目次

第3章 落ち物パズル

第4章 ボールアクションゲーム

第5章 横スクロールアクション

213

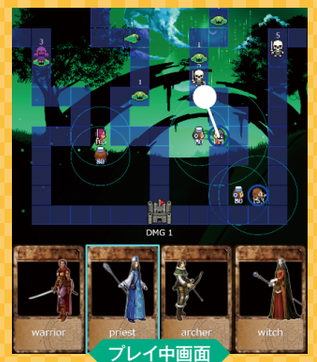
タイトル画面


プレイ中画面

第6章 タワーディフェンス

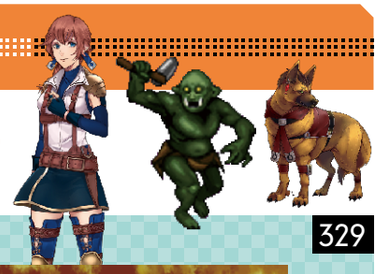
263

タイトル画面



プレイ中画面

第7章 ロールプレイングゲーム 前編

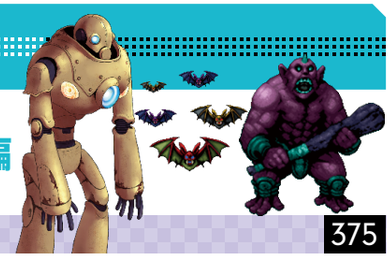


329

タイトル画面

プレイ中画面

第8章 ロールプレイングゲーム 後編



375

2-2 この章で制作するゲーム内容

この章では、往年の2Dスクロールシューティングゲームの作り方を解説し、ゲーム開発のテクニックとアルゴリズムを学んでいきます。ここでは、どのような内容のゲームを制作するかを見ていきましょう。

ゲーム画面

プレイヤーの操作する機体を「自機」、敵として登場する機体を「敵機」と称して説明します。

自機を操作し、敵や障害物を避けながら、弾を撃って敵機を撃ち落としていくゲームで、次のような画面になります。

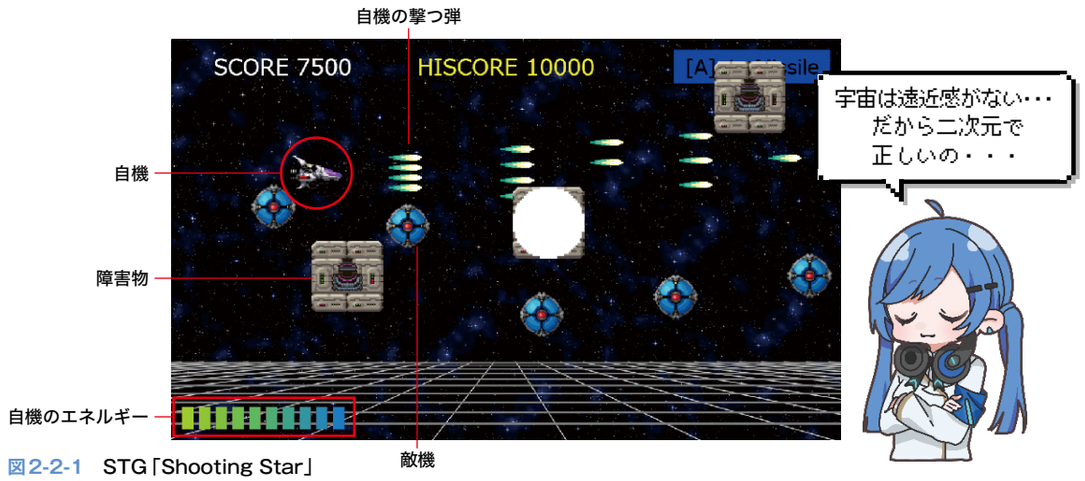


図2-2-1　STG「Shooting Star」

ルールと操作方法

ゲームルールと操作の仕方を説明します。

- 画面は右から左へスクロールし、画面右から敵が出現する
- 方向キーで自機を8方向に動かし、スペースキーで弾を撃つ
- エネルギー制とし、敵機や敵の弾に触れるとエネルギーが減り、0になるとゲームオーバー
- エネルギーは出現するアイテムを取ると回復する
- 武器がパワーアップする2種類のアイテムが出現する
  1つは複数の弾が同時に発射されるようになるもの
  もう1つは貫通レーザーを撃てるようになるもの

