初心者がモチベーション上げながらプログラミングをしてシューティング（っぽい）ゲームを 1 本作る！

Alisue (http://hashnote.net/)

2013/12/25

はじめに

この記事は Kawaz Advent Calendar 2014 のクリスマスの記事として書かれました。この記事は「初心者がモチベーション上げながらプログラミングをしてシューティング（っぽい）ゲームを 1 本作る！」という目標の元書かれています。この目的のために、以下のような方針を打ち立てました。

1. インストールが難しい言語はご法度（例 C/C++ や Java など）
2. 可能な限りプログラミング以外の部分のイザコザをなくす（Python の文字コード論外）
3. Windows および Mac で共に同じように動作すること（.net framework vs Mono 論外）
4. 「仕組み」の理解に重点を置くためフレームワークなどは一切使用しない（ああいうのは内部を知っている人が使うべきものです）
5. つまづきやすいオブジェクト指向は使わない（つまづいてしまったら意味がないので）
6. 嘘でもいいのでなるべくわかりやすく説明する（レベルが上がれば自ずと間違いに気づくでしょう）

したがって既にゲームをつくったことがある方や、オブジェクト指向バリバリな方などには全く参考になりません。 そのあたりはご了承ください。

どこでもなるべく同じように動き、インストールなどが簡単な言語として今回は JavaScript を選ぶことにしました。まぁこの時点でご存じの方は冷や汗ものだと思いますが、いろいろと制限をすれば JavaScript は初心者でも扱える言語です（だと信じたい）。

では「初心者がモチベーション上げながらプログラミングをしてシューティング（っぽい）ゲームを 1 本作る！」を開始しましょう。まずは準備からです。

ブラウザの準備

すでに Google Chrome の最新版を使用している方は読み飛ばして ください。それ以外の方で Internet Explorer を使用している人以外は**最新版の Google Chrome を インストール**してください。

もしも**普段から Internet Explorer を使用している人**がいた場合は下記のステップを実行したあと 最新版の Google Chrome をインストールしてください。

1. ホームセンターに行き 1) 着火剤 2) 炭 3) 机に使用出来るくらいのサイズの鉄板 を購入。またスーパーに行き豚肉 100 g を購入
2. 購入した鉄板が乗る程度に炭を広げ、着火剤を用いて全炭が最高温度になる位までしっかりと火をつける（なお、この作業は外で行なってください）
3. 鉄板を炭の上に載せ、しっかりと温める。この際購入した豚肉がちゃんと焼けるかにより温度を確認してください
4. 鉄板の上で 10 秒以上土下座を行ってください

参考

- 焼き土下座
- [IE よ滅べ！の意味がわからない方] いざ、爆速と美麗の WWW へ！IE がダメな理由教えます

とりあえず **Google Chrome が最低の条件なのでこいつをインストールしてください。**

テキストエディタの準備

普段プログラミングなどを行わない方にとって、文字を書くソフトと言えばワードかと思います。しかしワードはプログラミングをするという目的においてカスなので、プログラミングに向いた軽量のテキストエディタを準備してください。以下お勧めです。

- **Notepad++**
  Windows を使用している人向け。おそらく現存するメモ帳系アプリにおいて最強です。軽量でかつ高機能なので Windows の方はこれを使えばよいでしょう
- **CotEditor**
  Mac OS X を使用している人向け。僕が知っている限りでは、このアプリが軽量かつ高機能なので一択です。Mac はデフォルトのメモ帳系アプリが本当にクズなので、使わないと思ってもインストールはしておいて欲しい一品

- **GVim/MacVim**
  変態が使用するエディタです。変態になりたい場合は使ってください。Mac を購入したら絶対に導入したい！私が 3 年間で厳選した超オススメアプリ 10 選！
- **Emacs**
  変態が使用するエディタ 2 です。 変態になりたい場合は使ってください。Mac を購入したら絶対に導入したい！超オススメアプリ 10 選！

参考

- エディタ戦争
- 2011 年テキストエディタ界の動向まとめと、来年次の vim エディタ普及に向けた対策資料

プログラミング前準備

この章では円滑にプログラミングをすすめるための前準備を行います。特に初心者のかたはフォルダ構造やファイルの位置、名前などに注意して読んでください。

骨組みの準備

JavaScript を用いてゲームを作成するため、まず骨組みとなる HTML ファイルおよび 骨組みとなるフォルダ構造を作成します。 なお HTML ファイルに関して詳しくは解説しませんので、興味がある方は下記リンクを参照してください。

- HTML5.jp
- HTML5 リファレンス
- Google HTML/CSS Style Guide
- 「Google HTML/CSS Style Guide」を適当に和訳してみた

では早速作成しましょう。なお本記事ではフォルダ構造は下記のようになっていると仮定しています

```
KawazAdventCalendar
    |
    +- src
        |
        +- index.html
        +- js
        |   |
        |   +- shooting.js
```

```
|
+- img
    |
    +- player.png
    +- enemy.png
    +- player_bullet.png
    +- enemy_bullet.png
```

まず上記に従い js フォルダおよび img フォルダを作成してください。その後下記に上げる画像ファイルをダウンロードし、保存してください。

- player.png : 
- enemy.png : 
- player_bullet.png : 
- enemy_bullet.png : 

次に、index.html の内容を下記に記載するので、これを参考に作成してください。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
```

最後にプログラムをメインに記述する shooting.js を js フォルダの中に下記の内容で作成してください。

```
"use strict"
```

とりあえず骨組みはこれで完成です。

画像ファイルの読み込み

通常はプログラム中で動的に画像ファイルを読み込みますが、今回は初心者向けということで **HTML の機能を利用して画像を読み込みます。**

HTML の機能を利用すると言いましたが、単純に<img>タグを使用するだけです。下記のように index.html を変更しましょう。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
<img src="img/player.png">
```

この状態で index.html を Google Chrome で開いてみましょう。どうでしょうか？左上に緑色の「バグ」が見えるでしょうか？今回はこの「バグ」を操作するシューティングゲームを作成します。

では、同様にして他の画像もすべて読み込みましょう。最終的な index.html は以下のようになります。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
<img src="img/player.png">
<img src="img/enemy.png">
<img src="img/player_bullet.png">
<img src="img/enemy_bullet.png">
```

キャンバスの作成

ゲームというのは実際のところ、動的なお絵かきに他なりません。ユーザーのキー入力によって主人公が移動し、主人公が移動したことはユーザーに対して「絵」という形で表現されます。

結局なにが言いたいかというと、プログラミングによってお絵かきが出来る「場所」さえあればそこにゲームを作ることが出来るという事です。まぁ今は何を言っているかわからないかもしれませんが、そのうち解ると思います。

さて、JavaScript で絵を描くには Canvas（キャンバス）タグというものを使用します。このキャンバスタグは HTML5 という新しい規格から表舞台に出てきた新機能で、ホームページ上に動的にお絵かきができます。 別に HTML の歴史を語るために筆を持ったわけでは無いので割愛しますが、今回はこの新機能を少しだけ使って初心者向けのシューティングゲーム作成を行なっていきます。

さて、うだうだと御託を述べるのはこの辺にして、早速このキャンバスタグを先程から編集している index.html に追加しましょう。以下のように index.html に修正を加えてください。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
<img src="img/player.png">
<img src="img/enemy.png">
<img src="img/player_bullet.png">
<img src="img/enemy_bullet.png">
<canvas width='240' height='320'></canvas>
```

<canvas width='240' height='320'></canvas>が追加した部分になります。追加部分において width='240' height='320' と書いて有りますが、この部分でキャンバスの横幅（width）と縦幅（height）をピクセル数で設定しています。

キャンバスのデザイン設定

さて、キャンバスの作成は終了しましたが今のままではどこがキャンバスなのかさっぱりわかりません。仕方がないので CSS という「装飾を施すための言語」を用いてキャンバスと背景のデザインを簡単に行います。 なお CSS に関しても本記事では取り扱わないので、興味のある方は下記リンクなどを参照してください。

- CSS3 リファレンス
- 一夜漬けで CSS3 をマスターするために見ておくべきコードのまとめ

では早速下記のように index.html を編集してください。なおコード中の# ... とはその部分が略されていることを示します。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
<style>
  body {
    background: #eee;
  }
  canvas {
    background: #000;
  }
</style>
<img src="img/player.png">
// ...
```

これで index.html を開くと少し大きめな黒い部分が表示されるようになりました。ただ、左上に鎮座しておりあまり美しくありません。 せっかくなら CSS を使用して画面の左右中央に表示するようにしましょう。以下のように再度修正を加えてください。

```
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Kawaz Advent Calendar 2014-12-25</title>
<style>
  body {
    background: #eee;
  }
  canvas {
    background: #000;
    display: block;
    margin: auto;
  }
```

```
</style>
<img src="img/player.png">
// ...
```

これで画面の左右方向中央にキャンバスが表示されるようになりました。なお上下方向中央に表示するには少し高度な CSS が必要になるためここでは割愛します。

名前を与える

ほぼ骨組みは完成しているのですが、このままではプログラムから使用することができません。プログラムからこれらの要素を使用するためには「名前」が必要になります。まぁ現実世界でもそうですよね？名前が無いものを呼んだりできませんから。

HTML で名前を与えるには `id="名前"` という記載をタグの中にしてやります。なおひとつ重要な注意ですが **id で与えられた名前の（1 ファイル内での）重複は許されていません。** 稀に HTML などを解説しているサイトでこの id を重複させているところがありますが、 間違いなので気をつけてください。

では、下記を参考にプログラムからアクセスする画像・キャンバス要素に名前を付けてください。

```
<!DOCTYPE html>
// ...
<img id="player" src="img/player.png">
<img id="enemy" src="img/enemy.png">
<img id="player_bullet" src="img/player_bullet.png">
<img id="enemy_bullet" src="img/enemy_bullet.png">
<canvas id="screen" width='240' height='320'></canvas>
```

ちなみに気づいた方も多いと思いますが `id` の指定と `width` の指定で引用符が異なります。ただこれは僕個人の趣味なので `id='名前'` としたり `width="240"` としても問題ありません（僕は文字列を"で、数値を'で囲むようにしてます。結構忘れますが）。ただ'で初めて"で終わるような使い方はできないので注意してください。初心者のうちは混乱するかもしれませんが、**基本的に'と"は同じように「値を囲む」時に使用します。**

JavaScript ファイルのロード

さて、次の章から実際にプログラミングを行い始めます。このプログラミングとはプログラムを書く作業のことを指します。したがって「プログラムを書く場所」が必要となります。

JavaScript はこの「プログラムを書く場所」というのが 2 種類あります。HTML の「中」か「外」です。気楽に書くには HTML の「中」が便利なのですが、今回は

ややこしくなるので「外」のみを使用します。もし何か他の入門書などで HTML の「中」に書いているコードがあった場合はそちらに従ってください。

さて、ではこの「外」とは一体どこなのか？それはもうお気づきかとは思いますが、すでに作成した `shooting.js` です。この記事ではこの `shooting.js` に JavaScript を書いて、この `shooting.js` を `index.html` で読み込むことで実行させます。

はい、なので下記のように `shooting.js` を読み込んでください。**なお、どこに `<script>` タグを置くかというのは非常に重要なので必ず位置を守ってください。**

```
<!DOCTYPE html>
// ...
<img id="player" src="img/player.png">
<img id="enemy" src="img/enemy.png">
<img id="player_bullet" src="img/player_bullet.png">
<img id="enemy_bullet" src="img/enemy_bullet.png">
<canvas id="screen" width='240' height='320'></canvas>
<script src="js/shooting.js"></script>
```

これでプログラミングを行う準備はすべて整いました。次の章から実際にプログラミングを始めましょう。

プログラミング事始め

この章ではとりあえずプログラミングを行なってみます。習うより慣れろ、この章は糞つまらないので多少わからないことがあっても立ち止まらずに進んでみてください。

Hello World

なぜかプログラミングの参考書などを見ると必ずこの Hello World が一番最初にあります。 まるで宗教のように存在する Hello World、文字を表示したってなんの面白みもありません。

さて、ここまでディスった癖に僕の記事でも Hello World を表示させます。理由は簡単。文字列の出力が出来ると「デバッグ情報」が出せるようになるからです。なんのことかはそのうち解ると思います（たぶんこの辺がどの参考書でも Hello World を最初に取り上げる理由）。

では早速 `shooting.js` を開いて下記のように修正してください（`index.html` じゃ ないので注意！）。

```
"use strict"

