シューティングゲーム

ゲームの基本的な流れ

スタートボタンを押すと
ゲームが開始されます。

画面内をクリックすると
弾を打つことができます。

敵に当たったらゲームオーバー

30秒経過すると
ゲーム終了です。

ゲームの基本ルール

- ゲームの時間は**30秒**
- 弾が敵に当たるとスコアが**+1**
- **敵に直接当たる**とゲームオーバー
- 敵は**右→左**に動き、ランダムに出現
- 弾は最大**同時5発**まで出力可能

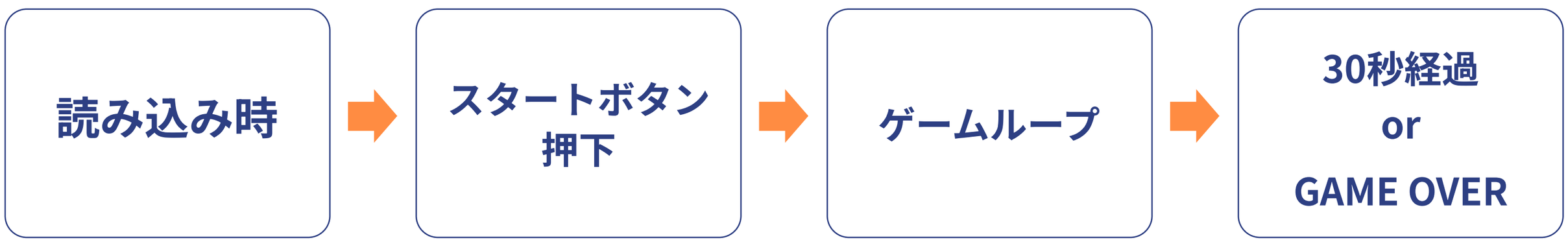
設計

必要なパーツ・データと
機能を洗い出す

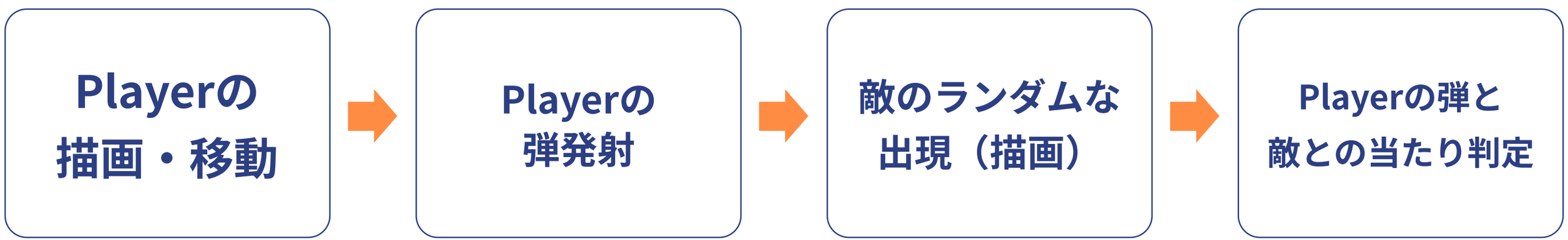
処理の流れを記述し
データと機能を割り当てる

点数 2

処理の流れ

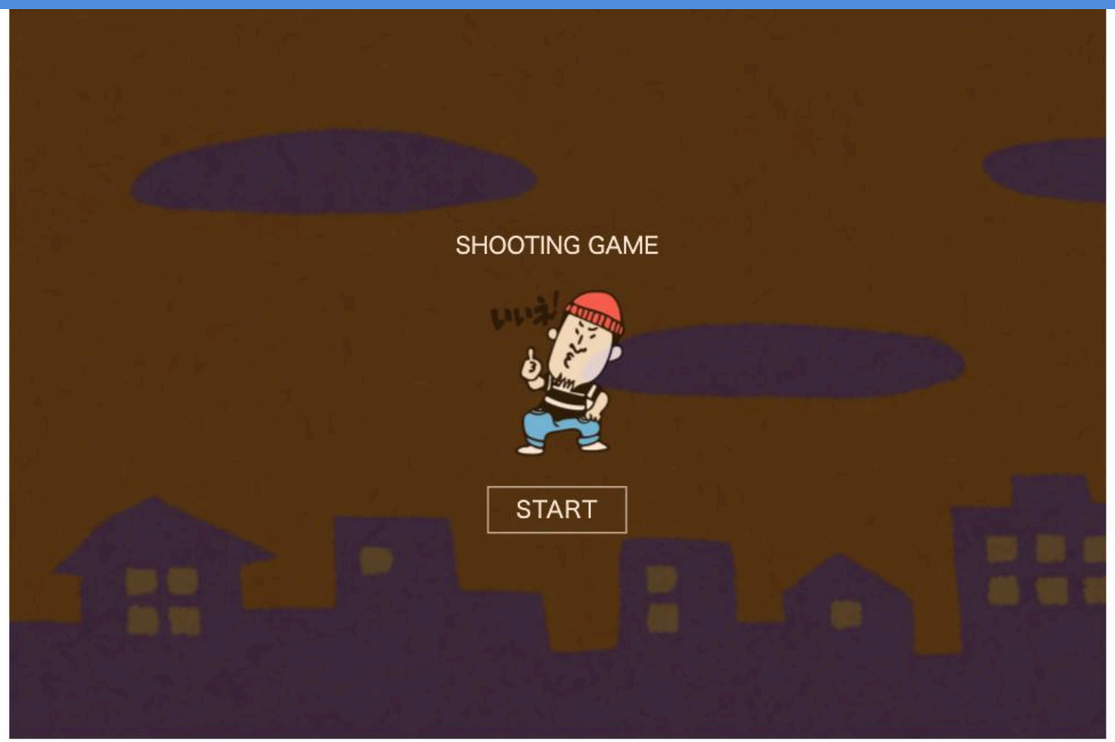


ゲームループ内の処理



スタートボタン押下時の処理

スタートボタン押下時



点数 0

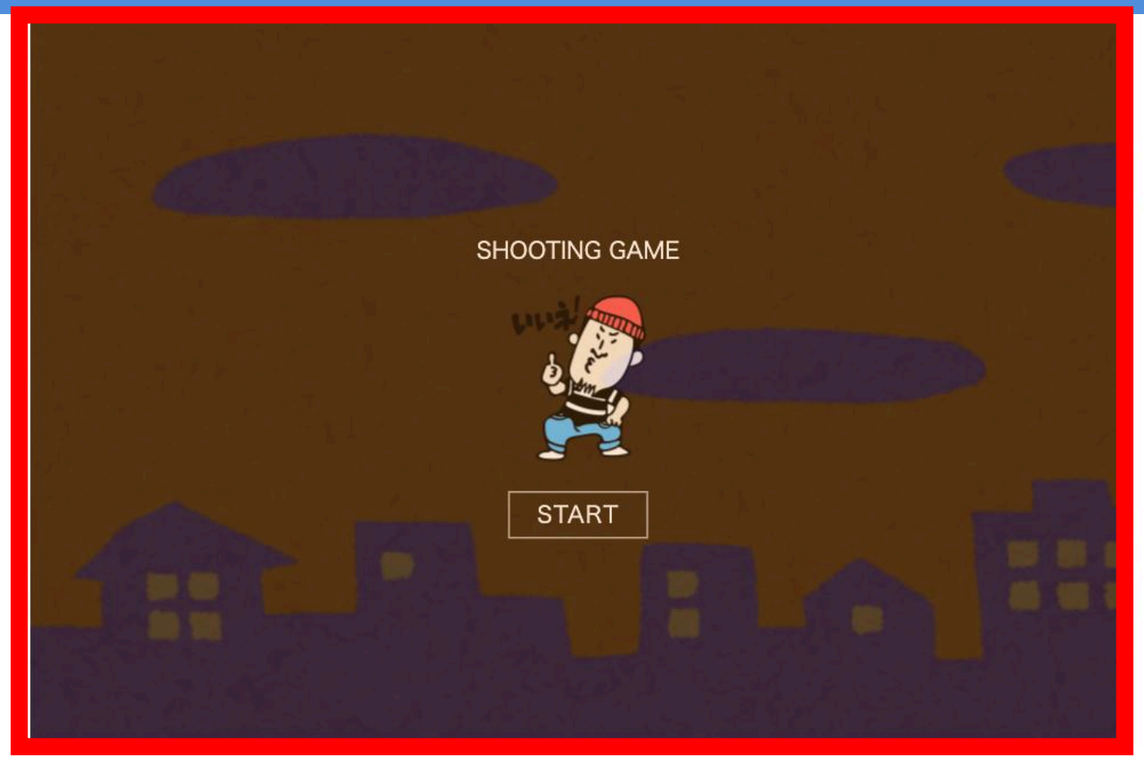
残り時間 30

点数 0

残り時間 30

- オーバーレイが非表示になる

オーバーレイを非表示にするためには



CSSとうまく組み合わせることで、指定したHTMLの表示・非表示をclassの付け替えだけで実装することができます！

overlayのHTML部分の構造

HTML

```
<div class="overlay">
  <div class="overlay-content overlay-start">
  </div>

  <div class="overlay-content overlay-gameover hide">
  </div>

  <div class="overlay-content overlay-finish hide">
  </div>
</div>
```

オーバーレイを非表示にするためには

CSS

```css
.hide {
    display: none;
}
```

JavaScript

```javascript
elem.classList.add('hide')
```

- ●● （オーバーレイ）の部分を非表示にして！

1. 直接非表示にするめ指示の仕方
2. 非表示用のclassを追加・削除

classList

JavaScript

```
elem.classList.add('hide')       // 特定のclassを加える

elem.classList.remove('hide')    // 特定のclassを削除する

elem.classList.toggle('hide')    // 特定のclassを切り替えする
```

clickイベントの登録

JavaScript

```
