Browsergame: A-Maze-Ing

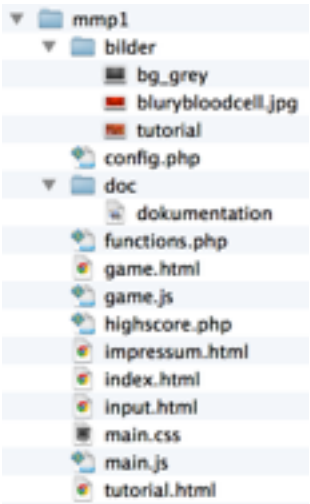
Erstellt von: Daniel Raudschus
24. Mai 2014

TECHNISCHE DOKUMENTATION

1. GLIEDERUNG

Die Dateien liegen in einem Ordner <mmp1> und besitzen zwei Unterordner.:

1. **<bilder>** beinhaltet das Tutoriumbild und die Hintergrundbilder für Webseite und Canvassowie
2. **<doc>** enthalt die technische Dokumentation



Im Hauptordner **<mmp1>** liegen sämtliche .html, .js und .php Files gespeichert:

1. index.html definiert die startseite.
2. game.html beinhaltet das Canvaselement.
   A. game.js enthät alle für das Canvaselement nötigen JavaScriptfunktionen
3. input.html enthält ein HTML Formular zur Namens- und Highscoreeingabe, welches an den PHP Server gesendet wird.

   B. main.js enthält JavaScript zum übertragen des Highscores an das HTML Formular

4. impressum.html enthät das geforderte Impressum.
5. main.css beinhaltet sämtliche css für die gesamte Website.
6. highscore.php greif auf eine MySql Datenbank zu, übermittel Spielername und Highscore und gibt diese in einer Tabelle für den Spieler aus.

   C. functions.php startet die session und inkludiert die Datei config.php in der die Anmeldedaten für die MySql Datenbank enthalten sind (config.php wird nicht mit an den Server gesendet oder zum Git repository gepusht!)

7. tutorial.html enthält eine Bilddatei zur Beschreibung des Spielablaufs.

2. CODE

Das Browsergame ist komplett in HTML5 geschrieben und wird im Canvas mit JavaScript animiert. Das gesammte Spiel läuft in der Funktion **gameBasics()** ab. Diese wird durch **requestAnimationFrame()** 60 mal pro Sekunde aufgerufen

clearRect() löscht den gesamten Canvas;
draw() zeichnet alle Kreise in den Canvas;
drasCreateCirle() zeichnet Spieler- und Gewinnkreis
movePlayer() ruft Steuerungsfunktion auf
win() ruft Gewinnfunktion auf
die() ruft Sterbefunktion auf

Weiterhin versucht eine if-else Abfrage eine Veränderung des Highscorewertes über den Startwert hinaus zu verindern. Wird der Startwert, z.B. durch Manipulation größer als sein Startwert, wird dieser auf 0 gesetzt.

Um randomiesierte Kreise zu erstellen gibt es eine Funktion **Circle()** welche als Konstruktor agiert und die Parameter **radius, speed, bubbleRadius, xPos, yPos, opacity, counter** sowie eine variable **direction** enthalten (Abb. 1.0).

```
// circle construktor for random circles
function Circle(rad, speed, circleWidth, xPos, yPos) {
    this.radius = rad;      // from rotationpoint
    this.speed = speed;
    this.bubbleRadius = circleWidth;
    this.xPos = xPos;
    this.yPos = yPos;

    this.opacity = Math.random() * .25;

    this.counter = 0;

    var direction = Math.floor(Math.random() * 2);

    if (direction == 1) {
        this.dir = -1;      // counterclockwise
    } else {
        this.dir = 1;       // clockwise
    }
}
```

(Abb. 1.0)

radius: Radius von einem randomisierten Punkt innerhalb des Canvas zum Mittelpunkt eines neuen Kreises (rotationsradius).
speed: Geschwindigkeit der Rotation.
xPos/yPos: Randomisierte Punkte innerhalb des Canvas.
opacity: Alphawert für Transparenz der Kreise.
counter: Wert der nach Pos. oder Neg. zählt und in Verbindung mit Math.cos() die Roation der Kreise implementiert.
direction: Legt Pos. oder Neg. Drehung fest.

