



# Lab Report

Master Projekt System Entwicklung, SS 2013  
(*Prof. Dr. J. Wietzke, Prof. Dr. E. Hergenröther*)

„Strömungssimulation in der erweiterten Realität“

vorgelegt von

T. Sturm (709794)

A. Holike (724986)

S. Arthur (715720)

M. Djakow (718531)

27. März 2014

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>2</b>
<b>2 Bestehende Arbeiten</b>	<b>3</b>
<b>3 Konzept</b>	<b>4</b>
3.1 Modularisierung . . . . .	4
<b>4 Realisierung</b>	<b>5</b>
4.1 Hardwareaufbau . . . . .	5
4.2 XNA Game Studio 4.0 + Kinect SDK . . . . .	7
4.3 Die Hauptanwendung . . . . .	8
4.4 Die Datencontainer . . . . .	9
4.5 Kinect Integration . . . . .	11
4.6 Das Terrain . . . . .	12
4.7 Das Kamerasytem . . . . .	16
4.8 Das Partikelsystem . . . . .	17
4.9 Die Physik . . . . .	21
4.10 Das Graphical User Interface . . . . .	25
4.11 Properties . . . . .	28
4.12 Reflection . . . . .	28
<b>5 Ergebnisse</b>	<b>30</b>
<b>6 Probleme</b>	<b>31</b>
6.1 Echtzeitfähigkeit . . . . .	31
6.2 Darstellung . . . . .	31
<b>7 Fazit &amp; Ausblick</b>	<b>32</b>

# 1 Einleitung



Die Technik der Strömungssimulation spielt heutzutage eine größere Rolle den je. Lange ist es her das Windkanäle nur zur Erforschung und Verbesserung der Aerodynamik von Flugzeugen genutzt wurde. Heute gibt es kaum noch ein Kraftfahrzeug das nicht im Windkanal optimiert wurde und auch Architekten und Statiker nutzen immer häufiger den Windkanal um ihre Konstruktionen und Berechnungen zu überprüfen. Ein relativ neues Thema in diesem Gebiet ist die Untersuchung kompletter Stadtteilen und Städten im Windkanal. Durch den enormen Anstieg von neuen Wohn-, Gewerbe- und Industriegebieten in den letzten Jahrzehnten, schrumpft der Anteil von freien und natürlichen Flächen immer weiter, wodurch die Frischluftzufuhr negativ beeinflusst wird und sich das Klima in Städten stetig weiter erwärmt und verschlechtert. Um dieser Entwicklung entgegen zu wirken nutzen auch Städteplaner immer häufiger die Vorteile des Windkanals.

So nützlich der Windkanal auch für alle vorgestellten Anwendungen ist, so ist dessen Nutzung auch immer mit sehr hohen Kosten verbunden. Durch die immer weiter steigende Leistung von Computern liegt es deshalb nahe, zu versuchen, erste Tests und Untersuchung von der Realität auf den Computer auszulagern und auf diesem zu simulieren. Genau aus diesem Gebiet bestand der Hauptteil unserer Forschung in diesem Semester. Um dieses doch sehr mathematische und trockene Thema für Außenstehende noch etwas attraktiver und greifbarer zu machen haben wir es jedoch, mit Hilfe der Kinect Kamera von Microsoft, noch um eine Interaktive Komponente erweitert.

Auf den Folgenden Seiten werden wir nun unser Vorgehen sowie die Fortschritte, Erfahrungen und Ergebnisse dieses Projekts vorstellen.

## 2 Bestehende Arbeiten



Abbildung 1: (a) Augmented Reality Sandbox. (b) A Particle System for Interactive Visualization of 3D Flows. (c) Synthetic Turbulence using Artificial Boundary Layers. (d) Scalable Fluid Simulation using Anisotropic Turbulence Particles

Bevor wir auf unsere eigene Arbeit eingehen, wollen wir kurz einige andere Arbeiten vorstellen die sich mit ähnlichen Themen beschäftigen und an denen wir uns zum Teil orientiert haben.

Beginnen wollen wir mit der *Augmented Reality Sandbox* [**Kreylos2010**] einem beeindruckenden Projekt der *University of California* in Zusammenarbeit mit weiteren Forschungsinstituten, welches die Grundidee zu unserer Integrierung der Kinect 3D Kamera von Microsoft und dem Sandkasten lieferte. Ziel dieses Projekts war es, mit Hilfe der Kinect Kamera, eine spielerische Möglichkeit zu entwickeln, die es erlaubt topologische Landschaft, durch formen des Sandes mit den eigenen Händen, zu erstellen. Zusätzlich nutzen sie auch ein physikalisches System zur Simulation von Wasser die man in Abbildung 1 gut erkennen kann.

Mit physikalischen System in Bezug auf Strömungssimulation beschäftigen sich auch die folgenden Arbeiten. *A Particle System for Interactive Visualization of 3D Flows* [**Krueger2005**] ist zwar schon etwas älter, aber es ist eine der ersten Arbeiten in der mit der Auslagerung der Physikberechnung auf die GPU, um realistische Echtzeitströmungen zu simulieren, experimentiert wird. Zudem werden sehr interessante Konzepte nutzt, wie zum Beispiel die Speicherung der Partikelpositionen in einer Textur die gleichzeitig als Ein- und Ausgabecontainer zwischen den einzelnen Berechnungsschritten dient.

Eine weitere interessante Arbeit ist *Synthetic Turbulence using Artificial Boundary Layers* [**Pfaff2009**]. Hier werden vorberechnete Strömungsfelder zur Simulation der Partikel genutzt umso extrem realistische Darstellungen zu realisieren. Durch die Vorberechnung verfügt dieses Verfahren aber leider über keine Echtzeitfähigkeit. In der Folgearbeit *Scalable Fluid Simulation using Anisotropic Turbulence Particles* [**Pfaff2010**] wird dieses Manko, durch starke Parallelisierung und der Berechnung vieler kleineren Einzelsimulationen, allerdings behoben.

# 3 Konzept

## 3.1 Modularisierung

Um eine parallele Entwicklung und spätere Erweiterbarkeit sicherzustellen entschieden wir uns für eine modellbasierte Entwicklung. Unser Projekt lässt sich in 3 Kategorien gliedern. Das ParticleSystem (**ParticleSystem**), die Kinect Ansteuerung (**SandstormKinect**) und einen Controller (**Sandstorm**) der Events entgegen nimmt und diese verarbeitet oder ggf. weiterleitet. Das XNA-Framework unterstützt einen modellbasierten Ansatz indem es jeder DrawableGameComponent ihren eigenen Kontext zuweist. Dies bedeutet das jede DrawableGameComponent ein eigenes kleines Projekt darstellt und unabhängig von den anderen Projekten betrieben werden kann.

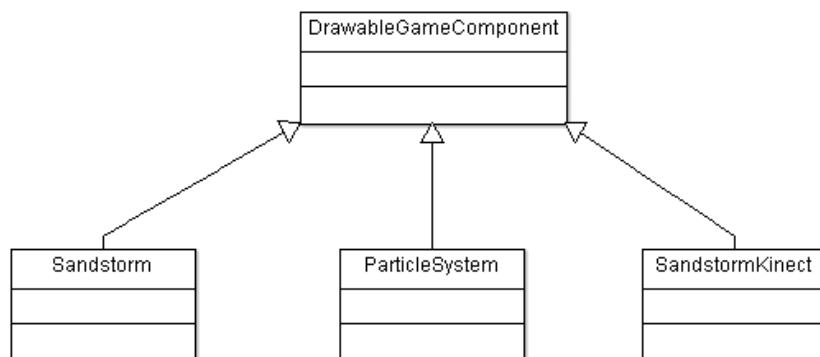


Abbildung 2: Komponenten

# 4 Realisierung

In diesem Kapitel wird die Realisierung des Projekts erläutert. Begonnen mit dem Hardwaeraufbau über die Wahl der Frameworks, das Grundgerüst der Anwendung, die Integration der *Kinect*-Kamera, dem Aufbau des Kamera-Systems und der Terrain-Darstellung bis zum Partikelsystem samt Physik.