window.addEventListener("load", function () {
  const startBtn = document.querySelector("#btn-start");

  startBtn.addEventListener("click", function () {
    hideOverlay();
  });;);
});
```

- 特定のボタンに対して人間がクリック操作をした時の指示を登録します！

どのoverlayを非表示にする？

JavaScript

```
function hideOverlay(className) {
  const overlay = document.querySelector(".overlay");
  const target = document.querySelector(className);
   overlay.classList.add("hide");
   target.classList.add("hide");
}


 hideOverlay(".overlay-start");
```

Playerの表示

画像を描画・drawImage()

context.drawImage(img, x, y, w, h)

img・・・表示したい画像の内容（画像オブジェクト）

x・・・画像を表示したい位置のx座標

y・・・画像を表示したい位置のy座標

w・・・画像の横幅

h・・・画像の高さ

xとyで指定した場所が画像の左上に相当します

Player機について

- ufo.gifを使用
- Player機を画面に表示するには以下が必要
  - 表示位置（x,y）
  - 大きさ（横幅と高さ）
  - 元画像データの置き場所（パス）
  - 画像オブジェクトの格納(drawImageの仕様上)
  - 弾が打てるかどうかのフラグ（後ほど出てきます）

Player機について

```javascript
const player = {
  x: 0,
  y: 0,
  w: 0,
  h: 0,
  src: "images/ufo.gif",
  texture: null,
}
```

関連するデータは「オブジェクト」に
まとめます！！！

画像をcanvasで扱う場合の注意

drawImage()で
画像を表示可能

drawImage()実行時点で
画像の読込完了必須

読み込み未完了だと
エラーになる

画像を先に読み込み

JavaScript

```
function initPlayer() {
  const img = new Image();
  img.src = player.src;
  img.onload = function () {
    player.w = img.width;
    player.h = img.height;
    player.texture = img;
  };
}
```

