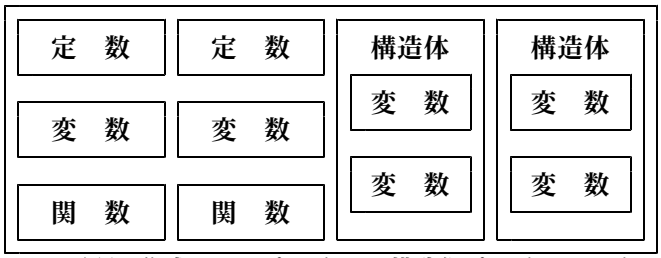
ESライブラリ＆＆
ゲームプログラミング

○　総集編　シューティングゲームの作成　○

C言語

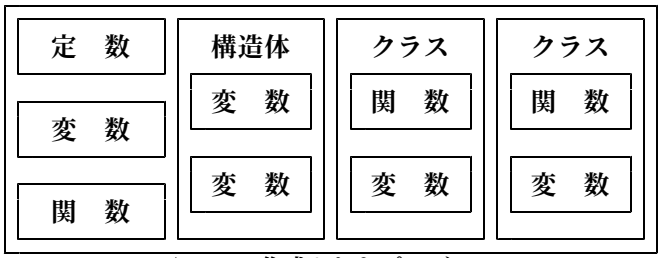
C言語は、1972年、AT&T社のベル研究所のD.M.RitchieとB.W.Kernighanによって開発された構造化プログラミングを行うための手続き型言語のひとつです。
C言語は、プログラムの機能を関数として記述し、データを変数や定数として記述します。基本的な制御文と豊富な演算子によって関数(function)を作り、その集まりをプログラムする関数型のプログラミング言語です。入出力も特別な命令があるわけではなく、標準で提供される関数(標準ライブラリ関数)に任されています。また、関連のあるデータをまとめることのできる構造体や、メモリアドレスを指し示すポインタなどもC言語の特徴の一つです。
一般的に、C言語はコンパイラ方式であり、入力されたプログラムは、コンピュータが実行できる形式に変換した後に実行されます。



▲C言語で作成されたプログラム(構造化プログラミング)

C++

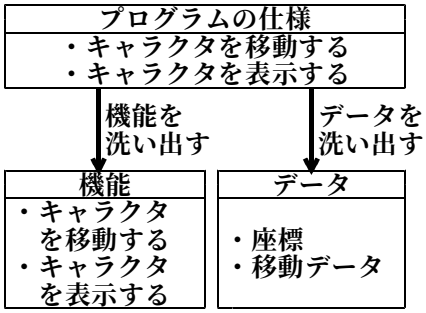
C++は、手続き型のC言語にオブジェクト指向の概念が取り入れられて改良されたオブジェクト指向言語です。AT&T社のベル研究所のBjarne Stroustrupによって、1980年頃から開発された「クラス付きC」とよばれた言語が元になっており、1983年にC++の名前になりました。
C++は、C言語にクラスを定義する機能を追加し、オブジェクト指向プログラミングを実現できるようにしたものです。したがって、C言語の機能だけで記述されたソースコードをC++のコンパイラでコンパイルすることもできます。構造体やポインタもそのまま使われています。C++は、C言語に慣れ親しんだプログラマが、構造化プログラミングからオブジェクト指向プログラミングに徐々に移行するためのものだと言えます。



▲C++で作成されたプログラム
(構造化プログラミング＋オブジェクト指向プログラミング)

構造化プログラミング

構造化プログラミングとは、プログラムの構造を機能(部品)単位に細分化するというものです。構造化プログラミングのための設計のことを構造化設計と呼びます。構造化設計では、プログラムの仕様の中に、どのような機能があるかを洗い出します。つまり、プログラムを機能の集まりと考えるのです。
たとえば、「キャラクタ移動プログラム」を作成するとしましょう。構造化設計では、プログラムの仕様の中から「キャラクタを移動する」「キャラクタを表示する」という機能を洗い出します。
構造化設計では、機能の洗い出しが終わったら、引き続きデータの洗い出しを行います。キャラクタ移動プログラムでは、「キャラクタの座標」および「移動データ」というデータが洗い出されます。



どのようなプログラムであっても、その構成要素は機能と機能の対象となるデータに大別されます。これは、構造化プログラミングでもオブジェクト指向プログラミングでも同じです。

関数

構造化プログラミングでは、プログラムを機能の集まりとして捉えますが、C言語では、機能を「関数」で記述します。

C言語は、関数の集まりで成り立っています。つまり、ある関数を実行することで、ある処理をするとも言えます。関数ひとつひとつは大した処理をしませんが、それらの関数を次々に実行することにより、複雑な処理を実現します。

C言語の関数は大きく２種類に分けることができます。C言語の規格であらかじめ用意されている標準ライブラリ関数と、プログラマが作成するユーザ関数です。標準ライブラリ関数には、画面に文字を表示するprintf関数をはじめ、文字列操作、ファイルの読み書き、メモリの読み書き、数学関数などさまざまな関数が提供されています。しかし、標準ライブラリ関数だけでは足りないので、独自の関数を作成できるようになっています。

変数

変数とは、プログラムで計算を行うときに使う数値を記憶するために、メモリに数値を角のするための箱のことをいいます。この箱(変数)を介してさまざまな演算を行うことができます。

```
int   a = 3, b = 10, c = 3;
```

a [3]　　b [10]　　c [3]

変数に付ける名前のことを変数名といいます。プログラムは変数名によって変数の区別を行います。上の例では、a, b, cという変数名がつけられ、それぞれ３，１０，３が入っています。

また、変数には型という属性があります。型によって、記憶できる数値の種類と範囲が異なります。C言語では、基本の型として以下の４つの型があります。上の変数a, b, cはint型なので、約±２１億の数値を記憶できます。

型名	説明	Windows上での値の範囲
char	文字を格納	-128～127
int	符号付き整数	-2147483684～2147483647
float	浮動小数点	±10の±38乗(有効桁数7桁)
double	倍精度浮動小数点	±10の±308乗(有効桁数15桁)

コンピュータは無限に大きな数字や、無限に大きな精度の数を記憶することができません。上の表のようにそれぞれの型には一定の範囲内の数しか格納することができないようになっています。それぞれの型の使用ビット数は、その型の変数を使うために使用する記憶容量の大きさです。ここで、Windows上と断っているのは、C言語の仕様では、これらの型で使う記憶容量の大きさが明確に規定されていないからです。Windowsやコンパイラによっては、int型が64ビットのものもあります。

さらに型の範囲を明確に定義するためにshortとlongと呼ばれる修飾子が用意されています。これを変数の宣言の前に付けて、その型が取りうる値の範囲を指定します。short, longを付けた型は、次の３つがあります。

型名	説明	Windows上での使用ビット数
short int	短い符号付き整数	16
long int	長い符号付き整数	32
long double	拡張倍精度浮動小数点	Windowsではdoubleと同じ内部表現

短い符号付き整数、長い符号付き整数のビット数も、コンピュータやＯＳなどによって異なります。このように同じプログラムでも使うコンピュータによって精度が異なるということは、当然プログラミングのさまざまな場面に影響を及ぼします。そこでＣ言語では、それぞれの型での最低限保証される精度が定められています。short intとintは少なくとも16ビットあり、long intは少なくとも32ビットであることが保証されています。

また、符号付き整数に対して符号なし整数というものもあります。int, short int, long int, charは、符号付きの整数型ですが、これらのすべての型に対して、符号なしの整数として扱うようにすることもできます。そのためには、unsignedを各型の前に付ければよいのです。

たとえば、char型は文字を格納しますが、その範囲は-128から127までです(注：0～255という環境もあります)。しかし、unsigned charにすると、0から255になります。なお、unsignedの反対の意味としてsignedがあります。実は今までの定義ではこのsignedが省略されていたのです。これらをまとめると以下のようになります。

組み合わせた型	慣例的な型	Windows上での型	Windows上でのサイズ(バイト)	範　囲
signed char	char	char	1	-128～127
unsigned char	unsigned char	BYTE		0～255
signed short int	short	short	2	-32768～32767
unsigned short int	unsigned short	WORD		0～65535
signed int	int	int	4	-2147483684～2147483647
unsigned int	unsigned int	UINT		0～4294967295
signed long int	long	long		intと同じ
unsigned long int	unsigned long	DWORD		unsigned intと同じ
float	float	float		±10の±38乗(有効桁数7桁)
double	double	double	8	±10の±308乗(有効桁数15桁)
long double	long double	long double		Windowsではdoubleと同じ

以上のように、たくさんの型がありますが、整数を扱う場合は、たとえ小さな数値しか必要としない場合でもint型かunsigned int型、実数の場合はdouble型を使います。これらの型は、OS(CPU)がもっとも高速に計算できるようになっています。そのほかの型は、特別な場合だけ使うようにします。たとえばDirect3D(DirectX Graphics)では、ほとんどの数値をGPUが高速に処理できるfloat型で扱うため、double型を使っても数値が切り捨てられたり、コンパイル時に「警告」が発生することがあります。

定数

123や3103のように直接値を指定する数値を定数(constant)といいます。定数も実は型を持ちます。123や3103と記述すると、これらの定数はintとして扱われます。intの範囲を超えるとunsigned intとして扱われます。

また、定数だけからなる式を定数式と呼びます。たとえば、次の式はすべて定数式です。

```
3103
1 + 2 + 3 + 4 / 5
123 * (310 - 3)
```

小数点の付いた数値はdouble型として扱われます。

```
3.14159      1.41421356      8.93
```

小数点の付いた数値の後ろにfまたはFを付けるとfloat型として扱われます。

```
3.14159f     1.414213f       8.93f
```