// デベロッパーツールにログとして表示
console.log("Hello World");
```

保存後に `index.html` を再表示（ `F5` か `Command + R` ）してください。 Hello World と表示されれば成功です。

え？表示されない？

おそらく「デベロッパーツール」が表示されていないためです。とりあえず気にせず読み進めましょう。

さて、今回のコードはすごーく短いですが、結構重要な要素を含んでいます。なので各行それぞれ解説します。

```
"use strict"
```

これを理解してもらうのは多分不可能なので**「おまじない」**だと思ってください。**JavaScript を書く場合はファイルの先頭に必ず書くようにしてください**。それを守るだけでプログラミングが上手になります。

```
// デベロッパーツールにログとして表示
```

日本語が書いて有りますね。日本語の意味も結構重要なのですが、ここで言いたいのはこの行が**コメント**であるということです。

コメントとは「プログラムの実行に直接関係がなく、実行時に無視される文字列」のことです。まぁ簡単に言うと**「コメントには基本的に何を書いても良い」**ということです（基本的と断ったのは環境依存文字を使った絵文字入りのコメントのせいでプロジェクトがコンパイルできなくなった事例などがあるため w）。

本記事ではこのコメントにいろいろな使い方や説明を記載していくので、見た瞬間に「あ、これはコメントだ」と分かるようになってください。

では、どのようなものがコメントなのかを説明します。まぁ説明ってほどじゃないですけど、JavaScript では以下のような部分がコメントになります。

1. `//` 以降、改行まで
2. `/*` と `*/` で囲まれている部分前部

個人的には//を使用したコメントが好きなので、この記事ではそれだけ覚えていれば大丈夫です。

```
console.log("Hello World");
```

ここが実際のコードになります。 **コンソール**というところに「Hello World」と出力するコードです。

で、このコンソールってのはどこかというと「デベロッパーツール」という場所にあります。**この「デベロッパーツール」ですが、普段は見えないので出せるようになってください**。Windows の場合は `F12` キーを、Mac の場合は `Command +`

Option + I を押してください。なお、このデベロッパーツールは詳しく説明すると記事が一本かけるくらい多機能なので解説は割愛します。詳しく知りたい方は下記リンクをご覧ください。

- Web 開発でよく使う、特に使える Chrome デベロッパー・ツールの機能
- 「Google Chrome Deveoper Tools」はこう使う！基本機能チュートリアル & GDD 2011 セッション概要
- （非開発者でも使える）「Chrome Developer Tools」を使ってみよう！

さて、「デベロッパーツール」の出し方もわかったので先ほどと同様に再表示をさせたあとこの「デベロッパーツール」を出してみてください。 下記参考画像のように Hello World と表示されていれば成功です。

Hello World 変数編

どんどん行きましょう。次はどうしても最初に解説しなくてはいけない「変数」について説明します。

「変数」というのは簡単に言うと「箱」です。プログラミングをしているといろいろな値が出てきます。すごく長い計算を使って出した数字やユーザーから入力してもらったチャットの内容などです。**変数はこれらの値を格納するための箱だと思っていただければだいたい正解です**。ただ、このように説明されてもわかりにくいと思うので以下の様に shooting.js を 書き換えて、実行（ index.html を再表示）してみてください。

```
"use strict"
// 変数 message を定義
var message;
// 変数 message に「文字列」を代入
message = "Hello World";

// デベロッパーツールにログとして表示
console.log(message);

// 変数 number を定義して「文字列」を代入
var message2 = "ハローワールド";

// デベロッパーツールにログとして表示
console.log(message2);
```

これを実行するとコンソールに下記のように表示されるはずです。

```
Hello World
ハローワールド
```

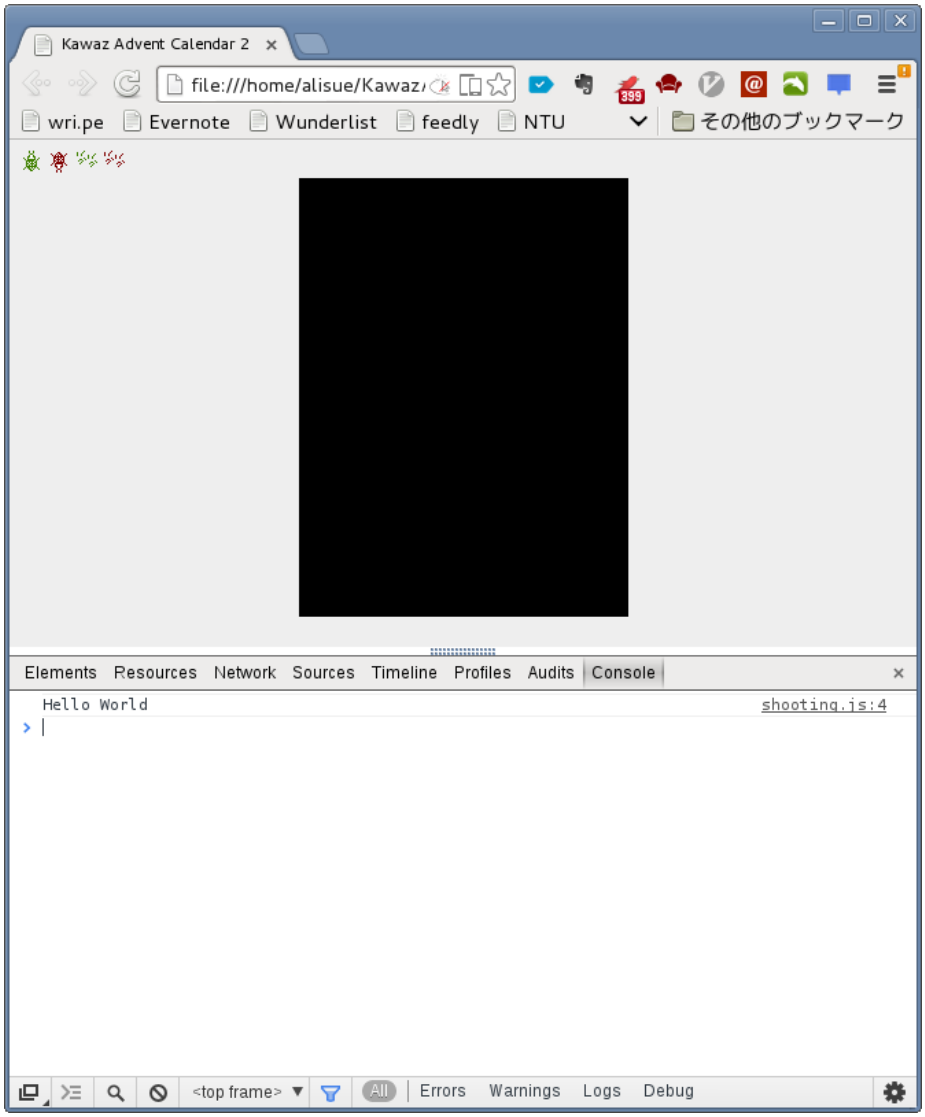

Figure 1: Hello World

変数の定義は var から始まる

変数というのは `var 変数名;` とすると**定義**することができます。変数という箱はどこにでも存在しているわけではなく、このように**定義**をしないと使うことができません。現実世界でも「箱」ってのは買うなり作るなりしないと存在しませんよね？もしも「俺は思っただけで箱が現れるけど、なにか？」というならば、こんな記事を読んでないで僕にその超能力の研究をさせてください。

さて、変数は定義しただけでは空っぽです。この空っぽの変数の中に値を入れることを**代入する**といいます。なので上記の `message = "Hello World";` という部分は「`message` という変数の中に `"Hello World"` という文字列を入れてね」という意味です。

変数はこのように**定義**して**代入**ということをよく行うので、もっと短く書く方法があります。 その方法が `var message2 = "ハローワールド";` の部分で、このように**定義**して**代入**というのが一行で書けます。単純に定義して代入しているだけなので、この先こういう書き方を見ても混乱しないでくださいね。

文字列と数値と真偽値

最後にどうしても説明しなきゃいけない**文字列**と**数値**と**真偽値**を説明します。

すでに変数を学んだ皆さんは、実は小学校の文章題くらいならプログラミングで解けるようになっています。 例を見ましょう。

> 【算数】たかしくんは 1 個 60 円のりんごを握りしめ、滅びゆく世界の中、決意の眼差しで空を仰ぎ、静かに呟きました。「例え明日世界が滅亡しようとも、今日僕はりんごの木を植えるよ」常に希望を捨てず、最後まで生き抜こうとするたかし君の目にはどんな未来が映っているでしょうか。
> (出典: http://matome.naver.jp/odai/2134324699886988501)

ではこの問題をプログラミングで解いてみます。この問題は文章題なので、よりわかりやすく整理すると

1. たかしくんの**所持金**は **500 円**
2. 1 個 **60 円**の**りんご**が **3 つ**
3. 1 個 **30 円**の**みかん**が **4 つ**

です。太字にしたところがまさにプログラミングで使う「名前」や「値」です。ではたかしくんの残金を計算してみましょう。 `shooting.js` を以下のように修正してください。

```
"use strict"
// --- 必要な変数をすべて定義（複数定義する場合は , で区切る）
```

```
// pocket -- 所持金
var pocket;
// apple_price -- りんごの値段
// apple_count -- りんごの個数
var apple_price, apple_count;
// orange_price -- みかんの値段
// orange_count -- みかんの個数
var orange_price, orange_count;

// --- 文章題で与えられた数値を代入

// 所持金は 500 円
pocket = 500;

// りんごの値段は 60 円
apple_price = 60;
// りんごの数は 3 つ
apple_count = 3;

// みかんの値段は 30 円
orange_price = 30;
// みかんの数は 4 つ
orange_count = 4;

// 残高を求める
//
// balance -- 残高
// 残高 = 所持金 - りんごの値段 x りんごの個数 - みかんの値段 x みかんの個数
//
// 注意: コンピュータでは掛け算は * を使う
//
var balance = pocket - apple_price * apple_count - orange_price * orange_count;

// 残高を表示
console.log("たかしくんの残金は" + balance + "円");
```

これを実行すると「たかしくんの残金は 200 円」と表示されると思います。ちゃんと計算されてますね。

さて、鋭い方はもうお気づきかと思いますが**変数に値を代入する部分で" や ' などの引用符が使われていません**。実は**引用符**で囲う値は**文字列**だけです。正確には**引用符で囲うと文字列になります**。

コンピューターはあまりにも正確すぎるので僕ら人間のような「推測」ができません。これとは対照的に、人間は「推測」が得意です。 ほら、小学校の頃に「1 + 1 =」とか流行りませんでしたか？僕の頃は文脈や状況によって「田んぼの田」

と答えたり、「2」と答えたり、柔軟な対応ができて初めて仲間として認められたものでした。しかしコンピュータは推測ができないので「1 + 1 =」と聞かれると必ず「2」と答え、小学校 4 年生くらいの男の子に馬鹿にされます。

たぶん、いくら戯言を述べても混乱するだけなので例を見せましょう。各コードの出力結果はコード下にコメントで示しました。

```
console(1 + 1);
// 出力: 2

console('Hello' + 'World');
// 出力: HelloWorld

console('1' + '1');
// 出力: 11
```

この例からも分かるように**文字列の連結は + で行います**。また当たり前ですが**数値の足し算も + で行います**。人間なら「推測」によって文字列の連結をすべきなのか、足し算をすべきなのかだいたいわかると思いますが、コンピューターにはそれができないため `console('1' + '1');` の結果が `11` になります。

ちなみに D 言語（Dark Language）と呼ばれる暗黒物質理論と量子力学、特殊相対性理論という現代物理学の最先端理論を元に研究開発された 新しいプログラミング言語は、文字列の連結に ~ を使うという誰も思いつかなかった素晴らしい方法でこの問題を解決しています（98%嘘です）。

話がそれましたが、このような「値の種類」というのは型と呼ばれ、他にも様々なものがあります。その中に、「イエス」か「ノー」の二択を表す型があり、その値は**真偽値**と呼ばれます。

（出典: http://www.personalised-gifts-engraved.com/acatalog/4_Angel_Devil_Pillow_Case.html）

プログラミングをしているとよく「イエス」か「ノー」の二択に迫られます。例えば、僕の夜の行動を「プログラミング言語なでしこ」 的な仮想言語で表現するならば下記のようになります。

