ソフトゼミ B 2016 第 7 回

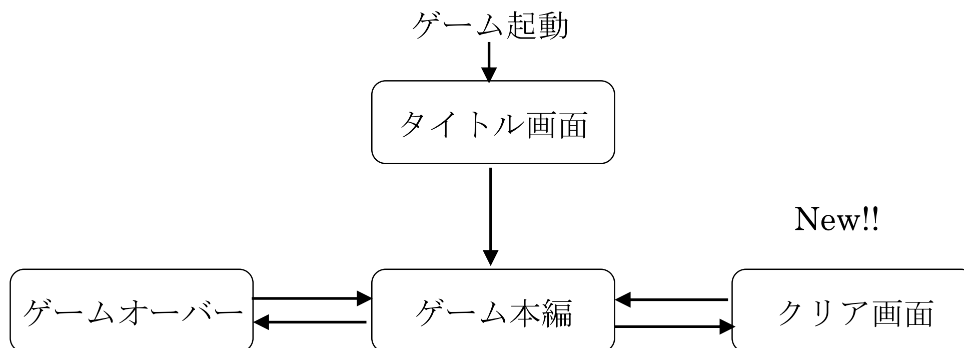
ブロック判定

今回はブロック判定です．壁判定ということもあります．1 番のポイントは「上下のブロック判定と，左右のブロック判定を分けて考える」ことです．どういうことか．順を追ってしっかりと理解していきましょうφ(*▽*)≡=—

今回もまた，新しいソースコードを配ります．新しいプロジェクトは作らなくていいので，ソースコードを前回作った main.cpp に上書きしてください．前回のソースを取っておきたいという方は，メモ帳などに取っておくといいでしょう．

➢ ゲーム全体の構造

今回のゲームは次のような構造をしています．前回なかったクリア画面が実装されています．



➢ 壁判定の考え方

第 6 回の「次回に向けて」をやった人にはわかると思いますが，前回までのあたり判定だけでは，横の壁にぶつかっても止まってくれない，という不具合が発生してしまいます．前回までのあたり判定は次のようになっていました．

```
void judge_v(int blx, int bly) {
	if (blx < (player.x + HERO_SIZE) && player.x < (blx + BLOCK) && bly <= (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {

		if (player.vy >= 0) {//落ちてる最中あるいはｙ方向には動いていないなら
			player.jfly = 0;
			player.y = (bly - HERO_SIZE);
			player.vy = 0;
		}
		else {//上向きに動いてるなら
			player.vy = 0;
			player.y = (bly + BLOCK);
		}

	}
	int white = GetColor(255, 255, 255);
	DrawFormatString(550,120,white,"J2:%4d",map[blx/BLOCK][blx/BLOCK]);
}
```

前回の不具合の原因は，要するに「主人公とブロックがぶつかったら，主人公をそのブロックの上にのせる」としてしまっていることです．これでは横からぶつかったときに止まってくれません．

そこで，左右の壁判定をする関数をつくってやります．次の通りです．

```
void judge_h(int blx, int bly) {//左右壁判定
	//右に壁があるか
	if (blx <= (player.x + HERO_SIZE) && player.x < (blx + BLOCK - 4) && bly < (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {	//右方向
		player.kabe_r = 1;
		if (map[blx / BLOCK][bly / BLOCK] == 3)
			//触れているブロックがゴールならクリア
			clear();


	}
```

```
	//左に壁があるか
	if (blx + 4 <= (player.x + HERO_SIZE) && player.x <= (blx + BLOCK) && bly < (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {	//左方向
		player.kabe_l = 1;
		if (map[blx / BLOCK][bly / BLOCK] == 3)
			//触れているブロックがゴールならクリア
			clear();
	}
	//デバック用
	int white = GetColor(255, 255, 255);
	DrawFormatString(450, 20, white, "J1:%4d", map[blx / BLOCK][blx / BLOCK]);
  }
```

右と左とで処理を分ける理由は簡単です. 右に壁があっても左には動けますし, 逆に左に壁があっても右には動けるからです.

右あるいは左に壁があった場合，壁をしめすフラグ，player.kabe_r と player.kabe_l とがそれぞれ ON になります．kabe_r が ON になっていれば右にはそれ以上動かず，kabe_l が ON になっていれば左にはそれ以上動かないように，主人公の移動を管理する move 関数で決められています.

