2011

Modellierung von 3D Wolken auf Basis von 2D Bewölkungsdaten und Rendering mittels Rastergrafik

Bachelor-Arbeit von Sascha Kolodzey
Fachbereich Informatik - Universität Osnabrück
18.02.2011

Gutachter:
Prof. Dr. Oliver Vornberger
Juniorprof. Dr.-Ing. Elke Pulvermüller

1 Danksagung

Hiermit möchte ich mich bei allen Menschen bedanken, die mich bei dieser Arbeit unterstützt haben. Insbesondere bei Henning Wenke, der immer Zeit hatte, um über Probleme zu diskutieren und mich mit hilfreichen Anregungen unterstützte. Ferner bei Frau Juniorprof. Dr.-Ing. Elke Pulvermüller und Herrn Prof. Dr. Oliver Vornberger, die diese Arbeit begutachten. Ebenfalls Dank an Friedhelm Hofmeyer für die technische Unterstützung. Außerdem möchte ich meiner Mutter für das Korrekturlesen und meinem Vater für die Unterstützung während des Studiums danken. Für die emotionale Unterstützung während der Arbeit möchte ich mich bei meiner Freundin Annika Wegener bedanken.

2 Abstract

Im Rahmen der Arbeitsgruppe „Medieninformatik“ an der Universität Osnabrück besteht ein Schwerpunkt der Arbeit in der „Interaktiven Visualisierung von Wetterdaten im Web“. Unter anderem werden Daten wie Temperatur, Wind, Niederschlag und Bewölkung gleichzeitig angezeigt. Eine wolkenähnlichere Darstellung könnte möglicherweise für den Benutzer hilfreich sein bei der Zuordnung der Daten. Deshalb ist das Ziel der Arbeit, mit Hilfe der Programmierung von modernen Grafikkarten und unter Verwendung von Grafik APIs Algorithmen zu entwickeln, die eine dreidimensionale Darstellung von 2D Bewölkungsdaten ermöglichen.

Inhaltsverzeichnis

3 Einleitung

3.1 Motivation

Die innerhalb der Arbeitsgruppe „Medieninformatik“ entwickelten Algorithmen zur Visualisierung von Bewölkung haben eine zweidimensionale Darstellung. Oft werden mehrere Daten gleichzeitig visualisiert, sodass eine genaue Zuordnung der Daten dem Benutzer unnötig schwer fällt.
Bei den zugrundeliegenden Daten handelt es sich um zweidimensionale Bewölkungsdaten, die auf Basis von Algorithmen zu einer dreidimensionalen Darstellung transformiert werden sollen. Um dies zu ermöglichen, werden Annahmen zugrunde gelegt, wie sich die Bewölkung in der dritten Dimension ausdehnen könnte. Der Fokus dieser Arbeit liegt in der Entwicklung von Algorithmen zur Modellierung von 3D Bewölkung und Rendering mittels Rastergrafik. Hierbei wird die Bewölkung bis zu einem bestimmten Grad verfälscht. Wie sich diese Verfälschung, und vor allem, wie sich die dreidimensionale Darstellung der Bewölkung auf das Verständnis der visualisierten Wetterdaten auswirkt, muss in einer anschließenden User-Study geprüft werden.

3.2 Methodisches Vorgehen

Zu Beginn wird ein Überblick über die Werkzeuge gegeben, die im Rahmen dieser Arbeit Verwendung finden. Um die gegebenen zweidimensionalen Bewölkungsdaten in die dritte Dimension zu transformieren, wird hauptsächlich die Technik *Noise* verwendet. Dieses Verfahren wird im Anschluss an die Werkzeugeinführung weitreichend erläutert. Im Verlauf werden zwei grundsätzlich verschiedene Ansätze vorgestellt:
Einer beschäftigt sich mit Algorithmen zur Visualisierung dreidimensionaler Daten. Der Bewölkungsgrad wird mit Hilfe einer dreidimensionalen Noise Textur transformiert und unter Anwendung von Volumen Rendering Techniken dreidimensional visualisiert. Ein anderer Ansatz generiert ein Partikelsystem und animiert dieses auf Basis des Bewölkungsgrades. Hierbei kommt ebenfalls die Technik *Noise* zum Einsatz.
Abschließend werden beide Ansätze dahingehend miteinander verglichen, inwieweit sie für eine Visualisierung von Bewölkungsdaten geeignet sind.

4 Format

Der Ausgangspunkt ist ein Datensatz, der als Bewölkungsgrad interpretiert wird. Der Bewölkungsgrad der Erdoberfläche oder ein Ausschnitt der Erde liegt in Form eines Skalarfeldes vor. Das heißt, dass jedem Punkt (x, y) auf der Oberfläche ein skalarer, reeller Wert aus dem Intervall [0,1] zugeordnet werden kann. In diesem Fall steht 0 für 0% und 1 für 100% Bewölkung im Punkt (x, y) (Abb.1). Besitzt der Datensatz eine zeitliche Abfolge von Bewölkungsdaten, ist weiterhin möglich, diese Bewölkungsgradsätze nacheinander einzulesen, um somit einen zeitlichen Verlauf der Bewölkung zu erhalten.

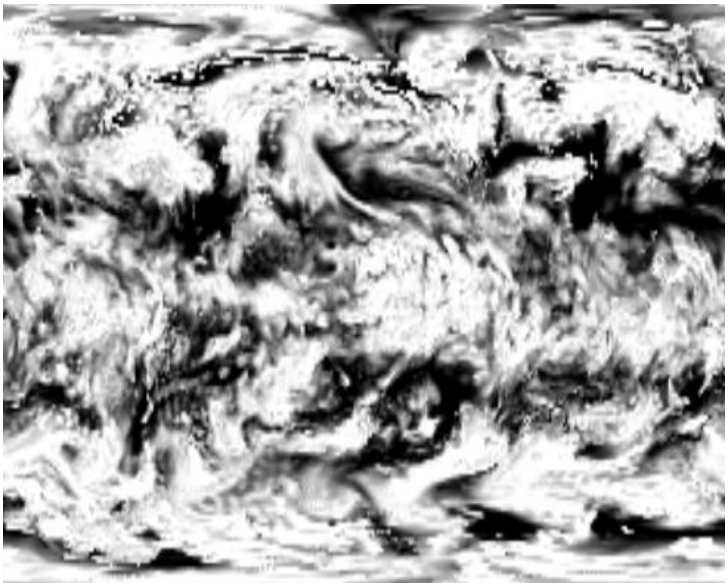

Abbildung 1: Bewölkungsgrad, keine Wolken(schwarz), Wolken(weiß)

5 Werkzeuge

5.1 OpenGL

OpenGL (Open Graphics Library) ist momentan eine der führenden 3D Grafik Spezifikationen. Seit Veröffentlichung der Version 1.0 im Jahr 1992, hat sich die Bibliothek stetig weiterentwickelt. OpenGL wurde anfangs von Silicon Graphics, Inc. (SGI) entwickelt. Seit dem Jahr 2006 liegt die Weiterentwicklung von OpenGL in der Verantwortung der Khronos Group. Zu den über 100 Mitgliedern der Khronos Group gehören unter anderem NVIDIA und Google [KHR]. OpenGL befindet sich aktuell in der Version 4.1, welche am 26. Juli 2010 veröffentlicht wurde [OGL]. Ein großer Vorteil gegenüber anderen Produkten (z.B. Direct3D) ist die Plattformunabhängigkeit.
Im September 2004, mit der Herausgabe der Version 2.0, wurde die OpenGL Shading Language 1.30 eingeführt. Diese machte es den Entwicklern erstmals möglich, die bis zu diesem Zeitpunkt *fixed* Rendering Pipeline zu programmieren. Dadurch eröffnete sich eine ganz neue Ebene, Objekte zu zeichnen und einzufärben (im Folgenden rendern genannt).

In diesem Kapitel wird nicht auf jedes Detail der OpenGL Bibliothek eingegangen. Es soll lediglich einen groben Überblick verschaffen.

5.1.1 Geometrie

Eine Geometrie besteht aus einer Menge von dreidimensionalen Punkten im Raum (im Folgenden Vertices genannt). Vertices enthalten in der Regel mehr Informationen als nur die Position. In vielen Fällen besitzt ein Vertex zusätzlich eine Normale oder eine Texturkoordinate. Wie OpenGL diese Vertices interpretiert, ist vom gewählten Primitive Typ abhängig. In der aktuellen Version 4.1 definiert OpenGL 12 Primitives. [SEG]

Im Folgenden werden drei dieser Typen vorgestellt. Die Nummerierung in den Abbildungen entspricht der Indizierung der einzelnen Vertices.

GL_TRIANGLE
Jeweils drei aufeinanderfolgende Vertices werden als Dreieck interpretiert (Abb.2).
Für die Darstellung von N Dreiecke(n) werden $3 * N$ Vertices benötigt.

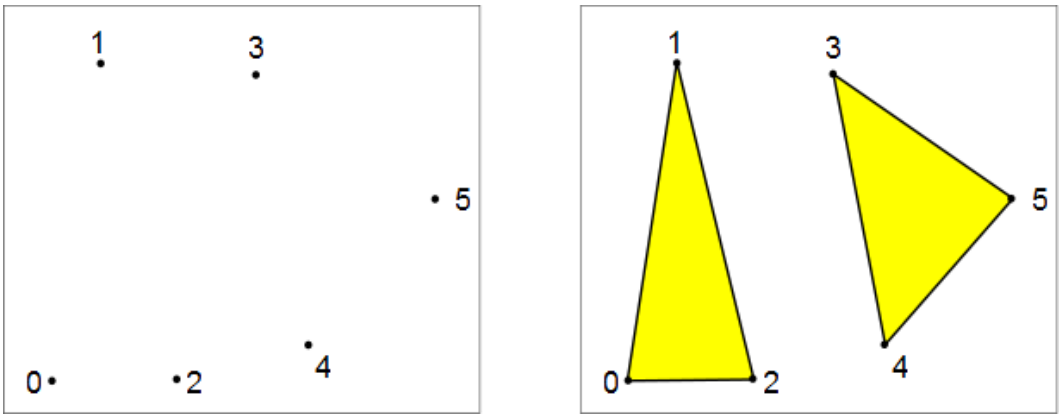

Abbildung 2: GL_TRIANGLE

GL_TRIANGLE_STRIP
Die ersten drei Vertices definieren das initiale Dreieck. Alle weiteren Dreiecke ergeben sich aus dem darauffolgenden Vertex und den letzten beiden Vertices (Abb.3).
Diese Methode ist dann hilfreich, wenn eine geschlossene Geometrie vorliegt und sich mehrere *Vertices* auf derselben Position befinden. Für die Darstellung von N Dreiecke(n) werden $2 + n$ Vertices benötigt. Gegenüber $GL\_TRIANGLE$ werden im Idealfall $2 * (n - 1)$ Vertices eingespart.

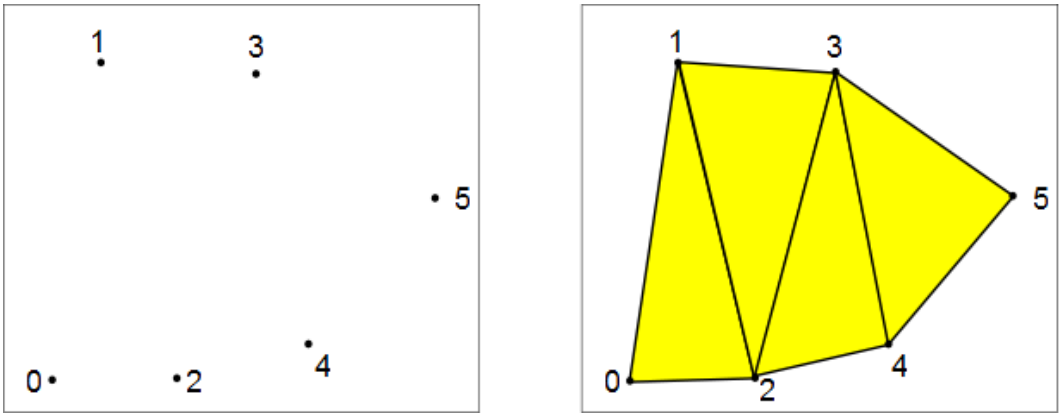

Abbildung 3: GL_TRIANGLE_STRIP

GL_POINT
Jeder Vertex steht für einen Punkt und ist nicht Teil einer Menge von Vertices, die zusammen zu einem Primitiv verbunden werden (Abb.4). Ferner besitzen Vertices, deren Typ *GL_POINT* entspricht, eine Größe. Die Größe hat Einfluss auf die Ausdehnung des Punktes in die x und y Richtung und somit auf die Anzahl der Bildpunkte (im Folgenden Pixel genannt).

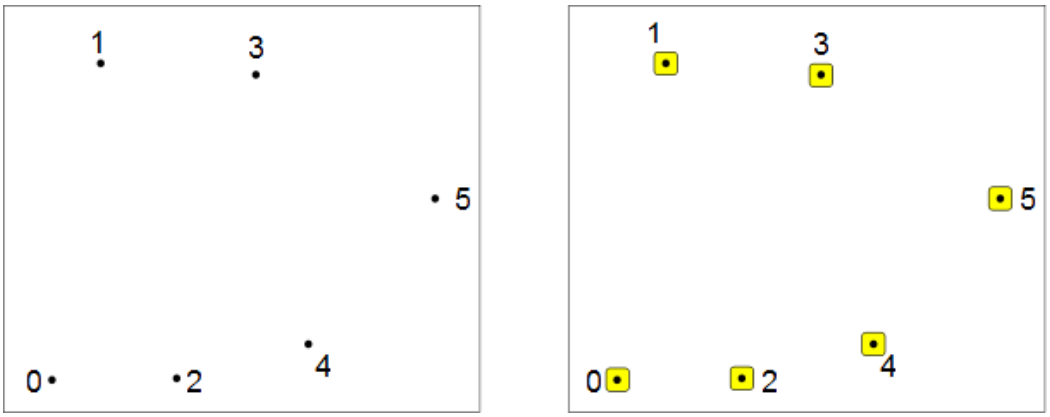

Abbildung 4: GL_POINT

5.1.2 Rendering Pipeline

Die Rendering Pipeline beschreibt den Vorgang eines Grafiksystems, dreidimensionale Szenen zu rastern und das Ergebnis auf einem Bildschirm anzuzeigen. Sie lässt sich in feste und dynamische Stationen unterteilen. Mit der Einführung programmierbarer Shadereinheiten ist es nun möglich, dynamische Stationen der Rendering Pipeline zu programmieren. Alle übrigen Stationen sind nicht modifizierbar und (soweit unterstützt) in der Grafikkartenhardware bereitgestellt [WRI]. Im Folgenden werden die Stationen der Rendering Pipeline (OpenGL Version 2.0 bis 3.1) aus Sicht des Programmierers vereinfacht dargestellt. Wie diese Stationen in der Hardware implementiert sind, ist vom Hersteller der Grafikkarten abhängig.

Vertexshader (dynamisch)
Für jeden Vertex wird eine Instanz des Vertexshaders aufgerufen. In dieser ist es möglich, Operationen auf dem Vertex durchzuführen.

Primitive Assembling (statisch)
Aus den einzelnen Vertices wird das definierte Primitiv zusammengesetzt.

Primitive Processing (statisch)
Durchführung von Clipping, Culling und Perspective Divide.

Rasterizer (statisch)
Der Rasterizer generiert Fragments auf Basis des Primitive Type und reicht diese an den Fragmentshader weiter.

Fragmentshader (dynamisch)
Für jedes generierte Fragment wird eine Instanz des Fragmentshaders aufgerufen. Im Fragmentshader kann die Farbe des Fragments zugewiesen werden.

PerFragmentOperations (statisch)
Ausführung des Tiefentests und Blending (falls aktiviert) für jedes Fragment. Besteht das Fragment den Tiefentest, kann der Farbwert in den Framebuffer geschrieben und angezeigt werden.

5.1.3 Hardware Shader

Mit dem Wort Shader werden Programme bezeichnet, die auf heutigen Grafikkarten ausgeführt werden können. Diese erlauben es dem Programmierer, bestimmte Teile der Rendering Pipeline zu programmieren. Prinzipiell werden bis zur OpenGL Version 3.1 Shader in zwei Arten unterteilt, den Vertex- und den Fragmentshadern. Während der Vertexshader dynamische Veränderungen der Vertices und geometrische Berechnungen ausführt, berechnet der Fragmentshader den Farbwert jedes Fragments. Mit DirectX 10 / OpenGL 3.2 wurde ein weiterer Shader eingeführt, der Geometryshader. Dieser kann, im Gegensatz zum Vertexshader, Geometrie der Szene hinzufügen oder eliminieren. Die letzte Errungenschaft in Bezug auf Shader sind die Tesselationshader (DirectX 11 / OpenGL 4.0). Die Tesselationshader sind in der Lage, Geometrie noch weiter zu verfeinern, um somit einen sehr hohen Detailgrad zu erreichen. Im Gegensatz zum Geometryshader kann der Tesselationshader keine beliebige Geometrie erzeugen oder löschen. Shader werden in einer dafür vorgesehenen Programmiersprache geschrieben. In OpenGL wird ein C-Dialekt mit

dem Namen OpenGL Shading Language (GLSL) verwendet. Der Quelltext wird zur Laufzeit der Grafikanwendung gelesen und vom Treiber der Grafikkarte in Maschinencode übersetzt, der auf der Grafikkarte ausgeführt werden kann. [WRI]

Im Weiteren wird auf den traditionellen Vertex- und Fragmentshader genauer eingegangen.

Vertex Shader
Für jedes Vertex einer Grafikanwendung wird der Vertexshader in einem Durchlauf der Rendering Pipeline genau einmal und unabhängig voneinander aufgerufen. Hierbei arbeitet der Vertexshader auf den hereinkommenden Vertexdaten. Wie in Abschnitt 5.1.1 erläutert, kann ein Vertex über weitaus mehr Informationen verfügen, als seine Position im Raum. Hierbei sei anzumerken, dass in einer Instanz eines Vertexshaders keine Informationen über benachbarte Vertices oder die zugrundeliegende Topologie vorliegen. Operationen, die ein solches Wissen benötigen, können im Vertexshader nicht ohne weiteres ausgeführt werden. [ROS]

Übliche Operationen, die der Vertexshader ausführt sind:

- Änderungen an der Position des Vertex
- Normalentransformation und Normalisierung
- Erzeugung von Texturkoordinaten
- Lichtberechnung

In Abbildung 5 wird ein relativ einfacher Vertexshader vorgestellt. Dieser erhält die Position als Eingabe und reicht diese an die nächste Stufe in der Pipeline weiter. Die Variable $vPos$ wird interpoliert und später im Fragmentshader zugänglich sein. Zum Schluss wird die Position des Vertex mit der $Model-View-Projection-Matrix$ multipliziert und somit die endgültige Position bestimmt.

```
#version 140
uniform mat4 modelViewProjetionMatrix;
in vec3 vPosition;
out vec3 vPos;
void main() {
      vPos = vPosition;
      gl_Position = modelViewProjetionMatrix * vec4( vPosition,1 );
}
```

Abbildung 5: Beispiel Vertexshader

Die built-in Variable $gl\_Position$ ist ein vierdimensionaler Vektor und muss im Vertexshader gesetzt werden, ansonsten würde ein Fehler geworfen werden. Mit den Schlüsselwörtern in und out werden hereinkommende Daten (in) und Daten, die weitergereicht werden (out), gekennzeichnet. In diesem Beispiel wird die Variable $vPos$ zur nächsten Station der Pipeline durchgereicht. Das Schlüsselwort $uniform$ deklariert globale Variablen, die in allen laufenden Shaderinstanzen sichtbar sind.

Fragment Shader
Wie bei dem oben genannten Vertexshader durchlaufen auch alle Fragments, die durch den Rasterizer generiert wurden, genau einmal und unabhängig voneinander den Fragmentshader. Die im Vertexshader deklarierten *out* Variablen werden interpoliert und sind im Fragmentshader für weitere Operationen zugänglich.

```
#version 140
in vec3 vPos;
out vec4 fragColor;
void main() {
      fragColor = vec4( vpos, 1 );
}
```

Abbildung 6: Beispiel Fragmentshader

Abbildung 6 zeigt einen einfachen Fragmentshader, der die übergebene Position als Farbwert setzt. Der Fragmentshader muss ebenfalls einen vierdimensionalen Vektor als *out* Variable definieren und diesen auch setzen. Sollte dies nicht der Fall sein, wird ein Fehler geworfen.

Übliche Operationen für den Fragmentshader sind:

- Verarbeitung interpolierter Variablen
- Zugriff auf Texturobjekte
- Nebel
- Alphawert-Berechnungen
- Posteffekte

Wie auch Vertices, haben Fragments keine Informationen über die Topologie, zu der sie gehören und über weitere Fragments des Primitive. [ROS]

5.1.4 Blending

Blending ist eine Methode, transparente Objekte darzustellen. Hierzu wird beim Durchführen der PerFragmentOperations der zu schreibende Farbwert mit dem bereits im Framebuffer stehenden Farbwert vermischt. OpenGL bietet für verschiedene Mischverhältnisse Blendkonstanten, die über die Funktion

$$glBlendFunc(source, destination)$$

gesetzt werden können.

OpenGL mischt zwei Farbwerte nach folgender Formel:

$$C_f = C_s * src + C_d * dst$$

Hierbei ist C_d die Farbe, die bereits im Framebuffer gespeichert ist und C_s die zu schreibende Farbe. Folglich ist C_f der resultierende Farbwert, der nach dem Blending im Framebuffer steht.

