

Philipps-Universität Marburg

Fachbereich 12 - Mathematik und Informatik



Diplomarbeit

Progressive Meshes mit CUDA

von
René Frank
Dezember 2009

Betreuer:
Prof. Dr. Thorsten Thormählen
und
Prof. Dr. Manfred Sommer

Arbeitsgruppe Grafik und Multimedia Programmierung

Erklärung

Ich Max Mustermann(Informatikstudent an der Philipps-Universität Marburg, Matrikelnummer xxxxxx), versichere an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die hier vorliegende Diplomarbeit wurde weder in ihrer jetzigen noch in einer ähnlichen Form keiner Prüfungskommission vorgelegt.

Marburg, 17 Dezember 2009

Max Mustermann

Zusammenfassung

Viele der in der Computergrafik verwendeten 3D-Modelle werden mit Hilfe der Dreiecksnetze repräsentiert. ... (max. 1 Seite)

Abstract

text text text text text text text text text text text text text text text text
text (exakte englische Übersetzung der deutschen Kurzfassung)

Inhaltsverzeichnis

Inhaltsverzeichnis	I
1. Einleitung	1
1.1. Motivation	1
1.2. Ziele	1
1.3. Aufbau der Arbeit	2
1.4. Verwandte Arbeiten	2
2. Grundlagen	5
2.1. Grundlage 1	5
2.1.1. subsection	5
2.2. Grundlage 2	5
2.3. Grundlage 3	5
3. Eigenes Verfahren	7
3.1. LaTeX-Editoren	7
3.2. Beispiel für eine Tabelle	7
3.3. Formel in Latex und Konventionen zur Verwendung mathematischer Symbole	7
3.4. Beispiel für eine Vektorgrafik	8
3.5. Beispiel für eine Vektorgrafik mit mathematischen Symbolen	9
3.6. Beispiel für eine Rastergrafik	9
3.7. Beispiel für die Darstellung von Algorithmen	11
3.8. Beispiel für die Darstellung von Quellcode-Auszügen	11
4. Ergebnisse und Evaluation	13
5. Zusammenfassung und Ausblick	15
Abkürzungsverzeichnis	17
Abbildungsverzeichnis	19
Tabellenverzeichnis	20
Liste der Algorithmen	22
Listings	24

A. Anhang	26
Danksagung	27

1

Kapitel 1.

Einleitung

Diese Diplomarbeit beschäftigt sich mit den Parallel View-Dependent Compressed Progressive Meshes und deren Umsetzung in die vom Grafikkartenhersteller NVIDIA entwickelte parallele Programmiersprache CUDA. Dazu gehört die Entwicklung einer für die parallele Verarbeitung geeignete effiziente Datenstruktur, sowie eine effiziente Datenverwaltung.

1.1. Motivation

Die aktuelle Entwicklung der Multimediaindustrie versucht zunehmend die Simulation von virtuellen Welten realistisch darzustellen. Die Ansprüche der Anwender werden mit der Zeit immer größer und dementsprechend die generierte virtuelle Realität immer komplexer. So eine Entwicklung ist unweigerlich mit der Steigerung der erforderlichen Rechenleistung verbunden, da die simulierten Objekte aus Millionen von Polygonen bestehen können und in Echtzeit dargestellt werden müssen. Im Laufe der Jahre sind viele verschiedene Verfahren entwickelt worden, die das Ziel hatten, die komplexen Objekte mit einem vertretbaren Qualitätsverlust in Echtzeit darzustellen. Der mit Abstand beste Ansatz, um den Kompromiss zwischen Qualität und Geschwindigkeit zu finden, ist die View-Dependent Progressive Meshes. Einer der Vorteile dieser Herangehensweise ist, dass dieses Verfahren hochgradig parallelisierbar ist, so dass sich mit einer geeigneter Programmiersprache und Hardware eine beachtliche Effizienzsteigerung erzielen lässt. Die von NVIDIA entwickelte parallele Programmiersprache CUDA setzt auf den aktuellen Trend der GPGPUs und ermöglicht es mit einer kostengünstigen Grafikkarte, die in fast jeden Desktoprechner vorhanden ist, Programme effizient zu parallelisieren. Aus diesem Grund ist CUDA für das Parallelisieren von View-Dependent Progressive Meshing besonders geeignet.

1.2. Ziele

Das Ziel dieser Arbeit ist die Entwicklung einer effizienten parallelen Implementierung von komprimierten View-Dependent Progressive Meshes in CUDA, welche in der Lage ist, Objekte die aus mehreren Millionen von Polygonen bestehen können, in Echtzeit zu verarbeiten.

