Inhaltsverzeichnis

Definitionen 4

1. Einleitung 6

2. Vorbetrachtungen 9

2.1. Grafische Datenverarbeitung und Grafikkarte 10

2.1.1. Übersicht der grafischen Datenverarbeitung 10

2.1.2. Die Renderpipeline 12

2.1.3. Entwicklung der Grafikbeschleuniger 15

2.1.4. Architektur aktueller Grafikbeschleuniger 18

2.1.5. Prinzipien der Grafikkomposition 21

2.2. Kennwerte zur Leistungsfähigkeit von Grafikkarten 26

2.2.1. Schnittstelle PCI-Express 26

2.2.2. Grafikspeichergröße 28

2.2.3. Grafikspeicherbandbreite 29

2.2.4. GPU Anzahl 29

2.2.5. Shadereinheiten und parallele Recheneinheiten 30

2.2.6. Ladezeiten 30

2.2.7. Bildwiederholrate 31

2.2.8. Leistungssteigerungsprognosen 32

2.2.9. Echtzeitfähigkeit, Messfehler und Interprozesskommunikation 33

2.3. Grundlegende Techniken der Beispielanwendungen 35

2.3.1. Beispielanwendungen 35

2.3.2. Bump Mapping 36

2.3.3. Shader-Programmierung 40

2.3.4. Compute Shader, GPGPU und Physikberechnungen 41

2.3.5. Partikeleffekte 43

2.3.6. Terrainkonstruktion 44

2.3.7. Objektmodellierung und Dateiformate 45

3. Lösungsvarianten zur Entwicklung der Rendersoftware 47

3.1. Basislösung 48

3.1.1. OpenGL 49

3.1.2. OpenGL SL 50
3.1.3. MathLib 51
3.1.4. GLEW 51
3.2. OpenCL 52
3.3. CUDA 53
3.4. Brook 54
3.5. CAL 55
3.6. Chromium 56
3.7. Rapidmind 57
3.8. Open Scene Graph 59
3.9. Visualization Library 61
3.10. Equalizer 63
3.10.1. Architektur 63
3.10.2. Erweiterte Kompositionsalgorithmen 65
3.11. Messwertermittlung 67
3.12. Tools 70
3.13. Vorstellung der Referenzsysteme 72
4. Entwurf ausgewählter Lösungsvarianten 75
4.1. Generelle Architektur 76
4.2. Basislösung 77
4.3. Equalizer und Open Scene Graph 82
4.3.1. Server-Client-Paradigma 84
4.3.2. Ansatz 85
5. Realisierung gewählter Lösungen 87
5.1. Basislösung 88
5.1.1. Ansatz der herstellerabhängigen Aufteilung der Renderpipeline 88
5.1.2. Bump Mapping 92
5.2. Testsystem Physikberechnung 100
5.2.1. x64-Portierung 100
5.2.2. Testprogramm 101
5.2.3. Schlussfolgerung 104
5.3. Equalizer und Open Scene Graph 106
5.3.1. Equalizer – Konfigurationsdateien 106
5.3.2. Nachrichtenwarteschlange 108
5.3.3. Anwendungsauswahl 109

