二次元配列を応用した当たり判定

二次元のゲームではよく描画を 2 次元配列で行いますが、その当たり判定も描画に用いている 2 次元配列のデータを利用してつくることができます。

どういうことを言っているかを簡単なサンプルプログラムを使って説明しましょう。まず描画するプログラムを書きます。

まずヘッダーに

```
#define SCREEN_WIDTH    (640)                           // 画面の横幅
#define SCREEN_HEIGHT   (480)                           // 画面の縦幅
#define CHIP_SIZE       (32)                            // 一つのチップのサイズ
#define MAP_WIDTH       (SCREEN_WIDTH / CHIP_SIZE)      // マップの横幅
#define MAP_HEIGHT      (SCREEN_HEIGHT / CHIP_SIZE)     // マップの縦幅
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE 管理のコンポーネント
    TTimer *Timer1;
    void __fastcall Timer1Timer(TObject *Sender);
    void __fastcall FormKeyDown(TObject *Sender, WORD &Key,
          TShiftState Shift);
    void __fastcall FormKeyUp(TObject *Sender, WORD &Key,
          TShiftState Shift);
private:    // ユーザー宣言

    TRect PL;   //操作キャラクターの現在位置など
    int i,j;

    //移動速度
    int MoveX,MoveY,DounSp;

    //キー入力
    bool L,R;

    //空中フラグ
    bool kutyu;

public:     // ユーザー宣言
    __fastcall TForm1(TComponent* Owner);
};
```

この#define を宣言せずに変数のみを使っていると、プログラミング中に「あれ？この変数なんの数字？」となるので宣言しておきましょう。

cpp ファイルのコンストラクタ関数の上に

```
int MapData[SCREEN_HEIGHT / CHIP_SIZE][SCREEN_WIDTH / CHIP_SIZE] =
{
    0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
    1,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,1,
    1,0,0,1,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,1,
    1,0,0,1,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,1,0,1,
    1,0,0,1,1, 1,1,0,0,0, 0,0,0,0,0, 0,0,1,0,1,

    1,0,0,0,0, 0,0,0,1,1, 0,0,0,0,0, 0,0,1,0,1,
    1,0,0,0,0, 0,0,0,0,0, 0,0,1,1,0, 0,0,1,0,1,
    1,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,1,0,1,
    1,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,1,
    1,0,0,0,0, 0,0,1,1,0, 0,0,0,0,0, 1,0,0,0,1,

    1,0,0,0,0, 1,1,1,1,1, 0,0,0,0,1, 1,0,0,0,1,
    1,0,0,0,0, 1,1,1,1,1, 0,0,0,1,1, 1,0,0,0,1,
    1,0,0,0,0, 0,0,0,0,0, 0,0,0,1,1, 1,0,0,0,1,
    1,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,1,
    1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1,
};
```

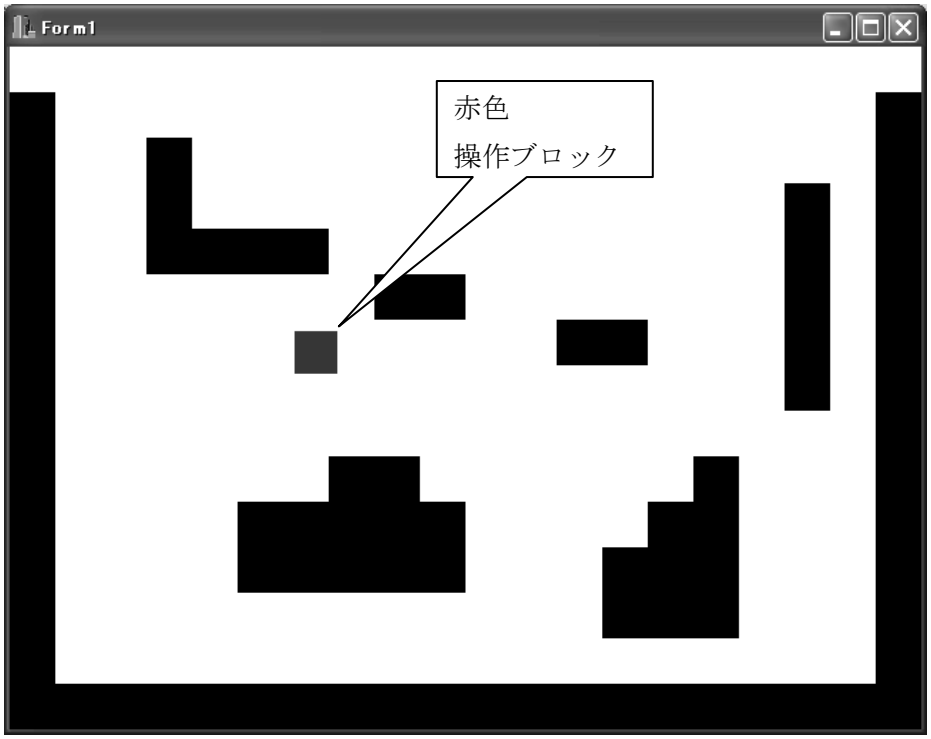
コンストラクタ関数とタイマー関数に

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    ClientWidth = SCREEN_WIDTH;
    ClientHeight = SCREEN_HEIGHT;

    PL = Bounds(200, 200, 30, 30);
}
//---------------------------------------------------------
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Canvas->Brush->Color = clWhite;
    Canvas->FillRect(Rect(0,0,ClientWidth,ClientHeight));