Tabelle 1: Blendkonstanten

source	*destination*
GL_ONE	*GL_ONE*
GL_ZERO	*GL_ZERO*
GL_DST_COLOR	*GL_SRC_COLOR*
GL_ONE_MINUS_DST_COLOR	*GL_ONE_MINUS_SRC_COLOR*
GL_SRC_ALPHA	*GL_SRC_ALPHA*
GL_ONE_MINUS_SRC_ALPHA	*GL_ONE_MINUS_SRC_ALPHA*
GL_DST_ALPHA	*GL_DST_ALPHA*
GL_ONE_MINUS_DST_ALPHA	*GL_ONE_MINUS_DST_ALPHA*
GL_SRC_ALPHA_SATURATE	

Tabelle 1 zeigt alle zugelassenen Blendkonstanten für die Übergabeparameter der Blendfunktion. Eine sehr häufig gewählte Kombination der Blendkonstanten ist *GL_SRC_ALPHA* für den Source und *GL_ONE_MINUS_SCR_ALPHA* für die Destination Komponente. Diese Kombination ist auch unter dem Namen Alpha Blending bekannt (Abb.7) [WRI]. Für das Alpha Blending ergibt sich folgende Formel:

$$C_f = C_s * a_s + (1 - a_s) * C_d$$

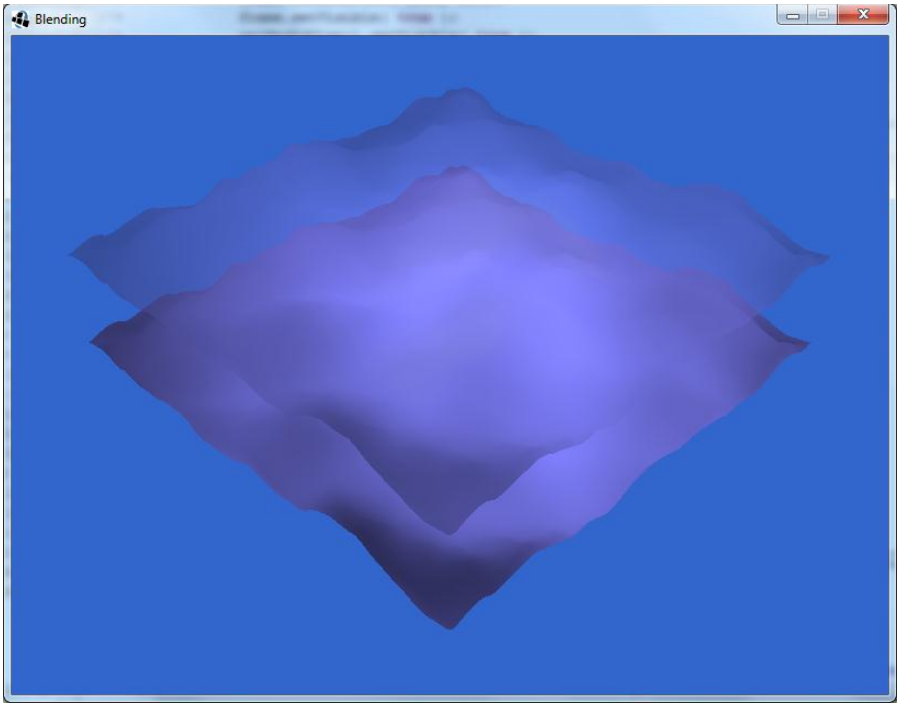

Abbildung 7: Alpha Blending ($a_s = 0.66$, $a_d = 1 - 0.66 = 0.33$)

5.1.5 Texture Mapping

Texture Mapping ist eine Technik, Bilder (im Folgenden Texturen genannt) auf eine Geometrie zu projizieren. Dadurch ist es mit relativ wenig Rechenaufwand möglich, die Oberfläche der Geometrie mit Oberflächendetails zu erweitern (Abb. 8).

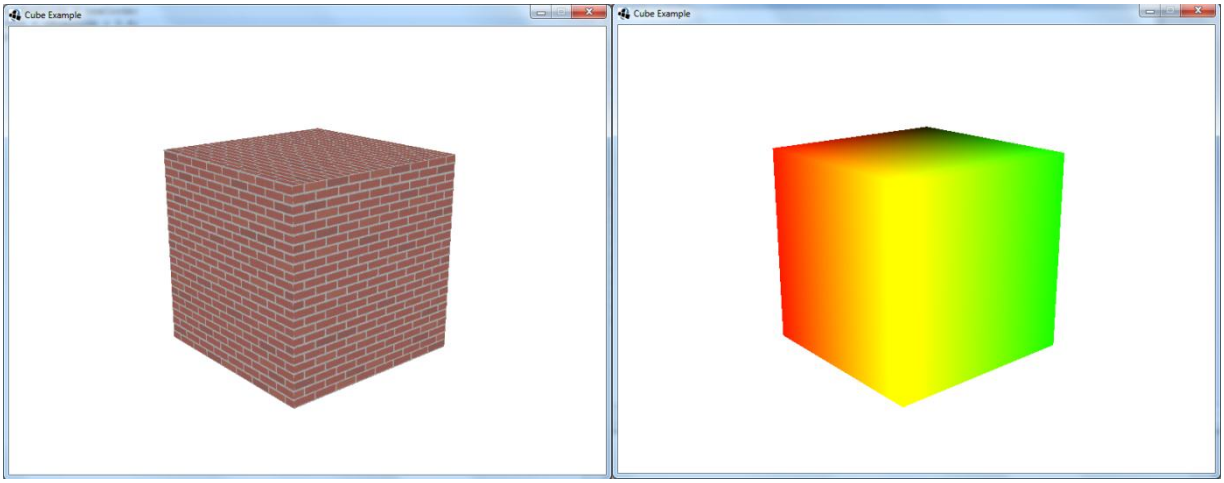

Abbildung 8: texturierter Würfel (links), nicht texturierter Würfel (rechts)

Für die Texturierung wird jedem Vertex eine Texturkoordinate zugeordnet. Um Irritationen vorzubeugen, liegen die zweidimensionalen Texturkoordinaten im uv-Koordinatensystem vor. Sollten die Texturkoordinaten von der Position abweichen, können diese immer noch über abweichende uv-Koordinaten eindeutig bestimmt werden. Die uv-Koordinaten werden für alle Fragments des Primitives interpoliert. Im Fragmentshader können nun mit Hilfe der interpolierten Koordinaten Texturobjekte ausgelesen und die Farbwerte dem jeweiligen Fragment zugewiesen werden. In Abbildung 9 wird dieses Vorgehen noch einmal verdeutlicht.

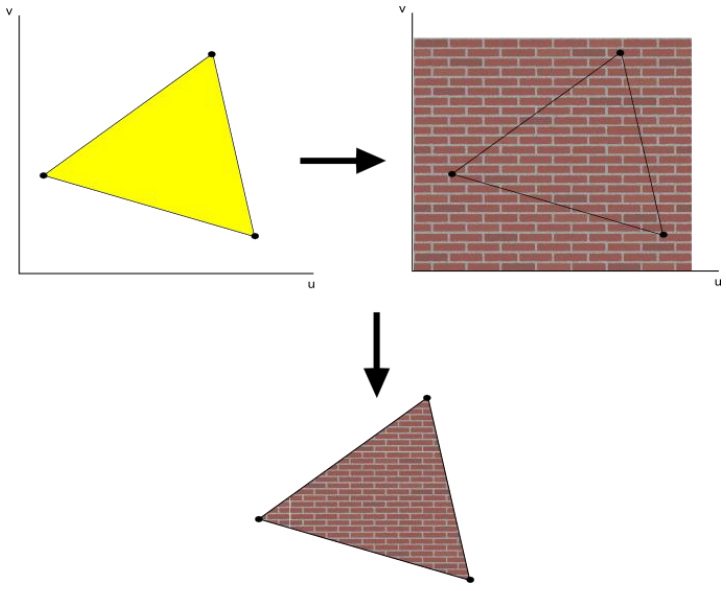

Abbildung 9 : Texture Mapping

5.1.6 Java und OpenGL

Das zu erstellende Programm soll später einmal als Applet in einem Webbrowser ausgeführt werden. Applets sind Java Programme, die von einem Webserver heruntergeladen werden und auf dem Clientrechner ausgeführt werden können. Zur Programmierung muss eine Java Wrapperbibliothek für OpenGL verwendet werden. Um eine Garantie der Lauffähigkeit zu gewährleisten, ist die Installation einer aktuellen Java Version und einer aktuellen OpenGL Version auf dem Clientrechner erforderlich.

Momentan stehen zwei nennenswerte OpenGL Wrapperbibliotheken zur Verfügung:

- **Java Bindings for OpenGL (JOGL)**
  JOGL wird seit 2003 von Sun Microsystems SGI entwickelt und ist ein Open Source Projekt. Es wurde dahingehend entwickelt, die Vorteile aller vorherigen Wrapperbibliotheken (gl4java, LWJGL und Magician) zu kombinieren. Außerdem sollte auch die Spieleindustrie auf Java aufmerksam gemacht werden. [JOGL]

- **Lightweight Java Game Library (LWJGL)**
  LWJGL ist momentan in der Version 2.6 (18.10.2010) verfügbar und wurde von einem Mitglied von PuppyGames ins Leben gerufen. Der Fokus der Bibliothek liegt auf der Spieleprogrammierung. Es wurde nach folgenden Richtlinien programmiert: Geschwindigkeit, Portabilität, Einfachheit, Minimalismus und Sicherheit. [LWJGL]

In dieser Arbeit wurde die Lightweight Java Game Library angewendet. Im Gegensatz zu LWJGL, arbeitet JOGL mit Swing Komponenten, was sich in der Geschwindigkeit widerspiegelt. JWJGL unterstützt Swing, läuft aber über ein AWT Canvas in der Standardinitialisierung. Diese schlanke Variante überzeugte und es wurde sich für LWJGL entschieden.

5.2 Framework

Das Basisprogramm wurde von Henning Wenke zur Verfügung gestellt [WEN]. Henning Wenke entwickelte im Rahmen seiner Masterarbeit einen objektorientierten Ansatz für eine OpenGL Entwicklungsumgebung in Java. Dieser Ansatz wurde übernommen und modifiziert. Im Laufe dieser Arbeit wurde das Framework stetig weiterentwickelt. Das Ziel des Framework ist es, die Funktionalität von OpenGL auf Klassen zu verteilen, wodurch sich die Entwicklung wesentlich einfacher gestaltet. Angemerkt sei hier, dass das Framework unter bestimmten Annahmen heraus entstanden ist. Diese Annahmen können in Abhängigkeit der Problemstellung als nützlich oder auch hinderlich betrachtet werden. Das Framework ist deshalb auf eine ganz bestimmte Art von OpenGL Anwendungen zugeschnitten und lässt in Bezug auf andere Problemstellungen weniger Spielraum. Um das Framework auf andere Problemstellungen zu erweitern, müssen möglicherweise Änderungen vorgenommen werden.

Der derzeitige Stand des Frameworks ist im folgenden UML Diagramm dargestellt (Abb.10). Alle Klassen, die mit dem Wort „Extended“ beginnen, stehen für konkrete Implementierungen der abstrakten Klassen und sind so im Framework nicht vorzufinden. Es werden nicht alle Klassen aufgeführt, sondern nur diejenigen, an denen die Struktur deutlich wird.

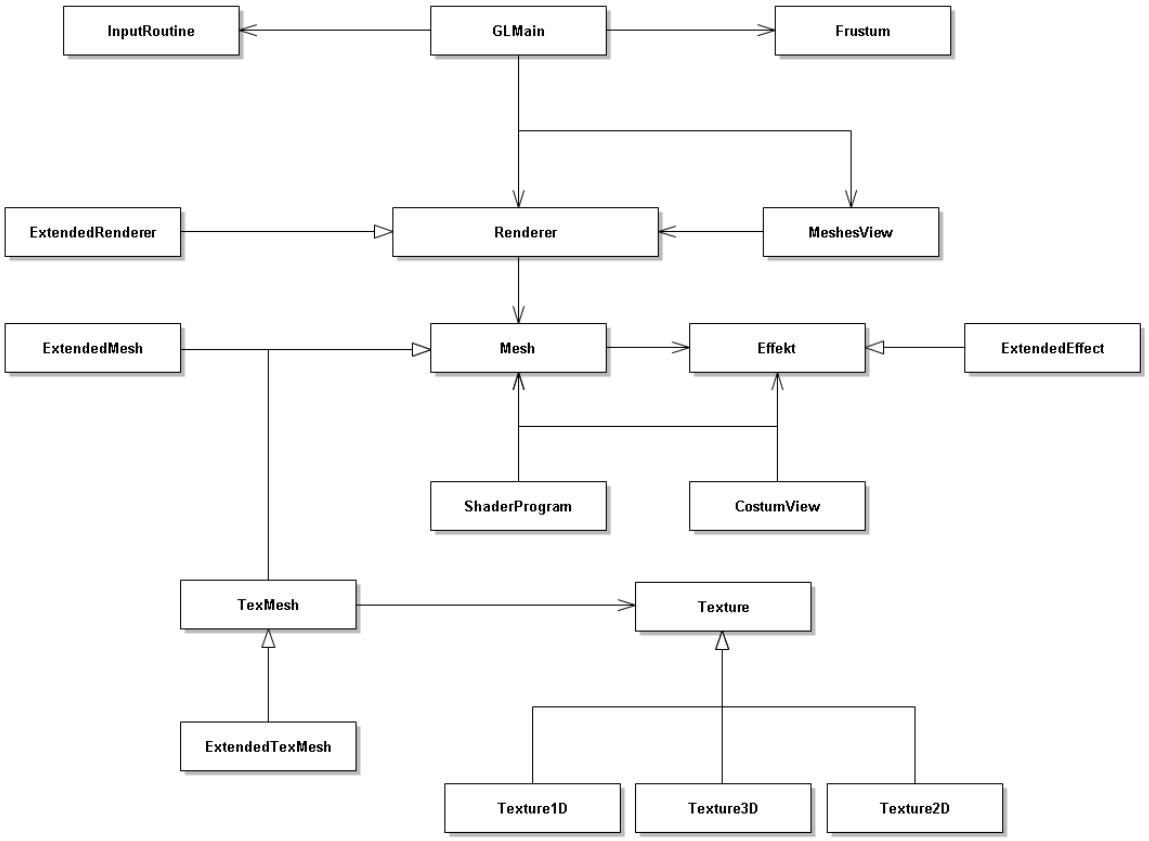

Abbildung 10: Framework

Es folgt eine kurze Erläuterung zu den relevanten Klassen:

Texture
Texturen werden mit der abstrakten Klasse $Texture$ verwaltet. Diese Klasse erzeugt eine OpenGL Textur Einheit mit allen grundlegenden Einstellungen. Jede erbende Klasse implementiert die Methode $initTexture()$, in der das Format, der Filter und die Größe definiert werden.

Mesh, TexMesh und ShaderProgram
Die abstrakte Klasse $Mesh$ kapselt alle OpenGL Befehle, die ausgeführt werden müssen, um die gewünschten Vertexdaten mit Hilfe von OpenGL in den Grafikspeicher zu kopieren. Jedes $Mesh$ kann sich selbst mit dem Aufruf der Methode $draw()$ zeichnen. Alle erbenden Klassen implementieren weiterhin die Methode $drawCustom()$. Hier können speziell für dieses $Mesh$ vorgesehene OpenGL Befehle ausgeführt werden. Ebenfalls besitzt jedes $Mesh$ ein Objekt der Klasse $ShaderProgram$. Ein $ShaderProgram$ kapselt eine OpenGL Shaderinstanz, welche mindestens aus einem Vertexshader besteht.

Die Klasse $TexMesh$ ermöglicht es, einem $Mesh$ Texturen hinzuzufügen. Hierbei wird gefordert, dass jede Implementierung der Klasse $TexMesh$ Texturkoordinaten bereitstellt.

GLMain und Renderer
Die Klasse $GLMain$ ist die Kernklasse, von der aus alle benötigten Klassen generiert und initialisiert werden. Um eine neue Szene zu erstellen, wird eine neue Klasse von der Klasse $Renderer$ abgeleitet. Eine Instanz der Klasse $Renderer$ besitzt eine Menge von $Mesh$ Objekten, die zu einer Szene gehören. Alle Implementationen der Klasse $Renderer$ müssen die Methode $render()$ implementieren. Die Methode $render()$ ruft die vom Benutzer festgelegte Reihenfolge auf, in der die Meshes gezeichnet werden sollen. Es wurde eine allgemeine Möglichkeit implementiert, alle Meshes eines Renderers zu zeichnen. Diese Methode hat den Namen $drawMeshes()$ und kann aus der Methode $render()$ direkt aufgerufen werden. Die Methode $drawMeshes()$ versucht, alle Meshes in der richtigen Reihenfolge zu zeichnen und fügt ebenfalls Effekte einzelner Meshes korrekt ein. Jedoch ist es in den meisten Fällen immer noch notwendig, die Methode $render()$ selbst zu implementieren, da sich bestimmte Anforderungen nicht so ohne weiteres mit der allgemeinen Implementation abbilden lassen. Ist eine neue Rendererklasse abgeleitet, kann diese instanziiert und zur Laufzeit dem $GLMain$ Objekt hinzugefügt werden. Die Klasse $GLMain$ verwaltet eine Liste von $Renderer$ Objekten, die vom Benutzer aktiviert werden können. Ist ein $Renderer$ aktiviert, wird in jedem Frame seine $draw()$ Methode aufgerufen. Die Methode $draw()$ ruft wiederum die implementierte $render()$ Methode auf. In der Methode $draw()$ werden vor jedem Aufruf der Methode $render()$ globale OpenGL Einstellungen geprüft und gegebenenfalls neu gesetzt.

InputRoutine
Die Klasse $InputRoutine$ ist für die Verarbeitung der vom Benutzer ausgelösten Eingaben zuständig.

Effect
Jedes $Mesh$ verfügt über eine Liste von Effekten. Diese Effekte werden gewöhnlich, nachdem das $Mesh$ gezeichnet wurde, auf das Ergebnis angewendet. Dieser Prozess wird auch $Postprocessing$ genannt. Ein typischer Effekt ist das Weichzeichnen eines Objektes, weil es sich möglicherweise mit einer sehr hohen Geschwindigkeit bewegt. In einer Implementierung der Klasse $Effekt$ werden konkrete Shaderprogramme geladen und auf ein gerendertes Ergebnis angewendet. Eine typische Implementierung eines Weichzeichners würde das gerenderte Ergebnis eines Meshes in Form einer Textur erhalten, dieses in einem weiteren Renderschritt mittels eines geeigneten Fragmentshaders weichzeichnen und in den Framebuffer schreiben. Des Weiteren verfügt jeder $Effect$ über eine grafische Repräsentation, die für die MeshesView benötigt wird.

MeshesView
Die Klasse $MeshesView$ ist die Hauptkontrolleinheit des Benutzers. Über dieses grafische Interface wird der aktuell aktive $Renderer$ der Klasse $GLMain$ dargestellt. Sobald ein Wechsel des Renderers geschieht, wird die $MeshesView$ aktualisiert und eine Auflistung aller Meshes des gerade aktiven Renderers wird angezeigt. Über Menüs können Einstellungen an vorhandenen Meshes vorgenommen werden. Die Grundeinstellungen eines Meshes werden in dafür vorgesehene Default Komponenten geladen. Ein $Mesh$ kann allerdings über eigene Viewkomponenten verfügen, die direkt in die $MeshesView$ geladen werden können (Abb.11).

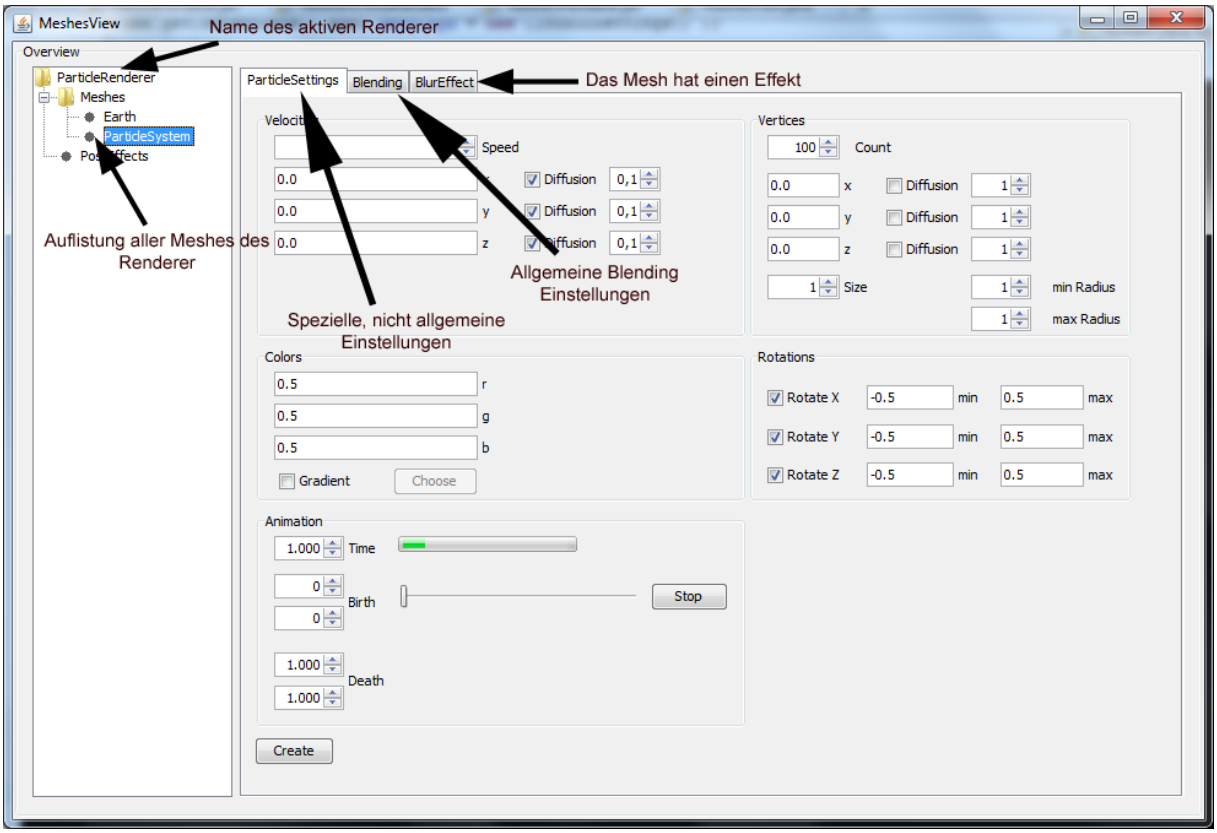

Abbildung 11: MeshesView im Detail

Frustum

Die Klasse *Frustum* ist zuständig für die Berechnung der $Model - View - Projection - Matrix$.