## 4.1 Hardwareaufbau

Die Hardware

Beschreibung wie der Tisch gebaut ist  
Was für Rechner dranne hängt  
Was für Beamer dranne hängt  
Was das mit dem Sand war

Die Konstruktion

Der Aufbau

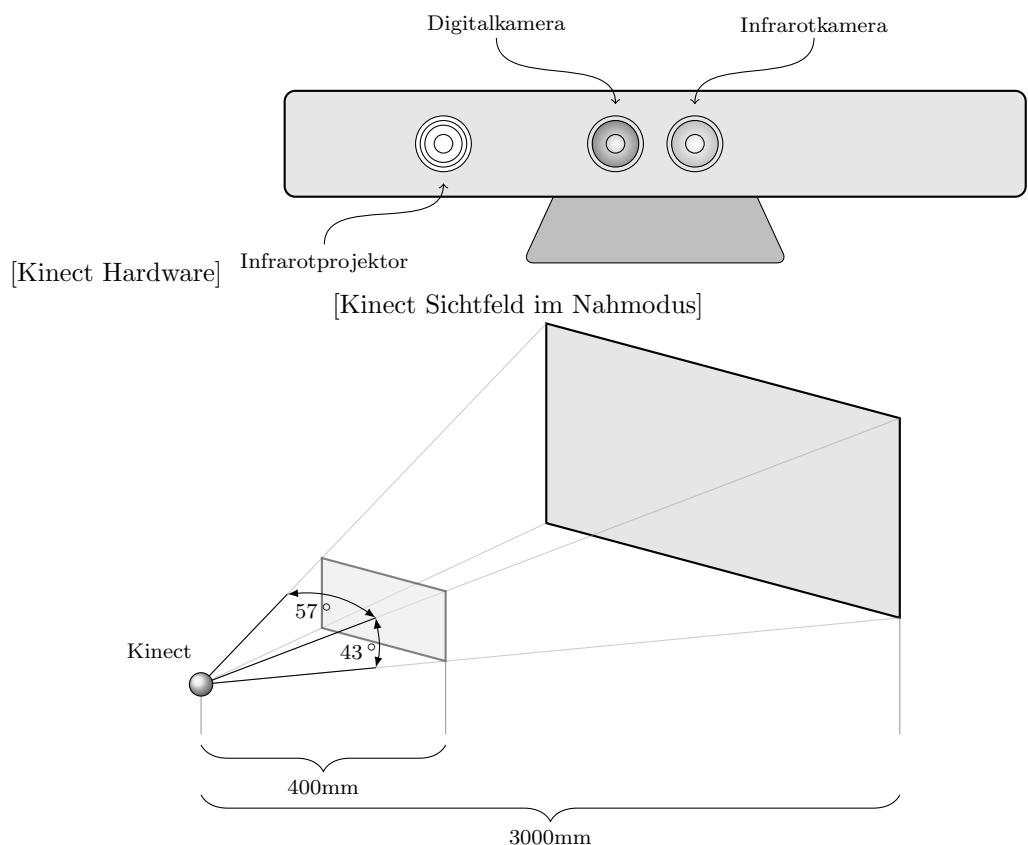


Abbildung 3: Kinect Sensor

## 4.2 XNA Game Studio 4.0 + Kinect SDK



Für die Interaktion mit der *Kinect*-Kamera und der Darstellung der Landschaft und des Partikelsystems haben wir uns für das *XNA Game Studio 4.0* und das *Kinect SDK* von *Microsoft* entschieden. Das *XNA Game Studio* ist eine Programmierumgebung die auf *Visual Studio* basiert und zur Entwicklung von Spielen für *Windows-Phone*, *XBox 360* und *Windows*-basierten Computern entworfen wurde. Bestandteil des *XNA Game Studio* ist das *XNA Framework*, welches mehrere auf dem *.Net-Framework* basierende Bibliotheken vereint und eine sehr einfache und angenehme Schnittstelle zu diesen bereitstellt.

Dazu gehören:

### **DirectX**

DirectX ist eine API für hochperformante Multimedia-Anwendungen und kommt meist bei der Hardware-beschleunigten Darstellung von 2D- und 3D-Grafiken zum Einsatz.

### **XInput**

XInput ist eine API zur Verarbeitung von Benutzereingaben über Maus, Tastatur und den *XBox 360* Kontroller.

### **XACT**

XACT(*Microsoft Cross-Platform Audio Creation Tool*) stellt einfache Schnittstellen zur Audiomodulierung und der Verknüpfung von Sounds an bestimmte Ereignissen bereit.

In unserer Implementierung wird ausschließlich DirectX für die Darstellung und XInput für die Verarbeitung der Benutzereingaben genutzt. Zudem kommt zusätzlich das *Kinect SDK* zur Ansteuerung der *Kinect*-Kamera zum Einsatz, welches auch auf dem *.Net-Framework* basiert und sich dadurch nahtlos und ohne weitere Anpassungen in das System integrieren lässt.

## **4.3 Die Hauptanwendung**

Nachdem der Hardwareaufbau und die Wahl der Frameworks vollendet war, ging es daran den Aufbau der Hauptanwendung, welche als Einstiegspunkt für alle weiteren Aufgaben fungiert, zu entwerfen.

### **4.3.1 Die Anforderungen**

### **4.3.2 Die Umsetzung**

Als Start der Umsetzung diente ein leeres XNA-Projekt, welches bereits

## 4.4 Die Datencontainer

Um die Echtzeitfähigkeit unseres Systems wieder herzustellen, benötigt es der GPU-Unterstützung. Bevor wir in technische Details verfallen, werden wir einen kleinen Exkurs machen wie GPUs eigentlich arbeiten.

### 4.4.1 Die GPU

CPUs und GPUs weisen grundlegend verschiedene Architekturen auf.



Abbildung 4: GPU-Architektur

TODO: geklaut hier: <http://www.tomshardware.de/CUDA-Nvidia-CPU-GPU,testberichte-240065-2.html>

Während eine CPU einen relativ großen Befehlssatz hat um Ganz- oder Fließkommazahlen zu verarbeiten, besitzt hingegen eine GPU einen sehr kleinen Befehlssatz und kann lediglich Fließkommazahlen verarbeiten. Der große Vorteil einer GPU jedoch ist das sie die Möglichkeit besitzt Berechnungsaufgaben an verschiedene kleinere CO-Prozessoren sogenannte Shader-Units abzugeben. Durch das zuweisen einer Aufgabe pro Shader-Unit erlaubt eine GPU somit das hoch-parallele abarbeiten von Aufgaben - solange diese unabhängig voneinander sind. Diese parallele Programmierung hat jedoch auch Nachteile. Nicht nur das es einer speziellen Programmierung benötigt - sogenannte Shader-Programmierung (Shader-Programme). Sondern es setzt auch Voraus, das jede Shader-Unit das gleiche Shader-Programm ausführt. Besitzen jedoch die zu verarbeitenden Berechnungen genug Unabhängigkeit, so kann eine erhebliche Beschleunigung durch den Einsatz einer GPU welche meist hunderte von Shader-Units besitzt erzielt werden.

### 4.4.2 GPU-Programmierung

GPUs und CPUs besitzen unabhängigen Speicher es benötigt somit nicht nur spezieller Programme, sondern auch den schwierigen Teil der GPU Programmierung - den Datentransport zwischen den Speichern. Zur Vereinfachung nehmen wir in nachfolgenden Kapiteln an, das wir folgendes Viereck zeichnen möchten.



Abbildung 5: Das Viereck

**Indexbuffer**

**GPU-Programmierung**

4.4.3 Datencontainer

## 4.5 Kinect Integration

hier wird die dll erklärt und wie sie eingebunden wird  
kinect baut metrik vom bild um veraenderungen wahrzunehmen  
sendet event nur wenn neues Tiefenbild vorhanden  
tiefenbild blur

## 4.6 Das Terrain

Nachdem das Grundgerüst der Anwendung stand und wir die Kinect Kamera erfolgreich eingebunden hatten ging es daran uns um die Umsetzung der Terraindarstellung zu kümmern.

### 4.6.1 Die Grundgeometrie

Als Grundgeometrie für die Darstellung unseres Terrains erzeugen wir ein flaches reguläres Gitter, dass in der X-Z-Ebene aufgespannt ist (s. Abbildung 6). Die Größe und die Anzahl der Unterteilungen des Gitters wurde variabel gestaltet, um im späteren Verlauf der Entwicklung, einfacher unterschiedliche Konfiguration zu testen.

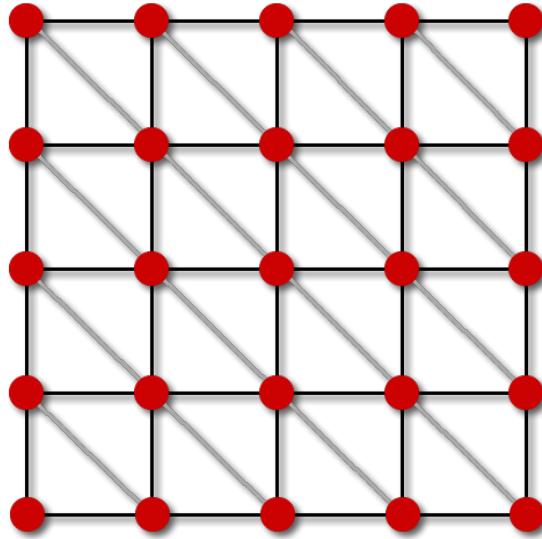


Abbildung 6: Reguläres Gitter des Terrains.

### 4.6.2 Die Höhendaten

Um mit dem zuvor erstellten regulären Gitter ein vollständiges Abbild unserer Sandkastenlandschaft zu repräsentieren, musste nun noch ein Weg gefunden werden die von der Kinect gelieferten Höhendaten in das Model zu integrieren.

Der erste Schritt auf diesem Weg besteht aus einer Vorverarbeitung der gelieferten Daten. Die Kinect liefert uns ein Short-Array mit 307200 Werten, was einer Auflösung von 640 x 480 Pixeln entspricht. Da der Aufbau unseres Sandkastens sowie unseres regulären Gitters allerdings quadratisch ist, müssen wir die gelieferten Daten auch auf dieses Maß beschneiden. Dazu durchlaufen wir das gelieferte Array und extrahieren daraus einen inneren quadratischen Bereich, für den, genau wie bei unserem regulären Gitter, eine variable Größe gewählt werden kann. Abbildung 7 zeigt diesen Vorgang.

