

## Contents

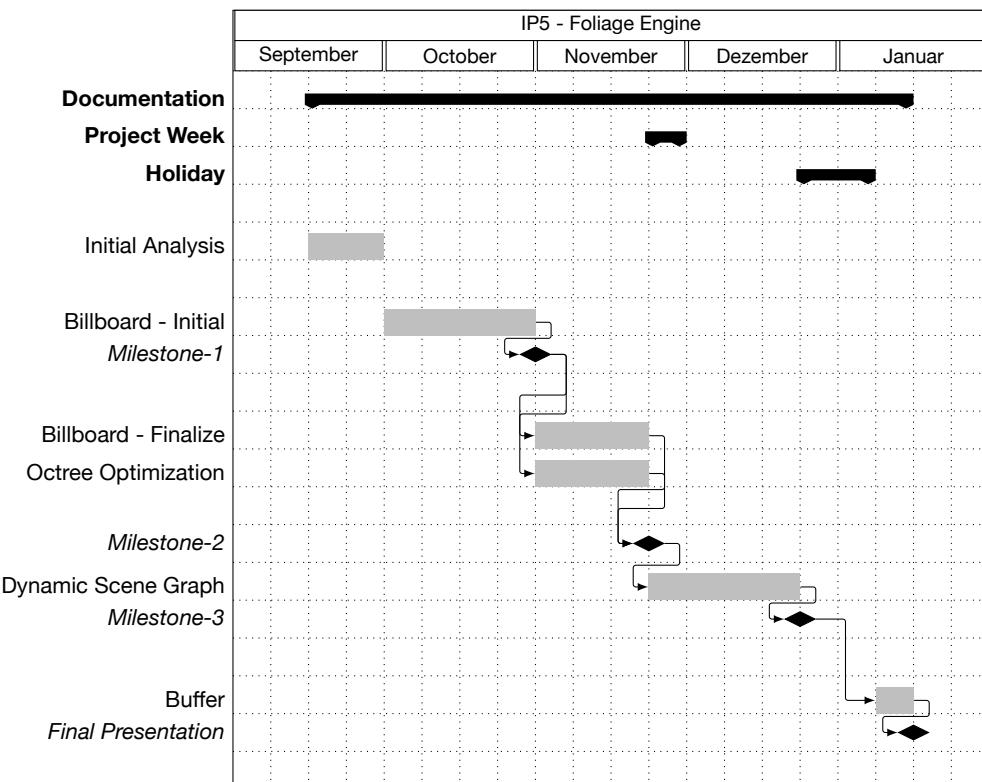
<b>Foliage Engine</b>	<b>2</b>
Introduction . . . . .	2
<b>Grobplanung</b>	<b>2</b>
Milestones . . . . .	2
Aktualisierte Grobplanung . . . . .	4
Aktualisierte Milestones . . . . .	4
<b>Analyse</b>	<b>5</b>
Ist-Analyse . . . . .	5
Ziel . . . . .	5
Struktur Sourcecode . . . . .	5
Debugging . . . . .	7
Performance . . . . .	7
Findings . . . . .	7
<b>Billboard</b>	<b>8</b>
Schritt 1) VertexShaderMaterial . . . . .	8
Schritt 2) Partikelsystem . . . . .	10
Schritt 3) Points . . . . .	10
Fail . . . . .	11
Schritt 4) Billboard/Sprite-Kombination . . . . .	11
Schritt 5) Billboard Implementierung . . . . .	11
<b>Octree</b>	<b>11</b>
Ausgangslage . . . . .	11
Update des LOD . . . . .	12
Ergebnis . . . . .	17
<b>Appendix</b>	<b>20</b>
Meetings . . . . .	20
Meeting (1) . . . . .	20
Meeting (2) . . . . .	21
Meeting (3) . . . . .	22
References . . . . .	23