6 Noise

In der OpenGL Standard Rendering Pipeline werden Geometrien nach Definition präzise gerendert. Diese Tatsache macht es dem Programmierer einfach, geometrische Objekte exakt zu zeichnen. Geht es jedoch um die Darstellung von Objekten aus dem alltäglichen Leben, hat dies einen erheblichen Nachteil. Denn eben genau diese Objekte besitzen keine hundertprozentige "Präzision". Den Stamm eines Baumes durch einen länglichen Quader darzustellen, wäre eine Möglichkeit, jedoch hätte das Aussehen wohl wenig mit der Realität zu tun. Baumrinde hat Struktur, sie ist verwinkelt, hat Senken und Ausbeulungen. Sie ist eben nicht "perfekt", wie es ein durch acht Punkte definierter Quader ist. Einen fabrikneuen Fußball darzustellen ist nicht schwer. Ihm jedoch ansehen zu können, dass die Nachbarskinder mit ihm schon mehrere Jahre im Innenhof spielen, ist weitaus komplizierter. Auch Wolken sind ein gutes Beispiel, sie besitzen keine harten Kanten, sind transparent und können Licht in ihrem Volumen ablenken oder absorbieren. Eine Wolke kann auseinander driften und sich in zwei unabhängige Wolken teilen oder sich mit Anderen vereinigen.

Mit diesem Problem beschäftigte sich Ken Perlin in den 90er Jahren, als er für die Firma Magi an dem Film „Tron“ arbeitete. Er entwickelte daraufhin ein nach ihm benanntes Verfahren, in dem er eine Technik benutzte, die er $Noise$ nannte. Heute ist das Verfahren auch besser bekannt als $Perlin\ Noise$ [ROS]. Wie dieses Verfahren genau aussieht und implementiert werden kann, wird in einem späteren Abschnitt gezeigt. Zunächst einmal wird auf die Definition und die Konstruktion von Noise Funktionen eingegangen.

6.1 Definition

Wird heutzutage von Noise (Rauschen) gesprochen, so ist damit meistens das zufällige Auftreten von Bildpunkten (TV Schnee) oder Geräuschfrequenzen (Gewitter stört das Radio) gemeint. Diese Arten von Noise haben die Eigenschaft *echt zufällig* zu sein. Dies bedeutet, dass keine Möglichkeit besteht, das Ergebnis in irgendeiner Art und Weise zu reproduzieren. Aber genau diese Eigenschaft ist sehr wichtig in der Computer Grafik. Ein mit Noise modifiziertes Objekt sollte aus jeder Kameraeinstellung und in einer laufenden Animation gleich aussehen. *Echt zufälliges* Noise würde das Objekt in jedem Bild der Animation und jeder Kameraeinstellung anders aussehen lassen. [ROS]

Die Idee ist, eine Funktion zu generieren, die auf Basis ihrer Eingabe immer das gleiche Ergebnis liefert. Gleichzeitig sollte das Ergebnis ein zufälliges Muster aufweisen. Diese Eigenschaften werden auch unter dem Begriff *Pseudorandomness* zusammengefasst.

Eigenschaften der Funktion:

- Stetigkeit
- Reproduzierbarkeit
- Die Rückgabewerte liegen in einem vordefinierten Intervall $[a, b]$
- Das erzeugte Muster lässt keine offensichtlichen Wiederholungen erkennen
- Isotropie – Unabhängigkeit in der Richtung
- Erweiterbarkeit auf höhere Dimensionen
- Unabhängigkeit in der skalaren Ausprägung

Diese Eigenschaften führen zu einer Art von Funktionen, die eine Vielzahl von *pseudorandom* Elementen einer regulären Struktur hinzufügen können. [PER3]

6.2 1D Noise Funktionen

Mit Hilfe der obigen Definition wird nun ein Beispiel vorgestellt, um eine relativ einfache Noise Funktion für den eindimensionalen Fall zu generieren. Im Folgenden wird dieser Ansatz auf den zweidimensionalen und dreidimensionalen Fall erweitert.

Im ersten Schritt wird jedem ganzzahligen Wert der x-Achse eine vorher berechnete Zufallszahl aus dem Intervall $[-1, 1]$ zugeordnet (Abb.12). Da das Ziel eine stetige Noise Funktion ist, wird im nächsten Schritt zwischen den diskreten Werten interpoliert. Die Qualität der Interpolation spielt eine zentrale Rolle in der Erzeugung, weil sie sich stark in dem endgültigen Ergebnis widerspiegelt. Aus diesem Grund wird auf lineare Interpolation verzichtet und eine kubische Interpolation bevorzugt, denn diese liefert weiche Übergänge zwischen den Werten (Abb.13).

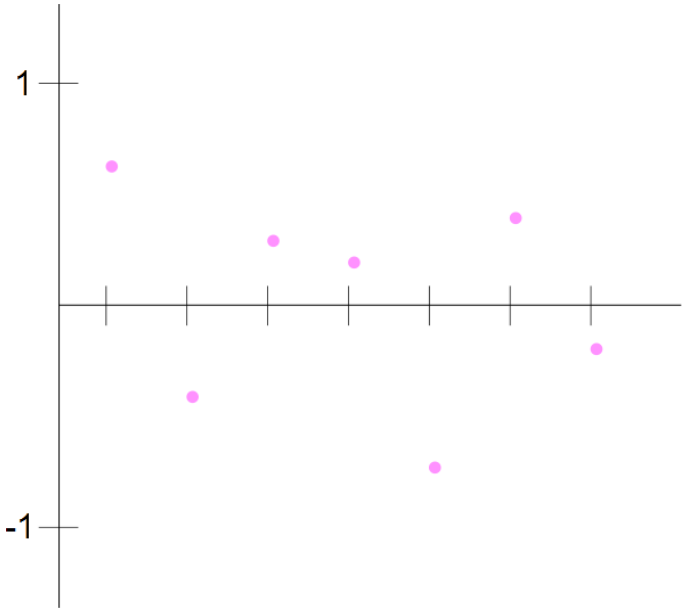

Abbildung 12: Diskrete Noise Funktion

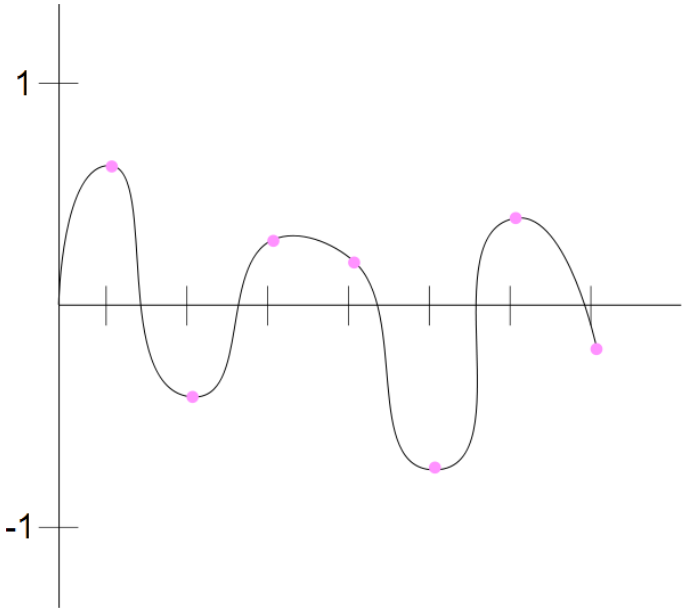

Abbildung 13: Stetige Noise Funktion

Durch Modifikation der Frequenzen und Amplituden der Funktionen, lassen sich viele verschiedene Noise Funktionen generieren. Abbildung 14 stellt drei verschiedene Noise Funktionen dar. Hierbei wurde jeweils die Frequenz verdoppelt und die Amplitude halbiert.
Es fällt auf, dass spezifische Details der Funktionen dichter und kleiner werden, je höher die Frequenz und kleiner die Amplitude gewählt wird.

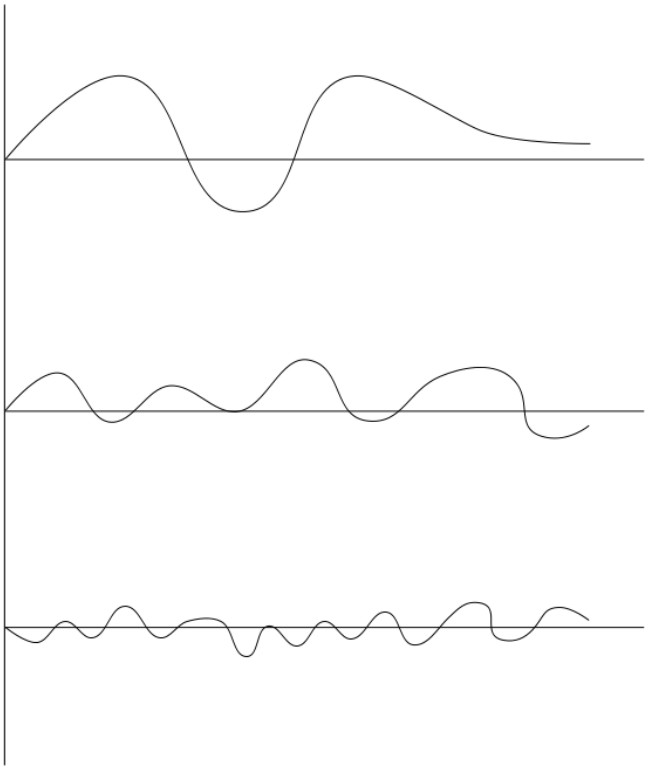

Abbildung 14: Verschiedene Frequenzen und Amplituden

Für sich betrachtet sehen die Funktionen nicht besonders ansprechend aus. Wird stattdessen die Summe der Funktionen gebildet, sieht das Ergebnis wesentlich interessanter aus. Abbildung 15 veranschaulicht die Summe der Funktionen aus Abbildung 14.

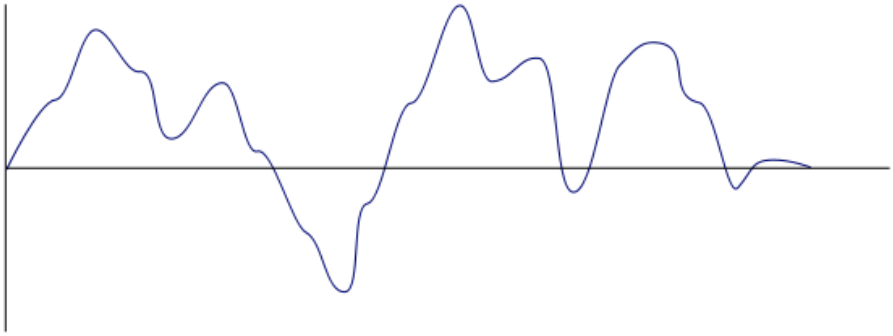

Abbildung 15: Verschiedene Frequenzen und Amplituden aufsummiert

Die endgültige Funktion zeigt Details in allen Größen (Abb.15). Wobei Funktionen mit geringeren Frequenzen und höheren Amplituden die grobe Form der endgültigen Funktion vorgeben. Hingegen generieren kleinere Amplituden und höhere Frequenzen feinere Details. Die Technik, die Frequenz zu verdoppeln und die Amplitude zu halbieren, ist auch unter dem Namen $1/f - Noise$ genannt. [EBE]

6.3 Mehrdimensionale Noise Funktionen

Die Erweiterung auf die zweidimensionale Ebene geschieht durch ein NxN Gitter. Wie im eindimensionalen Fall werden den Gitterpunkten gleichverteilte Zufallsvariablen zugeordnet. Dieser Ansatz wird auch $Lattice\ Noise$ genannt. $Lattice\ Noise$ ist relativ einfach aufgebaut, effizient und liefert gute Ergebnisse. [EBE]

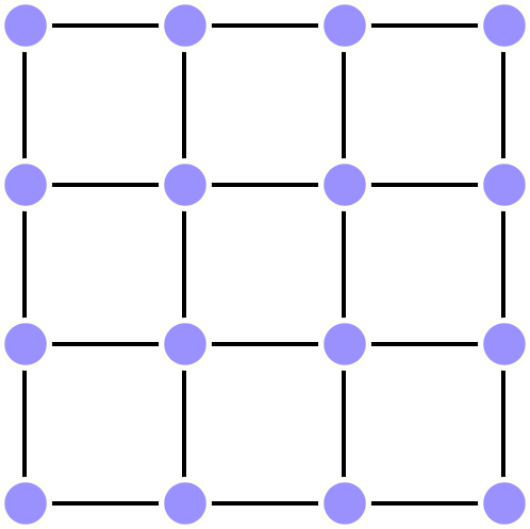

Abbildung 16: NxN Gitter

Jeder Punkt in der Abbildung 16 steht für eine ganzzahlige Position des NxN Gitters. Um eine stetige Funktion zu erhalten, wird auch hier wieder interpoliert. Abbildung 17 veranschaulicht die Interpolation in der Ebene. Der Wert an einer beliebigen Position im Gitter resultiert aus den Einflüssen der vier ganzzahligen Nachbarn. Hierbei werden drei Interpolationen benötigt. Die ersten zwei Interpolationen berechnen zwei Zwischenwerte. Mit den berechneten Zwischenwerten lässt sich eine weitere Interpolation durchführen, welche das endgültige Ergebnis liefert. Diese Art der Interpolation ist auch unter dem Namen Bilineare Interpolation bekannt. [WIK3]

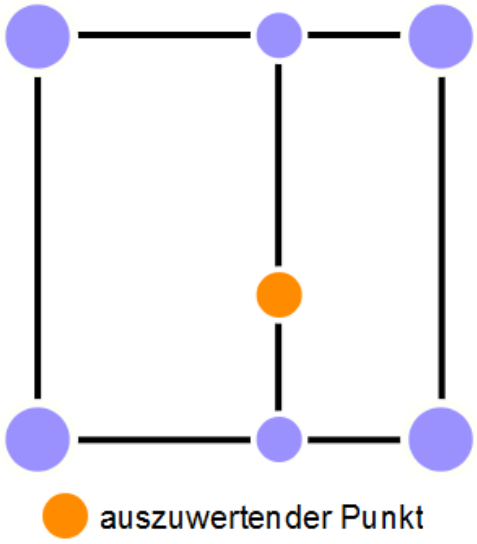

Abbildung 17: Bilineare Interpolation

Abbildung 18 zeigt eine nach Perlin generierte 2D Noise Textur. Die Textur wird initial auf der CPU (Central Processing Unit) generiert und mit OpenGL dargestellt.

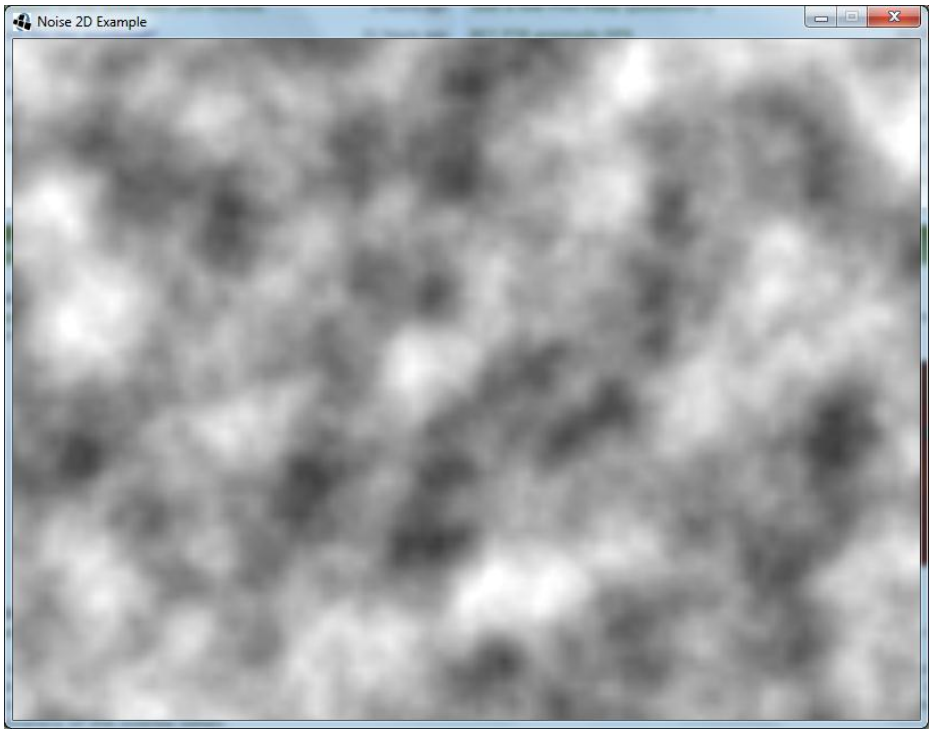

Abbildung 18: 2D Noise Textur, Auflösung 128 * 128 Pixel

3D Noise

Eine Erweiterung des 2D Noise Algorithmus um eine weitere Dimension entspricht dem 3D Noise. 3D Noise zu visualisieren, gestaltet sich als etwas schwieriger als 2D Noise. Abbildung 18 würde einem Schnitt durch eine 3D Noise Textur entsprechen. Nach oben und unten wären weitere Schnitte mit stetigen Übergängen vorhanden (Abb. 19).

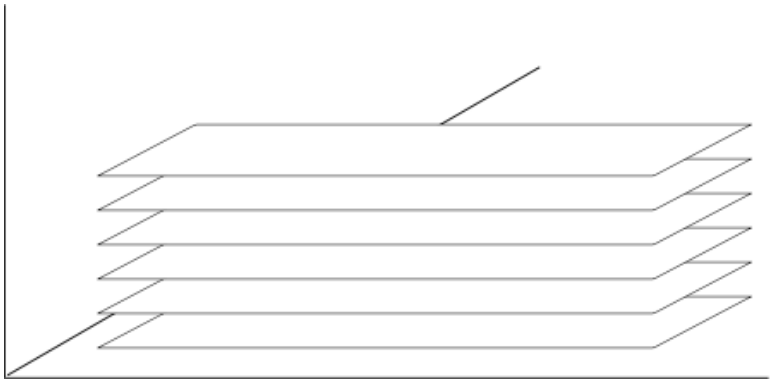

Abbildung 19: Aufbau 3D Noise

Wie bereits im Abschnitt über 2D Noise behandelt, wird auch hier wieder auf ein Gitter basierendes Verfahren zurückgegriffen. Da es sich um 3D Noise handelt, besteht das Gitter aus $NxNxN$ Punkten. Der Raum wird in gleichgroße Quadrate zerlegt, deren Randpunkte jeweils einer ganzzahligen Position im dreidimensionalen Raum entsprechen. [PER2]

Für die Auswertung eines Punktes innerhalb eines Quaders wird eine Trilineare Interpolation angewendet [WIK4]. Hierzu werden sieben Interpolationen benötigt. Vier Interpolationen in x-Richtung, zwei Interpolationen in y-Richtung und eine Interpolation in z-Richtung (Abb.20).

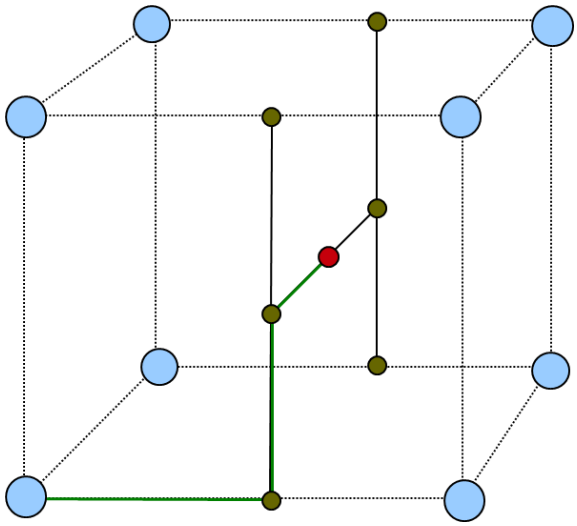

Abbildung 20: Trilineare Interpolation

7 3D Noise - Visualisierung

Die Visualisierung einer dreidimensionalen Noise Textur erfordert einen Rendering Ansatz, der nicht nur die Oberfläche, sondern auch das Volumen eines Objektes berücksichtigt. Werden optische Eigenschaften traditionell in einem Punkt der Geometrie-Oberfläche ermittelt, berücksichtigen Ansätze zur Darstellung des Volumens auch den inneren Bereich für die Visualisierung. Ein Ansatz ist, das Volumen in einer 3D Textur zu speichern und für jeden Bildpunkt einen Strahl in das Volumen zu schießen. Der Strahl wertet den Farbwert an diskreten Punkten innerhalb des Volumens aus und berechnet den resultierenden Farbwert durch Anwendung von Alpha Blending. Es muss also zusätzlich für jeden RGB Farbwert noch eine Alpha Komponente abgespeichert werden. [HAD]

7.1 BoundingBox

Ein möglicher Ansatz zum Rendern von dreidimensionalen Texturen ist die $BoundingBox$ Methode. Die Idee ist, das zu rendernde Volumen mit Hilfe eines Quaders ($BoundingBox$) aufzuspannen und zu begrenzen (Abb.21a). Im nächsten Schritt wird das Volumen durch weitere kleinere Quader innerhalb der $BoundingBox$ zerlegt. Das Ergebnis ist ein dreidimensionales Grid, bestehend aus Quadern mit gleichen Abmessungen (Abb. 21b).