```
もし、夕飯ができている ならば
    もし、お寿司 ならば
        「いまから帰る」と表示。
    もし、ビール ならば
        「いまから帰る」と表示。
    もし、冷えたご飯 ならば
        「もう食べた」と表示。
違えば
    「食べて帰る」と表示。
```

ここで示した条件はすべて「イエス」と「ノォー」で表される条件です。このように「イエス」か「ノー」と言う二択の値はよく使うため `true`（真） と `false`

Figure 2: 真偽値参考画像

（偽）というように表されます。ここではこれ以上解説しないですが、頭の片隅にでもおいておいてください。

この書いてる方も読んでる方も眠くなる章はこれで終わりです。次の章からはもっと楽しいことをしましょう。

キャラクターを表示する

文字列の表示なんて面白味のないことはやめて画像を表示しましょう。この章では「画像の表示」以外にも「ランダム処理」や「ループ処理」などゲームを作る上で欠かせない技術について説明します。

キャンバスに描画する

前準備でもさらっと説明したように、この記事では JavaScript でキャンバスに絵を書くことでゲームを作ります。この**キャンバス**というのは以下の 3 ステップを踏むことで使えるようになります。

1. キャンバスオブジェクトを取得
2. コンテキスト（とよばれるナニか）をキャンバスオブジェクトから作成
3. コンテキストの中に入っている道具（メソッド）を使ってお絵かき

詳しい説明は割愛するとして、実際に何か描画してみましょう。 下記のように `shooting.js` を書き換えてください。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;

// ページロード時に呼び出される処理を指定
// window.onload = function(){ から }; までの間が呼び出される。
window.onload = function() {
    // id を用いてキャンバスオブジェクトを取得し
    // canvas 変数に代入
    //
    //   オブジェクト = document.getElementById('id');
    //
    canvas = document.getElementById('screen');

    // 2次元用の描画コンテキスト（とよばれるナニか）を取得し代入
    ctx = canvas.getContext('2d');

    // 塗りつぶしの色を指定（白）
```

```
    ctx.fillStyle = '#fff';
    // 塗りつぶされた四角形（横, 縦 = 20, 30）を（8, 5）の位置に描画
    ctx.fillRect(8, 5, 20, 30);

    // 線の色を指定（赤）
    ctx.strokeStyle = '#f00';
    // からっぽの四角形（横, 縦 = 90, 10）を（40, 55）の位置に描画
    ctx.strokeRect(40, 55, 90, 10);
};
```

これで塗りつぶされた白色の四角形と赤い四角の枠が描画されていれば成功です。なお今回のコードに関しては詳しく説明する必要がないため、説明は割愛します。読み進めることを優先してください。

キャンバスに画像を描画する

四角形を描画してもなにも面白くないので、とっとと画像を描画しましょう。画像を描画するにはコンテキスト（ctx）の drawImage(img, x, y) というメソッド を利用します。下記コードを参考に shooting.js を書き換えてください。なお理解不要のお決まり部分のコメントは削除しました。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;
// プレイヤーの画像を保持する変数を定義
var img_player;

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // コンテキストを取得（おまじない）
    canvas = document.getElementById('screen');
    ctx = canvas.getContext('2d');

    // Playerの画像（id='player'で指定された<img>）を取得
    img_player = document.getElementById('player');

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
};
```

これでキャンバス（黒い部分）の中に緑色の「バグ」が表示されれば成功です。

さて、先程から何の気なしに

> Player の画像を (20, 50) の位置に描画

のように（数字，数字）という書き方をしてきましたが、これは**座標**を表します。**座標**と聞くと死に急ぎ野郎が出てくる昨今ですが、元々は**位置**を明確に表すためのものです。中学校で習っているはずなので、あんまり詳しくは解説しませんがコンピューターの場合原点座標 (0, 0) は左上に設定されることが多いです。 今回の場合も同様で (10, 20) と (20, 30) では前者のほうが左上よりです。座標がどのように描画位置を変えているかは実際に値を変化させて確かめてみてください。

キャンバスに画像をたくさん描画する

さて、せっかくプレイヤーを表示したので敵キャラも表示しましょう。以下のように `shooting.js` を改変してください。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;
// プレイヤーの画像を保持する変数を定義
var img_player;
// 敵キャラの画像を保持する変数を定義
var img_enemy;

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // コンテキストを取得（おまじない）
    canvas = document.getElementById('screen');
    ctx = canvas.getContext('2d');

    // Playerの画像（id='player'で指定された<img>）を取得
    img_player = document.getElementById('player');
    // 敵キャラの画像（id='enemy'で指定された<img>）を取得
    img_enemy = document.getElementById('enemy');

    // Playerの画像を（20, 50）の位置に描画
    ctx.drawImage(img_player, 20, 50);
    // 敵キャラの画像を（30, 60）の位置に描画
    ctx.drawImage(img_enemy, 30, 60);
};
```

これを実行すると、プレイヤー（緑バグ）のすぐ隣に寄り添うよう敵キャラ（赤バグ）が表示されると思います。

さて、今回作るのはシューティングゲームなので敵キャラはたくさん出てくるはずです。なのでとりあえず 15 匹くらい表示しましょう。以下のように `shooting.js` を書き換えてください。 なおコード中に`// ...` で示された部分は省略を表しています。その部分のコードは消さないで残しておいてくださいね。

```
// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
    // 敵キャラの画像をいろんなところに描画
    ctx.drawImage(img_enemy, 60, 20);
    ctx.drawImage(img_enemy, 60, 30);
    ctx.drawImage(img_enemy, 60, 40);
    ctx.drawImage(img_enemy, 60, 50);
    ctx.drawImage(img_enemy, 60, 90);
    ctx.drawImage(img_enemy, 90, 20);
    ctx.drawImage(img_enemy, 90, 30);
    ctx.drawImage(img_enemy, 90, 40);
    ctx.drawImage(img_enemy, 90, 50);
    ctx.drawImage(img_enemy, 90, 90);
    ctx.drawImage(img_enemy, 120, 20);
    ctx.drawImage(img_enemy, 120, 30);
    ctx.drawImage(img_enemy, 120, 40);
    ctx.drawImage(img_enemy, 120, 50);
    ctx.drawImage(img_enemy, 120, 90);
};
```

どうでしょうか？なんだかすごく恣意的に整列してますね。

敵キャラの位置をランダムに決定する

やはりせっかくプログラミングをしているので敵キャラの位置は**ランダム**に決めたいですよね？ ランダムな位置を決めるにはランダムな値が必要です。JavaScript でランダムな値を取得するには `Math.random()` というものを使用します。 この `Math.random()` は 0 から 1 までの値をランダムで返すので、欲しい値を掛け算してやると好きな大きさのランダムな数字を得ることができます。やってみましょう。 以下の要領で `shooting.js` を修正してください。

```
// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
```

```
    // 敵キャラの画像をランダムな位置に表示（とりあえず一匹）
    ctx.drawImage(img_enemy, Math.random() * 100, Math.random() * 100);
};
```

これを何度か実行してみてください。実行するたびに敵キャラの位置が変わっていれば成功です。

さて、何度か実行していると敵キャラがすごーく狭い範囲でしか動いていないことがわかります。 キャンバスのサイズは 240 x 320 で作成しましたよね？それにも関わらず上記コードでは 100 x 100 の中でランダムな位置を指定しています。じゃぁ `Math.random() * 320` とかやればいいじゃんということになりますが、せっかくなのでキャンバスのサイズをプログラミングで取得しましょう。

キャンバスのサイズは `canvas.width` と `canvas.height` で取得できます。 それぞれ横幅と縦幅ですね。これを踏まえて上記のコードを修正したものが下記になります。 参考にして `shooting.js` を書き換えてください。

```
// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
    // 敵キャラの画像をランダムな位置に表示（とりあえず一匹）
    // なお、下記のように () の中は , の位置で改行しても構わない
    ctx.drawImage(img_enemy,
                  Math.random() * canvas.width,
                  Math.random() * canvas.height);
};
```

さて、これでどうでしょうか？完成でしょうか？

これを何度か実行すると稀に敵キャラが右か下の端っこに隠れてしまうことがあると思います。 まぁこれは簡単な話で `drawImage(img, x, y)` というメソッドは**対象の画像の左上 を (x, y) の位置にあわせて描画する**関数だからです。ランダムに取得した値がキャンバスの幅ギリギリだと、そこから画像の描画を右下に向かって始めるので画像が全部表示されません。じゃぁどうすればいいかというと、画像の横幅をキャンバスの横幅から引いてあげればいいですね。こうすればランダムに取得した値の最大値でも画像はキャンバスの中に収まります。

上記を踏まえて修正したコードが下記です。 このように `shooting.js` を修正してください。

```
// ...
```

```
// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
    // 敵キャラの画像をランダムな位置に表示（とりあえず一匹）
    // なお、下記のように () の中は , の位置で改行しても構わない
    ctx.drawImage(img_enemy,
                  Math.random() * (canvas.width - img_enemy.width),
                  Math.random() * (canvas.height - img_enemy.height));
};
```

何度かリロードして敵キャラが常にキャンバス内に描画されることを確認してください。

ループ処理で馬鹿みたいにたくさん敵を描画する

さて、ランダムに敵の位置を決定できるようになったので大量に敵を表示しなおします。ただ先程のように一つづつ書いていくと、例えば 1000 匹の敵を表示しようとすると骨が折れます。せっかくプログラミングをしているのだから、こういうくだらない処理はコンピュータに任せたいところ。

JavaScript において、このような**繰り返し処理**を行う場合 **for 文**というものを使用します（他に **while 文**などもあるが扱わない）。この **for 文**は以下のような定義になっています。なお、より詳しくは for 文 - JavaScript 入門 あたりを参照すると良いと思います。

```
for(はじめに呼ばれるコード; 終了判定に使われるコード; 各ループ後に呼ばれるコード) {
    ループされる処理
}
```

```
// 主な使い方
// 下記コードはコンソールに 0 から 99 までを表示します
//  | var i=0; で変数 i というのを定義し、0 を入れます
//  | i<100; で i が 100 未満であれば処理を繰り返すという条件を与えます
//  | i++ で i の値を 1 増やします（インクリメント）
for(var i=0; i<100; i++) {
    console.log(i);
}
```

ではこの **for 文**を使って 1000 匹の敵キャラをランダムに描画してみましょう。下記を参照して `shooting.js` を書き換えてください。

```
// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像を (20, 50) の位置に描画
    ctx.drawImage(img_player, 20, 50);
    // 敵キャラの画像をランダムな位置に表示
    for(var i=0; i<1000; i++) {
        ctx.drawImage(img_enemy,
                    Math.random() * (canvas.width - img_enemy.width),
                    Math.random() * (canvas.height - img_enemy.height));
    }
};
```

下記参考画像のように気持ちが悪いくらいたくさんのバグが表示されれば成功です。ただこのままだと気持ち悪いので 10 匹だけ表示するように改良しておいてください。

キーボードでキャラクターを動かす

操作ができないゲームはただの映画です。ただの映画にしてはあまりにも画像のクオリティーが低いので、頑張ってキャラクターを動かせるようにしましょう。この辺から少しむずかしくなりますが、習うより慣れろ精神でどんどん読み進めることをおすすめします。ちなみにこの章で「イベント処理」と「if 文」、「配列」および「関数」を説明します。

イベントを補足する

ユーザーがキーボードを叩いたり、マウスをクリックしたり、幼馴染の女子高生が大学生の彼氏を作ったという話を聞くと**イベント**というものが発生します。したがって、この**イベント**が発生したことを補足できれば様々な局面に対応できます。

まぁとりあえずイベントの説明はこの辺にして、コードを見てみましょう。以下のように `shooting.js` を修正してください。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // キーボードが押されるとこの内部の処理が実行される
    console.log("キーボードが押されたよ");
```

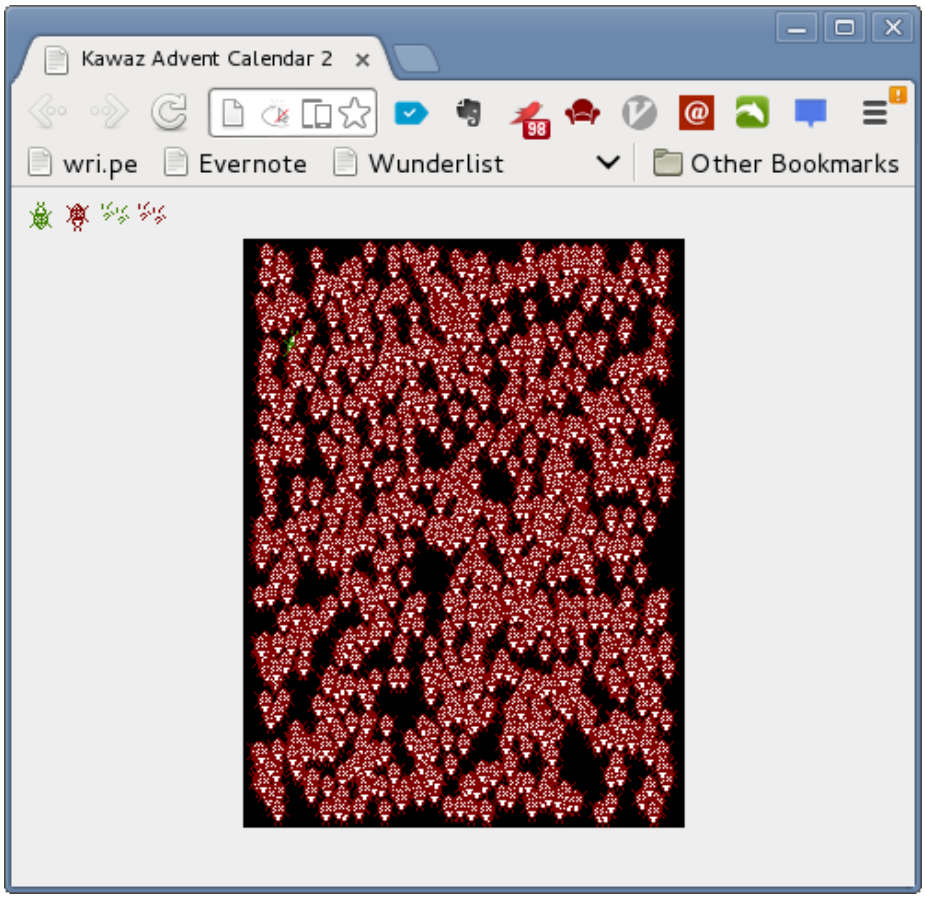

Figure 3: 大量のバグ

