

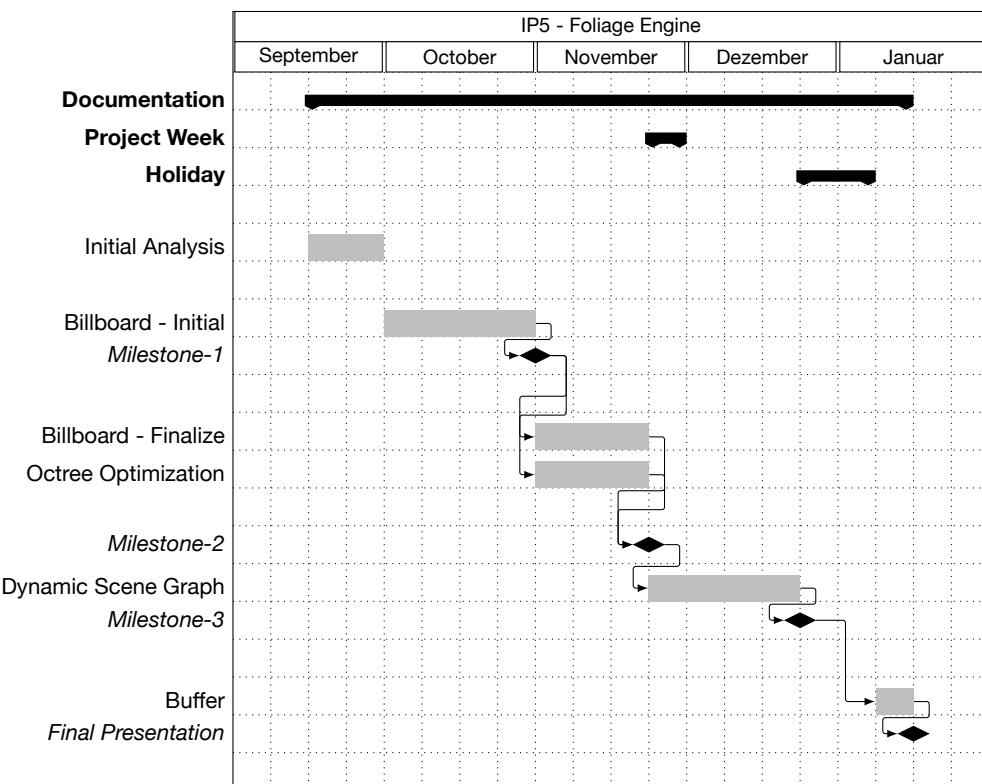
## Contents

<b>Foliage Engine</b>	<b>2</b>
Introduction . . . . .	2
<b>Grobplanung</b>	<b>2</b>
Milestones . . . . .	2
<b>Analyse</b>	<b>3</b>
Ist-Analyse . . . . .	3
Ziel . . . . .	3
Struktur Sourcecode . . . . .	3
Debugging . . . . .	5
Performance . . . . .	5
Findings . . . . .	5
<b>Billboard</b>	<b>6</b>
Schritt 1) VertexShaderMaterial . . . . .	6
Schritt 2) Partikelsystem . . . . .	8
Schritt 3) Points . . . . .	8
Schritt 4) Billboard/Sprite-Kombination . . . . .	9
Schritt 5) Billboard Implementierung . . . . .	9
<b>Octree</b>	<b>9</b>
<b>Dynamic Scene Graph</b>	<b>9</b>
<b>Appendix</b>	<b>10</b>
Meetings . . . . .	10
Meeting (1) . . . . .	10
Meeting (2) . . . . .	11
Meeting (3) . . . . .	12
References . . . . .	13