---

## Foliage Engine

### Introduction

### Grobplanung



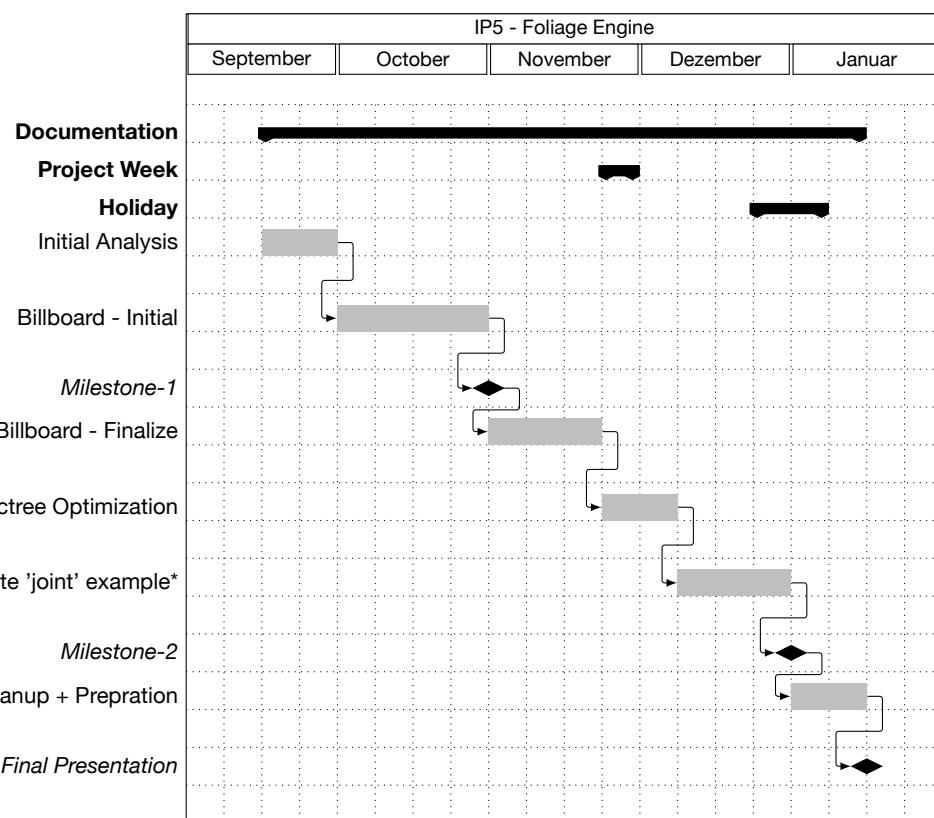
### Milestones

- **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.

## Aktualisierte Grobplanung

Auf Grund mehrerer Faktoren musste die Grobplanung grundsätzlich angepasst werden. Der Hauptgrund war dabei die Feststellung, dass die bestehende Foliage Engine schlecht erweiterbar ist (undurchsichtige Abhängigkeiten, hohe Komplexität der Implementation, Codequalität nicht zufriedenstellend etc.). Aus diesem Grund wurde zwischen dem Milestone 1 und Milestone 2 entschieden ein Example von Grund auf neu zu erstellen. Mit dieser Massnahme wurde sichergestellt, dass wir die Octree Erweiterung vornehmen können. Dabei wurde vereinbart, dass wir bestehende sinnvolle Konzepte und Überlegungen der Foliage Engine übernehmen. Der Milestone 3 wurde dabei mehrheitlich entfernt. Ziel ist jedoch, in unserer Implementation diese Erweiterbarkeit zu beachten.



## Aktualisierte 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:** Das Ziel ist es die beiden Teile (Billboard und Octree) in ein gemeinsames Beispiel zusammenzuführen.

- **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 Debuggmöglichkeiten 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.

