Table of Contents

初めに 1.1

- 本書の構成 1.1.1
- 本書の対象者 1.1.2
- 使用しているツールのバージョン 1.1.3

環境構築 1.2

- ディレクトリ構成 1.2.1
- gulp 1.2.2
- browserify 1.2.3
- browser-sync 1.2.4
- gulpfileの作成 1.2.5

ゲームループ 1.3

- 処理落ちとフレーム落ち 1.3.1
- CanvasAPIを使ってみる 1.3.2
- 終わりに 1.3.3

シーンとオブジェクト 1.4

- 概要 1.4.1
- シーン 1.4.2
- シーン切り替え 1.4.3
- オブジェクト 1.4.4
- シーンにオブジェクトの追加 1.4.5

描画 1.5

- 画像の読み込み 1.5.1
- スプライト 1.5.2
- スプライトアニメーション 1.5.3

入力 1.6
キーボード入力 1.6.1
それ以外のキー情報の保存方法 1.6.2
ゲームパッド入力 1.6.3
オブジェクト生成 1.7
ObjectBaseクラスの作成 1.7.1
ショット 1.7.2
敵 1.7.3
敵マスター 1.7.4
移動 1.8
キャラ移動 1.8.1
ベクトル移動 1.8.2
自機狙い 1.8.3
当たり判定 1.9
円同士の判定 1.9.1
矩形同士の判定 1.9.2
BGM/SE再生 1.10
HTML5Audio 1.10.1
WebAudio 1.10.2
流れ 1.10.2.1
コンテクスト 1.10.2.2
ノード 1.10.2.3
モジュラールーティング 1.10.2.4
iOS端末での再生 1.10.3
オーディオの読み込み 1.10.4
フェードイン／フェードアウト 1.10.5
その他 1.11

FPS計算 1.11.1
セーブ 1.11.2
画面最大化 1.11.3
Electron 1.11.4
参考文献 1.12
後書き 1.13

初めに

ゲームを作りたくありませんか。本書では、Webエンジニアを対象に、 ゲームを作る上で必要な知識な基本的な知識を解説します。 この本を読了することで、あなたは以下のようになることができます。

- ゲームに必要なプログラムを自力で組める
- 応用的なゲームプログラミングの情報を収集して、理解し知識として蓄えられる

本書の構成

本書では、シューティングゲームを作りながら、ゲームプログラミングについて学んでいきます。 シューティングゲームを題材としたのは、プログラムが比較的容易なこと、 また人に最低限遊んでもらえるまでにブラッシュアップする上で、必要な画像や BGM が少なく、 レベルデザインも容易であるからです。

なおゲームエンジンやフレームワークを使わないで進めます。これは、ゲームエンジンやフレームワークに 普段隠蔽された処理なども自分で組んでいくことで、プログラムの全体像を掴めるようになるためです。

本書の対象者

本書の下記の読者像を想定して制作されています。

- Web プログラマーだ
- ゲームを作りたい
- ゲームの作り方は全然わからない

事前知識としてある程度のプログラミングの経験、及び JavaScript の知識が 必要になるかと思います。巻末に参考文献を記載してますので、 内容が難しいようでしたら、まずそちらを読むのが良いと思います。

使用しているツールのバージョン

Chrome 55 にて確認しております(2017/2)

環境構築

本書では、タスクランナーである gulp を使い、開発環境を構築したいと思います。 これは、下記の目的を満たしたいためです。

- クラス毎にファイルを分割したいため
- コード修正をブラウザですぐに確認したいため

前者の「クラス毎にファイルを分割したいため」については、 ゲームプログラムは他のプログラムより、クラス数が多くなりがちだからです。 1つの js ファイルに全てのクラスを記述したり、あるいは分割した js を それぞれ `script` タグで一つ一つ読み込んでも良いのですが、コードの保守のしやすさのため、 クラス毎にファイルを分割することにし、それを `browserify` で結合することにします。 (`browserify` については後述します。)

後者の「コード修正をブラウザですぐに確認したいため」については、 プログラムを修正するたびに、ブラウザを開いてリロードボタンを押す手間を省くためです。 ゲーム開発は画面の配置調整など含めて、細かくコードを修正しては画面の確認を繰り返すことが多いです。 コードを修正するとブラウザを自動でリロードしてくれるタスクを導入して、この手間を減らします。

なお、以降の章では、特に `gulp` に依存した記述をしておりませんので、自分にとって得意な開発環境があるを作れる方は、そちらを使用していただいても構いません。 例えば、 `babel` であったり、 `webpack` を使用して頂いても、以降の章では差し支えありません。

ディレクトリ構成

ディレクトリ構成については下記のようにしたいと思います。

```
.
```

```
├── package.json
├── public
│   ├── bgm
│   ├── image
│   ├── js
│   │   ├── main.js
│   ├── index.html
│   └── sound
└── src
    └── js
        ├── asset_loader
        ├── logic
        ├── object
        ├── scene
        ├── game.js
        └── main.js
```

package.json
`npm init` により生成します。 `npm` についての説明は省略します。

public/bgm
ゲームで使用する BGM ファイルをここに置くことにします。

public/image
ゲームで使用する画像ファイルをここに置くことにします。

public/sound
ゲームで使用するSEファイル(効果音)をここに置くことにします。

public/js
`gulp` 及び `browserify` でビルドされた js ファイルをここに置くことにします。 ゲームは、この js ファイルを使用します。 `gulp` 及び `browserify` によって自動生成されるので、 このファイルを編集してはいけません。

public/index.html
ゲームを表示するためのHTMLファイルです。

src/js/logic
ゲームのソースコードのうち、画像や音声データの読み込みをするクラスをここに置くことにします。

src/js/logic
ゲームのソースコードのうち、ゲームロジックに値するクラスはここに置くことにします。

src/js/object
ゲームのソースコードのうち、オブジェクト(後述します)に値するクラスはここに置くことにします。

src/js/scene
ゲームのソースコードのうち、シーン(後述します)に値するクラスはここに置くことにします。

src/js/game.js
ゲームのソースコードのうち、ゲーム全体を表すクラスである Game クラスを記述するファイルです。

src/js/main.js
ゲームのソースコードのうち、Game クラスの生成及び実行をするファイルです。エントリポイントとも呼びます。

gulp

`gulp` はタスクランナーです。色々なタスクを自動化するために使います。 今回は、以下のタスクを自動化するために使用します。

- クラス毎に分割したファイルを結合するため
- コードの修正をするとブラウザをすぐリロードする

上記の2つのタスクは `browserify` と `browser-sync` モジュールを使用して実現します。

`gulp` の watch 機能 を使用することで、コードの修正にフックして、 自動でファイル結合と、ブラウザリロードを行うことにします。

browserify

`browserify` とは `require()` 関数を使うためのモジュールです。 `require()` とは、あるソースコード内から、他のソースコードを読み込むための関数です。node を利用したことがある人は、既に `require` 関数が用意されているので、使用されたことがある人も多いかもしれません。 `require()` 関数を使うことで、クラス毎に適切に js ファイルを分割することができ、最終的にそれらは一つの js ファイルになって出力されます。

browser-sync

`browser-sync` はファイル変更を監視し、自動でブラウザのリロードを行ってくれるツールです。 ローカルサーバーも起動してくれるので、ゲームの開発中は `browser-sync` を使用することで、 ゲームの動作確認の手間を大幅に減らすことができます。

gulpfileの作成

それでは、上記のタスクを利用するために必要な npm モジュールをインストールします。

```
npm init
npm -i browser-sync browserify gulp gulp-notify gulp-plumber gulp-r
ename run-sequence vinyl-source-stream
```

下記のgulpfileを用意します。

```
'use strict';
```

```
// ソース元の対象ファイル
var source_file = './src/js/main.js';
// 出力ディレクトリ
var dist_dir = './public/js/';
// アプリファイル
var appjs = 'main.js';

// gulp watch で開く html
var html_dir = "public";
var html = "index.html";

var browserify = require('browserify');
var gulp       = require('gulp');
var source     = require('vinyl-source-stream');
var rename     = require('gulp-rename');
var plumber    = require('gulp-plumber');
var runSequence= require('run-sequence');
var path       = require('path');
var notify     = require('gulp-notify');
var browserSync= require('browser-sync').create();

gulp.task('browserify', function() {
    return browserify(source_file)
        .bundle()
        .on('error', function(err){   //ここからエラーだった時の記述
            // デスクトップ通知
            var error_handle = notify.onError('<%= error.message %>
');
            error_handle(err);
            this.emit('end');
        })
        .pipe(source(appjs))
        .pipe(gulp.dest(dist_dir));
});

gulp.task('reload', function () {
    return browserSync.reload();
```

```
});

gulp.task('build', function(callback) {
    return runSequence(
        'browserify',
        'reload',
        callback
    );
});

gulp.task('browser-sync', function() {
    return browserSync.init({
        server: {
            baseDir: html_dir,
            index: html,
        }
    });
});

gulp.task('watch', ['browser-sync'], function() {
    gulp.watch('src/js/**/*.js', ['build']);
});
```

これで `gulp watch` をコマンドライン上で実行することで、gulp の監視タスクが立ち上がります。 ソースコードの変更があると、自動でファイルの結合とブラウザのリロードを行ってくれます。

また、 `browser-sync` が Web アプリケーションサーバーも立ち上げてくれるため、 `http://localhost:3000/` にアクセスすると、ゲームにアクセスすることができます。

ゲームループ

Webエンジニアが初めてゲームを作るときにまず戸惑うのが「ゲームループ」という考え方だと思います。 本稿では、ゲーム画面上で、キャラクターの動きがどのようにして更新されていくのかを、「ゲームループ」という考え方でご説明させていただきます。

FPS(frame per second)という言葉を聞いたことがありますでしょうか。「60FPS」「30FPS」という言葉を聞いたことがあるかと思います。60FPSという言葉について解説します。「60FPS」のゲームでは、1秒間に60回、画面が更新されています。つまり、1/60秒ごとに少しずつ違う画面を計算・描画することで、私たちの目にゲームが「動画」として映っているのです。下記のコードを見てみてください。

```
function update() {
    // ゲーム更新
}

function run() {
    update();

    requestAnimationFrame(run);
}

run();
```

`requestAnimationFrame` メソッドは、ブラウザの次の再描画の直前に、指定した関数を呼び出すようにブラウザに要求します。 ブラウザの再描画は基本的に1秒に60回行われるため、上記のコードを書くと、1秒間に60回、update 関数が呼び出されます。

この update 関数の中で、ゲームの描画であったり、描画に必要な情報の更新を行います。

処理落ちとフレーム落ち

ゲームループは2種類の実装方法があります。これは、処理に時間がかかった際に「処理落ち」させるか「フレーム落ち」させるかの違いです。

1秒間に60回処理する場合、最大16.666ms以内に1処理を終わらせないといけません。 それ以上時間、処理に時間をかけた場合に、「処理を遅延させるか」「描画をスキップするか」が 処理落ちとフレーム落ちの違いです。

「処理落ち」が発生すると、画面の動きがすごくゆっくりになります。画面の動きはゆっくりになりますが、描画はスキップしません。 弾幕シューティングゲームなどは描画がスキップされると、プレイヤーが理解できないまま自機が死んでしまいます。描画がスキップされると困る場合には、処理落ちする方のゲームループを使用します。先ほどのサンプルコードのゲームループは、処理落ちするゲームループです。

一方「フレーム落ち」が発生すると、画面の動きがいきなり飛んだように見えます。処理自体は時間経過どおりに進みます。 そのフレームの描画を諦めて次のフレームの描画に移るため、画面の動きがカクカクっと飛んだ感じになります。 ただ実際の時刻との同期は守られるため、動き自体が遅くなることはありません。

処理落ちとフレーム落ちでは、フレーム落ちの方が実装がちょっと面倒くさいです。 フレーム落ちさせる場合のコードは以下となります。

```
var deltaTime = 1.0 / 60.0;
var currentTime;
var accumulator;

function update() {
    // ゲーム更新
}

function run() {
    var newTime = performance.now();
    var fTime = (newTime - currentTime) / 1000;
```

```
    // 追いつくための実行回数があまりにも増えないようにする
    if (fTime > 0.25) fTime = 0.25;
    var currentTime = newTime;
    accumulator += fTime;

    // 実行回数を増やして処理を追いつかせる
    while (accumulator >= deltaTime) {
        update();
        accumulator -= deltaTime;
    }
    requestAnimationFrame(run);
}

run();
```

描画毎に、前回の実行時との経過時間を計算し、1/60 ms 以内に処理が収まってなければ、 次の実行時に複数回実行することで、処理を追いつかせています。

CanvasAPIを使ってみる

それでは、Canvas API を使って、試しに描画をしてみましょう。
`public/index.html`

```
<html>
    <canvas id="mainCanvas" width="640" height="480"></canvas>
</html>
```

index.html に `canvas` タグを追加します。ここにゲーム内容を描画します。 `canvas` タグの id 属性は `mainCanvas` です。あとで JavaScript 側から Canvas の DOM を取得する際に使用します。

