

2019年5月14日

ソフトウェア設定および実験 SDL2プログラミング (UI1)

光原弘幸 (情報系B4講座)
mituhara@is.tokushima-u.ac.jp

本日の授業 (12:50～16:05)

▶ 目的

- ▶ ゲームに必要な不可欠なUI (User Interface) のプログラミングを学び、ゲーム開発に必要な基本知識・スキルを身につける
 - ▶ ウィンドウ処理, 図形・文字描画処理, イベント処理, 画像描画処理
- ▶ SDL2プログラミングについて説明します
 - ▶ UIについての説明は省きます

▶ 流れ

1. manabaにアクセス
 - ▶ 第5・6回 UI1・2の授業資料, サンプルプログラムをダウンロード
- 2. スライドを使った説明を聞く
 - ▶ 主要な項目ごとに説明
3. サンプルプログラムを動作させる
4. 演習に取り組む
 - ▶ サンプルプログラムに演習内容が記載されている
5. レポート課題に取り組む

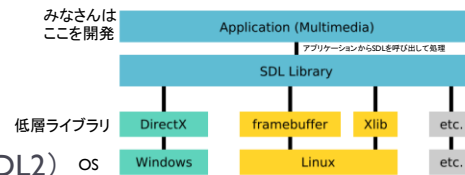


SDL2プログラミング

▶ SDL (Simple DirectMedia Layer)

- ▶ マルチメディアを簡単かつ直接に扱うためのライブラリ
 - ▶ グラフィックス、サウンド、ジョイスティック、スレッド、タイマなどの機能を扱うことができ、ゲーム開発に多く用いられる
 - ▶ 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
 - ▶ C言語だけでなくC++, JavaやPerlといった多くのプログラミング言語にもバインド

▶ <https://www.libsdl.org/>



- ▶ 2019年4月現在 Ver.2.0.9 (SDL2) OS
- ▶ ソフト実験では、SDL2を使う
 - ▶ SDL2では、グラフィックス系処理など多くの機能が拡張されている

SDL2プログラミング 事始め

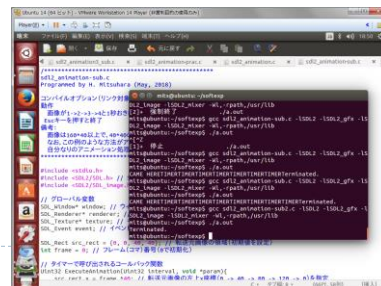
▶ 電算室ライクな開発環境を構築する

▶ 仮想マシンの導入

- ▶ あるOSの上で別のOSを動作可能にするソフトウェア
- ▶ 例. Windowsの上でUbuntuを動作させる

▶ VMware Workstation Player (以下, VMP)

- ▶ 非営利利用であれば無償の仮想マシンソフトウェア
- ▶ <https://www.vmware.com/jp.html> で「ダウンロード」をクリックし、当該ファイルをダウンロード
- ▶ インストーラに従ってインストール

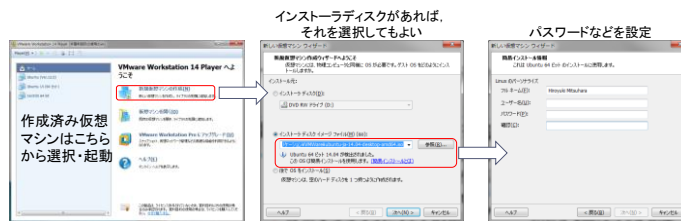


SDL2プログラミング 事始め

▶ 電算室ライクな開発環境を構築する

▶ VMP(Windows)にUbuntuをインストールする

1. Ubuntuのイメージファイル(ISOイメージ)を予めインストール
<https://www.ubuntulinux.jp/>
2. VMPを起動し、「新規仮想マシンの作成」からイメージファイルを選択



3. 作成済み仮想マシンを選択して, Ubuntuを起動



SDL2プログラミング 事始め

▶ SDL2のインストール

▶ Terminalを起動



▶ apt-getコマンドでSDL2をダウンロード・インストール

- 主要ないくつかのライブラリをインストールする場合

```
sudo apt-get install -y libSDL2-dev libSDL2-image-dev
libSDL2-mixer-dev libSDL2-net-dev libSDL2-ttf-dev
libSDL2-gfx-dev
```

- sudoコマンド: 指定したユーザでコマンドを実行する
 - デフォルト(ユーザ指定しなければ)rootユーザで
- apt-getコマンド: パッケージのインストール/更新
 - Debian系のディストリビューション(Ubuntuなど)で有効

SDL2プログラミング 事始め

- ▶ 電算室環境仮想イメージの個人PCへの導入方法
 - ▶ スライド3～5枚目に記述した開発環境構築の方法よりも、下記のURLに記述された方法で構築するのがベター
 - ▶ <http://netadm.iss.tokushima-u.ac.jp/soft/vm.html>
 - ▶ こちらのページ内容に従って構築してみてください
 - ▶ イメージファイルが大きいので、大容量のUSBメモリが必要



SDL2プログラミング基礎

- ▶ 簡単なプログラム
 - ▶ ウィンドウ表示プログラム例

```
#include <stdio.h>
#include <SDL2/SDL.h> // SDLを用いるために必要なヘッダファイルをインクルード

int main(int argc, char *argv[]){
    SDL_Init(SDL_INIT_VIDEO); // SDLの初期化
    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
    SDL_Delay(5000); // 5秒スリープ
    return 0;
}
```

SDL2では、SDL1.2で使用していた

- SDL_SetVideoMode関数
- SDL_WM_SetCaption関数などは廃止



SDL2プログラミング基礎

プログラムの構成

ヘッダファイルのインクルード

変数や関数のプロトタイプ宣言

- 関数では、引数や戻り値の定義

→ライブラリ内の変数や関数を利用するために必要

- 必要なヘッダファイルを適宜インクルード

例.

```
#include <SDL2/SDL.h> // 基本的処理系
#include <SDL2/SDL_image.h> // 画像処理系
#include <SDL2/SDL_mixer.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL2_gfxPrimitives.h>
#include <SDL2/SDL_net.h>
#include <SDL2/SDL_opengl.h>
```