drawImage()を実行する手前の段階で、
画像が読み込み完了することを保証するた
めに専用の処理を用意します！！

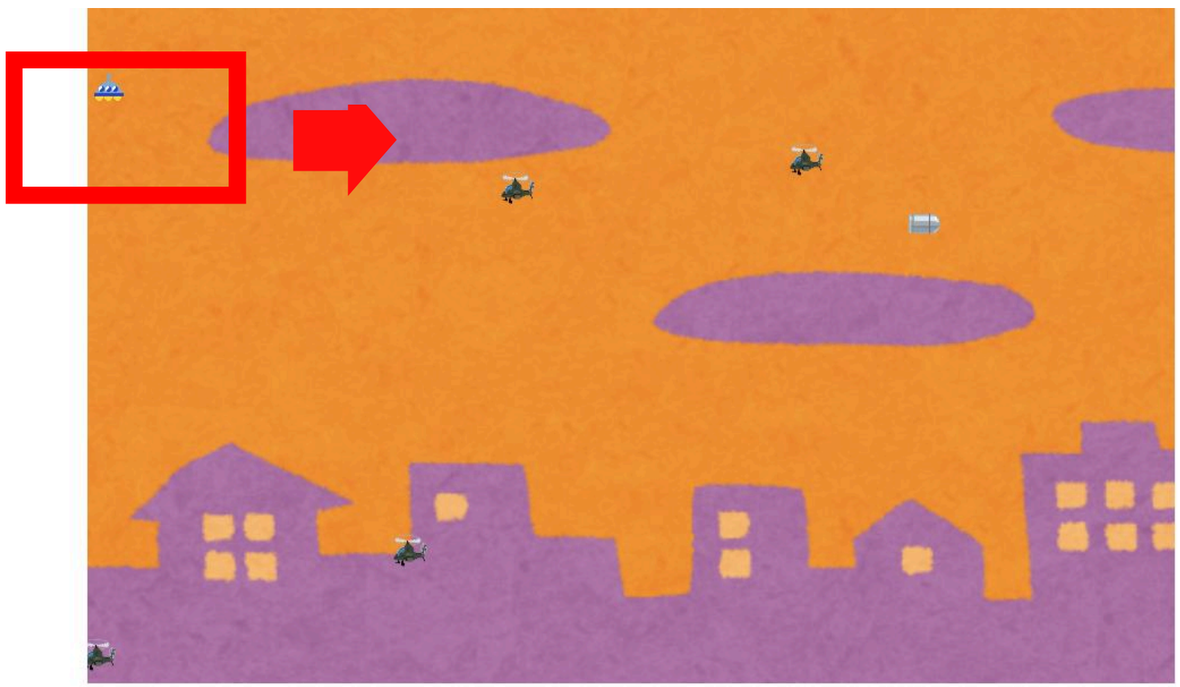
画像を先に読み込み

JavaScript

```
window.addEventListener("load", function () {
  initPlayer(); // 追加

  const startBtn = document.querySelector("#btn-start");
  startBtn.addEventListener("click", function () {
    hideOverlay();
  })
});
```

Player機をマウスに追従させる

マウス位置にPlayer機を追従させるには...？



- マウスの位置にPlayer機の真ん中を配置

canvas上をマウスが動く際に、プログラムでマウス位置を確認することができる仕組みがあればGood！！！

イベントオブジェクト

JavaScript

```
canvas.addEventListener("mousemove", function (e) {
    player.x = e.offsetX;
    player.y = e.offsetY;
});
```

イベントオブジェクトというものを使うと、マウスが動いたときに
関連する情報をまとめて取得することができます！

（console.log()で中身を確かめよう）

ゲームループの基本処理

JavaScript

```
function gameMainLoop() {

  // 全部消す

  // Player機の位置移動は今回マウスに追従させるのでここには書かない

  // Player機の描画

  requestAnimationFrame(gameMainLoop);
}
```

Playerの弾の管理

Player機の弾を配列で管理

```JavaScript
const playerBullets = [●●, ●●, ●●, ●●, ●●];
```

配列が弾の補充口になるイメージ

- 今回は初めから「5発まで」の弾数制限あり。これは「同時に画面上に5発までしか弾の表示ができないようにする」という意味。
- 配列で弾のデータ管理をすることで、管理を容易にすることが可能。

Player機の弾単体のデータ

JavaScript

```
const bullet = {
  x: 0,
  y: 0,
  w: 30,
  h: 30,
  src: "images/fig_bullet_ltr.gif",
  texture: null,
  isShotted: false, // ここがポイント
}
```

- 弾データのオブジェクトに
  isShottedというものを用意
- **isShotted**が**true**なら「発射済の弾」
  isShottedが**false**なら「未発射の弾」

isShottedの考え方

1. 初期状態（全部ストックにある）

全て isShotted: false
→ 撃てる状態

ストックにある（撃てる）

飛んでる最中（撃てない）

2. クリックで2発撃った

1,2・・・isShotted: true
3,4,5・・isShotted: false

画面上で飛んでる弾 →

弾1

弾2

3. 画面外に出たら → ストックに戻る

1・・・isShotted: false に戻る
→ また撃てる！

for文で5発分の弾のセットをする

JavaScript

```
const PLAYER_BULLET_MAX_COUNT = 5;

for (let i = 0; i < PLAYER_BULLET_MAX_COUNT; i++) {
  配列(playerBullets)に弾のデータを設定

  画像だけは読み込みに時間がかかるので後から設定
}
```

- loadイベント内で処理を実行する

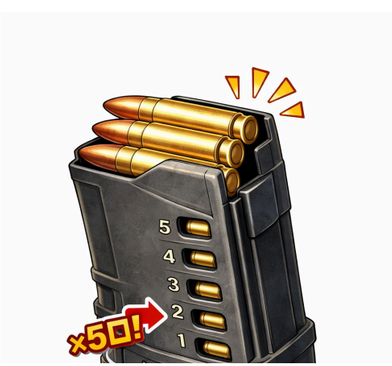
Playerの弾の発射・1

クリック操作時にやりたいこと

弾の補充口の中から
まだ弾があるかを
チェック

isShotted = falseがあれば
まだ使える（撃ってない）
弾と判断

isShotted = trueに
切り替え（事実上の発射）



Playerの弾の発射・2

弾を動かす

JavaScript

```
function moveBullet() {
  for (let i = 0; i < PLAYER_BULLET_MAX_COUNT; i++) {
```

弾の位置の移動

画面外に弾が出たらisShottedをfalseにする（また撃てるようにする）

弾を描画しなおす

```
  }
}
```

敵の管理

敵機を配列で管理

```JavaScript
const enemies = [●●, ●●, ●●, ●●, ●●];
```

- 敵はランダムに出現し右から左へ移動。縦軸座標はランダムに変わる。
- 敵機についても同時出現は最大5機までとする。

敵機の単体のデータ

JavaScript

```
const enemy = {
  x: 0,
  y: 0,
  w: 30,
  h: 30,
  speed: 0,
  src: "images/enemy.gif",
  texture: null,
  valid: false, // ここがポイント
}
```