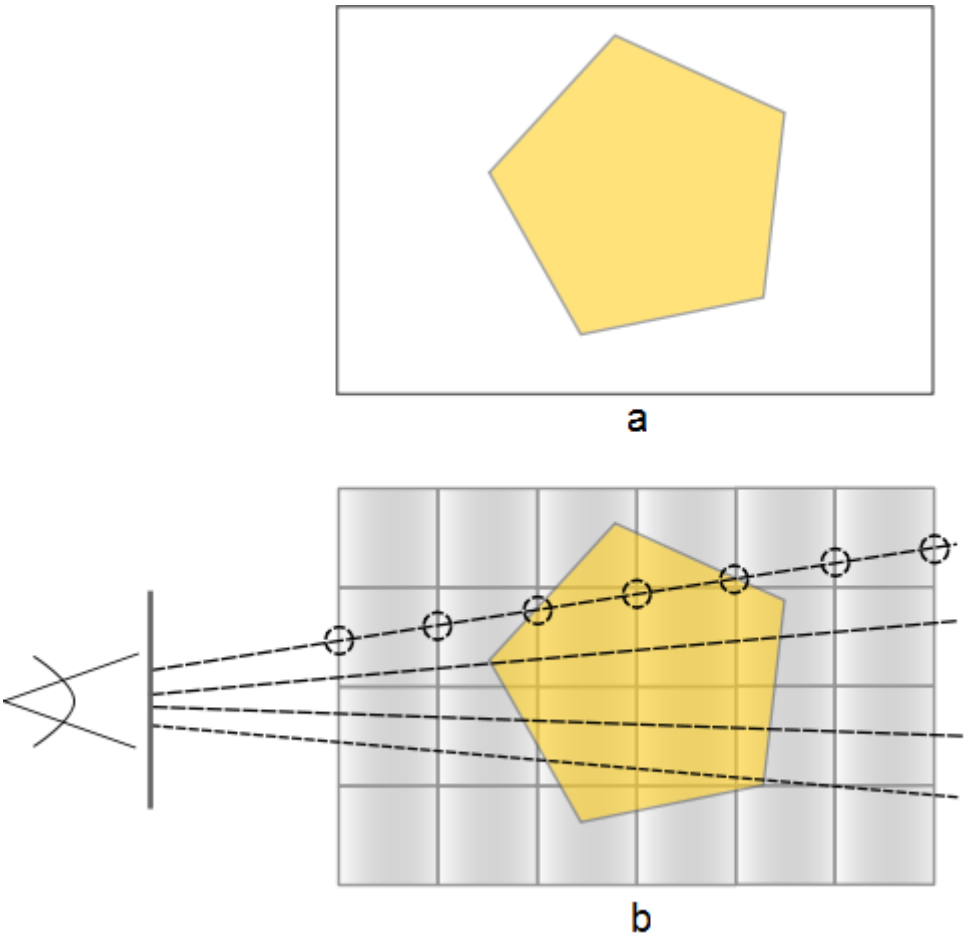

Abbildung 21: BoundingBox (a); verfeinerte BoundingBox(b), Kreise stehen für Punkte zur Farbauswertung

Jeder dieser Quader repräsentiert einen diskreten Farbwert, der zur endgültigen Farbberechnung benötigt wird. Die Feinheit dieser Zerlegung zu bestimmen, ist ein wichtiger Bestandteil dieser Methode. Wird das Grid zu grob gewählt, werden möglicherweise entscheidende Änderungen übersprungen. Andererseits kann eine zu fein gewählte Auflösung dazu führen, dass mehrere Punkte in einem Intervall der Auswertung den gleichen Farbwert liefern. Dies führt zu zusätzlichem Rechenaufwand, der sich im Ergebnis nicht widerspiegelt. Der Verfeinerungsgrad des Grids ist von der Farbänderungsrate des zu rendernden Volumens abhängig. Die Implementierung bestimmt wiederum, an welchen Positionen die Farbwerte ausgewertet werden. In Abbildung 21 wird der Farbwert jeweils an der Seite des Quaders ausgewertet, der von dem Strahl als erstes durchstoßen wird (schwarze Kreise). Ein offensichtliches Problem ist die Behandlung von leeren Quadern innerhalb der $BoundingBox$ (Abb.21). Leere Quader erhöhen die Laufzeit unnötig, da sie in die Berechnung mit einfließen. Um dieses Problem zu beheben, gibt es verschiedene Ansätze, die aber hier nicht weiter erläutert werden. In dieser Arbeit wird mit Texturen gearbeitet, die den gesamten Bereich der $BoundingBox$ ausfüllen, wodurch das genannte Problem nicht relevant ist.
Anzumerken ist, dass die $BoundingBox$ Methode nicht auf die Visualisierung von 3D Noise beschränkt ist, sondern beliebige dreidimensionale Rasterdaten darstellen kann.

7.1.1 Implementierung

In der konkreten Implementierung werden in Abhängigkeit vom Feinheitsgrad der $BoundingBox$ Ebenen generiert. Jede Ebene repräsentiert einen Querschnitt des Volumens. Diese Ebenen werden immer orthogonal zur Kamera rotiert, um sicherzustellen, dass im korrekten Winkel durch das Volumen gerendert wird. Die Rotation ergibt sich aus der aktuellen Position der Ebene und der Position der Kamera. Des Weiteren benötigen alle Ebenen eine sich unterscheidende z-Komponente, über die die Tiefe der aktuellen Ebene im Volumen bestimmt werden kann. Die Tiefe der Ebenen ist in diesem Fall normiert, um diese für den Texturzugriff im Fragmentshader zu nutzen. Um sicherzustellen, dass beim Alpha Blending keine Fehler entstehen, wird die Ebene mit der größten z-Komponente als erstes gerendert, die mit der kleinsten z-Komponente als letztes. Es wird also, wie üblich, von „hinten“ nach „vorne“ durch das Volumen gerendert. Sind die Positionen der Ebenen normiert, befinden sich die Ebenen innerhalb des Einheitswürfels. Ist dies der Fall, kann direkt über die Vertexkoordinaten ein Zugriff auf die 3D Textur im Fragmentshader stattfinden. Ist dies nicht der Fall, müssen die Texturkoordinaten für die x und y Dimensionen angegeben werden.

```
#version 140
uniform sampler3D noise;
in vec2 texCoords;
in vec3 vPos;
out vec4 fragColor;
void main() {
      fragColor = texture( noise, vec3( texCoords, vPos.z );
}
```

Abbildung 22: BoundingBox Fragmentshader

Der Fragmentshader liest mit Hilfe der übergebenen Texturkoordinaten der Ebene und der normierten z-Position die Farbe aus der 3D Textur (Abb. 22). Es wird davon ausgegangen, dass die übergebene Textur im Format $RGBA$ vorliegt. Bevor die Ebenen gerendert werden können, muss das Alpha Blending aktiviert werden. Dies geschieht über den Aufruf der Blendfunktion mit folgenden Parametern:

$$glBlendFunc(GL\_SRC\_ALPHA,\ GL\_ONE\_MINUS\_SRC\_ALPHA).$$

7.1.2 Ergebnis

Abbildung 23 zeigt eine 3D Noise Textur, die mit Hilfe der $BoundingBox$ Methode gerendert wurde. Es wurden 100 Ebenen übereinander gelegt und mittels Alpha Blending verrechnet. Die 3D Noise Textur hat eine Auflösung von $128 * 128 * 128$ Farbwerten. Die Startfrequenz beträgt 3.4 und die Startamplitude 0.64. In diesem Beispiel wurde eine vierstufige $1/f$ - Noisefunktion verwendet.

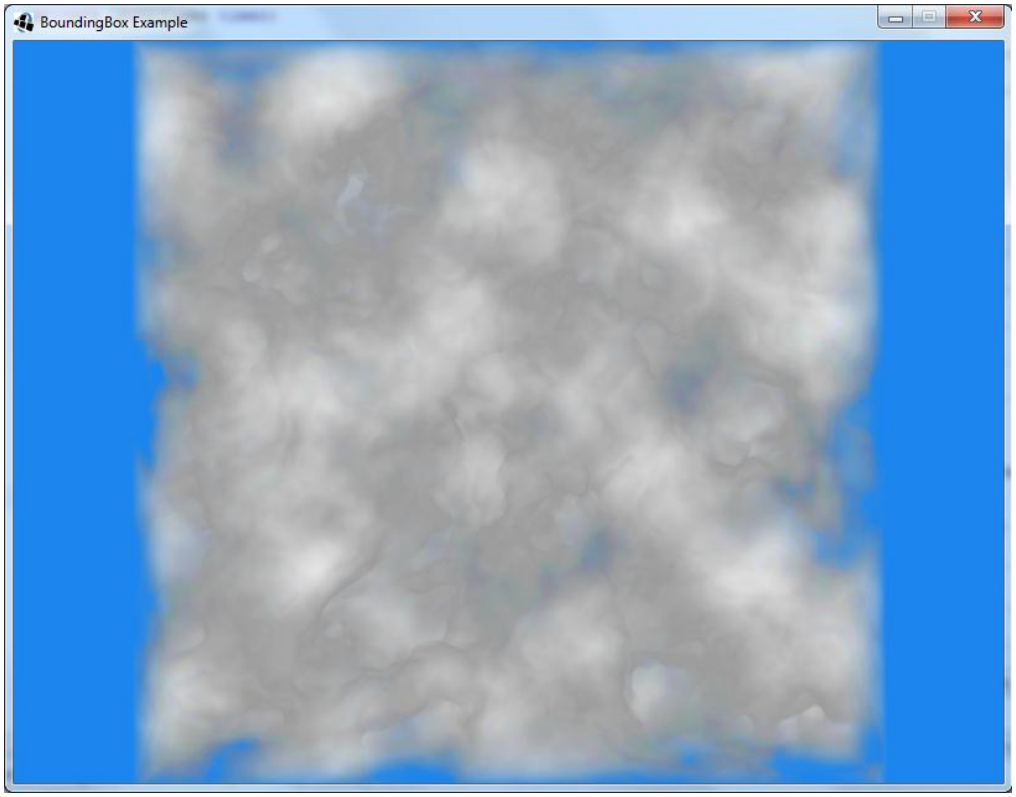

Abbildung 23: 3D Noise visualisiert mit der BoundingBox Methode

7.2 BoundingSphere

In vielen Fällen ist es sinnvoll, eine andere Geometrie zu wählen, auf die die 3D Textur projiziert wird. Für diese Arbeit ist vorgesehen, dass die Bewölkung auf der Erdkugel angezeigt werden kann. In der Implementierung der $BoundingBox$ Methode wurden Querschnitte der 3D Textur auf Ebenen projiziert, die dann wiederum übereinander geblendet wurden. Die naheliegende Idee ist, die Ebenen der $BoundingBox$ durch Kugeln zu ersetzen und auf jede Kugel eine Schicht der 3D Textur zu projizieren.

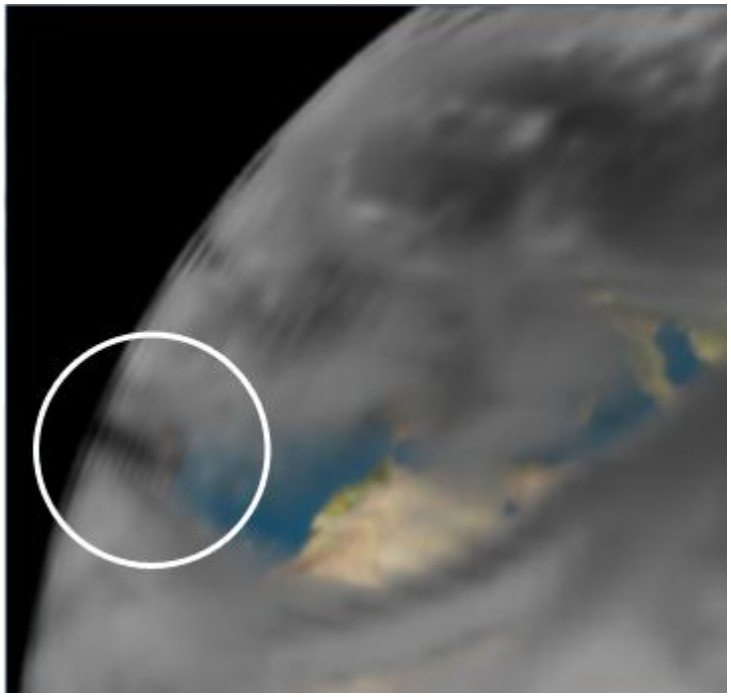

Abbildung 24: BoundingSphere, fehlerhafte Darstellung an den Rändern

Abbildung 24 macht den erheblichen Nachteil dieser Methode deutlich. An den Randstellen entstehen unschöne Effekte, weil die Kugeln der $BoundingSphere$ klar erkennbare Streifen verursachen. Dies ist auf die nicht senkrecht zur Kamera ausgerichtete Kugeloberfläche zurückzuführen (Abb. 25).

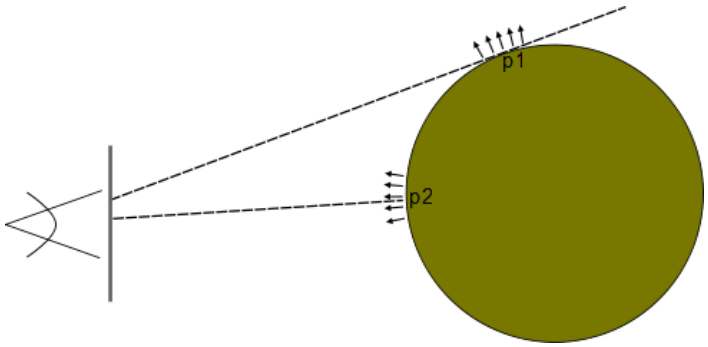

Abbildung 25: Fragments, deren Normale nahezu orthogonal zur Kamera stehen (p1), generieren unschöne Effekte

Aus diesem Problem heraus entstand ein modifizierter Ansatz der $BoundingSphere$. Dieser basiert immer noch auf Kugeln, löst jedoch das oben genannte Problem.

7.3 Advanced BoundingSphere: Bounding VoxelSphere

In der $Bounding\ VoxleSphere$ Methode wird das zu rendernde Volumen durch zwei Kugeln begrenzt. Eine Innere Kugel $k1$ und eine äußere Kugel $k2$. Das Begrenzungsvolumen ergibt sich aus der Differenz der Volumina von Kugel $k1$ und $k2$. Im nächsten Schritt wird das entstandene Begrenzungsvolumen weiter durch kleinere Quader zerlegt. Diese Quader sind für die Abtastung der zu rendernden 3D Textur vorgesehen (Abb. 26).

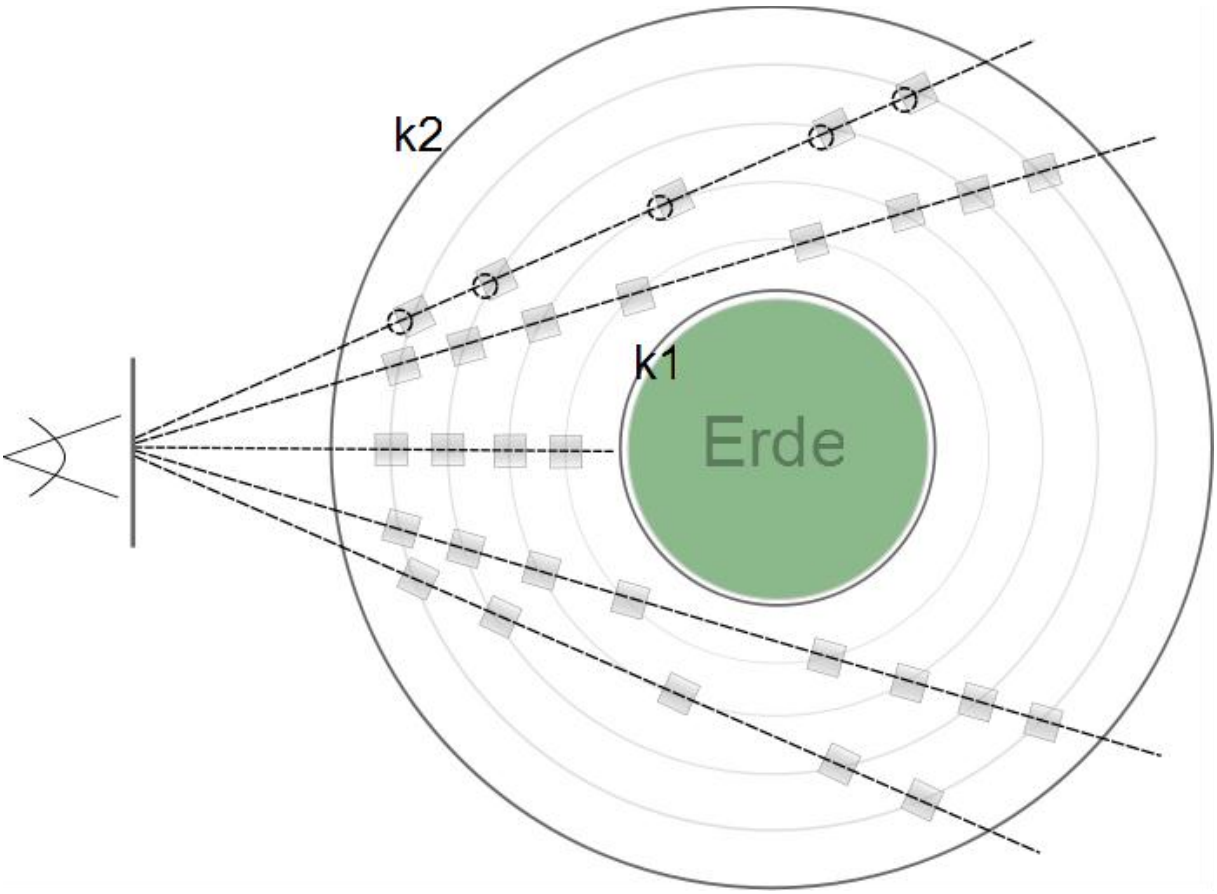

Abbildung 26: Bounding VoxelSphere

Im Gegensatz zur $BoundingSphere$, ist die Geometrie, auf die das Volumen projiziert wird, immer senkrecht zur Kamera ausgerichtet (Abb.26).

7.3.1 Implementierung

Wie in der $BoundingBox$ Methode, ist es auch bei der Implementierung der $Bounding\ VoxelSphere$ Methode erforderlich, Ebenen innerhalb des Begrenzungsvolumens zu platzieren. Es werden jedoch wesentlich mehr und kleinere Ebenen generiert. Auf diese Ebenen werden anschließend Farbwerte der 3D Textur projiziert und übereinander geblendet.

Der Durchmesser der $Bounding\ VoxelSphere$ ergibt sich aus der Differenz der vordefinierten Radien der Kugeln $k1$ und $k2$ ($r = r2 - r1$). Die Grundidee des Algorithmus

ist es, auf Basis von Kugelkoordinaten, Ebenen innerhalb des Volumens zu generieren. Hierbei werden für eine Anzahl von Radien aus dem Intervall $[r1, r2]$ Kugeloberflächen durchlaufen und Ebenen in jedem Punkt eines Intervalls der Kugel generiert. Das Ergebnis sind gleichverteilte Ebenen innerhalb des $Bounding\ VoxelSphere$ Volumens. Die Gleichverteilung würde jedoch beim späteren Rendern zu Problemen führen. Der Benutzer würde eine „Struktur" innerhalb der $Bounding\ VoxelSphere$ sofort erkennen (Abb. 30). Um dies zu vermeiden, werden die Positionen jeder Ebene leicht variiert (Abb. 27).

$$x = \mathrm{r} * \sin(\theta) * \cos(\varphi)$$
$$y = \mathrm{r} * \sin(\theta) * \sin(\varphi)$$
$$z = \mathrm{r} * \cos(\theta)$$
$$(x_f, y_f, z_f) = (x + a(x), \quad y + a(y), \quad z + a(z))$$

Abbildung 27: Modifizierte parametrisierte Kugelkoordinaten, a(.) addiert jeweils zu jeder Komponente einen Rauschterm, um diese zu variieren.

Durch Addition des Rauschterms $(a(.))$ wird die endgültige Position der Ebene in einer Umgebung um den Mittelpunkt gestreut (Abb. 30).

