Projektabschlussbericht

Universe Simulation

02.01.2020

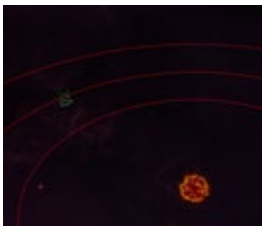
1 Management- und Dokumentationsattribute

Dokumentationsattribute	
Autor(en)	Simon Weck, Paul Amelingmeyer, Patryk Watola, Benjamin Jäger
Eindeutige Teamnummer	C05
Namen der Teammitglieder	Simon Weck, Paul Amelingmeyer, Patryk Watola, Benjamin Jäger

2 Problemstellung

Die grundlegende Problemstellung des Projektes bestand darin, die Gebiete „Computergrafik“ und „Bildverarbeitung“ in einer gemeinsamen Anwendung sinnvoll zu kombinieren.

Hierbei wird die Bildverarbeitung genutzt, um ein vom Nutzer handgezeichnetes Bild einzulesen. Anhand der aus dem Bild analysierten geometrischen Parameter wird in der Computergrafik ein Modell erstellt. Der besondere Reiz der Anwendung liegt darin, dass eine ausgesprochen einfache und für fast jede zugängliche Art der „Eingabe“ (Stift und Papier) genutzt wird, um komplexe Ergebnisse zu erzielen.

Als Thema wurde hier die Erstellung von Planetensystemen gewählt. Der Schwerpunkt liegt hierbei nicht auf der physikalisch korrekten Simulation eines Sonnensystems, sondern darauf, dem Nutzer viele kreative Freiheiten zu lassen, optisch ansprechende unterhaltsame Bilder zu generieren. So kann der Nutzer beispielsweise mehrere Sonnen um einen Kristallplaneten kreisen lassen und betrachten, wie dieser von den Sonnen beleuchtet wird, oder das animierte Wasser auf einem erdähnlichen Planeten bestaunen.

Eine Anwendung wäre im pädagogischen Bereich denkbar. Schüler könnten spielerisch und kreativ an physikalische Problemstellungen wie die Himmelsmechanik herangeführt werden.

~ Paul Amelingmeyer

3 Abweichungen zum Pflichtenheft

Bildverarbeitung

Im Pflichtenheft war geplant neben der Art der Planeten auch Symbole zu erkennen, die Besonderheiten zu einem Planeten hinzufügen können (z.B. ein Vulkan, welcher auf den Planeten gesetzt wird). In der finalen Version wird lediglich die Art eines Planeten mittels der Farbe des Kreises erkannt, da eine das Zeichnen von Symbolen innerhalb eines Kreises dessen Erkennung erschwert. (Siehe Lösungsstrategie)

Computergrafik

Es war geplant einen Modus zu implementieren, in welchem der Nutzer einen Planeten aus der First-Person-Sicht hätte erkunden sollen. Dabei sollte es eine richtige Kollision mit der Oberfläche des Planeten geben, sodass der Benutzer auf der Oberfläche “spazieren gehen” und sich alles genauer anschauen kann. Dieser Modus hat es nicht in das Finale System geschafft, ist aber als „Future Work” eine gute Idee.

~ Paul Amelingmeyer

4 Verwendete Technologie

OpenJDK 11 (LTS) from Adopt OpenJDK (adoptopenjdk.net)

4.1 Bildverarbeitung:

-OpenCV 3.3.0

4.2 Computergrafik:

-Jogl 2.3.2 (jogamp.org)

-JavaFX11 (openjfx.io)

-JavaFX um Videos in Swing einzubinden

-Java Swing für die Oberfläche

-Webcam capture 0.3.12 (github.com/sarxos/webcam-capture)

-Bibliothek, um Kameras in Java Swing einzubinden

5 Lösungsstrategie

In der Bildverarbeitung traten einige Probleme auf bezüglich unvollständig erkannten Linien und später bezogen auf die weitere Verarbeitung der Bilddaten für den mittels OpenGL realisierten Teil der Software.

5.1 Linienerkennung

Das erste größere Problem zeigte sich bei der Linienerkennung als der HoughLinesP-Algorithmus nur Teile der Linie erkannte und die Endpunkte nicht eindeutig zu bereits erkannten Kreisen zugeordnet werden konnten. Der erste Lösungsansatz war ein Rechteck aufzuspannen, welches einen Bereich zwischen den Mittelpunkten zweier Kreise begrenzt und dann zu überprüfen, ob der End- und Startpunkt innerhalb des besagten Rechtecks liegt. Dazu ist zunächst die Fläche des Rechtecks bestimmt worden. Dann wurden Dreiecke

zwischen zwei beieinander liegenden Eckpunkten des Rechtecks und einem Endpunkt der Gerade gebildet. Auf diese Weise erhält man vier Dreiecke und die Summe der Flächen dieser Dreiecke muss gleich der Fläche des Rechtecks sein, damit der Punkt innerhalb des Rechtecks liegt. Wenn beide Punkte innerhalb des Rechtecks lägen, würden beide Planeten als eine „Relation" (die Datenklasse zum Speichern von verbundenen Kreisen) gespeichert.

Dies ergab dann aber neue Probleme, da viele Ausnahmefälle existierten, welche zu großen Teilen nicht sinnvoll abgedeckt werden konnten.

Die neue, und finale, Lösungsstrategie war in der Implementierung theoretisch ineffizienter als der erste Ansatz, erzielte in der Praxis aber trotzdem sehr gute Zeiten. Dabei wurden die einzelnen Liniensegmente durch Vektoren künstlich verlängert. Zuerst wird ein Vektor von dem Start- zu dem Endpunkt eines Liniensegments gebildet. Der Vektor wird dann auf den Startpunkt aufaddiert bzw. davon subtrahiert bis er sich innerhalb eines Kreises befindet. So wird in beide Richtungen geprüft, welche Planeten verbunden sind. Läuft ein Vektor aus dem Bild heraus, so wird das Liniensegment ignoriert.

~ Patryk Watola

5.2 Strukturierung

Das andere größere Problem war die ordentliche Strukturierung der Daten für die Weiterverarbeitung. Nach den ersten zwei Schritten, welche Kreise und Relationen erkannt haben, war noch nicht bekannt, welcher Planet sich um welchen dreht und wie die Planetennetze zusammenhängen.

Der Lösungsansatz hier war eine Art Baumstruktur aufzubauen, indem jedem Objekt mitgegeben wird, welche Objekte sich um es drehen (die "Children"). Diese wurden direkt im Objekt gespeichert. Begonnen wird mit einem Planeten, welcher ein Zentrum ist. Das wird im Voraus bestimmt. Dann werden in den Relationen alle Objekte herausgefiltert, welche mit dem Zentrum zusammenhängen und die Daten für diese (Distanz zum Planeten, um welchen sie kreisen zum Beispiel) gespeichert. Dasselbe wird mit all den neu gefundenen Planeten wiederholt und dann ebenso mit deren "Children" und so weiter.

Zum Schluss wird dann noch überprüft ob Planeten existieren, welche noch nirgendwo zusammenhängen. Aus diesen wird dann wieder ein neues Zentrum ermittelt und die Prozedur wiederholt, bis alle Planeten sich in einem Planetensystem befinden (ein Planetensystem kann auch aus nur einem Planeten bestehen).

~ Patryk Watola

5.3 Erkennung des Planetentypen

Ein weiteres Problem war die Erkennung des Planetentypen. Hierzu wurden Algorithmen genutzt, welche verschiedene Shapes wie Dreiecke, Vierecke, Fünfecke etc. erkennen. Beim Testen wurde jedoch klar, dass die Kreiserkennung in Kombination mit der Shape-Erkennung zu Problemen führt, da Kreise teilweise nicht erkannt werden, oder die Symbole innerhalb dieser fälschlicher weise als Kreise erkannt werden. Nach einigen Tests wurde der Schluss gezogen , dass es zuverlässiger funktioniert, wenn die Kreise in verschiedenen Farben gezeichnet werden, und diese dann als Erkennungsmerkmal für die verschiedenen Planeten verwendet.

Des Weiteren wird eine Möglichkeit benötigt das Zentrum des Systems festzulegen, damit im Anschluss die Planeten-Hierarchie korrekt aufgebaut werden kann. Hierzu war zunächst

geplant das Zentrum mit einem „X“ zu markieren. Ähnlich wie zuvor interferierte dies mit der Kreiserkennung. Hierzu wurde jedoch recht schnell eine zuverlässigere und intuitivere Lösung gefunden: Das Zentrum eins Systems ist immer der größte Planet.

~ Paul Amelingmeyer

6 Algorithmus

6.1 Bildverarbeitung

6.1.1 Relationen erkennen:

Liste mit Liniensegmenten LINIENLISTE;
for (jede LINIE in LIENIENLISTE) {
    bestimme Vektor von Start bis Endpunkt der Linie;
    gehe vom Startpunkt der Linie in Richtung des Vektors bis innerhalb eines Kreises oder außerhalb des Bildes;

    Wenn in Kreis:
    Gehe von Startpunkt in umgekehrte Richtung des Vektors bis innerhalb eines Kreises oder außerhalb des Bildes;
    Wenn in zweitem Kreis: speichere beide Kreise und deren Distanz als Relation;
    Wenn außerhalb des Bilds: verwerfe Liniensegment;

    Wenn außerhalb des Bilds: verwerfe Liniensegment;
}

~ Patryk Watola

6.1.2 Ansatz Relationen erkennen vorher (nicht verwendet, weil es nicht funktioniert hat):

Rechteck aufspannen von Kreismittelpunkt A und B mit beliebiger Breite;
Fläche von Rechteck bestimmen;
Start- und Endpunkt des Liniensegments nehmen;
Dreiecke bilden zwischen jeweils zwei nebeneinander liegenden Punkten des Rechtecks und dem Punkt auf der Linie;
Überprüfen ob Summe der vier Dreiecksflächen mit Rechtecks Fläche übereinstimmt;
Falls ja: Relation zwischen beiden Planeten bilden;
Falls nein: andere Planetenkombinationen versuchen;

Erklärung: HoughLinesP erkannte nicht immer ganze Linien (selbst bei unterschiedlichen Werten) daher war der Algorithmus nötig.
Der aktuell verwendete Ansatz verfolgt eine Art „Raytracing”, welche durch Vektoren in Richtung des Geradensegments (Von Start- zu Endpunkt) einen Ray erzeugen. Dadurch wird geprüft auf welche zwei Kreise das Liniensegment zeigt. Diese werden dann gespeichert, um später ordentlich organisiert zu werden.
Der zweite Ansatz ist effizienter was die Laufzeit betrifft, da er beinahe ausschließlich ganzzahlige Operationen verwendet und nur wenige for-Schleifen ineinander verschachtelt

sind. Daher wäre es in Zukunft eine Möglichkeit den Algorithmus funktionsfähig zu machen, obwohl es sich nur um eine minimale Zeitersparnis handeln würde.

~ Patryk Watola

6.1.3 Planetentyp erkennen

Um den Planetentyp zu erkennen wird die Farbe des Kreises ausgewertet. Hierzu wird zuerst das Ausgangsbild in den HSV-Farbraum konvertiert. Anschließend wird geprüft welchen Farbwert das Pixel im Zentrum des Kreises hat. Dieser wird von open Cv mit einer Zahl zwischen 0 und 180 beschrieben.
Der erste Lösungsansatz war diese Zahl durch 30 zu teilen und das Ergebnis anschließend kaufmännisch zu runden. Somit würde man eine Zahl zwischen 1 und 6 erhalten von der jede eine der Primär- und Sekundärfarben darstellt.
Dies führte allerdings zu ungenauen Ergebnissen, da nicht jede Farbe einen gleich großen Anteil im Farbkreis hat. Der Anteil der Farbe Grün ist beispielsweise um ein Vielfaches größer als der von Gelb. Dies führte dazu, dass eigentlich grüne Planeten von der Software fälschlicher Weise als Gelb erkannt wurden.
Aus diesem Grund wird nun mit if-Abfragen geprüft in welchem Bereich sich der Farbwert aufhält.

