2019年5月21日

# ソフトウェア設定および実験
# SDL2プログラミング（UI2）

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
  - ゲームを彩るUI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
    - サウンド処理，アニメーション処理 （，アタリ判定）
- 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

# サウンド処理

- SDL2_mixerサブシステムを使う
  - 基本的な処理の流れ
    1. SDL2_mixerサブシステムを初期化
    2. オーディオデバイスを開き
    3. 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
    4. サウンドデータを再生したり停止したりする
  - ヘッダファイルのインクルード

```
#include <SDL2/SDL_mixer.h> // コンパイルオプション -lSDL2_mixer
```

  - SDL2_mixerの初期化

```
SDL_Init(SDL_INIT_AUDIO);
```

  - Mix_Init関数
    - WAVE形式以外のサウンドファイルを読み込む場合

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

フラグ値：MIX_INIT_FLAC, MIX_INIT_MOD, MIX_INIT_MP3, MIX_INIT_OGG

# サウンド処理

- サウンドデータの種類
  - Music型
    - 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(メモリを節約できる)
    - 同時に複数のMusic型サウンドは再生できない
      →BGM(ある程度長いサウンド)再生に用いられる
    - Mix_Music構造体(不透明構造体のためメンバ(変数)は不明)
  - Chunk型
    - サウンドデータをメモリに読み込んで再生
      - 容量の小さいサウンドファイルを対象とすることが多い
    - 複数のサウンドを同時に再生できる
      →効果音(即応性の高いサウンド)再生に用いられる
    - Mix_Chunk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;
```

# サウンド処理

ビット深度
- 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（サンプルビット数）
- 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

- オーディオデバイスを開く
  - Mix_OpenAudio関数
    - 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

出力サンプリング周波数　出力フォーマット　出力チャネル数　出力チャンクサイズ（byte）　返値（失敗：-1，成功：0）

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

- 出力サンプリング周波数：大きい値ほど高音質（例．44,100Hz）
  - デフォルトは，22,050Hz
- 出力フォーマット：ビット深度などに応じて指定
  - デフォルトは，AUDIO_S16SYS（16ビット符号あり．システムのバイト順）
- 出力チャネル数：モノラル（1）かステレオ（2）か
- 出力チャンクサイズ：サウンド再生に使うメモリサイズ
  - 小さい値だと，音飛びする
  - 大きい値だと，再生のタイミングが遅れる
  - デフォルトは，1,024byte

# サウンド処理

- Music型サウンドの読み込み
  - Mix_Music構造体の宣言

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

  - Mix_LoadMUS関数
    - サウンドファイルを読み込む

読み込むサウンドファイル名

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

      - 読み込みに成功した場合は，Mix_Musicへのポインタを返す
- Chunk型サウンドの読み込み
  - Mix_Chunk構造体の宣言

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

  - Mix_LoadWAV関数
    - サウンドファイルを読み込む

読み込むサウンドファイル名

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

      - WAVE，AIFF，RIFF，OGG，VOCファイルを読み込むことができる
      - 読み込みに成功した場合は，Mix_Chunkへのポインタを返す

# サウンド処理

- Music型サウンドを制御する
  - Mix_PlayMusic関数
    - サウンドを再生する

再生するサウンドデータ
ループ回数

```
Mix_PlayMusic(music, -1); // Music型サウンドを無限に再生
```

  - Mix_VolumeMusic型
    - サウンドの音量を設定する

音量（最小0～最大128）

```
printf("PreVol:%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));
```

-1を与えると現在の音量を返す

  - Mix_PauseMusic関数，Mix_ResumeMusic関数
    - サウンドを一時停止／一時停止解除する

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

# サウンド処理

- Music型サウンドを制御する
  - Mix_RewindMusic関数
    - サウンドを先頭まで巻き戻す

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

      - 再生中，一時停止中，停止中に関わらず巻き戻せる
      - 開始位置を指定して再生する場合は，Mix_SetMusicPosition関数
  - Mix_HaltMusic関数
    - サウンドを停止する

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

      - フェードアウトして停止させる場合は，Mix_FadeOutMusic関数
  - Mix_GetMusicType関数
    - サウンドの形式を取得する

対象サウンド（NULLの場合は，再生中のサウンド）