先ほどのコードでcanvas タグに文字を表示してみたいと思います。

```
var mainCanvas = document.getElementById('mainCanvas');
var context = mainCanvas.getContext('2d');
// ゲーム更新
function update() {
    // 画面をクリア
    context.clearRect(0, 0, mainCanvas.width, mainCanvas.height);
    context.font = "18px 'MS ゴシック'";
    context.fillStyle = 'rgb( 255, 255, 255 )'; // 黒

    // 文字を描画
    context.fillText("Hello World!", mainCanvas.width/2, mainCanvas
.height);

}

function run() {
    update();

    requestAnimationFrame(run);
}

run();
```

画面中央に「Hello World!」という文字列が表示されたかと思います。

終わりに

それでは実際のゲームを作っていきたいと思います。まずゲームクラスを作っておきます。 ゲームクラスに、ゲームループに必要な処理を実装することにします。

`src/game.js

```
var Game = function (canvas) {
    this.ctx = canvas.getContext('2d'); // Canvas への描画は、ctx プ
ロパティを通して行うこととする。
```

```
    // 画面サイズ
    this.height = canvas.height;
    this.width = canvas.width;

};
Game.prototype.startRun = function () {

    this.run();
};

Game.prototype.run = function () {
    // ここにゲームの処理を書く

    this.request_id = requestAnimationFrame(this.run.bind(this));
};
module.exports = Game;
```

`src/main.js` はゲームのエントリポイント(最初の開始地点)です。 ゲームクラスのインスタンスを生成し、ゲームを起動します。

`src/main.js`

```
var Game = require("game");
var mainCanvas = document.getElementById('mainCanvas');
var game = new Game(mainCanvas);
game.startRun();
```

シーンとオブジェクト

概要

ゲームのプログラムを書いていく上で重要な概念として、シーン(Scene)とオブジェクト(Object)があります。オブジェクトはアクター(Actor)と呼ばれる場合があるかもしれません。また、シーンもオブジェクトの1種として扱うゲームエンジンもあります。

シーンは、Webサイトでいうところのページです。画面1つを表します。例えば、タイトルシーン、ステージシーン、エンディングシーンなどがあります。シーンは遷移します。タイトルシーンの次は、ステージ選択シーンに変わるでしょうし、ステージシーンでゲームオーバーになれば、ゲームオーバーシーンに変わります。

オブジェクトは Web サイトでいうところの DOM です。例えばシューティングゲームであれば、敵オブジェクト、自機オブジェクト、弾オブジェクト、文字オブジェクトなどがあります。シーンとオブジェクトの仕組みを採用したゲームエンジンでは、シーン上に存在する全ての要素を、オブジェクトとして扱うことになります。そしてオブジェクトには親子関係があります。シューティングゲームで言えば、自機と自機オプションは、親子関係になるでしょう。

シーンとオブジェクトは、シューティングゲームでもアクションゲームでも通用する概念です。また2Dでも3Dでも通用するゲームの設計において普遍的な概念です。

シーン

それでは、実際にシーンを作ってみましょう。まずはステージシーンを作ろうと思います。

`src/scene/stage.js`

```
var StageScene = function (game) {
    this.game = game;

    // 経過フレーム数
    this.frame_count = 0;
};
// 更新
StageScene.prototype.update = function () {
    this.frame_count++;
};
// 描画
StageScene.prototype.draw = function () {

};
module.exports = StageScene;
```

`StageScene` クラスを作りました。 `StageScene` クラスはインスタンスメソッドとして、 `update` と `draw` メソッドを持っています。

`update` と `draw` メソッドは、1秒間に 60 回呼び出すこととします。 `update` メソッドはゲームの状態の更新を担当し、 `draw` はその後に状態を元に描画することを担当することにします。

`update` 関数の中では、経過フレーム数を数えています。ゲームの種類にもよりますが、 シューティングゲームなどでは、敵の登場タイミングを計る際などに経過フレーム数を使用するので、 経過フレーム数を数えておくと、他の実装でも使うことができます。

コンストラクタの引数に `game` 引数を取っています。 `game` は `Game` クラスのインスタンスを渡すことにします。

シーン切り替え

シーンを `Game` クラス側から呼び出したいと思います。

`src/game.js`

```
var StageScene = require("scene/stage");

var Game = function (canvas) {
    this.ctx = canvas.getContext('2d'); // Canvas への描画は、ctx プ
ロパティを通して行うこととする。
    // 画面サイズ
    this.height = canvas.height;
    this.width = canvas.width;

    this.scenes = {}; // シーン一覧
    this.next_scene = null;
    this.current_scene = null;
    this.prev_scene = null;

    this.addScene("stage", new StageScene(this)); // シーンを追加
    this.changeScene("stage"); // 最初のシーンに切り替え
};
Game.prototype.run = function () {
    this.toNextSceneIfExists(); // 次のシーンが前フレームにて予約され
ていればそちらに切り替え

    this.scenes[this.current_scene].update();
    this.scenes[this.current_scene].draw();

    this.request_id = requestAnimationFrame(this.run.bind(this));
};
Game.prototype.toNextSceneIfExists = function () {
    if(!this.next_scene) return;

    this.current_scene = this.next_scene;
    this.next_scene = null;
};

// ゲームで使用するシーンを追加
Game.prototype.addScene = function (name, scene) {
    this.scenes[name] = scene;
};
```

```
// シーン切り替え
Game.prototype.changeScene = function (name) {
    this.next_scene = name;
};
// 前のシーンに戻る
Game.prototype.changePrevScene = function () {
    if(!this.prev_scene) return;

    this.current_scene = this.prev_scene;
    this.prev_scene = null;
};
```

`Game` クラスのコンストラクタ及び `run` 関数を修正しております。また、 `addScene` , `changeScene` , `toNextSceneIfExists` , `changePrevScene` メソッドを追加しております。一つずつ解説したいと思います。

```
var Game = function (canvas) {
    this.ctx = canvas.getContext('2d'); // Canvas への描画は、ctx プロパティを通して行うこととする。
    // 画面サイズ
    this.height = canvas.height;
    this.width = canvas.width;

    this.scenes = {}; // シーン一覧
    this.next_scene = null;
    this.current_scene = null;
    this.prev_scene = null;

    this.addScene("stage", new StageScene(this)); // シーンを追加
    this.changeScene("stage"); // 最初のシーンに切り替え
};
```

`Game` クラスのコンストラクタです。大きな変更は2点です。

まずシーン管理用に `scenes` , `next_scene` , `current_scene` , `prev_scene` といったプロパティの追加をしています。

`scenes` には、ゲームで使用するシーン一覧を追加します。後述しますが、追加は `addScene` メソッドを使って行います。 基本的に、 `addScene` は、動的にシーンを生成して追加する要件がないかぎり、 ゲームを初期化する際に、全てのシーンを追加することとなるかと思います。

`next_scene` , `current_scene` , `prev_scene` はそれぞれ、「次に遷移するシーン」「現在のシーン」「前のシーン」です。 後述しますが、シーンを切り替える際は、切り替えを実行して、すぐに切り替えるのではなく、次のフレームの最初にシーン切り替えを行います。そのため、切り替え先のシーンを一旦保存するための `next_scene` が必要になります。また前のシーンに戻りたい場合がゲームではあります。例えばステージシーンで、メニューを開いて、メニューを閉じたらまた元のステージに戻る場合などです。そういった用途のために、前のシーンを保存する `prev_scene` を使用します。

```
Game.prototype.run = function () {
    this.toNextSceneIfExists(); // 次のシーンが前フレームにて予約されていればそちらに切り替え

    this.scenes[this.current_scene].update();
    this.scenes[this.current_scene].draw();

    this.request_id = requestAnimationFrame(this.run.bind(this));
};
```

`run` メソッドは、requestAnimationFrame 関数により、ブラウザの描画タイミング毎(1秒に60回)呼ばれます。 `run` メソッド内にて、 `toNextSceneIfExists` メソッドを呼び出しています。先述しましたが、 `toNextSceneIfExists` 関数は、 `next_scene` プロパティに、切り替え先のシーンが予約されていれば、そのシーンに切り替えを行います。

また、現在のシーンの `update` と `draw` メソッドを呼び出しています。 `run` メソッドが1秒に60回呼ばれるので、 シーンの `update` と `draw` も1秒に60回呼ばれるわけです。

```
Game.prototype.toNextSceneIfExists = function () {
    if(!this.next_scene) return;

    this.current_scene = this.next_scene;
    this.next_scene = null;
};
```

`addScene`, `changeScene` メソッドについては以下の通りです。 `name` にはシーン名を、 `scene` にはシーンのインスタンスを渡します。

```
Game.prototype.addScene = function (name, scene) {
    this.scenes[name] = scene;
};
```

```
Game.prototype.changeScene = function (name) {
    this.next_scene = name;
};
```

また、 `changePrevScene` メソッドも追加しておきます。前のシーンに戻る関数です。

```
Game.prototype.changePrevScene = function () {
    if(!this.prev_scene) return;

    this.current_scene = this.prev_scene;
    this.prev_scene = null;
};
```

なお今回は省略しましたが、これらのシーン管理は、Game クラスに実装せず、 `SceneManager` のような専用のクラスを用意して、そこに `addScene`, `changeScene` といったメソッドをまとめる場合も多いです。

オブジェクト

オブジェクトを作っていきましょう。自機オブジェクトを作成します。

`src/object/chara.js`

```
var Chara = function (scene) {
    this.scene = scene;
    this.game = game;

    this._id = "chara";

    this.x = 0;
    this.y = 0;

    // 経過フレーム数
    this.frame_count = 0;
};
Chara.prototype.id = function () {
    return this._id;
};
// 更新
Chara.prototype.update = function () {
    this.frame_count++;
};
// 描画
Chara.prototype.draw = function () {

};
module.exports = Chara;
```

Chara クラスを作りました。Chara クラスはインスタンスメソッドとして、 `update` と `draw` メソッドを持っています。また、パブリックなアクセサとして、 `id` を持っています。

`Scene` クラスと同じように、 `update` と `draw` メソッドは、1秒間に 60 回呼び出すこととします。 `update` メソッドはゲームの状態の更新を担当し、 `draw` はその後に状態を元に描画することを担当することにします。

`id` はゲーム内でオブジェクトを一意に識別するIDです。例えば、敵の弾などは画面上に同じ種類の弾が複数存在するので、 それらを個別に識別するIDがあると便利です。今回、キャラは画面上に1機だけなので、 `"chara"` 固定にしたいと思います。

コンストラクタの引数に `scene` 引数を取っています。 `scene` は シーンクラスのインスタンスを渡すことにします。

シーンにオブジェクトの追加

ステージシーンに、先ほど追加したキャラクラスのインスタンスを追加したいと思います。

`src/scene/stage.js`

```
var Chara = require("../object/chara");
var StageScene = function (game) {
    this.game = game;

    // シーン上のオブジェクト一覧
    this.objects = {};

    // 経過フレーム数
    this.frame_count = 0;

    this.addObject(new Chara(this));
};
StageScene.prototype.addObject = function (object) {
    this.objects[object.id()] = object;
};
// 更新
StageScene.prototype.update = function () {
    this.frame_count++;

    this.updateObjects();
};
StageScene.prototype.updateObjects = function () {
```

```
    for (var id in this.objects) {
        this.objects[id].update();
    }
};

// 描画
StageScene.prototype.draw = function () {
    this.drawObjects();

};
StageScene.prototype.drawObjects = function () {
    for (var id in this.objects) {
        this.objects[id].draw();
    }
};

module.exports = StageScene;
```

修正点をピックアップして解説していきます。

```
StageScene.prototype.addObject = function (object) {
    this.objects[object.id()] = object;
};
```

`addObject` メソッドを追加して、ステージシーンにキャラインスタンスを追加しています。 追加されたキャラインスタンスは、他のオブジェクトと一緒に、 `objects` プロパティ内で管理されます。 この辺りは、 `Game` クラスに、シーン管理を追加したときと同じですね。

```
// 更新
StageScene.prototype.update = function () {
    this.frame_count++;

    this.updateObjects();
};
StageScene.prototype.updateObjects = function () {
```

```
    for (var id in this.objects) {
        this.objects[id].update();
    }
};
```

シーンの `update` を実行すると、シーンが管理している各オブジェクトの `update` も実行されるようにします。

```
// 描画
StageScene.prototype.draw = function () {
    this.drawObjects();

};
StageScene.prototype.drawObjects = function () {
    for (var id in this.objects) {
        this.objects[id].draw();
    }
};
```

同様に、シーンの `draw` を実行すると、シーンが管理する各オブジェクトの `draw` も実行されるようにします。

描画

ここまでで、シーンとオブジェクトをゲームに追加しました。シーンやオブジェクトを追加しましたが、まだ描画を行っていないため、画面には何も表示されていません。いよいよこれらのシーンやオブジェクトを描画していきたいと思います。

画像の読み込み

まずは、画像をゲームに読み込む処理をしたいと思います。JavaScript は非同期に処理される言語なので、 画像を読み込みしている最中も処理が進行します。画像の読み込みをしている間は、 ゲームを進行させない(=ローディング画面を表示する)ようにしないと、画像が表示されないまま ゲームが進行してしまいます。 `ImageLoader` クラスを作成し、画像の読み込み処理や、どこまで画像を読み込んだかを管理することにします。

