第2回　画像の制御と初期化

今回は、前回作成した関数の中に、画像を表示する命令を加え、さらに自機(Player)を動かせるようにします。自機を動かせるようにするためのキーボードからの入力を受け取る処理、入力に応じて座標を動かす処理、画像データをメモリ上にロードする処理、メモリ上の画像データを描画する処理を記述していきます。

◯構造体の定義の追加

DxLib.hのインクルードの後に以下の構造体を定義します。以下の構造体はそれぞれキーボードからの入力、自機に関する情報、ロードした画像の情報を記録するための構造体です。

```
// 入力情報
struct SInput{
	int button[ BUTTON_MAX ] ;	//ボタンを押し続けたフレーム数
}input ;
// 主人公の情報。各々のプレイヤー一人に一つの配列
struct SPlayer{
	int life ;
	int pos[ 2 ] ;
}player ;
// 画像の情報
struct Simg{
	int player ;
	int chip ;
	int enemy[ 8 ] ;
}img ;
```

構造体の使用方法は覚えていますか？最初のSInput構造体ならば、SInputは構造体の定義を表す「タグ名」、最後のinputは変数名「input」での構造体変数の宣言です。構造体変数がグローバル変数として宣言されていることに注意しましょう。

各構造体のタグ名の先頭の大文字の「S」は構造体、つまりstructのSです。こうしておくことであとあと見やすいソースコードになります。今回のプログラムでは複数の単語が連なる関数名や構造体名は各単語の先頭文字を大文字で記述します。これをPascal記法と呼びます。変数名については単語は全て小文字で区切りはアンダーバーで記述します。こちらの名前は申し訳ありませんがわかりませんでした。また、人によっては構造体変数であることを表すために、構造体変数の変数名の先頭にs_などの文字列をつけるひともいます。

playerには体力を現すlifeと、座標を表すposという配列があります。このposについてですが、pos[0]はx座標、pos[1]はy座標ということになっています。配列の中の添え字が0か1かによって座標軸を表すこの表現は、このゼミBでは割と頻繁に用いることになるので、覚えておいてください。

imgには障害物（？）の画像用のchipという変数と、敵の画像用のenemyという配列があります。何故配列なのかはこの後判ります。では、じわじわと本題に入っていきましょう。

◯初期化について

ゲームを始めるときには初期配置やら体力の設定などが必要ですよね。ここでちょっとゼミAのおさらい。

例えば、適当にint x;と記述しても。int型の変数xの値は0ではなく、任意の数値が入っており、数値で初期化せずに中の値を扱うのは非常に危険です。ですから、以下のように初期化していきます。

```
int Game(void){
	// 変数の初期化
	player.life = 100 ;			//自機のライフ
	player.pos[ 0 ] = 320 ;		//自機の初期X座標
	player.pos[ 1 ] = 240 ;		//自機の初期Y座標

	// ループ突入
```

関数 Game の先頭にこれらのコードを記述します。「// ループ突入」以後は以前書いたソースコードですので書く必要はありません。以後、コードを追記する際は断りなく前後のコードを併記します。

ライフは 100、自機の初期出現位置の座標を(320,240)と設定しています。また、この時点で DX ライブラリの関数を用いて、画像をロードしていきます。LoadImg 関数内に DX ライブラリの関数を用いて画像ロードの命令を記述してきます。いずれについても、ループ内にこの情報を書かない様に。

LoadImg 関数の中身は以下のとおりです。いつも通り、プロトタイプ宣言をするか、呼び出し以前に書かないとコンパイルエラーになりますので注意しましょう。今回は関数内において先程記述したグローバル変数を利用するので、グローバル変数よりも後に関数を記述してください。

```
void LoadImg( void ){
	img.player = LoadGraph( "img\\player.png" ) ;
	img.chip = LoadGraph( "img\\tile.png" ) ;
	if( LoadDivGraph( "img\\enemy.png" , 8 , 4 , 2 , 32 , 32 , img.enemy ) == -1
){
		printfDx( "ロード失敗\n" );
	}
	return;
}
```