#### Texturen

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 entweder 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:

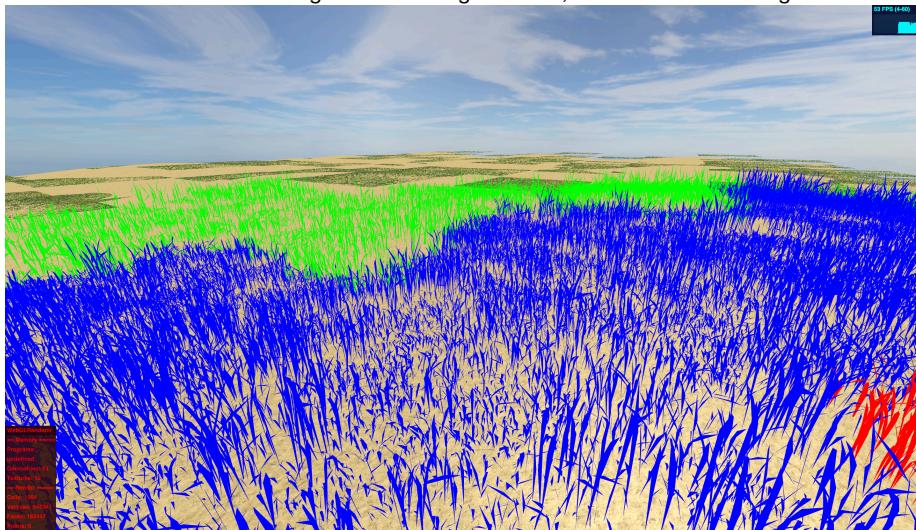
```

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:

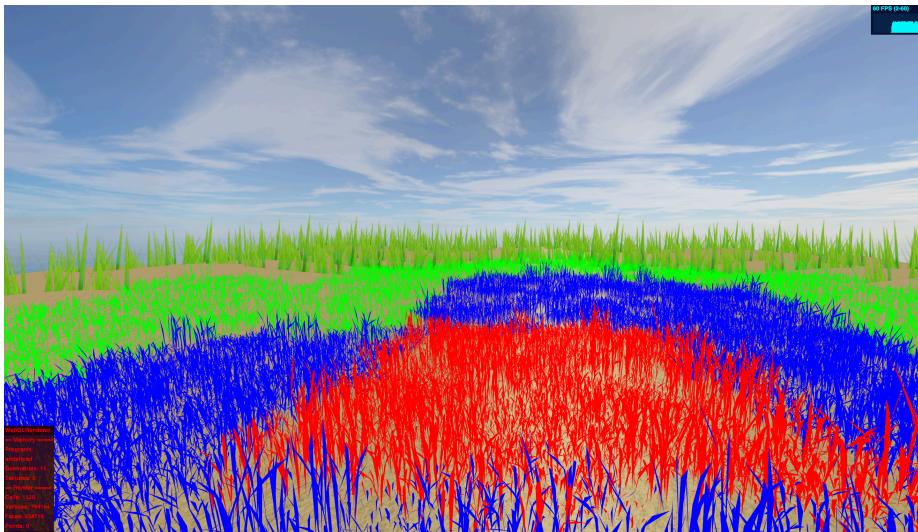


Figure 1: Resultat mit ShaderMaterial

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.PartikelSystem(geometry, material)` und verwendeten dabei wieder unser ShaderMaterial. Jedoch wurden wir dabei schnell von unserem Chrome-Browser beim testen darauf Aufmerksam gemacht, dass das PartikelSystem 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.sizeAutenuation = 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:

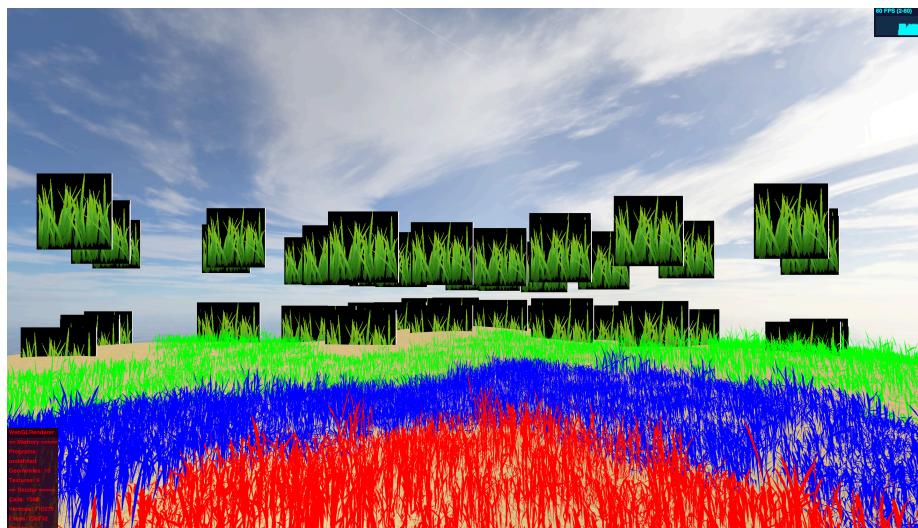


Figure 2: Resultat mit ShaderMaterial

## **Fail**

commit “a9446a61b578dec97f7f09f9617f28b7c59d3917”

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

TDB

### **Schritt 5) Billboard Implementierung**

TBD

## **Octree**

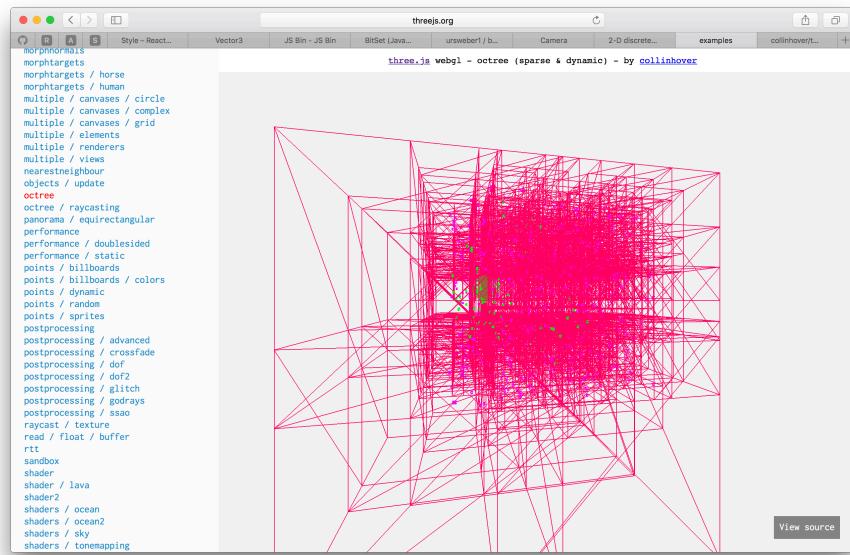
