

第2回 画像の制御と初期化

今回は、前回作成した関数の中に、画像を表示する命令を加え、さらに自機(Player)を動かせるようにします。自機を動かせるようにするためのキーボードからの入力を受け取る処理、入力に応じて座標を動かす処理、画像データをメモリ上にロードする処理、メモリ上の画像データを描画する処理を記述していきます。

○構造体の定義の追加

DxLib.h のインクルードの後に以下の define 及び構造体を定義します。以下の構造体はそれぞれキーボード・ジョイパッドからの入力、自機に関する情報、ロードした画像の情報を記録するための構造体です。

```
// 記号定数
#define BUTTON_MAX ( 5 )

// 構造体
// 入力情報
struct SInput{
    int button[ BUTTON_MAX ]; // ボタンを押し続けたフレーム数
}input;

// 主人公の情報 各々のプレイヤーに配列の要素一つが対応
// 今回はプレイヤーが一人だけなので、配列にはしない。
struct SPlayer{
    int life; // プレイヤーのライフ
    int pos[ 2 ]; // プレイヤーの画面上の座標
}player;

// 画像の情報
struct SImg{
    int player; // プレイヤーの画像
    int chip; // マップチップの画像
    int enemy[ 8 ]; // 敵の画像
}img;
```

この時点でのmain.cはmain_02a.cppを参照してください。最初の行については後で説明します。構造体の使用法は覚えていますか？最初のSInput構造体ならば、SInputは構造体の「タグ名」、最後のinputは変数名「input」での構造体変数の宣言です。構造体変数がグローバル変数として宣言されていることに注意しましょう。

各構造体のタグ名の先頭の大文字の「S」は構造体、つまりstructのSです。こうしておくことであとと見やすいソースコードになるでしょう。今回のプログラムでは複数の単語が連なる関数名や構造体名は各単語の先頭文字を大文字で記述します。これをPascal記法と呼びます。変数名については単語は全て小文字で区切りはアンダーバーで記述します。これらの名前は申し訳ありませんがわかりませんでした。また、人によっては構造体変数であることを表すために、構造体変数の変数名の先頭にs_などの文字列をつける人もいます。

playerには体力を現すlifeと、座標を表すposという配列があります。このposについてですが、pos[0]はx座標、pos[1]はy座標ということになっています。配列の要素の添え字が0か1によって座標軸を表すこの表現は、このゼミBでは割と頻繁に用いることになるので、覚えておいてください。

imgには障害物(?)の画像用のchipという変数と、敵の画像用のenemyという配列があります。

後者が何故配列なのかはこの後判ります。では、じわじわと本題に入っていきます。

○初期化について

ゲームを始めるときには初期配置やら体力の設定などが必要ですよね。ここでちょっとゼミAのおさらい。

例えば、適当に `int x;` と記述しても、`int` 型の変数 `x` の値は `0` ではなく、任意の数値が入っており、数値で初期化せずに中の値を扱うのは非常に危険です。ですから、以下のように初期化していきます。

```
int Game( void ){
    // 変数の初期化
    player.life = 100;    // 自機のライフに100を代入
    player.pos[ 0 ] = 320; // 自機のX座標に320を代入
    player.pos[ 1 ] = 240; // 自機のY座標に240を代入

    // ループ突入
```

関数 `Game` の先頭にこれらのコードを記述します。「`// ループ突入`」以後は以前書いたソースコードですので書く必要はありません。以後、コードを追記する際は断りなく前後のコードを併記します。ライフは100、自機の初期出現位置の座標を(320, 240)と設定しています。この時点でのソースコードは `main02b.c` を参照してください。

次に、変数の初期化後に `DX` ライブラリの関数を用いて、画像をロードしていきます。`LoadImg` 関数内に `DX` ライブラリの関数を用いて画像ロードの命令を記述してきます。いずれについても、ループ内にこの情報を書かない様に。

`LoadImg` 関数の中身は以下のとおりです。いつも通り、プロトタイプ宣言をするか、呼び出し以前に書かないとコンパイルエラーになりますので注意しましょう。今回は関数内において先程記述したグローバル変数を利用するので、グローバル変数よりも後に関数を記述してください。

```
// 画像のロード
void LoadImg( void ){
    img.player = LoadGraph( "img\\player.png" );
    img.chip = LoadGraph( "img\\tile.png" );
    if( LoadDivGraph( "img\\enemy.png", 8, 4, 2, 32, 32, img.enemy ) == -1 ){
        printfDx( "ロード失敗\n" );
    }
    return;
}
```