```
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...
};
```

これを実行するとキーボード入力時に「キーボードが押されたよ」と表示されると思います。なお、うまく動かない場合は一度マウスでバグをクリックしてください（デベロッパーツールにフォーカスが取られているとうまく動作しないため）。

押されたキーを取得する

キーが押されたことだけがわかってもゲームが成り立つとは思えません。したがって「どのキーが押されたのか？」が分かる必要があります。 これは `e.keyCode` とすると取得できます。下記コードを参照して書き換えてください。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // キーボードが押されるとこの内部の処理が実行される
    console.log(e.keyCode + "番のキーが押されたよ");
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...
};
```

キーを入力した際に「83 番のキーが押されたよ」のように番号が表示されれば成功です。各キーには固有の番号がふられているのでゲームで使用するキー番号はメモしておきましょう。

なお日本語入力が ON になっていると動かないので注意してください。

もしも～なら～する（if 文）

さて、押されたキーを判別することができたので、ナニが押されたか？に対して処理を分けることが出来ればキャラクターを動かせそうなもんです。この「もしも～なら～する」という処理は **if 文**というものを使えば行えます。この **if 文**ですが、構文は下記のようになっています。

```
// 条件が 1 つの場合
if(条件式) {
    条件が正しい場合実行されるコード
}

// 条件が 1 つの場合 2
if(条件式) {
    条件が正しい場合実行されるコード
} else {
    条件が正しく無いときに実行されるコード
}

// 条件が 2 つ以上の場合
if(条件式 1) {
    条件が正しい場合実行されるコード
} else if(条件式 2)  {
    条件式 1 が正しくなく、条件式 2 が正しい場合に実行されるコード
} else {
    すべての条件が正しくない場合に実行されるコード
}
```

さて、ここで**条件式**というのが出てきました。**条件式**とは、なんとも説明がしがたいのですが下記のようなものを言います。

```
A == B // A と B が等しい（=は 2 つなので注意！）
A != B // A と B が等しくない
A >= B // A は B と等しいか大きい
A <= B // A は B と等しいか小さい
A > B  // A は B より大きい
A < B  // A は B より小さい
```

おそらく数学で習ったと思います。 ただ**プログラミングの等しいは== で表されることにむっちゃ注意（= は代入）**。結構この辺のミスが多いと思います。

さて、この **if 文**と**条件式**を駆使してプログラマは処理を分けていきます。では実際に**スペースが押された時だけ「スペースが押されたよ」と表示してみましょう。**以下のように `shooting.js` を書き換えてください。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // スペース（32 番）が押されたか確かめる
    if(e.keyCode == 32) {
        // keyCode が 32 の時だけ実行される部分
```

```
        console.log("スペースキーが押されたよ");
    } else {
        // スペースキー以外の場合は単純に番号を表示
        console.log(e.keyCode + "番のキーが押されたよ");
    }
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...
};
```

スペースキーを押した時と他のキーを押した時で表示される内容が変化していれば成功です。

プレイヤーを動かしてみる

だいたい準備はそろったので、そろそろプレイヤーを動かすことを考えます。

動くってナニ？

まず「動く」ってなんだ？というところをちょっと考えてみましょう。現実世界では「動く」というのは物理的な位置が変わることを指します。ただ、今はコンピュータの画面上に描画されたキャラクターが動くかどうかの議論なのでこれは違います（物理的に動かすってことは画面を持って走るってことですよね？）。

まぁ結論を述べると、コンピューター上に描画されたキャラクターが動くというのは「今現在の描画が消され、少し離れた位置に新しく描画される」ということです。なんだか難しい話に聞こえますが、ようは「パラパラアニメ」です。

要点をまとめると、キャラクターを動かすためには

1. 現在の描画を消す
2. 少し離れた位置に再度描画する

となります。 ではこれを実際に行なってみましょう。

現在位置を定義する

「少し離れた場所」というのを知るためには「今現在どこにいるか？」がわからなくてはなりません。したがって、今現在のプレイヤーの位置を保存しておく変数を定義します。描画はこの変数の値を使用して行い、プレイヤーを移動させる場合はこの変数の値を少し増やせば（減らせば）良いことになります。

ではプレイヤーの位置を保存する変数を定義して、その変数を用いて描画するように書き換えてみましょう。 `shooting.js` を以下のようにしてください。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;
// プレイヤーの画像を保持する変数を定義
var img_player;
// 敵キャラの画像を保持する変数を定義
var img_enemy;
// プレイヤーの現在位置を保持する変数を定義
// player_x -- プレイヤーの x 座標
// player_y -- プレイヤーの y 座標
var player_x, player_y;

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // ...
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Player の初期位置を指定
    // player_x = キャンバスの左右中央
    // player_y = キャンバスの下から 20px 上
    player_x = (canvas.width - player.width) / 2;
    player_y = (canvas.height -player.height) - 20;

    // Player の画像を (player_x, player_y) の位置に描画
    ctx.drawImage(img_player, player_x, player_y);
    // 敵キャラの画像をランダムな位置に表示
    for(var i=0; i<10; i++) {
        ctx.drawImage(img_enemy,
                    Math.random() * (canvas.width - img_enemy.width),
                    Math.random() * (canvas.height - img_enemy.height));
    }
};
```

ついでにプレイヤーの初期位置をシューティングゲームっぽい位置に設定しました。実行した際にキャンバスの左右中央下の方にプレイヤーが表示されていれば成功です。

プレイヤーを動かす

プレイヤーの「現在位置」が定義できたので次は右矢印キーが入力された時にプレイヤーを右に動かすようにしてみましょう。 これはどのように行うかというと

1. キー入力を取得
2. 右矢印キーか判定
3. プレイヤーの現在位置を少し右にずらす
4. キャンバスをクリアする
5. プレイヤーを新しい現在位置に描画する

というステップになります。 では下記を参考に `shooting.js` を書き換えてください。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // 右矢印（39番）が押されたか確かめる
    if(e.keyCode == 39) {
        // プレイヤーのx座標を少し増やす
        //   XXX += 2 という書き方は XXX = XXX + 2 を短くした書き方
        player_x += 2;

        // キャンバスをクリアする
        ctx.clearRect(0, 0, canvas.width, canvas.height);

        // 新しい位置にプレイヤーを描画
        ctx.drawImage(img_player, player_x, player_y);
    }
};

// ...
```

これを実行して右矢印キーを押すと敵キャラがすべて消えて（キャンバスをクリアしたため）、プレイヤーが右に移動します。

同様にして左右に動くようにもしてみましょう（上下左右でもいいのですが、ゲームの特性上左右だけのほうがゲームっぽくなるので今回は左右だけです）。コードはほぼ同じなのでご自身で行なってみてください。下記に各矢印キーの番号を明記します。

- 左矢印キー: 37
- 右矢印キー: 39

念の為僕のコードも貼り付けておきます。うまく動かない場合は参考にしてください。 また、`if` と `else if` の使い分けや、どのようにして同じ処理を繰り返し書かないようにしているか、どうすればコードがわかりやすくなるのか？などに注意してみてください。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // 上下左右の移動速度を定義
    var SPEED = 2;

    // キー番号だとわかりにくいため予め変数に格納
    var RIGHT = 39;
    var LEFT  = 37;

    // 移動処理を行ったかどうか（Yes/No）を表す変数を定義し
    // 移動していない（false）で初期化
    var moved = false;

    if(e.keyCode == RIGHT) {
        // プレイヤーの x 座標を少し増やす
        player_x += SPEED;
        // 移動したので true を代入
        moved = true;
    } else if(e.keyCode == LEFT) {
        // プレイヤーの x 座標を少し減らす
        player_x -= SPEED;
        // 移動したので true を代入
        moved = true;
    }

    // キー入力により移動したか調べる
    // 注意: 真偽値なので moved == true のようにしなくても同じ意味になる
    if(moved) {
        // キャンバスをクリアする
        ctx.clearRect(0, 0, canvas.width, canvas.height);

        // 新しい位置にプレイヤーを描画
        ctx.drawImage(img_player, player_x, player_y);
    }
};

// ...
```

敵キャラを消えないようにする（配列を使用）

プレイヤーは左右に動くようになりましたが、敵キャラが消えてしまいます。もちろん敵キャラも再描画するようにすればいいのですが、素直に下記のようにすると敵キャラの位置が毎回変わってしまいます。

```
// ...

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // ...

    // キー入力により移動したか調べる
    // 注意: 真偽値なので moved == true のようにしなくても同じ意味になる
    if(moved) {
        // キャンバスをクリアする
        ctx.clearRect(0, 0, canvas.width, canvas.height);

        // 新しい位置にプレイヤーを描画
        ctx.drawImage(img_player, player_x, player_y);

        // 敵キャラの画像をランダムな位置に表示
        // これだとプレイヤーが動くたびに敵キャラの位置が変わってしまう
        for(var i=0; i<10; i++) {
            ctx.drawImage(img_enemy,
                          Math.random() * (canvas.width - img_enemy.width),
                          Math.random() * (canvas.height - img_enemy.height));
        }
    }
};

// ...
```

したがって敵キャラの位置もプレイヤーのように変数で保存する必要があるのですが、敵キャラはプレイヤーと違って 10 匹もいます。プレイヤーの場合と同様に変数を定義すると、下記のように非常に面倒くさいことになります。

```
// 敵キャラの現在位置を保持する変数を定義？？？面倒くさ過ぎない？
var enemy_1_x, enemy_1_y;
var enemy_2_x, enemy_2_y;
// ...
var enemy_10_x, enemy_10_y;
```

さて、こういう場合に「配列」を使うと非常にスッキリと書けます。「配列」の中には変数がたくさん入っています。中の変数には添字と呼ばれる番号でアクセスできるため **for 文**で使用するカウンター変数（i）などで簡単に個々の変数にアクセスできます。以下例です。

```
// 配列を格納する変数 a を定義
var a;
// a に 100 個の要素を持つ配列を代入
```

```
a = new Array(100);
// a の各要素に数字を代入
// なお a.length は要素数（100）を返す
for(var i=0; i<a.length; i++) {
    // i 番目の要素に自分の番目（i）を代入
    a[i] = i;
}
```

これを使用すれば 1000 匹だろうが 10000 匹だろうが手間はそれほど変わりません。以下のように shooting.js を編集してください。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;
// 敵キャラの数を定義
var ENEMIES = 10;
// ...
var player_x, player_y;
// 敵キャラの現在位置（配列）を保持する変数を定義し
// ENEMIES 分だけ要素数を持つ配列を代入
var enemies_x = new Array(ENEMIES);
var enemies_y = new Array(ENEMIES);

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // ...

    // キー入力により移動したか調べる
    // 注意: 真偽値なので moved == true のようにしなくても同じ意味になる
    if(moved) {
        // キャンバスをクリアする
        ctx.clearRect(0, 0, canvas.width, canvas.height);

        // 新しい位置にプレイヤーを描画
        ctx.drawImage(img_player, player_x, player_y);

        // 敵キャラの画像を (enemies_x[i], enemies_y[i]) の位置に表示
        for(var i=0; i<ENEMIES; i++) {
            ctx.drawImage(img_enemy, enemies_x[i], enemies_y[i]);
        }
    }
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
```

```
    // ...