変数の定義と初期化

変数を定義するためには以下のように記述します。

```
型　変数名;
```

たとえば、int型の変数を定義するには、

```
int　h;
int　s;
int　t;
int　m;
```

とします。これでh, s, t, mという名前の箱が用意されるわけです。また、カンマ記号を用いて、次のようにまとめて書くこともできます。

```
int　h, s, t, m;
```

このように、C言語では使用する変数をすべて定義しなくてはいけません。また、変数は使用する前に定義しておく必要があります。たとえば、以下のようなプログラムはエラーになります。

```
int　h, s, t;

h = 3;
s = 1;
t = 0;
m = 3;  // エラー！

int　m;
```

変数は定義されることにより、その箱がメモリに確保されます。しかし、その際に自動的に初期値として0が代入されるわけではありません。つまり、どんな値が入っているかわからない状態(不定)になっています。

```
int　x, y;

y = x + 100;
```

このプログラムを実行すると、yの値は意味のないものとなります。このように、初期化(最初に値をセットしておくこと)をしていないと意図しない結果になってしまいます。

const修飾子

円周率や光速度のような定数を用いて計算を行うことは多々あります。しかし、円周率や光速度のように常に、一定の値を変数に格納して計算に用いるというのは問題があります。なぜなら、変数の値は、その名の示すとおり、その内容を変更できるので、ミスや勘違いによってその定数の値が別のものに書き換えられてしまうことがよくあるからです。プログラムの実行中、ずっと同じ値である場合には、const修飾子を付けてその値を変えることができないようにします。

```
const double　pi = 3.14159265358979;　// 円周率
pi = 2.99792458;　　　　　　　　　　　// エラー！
```

演算子

C言語の演算子は、加算と減算はそれぞれ+と-を用いますが、乗算は×ではなく*を用います。また、キーボードに÷のキーがないので、除算は/(スラッシュ)で代用します。すなわち、a × bはa * bと記述し、a ÷ bはa / bと記述します。括弧があれば、その中が先に計算されます。代表的な演算子を紹介します。

演算子名	記号	名　前	記述例	機　能
算術演算子	++	インクリメント	a++	値に1を加える
	--	デクリメント	a--	値から1を引く
	+	正符号	+a	正符号
	-	負符号	-a	負符号
	*	乗算	a * b	乗算
	/	除算	a / b	除算
	%	剰余	a % b	割り算の余りを求める。
	+	加算、文字列の連結	a + b	加算、両辺が文字列なら連結する
	-	減算	a - b	減算

演算子名	記号	名　前	記述例	機　能
ビット演算子	<<	左シフト	a << b	ビットを左にシフトする
	>>	算術右シフト	a >> b	ビットを右にシフトする

演算子名	記号	名　前	記述例	機　能
関係演算子	==	等値	a == b	両辺が等しければtrue、そうでなければfalseとなる
	!=	非等値	a != b	両辺が等しくなければtrue、そうでなければfalseとなる
	<	左不等	a < b	左辺より右辺が大きければtrue、そうでなければfalseとなる
	<=	等価左不等	a <= b	左辺が右辺以上ならtrue、そうでなければfalseとなる
	>	右不等	a > b	左辺より右辺が小さければtrue、そうでなければfalseとなる
	>=	等価右不等	a >= b	左辺が右辺以下ならtrue、そうでなければfalseとなる

演算子名	記号	名　前	記述例	機　能
論理演算子	&&	条件 AND	a && b	両辺がtrueのときのみtrueとなる
	||	条件 OR	a || b	少なくとも一方がtrueならtrueとなる

演算子名	記号	名　前	記述例	機　能
代入演算子	=	単純代入	a = b	右辺の式の値を左辺に代入する
	/=	除算代入	a /= b	a = a / bと同じ
	%=	余剰代入	a %= b	a = a % bと同じ
	*=	乗算代入	a *= b	a = a * bと同じ
	+=	加算代入	a += b	a = a + bと同じ
	-=	減算代入	a -= b	a = a - bと同じ

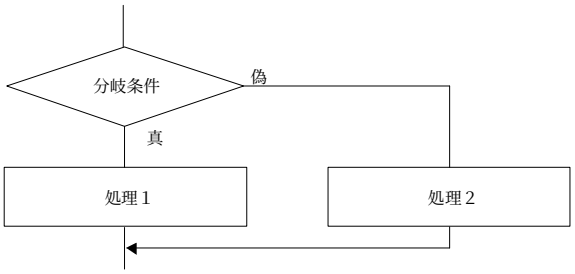
制御文

制御文とは、プログラムの流れを制御するための文です。C／C++では上から順番にプログラムが実行されますが、制御文を使うことにより、プログラムを分岐させたり、反復したりすることができます。
分岐させる制御文としてif-else、switch-caseがあり、反復(ループ)させる制御文としてwhile、do-while、forがあります。

1．if-else

if-else文は、条件が成り立つ場合は「処理１」が行われ、成り立たない場合は「処理２」が実行されます。

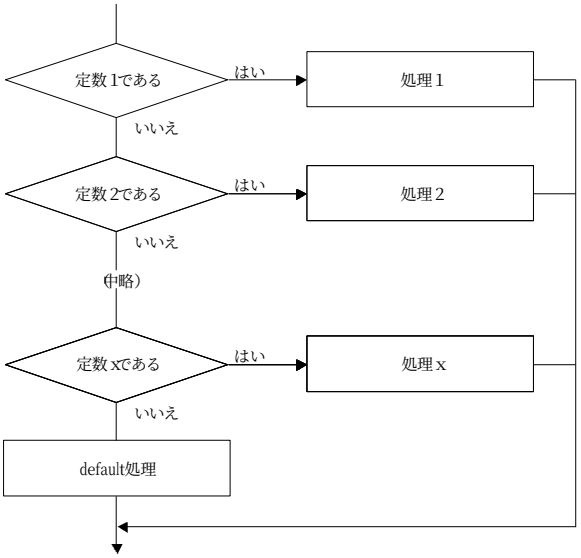
```
if(条件式) {
    条件式を満たしたときに実行する処理
} else {
    条件式を満たさなかったときに実行する処理
}
```



２．switch-case

switch文は、値によって実行する処理を多分岐させることができます。

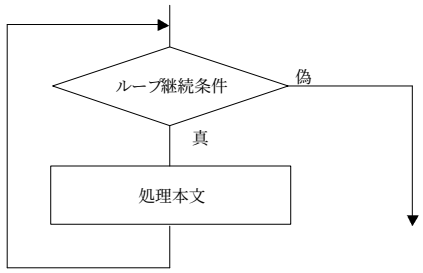
```
switch(変数) {
case 定数式１:
    処理１
    break;

case 定数式２:
    処理２
    break;
  (中略)
case 定数式ｘ:
    処理ｘ
    break;

default:
    すべての定数式を満たさなかったときの処理
    break;
}
```



３．while

while文は、ループに入る前に条件判定を行い、条件が成り立つ間処理を反復します。

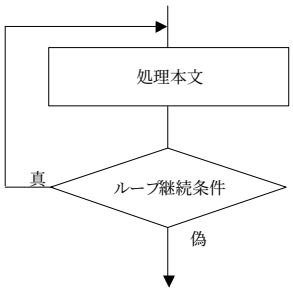
```
while(ループ継続条件) {
    処理本文
}
```



４．do-while

do-while文は、ループの最後に条件判定を行い、条件が成り立つ間処理を反復します。

```
do {
    処理本文
} while(ループ継続条件);
```

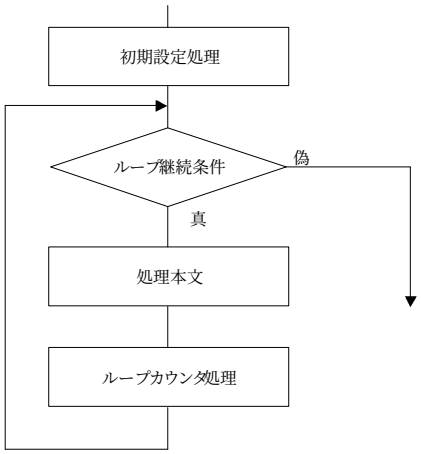


５．for

for文は、条件が成り立つ間、処理を反復させることができます。

```
for(初期設定処理; ループ継続条件; ループカウンタ処理) {
    反復させたい処理
}
```

for文は、指定した回数だけ反復させたいときに使用します。反復回数を管理するために、ループカウンタという変数を使用します。



配列

プログラムでは、データを変数に入れて扱います。基本的に１つのデータにつき１つの変数を宣言する必要がありますが、この方法ではプログラム作成上、困難になる場合があります。
たとえば、敵キャラクターの座標が、teki_x1, teki_y1, teki_x2, teki_y2…というint型の変数に格納されているとします。

```
int   teki_x1 =  0, teki_y1 =  3;   // 敵キャラNo.1
int   teki_x2 = 10, teki_y2 =  3;   // 敵キャラNo.2
int   teki_x3 = 50, teki_y3 = 10;   // 敵キャラNo.3
```

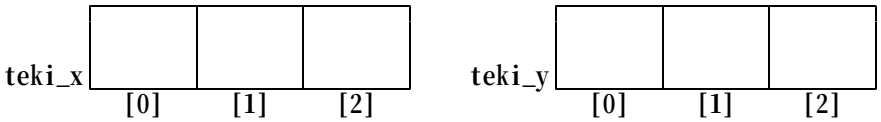
これは、以下のように個々の変数を用意していることになります。

0	3	10	3	50	10
teki_x1	teki_y1	teki_x2	teki_y2	teki_x3	teki_y3

ここで、敵キャラクターの数が多くなった場合、たとえば、1000になったとしたら、teki_x1000とteki_y1000まで２０００個もの変数を定義しなければいけなくなり、かなり面倒です。

