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 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.

2.1 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.

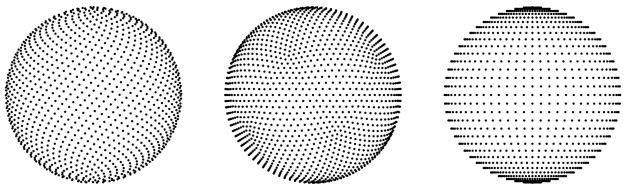

Abbildung 1: Vergleich der verschiedenen Resultate der Algorithmen zum Erzeugen von Punkten auf einer Kugel. Links: Fibonacci-Gitter, Mitte: Icosphere, Rechts: Längen-Breitengrad-Gitter.

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 [5].

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.

2.2 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* Φ und der *Golden-Angle* α [5, 6].

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

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.

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

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

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.

2.3 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 [4]. 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].

2.4 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].

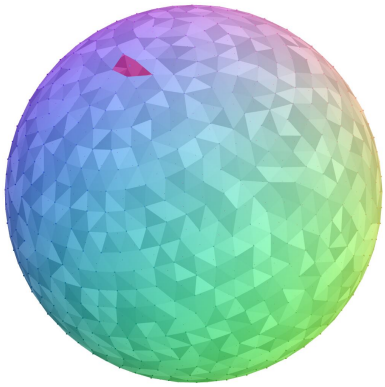

Abbildung 2: Loch nach dem Ausführen einer Delaunay-Triangulation auf einem Fibonacci-Gitter.

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.

3 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.

3.1 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

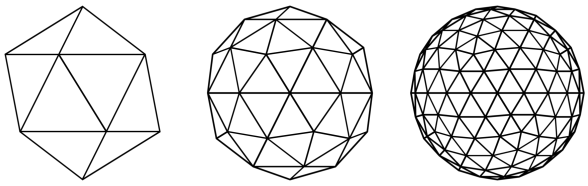

Abbildung 3: Vergleich zwischen unterschiedlichen Rekursionslevel von Algorithmus 1. Links befindet sich das Basismodell, in der Mitte die erste und rechts die zweite Rekursionsstufe.

für jede Dreiecksfläche 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

3.2 Fractal Brownian Motion

Eine Textur zu erzeugen, die zu jeden Zeitpunkt gleich aussieht wirkt oft langweilig und uninteressant. Auch, wenn man das Vorbild dieser Arbeit betrachtet – die Sonne –, unterscheidet sich die Oberfläche zu jedem Zeitpunkt von einer vorherigen Messung. Um ein solchen Phänomen zu modellieren, bedient man sich häufig dem Rauschen. Rauschen hat für unterschiedliche Fachgebiete eine unterschiedliche Bedeutung. Musiker verbinden damit störende Geräusche und Astrophysiker denken dabei an kosmische Hintergrundstrahlung [10]. In der Computergrafik versucht man hingegen, dieses Rauschen künstlich zu generieren, um beispielsweise eine Textur prozedural zu erzeugen. Auch in dieser Arbeit soll die Oberflächentextur der Sonne durch ein künstliches Rauschen erzeugt und verändert werden. Dies entspricht natürlich nicht der Realität, denn die Sonnenoberfläche hängt von vielen Faktoren, wie der Temperatur und dem Magnetfeld der Sonne ab. Auch kann man sogenannte Sonnenflecken beobachten, welche sich als dunkle Flecken auf der Oberfläche äußern. Diese sind nicht wirklich schwarz, sondern erscheinen dunkler als ihre Umgebung, da diese Stellen niedrigere Temperaturen aufweisen.

Anstatt nun zu versuchen, die Oberfläche der Sonne so realitätsgetreu wie möglich nachzubilden, wird in diesem Projekt versucht, mithilfe von künstlich erzeugtem Rauschen eine ähnliche Oberfläche zu generieren. Hierfür wird, wie der Titel des Abschnitts andeutet, *Fractal Brownian Motion* (fBM) verwendet. Oft werden hierbei Begriffe wie *Oktaven*, *Porosität* und *Zuwachs* genannt. Eine Oktave beschreibt eine Summe aus mehreren Rauschfunktionen, die Porosität die Multiplikation der Frequenz um einen konstanten Wert und der Zuwachs die Verringerung der Amplitude. Algorithmus 2 zeigt ein einfaches zweidimensionales frakturiertes Rauschen basierend auf [10].