## Foliage Engine

### Introduction

### Grobplanung



### Milestones

- **Milestone - 1:** Es existiert eine erste Version der Billboard Implementation in der Foliage Engine. Die Implementation ist unter Umständen noch nicht ganz fertig, jedoch mehrheitlich. Nachbesserungen (der Billboard Implementation) sowie eine Analyse der Octtree Implementation sollen bis zur Projektwoche vorgenommen werden.
- **Milestone - 2:** Es existiert eine verbesserte Version der Billboard Implementation, des Weiteren wurde die Octtree Implementation analysiert. Sofern die Resultate befriedigend sind, kann mit der Analyse der Dynamic Scene Graph Implementation in der Projektwoche gestartet werden.
- **Milestone - 3:** Es existiert eine einfache Implementation eines Dynamic Scene Graph (abhängig vom Projektfortschritt).
- **Final Presentation:** Finale Abgabe / Präsentation.

## Analyse

### Ist-Analyse

#### Ziel

Die Foliage-Engine soll die three.js Library mit einer Gras-Engine ergänzen. Ziel war es ein Example für threejs.org zu erstellen. Das Example besteht aus einer Fläche mit Gras in der man sich frei bewegen und so die Engine testen kann.

#### Struktur Sourcecode

Übernommen haben wir die bestehende Foliage-Engine als komprimierter Ordner. Darin enthalten ist ein Readme mit einer Anleitung zum Erstellen von Foliage-Objekten und deren Konfiguration enthalten.

#### Foliage-Engine

Das "index.html" zusammen mit "three.foliageengine.js" ist die finale Demo bzw. die finale Implementatin von der Foliage-Engine die wir erhalten haben.

#### Librarys

In einem Ordner "js" sind alle verwendeten Librarys gespeichert die für das Ausführen von Three.js und den Anzeigen von Debugmöglichkeiten notwendig sind.

#### Models

Die Models der verschiedenen LOD-Levels sind im "model" Ordner untergebracht und nach LOD-Level sortiert.

**Scenes** Im Ordner "scenes" findet man dann verschiedene HTML-Files die als Muster dienen, in denen verschiedene Gras-Models und LOD-Levels für die Foliage-Engine verwendet werden.

#### Textures

Im "textures" Ordner sind verschiedene Texturen für Gras und Terrain vorhanden.

Will man nun die Engine testen muss man im Hauptordner einen Server starten und kann dann mittels Browser entwerder das "index.html" oder eines der anderen Beispiele aus "scenes" aufrufen.

```
foliageengine/
    index.html
    js/
        PointerLockControls.js
        three.js
        three.min.js
        threex.renderstats.js
    models/
        grass/
            2.5D/
                models
                bilder
            lod0/
                models
                bilder
            lod1/
                models
                bilder
            lod2/
                models
                bilder
            lod3/
                models
                bilder
            lod4/
                models
                bilder
            lod5/
                models
                bilder
    readme.txt
    scenes/
        2.5d.html
        32.52d.html
        50fps.html
        demo.html
        phd.html
        unity.html
    textures/
        grass_normal.png
        grass.png
        lod2d_normal.png
        lod2d.png
        sand_clean_bumpy_01_b_normal.png
        sand_clean_bumpy_01_b.png
        skybox/
three.foliageengine.js
```

## Debugging

Um den Code besser zu verstehen und später auch einfacher die Veränderungen die wir programmieren zu erkennen, haben wir uns eine Debugversion von der Foliage-Engine gebaut. Dabei haben wir den verschiedenen LOD eine andere Farbe gegeben.

```

1   // Register new loaded LOD Level
2   var mesh;
3   if(level == 1){
4       mesh = new THREE.Mesh(
5           new THREE.BufferGeometry().fromGeometry(geometry),
6           //new THREE.MeshFaceMaterial(material);
7           new THREE.MeshBasicMaterial({color : 0xff0000}));
8   } else if(level == 2){
9       mesh = new THREE.Mesh(
10          new THREE.BufferGeometry().fromGeometry(geometry),
11          new THREE.MeshBasicMaterial({color : 0x0000ff}));
12   } else if(level == 3){
13       mesh = new THREE.Mesh(
14          new THREE.BufferGeometry().fromGeometry(geometry),
15          new THREE.MeshBasicMaterial({color : 0x00ff00}));
16   } else {
17       mesh = new THREE.Mesh(
18          new THREE.BufferGeometry().fromGeometry(geometry),
19          new THREE.MeshBasicMaterial({color : 0x000000}));
20   }

```

Dabei färbten wir alle Gras-Objekte im LOD1 Rot, im LOD2 Blau und im LOD3 Grün ein. Das letzte LOD liessen wir, da dort eh nur eine Textur geladen wird.