Echtzeit

Das entwickelte Programm soll selbst sehr große Polygonnetze effizient verarbeiten können. Die Eingaben des Benutzers für die Translation und Rotation des Objekts sollen

in Echtzeit umgesetzt werden. Die durchschnittliche Laufzeit des Programms pro Frame soll höchstens drei Mal soviel Zeit wie das Rendering des gegebenen Modells benötigen, um eine Echtzeitdarstellung des Modells zu ermöglichen. Dabei können die Modelle aus mehreren Millionen von Dreiecken bestehen.

Kosten

Das Programm sollte mit der normalen, kostengünstigen Privatanwender-Hardware laufen, sodass für die Ausführung keine Spezialrechner benötigt werden. Die einzige Voraussetzung an das System ist eine NVIDIA-Grafikkarte die CUDA 1.1 unterstützt. Diese ist aber in den meisten Desktoprechnern vorhanden oder kann kostengünstig nachgerüstet werden.

1.3. Aufbau der Arbeit

Im ersten Abschnitt des Kapitels ?? soll zunächst die Bedeutung der Grafikkarte als Berechnungseinheit verdeutlicht werden. Dann soll im zweiten Abschnitt die Hard- und Softwarearchitektur der Programmiersprache CUDA beschrieben werden, sowie einige Vorschläge zu Codeoptimierung diskutiert, bevor im Kapitel ?? ein Überblick über die wichtigsten Verfahren zur Echtzeitdarstellung komplexer Objekte geben wird. An dieser Stelle werden auch das View-Dependent Progressive Meshing, sowie einige Simplifizierungstechniken genauer erläutern. Kapitel ?? beschäftigt sich mit der Theorie des im Rahmen dieser Diplomarbeit entwickelten Algorithmus. Dabei sollen die Datenstrukturen, die Kompression, sowie die einzelnen Schritte des Algorithmus genauer erläutert werden. Die Implementierung des Algorithmus in CUDA wird im Kapitel 5 besprochen, dabei sollen die benutzten Bibliotheken, sowie die CUDA-spezifische Umsetzung des Programms beschrieben werden. Anschließend werden im Kapitel 6 die durchgeführten Tests und deren Ergebnisse dokumentiert und diskutiert, sowie im Kapitel 7 ein Ausblick auf weiterführende Arbeiten gegeben.

1.4. Verwandte Arbeiten

Im Themenbereich der Progressive Meshes und View-Dependent Progressive Meshes gab es schon am Ende des letzten Jahrzehnts einige Veröffentlichungen [?, ?]. Diese waren zwar eine gute und notwendige Weiterentwicklung vom klassischen LOD-Algorithmus, ermöglichten aber nicht eine effiziente Echtzeitdarstellung von großen Modellen. In [?] wurde schließlich ein Versuch unternommen eine effizientere Datenstruktur zu entwickeln, um den Speicherverbrauch zu optimieren und bessere Geschwindigkeit zu erreichen. Diese effizientere Datenstruktur brachte zwar einige Verbesserungen, ermöglichte aber dennoch keine Echtzeitdarstellung von großen Modellen. Seit dem gab es eine Reihe von Verfahren, die das Ziel hatten eine effiziente Echtzeitdarstellung von großen Modellen zu ermöglichen. Einige von diesen Verfahren nutzten Multi-Triangulationen [?], andere Versuchten die View-Dependent Progressive Meshes weiterzuentwickeln [?, ?, ?]. Doch keins dieser Verfahren konnte die Anforderungen vollständig erfüllen.

Eine erst kürzlich veröffentlichte Arbeit [?] machte endlich einen Schritt in die richtige Richtung. Die in dieser Arbeit implementierte GPU-Variante von parallelen View-Dependent Progressive Meshes ermöglichte eine akzeptable Echtzeitdarstellung von großen Modellen. Diese braucht durchschnittlich das dreifache der Zeit, die für das Rendering des Modells benötigen wird und lässt somit einen großen Spielraum für die Optimierung offen.

2 **Kapitel 2.**

Grundlagen

text

2.1. Grundlage 1

2.1.1. subsection

text

2.2. Grundlage 2

Wie schon in Kapitel 2.1 beschrieben ...

2.3. Grundlage 3

text

Kapitel 3.

Eigenes Verfahren

In diesem Kapitel soll das eigene Verfahren beschrieben werden. Es geht dabei nicht nur darum zu beschreiben was gemacht wurde, sondern ebenfalls darum zu begründen, weshalb bestimmte Design-Entscheidungen getroffen wurden.

