ゲーム開発のための SDL プログラミング

ユーザインタフェイスを中心として

徳島大学 理工学部 理工学科 情報光システムコース・情報系

光原 弘幸

2019 年 5 月

目次

第 1 章

はじめに

1.1　ソフトウェア設計および実験

ソフトウェア設計および実験（以下，ソフト実験）は，徳島大学理工学部理工学科情報光システムコース・情報系（旧：徳島大学工学部知能情報工学科）の 2 年時に開講されている実験科目（通年・必修）であり，デジタルゲーム（以下，ゲーム）の開発をテーマとしている．ゲームは現代社会において欠かせない“娯楽（エンタテインメント）”のひとつであり，娯楽を超えて世界共通の“文化”として扱われることもある．

日常的にゲームを楽しんでいる学生は多いだろう．そして，「どのようにゲームを創るのだろう？」という関心や「ゲームを創ってみたい！」という意欲をもっている学生も多いはずである．このような背景もあって，ソフト実験は 2012 年度から，前期は個人によるゲーム開発，後期はグループによるネットワーク対戦型ゲーム開発という構成になり，本コースの情報教育を代表する授業になっている[*1]．

このソフト実験で扱う内容は多岐にわたる．これは，テーマとするゲーム開発のために学ぶべきことが多岐にわたるからであり，ソフト実験が学生に大きなやり甲斐を提供することを意味している．例えば，キーボード・マウスやその他の多様なデバイスを取り入れたり，アニメーションや 3 次元 CG の処理に凝ってみたり，多人数がストレスなく同時参加できるネットワーク通信を実現したり，と挙げれば切りがない．ゲームは総合芸術とも言われるが，よりよいゲームを完成させるには，プログラミングだけでなく，企画，グラフィックスやサウンド，開発の進捗管理なども重要となってくる．ゲーム開発では，乗り越えなければならないさまざまな問題・課題に直面する．ソフト実験でも，問題・課題を根気強く着実に乗り越えていくことが求められる．ソフト実験に合格すれば，ゲームを含めたソフトウェア開発に関する知識・スキルを獲得することはもちろん，「思い描いたゲームを開発できた」や「グループで協力して開発をやり遂げた」など，今後の人生にプラスになる自信をもてるようになるだろう[*2]．

近年，ゲーム開発の効率化・高機能化を実現するゲームエンジン（統合開発環境）の登場により，ゲーム開発がより身近なものになりつつある．例えば，普及しているゲームエンジンとして，Unity[*3]や Unreal Engine[*4]が挙げられる．「ゲームエンジンを使ってゲームを開発したい」と思うかもしれないが，ソフト実験ではゲームエンジンは使わない．ソフト実験は，1 年時（プログラミング入門）で学んだ C 言語に関する実践的かつ発展的な授業に位置づけられ，“ゲームエンジンに頼らず 1 から開発してもらうことで，実力をつけて

[*1] 以前は，前期に RoboCup サッカー（ソフトウェアエージェント開発），後期にゲーム開発を実施していた．
[*2] 実際，有名ゲームメーカに就職し活躍している先輩が複数人いる．
[*3] https://unity3d.com/jp
[*4] https://www.unrealengine.com/ja/what-is-unreal-engine-4

もらう”ことを目的としているからである．ゲーム開発には基本的に C 言語を使用することになるが，ゲームのすべてを 1 から C 言語だけで開発することには多くの困難が伴う．そこで，ソフト実験では，ゲームエンジンほどではないが，開発の効率化・高機能化につながるライブラリを使用することとする．

1.2 SDL

ソフト実験で使用する主要なライブラリのひとつが，SDL（Simple DirectMedia Layer）[*5]である．SDL は，Sam Lantinga 氏（当時，アメリカのゲーム開発会社 Blizzard Entertainment の Lead Software Engineer）らによって開発された，マルチメディアを簡単かつ直接に扱うためのオープンソースライブラリ（zlib License[*6]）である．グラフィックス, サウンド, ジョイスティック, スレッド, タイマなどのゲーム開発に欠かせなマルチメディア処理に力を発揮することから，SDL を使用して開発されたゲームも多い[*7]．さらに SDL は，Linux はもちろん Windows や MacOS など主要なオペレーティングシステム（OS）で使用可能つまりマルチプラットフォームであるとともに，C 言語だけでなく C++，Java や Python といった複数のプログラミング言語にバインドされており，スマートフォン／タブレットアプリ（Android，iOS）の開発にも対応する．できるだけ簡潔にプログラムしたい画面描画等（ハードウェアに依存する部分が多い）に関して，SDL は OS の描画ライブラリ（API : Application Programming Interface）の上位インタフェイスとして提供される（図 1.1）．Linux であれば基本的には Xlib，Windows であれば DirectX の上位インタフェイスとして提供される．

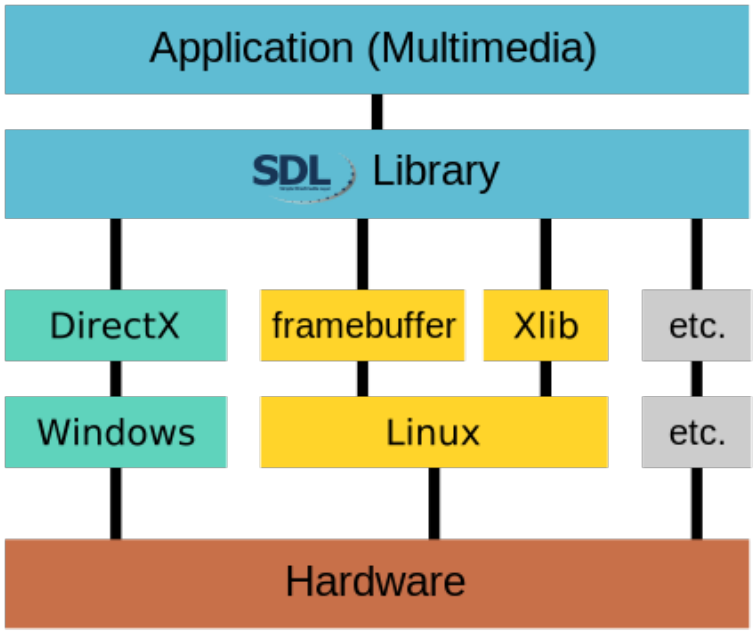

図 1.1　SDL の構造（https://ja.wikipedia.org/wiki/SDL#/media/File:SDL_Layers.svg）

[*5] https://www.libsdl.org/
[*6] http://zlib.net/zlib_license.html
[*7] https://en.wikipedia.org/wiki/List_of_games_using_SDL

1.2.1 SDL1.2 から 2.0 へ

SDL は 1998 年に誕生し，バージョン 1.2（SDL1.2）がメジャーになった後，大きな改良により 2019 年 4 月現在のバージョンは 2.0.9（安定版）になっている．このバージョン 2.0 系は“SDL2”と呼ばれる．

SDL を使用すれば，2 次元ゲームはもちろん，3 次元 CG 描画ライブラリである OpenGL[*8]を活用して 3 次元ゲームも開発できる．SDL2 は，SDL1.2 を引き継ぎながら，以下のようなさまざまな改良・追加がほどこされている．

- 描画の高速化（3 次元ハードウェアアクセラレーションなど），OpenGL の拡張への対応
- 32bit オーディオ，複数のオーディオデバイス，録音への対応
- 複数ウィンドウ，非矩形ウィンドウ，メッセージボックス，ディスプレイ解像度フルスクリーン，マルチディスプレイへの対応
- 垂直同期，クリップボードへの対応
- Android，iOS，マルチタッチやジェスチャへの対応
- 力覚フィードバック，テキスト入力

SDL2 は SDL1.2 を引き継いではいるが，構造体や関数などいくつかの仕様が変更されたり削除されたりしているため，SDL1.2 用のプログラムソースを単に SDL2 にリンクさせてコンパイルしても失敗するだろう．つまり，SDL2 で定められた構造体や関数などの仕様に従ってプログラミングする必要がある．SDL1.2 と SDL2 でのウィンドウ生成・表示のプログラムの一部を示す．

SDL1.2

```
SDL_WM_SetCaption("Test", "Software Exp"); // ウィンドウのタイトルなどを設定
SDL_Surface* window = SDL_SetVideoMode(640, 480, 32, SDL_SWSURFACE); // ウィンドウの
生成・表示
```

SDL2

```
SDL_Window* window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, 0); // ウィンドウの作成・表示
```

ウィンドウの生成・表示について，SDL1.2 では **SDL_Surface** 構造体，**SDL_SetVideoMode** 関数（そして，この関数に付随する **SDL_WM_SetCaption** 関数）が使用される．一方，SDL2 では **SDL_SetVideoMode** 関数は廃止され，**SDL_Window** 構造体と **SDL_CreateWindow** 関数が使用される．ゲームに限らず多くのソフトウェアはウィンドウの存在を前提とすることから，SDL2 ではこの前提に関する記述が大きく変わったことになる．

SDL2 の **SDL_CreateWindow** 関数はハードウェアレンダリングという描画の高速化やディスプレイ解像度への自動対応を実現し，描画内容の拡大縮小も容易にしていることから，特にゲーム開発に恩恵をもたらす．したがって，ソフト実験では SDL2 を使うことにする[*9]．

[*8] https://www.opengl.org/

[*9] 2018 年度までは SDL1.2 と SDL2 の両方を扱っていたが，2019 年度からは SDL2 のみを扱う．

1.3　本教材の目的

本教材の目的は，C 言語の基礎を習得した学生を対象として，ソフト実験におけるゲーム開発で大きなウェイトを占めるユーザインタフェイス（User Interface．以下，UI）に焦点を当て，より良い UI を実現する SDL プログラミング（SDL2）を習得してもらうことである．ソフト実験では，前期からゲームを個人開発しなければならないことからもわかるように，“受動的”ではなく“能動的”に学習しながら開発に取り組むことが求められる．残念ながら，この教材は“SDL2 のすべてが分かる”や“手取り足取り教える”を標榜しておらず，“かゆいところに手が届く”という絶妙な内容も備えていないが[*10]，能動的な学習・開発の初期において多少の助けになることを期待する．

SDL2 には多くの構造体や関数が存在することから，それらすべてを理解するには多くの時間が必要となる．よって，本教材は網羅的な内容ではなく，ゲーム開発で重要となるトピック（場面）に的を絞って，それらトピックを実現する構造体や関数の使い方を概説する形を採用している．例えば，「ウィンドウを生成・表示する」，「画像を描画する」，「タイマ割込を使う」といったトピックごとに，構造体や関数の仕様，サンプルプログラムの一部（構造体や関数の記述例）を示すことにしている．サンプルプログラムは，ソフト実験の授業資料などをダウンロードできる manaba[*11]に公開しているので，本教材と合わせて参考になれば幸いである．

なお，本教材は下記の SDL に関する Web サイトを参考に作成している．

- SDL2.0 日本語リファレンスマニュアル （http://sdl2referencejp.osdn.jp/index.html）
- Simple DirectMedia Layer - Homepage （https://www.libsdl.org/）
- FrontPage - SDL Wiki’ （https://wiki.libsdl.org/FrontPage）

よって，本教材で説明できなかった内容や詳細については，これらの Web サイトなどにアクセスして学習・確認してほしい．

[*10] 内容の不十分さは，教材作成に時間をかけられなかったことが要因のひとつといえる・・・orz
[*11] https://manaba.lms.tokushima-u.ac.jp/

第 2 章

SDL プログラミング

ソフト実験では，C 言語から SDL2 を呼び出すという形を採用してゲームを開発する．本教材では以降，説明文内の SDL は SDL2 を意味する．1 年時に学んだ C 言語プログラミングの復習も兼ねて，実際に手を動かしながら（サンプルプログラムを実行し，そのソースを改良しながら）理解につなげてほしい．

2.1 事始め

SDL プログラミングを始めるにあたり，覚えておくべきことがいくつかある．

2.1.1 プリプロセッサを指定する

SDL を使用するには，まず，ソースコード（*.c）の冒頭で SDL に関するヘッダファイル（*.h）をインクルードする．これにより，SDL の関数や定数，グローバル変数などの定義がプリプロセッサとしてソースコードに埋め込まれる．関数の仕様（使用法など）を確認したい時は，ヘッダファイルの中身を参照することになる．インクルード文の例を以下に示す．

記述例

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL2_gfxPrimitives.h>
#include <SDL2/SDL_net.h>
```

開発環境によって異なるが，ヘッダファイルは usr/include/SDL2/に格納されている．ヘッダファイルは複数存在し，サブシステムと呼ばれる機能ごとに分けられたライブラリと対応づけられている（表 2.1）．SDL の基本機能の定義をまとめる SDL.h ファイルには，多くのヘッダファイルがインクルードされている．

sdl2-config コマンドで，ヘッダファイルを格納しているディレクトリなど SDL の情報（設定）を確認できる．以下にコマンドの使用法（用意されているオプション）を示す．例えば，ヘッダファイルの格納ディレクトリを確認したい場合は cflags オプションを，SDL のバージョンを確認したい場合は version オプションを指定する．

表 2.1　主な SDL サブシステム

処理	SDL2
基本処理	SDL2
グラフィクス処理	SDL2_image
サウンド処理	SDL2_mixer
フォント処理	SDL2_ttf
図形処理	SDL2_gfx
ネットワーク処理	SDL2_net

sdl2-config コマンド

```
sdl2-config
Usage: /usr/local/bin/sdl2-config [--prefix[=DIR]] [--exec-prefix[=DIR]] [--version]
[--cflags] [--libs] [--static-libs]
```

2.1.2　メイン関数を記述する

C 言語にはメイン関数が存在し，そこに記述された処理が最初に実行される．ゲームにもメイン関数は必要であるが，ゲーム実行開始時すなわち実行ファイル起動時に，コマンドに付随していくつかの引数（コマンドライン引数）を受け取って，ゲームを制御したい場合がある．例えば，ゲームの難易度やモード，プレイヤの名前や人数などを指定して，ゲームを実行したい場合があるだろう．このようにターミナルからコマンドライン引数を受け取るには，以下のようにメイン関数に仮引数を指定する必要がある．

記述例

```
main(int argc, char *argv[]){ }
```

argc には引数の総数（コマンド名すなわち実行ファイル名も数に含む）が格納され，*argv には引数の値（コマンド名 argv[0] も含む．argv[1] が第 1 引数，argv[2] が第 2 引数，・・・）が文字列（文字列を指すポインタの配列）として格納されている．例えば，ターミナルから

記述例

```
./a.out 5 easy
```

と入力した場合，argc=3，argv[0]=”./a.out”，argv[1]=”5”，argv[2]=”easy” が格納されることになる．

2.1.3　SDL を初期化・終了する

SDL を使用するにはまず，**SDL_Init** 関数で SDL を初期化する必要がある．SDL はいくつかのサブシステムに分かれていることから，ひとつ以上のサブシステムに対して初期化することになる．

そして，プログラムの最後で，SDL の使用を（明示的に）終了する必要もある．

表 2.2　SDL 初期化用フラグ

用途	フラグ（値）
描画を使う場合	SDL_INIT_VIDEO
音声を使う場合	SDL_INIT_AUDIO
入出力を使う場合	SDL_INIT_EVENTS
タイマーを使う場合	SDL_INIT_TIMER
ジョイスティックを使う場合	SDL_INIT_JOYSTICK
すべてを使う場合	SDL_INIT_EVERYTHING

SDL（サブシステム）を初期化する

SDL_Init 関数

int SDL_Init(Uint32 flags)

- flags: サブシステム初期化用フラグ

どのサブシステムを使用するかによって，**SDL_Init** 関数に与える引数（列挙体[*1]によるフラグ）が異なる（表 2.2）．例えば，ウィンドウを表示させ，そこに描画を行うプログラムでは，フラグとして **SDL_INIT_VIDEO** を与えることになる．初期化用フラグは，論理和（OR 演算子 |）として列挙できるほか，**SDL_INIT_EVERYTHING** 列挙体を与えれば，すべてのサブシステムを初期化できる．その他，**SDL_INIT_HAPTIC** や **SDL_INIT_GAMECONTROLLER** などのフラグがある．

SDL 初期化の記述は，初期化の失敗を考慮してエラー処理を含めるべきである．次の例では，2 つのフラグを与え，**SDL_Init** 関数の返値が負値（エラーコードに対応）であれば，初期化失敗としてプログラムを終了する．初期化成功の場合は，0 が返値となる．

記述例

```
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
    printf("Failed to initialize SDL due to %s.\n", SDL_GetError()); // 最後に発生したエラーメッセージを表示
    exit(-1);
}
```

SDL の使用を終了する

SDL_Quit 関数

void SDL_Quit(void)

引数・返値ともに存在しない **SDL_Quit** 関数は，SDL プログラム終了時に呼び出せば，初期化した SDL（サブシステム）すべてを終了できる．

[*1] 列挙体は整数定数に名前をつけたものであり，列挙型とも呼ばれる．

記述例

```
SDL_Quit();
```

次の例では，atexit 関数で **SDL_Quit** 関数を指定することで，プログラム終了時に自動的に **SDL_Quit** 関数を呼び出すようにしている．

記述例

```
atexit(SDL_Quit);
```

サブシステムを個別に終了する

SDL_QuitSubSystem 関数

void SDL_QuitSubSystem(Uint32 flags)

使用したサブシステムを個別に終了させる場合は，対応するフラグを **SDL_QuitSubSystem** 関数に与える．

記述例

```
SDL_QuitSubSystem(SDL_INIT_VIDEO);
```

2.1.4 コンパイルする

ソースファイルを保存したら，コンパイルを試みる．ヘッダファイルをインクルードすることでプリプロセッサが機能し，SDL の使用がコパイラに通知されるが，最終的に実行ファイルを生成する際，SDL のオブジェクトファイル（SDL の実体である*.a や*.so ファイル）とのリンクに成功するかはコンパイルオプションの指定に依存する．つまり，SDL を使用するには，コンパイルオプションを適切に指定し，SDL のオブジェクトファイルを検索・リンクする必要がある．具体的には，

-l オプション　リンク対象を指定
-L オプション　ライブラリ検索ディレクトリを追加
-Wl オプション　リンカへのオプションを通知（共用ライブラリ使用時）
-I オプション　ヘッダファイル検索ディレクトリを追加

を指定することになる．環境によって異なるが，SDL のオブジェクトファイルは/usr/lib/以下や/usr/local/lib 以下に格納されている．test.c において，SDL の主要な機能（サブシステム）を使用している場合のコンパイルコマンド例を以下に示す（開発環境によっては，-L オプションや-Wl オプション，-I オプションは指定する必要がないかもしれない）．

記述例

```
gcc test.c -lSDL2 -lSDL2_gfx -lSDL2_ttf -lSDL2_image -lSDL2_mixer -L/usr/local/lib
-I/usr/local/include/SDL2 -Wl,-rpath,/usr/local/lib
```

-Wl オプションには，ライブラリのあるディレクトリを明示する必要があり，-Wl の直後に，複数のオプション（引数）をカンマでつなげて明示する（,-rpath, ディレクトリ名）．

2.2　ウィンドウ

ゲームに限らず多くのソフトウェアにおいて，ウィンドウは必須といえる．SDL を使用することで，ウィンドウを簡単に生成・表示できる（図 2.1）．ウィンドウ処理は SDL の基本処理に該当するため，SDL.h をインクルードし，コンパイルオプションとして-lSDL2 を付加する．

ウィンドウを生成・表示するには，**SDL_Init** 関数に **SDL_INIT_VIDEO** を与えて SDL を初期化する必要がある．ウィンドウのデータ（タイトル，座標やサイズなど）は，**SDL_Window** 構造体に格納される．

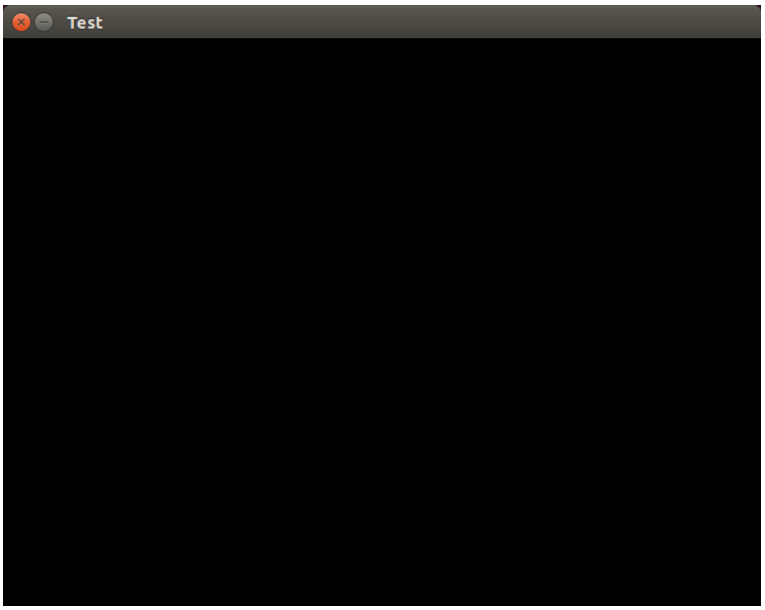

図 2.1　ウィンドウ

2.2.1　ウィンドウを生成・表示する

ウィンドウは，**SDL_Window** 構造体として生成される．ウィンドウを生成するには，ウィンドウの実体である **SDL_Window** 構造体を以下のように宣言する．

記述例