Zusätzlich haben wir noch die Distanz bei der das LOD wechseln soll verkürzt, damit wir alle 4 LOD auf einem Blick sehen können.

So können wir nun den Code verändern und sehen gleich die Wirkung davon. Auch lassen sich so leichter vorhandene Fehler finden.

## Performance

### Findings

Bei der Codeanalyse um uns in das Thema einzuarbeiten haben wir auch ein paar Fehler entdeckt. Die wier selbstverständlich auch dokumentieren möchten.

### Texturen

Als wir uns entschieden die Foliage-Engine so zu manipulieren, damit wir leicht die verschiedenen LOD erkennen können, viel uns auf dass die Texturen für das letzte LOD garnie geladen wurden. Bei unserer Suche nach dem fehler mit der Hilfe von Debugging-Tools vom Chrome-Browser fanden wir heraus, dass die Funktion THREE.Foliage.prototype.handle2DLevel gar nie aufgerufen wird.

Bei weiterem suchen stiessen wir dann schliesslich auf folgenden Codeabschnitt:

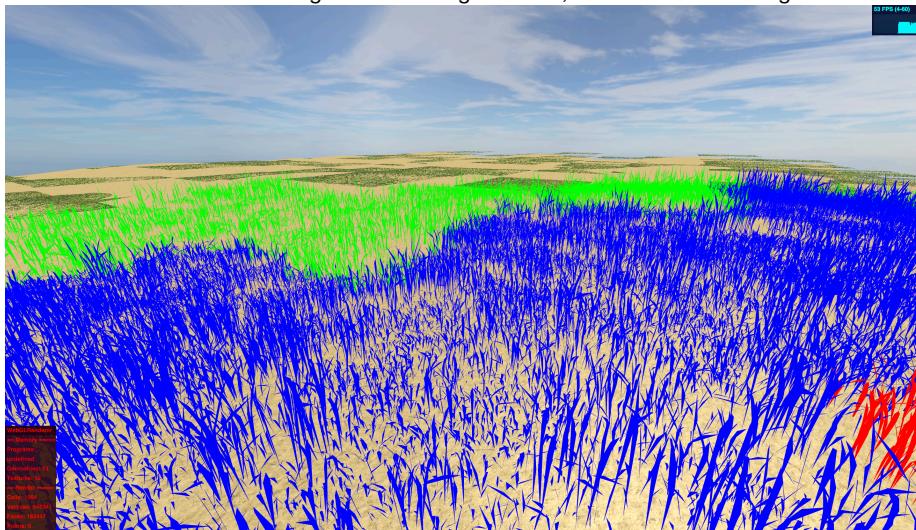
```

1   if (textures) {
2     var modelTextur = textures();
3     //var modelTextur = textures();
4     for (var x = 0; x < modelTextur.length; x++) {
5       this.totalModels++;
6       this.handle2DLevel(modelTextur[x], x, level);
7     }
8   }

```

Dabei entdeckten wir, dass in die Variable `modelTextur` eine Funktion statt einem Array geladen wird. Leider warf der Browser bei der Verwendung einer Funktion in der For-Schlaufe keinen Fehler. Den konnten wir leicht beheben in dem wir `textures()` mit `(textures())()` ersetzen. Nun wird auch das Array in die Variable übergeben. Jetzt sehen wir auch die Textur für das letzte LOD und entdeckten gleich den nächsten Fehler.

