# Simulation von Protuberanzen und Erstellen von prozeduralen Kugeloberflächen in Unity

Florian Hansen
Hochschule Flensburg
Germany

## ZUSAMMENFASSUNG

test

## 1 EINLEITUNG

Häufig steht man als 3D-Programmierer vor dem Problem, die in der Natur vorkommenden Phänomene effizient und in Echtzeit darzustellen zu wollen, um diese dann anschließend in andere Projekte wie Videospiele einzubinden. Aber auch bei Filmen sollte darauf geachtet werden, dass die Zeit für das Rendern nicht ausartet. Am interessantesten erscheint in diesem Zusammenhand die Nutzung von Echtzeitsimulationen, um natürliche Prozesse näher untersuchen zu können, die nicht so häufig vorkommen.

Im Grunde existieren zwei Lösungsansätze für das Problem, rechenintensive Echtzeitsimulationen durchzuführen. Zum einen kann man fiktive Effekte definieren, die durch iteratives Ausprobieren ähnliche Ergebnisse wie das reale Vorbild erzielen. Diese Ergebnisse sind dann jedoch rein fiktiv und haben selten etwas mit der Realität zu tun. Zum anderen kann man versuchen, physikalische Prozesse abzubilden, um so möglichst nah an der Realität zu bleiben. Diese wissenschaftliche Arbeit soll sich diesem Problem anhand der uns bekannten Sonne widmen. Hierbei wird sich auf die äußerliche Erscheinung der Sonne und besonders ihrer Protuberanzen (umgangssprachlich Sonnenstürme) bezogen. Es soll also eine Verbindung zwischen echten, beobachtbaren Phänomenen bzw. physikalischen Eigenschaften der Sonne und Computersimulationen umgesetzt werden.

## 2 GRUNDLAGEN

### 2.1 GLSL-Shader

### 2.2 HLSL-Shader

### 2.3 CG-Shader

## 3 BERECHNUNG EINER KUGEL

Damit später eine Textur für die Sonne erstellt werden kann, muss zuerst eine Oberfläche generiert werden, die diese Textur aufnehmen kann. Hierfür existieren verschiedenste Möglichkeiten. Im Falle der Sonne betrachten wir das Modell einer Kugel, die später manipuliert werden soll, um visuelle Effekte auf ihr abzubilden. Beim Betrachten der folgenden Algorithmen zur Erzeugung von Kugeloberflächen wird klar, dass nicht alle Methoden für dieses Vorhaben geeignet sind. Aus diesem Grund werden die einzelnen Verfahren näher erläutert und anschließend abgewogen, welcher Algorithmus zur Erzeugung einer Sonnenoberfläche verwendet werden sollte.

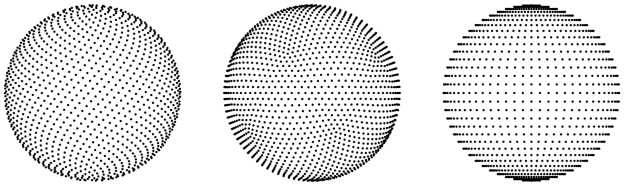


**Abbildung 1: Vergleich der verschiedenen Resultate der Algorithmen zum Erzeugen von Punkten auf einer Kugel. Links: Fibonacci-Gitter, Mitte: Icosphere, Rechts: Längen-Breitengrad-Gitter.**

### 3.1 Längengrad-Breitengrad-Gitter

Die einfachste Möglichkeit, eine Kugeloberfläche zu erzeugen, ist die Verwendung von Segmentlinien. Im Grunde unterteilt man die gesamte Kugel in Schichten, deren Umfang schließlich die Oberfläche der Kugel darstellen. Die Mesh-Punkte (*Vertices*) dieser Schichten, welche als einfache Kreise wahrgenommen werden können, werden mithilfe von Längen- und Breitengradinformationen erstellt. Das Resultat ist eine Kugeloberfläche mit erkennbaren Polen an der Ober- und Unterseite der Oberfläche. Diese entstehen dadurch, dass auch am Rand der Kugel die gleiche Anzahl an Kanten vorhanden ist, wie am Äquator. Diese Schichten sind dementsprechend lediglich herunterskaliert und in einem immer kürzer werdenden Abstand aufeinander geschichtet und Erzeugen eine Überabtastung am oberen und unteren Rand der Kugel [4].