5.3.4. Terraingenerierung ... 111
5.3.5. Ermittlung der Parallelisierungstechnologie ... 113
5.3.6. Physikberechnung ... 115
5.3.7. Laufzeitmessung ... 117
5.3.8. Installationsdateien ... 117
5.4. Messwerte ... 118
6. Bewertung ... 122
6.1. Messergebnisse ... 123
6.2. Software Engineering ... 125
6.3. Programm ... 128
7. Zusammenfassung ... 130
7.1. Ergebnisse und Schlussfolgerungen zur Leistungsuntersuchung ... 130
7.2. Zukünftige Erweiterung der Software ... 132
7.3. Technologieausblick ... 133
Literaturverzeichnis ... 134
Bildverzeichnis ... 136
Tabellenverzeichnis ... 137
Formelverzeichnis ... 137
DVD-Verzeichnis ... 137
Anhang ... 140
Allgemeine Fallstudie zur professionellen Nutzung der grafischen Datenverarbeitung ... 140
Leistungsprognose-Tabellen ... 141
Shader-Architektur ... 143
OpenCL Architekturskizze ... 144
CUDA Architekturskizze ... 145
Open Scene Graph Entwurfsskizze ... 145
Visualization Library Architekturskizzen ... 146
Equalizer-Skizzen ... 147
Gesamtaufbau der entwickelten praktischen Lösung ... 147
Skizzen zur Basislösung und manueller Aufteilung der Renderpipeline ... 150
Equalizer- und eqOSG-Diagramme ... 152
Konzept Physikengine ... 156
Handbuch zur Software ... 156

Definitionen

Paradigma

[…]Kuhn meint mit Paradigma also ein vorherrschendes Denkmuster in einer bestimmten Zeit. Paradigmen spiegeln einen gewissen allgemein anerkannten Konsens über Annahmen und Vorstellungen wider, die es ermöglichen, für eine Vielzahl von Fragestellungen Lösungen zu bieten. In der Wissenschaft bedient man sich in diesem Zusammenhang auch oft Modellvorstellungen, anhand derer man Phänomene zu erklären versucht. (Leitbild) […]

API (Application Programmung Interface)

Eine Programmierschnittstelle ist eine Schnittstelle, die von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Oft wird dafür die Abkürzung API (für engl. *application programming interface*, deutsch: „Schnittstelle zur Anwendungsprogrammierung“) verwendet. […]

Im weiteren Sinne wird die Schnittstelle jeder Bibliothek (*Library*) als API bezeichnet. […]

HPC (High Performance Computing)

Hochleistungsrechnen (englisch: *high-performance computing* – HPC) ist ein Bereich des computergestützten Rechnens. Er umfasst alle Rechenarbeiten, deren Bearbeitung einer hohen Rechenleistung oder Speicherkapazität bedarf. Hochleistungsrechner sind Rechnersysteme, die geeignet sind, Aufgaben des Hochleistungsrechnens zu bearbeiten. […]

Für die exakte Abgrenzung des Hochleistungsrechnens von den anderen Bereichen des computergestützten Rechnens gibt es aufgrund der schnellen Entwicklung der Rechentechnik keine dauerhaften formalen Kriterien. Es ist aber allgemein anerkannt, dass Rechenanwendungen, deren Komplexität oder Umfang eine Berechnung auf einfachen Arbeitsplatzrechnern unmöglich oder zumindest unsinnig macht, in den Bereich des Hochleistungsrechnens fallen.

Hochleistungsrechnen wird vor allem durch die auf parallele Verarbeitung ausgerichtete Architektur von Hochleistungsrechnern überhaupt erst möglich. […]

Grid Computing

Grid-Computing ist eine Form des verteilten Rechnens, bei der ein virtueller Supercomputer aus einem Cluster lose gekoppelter Computer erzeugt wird. Es wurde entwickelt, um rechenintensive wissenschaftliche – insbesondere mathematische – Probleme zu lösen.

CAS (Computer Assisted Surgery)

Computerunterstützte Chirurgie, d. h. Assistenz des Computers bei Therapieplanung und -durchführung. [Wikipedia dt.]

Computer Assisted Surgery (CAS) stellt ein computergestütztes, medizinisches Behandlungskonzept sowie eine Menge von Behandlungsmethoden dar. Es wird zur Planung von medizinischen Voruntersuchungen sowie zur Leitung und Durchführung medizinischer Eingriffe genutzt. […] [Übersetzung Wikipedia engl.]

