

## 第5回 マップと自機の当たり判定

自機が障害物にめり込んではいけませんので、今回は自機と障害物との衝突判定を作成します。

### ○自機の移動とマップ

今まで、自機の移動の際、**MovePlayer** 関数内で、**int** 型の配列 **f\_pos** に直前の座標を保存し、時期の座標を移動してきました。これは、実は今回のマップと自機の当たり判定を行うための伏線だったのです。1 ドット X(または Y)座標を動かす度に自機と壁との衝突判定を行い、衝突していた場合に直前の座標である **f\_pos** に自機の座標を戻してやればいい、というのが今回のプログラムの基本的な考え方となります。

まずはマップデータを表す構造体を作成します。マップの最大の大きさを表すための定数を **define.h** に定義し、**struct.h** のインクルードガードの前に以下の構造体定義を追記してください。

・ **define.h**

```
#define MAP_X_MAX ( 100 )
#define MAP_Y_MAX ( 100 )
#define CHIP_WIDTH ( 32 )
#define CHIP_HEIGHT ( 32 )
```

・ **struct.h**

```
// マップの情報
struct SMap{
    // マップサイズ(マス)
    int width ;
    int height ;
    // 各マスのデータ
    int chipID[ MAP_X_MAX ][ MAP_Y_MAX ] ;
    int eventID[ MAP_X_MAX ][ MAP_Y_MAX ] ;
};
```

今回作るプログラムにおけるマップは、マップチップと呼ばれる 32 ドット×32 ドットのマスの集合からできています。マップのが縦横何マスからなるか(以下マップサイズ)を **SMap** 構造体のメンバ変数 **width** と **height** に保存しています。この値がそれぞれ定数 **MAP\_X\_MAX** や **MAP\_Y\_MAX** よりも大きくなると異常動作のもとですので、注意しましょう。**chipID** にはマップチップの **ID** が、**eventID** にはイベントの **ID** が格納されます。後者は敵を出現させるのに使用します(今回は使用しません)。

上記の構造体を実際に使用するため、以下のグローバル変数の宣言を **global.h** に追加して下さい。

```
struct SMap map ; // マップ
int abs_pos[ 2 ] ; // 絶対座標
```

当然インクルードガードよりも前に記述して下さい。また、上記の変数の **extern** を **extern.h** で今まで同様に行って下さい。

さて、変数 **map** がマップであるの是一目瞭然ですが、配列 **abs\_pos** とは何でしょうか？これは画面の左上がマップ上のどの位置にあたるか、という情報です。気をつけなければならないのは、こちらの座標はマス単位での座標ではなく、ドット単位の座標である、ということです。このあたりの詳細はゼミ B 第7回で行います。今回はひとまず配列 **abs\_pos** の要素には両方とも 0 を代入しておき、マップ描画関数も仮組みのものを使用します。ソースファイルに **map.cpp** を追加し、マップを描画する **DrawMap** 関数を以下のように記述してください。

```
// DX ライブラリのヘッダのインクルード
#include "DxLib.h"

// 自作ヘッダのインクルード
#include "extern.h"
#include "function.h"

// マップ描画関数
void DrawMap( void ){
```

```

// i , j : マスのインデックス x , y : 画面上の座標
int i , j , x = 0 , y = 0 ;
// 描画の起点が画面内かつ、マスのインデックスが指定範囲内である限りループ
for( i = 0 , x = 0 ; x < 640 && i < map.width ; i++ , x += CHIP_WIDTH ){
    for( j = 0 , y = 0 ; y < 480 && j < map.height ; j++ , y += CHIP_HEIGHT ){
        // 任意のマスの chipID が 1 の時
        if( map.chipID[ i ][ j ] == 1 ){
            // 描画！
            DrawGraph( x , y , img.chip , TRUE ) ;
        }
    }
}
}
}

```

DrawGraph 関数は指定した座標を描画する画像の左上頂点として描画しますので、上記のようなプログラムになります。i、j はそれぞれマップ上で何マス目かを表し、x、y はそれらのマス左上の画面上でのドット単位での座標を表します。非常にややこしいですが、プログラムを追えば自ずとその意味はわかるかと思います。配列 chipID の任意の要素が 1 の時にチップを描画しています。

さて function.h に DrawMap 関数のプロトタイプ宣言を追加したら、collision.cpp 内に実際に当たり判定を行う関数 CollisionPlayerToWall 関数を作成して行きましょう。今回のプログラムでは自機と壁の衝突判定において、チップを 32×32 ドットの長方形、自機を 8×8 ドットの長方形とみなして当たり判定を行います。define.h に以下の定義を追加してください。

```

#define PLAYER_WIDTH_TO_WALL ( 8 )
#define PLAYER_HEIGHT_TO_WALL ( 8 )