```
SDL_Window* window; // ウィンドウを示すポインタを宣言
```

ウィンドウを生成・表示する

SDL_CreateWindow 関数

SDL_Window SDL_CreateWindow(const char* title, int x, int y, int w, int h, Uint32 flags)*

- title: ウィンドウのタイトル
- x: ウィンドウの x 座標

（SDL_WINDOWPOS_CENTERED または SDL_WINDOWPOS_UNDEFINED も指定可）

- y: ウィンドウの y 座標
- w: ウィンドウの幅（ピクセル）
- h: ウィンドウの高さ（ピクセル）
- flags: SDL_WindowFlags 列挙体のフラグ（論理和可能）

SDL_CreateWindow 関数により，生成されたウィンドウのデータが **SDL_Window** 構造体変数の各メンバ（変数）に返値として格納される．**SDL_Window** 構造体へはポインタを介してアクセスすることになる．**SDL_WindowFlags** 列挙体の値はウィンドウの状態と対応づけられており（表 2.3），0 を指定すれば一般的なウィンドウが生成される．ウィンドウのタイトルを”Test”，表示座標をスクリーンの中心とし，サイズを 640 × 480 ピクセルで，一般的なウィンドウを生成・表示する際の記述例を以下に示す．

記述例

```
SDL_Window* window; // ウィンドウのデータを格納する構造体
window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
640, 480, 0);
if(window == NULL){
    printf("Failed to create window.\n");
    exit(-1);
}
```

または，このように，宣言と同時にウィンドウデータを構造体に格納してもよい．

記述例

```
SDL_Window* window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, 0);
```

SDL_CreateWindow 関数によりウィンドウが生成・表示されるが，ウィンドウ内の描画内容までは設定されていないため，ウィンドウの枠だけが生成された状態となっている[*2]．**SDL_CreateWindow** 関数の返値が NULL の場合，ウィンドウ生成に失敗したことになる．

複数の **SDL_CreateWindow** 関数を記述することで，複数のウィンドウを生成・表示することができる．

さて，ウィンドウの生成・表示に関して，**SDL_Window** 構造体がポインタ型として宣言されていることに注意したい．構造体変数を関数に引数として与えたり，返値として受け取る場合，構造体全体を値渡しすると，メモリ容量（スタック）や処理時間を無駄に消費することになりかねない．よって，構造体変数の関数への受け渡しには通常，参照渡し（ポインタで構造体のアドレスを渡す）が好まれる．参照渡しにするには，構造体名の直後または構造体変数名の直前に*を付加して構造体ポインタを宣言し，そこに当該の（関数へ渡したい）構造体変数の先頭アドレスを格納する．構造体ポインタ型として宣言された構造体変数内のメンバ（変数）にアクセスする場合は，.（ドット演算子）ではなく，->（アロー演算子）を使用する．構造体やポインタについて理解が十分でなければ，SDL プログラミングをきっかけに復習するとよい．

[*2] おそらく，プログラム実行時のデスクトップ画面がウィンドウ内に描画されるだろう．場合によっては，ウィンドウが表示されないかもしれない．

表 2.3　SDL_WindowFlags 列挙体の主な値

ウィンドウの状態	フラグ（値）
フルスクリーン（ビデオモードを変更）	SDL_WINDOW_FULLSCREEN
疑似フルスクリーン （現在のデスクトップの解像度に合わせてフルスクリーン）	SDL_WINDOW_FULLSCREEN_DESKTOP
表示された状態	SDL_WINDOW_SHOWN
非表示の（隠された）状態	SDL_WINDOW_HIDDEN
枠がない	SDL_WINDOW_BORDERLESS
サイズ可変の状態	SDL_WINDOW_RESIZABLE
最小化の状態	SDL_WINDOW_MINIMIZED
最大化の状態	SDL_WINDOW_MAXIMIZED
入力のフォーカスがグラブされている	SDL_WINDOW_INPUT_GRABBED
入力のフォーカスがある	SDL_WINDOW_INPUT_FOCUS
マウスのフォーカスがある	SDL_WINDOW_MOUSE_FOCUS
OpenGL コンテキスト使用	SDL_WINDOW_OPENGL

2.2.2　ウィンドウを設定する

ここでは，ウィンドウの設定およびデータの取得用の主要な関数を説明する．ここで取り上げなかった関数については，SDL リファレンスマニュアル（Web サイト）やサンプルプログラム（manaba にアップロードされている）を参照のこと．

ウィンドウ ID を取得する

SDL_GetWindowID 関数

Uint32 SDL_GetWindowID(SDL_Window window)*

- window: 対象のウィンドウ

記述例

```
Unit32 window_id = SDL_GetWindowID(window);
printf("Window ID=%d\n", window_id);
```

ウィンドウ ID は，複数のウィンドウがある中で特定のウィンドウを処理する場合などに必要となる．
返値として，取得に成功した場合はウィンドウ ID を，失敗した場合は 0 を返す．

ウィンドウのタイトルを設定（変更・反映）する

SDL_SetWindowTitle 関数

void SDL_SetWindowTitle(SDL_Window window, const char* title)*

- window: 設定対象のウィンドウ
- title: タイトル文字列

記述例

```
SDL_Window* window;
SDL_SetWindowTitle(window, "New Title")
```

ウィンドウの位置（座標）を設定する

SDL_SetWindowPosition 関数

void SDL_SetWindowPosition(SDL_Window window, int x, int y)*

- window: 設定対象のウィンドウ
- x: ウィンドウの x 座標
- y: ウィンドウの y 座標

x および y には，SDL_WINDOWPOS_CENTERED または SDL_WINDOWPOS_UNDEFINED も指定できる．言うまでもないが，ウィンドウの座標はディスプレイ左上を基点とする．

記述例

```
SDL_SetWindowPosition(window, 300, 300);
```

ウィンドウの位置（座標）を取得する

SDL_GetWindowPosition 関数

void SDL_GetWindowPosition(SDL_Window window, int* x, int* y)*

- window: 設定対象のウィンドウ
- x: 取得したウィンドウの x 座標を格納する変数のポインタ（取得の必要がなければ，NULL でもよい）
- y: 取得したウィンドウの y 座標を格納する変数のポインタ

記述例

```
int x, y;
SDL_GetWindowPosition(window, &x, &y);
printf("Window position: (%d, %d)\n", x, y);
```

ウィンドウの最大サイズを設定する

SDL_SetWindowMaximumSize 関数

void SDL_SetWindowMaximumSize(SDL_Window window, int max_w, int max_h)*

- window: 設定対象のウィンドウ
- max_w: ウィンドウの最大の幅（ピクセル）
- max_h: ウィンドウの最大の高さ（ピクセル）

記述例

```
SDL_SetWindowMaximumSize(window, 480, 320);
```

ウィンドウの最小サイズを設定する

SDL_SetWindowMinimumSize 関数

void SDL_SetWindowMinimumSize(SDL_Window window, int min_w, int min_h)*

- window: 設定対象のウィンドウ
- max_w: ウィンドウの最小の幅（ピクセル）
- max_h: ウィンドウの最小の高さ（ピクセル）

記述例

```
SDL_SetWindowMinimumSize(window, 320, 240);
```

ウィンドウのサイズを設定する

SDL_SetWindowSize 関数

void SDL_SetWindowSize(SDL_Window window, int w, int h)*

- window: 設定対象のウィンドウ
- w: ウィンドウの幅（ピクセル）
- h: ウィンドウの高さ（ピクセル）

記述例

```
SDL_SetWindowSize(window, 480, 60);
```

ウィンドウの最大サイズまたは最小サイズが設定されている場合，ウィンドウはそれらの値を逸脱したサイズにはならない（変更後のウィンドウは最小サイズから最大サイズの間に収まる）.

ウィンドウのサイズを取得する

SDL_GetWindowSize 関数

void SDL_GetWindowSize(SDL_Window window, int* w, int* h)*

- window: 設定対象のウィンドウ
- w: 取得したウィンドウの幅を格納する変数のポインタ（取得の必要がなければ，NULL でもよい）
- h: 取得したウィンドウの高さを格納する変数のポインタ

記述例

```
int w, h;
SDL_GetWindowSize(window, &w, &h);
printf("Window size: (%d, %d)\n", w, h);
```

ウィンドウを最大化する

SDL_MaximizeWindow 関数

void SDL_MaximizeWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

この関数は，ウィンドウのサイズを **SDL_SetWindowMaximumSize** 関数で設定した最大サイズに変更する．

記述例

```
SDL_MaximizeWindow(window);
```

ウィンドウをアイコン化する

SDL_MinimizeWindow 関数

void SDL_MinimizeWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

記述例

```
SDL_MinimizeWindow(window);
```

ウィンドウを隠す

SDL_HideWindow 関数

void SDL_HideWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

記述例

```
SDL_HideWindow(window);
```

ウィンドウの枠の状態を設定する

SDL_SetWindowBordered 関数

void SDL_SetWindowBordered(SDL_Window window, SDL_bool bordered)*

- window: 設定対象のウィンドウ
- bordered: SDL_FALSE（0 or false）を指定すれば枠を取り除き，SDL_TRUE（1 or true）であれば枠をつける

この関数は，SDL_WINDOW_BORDERLESS フラグを変更することで，ウィンドウの枠の状態を変更する．

記述例

```
SDL_SetWindowBordered(window, 0);
```

ウィンドウを最前面にして入力フォーカスを与える

SDL_RaiseWindow 関数

void SDL_RaiseWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

記述例

```
SDL_RaiseWindow(window);
```

この関数は，複数のウィンドウを生成している際に使用されることが多いだろう．

最大／最小化されたウィンドウを元のサイズと位置に戻す

SDL_RestoreWindow 関数

void SDL_RestoreWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

記述例

```
SDL_RestoreWindow(window);
```

ウィンドウのフルスクリーン状態を設定する

SDL_SetWindowFullscreen 関数

int SDL_SetWindowFullscreen(SDL_Window window, Uint32 flags)*

- window: 設定対象のウィンドウ
- flags: フルスクリーンに関するウィンドウ状態のフラグ
  （SDL_WINDOW_FULLSCREEN，SDL_WINDOW_FULLSCREEN_DESKTOP または 0）

記述例

```
SDL_SetWindowFullscreen(window, SDL_WINDOW_FULLSCREEN);
```

この関数では， **SDL_WindowFlags** 列挙体の SDL_WINDOW_FULLSCREEN や SDL_WINDOW_FULLSCREEN_DESKTOP を指定することで，ビデオモードをフルスクリーンに切り替える．0 を指定すると，ウィンドウモードになる．

ウィンドウを破棄する

SDL_DestroyWindow 関数

ivoid SDL_DestroyWindow(SDL_Window window)*

- window: 設定対象のウィンドウ

この関数は，ウィンドウをメモリから破棄（解放）する．破棄に伴って，表示されていたウィンドウは消去される．プログラムの終了処理として記述されることが多い．

記述例

```
SDL_DestroyWindow(window);
```

メッセージボックスを生成・表示する

メッセージボックスは確認事項やエラーの表示に使用される．

SDL_ShowMessageBox 関数

int SDL_ShowMessageBox(const SDL_MessageBoxData messageboxdata, int* buttonid)*

- messageboxdata: 生成・表示するメッセージボックスの **SDL_MessageBoxData** 構造体
- buttonid: ユーザが押したボタンの ID を格納する変数

メッセージボックスの生成・表示に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．**SDL_MessageBoxData** 構造体の定義を以下に示す．

SDL_MessageBoxData 構造体

```
typedef struct SDL_MessageBoxData {
    Uint32 flags; // SDL_MessageBoxFlags の 1 つ
    SDL_Window* window; // 親ウィンドウ（NULL も可）
    const char* title; // メッセージボックスのタイトル（UTF-8）
    const char* message; // メッセージボックスのメッセージ（UTF-8）
    int numbuttons; // 表示するボタンの数
    const SDL_MessageBoxButtonData* buttons; // ボタンのデータ
（SDL_MessageBoxButtonData 構造体の配列）
    const SDL_MessageBoxColorScheme* colorScheme; // メッセージボックスの色データ
（SDL_MessageBoxColorScheme 構造体）
} SDL_MessageBoxData;
```

メッセージボックスの用途に合わせて，**SDL_MessageBoxFlags** 列挙体（表 2.4）のフラグを指定する．**SDL_MessageBoxButtonData** 構造体の定義を以下に示す．

SDL_MessageBoxButtonData 構造体

```
typedef struct SDL_MessageBoxButtonData {
    Uint32 flags; //SDL_MessageBoxButtonFlags の 1 つ
    int buttonid; // ユーザが定義したボタンの ID
    const char* text; // ボタンの文字（UTF-8）
} SDL_MessageBoxButtonData;
```

SDL_MessageBoxColorScheme 構造体は配列として，メッセージボックス背景色，メッセージボック

表 2.4　SDL_MessageBoxFlags 列挙体の値

メッセージボックスの種類	フラグ（値）
エラーダイアログ	SDL_MESSAGEBOX_ERROR
警告ダイアログ	SDL_MESSAGEBOX_WARNING
情報ダイアログ	SDL_MESSAGEBOX_INFORMATION

ス文字色，ボタン外枠色，ボタン背景色，選択されたボタンの色のそれぞれに，RGB（0〜255）の値を指定する．

リターンキーまたはエスケープキーの押下にデフォルト対応する（メッセージボックス上の）ボタンを**SDL_MessageBoxButtonFlags** 列挙体（表 2.5）のフラグで指定する．

メッセージボックス設定の記述例を以下に示す．

記述例

```
int buttonid;
const SDL_MessageBoxButtonData buttons[] = { // ボタンの設定
    {0, 0, "Yes"}, // フラグ, ボタン ID, ボタン文字
    {SDL_MESSAGEBOX_BUTTON_RETURNKEY_DEFAULT, 1, "No"},
    {SDL_MESSAGEBOX_BUTTON_ESCAPEKEY_DEFAULT, 2, "Cancel"},
};
const SDL_MessageBoxColorScheme colorScheme = { // 色の設定
    {
        // メッセージボックス背景色
        {255, 255, 255}, // R, B, G
        // メッセージ文字色
        {0, 0, 0},
        // ボタン外枠色
        {100, 100,  100},
        // ボタン背景色
        {200, 200, 200},
        // 選択されたボタンの色
        {0, 0, 120}
    }
};
const SDL_MessageBoxData messageboxdata = { // メッセージ文字の設定
    SDL_MESSAGEBOX_INFORMATION, // フラグ
    window, // 親ウインドウ（親ウィンドウをもたない場合は NULL を指定）
    "サンプル", // タイトル
    "Press a button.", // メッセージ
    SDL_arraysize(buttons), // 配置するボタン数
    buttons, // 設定したボタン
    &colorScheme // 設定した色
};
if (SDL_ShowMessageBox(&messageboxdata, &buttonid) < 0) { // メッセージボックスの生
成・表示
    SDL_Log("Failed to show message box.\n");
    return 0;
}
if (buttonid == -1) {
    SDL_Log("Button not selected.\n");
} else {
   SDL_Log("%s was selected.\n", buttons[buttonid].text); // 選択されたボタン文字を表
示
}
```

表 2.5　SDL_MessageBoxButtonFlags 列挙体の値

リターンキーに対するデフォルト	SDL_MESSAGEBOX_BUTTON_RETURNKEY_DEFAULT
エスケープキーに対するデフォルト	SDL_MESSAGEBOX_BUTTON_ESCAPEKEY_DEFAULT

シンプルなメッセージボックスを生成・表示する

SDL_ShowSimpleMessageBox 関数

int SDL_ShowSimpleMessageBox(Uint32 flags, const char title, const char* message, SDL_Window* window)*

- flags: **SDL_MessageBoxButtonFlags** 列挙体の 1 つ
- title: メッセージボックスのタイトル（UTF-8）
- message: メッセージボックスのメッセージ（UTF-8）
- window: 親ウィンドウ（NULL も可）

メッセージボックスの生成・表示に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．

記述例

```
SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_ERROR, "Sample Simple Message",
"Please press the button.", NULL);
```

2.3　図形・文字描画

ウィンドウを生成すれば，そこに図形や文字，画像を描画する必要が生じるだろう．図形・文字・画像描画においては描画対象（描画先）が存在することになるが，それがサーフェイス (2.3.1)，レンダラー（2.3.2），またはテクスチャ（2.3.6）である．

図形描画処理には SDL_gfx サブシステムを，文字描画処理には SDL_tff サブシステムを用いてプログラムするが効率的である．

2.3.1　サーフェイス

描画対象であるサーフェイスは，描画データを格納（記録）するフレームバッファ（メインメモリ領域）である（図 2.2）．メインメモリ上のサーフェイス（背面バッファとも呼ばれる）に対する描画はソフトウェアレンダリングと呼ばれ，サーフェイスに描画データが格納されることを意味する．格納された描画データがディスプレイ（スクリーン）へ出力されることで，その描画内容がディスプレイに表示（反映）される．

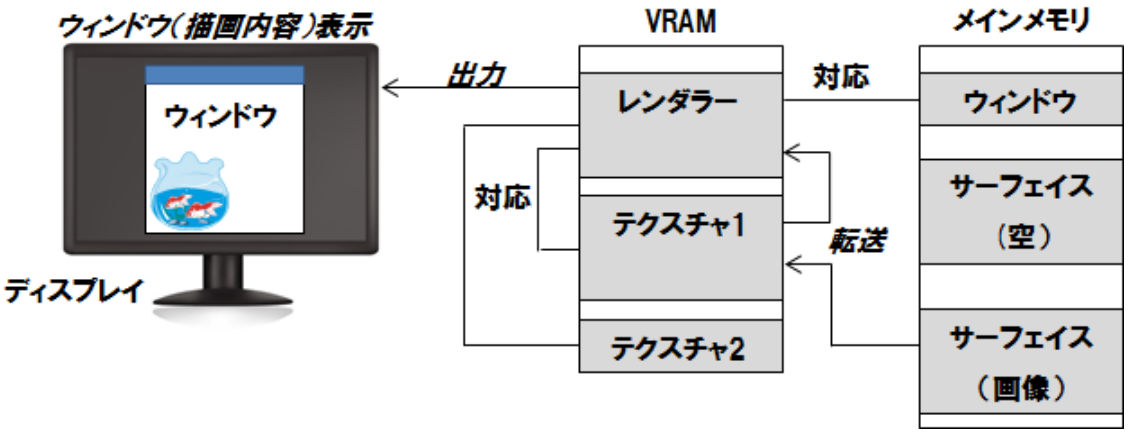

図 2.2　サーフェイス

サーフェイスは主に，画像ファイルから画像データを読み込んで格納するために使用される．読み込まれた画像データは VRAM 上のテクスチャに転送され，テクスチャに対して描画・画像処理が行われる．

サーフェイスのサイズなど概要データが格納される **SDL_Surface** 構造体は，以下のように定義される[*3]．内部使用変数については，プログラミングの際に意識する必要はない．

[*3] SDL1.2 から SDL2 に移行する際に，いくつかの変数が追加／削除／更新されたかもしれない．

SDL_Surface 構造体

```
typedef struct SDL_Surface {
    Uint32 flags; // （内部使用）
    SDL_PixelFormat *format; // サーフェイスのピクセル形式（読込専用）
    int w, h; // サーフェイスの幅と高さ（ピクセル）（読込専用）
    Uint16 pitch; // サーフェイスの幅の byte 数（読込専用）
    void *pixels; // 実際のピクセルデータへのポインタ（読み書き可）
    int locked; // サーフェイスのロック用（内部使用）
    void lock_data; // サーフェイスのロック用（内部使用）
    SDL_Rect clip_rect; // クリッピング処理用（読込専用）
    struct SDL_BlitMap *map; // 他サーフェイスへの高速コピーマッピング用データ（内部使用）
    int refcount; // 参照カウント（ほとんど読込）
} SDL_Surface;
```

pitch には，サーフェイス 1 行分の byte 数（w に依存）がセットされる．pixcels は，メモリ上の描画データを直接操作する際に用いることになる．clip_rect には，SDL_SetClipRect 関数により，**SDL_Rect** 構造体として以下のような矩形領域が設定される[*4]．

SDL_Rect 構造体

```
typedef struct{
    int x, y; // 矩形の左上の座標（x, y）
    int w, h; // 矩形の横幅（w）と高さ（y）
} SDL_Rect;
```

サーフェイスはピクセルの集合といえる．**SDL_PixelFormat** 構造体にはサーフェイスを構成するピクセル（の集合）のデータが格納されるが，この構造体を意識してプログラミングすることは少ない．

SDL_PixelFormat 構造体

```
typedef struct{
    SDL_Palette *palette; // SDL_Palette 構造体へのポインタまたは NULL
    Uint8  BitsPerPixel; // 各ピクセルの表現 bit 数（通常，8,16,24 または 32）
    Uint8  BytesPerPixel; // 各ピクセルの表現 byte 数（通常，1,2,3 または 4）
    Uint32 Rmask, Gmask, Bmask, Amask; // 各チャンネル値取得用ビットマスク
    Uint8  Rshift, Gshift, Bshift, Ashift; // ピクセル値の各チャンネルの 2 進数の左シフト数
    Uint8  Rloss, Gloss, Bloss, Aloss; // 各チャンネルで失われる精度
    Uint32 colorkey; // 透明ピクセルのピクセル値
    int refcount; // 参照カウント
    SDL_PixelFormat next;
} SDL_PixelFormat;
```

[*4] SDL_Rect 構造体は，描画データの部分転送など，領域の指定に多用される．

pallete は 8bit ピクセルフォーマット用のカラーパレット（256 色）で，SDL がサーフェイス用に SDL_PixelFormat を確保する際に自動的に生成される．なお，32bit のピクセルフォーマットは全く異なる．

サーフェイスを生成する

SDL_CreateRGBSurface 関数

SDL_Surface SDL_CreateRGBSurface(Uint32 flags, int width, int height, int depth, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask)*

- flags: 未使用（常に 0 にする）[*5]
- width: サーフェイスの幅（ピクセル）
- height: ウィンドウの高さ（ピクセル）
- bepth: 色深度
- Rmask: ピクセルの赤マスク
- Gmask: ピクセルの緑マスク
- Bmask: ピクセルの青マスク
- Amask: ピクセルの α マスク

色深度はピクセル深度とも呼ばれ，色表現ビット数（1 ピクセルのビット数）のことであり，32 を指定すれば 1677 万色（RGB 各 256 階調＋ α 値[*6]）となる．各マスクには，色指定ビット中のどのビットが R，G，B，または α チャンネル[*7]かを f で指定する．なお，マスクは各マシンのエンディアン[*8]（Byte の並び）に依存する．すべて 0 を指定すれば，マスクは考慮されない．

返値として，サーフェイスの生成に成功した場合はそのサーフェイスを，失敗した場合は NULL を返す．

[*5] フラグとして SDL_SWSURFACE，SDL_HWSURFACE，SDL_SRCCOLORKEY，SDL_SRCALPHA を論理和でも指定可．

[*6] ピクセル単位で色を混ぜ合わせる（重ね合わせる）比率で，透過度を意味する．

[*7] 画像等の α 値をまとめたデータ．α 値は 0〜255 までの値をとり，完全に透明にする場合は 0x00（0），不透明にする（混ぜ合わせない）場合は 0xff（255）となる．

[*8] ビッグエンディアンは，連続したデータを桁の大きい方からメモリに Byte ごとに格納する方式．リトルエンディアンはその逆．

記述例

