Eprobots JS Tutorial

In diesem Tutorial werde ich Schritt für Schritt eine Eprobots-Welt (Simulation genannt) aufbauen. Wir erschaffen die Simulation einer Welt mit künstlichen Lebewesen, die echte Evolution durchmachen und interessante Verhaltensweisen entwickeln.

Das Ganze kommt ohne externe Javascript-Bibliotheken und CSS aus. Es sind Grundkenntnisse in HTML und Javascript notwendig. Im Idealfall auch Erfahrungen mit Javascript-Klassen.

1. Grundgerüst / User-Interface

Wir fangen mit der index.html an. Alles was wir für das User-Interface brauchen ist ein Canvas zum Zeichnen der Welt und einen Button zum Starten und Stoppen der Simulation. Er bekommt die id `btn_start_stop`. Das Canvas-Element wird mit der Größe 800x400 vorbereitet. Die Größe kann hier verändert werden.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="main.js"></script>
</body>
</html>
```

index.html (komplett)

main.js als Einstiegspunkt

Andererseits wird auch schon ein erstes Script namens mains.js geladen. Dieses Script dient später auch als Einstiegspunkt für den Code und erledigt die notwendigen Initialisierungen.

main.js gibt vorerst nur einen Text aus.

```
console.log("Eprobots JS Tutorial - Start");
```

main.js (komplett)

Abschluss

Öffnet man die index.html im Browser sollte man in der Javascript-Konsole die Ausgabe “Eprobots JS Tutorial - Start” erhalten.

2. Simulations-Klasse und Settings-Modul

Im zweiten Teil des Tutorials werden wir die erste Klasse erstellen und ein Modul einführen, dass wichtige Einstellungen aufnimmt.

Die Simulations-Klasse ist der Dreh- und Angelpunkt des Programms:

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
    }
}
```

simulation.js (komplett)

Im Konstruktor soll später unser Canvas-Objekt übergeben werden, damit die Simulation weiß, wohin gezeichnet werden soll. Als erstes werden dann die Dimensionen des Canvas in Pixeln ausgegeben.

Anschließend wird die interne Weltgröße ausgerechnet. Man kann die Welt auch genauso groß wie das Canvas machen, beispielsweise 800x400, aber dann werden wir unsere Objekte kaum sehen können. Daher ist es gerade für Übungszwecke besser, die einzelnen Objekte größer zu machen. Zum Beispiel 8x8 Pixel. Dieser Wert ist aber leicht veränderbar, da er in ein Modul namens settings.js ausgelagert wird. Die Anzahl der Objekte in der Breite wird dann mit `canvas_width/settings.pixel_size` berechnet. Analoges gilt für die Höhe.

Aber was ist mit diesem `+2`?

Es ist ein gerne angewandter Trick die “Spielfläche” um einen Rand von einer Zelle zu erweitern (zum Beispiel im Computerschach). Wollen wir beispielweise eine Welt von 10x10 Zellen haben, initialisieren wir die Welt intern mit einer Größe von 12x12 (jeweils eine weitere Zelle oben/unten und links/rechts). *Sichtbar* wird nur der innere Bereich sein, dass heißt nur dieser Bereich wird tatsächlich gezeichnet.

Dies vereinfacht die spätere “Spiellogik” enorm, da man dann nicht ständig abfragen muss, ob man sich noch innerhalb der Grenzen befindet. Dazu besetzt man später die Randzellen mit besonderen Barriere-Objekten. So reicht es dann aus zu prüfen, ob sich an der Zielstelle ein Barrieren-Objekt befindet oder nicht. Im Laufe des Tutorials sollte dies klarer werden.

Anschließend werden die errechneten internen Welt-Dimensionen mit `console.log()` auf der Konsole ausgegeben.

Settings-Modul

An dieser Stelle bzw. in dieser Datei sollen später bequem und zentral (Simulations-)Einstellungen aufgenommen und verändern werden können. Alles was man gerne zentral als Parameter hätte sollte man dorthin auslagern, um keine Werte irgendwo im Code und vielleicht sogar mehrfach zu haben.

In Zukunft werden im Tutorial weitere Werte in die settings.js ausgelagert werden.

```
var settings = {
    pixel_size: 8
}
```

settings.js (komplett)

Neue Dateien in index.html aufnehmen

Ganz wichtig: Die beiden neuen Dateien/Module müssen in der index.html aufgeführt werden. Dies muss bei allen neuen Dateien gemacht werden. Ich werde in den Codeblöcken immer vollständige Dateien zeigen. Neue Zeilen und diejenigen die verändert werden müssen, sind hervorgehoben.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="simulation.js"></script>
    <script src="settings.js"></script>
    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

Simulation-Instanz erstellen

Da jetzt die neuen Dateien erstellt und eingebunden sind, können wir in der main.js eine Simulations-Instanz erstellen. Dazu sollte die main.js an dieser Stelle ausnahmsweise komplett mit diesem Code ersetzt werden:

```
document.addEventListener("DOMContentLoaded", function() {
    console.log("Eprobots JS Tutorial - Start");

    let canvas = document.getElementById("canvas");
    var simulation = new Simulation(canvas);
});
```

main.js (komplett)

`document.addEventListener("DOMContentLoaded", function() {});` sorgt dafür, dass der Code innerhalb der Funktion erst dann ausgeführt wird, wenn alles vollständig geladen wurde (im DOM).

Da die Simulation-Klasse in ihrem Konstruktor eine Canvas-Instanz benötigt, auf die gezeichnet werden soll, holen wir uns eine Referenz auf unser Canvas mit `document.getElementById("canvas");`. Der Zugriff erfolgt über das id-Attribut. Da unser Canvas schlicht “canvas” heißt, greifen wir auch so auf das Canvas-Element zu.

Anschließend wird dann die Simulations-Instanz mit dem canvas-Objekt erstellt.

3. Welt initialisieren mit Terrain-Objekten

Als erstes müssen wir die Welt erzeugen. Dazu führen wir zwei neue Klassen ein: World und Terrain.

```
class World {

    constructor(s, width, height) {
        console.log("World constructor");

        this.s = s;

        // init
        this.worldarr = new Array(width);
        for (var x=0;x<this.worldarr.length;x++){
            this.worldarr[x] = new Array(height);
```

```
            for (var y=0;y<height;y++){
                this.worldarr[x][y] = new Terrain(s, x, y);
            }
        }
    }

    get_terrain(x, y){
        return this.worldarr[x][y];
    }

}
```

world.js (komplett)

Um die Welt in der World-Klasse abbilden zu können gibt es ein Array namens `worldarr`. In diesem Array befindet sich unsere zweidimensionale Welt. Dies wird dadurch erreicht, das jedes Element im Array ebenfalls ein Array enthält. Die Breite und Höhe unserer Welt wird im Konstruktur übergeben. Im Konstruktor wird dann die zweidimensionale Welt mit zwei verschachtelten For-Schleifen erzeugt. Die einzelnen Zellen werden mit einem sogenannten *Terrain*-Objekt initialisiert. Diese Terrain-Objekte dienen dazu den Zustand einer Zelle aufzunehmen. So kann die Spielwelt später bequem erweitert werden.

Außerdem wird im Konstruktor auch eine Referenz auf die Simulationsklasse mit der Variablen `s` übergeben. Das ist notwendig, um innerhalb der World-Instanz auf die Simulation zurückgreifen zu können. Dieses “Muster” werde ich bei allen neuen Klassen anwenden.

Wichtig ist noch die Methode `get_terrain(x, y)`. Über diese Methode kann später auf die einzelnen Terrain-Felder über die Koordinaten zugegriffen werden. Im weiteren Verlauf werden wir nur noch über diese Methode auf Terrain-Objekte zugreifen und brauchen uns nicht mehr um die interne Implementierung mit dem zweidimensionalen Array kümmern.

Terrain-Klasse

Die Klasse Terrain ist erst einmal sehr simpel.

```
class Terrain {

    constructor(s, x, y) {
        this.s = s;

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

        this.dirty = false;

        this.slot_object = null;
    }
}
```

terrain.js (komplett)

Eine Instanz *kennt* seine Koordinaten. Das wäre theoretisch nicht nötig, aber ist aus Performance-Gründen sehr hilfreich. Außerdem gibt es das `dirty`-Flag, was für den später eingeführten *Drawer* wichtig sein wird. Damit das Terrain-Objekt auch tatsächlich etwas speichern kann gibt es das Feld `slot_object`. Das kann man sich als eine Art Behältnis vorstellen, in das man etwas setzen oder herausnehmen kann. Zum Beispiel unsere Eprobots oder auch die bereits angesprochenen *Barrieren-Felder*.

simulation.js erweitern

In der Konstruktor der *simulation.js* wird nichts weiter gemacht, als genau eine Instanz der World-Klasse zu erzeugen. Es bekommt die errechneten Werte für Breite und Höhe der Welt übergeben.

```
class Simulation {
```

```
    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);

        this.world = new World(this, this.world_width, this.world_height);
    }
}
```

simulation.js erweitern

Abschluss

Anschließend muss wie immer beim Einführen neuer Dateien die index.html erweitert werden.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="simulation.js"></script>
    <script src="settings.js"></script>
    <script src="world.js"></script>
    <script src="terrain.js"></script>

    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

4. Drawer und ein Eprobot

Es wird Zeit sich um das Zeichnen der Welt zu kümmern. Es gibt da verschiedene Strategien. Naiv könnte man einfach jedes einzelne Feld gemäß seines Zustandes zeichen. Dies würde zum Beispiel auch leere Zellen beinhalten und Zellen die sich gar nicht verändert haben. Die Simulation wird später so ablaufen, dass alle Eprobots “durchgerechnet” werden und anschließend wird das Spielfeld gezeichnet. Danach wird das ganze für den nächsten Simulationsschritt wiederholt usw. Würde man nun nach jedem Simulationsschritt die komplette Welt zeichnen, erzeugt das erhebliche CPU-Last, da ständig Felder neugezeichnet werden müssten, die sich meistens gar nicht verändert haben. Das ist sehr ineffizient. Gerade bei großen Welten mit vielen Eprobots. Dies würde dann die “Simulationsgeschwindigkeit” beeinflussen und die “Framerate” einschränken.