- 敵データのオブジェクトに
  validというものを用意
- validがtrueなら「画面上にいる」
  validがfalseなら「画面上にいない」

validの考え方

1. 初期状態（全部ストックにある）

全て valid: false
→ 待機中

ストックにある（待機中）

画面上にいる（出現中）

2. 2機出現した

1,2・・・valid: true
3,4,5・・valid: false

画面上にいる敵 →

敵1

敵2

3. 画面外に出たら → 待機に戻る

1・・・valid: false に戻る
→ た出現できる！

for文で5発分の敵のセットをする

JavaScript

```
const ENEMY_MAX_COUNT = 5;

for (let i = 0; i < ENEMY_MAX_COUNT; i++) {
  配列(enemies)に敵のデータを設定

  画像だけは読み込みに時間がかかるので後から設定
}
```

- loadイベント内で処理を実行する

敵のランダムな出現

敵の出現について

敵は同時に5体までが
出現上限数

うまく制御しないと
ゲーム開始直後にすぐ5体
出現してしまう

確率制御をして
敵がすぐ出現しすぎ
ないようにする

敵の縦軸座標をランダムに

JavaScript

```
function showEnemy() {
  const randomNum = Math.ceil(Math.random() * 100);
  if (randomNum < 3) {
    for (let i = 0; i < ENEMY_MAX_COUNT; i++) {
      if (!enemies[i].valid) {
        enemies[i].x = canvas.width;
        enemies[i].y = Math.ceil(Math.random() * canvas.height);
        enemies[i].speed = 5;
        enemies[i].valid = true;
        break;
      }
    }
  }
}
```

敵を動かす

敵を動かす

JavaScript

```
function moveEnemy() {
  for (let i = 0; i < ENEMY_MAX_COUNT; i++) {
```

敵の位置の移動

画面外に敵が出たらvalidをfalseにする

敵を描画しなおす

```
  }
}
```

タイマーを作る

setInterval()

JavaScript

```
let timer;
function timeMeasure() {
  let gameSeconds = 30;
  timer.setInterval(function () {
    gameSeconds--;
    document.querySelector("#time").innerHTML = gameSeconds;
    if (gameSeconds <= 0) {
      clearInterval(timer);
      isRunning = false;
    }
  }, 1000);
}
```

30秒経過の場合

JavaScript

```
let isRunning = false;
function gameMainLoop() {

  // 省略

    if (!isRunning) {
      return false;
    }

  // 省略

}
```

スタート時点でゲームRunをONに

JavaScript

```
startBtn.addEventListener("click", function () {
  hideOverlay(".overlay-start");
  timeMeasure()
  gameMainLoop()
});;);
```

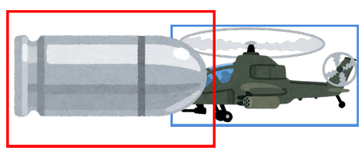
シューティングゲームの続き

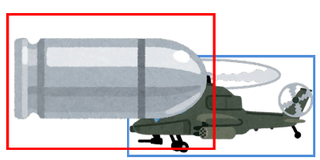
この章で扱うこと

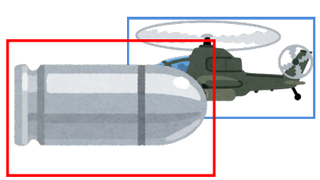
- 弾が敵に当たったらスコア加算
- 敵が自機に当たったらゲームオーバー
- タイマー終了時にクリア画面を表示
- リトライボタン

当たり判定について

当たり判定基礎

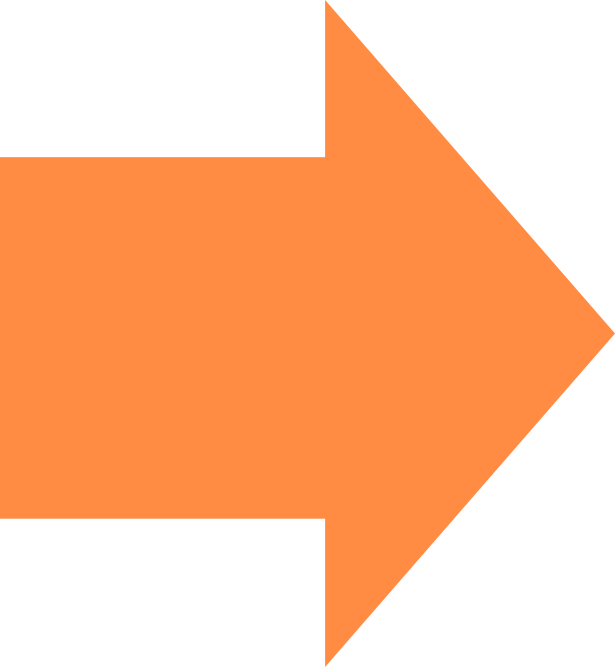






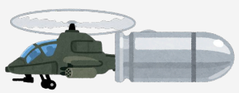
どれも設定している
つまり、「当たり」とみなすことができる！！

- 今回はPlayerの弾も敵も横方向に動く
- 敵の出現位置のY座標はランダム



2つの画像の当たり判定

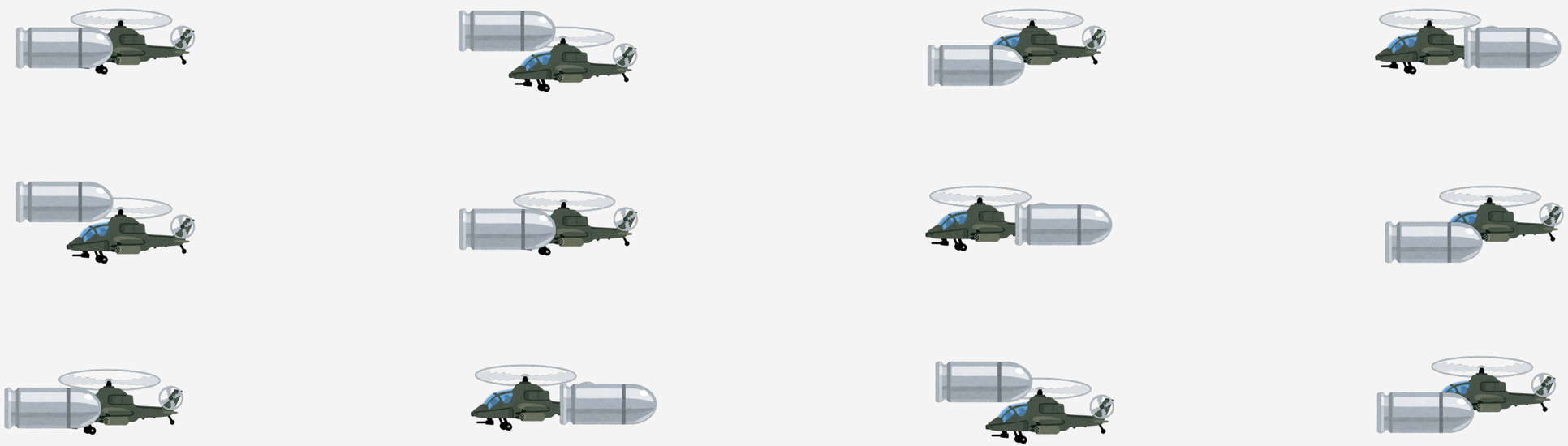


■2つの画像が重なるかどうか!?