Die Überabtastung an den Polen bzw. die Unterabtastung am Äquator bringt viele Probleme mit sich. Vor allem, wenn die Oberfläche animiert werden soll, erscheint die Animation als sehr unregelmäßig. Hierauf wird jedoch in einem späteren Abschnitt eingegangen, wenn die hier vorgestellten Methoden miteinander verglichen werden.

### 3.2 Fibonacci-Gitter

Um das Problem der Inhomogenität von Längengrad-Breitengrad-Gittern zu lösen, können Fibonacci-Gitter verwendet werden. Hierbei werden die Punkte des Gitters entlang einer Spirale, die sogenannte Fibonacci-Spirale, angeordnet. Wichtig bei der Konstruktion sind die *Golden-Ratio* $\Phi$ und der *Golden-Angle* $\alpha$ [4, 5].

$$\Phi = \frac{1+\sqrt{5}}{2} \qquad \alpha = 2\pi\,(2-\Phi)$$

Besonders einfach lässt sich bei diesem Algorithmus die Anzahl an Oberflächenpunkte kontrollieren. Das folgende Beispiel behandelt das Erstellen von $N$ Punkten, die gleichmäßig verteilt auf der Oberfläche einer Kugel liegen. Zuerst werden die Längen- und Breitengrade der einzelnen Punkte berechnet.

$$lat_i = arcsin\left(\frac{2i}{N+1} - 1\right) \qquad lon_i = \alpha \cdot i$$

Anschließend können die räumlichen Koordinaten als Positionsvektor $\vec{p} \in \mathbb{R}^3$ wie folgt berechnet werden.

$$\vec{p} = \begin{pmatrix} \cos(lon)\, cos(lat) \\ \sin(lon)\, sin(lat) \\ \sin(lat) \end{pmatrix}$$

Wie man sieht, ist die Erzeugung von homogen verteilten Oberflächenpunkten einer Kugel mithilfe von Fibonacci-Gittern ziemlich schnell implementiert. Nachteil dieser Lösung ist, dass lediglich die Punkte bzw. Vertices eines Meshes erzeugt werden. Um Flächen darstellen zu können, muss nachträglich eine Triangulation stattfinden.

## 3.3 Icosphere

Genau wie das Fibonacci-Gitter erzeugt ein Ikosaeder gleichmäßig verteilte Punkte auf der Oberfläche einer Kugel. Das Verfahren ist trotzdem komplett unterschiedlich. Anstatt die Vertices des Meshes zu berechnen und anschließend diese miteinander in Dreiecke zu verbinden, wird eine initiale Geometry erzeugt, dessen Flächen wiederum in weitere Flächen unterteilt wird. Dieses Vorgehen wird beliebige Male wiederholt, um eine immer glattere Oberfläche zu erzeugen. Das Problem bei diesem Algorithmus ist, dass durch die Unterteilung der Flächen in weitere, kleinere Flächen auf den ersten Blick nicht abgeschätzt werden kann, wie viele Vertices genau berechnet werden müssen. Tabelle 1 zeigt den Anstieg der benötigten Punkte, die durch $N$ wiederholten Unterteilungsschritten berechnet werden müssen [3]. Anhand der Anzahl an Dreiecken fällt ein exponentielles Wachstum auf, weshalb eine Erhöhung der Unterteilungsiterationen mit Bedacht durchgeführt werden sollte. Eine kleine Veränderung von $N$ hat große Auswirkung auf die benötigte Rechenleistung für das Kugelmesh [1].

**Tabelle 1: Vergleich zwischen Anzahl der Unterteilungsiterationen $N$ und den erzeugten neuen Flächen bzw. Vertices.**

| N | Anzahl Vertices | Anzahl Dreiecke |
|---|---|---|
| 1 | 12 | 60 |
| 2 | 42 | 240 |
| 3 | 162 | 960 |
| 4 | 642 | 3840 |
| 5 | 2562 | 15360 |