Der Feinheitsgrad der $Bounding\ VoxelSphere$ ist hierbei von zwei Parametern abhängig:

- Auf der Strecke von $r1$ nach $r2$ ergeben sich mögliche Abtastradien für die Berechnung der Kugelkoordinaten. Je kleiner das Abtastintervall gewählt wird, desto öfter werden Kugeln durchlaufen und Ebenen-Positionen gesetzt.
- Die Auflösung der Kugeln kann über die Erhöhung der Winkel Phi und $Theta$ nach jeder Berechnung gesteuert werden.

Abbildung 28 zeigt den Feinheitsgad einer $Bounding\ VoxelSphere$. An jedem Schnittpunkt der horizontalen und vertikalen Linien wird durch den Algorithmus eine Ebene platziert. Diese Ebene hat ihren Mittelpunkt genau in dem berechneten Punkt und richtet sich, wie in der $BoundingBox$ Methode beschrieben, orthogonal zur Kamera aus. Die Größe der Ebene ist von der gewählten Auflösung der $Bounding\ VoxelSphere$ und vom Abstand der Kamera abhängig. Allgemein gilt, je weiter die Kamera entfernt ist, umso größer werden die Ebenen und umso kleiner ist die Auflösung der Kugeln. Die Anpassung der Ebenen-Größe und die Auflösung der $BoundingSphere$ wird nach der Initialisierung der Default Werte in den entsprechenden Shader Programmen vorgenommen. Somit sind Änderungen der Auflösung und der Ebenen-Größe zur Laufzeit möglich.

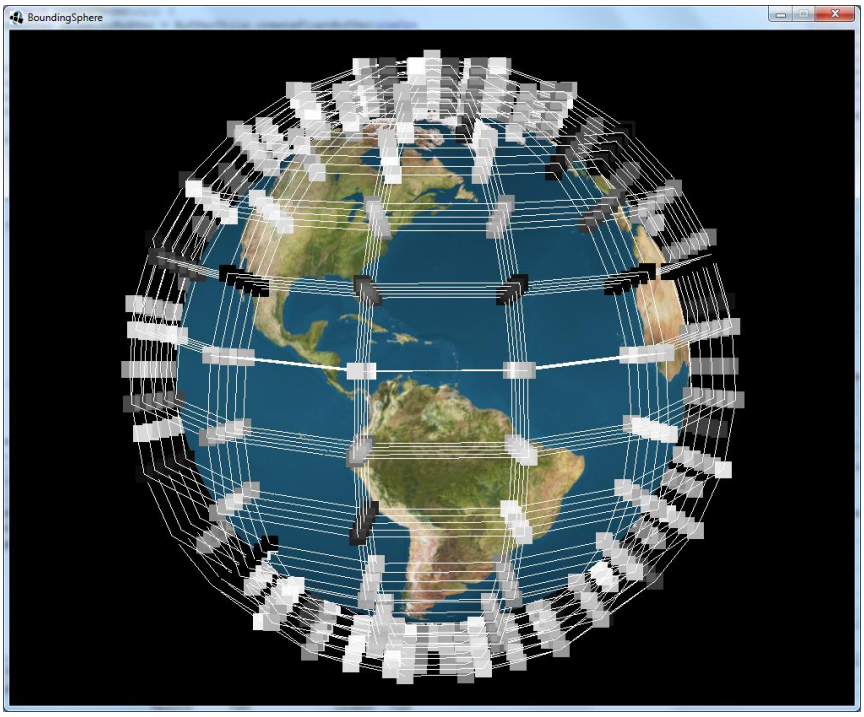

Abbildung 28: BoundingSphere, fünf Kugeln mit einer Auflösung von 15*15 Ebenen

Die Kugeln der *Bounding VoxelSphere* werden von innen nach außen generiert. Somit entstehen beim Blending keine Fehler, da die Ebenen genau in dieser Reihenfolge gezeichnet werden. Allerdings muss darauf geachtet werden, ob die Kamera von oben oder von unten auf die *Bounding VoxelSphere* gerichtet ist. Die *Bounding VoxelSphere* wird initial so aufgebaut, dass die Positionen auf den Oberflächen der Kugeln von unten nach oben generiert werden. Um Fehler beim Blending zu vermeiden, muss diese Reihenfolge umgekehrt werden, sobald die Kamera von unten nach oben gerichtet ist (Abb.29).

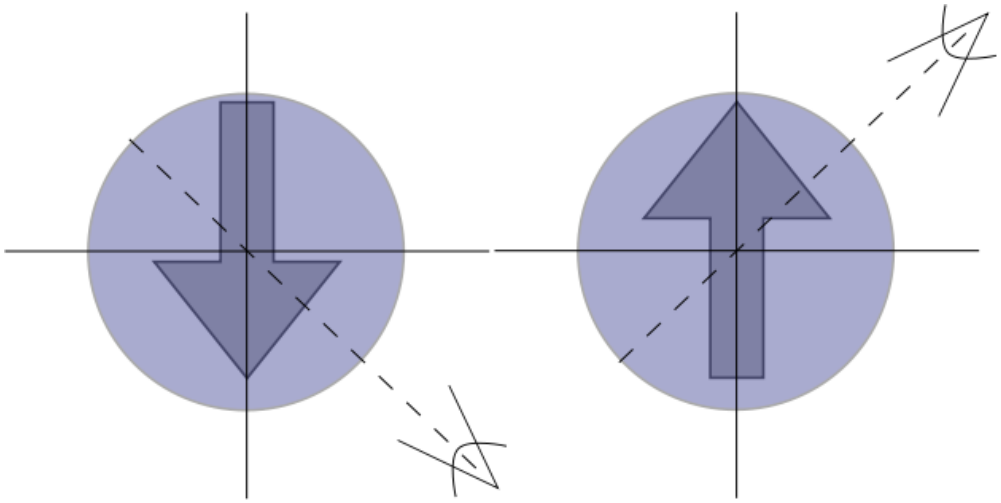

Abbildung 29: Umkehrung der Zeichenreihenfolge (Pfeil) in Abhängigkeit zur Kameraposition

In dieser Implementierung wird davon ausgegangen, dass sich in der Mitte der *Bounding VoxelSphere* ein fester, also nicht transparenter Körper befindet. Unerheblich ist, ob erst die hintere Seite oder erst die vordere Seite der *Bounding VoxelSphere* gezeichnet wird. Fragments, die hinter der Erdkugel liegen, fallen nicht ins Gewicht, wenn die Erdkugel zuerst gezeichnet wird. Die Tiefenwerte der Erdkugel verhindern, dass Fragments, die weiter entfernt sind, beim Blending berücksichtigt werden. Zu Problemen kommt es lediglich an den Randstellen, die nicht durch die Erde nach hinten hin abgedeckt werden (Abb.31). Diese Fehler sind jedoch minimal, wenn die *Bounding VoxelSphere* sehr nah an der Erdkugel liegt.

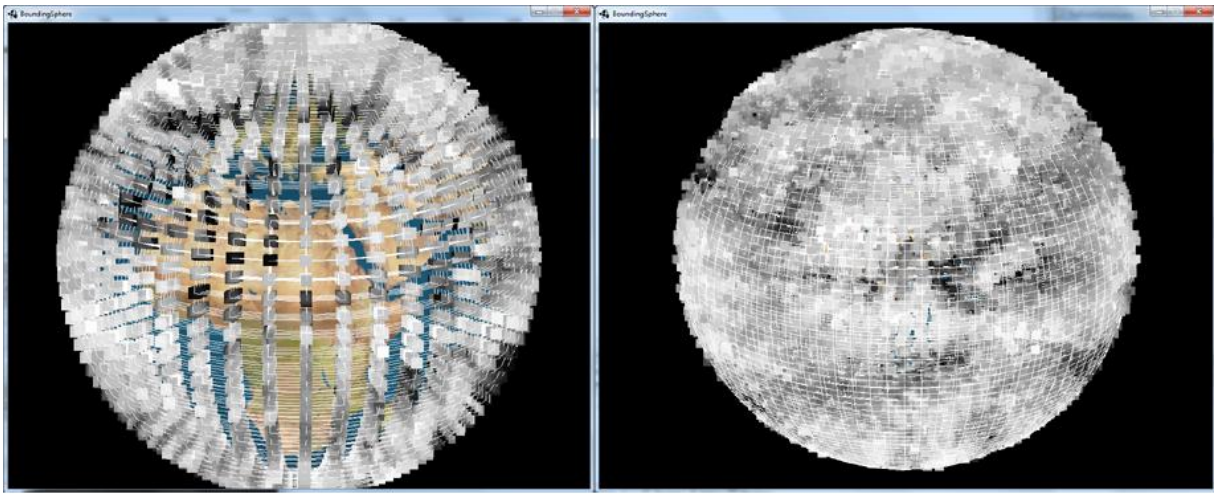

Abbildung 30: Bounding VoxelSphere ohne Rauschterm (links), mit Rauschterm (rechts)

Abbildung 31:Blending-Fehler

In dieser Implementierung ist das zu rendernde Volumen auf das vorgegebene Begrenzungsvolumen der *Bounding VoxelSphere* beschränkt. Ist das Volumen größer, werden möglicherweise Teile des Volumens nicht berücksichtigt. Es ist daher wichtig, dass das zu rendernde Volumen vollständig von dem Begrenzungsvolumen der *Bounding VoxelSphere* aufgespannt werden kann.

7.3.2 Ergebnis

In der Abbildung 32 wurde eine 3D Noise Textur mit Hilfe der *Bounding VoxelSphere* Methode gerendert.

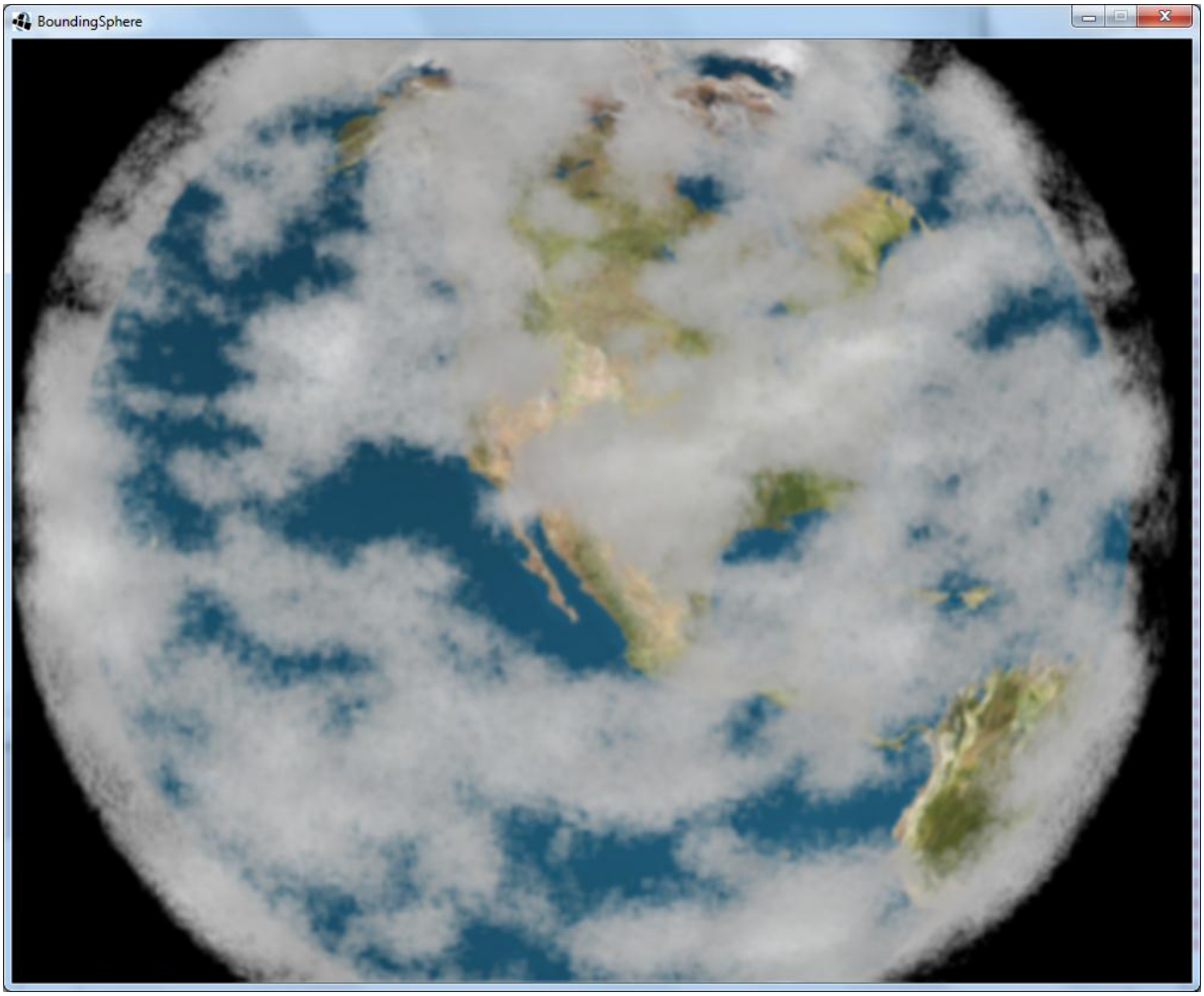

Abbildung 32: Bounding VoxelSphere, angewendet auf eine 3D Noise Textur; 10 Kugeln mit jeweils einer Auflösung von 1000*1000 Ebenen

8 2D Bewölkungsgrad und 3D Noise

Mit den vorgestellten Verfahren zur Visualisierung von 3D Texturen ist es möglich, eine dreidimensionale Modellierung des Bewölkungsgrades darzustellen. Ziel ist es, den zugrundeliegenden Bewölkungsgrad in Form einer 2D Textur mit Hilfe von 3D Noise in eine dreidimensionale Darstellung zu transformieren. Das Ergebnis wird in einer 3D Textur gespeichert und mit der $Bounding\ VoxelSphere$ Methode gerendert.

Der Ausgangspunkt ist der Bewölkungsgrad in Form einer zweidimensionalen Textur. Diese Textur wird später die grobe Form der Wolken bestimmen. Die Idee ist, ein grobes Wolkenkonstrukt zu erstellen und dieses mit Hilfe einer 3D Noise Textur zu verfeinern. Hierbei soll sich der Skalierungsgrad der Kameraposition anpassen. Je näher der Benutzer an die Erde heranzoomt, desto mehr Details sollen sichtbar werden.

Der Bewölkungsgrad wird auf jede Kugel der $Bounding\ VoxelSphere$ projiziert und mit Hilfe von 3D Noise leicht in seiner Ausprägung manipuliert. Dadurch entsteht ein dreidimensionaler Eindruck, dem nicht anzusehen ist, dass er aus einer einzigen zweidimensionalen Textur generiert wurde.

8.1 Implementierung

Der Detailgrad der gerenderten Bewölkung soll von der Entfernung der Kamera abhängig sein. Ist die Kamera weit entfernt, kann der Detailgrad relativ niedrig gewählt werden, um somit Rechenleistung zu sparen. Ist die Kamera sehr nahe, sollen alle Details für den Benutzer sichtbar werden. Die $Bounding\ VoxelSphere$ Methode ist bereits so implementiert, dass sie, in Abhängigkeit von der Entfernung zur Kamera, die Größen der Ebenen und die Auflösung der Kugeln anpasst, auf die die Textur projiziert wird. Ebenfalls kann mit Hilfe der Noise Textur der Detailgrad des Volumens noch weiter verfeinert werden. Hierbei wird mehrfach in die Textur hineingezoomt und mehrere Zoomstufen zueinander addiert (Abb.33 und Abb.34).

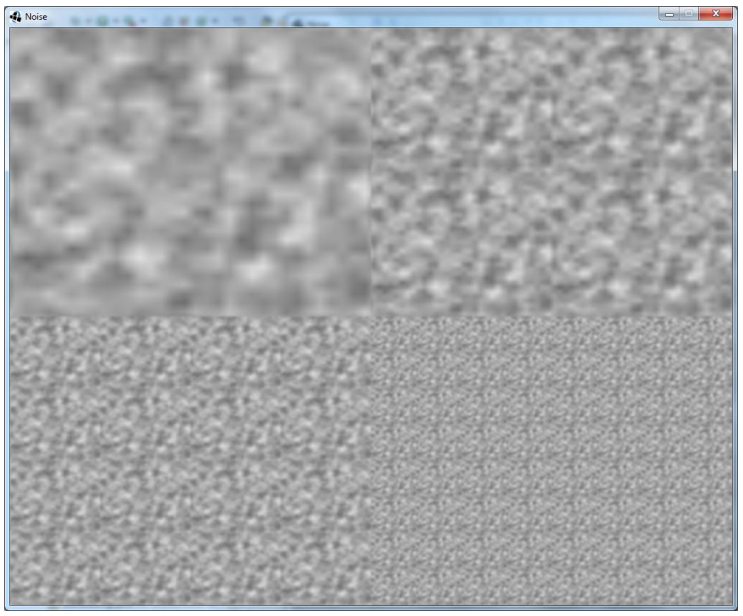

Abbildung 33: Vier Skalierungen einer Noise Textur (32*32 Pixel), 2^n Skalierungen ($0 <= n <= 3$)

In diesem Beispiel liegt die Größe der Noise Textur bei $32 * 32$ Pixel und ist somit relativ klein. Betrachtet man nur die Ausgangstextur im linken oberen Quadranten, lassen sich nicht sehr viele Details erkennen (Abb.33). In den weiteren Quadranten ist jeweils die nächste Skalierungsstufe zu erkennen. Abbildung 34 zeigt die Summe der skalierten Noise Texturen. Das Resultat lässt wesentlich mehr Details erkennen und zeigt, wie gut die Ergebnisse dieses Verfahrens sein können. Außerdem geschieht die Berechnung vollständig auf der Grafikkarte, in einem dafür programmierten Shader. Sämtliche Parameter können zur Laufzeit manipuliert werden und passen das Ergebnis entsprechend an. Je kleiner die Entfernung zur Kamera, desto höher wird der Skalierungsgrad der einzelnen Texturen gewählt, die im darauffolgenden Schritt addiert werden. Somit ist ein beliebiger Skalierungsgrad in Abhängigkeit zur Kameraentfernung möglich.

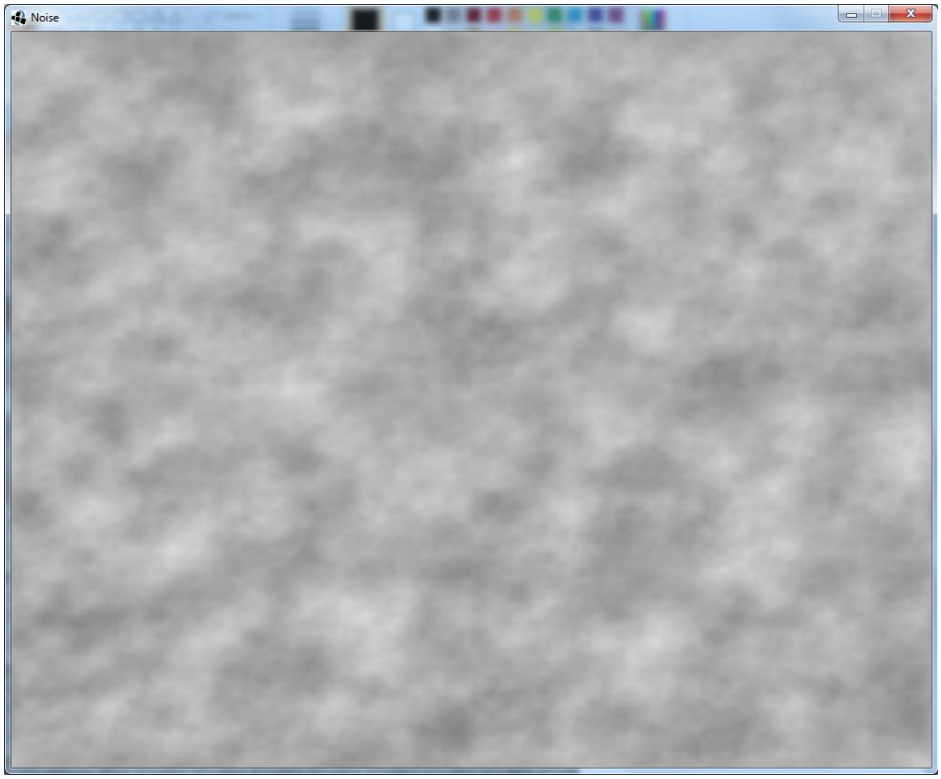

Abbildung 34: Summe der skalierten Noise Texturen

Ein Problem bei dem oben beschriebenen Ansatz zur dynamischen Erstellung der Noise Texturen ist die statische Generierung in Bezug auf die Zeit. Für die spätere Animation ist es jedoch wichtig, dass auch der Zeitpunkt einen Einfluss auf die Generierung der Noise Textur hat. Würde dies nicht geschehen, sieht der Benutzer statische Artefakte auf dem Bildschirm, die auf die generierte Noise Textur zurückzuführen wären. Um dieses Problem zu lösen, wird die z-Koordinate zum Zugriff der Noise Textur um den Faktor $1 + timeStep$ skaliert. Es wird also in jedem Zeitschritt eine andere Ebene aus der 3D Noise Textur gewählt. Die Variable $timeStep$ wird automatisch inkrementiert, sobald ein weiterer Animationsschritt erfolgt. Somit verschwinden statische Textur Merkmale und der Benutzer sieht eine stetige, dynamische Animation der generierten Bilder.