```
SDL_Surface *surface;
Uint32 rmask, gmask, bmask, amask;

// プリプロセッサ if（エンディアンによって定数を変える）
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
    rmask = 0xff000000;
    gmask = 0x00ff0000;
    bmask = 0x0000ff00;
    amask = 0x000000ff;
#else
    rmask = 0x000000ff;
    gmask = 0x0000ff00;
    bmask = 0x00ff0000;
    amask = 0xff000000;
#endif

surface = SDL_CreateRGBSurface(0, width, height, 32, rmask, gmask, bmask, amask);
```

ピクセルデータからサーフェイスを生成する

SDL_CreateRGBSurfaceFrom 関数

SDL_Surface SDL_CreateRGBSurfaceFrom(void* pixels, int width, int height, int depth, int pitch, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask)*

- pixels: 既存ピクセルデータへのポインタ
- width: サーフェイスの幅（ピクセル）
- height: ウィンドウの高さ（ピクセル）
- bepth: 色深度
- pitch: 水平方向のバイト数
- Rmask: ピクセルの赤マスク
- Gmask: ピクセルの緑マスク
- Bmask: ピクセルの青マスク
- Amask: ピクセルの α マスク

返値として，サーフェイスの生成に成功した場合はそのサーフェイスを，失敗した場合は NULL を返す．

サーフェイスを別サーフェイスに転送する

描画（画像）データを複数のサーフェイスに格納しておき，それらを適宜テクスチャに転送することで，ディスプレイに意図した描画内容を表示したり，サーフェイス間で描画データ転送したい場合（例えば，読み込んだ小さな画像をサーフェイスに敷き詰めてからディスプレイに表示する）もあるだろう．このようにサー

フェイスを別サーフェイスに転送するには，**SDL_BlitSurface** 関数を用いる．

SDL_BlitSurface 関数

int SDL_BlitSurface(SDL_Surface src, const SDL_Rect* srcrect, SDL_Surface* dst, SDL_Rect* dstrect)*

- src: 転送元（コピー元）のサーフェイス
- srcrect: 転送元の矩形領域（NULL を指定すればサーフェイス全体）
- dst: 転送先（コピー先）のサーフェイス
- dstrect: 転送先の矩形領域（NULL を指定すればサーフェイス全体）

返値として，サーフェイスの転送に成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

転送元／先の矩形領域は，**SDL_Rect** 構造体で指定される．dstrect は矩形の幅と高さを必要とせず，NULL が指定されると転送先座標は (0, 0) になる．

記述例

```
SDL_Surface *surface1, *surface2;

SDL_Rect src_rect = { 50, 100, 150, 200 }; // 矩形領域を (50,100)-(200,300) に設定
// src_rect.x = 50; src_rect.y = 100; src_rect.w = 150; src_rect.h = 200; のような記述も可
SDL_Rect dst_rect = { 200, 100 }; // 矩形領域の左上座標を指定（幅と高さは省略）

SDL_BlitSurface(surface1, &src_rect, surface2, &dst_rect); // surface1 の描画データを surface2 に転送
```

この記述例のように，**SDL_BlitSurface** 関数に矩形領域を指定することで，描画内容（サーフェイスの描画データ）の一部をウィンドウサーフェイスに転送することができる（図 2.3）．

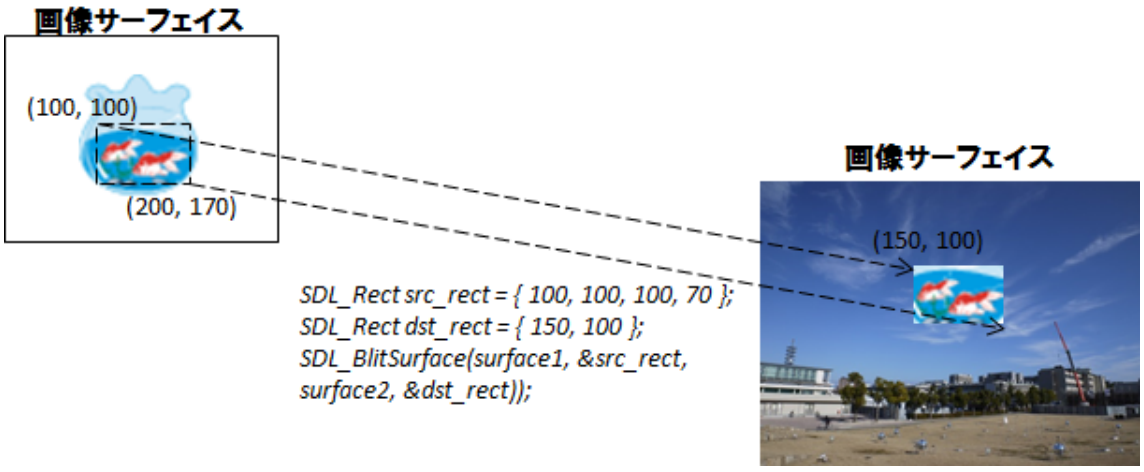

図 2.3　描画内容の部分転送

サーフェイスをロックする

複数のスレッド（処理）が同時並行で動作している場合（つまり，マルチスレッドの場合，あるスレッドがサーフェイス（フレームバッファ）を更新している最中に，他のスレッドが同じサーフェイスを更新してしまう状況が発生しうる．このような状況は，意図しない描画になってしまうことがあるため，避けるべきである．そこで，サーフェイス（描画データ）更新の際，他のスレッドがそのサーフェイスを更新できないようにするために，ロックという仕組み（排他制御，相互排除）が用意されている．当該スレッドのみがサーフェイスにアクセスできるようにロックする（鍵を掛ける）**SDL_LockSurface** 関数，ロックしたサーフェイスを解除する（他のスレッドによるアクセスを可能にする）**SDL_UnlockSurface** 関数がある．

SDL_LockSurface 関数

int SDL_LockSurface(SDL_Surface surface)*

- surface: 対象サーフェイス（ロックするサーフェイス）

返値として，ロックに成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

記述例

```
SDL_LockSurface(surface);
```

ロックしたサーフェイスを解除する

SDL_UnlockSurface 関数

int SDL_UnlockSurface(SDL_Surface surface)*

- surface: 対象サーフェイス（ロックを解除するサーフェイス）

返値として，ロックに成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

サーフェイスに対して複数のロックを設定することができるが，**SDL_LockSurface** 関数と **SDL_UnlockSurface** 関数は一対になっているため，ロックした数だけアンロックしなければ，他のスレッドがそのサーフェイスに対して描画できないままになる．あるサーフェイスへの描画処理の冒頭でロックし，その描画処理が終わった時点ですぐにアンロックするようにしたい．

記述例

```
SDL_UnlockSurface(surface);
```

サーフェイスを解放する

SDL_FreeSurface 関数

*void SDL_FreeSurface(SDL_Surface *surface)*

- surface: 対象サーフェイス

SDL_CreateRGBSurfaceFrom 関数で生成したサーフェイスの場合，ピクセルデータは解放されない．

記述例

```
SDL_FreeSurface(surface);
```

2.3.2　レンダラー

レンダラー（レンダリングコンテキスト）は描画対象を意味する．レンダラーは **SDL_CreateRenderer** 関数などで，ウィンドウに 1 対 1 で対応づけられて生成されることになる．SLD2 におけるレンダラーの採用は，SDL1.2 からの大きな変更のひとつである．レンダラーおよびテクスチャ[*9]の採用によりハードウェアレンダリング（VRAM+GPU）による高速かつ柔軟な描画を実現している（図 2.4）[*10]．

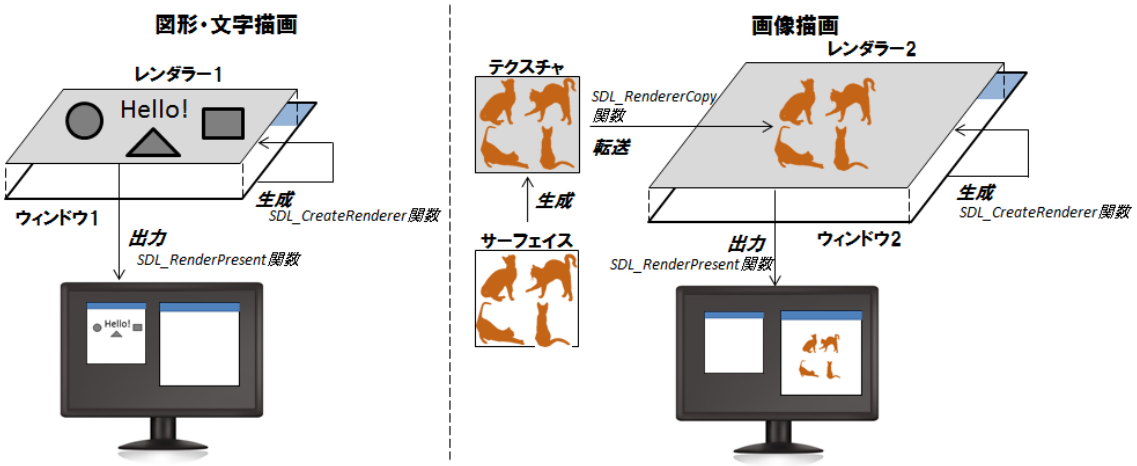

図 2.4　レンダラーに対する描画

レンダラーを生成する

レンダラーはウィンドウに対応づけて生成されるため，生成時に 1 つのウィンドウを指定する必要がある．生成されたレンダラーのデータは，レンダリングコンテキストとも呼ばれる描画設定も含んでおり，**SDL_Renderer** 構造体に格納される．レンダラーを生成するには，レンダラーの実体である **SDL_Renderer** 構造体を以下のように宣言する．

記述例

```
SDL_Renderer* renderer; // レンダラーを示すポインタを宣言
```

SDL_Renderer 構造体のメンバ（変数）は隠蔽されており，外部から直接読み書きができないようになっている．よって，描画設定の取得や更新の際は，関数を介して変数にアクセスすることになる．

SDL_CreateRenderer 関数

SDL_Renderer SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags)*

- window: 対象ウィンドウ

[*9] テクスチャについては，2.3.6 で説明する．
[*10] SDL1.2 におけるレンダリングは，主に CPU とメインメモリの組み合わせで処理されるソフトウェアレンダリングであった．

表 2.6　SDL_RendererFlags 列挙体の値

レンダラーの種類	フラグ（値）
ソフトウェア レンダラー	SDL_RENDERER_SOFTWARE
ハードウェア アクセラレーション	SDL_RENDERER_ACCELERATED
更新周期と同期	SDL_RENDERER_PRESENTVSYNC
テクスチャへのレンダリングに対応	SDL_RENDERER_TARGETTEXTURE

- index: 初期化するレンダリングドライバの番号
- flags: 0 または 1 つ以上の **SDL_RendererFlags** 列挙体（論理和）

index には通常，flags に対応した最初のドライバを表す-1 を指定する．flags には，**SDL_RendererFlags** 列挙体のフラグによりレンダラーの種類を指定する（表 2.6）．flags の値が 0 の場合，SDL_RENDERER_ACCELERATED のレンダラーで優先度が高いものになる．

返値として，レンダラーの生成に成功した場合はそのレンダリングコンテキストを，失敗した場合は NULL を返す．

記述例

```
SDL_Window* window; // ウィンドウのデータを格納する構造体
SDL_Renderer* renderer; // レンダリングコンテキスト（描画設定）を格納する構造体

window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
640, 480, 0);
renderer = SDL_CreateRenderer(window, -1, 0); // 生成したウィンドウに対してレンダラーを
生成
```

SDL_CreateWindowAndRenderer 関数

*int SDL_CreateWindowAndRenderer(int width, int height, Uint32 window_flags, SDL_Window** window, SDL_Renderer** renderer)*

- width: ウィンドウの幅（ピクセル）
- height: ウィンドウの高さ（ピクセル）
- window_flags: **SDL_WindowFlags** 列挙体のフラグ（論理和）
- window: 生成するウィンドウのポインタ（失敗時は NULL が格納される）
- renderer: 生成するレンダラーのポインタ（失敗時は NULL が格納される）

この関数は，ウィンドウとレンダラーを同時に生成する．ウィンドウとレンダラーの生成に成功すれば，そのウィンドウとレンダラーのアドレスが指定した引数に格納される．

返値として，生成に成功した場合は 0 を，失敗した場合は-1 を返す．

記述例

```
SDL_Window* window; // ウィンドウのデータを格納する構造体
SDL_Renderer* renderer; // レンダリングコンテキスト（描画設定）を格納する構造体

if(SDL_CreateWindowAndRenderer(640, 480, SDL_WINDOW_RESIZABLE, &window, &renderer))
{
    SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "Failed to create window and
renderer: %s", SDL_GetError());
    exit(-1);
}
```

なお，**SDL_CreateWindowAndRenderer** 関数では，**SDL_RendererFlags** 列挙体のフラグ，すなわち，レンダラーの種類を指定できないため，デフォルトフラグ（0）でレンダラーが生成される．

レンダラーの描画色を設定する

生成された直後のレンダラーは，描画設定（レンダリングコンテキスト）がデフォルトの状態になっている．特に図形・文字描画において，描画色の設定をデフォルト（黒）から変更する必要があるだろう．

SDL_SetRenderDrawColor 関数

int SDL_SetRenderDrawColor(SDL_Renderer renderer, Uint8 r, Uint8 g, Uint8 b, Uint8 a)*

- renderer: 対象レンダラー
- r: R（赤）成分
- g: G（緑）成分
- b: B（青）成分
- a: α 値（透過度）

それぞれの色成分には，0〜255 を指定する．α 値には通常，255（SDL_ALPHA_OPAQUE）を指定する．

返値として，設定に成功した場合は 0 を，失敗した場合は負値（エラーコード）を返す．

記述例

```
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255); // 生成したレンダラーに描画色と
して白を設定
```

レンダラーをクリアする（塗りつぶす）

レンダラーに図形などを描画した後，描画内容をクリアしたい場合がある．これは，レンダラーの全体を設定した描画色で塗りつぶすことに相当する．

SDL_RenderClear 関数

int SDL_RenderClear(SDL_Renderer renderer)*

- renderer: 対象レンダラー

返値として，クリアに成功した場合は 0 を，失敗した場合は負値（エラーコード）を返す．

記述例

```
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255); // 描画色として黒を設定
SDL_RenderClear(renderer); // 設定した描画色（黒）でレンダラーをクリア
```

レンダラーの描画内容を表示する

描画設定を終えて，図形や文字を描画する関数を呼び出しても，描画内容（描画データ）はディスプレイに表示（反映）されない．レンダラーの描画内容をディスプレイに表示するには，**SDL_RenderPresent** 関数を呼び出さなくてはならない．

SDL_RenderPresent 関数

void SDL_RenderPresent(SDL_Renderer renderer)*

- renderer: 対象レンダラー

記述例

```
stringColor(renderer, 0, 0, "TEST", 0xffffffff); // レンダラーに白で文字列を描画
SDL_RenderPresent(renderer); // レンダラー（描画データ）をディスプレイに表示
```

レンダラーを破棄する

レンダラーに描画する必要がなくなれば，メモリ領域の節約のためにも，レンダラーを破棄（解放）するとよい．なお，レンダラーの破棄により，テクスチャも破棄されることになる．

SDL_DestroyRenderer 関数

void SDL_DestroyRenderer(SDL_Renderer renderer)*

- renderer: 対象レンダラー

記述例

```
SDL_DestroyRenderer(renderer);
```

2.3.3 色を指定する

描画には色を指定する必要がある．

16 進数で色を指定する

サーフェイスの色深度が 32bit である場合，16 進数で色を指定するには，関数の色指定の引数に，0x から始まる 6 または 8 桁の 0〜f を指定する．色指定には，R（赤）成分，G（緑）成分，B（青）成分が必須で，8 桁がある場合はサーフェイス全体の α 値（透過度）が RGB に加わる．

ここで注意すべきは，エンディアンである．ビッグエンディアンかリトルエンディアンかによって，16 進数による色指定における色の配置（バイト順）が異なってくる．コンパイル時のバイト順を表すマクロである **SDL_BYTEORDER** の値からエンディアンを確認できる．

SDL_BIG_ENDIAN　この値であればビッグエンディアンであり，色指定は RGBA（赤・緑・青・α）の順となる．例えば，不透明の緑を指定するなら，0x00ff00ff となる．

SDL_LIL_ENDIAN　この値であればリトルエンディアンであり，色指定は ABGR の順となる．例えば，不透明の青を指定するなら，0xff00ff00 となる．

SDL_FillRect 関数を題材に，ビッグエンディアンにおける 16 進数による色指定の例を以下に示す．

記述例

```
SDL_FillRect(surface, NULL, 0xff0000); // 赤で塗り潰し
SDL_FillRect(surface, NULL, 0xff0000ff); // 赤で塗り潰し（不透明）
SDL_FillRect(surface, NULL, 0xff000000); // 赤？で塗り潰し（完全透明）
```

関数で色を指定する

描画関数における色指定のために，RGB（赤・緑・青）または RGBA（赤・緑・青・α）で指定した色を一意の整数値に変換する．

SDL_MapRGB 関数

Uint32 SDL_MapRGB(const SDL_PixelFormat format, Uint8 r, Uint8 g, Uint8 b)*

- format: 対象サーフェイスのピクセル形式
- r: R 値
- g: G 値
- b: B 値

format には，サーフェイスの各ピクセルのデータが格納されている．format に NULL を指定しても構わない．返値として，与えられたピクセル形式と RGB 値で最も近いピクセル値を返す．

記述例

```
SDL_FillRect(surface, NULL, SDL_MapRGB(surface->format, 0,0,0)); // 黒で塗りつぶす
```

SDL_MapRGBA 関数

Uint32 SDL_MapRGBA(const SDL_PixelFormat format, Uint8 r, Uint8 g, Uint8 b,Uint8 a)*

- format: （対象サーフェイスの）ピクセル形式
- r: R 値
- g: G 値
- b: B 値
- a: α 値

透過度を考慮して色指定ができる．

記述例

```
SDL_FillRect(surface, NULL, SDL_MapRGBA(window->surface, 0,0,0,255)); // 黒で塗りつぶす
```

2.3.4 SDL の基本機能で図形描画する

矩形領域を塗りつぶす

SDL_FillRect 関数

int SDL_FillRect(SDL_Surface dst, const SDL_Rect* rect, Uint32 color)*

- dst: 描画対象サーフェイス
- rect: 塗りつぶす領域
- color: 塗りつぶす色

rect には，**SDL_Rect** 構造体により領域を指定する．color には，16 進数，**SDL_MapRGB** 関数または **SDL_MapRGBA** 関数により色を指定する．返値として，塗りつぶしに成功した場合は 0 を，失敗した場合は負値（エラーコード）を返す．

記述例

```
SDL_Rect a = { 100, 100, 200, 200 }; // 矩形領域指定
SDL_FillRect(surface, a, SDL_MapRGBA(window->surface, 0,0,0,255)); // 黒で塗りつぶす
```

レンダラーに特化した（レンダラーのみを描画対象とする）以下のような図形描画関数が提供されている．ただし，円や三角形の描画関数は提供されていない．

点を描画する

SDL_RenderDrawPoint 関数

int SDL_RenderDrawPoint(SDL_Renderer renderer, int x, int y)*

- renderer: 描画対象レンダラー
- x: 点の x 座標
- y: 点の y 座標

描画に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．座標 (100, 100) に点を描画するには，以下のように記述する．

記述例

```
SDL_Window* window;
SDL_Renderer* renderer;
window = SDL_CreateWindow("Test",SDL_WINDOWPOS_CENTERED,SDL_WINDOWPOS_CENTERED,480,320,0);
renderer = SDL_CreateRenderer(window, -1, 0);

SDL_RenderDrawPoint(renderer, 100, 100);
```

複数の点を描画する

SDL_RenderDrawPoints 関数

int SDL_RenderDrawPoints(SDL_Renderer renderer, const SDL_Point* points, int count)*

- renderer: 描画対象レンダラー
- points: 点の配列
- count: 点の数

描画に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．points には，点の x 座標と y 座標を格納する **SDL_Point** 構造体の配列を指定する．(0, 0)，(50, 0)，(50, 50)，(0, 50) の 4 点を描画するには，以下のように記述する．

記述例