SDL2プログラミング基礎

プログラムの構成

メイン関数

- これまでに習ったCプログラムと同じ
- コマンドライン引数をとるようにしておく
 - 実行ファイルを呼び出す際に引数を渡す
 - 例. プレイモードや難易度, プレイヤの名前や人数をコマンドで指定する

argc: 引数の総数(コマンド名, すなわちプログラム名も含む)

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

*argv: 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, ...) (文字列を指すポインタの配列. つまり, 引数を文字列として受け取る)

コマンド(実行ファイル)に easy, Mitsuahara, I を渡す

```
$ ./a.out easy Mits 1
```

コンパイル

- SDL2ライブラリをリンクするために、オプションが必要
- リンクするライブラリ(オブジェクトファイル)を適宜指定
 - SDL2使用時のオプション例.

```
$ 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
```

-lSDL2 libSDL2.aまたはlibSDL2.soをライブラリのリンク対象とする
-L/usr/local/lib ライブラリ検索ディレクトリのリストに/usr/local/libを追加

SDL2プログラミング基礎

▶ SDL2ライブラリを使いこなすことでプログラミング

- ▶ 多数用意されている関数からの確に選んで使用
- ▶ 細分化されているSDL2ライブラリ(サブシステム)

	SDL2.0	コンパイルオプション
SDLの基本処理を担当	SDL2	-ISDL2
画像処理を担当	SDL2_image	-ISDL2_image
音声処理を担当	SDL2_mixer	-ISDL2_mixer
フォント処理(TrueType Font)を担当	SDL2_ttf	-ISDL2_ttf
図形処理を担当	SDL2_gfx	-ISDL2_gfx
ネットワーク処理	SDL2_net	-ISDL2_net

- ▶ SDL2 <http://sdl2referencejp.osdn.jp/index.html>

SDL2の初期化と終了

▶ SDL2の初期化

▶ SDL_Init関数

- ▶ SDLの用途に応じて, いくつかのフラグ(整数)を指定する

- フラグは, 細分化されたライブラリ(サブシステム)に対応

- 初期化の記述例.

複数指定する場合は, | (論理和) で連ねる

```
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
    printf("failed to initialize SDL.¥n");
    exit(-1);
}
```

▶ SDL_Quit関数

- ▶ SDL2の終了

```
SDL_Quit();
```

主なフラグ

用途	フラグ(列挙体)
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_Window構造体
 - ▶ ウィンドウに関するデータを格納
- ▶ SDL_CreateWindow関数
 - ▶ ウィンドウを生成・表示

ウィンドウのY座標 横サイズ 縦サイズ フラグ(ウィンドウの種類)
(ピクセル) (0を指定すれば、リサイズ不可)

- ウィンドウの座標
 - 直接、座標値を引数に与えてもよい
 - `SDL_WINDOWPOS_CENTERED` 中央にウィンドウを配置する
 - `SDL_WINDOWPOS_UNDEFINED` 自動的に座標値を決めてくれる
- フラグ

ウィンドウの種類	フラグ(列挙体)
□ 主なフラグ	

ウィンドウの種類	フラグ (列挙体)	フラグは, (OR演算子) で列 挙することも可能
フルスクリーン	SDL_WINDOW_FULLSCREEN	
リサイズ可能	SDL_WINDOW_RESIZABLE	
OpenGL利用	SDL_WINDOW_OPENGL	

ウィンドウ処理

- ▶ サンプルプログラム (sdl2 window.c) を見る

```
#include <stdio.h>
#include <SDL2/SDL.h> // コンパイルオプション -lSDL2
int main(int argc, char *argv[]){
    int i;
    SDL_Window* window; // ウィンドウのデータを格納する構造体
    // ---中略---
    // SDL初期化(初期化失敗の場合は終了)
    if(SDL_Init(SDL_INIT EVERYTHING) < 0) {
        printf("failed to initialize SDL.¥n");
        exit(-1);
    }
    // ウィンドウ生成・表示
    window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, 0);
    // SDL_Window* window = SDL_CreateWindow("Test",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 640, 480, 0);
    // 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
```

ウィンドウ処理

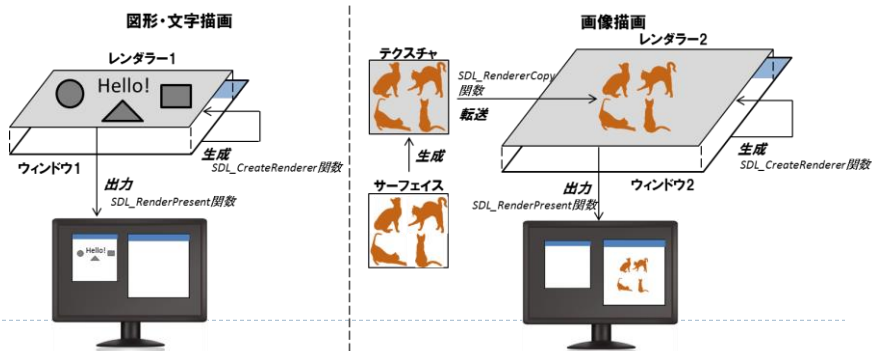
▶ サンプルプログラム(sdl2_window.c) つづき

```
// 描画処理
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_SetRenderDrawColor(renderer, 255, 255, 255, 0); // 生
成したレンダラーに描画色として白を設定
SDL_RenderClear(renderer); // 生成したレンダラーを白でクリア=
塗りつぶす(ただし、メモリに描画データを反映させただけなので、画面には表示
されない)
SDL_RenderPresent(renderer); // 描画データを表示
SDL_Delay(5000); // 5000ms(=5秒)待つ(sleep(5)と同等)
// ---中略---
SDL_DestroyWindow(window); // 生成したウィンドウの破棄(消去)
SDL_Delay(2000); // 2秒待つ
SDL_DestroyRenderer(renderer); // レンダラーの破棄(解放)
SDL_Quit(); // SDL使用の終了
return 0;
}
```