~ Paul Amelingmeyer

6.1.4 Planetensysteme organisieren:

Liste aller Relationen RELATIONEN;
Liste aller Planeten die ein Zentrum sind ZENTREN;
Liste an Planeten, welche in einem System sind, aber noch nicht bearbeitet wurden TODO;

```
while (Noch Planeten unbearbeitet sind) {
        while (Noch nicht abgearbeitete Planeten in TODO) {
                for (jeden Planeten in TODO) {
                        for (jede Relation in RELATIONEN) {
                        Beinhaltet aktuelle Relation den aktuellen Planeten?;
                        Ja: {
                        Füge Planeten aus Relation als “Child” des aktuellen Planeten hinzu;
                        Füge Planeten aus Relation zu TODO hinzu }
                        }
                }
        }

        Gibt es noch unbearbeitete Planeten die nicht Teil eines Planetensystems sind?;
        Ja: {
                Bestimme neues Zentrum;
                Füge Zentrum zu Zentren hinzu;
                Füge Zentrum zu TODO hinzu;
        }
        Nein: beende while-Schleife;
}
```

Erklärung: Diese Stelle ist der “kritische Moment”, da hier beide Teilprojekte aufeinandertreffen. Was sich am besten dafür angeboten hat war eine Baumklasse, da die Zeichnungen für die Erstellung streng genommen nichts anderes als aufgefächerte Baumstrukturen sind. Die Planeten, welche um etwas kreisen, werden direkt in besagtem Planeten gespeichert und diese so miteinander verkettet. Am Ende müssen lediglich die Zentren, welche jeweils die größten Planeten ihres Planetensystems sind, ausgelesen und die „Children”/ Planeten, welche um besagten Planeten kreisen verfolgt werden.
Um diese Struktur zu erreichen wird zuerst ein Planetenzentrum bestimmt (der größte Planet im Bild) und alle Relationen werden nach den Partnern durchsucht. Dabei wird jede Relation in einer for-each Schleife geprüft und die gefundenen Partner in einer TO-DO Liste gespeichert. Diese werden dann im nächsten Schritt auf dieselbe Art und Weise abgearbeitet.
Am Ende wird nur noch überprüft ob Planeten existieren, welche weder Teil eines Planetensystems sind noch ein Zentrum bilden bzw. alleinstehen. Dort wird dann wieder ein neues Zentrum gebildet und der Algorithmus von vorne gestartet.
Da eine solche Verkettung etwas unüblich ist könnte man in Zukunft über eine speziell dafür programmierte Datenstruktur nachdenken, um es etwas besser zu organisieren.

~ Patryk Watola

6.1.5 Normalisieren der Größen

```
Liste aller Planeten als Planetenliste;
double größteZahl = -1;
for (jeden Planeten in Planetenliste ) {
        wenn Größe des Planeten > größteZahl: setze größte Zahl auf Größe des Planeten;
        wenn Distanz zu Vorgänger > größteZahl: setze größte Zahl auf Distanz zu Vorgänger;
}

for (jeden Planeten in Planetenliste ) {
        setze Größe des Planeten auf Größe/größteZahl;
        setze Distanz des Planeten auf Distanz /größteZahl;
}
```

Erklärung: Auf diese Art und Weise sind alle Werte irgendwo zwischen 0 und 1 und lassen sich nachher beliebig skalieren. Absolute Werte (in diesem Fall die Größe in Pixeln) sind suboptimal für die Skalierung.
Vor allem wichtig wird dies, wenn mehrere größere/ komplexere Planetensysteme nebeneinander existieren sollen.

~ Benjamin Jäger

6.1.6 Linien(-segmente) erkennen:

```
Schritte zur Bildverbesserung für Algorithmus;
Erkenne Linien in Eingabebild mit HoughLinesP und speichere in Ergebnismatrix;

for (jedes Punktepaar aus Start- und Endpunkt in Ergebnismatrix) {
```

```
        Bilde “LineData”-Objekte;
        Speichere Start- und Endpunkt in “LineData”-Objekt;
}
```

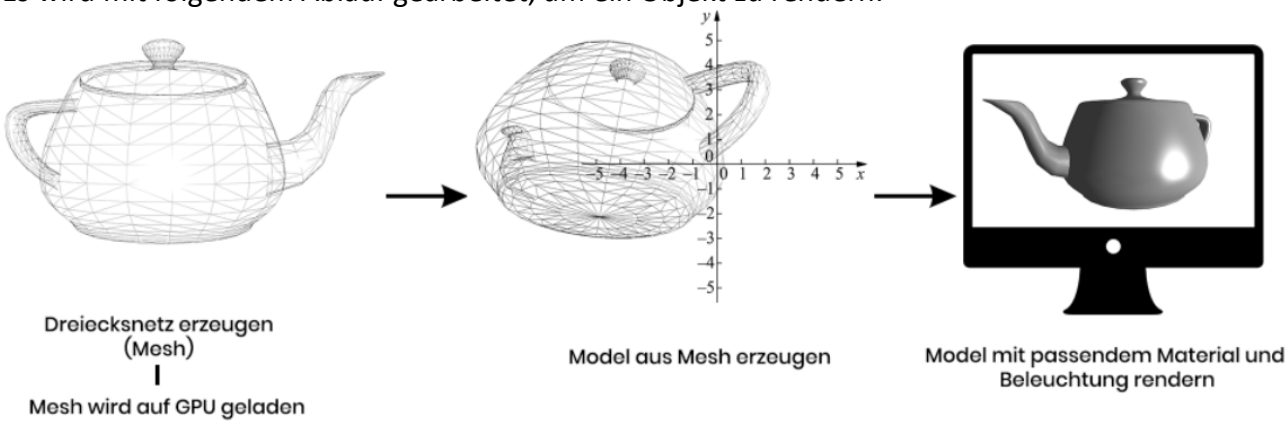
Erklärung: Die einzelnen Liniensegmente werden als eigene Klasse abgespeichert, um später besser weiterarbeiten zu können. OpenCV speichert die Daten in einer 4*x großen Matrix, welche mit rows() und cols() mit zwei ineinander verschachtelten for-Schleifen durchlaufen werden muss.
Da die Daten aus dieser Matrix häufiger benötigt werden und sie folglich häufiger durchlaufen wird, würde eine beachtliche Menge an zusätzlicher Zeit aufzubringen sein. Indem sie also einmal durchlaufen wird und alle separat abgespeichert werden reicht später nur eine for-each-Schleife. Da die Zeitersparnis besonders wichtig ist, ist dieser Zwischenschritt notwendig.

~ Patryk Watola

7 Computergrafik

7.1 Allgemeine Render Struktur

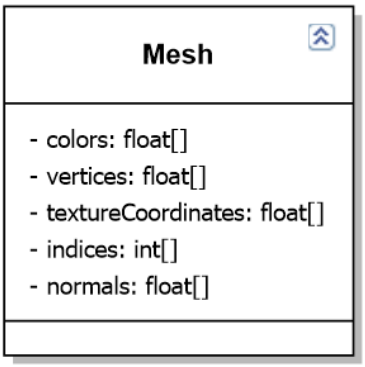
Es wird mit folgendem Ablauf gearbeitet, um ein Objekt zu rendern.



Ablauf, um ein Objekt zu rendern

Der erste Schritt besteht darin ein Mesh Objekt zu erzeugen, welches Informationen über die Eckpunkte des Netzes enthält. Ein Eckpunkt im Mesh wird definiert durch die Position (XYZ), Farbe (RGB), Normal Vektor (XYZ) und ggf. Textur Koordinaten. Ein Netz besteht aus einer Vielzahl an Eckpunkten. Wird im weiteren Verlauf von Vertices gesprochen, ist die Position des Eckpunktes gemeint. Ein weiteres Element eines Mesh Objektes ist eine Index Liste, die beschreibt wie die Eckpunkte verbunden werden sollen. Im weiteren Verlauf sind Indices Inhalte dieser Index Liste.

~ Benjamin Jäger



wichtige Klassenattribute der Mesh Klasse

```
public Mesh(float[] vertices, int[]indices ) {
    baseVertices=vertices;
    this.indexCount = indices.length;
    this.vertices=vertices;
    this.indices=indices;
    calculateNormals();
    loadToGPU.loadMeshToGPU(this);
}
```

Meist genutzter Konstruktor der Mesh Klasse

Für die meisten Objekte wird der obige Konstruktor genutzt, welcher zuvor berechnete Vertices und Indices annimmt. Dieser berechnet auch aus den Vertices und Indices die Normalen, welche später für die Beleuchtung genutzt werden.

Die wichtigste Methode hier ist die statische Methode *loadMeshToGPU(Mesh mesh).* Diese Methode erstellt ein Vertex Array Object, welches mehrere Vertex Buffer Objects besitzt. Diese Vertex Buffer Objects (kurz VBO) sind Puffer auf der GPU, welche dazu gemacht sind Informationen über ein zu renderndes Objekt zu speichern.

Vertex Array Object (VAO)	
0	Vertices (x0,y0,z0,x1,y1,z1.....xn,yn,zn)
1	Normals (x0,y0,z0,x1,y1,z1.....xn,yn,zn)
2	Colors (r0,g0,b0,r1,g1,b1.....rn,gn,bn)
3	Texture Coordinates (u0,v0,u1,v1....un,vn)

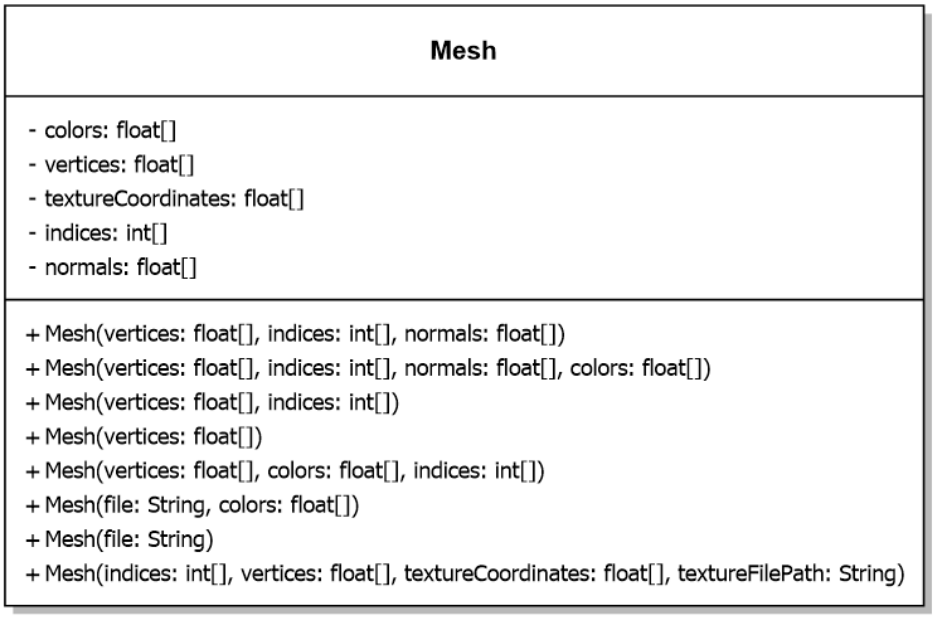
← Vertex Buffer Object (VBO)

Aufbau eines Vertex Array Objects

Oben ist der Aufbau von fast allen VAOs abgebildet. Einige spezielle Objekte benötigen mehr VBOs, aber darauf wird später eingegangen. Auch Textur Koordinaten werden nur für ein Objekt im System gebraucht.

Durch eine Vielzahl an Konstruktoren können viele Netze mit unterschiedlichen Eigenschaften erstellt werden. Diese Netze werden dynamisch abhängig von ihren Eigenschaften auf die GPU geladen. Beispielsweise ein Netz mit Textur oder ein anderes Netz welches Vertices, Indices und Farbwerte besitzt. Des Weiteren speichert ein Mesh Objekt

auch die IDs der Erstellten VAOs und VBOs, um sie im Verlauf des Programms noch aktualisieren zu können.