Die Folgende **Circle.prototype.update()** Funktion etabliert für jeden Kreis der entsteht mithilfe des Konstruktors Grösse, Drehrichtung, Geschwindigkeit und Abstand zum Rotationspunkt. Die Anschließende **drawCircles()** Funktions zeichnet diese Kreise dann und fügt sie mit push in ein vorher erstelltes Array ein. (Abb. 1.1).

```
Circle.prototype.update = function () {

    this.counter += this.dir * this.speed; // defines rotation, direction and speed of the bubbles


    context.beginPath();
    context.arc(this.xPos + Math.cos(this.counter / 100) *
        this.radius, this.yPos + Math.sin(this.counter / 100) *
        this.radius, this.bubbleRadius, 0, 2 * Math.PI, false);
    context.closePath();
    context.fillStyle = 'rgba(255, 0, 0,' + this.opacity + ')';
    context.fill();
};


// array for all the random created circles
var circles = new Array();


function drawCircles() {
    for (var i = 0; i < difficulty; i++) {
        var rad = Math.round(Math.random() * 200);      // from random rotation point! This is what
        var randomX = Math.round(Math.random() * (canvas.width + 200));
        var randomY = Math.round(Math.random() * (canvas.height + 200));
        var speed = Math.random() * 1;
        var circleWidth = Math.random() * 75;          // radius of the circles

        var circle = new Circle(rad, speed, circleWidth, randomX, randomY);
        circles.push(circle);                          // stack it to the array

    }
}
drawCircles(); // dont call in rAF
```

(Abb. 1.1)

drawCircles() darf aufgrund der seiner Math.random Werte nicht im **requestAnimationFrame()** aufgerufen werden, da sonst 60 mal pro Sekunde neue Werte entstehen. Das Array wird mittels **draw()** in **gameBasics()** aufgerufen

createCircle() und **drawCreateCircle()** sind den gerade vorgestellten Funktionen ähnlich, aber dafür gedacht „manuell“ Kreise zu erstellen und zu zeichen. In diesem Fall den Spiele- und Gewinnstein. (Abb. 1.2.)

```
// circle constructor
function createCircle(xPos, yPos, radius, color, border, borderwidth) {
    this.xPos = xPos;
    this.yPos = yPos;
    this.radius = radius;
    this.fillStyle = color;
    this.strokeStyle = border;
    this.lineWidth = borderwidth;

    this.opacity = .1;
}


function drawCreateCirle(c) {
    context.beginPath();
    context.arc(c.xPos, c.yPos, c.radius, 0, Math.PI * 2, false);
    context.fillStyle = c.fillStyle

    context.shadowColor = 'red';
    context.shadowBlur = 0;

    context.fill();
    context.strokeStyle = c.strokeStyle;
    context.lineWidth = c.lineWidth;
    context.stroke();
}
```

```
/**
 * Playermovement with arrowkeys
 */
function movement() {
    var up = down = left = right = false;


    function keysUp(key) {
        // left
        if (key.keyCode == 39) {
            left = false;
        }
        // right
        if (key.keyCode == 37) {
            right = false;
        }
        // down
        if (key.keyCode == 40) {
            down = false;
        }
        // up
        if (key.keyCode == 38) {
            up = false;
        }
    }


    function keysDown(key) {

        // left
        if (key.keyCode == 39) {
            left = true;
        }
        // right
        if (key.keyCode == 37) {
            right = true;
        }
        // down
        if (key.keyCode == 40) {
            down = true;
        }
        // up
        if (key.keyCode == 38) {
            up = true
        }
    }

    document.onkeyup = keysUp;
    document.onkeydown = keysDown;
```

```
document.onkeyup = keysUp;
document.onkeydown = keysDown;

return function movePlayer() {
    // left
    if (left) {
        player.xPos += playerSpeed;
    }
    // right
    if (right) {
        player.xPos -= playerSpeed;
    }
    // down
    if (down) {
        player.yPos += playerSpeed;
    }
    // up
    if (up) {
        player.yPos -= playerSpeed;
    }

    /**
     * Wall detection and avoidance
     **/
    if (player.xPos < 0 + player.radius) {
        player.xPos = 0 + player.radius;
    }
    if (player.yPos < 0 + player.radius) {
        player.yPos = 0 + player.radius;
    }
    if (player.xPos > canvas.width - player.radius) {
        player.xPos = canvas.width - player.radius;
    }
    if (player.yPos > canvas.height - player.radius) {
        player.yPos = canvas.height - player.radius;
    }
}
```

Die Funktion **Movement()** lässt die Spieler die Spielfigur mit den Pfeiltasten steuern. Hierfür wird eine **onkeyup** und **onkeydown** erkannt und true oder false gesetzt. Anschließend wird die Funktion **movePlayer()** per return zurückgegeben, welche dann die Position der Spielfigur verändert, indem Werte der jeweiligen globalen Variable verändert werden.