- Aの右端 >= Bの左端
- Aの左端 <= Bの右端
- Aの下端 >= Bの上端
- Aの上端 <= Bの下端

当たり判定の対象



弾×敵の組み合わせを総当たりで当たり判定チェック！（もっといい方法ももちろんある）

どのタイミングで当たり判定！？

```JavaScript
function moveBullet(){


    // 当たり判定はここ



}
```

当たり判定は、moveBullet関数の中で
行います！！！

どのタイミングで当たり判定！？

JavaScript

```
function moveBullet(){

    // 当たり判定はここ
    if(a.x + a.w >= b.x &&
     a.x <= b.x + b.w &&
     a.y + a.h >= b.y &&
     a.y <= b.y + b.h ){

    }
}
```

JavaScript

```
function isColliding(a, b){
   return (a.x + a.w >= b.x &&
a.x <= b.x + b.w && a.y + a.h
>= b.y && a.y <= b.y + b.h
   );
}

function moveBullet(){
    if(isColliding(弾, 敵)){}
}
```

return

JavaScript

```
function isColliding(a, b){
   return (a.x + a.w >= b.x &&
a.x <= b.x + b.w && a.y + a.h
>= b.y && a.y <= b.y + b.h
   );
}

function moveBullet(){
    if(isColliding(弾, 敵)){}
}
```

別の関数で計算・処理をした内容を
別の関数で使いたいときに使用するのが
returnです！

スコア表示

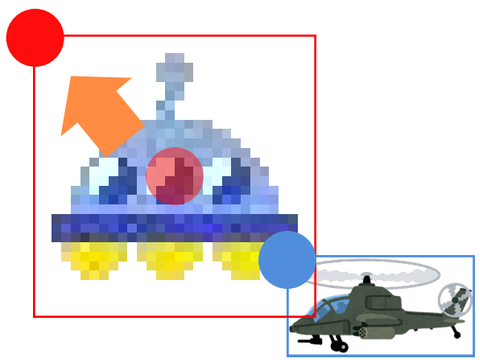
スコア表示

JavaScript

```
function updateScore(){
    let score = parseInt(document.querySelector("#score").innerHTML);
    document.querySelector("#score").innerHTML = score + 1;
}// 今のスコアを確認した上で、そのスコアに1を足し算した上で画面に表示

function moveBullet(){
    if(isColliding(弾, 敵)){
      updateScore() // スコア表示はここで
    }
}
```

敵と自機の当たり判定

敵と自機の当たり判定



Playerの位置情報は図形の真ん中で管理してるので
左上位置を基準にして考え直す必要がある！

JavaScript

```
const bullet = {
  x: player.x - player.w / 2,
  y: player.y - player.h / 2,
  w: player.w,
  h: player.h,
}
```

ゲームオーバー・クリア画面の表示

オーバーレイ表示のタイミング

30秒経過時

敵とPlayerの
当たり時

どのoverlayを表示にする？

JavaScript

```
function showOverlay(className) {
  const overlay = document.querySelector(".overlay");
  const target = document.querySelector(className);
   overlay.classList.remove("hide");
   target.classList.remove("hide");
}

 showOverlay(".overlay-gameover");
 showOverlay(".overlay-finish");
```

リトライボタン

リトライボタンは複数

リトライボタンは複数存在！！

どのボタンを押してもリトライ機能が
発動するように！！！

document.querySelectorAll()

JavaScript

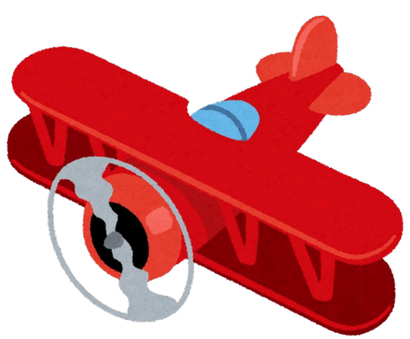
```
const retryButtons = document.querySelectorAll(".btn-retry");
```

document.querySelectorAll()を使うと、指定したパーツ（HTML）を全て取得可能！

リトライ = location.reload()で画面を再読み込みさせることで行います！

5回目・課題概要

別種類の敵を表示しましょう



- 別種類の敵はfig_biplane.png
- 別種類の敵も右→左に動き、ランダムに出現
- 敵に直接当たるとゲームオーバー
- それ以外も基本1種類目の敵と同じ条件