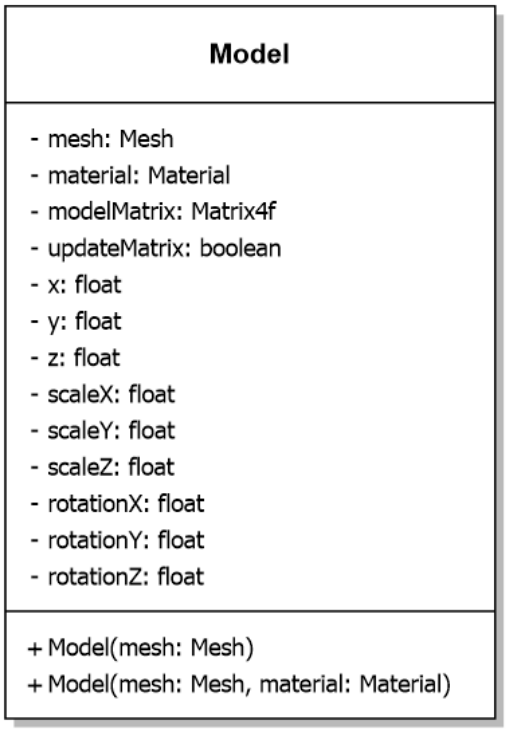
Konstruktoren der Mesh Klasse

Ein besonderer Konstruktor ist *Mesh(String file).* Hier übergibt man einen String der den Dateipfad zu einer .obj Datei angibt. Eine OBJ Datei ist ein offenes Dateiformat zum Speichern von dreidimensionalen geometrischen Formen. Ein Modifizierter OBJ-Loader aus dem Buch "Computer graphics programming in opengl with java" von V. Scott Gordon und John Clevenger durchsucht die Datei und befüllt Vertices, Indices, Normals und, falls vorhanden, Textur Koordinaten Arrays.

```
public Mesh(String file) {
    OBJParser parser=new OBJParser(file);
    vertices=parser.getVertices();
    indices=parser.getIndices();
    indexCount=indices.length;
    normals=parser.getNormals();
    loadToGPU.loadMeshToGPU(this);
}
```

Ein Mesh Objekt aus einer .obj Datei

Aus dem erzeugten Mesh Objekt wird nun ein Model Objekt erzeugt. Ein Model Objekt speichert auch die Transformation des Netzes. Außerdem besitzt ein Model Objekt auch ein Material, welches die Oberflächenbeschaffenheit des Netzes festlegt. Dieses Model Objekt kann dem Renderer übergeben werden, und dieser rendert es mit passender Transformation auf dem Display.

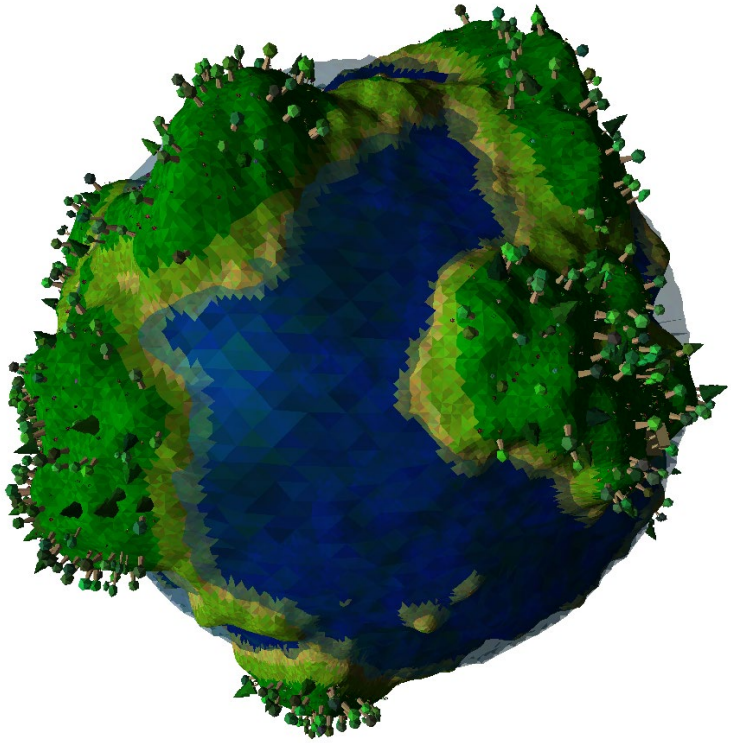


wichtige Klassenattribute der Model Klasse (unvollständig)

Ein Model besitzt x,y,z Translationen, Rotationen um die x-,y- und z- Achse sowie x,y,z Skalierungen. Die Model Klasse besitzt neben den im obigen Diagramm angegebenen Funktionen noch einige Konstruktoren in denen Skalierung, Rotation und Translation bereits angegeben werden können. Wichtig sind noch die Attribute *modelMatrix(Matrix4f)* und *updateMatrix boolean).* Die modelMatrix ist eine 4×4 Matrix bestehend aus floats. Diese Matrix fast alle Transformationen des Dreiecksnetzes in einer Matrix zusammen. Der Boolean wert *updateMatrix* besagt ob sich die Transformation des Objektes geändert hat. Ist dies der Fall, wird die *modelMatrix* beim nächsten Rendering erneut aus den Transformationswerten der Model Klasse errechnet.

~ Simon Weck

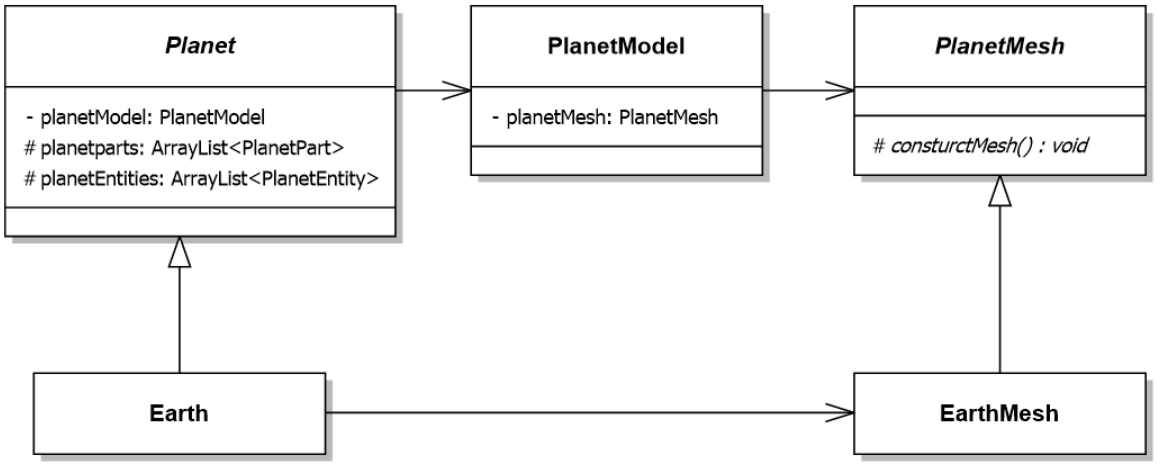
7.2 Prozedurale Generierung von Planeten



Beispiel eines erdähnlichen Planeten aus dem Programm

Ein Planet ist eine Einheit die aus „PlanetParts", „PlanetEntities" und dem Planeten selbst besteht. PlanetEntities sind dabei alle Objekte, die auf den Planeten gesetzt werden, also Bäume, Blumen oder auch Süßigkeiten wie Donuts. Als PlanetParts werden Objekte bezeichnet, die zu einem Planeten gehören aber nicht direkt auf den Planeten platziert werden. Dies sind zum Beispiel Ringe um einen Planeten oder das Wasser des oben abgebildeten Planeten.

Jeder Planet besitzt unterschiedliche Eigenschaften, jedoch haben alle Planeten eines gemeinsam: Sie werden prozedural generiert. Das bedeutet dem Computer wird eine ungefähre Bauvorschrift für die Planeten gegeben und er erzeugt aus dieser unterschiedliche Planeten. Beim oben abgebildeten Planeten könnte der nächste Planet der selben Sorte viel niedrigere Berge besitzen, somit ein viel größeren Wasseranteil, oder es werden nur wenige Bäume, aber viele Pilze auf dem Planeten generiert. Um nun einen dieser Planeten zu generieren wird als erstes der Planet selbst erstellt und danach PlanetParts sowie PlanetEntities hinzugefügt.

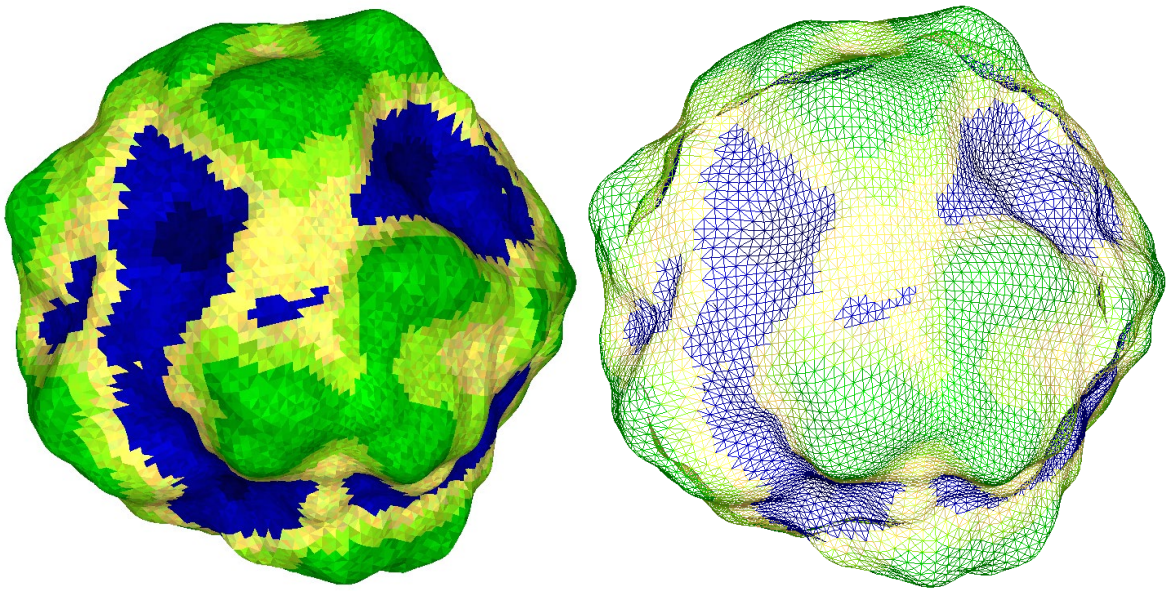


Bestandteile eines Planeten

Das oben abgebildete Klassendiagramm zeigt die Bestandteile eines Planeten um einen Planeten ohne PlanetParts und PlanetEntities zu erzeugen. Jeder Planet besitzt ein PlanetModel welches aus einem PlanetMesh besteht. Ein PlanetMesh ist das Dreiecksnetz des Planeten. Es enthält Vertices, Indices, die Normalen sowie Farbwerte des Netzes und Methoden, um dieses Netz zu erzeugen. Die Überklasse PlanetModel transformiert dieses Netz, also rotiert, skaliert oder translatiert es. Ein PlanetMesh besitzt auch ein Mesh Objekt welches wie bereits erklärt die Daten an die GPU weiterleitet. Diese Aufteilung in Mesh und Model wird später gebraucht, um ein Objekt zu rendern. Die Klasse Planet besitzt nun ein PlanetModel sowie die PlanetEntities und PlanetParts, um einen gesamten Planeten zu erzeugen. Ein erdähnlicher Planet besitzt also ein eigenes Dreiecksnetz (EarthMesh), welches erdähnliche Dreiecksnetze konstruiert. Das PlanetModel besitzt dann eines dieser Dreiecksnetze und die Überklasse Earth besitzt wiederum dieses PlanetModel. Die Abstrakte Methode *constructMesh()* in PlanetMesh ist dabei die wichtigste Methode, denn diese erzeugt das Dreiecksnetz welches als Grundlage für weitere Verformungen dient.

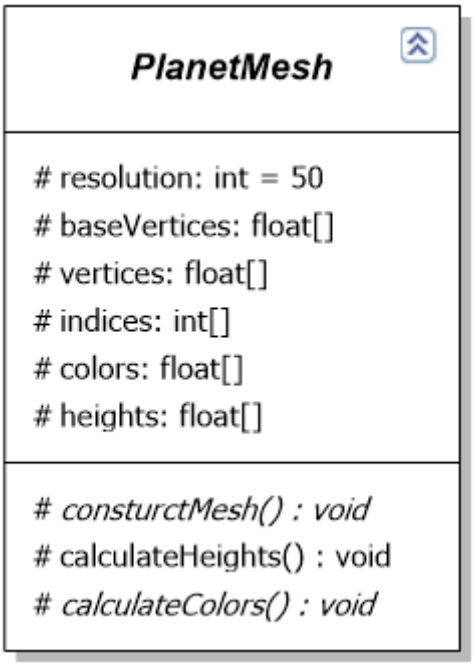
~ Patryk Watola

7.3 Erzeugung des Dreiecksnetzes eines Planeten (am Beispiel eines erdähnlichen Planeten)



Dreiecksnetz eines erdähnlichen Planeten ohne PlanetParts und PlanetEntities

Um eines dieser Dreiecksnetze zu erzeugen werden mehrerer Schritte benötigt. Dazu wird im Folgenden die PlanetMesh klasse genauer beschrieben.