ウィンドウ処理

▶ レンダラー(レンダリングコンテキスト)

- ▶ 描画対象を意味する
 - ▶ 描画設定を格納するデータともいえる
- ▶ ウィンドウに1対1で対応づけられて生成される
- ▶ ウィンドウに何か描画するには、レンダラーの生成が不可欠



ウィンドウ処理

▶ レンダラーの生成

▶ SDL_Renderer構造体

- ▶ レンダーに関するデータを格納

```
SDL_Renderer* renderer; // レンダーデータへのポインタ
```

▶ SDL_CreateRenderer関数

- ▶ 生成したウィンドウに対してレンダーを生成

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

生成対象ウィンドウ
レンダリングドライバ
(通常, -1を指定)

▶ レンダーの描画色設定

▶ SDL_SetRenderDrawColor関数

- ▶ 生成したレンダーに描画色(RGBA)を設定

```
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 0);
```

フラグ
(0またはSDL_RendererFlags列挙体の値)

R G B A(α):255で不透明

設定対象レンダー

ウィンドウ処理

▶ レンダーのクリア(塗りつぶし)

▶ SDL_RenderClear関数

- ▶ 生成したレンダーを設定色でクリア(塗りつぶす)
- ▶ メモリに描画データを反映させただけなので、画面には表示されない

```
SDL_RenderClear(renderer);
```

クリア対象レンダー

▶ レンダーの描画内容表示

▶ SDL_RenderPresent関数

- ▶ レンダーの描画内容を表示する

表示対象レンダー

```
SDL_RenderPresent(renderer);
```

▶ レンダーの破棄(解放)

▶ SDL_DestroyRenderer関数

破棄対象レンダー

```
SDL_DestroyRenderer(renderer);
```

15分間演習

ウィンドウ処理

- ▶ サンプルプログラム(sdl2_window.c)を実行した後, 演習に取り組んでみる
 - ▶ (1)タイトルやサイズを変更してウィンドウを生成・表示する
 - ▶ (2)SDL_WindowFlags列挙体の値を変更してウィンドウを生成・表示する
 - ▶ (3)複数のウィンドウを生成・表示する
 - ▶ (4)矩形領域の描画設定(色やサイズ)を変えて描画する
 - ▶ (5)RCを生成しない場合(ウィンドウに描画処理をしない場合＝ウィンドウを生成・表示するだけの場合), どのような動作になるか確認する
- ▶ 時間に余裕があれば, sdl2_window2.cも実行・演習



図形描画処理

- ▶ ウィンドウ内に図形を描画するには
 - ▶ SDLの基本機能で図形描画する
 - ▶ SDL_FillRect関数
 - 矩形領域を塗りつぶす
 - ▶ SDL_RenderDrawPoint関数
 - 点を描画する(複数の点を描画するなら, SDL_RenderDrawPoints関数)
 - ▶ SDL_RenderDrawLine関数
 - 直線を描画する(複数の直線を描画するなら, SDL_RenderDrawLines関数)
 - ▶ SDL_RenderDrawRect関数
 - 塗りつぶさない四角形を描画する
 - ▶ SDL2_gfxで図形描画する
 - ▶ 図形描画用のSDL2_gfxサブシステムを用いる



図形描画処理

▶ SDL2_gfx

▶ ヘッダファイルのインクルード

```
#include <SDL2/SDL2_gfxPrimitives.h> // コンパイルオプション -ISDL2_gfx
```

▶ pixelColor関数

▶ 点を描画する

```
pixelColor(renderer, 100, 100, 0xff0000ff);
```

▶ lineColor関数

▶ 直線を描画する

```
lineColor(renderer, 50, 50, 200, 100, 0xff00ff00);
```

▶ rectangleColor関数

▶ 塗りつぶさない四角形を描画する

```
rectangleColor(renderer, 50, 50, 200, 100, 0xffff00ff);
```

□ 塗りつぶす四角形の描画は、boxColor関数

描画色



図形描画処理

▶ SDL2_gfx

▶ circleColor関数

▶ 塗りつぶさない円を描画する

```
circleColor(renderer, 200, 200, 50, 0xff0000ff);
```

□ 塗りつぶす円の描画は、filledCircleColor関数

▶ pieColor関数／filledPieColor関数

▶ 塗りつぶさない／塗りつぶす半円を描画する

▶ trigonColor関数／filledTrigonColor関数

▶ 塗りつぶさない／塗りつぶす三角形を描画する



色指定

▶ 色指定の方法

▶ 16進数で指定する

- ▶ 色深度が32bitの場合, R(赤), G(緑), B(青), A(α値)を指定
 - 各色成分に対して, 8bit=2⁸=256階調
 - 各色成分に, 0x00(0)~0xff(255)を指定
 - α値(透過度=不透明度)は, 0x00で透明, 0xffで不透明
- ▶ RGBAの並びは, エンディアン(バイトオーダー)に依存
 - ビッグエンディアンでは, RGBA
 - リトルエンディアンでは, ABGR

```
// ビッグエンディアン
pixelColor(renderer, 100, 100, 0x00ff00ff); // 緑
// リトルエンディアン
pixelColor(renderer, 100, 100, 0x00ff00ff); // 紫(ピンク?)
```

▶ 10進数で指定する

- ▶ 関数の引数として, 各色成分を0~255で指定

```
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 0);
```

文字描画処理

▶ ウィンドウ内に文字を描画するには

- ▶ 文字を画像化してウィンドウ内に描画する
- ▶ ただし, レンダーには画像を直接描画できない

→テクスチャに対して描画する

- ▶ テクスチャ on レンダー on ウィンドウ

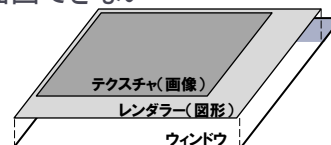
▶ テクスチャ

- ▶ 画像を描画する対象(VRAM上の描画データ)
 - ▶ VRAM上でGPUにより描画処理する仕組み
 - 高速かつ柔軟な描画処理(例. 拡大/縮小, 回転)が可能
 - ▶ 1つのレンダーに対して複数のテクスチャを生成できる

