# DXライブラリを用いてブロック崩しを作る

## Part6 自分が操作するブロックの作成。

# 今回やること

- ほぼブロック崩しができてきたが、今回は自分が操作するブロックを作成を目標とする。
- もちろん今回もクラスを用いて、自分が操作するブロックを作る。
- ソースはpart5で作ったものをそのまま用いる。

# 自分が操作するブロックのクラス

```
    void Graph();   //ボールの描画
    void Calc();    //ボールの計算
};

class MyBlock_obj{
private:
    int x,y;
    int sizeX,sizeY;
    int color;
public:
    int GetX(),GetY();
    int GetSizeX(),GetSizeY();
    void Set();   //この関数にてx,y,sizeX,sizeY,colorを決定する。
    void Graph(); //自分が操作するブロックの描画。
    void Calc();  //操作するのでその計算関数。
};

Block_obj Block[BLOCK_X_NUM][BLOCK_Y_NUM];
Ball_obj Ball;
MyBlock_obj MyBlock;

int Block_obj::GetX(){
    return x;
}
```

クラスのパラメータはほぼBlock_objと変わらない。
ただ、消失する可能性がないことと、自分が操作する点が違うので、
flag変数がない、その代わりにCalc関数を追加した。

# 自分が操作するブロックのクラス

```
int MyBlock_obj::GetX(){
    return x;
}
int MyBlock_obj::GetY(){
    return y;
}
int MyBlock_obj::GetSizeX(){
    return sizeX;
}
int MyBlock_obj::GetSizeY(){
    return sizeY;
}
void MyBlock_obj::Set(){
    x = 240;
    y = 400;
    sizeX = 100;
    sizeY = 30;
    color = GetColor(128,128,128);
}
void MyBlock_obj::Graph(){
    DrawBox(x,y,x+sizeX,y+sizeY,color,TRUE);
}

// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
```