## 3.4 Wahl eines Algorithmus' für die Sonne

Da in den voherigen Abschnitten einige Algorithmen zum Erzeugen von Punkten auf einer Kugeloberfläche besprochen wurden, muss nun ein geeigneter Algorithmus ausgewählt werden, welcher zur Beschaffenheit der Sonne passt. Außerdem muss berücksichtigt werden, dass die Oberfläche in späteren Schritten manipuliert und animiert werden soll. Dabei ist es wichtig zu berücksichtigen, dass sich die Oberfläche der Sonne überall gleichmäßig verhalten soll. Aus diesem Grund kann das Längengrad-Breitengrad-Gitter nicht verwendet werden. Das Problem mit diesem Gitter ist, dass es eine Überabtastung an den Polen bzw. eine Unterabtastung am Äquator besitzt. Eine Oberflächenbewegung wäre daher sehr unregelmäßig und entspricht auch nicht dem originalen Vorbild.

Eine elegantere Lösung wäre die Verwendung des Fibonacci-Gitters. Die Anzahl der Punkte sind frei wählbar und werden homogen auf der Oberfläche verteilt. Auch ist die Berechnung der Positionen nicht komplex. Jedoch stößt man auf Probleme beim erstellen der Flächen basierend auf den Punkten wie in Abbildung 2 zu sehen. Das Problem liegt beim Verbinden der letzten Vertices im Mesh, welche nach einer Delaunay-Triangulation ein Loch hervorrufen. Die Herausforderung ist, dieses Loch genau so homogen zu schließen, wie den Rest der Oberfläche [2].

Die wohl am besten geeignete Lösung für das Erzeugen einer Sonnenoberfläche ist daher die *Icosphere*. Auch, wenn die Anzahl der Punkte bzw. Dreiecke des Meshes nach jeder neuen Unterteilungsiteration drastisch ansteigen, kann bereits nach wenigen Iterationen eine erkennbare Kugel mit gleichverteilten Eckpunkten erzeugt werden. Auch ist die Geometrie geschlossen.

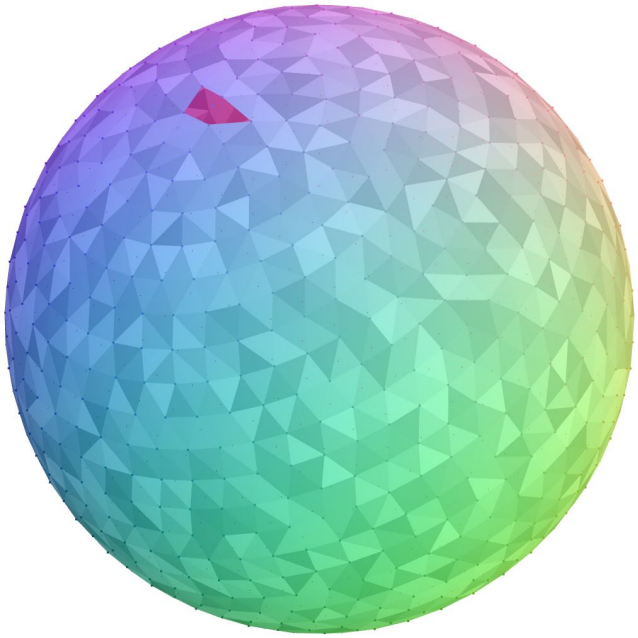


**Abbildung 2: Loch nach dem Ausführen einer Delaunay-Triangulation auf einem Fibonacci-Gitter.**

# 4 SONNENOBERFLÄCHE

In diesem Abschnitt wird besprochen, wie die Sonnenoberfläche in der Realität aussieht und wie die Erscheinung prodedural Nachgebildet werden kann. Dabei wird insbesondere auf die Entwicklung eines entsprechenden CG-Shaders eingengangen. Grundsätzlich besteht die Möglichkeit, eine Textur zu generieren, diese in eine Datei zu speichern und anschließend auf das Mesh zu übertragen. Dies hat jedoch den Nachteil, dass die Textur nur unter sehr hohem Aufwand über die Zeit verändert werden kann. Deshalb soll die Oberflächentextur im Shader selbst erzeugt werden, sodass diese in Echtzeit animiert werden kann. Auch wird das Verfahren erläutert, wie die Sonnengeometrie erzeugt wird und für den Shader vorbereitet wird.