Daher werde ich den Ansatz umsetzen, nur die tatsächlichen Zeichenoperationen in einem Simulationsschritt in einer Liste namens `paintlist` zu sammeln und nur diese Zeichenoperationen werden dann tatsächlich ausgeführt. Dies lohnt sich wirklich erheblich und steigert die Effizienz enorm und somit die mögliche Framerate. Es ist

dann so, dass im Extremfall in einem Simulationsschritt gar keine Zeichenoperationen ausgeführt werden, wenn sich zum Beispiel nichts verändert hat oder die Welt leer ist.

```
class Drawer {

    constructor(s, canvas) {
        this.s = s;
        this.canvas = canvas;
        this.canvas_ctx = canvas.getContext('2d', {alpha: false});
        this.x_step = null;
        this.y_step = null;

        this.paintlist = [];

        this.init_canvas();
    }

    init_canvas(){
        // korrigieren weil nur der innere bereich angezeigt wird
        this.x_step = this.canvas.width / (this.s.world_width_visible);
        this.y_step = this.canvas.height / (this.s.world_height_visible);
    }

    paint_fast(){
        for (let terrain of this.paintlist){

            this.canvas_ctx.fillStyle = terrain.get_color();
            // mit positionskorrektur für zeichenbereich
            this.canvas_ctx.fillRect((terrain.x - 1) * this.x_step, (terrain.y - 1) * this.y_step,
         this.x_step, this.y_step);
            terrain.dirty = false;
        }

        this.paintlist = [];
    }
}
```

drawer.js (komplett)

Die `paintlist` nimmt Terrain-Objekte auf, die ihren Zustand verändert haben. So erklärt sich nun auch das `dirty`-Flag. Der Hintergrund ist der, dass es innerhalb eines Simulationsschrittes auch möglich sein könnte, dass ein Terrain-Objekt mehrfach seinen Zustand ändert. Das dirty-Flag verhindert, dass ein Terrain-Objekt nicht mehrfach in der paintlist landen kann. Dies wird über die neue Methode im Terrain-Objekt namens `prepare_paint` sichergestellt. Sie setzt das dirty-Flag nur auf `true`, wenn es `false` ist. Anschließend packt sich das Terrain-Objekt selbst in die paintlist des Drawers.

Die Methode `paint_fast()` erledigt das eigentliche Zeichnen. Die zu zeichnende Farbe der Zelle wird vom Terrain-Objekt mit der Methode `get_color()` abgerufen. Dazu später mehr. Mit `fillRect` wird dann das Rechteck gezeichnet, wobei dort dafür gesorgt wird, dass nur der *innere Bereich* der Welt gezeichnet wird. Im Anschluss kann das dirty-Flag des Terrain-Objektes wieder auf false gesetzt werden.

Außerdem sehr wichtig: Nach der For-Schleife muss paintlist wieder “geleert” werden, da das Zeichnen der Welt abgeschlossen ist. Dies wird einfach durch Zuweisen einer neuen und somit leeren Array-Instanz bewerkstelligt: `this.paintlist = [];`

terrain.js erweitern

```
class Terrain {

    constructor(s, x, y) {
        this.s = s;

        this.x = x;
        this.y = y;
```

```
        this.dirty = false;

        this.slot_object = null;
    }

    prepare_paint(){
        if (this.dirty == false){
            this.dirty = true;
            this.s.drawer.paintlist.push(this);
        }
    }

    get_color(){
        if (this.slot_object){
            return this.slot_object.get_color();
        }
    }
}
```

terrain.js erweitern

Die Methode `get_color()` in der Terrain-Klasse macht nichts weiter, als die Frage nach der Farbe an das eventuell vorhandene `slot_object` zu delegieren. Wir haben allerdings noch keine Klasse für Slotobjekte definiert. Daher definieren wir nun eine: Die Klasse Eprobot.

Eprobots-Klasse

Die Eprobots-Klasse ist vorerst sehr einfach aufgebaut. Instanzen kennen ihre Position. Die Methode `get_color()` gibt die Farbe “rot” zurück, bzw. den RGB-Hexwert “#f50000”.

```
class Eprobot{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;
    }

    get_color(){
        return "#f50000";
    }
}
```

eprobot.js (komplett)

Einen Eprobot in die Welt setzen

Nun kommen wir langsam das erste mal dazu, etwas auf unserem Canvas anzuzeigen.

Um Objekte setzen zu können, erweitern wir die World-Klasse um die Methode `world_set(o, x_pos, y_pos)`. Sie kümmert sich um das Setzen und Aktualisieren der notwendigen Variablen und sorgt durch das Aufrufen der Methode `prepare_paint` auf dem Terrain-Objekt dafür, dass das Terrain-Objekt später auch gezeichnet wird.

```
class World {

    constructor(s, width, height) {
        console.log("World constructor");

        this.s = s;

        // init
        this.worldarr = new Array(width);
        for (var x=0;x<this.worldarr.length;x++){
            this.worldarr[x] = new Array(height);
            for (var y=0;y<height;y++){
                this.worldarr[x][y] = new Terrain(s, x, y);
            }
```

```
        }
    }

    get_terrain(x, y){
        return this.worldarr[x][y];
    }

    world_set(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.slot_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

}
```

world.js erweitern

simulation.js erweitern

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        let eprobot = new Eprobot(this);
        this.world.world_set(eprobot, 10, 10);

        this.drawer.paint_fast();
    }
}
```

simulation.js erweitern

`this.world_width_visible` und `this.world_height_visible` sind Hilfsvariablen welche die Dimensionen der sichtbaren Welt in der Simulations-Instanz speichern und vom Drawer abgerufen werden.

Wichtiger aber ist das Erzeugen einer Instanz vom Drawer, welcher das Canvas übergeben bekommt und das anschließende Erzeugen des ersten Eprobots, welcher mit der Methode `world_set()` der Klasse World in die Welt an Position (x=10, y=10) gesetzt wird. Mit `this.drawer.paint_fast();` wird das Zeichnen ausgelöst.

Abschluss

Nachdem die beiden neuen Dateien in die index.html eingefügt wurden kann man das Projekt im Browser neuladen, damit man ein wundervolles rotes Rechteck zu sehen bekommt :-) Nicht sehr spannend und aufregend für den bisherigen Aufwand aber doch ein Erfolg und eine gute Grundlage für die weiteren Vorhaben.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">
```

```
    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="simulation.js"></script>
    <script src="drawer.js"></script>
    <script src="settings.js"></script>
    <script src="world.js"></script>
    <script src="terrain.js"></script>
    <script src="eprobot.js"></script>

    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

5. Barriers

Nun kommen wir zu den Barrier-Objekten von denen schon öfter die Rede war. Wir brauchen sie spätestens jetzt, denn wir wollen im nächsten Kapitel einen Eprobot herumlaufen lassen. Damit dieser nicht über die Grenzen der Welt “stolpert”, müssen wir die Barrier-Objekte am Rand der Welt setzen, die wir ja speziell für diesen Zweck erweitert haben. Zuerst hier die Definition der Barrier-Klasse:

```
class Barrier{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;
    }

    get_color(){
        return "#ffffff";
    }
}
```

barrier.js (komplett)

Da der Hintergrund unserer Welt schwarz ist, definiere ich für Barrier-Objekte die Farbe weiß (“#ffffff”). Am Rand sind die Barrier-Objekte zwar nicht sichtbar, aber man könnte ja auch auf die Idee kommen Barrier-Objekte in den sichtbaren Teil der Welt zu setzen. Dies tun wir im Folgenden auch einmal zu Testzwecken in der simulation.js.

simulation.js erweitern

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
```

```
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        let eprobot = new Eprobot(this);
        this.world.world_set(eprobot, 10, 10);

        let b = new Barrier(this);
        this.world.world_set(b, 12, 12);

        this.drawer.paint_fast();
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

Das Setzen der Barrier-Felder erfolgt in der neuen Methode `add_borders` und wird durch zwei For-Schleifen erledigt. Die erste For-Schleife setzt die Barrier-Objekte oben und unten und die zweite For-Schleife tut das gleiche für links und rechts. Für das Setzen der Barrier-Objekte verwenden wir wieder die World-Methode `world_set`.

Abschluss

Wird dann die barrier.js in der index.html referenziert und lädt man neu, kann man zusätzlich zum roten Eprobot ein weißes Barrier-Objekt sehen.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>
```

```
    <script src="simulation.js"></script>
    <script src="drawer.js"></script>
    <script src="settings.js"></script>
    <script src="world.js"></script>
    <script src="terrain.js"></script>
    <script src="barrier.js"></script>
    <script src="eprobot.js"></script>

    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

6. Eprobot random walk und Game Loop

Dieser Schitt ist wieder etwas umfangreicher, aber wir nähern uns einer lauffähigen Simulation. Ziel ist es den einen Eprobot umherwandern zu lassen.

Das Umherwandern wird erst einmal durch den Zufall gesteuert werden. Später wird es dann interessanter, aber vorerst konzentrieren wir uns auf den Game Loop.

Um den Eprobot wandern zu lassen sind ein paar Vorbereitungen notwendig. Zuerst brauchen wir eine Liste welche die möglichen Bewegungsrichtungen definiert.

In unserer einfachen zweidimensionalen Welt gibt es nur acht Möglichkeiten sich zu bewegen. Die acht Bewegungsmöglichkeiten werden in der Datei definitions.js in der Variablen `DIRECTIONS` definiert.

