大幅改訂版

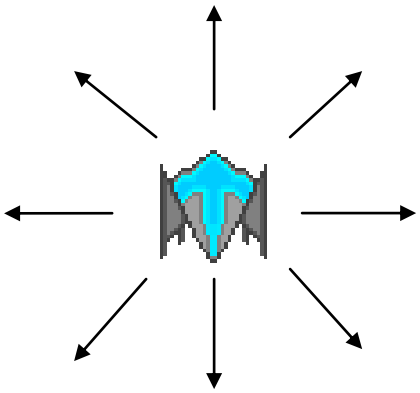
ソフトゼミＢ　第３回 キー入力Ｉ ～自・敵機の移動～

■　はじめに

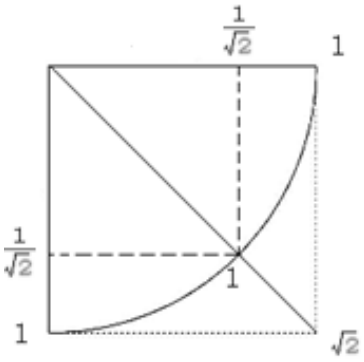
今回は、自機と敵の移動を勉強していきます。

■　自機の移動

まず自機の移動についてですが、下の図のように自機を 8 方向に動かせるようにしていきます。



ここで斜め移動について少し考えてみてください。
例えばｘ軸方向と、ｙ軸方向の移動速度が１のとき斜め移動を行うと、
下の図のように$\sqrt{2}$の移動速度で進んでしまいます。ゼミ B で作るゲームでは斜め移動の移動速度も１になるように、斜め移動のときは場合分けを行って、ｘとｙの移動速度を$\sqrt{2}$で割ることによって、斜め移動の速度を１にします。



明治大学エレクトロニクス研究部

大幅改訂版

■ 敵の移動

次は敵の移動についてです。市販で売られているようなシューティングゲームの敵の動き方は複雑ですが、今回は複雑な動きをさせず、下方向に一定の速さで移動させます。また、敵が画面外に出たらその敵に関する処理を行わせないようにしていきます。理由としてはすべての敵の処理を継続させると、重くなってしまうからです。

■ 下準備

➢ math.h (もしくは cmath)のインクルード ※改訂時に追加

これから、$\sqrt{2}$の値を出すために、平方根を求める標準ライブラリ関数 sqrt を使います。そのために必要なヘッダファイル「math.h」(もしくは「cmath」)をインクルードします。

#include “DxLib.h”の下に、

```
#include <math.h>
```

もしくは、

```
#include <cmath>
```

のどちらかを記述してください。(どちらでも良いですが、C++的には cmath の方が良いようです。詳しくはソフト班員まで。)

➢ マクロの追加

第二回で書いた#define 三つの後に、以下を追加してください。

```
/*  自機・敵機関連  */

//  自機のサイズ
#define PLAYER_SIZE 32

//  敵機のサイズ
#define ENEMY_SIZE 32

/************* #define  ここまで*************/
```

➢ 関数プロトタイプ宣言 ※改訂時に追加

今回は、2 つの新たな関数を自作します。それらの関数をどの位置に書いてもいいように、プロトタイプ宣言を行います。プロトタイプ宣言自体の説明はゼミ A 第 6 回を参照してください。

```
void movePlayer( void );
void moveEnemy( void );
```

大幅改訂版

➢ 速度を取り扱う構造体メンバの追加 ※改訂時に追加

プレイヤーと敵を動かすには、どのような速度で動かすかが必要不可欠です。というわけで、プレイヤーと敵の構造体に、速度を扱うパラメータ(網掛け部分のメンバ)を追加してみましょう。

```
//プレイヤー
struct SPlayer{
   int x, y;
   int velocity; //  自機の x, y 成分はキーボードによって決まるので、大きさのみ。
   int life, maxLife;
   int score;
};

//敵
struct SEnemy{
   int x, y;
   int vx, vy;
   int life, maxLife;
};
```

ここでいう「速度」とは 1 フレーム(ゲームループ 1 周)で動くピクセル数のことです。

これだけでは、こういうメンバがあるということしか宣言していないので、初期化の部分で、実際の値を代入します。**WinMain 関数上部**の初期化部分に、網掛け部分を追加してください。

```
// 初期化
player.x = 240;
player.y = 240;
player.velocity = 4;
player.life = player.maxLife;
player.score = 0;

enemy.x = 120;
enemy.y = 50;
enemy.vx = 0;
enemy.vy = 2;
enemy.maxLife = 5;
enemy.life = enemy.maxLife;
```

➢ GetHitKeyStateAll 関数と char 型配列 keyState の宣言　※改訂時に変更

次に、それぞれのキーボードが押されているかどうかの情報を教えてくれる関数、GetHitKeyStateAll 関数を以下のように、**ゲームループの一番上**に書いてください。

```
// キーの取得
GetHitKeyStateAll( keyState );
```

引数の keyState が宣言されてないぞ、と出てくるので、宣言します。keyState は、char 型 256 要素の配列で、各要素にキーが押されているかどうかの情報が入ります。以下の網掛けで示した部分のように、**グローバル変数の宣言の一番上**の部分に入れてください。

```
※↑この上は構造体の定義
/*グローバル変数の宣言 */
char keyState[ 256 ];
struct SImgList imgList;
struct SPlayer player;
struct SEnemy enemy;
※↓この下は WinMain 関数
```