このような時、「配列」の出番となります。配列は、「個々の変数をひとかたまりに宣言できるもの」といえます。いままでは、プログラムの中で値を格納するためには、必要な分だけ変数を作ってきました。配列を使うと、同じ型のデータをまとめて宣言できるようになります。

配列を図で表現すると、以下のように変数が連続しているイメージになります。上の図との違いは、変数が連続している点にあります。



配列とは、まとめて箱を確保し、位置(先頭からの距離)を指定してアクセスできる変数といえます。

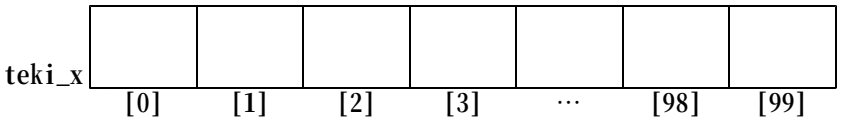
配列の定義

変数を使用するには定義が必要であったのと同じように、配列を使うためにも定義が必要です。

```
int    teki_x[100];
```

これは、teki_xという名前のint型の配列を定義しています。変数名の後に角括弧[]でくくって必要な個数を記述することで、配列の数(これを要素数といいます)を定義することができます。この例ではint型の１００個のデータが格納できる配列を定義しています。

配列を定義すると、メモリ上には以下の図のような変数の格納領域ができあがります。ちょうど、teki_x[0], teki_x[1], …, teki_x[99]という名前の１００個の変数が並んだ形になります。先頭がteki_x[1]ではなく、teki_x[0]になっていることに注意してください。



配列の定義の最初のintは、この配列がint型の変数を格納することを意味しています。もちろん、double型の配列やchar型の配列、そしてポインタや構造体などさまざまな変数を格納する配列を定義できます。
ただし、１つの配列を構成する要素は、すべて同じ型である必要があります。たとえば、１個目から３個目まではint型で、４個目から７個目まではdouble型にしたい、というような定義はできません。
配列を定義する際の括弧[]で囲まれた配列の要素数は、正の整数の定数式でなければなりません。小数や変数を指定することはできません。

```
int    array1[0.5];       // 小数はエラー

int    i = 100;
int    array2[i];         // iは変数なのでエラー

int    array3[310 / 3];   // 定数式なのでＯＫ
int    array4[0];         // 要素数が０の配列はエラー
```

配列の要素数は、プログラム実行中に変更することはできません。コンパイル時に決定されるため、プログラムの設計時によく考えておくことが必要です。

配列の添字

配列の個々の変数(これを配列の要素と呼びます)は、括弧[]を使ってアクセスすることができます。たとえば、前節の配列teki_xでは、teki_x[0]やteki_x[1]とすることで各要素にアクセスできます。この括弧でくくられた数字の部分を配列の添字と呼びます。Ｃ言語で配列の要素を扱う場合、次のことを注意してください。

配列の最初の要素は添字が０であり、最後の添字は「配列の要素数－１」である

配列の最初の要素には、添字を0にして(たとえば、teki_x[0])としてアクセスします。その次が1でアクセスします。0から始まることに注意してください。

１００個の要素数の場合、最後の１００番目の要素にアクセスするためには、添字を99とします。100にしてはいけません。0から始まるので、そこから100数えると最後は99になるわけです。また、添字には変数を使用することもとできます。たとえば、

```
int    array[100];
int    i = 9;
int    j = array[i];
```

というように変数を使ってアクセスすることもできます。

配列の値の参照と代入

配列の要素は、添字が必要ということを除けば、通常の変数と同じように使用することができます。たとえば、「int array[100]」として定義した場合、array[添字]はint型の変数として扱われます。
array[50]に0を代入したければ、

```
array[50] = 0;
```

とします。配列は要素ごとにしかアクセスできません。=演算子である配列を別の配列にコピーしたり、+=演算子で「配列＋配列」などとすることはできません。

```
int    a[5], b[5];

a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;

b  = a;      // 配列のコピー？→エラー
b += a;      // 配列の加算？  →エラー
```

このような場合、次のように要素１つずつ処理を行う必要があります。

```
b[0] = a[0];
b[1] = a[1];
b[2] = a[2];
b[3] = a[3];
b[4] = a[4];
```

配列をまとめて処理する場合は、ループを使うと効率よいプログラムを作成することができます。また、配列のコピー程度の操作であれば、メモリ操作関数でも行うことができます。

ところで、前述の配列a, bをまちがえて

```
b[10] = a[10];
```

とした場合、配列a, bともに要素数が５であり、10は存在しません。Ｃ言語では、配列の境界チェックを行わないため、コンパイルエラーにはなりません。このようなプログラムでは、メモリを破壊しながら処理を続けてしまいます。まったく関係のない変数やプログラムの一部が壊されてしまいます。

配列を使用するときは、範囲外にならないように、プログラム側で充分考慮する必要があります。

DirectX

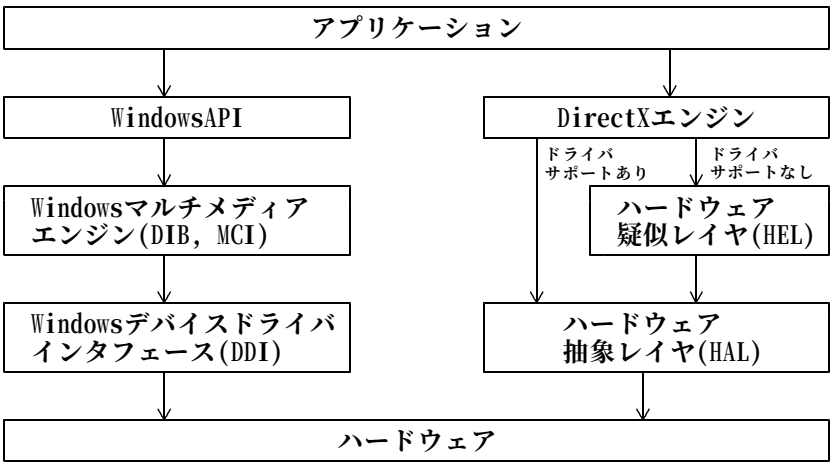
DirectXとは、グラフィックカードやサウンドカードなどのハードウェアを高速に扱うためのコンポーネント(部品)の総称です。Direct(直接)という言葉どおり、OSをとおさず、ハードウェアへの直接的なアクセスを提供しており、高速な処理を実現できます。

DirectXは、ハードウェアによって実装される部分(HAL)とソフトウェア的にエミュレートする部分(HEL)があり、これらは自動的に判別され、環境による機能の差を吸収できるようになっています。

HAL(Hardware Abstraction Layer：ハードウェア抽象レイヤ)は、アプリケーションソフトとハードウェアの間に入り、ハードウェアごとの振る舞いの違いを吸収するソフトウェアです。DirectXは、さまざまなハードウェアに対応していますが、これらの振る舞いの違いはHALが吸収しているため、プログラマは、その差を気にすることなく開発を行なうことができます。

HEL(Hardware Emulation Layer：ハードウェア疑似レイヤ)は、ハードウェアに実装されていない機能をソフトウェアでエミュレートするソフトウェアです。HALよりもスピードは劣るものの、あたかも機能をサポートしているかのような動作をします。ほとんどの機能はHELでサポートされているので、プログラマは、ハードウェアの機能の差を気にすることなく開発を行なうことができます。

このHALとHELにより、DirectXは環境に依存せず、ハードウェア支援が受けられる場合にはその能力を引き出すことができるようになっています。



DirectXの構成(DirectX9.0c)

○DirectX Graphics(version9)

画面にグラフィックを描画するコンポーネントであるDirect3DとDirect3DXの総称です。

○Direct3D(version9)

ポリゴンの描画や頂点座標変換、テクスチャマッピング、ブレンディング、フォグ、頂点バッファなど数多くの３Ｄの描画をサポートするコンポーネントです。グラフィックカードにDirect3Dアクセラレーション機能がある場合は、それを活用することで優れた3D描画処理が行えます。

○Direct3DX(version9)

Direct3Dによるアプリケーションの開発を支援するための拡張ライブラリです。テクスチャ、メッシュ、スタック行列、ｘファイル、エフェクト、算術関数といった3D関連だけでなく、2D描画を行うためのサーフェス、スプライトや、エラーメッセージを取得するライブラリも提供されています。

○DirectDraw(version7)

DirectX7以前でサポートされていたコンポーネントで、おもにビットマップの描画処理を行います。グラフィックカード上のメモリ(VRAM)への直接的なアクセス、高速なブロック転送、ダブルバッファおよびトリプルバッファによるフリッピング機能などを有しており、高速で滑らかなアニメーションを実現します。DirectX8からは、Direct3Dに統廃合されています。また、最近の環境では正常に動作しないケースも増えてきています。

○DirectX Audio(version8)

ＢＧＭや効果音を再生､録音するためのコンポーネントであるDirectMusicとDirectSoundの総称です。

○DirectMusic(version8)

主にMIDI音源を制御するコンポーネントです。MIDI音源のない環境でも、Roland GS音色データを搭載したソフトウェアMIDIエンジンにより、MIDIデータを再生することができます。DirectX SDK August 2007を最後にSDKから完全に削除され、将来的に廃止の予定となっています。

○DirectSound(version8)

Wave音源を制御するためのコンポーネントです。複数のサウンドを同時に再生でき、再生までのタイムラグも少ないので効果音を鳴らすのに適しています。サウンドデータを格納しているメモリ領域への直接的なアクセスができるため、サウンドデータを編集してエフェクトを加えることができます。

○DirectSound3D(version8)

サウンドを3D空間で再生するためのコンポーネントです。現実世界のような臨場感を再現することができます。

○DirectInput(version8)