```
var DIRECTIONS = [
    {x:-1,y:0},  // left
    {x:-1,y:-1}, // top left
    {x:0,y:-1},  // top
    {x:1,y:-1},  // top right
    {x:1,y:0},   // right
    {x:1,y:1},   // bottom right
    {x:0,y:1},   // bottom
    {x:-1,y:1}   // bottom left
];
```

definitions.js (komplett)

Das Modul Tools

Zusätzlich führen wir ein weiteres neues Modul hinzu, welches Hilfsfunktionen aufnimmt. Die erste Funktion gibt uns ganzzahlige Zufallszahlen bis zu einer definierten Obergrenze zurück. Davon werden wir im weiteren Verlauf oft Gebrauch machen. Die in Javascript eingebaute Zufallsfunktion liefert nämlich nur Fließkommazahlen (von 0.0 bis kurz vor 1.0). `tools_random(3)` gibt uns dafür beispielsweise Zufallszahlen mit den Werten 0, 1 und 2 zurück.

```
// liefert ganzzahlen von 0 bis max-1
function tools_random(max){
    return Math.floor(Math.random()*max);
}
```

tools.js (komplett)

eprobot.js erweitern

Nun kommen wir zur Erweiterung der Eprobots-Klasse.

```
class Eprobot{
    constructor(s) {
        this.position_x = null;
```

```
        this.position_y = null;

        this.s = s;
    }

    get_color(){
        return "#f50000";
    }

    step(){
        let moveval = tools_random(9);
        if (moveval<DIRECTIONS.length){
            let vec = DIRECTIONS[moveval];
            let movepos_x = this.position_x + vec.x;
            let movepos_y = this.position_y + vec.y;

            let t_new = this.s.world.get_terrain(movepos_x, movepos_y);
            if (t_new.slot_object==null){
                this.s.world.world_move(this, this.position_x, this.position_y, movepos_x,
         movepos_y);
            }else{
                console.log("autsch!");
            }
        }
    }
}
```

eprobot.js erweitern

Zuerst holen wir das Speichern der Referenz des Simulationsobjektes nach. Denn dieses brauchen wir in der sogenannten `step`-Methode. Die Idee ist Folgende: Der Eprobot hat eine Methode die ihn veranlasst etwas zu tun und dabei evt. seinen Zustand zu verändern. Unser Ziel ist ja, dass der Eprobot über die Welt wandert. Genauer gesagt soll er pro Simulationsschritt in eine der acht möglichen Richtungen gehen oder nichts tun. Es gibt also neun verschiedene Aktionen. Daher brauchen wir zuerst eine Zufallszahl.

`let moveval = tools_random(9);` liefert uns genau eine Variante dieser neun verschiedenen Aktionen (0 bis 8). Wir definieren, dass der Eprobot bei 0 bis 7 in eine Richtung laufen soll und bei einer 8 stehen bleiben soll. Dies prüft die Bedingung `if (moveval<DIRECTIONS.length)`, denn `DIRECTIONS.length` ist 8 und nur bei Werten kleiner als 8 soll er sich bewegen.

Soll eine Bewegung stattfinden, kann die neue Position des Eprobots basierend auf seiner aktuellen Position und der gewählten Bewegungsrichtung errechnet werden.

Als nächstes holen wir uns dann das Terrain-Objekt an der Zielposition und prüfen, ob es auch leer ist, sich dort also kein anderer Eprobot oder eine Barriere befindet. Ist das Terrain-Objekt leer, kann sich der Eprobot an diese Stelle bewegen. Dazu verwenden wir eine neue World-Methode auf die ich gleich eingehen werde.

world.js erweitern

```
class World {

    constructor(s, width, height) {
        console.log("World constructor");

        this.s = s;

        // init
        this.worldarr = new Array(width);
        for (var x=0;x<this.worldarr.length;x++){
            this.worldarr[x] = new Array(height);
            for (var y=0;y<height;y++){
                this.worldarr[x][y] = new Terrain(s, x, y);
            }
        }
    }
```

```
    get_terrain(x, y){
        return this.worldarr[x][y];
    }

    world_set(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.slot_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

    world_move(o, old_pos_x, old_pos_y, new_pos_x, new_pos_y){
        var t = this.get_terrain(new_pos_x, new_pos_y);
        t.slot_object = o;
        o.position_x = new_pos_x;
        o.position_y = new_pos_y;
        t.prepare_paint();

        var t_old = this.get_terrain(old_pos_x, old_pos_y);
        t_old.slot_object = null;
        t_old.prepare_paint();
    }

}
```

world.js erweitern

`world_move(o, old_pos_x, old_pos_y, new_pos_x, new_pos_y)` ist ähnlich aufgebaut wie world_set(). Ihre Aufgabe ist es einen Eprobot `o` von `old_pos_x` und `old_pos_y` nach `new_pos_x` und `new_pos_y` zu setzen. Zusätzlich kümmert es sich auch um das Neuzeichnen/Aktualisieren der beiden Terrain-Felder (alte Position und neue Position).

Game Loop

Kommen wir nun zum Game Loop um die ganze Sache in's Rollen zu bringen. Der Game Loop soll mit dem Start/Stopp-Button gestartet und angehalten werden können. Dazu brauchen wir einen Eventlistener auf dem Button der auf Klick-Events reagiert: `document.getElementById("btn_start_stop").addEventListener("click", startstop_button_handler);`. Bei einem Klick-Event wird die Funktion `startstop_button_handler` aufgerufen. Ganz oben in der main.js definieren wir dazu eine Zustandsvariable namens `running` die standardmäßig den Wert false hat. Die Aufgabe der Funktion startstop_button_handler ist es nun, genau diesen Wert zu "toggeln", also von false auf true und von true auf false zu setzen. Wird running von false auf true gesetzt, wird der Game Loop mit dem Aufruf der Funktion `simulation_loop` gestartet.

```
document.addEventListener("DOMContentLoaded", function() {
    console.log("Eprobots JS Tutorial - Start");
    let running = false;

    let canvas = document.getElementById("canvas");
    var simulation = new Simulation(canvas);

    function simulation_loop(){
        let steptime_start = new Date().getTime();

        simulation.simulation_step();
        simulation.drawer.paint_fast();

        let steptime_end = new Date().getTime();
        let current_frame_time = steptime_end - steptime_start;

        if (running) {
            let st = settings.frame_time - current_frame_time;
            setTimeout(()=>{simulation_loop()}, st);
        }
```

```
    }

    function startstop_button_handler(){
        console.log("start stop button clicked");
        running = !running;
        if (running){
            simulation_loop();
        }
    }

    document.getElementById("btn_start_stop").addEventListener("click", startstop_button_handler);
});
```

main.js erweitern

Die Hauptaufgabe von 'simulation_loop' ist es, die Methode `simulation_step` in der Simulationsklasse aufzurufen, die später noch definiert wird und anschließend die Welt zu zeichnen.

Framerate

Da wir eine fortlaufende Simulation haben wollen, muss dieser Prozess nun immer wieder von neuem ausgelöst werden. Nun könnte man eine Endlosschleife bauen, was aber viel zu schnell wäre. Daher ist die Strategie simulation_loop() nach einer gewissen Wartezeit wieder neu aufrufen zu lassen, was mit `setTimeout` erledigt werden kann. Bleibt noch die Frage nach der Wartezeit die in Millisekunden angegeben wird.

In der settings.js definieren wir eine Wartezeit von 100ms pro Frame. Dies entspricht einer Framerate von 10frames pro Sekunde.