```
if(Mix_GetMusicType(NULL)==MUS_MP3){
    printf("MP3¥n");
}
```

# サウンド処理

- Chunk型サウンドを制御する
  - Mix_PlayChannel関数
    - サウンドを再生する

再生対象サウンド / チャネル / ループ回数

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

      - チャネル：Chunk型サウンド（第2引数）にチャネルIDを指定して，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
      - 再生対象サウンド：Chunk型のサウンドデータ
      - ループ回数：0を指定すると1回再生，1だと2回再生
  - Mix_Volume関数
    - 音量を設定する

設定対象チャネル / 音量

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

      - 設定対象チャネル：音量を設定するチャネル（-1を指定すると，すべてのチャネル）
      - 音量（最小0～最大128）

# サウンド処理

- Chunk型サウンドを制御する
  - Mix_Pause関数
    - サウンドを一時停止する

対象チャネル

```
Mix_Pause(-1); // 全サウンド（Chunk型）を一時停止
```

      - チャネルに-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
  - Mix_Resume関数
    - サウンドの一時停止を解除する

対象チャネル

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

  - Mix_HaltChannel関数
    - サウンドを停止する

対象チャネル

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

      - フェードアウトして停止させる場合は，Mix_FadeOutMusic関数
  - Mix_Playing関数
    - サウンドの再生状態を取得する

-1を指定すると，返値として再生中のチャネル数を返す
チャネルIDを指定すると，再生中なら1，再生中でないなら0を返す

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

# サウンド処理

- サウンドを終了する
  - Mix_FreeMusic関数
    - 読み込んだMusic型サウンドを解放する　対象のMusic型サウンドデータ

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

  - Mix_FreeChunk関数
    - 読み込んだChunk型サウンドを解放する　対象のChunk型サウンドデータ

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

  - Mix_CloseMixer関数
    - すべてのサウンドを停止し，オーディオデバイスを閉じる

```
Mix_CloseAudio();
```

  - Mix_Quit関数
    - SDL2_mixerサブシステムをメモリから解放する

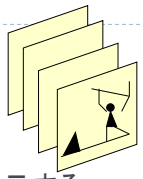
```
Mix_Quit();
```

15分間演習

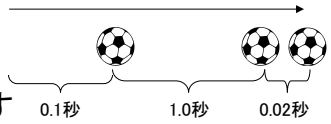
# サウンド処理

- サンプルプログラム（sdl2_sound.c）を実行した後，演習に取り組んでみる
  - （1）別のサウンドファイルを再生してみる
  - （2）BGMをキーまたはマウスボタンの押下により一時停止／解除できるようにする
  - （3）複数の効果音を用意し，キーの押下によりそれらが同時に再生されるようにする
  - （4）複数のBGMを用意し，マウスボタンの押下により切り替えて再生されるようにする

# アニメーション処理



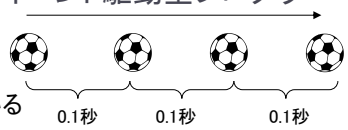
- どうやってアニメーションを実現するか？
  ↓
- パラパラ漫画の要領
  - 複数枚の画像を用意して，一定の時間間隔で次々に表示する
- どうやって“一定の時間間隔で次々に表示する”を実現するか？
  ↓
- Whileループの中で描画関数を呼び出す




  - マシンの性能やマシンの負荷に依存 ＝ コマ表示が安定しない
    - 性能の低いCPUやプロセスが多く走っている→アニメーションはゆっくり
    - 性能の高いCPUやプロセスが少ない→アニメーションは速い
  - SDL_Delayを入れると，他の処理を同時並行で実行できない

# アニメーション処理

- どうやってアニメーション処理を実現するか？
  - タイマー割り込みでアニメーション ＝ イベント駆動型プログラミング
    - マシンの性能や負荷に依存しない
      - OSにより割込のタイミングが管理されている




  - 処理の流れ
    1. タイマーを使うためにSDLを初期化
       - SDL_Init関数の引数に，SDL_INIT_TIMERを指定