キーボード、マウス、ジョイスティックなどの入力デバイスを制御するコンポーネントです。フォースフィードバック機能を使って反動を持たせたり、振動させたりすることができます。

○DirectShow(version9)

AVI、MPEG、Wave、MIDIといったさまざまなマルチメディアファイルを操作することができます。MP3(MEPG1 Layer-3 Audio)やDVD-Videoにも対応しています。DirectXからは削除されましたが、Windows標準の機能としてサポートされています。

○DirectPlay(version8)

ネットワークを利用した通信機能を提供するコンポーネントです。２台以上のコンピュータがネットワークを経由して互いに通信しあって協力、対戦するようなゲームを実現できます。また、「ロビーサーバー」も構築できるため、ネットワーク上のプレイヤーを探すための機能も実現できます。
(※DirectPlayは今後廃止の予定で、Microsoftは使用しないことを推奨しています)

○DirectSetup(version9)

DirectXのランタイム(DirectXを用いたアプリケーションを動かすためのドライバやDLLファイルなど)をインストールするためのコンポーネントです。更新の必要なコンポーネントを自動的に検索するので、最新版のランタイムを簡単にインストールすることができます。

DirectX SDK

DirectXを用いたアプリケーションを開発するには、DirectX SDKが必要になります。SDK(ソフトウェア開発キット)とは、C／C++用のヘッダファイル(.h)、リンクライブラリ(.lib)、サンプルプログラム、ドキュメント、ランタイムといった開発に必要なものをまとめたものです。
DirectX SDKは、Microsoftのホームページからダウンロードしたり、開発系の本や雑誌などに付属のCD-ROMから入手することができます。最新版は、「DirectX SDK June 2010」になります。

課　題

DirectX SDKをインストールし、VisualStudioで認識できるように設定しましょう。

ES Game Library

ES Game Libraryは、WindowsとDirectXを使ったゲームプログラミングを簡単に行えるようにしたゲーム用ライブラリです。Windowsとゲームの基本的な処理を行うフレームワーク群、DirectX Graphics, DirectX Audio, DirectInputをまとめたクラスライブラリに、ゲームループをコールバック式にするためのGameMain.hとGameMain.cpp、さらにクロスプラットフォームゲームライブラリXNAのように開発できるよう、関数名とクラスを定義したヘッダーファイルXNA.hから構成されています(ゲームループは無限ループのようにすることもできます)。

DirectX9.0cの基本的な機能を簡単な関数呼び出しで使用することができるようになっており、Win32 APIも使用できます。また、近年はハードウェアアクセラレータがなくなってしまったものの、豊富な描画命令を備えるGDIも使用できます。さらに、DirectX Graphicsにおいては、シェーダーにも対応できるので、GPUの性能を大きく引き出すことができます。

本来、WindowsとDirectXでゲームを作成するには、ウィンドウの管理やDirectXの制御など、かなりの量のコードを書く必要があります。特にデバイスの管理は複雑になる傾向があり、合わせて1,000行以上のプログラムを書かなければなりません。

ES Game Libraryでは、そういった面倒な部分をクラスに閉じ込めてあります。また、ソースコードも公開していますので、自由に変更・拡張して使用することができます。配布も自由です。

GameMainクラス(GameMain.h/GameMain.cpp)

ESライブラリでプログラミングの核となるのがGameMainクラスです。Windowsで動作するゲームの最小の機能のみ定義してあります。このクラスには、Initialize, LoadContent, UnloadContent, Update, Drawの５つの関数のひな形が用意されています。

Initialize関数

変数の初期化、コンフィグやユーザデータといった画像や音声以外のデータの設定を行います。

LoadContent関数

グラフィクスやサウンドといったマルチメディアデータの読み込みを行います。

UnloadContent関数

LoadContent関数で読み込んだグラフィクスやサウンドといったマルチメディアデータの解放を行います。

CやC++では、生成したリソースは解放しないとメモリに残ったままになるケースが多々あります(これをメモリリークやリソースリークと呼びます)。この関数には、リークを防ぐため、リソースを解放するプログラムを記述します。

ESライブラリでは、C++のデストラクタを使い、リークを防ぐための工夫をしていますが、必要に応じてリソースを解放するプログラムをここに記述します。

Update関数

ゲームの状態変更や背景、キャラクタの座標更新、衝突判定など描画以外の処理をします。デフォルトでは16.6ms毎、秒間60回呼ばれるようになっています。

Draw関数

画面に描画するための処理を記述します。

ゲームループの構成

ゲームループの構成方法には、内部処理と描画処理を無限ループ内で自由に実行する方法と、OSなどのシステムにより、必要に応じて自動的に実行されるコールバック式の２つが主に使われています。

無限ループ式は、古くからある方法で、OSなどが搭載されていない純粋なゲーム機でよく使われる方法です。現在もNintendoDSで使われています。名前のとおり、メインとなる関数で無限ループを構成し、そのなかでゲームの処理を実行し続けます。仕組みが単純なため、開発しやすいのが特長です。

コールバック式は、近年主流となっている方法です。(目に見える形での)ゲームループは構成せず、内部処理や描画処理を行う関数は、システムによって必要なときに呼び出されて実行されます。OSを搭載したPCや携帯機、ワークステーションに近い構成のゲーム機などで採用されています。無限ループ式に比べると、構成が少しだけ複雑になりますが、あらゆるプラットフォームに対応することができるというのが利点です。

ES Game Library必要環境

ES Game Libraryでゲーム開発を行うには、VisualC++とDirectX SDKの導入が必要になります。VisualC++は、仕様を満たすDirectX SDKがビルドできるバージョンであればどれでもかまいませんが、同梱しているのはVisualC++2005，2008，2010のものとなっています。開発自体は2005で行っています。

DirectX SDKは、本来は9.0c以降ならどのバージョンでもかまいませんが、同じ9.0cでも、年月によって引数が変更されたり、機能追加されたり(もしくは削除されたり)しているため、ID3DXFontインタフェースに機能追加があった以降のバージョン(2007以降)が必要です。それらの機能を使わないようにもしくは引数の変更を行えば、ビルド自体はできるようにはなっています。また、August2007に削除されたDirectMusicを使用しているので、DirectMusic関連のヘッダーファイルとライブラリファイルが必要です。さらにサウンド関連で、ogg形式に対応するため、ogg vorbis SDKも必要となっています。

これらは、SDKをインストールまたはコピーしただけでは認識されません。VC++2010版では、プロジェクトごとにフォルダを設定するようになっているため、プロジェクトプロパティでES Game Libraryの各フォルダにパスを設定済みとなっていますが、そのほかのバージョンでは、インクルードフォルダとライブラリフォルダの追加設定が必要です。

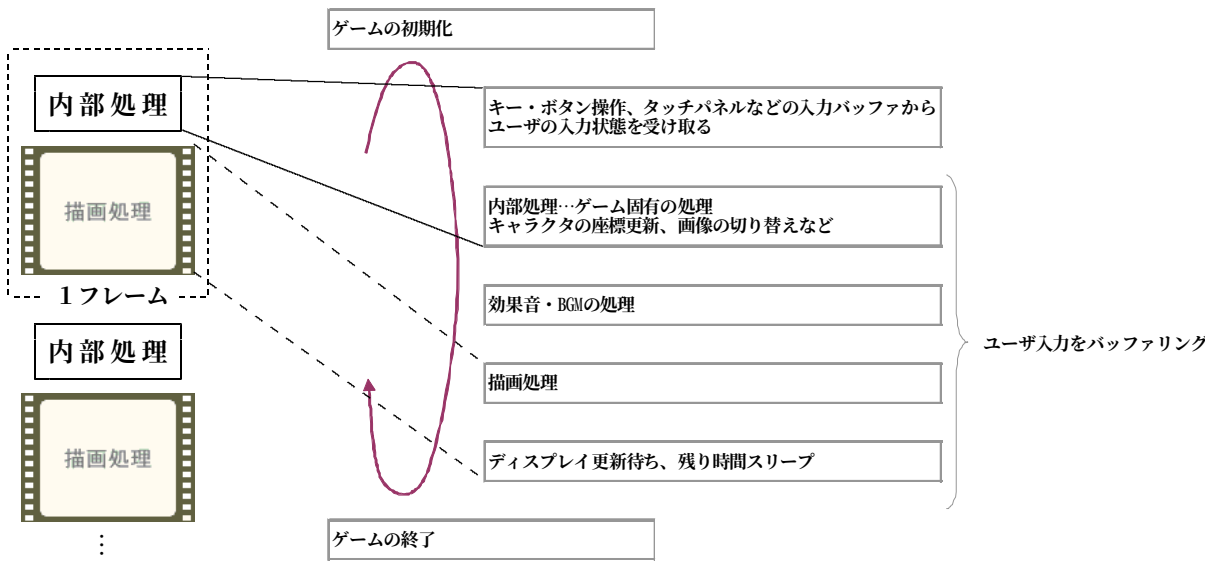
ゲームメインループ(ゲームループ)

ゲームプログラムは、キャラクタや背景など、いろいろな「もの」の状態を時間にそって更新し、画面に描画するプログラムと考えることができます。これを非常に単純化すると、

- 「もの」の状態を更新(内部処理＝Update関数)
- 「もの」を画面に描画(描画処理＝Draw関数)

の２つにまとめることができます。この処理の繰り返しがゲームのメインループになります。ループ１回ぶんを「１フレーム」と呼びます。１フレームごとに、キャラクタの位置や状態、背景やそのほかの表示状態などを更新し、その状態を反映して描画を行う、という処理を繰り返しています。