```
var settings = {
    pixel_size: 8,
    frame_time: 100
}
```

settings.js erweitern

Dabei bitte an das Komma nach `pixel_size: 8` denken. Das trifft auf alle Erweiterungen in der settings.js zu.

Zurück zur Berechnung der Wartezeit. Das Berechnen von `simulation_step` und das Zeichnen der Welt dauert ja auch eine Weile. Daher wird diese Zeit gemessen und von der gewünschten frame_time abgezogen. Dauert das Berechnen und Zeichnen also zum Beispiel 10ms, braucht nur noch 90ms bis zum nächsten Frame gewartet werden. Somit ist eine relativ stabile Framerate sichergestellt. Erwähnenswert ist noch, dass der Aufruf des nächsten Simulationsschrittes noch von der Variablen `running` abhängig gemacht wird. Klickt man also in der Zwischenzeit auf den Start/Stopp-Button, erhält running den Wert false und die Simulation stoppt.

simulation.js erweitern

Die simulation.js benötigt erfreulicherweise nur wenig Anpassungen. Aus der lokalen Variablen eprobot wird mit `this.eprobot` eine Instanzvariable gemacht. Somit können wir auch in anderen Methoden darauf zugreifen, also auch von der neuen Methode `simulation_step` aus. Diese hat hier nichts weiter zu tun, als die `step()`-Methode des einen Eprobots aufzurufen.

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
```

```
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.eprobot = new Eprobot(this);
        this.world.world_set(this.eprobot, 10, 10);

        this.drawer.paint_fast();
    }

    simulation_step(){
        this.eprobot.step();
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

terrain.js erweitern

Eine kleine Änderung ist noch in der terrain.js notwendig. Es fehlte noch der Fall in `get_color()`, dass kein slot_object gesetzt ist. Wenn ein Eprobot umherwandert, ist aber genau das der Fall. Auf dem Terrain-Feld der alten Position befindet sich dann kein Eprobot mehr. Welche Farbe soll dann zurückgegeben werden? Natürlich die Hintergrundfarbe, in unserem Fall also schwarz (“#000000”).

```
class Terrain {

    constructor(s, x, y) {
        this.s = s;

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

        this.dirty = false;

        this.slot_object = null;
    }

    prepare_paint(){
        if (this.dirty == false){
            this.dirty = true;
            this.s.drawer.paintlist.push(this);
        }
    }

    get_color(){
        if (this.slot_object){
```

```
            return this.slot_object.get_color();
        }else{
            return "#000000";
        }
    }
}
```

terrain.js erweitern

Abschluss

Am Ende dieses langen Kapitels müssen wir nur noch die neuen Dateien in die index.html eintragen, können den Browser aktualisieren und uns über einen laufenden Eprobot freuen.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="definitions.js"></script>
    <script src="tools.js"></script>
    <script src="simulation.js"></script>
    <script src="drawer.js"></script>
    <script src="settings.js"></script>
    <script src="world.js"></script>
    <script src="terrain.js"></script>
    <script src="barrier.js"></script>
    <script src="eprobot.js"></script>

    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

7. Mehrere Eprobots

Da wir es nun geschafft haben einen Eprobot durch die Gegend wandern zu lassen, ist es auch keine Hürde viele Eprobots laufen zu lassen. Dazu müssen wir die simulation.js ein wenig anpassen. Statt der Variable `this.eprobot` aus dem vorherigen Kapitel, nehmen wir nun für diesen Zweck ein Array: `this.list_eprobots = [];`, welches anfangs leer ist.

Als nächstes brauchen wir eine For-Schleife in der wir unsere Eprobots erzeugen. Die Start-Positionen der Eprobots sollen zufällig sein. Also werden innerhalb der For-Schleife zuerst die Positionskoordinaten ausgewürfelt. Statt jetzt einfach den Eprobot an der entsprechenden Stelle in die Welt zu setzen, sollten wir vorher abchecken, ob sich an dieser Stelle auch nichts Anderes befindet (keine Barriere und kein anderer Eprobot): `if (this.world.get_terrain(rand_x, rand_y).slot_object==null)`. Ist das der Fall, kann der Eprobot an diese Stelle gesetzt werden. Wenn nicht, dann nicht.

```
class Simulation {
```

```
    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.list_eprobots = [];
        for (let i=0;i<10;i++){
            let eprobot = new Eprobot(this);
            let rand_x = tools_random(this.world_width_visible);
            let rand_y = tools_random(this.world_height_visible);

            if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                this.world.world_set(eprobot, rand_x, rand_y);
                this.list_eprobots.push(eprobot);
            }
        }

        this.drawer.paint_fast();
    }

    simulation_step(){
        for (let o of this.list_eprobots) {
            o.step();
        }
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

Das war die Initialisierung. Der zweite Schritt ist einfacher. Statt die `step`-Methode des einen Eprobots aufzurufen, iterieren wir über die Liste der Eprobots und rufen die `step`-Methoden der jeweiligen Eprobots aus der Liste auf.

8. Plants und Plant seeding

In diesem Kapitel wollen wir eine weitere Art von Objekten einführen: Energieobjekte, die später von der Eprobots konsumiert werden können. Ich nenne sie “Plants”. Plants sollen fortlaufend verteilt werden (seeding) und es soll für diese Objekte eine Maximalanzahl geben. Wird eines oder mehrere gefressen, wird die Anzahl vor dem nächsten Simulationsschritt wieder ausgeglichen. Die Energie-Objekte legen wir in einen eigenen “Slot” im Terrain-Objekt namens `energy_object`.

Dafür schreiben wir zuerst die Plant-Klasse in der neuen Datei plant.js.

```
class Plant{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;

        this.s = s;
    }

    get_color(){
        return "#00ff00";
    }
}
```

plant.js (komplett)

Die Farbe dieser Objekte soll grün sein (“#00ff00”).

settings.js erweitern

```
var settings = {
    pixel_size: 8,
    frame_time: 100,
    number_of_plants: 10
}
```

settings.js erweitern

Weiterhin wollen wir die Gesamtanzahl der Plant-Objekte in der Variablen `number_of_plants` im Settings-Modul festhalten.

simulation.js erweitern

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.plant_counter = 0;

        this.list_eprobots = [];
        for (let i=0;i<10;i++){
            let eprobot = new Eprobot(this);
            let rand_x = tools_random(this.world_width_visible);
```

```
            let rand_y = tools_random(this.world_height_visible);

            if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                this.world.world_set(eprobot, rand_x, rand_y);
                this.list_eprobots.push(eprobot);
            }
        }

        this.drawer.paint_fast();
    }

    seed_plants(){
        if (this.plant_counter<settings.number_of_plants){
            let pc = this.plant_counter;
            for (let i=0;i<settings.number_of_plants-pc;i++){
                let p = new Plant(this);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                let t = this.world.get_terrain(rand_x, rand_y);
                if (t.energy_object==null && t.slot_object==null){
                    this.world.world_set_energy(p, rand_x, rand_y);
                    this.plant_counter++;
                }
            }
        }
    }

    simulation_step(){
        this.seed_plants();

        for (let o of this.list_eprobots) {
            o.step();
        }
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

Im Konstruktor der Simulations-Klasse definieren wir mit `this.plant_counter = 0` unseren Counter, der mit 0 initialisiert wird. Die Methode `simulation_step` wird so erweitert, dass vor dem Durchrechnen der Eprobots-Liste eine andere Methode namens `seed_plants` aufgerufen wird. In der Methode wird zunächst geprüft, ob der Wert in `plant_counter` kleiner ist als die gewünschte Menge die im Settings-Modul unter `number_of_plants` definiert ist. Der anschließende For-Loop läuft genau so oft, wie zusätzliche Plant-Objekte erzeugt werden müssen, um die gewünschte Menge zu erreichen. Die Verteilung soll zufällig erfolgen, also wird wieder eine zufällige Position erzeugt und es wird geprüft, ob das Terrain-Objekt auch leer ist. Leer bedeutet hier, dass weder ein `slot_object` noch ein `energy_object` gesetzt ist.

Anmerkung: Es könnte sein, dass in einem Simulationsschritt ausnahmsweise mal nicht die vollen 10 Plants in der Welt sind, aber das ist nicht weiter schlimm für die Eprobots und wird in einem nachfolgenden Simulationsschritt ausgeglichen.

terrain.js erweitern

```
class Terrain {

    constructor(s, x, y) {
        this.s = s;

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

        this.dirty = false;

        this.slot_object = null;
        this.energy_object = null;
    }

    prepare_paint(){
        if (this.dirty == false){
            this.dirty = true;
            this.s.drawer.paintlist.push(this);
        }
    }

    get_color(){
        if (this.slot_object){
            return this.slot_object.get_color();
        }else{
            if (this.energy_object){
                return this.energy_object.get_color();
            }else{
                return "#000000";
            }
        }
    }
}
```

terrain.js erweitern

Die Datei terrain.js wird so erweitert, dass sie ein neues Feld namens `energy_object` erhält. Außerdem wird die Methode `get_color` angepasst, so dass sie die Objekte in `energy_object` auch berücksichtigt.

world.js erweitern

```
class World {

    constructor(s, width, height) {
        console.log("World constructor");

        this.s = s;

        // init
        this.worldarr = new Array(width);
        for (var x=0;x<this.worldarr.length;x++){
            this.worldarr[x] = new Array(height);
            for (var y=0;y<height;y++){
                this.worldarr[x][y] = new Terrain(s, x, y);
            }
        }
    }

    get_terrain(x, y){
        return this.worldarr[x][y];
```

```
    }

    world_set(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.slot_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

    world_set_energy(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.energy_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

    world_move(o, old_pos_x, old_pos_y, new_pos_x, new_pos_y){
        var t = this.get_terrain(new_pos_x, new_pos_y);
        t.slot_object = o;
        o.position_x = new_pos_x;
        o.position_y = new_pos_y;
        t.prepare_paint();

        var t_old = this.get_terrain(old_pos_x, old_pos_y);
        t_old.slot_object = null;
        t_old.prepare_paint();
    }

}
```

world.js erweitern

Die world.js erhält für `energy_object` ebenfalls eine neue Methode namens `world_set_energy`, die sich analog zu `world_set` verhält.