```
SDL_Init(SDL_INIT_TIMER);
```

    2. SDL_AddTimer関数でタイマーを作成（設定）
    3. イベントループ（無限ループ）に入る
    4. コールバック関数を呼び出す
       - コールバック関数内でタイマー割込時の処理を実行する

# アニメーション処理

- タイマー割込イベント処理
  - SDL_TimerID変数を宣言

```
SDL_TimerID timer_id; // タイマID を格納する変数
```

    - タイマーIDを格納する変数
  - SDL_AddTimer関数
    - タイマーを作成（設定）する

割込間隔（ミリ秒）　コールバック関数　引数

```
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
```

      - 割込間隔：コールバック関数を呼び出す時間間隔（ミリ秒）
      - コールバック関数：呼び出されるコールバック関数名
      - 引数：コールバック関数に渡す引数（ポインタ）
        - 何も値を渡さない場合は，NULL
      - 作成成功の場合，返値としてタイマーIDが割り振られる（SDL_TimerID変数に格納）
      - タイマーは複数作成可能

# アニメーション処理

- タイマー割込イベント処理
  - コールバック関数の特徴

```
Uint32 callbackfunc (Uint32 interval, void *param){
    int *times = (int*)param;
    printf(“%d times Called¥n", *times);
    return interval;
}
```

```
timer_id = SDL_AddTimer(1000, callbackfunc, &n);
```

    - 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）の割込間隔（interval）とvoid型ポインタ（*param）
      - 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

# アニメーション処理

- タイマー割込イベント処理
  - 2つのコールバック関数を呼び出す
    - 1秒おきにcallbackfunc，2秒おきにcallbackfunc2（引数あり）

```
Uint32 callbackfunc(Uint32 interval, void *param){
    printf("1: 1sec Passed¥n");
    return interval;
}
Uint32 callbackfunc2(Uint32 interval, void *param){
    int *times = (int*)param;
    printf("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 n=0;
    timer_id1 = SDL_AddTimer(1000, callbackfunc, NULL); // 1秒おき
    timer_id2 = SDL_AddTimer(2000, callbackfunc2, &n); // 2秒おき
    while(1){ }
    SDL_Quit();
}
```

# アニメーション処理

- タイマー割込イベント処理
  - SDL_RemoveTimer関数
    - タイマーを削除する

削除するタイマー（ID）

```
SDL_RemoveTimer(timer_id);
```

  - SDL_GetTicks関数
    - SDL初期化からの経過時間（ミリ秒）を取得する

```
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
}
```

  - SDL_Delay関数
    - 処理を一定時間（ミリ秒）待機させる