▶ SDL_Texture構造体

- ▶ テクスチャ(描画データ)を格納

```
SDL_Texture* texture; // テクスチャデータへのポインタ
```



文字描画処理

▶ テクスチャの生成

▶ SDL_CreateTexture関数

▶ テクスチャを生成する

対象(このテクスチャを対応させる)レンダラー
 ピクセル形式 → `texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_TARGET, 800, 600);`
 テクスチャのアクセスパターン テクスチャの幅 テクスチャの高さ

- ピクセル形式には, SDL_PixelFormatEnum列挙体の値を指定
- テクスチャのアクセスパターンには, SDL_TextureAccess列挙体の値を指定

SDL_PixelFormatEnum列挙体		SDL_TextureAccess列挙体	
ピクセル形式	フラグ(値)	アクセスパターン	フラグ(値)
不明	SDL_PIXELFORMAT_UNKNOWN	ほとんど変更無し	SDL_TEXTUREACCESS_STATIC
RGB各8ビット	SDL_PIXELFORMAT_RGB888	頻繁に変更される	SDL_TEXTUREACCESS_STREAMING
RGBA各8ビット	SDL_PIXELFORMAT_RGBA8888	レンダリング対象	SDL_TEXTUREACCESS_TARGET



文字描画処理

▶ テクスチャの生成

▶ SDL_CreateTextureFromSurface関数

- ▶ サーフェイス(描画データ)からテクスチャを生成する
- ▶ テクスチャ生成と同時に, サーフェイスの描画データを転送する
 - 画像を表示する場合, この関数を使ってテクスチャを生成することが多い

```

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作成
  
```

▶ サーフェイスとは何か？

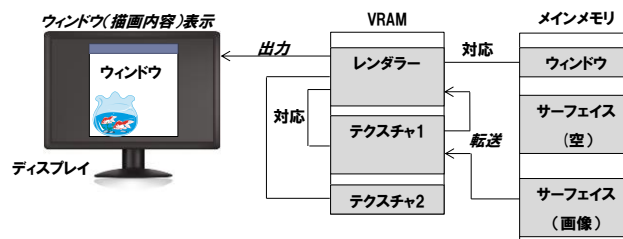
対象レンダラー テクスチャを生成したい画像データ
 が格納されているサーフェイス



文字描画処理

▶ サーフェイス

- ▶ 画像を描画する対象(メインメモリ上の描画データ)
 - ▶ フレームバッファ(メインメモリ領域)
 - ▶ サーフェイスの描画データはそのまま表示しない(≒不可視)
- ▶ 主に, 画像ファイルから画像データを読み込んで格納
 - ▶ テクスチャ(→レンダラー)に描画データを転送して, 描画データをウィンドウに表示



文字描画処理

▶ サーフェイス

- ▶ SDL_Surface構造体
 - ▶ サーフェイス(描画データ)を格納

```
SDL_Surface* surface; // サーフェイスデータへのポインタ
```

- ▶ SDL_Surface構造体の内部構造

```
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;
```

文字描画処理

▶ サーフェイス

▶ SDL_CreateRGBSurface関数

▶ サーフェイスを生成する

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

各色成分ごとのマスク(エンディアンに依存)

未使用(常に0) サーフェイスの幅
(ピクセル) サーフェイスの高さ
(ピクセル) 色深度
(ビット)

```
Uint32 rmask, gmask, bmask, amask;
// プリプロセスif(エンディアンによって定数
// を変える)
#ifdef SDL_BYTEORDER == SDL_BIG_ENDIAN
    rmask = 0xff000000;
    gmask = 0x00ff0000;
    bmask = 0x0000ff00;
    amask = 0x000000ff;
#else
    rmask = 0x000000ff;
    gmask = 0x0000ff00;
    bmask = 0x00ff0000;
    amask = 0xff000000;
#endif
```

▶ 実際のところ、画像ファイルから直接サーフェイスを生成することが多い

▶ IMG_Load関数

- 画像ファイルの画像データからサーフェイスを生成する(画像データをサーフェイスに格納)

文字描画処理

▶ サーフェイス

▶ SDL_BlitSurface関数

- ▶ サーフェイス(描画データ)を別のサーフェイスに転送する
- ▶ 描画データの矩形領域(部分)を指定して転送できる

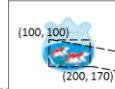
```
SDL_Surface *surface1, *surface2;
SDL_Rect src_rect = { 50, 100, 150, 200 }; // 矩形領域を
// (50,100)-(200,300) に設定
SDL_Rect dst_rect = { 200, 100 }; // 矩形領域の左上座標を指
// 定(幅と高さは省略)
SDL_BlitSurface(surface1, &src_rect, surface2, &dst_rect);
// surface1の描画データをsurface2に転送
```

▶ SDL_Rect構造体

- 矩形領域のデータを格納

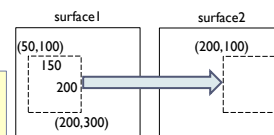
```
typedef struct{
    int x, y; // 矩形の左上座標(x, y)
    int w, h; // 矩形の幅(w) と高さ(y)
} SDL_Rect;
```

画像サーフェイス



```
SDL_Rect src_rect = { 50, 100, 150, 200 };
SDL_Rect dst_rect = { 200, 100 };
SDL_BlitSurface(surface1, &src_rect,
                 surface2, &dst_rect);
```