6回目・課題概要

敵から弾が出るようにしましょう

- 敵から弾が発射されるようにしてください
- 敵の弾は四角形（赤色）で表現してください
- 敵の弾がPlayerに当たるとゲームオーバー

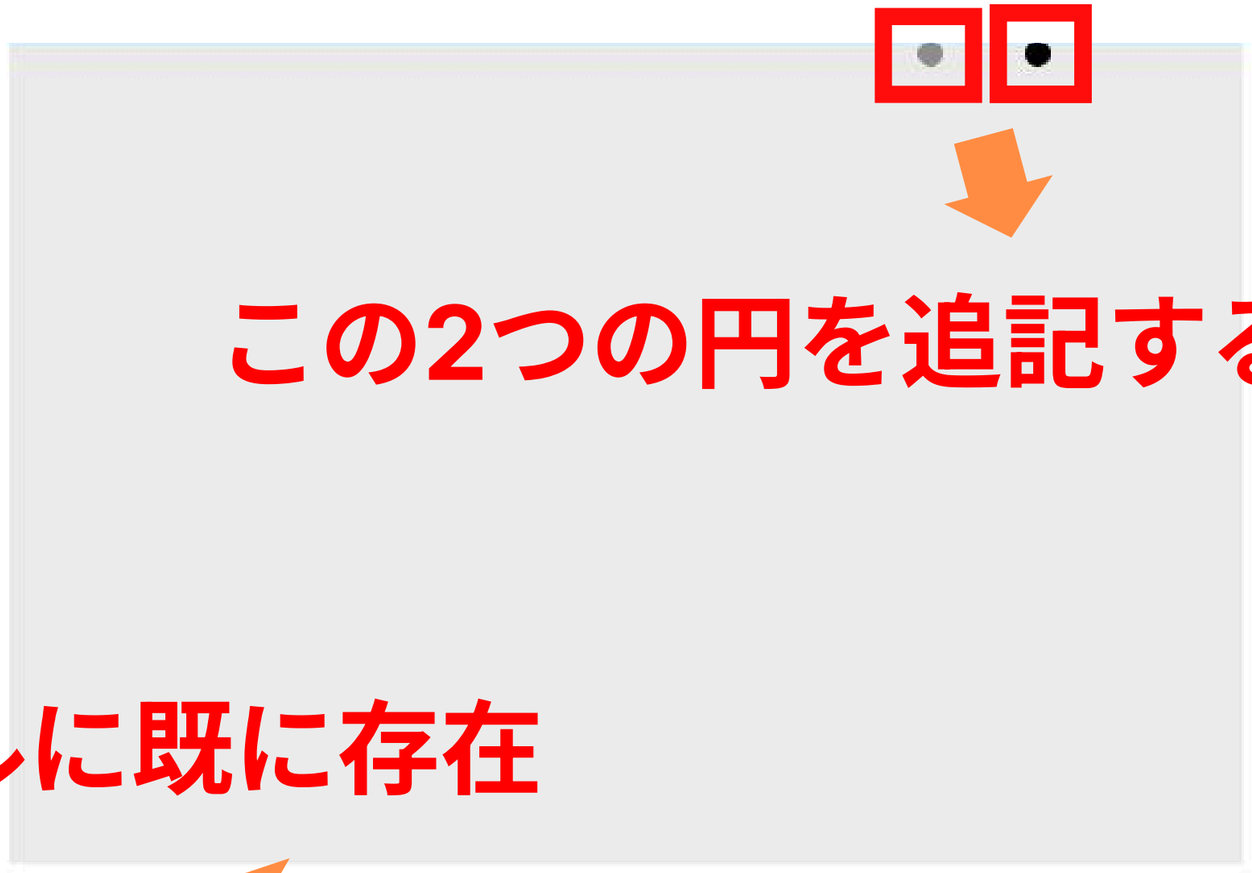
シンプルに左に弾が出るようにすればOK！できる方はPlayerに向かって弾が出るようにしてみてください！

第8回課題・解説

設計

必要なパーツ・データと
機能を洗い出す

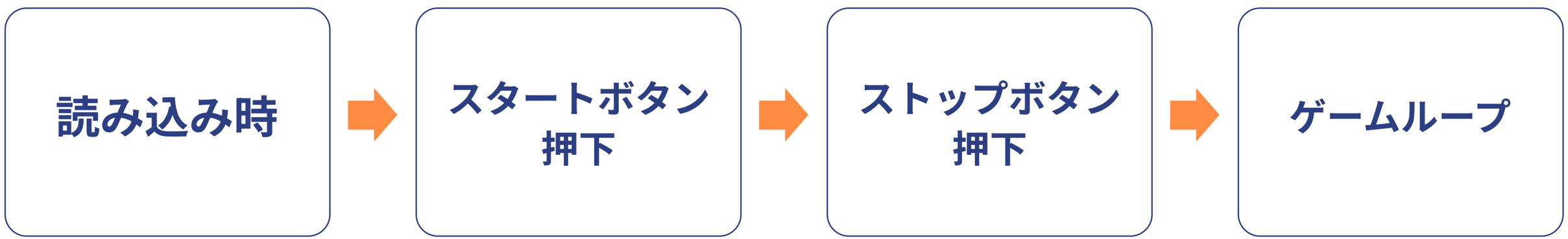
処理の流れを記述し
データと機能を割り当てる



この2つの円を追記する必要がある

配布ファイルに既に存在

処理の流れ



読み込み時

- ターゲットの黒円（※以降、円Bと呼びます）を描く
- loadイベントで上の処理を実行する

loadイベント・・・ページ読み込みした際の指示を記載

スタートボタン押下時

3

ゲームスタート ゲームストップ

- 数値がルーレットのようにランダム表示し続ける
- ランダムな数値は、後ほど灰色円（※以降、円Aと呼びます）の移動距離に反映される

ランダム表示の繰り返し・・・・setInterval()

ストップボタン押下時

- 数値の停止→確定した数値の変数speedへの反映
- 円Aを描画する（一旦はシンプルに描画→後ほど動かす）

「円を描く」が2回出てきます！ここが後ほどのポイントになってきます！！！

ゲーム進行（ループ）

- 先に決定した数値（変数speed）を元に、円Aを動かす
- 右端と左端にぶつかったら反転させる処理を入れる
- 移動中の摩擦処理と端衝突時の摩擦処理を加える
- 円Aと円Bの当たり判定 → 結果表示

段階的に細かく関数に分けて処理を作っていきます！！！！

目的地の円を描画する

円の描画

JavaScript

```
context.fillStyle = "#000";
context.beginPath();
context.arc(800, 10, 10, 0, Math.PI * 180, false);
context.fill();

// arc()は以下を当てはめていく
//  ・中心のx座標
//  ・中心のy座標
//  ・半径
//  ・開始角度
//  ・終了角度
//  ・時計回りか半時計回りか
```

- 「色・大きさ・位置」を教える
- 正円を描く場合、4つ目以降の値は左事例と同じようになります！

loadイベント

JavaScript

```
function initDestination() {
 // 中身は省略
}

window.addEventListener("load", function () {
  initDestination();
});
```

- 可読性を高めるため、処理は関数にして細かく分けていきます！！！

スタートボタン押下時の処理

スタートボタン押下時

3

ゲームスタート ゲームストップ

- 数値がルーレットのようにランダム表示し続ける
- ランダムな数値は、後ほど灰色円（※以降、円Aと呼びます）の移動距離に反映される

ランダム表示の繰り返し・・・・setInterval()

setInterval()

JavaScript

```
let timer;
let speed;

timer.setInterval(function () {
    speed = Math.ceil(Math.random() * 20);
    document.querySelector("#randomNumber").innerHTML = speed;
});
```