Abschluss

Im Anschluss müssen wir die neue Datei plant.js noch der index.html hinzufügen. Im Browser sollten wir nach einem Reload 10 Pflanzen verteilt in der Welt sehen. Allerdings gibt es noch keine Interaktion. Dazu mehr im nächsten Kapitel.

```
<!doctype html>
<html lang="de">
<head>
    <meta charset="utf-8">

    <title>Eprobots JS Tutorial</title>
    <meta name="description" content="Eprobots JS Tutorial">
</head>

<body>
    <h1>Eprobots JS</h1>
    <div>
        <canvas id="canvas"  width="800" height="400" style="background-color:#000000;"></canvas>
    </div>
    <div>
        <button type="button" id="btn_start_stop">Start/Stopp</button>
    </div>

    <script src="definitions.js"></script>
    <script src="tools.js"></script>
    <script src="simulation.js"></script>
```

```
    <script src="drawer.js"></script>
    <script src="settings.js"></script>
    <script src="world.js"></script>
    <script src="terrain.js"></script>
    <script src="barrier.js"></script>
    <script src="eprobot.js"></script>
    <script src="plant.js"></script>

    <script src="main.js"></script>
</body>
</html>
```

index.html erweitern

9. Plants konsumieren

Wieder ein kurzes Kapitel. Es geht darum, dass die Eprobots die Plants auch konsumieren können sollen.

Dazu ist nur eine kleine Änderung in der `step`-Methode der Eprobot-Klasse notwendig. Nachdem der Eprobot einen Schritt gelaufen ist, muss auf dem neuen Terrain-Objekt nachgesehen werden, ob sich auf ihm ein Plant-Objekt befindet. Ist dies der Fall wird eine Meldung ausgegeben, das Plant-Objekt wird entfernt und der `plant_counter` um 1 verringert.

Dadurch das `plant_counter` verringert wird, werden beim nächsten Simulationsschritt automatisch neue Plant-Objekte erzeugt.

```
class Eprobot{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;

        this.s = s;
    }

    get_color(){
        return "#f50000";
    }

    step(){
        let moveval = tools_random(9);
        if (moveval<DIRECTIONS.length){
            let vec = DIRECTIONS[moveval];
            let movepos_x = this.position_x + vec.x;
            let movepos_y = this.position_y + vec.y;

            let t_new = this.s.world.get_terrain(movepos_x, movepos_y);
            if (t_new.slot_object==null){
                this.s.world.world_move(this, this.position_x, this.position_y, movepos_x,
         movepos_y);
                if (t_new.energy_object){
                    console.log("mampf");
                    t_new.energy_object = null;
                    this.s.plant_counter--;
                }
            }else{
                console.log("autsch!");
            }
        }
    }
}
```

eprobot.js erweitern

10. Limitierte Lebenszeit

Langsam nähern wir uns der Simulation einer Welt mit künstlichen Lebewesen. Dafür ist es aber auch notwendig, dass unsere Lebewesen, die Eprobots, einen Lebenszyklus haben, also auch irgendwann sterben. Für diesen Zweck bekommen die Eprobots eine neue Variable namens `age`. Diese wird in jedem Simulationsschritt um 1 erhöht (`this.age++` in der `step`-Methode). Außerdem definieren wir eine neue Variable namens `max_age` um die maximale Lebenszeit eines Eprobots auf einen festen Wert zu begrenzen. Dieser leitet sich aus einem neuen Settings-Wert namens `eprobots_max_age` ab. Damit nicht alle Eprobots gleichzeitig sterben, wird zur tatsächlichen Lebenszeit noch ein zufälliger Wert hinzuaddiert.

```
class Eprobot{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;

        this.s = s;

        this.age = 0;
        this.max_age = settings.eprobots_max_age + tools_random(100);
    }

    get_color(){
        return "#f50000";
    }

    get_max_age(){
        return this.max_age;
    }

    step(){
        let moveval = tools_random(9);
        if (moveval<DIRECTIONS.length){
            let vec = DIRECTIONS[moveval];
            let movepos_x = this.position_x + vec.x;
            let movepos_y = this.position_y + vec.y;

            let t_new = this.s.world.get_terrain(movepos_x, movepos_y);
            if (t_new.slot_object==null){
                this.s.world.world_move(this, this.position_x, this.position_y, movepos_x,
         movepos_y);
                if (t_new.energy_object){
                    console.log("mampf");
                    t_new.energy_object = null;
                    this.s.plant_counter--;
                }
            }else{
                console.log("autsch!");
            }
        }
        this.age++;
    }
}
```

eprobot.js erweitern

settings.js erweitern

In der settings.js führen wir die neue Variable `eprobots_max_age` ein und setzen sie auf den Wert 500.

```
var settings = {
    pixel_size: 8,
    frame_time: 100,
    number_of_plants: 10,
    eprobots_max_age: 500
}
```

settings.js erweitern

Listen-Management

Jetzt kommen wir zur simulation.js. Die Aufgabe ist es jetzt, diejenigen Eprobots aus unserer Eprobots-Liste auszusortieren, die das Ende ihrer Lebenszeit erreicht haben. Dinge einfach aus einer Liste rauszuschmeißen, über die man gerade iteriert ist aber nicht gerade optimal. Wir könnten nun die toten Eprobots in eine anderen Liste tun und diese Eprobots im Anschluss entfernen. Wir machen es aber noch ein wenig anders: Beim Prozessieren legen wir alle *noch lebenden* Eprobots in eine vor der Iteration neu angelegte Liste. Dort befinden sich dann nach der Iteration alle überlebenden Eprobots. Wenn wir diese Liste haben, können wir diese dann einfach unserer Variable `list_eprobots` zuweisen und wir haben das Ziel erreicht. Und das ohne die Eprobots während des Iterierens aus der Liste zu schmeißen oder im Anschluss eine weitere For-Schleife ausführen zu müssen.

Ob ein Eprobot noch lebt wird mit diesem Ausdruck überprüft: `o.age<o.get_max_age()`

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.plant_counter = 0;

        this.list_eprobots = [];
        for (let i=0;i<10;i++){
            let eprobot = new Eprobot(this);
            let rand_x = tools_random(this.world_width_visible);
            let rand_y = tools_random(this.world_height_visible);

            if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                this.world.world_set(eprobot, rand_x, rand_y);
                this.list_eprobots.push(eprobot);
            }
        }

        this.drawer.paint_fast();
    }

    seed_plants(){
        if (this.plant_counter<settings.number_of_plants){
            let pc = this.plant_counter;
            for (let i=0;i<settings.number_of_plants-pc;i++){
                let p = new Plant(this);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                let t = this.world.get_terrain(rand_x, rand_y);
                if (t.energy_object==null && t.slot_object==null){
                    this.world.world_set_energy(p, rand_x, rand_y);
                    this.plant_counter++;
                }
            }
        }
    }
```

```
    simulation_step(){
        this.seed_plants();

        let list_eprobots_next = [];
        for (let o of this.list_eprobots) {
            if (o.age<o.get_max_age()){
                o.step();
                list_eprobots_next.push(o);
            }else{
                console.log("dead");
                this.world.world_unset(o, o.position_x, o.position_y);
            }

        }

        this.list_eprobots = list_eprobots_next;
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

world.js erweitern

Bisher brauchten wir es nicht, jetzt benötigen wir aber eine Möglichkeit Eprobots aus der Welt zu entfernen. In der simulation.js haben wir diesen Methodenaufruf: `this.world.world_unset(o, o.position_x, o.position_y);` Diese Methode muss nun noch in der World-Klasse implementiert werden.

Anzumerken ist, dass auch beim Entfernen eine Zeichenoperation ausgelöst werden muss, weil sich der Zustand des Terrain-Felds ja ändert, also von z.B. rot auf schwarz wechselt.

```
class World {

    constructor(s, width, height) {
        console.log("World constructor");

        this.s = s;

        // init
        this.worldarr = new Array(width);
        for (var x=0;x<this.worldarr.length;x++){
            this.worldarr[x] = new Array(height);
            for (var y=0;y<height;y++){
                this.worldarr[x][y] = new Terrain(s, x, y);
            }
        }
    }

    get_terrain(x, y){
```

```
        return this.worldarr[x][y];
    }

    world_set(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.slot_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

    world_unset(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.slot_object = null;

        t.prepare_paint();
    }

    world_set_energy(o, x_pos, y_pos){
        var t = this.get_terrain(x_pos, y_pos);
        t.energy_object = o;
        o.position_x = x_pos;
        o.position_y = y_pos;

        t.prepare_paint();
    }

    world_move(o, old_pos_x, old_pos_y, new_pos_x, new_pos_y){
        var t = this.get_terrain(new_pos_x, new_pos_y);
        t.slot_object = o;
        o.position_x = new_pos_x;
        o.position_y = new_pos_y;
        t.prepare_paint();

        var t_old = this.get_terrain(old_pos_x, old_pos_y);
        t_old.slot_object = null;
        t_old.prepare_paint();
    }

}
```

world.js erweitern

11. Reproduktion

In diesem Kapitel kommt ein weiterer Aspekt einer biologischen Simulation zum Tragen: Reproduktion. Wenn die Eprobots etwas “fressen”, sollen sie Energie erhalten und diese soll zur Fortpflanzung genutzt werden können.