画像サーフェイス



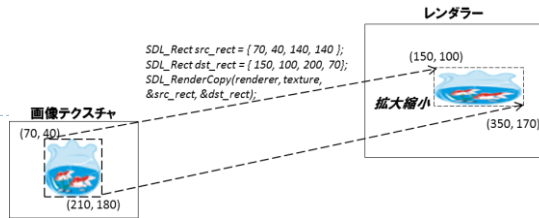
文字描画処理

▶ テクスチャ

▶ SDL_RenderCopy関数

- ▶ テクスチャの描画データをレンダラーに転送する
- ▶ 描画データの矩形領域(部分)を指定して転送できる
- ▶ 転送先の領域(サイズ)を指定することで拡大縮小もできる

```
SDL_Renderer* renderer =
    SDL_CreateRenderer(window, -1, 0);
SDL_Surface *image = IMG_Load("test.png");
SDL_Texture* texture =
    SDL_CreateTextureFromSurface(renderer, image);
SDL_Rect src_rect = {0, 0, 100, 100}; // 転送元画像の領域
SDL_Rect dst_rect = {0, 0, 200, 200}; // 画像の転送先の座標
と領域
SDL_RenderCopy(renderer, texture, &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. 2倍のサイズ)
SDL_RenderCopy(renderer, texture, NULL, NULL); // テクスチャ
全体をレンダラーに転送
```



文字描画処理

▶ テクスチャ

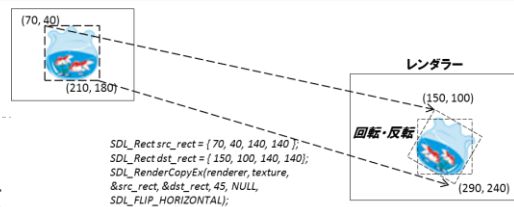
▶ SDL_RenderCopyEx関数

- ▶ 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(renderer, texture, &src_rect, &dst_rect,
    45, NULL, SDL_FLIP_HORIZONTAL);
```

回転角度(時計回り) 回転の中心座標 (NULLの場合、転送先領域の中心座標に設定される) 反転の種類(SDL_RendererFlip列挙体)

反転ボタン	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



文字描画処理

▶ ウィンドウ内に文字を描画するには

▶ SDL2_ttfで文字描画する

- ▶ 文字描画用のSDL2_ttfサブシステムを用いる
- ▶ TTF (True Type Font)
 - *.ttf(または*.ttc)というファイルがTrueType フォントのデータ
 - おそらくLinux では、/usr/share/fonts ディレクトリ内に存在
 - *.ttf ファイルをフォントビューアで開いてみればよい(アイコンをダブルクリック)

▶ ヘッダファイルのインクルード

```
#include <SDL2/SDL_ttf.h> // コンパイルオプション -lSDL2_ttf
```

▶ TTF_Font構造体

- ▶ フォントに関するデータを格納

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



文字描画処理

▶ 文字は画像化されて描画される

- ▶ 画像化された文字の描画データを格納する領域＝サーフェイスが必要

```
SDL_Surface *strings; // サーフェイスデータへのポインタ
```

▶ SDL2_ttf の主要な関数

▶ TTF_Init関数

- ▶ SDL2_ttfを初期化する

```
TTF_Init();
```

- 初期化に成功した場合は0, 失敗した場合は-1を返す

▶ TTF_OpenFont関数

- ▶ フォントを読み込む

フォントファイル名

フォントサイズ

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

- 読み込みが成功した場合は、フォントデータを返す(TTF_Font 構造体変数に格納)



文字描画処理

▶ SDL2_ttf の主要な関数

▶ TTF_Render○○_○○関数

▶ 文字列を画像化する(→文字描画する)

```
SDL_Surface *strings; R G B
SDL_Color white = {0xFF, 0xFF, 0xFF}; // フォントの色を白に
strings = TTF_RenderUTF8_Blended(font, "Hello!", white);
```

フォントデータ (描画したい)文字列 文字の描画色

- TTF_Render関数群(例. TTF_RenderUTF8_Solid関数)
- 使用する文字コードやフォントの荒さで, 接尾語が異なる
 - 文字の背景色を指定できる関数もある
- SDL_Color構造体

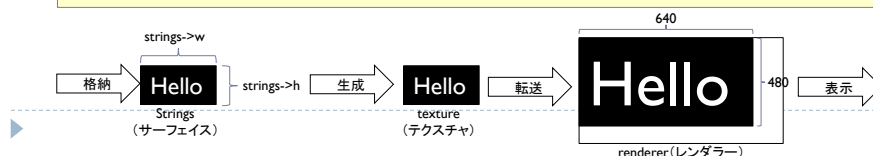
```
typedef struct{
    Uint8 r; // 赤
    Uint8 g; // 緑
    Uint8 b; // 青
    Uint8 unused;
} SDL_Color;
```

文字描画処理

▶ 文字描画プログラム例

```
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
```



図形・文字描画処理

▶ 終了処理

▶ SDL_DestroyWindow関数

- ▶ ウィンドウを破棄する