### **Ausgangslage**

Wir wollen einen Octree verwenden um damit effizient die Distanz von der Kamera zu einem Gras-Objekt zu berechnen. Die Elemente, die sich in einem Octree befinden, werden

dynamisch in Nodes aufgeteilt, die nie mehr als 8 Kinder haben können.

So entsteht ein Baum mit maximal 8 Kindern pro Node. Diese Nodes haben dann immer einen Raum den sie Belegen. Um jetzt effizient berechnen zu können, in welchem LOD so ein Node liegt, kann man jeweils die äusseren Punkte überprüfen. Liegt so ein Node-Raum in mehreren LOD so muss weiter bei dessen Kinder gesucht werden.

Als Ausgangslage verwendeten wir den Octree von [collinhover](#) das die ganzen Eigenschaften von einem Octree mit bringt. Was nicht vorhanden ist, sind Funktionen mit denen man den LOD der Nodes überprüfen kann und diese dann auch ein entsprechendes Event auf die betrof-



fenen Objekte auslöst.

*Octree example*

### Update des LOD

OctreeNode sind die Objekte welche alle notwendigen Informationen und die Kinder enthalten. Mit ihnen wird der eigentliche Octree aufgebaut.

Um nun unsere LOD-Überprüfungen auf dem OctreeNode machen zu können, müssen wir ihn um einen Parameter ergänzen.

