

Lab Report

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

„Was Sandkastenspiele mit der Frischluftzufuhr in Städten zu tun
haben“

vorgelegt von

T. Sturm (709794)

A. Holike (724986)

S. Arthur (715720)

M. Djakow (718531)

01.05.2013

Inhaltsverzeichnis

1	Einleitung	2
2	Bestehende Arbeiten	3
3	Konzept	4
4	Grundlagen	5
4.1	Mathematische Verfahren	5
5	Realisierung	6
5.1	Hardwareaufbau	6
5.2	XNA Game Studio 4.0 + Kinect SDK	7
5.3	Kinect Integration	8
5.4	Der Renderer	9
5.5	Das Kamerasystem	10
5.6	Terrain	11
5.7	Partikelsystem	15
5.8	Die Benutzeroberfläche	16
6	Zusammenfassung	17
7	Probleme	18
7.1	Echtzeitfähigkeit	18
7.2	Darstellung	18
8	Ausblick	18

1 Einleitung

hier kommt einleitendes gequatsche

2 Bestehende Arbeiten

wie wurde das in den usa gemacht

[**Kreylos2010**]

was gibt es für ähnliche ansätze

beispiele für kinect

beispiele für xny

beispiele für partikelsystem

3 Konzept

wie sieht unser konzept aus

4 Grundlagen

Hier kommt immer die Kapitelüberschrift hin, ein kleines Vorgeplänkel was im Kapitel behandelt wird.

4.1 Mathematische Verfahren

4.1.1 Verzerrung von Bildern

4.1.2 Billboarding

Um unsere Vorgabe der Echtzeitfähigkeit zu erfüllen benötigt es ein paar Tricks, die es erlauben die Komplexität unseres Renderers zu minimieren, gleichzeitig jedoch darf dem Zuschauer diese Manipulation nicht bemerken. Eine beliebte Technik hierfür ist das Billboarding. Die Idee des Billboardings basiert darauf, komplexe geometrische 3D-Objekte auf ein zweidimensionales Rechteck das sogenannte Billboard runterzubrechen. Bei dem Billboard handelt es sich meist um ein vorher berechnetes Bild von dem ursprünglich darzustellenden 3D-Objekts. Anschließend wird dieses Billboard zur Kamera ausgerichtet, dem Zuschauer fällt es somit sehr schwer zu erkennen, das es sich bei dem gezeigten Objekt um eine zweidimensionale Kopie des 3D-Objektes handelt. Diese Technik wird hauptsächlich dazu verwendet die benötigten Rechenoperationen für Objekte welche in der Ferne liegen zu minimieren. Kommt die Kamera dem tatsächlichen Objekten sehr nahe, wird meist mit einer Interpolation zwischen dem Billboard und dem tatsächlichen 3D-Objekt umgeschaltet.

5 Realisierung

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

5.1 Hardwareaufbau

Die Hardware

Die Konstruktion

Der Aufbau

5.2 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* Controller.

XACT

XACT(Microsoft Cross-Platform Audio Creation Tool) stellt einfache Schnittstellen zur Audiowiedergabe 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.

5.3 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

5.4 Der Renderer

wie wird es erzeugt

einfärbung

höhenlinien

5.5 Das Kamerasystem

Um eine Navigation in unserer 3D Szene, sowie eine einfache Art der Kalibrierung zu ermöglichen, wurde ein kleines, erweiterbares Kamerasystem 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 *Eulerwinkeln* zur Rotation entstehen kann und in speziellen Fällen den Verlust eines kompletten Freiheitsgrades bewirkt, setzten 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.

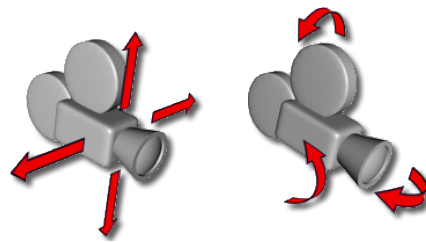


Abbildung 1: Grundfunktionen der Kamera.

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 2 zeigt den groben Aufbau des Kamerasystems.



Abbildung 2: Aufbau des Kamerasystems.

5.6 Terrain

Nachdem wir die Kinect eingebunden hatten, das Grundgerüst des Renderers stand und wir mit Hilfe unseres Kamerasystem in der 3D Szene navigieren konnten, ging es daran uns um die Umsetzung der Terraindarstellung zu kümmern.

5.6.1 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 3). Die Größe und die Anzahl der Unterteilungen des Gitters wurde variabel gestaltet, um im späteren Verlauf der Entwicklung, einfacher unterschiedliche Konfiguration zu testen.