`src/asset_loader/image.js`

```
var ImageLoader = function(game) {
    this.images = {};

    this.loading_image_num = 0;
    this.loaded_image_num = 0;
};
ImageLoader.prototype.loadImage = function(name, path) {
    var self = this;

    self.loading_image_num++;

    // it's done to load image
    var onload_function = function() {
        self.loaded_image_num++;
    };
```

```
    var image = new Image();
    image.src = path;
    image.onload = onload_function;
    this.images[name] = image;
};

// 画像が全て読み込まれたかどうか
ImageLoader.prototype.isAllLoaded = function() {
    return this.loaded_image_num > 0 && this.loaded_image_num === t
his.loading_image_num;
};

// 画像データの取得
ImageLoader.prototype.get = function(name) {
    return this.images[name];
};

// 画像データをメモリから解放
ImageLoader.prototype.remove = function(name) {
    delete this.images[name];
};

module.exports = ImageLoader;
```

それでは `ImageLoader` クラスを、一つずつ解説していきたいと思います。

```
var ImageLoader = function(game) {
    this.images = {};

    this.loading_image_num = 0;
    this.loaded_image_num = 0;
};
```

`ImageLoader` クラスのコンストラクタです。読み込んだ画像データの一覧を保持する `images` プロパティ、 読み込み中の画像の数である `loading_image_num` , 読み込んだ画像数である `loaded_image_num` を持ちます。

```
ImageLoader.prototype.loadImage = function(name, path) {
    var self = this;

    self.loading_image_num++;

    // it's done to load image
    var onload_function = function() {
        self.loaded_image_num++;
    };

    var image = new Image();
    image.src = path;
    image.onload = onload_function;
    this.images[name] = image;
};
```

`loadImage` メソッドを使用すると、画像の読み込みを開始します。 `name` はプログラム内での画像の識別子(名前)です。 `path` に画像のパスを指定します。非同期に読み込みを行うので、画像の読み込みを待たずに、関数は返ります。

```
ImageLoader.prototype.isAllLoaded = function() {
    return this.loaded_image_num > 0 && this.loaded_image_num === t
his.loading_image_num;
};
```

`isAllLoaded` メソッドは画像の読み込みが全て完了したことをチェックするメソッドです。

`ImageLoader` クラスを使って、画像を読み込みする際は、読み込みたい画像の数だけ `loadImage` メソッドを呼び出し、 `Game` クラスの `run` メソッドなどで、定期的に、 `isAllLoaded` メソッドが `true` であることを確認する必要があります。

もし、 `isAllLoaded` メソッドが `false` ならば、ローディング画面を表示する仕組みにしておくと、 のちのちプログラムを組み際に、画像の読み込みが必要な場合に、 `loadImage` メソッドを呼ぶだけで済むので楽になります。

```
ImageLoader.prototype.get = function(name) {
    return this.images[name];
};
```

`get` メソッドは、 `loadImage` メソッドで既に読み込みが完了した画像データを取得するメソッドです。 取得したデータは、 `Image` インスタンスなので、後述しますが Canvas の `drawImage` メソッドに そのまま引数として渡すことで、画面上に画像を描画することができます。

```
ImageLoader.prototype.remove = function(name) {
    delete this.images[name];
};
```

`remove` メソッドは、読み込んだ画像を破棄するメソッドです。小さなゲームであれば、画像を破棄せずに、 メモリ上に乗せ続けていても、問題にはなりにくいです。 しかし大きな規模のゲームになると、画像を大量に使うため、必要のなくなった画像は破棄しなくては、 メモリを大量に消費してしまい、メモリが足りなくてゲームが終了してしまうことが発生してしまいます。

スプライト

スプライトとは、複数の画像データを一つの画像ファイルにまとめたものです。同じ画像データでも、複数の画像ファイルをそれぞれ読み込むのと、複数の画像データをまとめた1つのファイルで読み込んで、プログラム内で画像を分割して使うのとでは、後者の方が読み込み速度が早くなります。そのためゲームでは、キャラのアニメーションなどに使用する画像データや、敵の弾の種類は、一つの画像データにまとめることが多いです。

スプライトの使用方法ですが、プログラムでスプライト画像を読み込み、スプライト画像の指定の領域のみを切り取って表示することで使用します。

それでは、 `Chara` クラスにスプライトを使用して、ゲーム上にキャラ画像を表示するようにしてみましょう。 いくつかのスプライト画像の表示に使う定数の追加と、 `draw` メソッドの変更を行います。

スプライト画像

`public/image/chara.png`

キャラのスプライト画像は、本書では用意できなかったので、 読者の皆様の方でご用意頂けましたらと思います。

フリー素材としては、例えば東方弾幕風向けの素材として、以下があったりします。

http://danmakufu.wiki.fc2.com/wiki/%E7%B4%A0%E6%9D%90%E3%83%AA%E3%83%B3%E3%82%AF

星蓮船自機ドット絵 霊夢・魔理沙・早苗の星蓮船風ドット絵 等

`src/object/chara.js`

```
// 描画
Chara.prototype.draw = function () {
    var image = this.core.image_loader.getImage(this.spriteName());

    var ctx = this.core.ctx;
```

```
    ctx.save();

    // set position
    ctx.translate(this.x, this.y);

    var sprite_width  = this.spriteWidth();
    var sprite_height = this.spriteHeight();

    ctx.drawImage(image,
        // sprite position
        sprite_width * this.spriteIndexX(), sprite_height * this.sp
riteIndexY(),
        // sprite size to get
        sprite_width,                       sprite_height,
        // adjust left x, up y because of x and y indicate sprite c
enter.
        -sprite_width/2,                    -sprite_height/2,
        // sprite size to show
        sprite_width,                       sprite_height
    );

    ctx.restore();
};

Chara.prototype.spriteName = function(){
    return "chara";
};
Chara.prototype.spriteIndexX = function(){
    return 0;
};
Chara.prototype.spriteIndexY = function(){
    return 0;
};
Chara.prototype.spriteWidth = function(){
    return 64;
};
Chara.prototype.spriteHeight = function(){
    return 64;
```

```
};
module.exports = Chara;
```

先ほど紹介した Chara クラスを改良しました。それではコードを追っていきましょう。

```
// 描画
Chara.prototype.draw = function () {
    var image = this.core.image_loader.getImage(this.spriteName());

    var ctx = this.core.ctx;

    ctx.save();

    // set position
    ctx.translate(this.x, this.y);

    var sprite_width  = this.spriteWidth();
    var sprite_height = this.spriteHeight();

    ctx.drawImage(
        image,
        // sprite position
        sprite_width * this.spriteIndexX(), sprite_height * this.sp
riteIndexY(),
        // sprite size to get
        sprite_width,                       sprite_height,
        // adjust left x, up y because of x and y indicate sprite c
enter.
        -sprite_width/2,                    -sprite_height/2,
        // sprite size to show
        sprite_width,                       sprite_height
    );

    ctx.restore();
};
```

`draw` 関数です。オブジェクトから Canvas の API を利用するには、 `this.core.ctx` を参照します。 `draw` メソッドの中身を解説したいと思います。

```
var image = this.core.image_loader.getImage(this.spriteName());
```

先ほど解説した、 `ImageLoader` クラスの `getImage` メソッドを使用して、読み込んだ画像のデータを取得しています。画像のデータは、 `Image` インスタンスです。

```
ctx.translate(this.x, this.y);
```

Canvas 上のどこに画像を描画するかを、(x, y)で指定しています。Canvas API では、(x, y) 座標の開始地点は、左上になります。 画面の一番左が、x = 0 で、画面の一番上が、y = 0 です。

```
ctx.drawImage(
    image,
    // sprite position
    0,                                  0,
    // sprite size to get
    sprite_width,                       sprite_height,
    // adjust left x, up y because of x and y indicate sprite cente
r.
    -sprite_width/2,                    -sprite_height/2,
    // sprite size to show
    sprite_width,                       sprite_height
);
```

取得した画像を `drawImage` 関数を利用して Canvas に描画しています。 `drawImage` 関数は、引数に指定した数によって挙動が変わる関数です。以下の3種類の引数の取り方があります。

- drawImage(image, dx, dy)

- drawImage(image, dx, dy, dw, dh)
- drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)

今回は引数が9つの使い方をします。

第1引数の image には、描画したい画像データを指定します。

第2引数・第3引数には、画像のどこからを描画したいかを x, y 座標で指定します。 画像の左上が(0, 0)となるので、今回は (0, 0)を指定することにします。

第4引数,第5引数には、Canvas 上のどこに画像を描画するか、x, y 座標で指定します。 `translate` で指定した描画位置から、相対的に描画位置をずらします。 これは、 `this.x` と `this.y` が指定する画像の位置について、基準を画像の真ん中にするためです。

`translate` で描画位置を指定し、さらに `drawImage` で描画位置をずらすのは二度手間に見えるかもしれませんが、 のちのち、画像の回転を行う上で大切になってきます。また画像の回転の項目で後述します。

```
Chara.prototype.spriteName = function(){
    return "chara";
};
Chara.prototype.spriteWidth = function(){
    return 64;
};
Chara.prototype.spriteHeight = function(){
    return 64;
};
```

`spriteName` は読み込んだ画像名を指定する定数です。 `spriteWidth` `spriteHeight` には、 読み込んだ画像のどこまでを切り取るか、横幅と高さを指定します。

スプライトアニメーション

先ほどの項では、スプライト画像から画像データを一部切り取って表示することを行いました。 これにより自機のキャラ画像が表示されたことかと思います。

さらに自機の画像をアニメーションさせてみたいと思います。

アニメーションの方法は、パラパラ漫画と同じように、 N秒毎に、表示するスプライトを変更することで行います。

それでは、自機のスプライト画像2種類を 10 フレーム毎に切り替えて、 アニメーションさせてみましょう。 `Chara` クラスを変更します。

`src/object/chara.js`

```
var Chara = function (scene) {
    this.scene = scene;
    this.game = game;

    this._id = "chara";

    this.x = 0;
    this.y = 0;

    // 経過フレーム数
    this.frame_count = 0;

    // 現在表示するスプライト
    this.current_sprite_index = 0;
};
Chara.prototype.id = function () {
    return this._id;
};
// 更新
Chara.prototype.update = function () {
    this.frame_count++;

    // animation sprite
    if(this.frame_count % this.spriteAnimationSpan() === 0) {
```

```
        this.current_sprite_index++;
        if(this.current_sprite_index >= this.spriteIndices().length
) {
            this.current_sprite_index = 0;
        }
    }

};
// 描画
Chara.prototype.draw = function () {
    var image = this.core.image_loader.getImage(this.spriteName());

    var ctx = this.core.ctx;

    ctx.save();

    // set position
    ctx.translate(this.x, this.y);

    var sprite_width  = this.spriteWidth();
    var sprite_height = this.spriteHeight();

    ctx.drawImage(
        image,
        // sprite position
        sprite_width * this.spriteIndexX(), sprite_height * this.sp
riteIndexY(),
        // sprite size to get
        sprite_width,                       sprite_height,
        // adjust left x, up y because of x and y indicate sprite c
enter.
        -sprite_width/2,                    -sprite_height/2,
        // sprite size to show
        sprite_width,                       sprite_height
    );
    ctx.restore();
};
```

```
Chara.prototype.spriteName = function(){
    return "sanae";
};
Chara.prototype.spriteIndexX = function(){
    return this.spriteIndices()[this.current_sprite_index].x;
};
Chara.prototype.spriteIndexY = function(){
    return this.spriteIndices()[this.current_sprite_index].y;
};
Chara.prototype.spriteAnimationSpan = function(){
    return 10;
};
Chara.prototype.spriteIndices = function(){
    return [{x: 0, y: 0},{x: 1, y: 0}];
};
Chara.prototype.spriteWidth = function(){
    return 64;
};
Chara.prototype.spriteHeight = function(){
    return 64;
};
module.exports = Chara;
```

あらためて、Chara クラスを修正して全て掲載しました。それでは少しずつ解説していきます。

```
var Chara = function (scene) {
    this.scene = scene;
    this.game = game;

    this._id = "chara";

    this.x = 0;
    this.y = 0;

    // 経過フレーム数
    this.frame_count = 0;
```

```
    // 現在表示するスプライト
    this.current_sprite_index = 0;
};
```

`this.current_sprite_index` というプロパティを追加しました。これは、現在表示するスプライトがどれかを表します。今回は、スプライト画像が2種類なので、0 か 1 の値を取り、 また、10フレームごとに変更します。

```
Chara.prototype.spriteAnimationSpan = function(){
    return 10;
};
```

`spriteAnimationSpan` はスプライトを切り替える間隔を定義しています。10 フレームごとに切り替えるので、10を指定しました。

```
Chara.prototype.spriteIndices = function(){
    return [{x: 0, y: 0},{x: 1, y: 0}];
};
```

`spriteIndices` にスプライト画像の位置を指定しています。今回は2種類のスプライト画像を使用するので、 左から0番目、上から0番目のスプライト画像を1つと、左から1番目、上から0番目のスプライト画像を表示するので、 `[{x: 0, y: 0},{x: 1, y: 0}]` を指定しました。

```
Chara.prototype.spriteIndexX = function(){
    return this.spriteIndices()[this.current_sprite_index].x;
};
Chara.prototype.spriteIndexY = function(){
    return this.spriteIndices()[this.current_sprite_index].y;
};
```