➢ ソースコード

それではソースコードを以下に示します．前回と変わっていないところは一部省略したところがあります.

```
#include "DxLib.h"
#include<stdio.h>



#define WINDOW_WIDTH 640	//ウィンドウの大きさ(横)
#define WINDOW_HEIGHT 480 //ウィンドウの大きさ(縦)
#define GRA 1				//重力加速度
#define MAXSPEEDY 20			//自由落下の最高速度
#define CENTER 288			//主人公の中央座標
#define HERO_SIZE 32		//主人公の大きさ
```

```
#define ATTACK_SIZE 32                    //攻撃の大きさ
#define WIDTH_SIZE 35                     //スクロールの幅(横)
#define HEIGHT_SIZE 15                    //スクロールの幅(縦)
#define BLOCK 32                          //タイルの大きさ[px]

#pragma warning(disable : 4996)//fopen,fscanf でエラーが出るのを防ぐ
                //このエラーの原因は「マイクロソフトの余計なおせっかい」です

struct SImg{
        int migi;
        int r_attack;
        int kabe;
        int yuka;
        int goal;
        int r_enemy;
        int gameover;
        int title;
        int clear;
        int clear2;
        int haikei;
        int null;
};

//主人公関連の構造体
struct SPlayer{
        int x, y;           //主人公の X,Y 座標
        int vx, vy;                 //主人公の X,Y 軸方向の速度
        int jfly;           //飛んでいるかどうかの判別　0:着地/1:それ以外
        int dire;           //主人公の向いている方向 0:右/1:左
        int attack;                 //攻撃中かどうかの判別              0:通常/1:攻撃中
        int kabe_r;                 //右方向に壁があるかどうかの判定
        int kabe_l;                 //左方向に壁があるかどうかの判定
        int kakusi;
};
```

```
//敵関連の構造体
struct SEnemy{
	int x, y;			//敵の X,Y 座標
	int vx, vy;				//敵の X,Y 軸方向の速度
	int life;			//敵の体力
	int size_x, size_y;		//敵の大きさ
	int count;				//敵の動く際のカウント
	int dire;			//敵の向いている方向	0:右/1:左
};


//構造体の宣言
struct SImg img;
struct SPlayer player;
struct SEnemy enemy;
int map[WIDTH_SIZE][HEIGHT_SIZE];
char keyState[256];


//マップチップの読み込み
void Map(void) {
	FILE *file;

	if ((file=fopen("map/map.txt","r"))== NULL){
	  //もしマップデータが開くことが出来なかったら
	      デバッグ画面に"Map Data Read Error"と表示
		OutputDebugString("MapData Read Error¥n");
		exit(EXIT_FAILURE);
	}
	for (int j = 0; j < HEIGHT_SIZE; j++) {
		for (int i = 0; i < WIDTH_SIZE; i++) {
			fscanf(file, "%d,", &map[i][j]);//カンマに気を付けてください
		}
	}
	fclose(file);
}
```

```
//画像読み込み
void image(void) {
   /*略*/
}

//変数の初期化
void init(void) {
   /*略*/
}

void title(void) {
   /*略*/
}

//ゲームオーバー画面の表示＋マップのリロード＋変数の初期化
void gameover(void) {
   /*略*/
}

//クリア画面の表示マップのリロード＋変数の初期化
void clear(void) {

        ClearDrawScreen();
        if (player.kakusi >= 10)
                DrawGraph(0, 0, img.clear2, TRUE);
        else
                【穴埋め１】;      //クリア画面(img.clear)を描画
        ScreenFlip();

        while(keyState[【穴埋め２】]!=1&&keyState[KEY_INPUT_X]!=1) {
        //エスケープかつ X が押されていないとき
                GetHitKeyStateAll(keyState);
                if (ProcessMessage() == -1)
                        break;              //エラーが発生したらループを抜ける
        }
```

```
		init(); //変数の初期化

		Map();	//マップのリロード
}

void judge_h(int blx, int bly) {
		//右に壁があるか
		if (blx <= (player.x + HERO_SIZE) && player.x < (blx + BLOCK - 4) && bly < (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {	//右方向
				player.kabe_r = 1;
				if (map[blx / BLOCK][bly / BLOCK] == 3)				// 触れているブロックがゴールならクリア
						clear();

		}
		//左に壁があるか
		if (blx + 4 <= (player.x + HERO_SIZE) && player.x <= (blx + BLOCK) && bly < (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {	//左方向
				player.kabe_l = 1;
				if (map[blx / BLOCK][bly / BLOCK] == 3)				// 触れているブロックがゴールならクリア
						clear();
		}
		//デバック用
		int white = GetColor(255, 255, 255);
		DrawFormatString(450, 20, white, "J1:%4d", map[blx / BLOCK][blx /BLOCK]);
}

//壁判定　(上下判定/ブロックの上に乗る)
void judge_v(int blx, int bly) {
		//主人公とブロックが重なっていたら
		if (blx < (player.x + HERO_SIZE) && player.x < (blx + BLOCK) && bly <= (player.y + HERO_SIZE) && player.y < (bly + BLOCK)) {

				if (player.vy >= 0) {//落ちてる最中あるいは y 方向には動いていないなら
						player.jfly = 0;
```

```
				player.y = (bly - HERO_SIZE);
				player.vy = 0;
			}
			else {//上向きに動いてるなら
				player.vy = 0;
				player.y = (bly + BLOCK);
			}
			if (map[blx / BLOCK][bly / BLOCK] == 3) {//触れているブロックがゴー
		ルなら
				【穴埋め３】;
			}
		}

		int white = GetColor(255, 255, 255);
		DrawFormatString(550, 120, white, "J2:%4d", map[blx/ BLOCK][blx/BLOCK]);

}

//プレイヤーの移動
void move(void) {

		player.vx = 0;						//横移動リセット

		if (player.jfly != 0 && player.vy < MAXSPEEDY)	//重力(空中にいれば重力を