Ball_obj::Calc()関数と
Main関数の間に
左のMyBlock_objの関数を
追加する。

基本的に左の関数については
説明はあまりいらないだろう。

もちろん、今回もGetXなどの
関数を用いて、
当たり判定を実装する。

# 自分が操作するブロックのクラス

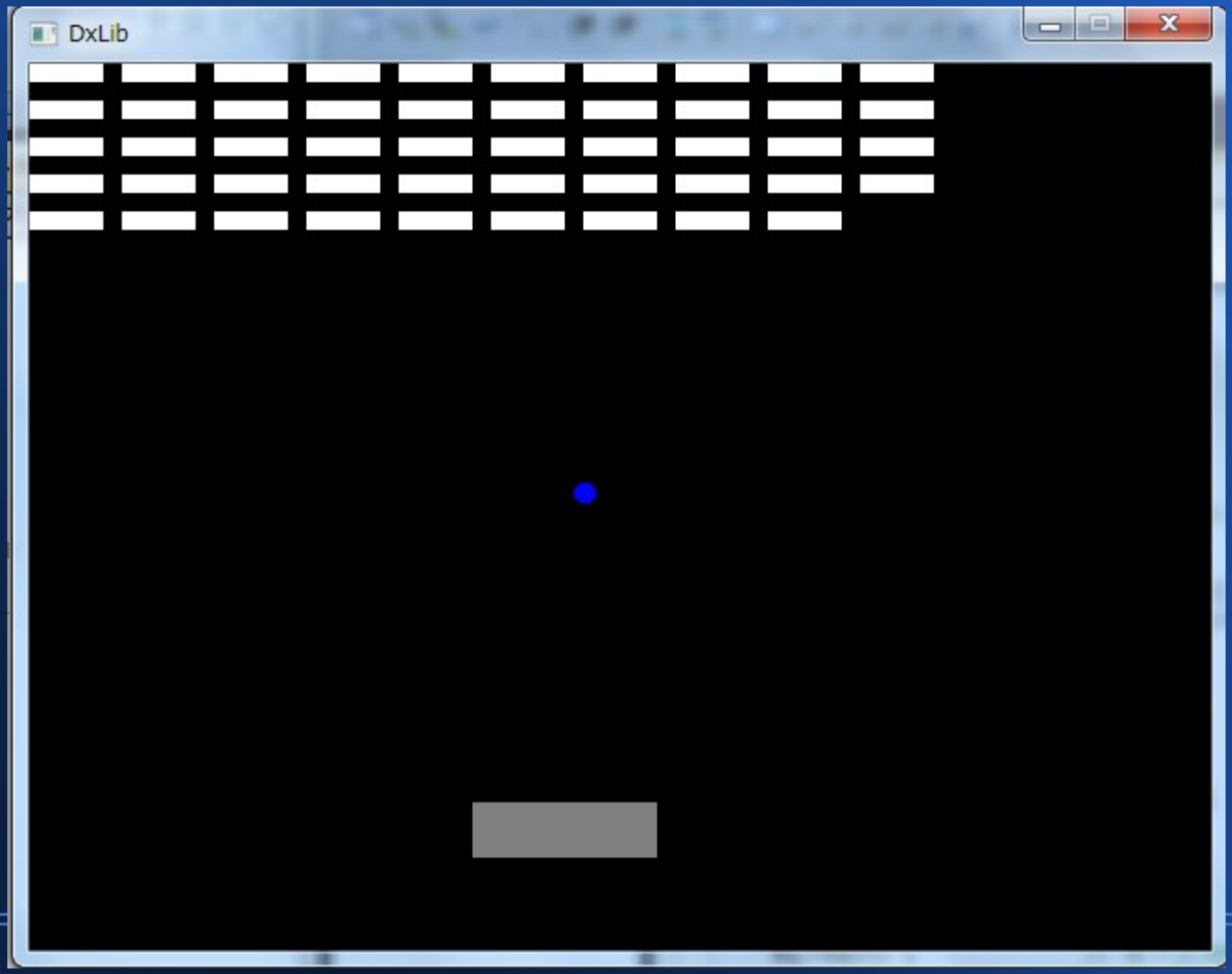
```
//自分が操作するブロックの設定
MyBlock.Set();

while(ProcessMessage()==0){

    //描画されているものを消す。
    ClearDrawScreen();
    //メインループの動作はここに書く！！

    //ブロックの描画
    for(int i=0;i<BLOCK_X_NUM;i++){
        for(int j=0;j<BLOCK_Y_NUM;j++){
            Block[i][j].Graph();
        }
    }
    //ボールの描画＋移動
    Ball.Graph();
    Ball.Calc();
    //自分が操作するブロックの描画
    MyBlock.Graph();
    //裏画面の描画状態を表に反映
    ScreenFlip();
}
```

さて、main関数内に、
MyBlock.Set()関数と
MyBlock.Graph()関数を追加する。

追加する位置はもちろん
最初の初期設定のやつはループ外