Für die Textur erstellte sie ein flaches Objekt das auf dem Boden liegt. Darüber legten sie ihre Textur. Was sich im ersten Augenblick noch gut anhört, sieht dann im Endergebnis so aus:



Bei einem Terrain das keine gerade Fläche ist, verschwinden die Graden Flächen in den Hügeln und hinterlassen so entweder Löcher in der Grassimulation oder es fliegen Quadratische Grastexturen irgendwo in der Luft herum.

Dieser Fehler wurde nicht behoben und besteht immer noch.

## Billboard

### Schritt 1) VertexShaderMaterial

In einem ersten Schritt haben wir aufbauend auf bestehendem Code ??? ein ShaderMaterial implementiert und den letzten LOD als Mesh abgebildet.

Gegebenes ShaderMaterial:

```

1   // Register new loaded LOD Level
2   var mesh;
3   if(level == 1){
4       mesh = new THREE.Mesh(
5           new THREE.BufferGeometry().fromGeometry(geometry),
6           //new THREE.MeshFaceMaterial(material));
7           new THREE.MeshBasicMaterial({color : 0xff0000}));
8   } else if(level == 2){
9       mesh = new THREE.Mesh(
10          new THREE.BufferGeometry().fromGeometry(geometry),
11          new THREE.MeshBasicMaterial({color : 0x0000ff}));
12 } else if(level == 3){
13     mesh = new THREE.Mesh(
14        new THREE.BufferGeometry().fromGeometry(geometry),
15        new THREE.MeshBasicMaterial({color : 0x00ff00}));
16 } else {
17     mesh = new THREE.Mesh(
18        new THREE.BufferGeometry().fromGeometry(geometry),
19        new THREE.MeshBasicMaterial({color : 0x000000}));
20 }

```

Eigenes ShaderMaterial:

```

1  var vertexShader = [
2      "varying vec2 vUv;",
3      "void main() {",
4      "vUv = uv;",
5      "gl_Position = projectionMatrix * ",
6      "(modelViewMatrix * vec4(0.0, 0.0, 0.0, 1.0) + ",
7      "vec4(position.x, position.y, 0.0, 0.0));",
8      "}"
9  ].join("\n");
10 var fragmentShader = [
11     "uniform sampler2D texture;",
12     "varying vec2 vUv;",
13     "void main() {",
14     "vec4 tex = texture2D ( texture, vUv);",
15     "gl_FragColor = vec4(tex.r, tex.g, tex.b, tex.a);",
16     "}"
17 ].join("\n");

```

Für einen ersten Test verwendeten wir dabei das Beispielprogramm von threejs.org welches normalerweise einen Würfel rotieren lässt. Nun ersetzen wir das Material des Würfels durch unser eigenes ShaderMaterial. Dabei soll, wenn alles klappt, uns unsere Textur die ganze Zeit anschauen, obwohl sich das Objekt eigentlich dreht.

Als sich das Ergebnis als erfolgreich heraus stellte, haben wir einen Test innerhalb der Foliage-Engine gewagt. Der einfachheit halber haben wir unser Experiment im letzten LOD-Level versucht, da es sich auch um eine Textur handelt und dieser einfacher zu verändern ist.

Billboard-Simulation mit PlaneBufferGeometry:

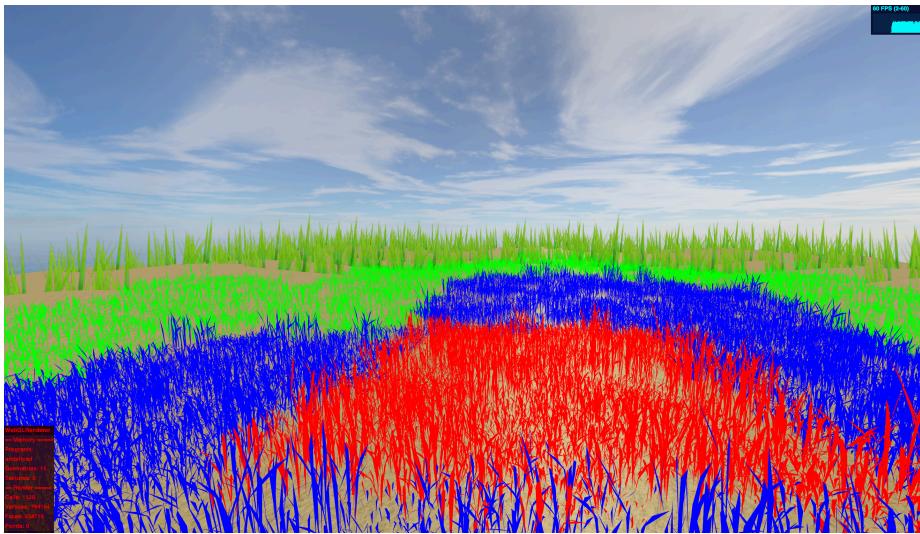


Figure 1: Resultat mit ShaderMaterial

Beim präsentieren von diesem Ergebnis wurden wir darauf Aufmerksam gemacht, dass das kein Partikelshader ist, sondern mehr ein Workaround der das Verhalten von einem Partikelshader imitiert. In unserer Lösung hat jedes Objekt immer noch 4 Vertices um es darzustellen. Beim eigentlichen Partikelshader ist für eine Darstellung nur ein Vertix notwendig, nämlich das für die Position des Objekts.

Wir wurden darauf hingewiesen, dass wir es mit dem Partikelsystem realisieren können.

## Schritt 2) Partikelsystem

Nun machten wir das gleiche Experiment nochmal, diesmal jedoch mit dem Partikelsystem. Dabei versuchten wir statt `THREE.Mesh(geometry, material)` die Funktion `THREE.ParticleSystem(geometry, material)` und verwendeten dabei wieder unser `ShaderMaterial`. Jedoch wurden wir dabei schnell von unserem Chrome-Browser beim testen darauf Aufmerksam gemacht, dass das `ParticleSystem` veraltet ist und wir `Points` verwenden sollen.

## Schritt 3) Points

Beim recherchieren für die Verwendung von `Points` stiessen wir darauf, dass für eine einfache Implementierung von einem Billboard kein eigener Shader notwendig ist. Es wird uns `THREE.PointsMaterial()` angeboten welcher den Shader komplett implementiert hat. Dort müssen wir lediglich noch ein paar Einstellungen betätigen.

```

1  var material = new THREE.PointsMaterial();
2  material.map = texture;
3  material.size = 5.0;
4  material.sizeAttenuation = false;
5  material.transparent = false;
```

Als Geometry wird hier nur noch ein Vertix benötigt.

Auch hier haben wir zuerst ein Beispiel mit dem Beispielprogramm von threejs.org umgesetzt. Anschliessend haben wir wieder das letzte LOD mit unseren Points-Objekten ersetzt um es innerhalb von der Foliage-Engine zu testen. Für den Test Haben wir noch ein paar Objekte in die Luft gesetzt und die Transparenz deaktiviert, um zu sehen ob der Billboard-Effekt wirklich korrekt funktioniert.

Ergebnis mit den Points-Objekten:

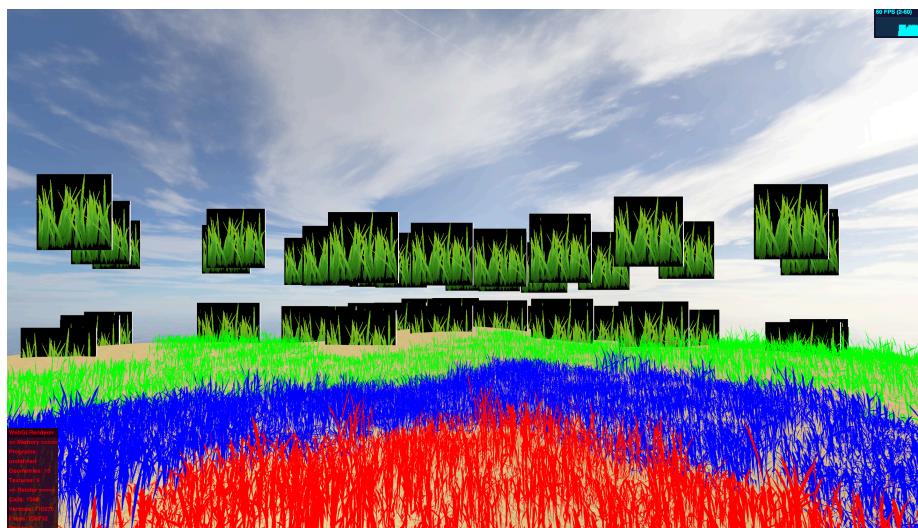


Figure 2: Resultat mit ShaderMaterial

#### Schritt 4) Billboard/Sprite-Kombination

TDB

#### Schritt 5) Billboard Implementierung

TBD

#### Octree

#### Dynamic Scene Graph

## Appendix

### Meetings

#### Meeting (1)

**Datum:** 29. September 2015 (11:00 - 11:15)

**Teilnehmende:** Livio Bieri, Raphael Brunner, Stefan Arizona, Roman Bolzern

#### Protokollpunkte / Themen

- Besprechung der erledigten Arbeiten seit letzten Meeting.
- Erläuterungen der drei Teilaufgaben des Projekts durch Auftraggeber (*Billboard Implementation, Optimierung mittels Octree Implementation, Dynamic Scene Graph Implementation durch Auftraggeber*).

#### Pendenzenliste / Beschlussee

Bis zum nächsten Meeting am 7. Oktober sind die folgenden Pendenzen zu erledigen:

1. Ausarbeiten einer kurzen Beschreibung der drei möglichen Teilaufgaben des Projekts.
2. Ausarbeiten einer kurzen Planung der wichtigsten Milestones (*TDB* vor dem 7. Oktober, via Mail).

#### Sonstiges - None

**Meeting (2)**

**Datum:** 07. Oktober 2015 (11:00 - 11:30)

**Teilnehmende:** Livio Bieri, Raphael Brunner, Roman Bolzern

**Protokollpunkte / Themen**

- Besprechung der erledigten Arbeiten seit letzten Meeting.
- Vorstellung der ersten Implementation der VertexShaderMaterials<sup>1</sup> basierend auf dem Beispiel von Herr Bolzern ???.
- *Feedback: Bisherige Implementation verwendet Mesh; sollte aber Points verwenden.*

**Pendenzenliste / Beschlüsse**

*Bis zum nächsten Meeting sind die folgenden Pendenzen zu erledigen:*

1. Implementation unter der Verwendung von Points<sup>2</sup>
2. Starten mit der Dokumentation.

**Sonstiges**

- None

---

<sup>1</sup>[three.js Dokumentation: ShaderMaterial](#)

<sup>2</sup>[three.js Dokumentation: Points](#)

### **Meeting (3)**

**Datum:** 20. Oktober 2015 (11:15 - 11:40)

**Teilnehmende:** Livio Bieri, Raphael Brunner, Roman Bolzern, Stefan Arizona

#### **Protokollpunkte / Themen**

- Besprechung der erledigten Arbeiten seit letzten Meeting.
- Vorstellung der Points Implementation.<sup>3</sup>

#### **Pendenzenliste / Beschlüsse**

*Bis zum nächsten Meeting sind die folgenden Pendenzen zu erledigen:*

1. Nachdokumentieren der bisher erledigten Arbeit.
2. Zeiten aktualisieren.

#### **Sonstiges**

- *None*

---

<sup>3</sup>Commit auf Github

**References**