- 円Aのスピード決定＆変数格納 & 画面上に数字を表示という2つの役割があります！

clickイベントの登録

JavaScript

```
window.addEventListener("load", function () {
  const startBtn = document.querySelector("#btnStart");
  startBtn.addEventListener("click", startRandomNumber);

  initDestination();
});
```

- 特定のボタンに対して人間がクリック操作をした時の指示を登録します！

ストップボタン押下時の処理

clickイベントの登録

JavaScript

```
function stopAndShoot() {
 // 中身は省略
}

window.addEventListener("load", function () {
  // 途中省略
  const stopBtn = document.querySelector("#btnStop");
  startBtn.addEventListener("click", stopAndShoot);
  // 途中省略
});
```

ストップボタン押下時の詳細処理

JavaScript

```
function stopAndShoot() {

  // setInterval()の解除
   clearInterval(timer);


  // 円Aの描画


}
```

円の描画処理をClassでまとめる

円描画処理を見直す

```JavaScript
context.fillStyle = "#000";
context.beginPath();
context.arc(800, 10, 10, 0, Math.PI *
180, false);
context.fill();
```

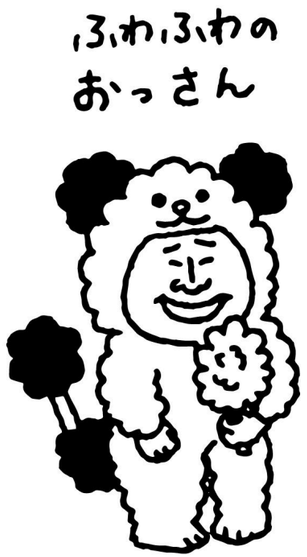
- 似たような処理が2回（複数回）記載されている...!

Classを使うことで効率的にデータや処理をまとめる！

Classとは

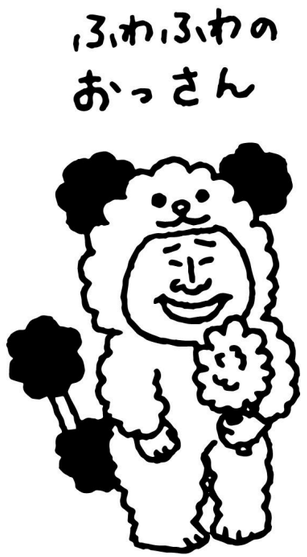
- 関連するデータと処理を1つにまとめたもの
- クラスはclass ●● と記載し、後ろに { } を記載する
- ●●の部分は、表現したいモノの概念を表す名前を示し、頭文字を大文字にする

円を描画するのに必要はデータは？



- 円の中心の座標（x, y）
- 半径の大きさ
- 円の色
- 円の移動スピード（今後、円Aを動かすので必要になってくる）

円に関して必要な処理は？



- 「円を描画する」今回はこれのみ！！！

Class Ballの構成

JavaScript

```
class Ball {

    constructor(初期値を受け取る場所){
      // 初期値を設定する場所
    }

    draw(){
     共通する処理を記載していく！！
    }（今回はdraw()→円の描画のみ）
}
```

- constructor()は初期値の設定場所
- 共通する処理はconsructor()の下に
  どんどん追記していく
- 共通処理の中で初期値を使いたい場合は
  this.●●のような書き方をする

Classの使用時

JavaScript

```
let destination;

// 途中省略


destination = new Ball(800, 10, 10,
0, "#000");
destination.draw();
```

- new クラス名()をで新しいデータ生成
- 初期値を()の中に注入する
- 生成したデータは変数に格納
- 共通処理を用いる場合は
  変数.処理名()のように記述

円を動かす

ストップボタンの処理に追加

JavaScript

```
function stopAndShoot() {

  // 途中省略

  gameMainLoop();
}
```

ストップボタンを押した時に
円を動かす（ゲームループ）の処理を
したいので左記のように追記をします！

ゲームループの基本処理

JavaScript

```
function gameMainLoop() {

  // 全部消す

  // 円の位置を移動

  // 円を描画

  requestAnimationFrame(gameMainLoop);
}
```

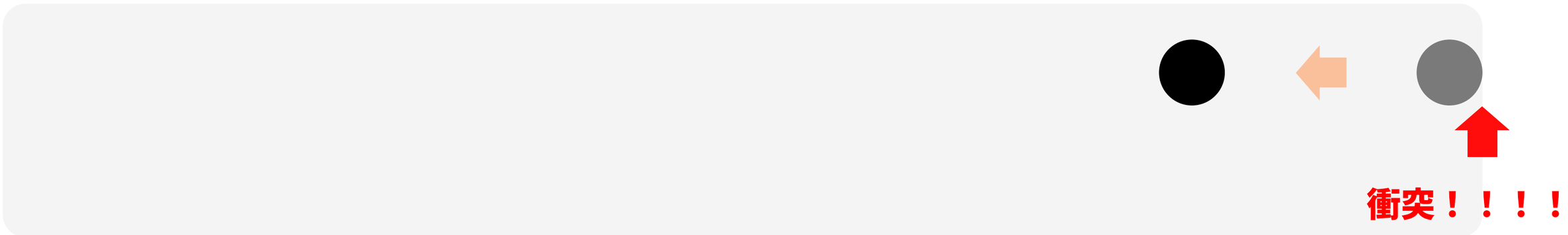
Class Ballに必要な処理を追加

JavaScript

```
class Ball {
    constructor(初期値を受け取る場所){
      // 初期値を設定する場所
    }
    draw(){
    }

    move(){
      // ボールを移動させるための処理
    }
}
```