```
SDL_Point points[4] = {{0, 0}, {50, 0}, {50, 50}, {0, 50}}; // 点の座標 (x, y) を配列に格納
SDL_RenderDrawPoints(renderer, points, 4); // 複数の点を描画
```

直線を描画する

SDL_RenderDrawLine 関数

int SDL_RenderDrawLine(SDL_Renderer renderer, int x1, int y1, int x2, int y2)*

- renderer: 描画対象レンダラー
- x1: 始点の x 座標
- y1: 始点の y 座標
- x2: 終点の x 座標
- y2: 終点の y 座標
- color:描画色

描画に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．(100, 100)-(200, 200) に直線を描画するには，以下のように記述する．

記述例

```
SDL_RenderDrawLine(renderer, 100, 100, 200, 200);
```

複数の直線を描画する

SDL_RenderDrawLines 関数

int SDL_RenderDrawLines(SDL_Renderer renderer, const SDL_Point* points, int count)*

- renderer: 描画対象レンダラー
- points: 直線の端点の座標（配列）
- count: 直線の端点の数（count-1 本の直線が描かれる）

描画に成功すれば 0，失敗すれば負値（エラーコード）が返値となる．(0, 0)-(50, 0)-(50, 50)-(0, 50)-(0, 0) の 5 点をつなぐ 4 本の直線を描画するには，以下のように記述する．

記述例

```
SDL_Point points[5] = {{0, 0}, {50, 0}, {50, 50}, {0, 50}, {0, 0}}; // 端点の座標(x, y)を配列に格納
SDL_RenderDrawLines(renderer, points, 5); // 複数の線を描画
```

塗りつぶさない四角形を描画する

SDL_RenderDrawRect 関数

int SDL_RenderDrawRect(SDL_Renderer renderer, const SDL_Rect* rect)*

- renderer: 描画対象レンダラー
- rect: 四角形の領域データ

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．rect には，矩形領域を格納する **SDL_Rect** 構造体を指定する．四角形の左上の頂点を (0, 0)，幅 100 ピクセル，高さ 100 ピクセルの四角形（塗りつぶしなし）を描画するには，以下のように記述する．

記述例

```
SDL_Rect rect = { 0, 0, 100, 100 }; // 矩形領域データ（左上頂点の x,y 座標，幅，高さ）を格納する構造体
SDL_RenderDrawRect(renderer, &rect);
```

塗りつぶした四角形を描画するには，**SDL_RenderFillRect** 関数を用いる．その引数は，**SDL_RenderDrawRect** 関数と同様である．

2.3.5 SDL_gfx で図形描画する

SDL_gfx サブシステムのバージョンアップに伴い，関数が削除または変更されているが，ここでは基本的な関数を紹介する[*11]．

まずは，SDL_gfx サブシステムを用いる場合のインクルード文とコンパイルオプションの記述例を以下に示す．

記述例

```
#include <SDL2/SDL2_gfxPrimitives.h>
```

コンパイルオプション

```
-lSDL2_gfx
```

点を描画する

pixelColor 関数

[*11] その他の関数は下記を参照．
http://www.ferzkopp.net/Software/SDL2_gfx/Docs/html/index.html

*int pixelColor (SDL_Renderer *dst, Sint16 x, Sint16 y, Uint32 color)*

- dst: 描画対象レンダラー
- x: 点の x 座標
- y: 点の y 座標
- color: 描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．座標 (100, 100) に点を描画するには，以下のように記述する．

記述例

```
pixelColor(renderer, 100, 100, 0xffffffff);
```

水平線を描画する

hlineColor 関数

*int hlineColor(SDL_Renderer *dst, Sint16 x1, Sint16 x2, Sint16 y, Uint32 color)*

- dst: 描画対象レンダラー
- x1: 始点の x 座標
- x2: 終点の x 座標
- y: y 座標
- color: 描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(100, 50)-(200, 50) に水平線を描画するには，以下のように記述する．

記述例

```
hlineColor(renderer, 100, 200, 50, 0x0000ffff);
```

垂直線を描画する

vlineColor 関数

*int vlineColor(SDL_Renderer *dst, Sint16 x, Sint16 y1, Sint16 y2, Uint32 color)*

- dst: 描画対象レンダラー
- x: x 座標
- y1: 始点の y 座標
- y2: 終点 y 座標
- color: 描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(100, 50)-(100, 200) に垂直線を描画するには，以下のように記述する．

記述例

```
vlineColor(renderer, 100, 50, 200, 0xff0000ff);
```

直線を描画する

lineColor 関数

*int lineColor (SDL_Renderer *dst, Sint16 x1, Sint16 y1, Sint16 x2, Sint16 y2, Uint32 color)*

- dst: 描画対象レンダラー
- x1: 始点の x 座標
- y1: 始点の y 座標
- x2: 終点の x 座標
- y2: 終点の y 座標
- color:描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(100, 100)-(200, 200) に直線を描画するには，以下のように記述する．

記述例

```
lineColor(renderer, 100, 100, 200, 200, 0x00ff00ff);
```

塗りつぶさない四角形を描画する

rectangleColor 関数

*int rectangleColor (SDL_Renderer *dst, Sint16 x1, Sint16 y1, Sint16 x2, Sint16 y2, Uint32 color)*

- dst: 描画対象レンダラー
- x1: 四角形の左上の x 座標
- y1: 四角形の左上の y 座標
- x2: 四角形の右下の x 座標
- y2: 四角形の右下の y 座標
- color:描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(50, 50)-(200, 100) を対角線にもつ四角形（塗りつぶしなし）を描画するには，以下のように記述する．

記述例

```
rectangleColor(renderer, 50, 50, 200, 100, 0xaaaaaaff);
```

塗りつぶした四角形を描画するには，**boxColor** 関数を用いる．その引数は，**rectangleColor** 関数と同様である．

塗りつぶさない円を描画する

circleColor 関数

*int circleColor(SDL_Renderer * dst, Sint16 x, Sint16 y, Sint16 r, Uint32 color)*

- dst: 描画対象レンダラー
- x: 中心 x 座標
- y: 中心 y 座標
- r: 半径（ピクセル）
- color:描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(50, 50) を中心として半径 25 ピクセルの円（塗りつぶしなし）を描画するには，以下のように記述する．

記述例

```
circleColor(renderer, 50, 50, 25, 0xffff00ff);
```

塗りつぶした円を描画するには，**filledCircleColor** 関数を用いる．その引数は，**circleColor** 関数と同様である．

塗りつぶさない半円を描画する

pieColor 関数

*int pieColor(SDL_Renderer * dst, Sint16 x, Sint16 y, Sint16 r, Sint16 start, Sint16 end, Uint32 color)*

- dst: 描画対象レンダラー
- x: 中心 x 座標
- y: 中心 y 座標
- r: 半径（ピクセル）
- start: 開始角
- end: 終了角
- color:描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(100, 100) を中心として半径 50 ピクセルの円を 0 度から 90 度までの半円（塗りつぶしなし）で描画するには，以下のように記述する．

記述例

```
pieColor(renderer, 100, 100, 50, 0, 90, 0xff00ffff);
```

塗りつぶした半円を描画するには，**filledPieColor** 関数を用いる．その引数は，**pieColor** 関数と同様である．

塗りつぶさない三角形を描画する

trigonColor 関数

*int trigonColor(SDL_Renderer * dst, Sint16 x1, Sint16 y1, Sint16 x2, Sint16 y2, Sint16 x3, Sint16 y3, Uint32 color);*

- dst: 描画対象レンダラー

- x1: 第 1 頂点の x 座標
- y1: 第 1 頂点の y 座標
- x2: 第 2 頂点の x 座標
- y2: 第 2 頂点の y 座標
- x3: 第 3 頂点の x 座標
- y3: 第 3 頂点の y 座標
- color:描画色

返値として，描画に成功した場合は 0 を，失敗した場合は-1 を返す．(0, 0)，(50, 0)，(25, 25) を頂点とした三角形（塗りつぶしなし）で描画するには，以下のように記述する．

記述例

```
trigonColor(renderer, 0, 0, 50, 0, 25, 25, 0x00ffffff);
```

塗りつぶした半円を描画するには，**filledTrigonColor** 関数を用いる．その引数は，**trigonColor** 関数と同様である．

2.3.6 テクスチャ

テクスチャは SDL2 で採用された，VRAM 上で GPU により描画処理する仕組みであり，高速かつ柔軟な描画処理を実現できる．図形描画は比較的単純であるため，高速かつ柔軟な描画処理は必ずしも求められない．一方，文字や画像は色数の多さやアンチエイリアシングなどを含めた比較的複雑な描画になりがちなため，メインメモリではなく VRAM 上，CPU ではなく GPU での描画処理が求められる．このことから SDL2 では，文字・画像はレンダラーではなく，テクスチャに対して描画される．

テクスチャは高速かつ柔軟な描画処理のための一時的な VRAM 上の領域といえ，**SDL_Texture** 構造体に描画データが格納される．テクスチャを生成するには，テクスチャの実体である **SDL_Texture** 構造体を以下のように宣言する．

記述例

```
SDL_Texture* texture; // テクスチャを示すポインタを宣言
```

レンダラーと同様に，そのメンバ（変数）は隠蔽されており，外部から直接読み書きができない．また，**SDL_Texture** 構造体は同一スレッド内で扱う必要がある．さらに，描画内容（格納された描画データ）をディスプレイに表示するには，テクスチャからレンダラーへ描画データを転送（コピー）した後，**SDL_RenderPresent** 関数で描画内容をディスプレイに表示する必要がある．テクスチャはレンダラーに，レンダラーはウィンドウに対応づけられるため，多くの場合，テクスチャへの描画内容はウィンドウに反映される（図 2.5）．

テクスチャを生成する

テクスチャはレンダラーに対応づけて生成されるが，必ずしも 1 対 1 対応でなくてもよい．言い換えれば，1 つのレンダラーに対して複数のテクスチャを生成できる．

SDL_CreateTexture 関数

SDL_Texture SDL_CreateTexture(SDL_Renderer* renderer, Uint32 format, int access, int w, int h)*

図 2.5　テクスチャに対する描画

表 2.7　SDL_PixelFormatEnum 列挙体の主な値

ピクセル形式	フラグ（値）
不明	SDL_PIXELFORMAT_UNKNOWN
RGB 各 8 ビット	SDL_PIXELFORMAT_RGB888
RGBA 各 8 ビット	SDL_PIXELFORMAT_RGBA8888

- renderer: 対象レンダラー
- format: ピクセル形式
- access: テクスチャのアクセスパターン
- w: テクスチャの幅（ピクセル）
- h: テクスチャの高さ（ピクセル）

format には，ピクセル形式として **SDL_PixelFormatEnum** 列挙体の値（表 2.7）を指定する．access には，テクスチャのアクセスパターンとして **SDL_TextureAccess** 列挙体の値（表 2.8）を指定する．

返値として，生成に成功した場合はテクスチャへのポインタを，失敗した場合（レンダラー使用不可，format 非対応，範囲外の幅や高さ）は NULL を返す．

記述例

```
SDL_Window* window;
SDL_Renderer* renderer;
SDL_Texture* texture;
window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
640, 480, 0);
renderer = SDL_CreateRenderer(window, -1, 0);
texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA8888,
SDL_TEXTUREACCESS_TARGET, 800, 600);
```

サーフェイスからテクスチャを生成する

画像や文字（列）をサーフェイスに読み込み，その描画内容（画像や文字）からテクスチャを生成できる．ゲーム開発においても，この生成手法は空のテクスチャを生成する **SDL_CreateTexture** 関数よりも多用されるだろう．

SDL_CreateTextureFromSurface 関数

表 2.8　SDL_TextureAccess 列挙体の値

アクセスパターン	フラグ（値）
ほとんど変更されない（ロック不可）	SDL_TEXTUREACCESS_STATIC
頻繁に変更される（ロック可）	SDL_TEXTUREACCESS_STREAMING
レンダリング対象として使用可	SDL_TEXTUREACCESS_TARGET

SDL_Texture SDL_CreateTextureFromSurface(SDL_Renderer* renderer, SDL_Surface* surface)*

- renderer: 対象レンダラー
- surface: サーフェイス

surface には，描画データ（ピクセルデータ）をもつサーフェイスを指定する．したがって，予めサーフェイスに画像や文字を描画しておく必要がある．

返値として，生成に成功した場合はテクスチャへのポインタを，失敗した場合は NULL を返す．

記述例

```
SDL_Window* window = SDL_CreateWindow("Test",SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,320,240,0);
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);
SDL_Surface *image = IMG_Load("test.png"); // 画像を（サーフェイスに）読み込む

SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, image); // 読み込んだ
画像からテクスチャを生成
```

テクスチャの情報を取得する

SDL_QueryTexture 関数

int SDL_QueryTexture(SDL_Texture texture, Uint32* format, int* access, int* w, int* h)*

- texture: 対象テクスチャ
- format: テクスチャのピクセル形式を格納するポインタ
- access: テクスチャのアクセスパターンを格納するポインタ
- w: テクスチャの幅を格納するポインタ
- h: テクスチャの高さを格納するポインタ

実際的には，format および access には NULL を指定し，w と h を取得することが多い．

返値として，情報取得に成功した場合は 0 を，失敗した場合は負値（エラーコード）を返す．

記述例

```
SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, image); // 読み込んだ画像からテクスチャを作成

SDL_QueryTexture(texture, NULL, NULL, &image->w, &image->h); // 画像（テクスチャ）の情報（サイズなど）を取得
```

テクスチャの描画内容をレンダラーに転送する

テクスチャの描画内容をウィンドウに反映（表示）させるには，テクスチャに格納された描画データをレンダラーに転送（コピー）する必要がある．

SDL_RenderCopy 関数

int SDL_RenderCopy(SDL_Renderer renderer, SDL_Texture* texture, const SDL_Rect* srcrect, const SDL_Rect* dstrect)*

- renderer: 転送先（コピー先）レンダラー
- texture: 転送元（コピー元）のテクスチャ
- srcrect: 転送元の矩形領域（NULL を指定すればテクスチャ全体）
- dstrect: 転送先の矩形領域（NULL を指定すればレンダラー全体）

転送元／先の矩形領域は，**SDL_Rect** 構造体で指定される．srcrect で矩形領域を指定できることが，テクスチャの一部だけをレンダラーに転送することを可能にしている．dstrect で指定された矩形領域（サイズ）に応じて，描画データが拡大／縮小されレンダラーに転送される．

返値として，テクスチャの転送に成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

記述例

```
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);

SDL_Surface *image1 = IMG_Load("test1.png"); // 画像を（サーフェイスに）読み込む
SDL_Surface *image2 = IMG_Load("test2.png"); // 画像を（サーフェイスに）読み込む

SDL_Texture* texture1 = SDL_CreateTextureFromSurface(renderer, image1); // 読み込んだ画像からテクスチャを作成
SDL_Texture* texture2 = SDL_CreateTextureFromSurface(renderer, image2);

SDL_QueryTexture(texture1, NULL, NULL, &image1->w, &image1->h); // 画像（テクスチャ）の情報（サイズなど）を取得
SDL_QueryTexture(texture2, NULL, NULL, &image2->w, &image2->h);

SDL_Rect src_rect = {0, 0, image1->w, image1->h}; // 転送元画像の領域（この場合，画像全体が設定される）
SDL_Rect dst_rect = {0, 0, 100, 100}; // 画像の転送先の座標と領域
SDL_RenderCopy(renderer, texture1, &src_rect, &dst_rect); // テクスチャをレンダラーに転送（設定のサイズで）
SDL_RenderCopy(renderer, texture2, NULL, NULL); // テクスチャ全体をレンダラーに転送
```

テクスチャの描画内容を回転・反転してレンダラーに転送する

SDL_RenderCopyEX 関数

int SDL_RenderCopyEx(SDL_Renderer renderer, SDL_Texture* texture, const SDL_Rect* srcrect, const SDL_Rect* dstrect, const double angle, const SDL_Point* center, const SDL_RendererFlip flip)*

- renderer: 転送先（コピー先）レンダラー
- texture: 転送元（コピー元）のテクスチャ
- srcrect: 転送元の矩形領域（NULL を指定すればテクスチャ全体）
- dstrect: 転送先の矩形領域（NULL を指定すればレンダラー全体）
- angle: テクスチャの回転の角度（度数法・時計回り）
- center: テクスチャの回転の中心座標（NULL を指定すれば，dstrect.w/2, dstrect.h/2）
- flip: テクスチャの反転の種類

SDL_RenderCopy 関数と同様，転送元／先の矩形領域は **SDL_Rect** 構造体で指定され，部分転送および拡大縮小を可能にしている（図 2.6）．意図した回転になるよう，angle と center の値を指定する．center には **SDL_Point** 構造体で座標を指定する．flip には，**SDL_RendererFlip** 列挙体のフラグ（表 2.9）を指定する．

返値として，テクスチャの転送に成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

表 2.9　SDL_RendererFlip 列挙体の値

反転パターン	フラグ（値）
反転しない	SDL_FLIP_NONE
水平方向に反転	SDL_FLIP_HORIZONTAL
垂直方向に反転	SDL_FLIP_VERTICAL
対角線で反転	SDL_FLIP_HORIZONTAL | SDL_FLIP_VERTICAL

記述例

```
SDL_RenderCopyEx(renderer, texture, &src_rect, &dst_rect, 45, NULL,
SDL_FLIP_HORIZONTAL); // テクスチャをその中心で 45 度（時計回り）回転させ，水平方向に反転させてレンダラーに転送
```

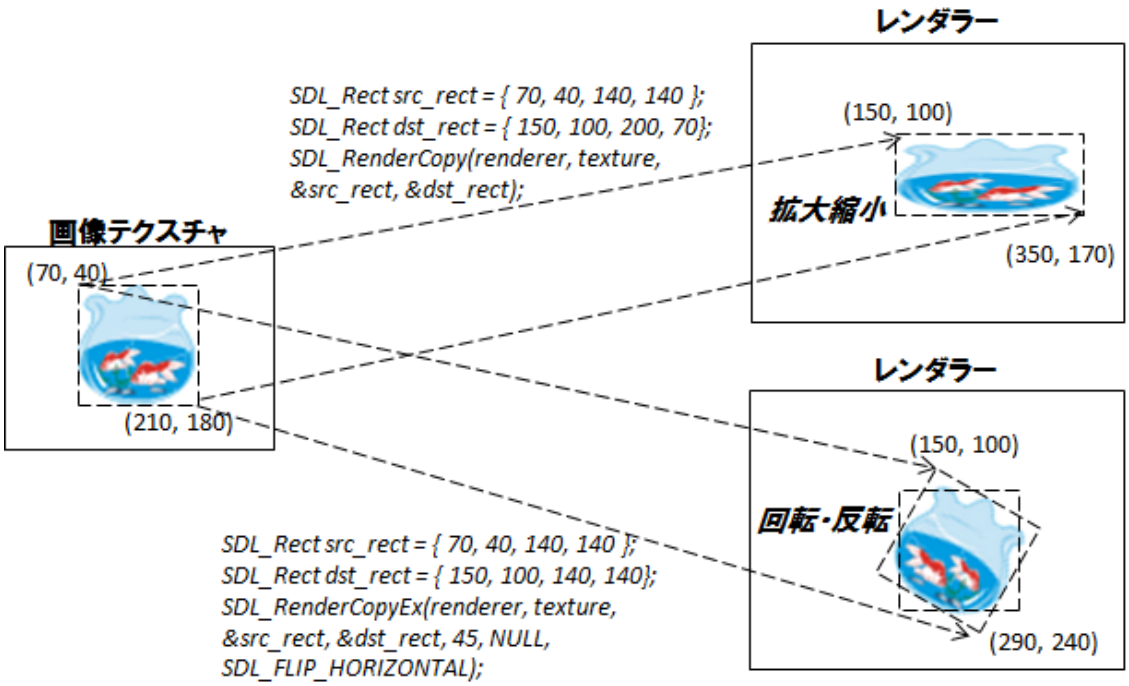

図 2.6　描画内容の部分転送および拡大縮小

描画内容を回転・反転させるプログラムをソフトウェアレンダリングで実現するとなると，各ピクセルに対して処理が必要になるだろう（多くの処理時間もかかるはずである）．テクスチャを導入することで，回転・反転を容易に実現できることが SDL2 の利点のひとつでもある．

テクスチャをロックする

テクスチャを(領域指定して)ロックすることができる．ただし，アクセスパターンが SDL_TEXTUREACCESS_STREAMIN
であるテクスチャのみロック可能である．

SDL_LockTexture 関数

int SDL_LockTexture(SDL_Texture texture, const SDL_Rect* rect, void** pixels, int* pitch)*

- texture: 対象テクスチャ（ロックするテクスチャ）
- rect: ロックする領域（NULL を指定すればテクスチャ全体をロックする）
- pixcel: ロックされた領域のピクセルのオフセットへのポインタを格納するポインタ
- pitch: ロックされたピクセルの水平方向のバイト数を格納するポインタ

返値として，ロックに成功した場合は 0 を，失敗した場合はエラーコード（負値）を返す．

記述例

```
void *pixels;
int pitch;
SDL_LockTexture(texture, NULL, &pixels, &pitch);
```

ロックしたテクスチャを解除する

SDL_UnlockTexture 関数

void SDL_UnlockTexture(SDL_Texture texture)*

- texture: 対象テクスチャ（ロックを解除するテクスチャ）

記述例

```
SDL_UnlockTexture(texture);
```

テクスチャを破棄する

テクスチャが必要なくなれば，メモリ領域の節約のためにも，テクスチャを破棄（解放）するとよい．

SDL_DestroyTexture 関数

void SDL_DestroyTexture(SDL_Texture texture)*

- texture: 対象テクスチャ

記述例