eprobot.js erweitern

Dazu erweitern wir die Eprobot-Klasse um eine Instanzvariable namens `energy` die mit 0 initialisiert wird. Weiterhin wird dieser Wert im Falle des Konsums eines Plant-Objektes um 1 erhöht (`step`-Methode).

```
class Eprobot{
    constructor(s) {
        this.position_x = null;
        this.position_y = null;

        this.s = s;

        this.age = 0;
        this.max_age = settings.eprobots_max_age + tools_random(100);
        this.energy = 0;
```

```
    }

    get_color(){
        return "#f50000";
    }

    get_max_age(){
        return this.max_age;
    }

    step(){
        let moveval = tools_random(9);
        if (moveval<DIRECTIONS.length){
            let vec = DIRECTIONS[moveval];
            let movepos_x = this.position_x + vec.x;
            let movepos_y = this.position_y + vec.y;

            let t_new = this.s.world.get_terrain(movepos_x, movepos_y);
            if (t_new.slot_object==null){
                this.s.world.world_move(this, this.position_x, this.position_y, movepos_x,
         movepos_y);
                if (t_new.energy_object){
                    console.log("mampf");
                    t_new.energy_object = null;
                    this.s.plant_counter--;
                    this.energy++;
                }
            }else{
                console.log("autsch!");
            }
        }
        this.age++;
    }
}
```

eprobot.js erweitern

simulation.js erweitern

In der simulation.js erweitern wir die `simulation_step`-Methode. Nachdem dort die `step`-Methode eines lebenden Eprobots ausgeführt wurde, soll er die Möglichkeit zur Fortpflanzung erhalten. Dies soll in die neue Methode `try_fork(o, list_eprobots_next)` ausgelagert werden. Der Methode wird die Variable `list_eprobots_next` mitgegeben. Die Methode erzeugt im Falle einer erfolgreichen Fortpflanzung den neuen Eprobot und kann ihn gleich in diese Liste einfügen, denn wir brauchen ja auch die neuen Eprobots in unserer Liste der aktiven Eprobots.

Die `try_fork`-Methode macht im wesentlichen dies: Zuerst wird geprüft, ob die Energie des Eprobots größer als 0 ist, weil er nur dann das recht hat sich fortzupflanzen. Wenn er sich fortpflanzen darf, wird eine zufällige Position um den Eprobot herum ausgewürfelt. Dann wird nachgesehen, ob das Terrain-Objekt an dieser Position auch leer ist. Ist auch das der Fall, findet die Reproduktion statt, der neue Eprobot wird in die Welt gesetzt und der Liste der aktiven Eprobots hinzugefügt. Ganz wichtig: Am Ende muss der Energie-Wert des Eltern-Eprobots wieder um 1 vermindert werden. Fortpflanzung verbraucht Energie. Andernfalls hätten wir eine Reproduktionsexplosion.

Was aber wenn das Feld besetzt ist? Dann wird kein Eprobot erzeugt. Das kann man als Regel in unserer Eprobots-Welt durchaus akzeptieren und die Eprobots werden sich später auch an diesen Umstand anpassen. Es sind hier aber auch viele Erweiterungen denkbar z.B. in einer Schleife mehrere Felder durchprobieren. Oder auch weiter entfernte Felder. Der Phantasie sind keine Grenzen gesetzt.

Die index.html muss ausnahmeweise mal nicht erweitert werden. Nach einem Reload im Browser wird man wahrscheinlich sich vermehrende Eprobots sehen können. Es kann aber auch sein, dass die “Population” ausstirbt. Dann einfach neu laden und erneut probieren.

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.plant_counter = 0;

        this.list_eprobots = [];
        for (let i=0;i<10;i++){
            let eprobot = new Eprobot(this);
            let rand_x = tools_random(this.world_width_visible);
            let rand_y = tools_random(this.world_height_visible);

            if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                this.world.world_set(eprobot, rand_x, rand_y);
                this.list_eprobots.push(eprobot);
            }
        }

        this.drawer.paint_fast();
    }

    seed_plants(){
        if (this.plant_counter<settings.number_of_plants){
            let pc = this.plant_counter;
            for (let i=0;i<settings.number_of_plants-pc;i++){
                let p = new Plant(this);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                let t = this.world.get_terrain(rand_x, rand_y);
                if (t.energy_object==null && t.slot_object==null){
                    this.world.world_set_energy(p, rand_x, rand_y);
                    this.plant_counter++;
                }
            }
        }
    }

    simulation_step(){
        this.seed_plants();

        let list_eprobots_next = [];
        for (let o of this.list_eprobots) {
            if (o.age<o.get_max_age()){
                o.step();
                list_eprobots_next.push(o);
                this.try_fork(o, list_eprobots_next);
            }else{
                console.log("dead");
                this.world.world_unset(o, o.position_x, o.position_y);
            }

        }
```

```
        this.list_eprobots = list_eprobots_next;
    }

    try_fork(o, list_eprobots_next){
        if (o.energy>0){
            let spreadval = tools_random(8);
            let vec = DIRECTIONS[spreadval];
            let spreadpos_x = o.position_x + vec.x;
            let spreadpos_y = o.position_y + vec.y;
            let spreadterrain = this.world.get_terrain(spreadpos_x, spreadpos_y);
            if (spreadterrain.slot_object==null){
                console.log("spread");
                let eprobot = new Eprobot(this);
                this.world.world_set(eprobot, spreadpos_x, spreadpos_y);
                list_eprobots_next.push(eprobot);
                o.energy--;
            }
        }
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

12. Eprobot counter und Seeding im Game Loop

Aus dem letzten Kapitel habe ich am Schluss vom Fall gesprochen, dass die Population aussterben kann. Das kann passieren, aber jetzt wäre es ja schön, wenn die Simulation selbst dies bemerken würde und wieder ein erneutes “Seeding” von Eprobots veranlassen würde, sprich: das initiale Aussetzen von Eprobots.

Genau darum kümmert sich dieses Kapitel. Dazu brauchen wir in der Simulations-Klasse zuerst einen `eprobot_counter` analog zum `plant_counter` der mit 0 initialisiert wird. Desweiteren müssen wir das initiale Erzeugen von Eprobots aus dem Konstruktor der Simulationsklasse in eine eigene Methode namens `seed_eprobots` verschieben. Diese Methode wird dann von der Methode `simulation_step` aufgerufen.

`seed_eprobots` enthält die gleiche For-Schleife die vorher im Konstruktor stand und erzeugt 10 Eprobots an zufälligen Positionen. Allerdings wird die For-Schleife nur dann ausgeführt, wenn `eprobot_counter` den Wert 0 enthält. Außerdem müssen wir daran denken, unseren `eprobot_counter` auch zu pflegen, also sollten wir ihn auch tatsächlich inkrementieren (erhöhen um 1) nachdem ein Eprobot erzeugt wurde. Das wird in der Methode `seed_eprobots` getan, wie auch in der Methode `try_fork`. Irgendwo muss der Counter natürlich auch dekrementiert (vermindern um 1) werden. Dies machen wir in der Methode `simulation_step` im Zweig für die toten Eprobots.

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
```

```
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.plant_counter = 0;

        this.list_eprobots = [];
        this.eprobot_counter = 0;

        this.drawer.paint_fast();
    }

    seed_eprobots(){
        if (this.eprobot_counter==0){
            for (let i=0;i<10;i++){
                let eprobot = new Eprobot(this);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                    this.world.world_set(eprobot, rand_x, rand_y);
                    this.list_eprobots.push(eprobot);
                    this.eprobot_counter++;
                }
            }
        }
    }

    seed_plants(){
        if (this.plant_counter<settings.number_of_plants){
            let pc = this.plant_counter;
            for (let i=0;i<settings.number_of_plants-pc;i++){
                let p = new Plant(this);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                let t = this.world.get_terrain(rand_x, rand_y);
                if (t.energy_object==null && t.slot_object==null){
                    this.world.world_set_energy(p, rand_x, rand_y);
                    this.plant_counter++;
                }
            }
        }
    }

    simulation_step(){
        this.seed_plants();
        this.seed_eprobots();

        let list_eprobots_next = [];
        for (let o of this.list_eprobots) {
            if (o.age<o.get_max_age()){
                o.step();
                list_eprobots_next.push(o);
                this.try_fork(o, list_eprobots_next);
            }else{
                console.log("dead");
```

```
                this.world.world_unset(o, o.position_x, o.position_y);
                this.eprobot_counter--;
            }

        }

        this.list_eprobots = list_eprobots_next;
    }

    try_fork(o, list_eprobots_next){
        if (o.energy>0){
            let spreadval = tools_random(8);
            let vec = DIRECTIONS[spreadval];
            let spreadpos_x = o.position_x + vec.x;
            let spreadpos_y = o.position_y + vec.y;
            let spreadterrain = this.world.get_terrain(spreadpos_x, spreadpos_y);
            if (spreadterrain.slot_object==null){
                console.log("spread");
                let eprobot = new Eprobot(this);
                this.world.world_set(eprobot, spreadpos_x, spreadpos_y);
                list_eprobots_next.push(eprobot);
                this.eprobot_counter++;
                o.energy--;
            }
        }
    }

    add_borders(){
        for (let x=0;x<this.world_width;x++){
            let b = new Barrier(this);
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

Abschluss

Nach einem Browser-Reload und dem Starten der Simulation, sollte von nun an immer etwas im Canvas los sein.