ずっと繰り返したい、ゲーム中の動作は
ループ内に書く。

# 実行結果




このように灰色のブロックが
下に描画されただろうか。

さて、まだ描画命令しかしてないので
動くわけもない。

というわけで、Calc関数の中に
ブロックを操作する命令を追加したい

そこで登場するのが、
DxLib.hにある、
GetHitKeyStateAll関数
というものを用いる。

# GetHitKeyStateAll関数

```
例　Ｚキーの状態を知りたい場合

char Buf[ 256 ] ;

GetHitKeyStateAll( Buf ) ;

if( Buf[ KEY_INPUT_Z ] == 1 )
{
        // Ｚキーが押されている
}
else
{
        // Ｚキーは押されていない
}
```

サンプルソースはDXライブラリの公式ホームページより引用してきました。

このように、まずchar型の要素数が256個の配列を宣言。
そして、GetHitKeyStateAll(Buf)とやると、そのchar型の配列に、キーが押されているのかどうかが、格納される。押されている時は1,押されていない時は0となる。
これにより、ボタンが押されているのかどうかが判定できるということである。
Zボタンが押されているか知りたい時はKEY_INPUT_Zのように、
←キーが押されているか調べる時はKEY_INPUT_LEFTを用いる。
キーの定義値はpart6ソースフォルダの中に入れておいたので、参考にするとよい。