Um die vorarbeitenden Daten nun auf das reguläre Gitter zu übertragen, kamen zwei unterschiedliche Ansätze in Frage:

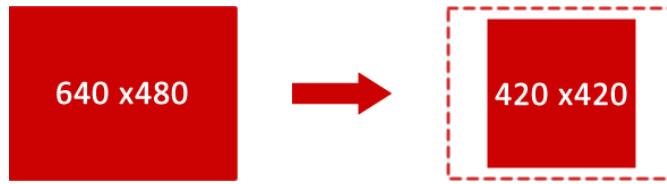


Abbildung 7: Zuschnitt der gelieferten Daten.

## 1. Mit Hilfe der CPU

Beim ersten Ansatz werden die Daten direkt auf der CPU verarbeitet und in das reguläre Gitter integriert. Ein großer Nachteil dieses Ansatzes ist allerdings, dass bei jeder Aktualisierung der sogenannte *VertexBuffer*, mit dessen Hilfe die Geometriedaten an die GPU übertragen werden, komplett neu aufgebaut werden muss. Dieser Vorgang ist sehr kostenintensiv und würde die Echtzeitfähigkeit unserer Anwendung stark einschränken.

## 2. Mit Hilfe der GPU

Beim zweiten Ansatz kann dieser kostenintensive Neuaufbau des Vertexbuffers durch moderne Shader-basierte Verfahren umgangen werden. Dazu wird aus den Daten eine Textur (Heightmap s. Abbildung 8) erzeugt und anschließend mit den Geometriedaten des regulären Gitters zusammen an die GPU übertragen. Im Vertex-Shader kann jetzt mit Hilfe des *Vertex Texture Fetch* (VTF) Verfahrens direkt auf diese Textur und die enthaltenen Höhendaten zugegriffen und für die Manipulation der Y-Position der einzelnen Vertizes genutzt werden.

Entschieden haben wir uns letztendlich für den zweiten Ansatz, da er das weitaus höhere Potenzial zur Echtzeitfähigkeit bietet, welche für unsere Anwendung eine sehr wichtige Rolle spielt und darüber hinaus auch ressourcensparender ist.

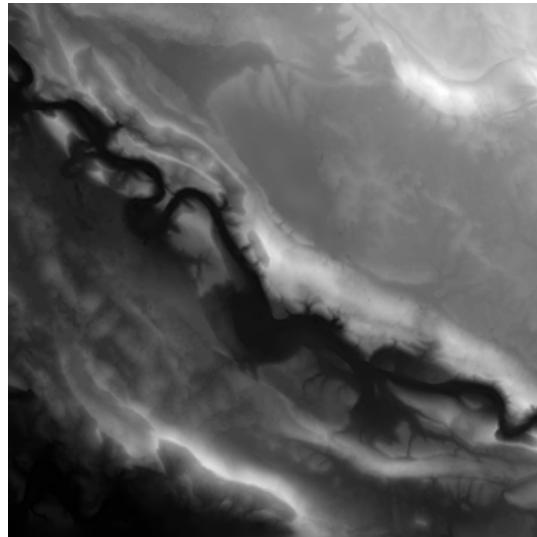


Abbildung 8: In einer Textur abgelegte Höhendaten (Heightmap).

#### 4.6.3 Die Darstellung

Nachdem nun die Grundgeometrie erzeugt, die Höhendaten vor verarbeitet an die GPU übertragen und die Y-Position der Vertices manipuliert wurden, können wir nun unser Terrain endlich darstellen. Einheitlich eingefärbt, bekommen wir allerdings ein Ergebnis dass nicht wirklich an eine Landschaft erinnert (s. Abbildung 9).

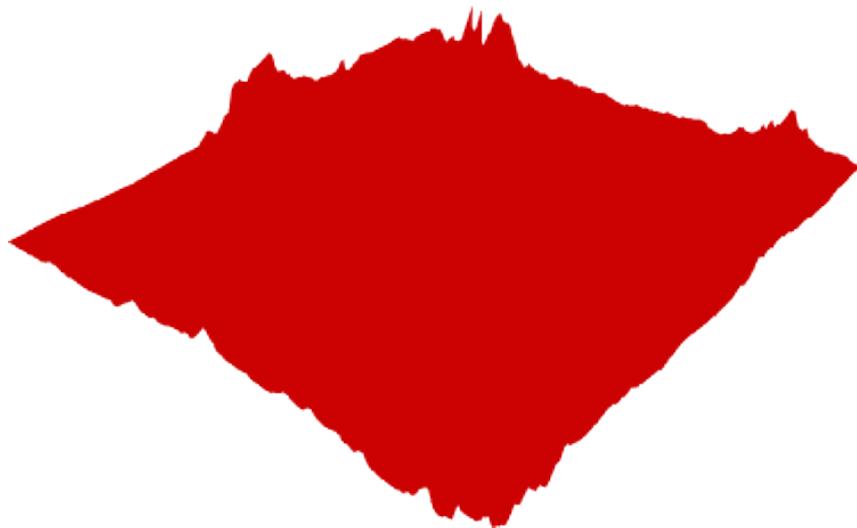


Abbildung 9: Darstellung mit nur einer Farbe.

Dieses Erscheinungsbild lässt sich durch ein fehlendes Beleuchtungssystem und die dadurch fehlende Schattierung der Szene erklären. Da die Implementierung eines kompletten Beleuchtungssystems für unsere Anwendung allerdings wenig Sinn machen würde, lösen wir das oben gezeigte Darstellungsproblem mit Hilfe von verschiedenen Farben für die unterschiedlichen Höhenwerte. Die einfachste Umsetzung dafür wäre direkt die Farbwerte aus der Höhentextur zu nutzen, wodurch ein Graustufenverlauf von dunkel (niedrig) zu hell (hoch) entstehen würde (s. Abbildung 10a). Hierdurch erhalten wir zwar eine korrekte Darstellung unseres Terrains, jedoch sind Graustufen mehr als ungeeignet für die spätere Projektion auf den Sand. Aus diesem Grund haben wir eine Möglichkeit zur benutzerdefinierten Wahl des Farbverlaufs implementiert. Diese besteht aus vier frei wählbaren Farben für vier unterschiedliche Höhenbereiche welche im Shader linear interpoliert werden (s. Abbildung 10b,c).

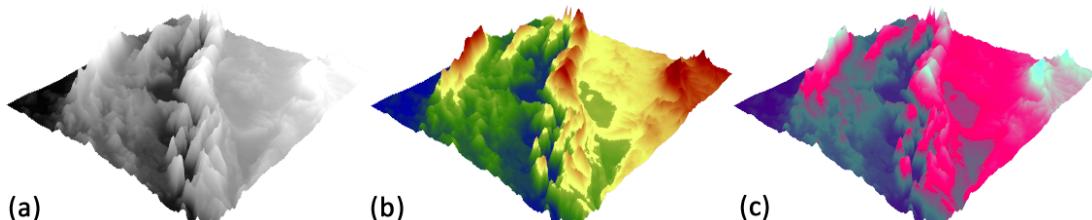


Abbildung 10: (a) Einfärbung anhand der Höhentextur. (b, c) Einfärbung anhand eines benutzerdefinierten Farbverlaufs.

Um die Höhenunterschiede bei der Projektion auf den Sand noch deutlicher erkennbar zu machen, haben wir am Ende des ersten Projektsemesters noch mit der Darstellung von Höhenlinien experimentiert und im Laufe des zweiten Projektsemesters letztendlich vollständig integriert. Die erste experimentelle Version (s. Abbildung 11a) arbeitete ausschließlich auf den reinen Höhendaten an einem einzelnen Punkt, weshalb an manchen Stellen noch sehr großflächige, schwarze Bereiche auftraten. Bei der endgültigen Version (s. Abbildung 11b) flossen schließlich noch zusätzliche Informationen aus den Nachbarbereichen mit in die Berechnung ein, um eine einheitliche Stärke der Höhenlinien zu gewährleisten und größere, schwarze Bereiche auszuschließen.

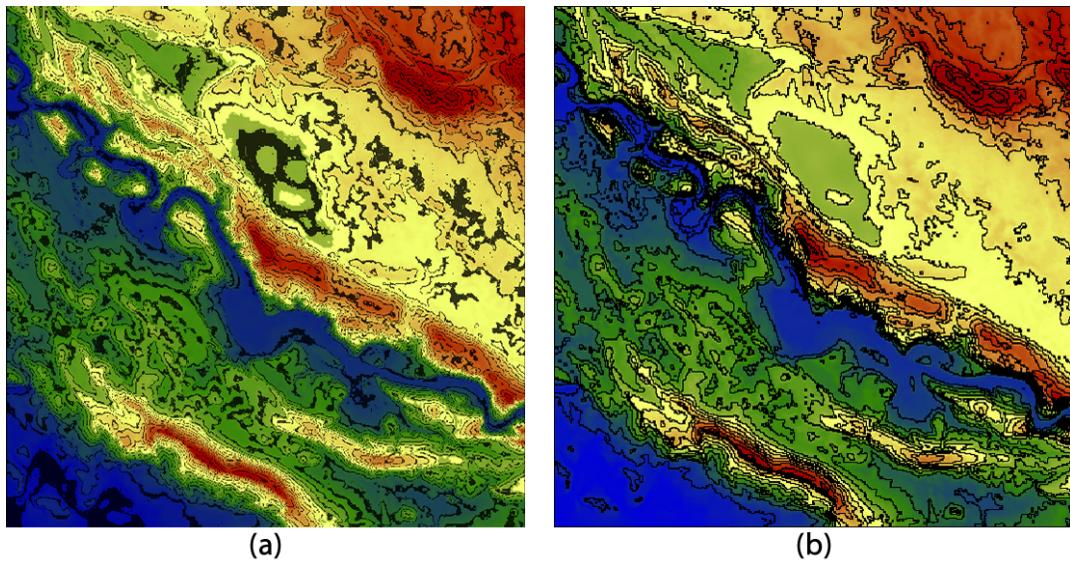


Abbildung 11: (a) 1. experimentelle und (b) endgültige Darstellung der Höhenlinien.