Algorithmus 2: Fractal Brownian Motion mit eindimensionaler Eingabe

Eingabe : $x \in \mathbb{R}$, $octaves \in \mathbb{N}$
Ausgabe : $y \in \mathbb{R}$

$y \leftarrow 0$;
$lacunarity \leftarrow 2$;
$gain \leftarrow 0.5$;
$amplitude \leftarrow 0.5$;
$frequency \leftarrow 1$;
for $i \leftarrow 0$ **to** $octaves$ **do**
  $y = y + amplitude * noise(frequency \cdot x)$;
  $frequency = frequency \cdot lacunarity$;
  $amplitude = amplitude \cdot gain$;
end
return y;

Die Abbildung zum Erzeugen von Rauschen (*noise*) kann dabei beliebig gewählt werden. In den meisten Fällen erzeugt man jedoch kein echtes Rauschen, welches eine nicht stetische Funktion darstellen würde. Stattdessen wird mithilfe von überlagerten Wellenfunktionen künstliches stetiges Rauschen implementiert. Dies hat den Vorteil, dass sich das Rauschverhalten periodisch ist, d.h. dass sich die Werte abhängig von der Zeit irgendwann wiederholen. Dies ist unter anderem daher sinnvoll, da im Falle einer Textur sonst Sprünge bzw. harte Kanten erzeugt werden. Um fBM nun in der Textur verwenden zu können muss die Funktion *noise* näher betrachtet werden. Im Algorithmus 2 wird die Abbildung noise : $\mathbb{R} \rightarrow \mathbb{R}$ verwendet. Als Eingabe wird also eine reelle Zahl genommen und auf eine andere reelle Zahl abgebildet. Um nun eine Textur zu erzeugen muss beachtet werden, dass das Mesh, dessen Oberfläche mit der Textur ausgestattet werden soll, sogenannte UV-Koordinaten pro Eckpunkt definiert. Anhand dieser UV-Koordinaten wird dann die entsprechende Stelle in der Textur verwendet und auf den Eckpunkt projeziert. Da eine Textur häufig zweidimensional ist, sind die UV-Koordinaten ebenfalls zweidimensionale Angaben. Die Abbildung

muss nun also umgewandelt werden, sodass noise : $\mathbb{R}^2 \to \mathbb{R}$ gilt. Es ändert sich am Algorithmus selbst jedoch nichts, außer dass als Eingabe UV-Koordinaten $\in \mathbb{R}^2$ genommen werden, anstatt eine Ganzzahl $\in \mathbb{R}$. Jedoch muss dadurch ebenfalls die *noise*-Funktion von einem 2D-Vektor abhängen. Algorithmus 4 und 3 zeigen Beispielimplementationen dieser Funktionen. Damit wird bei einer Eingabe eines zweidimensionalen Vektors eine reelle Zahl abgebildet, die im darauf folgenden Schritt als Farbwert verwendet werden kann. Mit anderen Worten, der Algorithmus 5 liefert dann zu jeder UV-Koordinate einen entsprechenden Graustufenwert der Textur.

Algorithmus 3: Beispiel für die Hashfunktion *hash* mit einer zweidimensionalen Eingabe.

Eingabe : $\vec{n} \in \mathbb{R}^2$
Ausgabe : $h \in \mathbb{R}$

$a \leftarrow 123.456789$;
$b \leftarrow 987.654321$;
$c \leftarrow 54321.9876$;
$s \leftarrow \sin(\text{dot}(\vec{n}, (a, b)^T))$;
$h \leftarrow \text{frac}(s \cdot c)$;
return h;

Algorithmus 4: Beispiel für die Rauschfunktion *noise* mit einer zweidimensionalen Eingabe.

Eingabe : $\vec{p} \in \mathbb{R}^2$
Ausgabe : $r \in \mathbb{R}$

$\vec{i} \leftarrow \text{floor}(\vec{p}) \in \mathbb{R}^2$;
$\vec{u} \leftarrow \text{smoothstep}(0, 1, \text{frac}(\vec{p})) \in \mathbb{R}^2$;
$a \leftarrow \text{hash}(i + (0, 0)^T)$;
$b \leftarrow \text{hash}(i + (1, 0)^T)$;
$c \leftarrow \text{hash}(i + (0, 1)^T)$;
$d \leftarrow \text{hash}(i + (1, 1)^T)$;
$r \leftarrow \text{lerp}(\text{lerp}(a, b, x_u), \text{lerp}(c, d, x_u), y_u)$;
return r^2;