# MyBlock_obj::Calc()関数の編集

```
void MyBlock_obj::Graph(){
    DrawBox(x,y,x+sizeX,y+sizeY,color,TRUE);
}

void MyBlock_obj::Calc(){
    char Buf[256];
    GetHitKeyStateAll(Buf);//キーの状態をBufに格納

    //左が押されていたら、MyBlockのx座標を減らす。（左に動かす）
    if(Buf[KEY_INPUT_LEFT]==1){
        x-=5;
    }
    //右が押されていたら、MyBlockのx座標を増やす。(右に動かす)
    else if(Buf[KEY_INPUT_RIGHT]==1){
        x+=5;
    }
}

// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
```

前ページのサンプルを
参考に、キーの入力状態を
調べる。

ちなみに、char型の配列の
名前はBufにする必要はない。

このようにすることで、
左や右が押されていることが
わかるので、
それに応じて、MyBlockの
x座標をいじってやることにする

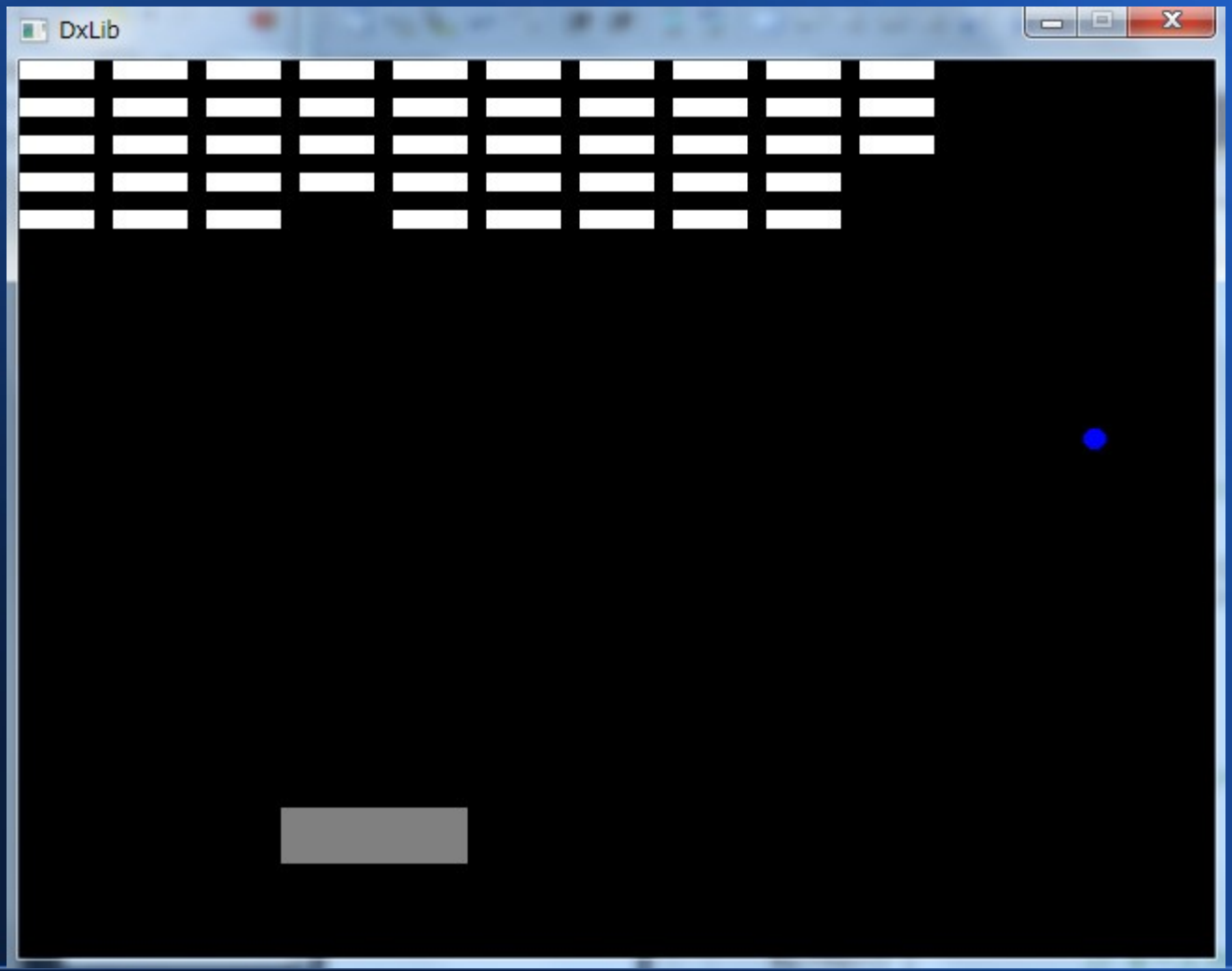
# main関数も編集する。

```
while(ProcessMessage()==0){

    //描画されているものを消す。
    ClearDrawScreen();
    //メインループの動作はここに書く！！

    //ブロックの描画
    for(int i=0;i<BLOCK_X_NUM;i++){
        for(int j=0;j<BLOCK_Y_NUM;j++){
            Block[i][j].Graph();
        }
    }
    //ボールの描画＋移動
    Ball.Graph();
    Ball.Calc();
    //自分が操作するブロックの描画＋計算
    MyBlock.Graph();
    MyBlock.Calc();
    //裏画面の描画状態を表に反映
    ScreenFlip();
}
```