パソコンのキーボードだけでなく、スマートフォンでもプレイできるようにします。自動で弾を発射する機能と、タップした位置に自機が移動する仕組みを組み込みます。スマートフォン対応は2-12で行います。

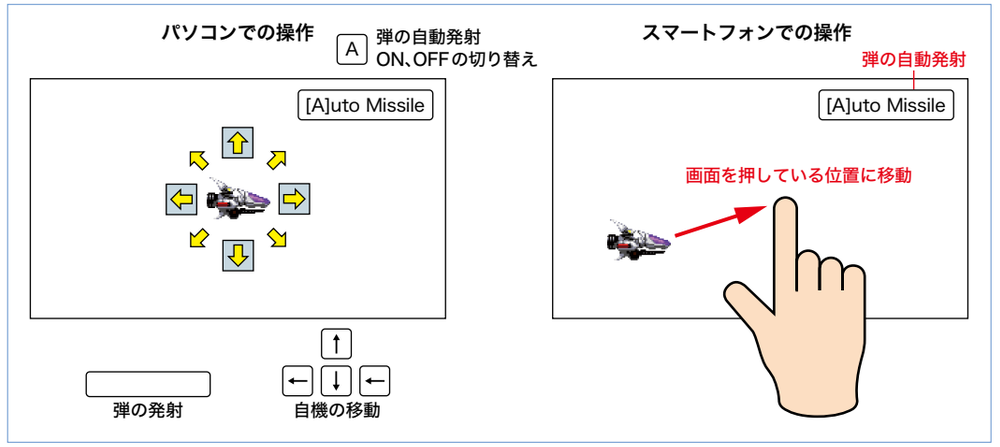


図2-2-2　操作方法

ゲームのフロー

ゲーム全体の流れを示します。この章で制作するSTGは、多くのゲームに共通する、タイトル画面→ゲームをプレイ→ゲーム結果の表示という大きな処理の流れを組み込みます。

次に、ゲームのメインの部分の流れを示します。

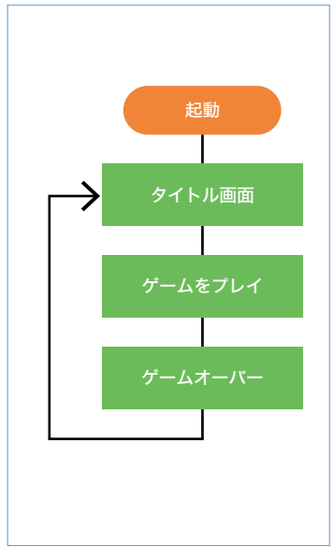


図2-2-3　全体の流れ

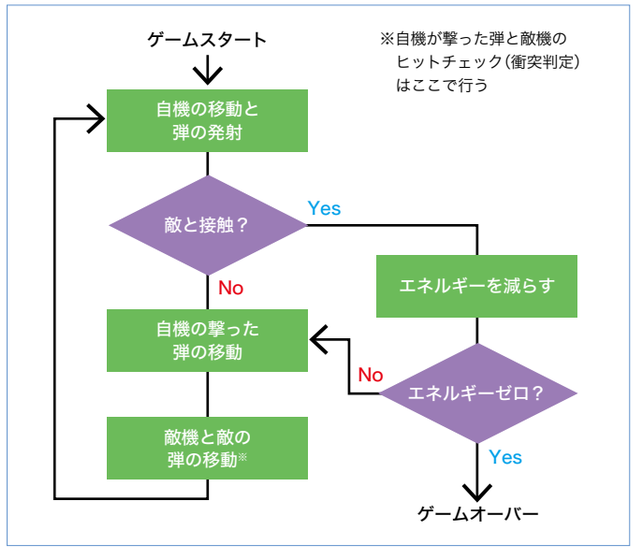


図2-2-4　主要部分の流れ

```
function mainloop() {
    drawPzl();
}

var masu = [
    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
    [-1, 1, 0, 0, 0, 0, 0, 0, 1,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 1, 0, 0, 0, 0,-1],
    [-1, 0, 0, 1, 1, 1, 0, 0, 0,-1],
    [-1, 0, 0, 0, 1, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 0, 0, 0, 0, 0, 0, 0, 0,-1],
    [-1, 1, 0, 0, 0, 0, 0, 0, 1,-1],
    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
];

function drawPzl() {//ゲーム画面を描く関数
    var x, y;
    drawImg(0, 0, 0);
    for(y=1; y<=11; y++) {
        for(x=1; x<=7; x++) {
            if(masu[y][x] > 0) drawImgC(masu[y][x], 80*x, 80*y);
        }
    }
}
```