## 4.7 Das Kamerasytem

Um eine Navigation in unserer 3D Szene, sowie eine einfache Art der Kalibrierung zu ermöglichen, wurde ein kleines, erweiterbares Kamerasytem entwickelt. Das System besteht aus zwei Hauptkomponenten, der Kamera-Klasse und der Kamerakontroller-Klasse.

### Kamera

Die Kamera-Klasse stellt alle Grundfunktionen einer virtuellen Kamera zur Verfügung. Dazu gehören neben der Translation und der Rotation auch unterschiedliche Arten der Projektion (Perspektivisch, Orthografisch) und verschiedene Kamera-Modi (Orbital, Walk, Fly) zur Navigation.

Um den sogenannten *Gimbal Lock* zu vermeiden, welcher bei der Verwendung von *Eulerwinkel*n zur Rotation entstehen kann und in speziellen Fällen den Verlust eines kompletten Freiheitsgrades bewirkt, setzen wir in unserem System auf den Einsatz von *Quaternionen* zur Rotation der Kamera. Diese bieten neben der Vermeidung des *Gimbal Locks* auch eine weitaus effizientere Berechnung der Transformationen.

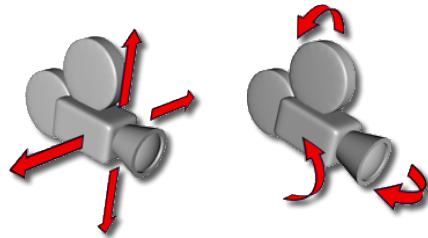


Abbildung 12: Grundfunktionen der Kamera.

### Kamerakontroller

Die Kamerakontroller-Klasse dient als Schnittstelle zwischen der Peripherie und der eigentlichen Kamera und ermöglicht somit eine saubere Trennung zwischen der Verarbeitung von Benutzereingaben und der eigentlichen Funktionalität der Kamera. Abbildung 13 zeigt den groben Aufbau des Kamerasytems.



Abbildung 13: Aufbau des Kamerasytems.

## 4.8 Das Partikelsystem

Nachdem wir im ersten Projektsemester mit unserem CPU-basierten Partikelsystem sehr schnell an die Grenzen des machbaren gestoßen waren, haben wir uns im zweiten Projektsemester kurzfristig dafür entschieden, das System noch einmal komplett zu überarbeiten und dieses Mal auf eine reine GPU Implementierung zu setzen.

### 4.8.1 Die Anforderungen

Als Anforderungen haben wir uns gesetzt, ein hoch flexibles und vom restlichen System getrenntes Partikelsystem zu entwickeln, welches die Fähigkeit bietet mehrere hunderttausend oder sogar Millionen von Partikeln in Echtzeit darzustellen.

### 4.8.2 Die Umsetzung

Um unser angestrebtes Ziel zu erreichen, haben wir auf eine Kombination aus verschiedenen Techniken gesetzt, die wir folgend etwas genauer Beschreiben werden.

#### Billboarding

Für die Darstellung unserer Partikel haben wir zwei unterschiedliche Techniken in Betracht gezogen. Bei der ersten und einfacheren Technik wird ein Partikel durch einen einzelnen Vertex repräsentiert und anschließend, als farbiger Punkt, auf den Bildschirm gezeichnet. Da wir aber nicht auf die Möglichkeit verzichten wollten, bei Bedarf unseren Partikeln auch eine Textur zuzuweisen, haben wir uns für die zweite, etwas aufwendigere, Technik das *Billboarding* entschieden.

*Billboards* bestehen aus zwei dreieckigen Polygonen die ein Rechteck bilden (s. Abbildung 14). Dieses Rechteck wird anschließend im Vertexshader, mit Hilfe der Viewmatrix der Kamera, so transformiert damit es immer in Richtung des Betrachters ausgerichtet ist. Durch diese Eigenschaft, lassen sich, mit sehr geringem Rechenaufwand, unterschiedlichste Effekte realisieren. In unserem Anwendungsfall, die Darstellung von Rauch beziehungsweise Nebel.

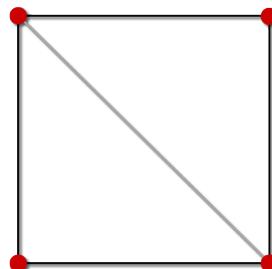


Abbildung 14: Aufbau des Rechtecks für ein Billboard.

## Instancing

Bei dieser Technik handelt es sich, um eine von der Grafikhardware bereitgestellten Funktionalität, zur Reduzierung von sogenannten *Drawcalls*. Unter einem *Drawcall* versteht man im Allgemeinen, das Zeichen eines Objekt mit einem bestimmten Material, einer Transformation und gegebenenfalls weiteren Eigenschaften. In unserem Fall wäre also jedes gezeichnete Partikel (Billboard) ein *Drawcall*. Diese sind allerdings sehr teuer und bei der angestrebten Anzahl von über 1.000.000 Partikeln, wäre an eine Echtzeitfähigkeit nicht mehr zu denken gewesen. Somit entschieden wir uns für das *Instancing*. Diese Technik erlaubt es 1.048.576 Instanzen der gleichen Geometrie, in unserem Fall die Billboards, in nur einem einzigen *Drawcall* zu Zeichen.

## Rendertargets

In unserer vorherigen CPU-basierten Implementierung des Partikelsystems, war es ein leichtes, die benötigten Eigenschaften (Position, Geschwindigkeit, usw.) unserer Partikel, in einer dazu passenden Datenstruktur im RAM abzulegen und zu manipulieren. Bei der Neuimplementierung musste nun ein Weg gefunden werden, die Speicherung und Manipulation der Partikeleigenschaften, performant auf die GPU zu übertragen.

Hierfür entschieden wir uns, für die Nutzung sogenannter *Rendertargets*. Diese repräsentieren eine spezielle Art von Texturen, für die neben dem standardmäßigen Lesezugriff auch Schreibzugriff zur Verfügung steht. Durch diese Eigenschaft, wird es möglich, *Rendertargets* als Datencontainer zu nutzen und deren Inhalt direkt auf der GPU zu manipulieren.

Zusätzlich zu den normalen *Rendertargets*, unterstützt XNA auch *Multiple Rendertargets*. Welche es ermöglichen, während eines Passes, in bis zu vier Rendertargets gleichzeitig zu schreiben.

Mit dieser Technik war es uns nun möglich, unsere einzelnen Partikeleigenschaften in den einzelnen Farbkanälen der vier verfügbaren Rendertargets abzulegen (s. Abbildung 15) und direkt auf der GPU zu manipulieren.

	RED	GREEN	BLUE	ALPHA
RT0	Position X	Position Y	Position Z	Birth Time
RT1	Velocity X	Velocity Y	Velocity Z	Life Time
RT2	Act Size	Start Size	End Size	X
RT3	Color Red	Color Green	Color Blue	Color Alpha

Abbildung 15: Kanalbelegung der einzelnen Rendertargets.

## Ping-Pong

Im Normalfall können *Rendertargets*, innerhalb eines Passes, entweder nur gelesen oder nur geschrieben werden. Um diese Einschränkung zu umgehen und einen weiteren Pass einzusparen, setzen wir das sogenannte *Ping-Pong* Verfahren ein. Bei diesem Verfahren, existiert von jedem *Rendertarget* ein Duplikat (s. Abbildung 16). Während eines Passes wird nun eines der Duplikate genutzt um Daten daraus zu lesen und das andere um die manipulierten Daten zurück zuschreiben. Im Anschluss an den Pass, werden die beiden *Rendertargets* dann einfach ausgetauscht. Somit können im nächsten Pass, die zuvor geschriebenen Daten gelesen werden und die alten Daten können überschrieben werden.

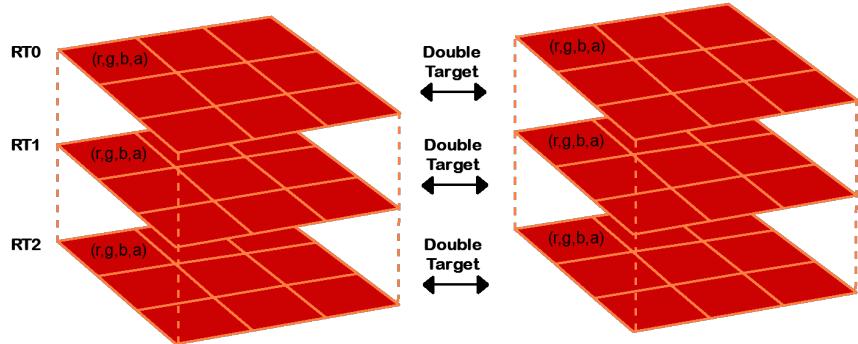


Abbildung 16: Duplierte Rendertargets für Ping-Pong Verfahren.