## 4.1 Generierung der Geometrie

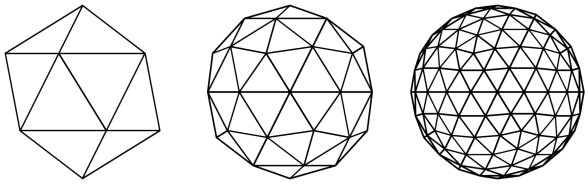
Bevor eine Textur für die Sonne dargestellt werden kann, müssen zuerst alle Rahmenbedingungen stimmen. Dies beinhaltet vor allem die Vorbereitung eines geeigneten Meshes für die Sonnenoberfläche. Im Folgenden wird der Algorithmus vorgestellt, welcher eine homogene Geometrie für die Sonne erstellt. Dieser erstellt in erster Linie die Eckpunkte einer Icosphere. Eine Icosphere besitzt 20 Dreiecke bestehend aus 12 Eckpunkten, welche die Grundlage für weitere Unterteilungsschritte sind. Dementspreched müssen diese initial berechnet werden. Anschließend startet der eigentliche Algorithmus zum Unterteilen der Flächen in kleinere Flächen. Dieser berechnet auf jeder Kante eines Dreiecks einen Mittelpunkt, sodass insgesamt drei neue Eckpunkte entstehen. Anschließend müssen die neu erzeugten Punkte auf die Einheitskugel abgebildet werden, indem die Positionsvektoren normalisiert werden. Algorithmus 1 soll diesen Vorgang verdeutlichen.

**Algorithmus 1:** Unterteilen von Dreiecksflächen auf einer Icosphere

**Eingabe :** Flächen des Basismeshes $M$, Rekursionslevel $n$
**Ausgabe :** Mesh einer Icosphere mit unterteilten Flächen

**for** $i \leftarrow 0$ **to** $n$ **do**
  $M' \leftarrow \{\}$;
  **foreach** $d \in M$ **do**
    $(\vec{a},\ \vec{b},\ \vec{c}) \leftarrow d$;
    $\vec{a}' \leftarrow$ calculateMiddlePoint$(\vec{a}, \vec{b})$;
    $\vec{b}' \leftarrow$ calculateMiddlePoint$(\vec{b}, \vec{c})$;
    $\vec{c}' \leftarrow$ calculateMiddlePoint$(\vec{c}, \vec{a})$;
    append$(M', (\vec{a},\ \vec{a}',\ \vec{c}'))$;
    append$(M', (\vec{b},\ \vec{b}',\ \vec{a}'))$;
    append$(M', (\vec{c},\ \vec{c}',\ \vec{b}'))$;
    append$(M', (\vec{a}',\ \vec{b}',\ \vec{c}'))$;
  **end**
  $M \leftarrow M'$;
**end**

**Abbildung 3: Vergleich zwischen unterschiedlichen Rekursionslevel von Algorithmus 1. Links befindet sich das Basismodell, in der Mitte die erste und rechts die zweite Rekursionsstufe.**



## 4.2 Fractal Brownian Motion

## 4.3 Cellular Noise

## 4.4 Entwicklung eines Shaders

# 5 PROTUBERANZEN

## 5.1 Wie Sonnenstürme entstehen

## 5.2 Magnetische Felder am Beispiel eines Dipols

## 5.3 Erstellung eines Vektorenfelds

## 5.4 Aufbau eines Partikelsystems

# 6 FAZIT UND AUSBLICK

# LITERATUR


[1] Blender 2.80 Manual Primitives. https://docs.blender.org/manual/en/2.80/modeling/meshes/primitives.html. Letzter Zugriff: 11.02.2021.

[2] Red Blob Games Delaunay + Voronoi on a sphere. https://www.redblobgames.com/x/1842-delaunay-voronoi-sphere/. Letzter Zugriff: 11.02.2021.

[3] M. Eder, T. Price, T. Vu, A. Bapat, and J. Frahm. Mapped convolutions. *CoRR*, abs/1906.11096, 2019.

[4] Á. González. Measurement of areas on a sphere using fibonacci and latitude–longitude lattices. *Mathematical Geosciences*, 42(1):49, Nov 2009.

[5] R. Marques, C. Bouville, M. Ribardière, L. P. Santos, and K. Bouatouch. Spherical fibonacci point sets for illumination integrals. *Computer Graphics Forum*, 32(8):134–143, 2013.