```
this.levelOfDetail = parameters.levelOfDetail ? parameters.levelOfDetail : 0;
```

Die Informationen über den besetzten Raum von jedem Node können wir mit folgenden Attributen erhalten:

```
this.left;
this.right;
this.front;
this.back;
this.top;
this.bottom;
```

Mit diesen Argumenten lassen sich nun die Eckkoordinaten der Nodes ermitteln.

Will man nun die Objekte erhalten die in einem Node verpackt sind, macht man dies mit `this.objects[index].object`. In den Objekten sind jetzt die Daten enthalten die wir dem Octree übergeben haben. Dort müssen wir auch die Position des Objektes extrahieren um damit die Distanz zur Kameraposition zu berechnen. Damit wissen wir schon alles das wir für die Berechnung der entsprechenden LOD wissen müssen.

**Update LOD auf Octree** Die Berechnung des LOD wird mit der Funktion `updateLevelOfDetail(fromPosition)` ausgelöst. Initial wird diese von aussen auf den Octree aufgerufen, und übergibt dabei den aktuellen Standort der Kamera. Diese Funktion ruft wiederum die Funktion auf dem root auf, welcher ein OctreeNode ist.

```
// fromPosition: is usually the camera position
updateLevelOfDetail: function (fromPosition) {
    this.root.updateLevelOfDetail( fromPosition );
}
```

Bevor wir jedoch die Implementation davon auf dem OctreeNode anschauen, müssen wir einige andere notwendigen Funktionen zeigen.

**Distanz Berechnung** Zur berechnung der Distanz verwenden wir den euklidischen Abstand.

```
calculateDistance: function ( x1, y1, x2, y2 ) {
    var a = x1 - x2;
    var b = y1 - y2;
    return a * a + b * b;
}
```

Die ersten beiden Parameter repräsentieren die Kameraposition und die letzten zwei die Position des Objekts. Diese Distanz kann nun verwendet werden um das entsprechende LOD zu ermitteln.

Um das korrekte LOD berechnen zu können brauchen wir noch den Range der entsprechenden LOD's. Dafür haben wir dem Octree einen weiteren Parameter hinzugefügt `this.levelOfDetailRange;`, der bei der Instanzierung als Parameter mitgegeben wird. Damit wir aber diese Ranges nun mit den berechneten Distanzen verwenden können, müssen wir sie noch quadrieren.

```
function prepareMarshall( array ){
    var newArray = [];
    for(var i = 0; i < array.length; i++){
        newArray.push(array[i] * array[i]);
    }
    return newArray;
}
```

Dies wird einmalig bei der Instanzierung erledigt und dann so im Octree gespeichert. Dabei kann man auch Defaultwerte beim Octree angeben falls keine Ranges bei der Instanzierung mitgegeben werden.

```
this.levelOfDetailRange =
    prepareMarshall( parameters.levelOfDetailRange )
    || [1 ,25, 64, 2500]
```

Das entsprechende LOD kann mit diesen Informationen ermittelt werden. Dafür iterieren wir über unser Range-Array und suchen solange bis wir einen Wert finden der grösser als unsere Distanz ist. Das LOD geben wir dann in Form eines Integers zurück.