```
SDL_DestroyWindow(window);
```

▶ SDL_FreeSurface関数

- ▶ サーフェイスを解放する

```
SDL_FreeSurface(strings);
```

▶ SDL_DestroyRenderer関数

- ▶ レンダラーを破棄する

```
SDL_DestroyRenderer(renderer);
```

▶ SDL_DestroyTexture関数

- ▶ テクスチャを破棄する

```
SDL_DestroyTexture(texture);
```

▶ TTF_CloseFont関数

- ▶ フォントを閉じる

```
TTF_CloseFont(font);
```

▶ TTF_Quit関数

- ▶ SDL2_ttfの使用を終了する

```
TTF_Quit();
```



15分間演習

図形描画・文字描画処理

▶ サンプルプログラム(sdl2_draw.c)を実行した後、演習に取り組んでみる

- ▶ (1) サンプルプログラムの図形の色や形状を変更してみる
- ▶ (2) 現在のプログラミング環境(コンピュータ)のエンディアンを確認し、必要であれば、16進数による色指定を調整する
- ▶ (3) α 値を変更することによって、どのような描画になるか確認する
- ▶ (4) SDL2_gfxの関数により、さまざまな図形を描画してみる

▶ サンプルプログラム(sdl2_font.c)を実行したり、中身を見たり、演習に取り組んでみる

- ▶ (1) 好みのフォントを描画する
- ▶ (2) 「出力デバイスの違いに対応」の2つの関数をコメントアウトした際の動作を確認する
- ▶ (3) フォントを適切なサイズで描画する(拡大しすぎないように)
- ▶ (4) SDL_RenderCopyEx関数により文字列を回転・反転させる

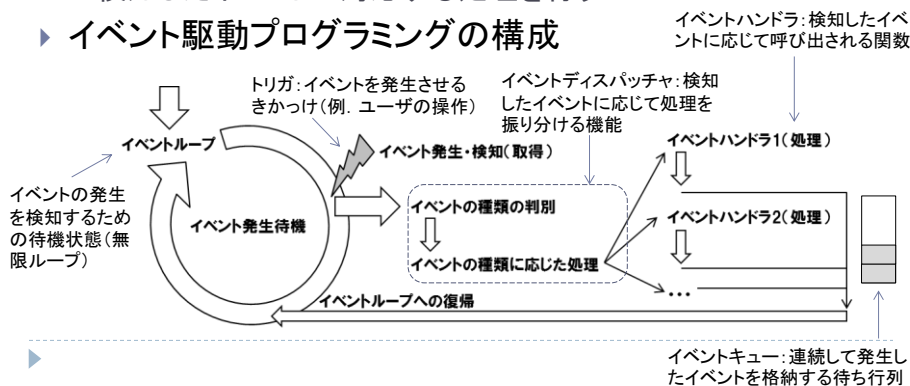


イベント処理

▶ イベント駆動型プログラミング

- ▶ 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
 - ▶ 例. 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
- ▶ 検知したイベントに対応する処理を行う

▶ イベント駆動プログラミングの構成



イベント処理

▶ 基本的な流れ

1. イベント処理の準備(イベント共用体宣言, SDL初期化)
2. イベントループ内でイベントの発生を待ち
 - ▶ そうしないと, 一瞬でプログラムが終了してしまう
3. 特定のイベントを検知して対応する処理を実行

イベントループへ

```
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
                ....
        }
    }
}
```

イベント処理

▶ SDL_Event共用体

▶ イベントデータを格納する

主要なイベントのみ記載

イベントごとに
用意され
た複数の構
造体

```
typedef union{
    Uint8 type; // イベントの種類
    SDL_KeyboardEvent key; // キーボードイベント
    SDL_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_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_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_Event;
```

▶ typeの値で処理を振り分ける

イベント処理

▶ 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
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
ウィンドウ	SDL_WINDOWEVENT
マウスホイール	SDL_MOUSEWHEEL

```
if(SDL_PollEvent(&event)){
    switch (event.type) {
        case SDL_MOUSEMOTION:
            ....
        case SDL_KEYDOWN:
            ....
    }
```

▶ まずは検知したイベントの種類をチェック

イベント処理

▶ イベントの検知

▶ SDL_PollEvent関数

- ▶ イベントキューから未処理のイベントを取得
- ▶ 未処理のイベントが存在する場合は1, 存在しない場合は0を返す

▶ SDL_WaitEvent関数

- ▶ 次のイベントが発生するまで無制限に待機
- ▶ 待機状態の場合は1, 待機中にエラーが発生した場合は0を返す

```
while(1){
    if ( SDL_PollEvent(&event) ){
        switch (event.type) {
            case SDL_MOUSEMOTION:
                ....
        }
    }
}
```

```
while(SDL_WaitEvent(&event)){
    switch (event.type) {
        case SDL_MOUSEMOTION:
            .....
    }
}
```

イベント処理

▶ 意図的なイベント発生

▶ SDL_PushEvent関数

- ▶ 意図的にイベントキューにイベントを追加する

```
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
```

- ▶ イベントを定義できる

```
SDL_Event event;
event.type = SDL_USEREVENT;
event.user.code = my_event_code; // コード(整数)
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
```

イベント処理

▶ キーボードイベント

- ▶ event.type
 - ▶ SDL_KEYDOWN(押された)
 - ▶ SDL_KEYUP(離された)
- ▶ event.key.keysym.symに、どのキーのイベントかが格納される

```
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;
    case SDLK_a: // aキーが押された場合
        ....
    }
}
```

イベント処理

▶ キーボードイベント

- ▶ 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
    Uint32 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;
```

- ▶ SDL_Keysym構造体

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

イベント処理

▶ キーボードイベント

▶ 主なキーコード(event.key.keysym.symの値)

イベント	フラグ(値)
Return (Enter)	SDLK_RETURN
Esc	SDLK_ESCAPE
スペース	SDLK_SPACE
Back Space	SDLK_BACKSPACE
Delete	SDLK_DELETE
↑, ↓, →, ←	SDLK_UP, SDLK_DOWN, SDLK_RIGHT, SDLK_LEFT
A, B, C, ...	SDLK_a, SDLK_b, SDLK_c, ...
キーパッドの0, 1, 2, ...	SDLK_KP_0, SDLK_KP_1, SDLK_KP_2, ...

```
switch(event.key.keysym.sym){
    case SDLK_ESCAPE: // Escキーが押された場合
        ....
}
```

イベント処理

▶ キーボードイベント

▶ キーリピートの設定

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)
- デフォルトでは, キーリピート有効

▶ SDL_EnableKeyRepeat関数

- キーリピートを設定する

リピートが開始されるまでの時間(ミリ秒)

リピート間隔の時間(ミリ秒)

```
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
```


イベント処理

▶ マウスイベント

- ▶ event.type
 - ▶ SDL_MOUSEMOTION(カーソルが移動した)
 - ▶ SDL_MOUSEBUTTONDOWN(ボタンが押された)
 - ▶ SDL_MOUSEBUTTONUP(ボタンが離された)
- ▶ event.button.buttonに、どのボタンのイベントかが格納される

```
switch (event.type) {
case SDL_MOUSEMOTION: // マウスが移動した時
    ...
    break;
case SDL_MOUSEBUTTONDOWN: // マウスボタンが押された時
    switch(event.button.button){
    case SDL_BUTTON_LEFT: // 左ボタンが押された時
        ...
        break;
```