```
SDL_Delay(5000);
```

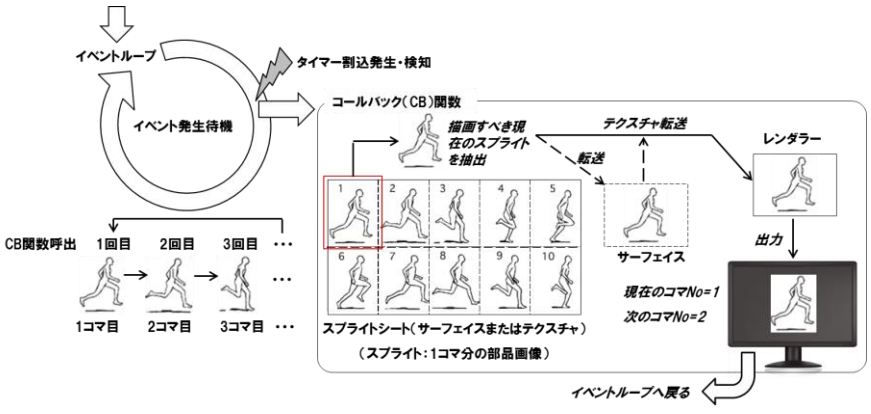
待機時間（ミリ秒）

# アニメーション処理

- アニメーション処理の流れ
  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（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

# アニメーション処理

- アニメーション処理の流れ（図）




  - 背景にキャラクタを重ねたい場合は，背景→キャラクタの順で描画

# アニメーション処理

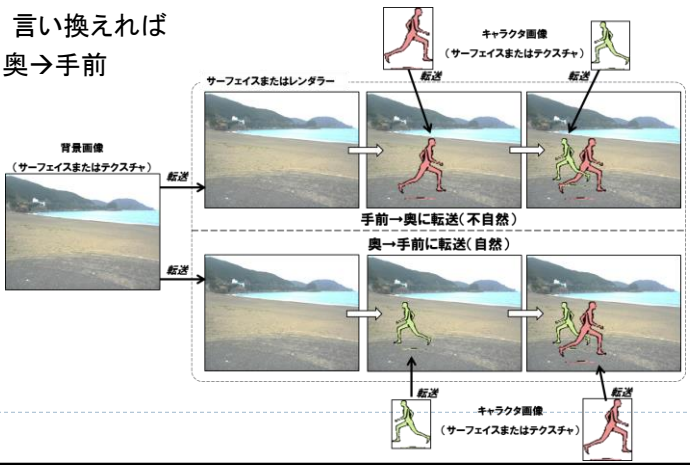
- 描画データの抽出
  - 一定時間ごとに，現在のコマに対応するスプライト領域を抽出する


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，複数キャラクタ分埋めたとする

  - カウンタ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
  - 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
  - 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す
  - 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

# アニメーション処理

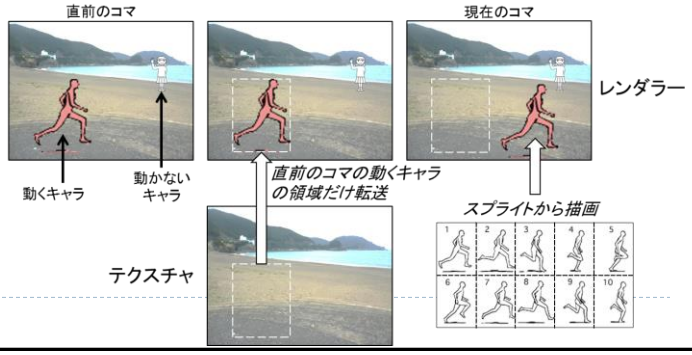
- 描画データの転送（テクスチャ→レンダラー）
  - 転送の順番を配慮する
    - 背景画像→前景画像（例．キャラクタ画像）
      言い換えれば
    - 奥→手前



# アニメーション処理

- 描画データの更新
  - キャラクタが背景画像上を移動した場合，どのように描画？
    - 移動先の座標にキャラクタを描画するだけだと，移動元（直前）のキャラクタ画像が描画されたままになる

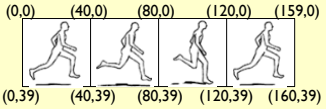
  →直前（1コマ前）のキャラクタ画像を消去する
    - 当該領域を背景画像で上書き描画する




当該領域を考慮せず，レンダラー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が，効率的でない場合がある

# アニメーション処理

- サンプルプログラム（sdl2_animation.c）を見てみる




```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
// -----グローバル変数-----
SDL_Renderer* renderer; // レンダラー（描画対象）
SDL_Texture* texture; // キャラクタ用テクスチャ
SDL_Event event; // イベントデータを格納する構造体
// -----タイマーで呼び出されるコールバック関数-----
Uint32 ExecuteAnimation(Uint32 interval, void *param){
    int *frame = param; // 受け取ったフレーム番号
    SDL_Rect src_rect = {0, 0, 40, 40}; // 転送元画像の領域（初期値を設定）
    src_rect.x = *frame *40; // 転送元画像の左上x座標（0->40->80->120->0）を指定
    SDL_RenderCopy(renderer, texture, &src_rect, NULL); // 転送元画像（テクスチャ）の指定した領域をレンダラーに転送
    // コールバック関数が呼び出されたことをユーザ定義イベントとしてイベントキューに追加
    event.type = SDL_USEREVENT; // イベントの種類をユーザ定義イベントにする
    event.user.code = 0; // 定義したイベントのコードに0を設定
    SDL_PushEvent(&event); // イベントキューにユーザ定義イベントを追加
    return interval;     // コールバックが呼び出される間隔を返す（必須）
}
```

# アニメーション処理

- サンプルプログラム（sdl2_animation.c） つづき