働かせる)
			player.vy += 1;

		player.kabe_l = 0;				//壁判定リセット
		player.kabe_r = 0;

		【穴埋め４】//壁判定（左右）の呼び出し

		GetHitKeyStateAll(keyState);	//キーボードのすべてのキーの状態を取得
				//その状態は変数 keyState に記録される

		if (keyState[KEY_INPUT_LEFT]) {				//主人公の左向き操作(もし
```

```
左ボタンが押されていたら)
			if (player.kabe_l == 0 && player.x > 0) {
				player.vx -= 4;	//主人公を x 方向に移動させる
			}
			player.dire = 1;//主人公は左向き
		}

		if (keyState[KEY_INPUT_RIGHT]) {			//主人公の右向き操作(もし
右ボタンが押されていたら)
			if (player.kabe_r == 0 && player.x + HERO_SIZE <= BLOCK *
WIDTH_SIZE) {
				player.vx += 4;
			}
			player.dire = 0;//主人公は右向き
		}

		if (player.jfly == 0 && keyState[KEY_INPUT_X]) {		//主人公のジャン
プ操作(空中にいないときに X キーが押されたら)
			player.jfly = 1;
			player.vy -= 15;		//主人公の y 速度を変化させる(上方向の速さ)
		}

		if (keyState[KEY_INPUT_R])				//デバッグ用ジャンプ
			player.vy = -5;

		player.jfly = 1;

		player.x += player.vx;
		player.y += player.vy;

		for (int j = 0; j<HEIGHT_SIZE; j++) {
			for (int i = 0; i<WIDTH_SIZE; i++) {
				if (map[i][j] != 0)
					judge_v(i*BLOCK, j*BLOCK);
			}
		}
```

```
	if ((enemy.x < player.x + HERO_SIZE && player.x < enemy.x + enemy.size_x)
&&(enemy.y < player.y + HERO_SIZE && player.y < enemy.y + enemy.size_y)) {
		if (enemy.life != 0) {
			gameover();
		}
	}



	if (player.y>WINDOW_HEIGHT) {//画面の外に出たら（下に落ちたら）ゲームオ
ーバー
		gameover();
	}


	if (player.x == 0 && (player.y > 400 && player.y < 402))
		player.kakusi += 1;
	}

//敵キャラの移動
void enemymove(void) {
   /*略*/
}

//主人公キャラの攻撃
void attack(void) {


	GetHitKeyStateAll(keyState);
	if (keyState[KEY_INPUT_Z]) {	//Z キーが押されていたら
		if (player.attack == 0)
			player.attack = 1;
	}
	else
		player.attack = 0;
}
```

```
void draw(void) {
	//画面クリア
	ClearDrawScreen();

	//背景の描画
	for (int j = 0; j < BLOCK; j++)
		DrawGraph(j * BLOCK, 0, img.haikei, TRUE);

	//マップチップからの描画
	for (int j = 0; j < HEIGHT_SIZE; j++) {
		for (int i = (player.x / BLOCK) -9; i < (player.x / BLOCK) + 12; i++) {
			if (i < WIDTH_SIZE && i >= 0) {	//横の部分より狭い範囲でマップ判断
				switch (map[i][j]) {
				case 0: break;
				case 1:		DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.yuka, TRUE); break;	//床
				case 2:		DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.kabe, TRUE); break;	//壁
				case 3:		DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.goal, TRUE); break;	//ゴール旗(触れたらゴール)
				default: DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.null, TRUE); break;		//null
				}
			}
			else if (j == 14)
				DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.yuka, TRUE);			//最下段は床を描画
			else DrawGraph(i*BLOCK - player.x + CENTER, j * BLOCK, img.kabe, TRUE);
		}
	}

	if (player.dire == 0)			//主人公の描画
		DrawGraph(CENTER, player.y, img.migi, TRUE);			//右向き
```

```
	else
		 DrawTurnGraph(CENTER,player.y,img.migi,TRUE);
		//DrawTurnGraph でその画像を左右反転させて表示させる

	if (player.attack) {				//攻撃の描画
		if (player.dire == 0)
			DrawGraph(【穴埋め 5】, 【穴埋め 6】, img.r_attack, TRUE);
		else
			DrawTurnGraph(【穴埋め 7】, 【穴埋め 6】, img.r_attack,
TRUE);
	}



	if (enemy.life > 0) {				//敵の描画
		if (enemy.dire == 0)			//向きの判別
			DrawGraph(enemy.x - player.x + CENTER, enemy.y,
img.r_enemy, TRUE);
		else
			DrawTurnGraph(enemy.x - player.x + CENTER, enemy.y,
img.r_enemy, TRUE);
		}

	//デバッグ用の変数描画
	int white = GetColor(255, 255, 255);
	DrawFormatString(450, 20, white, "X :%4d", player.x);
	DrawFormatString(550, 20, white, "Y :%4d", player.y);
	DrawFormatString(450, 40, white, "VX:%4d", player.vx);
	DrawFormatString(550, 40, white, "VY:%4d", player.vy);
	DrawFormatString(450, 60, white, "右壁:%4d", player.kabe_r);
	DrawFormatString(550, 60, white, "左壁:%4d", player.kabe_l);
	if (player.dire) {
		DrawString(450, 80, "左向き", white);
	}
	else {
		DrawString(450, 80, "右向き", white);
	}
```

```
	if (player.jfly) {
		DrawString(550, 80, "空中", white);
	}
	else {
		DrawString(550, 80, "着地", white);
	}
	if (player.attack) {
		DrawString(450, 100, "攻撃", white);
	}
	else {
		DrawString(450, 100, "通常", white);
	}

	ScreenFlip();
}


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR ipCmdLine, int nShowCmd)
	{
		/*略*/

	//初期化に失敗したらやめる
	if (DxLib_Init() == -1) { return -1; }

	////マップチップ読み込み
	Map();

	//画像のロード
	image();

	//変数の初期化
	init();

	SetDrawScreen(DX_SCREEN_BACK);
	//タイトル表示
```