13. OISC!

In diesem Kapitel geht es an’s Eingemachte. Wir werden die Simulation zum Leben erwecken indem wir Verhalten evolvieren lassen. Momentan werden die Bewegungswerte der Eprobots zufällig erzeugt. Das wollen wir mit etwas Besserem ersetzen. Dieses etwas sollte auch vererbbar und evolvierbar sein. Man sollte es in Form einer Datenstruktur vererben und Mutationen darauf anwenden können.

An dieser Stelle gibt es verschiedene Ansätze, zum Beispiel Neuronale Netze. Ich verwende hier aber einen anderen Ansatz. Die Eprobots sollen kleine Programme enthalten, die in jedem Schritt ausgeführt werden. Diese Programme haben natürlich auch Datenspeicher. Als Ausgabewert für die Bewegung verwenden wir dann einfach ein bestimmtes Element dieses Datenspeichers.

Jetzt stellt sich die Frage was für Programme man verwendet und wie man sie ausführt. Man könnte auch Javascript-Code verwenden, doch ist es schwierig so etwas zumindest auf der Text-Ebene zu mutieren. Stattdessen habe ich mich für sogenannte OISC-Programme entschieden. OISC steht für “One Instruction Set Computer”. Heutige Prozessoren haben viele oder sehr viele Befehle (RISC und CISC). Es ist aber theoretisch und praktisch möglich einen Prozessor zu entwerfen, der nur einen einzigen Befehl kennt und mit dem man trotzdem alle Arten von informationstechnischen Aufgaben realisieren kann, der also turing-vollständig ist. Dieser Befehl muss relativ mächtig sein und verschiedene Funktionalitäten in sich vereinen: also muss z.B. einen bedingten Sprung ausführen, etwas berechnen und etwas kopieren können. Genau das leistet so ein OISC-Befehl. Der Vorteil dieses Ansatzes ist es, dass man für die Ausführung eines OISC-Programmes gar keine Befehle mehr dekodieren muss, weil es ja nur einen einzigen Befehl gibt.

Es gibt verschiedene Implementierungsvarianten. Ich habe mich für diese entschieden: Subtract and branch if less than or equal to zero:
https://en.wikipedia.org/wiki/One_instruction_set_computer#Subtract_and_branch_if_less_than_or_equal_to_zero
Der Befehl hat drei Parameter und hinter dem Link gibt es eine Implementierungsvariante die ich leicht abgewandelt übernommen habe.

So weit so gut. Diesen Prozessor haben wir natürlich nicht in Hardware, sondern wir simulieren/emulieren bzw. virtualisieren diesen Prozessor. Aber auch dabei kommt es uns zugute, dass das “parsen” und verarbeiten dieser Programme sehr einfach ist. Um genau zu sein, handelt es sich dabei nur um eine einfache Funktion die den Programm- und Datenspeicher entgegen nimmt und den Code in einer while-Schleife ausführt, wobei man natürlich auch auf diverse Abbruch-Bedingungen achten muss, damit keine Endlosschleifen produziert werden.

Hier der Pseudocode dieses Befehls:

```
subleq a, b, c   ; Mem[b] = Mem[b] - Mem[a]
                 ; if (Mem[b] ≤ 0) goto c
```

Diesen Code bitte nicht in das Projekt einfügen. a, b und c sind unsere drei Parameter. Der “Prozessor” macht also Folgendes: Der Wert an der Adresse a wird vom Wert an der Adresse b abgezogen. Der neue Wert wird dann an der Adresse b gespeichert. Anschließend wird überprüft, ob der Wert an der Adresse b kleiner oder gleich 0 ist. Ist das der Fall geht der Programmablauf an der Adresse c weiter.

Weiterhin machen wir es so, dass wir getrennte Programm- und Datenspeicher verwenden. Man kann es aber auch genauso gut mit nur einem großen Programm/Datenspeicher realisieren.

tools.js erweitern

Wir erweitern nun also tools.js um eine neue Funktion namens `tools_compute`. Die Argumente sind `program`, `data` und `PS`. Für program und data werden Arrays mit Integer-Werten erwartet. Das ist später der Programm- und Datenspeicher eines Eprobots. PS ist ein Wert der angibt, nach wie vielen Verarbeitungsschritten das Programm spätestens abbrechen soll. Es ist wichtig hier einen Wert zu haben, auch wenn er relativ hoch ist. So schützt man sich vor Endlosschleifen.

Was noch an `tools_compute` erwähnenswert ist: Die Adresswerte für a, b und c werden mittels der Modulo-Operation so gemappt, das immer gültige Adresswerte entstehen! Das verbessert die Wahrscheinlichkeit für funktionsfähige Programme enorm.

`tools_random2` ist eine weitere kleine Hilfsfunktion im Stile von `tools_random`.

Mutation

Die Funktion `tools_mutate` erhält ein Programm in Form eines Arrays und liefert ein leicht modifiziertes Programm zurück. Die Modifizierung ist dabei vollkommen zufällig. `mutate_possibility` ist ein float-Wert der pro Datenspeicherzelle angibt, wie hoch die Wahrscheinlichkeit ist, dass sie überhaupt modifiziert wird. 0.0 bedeutet 0% Wahrscheinlichkeit und 1.0 100% Wahrscheinlichkeit. Ein guter Wert ist zum Beispiel 0.01, also eine 1%-Wahrscheinlichkeit oder anders gesagt: ungefähr alle 100 Zellen gibt es eine Mutation. Kommt es dann zu einer Mutation wird ein weiterer ganzzahliger Zufallswert zu dem vorhandenen Wert addiert. `mutate_strength` gibt dabei an, wie groß oder wie klein dieser Zufallswert werden kann. Dabei ist es so dass bei einem Wert von

z.B. 400, Werte von -200 bis 200 erzeugt werden. Es können also auch negative Werte addiert werden. Der neue Wert kann also maximal um +-200 vom alten Wert abweichen.

```
// liefert ganzzahlen von 0 bis max-1
function tools_random(max){
    return Math.floor(Math.random()*max);
}

// liefert ganzzahlen von min bis max-1
function tools_random2(min, max){
    var delta = max - min;
    return Math.floor(Math.random()*delta)+min;
}

function tools_compute(program, data, PS) {
    var program_counter = 0;
    var step_counter = 0;
    var a, b, c;

    while (program_counter >= 0 && (program_counter + 2) < program.length && step_counter < PS) {
        a = program[program_counter];
        b = program[program_counter + 1];
        c = program[program_counter + 2];

        a = a % data.length;
        b = b % data.length;
        c = c % program.length;

        if (a < 0 || b < 0) {
            program_counter = -1;
        }else{
            data[b] = data[b] - data[a];
            if (data[b] > 0) {
                program_counter = program_counter + 3;
            } else {
                program_counter = c;
            }
        }
        step_counter++;
    }
    return step_counter;
}

function tools_mutate(mutate_possibility, mutate_strength, memory) {
    var new_memory = [];
    for (var i=0;i<memory.length;i++){
        var copyval = memory[i];
        if (Math.random() < mutate_possibility) {
            copyval = copyval + tools_random(mutate_strength) - (mutate_strength / 2);
        }
        new_memory.push(copyval);
    }

    return new_memory;
}
```

tools.js erweitern

settings.js erweitern

Als nächstes gibt es einige neue Parameter für die settings.js. `programm_length` und `data_length` beschreiben die Größen des Programm- und Datenspeichers jedes Eprobots. `program_steps_max` ist die erwähnte Maximalanzahl an Verarbeitungsschritten, damit es nicht zu Endlosschleifen kommt. `mutate_possibility` und `mutate_strength` sind die Mutationsparameter für `tools_mutate`. `eprobots_max_individuals` schließlich ist eine Obergrenze für die Anzahl der Eprobots, was wichtig ist, da wir sonst in kurzer Zeit nur noch Eprobots auf dem Canvas sehen

würden. Die `frame_time` habe ich hier auf 20ms verringert. Ich finde diese Geschwindigkeit interessanter. Das ist aber eine Geschmacksfrage.

```
var settings = {
    pixel_size: 8,
    frame_time: 20,
    number_of_plants: 200,
    eprobots_max_age: 500,
    program_length: 100,
    data_length: 50,
    program_steps_max: 10000,
    mutate_possibility: 0.01,
    mutate_strength: 400,
    eprobots_max_individuals: 100
}
```

settings.js erweitern

eprobot.js erweitern

Als nächstes wenden wir uns den Anpassungen an der Eprobots-Klasse zu.

Zuerst sollten wir den Konstruktor so anpassen, das ein Eprobot mit Programm- und Datenspeicher initialisiert wird. Programm- und Datenspeicher werden dann in internen Feldern zur weiteren Verarbeitung gesichert. Wichtig an diese Stelle zu erwähnen ist, dass sich der Programmspeicher nie ändert, der Datenspeicher aber sehr wohl. Später wollen wir Programm- und Datenspeicher an die Nachkommen “vererben”. Daher wäre es vielleicht nicht sehr realistisch, den Datenspeicher, der sich ja im Laufe des Lebens eines Eprobots verändert, zu vererben (es sei denn wir sind Fans von Epigenetik). Besser wäre es den *initialen* Datenspeicher zu vererben. Daher arbeitet der Eprobot selbst auf einer Kopie des Datenspeichers namens `working_data`, die zu diesem Zweck im Konstruktor erstellt wird. Der initiale Datenspeicher wird zur Vererbung aufbewahrt.