ボタン	インデックス値
左	SDL_BUTTON_LEFT
中央	SDL_BUTTON_MIDDLE
右	SDL_BUTTON_RIGHT



イベント処理

▶ マウスイベント

▶ SDL_MouseMotionEvent構造体

一部変数を省略して記載

```
typedef struct{
    Uint32 type; // イベントの種類(SDL_MOUSEMOTION)
    Uint32 windowID; // フォーカスのあるウィンドウ
    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

一部変数を省略して記載

```
typedef struct{
    Uint32 type; // イベントの種類(SDL_MOUSEBUTTONDOWN または
    SDL_MOUSEBUTTONUP)
    Uint32 windowID; // フォーカスのあるウィンドウ
    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;
```



イベント処理

▶ マウスイベント

典型的なプログラム例

```
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;
}
```

イベント処理

▶ ウィンドウイベント

- ▶ ウィンドウの状態が変化した際に発生
 - ▶ 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)
 - ▶ プログラムによるウィンドウの設定変更(例えば, SDLSetWindowSize関数など)
- ▶ 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;
```

イベント処理

▶ ウィンドウイベント

主な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
ウィンドウがマウスのフォーカスを 得た／失った	SDL_WINDOWEVENT_ENTER SDL_WINDOWEVENT_LEAVE
ウィンドウがキーボードのフォーカスを 得た／失った	SDL_WINDOWEVENT_FOCUS_GAINED SDL_WINDOWEVENT_FOCUS_LOST



イベント処理

▶ ウィンドウイベント

典型的なプログラム例

```
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
                        変化
                        ....
                        break;
                    ....
                }
                break;
            }
        }
    }
}
```



15分間演習

イベント処理

- ▶ サンプルプログラム(sdl2_event.c)を実行した後、演習に取り組んでみる
 - ▶ (1)プログラムを他のイベントにも対応するように改良する
 - ▶ (2)マウスをクリックしている時だけ、点を描画するように改良する
 - ▶ (3)マウスホイールのイベント(操作)を検知できるようにする
 - ▶ (4)「○キーを押しながら□キーを押す」というイベント(操作)を検知できるようにする
 - ▶ (5)より多くのウィンドウイベントに対応する
 - ▶ (6)2つのウィンドウに対するイベントの動作を確認し、可能ならば改良する
 - ▶ (7)ジョイスティックがあれば、接続して動作を確認する



画像描画処理

- ▶ 処理の流れ
 1. 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む
 2. サーフェイスに読み込まれた画像ファイル(画像データ)をテクスチャに転送する
 3. レンダラー(ウィンドウと対応づけられている)に転送する
 4. レンダラーの描画データをディスプレイに表示する
- ▶ SDL2_imageで画像ファイルを読み込む
 - ▶ 画像ファイル読込用のSDL2_imageサブシステムを用いる
 - ▶ ヘッダファイルのインクルード

```
#include <SDL2/SDL_image.h>
// コンパイルオプション -lSDL2_image
```



画像描画処理

▶ 画像ファイルを読み込む

▶ IMG_Load関数

- ▶ 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
- ▶ 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

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

読み込む画像ファイル名

▶ 画像を表示する

1. SDL_CreateTextureFromSurface関数でテクスチャを作成
≡サーフェイスからテクスチャへ画像データを転送
2. SDL_RenderCopy関数でその画像データをレンダラーに転送
3. SDL_RenderPresent関数で画像を表示



画像描画処理

典型的なプログラム例

```
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

SDL_Rect src_rect = {0, 0, image->w, image->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
```



15分間演習

画像描画処理

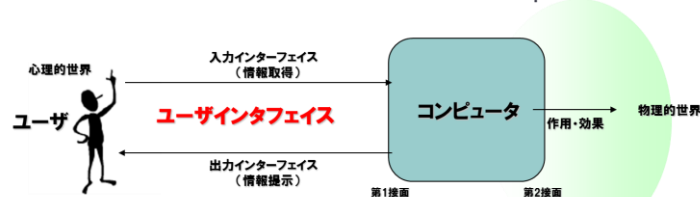
- ▶ サンプルプログラム(sdl2_image.c)を実行した後, 演習に取り組んでみる
 - ▶ (1) 画像の描画の位置やサイズを変更する
 - ▶ (2) test1.pngがtest2.pngより手前に来るように描画する
 - ▶ (3) 1つの画像でウィンドウを埋め尽くすように描画する
- ▶ サンプルプログラム(sdl2_image2.c)を実行した後, 演習に取り組んでみる
 - ▶ (1) 転送元と転送先の座標や領域を変更して描画する
 - ▶ (2) SDL_RenderCopyEx関数を使って, 回転・反転させて描画する
 - ▶ (3) 0.1秒おきに画像を回転して描画する



レポート課題を説明する前に

UI概説

- ▶ ユーザインタフェース (User Interface: UI)
 - ▶ 機械, 特にコンピュータとその機械の利用者(通常は人間)の間での情報をやりとりするためのインタフェースである。(Wikipediaより)



- ▶ UIを考える(設計・プログラムする)ということとは
 - ▶ “コンピュータを人間に合わせるにはどうすればよいか”を考えること
 - ▶ “人間に「使ってみよう」「使いやすい」と感じさせるにはどうすればよいか”を考えること
 - ▶ 良いUI→ユーザとコンピュータの調和による作業効率や安全性(≡現在社会におけるQuality of Life: QOL)の向上



ソフトウェアにおけるCUIとGUI

▶ CUI (Character User Interface)

- ▶ 利点: 複雑な処理をまとめて実行できる
 - ▶ 例. あるディレクトリのファイルをすべて削除せよ
- ▶ 欠点: コマンドを覚える必要がある
- ▶ エキスパート(専門家)に有益

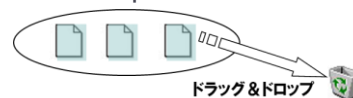
```
$ rm -fr dir
```



コンピュータの普及

▶ GUI (Graphical User Interface)

- ▶ 利点: アイコンやボタンの採用により, 直観的で分かりやすい
 = **ダイレクトマニピュレーション** (Direct Manipulation)
- ▶ 例. ファイルをすべて削除せよ



- ▶ 欠点: 特になし? (エキスパートには逆に使いにくい?)
- ▶ 多くのユーザに有益



認知工学に基づくUI設計

▶ 「使ってみたい」と思わせるには

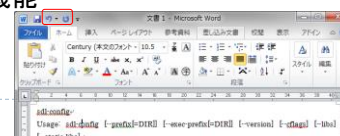
- ▶ 活性要素: 新奇性, 見た目の良さ, 社会的価値
- ▶ 不活性要素: 不安
 - ▶ 「使い方が分からない」「何ができるか分からない」「壊してしまいそう」

...