Die Auflösung, also die Anzahl an Vertices, die das Netz besitzen soll, wird durch *resolution* bestimmt. Die Arrays, die danach aufgelistet sind, speichern die Attribute des Netzes, um sie später auf die GPU zu laden. Heights speichert dabei die Höhenwerte der Vertices, um später

einen hohen Vertex zum Beispiel grau zu färben, sollte er besonders „hoch“ sein (z. B. ein Berg).

Drei Schritte werden benötigt, um ein fertiges Dreiecksnetz zu erzeugen:

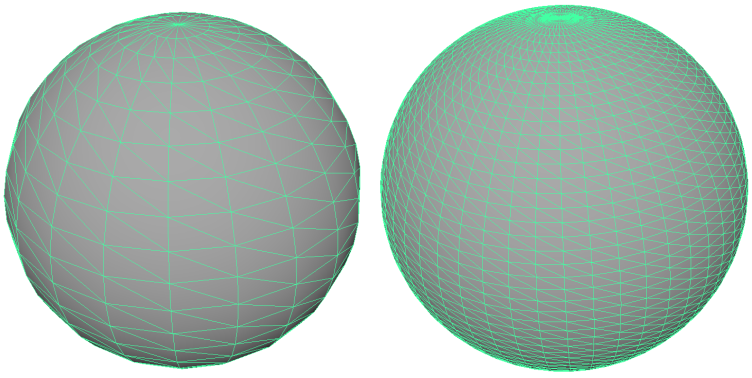
- Als erstes wird ein Basis Dreiecksnetz erzeugt, welches dann nach Belieben verformt werden kann. Die Methode *consturctMesh()* ist dafür zuständig und befüllt *baseVertices[]* und *indices[]*.
- Danach wird das Netz verformt, um ein erdähnliches Aussehen zu bekommen. Die Methode *calculateHeights()* ist dafür zuständig und basierend auf den *baseVertices* die in Schritt 1 errechnet wurden befüllt die Methode die Arrays *vertices[]* und *heights[]*.
- Der letzte Schritt gibt dem Planeten seine Farbe. *CalculateColors()* errechnet, basierend auf der Höhe eines Vertex, seine Farbe. Ein Vertex mit einem sehr niedrigen Wert im *heights[]* Array wird dann z.B. blau eingefärbt, um Wasser darzustellen. Aus den errechneten Vertices, Indices und Farbwerten wird dann ein Mesh Objekt erstellt, welches dann die Daten an die GPU übermittelt.

Diese 3 Schritte werden nun etwas näher beleuchtet.

~ Benjamin Jäger

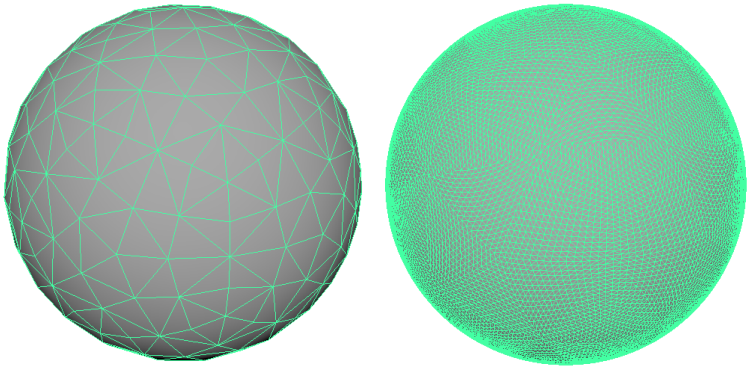
7.3.1 Erzeugung eines Basis Dreiecksnetzes

Um eine viel zahl an unterschiedlichen Dreiecksnetzen zu erzeugen, die einem Planeten ähneln, spielt die Auflösung des Netzes eine wesentliche Rolle. Auch bei einem angestrebten low-poly Look ist die Anzahl der einzelnen Dreiecke sehr wichtig. Gesucht ist also eine Kugel, bei der die Auflösung gesteuert werden kann. Begonnen wird jedoch nicht mit einer normalen Kugel als Grundform, sondern mit einem Würfel. Ein Würfel als Grundform bietet einige Vorteile, denn damit ist es später einfacher viereckige, aber erdähnliche Planeten zu erzeugen. Auch die Generierung von Indices und Vertices ist damit wesentlich leichter. Vor allem wenn ein Low-Poly-Look angestrebt wird, auf den später eingegangen wird.



Kugel mit der Auflösung 20,20 (links) und eine Kugel mit der Auflösung 50,50 (rechts)

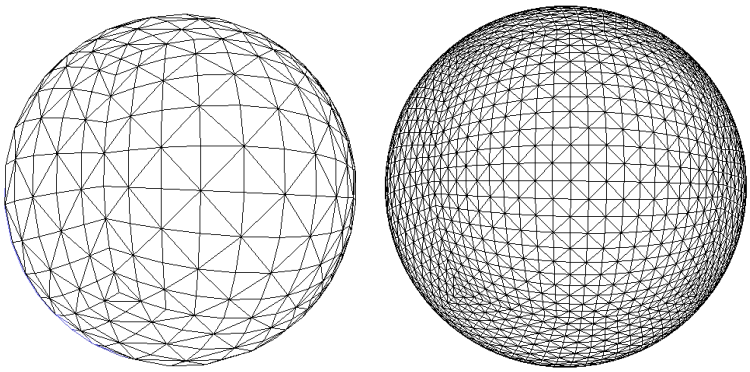
Erhöht man die Auflösung bei einer normalen Kugel werden die Dreiecke oben und unten immer kleiner, weshalb sie ungeeignet ist, um eine hohe Anzahl an Vertices darzustellen. Der obere und untere Teil des Planeten würde dann sehr verzerrt wirken.



Ikosaeder mit der Unterteilung 2 (links) und mit der Unterteilung 5 (rechts)

Wird ein Ikosaeder benutzt, so wird es eine Kugel, bei der alle Vertices ungefähr gleich verteilt und gleich groß sind. Wird jedes Dreieck des Ikosaeders in 3 weitere Dreiecke unterteilt, so entsteht eine Kugel mit höherer Auflösung. Jedoch steigt die Anzahl der Vertices exponentiell, weshalb dieser Ansatz auch nicht geeignet ist, um viele unterschiedliche Auflösungen anzuzeigen.

Der Finale Ansatz ist also ein Würfel, wobei jede Seite aus n*n Vertices besteht. Wird der Abstand aller Vertices zum Ursprung auf 1 gesetzt, bzw. werden alle Punkte des Würfels normiert, erhält man eine Kugel, bei der alle Dreiecke ungefähr gleich verteilt sind. Die Einstellung der Auflösung der Kugel ist nun auch im größeren Umfang möglich.



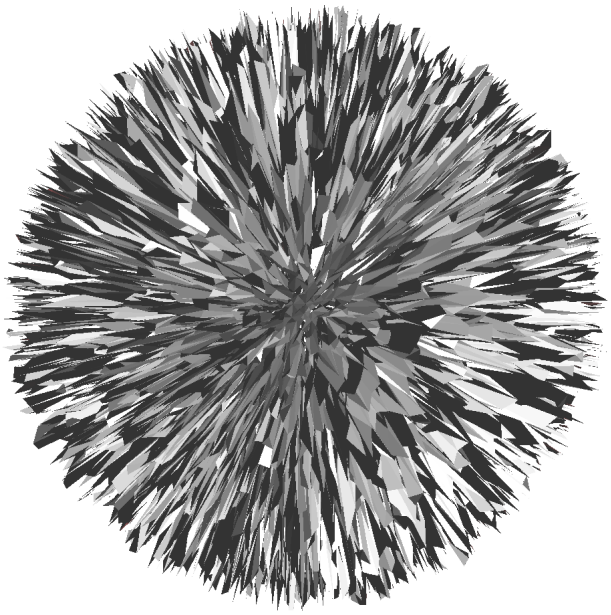
“aufgeblasener Würfel” mit Auflösung 10 (links) und Auflösung 25 (rechts)

Für fast alle Planeten ist dieses Dreiecksnetz die Grundlage, um verformt zu werden. Spannend ist dieser Planet jedoch noch nicht, am Ende sollen die Vertices nicht alle den Abstand 1 zum Ursprung haben. Einige Vertices sollen einen größeren Abstand haben, um Berge zu bilden und einige sollen einen geringeren Abstand haben, um Meere zu bilden.

~ Paul Amelingmeyer

7.3.2 Verformung des Planeten

Nun wird der zweite Schritt betrachtet, um ein erdähnliches Dreiecksnetz zu erzeugen: Die *calculateHeights()* Methode. Gesucht wird also für jeden Vertex ein Multiplikator, der bestimmt, wie weit er sich vom Ursprung entfernen soll. Eine Möglichkeit wäre immer zufällige Multiplikatoren zu benutzen, jedoch sieht das Ergebnis weniger ansprechend aus.

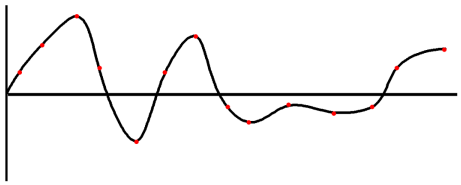


Jeder Vertex des Netzes wurde mit einer zufälligen Zahl in [0,1] multipliziert

Die Multiplikatoren der Vertices sollten ihren Wert nur minimal und kontinuierlich ändern, ohne dabei große Sprünge zu machen. Das klingt stark nach einer stetigen Funktion, die jedem Vertex einen Multiplikator zuordnet - also eine F(x,y,z).

Gewollt sind zufällige Werte von dieser Funktion und eben keine Funktion, die sich wiederholt (keine erkennbare Periodizität). Ebenfalls benötigt wird ein fest definierter Bereich, den diese Funktion zurückgibt, denn zu große oder zu kleine Zahlen führen zu Ausreißern. Darüber sind sich jedoch bereits anderweitig Gedanken gemacht und entsprechende Funktionen entwickelt worden. Noise-Funktionen werden sie genannt und werden oft in der Bildsynthese genutzt, um z.B. realistische Texturen für Holz oder Wolken zu erstellen.

Ein Beispiel einer Noise-Funktion wäre der “Value Noise”, welcher zunächst jedem X-Achsenabschnitt einen zufälligen Y-Wert zuweist (normalerweise im Bereich [0,1]) und diese Punkte interpoliert z.B. durch einen Kubischen Spline.

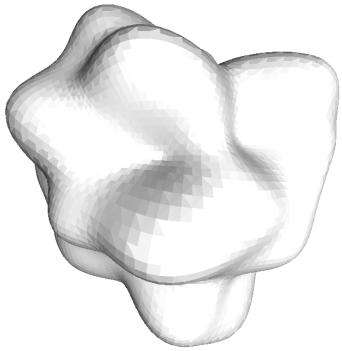


Value Noise in 1D

Benötigt wird eine dreidimensionale Funktion und keine eindimensionale. Die gewählten Y-Werte können hier zu zufällig sein und keine realistische Erdoberfläche simulieren, daher findet eine solide Open-Source-Java-Implementierung des Simplex-Noise Verwendung. Diese Funktion funktioniert in n-Dimensionen und ist Nachfolger des Perlin-Noise, eine sehr bekannte Noise-Funktion, die schon oft professionell angewendet worden ist, um realistische Bildsynthese zu betreiben.

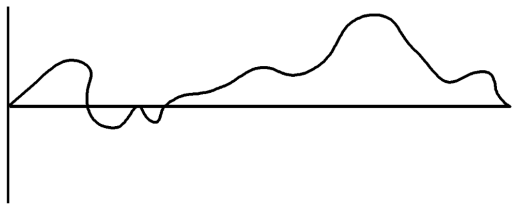
Die 3D-Simplex-Noise-Funktion nimmt drei Float-Werte an und gibt einen Wert zwischen [-1,1] zurück. Genommen wird also ein Punkt des Grund Dreiecksnetzes, welches zuvor

kalkuliert worden ist, und multiplizieren es mit dem Wert der Noise-Funktion der x,y,z Werte. Dies liefert zunächst folgende Ergebnisse.