`current_sprite_index` と `spriteIndices` を元に、スプライト画像のどの座標を切り取ればいいのかを返します。

```
// 描画
Chara.prototype.draw = function () {
    var image = this.core.image_loader.getImage(this.spriteName());

    var ctx = this.core.ctx;

    ctx.save();

    // set position
    ctx.translate(this.x, this.y);

    var sprite_width  = this.spriteWidth();
    var sprite_height = this.spriteHeight();

    ctx.drawImage(
        image,
        // sprite position
        sprite_width * this.spriteIndexX(), sprite_height * this.sp
riteIndexY(),
        // sprite size to get
        sprite_width,                       sprite_height,
        // adjust left x, up y because of x and y indicate sprite c
enter.
        -sprite_width/2,                    -sprite_height/2,
        // sprite size to show
        sprite_width,                       sprite_height
    );
    ctx.restore();
};
```

drawImage の第2引数と第3引数が変更になっております。 `spriteIndexX` と `spriteIndexY` を元に、 スプライト画像のどこを切り取るかを指定しています。この `spriteIndexX` と `spriteIndexY` は 先述した `this.current_sprite_index` を元に決定されます。

```
// 更新
```

```
Chara.prototype.update = function () {
    this.frame_count++;

    // animation sprite
    if(this.frame_count % this.spriteAnimationSpan() === 0) {
        this.current_sprite_index++;
        if(this.current_sprite_index >= this.spriteIndices().length
) {
            this.current_sprite_index = 0;
        }
    }

};
```

`update` メソッド内には表示する画像データを切り替える処理を追加しています。 `spriteAnimationSpan` で定義されたフレーム数ごとに、 `current_sprite_index` を +1 しています。 また、表示できるスプライトがなくなれば、また最初のスプライトに戻ります。

これで、自機キャラがアニメーションするようになります。

入力

前項までで、ゲームにシーンとオブジェクトを追加し、その中で画像を描画するところまで解説しました。 このままでは、画面は表示されていますが、プレイヤーがゲームに干渉することはできません。

本項では、プレイヤーがキーボードあるいはゲームパッド(ゲーム用のコントローラーのこと)で操作した内容をゲーム側で受け取る方法について解説します。

キーボード入力

まず、ユーザーのキーボード入力を受け取るクラスを作りたいと思います。

`src/constant.js`

```
var Constant = {
    BUTTON_LEFT:  0x01, // 0b00000001
    BUTTON_UP:    0x02, // 0b00000010
    BUTTON_RIGHT: 0x04, // 0b00000100
    BUTTON_DOWN:  0x08, // 0b00001000
    BUTTON_Z:     0x10, // 0b00010000
    BUTTON_X:     0x20, // 0b00100000
};
module.exports = Constant;
```

プレイヤーのキー入力は、定数を使ってプログラムから扱いたいと思います。後述しますが、どのキーが入力されたかどうかの情報はビットで管理することにします。 これは以下のメリットのためです。

- ビット演算子の方が通常の算術より高速
- 一つの変数で、複数のキー押下を管理することができる。

ES5では、2進数による値表記ができないため、16進数で定数を定義しています。 2進数にすると、どういう値になるかは、コメントに記載しました。

ご覧の通り、キー一つにつき、1bitだけ立てており、キーに対応するビットが立っていれば、押下されていると判定するようにしたいと思います。

src/input.js

```
var Consntant = require("./constant");
var Input = function() {
    // キー押下フラグ
    this.keyflag = 0x0;

    // 一つ前のフレームで押下されたキー
    this.before_keyflag = 0x0;
};
// キー押下
Input.prototype.handleKeyDown = function(e){
    this.keyflag |= this._keyCodeToBitCode(e.keyCode);
    e.preventDefault();
};
// キー押下解除
Input.prototype.handleKeyUp = function(e){
    this.keyflag &= ~this._keyCodeToBitCode(e.keyCode);
    e.preventDefault();
};
// 指定のキーが押下状態か確認する
Input.prototype.isKeyDown = function(flag){
    return this.keyflag & flag;
};
// 指定のキーが押下されたか確認する
Input.prototype.isKeyPush = function(flag){
    // 1フレーム前に押下されておらず、現フレームで押下されてるなら true
    return !(this.before_keyflag & flag) && this.keyflag & flag;
};
// キーコードをBitに変換
Input.prototype._keyCodeToBitCode = function(keyCode){
```

```
    var flag;
    switch(keyCode) {
        case 16: // shift
            flag = Constant.BUTTON_SHIFT;
            break;
        case 32: // space
            flag = Constant.BUTTON_SPACE;
            break;
        case 37: // left
            flag = Constant.BUTTON_LEFT;
            break;
        case 38: // up
            flag = Constant.BUTTON_UP;
            break;
        case 39: // right
            flag = Constant.BUTTON_RIGHT;
            break;
        case 40: // down
            flag = Constant.BUTTON_DOWN;
            break;
        case 88: // x
            flag = Constant.BUTTON_X;
            break;
        case 90: // z
            flag = Constant.BUTTON_Z;
            break;
    }
    return flag;
};
Input.prototype.bindKey = function(){
    var self = this;
    window.onkeydown = function(e) { self.handleKeyDown(e); };
    window.onkeyup   = function(e) { self.handleKeyUp(e); };
};
Input.prototype.saveBeforeKey = function(){
    this.before_keyflag = this.keyflag;
    this.keyflag = 0x0;
}
```

src/game.js

```
var Game = function (canvas) {
    this.input = new Input();
    this.input.bindKey();

    /* ~ 以下省略 ~ */
};
Game.prototype.run = function() {
    /* ~ 以上省略 ~ */

    // 押下されたキーを保存しておく
    this.input.saveBeforeKey();
};
```

`Input` クラスを作りました。また `Game` クラスから `Input` クラスを呼び出しています。 それでは少しずつ解説していきます。

```
var Input = function() {
    // キー押下フラグ
    this.keyflag = 0x0;

    // 一つ前のフレームで押下されたキー
    this.before_keyflag = 0x0;
};
```

`Input` クラスのコンストラクタです。プロパティとして、押下されたキーを保存する `keyflag` と、1フレーム前に押下されたキーを保存する `before_keyflag` を持ちます。先述の通り、 `keyflag` は 8bit の値として使い、各 bit 毎に bit が立っているかどうかをチェックすることで、どのキーが押下されているかを判定します。

`before_keyflag` には、1フレーム前に押下されたキーを保存します。なぜこの変数が必要なのかを解説します。キーの押下判定を、JavaScript で取得するには、 `onkeyup` と `onkeydown` イベントハンドラを利用するのですが、 `onkeydown` は、キーが押下さえされていれば発火されます。この性質は、ボタンが押しっぱなしであるかどうかを判定する分には、便利なのです。一方ボタンが押下された時を取得する場合、現在のフレームでキーが押下されたことを確認し、1フレーム前にキーが押下されていないことを確認しなければいけません。そのために、1フレーム前のキー入力の情報を保存する `before_keyflag` を用意したわけです。

```
Input.prototype.bindKey = function(){
    var self = this;
    window.onkeydown = function(e) { self.handleKeyDown(e); };
    window.onkeyup   = function(e) { self.handleKeyUp(e); };
};
Input.prototype.saveBeforeKey = function(){
    this.before_keyflag = this.keyflag;
    this.keyflag = 0x0;
}
```

`bindKey` メソッドはゲームが起動した時に最初に一度だけ呼び出します。ブラウザのキー入力時へのフックを追加します。

`saveBeforeKey` メソッドは、フレーム処理の最後に毎回実行します。そのフレームで、プレイヤーが入力されたキーの履歴を保存します。

```
// キーコードをBitに変換
Input.prototype._keyCodeToBitCode = function(keyCode){
    var flag;
    switch(keyCode) {
        case 16: // shift
            flag = Constant.BUTTON_SHIFT;
            break;
        case 32: // space
            flag = Constant.BUTTON_SPACE;
```

```
            break;
        case 37: // left
            flag = Constant.BUTTON_LEFT;
            break;
        case 38: // up
            flag = Constant.BUTTON_UP;
            break;
        case 39: // right
            flag = Constant.BUTTON_RIGHT;
            break;
        case 40: // down
            flag = Constant.BUTTON_DOWN;
            break;
        case 88: // x
            flag = Constant.BUTTON_X;
            break;
        case 90: // z
            flag = Constant.BUTTON_Z;
            break;
    }
    return flag;
};
// キー押下
Input.prototype.handleKeyDown = function(e){
    this.keyflag |= this._keyCodeToBitCode(e.keyCode);
    e.preventDefault();
};
// キー押下解除
Input.prototype.handleKeyUp = function(e){
    this.keyflag &= ~this._keyCodeToBitCode(e.keyCode);
    e.preventDefault();
};
```

`_keyCodeToBitCode` 関数は、 `onkeyup` もしくは `onkeydown` イベントが発生した際に渡される `KeyboardEvent` オブジェクトの `keyCode` です。
keyCode のどの値が、キーボードの どのキーに紐づくかは、

MDN(https://developer.mozilla.org/ja/docs/Web/API/KeyboardEvent/keyCod
) などを参照してください。

`handleKeyDown` は、キーが押下された際に実行される関数です。内部では、キーコードを Bit に変換し、 OR 演算子を使って、 `keyflag` に保存しています。ビット演算子は、Webエンジニアにとって慣れ親しまないと思うので、 詳しく解説します。

Zボタンが押されると、 `handleKeyDown` は以下を実行します。

```
//BUTTON_Z => 0b00010000
this.key_flag = 0b00000000 | 0b00010000;
```

OR 演算子なので、各bit それぞれについて、どちらか片方の値が 1 であれば、そのbitは1となります。 つまり、 `key_flag` は `0b00010000` となります。

Zボタンと同時にXボタンも押されていればどうなるでしょうか

```
//BUTTON_X => 0b00100000
this.key_flag = 0b00010000 | 0b00100000;
```

既にZボタンが押されているので、 `0b00010000` に対して OR 演算を行います。 各ビットに対して、片方の値が 1 になります。よって、既に押下されている Z ボタンの状態は残ったまま、 Xボタンも押された状態になります。 `0b00110000` となります。

`handleKeyUp` は、キーの押下が解除された際に実行される関数です。 内部では、キーコードを Bit に変換し、否定演算子で、ビットを反転させてから、 AND 演算子を使って、 `keyflag` に保存しています。 また順を追って解説していきます。

Zボタンが押されるのが解除されると、 `handleKeyUp` は以下を実行します。

```
//  BUTTON_Z => 0b00010000
// ~BUTTON_Z => 0b11101111
this.key_flag = 0b00010000 & 0b11101111;
```

AND演算子は、各bit それぞれについて、どちらか両方の値が 1 であれば、そのbitは1となります。 つまり、key_flag は `0b00000000` となります。

Zボタンと同時にXボタンも押されていればどうなるでしょうか

```
//  BUTTON_X => 0b00100000
//  BUTTON_Z => 0b00010000
// ~BUTTON_Z => 0b11101111
this.key_flag = 0b00110000 & 0b11101111;
```

既にZ, Xボタンが押されているので、 `0b00110000` に対して AND 演算を行います。 結果は、 `0b00100000` となり、Xボタンだけ押された状態は残ったまま、Zボタンが押された状態は解除されました。

```
// 指定のキーが押下状態か確認する
Input.prototype.isKeyDown = function(flag){
    return this.keyflag & flag;
};
```

`isKeyDown` メソッドは、指定されたキーが押下されているか確認するメソッドです。 `this.input.isKeyDown(Constant.BUTTON_Z)` のように呼び出します。

AND 演算子を使うことで、 該当のキーが押下されていれば `true` を、押下されてなければ、 `0x0` つまり `false` を返します。

`isKeyDown` は押下さえされていれば、キーを押しっぱなしでも常に `true` を返し続けます。 例えば自機のショットを撃つ際などの判定には向いていますが、 最初に押下された時に実行したいこと(メニューの決定など)の判

定には向いていません。 (後者はで `isKeyDown` メソッドを使うと、毎フレーム決定が押下された扱いになってしまいます) その場合は、 `isKeyPush` メソッドを使用します。

```
// 指定のキーが押下されたか確認する
Input.prototype.isKeyPush = function(flag){
    // 1フレーム前に押下されておらず、現フレームで押下されてるなら true
    return !(this.before_keyflag & flag) && this.keyflag & flag;
};
```

指定のキーが最初に押下されたかどうかを判定します。 内部では、AND 演算子を用いて、該当のキーが現在のフレームで押下されており、 1つ前のフレームで押下されてなければ、true を返します。

この一つ前のフレームでの押下を判定するために、 `before_keyflag` を使用しています。

`src/game.js`

```
var Game = function (canvas) {
    this.input = new Input();
    this.input.bindKey();

    /* ~ 以下省略 ~ */
};
Game.prototype.run = function() {
    /* ~ 以上省略 ~ */

    // 押下されたキーを保存しておく
    this.input.saveBeforeKey();
};
```

最後に、キー入力を受け取るために、 `Game` クラスを少し修正します。

それ以外のキー情報の保存方法

キー情報の保存方法は、いくつか種類があります。前の項目で解説したのは、 「前の押下情報だけを保存しておく」方法でした。これは、キーが押下されているか、あるいはキーが押下されたかを 判定するだけのゲームであればこの方法で充分です。