In der `step`-Methode erhalten wir zukünftig unseren Aktionsswert nicht mehr vom Zufall, sondern lassen ihn uns von einer neuen Methode namens `get_output_OISC` geben, die wir noch definieren werden.

Die `get_output_OISC`-Methode ruft schließlich `tools_compute` mit dem Programm- und Datenspeicher des Eprobots auf. Dann wird der Ausgabewert für die Bewegung aus der ersten Speicherzelle des Datenspeichers entnommen. Man könnte den Wert auch von jeder anderen Stelle nehmen.

Im Anschluss wird der Ausgabewert durch die Methode `map_output_val(val, number_of_values)` verarbeitet und evt. verändert. In dieser Funktion kommt nämlich wieder die Modulo-Operation zum Tragen. Das bedeutet, dass wir immer einen Aktionswert von 0-8 (8 Bewegungsrichtungen + Nichts tun) erhalten. Egal wie hoch oder wie klein der tatsächliche Wert auch sein mag.

Zusätzlich gibt es noch einen Schutz gegen nicht-finite Werte. Es ist theoretisch möglich, dass ein Integer-Overflow auftritt und der Wert Infinity oder -Infinity annehmen kann. Ist dies in der Ausgabezelle der Fall wird ein zufälliger Wert zurückgegeben. Nicht-finite Werte treten selten auf und ein random-Verhalten wird sich in der Eprobots-Population nicht durchsetzen.

```
class Eprobot{
    constructor(s, program, init_data) {
        this.position_x = null;
        this.position_y = null;

        this.s = s;

        this.age = 0;
        this.max_age = settings.eprobots_max_age + tools_random(100);
        this.energy = 0;

        this.program = program;

        this.init_data = init_data;
        this.working_data = init_data.slice(0);
    }
```

```
    get_color(){
        return "#f50000";
    }

    get_max_age(){
        return this.max_age;
    }

    map_output_val(val, number_of_values){
        if (isFinite(val)){
            var mapped_val = Math.abs(val) % (number_of_values);
        }else{
            var mapped_val = tools_random(number_of_values);
        }
        return mapped_val;
    }

    get_output_OISC(){
        tools_compute(this.program, this.working_data, settings.program_steps_max);

        let moveval_raw = this.working_data[0];
        let moveval = this.map_output_val(moveval_raw, DIRECTIONS.length + 1);

        return moveval;
    }

    step(){
        let moveval = this.get_output_OISC();
        if (moveval<DIRECTIONS.length){
            let vec = DIRECTIONS[moveval];
            let movepos_x = this.position_x + vec.x;
            let movepos_y = this.position_y + vec.y;

            let t_new = this.s.world.get_terrain(movepos_x, movepos_y);
            if (t_new.slot_object==null){
                this.s.world.world_move(this, this.position_x, this.position_y, movepos_x,
         movepos_y);
                if (t_new.energy_object){
                    console.log("mampf");
                    t_new.energy_object = null;
                    this.s.plant_counter--;
                    this.energy++;
                }
            }else{
                console.log("autsch!");
            }
        }
        this.age++;
    }
}
```

eprobot.js erweitern

simulation.js erweitern

Das war es fast. Kommen wir zur simulation.js. Da wir den Konstruktor der Eprobots verändert haben und dort jetzt initiale Programm- und Datenspeicher brauchen, müssen diese beim Erstellen der Eprobots erst einmal erzeugt werden. Die Initialisierung der Programm- und Datenspeicher ist anfangs zufällig. Die Werte dafür kommen aus den beiden Methoden `get_random_program` und `get_random_data`.

Es werden anfangs viele “Schrott-Programme” erzeugt werden, aber auch einige, die den Ausgabewert/Bewegungswert so verändern, dass ein *Bewegungsmuster* entsteht. Diese Programme werden erfolgreich sein und sich fortpflanzen können. In diesem Moment kommt das Evolutionsprinzip zum Tragen, denn es werden die erfolgreichen Programme vererbt. Aber nicht zu 100%, denn sie werden vorher noch durch unsere Mutationsfunktion geschickt, die kleine zufällige Änderungen vornimmt. Manche Programme werden so

wieder unbrauchbar, sind also unfähig sich zu vermehren. Andere sind aber vielleicht ein bißchen besser als ihr Vorgänger. Der Kreis schließt sich und es werden immer besser angepasste Programme entstehen.

Die Vererbung mit Mutation wird in der Methode `try_fork` vorgenommen. Am Anfang von `try_fork` wird noch überprüft ob die Maximalzahl an Eprobots schon erreicht ist (`eprobots_max_individuals` aus den Settings). Ist sie erreicht können sich die Eprobots erst wieder fortpflanzen, wenn mindestens ein Eprobots gestorben ist.

```
class Simulation {

    constructor(canvas) {
        console.log("Simulation class init")
        let canvas_width = canvas.width;
        let canvas_height = canvas.height;
        console.log("Canvas: "+canvas_width+"x"+canvas_height);

        this.world_width = canvas_width/settings.pixel_size+2;
        this.world_height = canvas_height/settings.pixel_size+2;
        console.log("world dimensions: "+this.world_width+"x"+this.world_height);
        this.world_width_visible = this.world_width-2;
        this.world_height_visible = this.world_height-2;

        this.world = new World(this, this.world_width, this.world_height);
        this.drawer = new Drawer(this, canvas);

        this.add_borders();

        this.plant_counter = 0;

        this.list_eprobots = [];
        this.eprobot_counter = 0;

        this.drawer.paint_fast();
    }

    get_random_program(){
        var program = [];
        for (var pi = 0; pi < settings.program_length; pi++) {
            var val = tools_random(settings.program_length * 10) - settings.program_length;
            program.push(val);
        }

        return program
    }

    get_random_data(){
        var init_data = [];
        for (var di = 0; di < settings.data_length; di++) {
            var val = tools_random2(-720, 720);
            init_data.push(val);
        }

        return init_data;
    }

    seed_eprobots(){
        if (this.eprobot_counter==0){
            for (let i=0;i<10;i++){
                let init_program = this.get_random_program();
                let init_data = this.get_random_data();
                let eprobot = new Eprobot(this, init_program, init_data);
                let rand_x = tools_random(this.world_width_visible);
                let rand_y = tools_random(this.world_height_visible);

                if (this.world.get_terrain(rand_x, rand_y).slot_object==null){
                    this.world.world_set(eprobot, rand_x, rand_y);
                    this.list_eprobots.push(eprobot);
                    this.eprobot_counter++;
```

```
            }
        }
    }
}

seed_plants(){
    if (this.plant_counter<settings.number_of_plants){
        let pc = this.plant_counter;
        for (let i=0;i<settings.number_of_plants-pc;i++){
            let p = new Plant(this);
            let rand_x = tools_random(this.world_width_visible);
            let rand_y = tools_random(this.world_height_visible);

            let t = this.world.get_terrain(rand_x, rand_y);
            if (t.energy_object==null && t.slot_object==null){
                this.world.world_set_energy(p, rand_x, rand_y);
                this.plant_counter++;
            }
        }
    }
}

simulation_step(){
    this.seed_plants();
    this.seed_eprobots();

    let list_eprobots_next = [];
    for (let o of this.list_eprobots) {
        if (o.age<o.get_max_age()){
            o.step();
            list_eprobots_next.push(o);
            this.try_fork(o, list_eprobots_next);
        }else{
            console.log("dead");
            this.world.world_unset(o, o.position_x, o.position_y);
            this.eprobot_counter--;
        }

    }

    this.list_eprobots = list_eprobots_next;
}

try_fork(o, list_eprobots_next){
    if (o.energy>0 && this.eprobot_counter<settings.eprobots_max_individuals){
        let spreadval = tools_random(8);
        let vec = DIRECTIONS[spreadval];
        let spreadpos_x = o.position_x + vec.x;
        let spreadpos_y = o.position_y + vec.y;
        let spreadterrain = this.world.get_terrain(spreadpos_x, spreadpos_y);
        if (spreadterrain.slot_object==null){
            console.log("spread");
            let new_program = tools_mutate(settings.mutate_possibility, settings.mutate_strength,
 o.program);
            let new_data = tools_mutate(settings.mutate_possibility, settings.mutate_strength,
 o.init_data);
            let eprobot = new Eprobot(this, new_program, new_data);
            this.world.world_set(eprobot, spreadpos_x, spreadpos_y);
            list_eprobots_next.push(eprobot);
            this.eprobot_counter++;
            o.energy--;
        }
    }
}

add_borders(){
    for (let x=0;x<this.world_width;x++){
        let b = new Barrier(this);
```

```
            this.world.world_set(b, x, 0);

            let b2 = new Barrier(this);
            this.world.world_set(b2, x, this.world_height-1);
        }

        for (let y=1;y<this.world_height-1;y++){
            let b = new Barrier(this);
            this.world.world_set(b, 0, y);

            let b2 = new Barrier(this);
            this.world.world_set(b2, this.world_width-1, y);
        }
    }
}
```

simulation.js erweitern

Abschluss

Nach einem Reload sollte man relativ schnell Populationen mit interessanten Bewegungsmustern sehen können. Manchmal dauert es ein wenig, aber nach spätestens 5 Minuten sollte man etwas zu sehen bekommen.