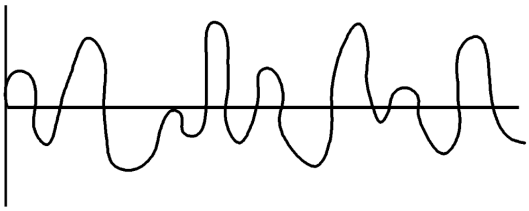
Grundnetz eines Planeten multipliziert mit Simplex-Noise-Werten

Dieses Ergebnis ist schon mal wesentlich besser als eine normale Kugel als Planeten. Hiermit können auch immer wieder andere Netze erzeugt werden, die nicht identisch sind. Erdähnlich ist das Ergebnis aber immer noch nicht, es ähnelt eher einem Kaugummi. Dieser Planet ist viel zu abgerundet. Es fehlen Details wie kleinere Hügel und das Gesamtbild sollte etwas kantiger sein. Um mehr Details zu bekommen, wird die Funktion gestaucht. Dies bedeutet im Endeffekt, dass die Frequenz, also die Anzahl der Schwingungen, erhöht wird. Die gestauchte Funktion wird dann auf die Ursprungsfunktion aufaddiert, um die finale Noise-Funktion zu erhalten.



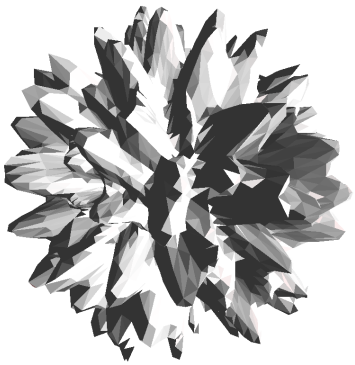
Erhöhung der Frequenz

Diese Funktion hat schon etwas mehr Detail, sieht aber immer noch nicht erdähnlich genug aus.



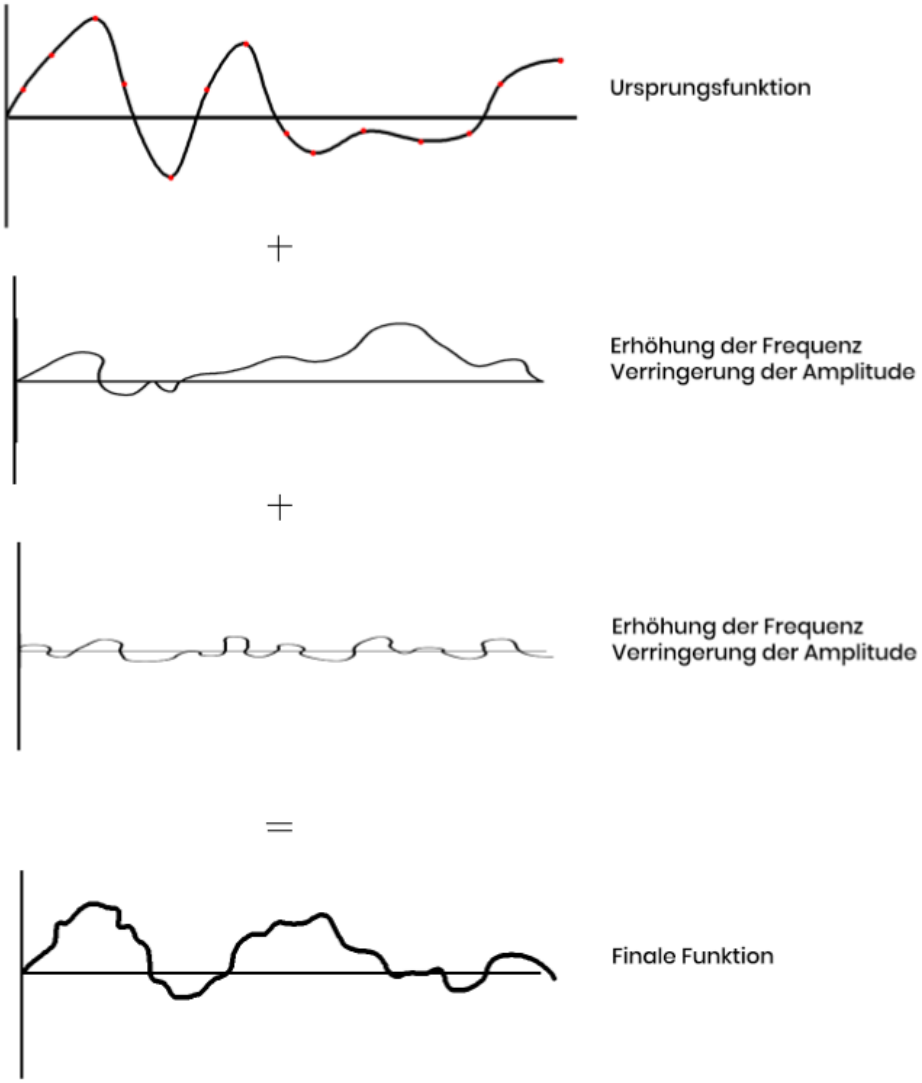
Weitere Erhöhung der Frequenz

Aus weiterer Erhöhung der Frequenz folgt zwar einerseits mehr Detail, jedoch geht die Ursprungsform der Planeten verloren.



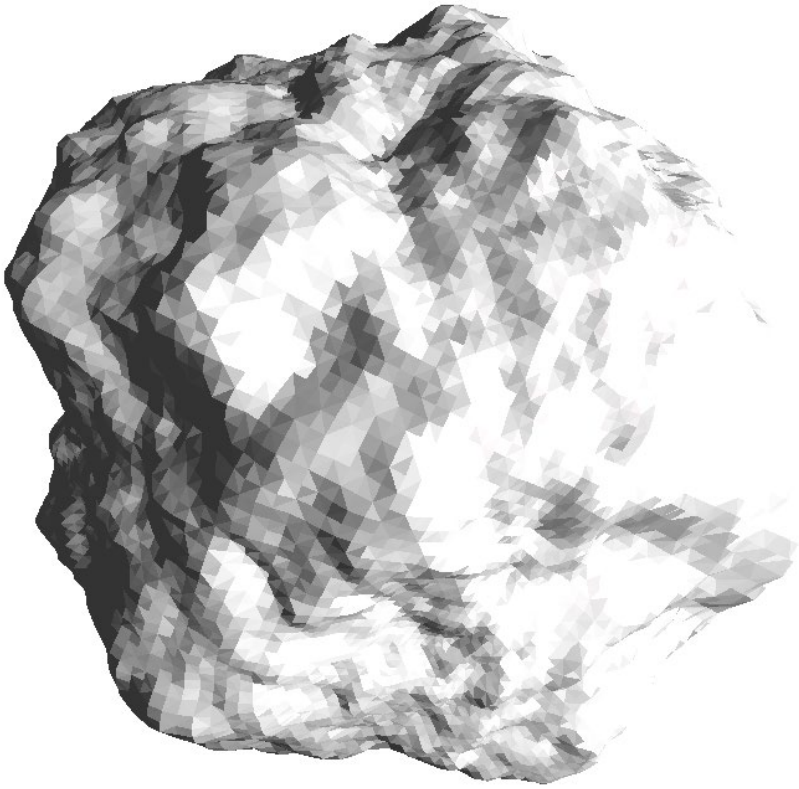
Verlust der Ursprungsform durch hohe Frequenz der Noise-Funktion

Benötigt werden mehr Details, jedoch sollte die Ursprungsform dabei erhalten bleiben. Um dies zu schaffen, muss der Einflussbereich jeder weiteren Funktion, die auf die Ursprungsfunktion aufaddiert wird, verringert werden. Das heißt, dass die Amplitude der nachfolgenden Funktionen verringert werden sollte.



Ablauf, um die finale Noise-Funktion zu berechnen.

Die finale Noise-Funktion hat genug Detail und behält dabei auch ihre ursprüngliche Form bei.



Finale Noise-Funktion am Planeten angewendet

```
public float calculateNoiseValue(float x,float y,float z) {
    float noiseValue=0;
    float frequency=noiseBaseRoughness;
    float amplitude=1;

    for (int i = 0; i <noiseLayers; i++) {
        float noiseX= x*frequency+noiseOffsetX;
        float noiseY= y*frequency+noiseOffsetY;
        float noiseZ= z*frequency+noiseOffsetZ;
        noiseValue+=(float)(SimplexNoise.noise(noiseX,noiseY,noiseZ)+1)*0.5f*amplitude;
        frequency*=noiseRoughness;
        amplitude*=noisePeristance;
    }

    noiseValue=Math.max(0, noiseValue-noiseMinValue);
    return noiseValue*noiseStrength;
}
```

Funktion um Noise-Wert für x,y,z keines Punktes zu errechnen
In Anlehnung an die Funktion aus dem Artikel:
https://flafla2.github.io/2014/08/09/perlinnoise.html

Die obige Funktion kalkuliert für einen Punkt den entsprechenden Noise-Wert.
Hier gibt es einige Möglichkeiten die Funktion zu bearbeiten, um mehr oder weniger Detail zu erhalten. Der Wert *noiseValue,* der zu Beginn initialisiert wird, ist der finale Wert nach sämtlicher Bearbeitung, also der Multiplikator, der den Punkt letztendlich an seine korrekte Stelle setzt. Die Werte *frequency* und *amplitude* sind die bereits erwähnten Werte, um den

Planeten mehr oder weniger Detail zu geben. Der Wert *noiseBaseRoughness* ist dabei der Startwert der Frequenz der Funktion. Dieser kann bereits zu Beginn sehr hoch eingestellt werden, um viel Details zu erhalten, was jedoch in sehr kantigem Terrain enden kann. Die Anzahl der Durchläufe der folgenden Schleife wird durch den Wert *noiseLayers* kontrolliert. Die Schleife addiert die einzelnen Noise-Werte auf, um den finalen Noise-Wert zu kalkulieren. Während der Schleife wird dann wie bereits erklärt die Frequenz der Funktion erhöht, aber dabei ihre Amplitude verringert und dann auf den Wert der zuvor errechneten Funktion aufaddiert. Der Wert *noiseLayers* kontrolliert also wie viele Funktionen aufaddiert werden.
Die aufaddierten Funktionen werden dann Oktaven genannt.

```
float noiseX= x*frequency+noiseOffsetX;
float noiseY= y*frequency+noiseOffsetY;
float noiseZ= z*frequency+noiseOffsetZ;
noiseValue+=(float)(SimplexNoise.noise(noiseX,noiseY,noiseZ)+1)*0.5f*amplitude;
frequency*=noiseRoughness;
amplitude*=noisePeristance;
```

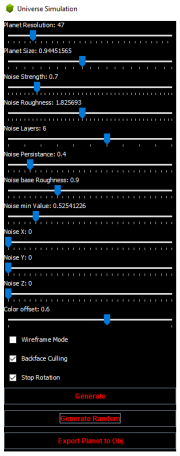
Das Schleifeninnere der Noise-Funktion

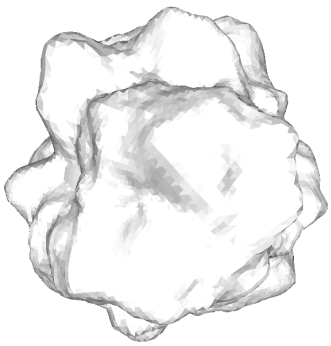
Innerhalb der Schleife werden zunächst die neuen x,y,z-Komponenten berechnet. Dazu werden alle Komponenten des Punktes des Planeten zunächst mit dem Frequenzwert multipliziert. Das heißt, die Funktion wird um den *frequency* Wert gestaucht. Danach wird zu jeder Komponente ein Offset addiert, welches genutzt werden kann, um bei gleichbleibender Frequenz und Amplitude andere Werte zu generieren. Danach werden die neu errechneten x,y,z Werte des Punktes genutzt und dem Simplex-Noise-Algorithmus übergeben. Dieser gibt einen Wert im Intervall [-1,1] zurück. Dieser Wert wird dann auf das Intervall [0,1] umgerechnet, um ihn später besser weiterverwenden zu können (z.B. für das Einfärben eines Planeten). Der resultierende Wert wird mit der Amplitude der Funktion multipliziert, um den finalen Noise-Wert dieser Funktion zu erhalten. Der Noise Wert wird mit dem errechneten Noise-Wert der Funktion aufaddiert, um nach Ende der Schleife einen Wert bestehend aus n-Oktaven zu erhalten. Dabei gilt: n = *noiseLayers*. Zuletzt wird die Frequenz um den *noiseRoughness*-Faktor erhöht und die Amplitude der nächsten Funktion wird um den Faktor *noisePeristance* verringert.

```
noiseValue= Math.max(noiseValue, noiseMinValue);
return noiseValue*noiseStrength;
```