他の保存方法には、「コマンド履歴を保存する」「押下時間も保存する」方法などがあります。 「コマンド履歴を保存する」方法は、格闘ゲームなどの、プレイヤーが入力したコマンド履歴をプログラムが参照したい場合に用いられます。 「押下時間も保存する」タイプは、例えばシューティングゲームのチャージショットなどを利用する場合に用いられます。

今回は解説しませんが、覚えておくと他のゲームの実装時にも応用が利くでしょう。

ゲームパッド入力

前の項目で、プレイヤーのキーボード入力を取得する方法を紹介しました。 本項では、ゲームパッド(ゲームのコントローラー)からの入力も取得できるようにします。 HTML5 には GamePad API があるので、それを使ってゲームパッドからの入力を取得します。

`src/input.js`

```
Input.prototype.handleGamePad = function() {
    var pads = navigator.getGamepads();
    var pad = pads[0]; // 1Pコン

    if(!pad) return;

    this.keyflag = 0x00;
    this.keyflag |= pad.buttons[0].pressed ? constant.BUTTON_Z:
  0x00;
    this.keyflag |= pad.buttons[1].pressed ? constant.BUTTON_X:
  0x00;

    this.keyflag |= pad.axes[1] < -0.5 ? constant.BUTTON_UP:
```

```
    0x00;
      this.keyflag |= pad.axes[1] >  0.5 ? constant.BUTTON_DOWN:
    0x00;
      this.keyflag |= pad.axes[0] < -0.5 ? constant.BUTTON_LEFT:
    0x00;
      this.keyflag |= pad.axes[0] >  0.5 ? constant.BUTTON_RIGHT:
    0x00;
  };
```

`Input` クラスの更新箇所です。 `handleGamePad` 関数は、ゲームパッドからの入力を受け取って、 `keyflag` プロパティを更新するメソッドです。

`navigator.getGamepads` 関数により、ゲームパッドの情報を取得できます。 `navigator.getGamepads` は、 配列を返します。複数のゲームパッドが接続されていた場合にもそれぞれのゲームパッドの値が取得できるように、 `navigator.getGamepads` は配列を返すようになっているのです。今回は1Pコントローラしか使用しないので、 `pads[0]` のみを取得します。

`pad` 変数には、 `pad.buttons` と、 `pad.axes` があります。buttons は各種ボタン、axes は各種方向キーです。 今回は、 `buttons[0]` を、キーボードのZボタンと同等に、 `buttons[1]` をキーボードのXボタンと同等に扱います。

`pad.axes` は、2つの値の配列であり、それぞれ -1 ~ 1 の値を取ります。これは、スティックをどの方向にどれだけ倒したかによって強弱の概念があるからです。 今回は、0.5 と -0.5 を閾値として入力されたかされていないかをチェックしています。

`src/game.js`

```
  Game.prototype.run = function() {
      this.input.handleGamePad();
      /* ~ 以下省略 ~ */
  };
```

最後に、 `Game` クラスの `run` メソッド内で、 `handleGamePad` を呼び出します。 これで、ゲームプログラムから、ゲームパッドの値を取得することができます。

オブジェクト生成

前項で、プレイヤーからの入力を取得する処理を実装しました。これでシューティングゲームに必要な、「自機がショットを撃つ」「自機が移動する」を実装することができるようになりました。本稿では、まず「自機がショットを撃つ」を実装したいと思います。

ObjectBaseクラスの作成

今まで画面上のオブジェクトには、 `Chara` クラスしかありませんでした。 `Chara` クラスに加えて追加で `Shot` クラスを実装したいと思います。 `Chara` クラスには、自機固有の処理とオブジェクトとして共通の処理が実装されていました。オブジェクトで共通している処理は、 `ObjectBase` クラスに切り出して、 `Chara` クラスと、 `Shot` クラスは、 `ObjectBase` クラスを継承するようにしてしまいましょう。

`src/util.js`

```
'use strict';
var Util = function (){};
Util.inherit = function( child, parent ) {
    var getPrototype = function(p) {
        if(Object.create) return Object.create(p);

        var F = function() {};
        F.prototype = p;
        return new F();
    };
    child.prototype = getPrototype(parent.prototype);
    child.prototype.constructor = child;
};
module.exports = Util;
```

src/object/base.js

```
var id = 0;

var ObjectBase = function (scene) {
    this.scene = scene;
    this.game = game;

    this._id = ++id;

    this.x = 0;
    this.y = 0;

    // 経過フレーム数
    this.frame_count = 0;

    // 現在表示するスプライト
    this.current_sprite_index = 0;
};
ObjectBase.prototype.id = function () {
    return this._id;
};
// 更新
ObjectBase.prototype.update = function () {
    this.frame_count++;

    // animation sprite
    if(this.frame_count % this.spriteAnimationSpan() === 0) {
        this.current_sprite_index++;
        if(this.current_sprite_index >= this.spriteIndices().length
) {
            this.current_sprite_index = 0;
        }
    }

};
// 描画
ObjectBase.prototype.draw = function () {
```

```
    var image = this.core.image_loader.getImage(this.spriteName());

    var ctx = this.core.ctx;

    ctx.save();

    // set position
    ctx.translate(this.x, this.y);

    var sprite_width  = this.spriteWidth();
    var sprite_height = this.spriteHeight();

    ctx.drawImage(
        image,
        // sprite position
        sprite_width * this.spriteIndexX(), sprite_height * this.sp
riteIndexY(),
        // sprite size to get
        sprite_width,                       sprite_height,
        // adjust left x, up y because of x and y indicate sprite c
enter.
        -sprite_width/2,                    -sprite_height/2,
        // sprite size to show
        sprite_width,                       sprite_height
    );
    ctx.restore();
};

ObjectBase.prototype.spriteName = function(){
    throw new Error("must be implemented");
};
ObjectBase.prototype.spriteIndexX = function(){
    return this.spriteIndices()[this.current_sprite_index].x;
};
ObjectBase.prototype.spriteIndexY = function(){
    return this.spriteIndices()[this.current_sprite_index].y;
};
ObjectBase.prototype.spriteAnimationSpan = function(){
```

```
    throw new Error("must be implemented");
};
ObjectBase.prototype.spriteIndices = function(){
    throw new Error("must be implemented");
};
ObjectBase.prototype.spriteWidth = function(){
    throw new Error("must be implemented");
};
ObjectBase.prototype.spriteHeight = function(){
    throw new Error("must be implemented");
};
module.exports = ObjectBase;
```

src/object/chara.js

```
var Util = require("../util");
var ObjectBase = require("./base");
var Chara = function (scene) {
    ObjectBase.apply(this, arguments);
};
Util.inherit(Chara, ObjectBase);

Chara.prototype.spriteName = function(){
    return "chara";
};
Chara.prototype.spriteAnimationSpan = function(){
    return 10;
};
Chara.prototype.spriteIndices = function(){
    return [{x: 0, y: 0},{x: 1, y: 0}];
};
Chara.prototype.spriteWidth = function(){
    return 64;
};
Chara.prototype.spriteHeight = function(){
    return 64;
};
```

```
module.exports = Chara;
```

`Util` クラスは、ゲームに必要なユーティリティ関数をまとめた静的なクラスです。 JavaScript はプロトタイプベースの言語であり、継承を行うには少し特殊な実装が必要です。 継承を行う `inherit` 関数を `Util` クラスに作ることにします。 `Util.inherit(ChildClass, ParentClass)` で、親クラスのプロパティやメソッドを子クラスに継承することができます。

`ObjectBase` クラスは、 `Chara` クラスや `Shot` クラスが継承する基底クラスです。 前回の描画の項目などで記載したスプライトアニメーションなどの処理はここにまとめておきます。 (正確に言うと、スプライトを使わなかったり、スプライトのアニメーションをしないオブジェクトも今後増えていくので、 `SpriteAnimationObjectBase` 等のような名前が適切かと思いますが、 今回作るゲームの要件では、そこまで必要ではないので、このままで行こうと思います)

`Chara` クラスは、 `ObjectBase` クラスを継承します。今まで `Chara` クラスに実装したほとんどの機能が、 `ObjectBase` クラスに移動したので、 `Chara` クラス自身は、 スプライトの設定などの `Chara` 特有の設定を記載するだけのコードになりました。

ショット

それでは、 `Shot` クラスを実装したいと思います。

スプライト画像

`public/image/shot.png`

弾のスプライト画像は、本書では用意できなかったので、 読者の皆様の方でご用意頂けましたらと思います。

フリー素材としては、例えば東方弾幕風向けの素材として、以下があったりします。
http://danmakufu.wiki.fc2.com/wiki/%E7%B4%A0%E6%9D%90%E3%83%

AA%E3%83%B3%E3%82%AF
弾画像 弾のズレが最も少ない画像。imgフォルダに入れるだけで使えるタイプ。

`src/object/shot.js`

```
var Util = require("../util");
var ObjectBase = require("./base");
var Shot = function (scene, x, y) {
    ObjectBase.apply(this, arguments);

    this.x = x;
    this.y = y;
};

Util.inherit(Shot, ObjectBase);

Shot.prototype.spriteName = function(){
    return "shot";
};
Shot.prototype.spriteAnimationSpan = function(){
    return 0;
};
Shot.prototype.spriteIndices = function(){
    return [{x: 0, y: 0}];
};
Shot.prototype.spriteWidth = function(){
    return 16;
};
Shot.prototype.spriteHeight = function(){
    return 16;
};
module.exports = Shot;
```

スプライトアニメーションを使用しないため、スプライトを1つだけ指定しています。それ以外については、 `Chara` クラスと同様に、基本的な実装は、 `ObjectBase` クラスを継承し、 `Shot` クラス特有の情報だけ定数で定

義しています。

それでは、先ほどの `Chara` クラスに、プレイヤーからの入力を受け付けて、ショットを生成する処理を追加したいと思います。

`src/object/chara.js`

```
Chara.prototype.update = function () {
    BaseObject.run.apply(this, arguments); // 親クラスの run を実行

    // Zが押下されていればショット生成
    if(this.core.isKeyDown(Constant.BUTTON_Z)) {
        this.scene.addObject(new Shot(this, this.x, this.y));
    }
};
```

先述の入力の項目で述べたプレイヤーからの入力を取得する方法に従い、`Chara` クラス内で、プレイヤーの Z ボタンの押下を取得して、 `Shot` インスタンスを生成し、ステージに追加しています。

これで、Zボタンを押下することで、キャラが弾を撃つようになりました。しかしショットの動きをまだ追加していないため、ステージ上の自機の位置にショットが 描画されたまま動かないと思います。これは、事項の移動の項目で修正するので、一旦敵の生成について 解説していきたいと思います。

敵

敵クラスの作成も行います。これは今までに解説してきたことだけで実装できます。

スプライト画像

`public/image/enemy.png`

敵のスプライト画像は、本書では用意できなかったので、 読者の皆様の方でご用意頂けましたらと思います。

フリー素材としては、例えば東方弾幕風向けの素材として、以下があったりします。

http://danmakufu.wiki.fc2.com/wiki/%E7%B4%A0%E6%9D%90%E3%83%AA%E3%83%B3%E3%82%AF
KMAPさんの素材詰め合わせ

`src/object/enemy.js`

```
var Util = require("../util");
var ObjectBase = require("./base");
var Enemy = function (scene, x, y) {
    ObjectBase.apply(this, arguments);

    this.x = x;
    this.y = y;
};
Util.inherit(Enemy, ObjectBase);
Enemy.prototype.spriteName = function(){
    return "enemy";
};
Enemy.prototype.spriteAnimationSpan = function(){
    return 0;
};
Enemy.prototype.spriteIndices = function(){
    return [{x: 0, y: 0}, {x: 0, y: 1}, {x:0, y:2}];
};
Enemy.prototype.spriteWidth = function(){
    return 16;
};
Enemy.prototype.spriteHeight = function(){
    return 16;
};
module.exports = Enemy;
```

敵を表すクラスである `Enemy` クラスです。これまでの `Chara` クラスや、 `Shot` クラスと同様に、 `ObjectBase` クラスを継承して、実装しています。 この敵の出現処理ですが、次の項目の敵マスタークラスに委ねることにします。

敵マスター

敵の出現については、いつどこから出現するのかのロジックを考えて実装しなくてはなりません。

こうしたロジックについて、敵マスターという概念を導入し、敵マスターに出現ロジックをまとめると便利です。

`src/logic/master.js`

```
var Master = function (scene) {
    this.scene = scene;

    this.frame_count = 0;
};
Master.prototype.update = function () {
    this.frame_count++;

    if(this.frame_count % 100 === 0) {
        var x = Math.floor(Math.random() * this.core.width); // x軸
はランダム
        var y = 0; // 画面上部から
        this.scene.addObject(new Enemy(this.scene, x, y));
    }
};
```

敵マスターは、概念上存在するだけで、ゲームでは実際に見えないしプレイヤーからも認知されない存在です。 こうした敵を出現させる存在を、透明のオブジェクトとして `Object` クラスで実装し、扱うゲームエンジンも

存在します。 その方が、既存のオブジェクト管理の仕組みに乗っかって 実装できるので楽なのですが、今回は、オブジェクトの仕組みに乗っからずに実装したいと思います。

出現ロジックは極めてシンプルです。100フレームごとに1つ、敵を出現させます。 出現位置は、画面上部のどこからかランダムです。

`src/scene/stage.js`

```
var Master = require("../logic/master");
var StageScene = function (game) {
    /* ~ 以上省略 ~ */
    this.master = new Master(this);
};
// 更新
StageScene.prototype.update = function () {
    /* ~ 以上省略 ~ */

    this.master.update(); // 敵の出現
};
```