    // 敵キャラの初期位置を指定
    for(var i=0; i<ENEMIES; i++) {
        enemies_x[i] = Math.random() * (canvas.width - img_enemy.width);
        enemies_y[i] = Math.random() * (canvas.height - img_enemy.height);
    }

    // Playerの画像を (player_x, player_y) の位置に描画
    ctx.drawImage(img_player, player_x, player_y);
    // 敵キャラの画像を (enemies_x[i], enemies_y[i]) の位置に表示
    for(var i=0; i<ENEMIES; i++) {
        ctx.drawImage(img_enemy, enemies_x[i], enemies_y[i]);
    }
};
```

これを実行してプレイヤーを動かしてみてください。敵キャラが消えたり毎度場所が変わったりしなければ成功です。

描画処理を関数化する

プレイヤーと敵キャラの描画はページロード時処理とキー入力処理の部分で全く同じコードをコピペして使用しています。このように全く同じ処理（もしくはひどく似た処理）は**「関数化」**することでもっとシンプルに行えるようになります。この先コードがどんどん複雑になることが予想できるので、この辺でコードをシンプルにするための**「関数化」**を行いましょう。

関数とは「ある値を受け取り、ある値を返すもの」です。別な言い方をすると「ある処理をまとめたもの」です。例えば、普段良く使用する「掛け算」も関数の一例です。何故ならば「掛け算」とは「2 つの数値 a, b をとり a を b 個足した数を返すもの」と 言う事ができるからです。

では**関数**の定義の仕方と具体的な使用例を見てみましょう。

```
// 関数の定義（引数（ひきすう）は任意の数取れる。また値は返さなくても良い）
var 関数名 = function(引数 1, 引数 2, ...) {
    処理
    return 戻り値（返す値）;
};

// 掛け算の場合（引数をとり、値を返す関数）
var multiple = function(a, b) {
    // 結果を代入する変数を定義し 0 で初期化
    var result = 0;
    // b 個分ループする
```

```
    for(var i=0; i<b; i++) {
        // 結果に a を足す
        result += a;
    }
    // 最終的な結果を返す
    return result;
};
// 掛け算の使い方
console.log(multiple(2, 3));
// 出力: 6

// 掛け算結果を表示する関数（引数をとり、値を返さない関数）
var multipleDisplay = function(a, b) {
    // 先ほど作った multiple 関数を使用して結果を計算し代入する
    var result = multiple(a, b);
    // 結果を表示する（値は返さない）
    console.log(result);
};
// 使い方
multipleDisplay(2, 3);
// 出力: 6

// 2 x 3 を返す関数（引数を取らず、値を返す関数）
var twoTimesThree = function() {
    // 先ほど作った multiple 関数を使用して結果を計算し代入する
    var result = multiple(2, 3);
    // 最終的な結果を返す（返さないで表示しても良い）
    return result;
};
// 使い方
console.log(twoTimesThree());
// 出力: 6
```

この**関数**を使用して描画部分を共通化しましょう。 下記のように `shooting.js` を修正してください。

```
// ...

// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // キャンバスをクリアする
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // 新しい位置にプレイヤーを描画
    ctx.drawImage(img_player, player_x, player_y);
```

```
    // 敵キャラの画像を (enemies_x[i], enemies_y[i]) の位置に表示
    for(var i=0; i<ENEMIES; i++) {
        ctx.drawImage(img_enemy, enemies_x[i], enemies_y[i]);
    }
};

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // ...

    // キー入力により移動したか調べる
    // 注意: 真偽値なので moved == true のようにしなくても同じ意味になる
    if(moved) {
        // 再描画する
        redraw();
    }
};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // 敵キャラの初期位置を指定
    for(var i=0; i<ENEMIES; i++) {
        enemies_x[i] = Math.random() * (canvas.width - img_enemy.width);
        enemies_y[i] = Math.random() * (canvas.height - img_enemy.height);
    }

    // (再)描画する
    redraw();
};
```

このように関数を定義すると繰り返し行う処理や、あちこちで行なっている共通した処理を一箇所にまとめることができ便利です。また、このように処理を 1 箇所にまとめることで修正を行う箇所を限定できるのでバグが発生しづらく、修正も楽に行えます。

滑らかにプレイヤーを動かす

ここまでで、プレイヤーを自由自在に動かすことが出来るようになりましたが、下記の点で不満が残ります。

1. 移動がなんとなくカクカクしている

2. 上と右を入力した時にどちらかにしか移動できない（斜め移動ができない）

この章では一般的なゲームで使用されている**メインループ**を用いてこれらの問題点を解決する方法を解説します。

なぜ滑らかに動かないのか

先の移動処理は簡略化すると以下のようなステップで行われていました。

1. ユーザーがキーを入力する
2. ブラウザがキー入力を察知し、`onkeydown` イベントを発生させる
3. `onkeydown` イベントに渡されたキー情報を元にプレイヤーを移動させる

このように**イベント**が発生した段階で処理を走らせる形式をここでは**イベント駆動方式**と呼ぶことにします。対照的に、一般的なゲームでは下記のように移動処理を行なっています。

1. キーボードの状態を調べる
2. キーが入力されていた場合はプレイヤーを移動させる
3. 最初に戻る

これらの処理に当たり判定や描画処理などを加えたものを**メインループ**と呼びます。なお、これら全体の処理はゲーム種により違いはありますが、たいていミリ秒単位で行われます（60 FPS のゲームの場合約 17 ms）。

一方、イベント駆動方式の場合ユーザーがキーを押しっぱなしにすると OS が自動的に一定の速度でキーを連打します（そのため KeyDown イベントなのに押しっぱなしでも 連続的に呼ばれる）。この連打の速度は OS の設定などにより違いますが、メモ帳などで `a` キーを押し続けた 時に `a` が入力される速度と等しいのですごく遅いです。したがってイベント駆動方式でキー入力処理を行なっている今作成中のゲームは動きがカクカクしています。

また、捕捉ですが `onkeydown` イベントはあくまでもキーが押されたことを通知するイベントなので同時押しに対応していません（e.keyCode には 1 つのキーを表す番号しか反映されていない）。そのため斜め移動など、同時に複数のキーの入力を取得しなければいけない処理はできません。

メインループ方式に変更する

ストレスが無いゲームを作成するためにはメインループ方式に移行する以外手段はありません。このメインループ方式ですが、名前の通り通常は**無限ループ**を用いて下記のように記載します。

```
// for(;;){} で無限ループになる（ただしブラウザが固まるので実用性はない）
for(;;) {
    // メインループ処理
}
```

ただし、コメントにも書きましたが JavaScript ではブラウザが固まってしまうためこの方法は取れません（C/C++など、別言語ならば OS に処理を戻す処理を記載出来るためフリーズが避けられる）。

そのため仮想的にこのメインループを実装するために **setTimeout** というタイマー を利用します。

setTimeout は関数と時間を受け取り、渡された関数を渡された時間後に実行する関数です。したがって、以下のようにすると無限ループを作ることができます（馴れないと何故無限ループするのか理解に苦しむかもしれませんが）。

```
// 無限ループする関数を定義
var infinityLoop = function() {
    // コンソールに出力
    console.log("ループ中");
    // 1000 ミリ秒後に infinityLoop（この関数）を実行
    setTimeout(infinityLoop, 1000);
};
// infinityLoop を実行（無限ループ開始）
infinityLoop();
```

ただし、このままだと**実行環境によってゲーム速度が変化します**。コンピュータのスペックは個々で異なるのでメインループを処理できる時間も異なります。この違いを可能な限り小さくするために FPS（frame per second）制御と呼ばれるゲーム 時間の制御を行う必要があります。

FPS 制御の仕組みは以下のとおりです。

1. フレーム開始時間を取得
2. フレーム処理を行う
3. フレーム処理終了時間を取得
4. 開始・終了時間からフレーム処理に何秒かかったのかを計算
5. 大抵の場合処理が早すぎるので一定の FPS になるように待つ

この処理をさきほどの setTimeout と組み合わせると JavaScript でメインループ方式 を使うことができます。とりあえずキー入力によりプレイヤーの操作は後回しにするので、下記のように shooting.js を修正しメインループ方式に変更してください。

```
"use strict"
// 全体で使用する変数を定義
```

```
var canvas, ctx;
// FPS管理に使用するパラメータを定義
var FPS = 30;
var MSPF = 1000 / FPS;
// ...

// メインループを定義
var mainloop = function() {
    // 処理開始時間を保存
    var startTime = new Date();

    // 描画処理
    redraw();

    // 処理経過時間および次のループまでの間隔を計算
    var deltaTime = (new Date()) - startTime;
    var interval = MSPF - deltaTime;
    if(interval > 0) {
        // 処理が早すぎるので次のループまで少し待つ
        setTimeout(mainloop, interval);
    } else {
        // 処理が遅すぎるので即次のループを実行する
        mainloop();
    }
};

// 下記は不適とわかったためコメントアウトしている（消しても構わない）
// キーが押された時に呼び出される処理を指定
//window.onkeydown = function(e) {
//  // ...
//};

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // 敵キャラの初期位置を指定
    for(var i=0; i<ENEMIES; i++) {
        enemies_x[i] = Math.random() * (canvas.width - img_enemy.width);
        enemies_y[i] = Math.random() * (canvas.height - img_enemy.height);
    }

    // メインループを開始する
    mainloop();
};
```

まだなにも動かないので正しくメインループ方式に移行できたかわかりません。とりあえず次に進みましょう。

キーボードの状態を取得する

メインループ方式に変更したため、今までのようにイベントでキーの入力状態を取得する方法は使えません。しかし JavaScript には現在のキーの状態を取得する方法がありません。したがって onkeydown と onkeyup イベントを用いて自分で現在のキー状態を管理 します。

onkeyup イベントはキーが離された時に発生します。キーが押された時に発生する onkeydown イベントとちょうど反対の役割をします。したがって全キーの状態を配列で保持し onkeydown イベントでその配列の e.keyCode 番目を true にし onkeyup イベントで false にすればすべてのキーの状態を配列で保持できます。まずは以下のようにキー状態の保存変数の定義とイベントによる状態更新のコードを shooting.js に加えてください。なおメインループ方式に変更した際にコメントアウトした部分（window.onkeydown）は今回のコードとバッティングするので削除してください。

```
"use strict"
// 全体で使用する変数を定義
var canvas, ctx;
// FPS 管理に使用するパラメータを定義
var FPS = 30;
var MSPF = 1000 / FPS;
// キー状態管理変数の定義（確か 256 以上のキーコードは無いと思う…ちょっと怪しい）
var KEYS = new Array(256);
// キーの状態を false （押されていない）で初期化
for(var i=0; i<KEYS.length; i++) {
    KEYS[i] = false;
}

// ...

// メインループを定義
var mainloop = function() {
    // ...
};

// キーが押された時に呼び出される処理を指定
window.onkeydown = function(e) {
    // キーを押された状態に更新
    KEYS[e.keyCode] = true;
}
// キーが離された時に呼び出される処理を指定
```

```
window.onkeydown = function(e) {
    // キーを離された状態に更新
    KEYS[e.keyCode] = false;
}

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...
};
```

これでどこでも KEYS[キー番号] のようにすれば、そのキーが押されていれば true 押されていなければ false と入力状態が取得できるようになりました。

プレイヤーを滑らかに動かす

これですべての準備が整ったので、再度プレイヤーを動かすコードを書きましょう。今回はメインループから呼び出すので movePlayer() という関数を定義し、その中 にプレイヤーの移動処理を記載しました。下記を参考にして shooting.js を修正してください。

```
// ...

// プレイヤーの移動処理を定義
var movePlayer = function() {
    // 上下左右の移動速度を定義
    var SPEED = 2;

    // キー番号だとわかりにくいため予め変数に格納
    var RIGHT = 39;
    var LEFT  = 37;

    if(KEYS[RIGHT]) {
        // プレイヤーの x 座標を少し増やす
        player_x += SPEED;
    }
    if(KEYS[LEFT]) {
        // プレイヤーの x 座標を少し減らす
        player_x -= SPEED;
    }
};

// メインループを定義
var mainloop = function() {
    // 処理開始時間を保存
```

```
    var startTime = new Date();

    // プレイヤーの移動処理
    movePlayer();

    // 描画処理
    redraw();

    // 処理経過時間および次のループまでの間隔を計算
    var deltaTime = (new Date()) - startTime;
    var interval = MSPF - deltaTime;
    if(interval > 0) {
        // 処理が早すぎるので次のループまで少し待つ
        setTimeout(mainloop, interval);
    } else {
        // 処理が遅すぎるので即次のループを実行する
        mainloop();
    }
};

// ...
```

カクカクがなくなったことに気づけましたか？

プレイヤーの移動範囲を制限する

今のままだとプレイヤーはキャンバスの外にも移動できてしまいます。これじゃゲームにならないのでプレイヤーの移動可能範囲を制限しましょう。仕組みは簡単で、プレイヤーの x 座標によって移動できるかを決定します。またもしもプレイヤーがキャンバスの外にはみ出ていた場合は強制的に中に戻してやります。

下記コードを参考に `shooting.js` を修正してください。

```
// ...