```
calculateLevelOfDetailFromDistance: function ( distance ) {
    var i, l;
    l = this.tree.levelOfDetailRange;
    for(i = 0; i < l.length; i++){
        if(distance <= l[i]) return i;
    }
    return i;
}
```

Damit können wir nun alle LOD der entsprechenden Ecken von einem OctreeNode berechnen. Nun brauchen wir eine Funktion die berechnet welches LOD der ganze OctreeNode hat. Dafür müssen wir für die berechnete Distanz der Ecken das LOD ermitteln und vergleichen ob alle das selbe haben, falls ja, hat der ganze OctreeNode das entsprechende LOD, falls nein, ist sein LOD undefined und sie müssen weiter unten in der Hierarchie definiert werden.

```
calculateLevelOfDetail: function ( fromPosition ) {
    var distanceFrontLeftEdge =
        this.calculateDistance(fromPosition.x, fromPosition.z, this.left, this.front);

    var distanceFrontRightEdge =
        this.calculateDistance(fromPosition.x, fromPosition.z, this.right, this.front);

    var distanceBackLeftEdge =
        this.calculateDistance(fromPosition.x, fromPosition.z, this.left, this.back);

    var distanceBackRightEdge =
        this.calculateDistance(fromPosition.x, fromPosition.z, this.right, this.back);

    var levelFrontLeftEdge =
        this.calculateLevelOfDetailFromDistance(distanceFrontLeftEdge);
    var levelFrontRightEdge =
        this.calculateLevelOfDetailFromDistance(distanceFrontRightEdge);
    var levelBackLeftEdge =
        this.calculateLevelOfDetailFromDistance(distanceBackLeftEdge);
    var levelBackRightEdge =
        this.calculateLevelOfDetailFromDistance(distanceBackRightEdge);

    var hasAllEdgesSameLevelOfDetail = levelFrontLeftEdge ===
        levelFrontRightEdge && levelFrontRightEdge ===
            levelBackLeftEdge && levelBackLeftEdge ===
```

```

        levelBackRightEdge;

    if(hasAllEdgesSameLevelOfDetail) {
        return levelFrontLeftEdge;
    } else {
        return undefined;
    }
}

```

Die folgende Funktion wandelt die Information nun in einen boolean um, den wir später für die Funktion auf den OctreeNode verwenden werden.

```

hasAllEdgesWithinSameLevelOfDetail: function ( fromPosition ) {

    var levelOfDetail = this.calculateLevelOfDetail(fromPosition);

    var hasAllEdgesSameLevelOfDetail = levelOfDetail === undefined ? false : true;

    return hasAllEdgesSameLevelOfDetail;
}

```

Weiter brauchen wir noch einen Funktion der Prüft, ob das berechnete LOD verglichen mit dem vorgängigen LOD gewechselt hat oder nicht.

```

willLevelOfDetailChange: function ( from, to ) {
    return from !== to || to === undefined;
}

```

Damit haben wir alle Funktionen die wir brauchen um eine `updateLevelOfDetail( fromDistance )` Funktion auf dem OctreeNode zu definieren.

**Update LOD auf OctreeNode** Um zu prüfen ob ein OctreeNode ein Update braucht, speichern wir das alte und neue LOD, falls der Node nicht undefined ist. Danach prüfen wir ob sich das LOD verändert hat; falls ja lösen wir einen Event aus, falls nein machen wir nichts. Falls der OctreeNode undefined ist, rufen wir `updateLevelOfDetail( fromDistance )` auf dessen Kinder auf.

```

updateLevelOfDetail: function ( fromPosition ) {

    if ( this.hasAllEdgesWithinSameLevelOfDetail ( fromPosition ) ) {
        var oldLevelOfDetail = this.utilLevelOfDetail.nodeLevelOfDetail;
        var newLevelOfDetail = this.calculateLevelOfDetail ( fromPosition );
        var didLevelOfDetailChange = this.willLevelOfDetailChange( oldLevelOfDetail, newLevelOfDetail );

        // we do not need to go any deeper -> all children will be in this LOD
        this.utilLevelOfDetail.nodeLevelOfDetail = newLevelOfDetail;
    }
}

```