Die Generierung der Texturen für jede Kugel der $BoundingSphere$ geschieht in einem Shaderprogramm. Die Kugeln der $Bounding\,VoxelSphere$ besitzen jeweils eine Tiefeninformation, die aus dem maximalen Radius der $BoundingSphere$ und des Radius der aktuellen Kugel berechnet wird. Dieser Wert wird, in Verbindung mit den Texturkoordinaten der Kugel, für das Auslesen der 3D Noise Textur genutzt. Somit wird jeder Kugel der $Bounding\,VoxelSphere$ eine 2D Noise Textur aus der 3D Noise Textur zugewiesen.

Wie bereits erwähnt, soll der Bewölkungsgrad die grobe Form der Wolken vorgeben. Dies erfolgt, indem jede generierte 2D Noise Textur in einem letzten Schritt auf Basis dieses Grades transformiert wird. Eine Möglichkeit ist, den aktuellen Farbwert des Grades der Bewölkung und den berechneten Farbwert der 2D Noise Textur auf Basis eben dieses Grades zu interpolieren.

$$finalValue = farbWertBew * (1 - alphaBew) + alphaBew * farbWertNoise$$

Der resultierende Farbwert $finalValue$ ist also zu 100% der Farbwert der Noise Textur, wenn der Bewölkungsgrad ebenfalls 100% erreicht. Ist keine Bewölkung vorhanden (0%), setzt sich zu 100% der Farbwert des Bewölkungsgrades durch, welcher in diesem Fall auf Schwarz gesetzt ist und aufgrund des Alphawertes von 0 keinen Einfluss hat. Der endgültige Alphawert, der für das Alphablending der $Bounding\,VoxelSphere$ benötigt wird, ist der Alphawert des Bewölkungsgrades.

```
#version 140
...
uniform sampler3D noise3d;
out vec4 fragColor;
float noiseTexture( float scale, vec2 tex, float level ) {
      return texture(noise3d, vec3(tex * scale, level)).r;
}
void main() {
      ...
 float v  = noiseTexture( scaleTexture0, texCo, z*(1+timeStep) );
       v += noiseTexture( scaleTexture1, texCo, z*(1+timeStep) );
       v += noiseTexture( scaleTexture2, texCo, z*(1+timeStep) );
       v += noiseTexture( scaleTexture3, texCo, z*(1+timeStep) );
       v /= 4.0;
 float i = mix(cloudGradient, v, cloudGradientAlpha );
 fragColor = vec4( vec3( i ) , cloudGradientAlpha );
}
```

Abbildung 35: Fragmentshader der Bounding VoxelSphere für die Berechnung und Darstellung des dreidimensionalen Bewölkungsgrades

Die genaue Berechnung des Farbwertes ist in Abbildung 35 zu sehen. Die 2D Noise Textur wird mit vier Aufrufen der Funktion $noiseTexture(float\ scale, vec2\ xy, float\ z)$ generiert. Die jeweils ermittelten Werte werden addiert und normiert. Die Variablen $scaleTexture[N]$ enthalten im Normalfall jeweils die $N-te$ Zweierpotenz. Die Variable $texCo$ enthält die Koordinaten für den Texturzugriff.

Das letzte Argument ist die um den Faktor$(1 + timeStep)$ skalierte z-Texturkoordinate, für den Zugriff auf die 3D Texture $(noise3d)$. Im darauffolgenden Schritt wird die Interpolation, mit Hilfe der Funktion $mix(a, b, c)$, zwischen dem Bewölkungsgrad (a) und der Noise Textur (b) durchgeführt. Als Interpolationsfaktor (c) wird der Alphawert des Bewölkungsgrades verwendet. Das Resultat wird in die Variable $fragColor$ gespeichert. Alle auf diese Weise generierten Fragments werden in einem weiteren Schritt der Rendering Pipeline auf den dazugehörigen Pixel abgebildet. Die endgültige Farbe des Pixels ergibt sich aus den geblendeten Farbwerten aller Fragments, die auf diesen Pixel abgebildet werden (Abb. 36).

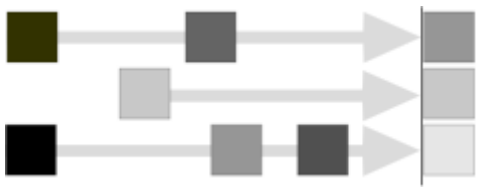

Abbildung 36: Der Farbwert eines Pixels ergibt sich aus den dazugehörigen Fragments (Blending aktiviert)

8.2 Ergebnis

Das Ergebnis zeigt die generierte Textur des dreidimensionalen Bewölkungsgrades. Die Textur in Abbildung 37 ist mit einer niedrigen bis mittleren Auflösung der *Bounding VoxelSphere* gerendert worden und lief flüssig mit 39.2 Frames (GPU: NVIDIA GTX 260 (2009)) [NVI].

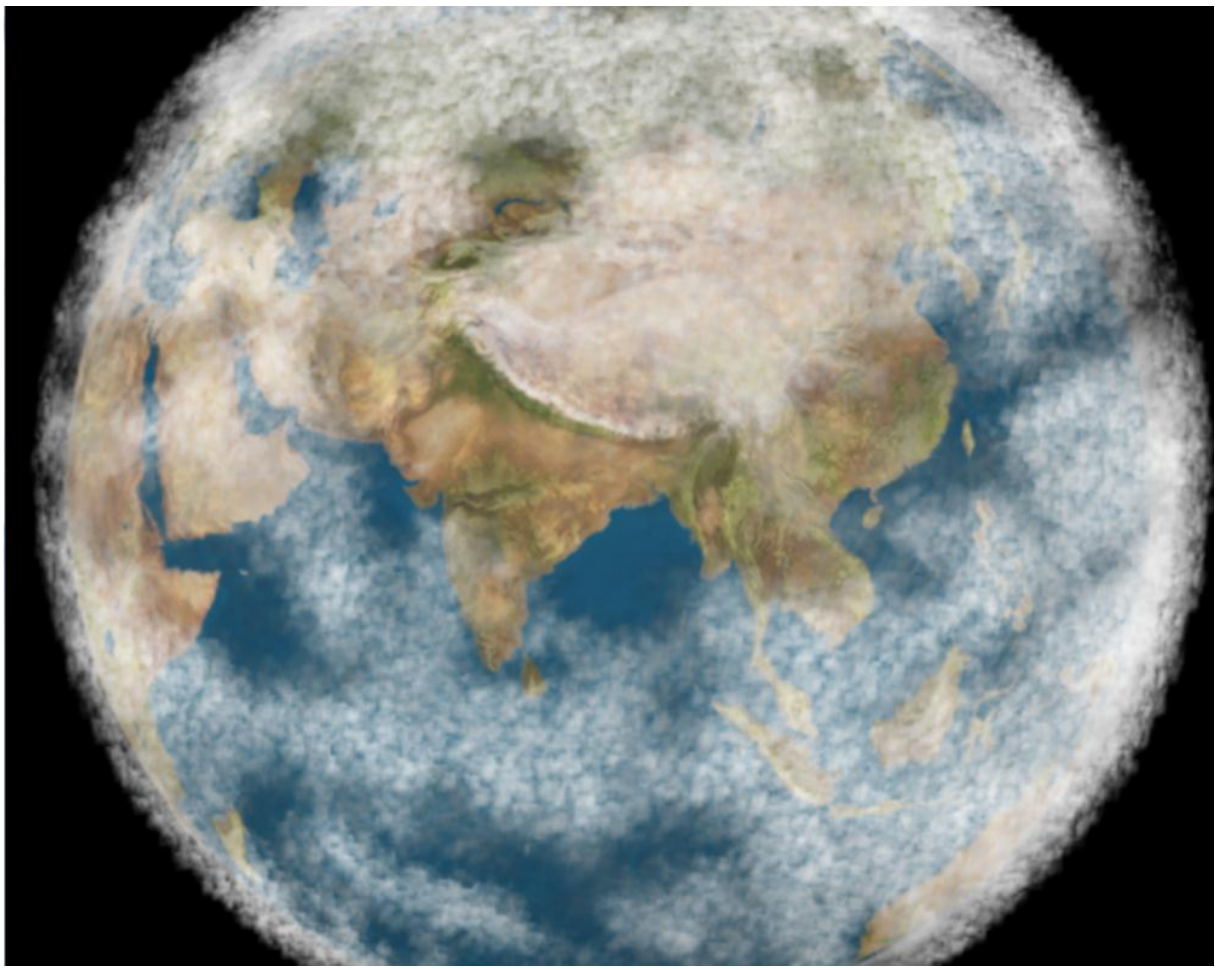

Abbildung 37: 3D Bewölkungsgrad gerendert mit der Bounding VoxelSphere Methode, 10 Kugeln mit einer Auflösung von 400*400 Ebenen

Abbildung 38 ist hingegen mit einer weitaus höheren Auflösung entstanden und lief dementsprechend nur mit 14.5 Frames, deutlich langsamer.

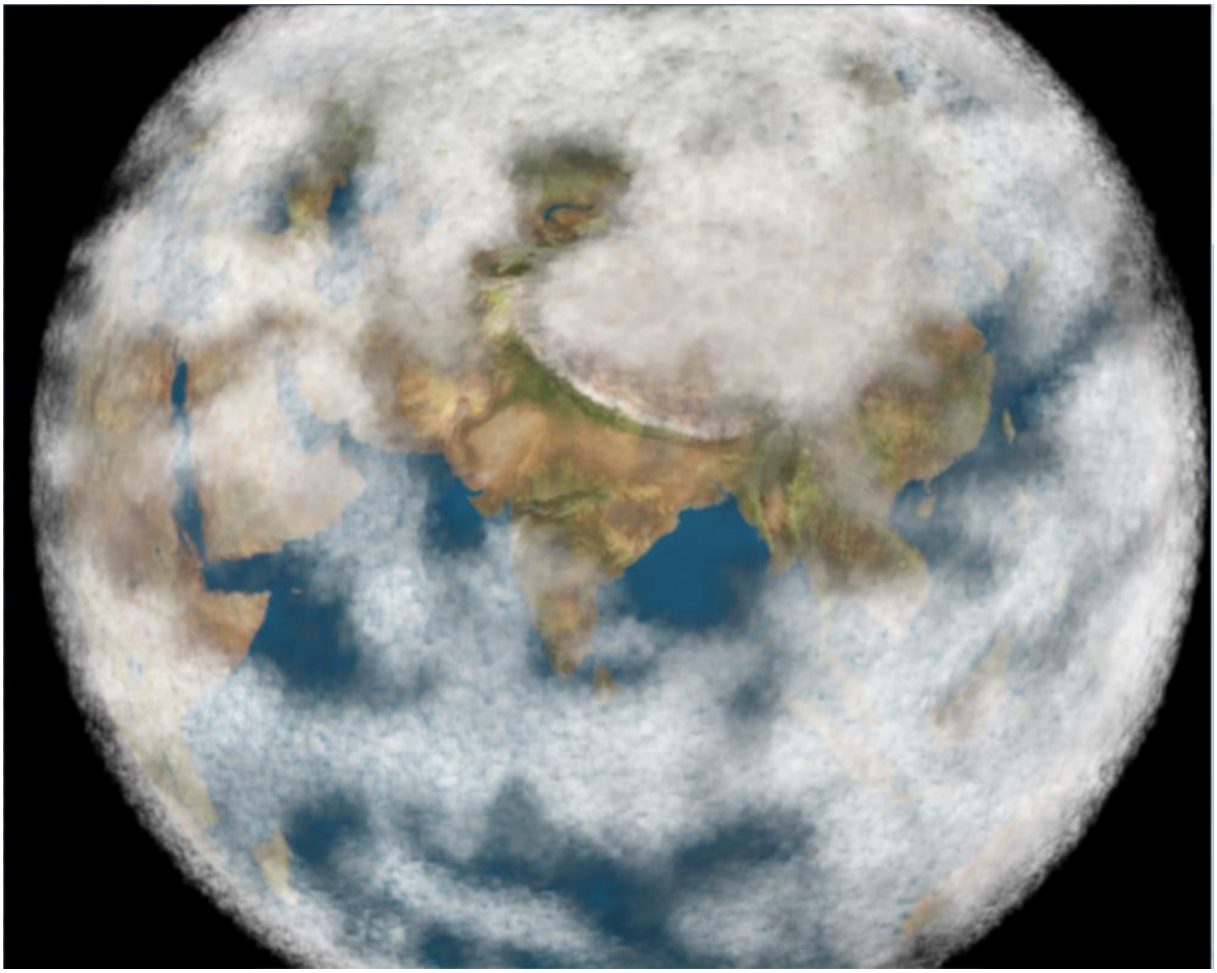

Abbildung 38: 3D Bewölkungsgrad gerendert mit der Bounding VoxelSphere Methode, 30 Kugeln mit jeweils einer Auflösung von 600*600 Ebenen

9 Partikelsysteme

Eine weitere Methode, um dem Problem der realistischen Darstellung von unscharfen Objekten wie Feuer, Wasser oder Wolken entgegen zu wirken sind $Partikelsysteme$. In den 80er Jahren wurde von Bill Reeves, der für die Firma „Lucasfilm“ an Animationen für den damaligen Film *Star Trek II: Wrath of Khan* arbeitete, ein weiteres Rendering Primitiv eingeführt. Nach Reeves Vorstellungen sollten sich $Partikelsysteme$ über die Zeit hin entwickeln. 1983 veröffentlichte Bill Reeves in seinem Paper: *Particle Systems – A Technique of Modeling a Class of Fuzzy Objects* eine Lösung solcher Probleme. Die Idee war, jedem Partikel einen initialen Zustand und eine Menge von stochastischen Übergangsregeln zuzuweisen. Anhand der Anfangsbedingung und der Übergangsregeln war es möglich, $Partikelsysteme$ über die Zeit zu animieren.

$Partikelsysteme$ unterscheiden sich grundsätzlich von der Geometrie, die durch Dreiecke oder Polygone definiert ist. Das Volumen eines $Partikelsystems$ wird durch eine Wolke von Primitives repräsentiert. Primitives können Punkte, Dreiecke oder auch Quader sein. Ferner besitzen $Partikelsysteme$ keine festen Oberflächen und Grenzen. Die Form eines solchen Systems ist nicht statisch sondern dynamisch. Partikel durchlaufen eine Prozesskette: Sie werden geboren, existieren und sterben. Während ihrer Lebenszeit verändern sie ihre Position und Form auf Basis der ihnen zugewiesenen stochastischen Regeln und dem initialen Zustand jedes einzelnen Partikels.

Oft gewählte Attribute für Partikel sind Position, Geschwindigkeit, Farbe, Lebenszeit, Größe, Form und Lichtdurchlässigkeit. Diese Attribute, mit Ausnahme der Lebenszeit, können sich dynamisch über die Zeit entwickeln. Für komplexe Systeme reichen meist die individuellen Attribute eines einzelnen Partikels nicht aus. Zum Beispiel könnte sich eine Menge von Kometen in einem Kraftfeld bewegen. Das Kraftfeld wäre in diesem Beispiel eine globale Größe und würde, in Abhängigkeit von der Entfernung des jeweiligen Kometen auf diesen wirken. Globale Größen sind wichtig, um Partikel in einen Kontext zu setzen. [ROS]

9.1 Implementierung

Die Implementierung lässt sich in zwei Stufen aufteilen. Zu Beginn werden alle benötigten initialen Zustände generiert und jedem Partikel zugewiesen. Im zweiten Schritt werden die Übergangsregeln generiert. Durch diese Zweiteilung ist es relativ einfach, zur Laufzeit die jeweiligen Übergangsregeln zu manipulieren und auf die initialen Zustände anzuwenden. Es ist ebenfalls möglich, die initialen Zustände neu zu generieren und bestehende Übergangsregeln anzuwenden.

Es wurde sich dafür entschieden, Point Sprites für die Geometrie der Partikel zu implementieren. Point Sprites sind seit der OpenGL Version 1.5 verfügbar und wurden zum Nachfolger der bis dahin bekannten GL_POINTS. Point Sprites verfügen im Gegensatz zu GL_POINTS über Texturkoordinaten, die im Fragmentshader zugänglich sind. Über die Funktion

$$glPointSize(float)$$

kann die Größe eines Point Sprites definiert werden. Mit einem einzigen Point Sprite ist es möglich, eine 2D Textur an einem beliebigen Ort auf dem Bildschirm zu platzieren.

Point Sprites haben den weiteren Vorteil, dass sie immer orthogonal zur Kamera stehen. Rotationen, die sicherstellen, dass das Primitiv orthogonal zur Kamera steht, fallen somit aus. Diese Eigenschaft ist wichtig in Verbindung mit *Partikelsystemen*, da ein Partikel keine definierte Vorder- beziehungsweise Rückseite besitzt.

Um *Partikelsysteme*, bestehend aus Point Sprites, dreidimensional wirken zu lassen, bietet es sich an, die Größe in Abhängigkeit von der Entfernung zur Kamera anzupassen. Dadurch wird der unschöne Effekt vermieden, dass alle Point Sprites, unabhängig von ihrer Entfernung zur Kamera, die gleiche Größe besitzen. Eine sehr oft verwendete Formel zur Berechnung der Größe ist:

$$size = \sqrt{\frac{1}{(a + b * d + c * d^2)}}$$

Die Variable d ist der Abstand des Point Sprites zur Kamera. Die Parameter a, b und c sind vom Benutzer einzustellen. Würde Beispielsweise $b = c = 0$ und $a > 0$ gewählt, hätte jeder Point Sprite eine konstante Größe. Wäre $a = c = 0$, würde die Größe, linear dem Abstand entsprechend, abfallen. [WRI]

Mit dem zu implementierenden *Partikelsystem* sollen später Wolken dargestellt werden. Generell wird für solche Anwendungen eine große Anzahl von Partikeln benötigt. Eine Möglichkeit, die Partikelanzahl gering zu halten und dennoch *dichte* Gebilde zu erhalten, ist die Verwendung von Weichzeichnungsfiltern. [PER3]

9.1.1 Weichzeichnungsfilter

Für das Weichzeichnen bietet sich ein gewöhnlicher Gaußscher Weichzeichner [WIK2] an (Abb.40 d). Der Gaußsche Weichzeichner ist nach dem deutschen Mathematiker Carl Friedrich Gauß benannt und spielt eine große Rolle in der Bildverarbeitung. Die mathematische Definition lautet:

$$g(x, y) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{(x^2+y^2)}{2\sigma^2}}.$$

Diese Funktion entspricht der Normalverteilung auf zwei Dimensionen und ist als Gewichtungsfunktion der umliegenden Bildpunkte zu verstehen. Jedem benachbarten Farbwert wird durch die oben genannte Funktion ein skalarer Wert zugewiesen, durch den sein Einfluss am resultierenden Farbwert bestimmt wird.

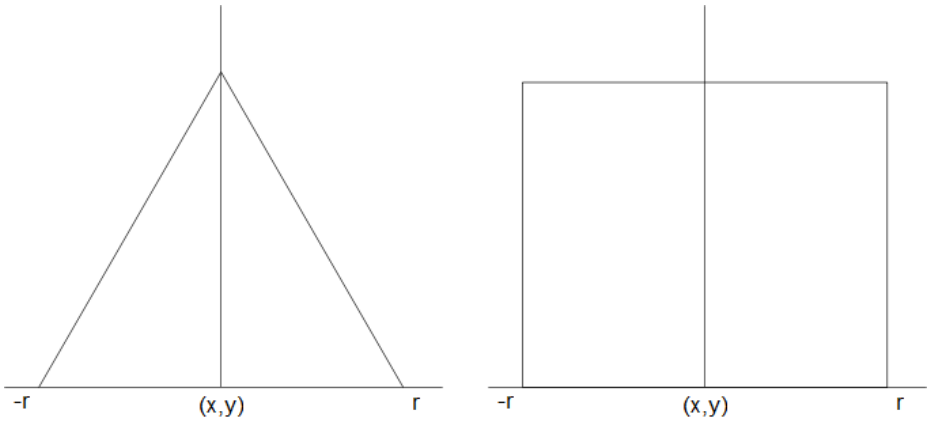

Abbildung 39: Tent-Filter (links) , Box-Filter (rechts)

Für die Implementierung ist es notwendig, den Radius der zu betrachtenden Nachbarn einzuschränken, da nur eine endliche Anzahl von Nachbarn berücksichtigt werden können. Ebenfalls wird der aufwendig zu berechnende Exponentialterm durch einen Tent-Filter approximiert [WIK4]. Dieser ist relativ einfach zu berechnen und ist ein Kompromiss zwischen dem Box- und Gaußfilter. Das Gewicht eines benachbarten Farbwertes berechnet sich mit dem Tent-Filter wie folgt:

$$g(x,y) = 1 - \frac{|\,x + y\,|}{radius}$$

Das Ergebnis ist die in Abbildung 39 dargestellte Tent-Funktion. Je weiter der Farbwert vom aktuellen Farbwert entfernt liegt, desto geringer ist sein Einfluss auf das endgültige Ergebnis.