// プレイヤーの移動処理を定義
var movePlayer = function() {
    // 上下左右の移動速度を定義
    var SPEED = 2;

    // キー番号だとわかりにくいため予め変数に格納
    var RIGHT = 39;
    var LEFT  = 37;

    if(KEYS[RIGHT] && player_x+img_player.width < canvas.width) {
```

```
        // プレイヤーの x 座標を少し増やす
        player_x += SPEED;
    }
    if(KEYS[LEFT] && player_x > 0) {
        // プレイヤーの x 座標を少し減らす
        player_x -= SPEED;
    }

    // プレイヤーがはみ出てしまった場合は強制的に中に戻す
    if(player_x < 0) {
        player_x = 0;
    } else if (player_x + img_player.width > canvas.width) {
        player_x = canvas.width - img_player.width;
    }
};
// ...
```

なお条件式で使用している `&&` は論理演算子とゆうもので日本語で言う「かつ」を表します。したがってプレイヤーがキャンバス内にいる場合のみキーの入力を受け付け、またプレイヤーがはみ出てしまった場合は強制的にキャンバス内に戻しています。

敵キャラを動かす

シューティングゲームなので敵キャラにも動いてもらう必要があります。すでにメインループ方式に移行しているので簡単です。

どう動かすのかを考える

通常シューティングには様々な敵キャラが存在し、様々な動きを見せます。隊列を組んで来るものやランダムに飛んでくるもの、その多様性がシューティングの醍醐味かもしれません。

こんだけ言っておきながら、今回は分量の関係上敵キャラは上から下に降りてくるだけにします。また上から下に降りてくるだけだとすぐに敵キャラがいなくなってしまうので下に抜けた敵キャラは再度上に戻るようにプログラムします。

実際のプログラム

「キャラクターを動かす方法」や「移動範囲を制限する方法」はすでに行ったので特に解説はありません。 下記コードを参照して `shooting.js` を書き換えてください。

```
// ...

// 敵キャラの移動処理を定義
var moveEnemies = function() {
    // 上下左右の移動速度を定義
    var SPEED = 2;

    // 各敵キャラごとに処理を行う
    for(var i=0; i<ENEMIES; i++) {
        // 敵キャラの y 座標を少し増やす
        enemies_y[i] += SPEED;

        // 敵キャラが下画面にはみ出た場合は上に戻す
        if (enemies_y[i] > canvas.height) {
            enemies_y[i] = -img_enemy.height;
            // せっかくなので x 座標を再度ランダムに設定
            enemies_x[i] = Math.random() * (canvas.width - img_enemy.width);
        }
    }
};

// メインループを定義
var mainloop = function() {
    // 処理開始時間を保存
    var startTime = new Date();

    // プレイヤーの移動処理
    movePlayer();
    // 敵キャラの移動処理
    moveEnemies();

    // ...
}

// ...
```

実行すると敵キャラが上から永遠と降ってくるのが確認できると思います。

当たり判定をする

ついにここまでたどり着きました。シューティングゲームの花形である当たり判定を解説します。まぁ当たり判定といってもいろいろな方法がありますが、今回は初心者向けという事で比較的簡単な「円を使った当たり判定」のみ解説します。

三平方の定理

別名ピタゴラスの定理ですね。 これさえわかっていれば簡単です。「直角三角形の斜辺の長さの二乗は他二辺の長さの二乗を足しあわせたものと等しくなる」という定理です。 思い出せましたか？

当たり判定への応用

応用もなにも下記図を見ていただければ一発だと思います。

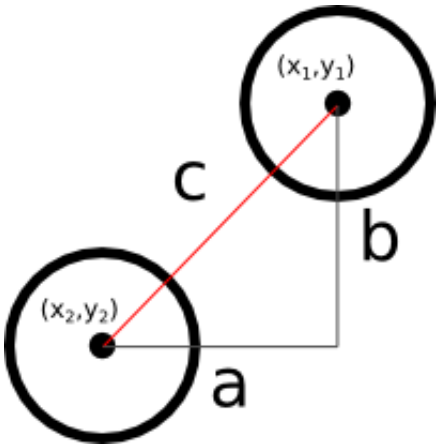

Figure 4: 円の当たり判定

図において各円はキャラクターや弾を表しているものとします。各キャラクターの中心座標からキャラクター同士の距離が計算できますが、この距離が各キャラクターの当たり判定用円の半径を足しあわせたものよりも短い場合は「当たっている」。それ以外の場合は「当たっていない」というシンプルな方法です。

汎用的な当たり判定関数の作成

シューティングゲームなので当たり判定の対象がキャラクターだったり弾だったりします。いちいちそれぞれの当たり判定を書いていられないので、汎用的な当たり判定関数を作成することにします。

上記「当たり判定への応用」を整理すると、当たり判定に必要なのは

1. 自分・対象の中心座標
2. 自分・対象の当たり判定用円の半径

の 2 つです。今回は簡易化のため、当たり判定用の円は画像と同じ大きさにします。また中心座標は自分の座標に横幅・縦幅の半分を足したものとします。した

がって下記のような関数を作れば良いことになります。下記関数を shooting.js の「// メインループを定義」の上あたりに定義してください。

```
var hitCheck = function(x1, y1, obj1, x2, y2, obj2) {
    var cx1, cy1, cx2, cy2, r1, r2, d;
    // 中心座標の取得
    cx1 = x1 + obj1.width/2;
    cy1 = y1 + obj1.height/2;
    cx2 = x2 + obj2.width/2;
    cy2 = y2 + obj2.height/2;
    // 半径の計算
    r1 = (obj1.width+obj1.height)/4;
    r2 = (obj2.width+obj2.height)/4;
    // 中心座標同士の距離の測定
    // Math.sqrt(d) -- d のルートを返す
    // Math.pow(x, a) -- x の a 乗を返す
    d = Math.sqrt(Math.pow(cx1-cx2, 2) + Math.pow(cy1-cy2, 2));
    // 当たっているか判定
    // ちなみに `return r1+r2 > d;` とだけ書いても OK
    if(r1 + r2 > d) {
        // 当たってる
        return true;
    } else {
        // 当たっていない
        return false;
    }
};
```

プレイヤーと敵との当たり判定を作成

汎用的な関数ができたので、これを用いてプレイヤーと敵の当たり判定を書きます。ただ、現在プレイヤーにも敵にもヒットポイントや死などの概念が無いため当たり判定をしても面白みがありません。したがってそれぞれにヒットポイントという変数を持たせ、ヒットポイントが 0 になったら描画をしないようにしてみます。

下記を参照しながら shooting.js を修正してください。

```
// ...
// 敵キャラの現在位置（配列）を保持する変数を定義し
// ENEMIES 分だけ要素数を持つ配列を代入
var enemies_x = new Array(ENEMIES);
var enemies_y = new Array(ENEMIES);
// プレイヤーのヒットポイント
var player_hp;
```

```
// 敵キャラのヒットポイント（配列）を保持する変数を定義し
// ENEMIES 分だけ要素数を持つ配列を代入
var enemies_hp = new Array(ENEMIES);


// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // キャンバスをクリアする
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // 生きている場合だけ新しい位置にプレイヤーを描画
    if(player_hp > 0) {
        ctx.drawImage(img_player, player_x, player_y);
    }

    // 敵キャラの画像を (enemies_x[i], enemies_y[i]) の位置に表示
    for(var i=0; i<ENEMIES; i++) {
        // 生きている場合だけ描画
        if(enemies_hp[i] > 0) {
            ctx.drawImage(img_enemy, enemies_x[i], enemies_y[i]);
        }
    }
};

// プレイヤーの移動処理を定義
var movePlayer = function() {
    // ヒットポイントを確認し、生きている場合のみ処理をする
    if(player_hp <= 0) {
        return;
    }

    // ...

};
// 敵キャラの移動処理を定義
var moveEnemies = function() {
    // 上下左右の移動速度を定義
    var SPEED = 2;

    // 各敵キャラごとに処理を行う
    for(var i=0; i<ENEMIES; i++) {
        // ヒットポイントを確認し、生きている場合のみ処理をする
        if(enemies_hp[i] <= 0) {
            // ループの残りのステップを無視して次のループに行く場合
            // は `continue` を指定する
            continue;
```

```
        }

        // ...
    }
};

// 汎用的当たり判定関数
var hitCheck = function(x1, y1, obj1, x2, y2, obj2) {
    // ...
};

// メインループを定義
var mainloop = function() {
    // ...

    // 敵キャラの移動処理
    moveEnemies();

    // プレイヤーと敵キャラの当たり判定（プレイヤーが生きている場合）
    if(player_hp > 0) {
        for(var i=0; i<ENEMIES; i++) {
            // 敵が生きている場合のみ判定する
            if(enemies_hp[i] > 0) {
                if(hitCheck(player_x, player_y, img_player,
                            enemies_x[i], enemies_y[i], img_enemy)){
                    // 当たっているのでお互いの HP を 1 削る
                    player_hp -= 1;
                    enemies_hp[i] -=1;
                }
            }
        }
    }

    // 描画処理
    redraw();

    // ...
};

// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Player の初期位置および HP を指定
```

```
    // player_x = キャンバスの左右中央
    // player_y = キャンバスの下から 20px 上
    player_x = (canvas.width - player.width) / 2;
    player_y = (canvas.height -player.height) - 20;
    player_hp = 10;

    // 敵キャラの初期位置および HP を指定
    for(var i=0; i<ENEMIES; i++) {
        enemies_x[i] = Math.random() * (canvas.width - img_enemy.width);
        enemies_y[i] = Math.random() * (canvas.height - img_enemy.height);
        enemies_hp[i] = 2;
    }

    // メインループを開始する
    mainloop();
};
```

弾を打って敵を倒す

さて、いい加減弾を打てるようにします。現段階でも見た目の調整（画像差し替えやエフェクト）をすれば結構ゲームぽくはなりますが、一応シューティングを作るのが目的なので外せません。

弾を発射する

基本的には敵を動かしたときと同様に **for 文**で処理をします。ただ弾は任意のタイミングで現れるので、必要になるまでは HP を 0 として保持し、発射された弾の HP のみ 1 とします。このように弾の状態を HP で管理しておけばプレイヤーや敵キャラと似た処理をおこなえます（HP が 0 なら表示や操作をしない）。それらを踏まえて下記コードを参考に `shooting.js` を書き換えてください。

```
// ...
// 弾の数を定義（同時に描画される弾の最大数より大きい必要あり）
var BULLETS = 5;
// ...
var img_player;
// プレイヤーの弾画像を保持する変数を定義
var img_player_bullet;
// ...
var player_x, player_y;
// プレイヤーの弾の現在位置（配列）を保持する変数を定義し
// BULLETS 分だけ要素数を持つ配列を代入
var player_bullets_x = new Array(BULLETS);
```

```
var player_bullets_y = new Array(BULLETS);
// ...
var player_hp;
// 弾のヒットポイント（配列）を保持する変数を定義し
// BULLETS 分だけ要素数を持つ配列を代入
var player_bullets_hp = new Array(BULLETS);
// ...


// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // ...

    // 生きている場合だけ新しい位置にプレイヤーを描画
    // ...

    // 弾の画像を (bullets_x[i], bullets_y[i]) の位置に表示
    for(var i=0; i<BULLETS; i++) {
        // 生きている場合だけ描画
        if(player_bullets_hp[i] > 0) {
            ctx.drawImage(img_player_bullet,
                          player_bullets_x[i],
                          player_bullets_y[i]);
        }
    }

    // 敵キャラの画像を (enemies_x[i], enemies_y[i]) の位置に表示
    // ...
};

// プレイヤーの移動処理を定義
var movePlayer = function() {
    // ...