```

面倒くさくても、実数は上記のようにできるだけ define を使って定数化しておくべきです。

次に CollisionPlayerToWall 関数を作成します。以下で使用している CollisionSquareToSquare 関数は今回のテキストの最後に載っていますので、前回作成した人は書き換え、作成していない人は collision.cpp 内に追加、function.h 内でプロトタイプ宣言を行って下さい。

```

int CollisionPlayerToWall( void ){
    int i , j , chip_pos[ 2 ] ;
    // プレイヤーが侵入しているマス全てのマップチップに対し当たり判定を行う
    for( i = ( abs_pos[ 0 ] + player.pos[ 0 ] - PLAYER_WIDTH_TO_WALL / 2 )
        / CHIP_WIDTH ;
        i <= ( abs_pos[ 0 ] + player.pos[ 0 ] + PLAYER_WIDTH_TO_WALL / 2 )
        / CHIP_WIDTH ;
        i++ ){
        for( j = ( abs_pos[ 1 ] + player.pos[ 1 ] - PLAYER_HEIGHT_TO_WALL / 2 )
            / CHIP_HEIGHT ;
            j <= ( abs_pos[ 1 ] + player.pos[ 1 ] + PLAYER_HEIGHT_TO_WALL / 2 )
            / CHIP_HEIGHT ;
            j++ ){
            // マップチップの ID を調べる
            if( map.chipID[ i ][ j ] == 1 ){
                // マップチップの画面上の座標(相対座標)を調べ、代入
                chip_pos[ 0 ] = i * CHIP_WIDTH - abs_pos[ 0 ] + CHIP_WIDTH / 2 ;
                chip_pos[ 1 ] = j * CHIP_HEIGHT - abs_pos[ 1 ] + CHIP_HEIGHT / 2 ;
                if( CollisionSquareToSquare( player.pos[ 0 ] , player.pos[ 1 ] ,
                    PLAYER_WIDTH_TO_WALL , PLAYER_HEIGHT_TO_WALL ,
                    chip_pos[ 0 ] , chip_pos[ 1 ] , CHIP_WIDTH , CHIP_HEIGHT ) ==
                    1 ){
                    return ( 1 ) ;
                }
            }
        }
    }
}

```

```

return ( 0 ) ;
}

```

はっきり言って難解です。紙面上の変な位置で改行されないよう頻繁に改行を入れているので注意して下さい。2つのfor文は自機の当たり判定である8ドット×8ドットの矩形がマップ上のどのマスに侵入しているかを調べ、各々のマスについて処理を実行するためのものです。次に、そのマスのチップIDを調べ、もしもそのチップIDが1、つまり壁だった場合は、配列abs\_posと、そのチップが縦横何マス目かの情報から、画面上でのチップの中心座標を配列chip\_posに代入しています。そして、プレイヤーとチップの衝突判定を行い、これが1だった場合は返り値として1を、どの地形ともあたっていない場合は0を返り値として返します。defineを使って定義した定数や、abs\_posの意味をよく考え、この関数のプロトタイプ宣言をfunction.hに追加して下さい。

最後に、最初に述べた自機とマップの衝突判定の呼び出し部分を記述します。今回は、Game関数内でのループにおいて実行するわけでない点に注意して下さい。MovePlayer関数内で呼び出します。

```

void MovePlayer(void){
                                中略
    // X座標をドット動かす
    f_pos[ 0 ] = player.pos[ 0 ] ;
                                中略
    if( CollisionPlayerToWall() == 1 ){
        player.pos[ 0 ] = f_pos[ 0 ] ;
    }
    // Y座標をドット動かす
    f_pos[ 1 ] = player.pos[ 1 ] ;
                                中略
    if( CollisionPlayerToWall() == 1 ){
        player.pos[ 0 ] = f_pos[ 0 ] ;
    }
                                後略
}

```

まず最初に自機の座標を保存して、移動する処理を行います。その結果、障害物に重なっていたら、座標を元に戻すという作業を行っており、こうすれば自機が障害物に乗り上げるのを防ぐことができます。

さて、以上でマップ表示の仮組み、マップ内の障害物との当たり判定の関数は作成しました。しかしマップの初期化は行っていません。Game関数内のループ内での適切な位置にDrawMap関数の呼び出しを追加して下さい。また、ループに入る前の初期化部分で、構造体変数mapのメンバ変数の初期化を行って下さい。width、heightは正の整数の好きな値で初期化し、配列chipIDは全ての要素を0で初期化、eventIDは全ての要素を255で初期化しましょう。また、障害物が表示されなくては意味が無いので、chipID[ x ][ y ] ( 0 ≤ x ≤ 31 , 0 ≤ y ≤ 14 )なる要素のいくつかを1で初期化しておきましょう。

#### ○矩形同士の衝突判定

CollisionCircleToCircle関数に引き続き、各々の座標は配列のポインタを、矩形の大きさは変数を使用します。

```

int CollisionSquareToSquare( int x1 , int y1 , int w1 , int h1 , int x2 , int y2 , int w2 , int h2 )
{
    int dx , dy ;                                // 矩形Aと矩形BのX座標の距離とY座標の距離
    dx = abs( x1 - x2 ) - abs( w1 + w2 ) / 2 ; // dxに2つの矩形のX軸における距離を代入
    dy = abs( y1 - y2 ) - abs( h1 + h2 ) / 2 ; // 同じくdy
    if( dx < 0 && dy < 0 ){                        // X軸上の距離、Y軸上の距離がいずれも0未満の場合
        return ( 1 ) ;                            // 衝突
    }
    return ( 0 ) ;                                // それ以外の場合は非衝突
}

```