## Fullscreen-Pass/Offscreen-Pass

Das Update unserer *Rendertargets* und somit die Manipulation unserer Partikeleigenschaften, findet in einem sogenannten *Fullscreen-Pass* statt, welcher vor dem eigentlichen Zeichen der *Billboards* durchlaufen wird. Hierbei wird ein im *Screenspace* positioniertes, Bildschirmfüllendes Rechteck genutzt, um ein 1-zu-1 Mapping des zu lesenden und des zu schreibenden *Rendertargets* zu erreichen. Da dieser Pass keine Bildschirmausgabe zur Folge hat, kann er auch als *Offscreen-Pass* bezeichnet werden.

## Blending

Um bestimmte Effekte realistischer Darzustellen, bietet das neue Partikelsystem, optional die Möglichkeit, additives Blending (s. Abbildung 17) zu aktivieren. Dabei werden die Farben übereinanderliegender Partikel aufsummiert, wodurch Bereiche mit vielen übereinanderliegenden Partikeln heller erscheinen (s. Abbildung 18).

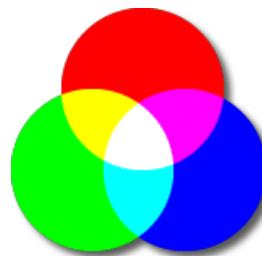


Abbildung 17: Additives Blending.

#### 4.8.3 Das Ergebnis

Das Ergebnis unserer Neuimplementierung war mehr als zufriedenstellend. Am Ende hatten wir ein, vom restlichen System getrenntes und sehr flexibles Partikelsystem entwickelt, welches nun komplett auf der GPU lief und uns somit wieder mehr Ressourcen für andere Aufgaben, auf Seiten der CPU zur Verfügung standen. Auch unser, am Anfang noch für utopisch gehaltenes Ziel, eine Anzahl von über 1.000.000 Partikel in Echtzeit darzustellen, konnten wir mit dem neuen System erreichen. Bei über 1.000.000 Partikeln, läuft unsere Anwendung nun immer noch mit über 100 Frames die Sekunde, solche Ergebnisse konnten wir bei der vorherigen CPU-basierten Implementierung, selbst mit minimaler Anzahl an Partikeln nicht erreichen. Abbildung 18 zeigt das Partikelsystem mit einigen unterschiedlichen Konfigurationen.

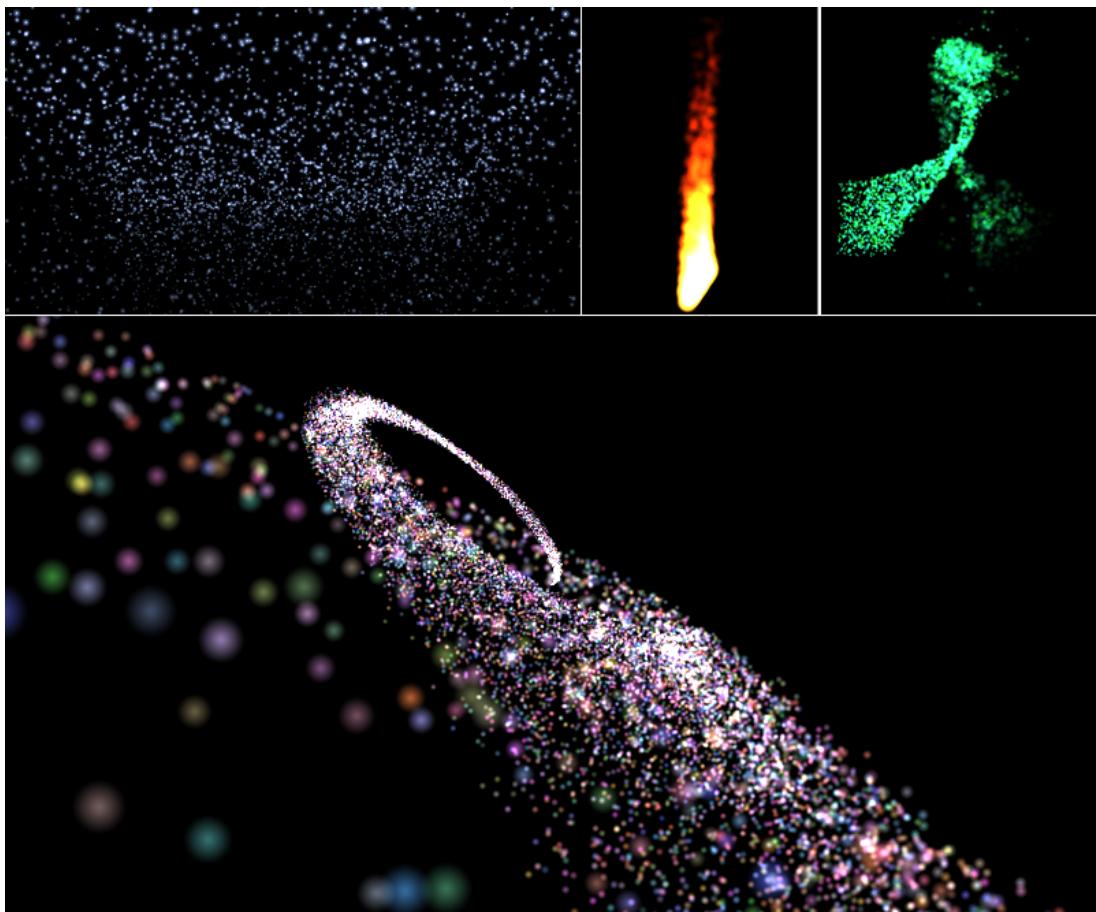


Abbildung 18: Partikelsystem mit unterschiedlichen Konfigurationen.

## 4.9 Die Physik

Die Physik soll eine möglichst echtzeitfähige und realitätsgerechte Simulation von Luftströmung über eine dynamische Landschaft berechnen. Zu diesem Zweck sollten die Partikel des Partikel-systems die einzelnen Luftteilchen darstellen und mithilfe einer einfachen Physik die Interaktion zwischen diesen und ihrer Umgebung simuliert werden.

### 4.9.1 Komponenten

Jedes Luftpartikel besteht aus zwei dreidimensionalen Vektoren. Der Vektor  $\vec{p}_i$  stellt dabei die Position des Partikels im Raum dar und der Vektor  $\vec{v}_i$  beschreibt die aktuelle Bewegung des Partikels. Außerdem besitzen alle Partikel den gleichen Radius  $r$  und einen Reibungskoeffizienten  $\mu$ , welcher bestimmt wie viel Energie ein Partikel bei einer Kollision verliert. Zusätzlich zu den Partikeln stehen Informationen über die Umgebung wie z.B. Höhendaten der Landschaft und verschiedene externe Kräfte ( $F$ ), wie z.B. Gravitation, zur Verfügung.

### 4.9.2 Bewegung der Partikel

Um die Bewegung der Partikel zu simulieren wird in jeder Iteration, für jedes einzelne Partikel, der Bewegungsvektor  $\vec{v}_i(t)$  auf den Positionsvektor  $\vec{p}_i(t)$  aufaddiert.

$$\vec{p}_i(t+1) = \vec{p}_i(t) + \vec{v}_i(t)$$

Auf diese Weise wird der Impuls solange erhalten bis der Bewegungsvektor durch Einwirkung äußerer Kräfte verändert wird.

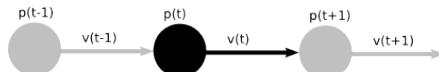


Abbildung 19: Konstante Bewegung

In diesem Fall wird angenommen das der zeitliche Abstand zwischen den einzelnen Iterationen konstant ist. Wenn die Dauer zwischen den Iterationen schwankt ist es ratsam dies zu kompensieren indem man den Bewegungsvektor  $\vec{v}_i$  mit dem zeitlichen Abstand ( $\Delta t$ ) zwischen den Iterationen multipliziert.

$$\vec{p}_i(t + \Delta t) = \vec{p}_i(t) + \vec{v}_i(t) * \Delta t$$

### 4.9.3 Verrechnung externer Kräfte

In jeder Iteration wird die Wirkung externer Kräfte ( $F$ ) auf das Partikel berechnet. Dabei wird jede der externen Kräfte ( $\vec{f}_i$ ) auf den Bewegungsvektor  $\vec{v}_i(t)$  aufaddiert.

$$\vec{v}_i(t+1) = \vec{v}_i(t) + \sum_{\vec{f}_i \in F} \vec{f}_i$$

Eine externe Kraft ist beispielsweise die Gravitation, welche die Partikel in die untere Richtung beschleunigt. Diese Wirkung kann dadurch erzielt werden indem beispielsweise der Vektor  $(0, -9.81, 0)$  zu der Menge der externen Kräfte hinzufügt wird. Diese schrittweise Aufaddierung

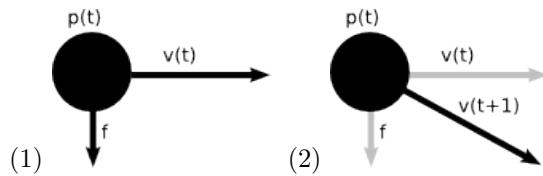


Abbildung 20: Bewegungsvektor vor(1) und nach(2) Modifikation durch eine externe Kraft

führt, über mehrere Iteration hinweg gesehen, zur Beschleunigung des Partikels, wie es in der Abbildung 20 dargestellt wird.