Die Berechnung des Weichzeichnungsprozesses geschieht nicht auf der CPU sondern ist in einem dafür programmierten Shader auf die Grafikkarte ausgelegt. Die Implementation in OpenGL lässt sich in mehreren Schritten erläutern:
Im ersten Schritt werden alle Objekte, die weichgezeichnet werden sollen, in eine dafür vorgesehene Textur gerendert. Dieser Prozess ist auch unter dem Namen „Render to Texture“ bekannt. Hierbei wird nicht, wie gewöhnlich, in den Framebuffer gerendert, sondern in eine vom Benutzer vorgegebene Textur. Diese Textur kann nun in weiteren Renderdurchläufen verwendet werden, um das endgültige Bild zu generieren.
Im zweiten Schritt werden alle weiteren nicht weichgezeichneten Objekte, in der Farbe Schwarz (0,0,0), ebenfalls in diese Textur gerendert. Dieser zweite Schritt ist notwendig, um gegebenenfalls verdeckte Objekte vom Prozess der Weichzeichnung auszuschließen. Ein schwarzer Farbwert wird beim Weichzeichnen keinen Einfluss haben. Es würde lediglich der Farbwert Schwarz (0,0,0) mit dem Gewicht $g(x,y)$ zum aktuellen Farbwert addiert werden.
In einem dritten Schritt werden alle Objekte, die nicht weichgezeichnet sind, nochmals in Farbe in eine weitere Textur gerendert. Das Ergebnis sind zwei Texturen, die in einem letzten Renderdurchlauf zusammengesetzt werden. Hierbei wird der Mittelwert aus beiden Texturen berechnet und dargestellt (Abb. 41).

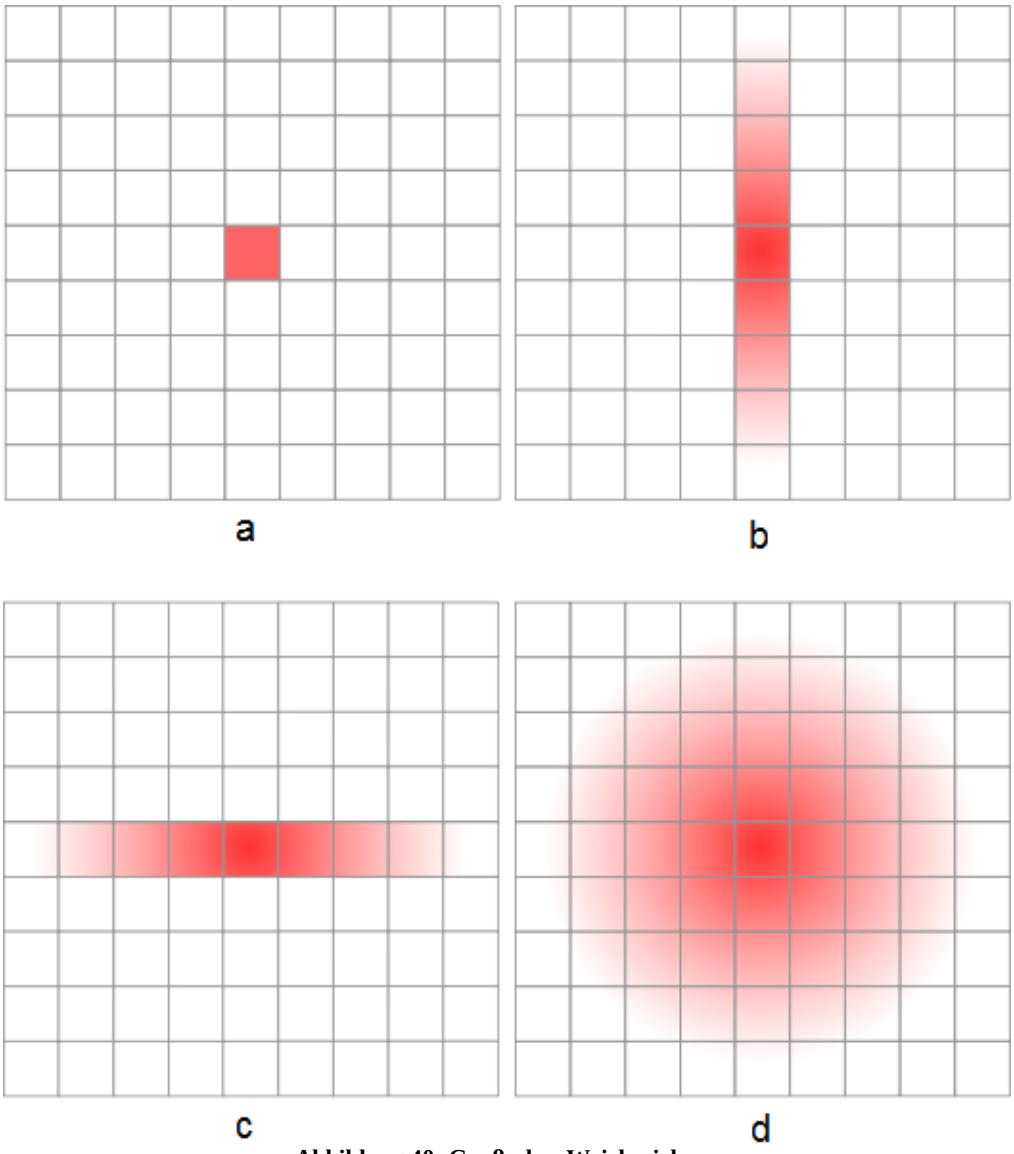

Abbildung 40: Gaußscher Weichzeichner

Für große Radien kann es sehr aufwendig werden, die Weichzeichnung zu berechnen. Es müssen schließlich für jeden Farbwert $radius^2$ viele Berechnungen durchgeführt werden. Um diese Laufzeit weiter zu minimieren, wird der Prozess des Weichzeichnens in zwei einzelne Schritte zerlegt, die nacheinander durchgeführt werden. Die Idee ist, den Prozess in einen vertikalen und einen horizontalen Weichzeichner zu unterteilen (Abb.40 b,c). Die Anzahl der Berechnungen reduziert sich somit auf $radius + radius$ für jeden Farbwert.

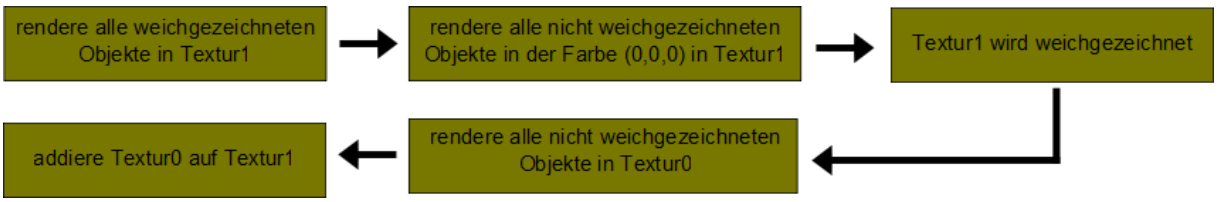

Abbildung 41: Renderpfad für Weichzeichner-Effekt

Mit Hilfe des Weichzeichners können die *Partikelsysteme* in einem weiteren Schritt bearbeitet werden. Mit möglichst wenigen Partikeln sollen wolkenähnliche Strukturen generiert werden. Wie in Abbildung 42 dargestellt, wird zuerst eine relativ geringe Anzahl an Partikeln generiert. In einem zweiten Schritt wird ein Gaußscher Weichzeichner über das Bild gelegt. Im letzten Schritt wird dem *Partikelsystem* eine Textur zugeordnet. Das Ergebnis ähnelt einer hochaufgelösten Wolkentextur. [PER1]

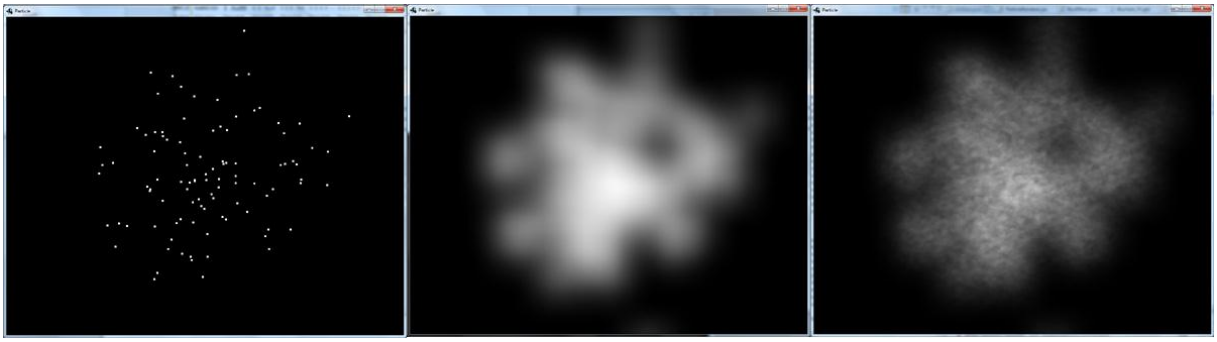

Abbildung 42: Nur Partikel (links), Partikel weichgezeichnet (mitte), Partikel weichgezeichnet und texturiert (rechts)

9.1.2 Zeitliche Animation

Um Partikel über die Zeit zu animieren, wird jedem Partikel ein dreidimensionaler Vektor zugeordnet. Die einzelnen Komponenten ergeben die Richtung, in der sich das Partikel bewegt. Die aktuelle Position des Partikels kann nun über folgende Formel eindeutig bestimmt werden:

$$Position = Startposition + Richtung * Zeitpunkt.$$

Diese Transformation wird im Vertexshader berechnet, wie auch, ob ein Partikel „lebt“ oder „tot“ ist. Jedes Partikel hat eine zu Anfang definierte Lebenszeit, die sich über einen definierten Punkt, an dem das Partikel zu leben beginnt und einem definierten Punkt, an dem das Partikel stirbt, berechnen lässt. Ist das Partikel zum aktuellen Zeitpunkt bereits tot oder noch gar nicht geboren, so wird lediglich eine Variable an den Fragmentshader weitergereicht, mit der Information, dass alle Fragments zu diesem Partikel eliminiert werden können. Eine Möglichkeit wäre, die Farbe aller nicht lebenden Partikel auf die Hintergrundfarbe zu setzen. Dies würde aber zu Problemen führen, denn ein nicht lebendes Partikel könnte demnach ein Lebendes verdecken. Auch würde das Alphablending tote Partikel berücksichtigen und somit falsche Ergebnisse generieren. Deshalb ist es in jedem Fall notwendig, Fragments toter Partikel zu zerstören. Fragments können mit dem Befehl *discard* von weiteren Berechnungen der Rendering Pipeline ausgeschlossen werden. Das Fragment wird demnach auch nicht weiter in den Framebuffer geschrieben und verursacht somit keine Fehler beim Alpha Blending.

10 2D Bewölkungsgrad und Partikelsysteme

Um die Dynamik der Bewölkung adäquat abbilden zu können und der räumlichen Begrenztheit des zuvor vorgestellten Verfahrens ($Bounding\ VoxelSphere$) entgegenzuwirken, wird mit Hilfe der Implementierung von $Partikelsystemen$ der Bewölkungsgrad dreidimensional modelliert. Diesem Vorgehen liegt die Annahme zu Grunde, dass die räumliche Begrenztheit aufgehoben und die Bewegung bzw. Dynamik der Wolken optimal realistisch dargestellt werden kann. Alle Informationen, die das $Partikelsystem$ benötigt, um über die Zeit animiert zu werden, werden auf Basis des zweidimensionalen Bewölkungsgrades generiert. Das Ergebnis wird, wie bei dem $Bounding\ VoxelSphere$ Verfahren, wieder in Verbindung mit einer Erdkugel zusammen dargestellt.

10.1 Implementierung

Der initiale Zustand des $Partikelsystems$ wird auf Basis des ersten Bildes des Bewölkungsgrades generiert. Hierbei werden für jeden gelesenen Farbwert des Bewölkungsgrades Partikelpositionen gesetzt. Die Anzahl der gesetzten Partikel ist davon abhängig, wie hoch der gelesene Bewölkungswert ist. Liegt der gelesene Wert sehr nah an 1, werden mehr Partikel generiert, als wenn der Wert 0.5 entsprechen würde. Jedem Partikel wird zusätzlich eine Tiefenkoordinate zugeordnet. Diese Koordinate ist ebenfalls vom gelesenen Bewölkungswert abhängig. Es wird kein Partikel generiert, falls der Farbwert nicht größer als Null ist. Somit wird erreicht, dass nur dort Partikel entstehen, an denen Bewölkung vorhanden ist. Das Ergebnis ist eine dreidimensionale Anordnung von Partikeln, welche in jedem Renderdurchlauf um die Erdkugel mittels Kugelkoordinaten transformiert wird. Hierbei wird die x-Koordinate für den Winkel $Theta$, die y-Koordinate für den Winkel Phi und die z-Koordinate für den Radius verwendet.

Um die Partikel in Abhängigkeit von der Zeit zu animieren, wird die Position jedes Partikels in einem dafür programmierten Shader auf Basis des aktuellen Bewölkungsgrades ($Zeit = t$) und dem Bewölkungsgrad im nächsten Zeitschritt ($Zeit = t + 1$) berechnet. Für die Berechnung der neuen Position wird ein Gradienten Vektor generiert, der auf die aktuelle Position addiert wird und somit die neue Position bestimmt. Der Gradient ist ein Differentialoperator, der auf ein Skalarfeld angewendet wird. Das Ergebnis ist ein Vektorfeld, welches die Änderungsrate und die Richtung der größten Änderung des Skalarfeldes bestimmt. Der Gradient für ein gegebenes Skalarfeld f berechnet sich wie folgt [WIK1]:

$$\nabla f = grad(f) = \frac{\delta f}{\delta x_1}\, e_1 + \frac{\delta f}{\delta x_2}\, e_2 + \cdots + \frac{\delta f}{\delta x_n}\, e_n = \begin{pmatrix} \frac{\delta f}{\delta x_1} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{pmatrix}$$

Das Ergebnis ist ein Spaltenvektor, der die Ableitung jeder Richtung als Komponenten enthält.

Um dieses Verfahren zu implementieren, ist es notwendig, die transformierten Positionen der Partikel nach jedem Zeitschritt zwischenzuspeichern, um im nächsten Renderschritt auf den zuletzt modifizierten Positionen weiterzuarbeiten. Dies wird mit Hilfe von Transform Feedback realisiert. Transform Feedback ist Teil der Rendering Pipeline und bietet die Möglichkeit, transformierte Output-Variablen des Vertex- oder Geometrieshaders im Speicher der Grafikkarten zu hinterlegen und somit für weitere Durchläufe der Pipeline bereitzustellen (Abb.43).

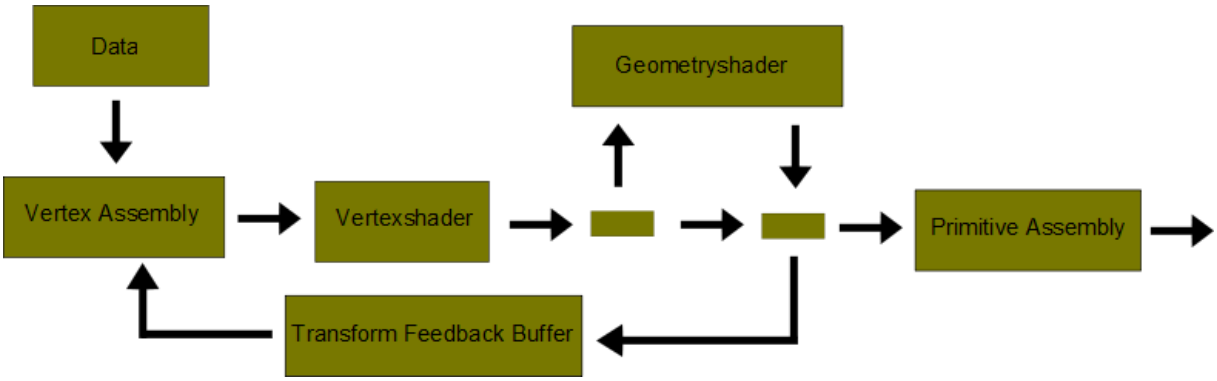

Abbildung 43: Transform Feedback

Der Gradient des Bewölkungsgrades berechnet sich aus dem Skalarfeld der zeitlichen Ableitung. Die zeitliche Ableitung ergibt sich wiederum aus dem Differenzenquotienten des Bewölkungsgrades zum Zeitpunkt t und dem darauffolgenden Zeitpunkt $t + 1$. Die endgültige Formel für die Berechnung des Vektors, der auf die aktuelle Position des Partikels addiert wird, lautet:

$$\nabla p = grad\left(\frac{\delta f}{\delta t}\right)$$

Abbildung 44 zeigt das Gradientenvektorfeld des Bewölkungsgrades zu einem gewählten Zeitpunkt t.

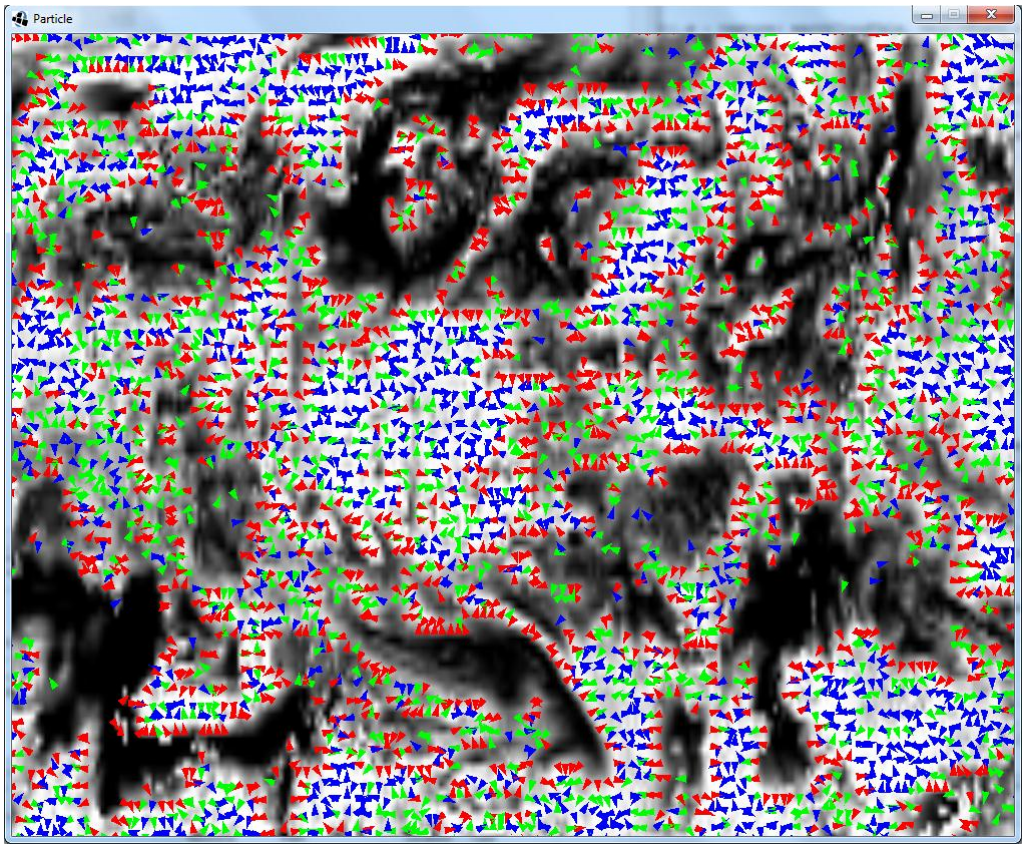

Abbildung 44: Gradient des Bewölkungsgrades (rote Pfeile: hohe Änderung, blaue Pfeile: mittlere Änderung, grüne Pfeile: geringe Änderung)

Mit dem berechneten Gradienten ist es jetzt möglich, die Partikel über die Zeit zu animieren.

Im nächsten Schritt werden die Partikel weichgezeichnet und schließlich mit einer Textur versehen. Die Texturierung erfolgt, wie im Kapitel $2D\ Bewölkung\ und\ 3D\ Noise$ erläutert, über die Summe skalierter Noise Texturen. Für die Partikel muss dieses Verfahren jedoch weiter modifiziert werden.
Als ein Problem der Texturierung stellt sich hier heraus, dass die Noise Textur aus jeder Kameraeinstellung die gleiche Struktur hat. In einer laufenden Animation wird dies schnell deutlich. Abbildung 45 zeigt zwei Bilder, in denen die Partikel mit einer Textur texturiert wurden, die nicht von der Kameraposition abhängig ist.

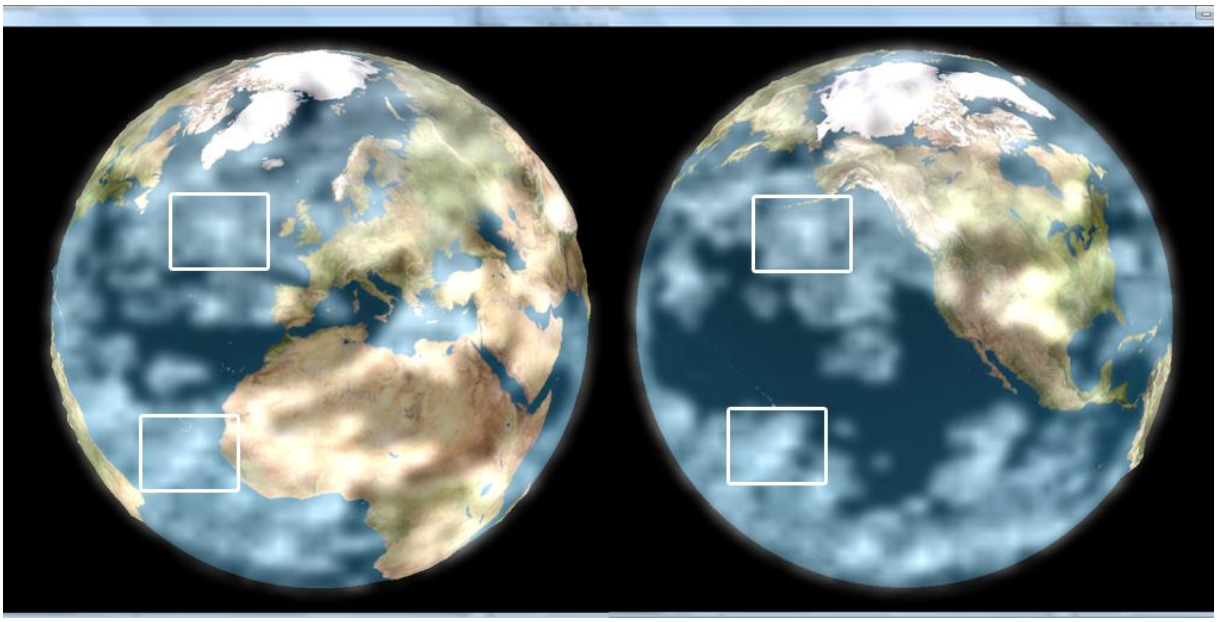

Abbildung 45: statische Texturierung der Partikel

Die markierten Stellen zeigen deutlich, dass es sich um dieselbe Textur handelt, mit der die Partikel texturiert worden sind. Um dieses Problem zu beheben, wird die Textur in Abhängigkeit von der Kameraposition skaliert, um somit für jede Kameraeinstellung eine neue Textur zu generieren. Die Position der Kamera rotiert in diesem Beispiel um den Mittelpunkt der Erde. Die Winkel *Phi* und *Theta* der Kugelkoordinaten können somit als Skalierungsfaktor genutzt werden.