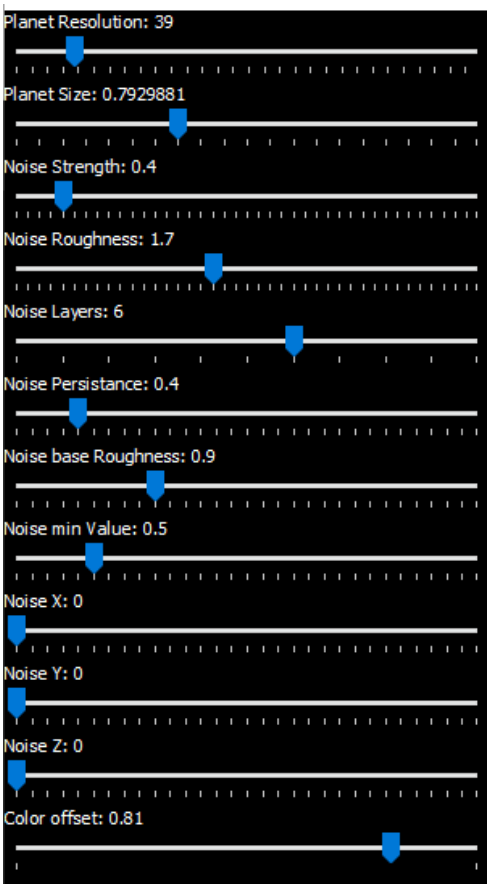
Ende der Noise Funktion

Zum Schluss wird noch ein minimaler Wert (*noiseMinValue*) eingeführt, der nicht vom finalen Noise-Wert unterschritten werden darf. Vor Rückgabe des Noise-Wertes wird der Wert noch einmal mit *noiseStrengh* multipliziert. Der Wert *noiseStrengh* verstärkt jeden Noise-Wert, um zum Beispiel Berge zu erhöhen und Meere zu vertiefen. Die erwähnten Werte *noiseBaseRoughness, noiseLayers, noiseOffsetX, noiseOffsetY, noiseOffsetZ, noiseRoughness, noisePeristance, noiseMinValue* und *noiseStrength* der Noise-Funktion können nach Belieben eingestellt werden, um beliebige Planeten zu erstellen.





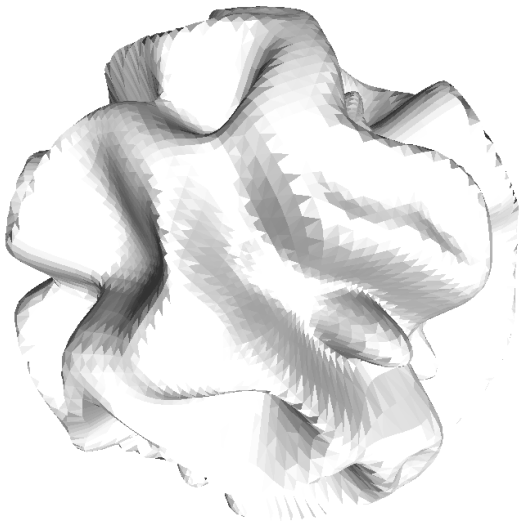
Ausschnitt aus dem Planet-Generator



Nahaufnahme der Schieberegler für den Planet-Generator

Die eben erwähnten Werte können im Planet-Generator innerhalb des Programms mittels Schiebereglern eingestellt werden, um einen individuellen Planeten zu erzeugen.

Die *calculateNoiseValue()*-Methode kann natürlich weiter angepasst werden, um andere Ergebnisse zu erzielen.



Abgeänderte calculateNoiseValue()-Methode

Bei diesem Beispiel wird nach der Berechnung des Noise-Wertes noch eine Cosinus-Funktion genutzt, um einen sehr welligen Planeten zu bekommen.

Die *calculateHeights()*-Methode macht genau das, denn sie errechnet für jeden Vertex des Basisnetzes einen Noise-Wert, multipliziert ihn mit dem Vertex und speichert die neu erzeugten Vertices in einem Array. Zudem werden alle Noise-Werte in einem zusätzlichen Array gespeichert, dem *heights[]* Array, welches zur Einfärbung der Vertices genutzt wird.

```
protected void calculateHeights() {
    heights=new float[baseVertices.length];
    vertices=new float[baseVertices.length];
    for (int j = 0; j < baseVertices.length; j+=3) {
        float noiseValue=noise.calculateNoiseValue(baseVertices[j],baseVertices[j+1],baseVertices[j+2]);
        vertices[j]=baseVertices[j]*(noiseValue+1);
        vertices[j+1]=baseVertices[j+1]*(noiseValue+1);
        vertices[j+2]=baseVertices[j+2]*(noiseValue+1);
        heights[j]=(noiseValue+1);
        heights[j+1]=(noiseValue+1);
        heights[j+2]=(noiseValue+1);
    }
    calculateColors();
}
```

Die calculateHeights()-Methode

Zu beachten ist, dass jeder Noise-Wert noch um eins erhöht wird, damit niemals ein Noise-Wert von 0 entsteht.

~ Simon Weck

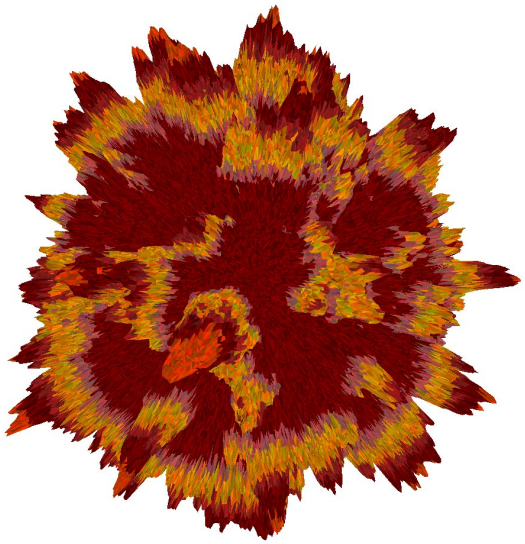
7.3.3 Farbgebung

Die *calculateHeights()* Methode ruft im Anschluss die *calculateColors()* Methode auf, welche die letzte Methode ist, um ein erdähnliches Dreiecksnetz zu konstruieren. Diese Methode speichert in dem *colors[]* Array eine RGB Farbe für jeden Vertex. Die Methode ist recht simpel, es läuft eine Schleife über alle vorhandenen Vertices und überprüft die Höhe dieses Vertex im *heights[]* Array. Anhand der Höhe wird dann eine Farbe im colors[] Array gespeichert, also zum Beispiel grün für ein Vertex mit einer Höhe von 1,5 oder grau für eine Höhe von 2,1.

```
protected void calculateColors() {
    colors=new float[heights.length];
    for (int i = 0; i < heights.length; i+=6) {
        //grey
        if (heights[i]<3) {
            colors[i]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
            colors[i+1]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
            colors[i+2]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
            colors[i+3]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
            colors[i+4]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
            colors[i+5]=0.82f*(colorOffset+((float)Math.random()*(1f-colorOffset)));
        }
```

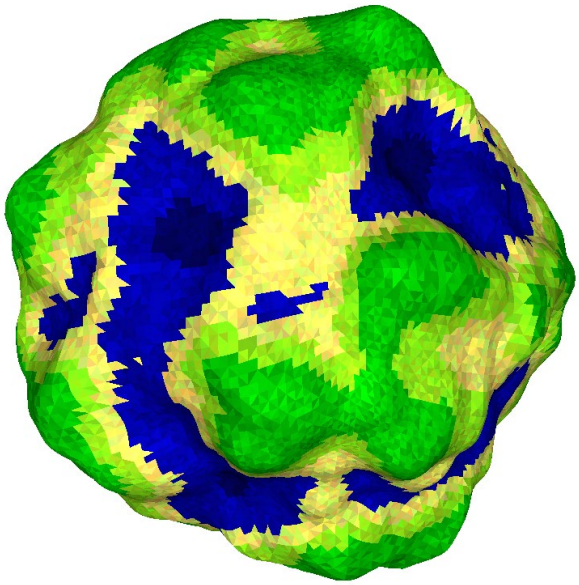
Ausschnitt der calculateColors() Methode

Hier ein Ausschnitt der *calculateColors()* Methode, welche für alle Vertices mit einer Höhe von bis zu 3 eine graue Farbe in das *colors[]* Array speichert. Zu erwähnen ist noch, dass die Farben immer etwas vom originalen Grauton abweichen, damit jedes Dreieck etwas anders eingefärbt ist. Der Wert *colorOffset* bestimmt dabei wie sehr die Farbe von der originalen Farbe abweichen soll. *colorOffset* liegt zwischen [0,1] wobei es bei 1 keine Abweichung zur originalen Farbe geben wird und bei 0 weicht die Farbe sehr stark von der originalen Farbe ab. Die Farbwerte sind frei wählbar, also könnte man für andere Planeten andere Werte anhand ihrer Höhe wählen.



Färbung eines Sonnen Planeten

Das Endergebnis ist ein erdähnliches Dreiecksnetz mit Vertices, Indices, Normalen und Farbwerten welches auf die GPU geladen werden kann, um es zu rendern.



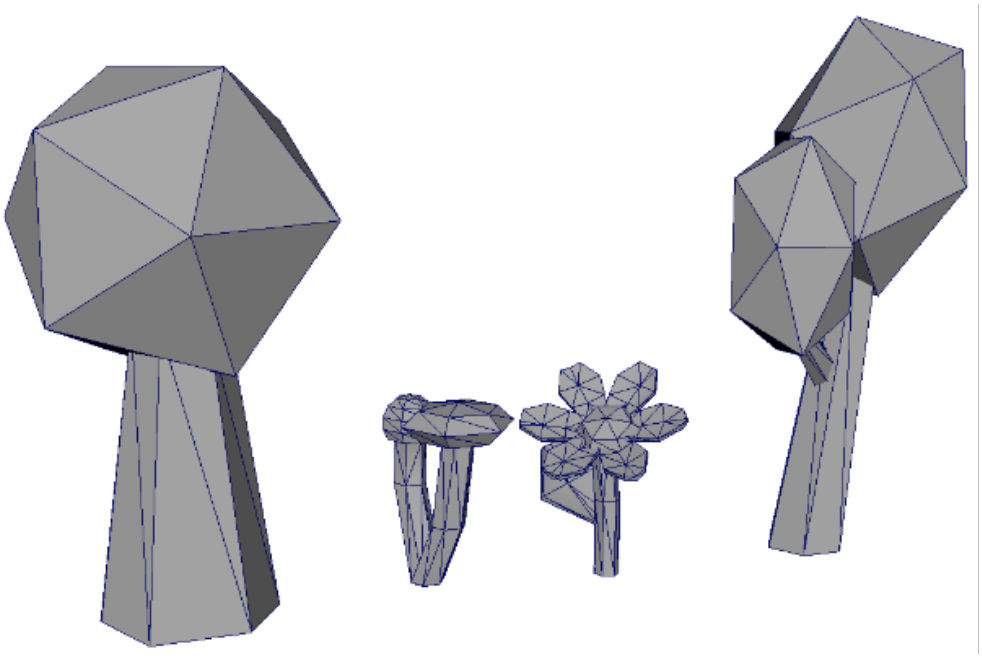
Erdähnliches Dreiecksnetz

~Paul Amelingmeyer

7.4 Einen Planeten besiedeln

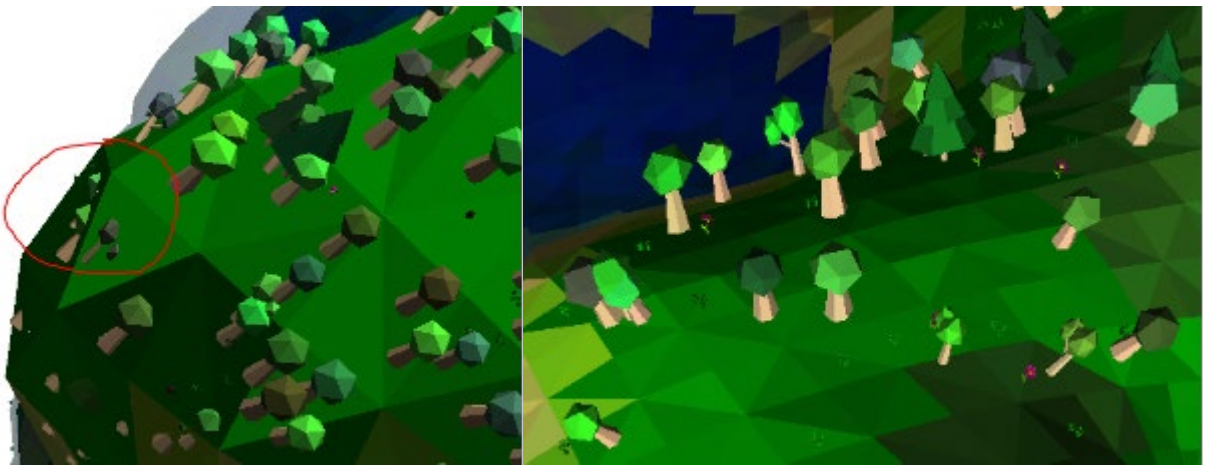
Wie bereits zu Beginn erwähnt besteht ein Planet nicht nur aus einem einzelnen Dreiecksnetz, sondern auch PlanetEntities und PlanetParts. Diese vervollständigen ihn und sorgen für den Feinschliff, um einen Planeten realistischer wirken zu lassen.