3.1. LaTeX-Editoren

Ein guter Cross-Plattform (Windows/Linux/Mac) Latex-Editor mit englischer und deutscher Rechtschreibkorrektur ist z.B. TexStudio (<http://texstudio.sourceforge.net/>). Unter Windows verwende ich diesen Editor gerne zusammen mit dem Sumatra PDF Viewer (<http://blog.kowalczyk.info/software/sumatrapdf/free-pdf-reader.html>), da dieser ein automatisches Neuladen unterstützt.

3.2. Beispiel für eine Tabelle

In Tabelle 3.1 sind die verwendeten Bibliotheken aufgelistet.

Bibliothek	Version
CUDA SDK	2.3
CUDA Toolkit	2.3
OpenGL	3.2
GLUT	3.7
GLEW	1.5.1
CUDPP	1.1

Tabelle 3.1.: Die verwendeten Bibliotheken.

3.3. Formel in Latex und Konventionen zur Verwendung mathematischer Symbole

Mathematische Symbole können in Latex sehr leicht erzeugt werden. Beispiel für Symbole im Text: Gegeben sei ein Skalar $a \in \mathbb{R}$. Tabelle 3.2 liste einige Konventionen zur Verwendung mathematischer Symbole. Abgesetzte Formel lassen sich ebenfalls leicht erzeugen:

$$f(x) = x^2 + 3 \tag{3.1}$$

Außerdem kann leicht auf abgesetzte Formel verwiesen werden (siehe Gleichung 3.1). Für mehrere ausgerichtete Formeln bietet sich die Umgebung `eqnarray` an:

$$f(x) = x^2 + 3 \quad (3.2)$$

$$g(\theta) = \cos(2\theta) = \cos^2 \theta - \sin^2 \theta \quad (3.3)$$

$$h(x) = \int_0^\infty e^{-x} dx \quad (3.4)$$

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$f, g(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$
Matrizen	Schreibmaschine	$\mathbf{A}, \mathbf{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, \{a, b\} \in \mathcal{B}$
Zahlenbereiche	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Tabelle 3.2.: Konventionen zur Verwendung mathematischer Symbole

3.4. Beispiel für eine Vektorgrafik

Wenn möglich sollten immer Vektorgrafiken verwendet werden. Rastergrafiken sollten nur dann eingesetzt werden, wenn die Original-Quelle ebenfalls eine Rastergrafik ist. Ein Cross-Plattform Editor zur Erstellung von Vektorgrafiken ist z.B. Inkscape: <http://inkscape.org/download/>. Nach der Erstellung in Inkscape sollte die Grafik zum einen zur späteren Weiterverarbeitung als SVG gespeichert werden. Zum anderen zwecks Import in Latex als PDF. Abbildung 3.1 zeigt ein Beispiel.

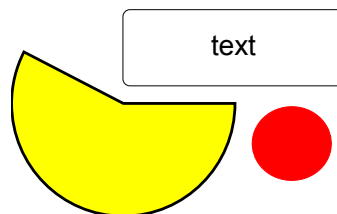


Abbildung 3.1.: Eine Vektorgrafik

3.5. Beispiel für eine Vektorgrafik mit mathematischen Symbolen

Um beliebigen Latex-Code in eine Vektorgrafik einzufügen (z.B. um mathematische Symbole zu setzen) kann in Inkscape beim Speichern der Datei als PDF die Option „Pdf+Latex: Text in PDF weglassen und Latex Datei erstellen“ angewählt werden (siehe Abb. 3.2)

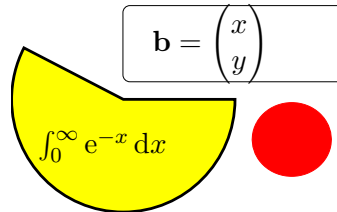


Abbildung 3.2.: Eine Vektorgrafik mit mathematischen Symbolen

3.6. Beispiel für eine Rastergrafik

Abbildung 3.3 zeigt, wie eine Rastergrafik eingebunden werden kann.