メイン処理を行う関数
ゲーム画面を描く関数を呼び出す

マス目を管理する二次元配列
周囲1マスを余白とし、そこに-1を代入
値1のところにタコが入る
値0は空白のマスになる

ゲーム画面を描く関数

背景画像を表示する
二重ループの繰り返しで

配列の値が1ならタコの画像を描く

配列の説明

masu[][]	マス目を管理する二次元配列

起動時に1回だけ実行されるsetup()関数と、毎フレーム実行されるメインループのmainloop()関数に、それぞれ必要な処理を記述しています。

setup()関数では画面の大きさ（キャンバスのサイズ）を指定し、背景とタコの画像を読み込んでいます。このゲームは画面サイズを幅960ドット、高さ1200ドットで設計します。

mainloop()関数ではゲーム画面を描くdrawPzl()関数を実行しています。

ブロックを描く

drawPzl()関数では、二重ループのfor文で外周を除く配列全体を調べ、タコが配置されている（値1）ならマス目上にタコの画像を描きます。その部分を抜き出して説明します。

```
for(y=1; y<=11; y++) {
    for(x=1; x<=7; x++) {
        if(masu[y][x] > 0) drawImgC(masu[y][x], 80*x, 80*y);
    }
}
```

変数yを用いたfor文の中に、変数xを用いたfor文があります。

yの値は1から始まり、xの値は1→2→3→4→5→6→7と変化し、masy[y][x]の値が1であればタコの画像を表示します。

xの繰り返しが終わるとyの値は2になり、再びxの値が1→2→3→4→5→6→7と変化し、masy[y][x]が1ならタコを描きます。

xの繰り返しが終わるとyの値は3になり、同様にyが11、xが7になるまで配列の値を調べ、1ならタコを描きます。以上の処理の流れを図示すると次のようになります。

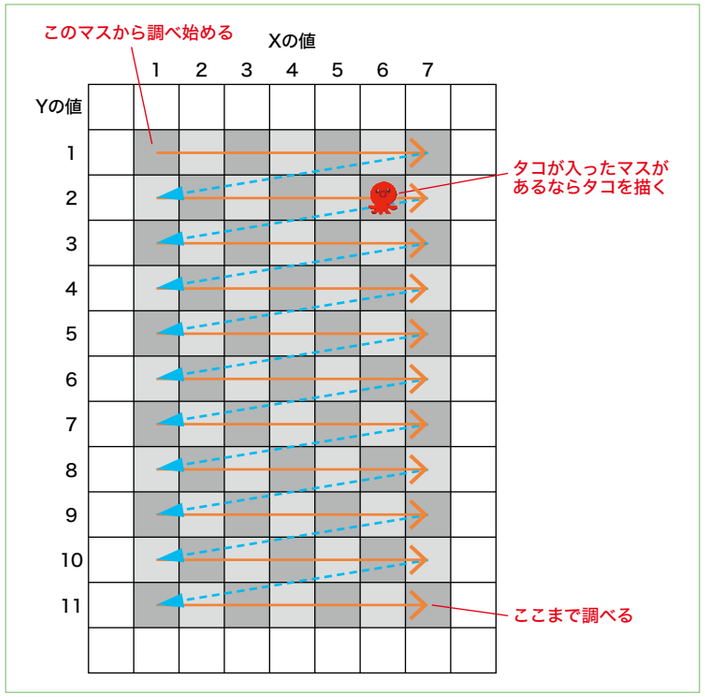


図3-3-4 二重ループのyとxの値の変化

二次元配列の値について

この落ち物パズルの二次元配列masu[y][x]の値をまとめます。

表3-3-2 マス目を管理する二次元配列の値

masu[y][x]の値	何の値か
-1	ブロックを動かしたり、揃ったかの判定を簡潔に記述するために代入しておく
0	空白（値0のマスには何も入っていない）
1～6	タコ、イカなどの6種類のブロック

二次元配列の外周に-1を代入する理由は、3-4のブロックを動かす処理と、3-7のブロックが揃ったかを判定する処理で明らかになります。ここではマス目を管理する配列の外周1マスを余分に確保していることを頭に入れ、次へ進みましょう。