最後に、 `Stage` クラスにて、 `Master` クラスのインスタンスを追加し、フレームごとに `Master` の `update` メソッドを実行するように修正しました。

移動

前回までの項で、画面内に敵とショットのオブジェクトを生成するところまで書きました。 しかし、これらの自機・敵・自機のショットはまだ画面上で動いていません。

本稿では、画面内に存在するオブジェクトの移動について書きたいと思います。

キャラ移動

入力の項で、プレイヤーのキーボードあるいはゲームパッドから情報を取得できるようになりました。 それでは、プレイヤーの入力を元に、キャラの移動を実装しようと思います。

`src/object/chara.js`

```
var Constant = require("../constant");

// キャラの移動速度
var SPEED = 2;
Character.prototype.run = function(){
    /* ~ 以上省略 ~ */

    // 自機移動
    if(this.core.input.isKeyDown(Constant.BUTTON_LEFT)) {
        this.x -= SPEED;
    }
    if(this.core.input.isKeyDown(Constant.BUTTON_RIGHT)) {
        this.x += SPEED;
    }
    if(this.core.input.isKeyDown(Constant.BUTTON_DOWN)) {
        this.y += SPEED;
    }
    if(this.core.input.isKeyDown(Constant.BUTTON_UP)) {
```

```
        this.y -= SPEED;
    }

    // 画面外に出させない
    this.forbidOutOfStage();
};
// 画面外に出させない
Character.prototype.forbidOutOfStage = function(){
    if(this.x < 0) {
        this.x = 0;
    }
    if(this.x > this.scene.width) {
        this.x = this.core.width;
    }
    if(this.y < 0) {
        this.y = 0;
    }
    if(this.y > this.core.height) {
        this.y = this.core.height;
    }
};
```

`Chara` クラスを少し修正しました。 `run` メソッド内で、プレイヤーの入力を受け取って、自機の(x, y)座標を移動させています。 また、 `forbidOutOfStage` メソッドにより自機が画面の外に出てしまわないように、 画面外に出てしまったら、画面内に座標を戻すように修正しています。

ベクトル移動

前の実装でキャラの移動について実装しました。(x, y)の加算・減算だけで実現しましたが、 この方法は、斜め方向の移動等、細かい移動をするには不便です。 オブジェクトを移動させたい際に、「速度」「角度」を指定することで移動できるようになると便利です。

キャラの移動について実装したので、次は自機が撃つショットの移動を実装しましょう。 自機の撃つショットは、前方向に進んでいきます。

`src/util.js`

```
Util.thetaToRadian = function(theta) {
    return theta * Math.PI / 180;
};
Util.calcMoveX = function(speed, theta) {
    return speed * Math.cos(Util.thetaToRadian(theta));
};
Util.calcMoveY = function(speed, theta) {
    return speed * Math.cos(Util.thetaToRadian(theta));
};
```

`src/object/base.js`

```
var Util = require("./util");
var ObjectBase = function (scene) {
    /* ~以上省略 ~ */

    this.speed = 0;
    this.theta = 0;
};
ObjectBase.prototype.update = function () {
    /* ~以上省略 ~ */

    this.move();
};
ObjectBase.prototype.move = function() {
    if (this.speed === 0) return;

    var x = Util.calcMoveX(this.speed, this.theta);
    var y = Util.calcMoveY(this.speed, this.theta);
    this.x += x;
    this.y += y;
};
```

```
ObjectBase.prototype.setMove = function(speed, theta) {
    this.speed = speed;
    this.theta = theta;
};
```

`ObjectBase` クラスを少し修正しました。 `ObjectBase` クラスを継承したオブジェクトは、 `setMove` メソッドに、 `speed` と `theta` を指定することで、後はプログラムが自動でそれに応じた移動をしてくれます。 `theta` はどの方向に向かうかの角度、 `speed` は1フレームにどのくらい進むかの速度です。

`theta` はシータと言います。記号では θ と記述します。0 ~ 360 の値を取り、 右が 0、下が、90、左が180, 上が 270 となります。(360 は 0 と同じく右です。)

`speed * cos(θ)` を計算すると、θ 方向に `speed` の分だけ移動する際に、x 座標はどのくらい進むのかわかります。 同じく、 `speed * sin(θ)` を計算すると、θ 方向に `speed` の分だけ移動する際に、y 座標はどのくらい進むのかわかります。

なお、JavaScript の `sin` `cos` 関数は、θ ではなく、ラジアンを渡さないといけないので、 `Util.thetaToRadian` 関数により、θ からラジアンに変換しています。

それでは、 `setMove` を使って、自機のショットに移動を設定してみたいと思います。

`src/object/chara.js`

```
// ショットの移動速度
var SHOT_SPEED = 3;
var SHOT_THETA = 270; // 画面前方
Chara.prototype.update = function () {
    BaseObject.run.apply(this, arguments); // 親クラスの run を実行

    // Zが押下されていればショット生成
```

```
    if(this.core.isKeyDown(Constant.BUTTON_Z)) {
        var shot = new Shot(this, this.x, this.y);
        shot.setMove(SHOT_SPEED, SHOT_THETA);

        this.scene.addObject(shot);
    }
};
```

`setMove` 関数を使用し、 `Chara` クラス内で、ショットを生成する際に、ショットの角度と速度を設定しています。 これで、Zボタンを押すと、自機から弾が発射され、画面前方に進んでいくようになったと思います。

自機狙い

自機狙いなど、オブジェクトがある特定の位置に向かって移動する設定をするにはどうすればいいでしょうか。 移動自体は、 `setMove` に `speed` と `theta` を渡してあげればいいのですが、オブジェクトの位置と、移動先の位置から、 `speed` と `theta` を計算する必要があります。

`src/util.js`

```
/* ~ 以下を追加 ~ */
Util.radianToTheta = function(radian) {
    return (radian * 180 / Math.PI) | 0;
};
```

`src/object/base.js`

```
/* ~ 以下を追加 ~ */
ObjectBase.prototype.setAimTo = function(x, y) {
    var ax = x - this.x;
    var ay = y - this.y;

    var theta = Util.radianToTheta(Math.atan2(ay, ax));
    return theta;
```

```
};
```

JavaScript には `atan2` 関数があるので、これを使うことで、2つの(x, y)座標から、 どの方向に向かうかの角度を取得することができます。 ただし、 `atan2` から返ってきた角度は、ラジアンなので、θに変換する必要があります。

これで例えば、敵を自機に向かって移動させたい場合は、

`src/object/enemy.js`

```
Enemy.prototype.update = function () {
    /* ~以上省略 ~ */

    if (this.frame_count % 60 === 0) {
        this.setAimTo(this.scene.chara.x, this.scene.chara.y);
    }
};
```

とすることで、自機に向かって移動してくる敵にすることができます。

毎フレーム、自機の移動先を計算すると処理に時間がかかるので、60フレームごとに 自機の位置を取得して、向かう先を修正しています。

当たり判定

先述の移動の項目にて、自機・敵・自機のショットの移動を実装しました。 少しずつゲームらしくなりましたが、敵にショットを当てて倒したり、逆に敵と接触して自機が死んだりはしません。

本項目では、当たり判定を実装し、以下を実現したいと思います。

- 自機と敵が接触すると、自機が死ぬ
- 自機の弾と敵が接触すると、敵が死ぬ

当たり判定について、自機や敵の画像というのは基本的に複雑な形をしており、 厳密に接触しているかどうかを判定すると、非常に計算に時間がかかります。 このため、自機や敵の形をすごく簡単な記号扱いすることで、当たり判定にかかる計算コストを減らす方法が取られます。 これは、キャラや敵を矩形(くけい、と読みます。四角形のこと)扱いして当たり判定する方法と、 円扱いして当たり判定する方法の大まかに2種類があります。

円同士の判定

`src/object/base.js`

```
ObjectBase.prototype.collisionRadius = function() {
    return 0;
};
ObjectBase.prototype.checkCollisionByCircle = function(obj) {
    if((this.x - obj.x)**2 * (this.y - obj.y)**2  <= (this.collisio
nRadius() + obj.collisionRadius())**2) {
        return true;
    }

    return false;
};
```

`ObjectBase` クラスに、円での当たり判定をチェックする処理を追加しました。 `collisionRadius` は当たり判定に使用する円の半径です。これは `ObjectBase` を継承した先のクラスで実装します。 また `checkCollisionByCircle` を用いて当たり判定を行います。 例えば、 `chara.checkCollisionByCircle(enemy)` を実行することで、自機と敵の当たり判定を行うことができます。

`checkCollisionByCircle` メソッドの中身を見ていきましょう。円と円の当たり判定については、 2つの円の中心点同士の距離が、2つの円の半径の合計より短ければ衝突しています。

円1: 中心点(x1, y1) 半径r1 円2: 中心点(x2, y2) 半径r2

2つの円の中心点同士の距離: √(x1-x2)^2 + (y1-y2)^2 2つの円の半径の合計: r1 + r2

√(x1-x2)^2 + (y1-y2)^2 ≦ r1 + r2

であるならば衝突しています。

JavaScript のコードにすると以下のようになります。

```
if(Math.sqrt((x1 - x2)**2 + (y1 - y2)**2) <= r1 + r2) {
    return true;
}
```

ただし、sqrt によるルートの計算は計算コストが高いので、二乗した値を用いて判定することが多いです。

```
if((x1 - x2)**2 + (y1 - y2)**2 <= (r1 + r2)**2) {
    return true;
}
```

矩形同士の判定

`src/object/base.js`

```
ObjectBase.prototype.collisionWidth = function() {
    return 0;
};
ObjectBase.prototype.collisionHeight = function() {
    return 0;
};
ObjectBase.prototype.checkCollisionByRect = function(obj) {
    if(Math.abs(this.x - obj.x) < this.collisionWidth()/2 + obj.col
lisionWidth()/2 &&
        Math.abs(this.y - obj.y) < this.collisionHeight()/2 + obj.c
ollisionHeight()/2) {
        return true;
    }

    return false;
};
```

`ObjectBase` クラスに、矩形での当たり判定をチェックする処理を追加しました。

矩形と矩形の衝突は、円と円の判定より多少面倒になります。 さらに言うと、回転した正方形と回転した正方形の計算となるとさらに面倒になります。

今回は、正方形が回転していることは考えずに、正方形の辺がX軸、Y軸に常に平行である場合を考えたいと思います。 x, y それぞれについて、2つの矩形の距離が、2つの矩形の幅の合計より短ければ衝突しています。

矩形1: 中心点(x1, y1) 横幅 w1 縦幅 h1 矩形2: 中心点(x2, y2) 横幅 w2 縦幅 h2

2つの矩形の中心点同士の距離(x座標): (x1-x2)の絶対値 2つの矩形の中心点同士の距離(y座標): (y1-y2)の絶対値 2つの矩形の横幅の合計: w1/2 + w2/2 2つの矩形の縦幅の合計: h1/2 + h2/2

(x1-x2)の絶対値 ≦ w1/2 + w2/2 かつ (y1-y2)の絶対値 ≦ h1/2 + h2/2

ならば衝突しています。

JavaScript のコードにすると以下のようになります。

```
if(Math.abs(x1 - x2) <= w1/2 + w2/2 && Math.abs(y1 - y2) <= h1/2 +
h2/2) {
    return true;
}
```

`checkCollisionByRect` メソッドは上記のコードを実装しています。 `Rect` は `Rectangle` の略称で、矩形(四角形)の英単語です。

BGM/SE再生

これまでの項で、シューティングゲームのゲームシステムに必要な最低限の実装は一通り紹介しました。 本項では、さらにBGM や SE の再生について解説します。

Web上での音声の再生は、HTML5 Audio を使う方法と、Web Audio を使う方法の二種類が存在します。

HTML5 Audio は、いわゆる `<audio>` タグを生成し、それを利用して、再生します。 簡単に音声が再生できる代わりに、再生の細かい制御ができません。

Web Audio は、JavaScript から利用する API です。細かい制御ができる代わりに、 制御に必要な細かい概念を知らなければなりません。

HTML5Audio

```
var element = new Audio();
element.addEventListener('canplay', function(e) {
    // 音声データの再生が可能になったら
});
element.src = "./path/to/file.mp3"; // 音声ファイルの指定
element.volume = 1.0; // 音量 0.0 ~ 1.0
element.loop = false; // ループするか否か
element.currentTime = 0; // 再生位置の指定
element.load(); // src で指定した音声ファイルの読み込みを開始する
element.pause(); // 再生の停止
element.play(); // 再生の停止
```

HTML5 Audio の使い方は上記のようになります。 `Audio` インスタンスを作成し、 `play` メソッドで再生します。 再生できるフォーマットは、ブラウザの種類にもよりますが、mp3 や wav, ogg, m4a 等が再生できます。