変数 `img.player`、変数 `img.chip` にそれぞれ `player.png`、`tile.png` をロードします。`player.png` には好きな画像を、`tile.png` には縦横32ドットの画像を用意してください。また、配列 `img.enemy` に、画像 `enemy.png` を、ロードしています。今回は縦横32ドットの画像を横に4列、縦に2列敷き詰めたものとして、配列の各々の要素にロードします。この際、ロードに失敗した場合 `LoadDivGraph` 関数は返り値として `-1` を返しますので、その際は `printfDx` 関数を用いて画面にロード失敗と表示しています。各々の関数の詳細については `DX` ライブラリのヘルプを参照してください。`LoadImg` 関数を書き終えたら、`Game` 関数内の変数の初期化後に `LoadImg` 関数を呼び出すコードを挿入してください。この時点でのソースコードは `main_02c.cpp` を参照してください。各々の画像は、`img` フォルダにいれ、そのフォルダをプロジェクトと同じディレクトリ(ソースコードがおりてあるディレクトリ)に置いてください。画像のサンプルは `zip` ファイル内に同梱してあります。

ロードした画像を描画するため、`DrawPlayer` 関数を以下のように書き換えます。

```
void DrawPlayer( void ){
    if( player.life > 0 ){    // 生存している場合のみ描画
        // 自機を描画
        DrawGraph( player.pos[ 0 ] - 8, player.pos[ 1 ]-8, img.player, TRUE );
    }
}
```

```
}
```

上記のように書き換えると、自機の座標を中心に自機の画像を描画するプログラムになります。ただし、DrawGraph 関数は与えられた座標を画像の左上頂点の座標として描画するので、自機の画像のサイズが縦横16ドットでなければ中心がずれて描画されます。自機の画像の大きさにあわせて、「-8」の部分を変えましょう。なお、DX ライブラリの2D 描画関数は、画面左上を(0,0)とし、デフォルトでは右下が(640,480)となっています。この時点でのソースコードはmain_02d.cpp を参照してください。

○入力の制御

自機の操作を行うため、構造体の定義の前、「#define BUTTON_MAX (5)」の後に以下の記号定数の定義(define)とグローバル変数の宣言を記述してください。

```
// ボタンの最大数
#define BUTTON_MAX      ( 5 )
// 各ボタンの ID
#define MY_INPUT_LEFT   ( 0 )
#define MY_INPUT_RIGHT  ( 1 )
#define MY_INPUT_UP     ( 2 )
#define MY_INPUT_DOWN   ( 3 )
#define MY_INPUT_A      ( 4 )
// 自機の移動速度
#define PLAYER_VELOCITY ( 8 )

// グローバル変数
char KeyBuf[ 256 ];      // キーボードの入力を保存する変数
```

次に、実際に操作を行うために、以下のプログラムを Game 関数のループ内の入力部に記述してください。

```
// キーボードからの入力を得る
GetHitKeyStateAll( KeyBuf );      // キーボード入力を全て配列KeyBuf に保存
// どのボタンをどれだけ押していたかを変数に保存
if( KeyBuf[ KEY_INPUT_LEFT ] == 1 ){ // 左キーを押しているなら
    input.button[ MY_INPUT_LEFT ]++; // 入力フレーム数を加算
}else{
    input.button[ MY_INPUT_LEFT ] = 0; // 押していない場合は0 を代入
}
// 以下同様
if( KeyBuf[ KEY_INPUT_RIGHT ] == 1 ){ // 右キーを押しているなら
    input.button[ MY_INPUT_RIGHT ]++;
}else{
    input.button[ MY_INPUT_RIGHT ] = 0;
}
if( KeyBuf[ KEY_INPUT_UP ] == 1 ){    // 上キーを押しているなら
    input.button[ MY_INPUT_UP ]++;
}else{
    input.button[ MY_INPUT_UP ] = 0;
}
if( KeyBuf[ KEY_INPUT_DOWN ] == 1 ){ // 下キーを押しているなら
    input.button[ MY_INPUT_DOWN ]++;
}else{
    input.button[ MY_INPUT_DOWN ] = 0;
```

```

    }
    if( KeyBuf[ KEY_INPUT_Z ] == 1 ){      // Zキーを押しているなら
        input.button[ MY_INPUT_A ]++;
    }else{
        input.button[ MY_INPUT_A ] = 0;
    }
}

```

最初のGetHitKeyStateAll関数で全てのキーボードのキーの入力状態を配列KeyBuf内に格納します。配列KeyBufの各々の要素は各々のキーが押されているとき1、押されていないとき0になっています。各々のキーに対応するインデックスの数値(KEY_INPUT_LEFT等)はDXライブラリのヘルプ内に記述されています。そして、if文によって、各要素を見て、然るべきキーがおされている場合に、配列input.buttonの各々の要素を加算し、押されていないときに0を代入します。こうすることで、どれだけ長く押していたのかを各要素に格納することができます。ちなみに、KEY_INPUT_ZはキーボードのZキー、MY_INPUT_Aはいわゆるゲーム内における「Aボタン」のことです。