4-9 ボール同士の衝突

ここでは、ボールの衝突と跳ね返りの動きをプログラミングしていきます。

“速さ”と“速度”について

ここから先は、ボールのX軸方向とY軸方向の1フレーム当たりの移動量（ドット数）を“速さ”、それらの2つの速さを合成したものを“速度”と呼んで説明します。速度とは、移動する物体の位置の変化を表すベクトル量であり、物理や数学に準じる呼び方で説明します。

図4-9-1 速さと速度

ボールの速度を入れ替える

プログラムのわかりやすさと作りやすさを重視し、このゲームは簡易的な計算でボールを跳ね返らせます。ボール同士が衝突した時、跳ね返らせる最も簡単な方法は、2つのボールの速度を入れ替えることです。

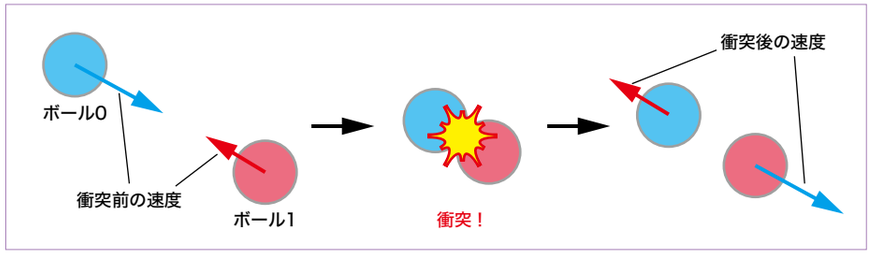


図4-9-2 ボールの速度の入れ替え

この方法なら、ボール0の速さの値をボール1の速さにし、ボール1の速さの値をボール0の速さにするだけで済みます。

次のHTMLを開き、ボールの速度を入れ替えるプログラムの動作を確認します。女の子のボールを飛ばし、モンスターの絵柄のボールに当て、それぞれの動きを見てみましょう。

動作の確認 ball0409.html

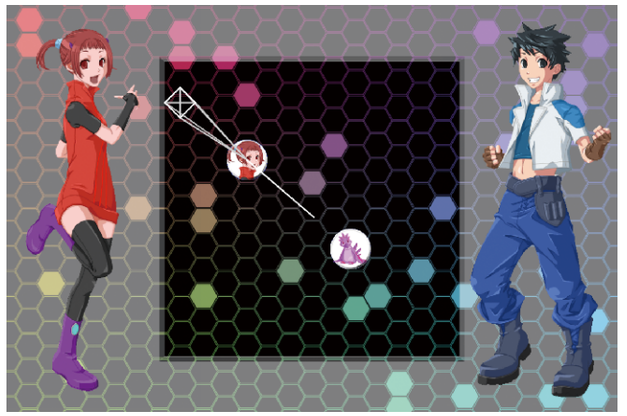


図4-9-3 動作画面

※このプログラムに新たな画像は用いていません。

ソースコード ball0409.jsより抜粋

■ボールを動かすmoveBall()関数に、ヒットチェックと速度の入れ替えを追記（太字部分）

```
function moveBall() {//ボールを動かす関数
    for(var i=0; i<BALL_MAX; i++) {
        ballX[i] = ballX[i] + ballVx[i];
        ballY[i] = ballY[i] + ballVy[i];
        if(ballX[i]< 340 && ballVx[i]<0) ballVx[i] = -ballVx[i];
        if(ballX[i]> 860 && ballVx[i]>0) ballVx[i] = -ballVx[i];
        if(ballY[i]< 140 && ballVy[i]<0) ballVy[i] = -ballVy[i];
        if(ballY[i]> 660 && ballVy[i]>0) ballVy[i] = -ballVy[i];
        ballVx[i] = ballVx[i] * 0.95;
        ballVy[i] = ballVy[i] * 0.95;
    }
    if(getDis(ballX[0], ballY[0], ballX[1], ballY[1]) <= 80) {
        var vx = ballVx[0];
        var vy = ballVy[0];
        ballVx[0] = ballVx[1];
        ballVy[0] = ballVy[1];
        ballVx[1] = vx;
        ballVy[1] = vy;
    }
}
```