movePlayer() überprüft ebenfalls ob die Spielfigur mit den Canvaswänden kollidiert. Dies geschieht durch prüfen der x und y Position des Spielers addiert mit deren Radius. Wird dieser Wert größer als der Canvas, wird die x und y Position des Spieler auf den maximalen Canvaswert minius des Spielfigurradius gesetzt.

```
function collide(c1, c2) {
    var dx = c1.xPos - c2.xPos;
    var dy = c1.yPos - c2.yPos;
    var distance = c1.radius + c2.radius;

    // Pythagorean Theorem
    return (dx * dx + dy * dy <= distance * distance);
}


function collideBubbles(c1, c2) {

    // moving/rotation xPos and yPos
    var bubbleX = c2.xPos + Math.cos(c2.counter / 100) * c2.radius;
    var bubbleY = c2.yPos + Math.cos(c2.counter / 100) * c2.radius;

    // white bloodcells
    var destroyerBubble = new createCircle(bubbleX, bubbleY, c2.bubbleRadius, 'rgba(255, 255, 255, .25)', 'rgba(151, 151, 170, .125)', 20);
    drawCreateCirle(destroyerBubble);

    var dx = c1.xPos - bubbleX;
    var dy = c1.yPos - bubbleY;
    var distance = c1.radius + c2.bubbleRadius;

    // Pythagorean Theorem for rot.
    return (dx * dx + dy * dy <= distance * distance);
}
```

Ähnlich wie die Kollision mit den Wänden funktioniert auch die Kollisionserkennung des Spielers mit den Kreisen. Hier muss jedoch die Distanz zwischen den jeweilgen x und y Positionen des Spielersteins und der Kreise errechnet werden. Dies geschieht mit der Pythagorasfunktion (siehe oben). Wird die Distanz von Spieler und Kreis kleiner als 0 findet eine Kollsion statt und die Spielfigur „stirbt“ und wird auf die Startposition zurückgesetzt.

Hierfür gibt es eine Funktion **collide()**, welche die generelle Kollision zwischen Spielerstein und Gewinnstein prüft und von der Funktion **win()** als booleanwert geprüft wird, sowie **collideBubbles()**, welche von der **die()** Funktion genutzt wird um eine Kollison zu erkennen und den Spieler sterben zu lassen. Zusätzlich werden hier die Kreise erstellt die den Spieler töten können (**destroyerBubbble**). Da **win()** und **die()** im **requestAnimationFrame()** eingebunden sind, werden Kollsionen sofort erkannt. Die hier entstehenden **destroyerBubble** bewegen sich in einer Diagonalen Auf- und Abwärtsbewegung in Relation zu den im Hintergrund rotierenden Kreisen.

3. DATENBANK UND SCORE

Um einen Highscore zu errechnen wird die in game.js gesetzte variable Highscore an eine versteckte Formularzeile gesendet, sobald die **win()** Funktion eine Kollision registriert. Der Spieler wird daraufhin auf die input.html Site geschickt. Hier kann er einen Namen eingeben. Dieser, sowie der übergebene Highscore, werden mittels onClick event an den Webserver gesendet. Dieses Vorgehen ist nötig, weil JavaScript lokal arbeitet, während php serverseitig arbeite. Ein einfachs Übergeben der Variablenwerte ist daher nicht möglich.

```
<form id="form" action="highscore.php" method="post">
    Name <input type="text" name="player" id="player" value="" required>
    <input type="hidden" name="score" id="score" value="">
    <input type="submit" name="submit" value="submit" onclick="myScore()">
</form>
```

Die MySql Datenbank ist sehr klein und besteht aus den Zeilen „Playername“ und „Score“. Diese werden mit Prepared Statements befüllt und als Tabelle ausgeben, welche der Spieler nach dem Spiel zu sehen bekommt.

```php
// gets the highscore from JS
$player = $_POST['player'];
$highscore = $_POST['score'];

// $sql = $dbh->query("INSERT INTO Score (playername, highscore) VALUES ('$player', '$highscore')");
$sql = $dbh->prepare("INSERT INTO Score (playername, highscore) VALUES (?, ?)");
$sql->execute(array($_POST['player'], $_POST['score']));

$sth = $dbh->query("SELECT * FROM Score ORDER BY highscore DESC LIMIT 0, 10");
$results = $sth->fetchAll();
```