# Dokumentation - Flight

Verwendete Technologien:
JavaScript, THREE.js, WebGL, Chrome Dev Tools, JQuery, Stats.js, node.js, MongoDB

Coding Conventions: JavaScript Standard Style (https://github.com/feross/standard): 2 Space Indentation, no Semicolons (ASI), daily Backups on GitHub (https://github.com/in0x/Fligth-Game-)

## Der Entry-Point:

Passiert über die window.onload() function.
Das gesamte Spiel passiert im Scope der onload() function, um Global Namespace Pollution zu vermeiden. Zu Beginn werden alle wichtigen Ressourcen (zum Beispiel Score-Counter, eine Liste aller Kollisions-Objekte…) angelegt.

## Die init() function:

Weist die zuvor angelegten Ressourcen zu, lädt das Mesh für die Kollisions-Objekte aus einem JSON-file mit dem eingebauten THREE.JsonLoader und legt die Szene an. Auch wird der erste Teil des Levels generiert. Zusätzlich wird ein Event-Listener gesetzt, der auf eine Enter-Eingabe wartet, um das Spiel zu starten und den Mauszeiger zu sperren.

## Die animate() function:

Dient als Render-Loop und startet gleichzeitig auch die Game-Loop. Der genaue Ablauf schaut folgend aus

- controls.update() : Rechnet aus dem Zeit-Delta seit dem letzten Aufruf die Änderung der Position und des Spieler und passt sie entsprechend an.
- checkCollisions(): Überprüft über den in THREE.js eingebauten RayCaster, ob zwischen dem Spieler und den Spiel-Objekten eine Kollision aufgetreten ist:
  Wenn ja:
  Ist das Kollisions-Objekt ein PickUp?
  Wenn ja: Überprüfe anhand des Mesh-Materials um welches Pickup es sich handelt und passe den Timer entsprechend an
  Hat der Spieler bereits ein Pickup dieser Art eingesammelt?
  Wenn ja, setze den Timer, der die PowerUp-Zeit misst wieder auf 0 (Spieler hat Modifikation solange, bis die maximale Zeit erreicht wird, zum Beispiel 4 Sekunden für Bonus Score).
  Wenn nein, starte den Timer für den jeweiligen Bonus.
  Wenn nein: Das Kollisions-Objekt ist ein Tree und somit das Spiel vorbei. Stoppe die rekursive Rendering-Loop und starte den End-Screen.
- handlePickups(): Überprüft ob Item-Timer gestartet sind. Wenn ja, überprüfe ob die Lifetime überschritten wurde und passe den Game-State entsprechend an. (Z.b. für Geschwindigkeiten: Setze sie auf die bei der Collision gespeicherten alten zurück)
- drawUI(): Update die Score und Distance HTML-Elemente
- animatePickups(): Übernimmt die Rotation der Pickup-Meshes

- reloadWorld(): Überprüft, ob der Spieler sich über eine Distance von mehr als 2000 bewegt hat:
  Wenn ja: Setze die nächste Überprüfungs-Position um 2000 zurück.
  resetFloor(): Ladet den Boden weiter
  spawnObstacles(): Entfernt alle Trees, die sich nun hinter dem Spieler befinden, und setzt nun neue:
  Traversiere ein 3000x3000 Feld vor dem Spieler. In jedem 1000x1000 Feld, setze einen Baum an eine zufällige Position. Mit einer 50% Prozent Chance, wird währenddessen zwischen zwei Trees auch ein Pickup gespawnt.

```
function spawnObstacles (mesh, curViewPos) {
    mesh_list.forEach(function (el) {
      if (el.position.z > controls.getObject().position.z)
        mesh_list.splice(mesh_list.indexOf(el), 1)
    })

    for (var y = 0; y < 3; y++)
      for (var x = 0; x < 3; x++) {
        //Spielfeld-Traversierung, 3000x3000 Feld, in 9 Subfeldern
        var x_pos = getRandom(-1500 + 1000 * x, -1500 + 1000 * (x + 1)),
          z_pos = getRandom(curViewPos, curViewPos - 1000 * y),
          y_pos = getRandom(-900, 0)

        if (y == 1 && x == 1 && getRandom(0, 11) > 6) {
          //Code zum Spanne von Pickups, der Kürze wegen hier ausgelassen
        }
        var tempMesh = new THREE.Mesh(treeGeo, new
        THREE.MeshLambertMaterial({ color: meshcolor}))
        tempMesh.scale.set(60, 100, 60)
        tempMesh.position.set(x_pos, y_pos, z_pos)
        tempMesh.rotation.y = (Math.random() * 10)
        scene.add(tempMesh)
        mesh_list.push(tempMesh)
    }
  }
```

*Die Funktion zum spawnen von Hindernissen*

## Kollision mit einem Tree:

Tritt dieser Fall ein, passiert folgendes:
Eine Hilfsvariable wird auf false gesetzt, die die rekursive Render-Loop bricht.
Als nächstes wird der End-Screen aus einem HTML-File geladen.
Nun werden die 5 Tipscores über getTopScores() geladen. Diese Funktion akzeptiert als drittes Argument ein Callback, in diesem Fall drawScoreBoard(). Diese Funktion baut eine Liste der Scores und fügt gleichzeitig Event-Listener für Submit (Score abschicken) und Restart (neue Spiel-Runde) ein.

# Das Backend:

Für die Highscore-Funktion habe ich eine einfache node.js API gebaut, die auf Redhat OpenShift gehostet ist. Sie hat 3 Routen:

- /scores: Fetcht alle hinterlegten Scores.
- /top/{number}: Fetcht die {number} besten Scores
- /post: Erhält über Post ein JSON-Objekt nach dem Template

```
{       „name“: name,
        „score“:score
}
```

- Alle diese Requests werden an eine über MongoLab gehostet MongoDB geschickt.