記号定数の定義は配列のインデックスを定数で記述するための記述、グローバル変数の宣言はキー入力の状態を保存するための変数の宣言です。defineの後に文字列と数値を続けて上記のように書くことで、それよりも後の部分で同じ文字列がでてきた時、それをその後の文字列で置換します。つまり、一度定義しておけば、その数値を変えたいときにdefineの部分の数値だけを変えてやればよいのです。定数を扱う際、積極的にdefineを用いると良いでしょう。

値の変動を確認するため、以下のプログラムを描画部分の先頭に記述してください。

```

clsDx() ;
printfDx( "右:%d" , input.button[ MY_INPUT_RIGHT ] ) ;

```

printfDxは、DXライブラリで画面に文字を表示させるために用います。しかし、あくまで簡易的なものなので、例えば今のように数値の変動を確認するために用いると便利です。動作の重い関数ですので、プログラム完成の際には使用しないことが望ましいです。

clsDxは、上記のような簡易画面出力をクリアします。これを入れないと、上記のprintfDxによる表示が残って、ご覧の有様になります。

上記の2行の文はとりあえず残して置いてください。ここまでのプログラムはmain_02e.cppを参照してください。

○座標の制御・画像の移動

下記の関数をDrawPlayer関数の前に記述してください。

```

// 自機の移動
void MovePlayer( void ){
    int v[ 2 ] = { 0, 0 }; // 各々のプレイヤーの速度を格納する変数宣言
    int f_pos[ 2 ];        // プレイヤーが移動する直前の座標を保存する変数
    if( player.life > 0 ){  // プレイヤーが生きているなら
        // 入力状況に応じて速度を算出
        if( input.button[ MY_INPUT_UP ] > 0 ){ // 上ボタンを押しているなら
            v[ 1 ] -= PLAYER_VELOCITY;        // Y方向への速度を減算
        }
        if( input.button[ MY_INPUT_DOWN ] > 0 ){ // 下ボタンを押しているなら
            v[ 1 ] += PLAYER_VELOCITY;        // Y方向への速度を加算
        }
        if( input.button[ MY_INPUT_LEFT ] > 0 ){ // 左ボタンを押しているなら
            v[ 0 ] -= PLAYER_VELOCITY;        // X方向への速度を減算
        }
        if( input.button[ MY_INPUT_RIGHT ] > 0 ){ // 右ボタンを押しているなら
            v[ 0 ] += PLAYER_VELOCITY;        // X方向への速度を加算
        }
    }
}

```

```

// 斜め移動の場合はルート2で割、速度を一定に保つ
if( v[ 0 ] != 0 && v[ 1 ] != 0 ){
    v[ 0 ] = ( int )( v[ 0 ] * 0.71f );
    v[ 1 ] = ( int )( v[ 1 ] * 0.71f );
}
// 実際に移動
while( ProcessMessage() != -1 ){           // 移動が完了するまでループ
    // 現在のX座標を f_pos[ 0 ]に保存
    f_pos[ 0 ] = player.pos[ 0 ];
    // X座標を1ドット動かす
    // 速度が正の場合加算、負の場合減算
    if( v[ 0 ] > 0 ){                       // X方向の速度が正なら
        player.pos[ 0 ]++; // 座標を加算
        v[ 0 ]--;          // 速度を減算
        // vは残りの移動すべきドット数を保存する役割を果たす
    }
    if( v[ 0 ] < 0 ){                       // X方向の速度が負なら
        player.pos[ 0 ]--; // 座標を減算
        v[ 0 ]++;          // 速度を加算
    }
    // 速度が0になったらループ脱出
    if( v[ 0 ] == 0 ){
        break;
    }
}
}
}

```

そして、プロトタイプ宣言部に MovePlayer 関数のプロトタイプ宣言を追加し、ループ内の移動部分に MovePlayer(); を追加してください。また、順にプログラムの内容を説明していきます。この時点でのソースコードは main_02f.cpp を参照してください。

まず、今回のループでどれだけ移動するのかを配列 v に保存します。キー入力の情報から配列 input.button に各ボタンの入力状態が保存されているので、これを参照し、if 文で分岐しながら加算・減算で算出します。PLAYER_VELOCITY には先程の define において 8 が定義されています。

さて、while 文の中身ですが、こちらでは f_pos[0] に player.pos[0] を代入しています。そして、この player.pos[0] に、v[0] の数値分、1 つずつ座標を動かすようにしています。その際、v[0] は 1 つずつ 0 に近づいていき、この v[0] の数が 0 になったとき、if 文の条件から抜け、自機は移動が完了します。

しかし、これだけでは X 軸の動きしか受け付けていません。そこで・・・

練習問題

自機を Y 軸方向にも動けるようにしてみましょう。ゼミ B の第 2 回はここまでです。ここから、どんなこのプログラムがゲームに成長していきます。ゴールを目指して頑張りましょう！解凍は main_02.cpp を参照してください。