2つのボールの距離が80ドット以下なら速度を入れ替える

※このプログラムに新たな変数は追加していません。

WWS.jsに備わる二点間の距離を求めるgetDis()関数で2つのボールの距離を求めています。ボールの半径は40ドットなので、ボールの中心間の距離が80以下なら衝突しています。その場合は太字で示したように、ballVx[0]とballVx[1]の値、及びballVy[0]とballVy[1]の値を入れ替えます。

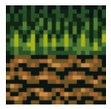
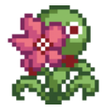
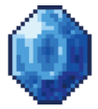
5-3 マップデータの管理

この章では各ステージの構成を「地形」と呼ぶことにします。その地形を定義する数値がマップデータです。
ゲーム画面を構成するマップデータの管理からプログラミングを始めていきます。

マップチップについて

ゲーム画面を構成する最小単位の画像を**マップチップ**と呼びます。これから制作するゲームで用いるマップチップとマップデータの値は次のようになります。

表5-3-1 マップチップとデータの値

値	0	1	2	3	4	5	6
マップチップ	無し						

これらの画像で地形を構成します。

値0はキャラクターが自由に移動できる空間です。何も表示する必要が無いので、マップチップの画像はありません。

1と2は**壁**になります。壁とはキャラクターが入ることができない場所を意味するゲーム用語です。このゲームでは1と2のチップの上にプレイヤーキャラが載り、左右に移動できるようにします。キャラクターが上に載ることを説明する時には、1と2を**床**と呼びます。

3と4はプレイヤーキャラが触れてはいけないもので、接触するとゲームオーバーになるようにします。

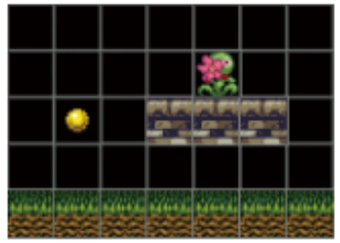
5は取るとタイム（残り時間）とスコアが増える宝で、ゴールドパールと呼ぶことにします。

6はゴール地点にあるクリスタルで、そこまで辿り着くとステージクリアです。

二次元配列で地形を定義する

2D（二次元）の画面構成のコンピューターゲームは、一般的に二次元配列でマップデータを扱います。二次元配列の扱い方は、3章の落ち物パズルで学びましたが、ここで改めて説明します。

例えば次の図のような地形は、右に羅列した数のように、マップチップに割り当てた番号を二次元配列でデータとして定義します。



0,0,0,0,0,0,0,
0,0,0,0,4,0,0,
0,5,0,2,2,2,0,
0,0,0,0,0,0,0,
1,1,1,1,1,1,1,

図5-3-1 数値による地形の定義

何もない空間は0、壁は1と2、敵は4、宝は5になっています。

二次元配列は、次の図のように、X軸方向とY軸方向の2つの添え字でデータを管理します。

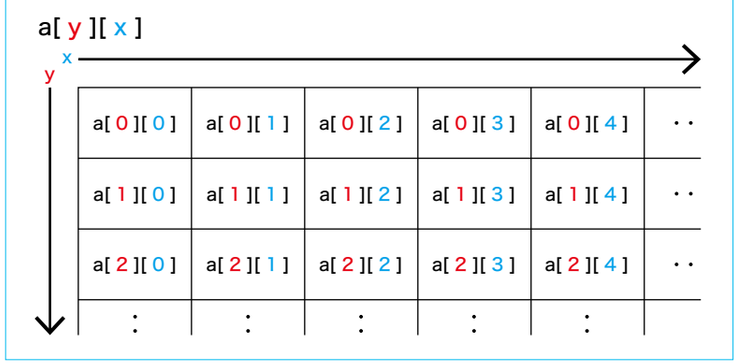


図5-3-2 二次元配列

地形を定義する

マップデータを二次元配列で管理するプログラムを確認します。次のHTMLを開いて動作を確認しましょう。**図5-3-3**のような実行画面が表示されます。

動作の確認 action0503.html

図5-3-3 実行画面

6-3 通路を定義する

このタワーディフェンスでは、4本ある通路のいずれかを敵が通って攻めてくるようにします。その通路を定義するところからプログラミングを始めます。

二次元配列を用いる

背景のパーツが格子状に並ぶ2Dゲームの地形は、二次元配列を使ってデータを保持します。このタワーディフェンスでは、次のような通路を二次元配列で定義します。矢印は通路の向き、すなわち敵が進む方向です。Cはプレイヤーが守る城です。