Durch hohe Ansammlungen von Partikeln, kann es durch den Weichzeichner oder auch Alpha Blending zu unerwünscht hellen Bereichen kommen (Abb. 46). Dieses generelle Problem ist auffällig in *Partikelsystemen* und kann möglicherweise auf die hohe Dynamik zurückgeführt werden.

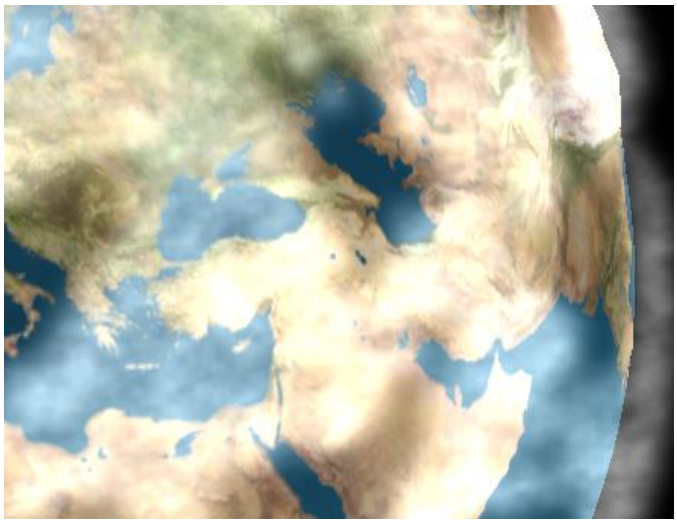

Abbildung 46: "Überbelichtung" durch Polarisierung der Partikel

10.2 Ergebnis

In Abbildung 47 werden die Partikel nochmals ohne Weichzeichner und ohne Texturierung gezeigt.

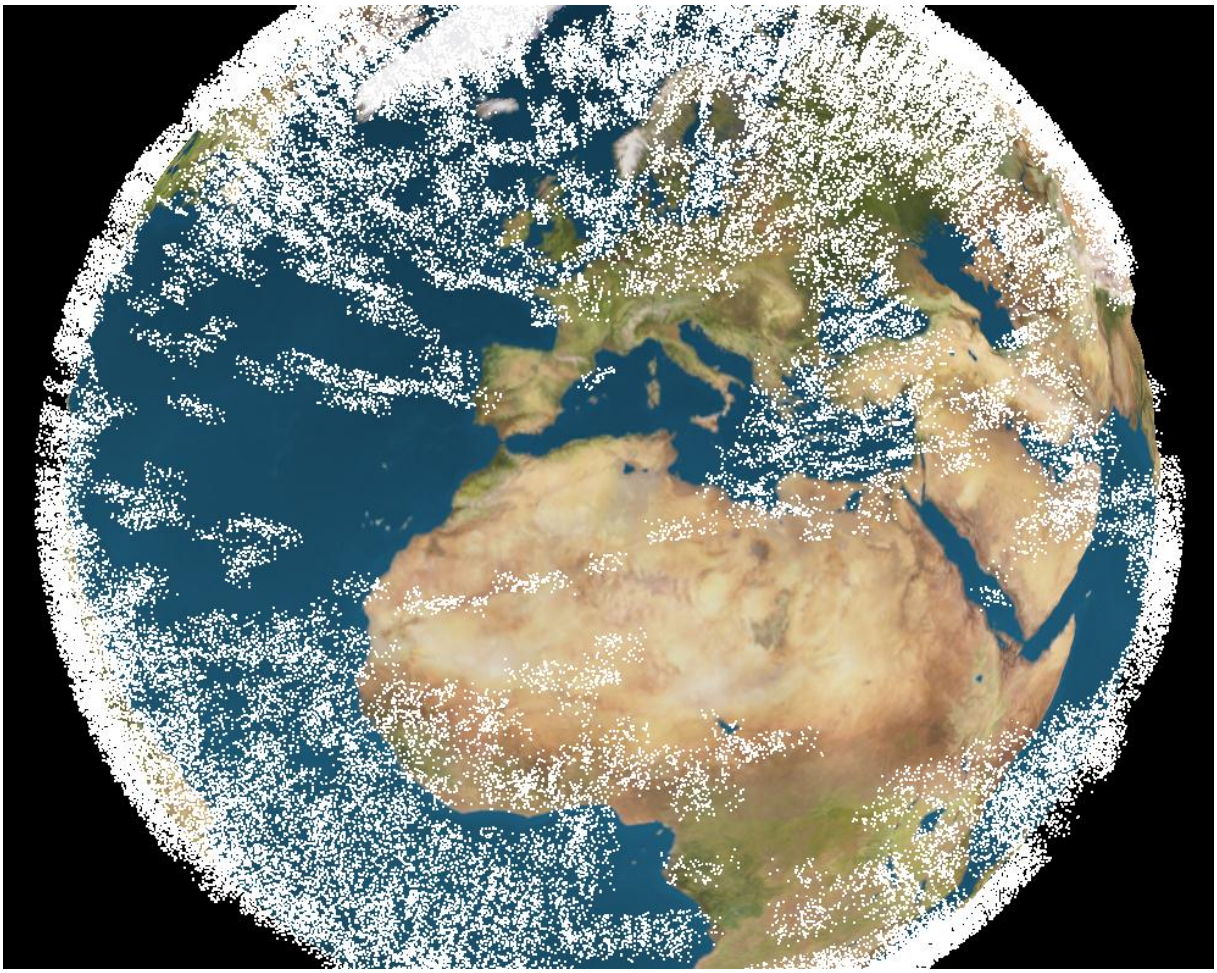

Abbildung 47: Bewölkungsgrad mit 290293 Partikeln visualisiert (ohne Weichzeichner und ohne Textur)

Als Ergebnis wird die fertige Visualisierung des Bewölkungsgrades auf Basis von *Partikelsystemen* dargestellt (Abb. 48). Das *Partikelsystem* besteht in diesem Fall aus 290293 Partikeln und läuft mit 59.9 Frames zufriedenstellend schnell (GPU: NVIDIA GTX 260 (2009)). [NVI]

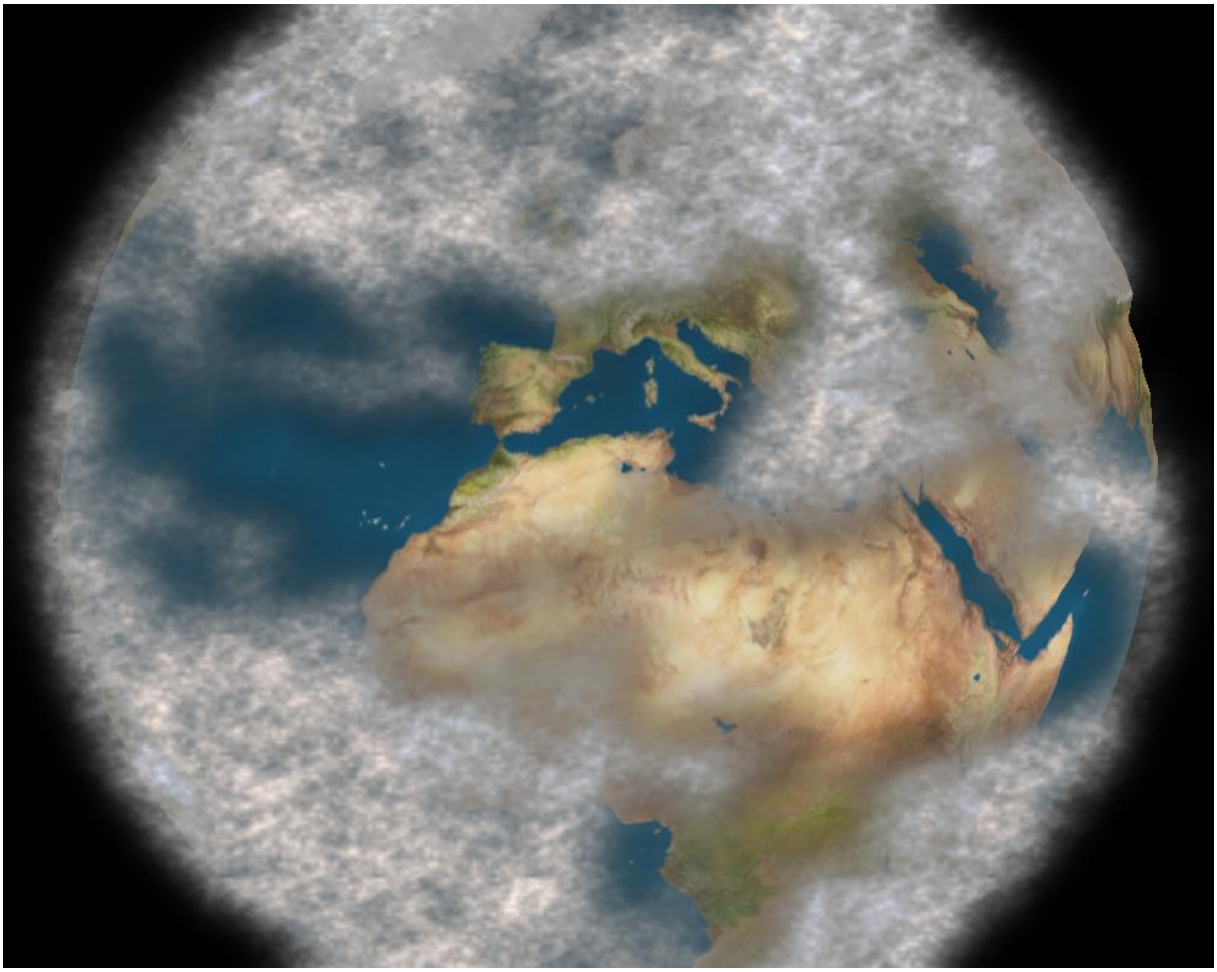

Abbildung 48: Bewölkungsgrad mit 290293 Partikeln visualisiert (mit Weichzeichner und Textur)

11 Vergleich

Im Rahmen der vorliegenden Arbeit wurden zwei Ansätze zur Modellierung und Visualisierung von dreidimensionaler Bewölkung vorgestellt. Hierbei handelt es sich einmal um die *Bounding VoxelSphere* Methode, welche das zuvor generierte dreidimensionale Bewölkungsvolumen darstellt. Zum anderen die Visualisierung durch ein *Partikelsystem*.
Unter der Fragestellung, inwieweit diese beiden Ansätze zur Modellierung und Visualisierung von dreidimensionalen Bewölkungsdaten geeignet sind, werden diese nach vier relevanten Kriterien verglichen:

- Performance
- Lokale Begrenztheit
- Implementierung
- Darstellungsqualität

Performance
Die *Bounding VoxelSphere* Methode ist in ihrer Skalierung flexibel und kann an das zu rendernde Volumen angepasst werden. Für die Generierung sehr guter Ergebnisse muss die Auflösung relativ hoch gewählt werden, dies spiegelt sich jedoch in der Performance wider und kann dazu führen, dass es zu einer nicht flüssigen Darstellung kommt. Allerdings sind auch geringe bis mittlere Auflösungen zufriedenstellend.
Im Gegensatz hierzu ist die implementierte Methode der *Partikelsysteme* sehr performant. Die Partikelzahl ist auf ein Minimum reduziert und wird mit Hilfe von Weichzeichnern beliebig skaliert. Die gewonnene Performance spiegelt sich jedoch in einer geringeren Qualität der dargestellten Daten wieder. Dies liegt daran, dass der Weichzeichner den Verlust von Informationen zur Folge hat.

Lokale Begrenztheit
Die *Bounding VoxelSphere* Methode stellt einen Raum zur Verfügung, in dem das Volumen visualisiert werden kann. Dieser Raum ist jedoch an einen festen Ort gekoppelt und hat definierte Grenzen. Volumina, die sich über die Grenzen erstrecken, werden nicht in der Visualisierung berücksichtig.
Das *Partikelsystem* hat dieses Problem nicht. Es ist hochdynamisch und kann sich im ganzen Raum ausbreiten.

Implementierung
Die Implementierung der *BoundingBox* und später der *Bounding VoxelSphere* ist im Vergleich zum *Partikelsystem* einfacher. Hinzu kommt, dass das grundlegende Modell der *BoundingBox* leicht auf andere Methoden, die das Visualisierungsproblem besser lösen, übertragen werden kann.
Im Kontrast hierzu ist das *Partikelsystem* relativ aufwendig zu implementieren, da sich eine Fülle von Freiheitsgeraden ergeben. Dazu kommt die sehr hohe Dynamik des Systems, welche nicht immer zu gewünschten Ergebnissen führen kann. Die Zeit, die ein Entwickler benötigt, um ein *Partikelsystem* zu entwerfen, spiegelt sich nicht unmittelbar in dem Ergebnis wieder.

Darstellungsqualität

Die *Bounding VoxelSphere* Methode liefert qualitativ hochwertige Ergebnisse bezüglich des zu visualisierenden Volumens. Lediglich an den Randstellen kann es zu Blending-Fehlern kommen, die sich jedoch nur minimal auf den optimalen Grad der Visualisierung auswirken.
Im Gegensatz hierzu gibt es bei dem *Partikelsystem* zwar auch gute Ergebnisse bezüglich der realistischen Darstellung des Bewölkungsgrades, jedoch durch Polarisierung einer großen Anzahl von Partikeln, können aufgrund des Weichzeichnens überproportional helle Bereiche entstehen. Diese Art „Überbelichtung“ kann als ein allgemeines Problem bei der Anwendung von *Partikelsystemen* angesehen werden. Durch die Anwendung des Weichzeichners ergibt sich zwar eine höhere Performance, die sich jedoch negativ auf die Qualität der dargestellten Daten auswirkt.

12 Fazit

Ziel der Arbeit war, mit Hilfe der Programmierung von modernen Grafikkarten und unter Verwendung von Grafik APIs Algorithmen zu entwickelt, die eine dreidimensionale Darstellung von 2D Bewölkungsdaten ermöglichen. Diese Zielsetzung konnte erfolgreich realisiert werden. Mit den entwickelten Algorithmen können zweidimensionale Bewölkungsdaten in eine dreidimensionale Darstellung transformiert und gerendert werden.

Für die Transformation der Daten stellten sich Noise Techniken durch ihr wolkenähnliches Aussehen als besonders geeignet dar. Es wurden zwei grundlegend verschiedene Ansätze zur dreidimensionalen Visualisierung entwickelt und implementiert. In beiden Ansätzen wurden Skalierungsprobleme, die beim Heranzoomen der Kamera entstehen können, geschickt mit Hilfe von Noise Texturen minimiert. Zuerst wurde eine Methode entworfen, die aus der 2D Bewölkung eine 3D Bewölkung modellierte. Das Resultat wurde im nächsten Schritt mit Hilfe der $Bounding\,VoxelSphere$ Methode gerendert. Hierbei gab die $BoundingBox$ Methode das grundlegende Modell vor, welches auf Ebenen basiert, um das Volumen zu rendern. Die $BoundingSphere$ Methode erweiterte diesen Ansatz auf eine Kugelgeometrie, die für die Projektion um die Erdkugel eingesetzt werden konnte. Allerdings brachte dieser Ansatz Nachteile mit sich: Das Problem war die nicht orthogonal ausgerichtete Geometrie der Kugeln an den Rändern. Eine Lösung fand sich in einer dritten Modifikation, der $Bounding\,VoxelSphere$ Methode. Die $Bounding\,VoxelSphere$ Methode platzierte kleine Ebenen im Volumen, die immer orthogonal ausgerichtet wurden.

Die im Rahmen dieses Ansatzes entworfenen Algorithmen zur Visualisierung von dreidimensionaler Bewölkung waren in der Lage, beliebige dreidimensionale Rasterdaten zu rendern. Obwohl die Ergebnisse der gerenderten 3D Bewölkung zufriedenstellend waren, zeigte sich, dass eine sehr hohe Qualität des gerenderten Volumens eine hohe Rechenleistung erforderte und somit zu einer nicht echtzeitfähigen Darstellung führen kann. Außerdem ist die Ausdehnung der Volumina durch das Begrenzungsvolumen der Verfahren beschränkt.

Um Performance Problemen entgegenzuwirken und eine dynamische Raumausdehnung zu erreichen, wurde die Bewölkung mit Hilfe eines $Partikelsystems$ dreidimensional visualisiert. Das $Partikelsystem$ generierte ebenfalls zufriedenstellende Ergebnisse, die gute Qualität mit relativ geringem Rechenaufwand vorwiesen. Das generelle Problem, dass $Partikelsysteme$ aus sehr vielen Partikeln bestehen müssen, um zusammenhängende Gebilde zu schaffen, wurde unter Anwendung von Weichzeichnungsfiltern erfolgreich behoben. Was bleibt, ist die komplizierte Implementierung mit vielen Freiheitsgraden als ein wesentlicher Nachteil von $Partikelsystemen$. Diese kann zu einer erhöhten Fehleranfälligkeit des Systems führen.

Eine mögliche Erweiterung wäre die Optimierung der $Bounding\,VoxelSphere$ Methode, um Volumina in noch besserer Qualität in Echtzeit zu rendern. Ebenfalls könnte die Anzahl der Partikel dynamisch in Abhängigkeit der Bewölkung zur Laufzeit variiert werden, um eine bessere Kopplung des $Partikelsystems$ an die momentane Bewölkung zu erreichen. Des Weiteren wäre darüber nachzudenken, die angewandten Verfahren mit geeigneten Beleuchtungsmodellen zu ergänzen.

Abschließend bleibt zu sagen, dass beide Methoden in ihrem jetzigen Stand der Realisierung durchaus zielführend zur Modellierung und Visualisierung von 3D Bewölkung eingesetzt werden können.

13 Literaturverzeichnis

[HAD] Markus Hadwiger, P. L. (2009). GPU-Based Volume Ray-Casting.

[OGL] OpenGL, *official Homepage*. Von http://www.opengl.org/

[SEG] Mark Segal, Kurt Akele. *The OpenGLRGraphics System:A Specification (Version 4.1 (Core Profile) - July 25, 2010)*

[JOGL] Java OpenGL Binding, *official Homepage.* Von http://www.jogl.info

[LWJGL] Lightweight Java Game Library, *official Homepage*. Von http://lwjgl.org/

[PER1] Perlin, K. (2001). *Improving Noise*. New York.

[PER2] Perlin, K. (1999). *Making Noise*. Von http://www.noisemachine.com/talk1/

[PER3] Perlin, K. *Puff Noise*. Von http://mrl.nyu.edu/~perlin/experiments/puff/

[PER4] Perlin, K. (1985). *An Image Synthesise. New York.* ISBN:0-89791-166-0

[ROS] Randi J. Rost, B. L.-K. (2009). *OpenGL Shading Language Third Edition.* Addison-Wesley. ISBN-13: 978-0321637635

[WIK1] Wikipedia, *Gradient*.Von http://de.wikipedia.org/wiki/Gradient_(Mathematik)

[WIK2] Wikipedia, *Weichzeichnen*. Von http://de.wikipedia.org/wiki/Weichzeichnen

[WIK3] Wikipedia, *Bilineare Interpolation*.
Von http://de.wikipedia.org/wiki/Bilineare_Filterung

[WIK4] Wikipedia, *Trilineare Interpolation*.
Von http://de.wikipedia.org/wiki/Trilineare_Filterung

[WIK5] Wikipedia, *Triangular function.*
Von http://en.wikipedia.org/wiki/Triangular_function

[WRI] Richard S. Wright, J. N. (2010). *OpenGL SuperBible Fifth Edition Comprehensive Tutorial and Referenze.* Addison-Wesley.
ISBN-13: 978-0321712615

[EBE] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steve Worley (2002). *Texturung and Modeling – Third Edition: A procedural Approach.* Morgan Kaufmann. ISBN-13: 978-1558608481

[WEN] Wenke, H. (2007). *World Weather3D.*
http://project.informatik.uni-osnabrueck.de/hwenke/EarthWeather3D.html

[KHR] Khronos Group, *official Homepage.* Von http://www.khronos.org/

[NVI] NVIDIA GTX 260 Grafikkarte.
Von http://www.nvidia.de/object/geforce_gtx_260_de.html

14 Erklärung zur selbstständigen Abfassung der Bachelor-Arbeit

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als die von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Greven, den 18.2.2011 _____