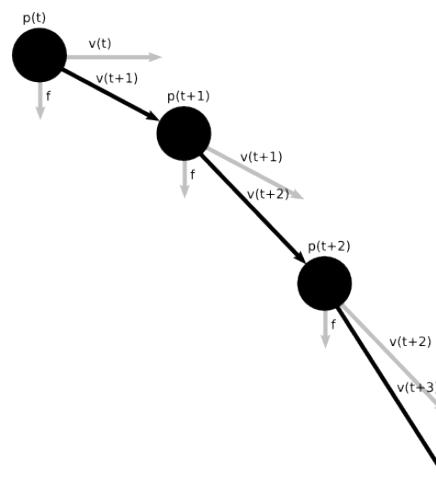


Abbildung 21: Die auf das Partikel wirkende Kraft( $f$ ) bleibt konstant und die Bewegung wird bei jedem Rechenschritt weiter in die Richtung der Kraft beschleunigt

#### 4.9.4 Kollisionserkennung von Partikeln mit der Umgebung

Nachdem ein Partikel bewegt wurde, wird überprüft ob eine Kollision mit der Heightmap vorliegt. Eine Kollision liegt vor wenn die vertikale Komponente des Positionsvektors des Partikels einen niedrigeren Wert enthält als der Höhenwert der Heightmap unterhalb der horizontalen Position des Partikels. Wenn dieser Fall vorliegt befindet sich das Partikel unterhalb der Heightmap und ist folglich während der letzten Positionsänderung in das Terrain eingedrungen und es liegt nun eine Kollision vor.

Zusätzlich kann der Durchmesser des Partikels berücksichtigt werden indem man den Radius zu dem Höhenwert der Heightmap aufaddiert und dadurch eine Kollision erkannt wird bevor der Positionsvektor unterhalb der tatsächlichen Heightmap liegt.

#### 4.9.5 Kollisionsauflösung von Partikeln mit der Umgebung

Wurde erkannt dass das Partikel mit der Heightmap kollidiert ist, wird eine Kollisionsauflösung durchgeführt. Dabei wird zuerst die Flächennormale der Heightmap ( $\vec{n}$ ) an dem Ort der Kollision berechnet.

Mithilfe der Flächennormale  $\vec{n}$  wird nun die Reflektion des Bewegungsvektor berechnet und die Bewegungsrichtung auf diese Weise verändert.

$$\vec{v}_i(t+1) = \vec{v}_i(t+1) - 2 \cdot \langle \vec{v}_i(t+1), \vec{n} \rangle \cdot \vec{n}$$

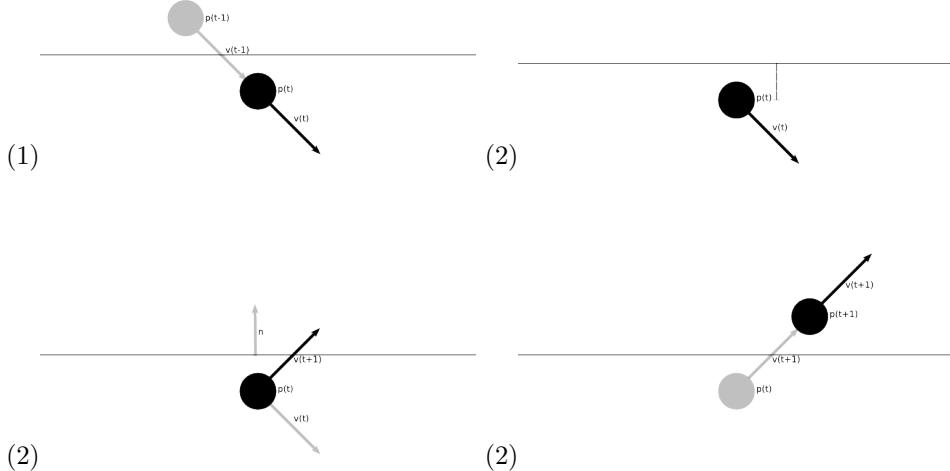


Abbildung 22: (1) Partikel dringt in die Heightmap ein. (2) Erkennung der Kollision. (3) Reflektion des Bewegungsvektors(v) entlang der Flächennormale(n). (4)

Allein mithilfe der Reflektion können bereits die meisten Kollisionen aufgelöst werden. Wenn beispielsweise ein Partikel senkrecht zur Oberfläche in die Heightmap eindringt wird der Bewegungsvektor auf diese Weise umgekehrt und das Partikel wird sich in der nächsten Iteration wieder aus der Heightmap herausbewegen.

Die Kombination aus Reflektion und Gravitation führt außerdem dazu, dass Partikel die auf der Heightmap liegenbleiben sind, sich entlang des Gefälles der Heightmap bewegen, oder im Falle einer horizontalen Ebene auf der aktuellen Position liegenbleiben. Dies wird dadurch verursacht, dass ein liegengebliebenes Partikel einen Nullvektor als Bewegungsvektor besitzt, wodurch nach Verrechnung der Gravitation der Bewegungsvektor dem Vektor der Gravitation entspricht und verursacht dass das Partikel in die Heightmap bewegt wird, wodurch eine Kollision hervorgerufen wird. Wenn die Flächennormale in die entgegengesetzte Richtung der Gravitation zeigt, wie es im Fall einer horizontalen Ebene vorliegt, wird der Bewegungsvektor durch die Reflektion vollständig umgekehrt. In der nächsten Iteration wird das Partikel wieder an seine Ursprungsposition bewegt und die Addition der Gravitation zum Bewegungsvektor führt zur Neutralisierung der Bewegung, wodurch der Bewegungsvektor wieder zu einem Nullvektor wird und somit die Ausgangssituation wiederhergestellt ist.

Wenn allerdings die Kollision an einem Gefälle stattfindet, neutralisieren sich Bewegung und Gravitation nur teilweise. Da die Gravitation keinen Einfluss auf den horizontalen Anteil der Bewegung hat, bleibt dieser erhalten und das Partikel bewegt sich in der nächsten Iteration in die horizontale Richtung der Flächennormale. Auf diese Weise bewegen sich Partikel auf der Oberfläche der Heightmap und folgen dabei der horizontalen Richtung der aktuellen Flächennormale. Bei einem Gefälle führt dieser Vorgang dazu, dass die Partikel sich, wie in Abbildung 21 dargestellt ist, entlang der schießen Oberfläche nach unten bewegen, bis sie schließlich in einem Tal liegenbleiben. Oder im Fall einer horizontalen Ebene auf der Oberfläche

liegenbleiben.

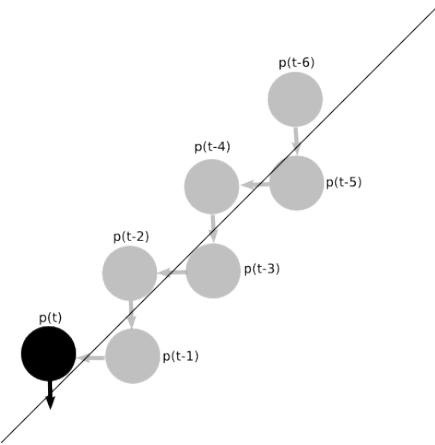


Abbildung 23: Bewegung eines Partikels entlang eines Gefälles.

Nach der Reflektion kann die Reibung einbezogen werden, indem der Bewegungsvektor  $\vec{v}_i$  von dem Reibungskoeffizienten  $\mu$  gekürzt wird. Wobei  $\mu$  einen festen Wert zwischen 0 und 1 besitzen muss und für alle Partikel gleichermaßen gilt.

$$\vec{v}_i(t+1) = \vec{v}_i(t+1) \cdot (1 - \mu)$$

Bei dem Wert 1 würde das Partikel alle Bewegungsenergie verlieren und auf der Aufprallstelle liegenbleiben. Je kleiner dieser Wert jedoch ist desto weniger Energie geht bei einer Kollision verloren und ein Wert von 0 führt dazu, dass die Reibung keinen Einfluss mehr auf den Bewegungsvektor hat.

Die Reibung sorgt dafür, dass die Partikel bei jeder Kollision immer langsamer werden, bis sie schließlich vollständig zum erliegen kommen und nicht dauerhaft ungebremst und durch die gesamte Welt springen. Der Verlust von Bewegungsenergie bedeutet allerdings auch, dass bei einem auf der Landschaft liegengebliebenen Partikel die Reflektion nicht mehr vollständig die Gravitation ausgleichen kann. Der Bewegungsvektor würde in der nächsten Iteration entgegen der Gravitation zeigen, dieser ist allerdings durch die Reibung schwächer als die Gravitation und kann diese nun nicht mehr vollständig neutralisieren. Die Reflektion reicht nun durch die Kombination mit der Reibung, nicht mehr alleine aus um zu verhindern, dass liegengebliebe Partikel immer weiter in die Oberfläche eindringen.

Um dies zu verhindern wir ein weiterer Mechanismus bei der Kollisionsauflösung benötigt.