```
// メイン関数
int main(int argc, char* argv[]) {
    SDL_Window* window; // ウィンドウデータ
    SDL_Surface *image; // 画像データ（サーフェイス）へのポインタ
    SDL_TimerID timer_id; // タイマ割り込みを行うためのタイマのID
    int frame = 0; // フレーム（コマ）番号（0で初期化）
    // SDL初期化
    SDL_Init(SDL_INIT_EVERYTHING);
    IMG_Init(IMG_INIT_PNG | IMG_INIT_JPG); // 描画する画像形式を指定して初期化
    // ウィンドウ生成・表示
    window = SDL_CreateWindow("Animation Test", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 120, 120, 0);
    // レンダラー（レンダリングコンテキスト）作成
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_SOFTWARE); //
SDL_RENDERER_SOFTWAREを指定
    // アニメーション用画像読み込み
    image = IMG_Load("characters.png"); // 画像の読み込み
    texture = SDL_CreateTextureFromSurface(renderer, image); // 読み込んだ画
像からテクスチャを作成
```

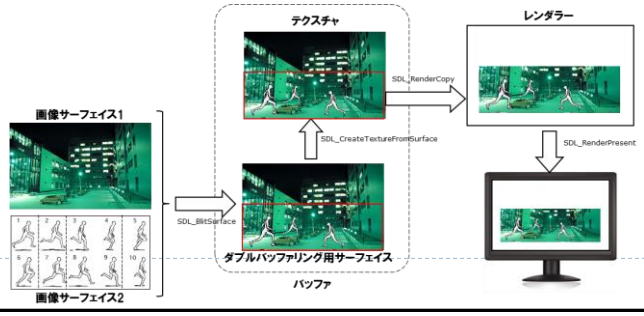
# アニメーション処理

- サンプルプログラム（sdl2_animation.c） つづき

```
    // タイマー作成
    timer_id=SDL_AddTimer(1000, ExecuteAnimation, &frame); // 1秒ごとにコール
バック関数を呼び出す（渡す引数はフレーム番号）
    // 無限ループ
    while(1){
        // イベント検知
        if(SDL_PollEvent(&event)){
            switch (event.type) {
            case SDL_USEREVENT: // ユーザ定義イベントなら（コールバック関数が呼び出
されたなら）
                // イベントコードを参照
                switch(event.user.code){
                case 0: // イベントコードが0なら
                    SDL_RenderPresent(renderer); // レンダラー（描画データ）を表示
                    frame ++; // フレーム番号を+1
                    if(frame == 4){ // フレーム番号が4になったら
                        frame = 0; // フレーム番号を初期化
                    }
                    break;
以下省略
```

# アニメーション処理

- ダブルバッファリング＝ちらつき対策
  - ちらつき：多くの画像（キャラ）をひとつずつ描画・表示することで発生（描画の過程を見せてしまっている）
  - 表示するすべての画像をレンダラーに転送し終えてから表示
    - SDL_RenderPresent関数を呼び出すタイミングを考える
    - サーフェイスも活用できる
      - 後で表示する描画データを先回りして描画し，サーフェイスに格納しておく




# アニメーション処理

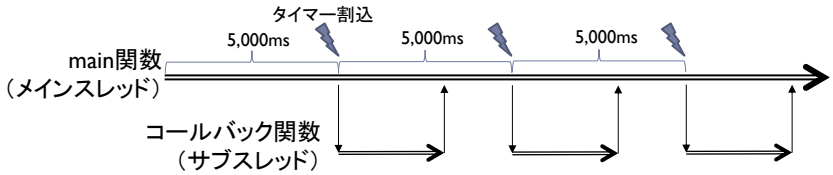
- ダブルバッファリング

サーフェイスを導入したダブルバッファリングの例