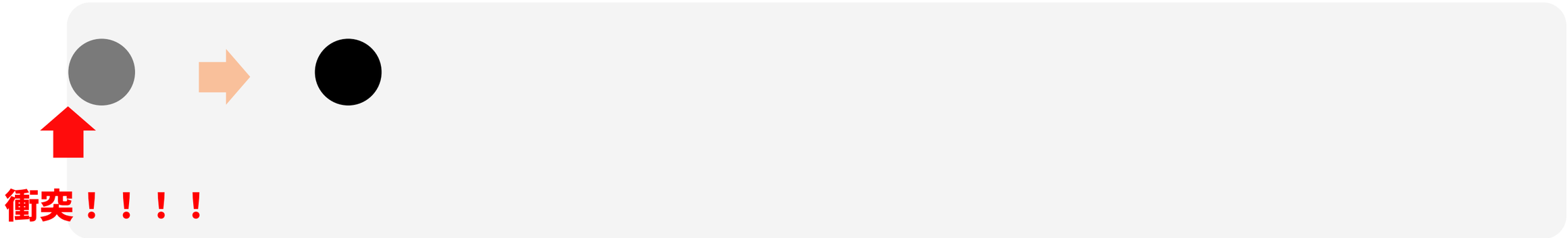
壁での反転

右端にぶつかった時



- **円の中心x座標＋円の半径**がゲームエリアの右端に衝突したら、**左方向に反転**させる

左端にぶつかった時



- **円の中心x座標 - 円の半径**がゲームエリアの左端に衝突したら、右方向に反転させる

円の反転処理について

JavaScript

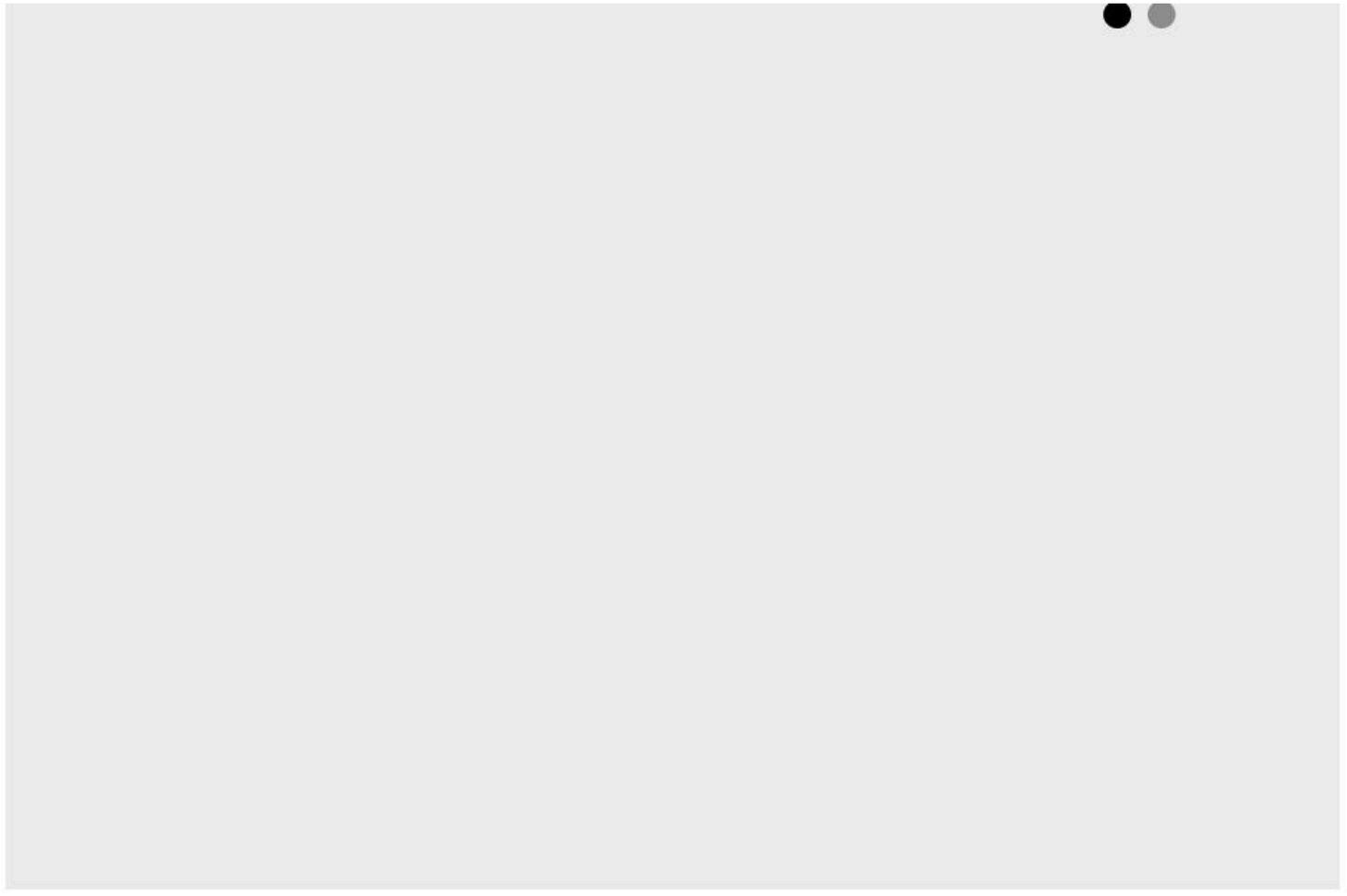
```
class Ball {

    // 途中省略

    turn(){
      // 円を逆方向に反転させる
      // マイナスを加えるだけ！！！

      this.speed = -this.speed
    }
}
```

摩擦処理を加える

摩擦の前提条件

- 摩擦は、床摩擦と衝突摩擦の2種類とする
- 床摩擦は0.99・衝突摩擦は0.8とする
- 床摩擦は「円の移動中」に発生
- 衝突摩擦は「右端 or 左端」に円が衝突した時に発生

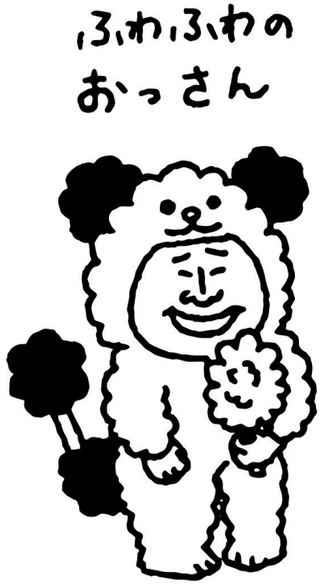
当たり判定



11

ゲームスタート ゲームストップ

当たり判定実施の前提条件



- 円Aの移動速度が0.01の以下になったときに
  当たり判定が実施される

2つの円の当たり判定

当たり！！！

はずれ・・・

※円A・・・灰色
円B・・・黒色

■2つの円が重なるかどうか!?

- 円Aの右端 >= 円Bの左端
- 円Aの左端 <= 円Bの右端

→ **両方が満たされればOK！**

結果表示