#### 4.9.6 Kollisionserkennung zwischen Partikeln

Um eine Kollision zwischen zwei Partikeln festzustellen errechnet man den Abstand zwischen beiden indem man beide Positionsvektoren voneinander subtrahiert und den Betrag des daraus resultierenden Vektors ermittelt. Wenn alle Partikel kugelförmig sind und den gleichen Durchmesser besitzen lässt sich eine Kollision sehr einfach ermitteln indem man den Abstand zwischen den Partikeln mit dem doppelten Radius der Partikel vergleicht.

$$|\vec{p}_i - \vec{p}_j| \leq 2r \rightarrow \text{Kollision liegt vor}$$

$$|\vec{p}_i - \vec{p}_j| > 2r \rightarrow \text{Keine Kollision}$$

Solange der Abstand zwischen beiden Partikeln größer ist als der doppelte Radius liegt keine Kollision vor.

#### 4.9.7 Kollisionsauflösung zwischen Partikeln

Wenn eine Kollision zwischen Partikeln festgestellt wurde, kann diese in ähnlicher Weise aufgelöst werden wie eine Kollision mit der Landschaft. Dabei

#### 4.9.8 Übertragung der Kräfte zwischen Partikeln

## 4.10 Das Graphical User Interface

Zur Interaktion und Einstellung der einzelnen Komponenten entschieden wir uns dafür, unserer Anwendung ein *Graphical User Interface* (GUI) zu spendieren.

### 4.10.1 Die Anforderungen

Als Anforderungen setzten wir uns ein übersichtliches und leicht verständliches System, welches uns im Laufe des Projekts ermöglichen sollte, schnell und ohne größeren Aufwand, neue Funktionalitäten hinzuzufügen.

### 4.10.2 Die Umsetzung

Um diese Anforderungen zu erreichen experimentierten wir im ersten Projektsemester mit einem *Multi-Window* Ansatz auf Basis von *Windows Forms*. Dieser Ansatz bestand aus zwei separaten Fenstern (s. Abbildung 22). Ein Fenster, das für die Projektion auf den Sand, mit Hilfe des Beamers genutzt wurde und ein weiteres Fenster für die 3D-Ansicht, Zusatzinformationen und den Bedienelementen zur Anpassung des Systems.

Leider stellte sich recht schnell heraus, dass dieser Ansatz doch nicht ganz so flexibel war, wie anfangs erwartet und das hinzufügen von neuen Bedienelementen jedes Mal mit relative viel Arbeit verbunden war. Zudem bedeuteten die beiden Fenster auch die doppelte Arbeit, da die komplette Szene jeweils zweimal gezeichnet werden musste. Aus diesen Gründen, entschlossen wir uns zu Beginn des zweiten Projektsemesters, diesen Ansatz zu verwerfen und auf ein einzelnes Fenster mit einer GPU-basierten GUI zu setzen.

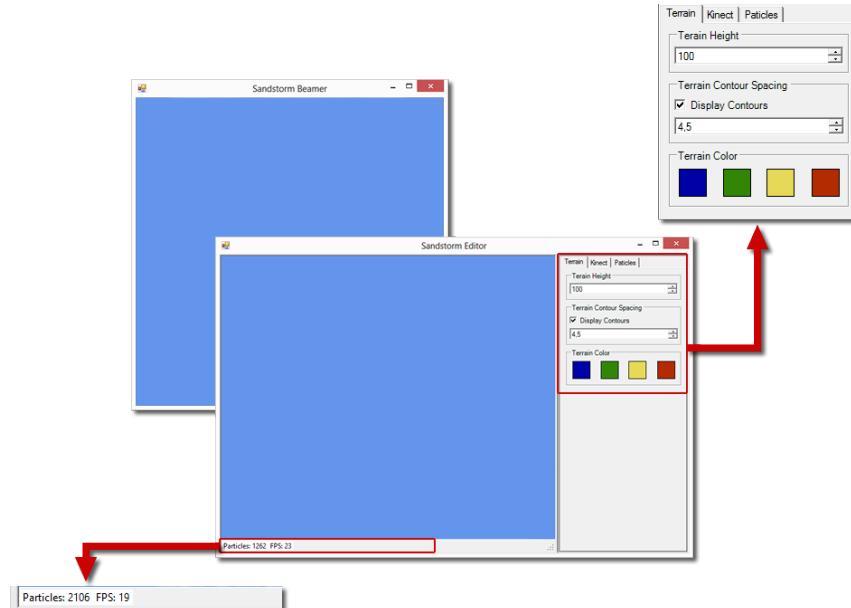


Abbildung 24: Multi-Window GUI aus dem ersten Projektsemester.

## Ruminate GUI

Nach längerer Recherche und Suche nach einer passenden Bibliothek zur Darstellung eines GUI unter XNA, entschieden wir uns letztendlich für die sehr minimalistische, Open-Source Bibliothek *Ruminate* [[Franks2013](#) ]. Diese bot uns Grundlegenden Bedienelemente (s. Abbildung) wie Buttons, Slider, usw. und war weitaus weniger überladen als die meisten anderen Bibliotheken.

Natürlich hatte auch diese Bibliothek nicht nur Vorteile. Da diese von einem Ein-Mann-Team als Hobbyprojekt entwickelt wurde, gab es hier und da noch das ein oder andere Problem und auch das Design der Bedienelemente war nicht das schönste. Dank Open-Source, konnten wir uns aber selbst um diese Probleme kümmern und das GUI individuell an unser System anpassen.



Abbildung 25: Einige Bedienelemente der Ruminate GUI [[Franks2013](#) ].

## Reflections

### 4.10.3 Das Ergebnis

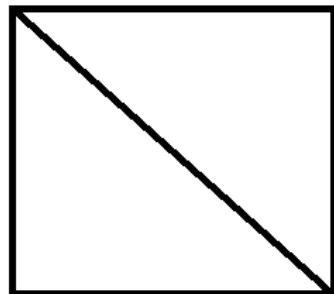


Abbildung 26: Reflection

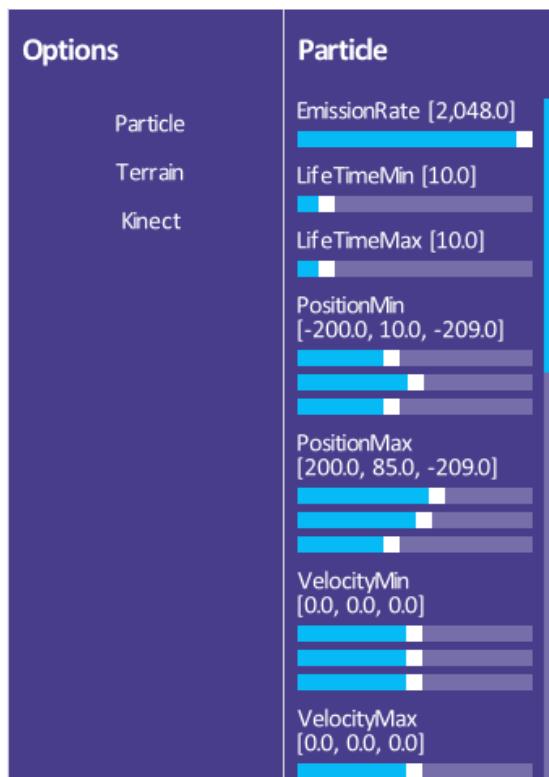


Abbildung 27: Die ingame GUI.

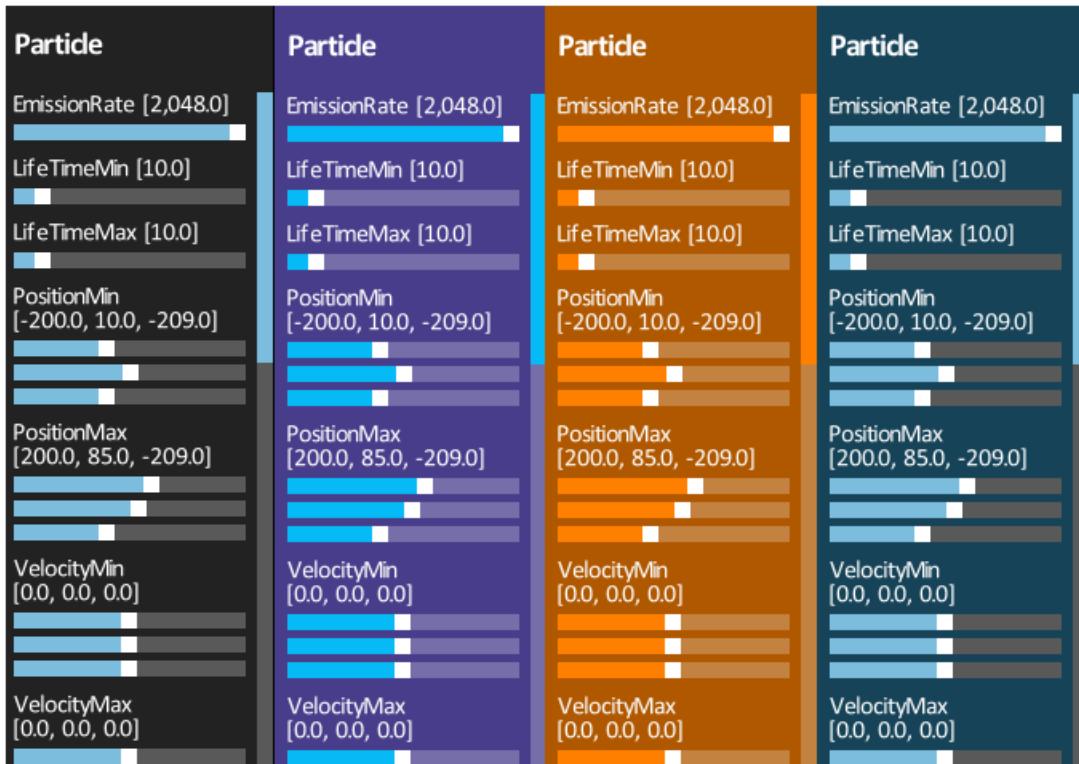


Abbildung 28: Unterschiedliche Erscheinungsbilder der GUI.