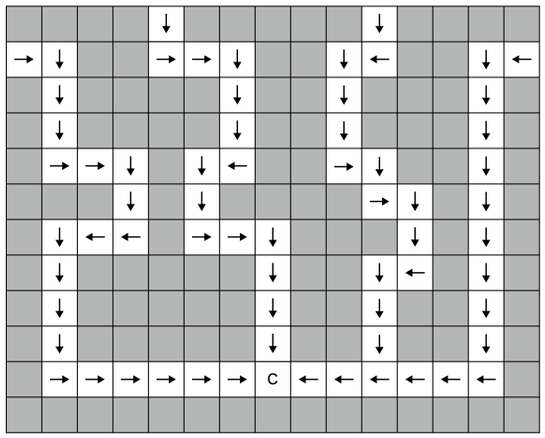


図6-3-1 タワーディフェンスの通路

それらを次の値で定義します。

表6-3-1 通路の向きと城のデータ値

1	2	3	4	5
↑	↓	←	→	C

二次元配列にすると、次のようになります。値0は何もないマスとします。

```
var stage = [
    [0,0,0,0,2,0,0,0,0,0,2,0,0,0,0],
    [4,2,0,0,4,4,2,0,0,2,3,0,0,2,3],
    [0,2,0,0,0,0,2,0,0,2,0,0,0,2,0],
    [0,2,0,0,0,0,2,0,0,2,0,0,0,2,0],
    [0,4,4,2,0,2,3,0,0,4,2,0,0,2,0],
    [0,0,0,2,0,2,0,0,0,0,4,2,0,2,0],
    [0,2,3,3,0,4,4,2,0,0,0,2,0,2,0],
    [0,2,0,0,0,0,0,0,2,0,0,2,3,0,2,0],
    [0,2,0,0,0,0,0,0,2,0,0,2,0,0,2,0],
    [0,2,0,0,0,0,0,0,2,0,0,2,0,0,2,0],
    [0,4,4,4,4,4,4,5,3,3,3,3,3,3,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
];
```

プログラムの確認

通路を定義したプログラムを確認します。次のHTMLを開いて動作を確認しましょう。

動作の確認 towet0603.html

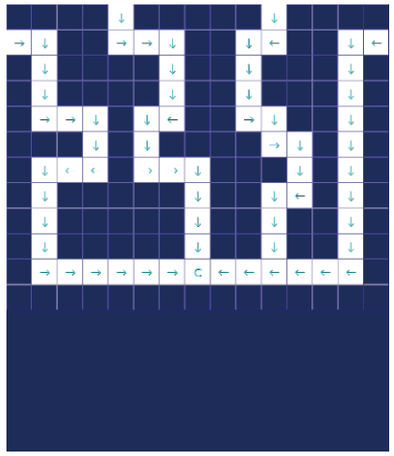


図6-3-2 実行画面

※このプログラムには画像を用いていません。

ソースコード tower0603.js

```
//起動時の処理
function setup() {
    canvasSize(1080, 1264);
}

//メインループ
function mainloop() {
    drawBG();
}

var SIZE = 72;
var stage = [//通路のデータ
    [0,0,0,0,2,0,0,0,0,0,2,0,0,0,0],
    [4,2,0,0,4,4,2,0,0,2,3,0,0,2,3],
    ≀ 省略
    [0,4,4,4,4,4,4,5,3,3,3,3,3,3,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
];
var arrow = ["", "↑", "↓", "←", "→", "C"];

function drawBG() {
    fill("navy");
    lineW(1);
    for(var y=0; y<12; y++) {
        for(var x=0; x<15; x++) {
```