```

        if (didLevelOfDetailChange) {
            this.dispatchEvent();
        }

    } else {

        // we need to go deeper
        var i, l; l = this.nodesIndices.length;
        if(l > 0){
            for ( i = 0; i < l; i ++ ) {
                this.nodesByIndex[ this.nodesIndices[ i ] ].updateLevelOfDetail( fromPosition );
            }
        } else {
            var k = this.objects.length;
            for(i = 0; i < k; i++) {
                var obj = this.objects[i].object;
                var dist = this.calculateDistance(obj.position.x, obj.position.z, fromPosition);
                var lod = this.calculateLevelOfDetailFromDistance(dist);
                obj.dispatchEvent( {
                    type: 'changed',
                    level: lod
                });
            }
        }
    }
}

```

**Event** Für das Event, dass ausgelöst wird, verwenden wir den von Threejs angebotenen `EventDispatcher`. Damit können wir ein Event direkt auf dem Objekt selbst auslösen. Um nun mit dem Ausgelösten Event etwas Sinnvolles anstellen zu können, brauchen wir eine Callback-Funktion die dem Octree bei der Instanzierung übergeben werden muss. Wird kein Callback übergeben, so wird diese durch eine leere Funktion ersetzt.

```
this.levelOfDetailChangedCallback = parameters.levelOfDetailChangedCallback || function () {
```

Dieser Callback wird direkt auf den Objekten im Octree ausgelöst und haben mit `this` direkten Zugriff auf dessen Attributen. So lassen sich die nötigen Schritte in der Function definieren die bei einem Event erledigt werden sollen. Das `dispatchEvent`, dass im `updateLevelOfDetail( fromPosition )` auf einem `OctreeNode` aufgerufen wird, iteriert über alle Objekte im `OctreeNode` und löst auf ihnen das Callback aus. Danach wird es rekursiv nach Unten weiter aufgerufen. Beim `dispatchEvent` wird noch ein Event-Objekt mitgegeben, dass das Level des Objektes enthält.

```
dispatchEvent: function () {
    var i, l;

    for(i = 0, l = this.objects.length; i < l; i++){
```

```

        this.objects[i].object.dispatchEvent( {
            type: 'changed',
            level: this.utilLevelOfDetail.nodeLevelOfDetail
        });

    }

    for ( i = 0, l = this.nodesIndices.length; i < l; i ++ ) {

        this.nodesByIndex[ this.nodesIndices[ i ] ].dispatchEvent();

    }

}

```

### Ergebnis

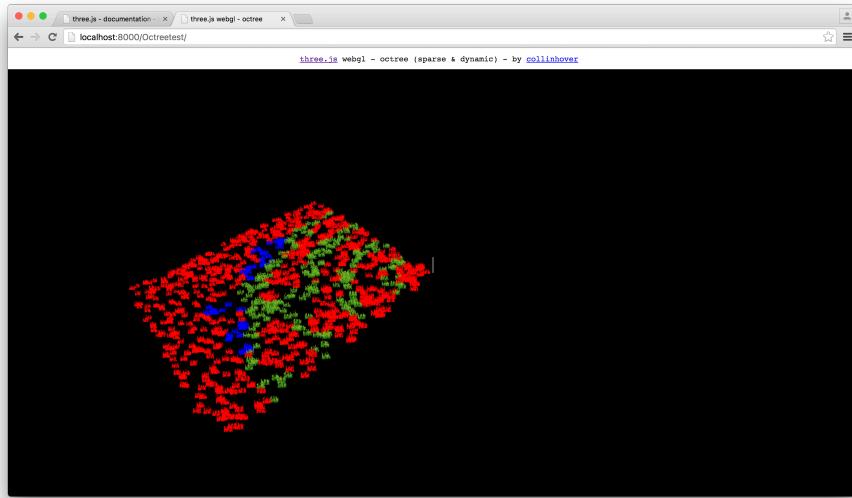
**Test 1** Um zu sehen ob sich unser Octree so verhält wie erwartet, haben wir eine das Example welches schon in der **Ausgangslage** gezeigt wurde, an unsere Situation angepasst. Wir lassen alle Punkte nur noch auf einer Ebene generieren und definierten eine Callback-Funktion die beim Auslösen des Events die Farbe der Objekte anhand des entsprechenden Levels ändert.