## 4.11 Properties

Um unser System so dynamisch wie möglich zu gestalten entschlossen wir uns Sammelcontainer für Systemparameter einzuführen - sogenannte Properties. Diese Properties lassen sich als Konfigurationsdateien ansehen. Ändert man einen Wert innerhalb einer Properties so wird er neue Parameter vom System sofort als neuer gültiger Wert angesehen. Dadurch das jegliche Parameter einer Kategorie (Physikengine, RenderEngine, Kinect..) mit solch einer Properties ausgestattet sind, ist es möglich für verschiedene Anwendungscenarios Default-Wert zu hinterlegen und diese bei Bedarf zu laden oder zu speichern.

## 4.12 Reflection

Im Laufe der Zeit wurde unser Projekt immer größer, dies brachte auch viele neue Funktionalitäten mit sich. All diese neuen Features mussten wir stetig unserer GUI-Oberfläche hinzufügen. Dieser sehr statische Ansatz wurde deshalb durch Reflektion in einen dynamischen überführt. Diverse moderne Programmiersprachen so auch unser verwendetes C-Sharp besitzen die Möglichkeit während des Programmablaufs Informationen über die Struktur eines gegebenen Objekts abzurufen.

Dieser Ansatz und die Tatsache das wir diverse Probleme mit unserer statischen Multi-Window GUI hatten, haben uns dazu bewegt unser altes GUI-System abzulösen. Dank unseren Properties war es uns mittels Reflection möglich ein neues dynamisches GUI System einzuführen,

hierbei verzichteten wir auch auf das Multi-Window System und haben die GUI direkt auf den Sandkasten projiziert.

## 5 Ergebnisse

hier schreiben wir unsere erfahrungen rein undwas wir genau hinbekommen haben. zudem sollen probleme die währed der arbeit aufgetreten sind erwähnt / erläutert werden.

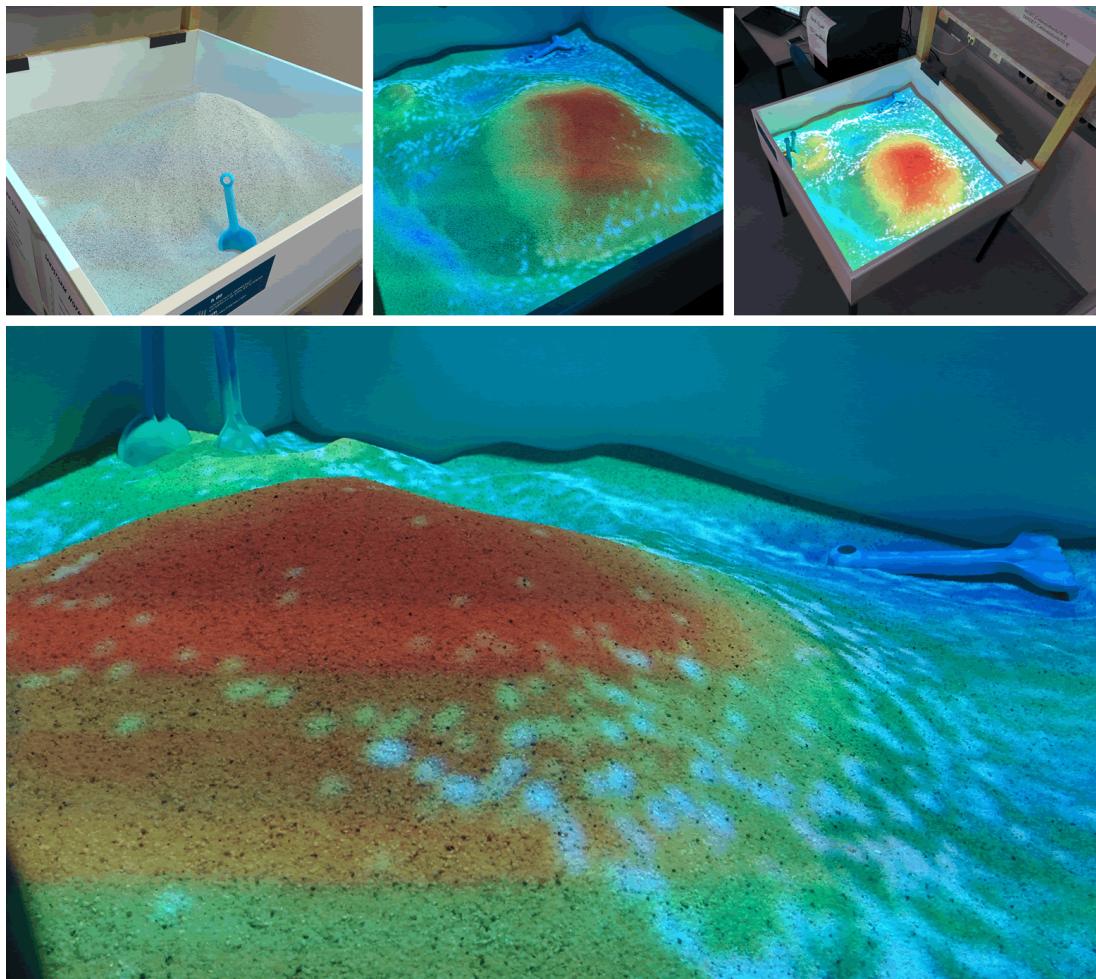


Abbildung 29: Sandstorm Projekt in Aktion.

# 6 Probleme

## 6.1 Echtzeitfähigkeit

Leider besitzt die derzeitige Ausarbeitung diverse kleinere Probleme, welche die Echtzeitfähigkeit des Systems gefährden. Diverse teile von Berechnungen werden noch wie in ?? beschrieben auf der CPU ausgeführt, während der Teil der Visualisierung bereits auf die GPU portiert wurde. Dies führt zu erheblichen Performanceproblemen, denn es muss bei jeder Physikberechnung (jeden Frame), die Partikeldaten zwischen GPU und CPU kopiert und synchronisiert werden.

## 6.2 Darstellung

Die Darstellung stellte sich um Laufe des Projektes als schwieriger heraus als vorher angedacht. Hierbei kann man die Probleme auf welche wir gestoßen sind grob in Hard- und Softwareprobleme unterscheiden.

### 6.2.1 Hardware

Trotz das wir einen Beamer von einem Grafiklabor der Hochschule zur Verfügung gestellt bekommen haben, bemerkten wir bereits bei ersten Tests, das ein großer Farbunterschied zwischen Beamer und Monitor vorhanden ist. Leider scheint das Spektrum unseres Beamers sehr begrenzt zu sein, so das wir einen Farbunterschied zwischen weiß und gelb kaum wahrnehmen können.

### 6.2.2 Software

Durch die physikalische Gegebenheit das Kinekt und Beamer sich an unterschiedlichen Orten befinden, entsteht bei der Projektion zusätzlich zur Verzerrung auch noch das Problem der Verschiebung. Die Kalibrierung stellte sich somit schwieriger heraus als bisher gedacht, deshalb wurden aus zeitlichen Gründen der Fokus auf Aufgaben gesetzt um schnellstmöglich eine lauffähige Version zu erstellen.

## 7 Fazit & Ausblick

Trotz das auf uns allerlei Probleme zukamen, entstand im Laufe eines Semesters eine Echtzeit Sandkastensimulation, die bereits grundlegende Funktionalität bietet. Das Projekt wurde im im Laufe des zweiten Semesters grundlegend neu struktuiert und somit anfängliche Performance Probleme erheblich verbessert. Nicht nur auf die Erweiterbarkeit (Properties) wurde sehr hohen Wert gelegt sondern es wurden viele neue zusätzliche Parameter unserer Engine hinzugefügt. Trotz der Tatsache, das man während der Entwicklung sehr schnell auf viele neue Ideen kommt, die man leider dann doch Aufgrund von Zeitmangel garnicht alle umsetzen kann haben wir uns nicht untekriegen lassen und so viel es ging umgesetzt. Durch diverse DirectX-9 probleme stoßen wir jedoch - wie bereits erwähnt, relativ schnell an die GPU-Grenzen des XNA-Frameworks. Eine Neuprogrammierung haben wir jedoch ausgeschlossen - denn es musste eine stabile lauffähige Version bis zum start der Hobit fertig gestellt sein. Dieses Projekt beinhaltet sehr viele Möglichkeiten und sollte in der Zukunft fortgeführt werden, jedoch benötigt es einen sehr großen Aufwand an Einarbeitungszeit deswegen wäre es schön wenn zukünftige Fächer der Hochschule bereits Grundlegende Kenntnisse im Bereich der GPU-Programmierung legen könnten.