音を ~/Library/Sounds/ に追加して使うこともできます。

なお、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」でも確認できますが、この一覧に表示される名称は必ずしも正確ではありません[9]。

上記のコンストラクタでは、対応するファイルが読めなかった場合は例外が発生します。エラーとして処理したい場合、try-catch 構文で捕捉する必要があります。

作成した Sound のインスタンスは、不要になったらデストラクタを適用して解放できます。

クラス Sound のデストラクタ

```
hg::Sound::~Sound()
```

サウンドデータ用に使用されたメモリは、対応するクラス Sound のインスタンスが破棄される際に自動的に解放されます。サウンドデータのサイズが大きい場合や数が多い場合には、不要になった時点でこまめに解放した方が良いでしょう。例えば、ゲームの開始時にしか使わないサウンドは再生終了とともに解放すると良いでしょう。

[8] 例えば macOS 15.6 (Sequoia) では、name() の引数として Basso, Bottle, Funk, Hero, Ping, Purr, Submarine, Blow, Frog, Glass, Morse, Pop, Sosumi, Tink が利用できます。

[9] 過去のインタフェースとの整合性、OS のバージョンや利用言語などが理由です。

13.3 音量と繰り返しの設定

クラス Sound のメンバ関数

```
void setVolume(double vol, bool loop = false)
```

引数　　vol:　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、$0.0 \leq vol \leq 1.0$ 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。第 2 引数を省略した場合は繰り返しません。

13.4 再生と停止

クラス Sound のメンバ関数

```
void play()

void stop()
```

メンバ関数 play() はサウンドの再生を開始します。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、メンバ関数 stop() を使います。

13.5 一時停止と再開

クラス Sound のメンバ関数

```
void pause()

void resume()
```

メンバ関数 pause() は、サウンドの再生を一時停止させます。メンバ関数 resume() 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 ビープ音

クラス Sound の静的メンバ関数

```
static void beep()
```

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。

◆ 音楽を再生しながらドラム音を鳴らせるプログラム

図 40 に、サウンドを再生するプログラムの例を示します。デフォルトの音楽以外に、再生するサウンドファイルをコマンドラインから指定できます。

プログラムを動作させると何も表示されていない小さなウィンドウが表示されますが、キーボードから ‘p’ を入力すると音楽の再生が開始されます。‘d’ を入力するとドラムの音が鳴りますので、音楽に合わせて簡単な演奏ができます。音楽の再生は停止しませんが、 ‘q’ を入力するとプログラムは停止します。

なお、Bluetooth のイヤホンやスピーカで再生している場合、キーの押下よりも音が若干遅れて聞こえることがあります。

```
#include <iostream>
#include <handy++>
// Radetzky Marsch : https://www.palaceconcertsvienna.com

const double wid = 300.0, hgt = 200.0;
static hg::Sound *drum = nullptr;

void beat(hg::Window& win) {
    drum->play();                   // ドラムを鳴らして
    win.setFillColor(HG_SKYBLUE);
    win.boxFill(0, 0, wid, hgt);
    hg::sleep(0.05);
    win.setFillColor(HG_WHITE);     // 一瞬ウィンドウの色が変わる
    win.boxFill(0, 0, wid, hgt);
}

int main(int argc, char *argv[])
{
    hg::Sound *music = nullptr;
    const char *sndfile = "../sound/RadetzkyMarsch.mp3";
    if (argc > 1)
        sndfile = argv[1];
    try {
        drum = new hg::Sound("../sound/drum.mp3"); // ドラムの音
        music = new hg::Sound(sndfile);            // 音楽
    }
    catch (...) {
        std::cerr << "ファイルが読めません" << std::endl;
        return 1;
    }
    music->setVolume(0.8, true);    // 音楽の音量。ループ再生を指定
    drum->setVolume(0.8);           // ドラムの音量
    bool playing = false;           // 再生中か停止中か
    bool music_on = false;          // 音楽は開始されたか
    auto win = hg::Window(wid, hgt);
    win.setEventMask(HG_KEY_DOWN);  // キー入力を扱う設定
    for ( ; ; ) {
        hg::event event;
        if (event.receiveEvent() != 0) continue; // イベントを取得
        if (event.type != HG_KEY_DOWN) continue; // イベントがキー入力?
        switch (event.ch) {
        case 'p': // 音楽の再生開始 or 一時停止
            if (music_on) {
                if (playing)
                    music->pause();
                else
                    music->resume();
                playing = !playing;
            }else {
                music->play(); // 最初の１回のみ
                playing = true;
                music_on = true;
            }
            break;
        case 'd': // ドラムを１回鳴らす
            beat(win); break;
        case 'q': // 終了
            goto EXIT;
        default:  // 他のキーは無視する
            break;
        }
    }
EXIT:
    return 0;
}
```