    Canvas->Brush->Color = clBlack;
    for(i = 0;i < SCREEN_WIDTH / CHIP_SIZE;i++){
        for(j = 0;j < SCREEN_HEIGHT / CHIP_SIZE;j++){

            if(MapData[j][i] == 1)
                Canvas->FillRect(
                    Bounds(i * CHIP_SIZE, j * CHIP_SIZE,
                        CHIP_SIZE, CHIP_SIZE));

        }
    }

    Canvas->Brush->Color = clRed;
    Canvas->FillRect(PL);
}
```

と書いてください。

先ほどのプログラムは二次元配列のデータを用いての描画をしています。詳しい内容は本冊子の「**二次元配列のマップデータへの応用**」をご覧ください。

これを実行してみると。こんな感じです。



この画像の白四角を背景、黒四角が判定ブロック、赤四角を操作ブロックとします。

これから操作ブロックを動かせるようにし、黒四角に当たり判定を作るのですが、いろいろと準備をしなければなりません。

ヘッダーにつぎのような関数を用意してください。

```
public:     // ユーザー宣言
    __fastcall TForm1(TComponent* Owner);

    // キャラクタをマップとの当たり判定を考慮しながら移動する関数
    void CharMove() ;

    // マップとの当たり判定　X、Y=プレイヤーの中心座標
    void MapHitCheck(int X, int Y, int *MX, int *MY) ;
];
```

これらの関数を使って操作ブロックの移動と当たり判定を製作します。

CharMove 関数は最終的にキャラクターがどの位置に置くかを決定する関数です。中身はこんな感じです。

```
void TForm1::CharMove()
{
    int Dummy = 0;

    //中心座標算出
    int SX = PL.Left + PL.Width()/2;
    int SY = PL.Top + PL.Height()/2;

    //上下移動成分だけチェック
    MapHitCheck(SX - PL.Width()/2, SY - PL.Height()/2, &Dummy, &MoveY);

    MapHitCheck(SX + PL.Width()/2, SY - PL.Height()/2, &Dummy, &MoveY);

    MapHitCheck(SX - PL.Width()/2, SY + PL.Height()/2, &Dummy, &MoveY);

    MapHitCheck(SX + PL.Width()/2, SY + PL.Height()/2, &Dummy, &MoveY);

    PL.Top += MoveY;//上下移動成分を加算

    //後に左右移動成分だけでチェック

    MapHitCheck(SX - PL.Width()/2, SY - PL.Height()/2, &MoveX, &Dummy);

    MapHitCheck(SX + PL.Width()/2, SY - PL.Height()/2, &MoveX, &Dummy);

    MapHitCheck(SX - PL.Width()/2, SY + PL.Height()/2, &MoveX, &Dummy);

    MapHitCheck(SX + PL.Width()/2, SY + PL.Height()/2, &MoveX, &Dummy);

    PL.Left += MoveX;// 左右移動成分を加算

    PL = Bounds(PL.Left, PL.Top, 30, 30);//座標の最終決定

    // 移動量の初期化
    MoveX = 0 ;
    MoveY = 0 ;
]
```

操作ブロックの座標はこの関数が決定しています。この関数の処理が終わって初めて動くことになります。

この関数は MapHitCheck 関数を使用しています。その関数の中身はこんな感じです。

```
void TForm1::MapHitCheck(int X, int Y, int *MX, int *MY)
{
    //このまま移動した時の座標
    int afX = X + *MX;
    int afY = Y + *MY;

    // 当たり判定のあるブロックに当たっているかチェック
    //if( GetChipParam( afX, afY ) == 1 ){
    if(MapData[afY / CHIP_SIZE][afX / CHIP_SIZE] == 1){
        // 当たっていたら壁から離す処理を行う
        int BL,BT,BR,BB;

        BL = (afX / CHIP_SIZE) * CHIP_SIZE ;        // 左辺の X 座標
        BR = (afX / CHIP_SIZE + 1) * CHIP_SIZE ;    // 右辺の X 座標
        BT = (afY / CHIP_SIZE) * CHIP_SIZE ;        // 上辺の Y 座標
        BB = (afY / CHIP_SIZE + 1) * CHIP_SIZE ;    // 下辺の Y 座標

        // 上辺に当たっていた場合
        if( *MY > 0){
            // 移動量を補正する
            *MY = BT - Y - 1;
            return ;
        }
        //下辺に当たっていた場合
        if( *MY < 0){
            // 移動量を補正する
            *MY = BB - Y + 1;
            return ;
        }
        //左辺に当たっていた場合
        if( *MX > 0){
            // 移動量を補正する
            *MX = BL - X - 1;
            return ;
        }
        //右辺に当たっていた場合
        if( *MX < 0){
            // 移動量を補正する
            *MX = BR - X + 1;
            return ;
        }
    }
}
```

この関数は現在位置と現在の速度を用いて、このまま進むとブロックにぶつかるかどうかを計算し、もしぶつかるなら、ぶつからないように移動量を補正する処理をしています。ここでマップデータを引き出してくることにより、現在位置と現在の速度を足した位置がブロックのある位置かどうかを判断できるということです。

ここまで関数ができたらあとは CharMove 関数を使うだけです。

しかし、ブロックにあたっているかの判定は操作ブロックの現在位置と操作ブロックの速さ MoveX、MoveY で処理しています。

よってブロックを移動させる処理は

PL.Left += ブロックをすすめる速さ;

ではなく、

MoveX += ブロックをすすめる速さ;

としてください。

もし座標に直接ブロックの進める速さをを足してしまうと、CharMove 関数はブロックの速さを 0 と勘違いしてしまいます。そうなるとブロックにあたったかの判定ができなくなります。

ブロックを移動させる処理は必ず座標そのものではなく速さを示す変数にしてください。

そのあとで CharMove 関数を使います。