変数 img.player、変数 img.chip にそれぞれ player.png、tile.png をロードします。player.png には好きな画像を、tile.png には縦横 32 ドットの画像を用意してください。また、配列 img.enemy に、画像 enemy.png を、ロードしています。今回は縦横 32 ドットの画像を横に 4 列、縦に 2 列敷き詰めたものとして、配列の各々の要素にロードします。この際、ロードに失敗した場合 LoadDivGraph 関数は返り値として-1 を返しますので、その際は printfDx 関数を用いて画面にロード失敗と表示しています。各々の関数の詳細については DX ライブラリのヘルプを参照してください。

ロードした画像を描画するため、DrawPlayer 関数を以下のように書き換えます。

```
void DrawPlayer(void){
	if(player.life>0){	//生存していない場合は描画しない
		//自機(円)を描画
		DrawGraph( player.pos[ 0 ] - 8 , player.pos[ 1 ] - 8 , img.player ,
TRUE ) ;
	}
}
```

上記のように書き換えると、自機の座標を中心に自機の画像を描画するプログラムになります。ただし、DrawGraph 関数は与えられた座標を画像の左上頂点の座標として描画するので、自機の画像のサイズが縦横 16 ドットでなければ中心がずれて描画されます。自機の画像の大きさにあわせて、「 -8」の部分を変えましょう。なお、DX ライブラリの 2D 描画関数は、画面左上を(0,0)とし、デフォルトでは右下が(640,480)となっています。

このままでは上記のプログラムはまだ適用されていません。LoadImg 関数を WinMain 関数内の Game 関数の呼び出し部分の前に記述しましょう。

o入力の制御

自機の操作を行うために、以下のプログラムを Game 関数のループ内の入力部に記述してください。

```
		GetHitKeyStateAll( KeyBuf ) ;			//キーボード入力全てを得る
		if( KeyBuf[ KEY_INPUT_LEFT ] == 1 ){		//左キーを押しているなら
			input.button[ MY_INPUT_LEFT ]++ ;	//入力フレーム数加算
		}else{
			input.button[ MY_INPUT_LEFT ] = 0 ;	//押していない場合は 0 にする
		}
		if( KeyBuf[ KEY_INPUT_RIGHT ] == 1 ){		//右キーを押しているなら
			input.button[ MY_INPUT_RIGHT ]++ ;
		}else{
			input.button[ MY_INPUT_RIGHT ] = 0 ;
		}
		if( KeyBuf[ KEY_INPUT_UP ] == 1 ){		//上キーを押しているなら
			input.button[ MY_INPUT_UP ]++ ;
```

```
            }else{
                  input.button[ MY_INPUT_UP ] = 0 ;
            }
            if( KeyBuf[ KEY_INPUT_DOWN ] == 1 ){        //下キーを押しているなら
                  input.button[ MY_INPUT_DOWN ]++ ;
            }else{
                  input.button[ MY_INPUT_DOWN ] = 0 ;
            }
            if( KeyBuf[ KEY_INPUT_Z ] == 1 ){           // Z キーを押しているなら
                  input.button[ MY_INPUT_A ]++ ;
            }else{
                  input.button[ MY_INPUT_A ] = 0 ;
            }
```

さらに構造体の定義の前に以下のプログラムを記述してください。

```
#define MY_INPUT_LEFT ( 0 )
#define MY_INPUT_RIGHT ( 1 )
#define MY_INPUT_UP ( 2 )
#define MY_INPUT_DOWN ( 3 )
#define MY_INPUT_A ( 4 )
#define PLAYER_VELOCITY ( 8 )

char KeyBuf[ 256 ] ;
```

最初のプログラムはキーボードからの入力を得て、その内容をゲーム内での入力として扱い易いよう変数に格納するプログラムです。最初の GetHitKeyStateAll 関数で全てのキーボードのキーの入力状態を配列 KeyBuf 内に格納します。配列 KeyBuf の各々の要素は各々のキーが押されているとき 1、押されていないとき 0 になっています。各々のキーに対応するインデックスの数値(KEY_INPUT_LEFT 等)は DX ライブラリのヘルプ内に記述されています。そして、if 文によって、各要素を見て、然るべきキーがおされている場合に、input.button 配列の各々の要素を加算し、押されていないときに 0 を代入します。こうすることで、どれだけ長く押していたのかを各要素に格納することができます。