    // キー番号だとわかりにくいため予め変数に格納
    var RIGHT = 39;
    var LEFT  = 37;
    var SPACE = 32;

    // ...

    if(KEYS[SPACE]) {
        // 未使用の弾があれば発射する
        for(var i=0; i<BULLETS; i++) {
            if(player_bullets_hp[i] == 0) {
                // 弾の初期位置はプレイヤーと同じ位置にする
```

```
                player_bullets_x[i] = player_x;
                player_bullets_y[i] = player_y;
                // 弾の HP を 1 にする。これにより次のループから描画や移動処理
                // が行われるようになる
                player_bullets_hp[i] = 1;
                // 弾は打ったのでループを抜ける
                // ループ処理を途中でやめる場合は `break` を使う
                break;
            }
        }
    }

    // プレイヤーがはみ出てしまった場合は強制的に中に戻す
    // ...
};
// プレイヤーの弾の移動処理を定義
var movePlayerBullets = function() {
    // 上下左右の移動速度を定義
    var SPEED = -6;

    // 各弾ごとに処理を行う
    for(var i=0; i<BULLETS; i++) {
        // ヒットポイントを確認し、生きている場合のみ処理をする
        if(player_bullets_hp[i] <= 0) {
            // ループの残りのステップを無視して次のループに行く場合
            // は `continue` を指定する
            continue;
        }

        // 弾の y 座標を少し増やす（減らす）
        player_bullets_y[i] += SPEED;

        // 弾が上画面にはみ出た場合は HP を 0 にして未使用状態に戻す
        if (player_bullets_y[i] < img_player_bullet.height) {
            player_bullets_hp[i] = 0;
        }
    }
};
// 敵キャラの移動処理を定義
// ...

// メインループを定義
var mainloop = function() {
    // ...

    // プレイヤーの移動処理
```

```
    movePlayer();
    // プレイヤーの弾の移動処理
    movePlayerBullets();

    // ...
};

// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
    // ...

    // Playerの画像（id='player'で指定された<img>）を取得
    img_player = document.getElementById('player');
    // Playerの弾画像（id='player_bullet'で指定された<img>）を取得
    img_player_bullet = document.getElementById('player_bullet');
    // ...

    // 弾の初期位置およびHPを指定
    for(var i=0; i<BULLETS; i++) {
        player_bullets_x[i] = 0;
        player_bullets_y[i] = 0;
        player_bullets_hp[i] = 0;
    }
    // 敵キャラの初期位置およびHPを指定
    // ...
};
```

実行しスペースを押すと「バグバグ」と書かれた弾が 5 個連なって発射されると思います。これは連射速度が早すぎて弾が 5 個では足りないために起こる現象です。

間隔をおいて弾を発射する

先の状態だと一気に弾が発射されてしまいました。 連射速度が早過ぎますね。したがって「カウンター」を使用して数フレームの間弾の発射を禁止します。詳しく説明すると、インターバルカウンターの値が 0 の時のみ発射を許可するようにします。弾を発射した後はこのカウンターに大きな値を入れます。そして毎フレームごとに 1 づつ値を減らします。するとカウンターの値は最初に設定した値のフレーム数後に 0 に戻るのでまた発射可能になります。

下記コードを参考に `shooting.js` を書き換えてください。

```
// ...
// 発射インターバルの値を定義（この値が大きいほど連射が遅くなる）
```

```
var FIRE_INTERVAL = 20;
// 弾の数を定義（同時に描画される弾の最大数より大きい必要あり）
// ...
var enemies_hp = new Array(ENEMIES);
// プレイヤーの発射インターバル
var player_fire_interval=0;

// ...

// プレイヤーの移動処理を定義
var movePlayer = function() {
    // ...

    // スペースキーが押され、なおかつ発射インターバルが 0 の時だけ発射する
    if(KEYS[SPACE] && player_fire_interval == 0) {
        // 未使用の弾があれば発射する
        for(var i=0; i<BULLETS; i++) {
            if(player_bullets_hp[i] == 0) {
                // ...
                // 弾を打ったので発射インターバルの値を上げる
                player_fire_interval = FIRE_INTERVAL;
                // 弾は打ったのでループを抜ける
                // ループ処理を途中でやめる場合は `break` を使う
                break;
            }
        }
    }
    // 発射インターバルの値が 0 より大きい場合は値を減らす。
    if(player_fire_interval > 0) {
        // 変数++; や 変数--; はそれぞれ 1 増やす、減らすという処理
        // そのため下記は `player_fire_interval -= 1;`と等価
        player_fire_interval--;
    }
    // ...
};
```

この変更により連射速度が大幅に落ち、同時に 5 個以上の弾がキャンバス内に存在することはなくなりました。

当たり判定処理

最後に弾と敵の当たり判定を行います。基本的にプレイヤーと敵の当たり判定と同様ですが、弾も敵も配列に入っているため二重ループを使わなければならない部分が異なります。**添字に注意して**下記のように shooting.js を修正してください。

```
// ...

// メインループを定義
var mainloop = function() {
    // ...

    // プレイヤーと敵キャラの当たり判定（プレイヤーが生きている場合）
    if(player_hp > 0) {
        for(var i=0; i<ENEMIES; i++) {
            // 敵が生きている場合のみ判定する
            if(enemies_hp[i] > 0) {
                if(hitCheck(player_x, player_y, img_player,
                            enemies_x[i], enemies_y[i], img_enemy)){
                    // 当たっているのでお互いの HP を 1 削る
                    player_hp -= 1;
                    enemies_hp[i] -=1;
                }
            }
        }
    }
    // プレイヤー弾と敵キャラの当たり判定（プレイヤーが生きている場合）
    if(player_hp > 0) {
        for(var i=0; i<ENEMIES; i++) {
            // 敵が死んでいる場合はスルーする
            if(enemies_hp[i] <= 0) {
                continue;
            }
            for(var j=0; j<BULLETS; j++) {
                // 弾が死んでいる場合はスルーする
                if(player_bullets_hp[j] <= 0) {
                    continue;
                }
                if(hitCheck(player_bullets_x[j],
                            player_bullets_y[j],
                            img_player_bullet,
                            enemies_x[i],
                            enemies_y[i],
                            img_enemy)){
                    // 当たっているのでお互いの HP を 1 削る
                    player_bullets_hp[j] -= 1;
                    enemies_hp[i] -=1;
                }
            }
        }
    }
```

```
    // ...
}

// ...
```

敵が弾に二回当たると消えれば成功です。

装飾する

「自分が死なないように弾を打って敵を倒す」というシューティングゲームの根幹は完成しましたが、どうもあまりゲームっぽくありません。この記事の締めくくりとして、ゲームっぽさを出すために画面を少し装飾します。

残り HP を表示する

残り HP を表示しましょう。 以下のように `shooting.js` を修正してください。

```
// ...
// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // ...

    // コンテキストの状態を保存（fillStyle を変えたりするので）
    ctx.save();
    // HP の最大値（10）x 5 の短形を描画（白）
    ctx.fillStyle = '#fff';
    ctx.fillRect(10, canvas.height-10, 10 * 5, 5);
    // 残り HP x 5 の短形を描画（赤）
    ctx.fillStyle = '#f00';
    ctx.fillRect(10, canvas.height-10, player_hp * 5, 5);
    // コンテキストの状態を復元
    ctx.restore();
};
// ...
```

左下に HP バーが表示されれば成功です。

倒した敵の数を表示

倒した敵の数を表示しましょう。 キャンバスには `fillText` という文字列を描画するメソッドがあるのでこれを使用します。 以下のように `shooting.js` を修正してください。

```
// ...
// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // ...

    // コンテキストの状態を保存（fillStyle を変えたりするので）
    ctx.save();
    // HP の最大値（10）x 5 の短形を描画（白）
    ctx.fillStyle = '#fff';
    ctx.fillRect(10, canvas.height-10, 10 * 5, 5);
    // 残り HP x 5 の短形を描画（赤）
    ctx.fillStyle = '#f00';
    ctx.fillRect(10, canvas.height-10, player_hp * 5, 5);

    // 「倒した敵の数/全敵の数」という文字列を作成
    var text = "Killed: " + killed + "/" + ENEMIES;
    // 文字列の（描画）横幅を計算する
    var width = ctx.measureText(text).width;
    // 文字列を描画（白）
    ctx.fillStyle = '#fff';
    ctx.fillText(text,
                 canvas.width - 10 - width,
                 canvas.height - 10);

    // コンテキストの状態を復元
    ctx.restore();
};
// ...
```

無敵時間を作る

通常、シューティングなどで攻撃をくらうと一定時間無敵になります。これは毎フレームごとに「当たった」と言う処理が走り HP がものすごい勢いで削られるのを防ぐための処理です。今回、敵の HP が 2 しか無いので敵に当たってもダメージは 2 で済みますが、この先とても硬い敵（プレイヤーにあたっても死なないような敵）を作るとすると同様の問題が発生します。 そこで、例に習って無敵時間を作りましょう。

無敵時間を作る際に必要な考え方は、弾の連射速度を調整した時の考え方と同様です。攻撃を受けた後、カウンターの値を一定値に設定し、その値が 0 に戻るまでは無敵と扱うことにします。

なお、この無敵時間は敵にも存在すべきなのですが、面倒くさいので割愛します。余力のある人は例に習って敵にも無敵時間を作ってみてください。では下記を参考に `shooting.js` を修正してください。

```
// ...
// 発射インターバルの値を定義（この値が大きいほど連射が遅くなる）
var FIRE_INTERVAL = 20;
// 無敵インターバルの値を定義（この値が大きいほど無敵時間が長くなる）
var STAR_INTERVAL = 20;
// ...
// プレイヤーの発射インターバル
var player_fire_interval=0;
// プレイヤーの無敵インターバル
var player_star_interval=0;

// ...

// メインループを定義
var mainloop = function() {
    // ...

    // プレイヤーと敵キャラの当たり判定（プレイヤーが生きている場合）
    // かつプレイヤーが無敵ではない場合
    if(player_hp > 0 && player_star_interval == 0) {
        for(var i=0; i<ENEMIES; i++) {
            // 敵が生きている場合のみ判定する
            if(enemies_hp[i] > 0) {
                if(hitCheck(player_x, player_y, img_player,
                            enemies_x[i], enemies_y[i], img_enemy)){
                    // ...

                    // プレイヤーを無敵状態にする
                    player_star_interval = STAR_INTERVAL;
                }
            }
        }
    }
    // プレイヤーの無敵インターバルを減少させる
    if(player_star_interval > 0) {
        player_star_interval--;
    }

    // ...
};

// ...
```

無敵状態時に点滅させる

せっかくなので無敵時間中は点滅するようにしましょう。 まず JavaScript では % という演算子で「あまり」を計算できます。 したがって X % 2 とすると 2 で割り切れる時は 0、割り切れないときは 1 が帰って きます。これを用いると毎フレームごとに ON と OFF が切り替えられます。

またキャンバスは描画に透過度を持たすことができます ctx.globalAlpha の値を変更してやると透過度が変わるので、これを使用して OFF の場合は半透明に描画する ようにします。

では下記を参考に shooting.js を修正してください。

```
// ...

// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // キャンバスをクリアする
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // 生きている場合だけ新しい位置にプレイヤーを描画
    if(player_hp > 0) {
        // 透過度を変えるのでコンテキストの状態を保存
        ctx.save();
        // 無敵時間の状態に応じて描画の透過度を変更
        if(player_star_interval % 2 != 0) {
            // 半透明に描画する
            ctx.globalAlpha = 0.5;
        }
        ctx.drawImage(img_player, player_x, player_y);
        // コンテキストの状態を戻す
        ctx.restore();
    }
    // 弾の画像を (bullets_x[i], bullets_y[i]) の位置に表示
    for(var i=0; i<BULLETS; i++) {
        // 生きている場合だけ描画
        if(player_bullets_hp[i] > 0) {
            ctx.drawImage(img_player_bullet,
                          player_bullets_x[i],
                          player_bullets_y[i]);
        }
    }
    // ...
};
// ...
```

ゲームオーバー画面を作る

現状、ゲームオーバになっても特にナニも起こりません。これじゃつまらないのでゲームオーバーになったら「Game Over」と表示するように しましょう。 下記を参考に `shooting.js` を修正してください。

```
// ...

// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // ...

    // コンテキストの状態を保存（fillStyle を変えたりするので）
    // ...

    // 「倒した敵の数/全敵の数」という文字列を作成
    // ...

    // プレイヤーが死んでいた場合ゲームオーバー画面を表示する
    if(player_hp <= 0){
        // 全体を半透明の黒い四角で覆う（オーバーレイ）
        ctx.globalAlpha = 0.5;
        ctx.fillStyle = '#000';
        ctx.fillRect(0, 0, canvas.width, canvas.height);
        ctx.globalAlpha = 1.0;

        // 真ん中に大きな文字でゲームオーバー（赤）と表示する
        ctx.font = '20px sans-serif';
        ctx.textBaseline = 'middle';    // 上下位置のベースラインを中心に
        ctx.fillStyle = '#f00';
        text = "Game Over";
        width = ctx.measureText(text).width;
        ctx.fillText(text,
                     (canvas.width - width) / 2,
                     canvas.height / 2);
    }

    // コンテキストの状態を復元
    ctx.restore();
};

// ...
```

ゲームクリア画面を作る

では逆にゲームクリア画面も作りましょう。 下記を参考に shooting.js を修正してください。

```
// ...