```

levelOfDetailChangedCallback: function ( event ) {
    if(event.level >= 4){
        this.material.color.setRGB( 0, 0, 255 ); // Blau
        console.log("updated LOD 4");
    } else {
        this.material.color.setRGB( 255, 0, 0 ); // Rot
        console.log("updated LOD 0 - 3");
    }
}

```

Ausgehend von der weißen Linie, die unsere Cameraposition imitiert, erwarten wir einen Viertelkreis in Rot und der Rest sollte Blau sein. Diesen Erfolg hatten wir leider nicht. Wie man auf dem Bild erkennen kann, sieht man zwar die Kreisform, die wir von unserem Octree erwarten, es hat jedoch noch grüne Bereiche, die nie ein Update erhalten und dann einen grossen roten Bereich, der eigentlich Blau sein sollte.



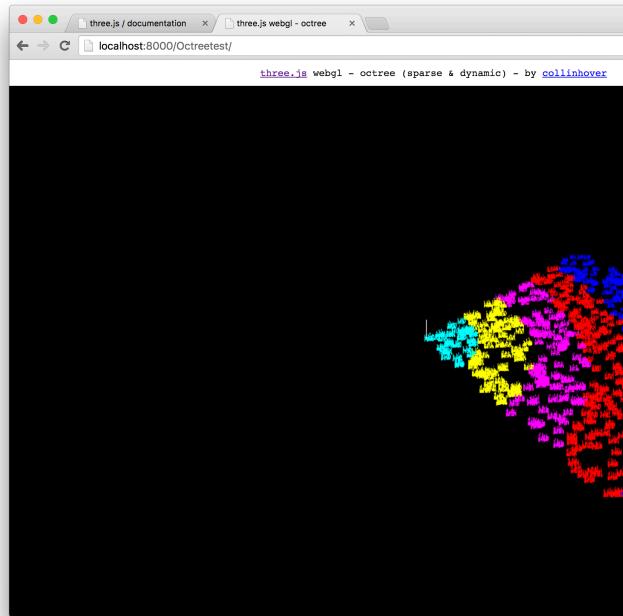
**Test 1** Dabei erkannten wir, dass wir beim LOD überprüfen stetig beim OctreeNodebleiben und diese immer eine Fläche repräsentieren. somit sind die Objekte welche sich an der Grenze des Ranges befinden immer undefiniert. Dies konnten wir später im Sourcecode beheben. Weiter hatten wir auch noch das Problem, dass Updates mit dem Event “undefined” ausgelöst wurden. Dabei erkannten wir das wir bisher die Octreeeigenschaft von Split und Expand vernachlässigt hatten. Dies konnte dann jedoch einfach gelöst werden.

Bei Expand mussten wir beim neu erstellten Octree den LOD default auf undefinedsetzen.

```
parent = new THREE.OctreeNode( {
    tree: this.tree,
    position: position,
    radius: radiusParent,
    levelOfDetail: undefined
} );
```

Beim Fall von einem split, wird in der funktion eine branch funktion aufgerufen, die dann einen neuen OctreeNode erstellt, auch dort konnten wir das Problem einfach beheben, indemm der neue Node einfach den LOD vom alten Node übernimmt.

```
node = new THREE.OctreeNode( {
    tree: this.tree,
    parent: this,
    position: position,
    radius: radius,
    indexOctant: indexOctant,
    levelOfDetail: this.levelOfDetail
} );
```



Nun haben wir ein erwartetes Verhalten als Ergebnis erhalten.

# 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**