HTML5 Audio は非常に簡単に音声ファイルの再生を行うことができますが、 BGMのループの開始位置／終了位置を指定することができません。

そのため、ちょっとややこしい概念は出てきますが、Web Audio を使用して、 BGMやSEを再生したいと思います。

WebAudio

```
// context 作成
var audio_context = new AudioContext();

// mp3 ファイルの読み込み
var request = new XMLHttpRequest();
request.open('GET', "./path/to/file.mp3", true);
request.responseType = 'arraybuffer';

request.send();
request.onload = function () {
    var res = request.response;
    // arraybuffer を audio 再生用に decode
    audio_context.decodeAudioData(res, function (buffer) {
        // 音量調整用の GainNode 作成
        var audio_gain = audio_context.createGain();

        // 音量
        audio_gain.gain.value = 1.0;

        // オーディオデータの出力を行う AudioBufferSourceNode 作成
        var source = audio_context.createBufferSource();

        // AudioBufferSourceNode に オーディオデータを追加
        source.buffer = buffer;

        // ループ再生する
        source.loop = true;
```

```
        // ループの再生開始位置を指定
        source.loopStart = 5.23;

        // ループの再生終了位置を指定
        source.loopEnd = 10.55;

        // AudioBufferSourceNode → GainNode を接続
        source.connect(audio_gain);

        // GainNode → destination に接続
        audio_gain.connect(audio_context.destination);

        // 再生開始
        source.start(0);

        // 再生停止
        source.stop(0);
    });
};
```

Web Audio の使い方は上記のようになります。

流れ

ゲームにおいて Web Audio を使用する際の基本的な流れは以下になります。 オーディオデータの加工やミキシング等を行うには より複雑な流れが必要ですが、BGMの再生やSEの再生程度であれば、 下記の流れで充分です。

1. コンテクスト(AudioContext) を生成
2. オーディオデータ(ArrayBuffer) を取得(Ajax)
3. オーディオデータからPCMオーディオデータを生成
4. オーディオデータの入力点(AudioBufferSourceNode)を生成
5. 音量調整ノード(GainNode)を生成
6. オーディオデータと音量調整ノードを接続
7. 音量調整ノードと出力を接続

8. 音源のスイッチをON

コンテクスト

Web Audioで何かする前に、まずコンテクストを作成する必要があります。 コンテキストはオーディオ処理やデコードを行うノードの作成を制御します。 (ノードについては後ほど説明します。)

コンテクストには、 `AudioContext` (リアルタイムレンダリング)と `OfflineAudioContext` (オフラインレンダリング)が存在しますが、 ゲームにおいては、 `AudioContext` を使います。

`AudioContext` インスタンスからは以下のノードを作ることができます。

- AudioBufferSourceNode
  - GainNode
  - ScriptProcessorNode
  - AnalyserNode
  - DelayNode
  - BiquadFilterNode
  - IIRFilterNode
  - WaveShaperNode
  - PannerNode
  - SpatialPannerNode
  - StereoPannerNode
  - ConvolverNode
  - ChannelSplitterNode
  - ChannelMergerNode
  - DynamicsCompressorNode
  - OscillatorNode

たくさんの種類がありますが、ゲームで主に使うのは、 `GainNode` と `AudioBufferSourceNode` くらいです。 この2つについては、ノードの項で説明します。

ノード

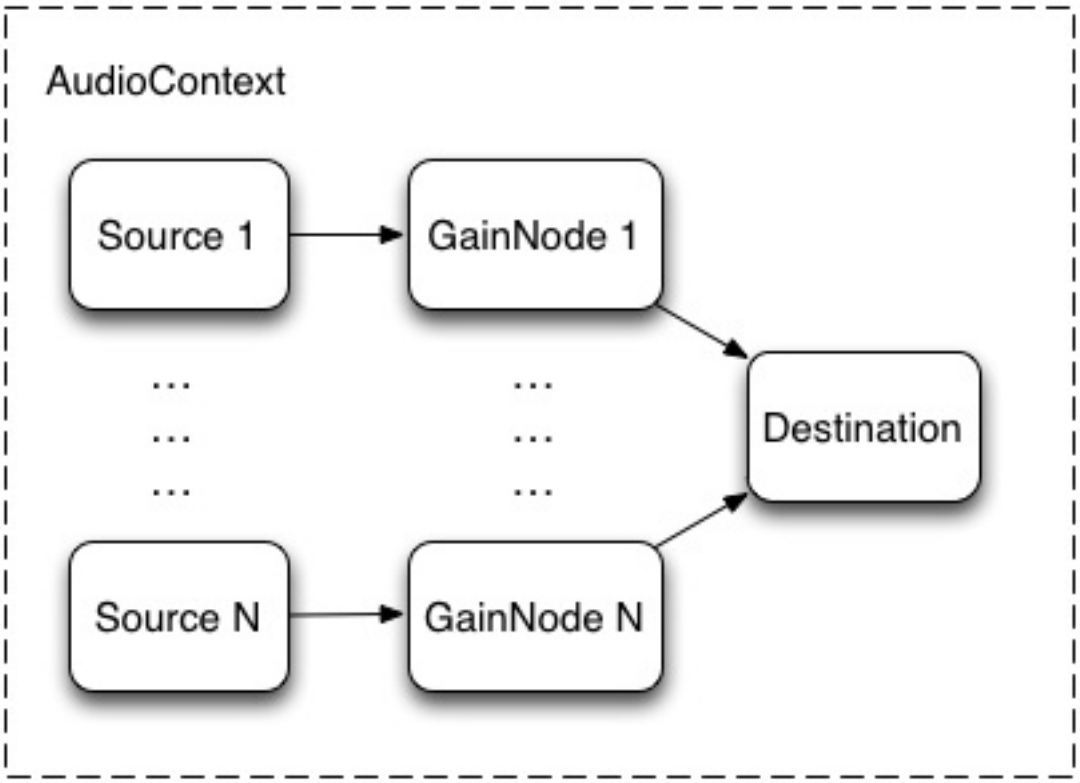
ノードとは、音声データのフィルターのようなものです。入力と出力を持ちます。 また、複数のノードを、入力と出力で接続して使います。 ノードの種類には例えば、入力された音声データにディレイをかけて出力したり、 入力された音声データの音量調整を行って出力したり、というのがあります。

一つのノードに対して複数のノードの出力／入力を接続することもできます。 接続されたノードの数はチャンネル数と呼ばれ、チャンネル数はノードの種類によって最大数がある場合もあります。

ゲームで主に使うのは、 `GainNode` と `AudioBufferSourceNode` 、 `AudioDestinationNode` です。 それぞれ、以下の役割があります。

- AudioBufferSourceNode はオーディオデータのバイナリを出力します。
- GainNode は音量調整を行うノードです。
- AudioDestinationNode はオーディオの最終的な出口です。

モジュラールーティング



Web Audio API の基礎
(*https://www.html5rocks.com/ja/tutorials/webaudio/intro/*) より引用

ノードの項目でも説明しましたが、ノードは入力と出力を持ち、 あるノードの出力と、別のノードの入力を接続することができます。

今回は、ゲームにおいて Web Audio を使用するので、以下のような ノードの接続の仕方をすることにします。

AudioBufferSourceNode -> GainNode -> AudioDestinationNode

それでは 1つずつ解説します。

```
var source = audio_context.createBufferSource();
```

`source` は `AudioBufferSourceNode` インスタンスです。入力を持たず出力を持ちます。

```
source.buffer = buffer;
```

`AudioContext.decodeAudioData` メソッドにより PCMデータに変換したオーディオデータを代入します。

```
source.loop = true;
source.loopStart = 5.23;
source.loopEnd = 10.55;
source.connect(audio_gain);
```

`AudioBufferSourceNode` インスタンスには、ループの再生や、ループ再生の開始／終了位置を設定することができます。

`AudioBufferSourceNode` インスタンスは使い捨てです。一度あるBGMやSEの再生に使用した後、 別のBGMやSEの再生に使用することはできません。BGMやSEの再生のたびに `AudioBufferSourceNode` インスタンスを 生成することになります。

```
var audio_gain = audio_context.createGain();
audio_gain.gain.value = 1.0; // 音量を 1.0 に設定
```

`audio_gain` は `GainNode` インスタンスです。入力と出力を持ちます。 入力されたデータの音量調整を行う加工を行い、それを出力します。

`GainNode` は一つのインスタンスを使いまわして、複数のBGMやSEを再生しても構いませんが、 音量調整やフェードイン／フェードアウト設定をいちいちリセットしなくてはならないため、 再生のたびに新しくインスタンスを生成することをオススメします。

```
audio_context.destination
```

`destination` は `AudioDestinationNode` インスタンスです。出力を持たず入力を持ちます。 オーディオの最終的な出力先です。

```
// AudioBufferSourceNode → GainNode を接続
source.connect(audio_gain);
```

```
// GainNode → destination に接続
audio_gain.connect(audio_context.destination);
```

各ノードのインスタンスが持つ `connect` メソッドにより、各ノードの入力と出力を接続することができます。

```
// 再生開始
source.start(0);

// 再生停止
source.stop(0);
```

`AudioBufferSourceNode` インスタンスの `start` , `stop` メソッドにより、再生の開始／停止ができます。(`AudioDestinationNode` ではなく、 `AudioBufferSourceNode` です。)

引数の 0 は、今すぐ再生するという意味です。N秒後に再生したい場合、引数に開始時刻を渡します。

iOS端末での再生

iOS ではユーザーのアクション契機でないとオーディオを再生できません。 一度ユーザーのアクション契機で再生すれば、以降はユーザーのアクション契機でなくとも オーディオを再生することができます。

よって、iOS向けの端末でのゲームでは、ユーザーの最初のタッチ時に 無音のオーディオを再生する処理をすると便利です。

```
var unlocked = false;

document.addEventListener('touchstart', function() {
    if (unlocked) {
        var node = audio_context.createBufferSource();
        node.start(0);
```

```
        unlocked = true;
    }
};
```

オーディオの読み込み

それでは、Web Audio でのオーディオデータの再生について解説しましたので、 いよいよそれを使って、ゲーム内で BGM や SEを再生する仕組みを整えていきたいと思います。

`src/asset_loader/audio.js`

```
var AudioLoader = function(game) {
    this.context = new AudioContext();
    this.audios = {}; // オーディオデータ

    this.loading_audio_num = 0;
    this.loaded_audio_num  = 0;

    this.playing_bgm_source = null; // 現在再生中のBGMの AudioBuffer
SourceNode インスタンス
    this.playing_bgm_gain   = null; // 現在再生中のBGMの GainNode イ
ンスタンス
    this.next_play_bgm      = null; // 次のフレームで再生予定のBGM
    this.next_play_se       = {};   // 次のフレームで再生予定のSE
};
AudioLoader.prototype.loadAudio = function(name, path, volume, is_l
oop, loop_start, loop_end) {
    var self = this;

    volume = volume || 1.0;

    self.loading_audio_num++;

    var request = new XMLHttpRequest();
    request.open('GET', path, true);
```

```
    request.responseType = 'arraybuffer';

    request.send();
    request.onload = function () {
        var res = request.response;
        // arraybuffer を audio 再生用に decode
        audio_context.decodeAudioData(res, function (buffer) {
            self.audios[name] = {
                buffer:     buffer,
                volume:     volume,
                is_loop:    is_loop,
                loop_start: loop_start,
                loop_end:   loop_end,
            };

            self.loaded_audio_num++;
        });
    };
};


AudioLoader.prototype.isAllLoaded = function() {
    return this.loaded_audio_num > 0 && this.loaded_audio_num === t
his.loading_audio_num;
};

// SE の再生
AudioLoader.prototype.playSE = function(name) {
    this.next_play_se[name] = true;
};
// BGM の再生
AudioLoader.prototype.playBGM = function(name) {
    this.next_play_bgm = name;

};

// SE, BGM の再生の実行
AudioLoader.prototype.executePlay = function() {
```

```
    // SE の再生
    for (var name in this.next_play_se) {
        var source = this._setupBufferSource(name);
        var audio_gain = this._setupAudioGain(name);

        source.connect(audio_gain);
        audio_gain.connect(this.context.destination);

        // 再生開始
        source.start(0);
    });

    // BGM の再生
    if (this.next_play_bgm) {
        this.playing_bgm_source = this._setupBufferSource(this.next
_play_bgm);
        this.playing_bgm_gain   = this._setupAudioGain(this.next_pl
ay_bgm);

        this.playing_bgm_source.connect(this.playing_bgm_gain);
        this.playing_bgm_gain.connect(this.context.destination);

        // 再生開始
        this.playing_bgm_source.start(0);
    });

    // reset
    this.next_play_se = {};
    this.next_play_bgm = null;
};

// BGM の再生停止
AudioLoader.prototype.stopBGM = function() {
    if(this.playing_bgm_source) {
        this.playing_bgm_source.stop(0);

        // reset
```

```
        this.playing_bgm_source = null;
        this.playing_bgm_gain   = null;
    }
};

AudioLoader.prototype._setupBufferSource = function(name) {
    var data = this.audios[name];

    var source = this.context.createBufferSource();
    source.buffer = data.buffer;

    if(data.is_loop) {
        source.loop = true;
    }

    if(data.loop_start) {
        source.loopStart = loop_start;
    }

    if(data.loop_end) {
        source.loopEnd = loop_end;
    }

    return source;
};
AudioLoader.prototype._setupAudioGain = function(name) {
    var data = this.audios[name];

    var audio_gain = this.context.createGain();
    audio_gain.gain.value = data.volume;

    return audio_gain;
};

AudioLoader.prototype.remove = function(name) {
    delete this.audios[name];
};
```

```
module.exports = AudioLoader;
```