```
SDL_DestroyTexture(texture);
```

2.3.7 SDL_ttf で文字描画する

ゲームに限らずソフトウェアにおいて，ウィンドウへの文字描画は必須といえるだろう．SDL_gfx サブシステムに文字描画関数（stringColor 関数）は存在するが，それよりも使い勝手の良い **SDL_ttf** サブシステ

ム[*12]を用いて文字を描画したい．SDL_ttf の ttf は TrueType Font[*13]の頭文字であり，一般的に流通・公開されているさざまざまフォントで文字描画できることを意味している．

.ttf（または.ttc）というファイルが TrueType フォントのデータであり，おそらく Linux では，/usr/share/fonts ディレクトリ内に存在する．フォントの見た目を確認するには，*.ttf ファイルをフォントビューアで開いてみればよい（Ubuntu では*.ttf ファイルをダブルクリックするとビューアが開く）．

まずは，SDL_ttf サブシステムを用いる場合のインクルード文とコンパイルオプションの記述例を以下に示す．

記述例

```
#include <SDL2/SDL2_ttf.h>
```

コンパイルオプション

```
-lSDL2_ttf
```

TrueType フォントで文字描画するには，フォントデータを*.ttf ファイルから読み込んで **TTF_Font** 構造体に格納する必要がある．

記述例

```
TTF_Font* font; // TrueType フォントデータを示すポインタ
```

SDL_ttf を初期化する

TTF_Init 関数

int TTF_Init()

返値として，初期化に成功した場合は 0 を，失敗した場合は-1 を返す．

記述例

```
if(TTF_Init()==-1) {
    printf("Failed to initialize TTF: %s\n", TTF_GetError());
    exit(-1);
}
```

フォントを読み込む

TTF_OpenFont 関数

*TTF_Font *TTF_OpenFont(const char *file, int ptsize)*

- file: TrueType フォントファイル名
- ptsize: フォントサイズ（ポイント）

file には，読み込む*.ttf ファイルのパスを記述する．

[*12] http://www.libsdl.org/projects/SDL_ttf/docs/SDL_ttf.html

[*13] TrueType フォントは，拡大縮小してもジャギー（ギザギザ）が生じず解像度に依存しないなどの特長を有している．無償公開されている TrueType フォントはインターネットからダウンロードできる．

返値として，読み込みが成功した場合は指定したフォントデータ（TTF_Font 構造体変数に格納される）を，失敗した場合は NULL を返す．

記述例

```
font = TTF_OpenFont("kochi-gothic-subst.ttf", 24);
```

TrueType フォントで文字描画する

SDL_ttf サブシステムによる文字描画では，文字（列）を画像のように扱う．言い換えれば，文字を書き込んだ画像をまずサーフェイスに格納し，いくつかの処理を経てディスプレイに表示する．

文字描画に用いるのは TTF_Render 関数群であり，使用する文字コードやフォントの荒さ[*14]に応じて TTF_Render に接尾語が追加された 12 の関数から構成される．どの文字描画関数も，文字（列），色，（読み込んだ）フォントを指定して描画する．日本語文字を描画したい場合は，文字コードが UTF8 の **TTF_RenderUTF8_Solid** 関数などを使うとよい．

まず文字をサーフェイスに描画し（描画データをサーフェイスに読み込み），そのサーフェイスからテクスチャを生成した後，レンダラーに転送して文字描画に至る．

TTF_Render 関数群

*SDL_Surface *TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderText_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_Color bg)*

*SDL_Surface *TTF_RenderText_Blended(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderUTF8_Solid(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderUTF8_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_Color bg)*

*SDL_Surface *TTF_RenderUTF8_Blended(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderText_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_Color bg)*

*SDL_Surface *TTF_Render Text_Blended(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderUNICODE_Solid(TTF_Font *font, const char *text, SDL_Color fg)*

*SDL_Surface *TTF_RenderUNICODE_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_Color bg)*

*SDL_Surface *TTF_RenderUNICODE_Blended(TTF_Font *font, const char *text, SDL_Color fg)*

- font: 読み込んだ TrueType フォント
- text: 描画する文字（列）
- fg: 文字色
- bg: 背景色

[*14] フォントの荒さは，（プログラムにより）実際に描画された文字列を見て確認してほしい（Solid が粗く，その他はなめらかになる．描画速度は Soild が最も速くなる）．

fg および bg では，以下のように定義された **SDL_Color** 構造体を用いて色を指定する．

SDL_Color 構造体

```
typedef struct{
    Uint8 r; // 赤
    Uint8 g; // 緑
    Uint8 b; // 青
    Uint8 unused;
} SDL_Color;
```

記述例

```
SDL_Surface *strings;
TTF_Font*  font = TTF_OpenFont("kochi-gothic-subst.ttf", 24);
SDL_Color white = {0xFF, 0xFF, 0xFF}; // フォントの色を白に指定

strings = TTF_RenderUTF8_Blended(font, "Hello!", white); // 文字列を画像として格納

texture = SDL_CreateTextureFromSurface(renderer, strings); // サーフェイス（文字列の描
画データが格納されている）からテクスチャを生成

SDL_Rect src_rect = {0, 0, strings->w, strings->h}; // 転送元
SDL_Rect dst_rect = {0, 0, 640, 480}; // 転送先（拡大／縮小）

SDL_RenderCopy(renderer, texture, &src_rect, &dst_rect); // テクスチャをレンダラーに
転送
SDL_RenderPresent(renderer); // 描画データを表示
```

フォントを閉じる

TTF_CloseFont 関数

*void TTF_CloseFont(TTF_Font *font)*

- font: 読み込んだフォント

記述例

```
TTF_CloseFont(font);
```

SDL_ttf の使用を終了する

TTF_Quit 関数

void TTF_Quit();

記述例

```
TTF_Quit();
```

2.4　イベント駆動型プログラミング

イベント駆動型プログラミングとは，ユーザの操作や OS からの割込といったイベントを検知し，検知したイベントに対応する処理を行うプログラムの動作方法または概念のことである．ゲーム開発に限らずソフトウェア全般において，イベント駆動型プログラミングは必須といえる．

イベントには多くの種類がある．例えば，典型的なイベントとして，「マウス（のボタン）がクリックされた」，「（キーボードの）キーが押された」，「ある時刻になった」などが挙げられる．特に近年では，デバイスの多様化に伴い，（画面の）スワイプやピンチイン／アウトといった新しいイベントにも対応する必要性も増してきている．

ここでは，一般的なデバイスであるマウス，キーボード，ジョイスティックに焦点を当て，それぞれの主要なイベントを取り上げて，イベント駆動型プログラミングを説明する．

2.4.1　イベント駆動型プログラムの構成

イベント駆動型プログラムは次の要素から構成される．

イベントループ　イベントの発生を検知するための待機状態（無限ループ）
トリガ　イベントを発生させるきっかけ
イベントディスパッチャ　検知したイベントに応じて処理を振り分ける機能
イベントハンドラ　検知したイベントに応じて呼び出される関数（処理）
イベントキュー　連続して発生したイベントを格納する待ち行列

イベント駆動型プログラムは，イベントループ内でイベントの発生を待ち→特定のイベントを検知して対応する処理を実行し→再びイベントループに戻る，という流れで実行される（図 2.7）．

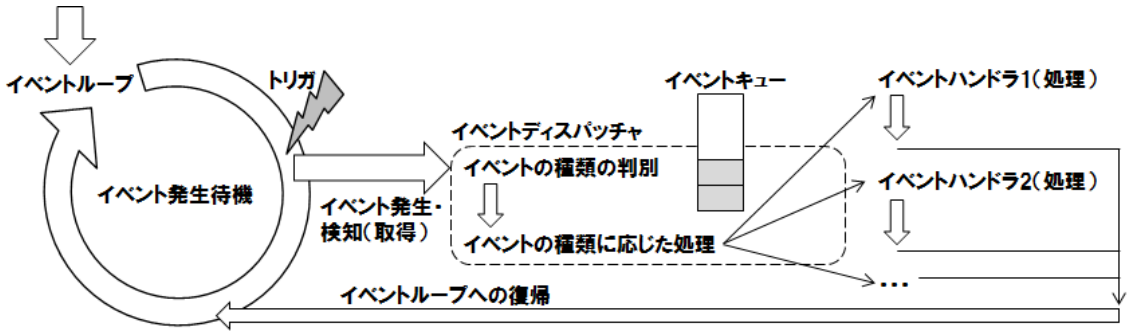

図 2.7　イベント駆動型プログラムの流れ

イベントの発生を検知する

SDL を介してイベントの発生を検知するには，**SDL_Init** 関数で **SDL_INIT_VIDEO** フラグを指定して SDL を初期化する必要がある．さらに，イベントデータを格納する **SDL_Event** 共用体[*15]を宣言し，無限ループ内で **SDL_PollEvent** 関数などによりイベントを検知する．イベントキューは，**SDL_Event** 共用体の列で構成されており，そのひとつひとつが未処理のイベントのデータである．イベントの種類は **SDL_Event** 共用体の type メンバ（変数）に格納されるので，この type メンバの値によって処理を振り分けることになる．

SDL_Event 共用体の各メンバには，以下に示すように，イベントごとにデータを格納する構造体が宣言されている[*16]．

[*15] イベントは同時には発生しないため，1 度に 1 つのイベントしかイベントキューに追加されない（場合によっては即座に処理される）．つまり，未処理イベントをイベントキューへ追加する一時的なメモリ領域として機能すればよいので，複数メンバ（変数）が同じメモリ領域を共有できる．よって，共用体を採用している．

[*16] SDL1.2 から SDL2 への移行に際して，いくつかのメンバが追加，変更または削除されている．

SDL_Event 共用体

```
typedef union{
    Uint8 type; // イベントの種類
    SDL_KeyboardEvent key; // キーボードイベント
    SDL_TextEditingEvent edit; // テキスト編集イベント
    SDL_TextIputEvent edit; // テキスト入力イベント
    SDL_MouseMotionEvent motion; // マウス移動イベント
    SDL_MouseButtonEvent button; // マウスボタンイベント
    SDL_JoyAxisEvent jaxis; // ジョイスティック軸イベント
    SDL_JoyBallEvent jball; // ジョイスティックボールイベント
    SDL_JoyHatEvent jhat; // ジョイスティックハットイベント
    SDL_JoyButtonEvent jbutton; // ジョイスティックボタンイベント
    SDL_QuitEvent quit; // 終了要求イベント
    SDL_UserEvent user; // アプリケーション定義イベント
    SDL_SywWMEvent syswm; // システム依存ウィンドウイベント

    SDL_CommonEvent common; // 共通イベント
    SDL_WindowEvent window; // ウィンドウイベント
    SDL_MouseWheelEvent wheel; // マウスホイールイベント
    SDL_JoyDeviceEvent jdevice; // ジョイスティック接続イベント
    SDL_ControllerAxisEvent caxis; // ゲームコントローラ軸イベント
    SDL_ControllerButtonEvent cbutton; // ゲームコントローラボタンイベント
    SDL_ControllerDeviceEvent cdevice; // ゲームコントローラ接続イベント
    SDL_AudioDeviceEvent adevice; // オーディオデバイスイベント
    SDL_TouchFingerEvent tfinger; // タッチイベント
    SDL_MultiGestureEvent mgesture; // マルチジェスチャーイベント
    SDL_DollarGestureEvent dgesture; // 複雑なジェスチャーイベント
    SDL_DropEvent drop; // ドラッグ＆ドロップイベント

} SDL_Event;
```

これらのメンバ構造体は，イベントの type に応じて上書きされる．例えば，motion という type のイベントを取得した場合，**SDL_MouseMotionEvent** 構造体により，**SDL_Event** 構造体が上書きされる．**SDL_MouseMotionEvent** 構造体にも type メンバが存在しているため，イベントの種類のデータは失われない．

イベント駆動型プログラムは，**SDL_PollEvent** 関数を介してイベントキューからイベントを取り出し，そのイベントの type の値（表 2.10）に応じて処理を振り分ける．基本的には OS が発生したイベントをイベントキューに追加するが，プログラマが意図的にイベントキューにイベントを追加することもできる．基本的なイベント駆動型プログラムは次のように記述される．

記述例

```
SDL_Event event; // イベントデータが格納される構造体を宣言

SDL_Init(SDL_INIT_VIDEO); // SDL を初期化

while(1){ // 無限ループ
    // イベントを取得したら
    if ( SDL_PollEvent(&event) )
    {
        switch (event.type) {
        // 以下にイベントごとの処理を記述
        case SDL_MOUSEMOTION: // マウスが移動した時
            ....
        }
    }
}
```

SDL_PollEvent 関数

int SDL_PollEvent(SDL_Event event)*

- event: イベント

この関数は，イベントキューから未処理のイベントを取得する．SDL_PeepEvents 関数はイベントキューに追加できるイベント数や取得対象イベントのフィルタリングなどを設定できるが，ここでは割愛する（Web リファレンスを参照のこと）．

返値として，未処理のイベントが存在する場合（イベントデータを取得した場合）は 1 を，イベントが存在しない場合は 0 を返す．

SDL_WaitEvent 関数

int SDL_WaitEvent(SDL_Event event)*

- event: イベント

この関数は，次のイベントが発生するまで無制限に待機することから，while の条件として採用できる．

返値として，待機状態（イベント発生時も含む）の場合は 1 を，待機中にエラーが発生すれば 0 を返す．

表 2.10　イベントの type

イベント	フラグ（値）
キーボード	SDL_KEYDOWN または SDL_KEYUP
テキスト編集	SDL_TEXTEDITING
テキスト入力	SDL_TEXTINPUT
マウス移動	SDL_MOUSEMOTION
マウスボタン	SDL_MOUSEBUTTONDOWN または SDL_MOUSEBUTTONUP
ジョイスティック軸	SDL_JOYAXISMOTION
ジョイスティックボール	SDL_JOYBALLMOTION
ジョイスティックハット	SDL_JOYHATMOTION
ジョイスティックボタン	SDL_JOYBUTTONDOWN または SDL_JOYBUTTONUP
終了要求	SDL_QUIT
アプリケーション定義	SDL_USEREVENT または SDL_RegisterEvents 関数で得られた値
システム依存ウィンドウ	SDL_SYSWMEVENT
ウィンドウ	SDL_WINDOWEVENT
マウスホイール	SDL_MOUSEWHEEL
ジョイスティック接続	SDL_JOYDEVICEADDED または SDL_JOYDEVICEREMOVED
ゲームコントローラ軸	SDL_CONTROLLERAXISMOTION
ゲームコントローラボタン	SDL_CONTROLLERBUTTONDOWN または SDL_CONTROLLERBUTTONUP
ゲームコントローラ接続	SDL_CONTROLLERDEVICEADDED または SDL_CONTROLLERDEVICEREMOVED
オーディオデバイス	SDL_AUDIODEVICEADDED または SDL_AUDIODEVICEREMOVED
タッチ	SDL_FINGERMOTION, SDL_FINGERDOWN または SDL_FINGERUP
マルチジェスチャー	SDL_MULTIGESTURE
複雑なジェスチャー	SDL_DOLLARGESTURE または SDL_DOLLARRECORD
ドラッグ&ドロップ	SDL_DROPFILE, SDL_DROPTEXT, SDL_DROPBEGIN, または SDL_DROPCOMPLETE

記述例

```
while(SDL_WaitEvent(&event)){ // イベント待機ループ
    switch (event.type) {
    // 以下にイベントごとの処理を記述（イベントハンドラを呼び出すか，単純な処理であれば直接記
述）
        ....
    }
}
```

SDL_PushEvent 関数

*int SDL_PushEvent(SDL_Event *event)*

- event: イベントキューに追加するイベント

この関数は，意図的にイベントキューにイベントを追加する．返値として，追加に成功した場合は 1 を，失敗した場合は負数（エラーコード）を返す．

記述例

```
SDL_Event quit_event = { SDL_QUIT }; // 対象のイベントを格納
SDL_PushEvent(&quit_event); // 意図的にイベントをイベントキューに追加
```

イベントを定義する

アプリケーション（言い換えればユーザまたはプログラマ）は，イベントデータの作成によりイベントを定義できる．アプリケーション定義イベントは，SDL によって発生されるのではなく（OS から発生が通知されるのではなく），ユーザにより意図的に発生させることができる．よって，**SDL_PushEvent** 関数によりアプリケーション定義イベントをイベントキューに格納し，対応するイベント処理に移行する必要がある．アプリケーション定義イベントのデータは **SDL_UserEvent** 構造体に格納される．

SDL_UserEvent 構造体

```
typedef struct{
    Uint32 type; // SDL_USEREVENT または SDL_RegisterEvents 関数で得られた値
    Uint32 timestamp; // イベントのタイムスタンプ
    Uint32 windowID; // フォーカスのあるウィンドウ
    Sint32 code; // ユーザ定義のイベントコード
    void *data1; // ユーザ定義のデータ（ポインタ）
    void *data2; // ユーザ定義のデータ（ポインタ）
} SDL_UserEvent;
```

SDL_RegisterEvents 関数

Uint32 SDL_RegisterEvents(int numevents)

- numevents: 確保するイベントの数

この関数は，複数のユーザ定義のイベントの領域を確保する．返値として，追加に成功した場合は先頭のイベント番号を，失敗した（イベントを領域を確保できない）場合は (Uint32)-1 を返す．

記述例

```
SDL_Event event;
event.type = SDL_USEREVENT;
event.user.code = my_event_code; // コード
event.user.data1 = significant_data; // データ 1
event.user.data2 = 0; // データ 2
SDL_PushEvent(&event); // 定義したイベントをイベントキューに追加
```

2.4.2 キーボードイベントを処理する

キーボードイベントは，キーの状態（キーが押されたか，離されたか）が変化することで発生する．よって，どのキーが押されたか／離されたかに応じて処理を振り分けることになる．キーボードイベントのデータは，以下のような **SDL_KeyboardEvent** 構造体に格納される．

SDL_KeyboardEvent 構造体

```
typedef struct{
    Uint32 type; // SDL_KEYDOWN または SDL_KEYUP
    Uint32 timestamp; // イベントのタイムスタンプ
    Unit32 windowID; // フォーカスのあるウィンドウ
    Uint8 state; // キーの状態（SDL_PRESSED または SDL_RELEASED）
    Uint8 repeat; // キーリピート（0 でなければ）
    SDL_Keysym keysym; // 押された／離されたキーのデータ
} SDL_KeyboardEvent;
```

押された（または，離された）キーのデータは，**SDL_Keysym** 構造体に格納される．

SDL_Keysym 構造体

```
typedef struct{
    SDL_Scancode scancode; // スキャンコード（SDL_Scancode 列挙体）
    SDL_Keycode sym; // キーコード（SDL_Keycode 列挙体）
    Uint16 mod; // 現在の修飾キー（SDL_Keymod 列挙体）
    Uint32 unused; // 未使用
} SDL_Keysym;
```

押された（または，離された）キーを検知するために，キーの物理的な位置を表すスキャンコードとキーの仮想的な位置を表すキーコードが用意されている．キーボードには（国や言語に応じた）JIS 配列や US 配列などのキー配列があるが，キーの物理的な配置が同じであれば，スキャンコードはキーに刻印された文字に関わらず同じになる．一方，キーコードはキーの仮想的な配置，すなわち，キーに刻印された文字または OS が割り当てた仮想キーに応じる．これら 2 つのコードは対を成しており[*17]，SDL2 において，スキャンコードは **SDL_Scancode** 列挙体として，キーコードは **SDL_Keycode** 列挙体（表 2.11）として与えられる．なお，SDL1.2 から SDL2 への移行において，変更されたキーコードがいくつか存在する．

現在の修飾キーは，「Alt キーを押しながら○キー」や「Ctrl キーを押しながら○キー」のように「押しながら」という状態を判別するために用意されており，**SDL_Keymod** 列挙体（表 2.12) として与えられる．

以下に，キーボードイベント処理プログラムの一例（一部）を示す．まず type の値からイベントを検知（イベントの種類を判別）し，「キーが押された」というキーボードイベントであることから，key.keysym.sym の値からどのキーが押されたかを検知して，対応する処理を実行している．

[*17] http://sdl2referencejp.osdn.jp/SDL_Scancode.html
http://sdl2referencejp.osdn.jp/SDL_Keycode.html

表 2.11　主なキーコード

キー	SDL_Keycode（値）
Back Space	SDLK_BACKSPACE
Tab	SDLK_TAB
Return（Enter）	SDLK_RETURN
Esc	SDLK_ESCAPE
スペース	SDLK_SPACE
Delete	SDLK_DELETE
↑, ↓, →, ←	SDLK_UP, SDLK_DOWN, SDLK_RIGHT, SDLK_LEFT
F1, F2, F3, ...	SDLK_F1, SDLK_F2, SDLK_F3...
A, B, C, ...	SDLK_a, SDLK_b, SDLK_c, ...
0, 1, 2, ...	SDLK_0, SDLK_1, SDLK_2, ...
キーパッドの 0, 1, 2, ...（SDL1.2）	SDLK_KP0, SDLK_KP1, SDLK_KP2, ...
キーパッドの 0, 1, 2, ...（SDL2）	SDLK_KP_0, SDLK_KP_1, SDLK_KP_2, ...

表 2.12　主な修飾キー

キー	修飾キーマスク（値）
修飾キーが押されていない (0)	KMOD_NONE
右 Ctrl キー押下	KMOD_RCTRL
左 Ctrl キー押下	KMOD_LCTRL
右 Shift キー押下	KMOD_RSHIFT
左 Shift キー押下	KMOD_LSHIFT
右 Alt キー押下	SDLK_RALT
左 Alt キー押下	SDLK_LALT

記述例