```
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);
image2 = IMG_Load("character.png"); // キャラクタ画像
texture2 = SDL_CreateTextureFromSurface(renderer, image2);
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
```

# Tips（コールバック関数の位置づけ）

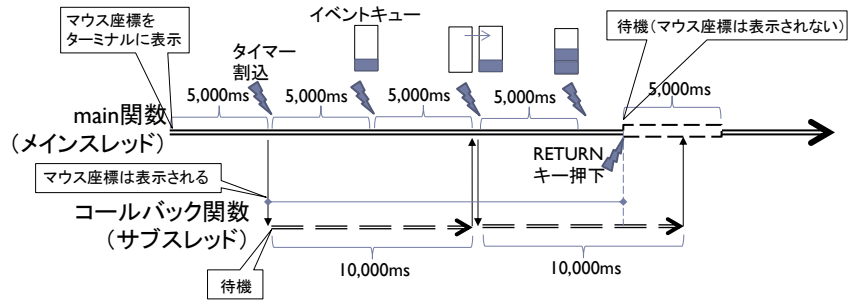
- コールバック関数は，メインスレッドと別のスレッド（サブスレッド）として同時並行で実行される
  - スレッド：処理の単位（別の回で扱います）
  - コールバック関数のあるプログラムは，マルチスレッドプログラム
  - main関数はメインスレッド




  - SDL_Texture構造体は同一スレッド内で扱う必要がある
    - テクスチャがらみの処理（レンダラーも含めた描画処理）はコールバック関数では扱えない？
      - 例．SDL_RenderPresent関数

# Tips（コールバック関数の位置づけ）

- sdl2_callback_test.cを開いて処理内容を確認し，実行してみる
  - ソースファイルは，UI1・2の資料としてmanabaに追加
  - コールバック関数の振る舞いと，メイン関数から呼び出された通常の関数の振る舞いの違いを確認してみよう



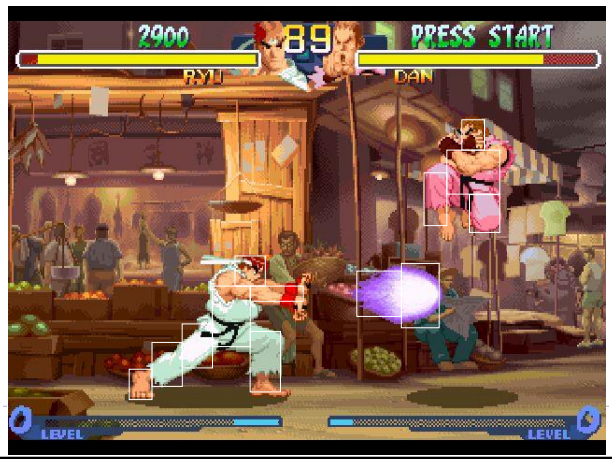
# アタリ判定

- アクションゲームやシューティングゲームに必須
  例えば
  - キャラクタ領域（四角）の重なりを検知


Street Fighter II

# アタリ判定

- アクションゲームやシューティングゲームに必須
  例えば
  - Iキャラクタに領域を複数用意



# アタリ判定

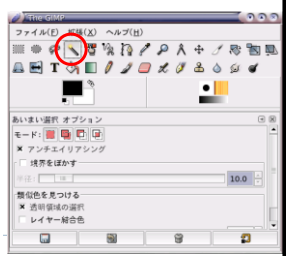
- アクションゲームやシューティングゲームに必須
  例えば
  - キャラクタの中心線の重なりを検知


演習に入る前に1

# 背景透過画像ファイルの作成

- GIMPを利用した背景透過画像ファイルの作成手順
  1. GIMPで画像ファイルを作成（または開く）

  2. アルファチャンネルを追加
     - メニューから，【レイヤー】→【透明部分】→【アルファチャンネルを追加】を選択
  3. ツールから隣接領域の選択ボタンを選択
     - 新規画像の場合は，画像全体を選択・削除



- このスライドに掲載したUIと電算室のGIMPのUIは異なるが，基本的な操作・手順は同じと思われる

# 背景透過画像ファイルの作成

- GIMPを利用した背景透過画像ファイルの作成手順
  4. 透過させたい背景部分を選択
     - （背景部分が点線で示される）

  5. 選択した背景部分を消去
     - メニューから，【編集】→【消去】または【切り取り】
     - （透過させる背景部分が格子模様で表現される）
  6. PNG等の形式で保存

演習に入る前に2

# コンピュータゲーム（ゲーム）

- ゲームは，単なる子どもの遊びではない
  - リハビリテイメント（リハビリ+エンターテイメント）
    - ゲームにより，高齢者や障害者が楽しみながら機能回復やトレーニングを行う
  - エデゥテイメント（教育+エンターテイメント）
    - ゲームにより，楽しみながら学習すること
  - その他
    - ソーシャルサイエンス（シチズンサイエンス）
      - Foldit
        - たんぱく質構造予想（Protein Structure Prediction：PSP）のための分散コンピューティング型パズルゲーム
        - ランキング機能（世界中のプレイヤと競える）