簡易図



ES Game Libraryでの開発

SDKのインストールと各フォルダの設定が終わったら、ES Game Libraryでの開発が行えます。.slnファイルをダブルクリックしてVisualC++を起動させてビルドすれば、空のウィンドウが表示されます。ESCキーを押すと終了です。これが、ES Game Libraryのフレームワークの仕事です。

これ以降のゲームプログラムをGameMain.hとGameMain.cppに記述していくことが、ゲーム開発の第一歩となります。

課　題

ES Game Libraryが使えるように開発環境を整備し、ソリューションをビルドしてプログラムを実行しましょう。成功すると、横長のウィンドウが表示されます。

ES Game Libraryの各フォルダ

ES Game Libraryでは、以下のフォルダが作成されています。

フォルダ	用途
Content	画像、サウンド、シェーダーなどの読み込みファイルを置くためのフォルダ
Debug	デバッグバージョンの実行ファイルがビルドされるフォルダ
Release	リリースバージョンの実行ファイルがビルドされるフォルダ
DirectX	DirectX関連ライブラリのソースおよびヘッダーファイルが格納されるフォルダ
Framework	ウィンドウ、基本ライブラリのソースおよびヘッダーファイルが格納されるフォルダ
GameScene	ゲームのシーン毎にクラス分けされたソースおよびヘッダーが格納されるフォルダ
Shader	エフェクトを記述したファイルをここに置いておくと、自動的にコンパイルされます
SDK	DirectXやogg vorbisのSDKが格納されています

ゲーム画像の準備

今回作成するシューティングゲームでは、以下のような画像を使用します。

目　的	背　景	プレイヤー	敵	プレイヤー弾	敵の弾	おしまい
画　像						
ﾌｧｲﾙ名	BG.jpg	PLY00.png～ PLY03.png	ENM00.png ENM01.png	SHT00.png～ SHT03.png	ENMSHT.png	OSHI.png
サイズ	1280 × 720	42 × 55	113 × 168	24 × 22	18 × 32	157 × 37
説　明	背景となる画像です。	プレイヤーが操作するキャラクタの画像です。 アニメーション用に、４コマ分用意してあります。	敵の画像がENM00.png、ダメージを与えたときの画像がENM01.pngです。	プレイヤーが発射する魔法弾です。 アニメーション用に、４コマ分用意してあります。	敵が発射する弾です。	ゲームが終了したときに表示する画像です。

課　題

シューティングゲームで使用する画像を読み込みファイルを置くためのフォルダに保存しましょう。

スクリーン座標

2Dでグラフィックを描画する場合、座標の指定はスクリーン座標で行います。スクリーン座標とは、画面のピクセルに１対１に対応する座標系のことです。スクリーン座標の原点は、左上にあります。また、ｙ軸の向きが逆になっていることも大きな特徴です。

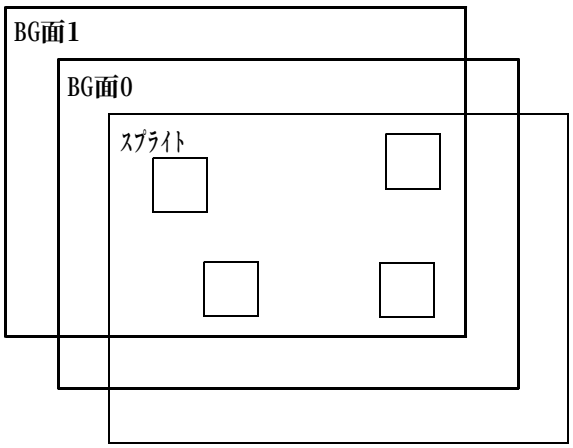
▲スクリーン座標系

スプライト

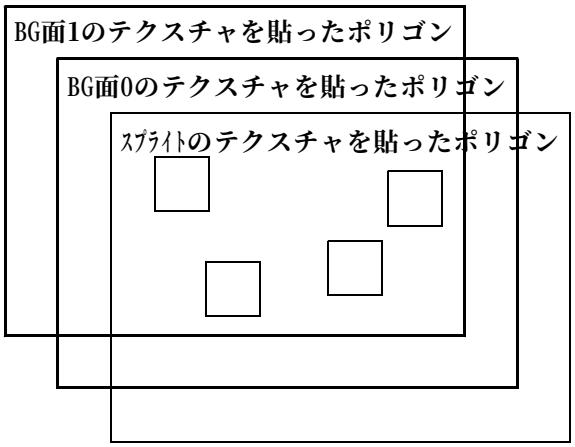
スプライトとは、キャラクターを高速に表示するための機能のことで、背景などほかの画像を壊すことなく独立して制御でき、透過色や重なり順序を指定することができます。

本来のスプライトは、ハードウェアで実現される機能で、背景画像(BG面)とは別に、多数の小さな画像を用意しておき、画面に合成して表示するものです。複数の背景やスプライトと重ね合わせて合成表示ができるようになっており、ほかのスプライトと重なったときに、どれを前面に表示するかが指定できるようになっています。CPUに負担をかけずにキャラクターを表示することができました。
現在では、CPUが高速化され、グラフィック機能も3Dに傾斜しているため、ハードウェアでスプライトをサポートしている機種はほとんどなく、ポリゴンとテクスチャを用いたり、SDKやライブラリでスプライトを擬似的に扱えるようにしています。

DirectX9も、2D表示はポリゴンを平面的に並べ、テクスチャを貼り付けて擬似的に再現する疑似スプライトです。内部では面倒な作業を行っているように感じますが、3Dの機能を用いて描画(レンダリング)されるので「スケーリング」「回転」「アルファブレンディング」といった機能をGPUで高速に行うことができます。



▲スーパーファミコンやセガサターンはBG面とスプライトをハードウェアで合成して画面に表示



▲DirectX9ではすべて「ポリゴン」＋「テクスチャ」で表示。重なり順序は描画する順序やｚバッファで指定

スプライトの読み込み

スプライトの読み込みと描画は、DirectX9本来の流れでは、テクスチャ画像を読み込んで描画領域の頂点データを宣言し、テクスチャを貼り付けるように設定したポリゴンを描画することで行います。これらの作業をDirect3D + ESライブラリにより、いつくかの関数を実行するだけで行うことができます。

画像を読み込み、スプライトとして扱えるようにするには、ESライブラリの「GraphicsDevice.CreateSpriteFromFile関数」にファイル名を指定するだけで行うことができます。この関数を実行すると、スプライトを制御するための変数が戻ってくるようになっているので、以下のように保存しておきます(関数を実行しただけでは、スプライトがなくなってしまいます)。

```
SPRITE    BG = GraphicsDevice.CreateSpriteFromFile( TEXT("bg.jpg") );
```

スプライトの描画

ESライブラリでのスプライトの描画は、SpriteBatch.Draw関数で行います。このときDraw関数の前にSpriteBatch.Begin関数を実行し、グラフィクスデバイスをスプライト(平面ポリゴン＋テクスチャ)の描画に適した状態にする必要があります。描画終了後にはSpriteBatch.End関数を実行し、描画が終了したことをグラフィクスデバイスに通知します。Begin～End関数内では、何度でもDraw関数の実行ができます。スプライトが実際に描画されるのはEnd関数を実行したときで、まとめてスプライトが描画(レンダリング)されるようになっています。

Draw関数は、スプライトを描画せず、スプライト描画リスト(スプライトバッチ)に登録するという作業をしています。End関数を実行したときに、リストに登録されたスプライトを指定順に並び替え、まとめて描画しています。

ESライブラリを使わず、DirectX9のみの場合も同じ流れになります。関数の名前が異なりますが、スプライトのBegin、Draw、Endという手順は同じです。

BeginSceneとEndScene

Direct3Dを用いて3Dグラフィクスを描画する場合、シーンの開始と終了をグラフィクスデバイスに通知する必要があります。シーンとは、映画などの「何々シーン」や「何々場面」と同じような、ゲーム内のワンシーン(世界観の一部)です。

3Dグラフィクスでは、3Dモデルを多数、空間内に置いて独自の世界観を表現します。この「モデルを空間内に置く」作業を始める宣言をするのが「シーンの開始」となります。作業終了を宣言するのが「シーンの終了」です。シーンの開始を宣言すると、グラフィクスデバイスが3Dグラフィクスを表現するのに適した状態になります。シーンの終了を宣言をすると、空間内に置かれたモデルがまとめて描画されるという仕組みです。

ESライブラリもDirect3Dを使用しているので、シーンの開始・終了宣言が必要となります。それぞれ「GraphicsDevice.BeginScene関数」「GraphicsDevice.EndScene関数」が対応しています。スプライトも3Dの機能を使っているので、これらの関数の実行が必要です。

課　題

背景とプレイヤーキャラクターを表示してみましょう。

(１)背景画像を保存するための変数を定義します。以下のプログラムをヘッダーファイル"GameMain.h"の"private:"の下に追加しましょう。

```
// 背景画像
SPRITE    BGSpr;
```

(２)背景を読み込みます。ソースファイル"GameMain.cpp"の"LoadContent関数"を以下のように変更しましょう。

```
void CGameMain::LoadContent()
{
    // TODO: use this.Content to load your game content here
    // 背景読み込み
    BGSpr = GraphicsDevice.CreateSpriteFromFile( TEXT("BG.jpg") );
}
```

(３)背景を描画します。ソースファイル"GameMain.cpp"の"Draw関数"を以下のように変更しましょう。

```
void CGameMain::Draw()
{
    GraphicsDevice.Clear(Color_CornflowerBlue);

    // TODO: Add your drawing code here
    GraphicsDevice.BeginScene();

    // スプライト描画
    SpriteBatch.Begin();

    // 背景
    SpriteBatch.Draw(BGSpr, Vector3(0.0f, 0.0f, 1.0f));

    SpriteBatch.End();

    GraphicsDevice.EndScene();
}
```