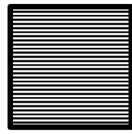
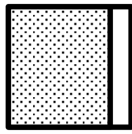
```
        title();

        /************************** ゲームループ ****************************/
        while(ProcessMessage()==0&& CheckHitKey(KEY_INPUT_ESCAPE) == 0) {

                //動き
                move();
                enemymove();

                //攻撃
                attack();

                //描画
                draw();

        }

        /************************ ゲームループおわり ***********************/
        //DX ライブラリ使用の終了処理
        DxLib_End();

        //SOFT の終了
        return 0;

}
```

➢ 今回使った DX ライブラリのライブラリ関数

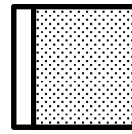
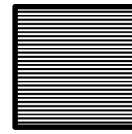
※前回と同じです.

➢ 左右壁判定の補足

左右の壁判定のあたり判定で＋4 とか-4 とか書いてあるところがあったと思います. このときの壁判定を図にするとこんな感じになります.

右の壁判定

左の壁判定

色がついているところが，あたり判定の範囲です．色の濃いほうが主人公で，薄いほうが壁（ブロック）です．

この図を見ると，ブロックのあたり判定の範囲が少し欠けています（ここが+4,-4 の部分です）．こうしないとうまく止まってくれません．プログラムから+4,-4 を消して実行してみるとその理由が分かるはずです．考えてみてください．

また，なぜ 4 なのかも考えてみましょう．ヒントは主人公の移動の仕方です．

➢ 次回

次回はシューティングと同じようにファイル分割をします．多分時間が余るので，質問とかサンプルゲームの改造案とかを考えておくといいかもしれません．