        - 専門家が約10年費やしても解けなかったPSPに関する問題を世界中のプレイヤがこのゲームを通じて3週間以内に解いた！
    - 健康科学，教育学，心理学，社会学，経済学などの研究対象

STOP DISASTER

Hazmat: HotZone

Foldit

# コンピュータゲーム（ゲーム）

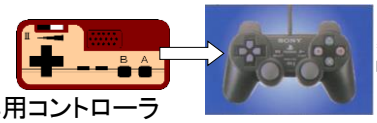
情報工学的には

- 最新のハードウェア／ソフトウェア技術を駆使したシステム／アプリケーション
  - AI・アルゴリズム
    - 囲碁，将棋，チェス，群衆シミュレーション　など
  - 3次元CG技術
    - 最近，3DCGが当たり前になってきている
  - VR技術
    - SONY PlayStation VR，Oculus Quest　など
  - ネットワーク技術
    - MMOG (Massively Multiplayer Online Game)
      - MMORPG　など
  - ユーザインタフェイス技術
    - いろいろ（VR技術も範疇に入る）




# ゲームの入力UI

- コンシューマゲーム
  - PC
    - キーボード，マウス，ゲームパッド　など
  - ゲーム専用機
    - 標準コントローラ（ゲームパッドなど）
    - 専用コントローラ





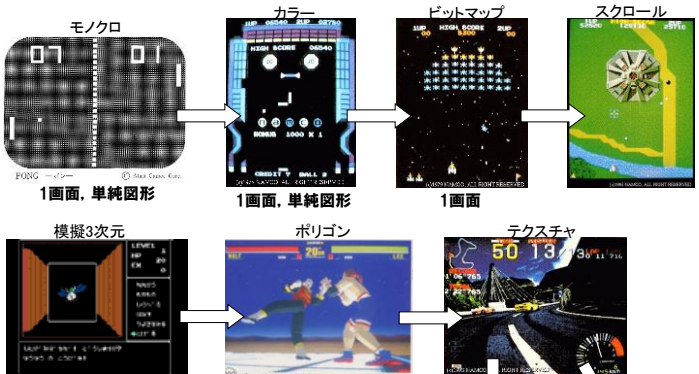



# ゲームの出力UI

- 映像

2次元

3次元




- サウンド
  - 電子音→FM音源→MIDI →CDデジタル

# ゲームのためのUI

- ソフトウェア
  - 「使ってみたい」「使いやすい」と感じさせるユーザインターフェイス
- ゲーム
  - 「使ってみたい」「使いやすい」はもちろん，「面白い」と感じさせるUI
    - 「面白さ」はゲーム特有の要素
  - 「面白い」と感じさせる要素には何があるか？

# ゲームのためのUI

- ソフトウェア
  - 「使ってみたい」「使いやすい」と感じさせるユーザインターフェイス
- ゲーム
  - 「使ってみたい」「使いやすい」はもちろん，「面白い」と感じさせるUI
    - 「面白さ」はゲーム特有の要素
  - 「面白い」と感じさせる要素には何があるか？
    - 没入感，爽快感，達成感などを最大限に引き出すUIを設計
    - ゲームの面白さはUIの良し悪しに依存することがある

  例えば
    - 没入感を高めるグラフィック ---- リアリティ
    - 爽快感を生み出す単純な操作 ---- 操作性
    - 達成感を生み出す難しい操作 ---- 操作性
    - （ストーリーを盛り上げる音楽） など

考えてみよう！
“くそゲー”だと思うゲームはUIに原因があるか？

# リアリティ

- 映像技術やサウンド技術の飛躍的進歩により，ゲームのリアリティは現実世界に近づきつつある
- リアリティを追求するシステム
  - シミュレーション
  - ゲーム

飛行機操縦シミュレーション

本物に忠実

目的：パイロットの訓練・育成

飛行機操縦ゲーム


簡略化

目的：パイロットの疑似体験

- すべてをリアルにするのではなく，対象を絞ってリアルにすることでゲームの面白さを向上することができる

# 操作性

- ゲーム内容や求める面白さで操作性を考える
  - 例．シューティングゲーム
    - 「シューティングゲームは爽快感が重要．マウス操作は難しそうだし，爽快感もないなぁ・・・」
      → ゲームパッドまたはキーボード
    - 「今までにない面白さのシューティングゲームを作りたい．爽快感よりも達成感を重視したい」
      →「操作は難しいけどマウス操作のシューティングゲームを作ってみるか！」

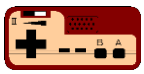
操作が単純＝爽快感が向上
⇕
操作が複雑＝達成感が向上

    - 操作性の観点から爽快感と達成感のバランスをとる