(４)プログラムを実行し、正しく描画されるかを確認しましょう。

(５)プレイヤー画像を保存するための変数を定義します。以下のプログラムをヘッダーファイル"GameMain.h"の"private:"の下に追加しましょう。

```
// プレイヤー
SPRITE   PlySpr;
```

(６)プレイヤー画像を読み込みます。以下のプログラムをソースファイル"GameMain.cpp"の"LoadContent関数"の適切な場所に追加しましょう。

```
// プレイヤー読み込み
PlySpr = GraphicsDevice.CreateSpriteFromFile( TEXT("PLY00.png"), Color(0, 0, 0) );
```

(７)プレイヤーを描画します。以下のプログラムをソースファイル"GameMain.cpp"の"Draw関数"の適切な場所に追加しましょう。

```
// プレイヤー
SpriteBatch.Draw(PlySpr, Vector3(0.0f, 0.0f, 0.0f));
```

(８)プログラムを実行し、正しく描画されるかを確認しましょう。

プレイヤーの移動

プレイヤーを移動しようとしてキーを押しても、まったく動きません。これは、移動のためのプログラムを記述していないためですが、もっとも問題なのは、つねに座標(０，０)に固定して描画しているという点です。このままでは、キー入力処理のプログラムを記述したとしても、キャラクターは移動できません。
キャラクターは、固定された数値ではなく、キーの入力によって数値を変更します。たとえば、右を１回押すたびに２つ移動する場合、(０，０)→(２，０)→(４，０)というように、ｘ座標の数値を増やしていかなければなりません。この動作を実現するために、座標に変数を導入します。

キーボード状態の取得

キーボードの状態は、Keyboard->GetState関数で取得できます。この関数を実行すると、キーの状態がKeyboardState型の変数に格納されます。

KeyboardState型の変数には、キーの状態を調べるための関数があります。IsKeyDown関数は、引数で指定したキーが押されているかを調べることができます。また、IsKeyUp関数は、引数で指定したキーが離されているかを調べることができます。

特定のキーの状態を調べるには、"Keys_"ではじまるキー定数またはDirectInputで定義される"DIK_"ではじまるキーボードデバイス定数を使用します。たとえば'Enter'キーは"Keys_Enter"、'A'キーは"Keys_A"となります。このように、"Keys_"にキーの名称をそのまま加えた名前となっています。

```
// キーボード状態の取得
KeyboardState    Key = Keyboard->GetState();

// キャラクター移動
if(KeyState.IsKeyDown(Keys_Left))
    // '←'が押されているときの処理
if(KeyState.IsKeyDown(Keys_Right))
    // '→'が押されているときの処理
```

課　題

キーの押下状態によって、プレイヤーを移動させましょう。

(１)プレイヤーのｘ座標とｙ座標を変数で管理します。プレイヤーのｘ座標とｙ座標を格納する変数をヘッダーファイル"GameMain.h"の適切な場所に追加しましょう。

```
// プレイヤー座標
float    PlyX, PlyY;
```

(２)ソースファイル"GameMain.cpp"の"Initialize関数"で変数を初期化し、プレイヤーの初期座標を設定します。CGameMain::Initialize関数を以下のように変更しましょう。

```
void CGameMain::Initialize()
{
    // TODO: Add your initialization logic here
    // 変数初期化
    PlyX = 0.0f;
    PlyY = 0.0f;
}
```

ｘ座標(PlyX)、ｙ座標(PlyY)ともに０を設定しているので、プレイヤーは左上に描画されます。これらの変数には、好きな値を設定しましょう。

(３)プレイヤーを描画プログラムを以下のように変更し、変数の座標にキャラクターが描画されるようにしましょう。

```
// プレイヤー
SpriteBatch.Draw(PlySpr, Vector3(PlyX, PlyY, 0.0f));
```

(４)この状態でプログラムを実行すると、変数PlyX, PlyYに設定した座標にプレイヤーが描画されます。

(５)キーの状態を調べ、キャラクターを移動させます。移動は、変数PlyX, PlyYを増減させることで行います。
ソースファイル"GameMain.cpp"の"Update関数"を以下のように変更しましょう。なお、(ア)～(ウ)は、各自考えてください。

```
int CGameMain::Update()
{
    // TODO: Add your update logic here
    // プレイヤー処理
    KeyboardState   Key = Keyboard->GetState();
    if(Key.IsKeyDown(Keys_Left))
        PlyX -= 3.0f;
    if(Key.IsKeyDown(Keys_Right))
        ※(ア)
    if(Key.IsKeyDown(Keys_Up))
        ※(イ)
    if(Key.IsKeyDown(Keys_Down))
        ※(ウ)

    return 0;
}
```

(６)実行して動作を確認しましょう。

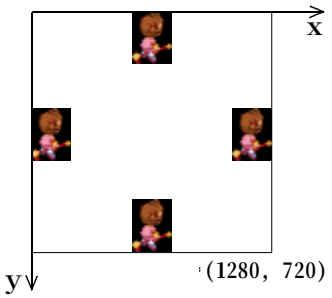
(７)キャラクターの移動範囲を画面内だけにしましょう。

・ヒント１
座標は、画像の左上を指しています。

・ヒント２
ｘ座標の最小値は、原点の0です。この値未満になるということは、原点より左側にいるということになり、画面外にいることになります。この場合は、ｘ座標を0にし、画面左に戻します。

・ヒント３
ｘ座標の最大値は、画面右端の1280になりそうですが、実は違います。ｘ座標は、キャラクターの左端の座標なので、ここが1280の場合は、すでに画面外に出ています。キャラクターの右端(左端 + 幅)が1280を越えているかどうかを調べるか、キャラクターの左端が、1280から幅を引いた値(1280 - 幅)を越えているかを調べます。越えている場合は、ｘ座標を1280から幅を引いた値にします。

・ヒント４
ｙ座標も同様に考えます。



敵の移動

敵は、プレイヤーと異なり、プログラムによって動かさなくてはなりません。動かし方はいろいろあり、一定のルートを通るだけのもの、一定のパターンで動くもの、プレイヤーや自分の状態によって異なる動作をするもの、時間によってパターンを変えるものなどさまざまな動作が考えられます。どのような動作をさせるかによって、ゲームの難易度や楽しさが変わってきます。

課　題

敵の画像を読み込んで描画し、さらにプログラムによって動かしましょう。

(１)敵の画像と座標を管理する変数を宣言します。以下のプログラムを適切な場所に追加しましょう。

```
// 敵
SPRITE    EnmSpr;
float     EnmX, EnmY;
```

(２)敵の座標変数を初期化します。以下のプログラムを適切な場所に追加しましょう。

```
// 敵
EnmX = 1160.0f;
EnmY = 0.0f;
```

ｘ座標(EnmX)に1160、ｙ座標(EnmY)に０を設定しているので、敵は右上に描画されます。この変数には、好きな値を設定しましょう。

(３)敵の画像を読み込みます。以下のプログラムを適切な場所に追加しましょう。

```
// 敵
EnmSpr = GraphicsDevice.CreateSpriteFromFile(TEXT("ENM00.png"), Color(0, 0, 0));
```

(４)敵を描画します。以下のプログラムを適切な場所に追加しましょう。

```
// 敵
SpriteBatch.Draw(EnmSpr, Vector3(EnmX, EnmY, 0.1f));
```

(５)プログラムを実行し、敵が描画されるかを確認しましょう。

(６)敵を動かす処理を作成します。

敵は、「動きデータ」を与えることによって動かすことにします。敵の行動は、８方向への移動と一時停止の９パターンとします。それぞれの行動に数字を割り当て、これを並べて動きデータとします。
動きデータは、以下の０～９の数字を並べることによって作成します。１フレームあたり１つの動きデータを処理します。

7 …↖(左上)	8 …↑(上)	9 …↗(右上)
4 …←(左)	5 …一時停止	6 …→(右)
1 …↙(左下)	2 …↓(下)	3 …↘(右下)
0 …動きデータ先頭へ戻る		

たとえば、下に２０回、上に２０回移動するというパターンを繰り返すようにするには、

動きデータ = 22222222222222222222888888888888888888880

というデータになります。

まず、動きデータを保存する変数と、何文字目の動きデータを処理するかを管理する変数を宣言します。以下のプログラムを適切な場所に追加しましょう。

```
// 敵動きデータ
tstring    EnmMoveData;
UINT       EnmMoveCount;
```

(７)EnmMoveCount変数を初期化します。以下のプログラムを適切な場所に追加しましょう。

```
EnmMoveCount = 0;
```

動きデータ変数EnmMoveDataは、あとで初期化します。

(８)敵の動きを処理するプログラムを作成します。以下のプログラムを適切な場所に追加しましょう。
なお、プログラム中の'?'には、'+'または'-'のいずれか適切なものを入れましょう。

```
// 敵移動
if(EnmMoveData.length() > EnmMoveCount) {
    if(EnmMoveData[EnmMoveCount] == '0')
        EnmMoveCount = 0;                          // 先頭に戻る
    switch(EnmMoveData[EnmMoveCount]) {
      case '1': EnmX ?= 4;  EnmY ?= 4;  break;  // 左下
      case '2':             EnmY ?= 4;  break;  // 下
      case '3': EnmX ?= 4;  EnmY ?= 4;  break;  // 右下
      case '4': EnmX ?= 4;              break;  // 左
      case '5':                         break;  // 一時停止
      case '6': EnmX ?= 4;              break;  // 右
      case '7': EnmX ?= 4;  EnmY ?= 4;  break;  // 左上
      case '8':             EnmY ?= 4;  break;  // 上
      case '9': EnmX ?= 4;  EnmY ?= 4;  break;  // 右上
    }
    EnmMoveCount++;                                // 次のデータへ
}
```