後者のプログラムは配列のインデックスを定数で定めるための記述と、キー入力の状態を保存するための変数の宣言です。define の後に文字列と数値を続けて上記のように書くことで、それよりも後の部分で同じ文字列がでてきた時、それをその後の文字列で置換します。つまり、一度定義しておけば、その数値を変えたいときに define の部分の数値だけを変えてやればいいのです。定数を扱う際、積極的に define を用いると良いでしょう。

値の変動を確認するため、以下のプログラムを描画部分に記述してください。

```
clsDx() ;
printfDx( "右:%d" , input.button[ MY_INPUT_RIGHT ] ) ;
```

printfDx は、ＤＸライブラリで画面に文字を表示させるために用います。しかし、あくまで簡易的なものなので、例えば今のように数値の変動を確認するために用いると便利です。動作の重い関数ですので、プログラム完成の際には使用しないことが望ましいです。

clsDx は、上記のような簡易画面出力をクリアします。これを入れないと、上記の printfDx による表示が残って、ご覧の有様になります。

上記の 2 行の文はとりあえず残して置いてください。画像の制御は次から本番です。

◯座標の制御・画像の移動

下記のプログラムを DrawPlayer 関数の前に記述してください。

```
void MovePlayer(void){
      int v[ 2 ] = { 0 , 0 } ; //各々のプレイヤーの速度を格納する変数宣言
      int f_pos[ 2 ] ;
      if( player.life > 0 ){          //プレイヤーが生きているなら
            //入力状況に応じて速度をとる。
            if( input.button[ MY_INPUT_UP ] > 0 ){
                  v[ 1 ] -= PLAYER_VELOCITY ;
            }
```

```
            if( input.button[ MY_INPUT_DOWN ] > 0 ){
                  v[ 1 ] += PLAYER_VELOCITY ;
            }
            if( input.button[ MY_INPUT_LEFT ] ){
                  v[ 0 ] -= PLAYER_VELOCITY ;
            }
            if( input.button[ MY_INPUT_RIGHT ] ){
                  v[ 0 ] += PLAYER_VELOCITY ;
            }
            // 斜め移動の場合はルートで割り、速度を一定に保つ
            if( v[ 0 ] != 0 && v[ 1 ] != 0 ){
                  v[ 0 ] = (int)( v[ 0 ] * 0.71f ) ;
                  v[ 1 ] = (int)( v[ 1 ] * 0.71f ) ;
            }
            // 実際に移動
            while( 1 ){
                  // 現在のX座標を変数に保存
                  f_pos[ 0 ] = player.pos[ 0 ] ;
                  // X座標をドット動かす
                  // 速度が正の場合と負の場合で分岐
                  if( v[ 0 ] > 0 ){
                        player.pos[ 0 ]++ ;      // 正なら座標を加算
                        v[ 0 ]-- ;               // 速度を減算。
                        // vは残りの移動すべきドット数を保存する役割を果たす
                  }else if( v[ 0 ] < 0 ){
                        player.pos[ 0 ]-- ;
                        v[ 0 ]++ ;
                  }

            }
      }
}
```

そして、ループ内にMovePlayer();を追加してください。また、順にプログラムの内容を説明していきます。

まず、今回のループでどれだけ移動するのかを配列vに保存します。キー入力の情報から配列input.buttonに各ボタンの入力状態が保存されているので、これを参照し、if文で分岐しながら加算・減算で算出します。PLAYER_VELOCITYには先程のdefineにおいて8が定義されています。

さて、while文の中身ですが、こちらではf_pos[0]にplayer.pos[0]を代入しています。そして、このplayer.pos[0]に、v[0]の数値分、１つずつ座標を動かすようにしています。その際、v[0]は1つずつ0に近づいていき、このv[0]の数が0になったとき、if文の条件から抜け、自機は移動しなくなります。

しかし、これだけではx軸の動きしか受け付けていません。そこで・・・

練習問題

自機をy軸方向にも動けるようにしてみましょう。第2回のゼミＢはここまでです。ここから、どんどんこのプログラムがゲームに成長していきます。ゴールを目指して頑張りましょう！