図 40: 音楽とドラム音を鳴らすプログラム (drum.cpp)

付録　関数の一覧

以下の説明において関数の記法は簡便なものであり、言語の文法とは異なります。

引数がポインタ型であることを表すために `*x` のように `*` を引数名の前に、参照型であることを表すために `str&` のように `&` を引数名の後ろに付けることがあります。

また、表では関数の戻り値の型、名前空間 hg の記述を省略しています。

「関数」の欄の記号は次の意味です。記号がない場合はメンバ関数を意味します。

- Ⓒ　コンストラクタ
- Ⓓ　デストラクタ
- Ⓢ　静的メンバ関数
- Ⓖ　グローバル関数

仮引数の型について特に説明がない場合、仮引数名が役割と型を表します。以下に、仮引数名の型、多くの場合の役割を示します。下線の付いた引数は省略可能です。ただし、これらは便宜的なものに過ぎませんので、詳細についてはそれぞれに関する説明を参照して下さい。

`x, y`	`double`	座標位置
`w, h`	`double`	幅、高さ
`r, a`	`double`	半径、角度
`str`	`std::string`	C++の文字列
`cstr`	`const char *`	C 言語の文字列
`fn`	`const char *`	ファイル名
`flag`	`bool`	ON / OFF
`rect`	`hg::rect`	長方形領域
`p`	`hg::point`	座標位置
`sz`	`hg::size`	幅と高さ

ウィンドウ

機能・役割	関数		関連するクラス	参照
ウィンドウを作成	`Window(w,h)`	Ⓒ	`Window`	3.1,6.1
	`Window(x,y,w,h)`	Ⓒ	`Window`	3.1, 6.1
	`Window(rect)`	Ⓒ	`Window`	3.1, 6.1
ウィンドウを閉じる	`~Window()`	Ⓓ	`Window`	6.2
	`close()`		`Window`	6.2
全ウィンドウを閉じる	`closeAll()`	Ⓢ	`Window`	6.2
ウィンドウタイトル	`setTitle(str&)`		`Window`	6.4
ウィンドウタイトル	`setTitleCText(cstr,...)`		`Window`	6.4
ウィンドウの大きさを得る	`getSize()`		`Window`	6.3
画面の大きさを得る	`screenSize()`	Ⓖ		6.1

図形の描画

機能・役割	関数	関連するクラス	参照
２点間に線分を描く	line(x0,y0,x1,y1)	Canvas	4.1
	line(p0,p1)	Canvas	4.1
カレントポイントを移動	moveTo(x,y)	Canvas	5.1
	moveTo(p)	Canvas	5.1
指定点まで線分を描く	lineTo(x,y)	Canvas	5.1
	lineTo(p)	Canvas	5.1
円を描く	circle(x,y,r)	Canvas	4.2
	circle(p,r)	Canvas	4.2
円を塗りつぶす	circleFill(x,y,r,strk)	Canvas	4.2
	circleFill(p,r,strk)	Canvas	4.2
楕円を描く	oval(x,y,r1,r2,a)	Canvas	5.4
	oval(p,r1,r2,a)	Canvas	5.4
楕円を塗りつぶす	ovalFill(x,y,r1,r2,a,strk)	Canvas	5.4
	ovalFill(p,r1,r2,a,strk)	Canvas	5.4
円弧を描く	arc(x,y,r,a0,a1)	Canvas	5.3
	arc(p,r,a0,a1)	Canvas	5.3
扇型を描く	fan(x,y,r,a0,a1)	Canvas	5.3
	fan(p,r,a0,a1)	Canvas	5.3
扇型を塗りつぶす	fanFill(x,y,r,a0,a1,strk)	Canvas	5.3
	fanFill(p,r,a0,a1,strk)	Canvas	5.3
長方形を描く	box(x,y,w,h)	Canvas	4.3
	box(p,sz)	Canvas	4.3
	box(rect)	Canvas	4.3
	rect(x,y,r1,r2,a)	Canvas	5.4
	rect(p,r1,r2,a)	Canvas	5.4
長方形を塗りつぶす	boxFill(x,y,w,h,strk)	Canvas	4.3
	boxFill(p,w,h,strk)	Canvas	4.3
	boxFill(rect,strk)	Canvas	4.3
	rectFill(x,y,r1,r2,a,strk)	Canvas	5.4
	rectFill(p,r1,r2,a,strk)	Canvas	5.4
折れ線を描く	lines(n,*x,*y)	Canvas	5.2
	lines(n,*p)	Canvas	5.2
多角形を描く	plygon(n,*x,*y)	Canvas	5.2
	plygon(n,*p)	Canvas	5.2
多角形を塗りつぶす	plygonFill(n,*x,*y,strk)	Canvas	5.2
	plygonFill(n,*p,strk)	Canvas	5.2