このプログラムでは、動きデータを１文字取りだし、switch文によって適切な処理に分岐させて敵の座標を増減させています。

(９)動きデータを定義します。以下のプログラムを適切な場所に追加しましょう。

```
// 動きデータ
EnmMoveData  = TEXT( "22222222222222222222222222222888888888888888888888888888880" );
```

－動きデータサンプル－

```
EnmMoveData = TEXT( "22222222222222244444444445555566666666668888888888888880" );
EnmMoveData = TEXT( "11111111111111111111111111111177777777777777777777777777770" );
```

(10)プログラムを実行すると、動きデータにそって敵が動きます。

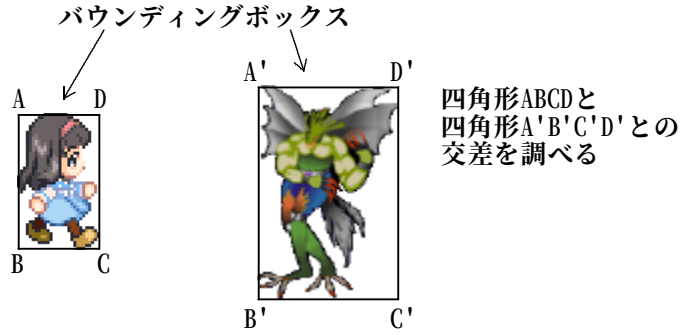
衝突判定

ゲームの中で、キャラクター同士の衝突検出(Collision Detection:衝突判定、当たり判定、ヒットチェックなどとも呼ばれます)は、ひんぱんに必要になります。シューティングゲームでは、プレイヤーと敵、プレイヤーの弾と敵、プレイヤーと敵の弾など、さまざまな種類の衝突検出が行われます。さらに、それらが１秒間に６０回の移動(秒間６０フレーム)が行われるとすると、そのぶんだけ衝突検出が必要となります。

バウンディングボックス

ゲームのようにリアルタイム性を追求する場合、キャラクターの複雑な形状を完全に考慮した衝突検出をするのはたいへんです。実際には、バウンディングボックス(Bounding Box:衝突範囲)と呼ばれる架空の領域を用いて、その領域の交差判定を行います。

２次元の場合には、矩形や円を用いることが多く、この領域ともう一方のキャラクターの交差判定を行うことで、擬似的に衝突を検出します。たいていの場合、キャラクターをすっぽりと包むような矩形にします。そうでない場合には、キャラクターのめり込みが起こる可能性が出てきます。



バウンディングボックスのプログラム

上の図のバウンディングボックスの頂点Ａの座標を(Ax, Ay)、頂点Ｃを(Cx, Cy)、頂点Ａ'を(A'x, A'y)、頂点Ｃ'を(C'x, C'y)と表します。このとき、２つのバウンディングボックスが交差しているかどうかを調べるプログラムは、次のようになります。

```
if(Ax <= C'x && Cx >= A'x && Ay <= C'y && Cy >= A'y)
    交差している
else
    交差していない
```

スクリーン座標系のｙ軸は、下方向に向いていることに注意しましょう。条件式をまとめると、

・頂点Ａが頂点Ｃ'の左側にあり、かつ、頂点Ｃが頂点Ａ'の右側にある
・頂点Ａが頂点Ｃ'の上側にあり、かつ、頂点Ｃが頂点Ａ'の下側にある

となります。この２つの条件を満たしたとき、２つの領域は交差しています。この条件を言い換えれば、四角形ＡＢＣＤの一部または全部が、四角形Ａ'Ｂ'Ｃ'Ｄ'の中にあるかどうかということです。

逆に言えば、

・頂点Ａが頂点Ｃ'の右側にある
・頂点Ｃが頂点Ａ'の左側にある
・頂点Ａが頂点Ｃ'の下側にある
・頂点Ｃが頂点Ａ'の上側にある

の条件を１つでも満たすと、絶対に交差していないということになります。これをＣ言語およびJavaで記述すると、次のようなプログラムになります。

```
// 衝突判定([ax1, ay1]が頂点A、[ax2, ay2]が頂点C、[bx1, by1]が頂点A'、[bx2, by2]が頂点C')
if(ax2 < bx1 || ax1 > bx2 || ay1 > by2 || ay2 < by1)
    // 衝突していない
else
    // 衝突している
```

課　題

衝突を判定する関数を作成しましょう。

(１)矩形の交差を判定するIsIntersect関数を作成します。関数本体を作成する前に、ヘッダーファイルで関数プロトタイプの宣言が必要です。

以下のプログラムをヘッダーファイル"GameMain.h"の適切な場所に追加しましょう。

```
// 関数プロトタイプ
bool IsIntersect(float ax1, float ay1, float ax2, float ay2,
                 float bx1, float by1, float bx2, float by2);
```

(２)IsIntersect関数本体を作成します。以下のプログラムをソースファイル"GameMain.cpp"の一番下に追加しましょう。

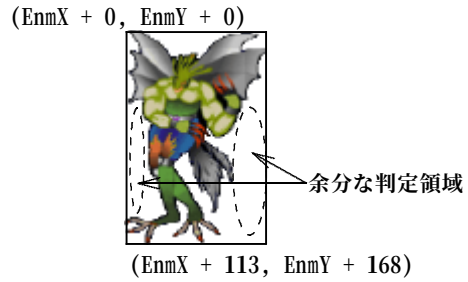
```
//--- 交差判定
bool CGameMain::IsIntersect(float ax1, float ay1, float ax2, float ay2,
                            float bx1, float by1, float bx2, float by2)
{
    if(ax2 < bx1 || ax1 > bx2 || ay1 > by2 || ay2 < by1)
        return false;   // 衝突していない
    return true;        // 衝突している
}
```

内部処理は、プレイヤーの移動→敵の移動→衝突判定という流れになります。

(４)プレイヤーと敵の衝突を検出し、衝突した場合に「あたり」と"出力"に表示されるようにしてみます。GameMain::Update関数に以下のプログラムを追加しましょう。

```
// プレイヤーと敵の衝突判定
if(IsIntersect(PlyX + 0.0f, PlyY + 0.0f, PlyX +  42.0f, PlyY +  55.0f,
               EnmX + 0.0f, EnmY + 0.0f, EnmX + 113.0f, EnmY + 168.0f)) {
    OutputDebugString( TEXT("あたり\n") );
}
```

プレイヤーと敵にバウンディングボックスを設定し、それが交差しているかどうかをIsIntersect関数で調べます。バウンディングボックスは、以下のようにキャラクターを完全に覆うようになっているので、画像が存在しない場所も衝突しているとみなされます。



左上の頂点(EnmX + 0, EnmY + 0)の数値を増やすと、領域が右下に狭まります。同様に、右下の頂点(EnmX + 113, EnmY + 168)の数値を減らすと、頂点が左上に狭まります。
たとえば、左上の頂点を(EnmX + 37, EnmY + 34)、右下の頂点を(EnmX + 50, EnmY + 50)とすると、顔の部分だけが衝突領域になります。

課　題

プレイヤーと敵にライフを設定し、画面上部に表示しましょう。また、スコアも表示しましょう。

(１)ライフは、変数で管理します。プレイヤーと敵のライフを格納する変数を宣言します。変数PlyHPとEnmHPを以下のように宣言しましょう。

```
// プレイヤーHP
int   PlyHP;

// 敵HP
int   EnmHP;
```

(２)ライフの初期値を設定します。以下のように初期化しましょう。

```
PlyHP = 5;
EnmHP = 10;
```

(３)ライフを描画します。以下のプログラムを適切な場所に追加しましょう。

```
// HP描画
SpriteBatch.DrawString(DefaultFont, TEXT("LIFE"), Vector2(120.0f, 0.0f), Color_Cyan);
SpriteBatch.DrawString(DefaultFont, Vector2(160.0f, 0.0f), Color_Pink,
                       TEXT("Player %d"), PlyHP);
SpriteBatch.DrawString(DefaultFont, Vector2(240.0f, 0.0f), Color_Lime,
                       TEXT("Enemy %d" ), EnmHP);
```

※GraphicsDevice.DrawString関数は、文字を描画する関数です。

(４)スコアを格納する変数を宣言します。以下のプログラムを適切な場所に追加しましょう。

```
int   Score;
```

(５)スコアを初期化します。以下のプログラムを適切な場所に追加しましょう。

```
// スコア
Score = 0;
```

(６)スコアを描画するようにします。以下のプログラムを適切な場所に追加しましょう。

```
// スコア描画
SpriteBatch.DrawString(DefaultFont, Vector2(0.0f, 0.0f), Color_White,
                       TEXT("SCORE %d"), Score);
```

課　題

プレイヤーまたは敵のライフが０以下になったらゲームが終了するようにしましょう。また、そのときに「おしまい」の画像を表示するようにしましょう。

(１)以下の方法(アルゴリズム)で作成します。

１．ゲームが進行している状態と、ゲームが終了した状態(ゲームオーバー)の２つの状態が存在することになります。２つの状態によって、処理を分岐させます。
２．ゲームの状態は、変数で管理します。
３．上記の変数が１なら通常状態、２ならゲームオーバーとします。
４．ゲームオーバーなら「おしまい」と表示します。そのとき、ゲームは進行させません。

(２)ゲームの状況を示す変数を宣言します。変数GameModeを以下のように宣言しましょう。

```
// ゲームモード
int   GameMode;
```

変数GameModeには、通常状態なら「１」、ゲームオーバーなら「２」を設定します。ゲームを拡張する場合、０ならタイトル、３ならエンディングというようにしていきます。

(３)変数GameModeを初期化します。ゲームを起動したときは、通常状態から始まるようにします。以下のように初期化しましょう。なお、「？」は各自考えてください。

```
// ゲームモード設定
GameMode = ?;
```