7.4.1 PlanetEntities

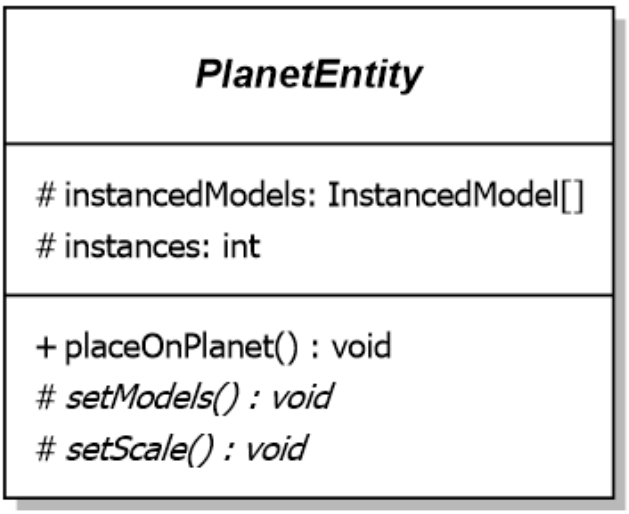


Beispiel für PlanetEntities Dreiecksnetze (Baum,Pilz,Blume,Baum). Erstellt in Maya

PlanetEntities sind Objekte die unmittelbar auf einen Planeten gesetzt werden. Bei einem erdähnlichen Planeten wären das zum Beispiel Bäume und Blumen. Sämtliche PlanetEntities sind in Maya modelliert worden und als .obj Datei importiert. PlanetEntities müssen direkt auf ein zufälliges Dreieck des Dreiecksnetzes des Planeten platziert werden. Zudem wird gefordert, dass diese Objekte richtig an dem Dreiecksnetz ausgerichtet sind, also entlang des normalen Vektors des Dreieckes auf dem das Objekt platziert wird. Zudem sollen die Objekte nur auf Dreiecke mit einer bestimmten Höhe platziert werden. Bäume sollen somit zum Beispiel auf Grass platziert werden und nicht auf Wasser.

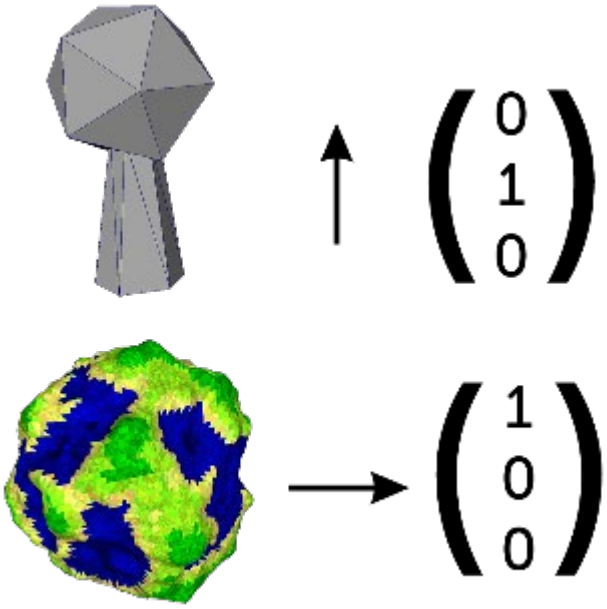


Falsch ausgerichtete Bäume (links) und richtig ausgerichtete Bäume (rechts)

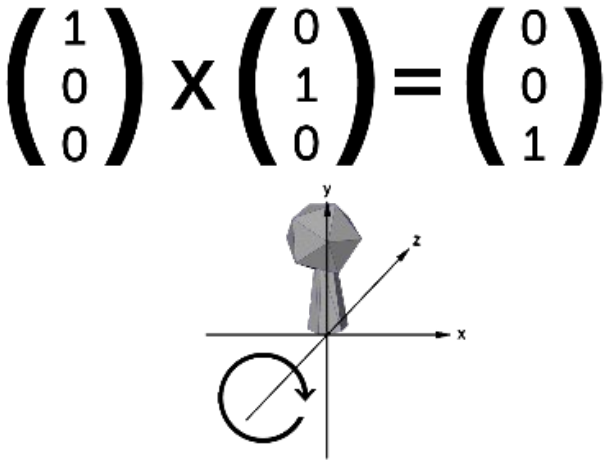


PlanetEntity Klasse

Die *placeOnPlanet()* Methode macht genau das. Sie sucht sich ein zufälliges Dreieck vom Netz des Planeten aus und speichert die korrekte Platzierung und Rotation um entlang des normalen Vektors ausgerichtet zu sein.

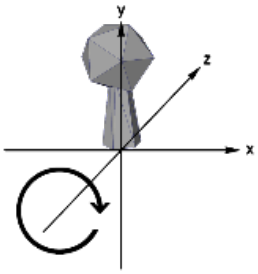


Abgenommen es soll ein Baum rechts außen auf den Planeten gesetzt werden. Um ihn richtig auf der Fläche auszurichten, wird zunächst der Vektor in Richtung des Baums benötigt. Dies ist für alle PlanetEntities der Vektor (0,1,0), da alle Objekte so modelliert wurden, dass sie in Richtung der positiven Y-Achse zeigen. Dazu wird noch der normalen Vektor des Dreieckes, auf das der Baum platzieren werden soll, benötigt. Daraus lässt sich eine Rotationsachse und ein Rotationswinkel berechnen, welche diesen Baum korrekt ausrichten.



Um die Rotationsachse zu bekommen wird das Kreuzprodukt des Normalen Vektors(1,0,0) und des Richtungsvektors des Baumes (0,1,0) berechnet. Bei diesem Beispiel erhält man den Vektor (0,0,1), welcher der positiven Z-Achse entspricht. Das ergibt auch Sinn, denn der Baum muss um -90° um die positive z-Achse gedreht werden um ihn richtig zu positionieren. Jetzt fehlt nur noch der Winkel:

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \bullet \begin{pmatrix}0\\1\\0\end{pmatrix} = 0$$



-Acos(0) = -90°

Um den Winkel zu erhaltenwird das Skalarprodukt zwischen dem Normalen Vektor(1,0,0) mit dem Richtungsvektor des Baumes (0,1,0) berechnet. Dieser liefert null. Wird der negativen Arkuskosinus vom errechneten Skalarprodukt berechnet erhälten man den gesuchten Winkel. Diese Berechnung funktioniert für jedes Dreiecksnetz um Objekte daran auszurichten. Diese Transformation um die errechnete Achse mit dem entsprechendem Winkel muss nur noch in einer model Matrix gespeichert werden, damit die Transformation im entsprechenden Shader stattfinden kann.

7.5 Instanziiertes rendering

Ein Planet besitzt normalerweise nicht nur einen Baum sondern eine menge davon. Man könnte ein Baum Objekt zum Beispiel 900 mal erzeugen und es jedes mal einzeln auf die GPU laden. Dies wäre jedoch nicht effizient, da es zum einem immer das selbe Dreiecksnetz ist, welches auf die GPU geladen wird, und der OpenGL draw call sorg für einen gewissen Overhead, da die GPU einige Vorbereitungen treffen muss um ein Objekt durch die Pipeline zu schicken. Viel einfacher wäre es das Dreiecksnetz einmal auf die GPU zu schicken und es dann immer wieder mit anderen Transformationen zu rendern. Das ist instanziiertes rendering. Für diese Zwecke werden weitere VBOs genutzt, die einzelne Model Matrizen speichern. Diese VBOs verhalten sich jedoch nicht wie normale VBOs, denn sie ändern ihren Wert nur pro Instanz. Das heißt eine riesige Liste an Model Matritzen wird in ein VBO geladen und benutzt jeweils eine Matrix für jede Instanz aber immer wieder andere Vertices.

Diese Liste an Matrizen wird dann jeden Frame aktualisiert um die Objekte zu bewegen.

	Vertex Array Object (VAO)
0	Vertices (x0,y0,z0,x1,y1,z1.....xn,yn,zn)
1	Normals (x0,y0,z0,x1,y1,z1.....xn,yn,zn)
2	Colors (r0,g0,b0,r1,g1,b1.....rn,gn,bn)
3	model Matrix(m0, m1, ... mn)

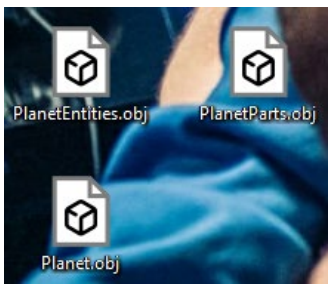
← Vertex Buffer Object (VBO)

In diesem Beispiel würden die Matrix m0 solange benutzt bis jeder Vertex einmal die Pipeline durchlaufen hat. Danach würden die nächste Matrix (m1) benutzet und damit wieder alle Vertices durch die Pipeline geschicket. Mit dieser Strategie ist es sehr einfach eine enorme menge an Bäumen auf einen Planeten zu platzieren ohne dabei Performance zu verlieren. Zu beachten ist jedoch, dass für jede Instanz auch eine Matrix errechnet werden muss, was die CPU sehr belasten kann. Eine Verbesserung wäre es diese auch auf der GPU zu berechnen, also die Transformationsdaten auf die GPU laden um dort die Matrix zu berechnen. Dessweiteren ist noch zu erwähnen, dass die maximale Länge an Daten, die auf die GPU geladen werden darf nur maximal 4 betragen darf, daher muss die Model Matrix auch auf 4 VBOs verteilt werden, damit eine 4x4 Matrix daraus gebildet werden kann.

~ Benjamin Jäger

7.6 Einen Planeten als .obj Datei exportieren

Objekte können mittels OBJ Loader bereits in das Projekt eingebunden werden aber umgekehrt ist dies auch möglich. Das Programm bietet die Möglichkeit Planeten und alle seine Bestandteile als OBJ Datei zu exportieren. Der Planet wird nicht als einzelne .obj Datei exportiert, sondern als 3 einzelne Dateien für Die PlanetEntities, PlanetParts und den Planeten selbst.