注:　引数 strk は bool 型で、図形の輪郭を描くかどうかを表す。引数 n は int 型で点の個数を表す。

文字列を描く

機能・役割	関数	関連するクラス	参照
文字列を描く	text(x,y,str&)	Canvas	4.4
	text(p,str&)	Canvas	4.4
	cText(x,y,cstr,...)	Canvas	4.4
	cText(p,cstr,...)	Canvas	4.4
文字列のサイズ	getTextSize(str&)	Canvas	5.7
	getCTextSize(cstr,...)	Canvas	5.7

描画領域の消去

機能・役割	関数	関連するクラス	参照
ウィンドウ消去	clear()	Window	4.6
レイヤ消去	clear()	Layer	9.4

色を作る

機能・役割	関数		関連するクラス	参照
色を RGB で作る	Color(red,green,blue)	Ⓒ	Color	4.5
	Color(red,green,blue,alpha)	Ⓒ	Color	5.5
灰色を作る	Color(gray)	Ⓒ	Color	4.5
	Color(gray,alpha)	Ⓒ	Color	5.5
カラーコード	Color(hex)	Ⓒ	Color	4.5
アルファ値を指定	Color(col,alpha)	Ⓒ	Color	5.5

注: 引数 red, green, blue, alpha, gray はいずれも double 型。引数 hex は unsigned long 型。引数 col は hg::Color 型。

線の太さ、色の指定

機能・役割	関数	関連するクラス	参照
線の太さを指定	setWidth(width)	Canvas	3.2
線の色を指定	setColor(col)	Canvas	3.3
塗りつぶし色を指定	setFillColor(col)	Canvas	3.3
透明色の塗り方	setComposite(compo)	Canvas	9.10
	setComposite(compo)	DoubleLayer	11.4

注: 引数 width は double 型。引数 col は hg::Color 型。引数 compo は int 型（マクロで指定）。

フォントの指定

機能・役割	関数	関連するクラス	参照
フォントを指定	setFont(font,size)	Canvas	3.4
	setFontByName(cstr,size)	Canvas	5.6

注: 引数 font は int 型（マクロで指定）。引数 size は double 型。

アプリケーション座標

機能・役割	関数	関連するクラス	参照
アプリケーション座標を設定	setCoordinate(x,y,scale)	Window	7.1
	setCoordinate(p,scale)	Window	7.1
座標変換 ウィンドウ座標→アプリケーション座標	transWtoA(x0,y0,*x1,*y1)	Window	7.2
	transWtoA(p)	Window	7.2
座標変換 アプリケーション座標→ウィンドウ座標	transAtoW(x0,y0,*x1,*y1)	Window	7.2
	transAtoW(p)	Window	7.2
有効／無効の切替	setCoordinateEnable(flag)	Window	7.3

注: 引数 scale は double 型で縮尺を表す。

イベント

機能・役割	関数	関連するクラス	参照
キー入力を得る	getKeyIn()	Window	8.1
イベントマスクを設定	setEventMask(mask)	Window	8.2
イベントマスクを得る	getEventMask()	Window	8.2
イベントを取得	receiveEvent()	event	8.3
	receiveEventNonBlocking()	event	8.4
タイマを設定	setIntervalTimer(t) Ⓖ		8.5
	setAlarmTimer(t) Ⓖ		8.5
ウィンドウ id を得る	getWindowID()	Window	8.3