Algorithmus 5: Erzeugen der Sonnentextur unter der Verwendung von Fractal Brownian Motion

Eingabe : UV-Koordinate uv, Anzahl der Oktaven n
Ausgabe : Farbvektor $\vec{c}$

$y_{fbm} \leftarrow \text{fbm}(uv + \text{fbm}(uv, n), n)$;
$\vec{c} \leftarrow (y_{fbm}, y_{fbm}, y_{fbm})$;
return $\vec{c}$;

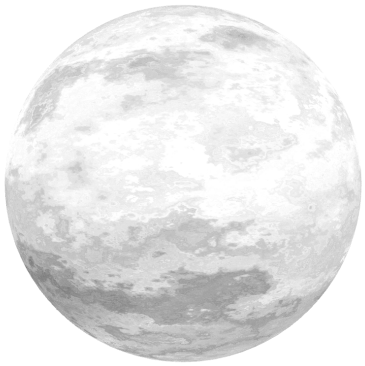

Abbildung 4: Anwendung von fBM auf einer Icosphere mit einer Amplitude von 0.86, einer Porosität von 4.56 und einem Zuwachs von 0.55.

4 PROTUBERANZEN

Ein weiterer wesentlicher Bestandteil dieser Arbeit besteht darin, die Protuberanzen der Sonne, umgangsprachlich Sonnenstürme, zu simulieren. Anders als bei der Sonnenoberfläche und ihrer Textur wird hier versucht, diese anhand von physikalischen Eigenschaften der Sonne zu berechnen. Ferner sollen Magnetfelder berechnet werden, die Einfluss auf Partikel der Sonne nehmen. Dadurch sollen Protuberanzen entstehen und simuliert werden.

Eine Protuberanz ist ein auf der Sonnenoberfläche auftretendes Phänomen. Sie entstehen durch abprupte Neuverbindungen von Magnetfeldlinien. Dies wird auch als Rekonnexion bezeichnet und geschieht immer dann, wenn zwei sich zwei Magnetfelder aufeinander zu ausbreiten und sich dann neu verbinden [8]. Dabei wird eine große Menge Energie freigesetzt, die dann Material aus der Sonne hinausschleudert [11]. Da die Sonne vermutlich nicht nur ein einfaches Magnetfeld besitzt, wäre eine vollständige Simulation aller Magnetfelder bzw. Pole nicht sinnvoll, da dies ab einer bestimmten Anzahl von Magnetfeldern einen zu hohen Rechenaufwand bedeuten würde und man die Phänomenen nicht in Echtzeit darstellen könnte. Deshalb wird sich im folgenden auf das Modell eines Dipols gestützt, welches einfach zu berechnen und trotzdem realitätsnahe Ergebnisse erzielen kann. Ein Dipol ist im Prinzip ein einfacher Magnet und besteht aus zwei unterschiedlichen Ladungen, anhand dessen ein Magnetfeld berechnet werden kann. Der Dipol ist das einfachste Modell, welches sich mit Magnetismus beschäftigt. Die magnetische Flussdichte $\vec{B}$ eines Dipols wird durch folgende Gleichung beschrieben [9].

$$\vec{B}(\vec{r}) = \frac{\mu_0}{4\pi r^2} \frac{3\vec{r}(\vec{m} \cdot \vec{r}) - \vec{m}r^2}{r^3}$$

Dabei beschreibt $\vec{r}$ den Positionsvektor des zu betrachtenden Punktes und $r = |\vec{r}|$ die Länge von $\vec{r}$, μ_0 die magnetische Feldkonstante und $\vec{m} = p(\vec{r}_+ - \vec{r}_-)$, wobei p die Polstärke, $\vec{r}_+$ die Position vom Pluspol und $\vec{r}_-$ die Position vom Minuspol sind [3]. Mithilfe dieser Gleichung lassen sich für alle Punkte in dem Magnetfeld eines Dipols die entsprechenden Kräfte berechnen.