```
SDL_Event event;
while(1){
    if( SDL_PollEvent(&event) ){
        // イベントの種類を判別
        switch (event.type) {
        case SDL_KEYDOWN: // キーが押された場合
            // どのキーが押されたかを判別
            switch(event.key.keysym.sym){
            case SDLK_ESCAPE: // Esc キーが押された場合
                ....
                break;
            ....
            }
        ....
        break;
        }
    }
}
```

キー名を取得する

SDL_GetKeyName 関数

const char SDL_GetKeyName(SDL_Keycode key)*

- key: 名前を取得するキーのキーコード

返値として，文字列（UTF-8）へのポインタを返す．

記述例

```
printf("The pressed key is %s.\n", SDL_GetKeyName(event.key.keysym.sym));
```

修飾キーの状態を取得する

SDL_GetModState 関数

SDL_Keymod SDL_GetModState(void)

返値として，修飾キーの組み合わせを論理和で返す．

記述例

```
printf("The modifier state is %x (%x).\n", SDL_GetModState(), event.key.keysym.mod);
```

キーリピートを設定する

キーリピートでは，キーを押したままの状態において，1 回だけキーを押したようにイベント取得するか（キーリピート無効），何回もキーを押しているようにイベント取得するか（キーリピート有効）を設定する．例えば，アクションゲームの開発において，矢印キーを押したままの状態でキャラクタをずっと移動させるには，キーリピートを有効にする．

キーリピートはデフォルトで有効になっている[*18]．

SDL_EnableKeyRepeat 関数

int SDL_EnableKeyRepeat(int delay, int interval)

- delay: リピートが開始されるまでの時間（ミリ秒）
- interval: リピート間隔の時間（ミリ秒）

delay，interval ともに 0 にすれば，キーリピートは無効になる．

返値として，設定に成功した場合は 0 を，失敗した場合は-1 を返す．

記述例

```
SDL_EnableKeyRepeat(1000,1000); // 1 秒間隔のキーリピート設定，1 秒後に開始
SDL_EnableKeyRepeat(0,0); // キーリピート無効を設定
```

[*18] SDL_KeyboardEvent 構造体の repeat の値が 0 以外になっている．

表 2.13　マウスボタン

ボタン	インデックス値
左	SDL_BUTTON_LEFT
中央	SDL_BUTTON_MIDDLE
右	SDL_BUTTON_RIGHT
戻る	SDL_BUTTON_X1
進む	SDL_BUTTON_X2

2.4.3　マウスイベントを処理する

マウスイベント処理を実現するために，マウスの移動に関する **SDL_MouseMotionEvent** 構造体とマウスのボタンに関する **SDL_MouseButtonEvent** 構造体が用意されている．

SDL_MouseMotionEvent 構造体

```
typedef struct{
    Uint32 type; // イベントの種類（SDL_MOUSEMOTION）
    Uint32 timestamp; // イベントのタイムスタンプ
    Uint32 windowID; // フォーカスのあるウィンドウ
    Uint32 which; // マウスインスタンス ID または SDL_TOUCH_MOUSEID
    Uint32 state; // ボタンの状態
    Sint32 x, y; // ウィンドウ内のマウスの x，y 座標
    Sint32 xrel, yrel; // マウスの x，y 方向の移動量（ピクセル）
} SDL_MouseMotionEvent;
```

SDL_MouseButtonEvent 構造体

```
typedef struct{
    Uint32 type; // イベントの種類（SDL_MOUSEBUTTONDOWN または SDL_MOUSEBUTTONUP）
    Uint32 timestamp; // イベントのタイムスタンプ
    Uint32 windowID; // フォーカスのあるウィンドウ
    Uint32 which; // マウスインスタンス ID または SDL_TOUCH_MOUSEID
    Uint8 button; // 状態の変化したボタン
    Uint8 state; // ボタンの状態（SDL_PRESSED または SDL_RELEASED）
    Uint8 clicks; // シングルクリック（1）または ダブルクリック（2）
    Sint32 x, y; // ボタンが押された／離された時点のマウスの x，y 座標
} SDL_MouseButtonEvent;
```

ボタンが押された場合は SDL_MOUSEBUTTONDOWN，離された場合は SDL_MOUSEBUTTONUP が type に格納される．状態の変化したボタンは，次のうちの 1 つである（表 2.13）．

マウスの状態を取得する

マウスはポインティングデバイスであることから，マウスの状態としてマウスの座標（マウスカーソルが指している座標）を取得したい．マウスの座標は **SDL_MouseMotionEvent** 構造体および **SDL_MouseButtonEvent** 構造体の x, y に，移動量のデータは **SDL_MouseMotionEvent** の xrel, yrel に格納される．以下に，マウスの座標と移動量などを表示するマウスイベント処理プログラムの一例（一部）を示す．

記述例

```
SDL_Event event;
while(1){
    if( SDL_PollEvent(&event) )
    {
        switch (event.type) {
        case SDL_MOUSEMOTION: // マウスが移動した時
            printf("Mouse moved by %d,%d to (%d,%d)\n",
event.motion.xrel, event.motion.yrel, event.motion.x, event.motion.y);
            break;
        case SDL_MOUSEBUTTONDOWN: // マウスボタンが押された時
            printf("Mouse button %d pressed at (%d,%d)\n",
event.button.button, event.button.x, event.button.y);
            switch(event.button.button){
            case SDL_BUTTON_LEFT: // 左ボタンが押された時
                printf("Left button pressed\n");
                break;
            }
            break;
        }
    }
}
```

マウスの状態を取得する関数として，SDL_GetMouseState 関数や SDL_GetRelativeMouseState 関数もあるが，上記のような方法でマウスの状態を取得できる．マウスのホイールに関する **SDL_MouseWheelEvent** 構造体もある．

SDL_MouseWheelEvent 構造体

```
typedef struct{
    Uint32 type; // イベントの種類（SDL_MOUSEWHEEL）
    Uint32 timestamp; // イベントのタイムスタンプ
    Uint32 windowID; // フォーカスのあるウィンドウ
    Sint32 which; // マウスインスタンス ID または SDL_TOUCH_MOUSEID
    Sint32 x; // 水平方向のスクロール量（正値が右，負値が左）
    Sint32 y; // 垂直方向のスクロール量（正値が奥，負値が手前）
    Uint32 direction; // SDL_MOUSEWHEEL_NORMAL または SDL_MOUSEWHEEL_FLIPPED
} SDL_MouseWheelEvent;
```

2.4.4 ジョイスティックイベントを処理する

アクションゲームやシューティングゲームなどジャンルによっては，入力デバイスとしてキーボードやマウスではなく，ジョイスティックを導入したい場合があるだろう．

ジョイスティックの導入には，**SDL_Event** 構造体に加えて，ジョイスティックを特定・利用するための **SDL_Joystick** 構造体も必要となる．そして，**SDL_Init** 関数で **SDL_INIT_JOYSTICK** を引数に指定し，SDL を初期化する．

記述例

```
SDL_Event event; // イベントを検知するための構造体
SDL_Joystick *joystick; // ジョイスティックを特定・利用するための構造体

SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK); // SDL 初期化（ビデオとジョイスティック）
```

利用可能なジョイスティックの数を取得する

SDL_NumJoysticks 関数

int SDL_NumJoysticks(void)

返値として，取得に成功した場合は（接続された）ジョイスティックの数を，失敗した場合は負値（エラーコード）を返す．

ジョイスティックをオープンする

SDL_JoystickOpen 関数

SDL_Joystick SDL_JoystickOpen(int device_index)*

- device_index: （オープンする）ジョイスティックの番号

返値として，オープンに成功した場合はジョイスティック識別子（**SDL_Joystick** 構造体）を，失敗した場合は NULL を返す．

表 2.14　ジョイスティックの状態（値）

状態	フラグ（値）
イベントを検知	SDL_ENABLE
イベントを検知しない	SDL_IGNORE
イベントの現態を取得する	SDL_QUERY

表 2.15　ジョイスティックイベント

イベントの種類	フラグ（値）
軸の状態が変化した（方向キーまたはアナログキー（スティック) が押された）	SDL_JOYAXISMOTION
ボタンが押された時	SDL_JOYBUTTONDOWN
ボタンが離された時	SDL_JOYBUTTONUP
トラックボールの状態が変化した	SDL_JOYBALLMOTION
ジョイスティックが接続された	SDL_JOYDEVICEADDED
ジョイスティックが外された（SDL2）	SDL_JOYDEVICEREMOVED

記述例

```
if(SDL_NumJoysticks() > 0)  // 接続されたジョイスティックがあるかチェック
{
    // ジョイスティックをオープンする
    joystick = SDL_JoystickOpen(0);
}
```

ジョイスティックイベントを有効または無効する

SDL_JoystickEventState 関数

int SDL_JoystickEventState(int state)

- state: ジョイスティックの状態（表 2.14）

返値として，ジョイスティックイベントの検知が有効の場合は 1，無効の場合は 0，失敗の場合は負値（エラーコード）を返す．

記述例

```
SDL_JoystickEventState(SDL_ENABLE); // ジョイスティックイベントを検知できるようにする
printf("%d\n", SDL_JoystickEventState(SDL_QUERY)); // 1 が表示される
```

ジョイスティックイベント処理は，キーボードやマウスのイベントと同様に **SDL_PollEvent** 関数を用いてイベントを取得し，type の値（表 2.15）に応じて処理を振り分ける．次いで，jaxis.axis や jbutton.button の値（どの軸なのか，どのボタンなのか）に応じてさらに処理を振り分ける．

以下に，ジョイスティックイベント処理プログラムの一例（一部）を示す．

記述例

```
while(1){
    if(SDL_PollEvent(&event)){
        switch (event.type) {
        case SDL_JOYAXISMOTION: // 方向キーまたはアナログキー（スティック）が押された時
            printf("The axis ID of the operated key is %d.\n",event.jaxis.axis);
// 操作された方向キーの方向軸を表示（0：アナログキー，1：アナログキー，2：方向キー左右方向，
3：方向キー上下方向）
            // 方向軸に応じた処理
            if(event.jaxis.axis==0){ ... }
            else if(event.jaxis.axis==1){ ... }
            else if(event.jaxis.axis==2){ ... }
            else if(event.jaxis.axis==3){ .... }
            printf("The axis value of the operated key is %d.\n",event.jaxis.value);
// ジョイスティックの操作された方向キーの値を表示（-32767（真左，真上）～32767（真右，真下））
            break;
        case SDL_JOYBUTTONDOWN: // ボタンが押された時
            printf("The ID of the pressed button is %d.\n", event.jbutton.button);
// 押されたボタンの ID を表示（0 から）
            // ボタン ID に応じた処理
                if(event.jbutton.button==0){ ... }
            break;
        case SDL_JOYBUTTONUP: // ボタンが離された時
            printf("The ID of the released button is %d.\n", event.jbutton.button);
// 離されたボタンの ID を表示（0 から）
            // ボタン ID に応じた処理
            if(event.jbutton.button==0){ .... }
            break;
        }
    }
}
```

ジョイスティックのイベント内容に応じて，**SDL_JoyAxisEvent** 構造体や **SDL_JoyButtonEvent** 構造体といったジョイスティックイベント用構造体うちの 1 つにデータが格納される．

SDL_JoyAxisEvent 構造体

```
typedef struct{
    Uint32 type; // イベントの種類（SDL_JOYAXISMOTION）
    Uint32 timestamp; // イベントのタイムスタンプ
    SDL_JoystickID which; // ジョイスティックのインスタンス ID
    Uint8 axis; // 状態が変化した軸
    Sint16 value; // 軸の現在位置（範囲: -32768〜32767）
} SDL_JoyAxislEvent;
```

SDL_JoyButtonEvent 構造体

```
typedef struct{
    Uint32 type; // イベントの種類（SDL_JOYBUTTONDOWN または SDL_JOYBUTTONUP）
    Uint32 timestamp; // イベントのタイムスタンプ
    SDL_JoystickID which; // ジョイスティックの（インスタンス）ID
    Uint8 button; // 状態が変化したボタン
    Uint8 state; // ボタンの状態（SDL_PRESSED または SDL_RELEASED）
} SDL_JoyButtonEvent;
```

ジョイスティックのインデックス／インスタンス ID を取得する

SDL_JoystickInstance 関数

*int SDL_JoystickInstance(SDL_Joystick *joystick)*

- joystick: （オープンした）ジョイスティック

返値として，取得に成功した場合はインスタンス ID を，失敗した時は負値（エラーコード）を返す．

ジョイスティックの方向キーの軸数を取得する

SDL_JoystickNumAxes 関数

int SDL_JoystickNumAxes(SDL_Joystick joystick)*

- joystick: （オープンした）ジョイスティック

返値として，取得に成功した場合は軸数を，失敗した時は負値（エラーコード）を返す．

ジョイスティックのボタンの数を取得する

SDL_JoystickNumButtons 関数

int SDL_JoystickNumButtons(SDL_Joystick joystick)*

- joystick: （オープンした）ジョイスティック

返値として，取得に成功した場合はボタン数を，失敗した時は負値（エラーコード）を返す．

ジョイスティックの軸の状態を取得する

SDL_JoystickGetAxis 関数

Sint16 SDL_JoystickGetAxis(SDL_Joystick joystick, int axis)*

- joystick: （オープンした）ジョイスティック
- button: 軸の番号（通常，x 軸が 0，y 軸が 1）

返値として，指定した軸の現在位置 (-32768〜32767) を返す.

ジョイスティックのボタンの状態を取得する

SDL_JoystickGetButton 関数

Uint8 SDL_JoystickGetButton(SDL_Joystick joystick, int button)*

- joystick: （オープンした）ジョイスティック
- button: ボタンの番号（0〜）

返値として，指定したボタンが押されている場合は 1 を，押されていない場合は 0 を返す.

記述例

```
printf("The joystick has %d axses.\n",SDL_JoystickNumAxes(joystick)); // 方向キー数を取得
printf("The joystick has %d buttons.\n",SDL_JoystickNumButtons(joystick)); // ボタン数を取得
printf("The joystick has %d Hat keys.\n",SDL_JoystickNumHats(joystick)); // Hat キー数を取得
printf("The joystick has %d balls.\n",SDL_JoystickNumBalls(joystick)); // ボール数を取得
```

2.4.5 ウィンドウイベントを処理する

ウィンドウイベントは，ユーザによるウィンドウの操作（例えば，位置やサイズ，フォーカスの変更）やプログラムによるウィンドウの設定変更（例えば，SDL_SetWindowSize 関数（SDL2）など，ウィンドウの状態が変化した際に発生する.

ウィンドウイベントデータ

ウィンドウイベントのデータは，**SDL_WindowEvent** 構造体に格納される.

表 2.16　ウィンドウイベント

イベントの種類	フラグ（値）
ウィンドウが見えるようになった	SDL_WINDOWEVENT_SHOWN
ウィンドウが隠れた	SDL_WINDOWEVENT_HIDDEN
ウィンドウが現れた	SDL_WINDOWEVENT_EXPOSED
ウィンドウが data1 から data2 へ移動した	SDL_WINDOWEVENT_MOVED
ウィンドウのサイズが data1 × data2 になった	SDL_WINDOWEVENT_RESIZED
ウィンドウのサイズが変化した	SDL_WINDOWEVENT_SIZE_CHANGED
ウィンドウが最小化された	SDL_WINDOWEVENT_MINIMIZED
ウィンドウが最大化された	SDL_WINDOWEVENT_MAXIMIZED
ウィンドウが通常の位置とサイズになった	SDL_WINDOWEVENT_RESTORED
ウィンドウがマウスのフォーカスを得た	SDL_WINDOWEVENT_ENTER
ウィンドウがマウスのフォーカスを失った	SDL_WINDOWEVENT_LEAVE
ウィンドウがキーボードのフォーカスを得た	SDL_WINDOWEVENT_FOCUS_GAINED
ウィンドウがキーボードのフォーカスを失った	SDL_WINDOWEVENT_FOCUS_LOST
ウィンドウマネージャが閉じることを要求した	SDL_WINDOWEVENT_CLOSE
ウィンドウにフォーカスを与えられた（SDL2.0.5 以降）	SDL_WINDOWEVENT_TAKE_FOCUS
ヒットテストが行われた（SDL2.0.5 以降）	SDL_WINDOWEVENT_HIT_TEST
（使用されない）	SDL_WINDOWEVENT_NONE

SDL_WindowEvent 構造体

```
typedef struct{
    Uint32 type; //イベントの種類（SDL_WINDOWEVENT）
    Uint32 timestamp; // イベントのタイムスタンプ
    Uint32 windowID; // フォーカスのあるウィンドウ
    Uint8 event; // イベントの内容（SDL_WIndowEventID）
    Sint32 data1; // イベントにより異なるデータ
    Sint32 data2; // イベントにより異なるデータ
} SDL_ExposeEvent;
```

SDL_WindowEvent 構造体のメンバである event には，SDL_WindowEvent 列挙体のフラグ（値）が格納される（表 2.16）.

「ウィンドウが現れた」というイベントは，ウィンドウのサイズや重なりが変化することで，ウィンドウの隠れていた部分が表出した場合に発生し，再描画の必要を生じさせる．通常，ウィンドウのサイズが変化すると，SDL_WINDOWEVENT_RESIZED → SDL_WINDOWEVENT_SIZE_CHANGED の順でイベントが発生するが，ユーザ操作やウィンドウマネージャ（OS）による変化の場合は逆の順序でイベントが発生する．

以下にウィンドウイベント処理プログラムの一例（一部）を示す．

記述例

```
while(1){
    if(SDL_PollEvent(&event)){
    // イベントの種類ごとに処理
    switch (event.type) {
        // ウィンドウイベント
        case SDL_WINDOWEVENT:
            // ウィンドウイベントの種類で処理を振り分け
            switch (event.window.event){
            case SDL_WINDOWEVENT_SIZE_CHANGED: // ウィンドウサイズが変化
                ....
                break;
            ....
            }
            break;
    }
}
```

2.4.6 タイマー割込イベントを処理する

タイマー割込イベントは，プログラム内の記述（設定）に基づいて OS の管理下で，一定の時間間隔で発生する．例えば，ゲームにおいて重要となるアニメーションの実現（ミリ秒単位で画像を切り替える）には，タイマー割込イベント処理は必須である．

タイマー割込イベントを発生させるには，**SDL_Init** 関数の引数に **SDL_INIT_TIMER** を指定して，SDL を初期化する必要がある．

記述例

```
SDL_Init(SDL_INIT_TIMER);
```

このように初期化しないと，コンパイルは通るものの，タイマーを利用できない．加えて，他のイベント処理と同様に，イベントループ（無限ループ）を用意していないと，タイマーが起動・実行される前に即座にプログラムが終了してしまい，結果的にタイマー割込が発生しない．イベントループ内で（処理が回っている間に）設定した時間間隔が経過すると，OS がタイマー割込を発生させる．タイマー割込イベント処理プログラムはこの割込の通知を受け取って，予め記述した処理（対応する関数）を実行する．

タイマーを作成する

SDL_AddTimer 関数

SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_TimerCallback callback, void param)*

- interval: コールバック関数を呼び出す時間間隔（ミリ秒）
- callback: コールバック関数（関数名）

- param: コールバック関数に渡す引数（callback に渡されるポインタ）

タイマー割込イベントを発生させるにはまず，**SDL_AddTimer** 関数でタイマーを作成（設定）する．複数のタイマーを作成可能で，作成されたタイマーにはタイマー ID が割り当てられ，**SDL_TimerID** 変数に格納される．コールバック関数はイベントハンドラに相当し，指定した時間間隔（interval）で呼び出される関数であり，次のような特徴を有している．

- コールバック関数名は自分で決めることができる．
- 受け取る引数は，整数型（Uint32 型）の interval と void 型ポインタ（汎用ポインタ）の*param である．void 型ポインタ（汎用ポインタ）を採用することで，どのような型の引数でも関数に渡すことができる．
- interval には，コールバック関数が呼び出されたタイミングが自動的に格納される．
- param はポインタ型であるため，引数（変数）のアドレスが格納される．
- param は void 型ポインタで受け取っているため，実際の引数の型にキャスト演算子[*19]で型変換して，新たな変数として格納する．
- 返値として，当該コールバック関数が次に呼び出されるタイミング interval（Uint32 型）が返される．

コールバック関数に渡す引数はポインタ型であるため，その引数（変数）のアドレスをコールバック関数に渡すことになる．引数を渡さない場合は，NULL を指定する．コールバック変数には構造体も渡すことができ，構造体に複数のメンバ（変数）を持たせることで，複数の変数をコールバック関数へ渡すことができる．

返値として，作成に成功した場合はタイマー ID を．失敗した場合は NULL（または 0）を返す．

1 秒（1000 ミリ秒）間隔で callbackfunc というコールバック関数を呼び出すタイマー割込は，以下のように記述される．なお，コールバック関数は別スレッド[*20]で実行される．

記述例

```
SDL_TimerID timer_id; // タイマ ID を格納する変数
timer_id = SDL_AddTimer(1000, callbackfunc, NULL); // 1000 ミリ秒間隔で引数なしで
callbackfunc を呼び出す

// イベントループ
while(1){
    .... // このループを回っている間に自動的にコールバック関数が呼び出される
}
```

1 秒間隔と 2 秒間隔で呼び出される 2 つの異なるコールバック関数を含むタイマー割込処理プログラムの一例を以下に示す．

[*19] 例えば，double 型の変数 a を int 型に変換して変数 b に格納したい場合は，キャスト演算子を用いて，b=(int)a; のようにする．
[*20] マルチスレッドプログラミングに関連するが，後日の授業で扱われることになっている．

記述例

```
Uint32 callbackfunc(Uint32 interval, void *param){
    printf("Callback Function 1: 1sec Passed\n");
    return interval;
}