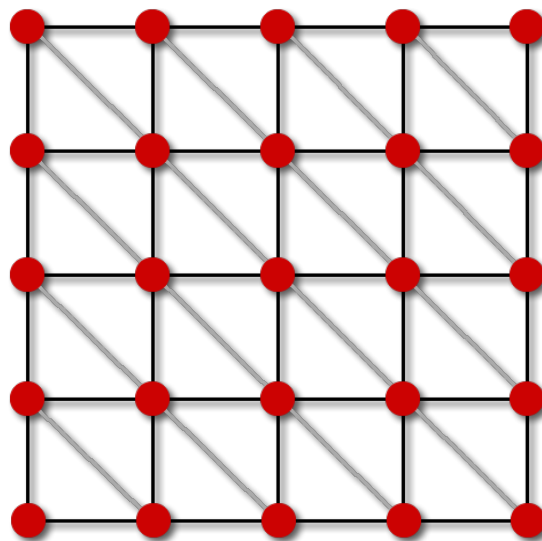


Abbildung 3: Reguläres Gitter des Terrains.

5.6.2 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 4 zeigt diesen Vorgang.



Abbildung 4: Zuschchnitt der gelieferten Daten.

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

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 5) 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 Vertices 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.

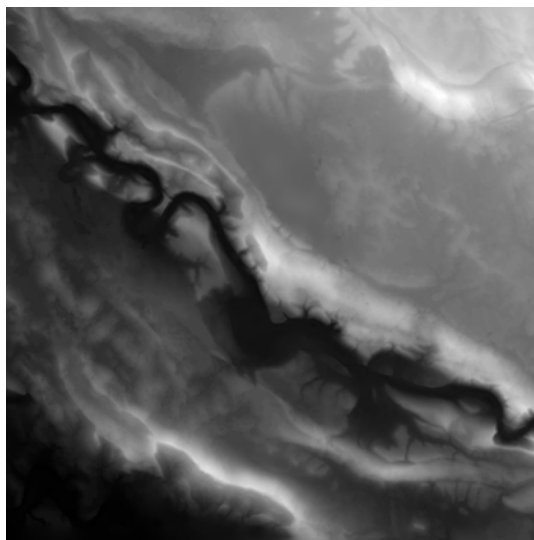


Abbildung 5: In einer Textur abgelegte Höhendaten (Heightmap).

5.6.3 Die Darstellung

Nachdem nun die Grundgeometrie erzeugt, die Höhendaten vorverarbeitet an die GPU übertragen und die Y-Position der Vertices manipuliert wurden, können wir unser Terrain endlich nun darstellen. Einheitlich eingefärbt, bekommen wir allerdings ein Ergebnis, das nicht wirklich an eine Landschaft erinnert (s. Abbildung 6).



Abbildung 6: 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 7a). 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 7b,c).

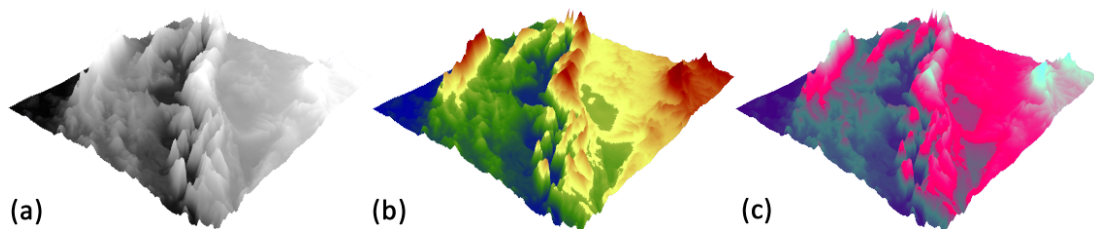


Abbildung 7: (a) Einfärbung anhand der Höhentextur. (b, c) Einfärbung anhand eines benutzerdefinierten Farbverlaufs.

Um die Höhenunterschiede bei der Projektion auf den Sand noch deutlicher zu machen, haben wir am Ende des Semesters noch mit der Darstellung von Höhenlinien experimentiert (s. Abbildung 8). Diese verbinden in bestimmten Abständen alle Punkte mit den gleichen Höhenwerten. Die aktuelle Implementierung ist noch nicht perfekt und benötigt noch einiges an Nachbearbeitung, aber schon jetzt kann man die Vorteile bei der Projektion auf den Sand erkennen.