4.1 Vektorenfelder und Partikel

Das Problem bei der Simulation von Wolken, Rauch, Feuer und auch von Protuberanzen ist, dass diese Gebilde nicht wohlgeformt sind, wie beispielsweise ein Würfel, eine Kugel oder sogar ein 3D-Modell eines Gebäudes. Anstatt zu versuchen Rauch mithilfe von starren Körpern darzustellen, kam man auf die Idee, Partikelsysteme zu definieren. Partikel repräsentieren dabei keine einzelnen Objekte, sondern haben die Aufgabe, Objekte aufgrund ihrer Eigenschaften selbst zu formen. Dabei sind die Partikel keine statischen Entitäten, sondern üben Kräfte untereinander aus und erfahren Kräfte von außen, die beispielsweise eine Bewegung der einzelnen Partikel hervorrufen. Ferner besitzen Partikel bestimmte Eigenschaften wie z.B. eine Lebensdauer, wodurch sie *geboren* werden, aber auch *sterben* können. Weitere übliche Eigenschaften sind Geschwindigkeit, Position, Größe, Farbe und Form. Ein Objekt, welches durch ein Partikelsystem dargestellt wird, ist nicht deterministisch, da seine Form nicht komplett beschrieben wird. Stattdessen werden stochastische Methoden angewandt, um das Aussehen dieser Objekte zu verändern [7].

Auch im Falle der Sonne, um die es in dieser Arbeit handelt, können Partikelsysteme dabei helfen, komplexe Phänomene realitätsnah darzustellen, anstatt sie in vorherigen Arbeitsprozessen statisch zu modellieren. In diesem Projekt werden Partikelsysteme von Unity verwendet. Genauer gesagt werden *Visual-Effect-Graphen* definiert, welche die Partikel und Kräfte, die auf sie einwirken, kontrollieren. Damit sollen vor allem versucht werden, die auf der Sonne auftretenden Protuberanzen zu simulieren. Hierfür wird der im Abschnitt 4 kurz besprochene Dipol verwendet werden. Die Idee ist, nicht alle Magnetfelder der Sonne abzubilden, Rekonnexionen ihrer Magnetfeldlinien zu berechnen und damit Protuberanzen zu erzeugen, sondern pro Protuberanz einen Dipol zu definieren, welcher die Form eines solchen Materiestroms beeinflusst bzw. hervorruft. Um nun ein Magnetfeld als Krafteinwirkung auf Partikel zu erstellen, wird ein Vektorenfeld aufgebaut und anschließend als 3-dimensionale Textur gespeichert. Diese Textur wird dann in das Partikelsystem eingebaut und beeinflusst damit die Bewegung und Geschwindigkeit der Partikel.

Ein Vektorfeld ist eine Sammlung von Vektoren, die jeweils eine Position und eine Richtung besitzen. Damit erfährt jedes Objekt, welches sich an einer bestimmten Stelle innerhalb des Vektorfeld befindet, eine Kraft in der jeweiligen Richtung. Um ein solches Feld zu erzeugen, werden Vektoren gleichmäßig im Raum verteilt. Anschließend werden die Richtungen mithilfe der magnetischen Flussdichten für jeden dieser Vektoren bestimmt, sodass sich diese anhand des Magnetfelds ausrichten. Jetzt, wo Vektoren im Raum verteilt und ihre Richtungen definiert wurden, wird eine 3D-Textur erzeugt, die all diese Informationen speichert. Dabei wird die Position eines Vektors durch den Index des Pixels und die Richtung durch eine Farbe innerhalb der Textur repräsentiert. Abbildung 5 stellt dies in einem zweidimensionalen Raum dar. Die damit erstellte 3D-Textur bestehend aus Richtungsvektoren, die von einer Position abhängen, kann nun innerhalb eines Partikelsystems verwendet werden. Damit die Partikel die Form einer Protuberanz annehmen, müssen deren Eigenschaften entsprechend justiert werden. Beispielsweise muss die initiale Position der Partikel innerhalb der

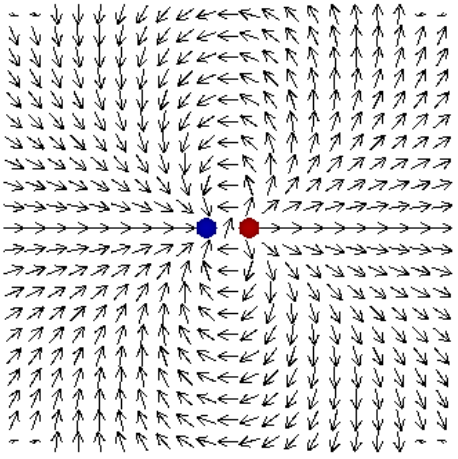

Abbildung 5: In Unity berechnete Magnetfeldlinien des Dipol-Modells. Der Einfachheit halber wird hier ein 2D-Vektorenfeld dargestellt.