上記が、 `AudioLoader` クラスの全容です。それではまた少しずつ解説していきたいと思います。

```
// SE の再生
AudioLoader.prototype.playSE = function(name) {
    this.next_play_se[name] = true;
};
// BGM の再生
AudioLoader.prototype.playBGM = function(name) {
    this.next_play_bgm = name;

};
```

ゲーム内で、 `playSE` 及び `playBGM` メソッドを呼び出すことで、オーディオを再生します。 ただし、メソッドの中身を見てもらえばわかるように、 `playSE` 及び `playBGM` 内では 再生予定のオーディオのフラグを立てているだけです。

これは、1フレーム内に同じオーディオが複数再生される場合に、 1度しか再生しないようにするためです。例えば、シューティングゲームであれば、 複数の敵が同時に死ぬ場合がありますが、この際に死んだ敵の回数だけ敵が死ぬ音が再生されると 音が大きくなるあるいは、再生タイミングによってはそれぞれの SE がそれぞれズレてしまうため、 このように再生するときは一旦フラグだけ立てて、後述する `executePlay` メソッドで実際の再生を行います。

```
// SE, BGM の再生の実行
AudioLoader.prototype.executePlay = function() {
    /* ~ 省略 ~ */

    // BGM の再生
    if (this.next_play_bgm) {
        this.playing_bgm_source = this._setupBufferSource(this.next
```

```
_play_bgm);
        this.playing_bgm_gain   = this._setupAudioGain(this.next_pl
ay_bgm);

        this.playing_bgm_source.connect(this.playing_bgm_gain);
        this.playing_bgm_gain.connect(this.context.destination);

        // 再生開始
        this.playing_bgm_source.start(0);
    });

    /* ~ 省略 ~ */
};
```

`executePlay` 関数では、 `playSE` `playBGM` メソッドによって、 フラグの立っている SE や BGM の再生を実際に行います。 BGMの再生のときのみ、 `AudioBufferSourceNode` インスタンスを変数に保存しています。 これは、 `stopBGM` メソッドによって、BGMの再生を後ほど停止できるようにするためです。

`executePlay` メソッドは `run` 関数内で一度だけ呼び出してください。

```
Game.prototype.run = function () {
    /* ~ 省略 ~ */
    this.audio_loader.executePlay();
};
```

フェードイン／フェードアウト

ゲームでは、BGMのフェードインやフェードアウトを実装することがあると思います。 これは、 `GainNode` インスタンスの `gain` プロパティに存在する `linearRampToValueAtTime` メソッドを使用すると、 BGM のフェードイン／フェードアウトを簡単に実装することができます。

`src/asset_loader/audio.js`

```
AudioLoader.prototype.fadeInBGM = function(duration) {
    var self = this;
    if(self.playing_bgm_gain) {
        var gain = self.playing_bgm_gain.gain;
        var startTime = self.context.currentTime;
        gain.setValueAtTime(gain.value, startTime);
        var endTime = startTime + duration;
        gain.linearRampToValueAtTime(1, endTime);
    }
},

AudioLoader.prototype.fadeOutBGM = function(duration) {
    var self = this;
    if(self.playing_bgm_gain) {
        var gain = self.playing_bgm_gain.gain;
        var startTime = self.context.currentTime;
        gain.setValueAtTime(gain.value, startTime);
        var endTime = startTime + duration;
        gain.linearRampToValueAtTime(0, endTime);
    }
};
```

ゲーム内から、 `AudioLoader` インスタンスの `fadeInBGM` あるいは `fadeOutBGM` を呼び出すことにより、 現在再生中のBGMをフェードイン／フェードアウトさせることができます。

`fadeInBGM` `fadeOutBGM` メソッドについて少しずつ解説していきたいと思います。

```
gain.setValueAtTime(gain.value, startTime);
```

現在時刻における音量を、現在の音量に設定しています。 設定しなくてもデフォルトの値(=現在の音量)になるのですが、 古いブラウザでは、デフォルトの値が 0 になっているブラウザもあるため、念のため設定します。

```
gain.linearRampToValueAtTime(1, endTime); // fade in
gain.linearRampToValueAtTime(0, endTime); // fade out
```

第一引数は最終的な音量です。第二引数に、いつまでにその最終的な音量まで減らす／増やすのかを設定します。 フェードインであれば、最終的な値は 1 になるので、第一引数に 1 を、フェードアウトならば、最終的な値は 0 になるので、 第一引数に 0 を指定しています。

その他

最後に、ゲームを作る上で知っておくと便利な処理を紹介させていただきます。

FPS 計算

開発時には、FPSを表示しておくと便利です。 例えば、弾幕シューティングゲームでは処理落ちが発生することがよくあるので、 どこで処理落ちしているのか、どのくらい重いのかを判断することができます。

`src/game.js`

```
var FPS_SPAN = 30;
var DEBUG = 1;

var Game = function () {
    /* ~ 省略 ~ */

    // 前回にFPSを計算した際の時刻(ミリ秒)
    this._before_calc_fps_time = 0;

    // 計測したFPS
    this._fps = 0;

    /* ~ 省略 ~ */
};

Game.prototype.run = function () {
    /* ~ 省略 ~ */

    if(DEBUG) { // デバッグの際のみFPSを表示する。
        this._calcFPS();
        this._renderFPS();
    }
```

```
    /* ~ 省略 ~ */
};
Game.prototype._calcFPS = function () {
    // FPS_SPAN 毎にFPSを計測する
    if((this.frame_count % FPS_SPAN) !== 0) return;

    // 現在時刻(ミリ秒)を取得
    var newTime = Date.now();

    if(this._before_calc_fps_time) {
        this._fps = parseInt(1000 * FPS_SPAN / (newTime - this._bef
ore_calc_fps_time));
    }

    this._before_calc_fps_time = newTime;
};
Game.prototype._renderFPS = function () {
    // FPSをレンダリング
    var ctx = this.ctx;
    ctx.save();
    ctx.fillStyle = 'rgb( 6, 40, 255 )';
    ctx.textAlign = 'left';
    ctx.fillText("FPS: " + this._fps, this.width - 70, this.height
- 10);
    ctx.restore();
};
```

1フレームごとに毎回、処理にどれくらい時間がかかったかを計算すると、FPSの計算自体に処理時間を使ってしまうので、30フレームごとに計算することにします。何フレームごとに計算するかは、 `FPS_SPAN` 定数に定義しています。

実装の内容ですが、 `_calcFPS` メソッドにより、30フレームの経過にかかった時間をマイクロ秒単位で計算し、 1秒で何フレーム処理できたかを計算して表示しています。

セーブ

ゲームのデータをクライアント側で保存する際は、HTML5 の API の `localStorage` を利用すると便利です。 `localStorage` とは WebStorage の一種で、JavaScript を用いてクライアント側にデータを保存する仕組みです。 ユーザーのデータを半永久的にデータを保持することができます。データの読み込み・更新も比較的簡単です。

`src/logic/save_storage.js`

```
var SaveStorage = function () {
    throw new Error("this class is static class");
}

SaveStorage.key = "save_key";

SaveStorage.save = function(data) {
    var self = this;
    localStorage.setItem(this.key, JSON.stringify(data));
};
SaveStorage.load = function() {
    return JSON.parse(localStorage.getItem(this.key));
};
```

`localStorage` には文字列しか保存できないため、JavaScript のオブジェクトを保存できるように、 `JSON.stringify` , `JSON.parse` を使用しています。

画面最大化

`src/main.js`

```
window.changeFullScreen = function () {
    var mainCanvas = document.getElementById('mainCanvas');
    if (mainCanvas.requestFullscreen) {
```

```
        mainCanvas.requestFullscreen();
    }
};
```

`public/index.html`

```
<html>
<head>
<style>
#mainCanvas:-webkit-full-screen {
    height: 100%;
    width:  100%;
}
</style>
</head>
<body>
    <canvas id="mainCanvas" width="640" height="480"></canvas>
    <input type="button" value="最大化" onclick="changeFullScreen()
;"><br />
    <script type="text/javascript" src="./js/main.min.js"></script>
</body>
</html>
```

Canvas の DOM の `requrestFullscreen` メソッドを呼び出すことで、 ゲーム画面を最大化することができます。

また、Canvas DOM の `-webkit-full-screen` に CSSを当てています。 これは、フルスクリーン時に適用されるCSSです。 今回は height と width を 100% にすることで、画面最大化時に、ゲーム画面も最大化して表示しています。 (これがないと、画面は最大化しても、ゲーム画面は 640 x 480px のままになります)

Electron

Electron は JavaScript でデスクトップアプリケーションが作成できるツールです。 本書では、JavaScript と HTML5 を利用して、Webブラウザ上で動くゲームを作成しましたが、 Electron を使用することで、Web ブラウザ上で動くゲームをPC向けアプリケーションとして出力することができます。

npm で `electron` と `electron-packager` をインストールします。 `electron` は、PCでの実行環境のことで、 `electron-packager` は、 実際に Web アプリケーションを、PC向けアプリとしてビルドする際に使用するツールです。

```
# コマンドラインで実行する
npm install --save-dev electron electron-packager
```

Electron 上で Webブラウザゲームを動かすスクリプトを作ります。

`public/electron.js`

```
'use strict';
var electron = require('electron');
var app = electron.app;
var dialog = electron.dialog;
var BrowserWindow = electron.BrowserWindow;

var mainWindow;

function createWindow () {
    mainWindow = new BrowserWindow({
        "width":          640,
        "height":         480,
        "useContentSize": true,  // フレームのサイズをサイズに含まない
        "resizable":      false, // ウィンドウのリサイズを禁止
        "alwaysOnTop":    true,  // 常に最前面
    });

    mainWindow.loadURL(`file://${__dirname}/index.html`);
```

```
    mainWindow.on('closed', function () {
        mainWindow = null;
    });
}

app.on('ready', createWindow);

app.on('window-all-closed', function () {
    if (process.platform !== 'darwin') {
        app.quit();
    }
});

app.on('activate', function () {
    if (mainWindow === null) {
        createWindow();
    }
});
```

640 × 480 のウィンドウを作成し、そこに `index.html` を表示しています。 Electron にはメインプロセスとレンダラプロセスの概念がありますが、 このように Webブラウザ上で既に動くゲームをPC向けアプリケーションとして起動する場合は、 レンダラプロセス上で index.html を実行するだけで、PC向けゲームを作成できます。

`package.json` を少し更新します。以下の行を `package.json` に追加します。

`./package.json`

```
"main": "electron.js",
"scripts": {
    "start": "electron ./public/electron.js",
    "build:win": "electron-packager ./public AppName --platform=win
32  --arch=x64 --icon=icon.ico",
    "build:mac": "electron-packager ./public AppName --platform=dar
```

```
win --arch=x64 --icon=icon.icns",
},
```

`main` に指定したファイルが、Electron で作成したPC向けアプリケーションが 起動する際に実行される JavaScript ファイルです。

ビルド前に、開発用として Electron で実行できるように、 `npm start` コマンドを定義しています。

また 64bit の Windows と Mac 向けに `npm run-script build:win` と `npm run-script build:mac` コマンドを定義しています。 これらは、 `electron-packager` によりビルドされます。

```
electron-packager <sourcedir> <appname> --platform=<platform> --arc
h=<arch> [optional flags...]
```

`electron-packager` コマンドの使い方は上記のようになります。 `sourcedir` がビルド対象の html や js があるディレクトリです。 minify 前の js は必要ないので、 今回は `public` ディレクトリを指定しました。

`appname` はアプリケーションの名前です。 `platform` は Windows や Mac あるいは Linux などどのOS向けにビルドするのかを指定します。 `arch` は 32bit もしくは 64bit 向けに出力するのかを指定します。 `icon` には、アプリケーションのアイコンを指定します。Windows と Mac では、使用できるアイコンのフォーマットが異なるので、2種類用意することに注意してください。

参考文献

本書の内容の補完として、下記の文献を挙げておきます。 下記の文献を読むことで、より理解が深まるかと思います。

HTML5 - Web developer guides | MDN
https://developer.mozilla.org/en/docs/Web/Guide/HTML/HTML5

JavaScript: The Good Parts
https://www.amazon.co.jp/dp/4873113911

逆引きゲームプログラミング
https://www.amazon.co.jp/dp/479801169X

シューティングゲーム アルゴリズムマニアックス
https://www.amazon.co.jp/dp/4797359978/

後書き

初めまして、さいと申します。本書を手にとってくださり 誠にありがとうございました。

同人誌を書くのも初めてで、かつコレだけ長い文章を書くのも初めてでした。 稚拙な文章／内容／技術ですが、何かしら皆様の参考になれば幸いです。 誤字脱字、プログラムのミスなどありましたらごめんなさい。

Special Thanks

表紙及び文中のイラストは、ラパメリ様にお願いしました。 お忙しい中、イラストを提供くださり、この場を借りてお礼申し上げます。

奥付

・本のタイトル
Webエンジニアのためのゲームプログラミング入門
・発行日
2017/4/9 (日)
・発行者
ひまわり鎮痛剤 さい
・連絡先
sairoutine@gmail.com
Twitter: @sairoutine
Web: https://sai-chan.com/
・印刷会社
株式会社ポプルス