    - どちらか一方に特化して，他の要素（ゲームの難易度，効果音など）でもう一方を補う

# 2次元と3次元の操作性

- 3次元ゲームの操作は複雑化する
  - 2次元のゲームの場合
    - 操作は比較的単純（直感的）

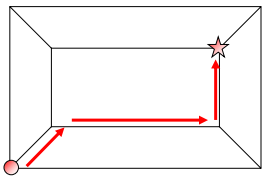
操作は比較的単純（直感的）



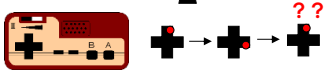
  - 3次元のゲームの場合
    - 操作は複雑

● ボタン数の増加
● 操作パタンの増加


2次元のゲームの場合



3次元のゲームの場合




3次元のゲームを開発する際は，操作性についてよく考える必要がある．

## アニメーション処理

残りの時間
演習＋レポート課題

- サンプルプログラム（sdl2_animation.c）を実行した後，演習に取り組んでみる
  - （1）タイマー割込の時間間隔を変えてアニメーション表示する
  - （2）オリジナルの画像を作成してアニメーション表示させる
  - （3）背景画像に前景（キャラクタ）画像を重ねてアニメーション表示させる

  →レポートのプログラミング課題に相当

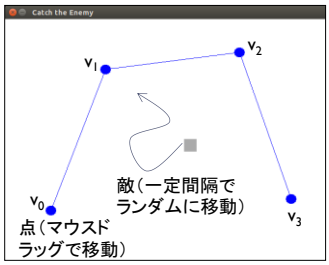
- sdl2_animation2.c，sdl2_animation3.cも実行した後，演習に取り組んでみる
- サンプルプログラム（sdl2_dbuffer.c）を実行しみる

## レポート課題

- プログラミング課題1：アニメーション
  - オリジナルのアニメーションを作成せよ
    - サンプルプログラムsdl2_animation.cをベースにしてもよい
      - ウィンドウサイズは大きくしたほうがよい
    - サウンドをつけるといった工夫を期待する
  - 注意
    - アニメーション動作中にコアダンプして，大きなcoreファイルが作成され，次回ログインができなくなる等の事例があった
    - ディスク使用量の限界があるので，容量を超過しないよう以下のように注意すること
      - コアダンプしたときには（デバッグ等に使用しない場合には），coreファイルを削除する
      - ゴミ箱に移動させた場合，ゴミ箱を空にする

# レポート課題

- プログラミング課題2：アタリ判定（自由課題：余裕のある人だけでOK）
  - 以下のルールをもつ簡単なゲーム“Catch the Enemy”を開発せよ
    - ウィンドウ内に4つの点（v0, v1, v2, v3）から構成される線があり，敵が1体いる
    - ユーザは4つの点をマウスドラッグで移動させ，線の形状を変えることができる
    - 敵は一定間隔でランダムに移動する
    - ユーザが4つの点から形成される3角形で敵を囲めばゲームクリアになる（v0≒v3）
    - 敵が4つの点から構成される線に触れるとゲームオーバーになる




  - マルチスレッドプログラミングに挑戦してみてもよいだろう
    - まだ習っていないが…

# レポート課題

- 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．アニメーションの原理は必須），課題内容，工夫した点，課題の結果，考察を書くこと．
       - 課題1（取り組んだなら課題2も）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
         - 提出されたプログラムを実行しなくても，どのようなアニメーションなのかレポートを読めば分かるように説明すること
         - 課題2については，どのようなアタリ判定アルゴリズムを考えたのか説明すること
       - 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（取り組んだなら課題2も）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