Uint32 callbackfunc2(Uint32 interval, void *param){
    int *times = (int*)param; // 受け取った引数（パラメータ）をキャスト演算子で int のポイ
ンタ型に変換し、変数 times に格納
    printf("Callback Function 2: 2sec Passed (%d times)\n", *times);
    return interval;
}

int main(int argc, char* argv[]) {
    SDL_TimerID timer_id1, timer_id2; // タイマ ID を格納する変数
    int times;

    timer_id1 = SDL_AddTimer(1000, callbackfunc, NULL); // 1 秒おきにコールバック関数
を呼び出す（引数なし）
    timer_id2 = SDL_AddTimer(2000, callbackfunc2, &times); // 2 秒おきにコールバック
関数を呼び出す（int 型変数のアドレスを引数として渡す）
    while(1){
    }
    SDL_Quit();
}
```

通常，タイマー割込による呼び出しを継続させるために，コールバック関数の返値として interval を返すようにする．タイマー割込を終了する場合は，0 を返すようにする．

タイマーを削除する

SDL_RemoveTimer 関数

SDL_bool SDL_RemoveTimer(SDL_TimerID id)

- id: 削除するタイマーの ID

SDL_AddTimer 関数で作成したタイマーを削除するために，削除したいタイマー ID を引数に指定する．削除が成功した場合は True（SDL_TRUE）を，失敗した場合は False（SDL_FALSE）を返値とする．

経過時間を取得する

SDL_GetTicks 関数

Uint32 SDL_GetTicks(void)

SDL 初期化からの経過時間を取得する **SDL_GetTicks** 関数を用い，OS から取得した経過時間に応じた処理をイベントループ内に記述することで，タイマー割込を実現できる．

SDL 初期化からの経過ミリ秒数を返値とする．なお，経過時間はプログラム実行後，約 49 日間までは有効である．

記述例

```
unsigned int previous_time=0, current_time;
while(1){
    current_time = SDL_GetTicks(); //  経過時間を整数型変数に格納
    if (current_time > previous_time + 1000){ // 前のタイミングから 1 秒経過
        printf("1 sec Passed.\n");
        previous_time = current_time; // 前のタイミングを現在で更新
    }
}
```

処理を一定時間待機させる

SDL_Delay 関数

void SDL_Delay(Uint32 ms)

- ms: 待機時間（ミリ秒）

Unix 標準ライブラリ（unistd.h）の sleep 関数と同じように，処理を一時的に止めることができる．ただし，sleep 関数では秒を指定するのに対し，**SDL_Delay** 関数ではミリ秒を指定する．

2.5 画像描画

ゲームの視覚的な魅力を高める上で，キャラクタや背景などの画像は欠かせない．例えば，背景画像（背面）にキャラクタ画像（前面）を重ねて表示したいことが多々あるが，基本的に画像は矩形で表されるため，キャラクタ画像にはキャラクタ本体以外の部分，つまり，画面に表示したくない背景部分がどうしても存在する．このとき，キャラクタ本体だけが背景画像の前面に表示され，キャラクタの周りは背景画像に対して透過させる必要がある．SDL は背景透過画像を（背景透過して）表示することができる．

背景透過画像は，GIMP[*21]などのペイントソフトで作成することができ，PNG や GIF 形式の画像ファイルとして保存される．PNG や GIF 形式の画像ファイルは画像データに透過度（α 値）を含むことができ，ピクセル全体の α 値が α チャンネルとしてまとめられている．

SDL プログラミングでは，汎用的な画像ファイル（PNG，JPEG，BMP など）をサーフェイスに読み込んだ後に，描画処理することになる．具体的には，サーフェイスに読み込まれた画像ファイル（画像データ）が高速描画処理可能なテクスチャに転送され，最終的にレンダラー（ウィンドウと対応づけられている）を介してディスプレイに表示される．

2.5.1 画像ファイルを読み込む

汎用的な画像ファイルである BMP[*22]は，基本的にデータ圧縮されない画像形式のため，データの欠損がなく鮮明な画像になる．その反面，ファイルサイズが大きくなってしまうことから，読み込みには比較的大きなサーフェイス（メモリ領域）が必要であり，処理にも時間がかかる．

BMP ファイルを読み込む関数は用意されているが，多くの場合，PNG や JPG(JPEG) などのデータ圧縮された画像ファイル（ファイルサイズが比較的小さい）を読み込むことになるだろう．

SDL の基本機能で BMP ファイルを読み込む

SDL_LoadBMP 関数

SDL_Surface SDL_LoadBMP(const char* file)*

- file: BMP ファイル名

読み込みに成功した場合は SDL_Surface 構造体を，失敗した場合は NULL を返値とする．つまり，サーフェイスに画像ファイル（画像データ）を読み込むために，**SDL_Surface** 構造体が必要となる．

記述例

```
SDL_Surface *image;
image = SDL_LoadBMP("a.bmp"); // a.bmp ファイルを image サーフェイスに読み込む
```

サーフェイスの描画データを BMP ファイルとして保存する SDL_SaveBMP 関数もあるが[*23]，本教材では説明を割愛する．

[*21] https://www.gimp.org/

[*22] BMP ファイル（Microsoft Windows Bitmap Image）は画像全体のピクセルデータを格納した配列とみなすことができる．

[*23] この関数をゲームで使うなら，例えば，プレイ画面のスナップショット保存などに使えるだろう．

SDL_image サブシステムで画像ファイルを読み込む

汎用的な画像ファイル（BMP, GIF, LBM, JPG, PCX, PNG, PNM, TGA, TIF, XCF, XPM）を読み込むには，**SDL_image** サブシステムを用いることになる．SDL_image サブシステムは，画像ファイルの読み込みおよび判定（正しい画像形式かどうか）に特化した関数群を有しており，以下のようにインクルード文とコンパイルオプションを記述する．

記述例

```
#include <SDL2/SDL2_image.h>
```

コンパイルオプション

```
-lSDL2_image
```

IMG_Load 関数

*SDL_Surface *IMG_Load(const char *file)*

- file: 画像ファイル名

読み込みに成功した場合は SDL_Surface 構造体を，失敗した場合は NULL を返値とする．つまり，SDL_LoadBMP 関数と同様に，読み込んだ画像ファイルは **SDL_Surface** 構造体に画像データとして格納される．

記述例

```
SDL_Surface *image;
image = IMG_Load ("a.jpg"); // a.jpg を image サーフェイスに読み込む
```

2.5.2 画像を表示する

サーフェイスに読み込んだ画像をディスプレイに表示するには，**SDL_CreateTextureFromSurface** 関数でテクスチャ（VRAM 上の画像データ）を作成し，**SDL_RenderCopy** 関数でその画像データをレンダラーに転送して，**SDL_RenderPresent** 関数で VRAM に反映させることになる．以下に，画像データ（サーフェイス全体）をディスプレイに表示する処理の一例（一部）を示す．

記述例

```
SDL_Window* window = SDL_CreateWindow("Test",SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,320,240,0); // ウィンドウの生成
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0); // レンダラーの生成
SDL_Surface *image = IMG_Load("test.png"); // 画像の（サーフェイスへの）読み込み

SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, image); // 読み込んだ
画像からテクスチャを作成

// 画像描画（表示）のための設定
SDL_Rect src_rect = {0, 0, image1->w, image1->h}; // コピー元画像の領域（この場合，画像
全体が設定される）
SDL_Rect dst_rect = {0, 0, 100, 100}; // 画像のコピー先の座標と領域（x, y, w, h）

SDL_RenderCopy(renderer, texture, &src_rect, &dst_rect); // テクスチャをレンダラーに
コピー（設定のサイズで）
SDL_RenderPresent(renderer); // 描画データを表示
```

このように，特別な関数を使わず，背景透過画像を背景透過して（そのまま）表示することができる．

2.6 サウンド

ゲームにはBGMや効果音といったサウンド（オーディオ）は欠かせない．サウンド処理プログラミングには，SDLの基本処理ライブラリとSDL_mixerサブシステムを使う方法があるが，使い勝手がよいのは後者である[*24]．したがって，ここではSDL_mixerサブシステムについて，代表的な関数を取り上げながら説明する．なお，サウンドファイルのビット深度[*25]の違いで，同じファイル形式（拡張子）であっても再生できたりできなかったりすることがある．プログラムが正しい（コンパイルに成功している）にも関わらず，サウンドが再生されない場合は，ビット深度の異なるサウンドファイルで試してみるとよい．

サウンド処理プログラムは基本的に，SDL_mixerサブシステムを初期化し→オーディオデバイスを開き→WAVE（*.wav），MP3（*.mp3），MIDI（.mid）といった汎用的なサウンドファイルを読み込み→サウンドデータを再生したり停止したりする，という流れで実行される．

まずは，SDL_mixerサブシステムを用いる場合のインクルード文とコンパイルオプションの記述例を以下に示す．

記述例

```
#include <SDL2/SDL_mixer.h>
```

コンパイルオプション

```
-lSDL2_mixer
```

サウンドファイル（サウンドデータ）の読み込みには，Music型とChunk型[*26]がある．Music型では，サウンドファイルのデータをメモリに全て読み込まず，ファイルからデコードしながら再生することになる．同時に複数のMusic型サウンドは再生できないが，メモリを節約でき，BGMといったある程度長いサウンドの再生に用いられる．Music型は**Mix_Music**構造体[*27]（ポインタ）と**Mix_LoadMUS**関数を用いてサウンドファイルを読み込む．

一方，Chunk型は，サウンドデータをメモリに読み込んで再生することになる．容量の小さいサウンドファイルを対象とすることが多く，複数のサウンドを同時に再生できるため，高い即応性が要求される効果音によく用いられる．Chunk型は**Mix_Chunk**構造体（ポインタ）と**Mix_LoadWAV**関数を用いてサウンドファイルを読み込む．

[*24] 競合する関数が存在することから，両者を同時に使うのは避けるべきである．

[*25] サンプリングしたサウンドの分割データに与えられるデータ量のことで，サンプルビット数とも呼ばれる．ファイルのプロパティ（属性）を参照すれば，サンプリング周波数やビットレートを確認できるだろう．ビットレート（bps）＝サンプリングレート（Hz）×ビット深度（bit）×チャンネル数であり，非圧縮のWAVEファイルであればビットレートは1,411,200bpsとなる．チャンネル数はモノラル（1）かステレオ（2）に該当する．

[*26] チャンクとは”（意味のある）ひとまとまりのデータ”という意味である．

[*27] 不透明構造体となっており，メンバ（変数）は隠蔽されている．

表 2.17　SDL_mixer 初期化用フラグ

扱う（サポートする）サウンド形式	フラグ（値）
FLAC	MIX_INIT_FLAC
MOD	MIX_INIT_MOD
MP3	MIX_INIT_MP3
OGG	MIX_INIT_OGG

Mix_Music 構造体

```
typedef struct Mix_Chunk {
    int allocated; // チャンクを解放するときに abuf を解放するか
    Uint8 *abuf; // サンプリングデータへのポインタ
    Uint32 alen; // abuf のバイト長
    Uint8 volume; // 音量（0=無音，128 =最大音量）
} Mix_Chunk;
```

記述例

```
Mix_Music *music; // Music 型でデータを格納する変数を宣言
Mix_Chunk *chunk // Chunk 型でデータを格納する変数を宣言;
```

2.6.1　SDL_mixer を初期化する

SDL_image サブシステムは，**SDL_Init** 関数の引数に **SDL_INIT_AUDIO** を指定することで初期化する．

記述例

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
```

Mix_Init 関数

int Mix_Init(int flags)

- flags: 扱うサウンド形式（論理和で複数指定可）

必須ではないようだが，WAVE 形式以外のサウンドファイルを読み込む場合は，**Mix_Init** 関数でも初期化しておくとよい．扱うサウンド形式に対応する flags（表 2.17）で指定したライブラリを読み込んで初期化する．

記述例

```
Mix_Init(MIX_INIT_MP3); // MP3 ファイルを読み込むための初期化
```

表 2.18　format に指定できるフラグ

フォーマット（サンプル形式）	フラグ（値）
8bit 符号なし	AUDIO_U8
8bit 符号あり	AUDIO_S8
16bit 符号なし（リトルエンディアン）	AUDIO_U16LSB
16bit 符号あり（リトルエンディアン）	AUDIO_S16LSB
16bit 符号なし（ビッグエンディアン）	AUDIO_U16MSB
16bit 符号あり（ビッグエンディアン）	AUDIO_S16MSB
AUDIO_U16LSB と同等 (後方互換性用)	AUDIO_U16
AUDIO_S16LSB と同等 (後方互換性用)	AUDIO_S16
16bit 符号なし，システムのバイト順	AUDIO_U16SYS
16bit 符号あり，システムのバイト順	AUDIO_S16SYS または MIX_DEFAULT_FORMAT

2.6.2　オーディオデバイスを開く

SDL_Mixer サブシステムを初期化した後，**Mix_OpenAudio** 関数により，指定された条件でオーディオデバイス（ミキサー API）を開く．なお，SDL_mixer サブシステムを構成する関数は，関数名の冒頭に”Mix_”がつく．

Mix_OpenAudio 関数

int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)

- frequency: 出力サンプリング周波数
- format: 出力フォーマット（サンプル形式）
- channels: 出力チャンネル数
- chunksize: 出力チャンクサイズ（Byte）

frequency の一般的な値は 11,025Hz，22,050Hz，44,100Hz であり，大きいほど高音質でサウンドが出力される．22,050Hz がデフォルト値（推奨）となっており，デフォルト値を採用する場合は **MIX_DEFAULT_FREQUENCY** を引数に指定する．format は SDL が対応するオーディオに基づいており，ビット深度などに応じて指定する（表 2.18）．デフォルト値は，**MIX_DEFAULT_FORMAT** で表される AUDIO_S16SYS である．channels は出力チャンネル数すなわちモノラルかステレオを表し，モノラルであれば 1 を，ステレオであれば 2 を値に指定する．chunksize はサウンド再生に使うメモリサイズを表し，値が小さいと音飛びが発生するが，大きすぎると再生のタイミングが遅れるなど処理に影響を与える（メモリ容量に起因する）．1,024Byte がデフォルト値のようである．

返値として，初期化に成功した場合は 0 を，失敗した場合は-1 を返す．

記述例

```
if(Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, 2, 1024)==-1) {
    printf("Failed to open audio device: %s\n", Mix_GetError());
    exit(-1);
}
```

2.6.3 サウンドファイルを読み込む

Music 型で読み込む

Mix_LoadMUS 関数

*Mix_Music *Mix_LoadMUS(const char *file)*

- file: サウンドファイル名

Music 型であるため，サウンドファイルはメモリに全て読み込まれず，ファイルからデコードしながら内部のサウンドプレイヤーで再生される．外部のプレイヤーで再生する（コマンドで当該プレイヤーを呼び出す）場合は，Mix_SetMusicCMD 関数を用いるが，説明は割愛する（Web リファレンスを参照のこと）．

返値として，読み込みに成功した場合は Mix_Music へのポインタを，失敗した場合には NULL を返す．

記述例

```
Mix_Music *music;
music = Mix_LoadMUS("test.wav"); // test.wav を Music 型で読み込み
```

Chunk 型で読み込む

*Mix_Chunk *Mix_LoadWAV(char *file)*

- file: サウンドファイル名

WAVE，AIFF，RIFF，OGG，VOC ファイルを読み込むことができる．

返値として，読み込みに成功した場合は Mix_Chunk へのポインタを，失敗した場合には NULL を返す．

記述例

```
Mix_Chunk *chunk;
chunk = Mix_LoadWAV("test.wav"); // test.wav を Chunk 型で読み込み
```

2.6.4 サウンドを制御する

サウンドファイルの読み込み方法（Music 型か Chunk 型か）によって，サウンドを制御（再生，一時停止，停止など）する関数が異なる．

Music 型サウンドを再生する

Mix_PlayMusic 関数

*int Mix_PlayMusic(Mix_Music *music, int loops)*

- file: 再生するサウンド（Mix_Music 構造体）
- loops: ループ回数

読み込んだ Music 型サウンドを最初から最後まで loops 回再生する．再生中の Music 型サウンドが停止またはフェードアウトするまでは，別の Music 型サウンドは再生されない（待機状態となる）．

返値として，読み込みに成功した場合は Mix_Music へのポインタを，失敗した場合には NULL を返す．

記述例

```
Mix_PlayMusic(music, -1); // Music 型サウンドを無限（繰り返し）再生
```

その他，フェードインしてループ再生する Mix_FadeInMusic 関数，開始位置からフェードインしてループ再生する Mix_FadeInMusicPos 関数などがあるが，説明は割愛する．

Music 型サウンドの音量を設定する

Mix_VolumeMusic 関数

int Mix_VolumeMusic(int volume)

- volume: 音量（最小 0〜最大 128）

channels に-1 を指定すると，現在の音量を取得できる．volume が 128（MIX_MAX_VOLUME）より大きい場合は，音量は 128 になる．

返値は，設定前または現在の音量になる．

記述例

```
printf("Previous volume: %d\n", Mix_VolumeMusic(MIX_MAX_VOLUME/2)); // 音量を半分にして，前の音量を表示
printf("Current volume: %d\n", Mix_VolumeMusic(-1)); // 現在の音量を表示
```

Music 型サウンドを一時停止する

Mix_PauseMusic 関数

void Mix_PauseMusic()

記述例

```
Mix_PauseMusic(); // 再生中のサウンドを一時停止
```

Music 型サウンドの一時停止を解除する

Mix_ResumeMusic 関数

void Mix_ResumeMusic()

記述例

```
Mix_ResumeMusic(); // 一時停止を解除
```

表 2.19　Mix_GetMusicType 関数で得られるサウンドファイル形式

サウンド形式	フラグ（値）
WAVE/RIFF	MUS_WAV
MOD	MUS_MOD
MIDI	MUS_MID
OGG	MUS_OGG
MP3	MUS_MP3
コマンドでの音楽	MUS_CMD

Music 型サウンドを先頭まで巻き戻す

Mix_RewindMusic 関数

void Mix_RewindMusic()

サウンドが再生中，一時停止中，停止中のどの状態にあっても巻き戻しできる．ただし，MOD，OGG，MP3，MIDI（実機）のみ有効である．

記述例

```
Mix_RewindMusic(); // 先頭まで巻き戻す
```

開始位置を指定して再生する場合は，Mix_SetMusicPosition 関数を用いる．

Music 型サウンドを停止する

Mix_HaltMusic 関数

int Mix_HaltMusic()

サウンドの停止に伴って，コールバック関数を指定した Mix_HookMusicFinished 関数が呼び出され，停止処理に移行することができる．

返値として常に 0 を返す．

記述例

```
Mix_HaltMusic(); // 再生中のサウンドを停止
```

その他，サウンドをフェードアウトして停止させる Mix_FadeOutMusic 関数がある．

Music 型サウンドの形式を取得する

Mix_GetMusicType 関数

*Mix_MusicType Mix_GetMusicType(const Mix_Music *music)*

- music: 対象サウンド（Mix_Music 構造体）

返値として，music が NULL の場合は現在再生中のサウンドファイルの形式（表 2.19）を，再生中でない場合は MUS_NONE を返す．

記述例

```
// 再生中の音楽の形式を表示する
if(Mix_GetMusicType(NULL)==MUS_MP3){
    printf("MP3\n");
}
```

Music 型サウンドの再生状態を取得する

Mix_PlayingMusic 関数

int Mix_PlayingMusic()

返値として，再生中なら 1 を，再生中でないなら 0 を返す．なお，一時停止中かどうかは取得（確認）できない．

記述例

```
printf("Music is playing? %s\n", Mix_PlayingMusic()?"Yes":"No"); // 再生状態を取得・表示
```

その他，一時停止状態を取得する Mix_PausedMusic 関数，フェード状態を取得する Mix_FadingMusic 関数などがある．

Chunk 型サウンドを再生する

Mix_PlayChannel 関数

*int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)*

- channel: 再生するチャネル ID
- chunk: 再生するサウンド（Mix_Chunk 構造体）
- loops: ループ回数

channel（チャネル）は，“モノラルかステレオか”ではなく，サウンドを再生させるための窓口（割り当て）のようなものであり，チャネル ID を指定することで，チャネルごとにサウンドを再生したり停止したりできる．Chunk 型サウンドにチャネル ID を指定することで，以後はそのチャネル ID を介して個別の制御が可能になる．channel に-1 を指定すると，サブシステムが予約されていない最初の空きチャンネルを選択する．chunk には，Chunk 型で読み込んだサウンドデータ（**Mix_Chunk** 型構造体）を指定する．loops に-1 を指定すると無限に再生が繰り返される．1 回だけ再生したい場合は 0 を指定する．1 を指定した場合は，1 回ループすなわちサウンドは 2 回再生される．

返値として，再生に成功した場合は再生したチャンネルを，失敗した場合には-1 を返す．

記述例

```
Mix_PlayChannel(1, music, 0); // チャネル 1 のサウンドを 1 回のみ再生
```

その他，指定時間ループ再生する Mix_PlayChannelTimed 関数，フェードインしてループ再生する Mix_FadeInChannel 関数，フェードインして指定時間ループ再生する Mix_FadeInChannelTimed 関数がある．

Chunk 型サウンドの音量を設定する

Mix_Volume 関数

int Mix_Volume(int channel, int volume)

- channels: 音量を設定するチャネル ID
- volume: 音量（最小 0〜最大 128）

channels に-1 を指定すると，すべてのチャネルに対して音量を設定することになる．volume が 0 より小さい場合は，音量は設定（変更）されない．音量のデフォルト値は，最大の 128（MIX_MAX_VOLUME）になっている．

返値として，指定したチャネル ID の現在の音量を返す．-1 が指定されている場合は全チャネルの音量の平均値を返す．

記述例

```
Mix_Volume(1, MIX_MAX_VOLUME/2); // 音量を半分にする
printf("Mean volume: %d\n",Mix_Volume(-1,-1)); // 平均音量を表示
```