つづく

起動時に実行される関数
キャンバスサイズの指定

メイン処理を行う関数
ゲーム画面を描く関数を
呼び出す

1マスのサイズ(ドット数)を定義

通路データを二次元配列で定義

通路の向きを表示するために定義

ゲーム画面を描く関数

二重ループの繰り返しで
マス目を描く

ア－ア

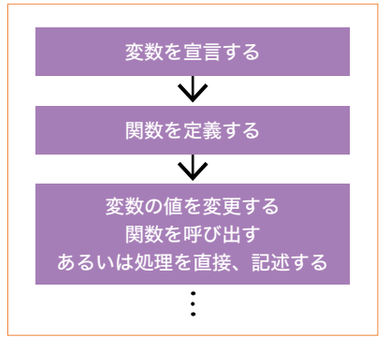
キャラクターを管理するクラスの定義

ここではキャラクターのクラスを定義し、パーティメンバーのパラメーターと、戦闘シーンに登場する敵のパラメーターを管理できるようにします。

クラスとインスタンスについて

クラスとはどのようなものかを説明します。クラスを理解するには、コンピュータのプログラムには**手続き型**と**オブジェクト指向**の２つの書き方があることを知る必要があります。JavaScript、C++、Java、Pythonなどの広く使われているプログラミング言語は、手続き型とオブジェクト指向の両方の書き方でプログラムを作ることができます。

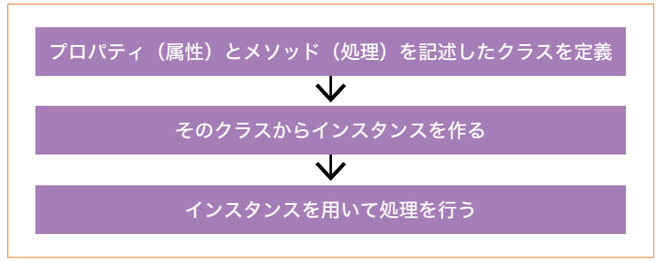
■手続き型プログラミングのイメージ



みなさんが本書で学んできたプログラムは、手続き型で書かれています。これに対し、オブジェクト指向では、**データ**（プロパティで扱う数値や文字列など）と**機能**（メソッドで定義した処理のこと）をひとまとめにした**クラス**を定義し、そのクラスから**インスタンス**（実体）を作ります。そして複数のインスタンスが関わり合ってシステム全体を動かす、という考え方を元にプログラムを組んでいきます。

■**オブジェクト指向プログラミングのイメージ**

※プロパティとメソッドについてはこの先で説明します。



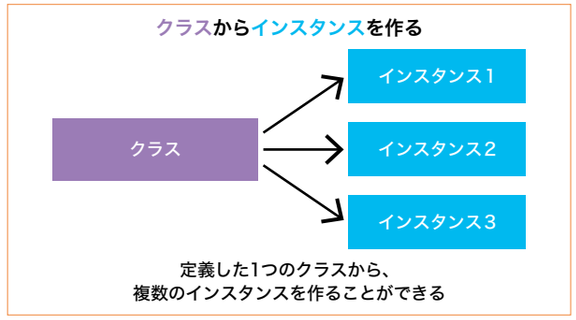


図7-7-1　クラスとインスタンスの関係

クラスから作り出すインスタンスを、そのクラスのオブジェクトと表現することもあります。本書では、キャラクターの設計図をクラス、そのクラスから作り出したゲームに登場するパーティメンバーや敵クリーチャーをインスタンスと（実体）と呼んで説明します。

オブジェクト指向プログラミングは一般的に難しく、プログラミング初心者向けではありません。これまで学んできた通り、本格的なゲームも手続き型プログラムで作ることができます。そこで、本書で制作するRPGはキャラクターの管理だけをクラスとインスタンスで行い、ゲームの処理は手続き型で記述します。

オブジェクト指向を取り入れる理由は、キャラクターのクラスを定義することで、パーティメンバーと戦闘シーンに登場する敵クリーチャーのパラメーター（体力や攻撃力など）をまとめて管理できるからです。キャラクター用のクラスを用意すると数値のやり取りなどを簡潔に行うことができます。

オブジェクト指向プログラミングは難しいと伝えましたが、キャラクター管理に用いるだけなら、それほど難しくはありません。手続き型で色々な配列を用意してパラメーターを管理するより、クラスを定義することでプログラムをすっきりと記述できます。