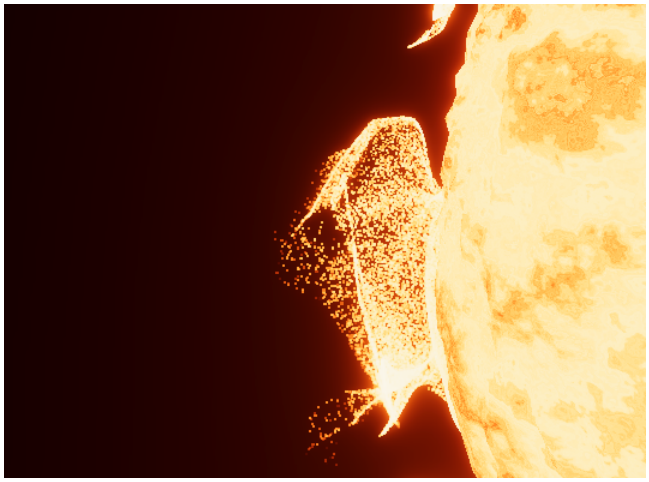

Abbildung 6: Kraftauswirkung eines Dipol-Magnetfeldes auf Partikel der Sonne. Das Ergebnis sind vereinfachte Protuberanzen.

Sonne sein. Dann muss eine explosionsähnliche Kraft auf sie einwirken, die die Partikel nach außen treibt. Diese Kraft simuliert damit die Rekonnexion von Magnetfeldlinien und steht senkrecht auf der Sonnenoberfläche und zeigt nach außen. Da das Magnetfeld ebenfalls vorhanden ist, werden sich die Partikel entlang der Magnetfeldlinien bewegen. Das Ergebnis sind Protuberanzen, die mithilfe eines Dipols simuliert und in Abbildung 6 betrachtet werden können.

5 FAZIT UND AUSBLICK

In dieser Arbeit wurde besprochen, wie eine Sonne mithilfe von prozeduralen Algorithmen simuliert werden kann. Dabei wurde

explizit auf verschiedene Möglichkeiten zum Generieren von Kugeloberflächen eingegangen und warum inhomogene Eckpunkte auf einer solchen Kugel problematisch sind. Es wurde sich für eine Icosphere entschieden, da dort Eckpunkte gleichmäßig verteilt auf einer Kugeloberfläche liegen und das Fibonacci-Gitter einige zusätzliche Probleme, wie beispielsweise ein Loch im Mesh nach einer Delaunay-Triangulation, bereitet. Anschließend wurde ein Algorithmus zum Berechnen von prozeduralen Texturen mithilfe von *Fractal Brownian Motion* erläutert, auf die Sonnenoberfläche übertragen und die Ergebisse dargestellt. Zum Schluss wurde der Dipol, sein Magnetfeld und magnetische Flussdichte kurz besprochen und die Theorie in ein Vektorenfeld übertragen. Dieses wurde dann als Kraftfeld in ein Partikelsystem übertragen, sodass sich die entsprechenden Partikel wie Protuberanzen der Sonne verhalten.

Zukünfig wäre es sinnvoll, den Dipol durch komplexere Modelle auszutauschen, die ein Magnetfeld hervorrufen. Dadurch könnte eine noch realistischere Simulation durchgeführt werden. Auch könnte untersucht werden, wie sich verschiedene Dipole beeinflussen und wie die daraus entstehenden Magnetfelder verwendet werden können. Darauf basieren wäre es ebenfalls interessant, wie Rekonnexionen und deren Energiefreigabe zur Simulation von Proutberanzen verwendet werden können.

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] W. Brown. *Magnetostatic Principles in Ferromagnetism.* Selected topics in solid state physics. North-Holland Publishing Company, 1962.

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

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

[6] 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.

[7] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, Apr. 1983.

[8] F. Spanier. Welt der Physik: Der Einfluss des Sonnenwinds auf die Erde. https://www.weltderphysik.de/gebiet/erde/erde/sonnenwind/. Letzter Zugriff: 14.02.2021.

[9] H. Stöcker. *Taschenbuch der Physik : -*. H. Deutsch, Frankfurt/Main, 2013.

[10] P. G. Vivo and J. Lowe. The Book of Shaders: Fractal Brownian Motion. https://thebookofshaders.com/13/. Letzter Zugriff: 13.02.2021.

[11] H. Zirin. Britannica: Prominences. https://www.britannica.com/place/Sun/Prominences. Letzter Zugriff: 14.02.2021.