Chunk 型サウンドを一時停止する

Mix_Pause 関数

void Mix_Pause(int channel)

- channel: 一時停止する（再生中の）チャネル ID

channel に-1 を指定すると，すべてのチャネルのサウンドが一時停止する．

記述例

```
Mix_Pause(-1); // 全サウンドを一時停止
```

Chunk 型サウンドの一時停止を解除する

Mix_Resume 関数

void Mix_Resume(int channel)

- channel: 一時停止を解除するチャネル ID

channel に-1 を指定すると，一時停止中のすべてのチャネルのサウンドが再開する．

記述例

```
Mix_Resume(1); // チャネル ID が 1 のサウンドの一時停止を解除
```

Chunk 型サウンドを停止する

Mix_HaltChannel 関数

int Mix_HaltChannel(int channel)

- channel: 停止するチャネル ID

channel に-1 を指定すると，すべてのチャネルのサウンドが停止する．サウンドの停止に伴って，コールバック関数を指定した Mix_ChannelFinished 関数が呼び出され，停止処理に移行することができる．

返値として常に 0 を返す．

記述例

```
Mix_HaltChannel(1); // チャネル 1 のサウンドを停止
```

Chunk 型サウンドの再生状態を取得する

Mix_Playing 関数

int Mix_Playing(int channel)

- channel: 調査対象のチャネル ID

channel に-1 を指定した場合，返値として，再生中のチャネル数を返す．channel にチャネル ID を指定した場合，再生中ならば 1 を，再生中でないなら 0 を返す．

記述例

```
printf("Playing channels: %d\n", Mix_Playing(-1));
```

その他，一時停止状態を取得する Mix_Paused 関数，フェード状態を取得する Mix_FadingChannel 関数，再生中のサウンド（データへのポインタ）を得る Mix_GetChunk 関数がある．

2.6.5 サウンドを終了する

Music 型サウンドを解放する

Mix_FreeMusic 関数

*void Mix_FreeMusic(Mix_Music *music)*

読み込んだ Music 型サウンドを解放する．これに伴って再生中の Music 型サウンドは停止する（フェードアウト中であれば停止を待つ）．

記述例

```
Mix_FreeMusic(music);
music=NULL; // 解放したことを明示するために
```

Mix_FreeChunk 関数

*void Mix_FreeChunk(Mix_Chunk *music)*

読み込んだ Chunk 型サウンドを解放する．これに伴って再生中の Chunk 型サウンドは停止する．

記述例

```
Mix_FreeChunk(chunk);
chunk=NULL; // 解放したことを明示するために
```

オーディオデバイスを閉じる

Mix_CloseMixer 関数

void Mix_CloseAudio()

すべてのサウンドが停止し，オーディオデバイスが閉じられる．もちろん，**Mix_OpenAudio** 関数で再開できる．

記述例

```
Mix_CloseAudio();
```

SDL_mixer を終了する

Mix_Quit 関数

void Mix_Quit()

SDL_mixer サブシステムをメモリから解放する．再び SDL_mixer サブシステムを使用するには，**Mix_Init** 関数を呼び出すなどの方法がある．

記述例

```
Mix_CloseAudio();
```

2.7　アニメーション

図形や画像を単に描画するだけでは，表現力の乏しいゲームになってしまう．例えば，キャラクタの動きや感情の表現力を高めるために，さまざまなポーズや表情を滑らかに描画することが望まれる．つまり，ゲームにおいてアニメーションは必要不可欠といえる．

これまでに学んだ SDL プログラミングの内容から，アニメーション処理をどのようにプログラムするかは想像できるだろう．基本的にはパラパラ漫画と同じ要領であり，一定時間ごとに描画内容（コマ）を切り替えて描画・表示すればよい．このような処理には，タイマー割込イベントや画像描画を駆使することになる．

2.7.1　アニメーション処理の流れ

キャラクタの動作に焦点を当てたアニメーション処理の流れは，基本的には以下のようになる（図 2.8）．

1. キャラクタ画像や背景画像をサーフェイスに読み込む（**IMG_Load** 関数など）
2. タイマーを設定・起動させる（**SDL_AddTimer** 関数）
3. イベントループ（無限ループ）に入る
4. 設定した時間間隔でコールバック関数が呼び出される
5. 描画すべきキャラクタの動作パターンに対応する画像領域を抽出する
6. 抽出した領域の画像データを描画対象（サーフェイスやレンダラー）に転送する
7. 描画対象（ウィンドウサーフェイスやレンダラー）をディスプレイに出力（表示）する
8. 現在または次に抽出する画像領域（領域番号など）を変数に格納する
9. コールバック関数からイベントループに戻る

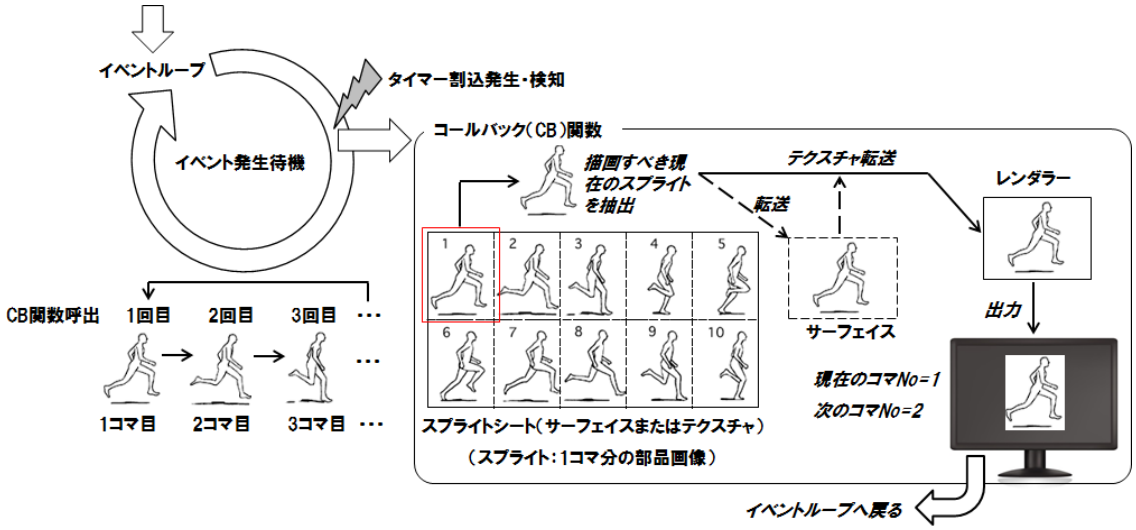

図 2.8　アニメーション処理の基本的な流れ

画像を用意する

アニメーションには，各コマに対応づけられるキャラクタの動作パターン（スプライト）を描いたキャラクタ画像（前景画像）が必要であり，一般的には，動作パタンをいくつかまとめて 1 枚の画像に描く（図 2.9）．このような画像は複数のスプライト領域（矩形領域）から構成される形となり，スプライトシート（またはキャラクタチップ）と呼ばれる．多くの場合，キャラクタ画像は背景透過画像（PNG ファイルなど）で描かれるだろう．GIMP による背景透過画像の作成については，**??**で概説する．

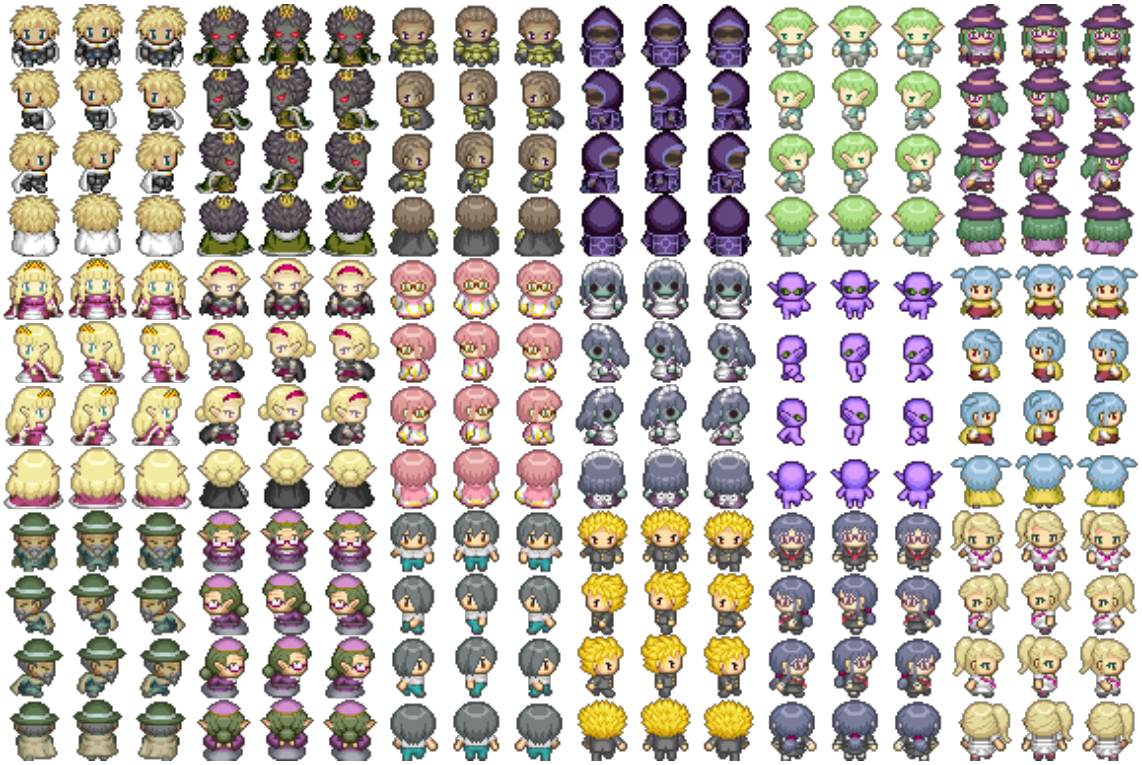

図 2.9　スプライトシート例（http://blog.pipoya.net/blog-entry-517.html）

スプライトシートは，背景画像にも適用できる．例えば，2 次元マップ上でゲームが展開する RPG[*28]では，草原，岩，海，城などを描いたスプライトが敷き詰められ，背景を構成している．このようなゲームにおいて，背景を構成するためのスプライトシートはマップチップと呼ばれる（図 2.10）．

スプライト領域を抽出する

アニメーション処理プログラムでは，一定時間ごとに，現在のコマに対応するスプライト領域を抽出することになる．よって，コマとスプライト領域の対応関係を予め把握しておくとともに，コマを管理する処理（例えば，現在または次はどのコマか，といったデータを記録しておく）が必要となる．例えば，幅 100 ×高さ 100 ピクセルのスプライトをキャラクタの種類別に横に 10 個，5 キャラクタ分埋めたとする[*29]．カウンター

[*28] 初期のドラゴンクエストやファイナルファンタジーに代表されるだろう．
[*29] このように，同じサイズのスプライトでスプライトシートを作成しておくと，アニメーション処理をプログラムしやすくなる．

図 2.10　マップチップ例（http://blog.pipoya.net/blog-entry-351.html）

変数を導入し一定時間ごと（タイマー割込発生のタイミング）に +1 して現在のコマを管理するならば，あるキャラクタの 1 コマ目として，抽出の基点座標（左上）を *(x, y)=(0, 0)* から幅・高さとも 100 ピクセルで抽出し，2 コマ目は *(x, y)=(100, 0)*，2 コマ目は *(x, y)=(200, 0)* ･･･のように規則的に基点座標をずらし，10 コマ目 *(x, y)=(900, 0)* の描画・表示を終えると，1 コマ目に戻るというアルゴリズムをプログラムすればよい．

描画データを転送する

抽出したスプライト領域の画像データは，描画対象（レンダラー）に転送される．この転送が画像の描画（表示まではされていない）に相当する．

背景画像の上に（前景として）キャラクタ画像を描画する場合，背景描画→キャラクタ画像の順で描画しなければならない（図 2.11）．複数のキャラクタ画像を重ねて描画する必要があるなら，もちろん，奥→手前の順で描画することになる．また，マップチップの部品を重ねて背景画像を構成する場合も，奥→手前の順で背景描画することになる（図 2.12）．

もし，キャラクタが背景画像上を移動した場合，直前（1 コマ前）のキャラクタ画像を消去する必要があり，この処理は当該領域を背景画像で上書き描画することで実現される．当該領域を考慮せず，ウィンドウやレンダラー（描画領域）全体の描画内容を一旦すべて消去して全体を再描画するという手法もある[*30]．

[*30] 一部しか描画（更新）しない場合であっても全体を消去・再描画するため，必ずしも効率的とはいえない．

アニメーションの 1 コマをディスプレイに表示させるには，一旦，背景画像にキャラクタ画像（スプライト領域）を重ねる形で，表示したいすべての画像データを不可視サーフェイスに転送する．逆を言えば，1 体ずつキャラクタ画像データを転送して即座にディスプレイに表示することをキャラクタの数だけ繰り返すことはしない．そのような逐次的な転送・表示では，画面のちらつきが発生しやすい．

SDL_BlitSurface 関数によるサーフェイス間転送のほか，**SDL_RenderCopy** 関数により，テクスチャの画像データをレンダラーに転送する（拡大縮小も可）．**SDL_RenderCopyEx** 関数を用いれば，画像データを拡大縮小・回転・反転させてレンダラーに転送できる．

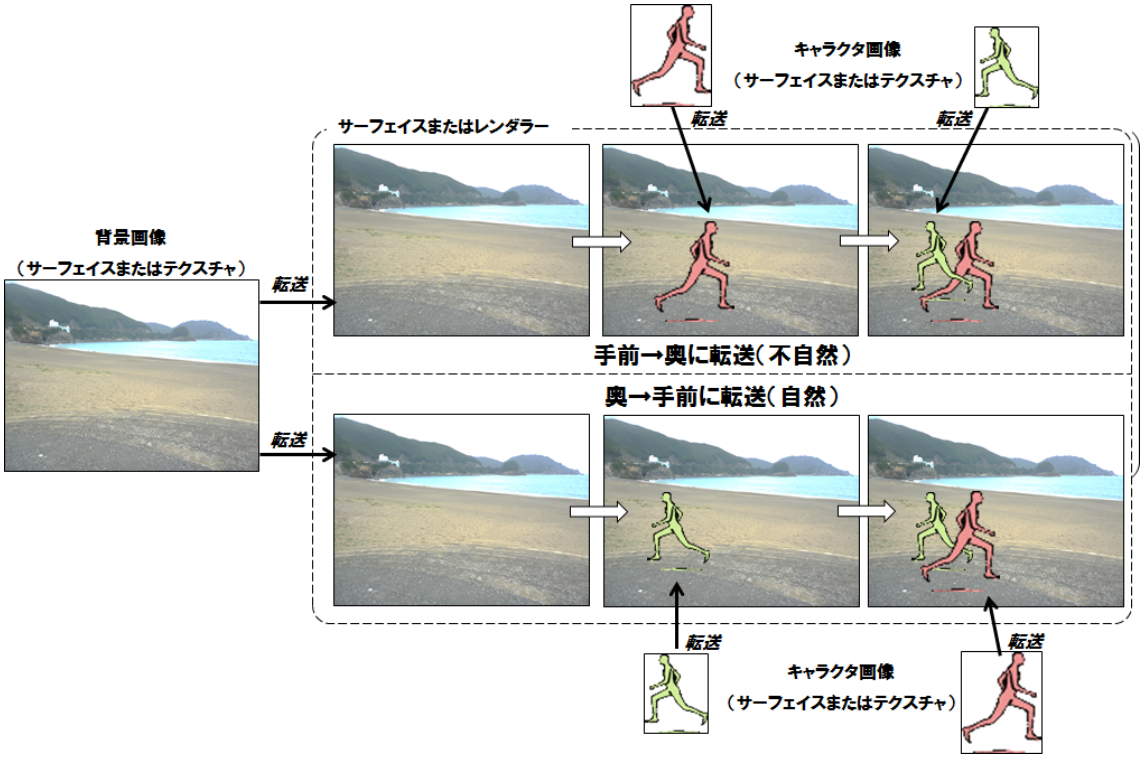

図 2.11　描画順を考慮した画像データの転送

ディスプレイに表示する

SDL_RenderPresent 関数によりレンダラーをディスプレイに出力する．

アニメーション処理プログラムの例は，manaba で公開しているサンプルプログラムを参照のこと．

2.7.2　ダブルバッファリング

アニメーションする複数のキャラクタが画面を縦横無尽に移動するゲームでは，キャラクタの描画・表示に伴って画面のちらつきが発生することがある．このようなちらつきの主な発生要因として，キャラクタ 1 体ずつに対して，その画像データを描画対象（ウィンドウサーフェイスやレンダラー）に転送し，その直後に描画内容をディスプレイに表示（出力）していることが考えられる．例えば，弾幕シューティングのように，自機に加えて無数の敵機や弾丸を描画する必要のあるゲームでは，ちらつき対策は必須といえよう．

図 2.12　マップチップの描画例（http://blog.pipoya.net/blog-entry-481.html）

ちらつき対策として有効なのが，ダブルバッファリングである[*31]．ダブルバッファリングとは，キャラクタを 1 体ずつ（逐次的に）描画対象に転送・表示するのはなく，すべてのキャラクタ（背景も含めて）を転送し終えてからディスプレイに表示する方法である（図 2.13）．描画対象であるレンダラーは，**SDL_RenderPresent** 関数が呼び出されるまでは，不可視の状態で描画内容の追加を受け入れるため，これらの関数を呼び出すタイミングに注意すればダブルバッファリングを実現できる．

レンダラーはウィンドウと 1 対 1 対応であるが，**SDL_CreateRGBSurface** 関数などによりダブルバッファリング専用の非可視サーフェイスを生成できる．つまり，生成したバッファに前もって描画データを格納しておき，その描画内容を必要なタイミングで即座に表示することを可能にする．RPG において例えば，フィールド画面，戦闘画面やステイタス画面を即座に切り替えるために，それらの描画データを前もってある程度バッファに格納しておけばよい．バッファに描画データを格納して表示するプログラムの例を以下に示す（詳細は manaba で公開しているサンプルプログラムを参照のこと）．

[*31] バッファとは，処理しきれなかったデータや後に出力するデータを一時的に格納しておくメモリ領域を意味する．

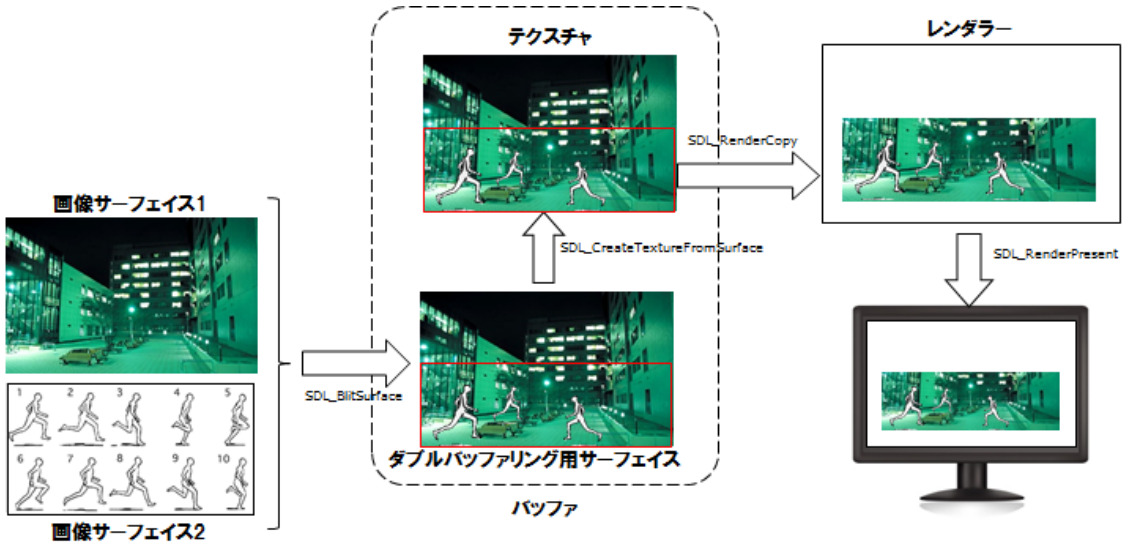

図 2.13　ダブルバッファリング

記述例

```
SDL_Surface *buffer, *image1, *image2;
SDL_Window *window;
SDL_Renderer *renderer;
SDL_Texture *texture, *texture1, texture2;
  ・・・中略・・・
buffer = SDL_CreateRGBSurface(SDL_HWSURFACE, 640, 480, 32, 0, 0, 0, 0); // ダブルバッファリング用サーフェイスを生成
image1 = IMG_Load("background.png"); // 背景画像
texture1 = SDL_CreateTextureFromSurface(renderer, image1); // 読み込んだ背景画像からテクスチャを作成
image2 = IMG_Load("character.png"); // キャラクタ画像
texture2 = SDL_CreateTextureFromSurface(renderer, image2); // 読み込んだキャラクタ画像からテクスチャを作成
SDL_BlitSurface(image1, NULL, buffer, NULL); // 先に背景画像をバッファに転送
SDL_BlitSurface(image2, NULL, buffer, NULL); // 次にキャラクタ画像をバッファに転送
  ・・・中略・・・
texture = SDL_CreateTextureFromSurface(renderer, buffer); // バッファ（描画内容）からテクスチャを作成
SDL_RenderCopy(renderer, texture, NULL, NULL); // テクスチャ（バッファから作成）をレンダラーに転送
SDL_RenderPresent(renderer); // レンダラーをディスプレイに表示
```