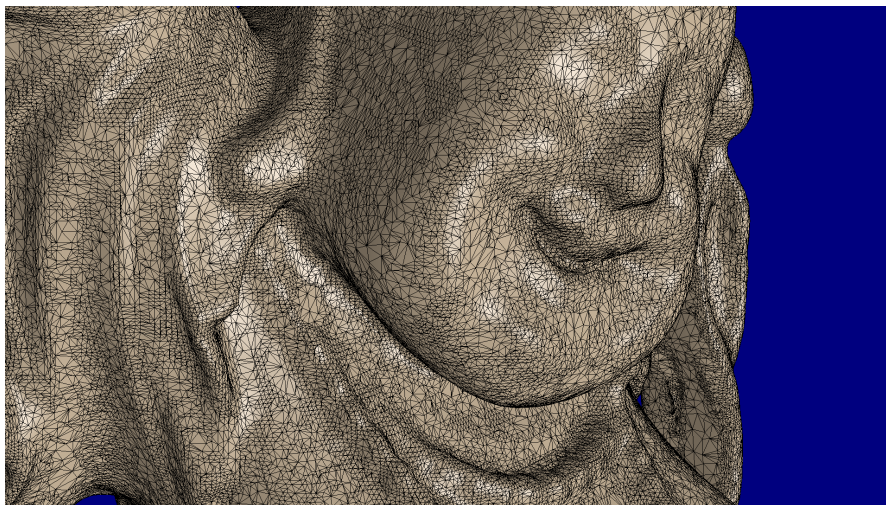


Abbildung 3.3.: Eine Rastergrafik

Bild- bzw. Tabellen-Beschriftungen sollten möglichst informativ sein. Aus der Beschreibung sollte die Bedeutung der Abbildung vollständig hervorgehen, so dass der Haupttext zum Verständnis nicht notwendigerweise gelesen werden muss.

3.7. Beispiel für die Darstellung von Algorithmen

Algorithmus 1 zeigt ...

Phase 1: Reduktion

```
for  $d := 0$  to  $\log_2 n - 1$  in parallel do
  for  $k := 0$  to  $n - 1$  by  $2^{d+1}$  in parallel do
     $x[k + 2^{d+1} - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$ 
```

Phase 2: Propagation

```
for  $d := \log_2 n$  to 0 in parallel do
  for  $k := 0$  to  $n - 1$  by  $2^{d+1}$  in parallel do
     $t := x[k + 2^d - 1]$ 
     $x[k + 2^d - 1] := x[k + 2^{d+1} - 1]$ 
     $x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]$ 
```

Algorithm 1: Pseudocode der zwei Phasen vom SCAN-Algorithmus [?].

3.8. Beispiel für die Darstellung von Quellcode-Auszügen

Listing 3.1 zeigt ...

```
1 void Adapt() {
2   cudaGLMapBufferObject( ... , vertVbo );
3   cudaGLMapBufferObject( ... , indVbo );
4   ...
5   cudaGLUnmapBufferObject( vertVbo );
6   cudaGLUnmapBufferObject( indVbo );
7 }
```

Listing 3.1: Pseudocode für die Kontrolle der VBOs.

4

Kapitel 4.

Ergebnisse und Evaluation

In diesem Kapitel sollen die Ergebnisse dieser Diplomarbeit diskutiert werden.

5 Kapitel 5.

Zusammenfassung und Ausblick

In diesem Kapitel sollen zunächst die erreichten Ziele diskutiert und abschließend ein Ausblick auf mögliche, weiterführende Arbeiten gegeben werden.

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
BTF	Bidirektionalen Textur Funktion
CPU	Central Processing Unit
CU	Control Unit
CUDA	Compute Unified Device Architecture
FLOPs	Floating Point Operations Per Second
FPU	Floating Point Unit
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HLOD	Hierarchische Level of Detail
IFS	Indexed-Face-Set
LOD	Level of Detail
MIMD	Multiple Instruction Multiple Data
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PCAM	Partitionierung Kommunikation Agglomeration Mapping
PM	Progressive Meshes
SFU	Spezial Funktion Units
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SLI	Scalable Link Interface
SP	Streaming-Prozessoren
SM	Streaming-Multiprozessoren
TPC	Textur Prozessor Clustern
VBO	Vertex Buffer Object

Abbildungsverzeichnis

3.1. Eine Vektorgrafik	8
3.2. Eine Vektorgrafik mit mathematischen Symbolen	9
3.3. Eine Rastergrafik	10

Tabellenverzeichnis

3.1. Die verwendeten Bibliotheken.	7
3.2. Konventionen zur Verwendung mathematischer Symbole	8

List of Algorithms

1. Pseudocode der zwei Phasen vom SCAN-Algorithmus [?]. 11

Listings

3.1. Pseudocode für die Kontrolle der VBOs.	11
---	----



Anhang A.

Anhang

Thema 1

Beispiel für einen Anhang

Thema 2

Danksagung

Hiermit möchte ich mich besonders bei Prof. Dr. XXXX, Prof. Dr. xxxx, Dipl. Inf. xxxx und Dipl. Inf. xxx für die Betreuung meiner Arbeit, hilfreiche Diskussionen und viel Geduld bei zahlreichen Fragen bedanken.