注: 引数 mask は unsigned int 型。引数 t は double 型。

レイヤ

機能・役割	関数	関連するクラス	参照
レイヤを追加	addLayer()	Window	9.2
ベースレイヤを得る	baseLayer()	Window	9.3
レイヤの描画を消去	clear()	Layer	9.4
表示／非表示を切り替える	show(flag)	Layer	9.5
	setVisible(*lay,flag,...)	Window	9.5
レイヤの順序を変更	moveLayer(lay&,n)	Window	9.6
レイヤをコピーする	copyTo(lay&)	Layer	9.7
レイヤを削除	removeLayer(lay&)	Window	9.8
レイヤをすべて削除	removeLayers()	Window	9.8
レイヤを含むウィンドウ	window()	Layer	9.9
レイヤの枚数	layers()	Window	9.6
ダブルレイヤを作る	addDoubleLayer()	Window	11.4
ダブルレイヤを切り替える	switchLayers()	DoubleLayer	11.5
ダブルレイヤの間で内容を複製	duplicate()	DoubleLayer	11.5
ダブルレイヤの順序を変更	moveLayer(wlay&,n)	Window	11.6
ダブルレイヤを削除	removeLayer(wlay&)	Window	11.6

注: 引数 lay は Layer 型。引数 wlay は DoubleLayer 型。

ビットマップ画像

機能・役割	関数		関連するクラス	参照
画像を読み込む	Image(fn)	Ⓒ	Image	10.2
画像を解放する	~Image()	Ⓓ	Image	10.2
画像サイズを得る	getSize()		Drawable	10.3
画像を複製する	dupImage(scale,flip)		Drawable	10.4
	dupRect(scale,x,y,w,h,flip)		Drawable	10.4
	dupRect(scale,rect,flip)		Drawable	10.4
画像を描く	imagePut(draw,x,y,scale,a)		Canvas	10.5
	imagePut(draw,p,scale,a)		Canvas	10.5
	imageDraw(draw,x,y)		Canvas	10.5
	imageDraw(draw,p)		Canvas	10.5
	imageDrawRect(draw,rect1,rect2)		Canvas	10.5
	imageDrawRect(draw,rect)		Canvas	10.5
	imageDrawRect(draw,p,rect)		Canvas	10.5

注: 引数 scale は double 型で倍率を表す。引数 flip は int 型で、画像の反転を指定する。引数 draw は Drawable 型（Image、または Layer）。

描画結果の保存

機能・役割	関数	関連するクラス	参照
ビットマップ画像を保存	save(fn)	Window	12.1
	save(fn)	Layer	12.1
描画履歴を記録	PDFRecord()	Layer	12.2
描画記録を破棄	PDFCancel()	Layer	12.2
レイヤを PDF で保存	PDFSave(fn)	Layer	12.3
複数レイヤを PDF で保存	PDFSave(fn)	Window	12.3

サウンド

機能・役割	関数		関連するクラス	参照
サウンドデータを読む	Sound(fn)	Ⓒ	Sound	13.2
	name(cstr)	Ⓢ	Sound	13.2
サウンドデータの解放	~Sound()	Ⓓ	Sound	13.2
音量と繰り返しの設定	setVolume(vol,loop)		Sound	13.3
再生	play()		Sound	13.4
停止	stop()		Sound	13.4
一時停止	pause()		Sound	13.5
再開	resume()		Sound	13.5
ビープ音	beep()	Ⓢ	Sound	13.6

注: 引数 vol は double 型で音量を表す。引数 loop は bool 型で繰り返し再生の有無を指定する。

その他

機能・役割	関数		関連するクラス	参照
point 型を作る	point(x,y)	Ⓒ	hg::point	4.1
size 型を作る	size(w,h)	Ⓒ	hg::size	4.3
rect 型を作る	rect(x,y,w,h)	Ⓒ	hg::rect	4.3
	rect(p,sz)	Ⓒ	hg::rect	4.3
指定した時間だけ待つ	sleep(t)	Ⓖ		5.9
パネルを表示	alert(str&,str0&,str1&,str2&)	Ⓖ		5.8
	alertCText(cstr,cstr0,cstr1,cstr2,...)	Ⓖ		5.8

目 次