// 再描画する関数（無引数、無戻り値）
var redraw = function() {
    // ...

    // コンテキストの状態を保存（fillStyle を変えたりするので）
    // ...

    // 「倒した敵の数/全敵の数」という文字列を作成
    // ...

    // プレイヤーが死んでいた場合ゲームオーバー画面を表示する
    if(player_hp <= 0){
        // ...
    }
    // 敵を殲滅していた場合はゲームクリア画面を表示
    else if(killed == ENEMIES){
        // 全体を半透明の黒い四角で覆う（オーバーレイ）
        ctx.globalAlpha = 0.5;
        ctx.fillStyle = '#000';
        ctx.fillRect(0, 0, canvas.width, canvas.height);
        ctx.globalAlpha = 1.0;

        // 真ん中に大きな文字でゲームクリア（白）と表示する
        ctx.font = '20px sans-serif';
        ctx.textBaseline = 'middle';    // 上下位置のベースラインを中心に
        ctx.fillStyle = '#fff';
        text = "Game Clear";
        width = ctx.measureText(text).width;
        ctx.fillText(text,
                     (canvas.width - width) / 2,
                     canvas.height / 2);
    }

    // コンテキストの状態を復元
    ctx.restore();
};

// ...
```

タイトル画面を作る（画面遷移）

では最後に、タイトル画面を作成しましょう。タイトル画面では「Hit SPACE to Start」と表示させ、スペースキーが押されたらゲームを開始するようにします。

さて、ゲームオーバー画面やゲームクリア画面と同じ方法でこのタイトル画面を作ることはとても難しいです。なので根本的に考え方を変え、**画面遷移（ステージ移行）**の考え方を用います。

画面遷移にはいくつかの方法がありますが、今回はループの内容まるごと変更する方法を取ります。この方法の利点として、どのような画面遷移にも応用が聞くという利点があります。今回はタイトル画面からゲーム画面への遷移ですが、原理的にはステージ遷移やポーズ画面への遷移、オプション画面への遷移など様々な応用が効きます。

以下画面遷移の概要です。

1. 通常どおりループ内処理を行う
2. ループの最後で特定条件にあてはまるかチェック（SPACE が押された、ステージクリアしたなど）
    1. 当てはまらない場合はスルーして通常通りループを継続
    2. 当てはまる場合はループを終了し、次のループ関数を呼び出す

今回はタイトル画面からゲーム画面への遷移だけなので下記のようにシンプルに行なえます。 下記を参照して `shooting.js` を修正してください。

```
// ...

// タイトルループを定義
var titleloop = function() {
    // 処理開始時間を保存
    var startTime = new Date();

    // キャンバスをクリアする
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Hit SPACE to Start と表示
    ctx.save();
    ctx.font = '20px sans-serif';
    ctx.textBaseline = 'middle';    // 上下位置のベースラインを中心に
    ctx.fillStyle = '#fff';
    var text = "Hit SPACE to Start";
    var width = ctx.measureText(text).width;
    ctx.fillText(text,
                 (canvas.width - width) / 2,
                 canvas.height / 2);
```

```
    ctx.restore();

    // スペースが押されていた場合は mainloop を呼び出して、titleloop を終了
    var SPACE = 32;
    if(KEYS[SPACE]) {
        // メインループを呼び出す
        mainloop();
        // 継続処理をせずに関数を終了（titleloop を抜ける）
        return;
    }

    // 処理経過時間および次のループまでの間隔を計算
    var deltaTime = (new Date()) - startTime;
    var interval = MSPF - deltaTime;
    if(interval > 0) {
        // 処理が早すぎるので次のループまで少し待つ
        setTimeout(titleloop, interval);
    } else {
        // 処理が遅すぎるので即次のループを実行する
        // Note: titleloop() を直接呼び出すとフリーズします。
        setTimeout(titleloop, 0);
    }
};


// メインループを定義
var mainloop = function() {
    // ...

    // 処理経過時間および次のループまでの間隔を計算
    var deltaTime = (new Date()) - startTime;
    var interval = MSPF - deltaTime;
    if(interval > 0) {
        // 処理が早すぎるので次のループまで少し待つ
        setTimeout(mainloop, interval);
    } else {
        // 処理が遅すぎるので即次のループを実行する
        // Note: mainloop() を直接呼び出すとフリーズするの忘れてた……
        setTimeout(mainloop, 0);
    }
};

// ...

// ページロード時に呼び出される処理を指定
window.onload = function() {
```

```
    // ...

    // タイトルループを開始する（メインループでは無いことに注意）
    titleloop();
};
```

これを実行するとまずタイトル画面が表示され、スペースを押すとゲームが開始します。この際、プレイヤーが弾を打ってしまいますが解決方法は 2 つあります。

1. 弾を打つキーを変える
2. ゲーム開始前にカウントダウンするなどし、画面遷移後に即ゲームを始めない

通常は 2 の方法を取ることが多いと思います。この方法はすでに学んだカウンターを利用すれば簡単に実装できるので、がんばってご自身で実装してみてください。

ちょっとカッコイイタイトル画面を作る（オプション）

ここからは完全にセンスの問題なのでオプションです。今までの技術を駆使して少しだけタイトル画面をかっこ良くしました。下記のように `shooting.js` を変更してみてください。

```
// ...
// タイトルループを定義
var titleloop_blinker = 0;
var titleloop = function() {
    // 処理開始時間を保存
    var startTime = new Date();

    // キャンバスをクリアする
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    ctx.save();
    // ラインで装飾
    // 参考: http://www.html5.jp/canvas/ref/method/lineTo.html
    ctx.strokeStyle = '#fff';
    ctx.beginPath();
    ctx.moveTo(20, 100);
    ctx.lineTo(canvas.width-20, 100);
    ctx.stroke();
    ctx.beginPath();
    ctx.moveTo(20, 145);
    ctx.lineTo(canvas.width-20, 145);
    ctx.stroke();
```

```
    ctx.strokeStyle = '#444';
    ctx.beginPath();
    ctx.moveTo(30, 90);
    ctx.lineTo(canvas.width-30, 90);
    ctx.stroke();
    ctx.beginPath();
    ctx.moveTo(30, 155);
    ctx.lineTo(canvas.width-30, 155);
    ctx.stroke();

    var text, width;
    // JavaScript Shooting と表示
    ctx.font = '20px serif';
    ctx.textBaseline = 'middle';    // 上下位置のベースラインを中心に
    ctx.fillStyle = '#fff';
    text = "JavaScript Shooting";
    width = ctx.measureText(text).width;
    ctx.fillText(text, (canvas.width - width) / 2, 120);

    // Hit SPACE to Start と表示
    titleloop_blinker++;
    if(titleloop_blinker > 20) {
        // 点滅処理様に透過度を調整
        ctx.globalAlpha = 0.5;
        // 100 を超えていたら 0 に戻す
        if(titleloop_blinker > 30) {
            titleloop_blinker = 0;
        }
    }
    ctx.font = '12px sans-serif';
    ctx.textBaseline = 'middle';    // 上下位置のベースラインを中心に
    ctx.fillStyle = '#ddd';
    text = "Hit SPACE to Start";
    width = ctx.measureText(text).width;
    ctx.fillText(text, (canvas.width - width) / 2, 240);
    ctx.globalAlpha = 1.0;
    ctx.restore();

    // ...
};
// ...
```

「Hit SPACE to Start」だけ表示されていたバージョンより少しカッコイイでしょ？これで「JavaScript で作る初めてのゲーム作成講座」は終了ですお疲れ様でした。

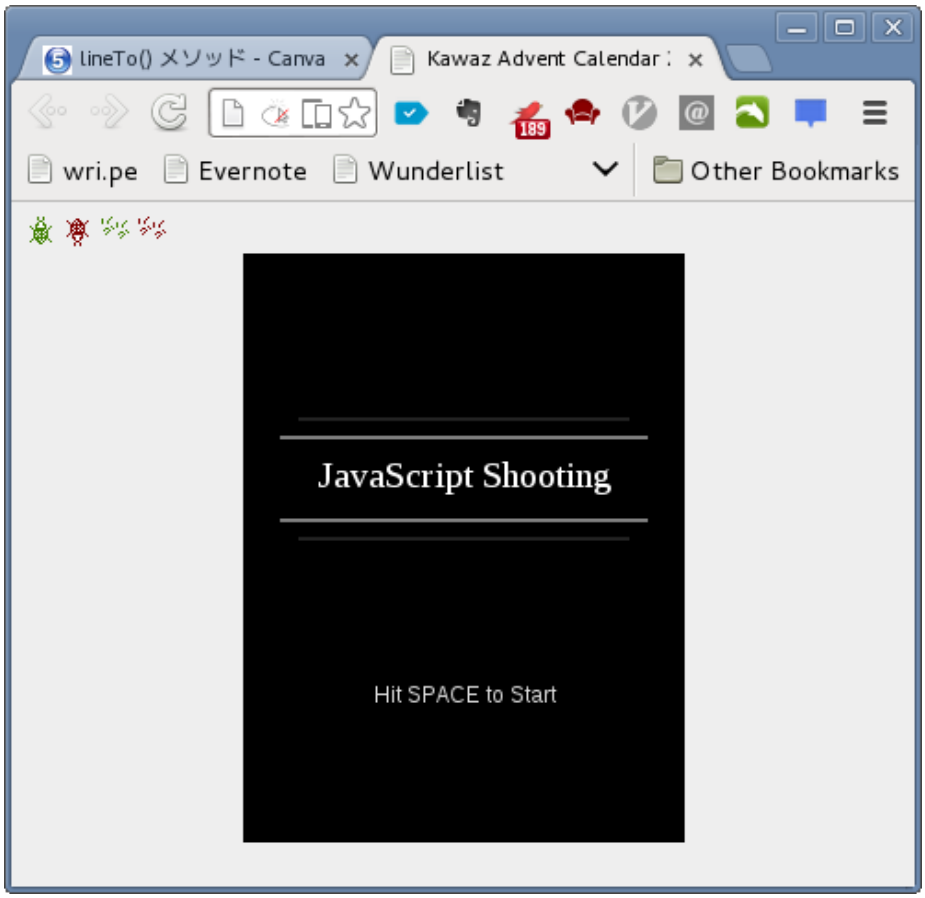

Figure 5: ちょっとカッコイイタイトル画面

おわりに

書く方もとってもしんどかったです。 正直初心者向け解説舐めてました。

今回作成したシューティングはゲームとしてはまだまだです。正直遊んでて面白くないですから（笑。

この先の改造はやりたい人が勝手にやればいいという事で、皆さんに丸投げします。まぁ僕が思いつく程度で言えば

1. 加速度を取り入れて操作を難しくしてもいいのでは？
2. 三角関数などを駆使して弾幕を作ってみても面白いかも
3. 敵にも弾を打たせようぜ
4. 敵の種類も増やそうぜ
5. 敵の動き方が単調すぎるので面白い動き方を考えてみては？
6. マップチップの考え方などを使ってステージをデザインしてもいいかも（今ランダムに配置しているだけなので）
7. せっかくならステージをいくつか作って画面遷移で切り替えとか？
8. ポーズ機能作ってみるとか？
9. アイテムをとったら強化させるとか？

まぁ上げて言ったらきりが無いですけどね（笑。ちなみに上記に書いた内容は（三角関数とか以外は）今回学んだ内容だけで全部実現可能です。

また、今回は初心者向けという事で割愛しましたが「オブジェクト指向」というのを取り入れるとコーディングがぐんと楽になります。僕はこの記事を書くために何本か似たようなシューティングを作ったのですが、この初心者向けのコードを書くのには一日くらいかかったのに対し、自分の楽な手法（オブジェクト指向などを駆使した方法）で書いたものは数時間で出来ました。あくまで一例ですが、それくらいオブジェクト指向は理解するとコーディング時に楽ができます（処理も追いやすくなりますし、理解もしやすくなります)。ただ、今回オブジェクト指向を取り入れなかったことからも分かるように、この考え方は結構初心者泣かせです。ただ大きなものを作るときに必須と言える考え方なので、この記事でゲーム作りやプログラミングに興味を持った方は是非一度学んでみてください。

感想などいただけるとすごく嬉しいです。 ではお疲れ様でした。