Aufteilung der Planeten Bestandteile in 3 einzelne Dateien

Eine .obj Datei kann Informationen über Vertices, Normalen sowie Texturkoordinaten speichern. Texturkoordinaten kommen nicht auf einem Planeten vor, weshalb sie hier wegfallen. Eine .obj Datei speichert aber auch welche Punkte wie miteinander verbunden werden sowie welche Normalen und Texturkoordinaten zu diesen Punkten gehören. Vertices

und Normalen werden bereits in der Mesh Klasse gespeichert, daher müssen diese lediglich in korrekter Formatierung in eine obj. Datei geschrieben. PlanetEntities und PlanetParts werden außerdem gruppiert, um sie später einzeln weiterverarbeiten zu können.

```
g Entity1Model1
v -1.1442457 -0.030231256 1.0141172
v -1.1440294 -0.022297978 1.0183998
v -1.1459295 -0.030396514 1.01450844
v -1.1457354 -0.024427006 1.0165821
v -1.1464796 -0.021269988 1.014713
v -1.1649879 -0.023141028 1.0151727
v -1.1650778 -0.026434246 1.013395
v -1.1643336 -0.029591255 1.015264
v -1.1634996 -0.029455043 1.0189109
v -1.1634098 -0.026161825 1.0206888
v -1.1544738 -0.05244313 1.0536785
v -1.1535158 -0.017322287 1.0726382
v -1.1670324 -0.028404124 1.0097831
v -1.1660744 0.0067167208 1.0287428
v -1.1654383 -0.028247666 1.0094128
v -1.1644803 0.006873179 1.0283724
vn 0.010669638 -0.00128525 -0.0025348798
vn 0.0057640993 4.59566E-4 0.0010457358
vn -0.0047523505 -0.0023962038 0.0024905768
vn -0.0101637645 3.1693105E-4 0.0043030363
vn -0.010769387 -0.0023716104 5.607629E-4
vn -0.005714225 0.0013688642 -5.867728E-5
vn 0.0047024763 5.6777365E-4 -0.0034776353
vn 0.010263513 0.0033399293 -0.0023289192
vn 0.01066964 -0.0012852498 -0.0025348803
vn 0.0057641002 4.595659E-4 0.0010457357
vn -0.0047523514 -0.002396204 0.0024905773
vn -0.010163766 3.1693088E-4 0.0043030367
vn -0.010769389 -0.0023716104 5.607633E-4
vn -0.005714226 0.0013688642 -5.8677164E-5
vn 0.0047024773 5.677738E-4 -0.0034776358
vn 0.010263515 0.0033399293 -0.0023289197
vn 0.33973736 0.018327188 -0.080123745
vn 0.17238553 0.017097708 -0.025877984
vn -0.16655545 -0.02825734 0.046255544
vn -0.33682233 -0.023907002 0.09031252
vn -0.3403985 -0.04256514 0.06703914
```

Ausschnitt der gespeicherten Vertices und Normals eines gruppierten PlanetEntities

Die Buchstaben v sowie vn leiten ein, dass es sich in einem Vertex und Normal handeln. Der Buchstabe g teilt alle folgenden Daten in eine Gruppe ein. Hier ist es das Entity Nummer 1 und das Model 1 also z.B. der Stamm eines Baumes. Wichtig ist zu beachten, dass die

```
for (int i = 0; i < vertices.length; i+=3) {
    Vector4f vertexVector = model.getModelMatrix().multiply(new Vector4f(vertices[i],vertices[i+1], vertices[i+2],1));
    String vertex = "v "+vertexVector.x+" "+vertexVector.y+" "+vertexVector.z;
    obj.add(vertex);
}
```

C5

Vertices noch nicht an der richtigen stelle sind. Die Vertices müssen zunächst mit der korrekten Model Matrix multipliziert werden um an die richtige stelle zu gelangen.

Ausschnitt der Methode um die ein Model in eine .obj Datei schreibt

Hier ist gut zu erkennen, dass der Vertex zunächst mit der Model Matrix multipliziert wird und das Ergebnis dann als String in einer Liste gespeichert wird. Diese wird dann später in eine .obj Datei geschrieben.

Zum Schluss muss in einer .obj Datei noch angegeben werden, welcher Vertex mit welchem normal zusammenhängt, und mit welchen weiteren Vertices es ein Dreieck bildet.

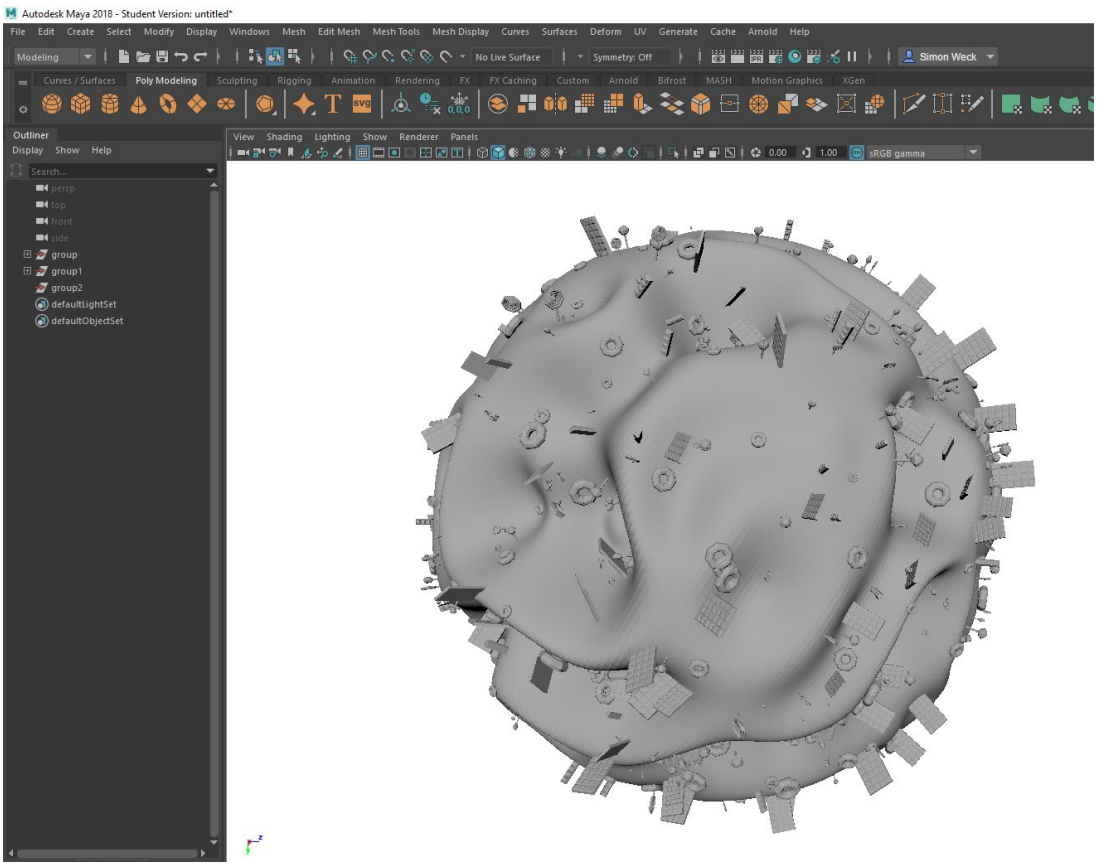
```
f 1//1 2//2 3//3
```

Das f, welches für „face“ steht, gibt als Index an, welcher Vertex mit welchem normal zusammenhängt. Der erste Eckpunkt besteht also aus dem Vertex 1 und dem Normal 1 aus den zuvor angegeben Daten. Dieser Eckpunkt ist verbunden mit dem 2. und 3. Vertex aus den Daten und bildet ein Dreieck. Um dies korrekt in eine .obj Datei zu speichern werden die Indices genutzt, die ebenfalls in der Mesh- Klasse jedes Objektes gespeichert sind.

```
for (int i = 0; i < indices.length; i+=3) {
    String face = "f "+(indices[i]+1+vertexCount)+"//"+(indices[i]+1+vertexCount)+
    obj.add(face);
}
```

Der erste Teil eines faces wird in einen String umgewandelt.

Hier werden die Indices des Models entnommen und in eine Liste geschrieben. Wichtig ist hier, dass eine .obj Datei nicht mit dem Index 0 beginnt, sondern mit 1. Also wird auf jeden Index 1 addiert. Auch zu beachten ist, dass die Indices um die aktuelle Anzahl der Vertices erhöht werden damit mehrere Modelle in eine Datei geschrieben werden können. Ohne das Problem, dass diese den falschen Index nutzen. Das Ergebnis sind 3 .obj Dateien, welche zum Beispiel in Maya weiterverarbeitet werden können, oder auch 3D gedruckt werden können.



Exportierter CandyPlanet in Maya importiert

Als „Future work" könnte man hier noch zu den einzelnen Modellen ihre Materialeigenschaften als .mtl Datei abspeichern. Die .mtl Dateien der einzelnen Modelle können dann in das .obj Datei Format eingebunden werden. Dadurch können auch Farben der einzelnen Objekte exportiert werden.

Weitere erwähnbare Features

Konvex Kombination von 2 Matrizen um sanft auf Planeten zu zoomen oder um die Kamera zu zentrieren.

Ray casting und entsprechende Kollisionserkennung von Linie und Kreis um Mouse Picking zu betreiben. Werden Planeten bei der Kollisionserkennung erkannt wird die Farbe der Planeten sowie der PlanetParts und PlanetEntities aktualisiert um den Nutzer zu benachrichtigen, das ein Planet auswählbar ist.

Größtenteils selbstgeschriebene Mathe Funktionen für Matrizen und Vektoren.

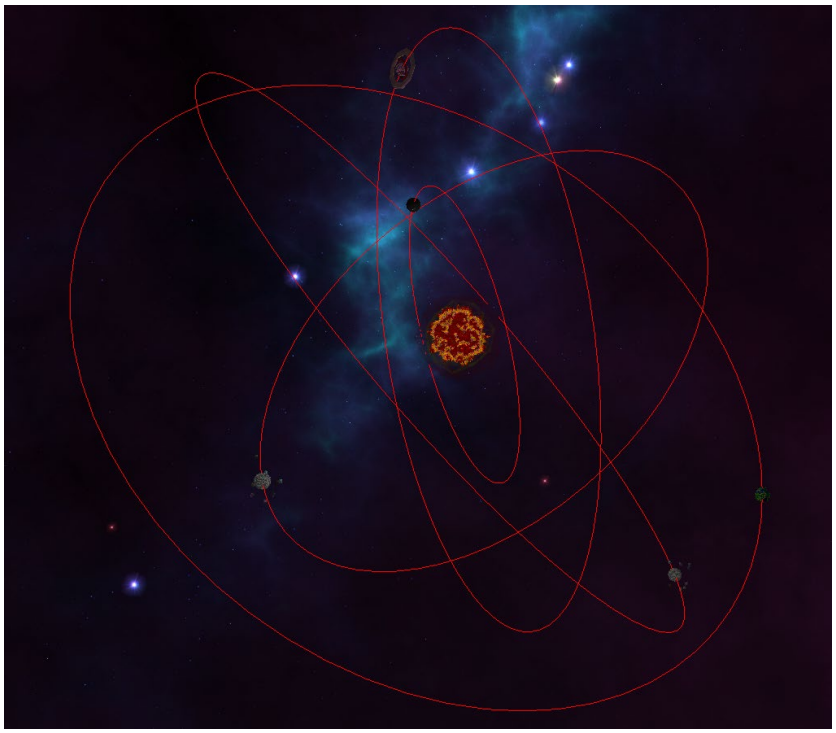
Dynamisches System um beliebig viele Punktlichter in eine Szene einzufügen.

Dreiecksnetze sowie Shader angepasst um Low-poly Look zu realisieren.

Skybox mit mehreren Texturen die zufällig ausgewählt werden.

Simulation von Wasser mit einem weiterem VBO für Multiplikatoren um Wellen zu simulieren.

Material System um Objekte wie Wasser transparent und glänzend wirken zu lassen.
Testklassen um Beleuchtung, den OBJ Loader und ähnliches zu testen.



Ergebnis einer Abfotografierten Zeichnung um ein Universum zu erstellen

~ Benjamin Jäger

8 Referenzen

8.1 Quellen

Inspiration für Matrix-Funktionen. Inverse Matrix Funktion aus lwjgl Matrix4f Klasse

https://github.com/LWJGL/lwjgl/blob/master/src/java/org/lwjgl/util/vector/Matrix4f.java

https://github.com/JOML-CI/JOML/blob/master/src/org/joml/Matrix4f.java

Punktlicht Attenuation-Werte

http://wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenuation

Simplex Noise algorithm (Bibliothek darf verwendet werden)

https://github.com/SRombauts/SimplexNoise/blob/master/references/SimplexNoise.java

Webcam Capture API (Bibliothek darf verwendet werden)

https://github.com/sarxos/webcam-capture

8.2 *Literatur*

Beiträge, um Noise zu verstehen

https://flafla2.github.io/2014/08/09/perlinnoise.html

http://media2mult.uos.de/pmwiki/fields/cg-II-09/index.php?n=ProceduralGraphicsI.Noise

Computergrafik Buch mit Java und Jogl

Teile des OBJ Loaders wurden dem Buch entnommen

Funktionen, um glsl Errors und OpenGL Errors zu finden

Gordon, V. Scott. Clevenger, John (2018). Computer graphics programming in OpenGL with Java. 2. Auflage. Dulles: Mercury Learning and Information.

Videoreihe, um Computergrafik und low-Poly zu verstehen

https://www.youtube.com/playlist?list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP

Website für Computergrafik

https://learnopengl.com/

https://opencv.org/

9 Anhang

9.1 Installation des Programms

Zur Installation des Proramms sollten folgende Schritte befolgt werden:

- Starten Sie “Intellij IDEA”
- Wählen Sie File > New > Project from existing sources ...
- wählen Sie den order “Universe Simulation”
- bestätigen Sie alles mit “OK”
- wählen Sie File > Project Structure und binden Sie Jogl 2.3.2 und OpenCv 411 ein
- binden Sie anschließend die Bibliotheken welche sich im Ordner „Libraries“ befinden ein