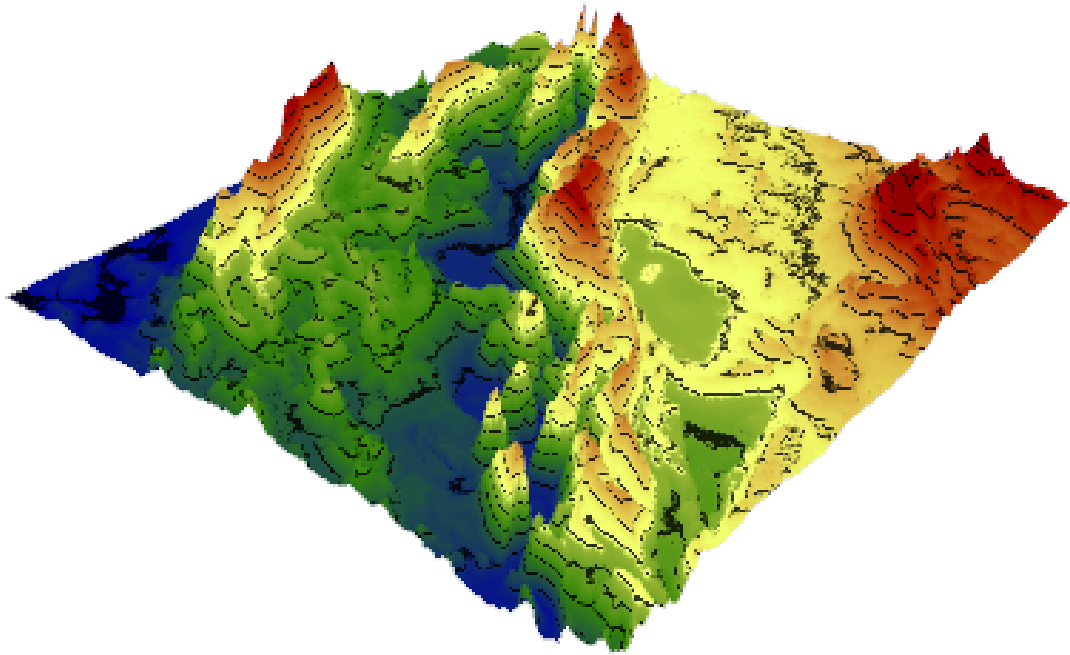


Abbildung 8: Experimentelle Darstellung mit zusätzlichen Höhenlinien.

5.7 Partikelsystem

Es gibt verschiedene Möglichkeiten ein Partikelsystem zu realisieren. In unserem Anwendungsfall haben wir uns dafür entschieden Visualisierung und Physik zu trennen. Dies führt dazu, dass es möglich ist, ohne große Abhängigkeiten voneinander parallel zu entwickeln.

5.7.1 Die Physik (Physik-Engine)

Damit die Partikel möglichst realitätsgetreu sich durch die Szene bewegen, benötigt der Computer die Information wie sich die entsprechenden Partikel verhalten sollen. Dies bedeutet dass physikalische Gesetze auf mathematische Funktionen abgebildet werden müssen. Durch diese Abbildung werden die Bewegungsabläufe der Partikel gesteuert. Um die Echtzeitfähigkeit des Systems sicher zu stellen müssen meist die Abbildungen (Gesetze) durch ein paar Tricks vereinfacht werden um Rechenkraft zu sparen.

Die Physik-Engine hat genau diese Aufgaben; Sie bewegt die Partikel durch den Raum, erkennt Kollisionen und bildet Gesetze der ?? ab. Durch die hohe Unabhängigkeit von den einzelnen Partikeln eignet sich eine GPU besonders gut zum berechnen entsprechender Gesetze. Wir haben uns dennoch im aktuellen Projektstatus dazu entschieden die Berechnungen auf der CPU durchzuführen. Der Grund hierfür liegt in der Problematik dass es nicht möglich ist Berechnungen welche auf der GPU stattfinden genauer zu untersuchen bzw. zu Debuggen.

5.7.2 Der Renderer (Draw-Engine)

Der Renderer ist der zweite Teil unseres Partikelsystems. Seine Aufgabe liegt darin, für jedes einzelnes Partikel die Eigenschaften (Farbe,Kraft,Größe,...) für den Benutzer zu visualisieren. Hierbei wird lediglich lesend auf den vorhandene Datenbestand zugegriffen. Auch hier ist eine hohe Parallelität möglich, denn jedes Partikel stellt eine unabhängige Einheit dar. Somit ist es möglich mit Hilfe von DirectX einen Shader für die GPU zu schreiben. Dieser erlaubt es, dass alle Shader-Units der GPU zusammen an einem Frame arbeiten.

5.8 Die Benutzeroberfläche

Nachdem die Grundfunktionalität der Hauptkomponenten unseres Systems standen ging es nun daran eine einfache aber dennoch funktionale Benutzeroberfläche zu entwerfen. Da das *XNA-Framework* von Haus aus auf *Windows-Forms* zur Darstellung von Benutzeroberflächen setzt, beschlossen auch wir vorerst diese Variante zu nutzen. Hielten uns aber die Möglichkeit offen eventuell später auf das etwas modernere *WPF*-System zu wechseln.