キーが押されているかどうかの判定については、前回説明した「CheckHitKey」関数でも可能ですが、この関数は効率が悪くなる場合があるので、普通は(ESC キーでゲームループを抜ける場合を除いて)使用しません。(詳しい理由は、資料末尾の関数の説明にて。)

これで、準備が整いました。

■ 敵・自分を動かす関数

以下の内容のソースを main.cpp の末尾に書いてください。(WinMain 関数の外)

```
/*関数 movePlayer*/
void movePlayer( void ){
  //プレイヤーがこのフレームで動く x 座標の距離
  int vx = 0;
  //プレイヤーがこのフレームで動く y 座標の距離
  int vy = 0;

  /* 押されているキーから、このフレームでの移動距離(x 成分, y 成分)の決定 */
  if( keyState[ KEY_INPUT_LEFT ] ){
    vx -= player.velocity;
(次ページへ続く)
```

```
    }
    if( keyState[ KEY_INPUT_DOWN ] ){
        vy += player.velocity;
    }
    if( keyState[ KEY_INPUT_UP ] ){
        vy -= player.velocity;
    }
    if( keyState[ KEY_INPUT_RIGHT ] ){
        vx += player.velocity;
    }
    if(vx && vy){
        vx = ( int ) ( vx / sqrt( 2.0 ) );
        vy = ( int ) ( vy / sqrt( 2.0 ) );
    }

    /* 移動 */
    player.x += vx;
    player.y += vy;

    /* 画面外移動の禁止 */
    if( player.x < - PLAYER_SIZE / 2 ){
        player.x = - PLAYER_SIZE / 2;
    }
    if( player.y < - PLAYER_SIZE / 2 ){
        player.y = - PLAYER_SIZE / 2;
    }
    if( player.x > WINDOW_WIDTH - SIDEBAR_WIDTH - PLAYER_SIZE / 2 ){
        player.x = WINDOW_WIDTH - SIDEBAR_WIDTH - PLAYER_SIZE / 2;
    }
    if( player.y > WINDOW_HEIGHT - PLAYER_SIZE / 2 ){
        player.y = WINDOW_HEIGHT - PLAYER_SIZE / 2;
    }
}
```

(次ページへ続く)

```
/*関数 moveEnemy*/
void moveEnemy( void ){
  if(enemy.life == 0){ return; }
  enemy.x += enemy.vx;
  enemy.y += enemy.vy;
  // 敵が完全に画面外に出たら、死ぬ。
  if( enemy.x < - ENEMY_SIZE ||
    enemy.x > WINDOW_WIDTH - SIDEBAR_WIDTH + ENEMY_SIZE ||
    enemy.y > WINDOW_HEIGHT + ENEMY_SIZE
  ){
    enemy.life = 0;
  }
}
```

movePlayer 関数は、キー入力の状況を判断してプレイヤーを移動させます。斜めの移動になる場合は、移動距離を $1/\sqrt{2}$にします。

moveEnemy 関数は、敵のライフが 0 でないときにのみ、敵を移動させます。画面外に敵が移動した場合は、ライフを 0 にしています。

最後に、下の文を WinMain 関数のゲームループの中に書いてください。準備段階で記述した、GetHitKeyStateAll の下に、網掛け部分を追加することになります。

```
    //キーの取得
    GetHitKeyStateAll( keyState );

    //プレイヤーの移動
    movePlayer();
      //プレイヤーの弾発射追加予定
    // 敵の移動
    moveEnemy();
      //敵の弾発射追加予定
※この下、ClearDrawScreen
```

今回の変更点は以上です。

※改定前の資料では、ESC キーが押された場合の中断を keyState に変更する作業を行なっていましたが、煩雑なので、改訂時に削除しました。

大幅改訂版

■ 新出関数

GetHitKeyStateAll 関数

int GetHitKeyStateAll(char *KeyStateBuf)

前回、特定のキーが押されているかどうかを判定するために、CheckHitKey という関数を紹介しました。しかし、同じフレーム内で 2 回以上同じキーの状態を見るときに CheckHitKey を 2 回呼び出しているようでは効率が悪いです。

そこで、GetHitKeyStateAll 関数の登場です。この関数は、引数に **char 型**、要素数 **256 個**の**配列**(実際のところはポインタなのですが、ここでは配列として説明)を指定すると、その配列の各要素に、対応するキーの入力状態(押されていない: 0, 押されている: 1)が代入されます。

例えば、char keyState[256];という宣言がある時に、キーの状態を取得する時には、

```
GetHitKeyStateAll( keyState );
```

と書きます。その中で、Z キーが押されているかどうかを見る時には、

```
if( keyState[ KEY_INPUT_Z] ){
  //押されている時の処理
}else{
  //押されていない時の処理
}
```

という風に書きます。

※ KEY_INPUT～というのは DxLib.h で定義されている定数で、各キーの入力状態を管理する配列の添字の番号です。例えば、Z キーを管理する添字の番号は KEY_INPUT_Z の値で、10 進数で 44(16 進数で 0x2C)です。

※ KEY_INPUT～と実際のキーボードとの対応は前回の CheckHitKey と同じです。前回の CheckHitKey の説明を見てください。

今回は、これによって Esc キーが押された時にループを抜ける判定を while 文の中で

```
  // Esc が押されてるなら抜ける
  if( keyState[KEY_INPUT_ESCAPE] ){ break; }
```

と書いたので、while 文頭にある条件式の

```
while( ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0){
```

の内、CheckHitKey の方の条件は不要になったため、ProcessMessage のみに変更をしました。