→ 不安を取り除く

▶ 求められる機能

- ▶ 対話機能(ウィザード)
- ▶ エラー防止機能
- ▶ フィードバックの明示化
- ▶ Undo(やり直せる)・Redo(再度できる)機能
- ▶ ヘルプ機能

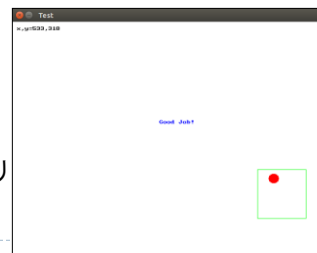


「使ってみたい」と思わせるには

- ▶ **アフォーダンス (Affordance)** – J.J. Gibson, D.A. Norman
 - ▶ 「物体の持つ属性(形, 色, 材質など)が物体自身をどう取り扱ったら良いかについてのメッセージをユーザに対して発している」
 - ▶ 取り扱い方法は, ユーザの頭の中にあるのではなく, 物体にある
 - ▶ ユーザは物体から取り扱い方法を受信して行動している
- 
- ▶ 初めて見る物体に対しても, 無意識的に瞬時に行動できる
→ 不安が取り除かれる
 - ▶ **UI設計にアフォーダンスを活用可能**

レポート課題

- ▶ **プログラミング課題1:ダイレクトマニピュレーション**
 - ▶ ウィンドウの生成と表示, 図形描画, マウス操作によるイベント駆動型プログラミングの習得を目的として, 次の要件を満たすプログラムを作成せよ. なお, 作成してほしいプログラムの実行ファイルをmanabaからダウンロードできるので, 確認すること.
 - ▶ 要件
 - ▶ ウィンドウに, 円と四角を描画する
 - ▶ マウス(カーソル)の現在座標をウィンドウ上に表示する (ヒント:sprintf関数)
 - ▶ 円をマウスでドラッグアンドドロップできる
 - ▶ 円を四角の中にドロップすると, “Good Job!”という文字列をウィンドウの中央付近に表示する
 - ▶ プログラムの終了条件を入れるなど, 自分なりのアレンジを加えても良い



レポート課題

- ▶ プログラミング課題2:アフォーダンスを考慮したGUI設計
 - ▶ 画像描画, 文字描画, キーボード操作によるイベント駆動型プログラミングなどの習得を目的として, 次の要件を満たすプログラムを作成せよ. なお, 作成してほしいプログラムの実行ファイルをmanabaからダウンロードできるので, 確認すること.
 - ▶ 要件
 - ▶ ウィンドウの上部に, アフォーダンスに満ちたボタンを3つ描画する. ボタンは図形描画または画像で作成すること.
 - ▶ 3つのボタンに文字を描画する. 文字内容やフォントは各自に任せる.
 - ▶ キーボードの“←”と“→”キーを押すことで, 3つのボタンから1つを選ぶようにする. その際, どのボタンが現在選択されているか分かるような視覚的効果を加えること.
 - ▶ “Enter/Return”キーを押すことで, 選択中のボタンに対応して以下の処理を行う.
 - ウィンドウ下部に画像を描画する.
 - 自分の好きな処理を行う.
 - プログラムを終了する.



レポート課題

- ▶ 調べ課題(自由課題:余裕のある人だけでOK)
 - ▶ メンタルモデルについて調べ, UI設計との関連について考察を述べよ.
- ▶ プログラミング課題に関する注意
 - ▶ 画像やサウンドをプログラムに取り入れる場合, それらの著作権や肖像権に十分配慮すること
 - ▶ 画像やサウンドのファイルサイズは極力小さくすること(大きすぎる場合, レポートファイルを受け取れない場合があります)

レポート課題

▶ 提出物

1. Latexで書いたレポートのファイル

- ▶ 下記のレポート要件を満たしていなければ、再提出となります
 - ▶ PDF形式で提出すること
 - ▶ レポートには、自分の名前や学籍番号、講義名などの情報を記載し、誰のどの講義に関するレポートか分かるようにすること
 - ▶ 実験ガイダンス資料・テクニカルライティングに従うこと
 - ▶ 講義の目的や取り扱った原理・理論(講義資料をコピー＆ペーストするだけでなく、自分なりにまとめる)、課題内容、課題の結果、考察を書くこと。特に、課題に関するキーワード(ダイレクトマニピュレーションやアフオーダンスなど)について調べ、説明を記述すること
 - ▶ 課題1、課題2の画面出力画像をレポートに入れ、その画像と対応づけながらプログラムの動作を説明すること
 - ▶ 講義の感想を記述すること
 - ▶ 参考にした参考文献はすべて記述すること



レポート課題

▶ 提出物

2. 課題1、課題2のプログラムソースと実行ファイル

- ▶ Makeファイル(作成していれば)、画像ファイルも含む
 - フォントファイル(*.ttf)は、外部の(ダウンロードした)無償＋配布可フォントの場合のみ含めてください
- ▶ プログラムソースには、コメントを多く入れること
- ▶ 提出物は、圧縮・アーカイブ化して提出してください
- ▶ 提出可能なファイル数、ファイルサイズには制限があります。確認の上、提出してください
- ▶ 圧縮・アーカイブ化したファイルは、解凍可能か確認してから提出してください

▶ 提出先

- ▶ manaba (<https://manaba.lms.tokushima-u.ac.jp/>)

▶ 提出期限

- ▶ manabaに記載(1週間後)
 - ▶ 未完成の状態でも、期限内に提出してください