といっても、ループ内に
MyBlock.Calc()を追加するだけである。

これにより、ついにブロックが動くと予想される。

# 実行結果




静止画じゃわかるわけもないが、
左を押したら左に
右を押したら右に動いたのが
確認できただろうか。

ここまでできたら後は
当たり判定だけである。

当たり判定の実装法は正直
ボールが上から来た時、どうするか
だけ考えればいいので、
そこまで難しくない。

さらに、
ほとんど前回と同じ計算をするので
コピペでも終わってしまう。

相変わらずボールの速度を
変更するので、
Ball_obj::Calc()関数を編集する。

# Ball_obj::Calc()関数を編集し、MyBlockとの当たり判定も作る。

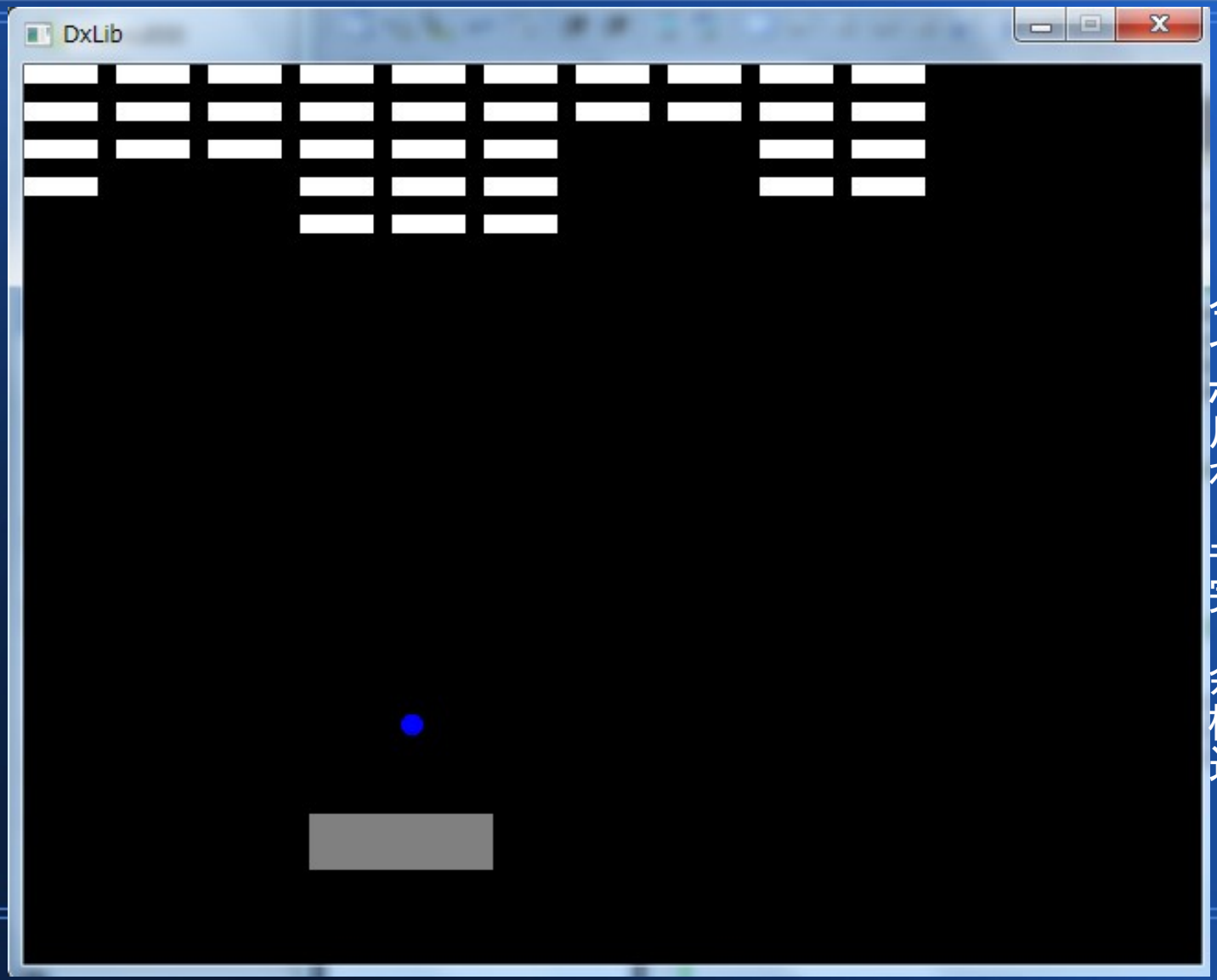
```
                }
            //ブロックの上に当たる時。
            if ( y < Block[i][j].GetY() &&
                y + vy + r >= Block[i][j].GetY() &&
                x + vx + r >= Block[i][j].GetX() &&
                x + vx - r <= Block[i][j].GetX() + Block[i][j].GetSizeX()){

                    vy *= -1;
                    Block[i][j].Delete();
            }
        }
    }
}

//自分が操作するブロックとボールの当たり判定。
//ブロックの上に当たる時。
if ( y < MyBlock.GetY() &&
        y + vy + r >= MyBlock.GetY() &&
        x + vx + r >= MyBlock.GetX() &&
        x + vx - r <= MyBlock.GetX() + MyBlock.GetSizeX()){

            vy *= -1;

}
x+=vx;
y+=vy;
}
```

自分が操作するといっても、
結局はただのブロックなので、
当たり判定はほとんど同じ、

というより、私はコピペした後、
Block[i][j]をMyBlockに
置き換えただけである。

これで実行すれば、自分が
操作するブロックと
ボールに当たり判定が
追加されたはずである。

# 実行結果




今回も静止画なのでわからないが
ついに自分が操作するブロックと
ボールに当たり判定ができ、
反射するようになったのが
わかるだろうか。

もうほとんどブロック崩しは
完成と言ってもいいかもしれない。

余裕があったら、MyBlockの
横や下の当たり判定も
追加するといいだろう。

# 総括

- とりあえず、ここまででブロック崩しのメイン部分の講座は終わりとします。
- この講座を経てクラスが大体理解できたなーとなった人がいたら嬉しいです。
- 次のpartでは、ポーズ画面の実装や、ブロック崩しの勝利条件、敗北条件の追加を目的とする予定です。