(４)「おしまい」画像を読み込みます。画像を格納する変数OsiSprを以下のように宣言しましょう。

```
// おしまい
SPRITE   OsiSpr;
```

(５)画像を読み込みます。以下のプログラムを適切な場所に追加しましょう。

```
// おしまい
OsiSpr = GraphicsDevice.CreateSpriteFromFile( TEXT("OSHI.png"), Color(0, 0, 0) );
```

(６)内部処理をゲームの状態によって分岐させます。ゲームオーバーの場合、「おしまい」と表示するだけなので、内部処理を行う必要はありません。ゲームオーバーでない場合は、これまでどおりプレイヤー、敵、衝突検出を行います。
内部処理を以下のように変更しましょう。なお、「？」は各自考えてください。

```
int CGameMain::Update()
{
    // TODO: Add your update logic here
    // ゲームモード分岐
    if(GameMode == ?)
        return 0;

    // プレイヤー処理
    KeyboardState   Key = Keyboard->GetState();
            ⋮
```

この変更により、ゲームオーバーの場合は内部処理が実行されなくなります。

(７)ゲームオーバーの場合、「おしまい」と表示するようにします。描画処理を以下のように変更しましょう。なお、「？」は各自考えてください。

```
// おしまい描画
if(GameMode == ?)
    SpriteBatch.Draw(OsiSpr, Vector3(560.0f, 340.0f, 0.0f));
```

(８)プレイヤーまたは敵のライフが０以下になったら、状態をゲームオーバーに変更します。この処理は、内部処理の最後に行います。
以下のプログラムを適切な場所に追加しましょう。

```
// おしまい判定
if(PlyHP <= 0 || EnmHP <= 0)
    GameMode = 2;
```

追加した判定により、どちらかのライフが０以下になると変数GameModeが２になり、ゲームオーバー状態となります。この状態になると、(６)により内部処理が実行されなくなり(＝ゲームが進行しない)、(７)により「おしまい」と表示されるようになります。

(９)プレイヤーと敵が衝突したら、プレイヤーのライフを０にします。以下のプログラムを適切な場所に追加しましょう。なお、「OutputDebugString(TEXT("あたり\n"));」は削除してかまいません。

```
PlyHP = 0;
```

(10)プレイヤーと敵が衝突した場合、プレイヤーのライフが０になってゲームオーバーになるかどうか確認しましょう。

課　題

プレイヤーが弾を発射できるようにしましょう。

(１)以下の方法(アルゴリズム)で作成します。

・スペースキーが押されたときに弾を発射
・発射できるのは１発のみ
・言い換えれば、画面に弾がないとき発射可能
・発射された弾は右に移動する
・画面の端または敵に当たると消滅(＝未発射状態となる)
・敵に当てるとダメージ１(敵のライフが１減る)、スコア1,000点加算
・弾は、「発射されている(画面内に存在する)状態」と「発射されていない(画面内に存在しない)状態」という２つの状態が存在する
・弾の状態は、変数で管理

(２)弾を管理する変数を宣言します。必要な変数は以下のものです。

・画像(PstSpr:SPRITE型)
・発射されているかどうか(pstShot:bool型…trueは発射中、falseは未発射)
・ｘ座標(PstX:float型)
・ｙ座標(PstY:float型)

以上の変数を宣言を適切な場所に作成しましょう。

(３)変数を初期化します。以下のプログラムを適切な場所に追加しましょう。

```
// プレイヤーショット
PstShot = false;
```

弾の初期化は、発射状態をfalseにして未発射状態にしておくだけで十分です。座標は発射したときに、プレイヤーの座標をもとに決定します。

(４)プレイヤーのショット画像を読み込むプログラムを適切な場所に作成しましょう。

(５)弾が未発射状態でスペースキーが押されたら弾を発射するようにします。以下のプログラムを適切な場所に追加しましょう。なお、「？」は各自考えてください。

```
// ショット発射
if(Key.IsKeyDown(Keys_?????)) {          // スペースキーが押されているか調べる
    if(PstShot == ?????) {               // 発射されていないか調べる
        PstShot = ????;                  // 発射状態にする
        PstX    = PlyX + 36;             // 座標の設定(杖の先)
        PstY    = PlyY + 22;
    }
}
```

弾が発射されていない状態(変数PstShotがfalse)のとき、スペースキーが押されたら弾を発射状態(変数PstShotをtrue)にし、弾の座標をプレイヤーの座標をもとに変数PstX, PstYに設定します。

(６)弾を描画するようにします。弾を描画するのは、発射されているときだけです。発射されていないときに描画してはいけません。
以下のプログラムを適切な場所に追加しましょう。

```
// プレイヤーショット描画
if(※ここは各自考えましょう)
    SpriteBatch.Draw(PstSpr, Vector3(PstX, PstY, 0.0f));
```

弾を描画するのは、発射されているとき(変数PstShotがtrueのとき)だけなので、描画の前に判定を行い、条件を満たすときだけ描画します。

(７)弾を発射できるようになったので、プログラムを実行して動作を確認してみましょう。

(８)弾を移動させる処理を作成します。必要な処理は、以下のものです。

・弾を移動させる
・弾が完全に画面外に出たら弾を未発射状態にする
・ただし、これらの処理を行うのは、弾が発射されているときだけ

以下のプログラムの「？」を埋め、適切な場所に追加しましょう。

```
// プレイヤーショット処理
if(※ここは各自考えましょう) {  // 弾が発射されているか調べる
    PstX += 6.0f;               // 弾の移動

    // 弾が画面内にあるか調べる
    // (弾の領域と画面の領域が交差しているか調べる＝交差していなければ、画面外にある)
    if(IsIntersect(PstX, PstY, PstX + 24.0f, PstY + 22.0f,
                   0.0f, 0.0f, 1280.0f, 720.0f) == false)
        PstShot = ?????;        // 未発射状態にする
}
```

(９)以上で弾が移動するようになったので、プログラムを実行して確認してみましょう。

(10)弾と敵の衝突検出を行います。衝突した場合、敵のライフを１減らし、スコアに1,000を加算します。以下のプログラムの「？」を埋め、適切な場所に追加しましょう。

```
// プレイヤーショットと敵の衝突検出
if(※ここは各自考えましょう) {  // 発射されているときだけ判定する
    if(???????????(PstX,         PstY,         PstX + 24.0f, PstY + 22.0f,
                   EnmX + 37.0f, EnmY + 34.0f, EnmX + 50.0f, EnmY + 50.0f)) {
        PstShot  = ?????;       // 弾を消す
        EnmHP   -= 1;           // 敵のHPを減らす
        Score   += 1000;        // スコアに1000を加算
    }
}
```

プレイヤーの弾と敵の衝突検出も、弾が発射されているときだけ行います。弾は、敵の領域を調整し、顔の部分だけにしか当たらないようにしています。
衝突した場合、弾を消し(状態を未発射にする)、敵のライフから１を引き、スコアを加算します。

課　題

敵が弾を発射するようにしましょう。

(１)以下の方法(アルゴリズム)で作成します。

・発射できるのは１発のみ
・言い換えれば、画面に弾がないとき発射可能
・発射する確率は、条件を満たしたときに１０％の確率
・発射された弾は左に移動する
・画面の端またはプレイヤーに当たると消滅(＝未発射状態となる)
・プレイヤーが当たるとダメージ１(プレイヤーのライフが１減る)
・プレイヤーの弾と敵の弾が当たった場合は、両方とも消滅し、スコアに100を加算
・弾は、「発射されている(画面内に存在する)状態」と「発射されていない(画面内に存在しない)状態」という２つの状態が存在する
・弾の状態は、変数で管理

(２)以下のプログラムを適切な場所に追加しましょう。なお、「？」は各自考えてください。

・乱数を扱うための変数の宣言
```
Random   Rand;
```

・敵の弾を管理する変数の宣言
```
// 敵ショット
SPRITE   EstSpr;
bool     EstShot;
float    EstX, EstY;
```

・敵の弾を初期化するプログラム
```
EstShot = false;
```

・敵の弾の画像を読み込むプログラム
```
// 敵ショット読み込み
EstSpr = GraphicsDevice.CreateSpriteFromFile(TEXT("ENMSHT.png"), Color(0, 0, 0));
```

・敵が弾を発射するプログラム
```
// 敵ショット発射
if(※ここは各自考えましょう) {       // 発射されていない
    if(Rand.Next(10) % 10 == 0) {    // １０%の確率
        EstShot = ????;              // 発射状態にする
        EstX    = EnmX + 24;         // 座標の設定(口のあたり)
        EstY    = EnmY + 32;
    }
}
```

敵の弾が未発射のとき、０～９のランダムな数値を作成し、それが０のとき(＝１／１０の確率)に弾を発射します。

・敵の弾を描画するプログラム
```
if(※ここは各自考えましょう)
    SpriteBatch.Draw(EstSpr, Vector3(EstX, EstY, 0.1f));
```

プレイヤーの弾と同様に、発射されているときだけ描画します。

(３)敵のショットを移動するプログラムを作成しましょう。

(４)プレイヤーと敵のショットの衝突を判定するプログラムを適切な場所に追加しましょう。

```
// プレイヤーと敵の弾の衝突判定
if(※ここは各自考えましょう) {  // 発射されているときだけ判定する
    if(???????????(PlyX, PlyY, PlyX + 42.0f, PlyY + 55.0f,
                   EstX, EstY, EstX + 18.0f, EstY + 32.0f)) {
        EstShot = ?????;         // 弾を消す
        PlyHP  -= 1;             // プレイヤーのHPを減らす
    }
}
```

(５)プレイヤーのショットと敵のショットの衝突を判定するプログラムを適切な場所に作成しましょう。なお、この場合、スコアに100を加算してください。