# Die Steuerung:

Funktioniert mit dem von Mr.Doob geschriebenen und mit THREE.js ausgeliefertem PointerLockControls-Module. Ich habe dieses auf meine Bedürfnisse angepasst:

Die Funktionalität für WASD entfernt, und durch eine stetige Beschleunigung die über die Game.Velocity gemessen wird ersetzt.

Den maximalen Yaw auf 120° beschränkt um Umdrehen durch den Spieler zu verhindern.

Eine maximale / minimale Y-Grenze eingebaut, um das Fliegen über Hindernisse, unter dem Boden zu verhindern.

Eine maximale / minimale X-Grenze eingebaut, um das Fliegen links/rechts der äußeren Hindernisse zu vermeiden.

PointerLockControl wird als neue Instanz als Kind des Camera-Objects instanziiert. Es enthält ein THREE.3dObject, welches es erlaubt die derzeitige Welt-Position, sowie die Facing-Direction abzufragen (wichtig für Raycasting bei Collision Detection).

Das PointerLockControls-Module wird nicht mit der derzeitigen R73 Release-Version von THREE.js ausgeliefert, wurde aber in früheren Versionen mitgeliefert, und ist noch immer unter der selben Lizenz im GitHub-Repo verfügbar. (https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html)

# THREE.js:

THREE.js ist eine JavaScript Bibliothek zur einfachen Nutzung von WebGL, der Browser zu GPU Schnittstelle. Sie bietet Abstraktionen für alle wichtigen GL Funktionen. Der WebGL Draft selbst basiert auf OpenGL ES 2.0, und somit bereits dem Modell der programmierbaren Grafik-Hardware (Im Gegensatz zum Fixed Function Model).

Der erste Schritt zur Nutzung der Bibliothek, ist die Erzeugung eines Renderer-Elements. Dies erzeugt einen OpenGL-Kontext, der bereits einige Flags wie zum Beispiel Anti-Aliasing oder Fog akzeptiert. Dieser Renderer fügt dann ein canvas element in das Dokument hinzu, in dem jeder gerenderte Frame angezeigt wird.

Zusätzlich wird noch eine Szene benötigt, die eine Kollektion aller Kontext-Elemente darstellt. Elemente werden erst gerendert, wenn sie in die aktuelle Szene eingefügt wurden. THREE.js unterstützt die Nutzung mehrerer Szenen, zum Beispiel für Virtual Reality Applikationen, die zwei simultane Bilder brauchen.

```
let scene = new THREE.Scene()
let renderer = new THREE.WebGLRenderer({antialias: true})
renderer.setSize(RENDER_WIDTH, RENDER_HEIGHT)
renderer.setClearColor(0x000, 1)
document.body.appendChild(renderer.domElement)
```

Die Projektionsmatrix wird wie in anderen Bibliotheken über eine Kamera erzeugt. Es wird perspektivische und orthogonale Projektion unterstützt.

```
camera = new THREE.PerspectiveCamera(75, RENDER_WIDTH /
RENDER_HEIGHT, 1, 8000)
camera.lookAt(0, 0, 0)
scene.add(camera)
let light = new THREE.AmbientLight(0xfff)
scene.add(light)
```

Die Bibliothek bietet  unterschiedliche Möglichkeiten, um Objekte in der Szene darzustellen. Einerseits können Objekte manuell aus Vertices in zum Beispiel indizierte Dreiecksnetze zusammengesetzt werden (THREE.geometry.vertices & .faces). Andererseits sind auch viele vorgefertigte Geometrien (THREE.SphereGeometry, .PlaneGeo, .IcosahedronGeo) vorhanden.

```
//Custom Geometry
let geo = new THREE.Geometry()

geo.vertices.push(new THREE.Vector3(-0.5, -0.5, 0), new
THREE.Vector3(0.5, -0.5, 0), new THREE.Vector3(0.5, 0.5, 0), new
THREE.Vector3(-0.5, 0.5, 0),)

geo.faces.push(new THREE.Face(this.vertices[0], this.vertices[1],
this.vertices[2]), new THREE.Face(this.vertices[0],
this.vertices[2], this.vertices[3]))

//Built-In
//width, height, segments
let geoThree = new THREE.PlaneGeometry(1, 1, 1)
```

Die Geometrie wird dann in ein Mesh eingefügt, in dem auch die Shading-Art (Lambert, Blind und Phong sind in der Bibliothek bereits implementiert) und weitere Materialeigenschaften (Color, bzw Diffuse Color, Specular Color, Shininess etc.) angegeben werden. Dieser Prozess abstrahiert somit Model-Generation, erstellen der Shader und Shader-Switching für unterschiedliche Objekte.

```
let mesh = new THREE.Mesh(geo, newTHREE.MeshLambertMaterial({color:
0xff0000})))
```

Transformationen können statt über Matrizen durch Eigenschaften der Meshes ausgeführt werden. So haben alle Meshes .rotation(.x.y.z), .scale(.x.y.z) und .position(.x.y.z) Eigenschaften, sowie eine .set(x,y,z) für alle Eigenschaften.

```
mesh.scale.set(2, 3, 0) //*2 in x, *3 in y
//mesh.rotation.z += (45 * (Math.PI / 180)) //radians
scene.add(mesh)
```

Weiters werden Loader für OBJ und JSON files angeboten. Wichtig ist hierbei, dass die Loader asynchrone Funktionen sind. Üblicherweise werden die Daten über den Loader geladen und dann mit Hilfe eines Callbacks im Code verwendet:

```
let loader = new THREE.JSONLoader()
 function callback (geometry, material) { //Callback
      other_geo = geometry
 }
loader.load('bla.json',callback(geometry))
render()
```

*Der finale Output*