次の図はキャラクターのクラスと、ゲーム内に実際に登場するキャラクター（インスタンス）のイメージです。

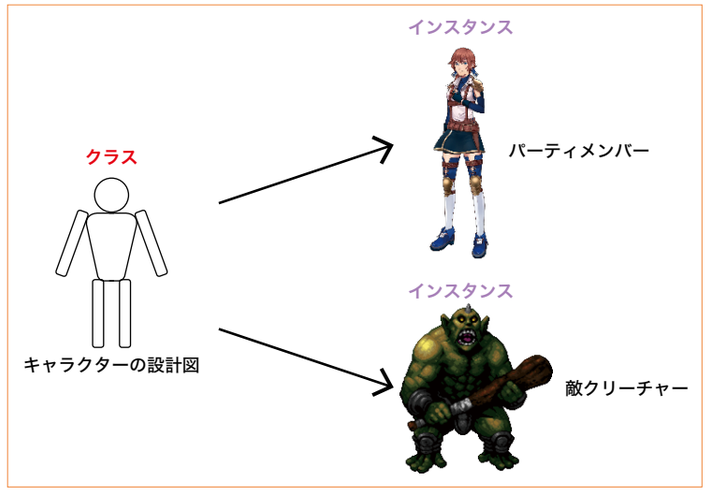


図7-7-2　キャラクタークラスとインスタンスのイメージ

8-4 ターン制を実装する

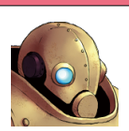
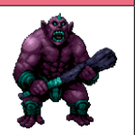
このゲームの戦闘はパーティメンバーと敵が交互に行動するターン制で行われるようにします。ここではターン制の骨組みとなる処理を組み込みます。

ターン制について

ターン制のRPGでは、素早さのパラメーターが大きいキャラクターから順に行動します。それを行うには、数値を大きな順、あるいは小さな順に並べ替える**ソート**というアルゴリズムを理解する必要があります。まずソートについて説明します。

パーティメンバー3人と敵3体が登場し、各キャラクターの素早さは次の値であるとします。この値は説明用のもので、ゲームに登場する実際のキャラクターの値とは違います。

表8-4-1　素早さの値

素早さ	80	60	120	40	70	110
キャラクター						

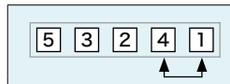
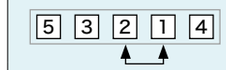
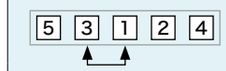
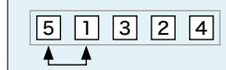
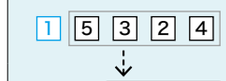
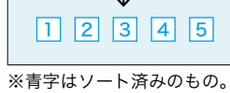
素早さの値が大きなキャラクターから行動する場合、順番は次の通りです。

この順番を決めるには、素早さ順にキャラクターを並べ替える必要があります。それがソート処理です。ソート処理には様々なアルゴリズムがありますが、本書では単純なプログラムで並べ替えできる**バブルソート**を用います。

バブルソートのアルゴリズム

バブルソートで小さな値から大きな値の順に並べ替えるには、データを右から左に向かって比較していき、小さい値を左に移していきます。データを左から右に向かって比較し、大きな値を右に移してもかまいません。バブルソートの流れを図解します。

図8-4-1　バブルソートの流れ

	データ右端の左右に並んだ値（ここでは4と1）を比較し、**左>右**なら入れ替えます。
	1つ左の位置の左右（この例では2と1）を比べ、**左>右**なら入れ替えます。
	比較と入れ替えを左に向かって進めていきます。
	左端までこれを行うと、最も小さな値が先頭に来ます。
	次は先頭を除いた枠内の範囲で比較と入れ替えを行います。 すると二番目に小さな値が左に移動します。
	残りの範囲も同じように比較と入れ替えを行います。
	最後は右端の2つの値の比較と入れ替えを行います。
	ソートが完了し、データが小さなものから大きなものの順に並びます。

※青字はソート済みのもの。

図8-4-1は小さな値から大きな値の順（**昇順**）に並べ替えるバブルソートの基本的な手順です。

RPG「ラストフロンティアの少女」は、素早さの値が大きなキャラクターから行動するので、大きな値から小さな値の順（**降順**）に並べ替えます。降順に並べるには、左から右に向かって比較し、大きな値を左に移す（＝小さな値を右に移す）プログラムがわかりやすいので、これから確認するrpg0804.jsはそのようにしてソートを行っています。

プログラムと動作の確認

次のHTMLを開き、探索シーンで戦闘に入ると、パーティメンバーと敵クリーチャーが交互に行動する様子がメッセージで表示されます。DOME Zeroに出現する「キラーワスプ」と「スパイダー」は、どのパーティメンバーよりも素早さが低いので、パーティメンバーが先に行動します。DOME XXXの敵は全てパーティメンバーより素早いので、敵の方が先に行動します。

戦闘中に「撤退」を選ぶと戦闘を中断します。3つのドームを行き来して動作を確認しましょう。