Hauptanforderungen waren ein übersichtliches Design und ein einfaches Hinzufügen von neuen Funktionalitäten. Um diese Anforderungen zu erfüllen entschieden wir uns für eine schlichte Statusleiste am unteren Rand des Editor-Fensters für einfache Anzeigen wie zum Beispiel die Frames Pro Sekunde(FPS) oder die Anzahl der Partikel und ein Tab-Panel an der rechten Seite des Editor-Fensters zur Konfiguration der einzelnen Komponenten. Durch die Nutzung des Tab-Panel lässt sich eine gute Separierung der einzelnen Komponenten in der Benutzeroberfläche realisieren.

Die Kommunikation zwischen der Benutzeroberfläche und den einzelnen Komponenten ist über das im *.Net-Framework* integrierte Event-System realisiert. Bei einer Interaktion mit der Benutzeroberfläche wird ein entsprechendes Event gefeuert, welches anschließend die benötigten Daten an alle Komponenten liefert, die sich zuvor für dieses Event registriert haben.

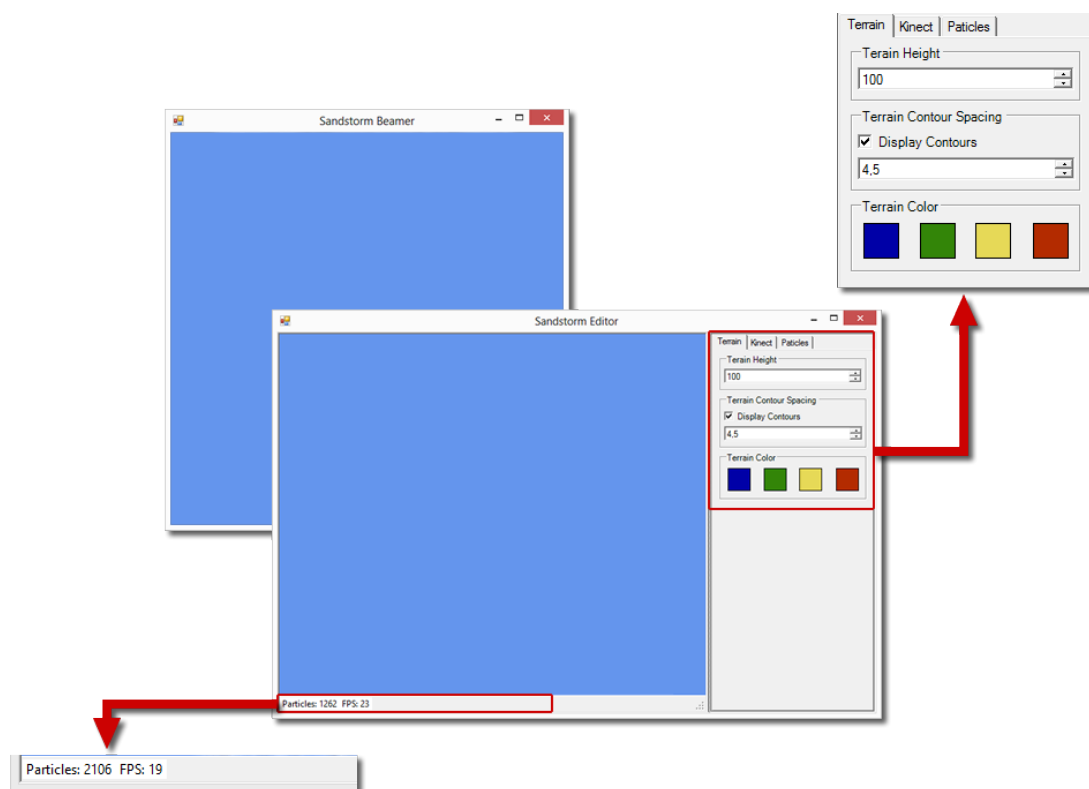


Abbildung 9: Die GUI

6 Zusammenfassung

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

7 Probleme

7.1 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.

7.2 Darstellung

Die Darstellung stellte sich im 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.

7.2.1 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.

7.2.2 Software

Durch die physikalische Gegebenheit das Kinect 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.

8 Ausblick

Trotz das auf uns allerlei Probleme zukamen, entstand im Laufe eines Semesters eine Echtzeit Sandkastensimulation, die bereits grundlegende Funktionalität bietet. Im Laufe des nächsten Semesters werden wir dann Aufgaben, welche in diesem Semester ein wenig vernachlässigt wurden wie z. B. die Kalibrierung nachbessern. Des Weiteren werden wir die bisherigen Physikberechnungen auf die GPU portieren um so hoffentlich wieder die Echtzeitfähigkeit des Sy-

stems zu erlangen. Auch neue Funktionalitäten sind geplant, welche notwendig sind um unser eigentliches ?? zu erreichen.