CAD (Computer Aided Design)

Der Begriff Rechnerunterstützte Konstruktion oder englisch Computer Aided Design (CAD) bezeichnet das Erstellen von Konstruktionsunterlagen für mechanische, elektrische oder elektronische Erzeugnisse mit Hilfe von spezieller Software, zum Beispiel im Anlagenbau, Maschinenbau, Autobau, Flugzeugbau, Schiffbau, in der Zahnmedizin und auch in der Architektur, im Bauwesen sowie im Grafik- und Modedesign. […]

DCC (Digital Content Creation)

Als Digital content creation wird der Bereich innerhalb einer Produktionspipeline bezeichnet, der sich mit der Erstellung von multimedialen Inhalten beschäftigt. Darunter werden verschiedenste Technologien zusammengefasst.
Generell fallen alle Werke, die grafischer Natur sind und computergestützt erzeugt wurden in den Bereich der DCC.
Da die Erstellung dieser Inhalte meistens ein kreativer Prozess ist, wird eine Fachkraft innerhalb eines Bereiches der DCC fast immer als *Artist* bezeichnet (3D-Artist, Compositing-Artist ...). […]

Demo („Demonstration“ im Sinne der Ing. Wissenschaft)

Eine Software zum Aufzeigen der Fähigkeiten des Programmierers und der verwendeten Plattform.

1. Einleitung

Die grafische Datenverarbeitung ist eine Disziplin der Informatik, die sich mit der grafischen Auswertung und Darstellung von Daten beschäftigt. Voraussetzung dafür ist das Vorhandensein von zweidimensionalen oder dreidimensionalen Daten wie Objekte, Texturen und Punkten. Desweiteren sind für Animationen auch Umgebungsdaten nötig. Diese können von vordefinierten Vektoren mit Transformationsart, Länge und Richtung bis hin zu physikalischen Daten wie Gravitation, Beschleunigung und Geschwindigkeit reichen, mit deren Hilfe dann die Bewegung zur jeweiligen Zeit berechnet werden können.

Anhand dieser Vielzahl von Daten die zu einer grafischen Präsentation benötigt werden ist erkennbar, dass es leistungsfähiger Rechentechnik in diesem Bereich der Informatik bedarf.

Die Verarbeitung der grafischen Daten geschieht in mehreren Schritten. Die Rohdaten wie Punkte, physikalische Umgebungsdaten und Bilder müssen vorverarbeitet werden. Ziel dabei ist es, anhand dieser Rohdaten grafische Daten wie Objekte, Texturen (an Objekte gebundene Oberflächenbilder) und Transformationsmatrizen für Bewegung und Effekte zu generieren. Diese Vorverarbeitung findet meist auf dem Hauptprozessor eines Systems statt.

Diese grafischen Daten werden an die Grafikkarte weitergeben. Auf dieser wird dann ein Algorithmus zur Transformation der verschiedenen Daten in eine Szene abgearbeitet, der üblicherweise als Renderpipeline bezeichnet wird. Dieser generiert am Ende ein Bild, welches folgend an ein Peripheriegerät wie Beamer oder Bildschirm oder spezielle Hardware wie Raumbeamer („Cave“) oder Stereo-Beamersystem weitergeleitet wird. Ausgehend von der Komplexität der Daten und den Parametern wird dieser Algorithmus mehrmals pro Sekunde (Echtzeit) durchlaufen oder braucht mehrere Stunden zur Generierung eines Bildes.

Heutige Grafikkarten sind sehr leistungsfähige Spezialchips die mit unterschiedlichsten Architekturen und Verfahren parallel eine Vielzahl von Daten sehr schnell verarbeiten können. Aufgrund neuester Technologien und dem rapiden Technologiefortschritt in diesem Bereich ist es möglich, auch generelle Berechnungsalgorithmen aus der Vorverarbeitung direkt auf der Grafikkarte berechnen zu lassen. Desweiteren wurde die Technologie von Hauptprozessoren adaptiert, mehrere Recheneinheiten (Processing Units) auf einer Karte unterzubringen. Dies ermöglicht ganz neue Ansätze und einen großen Fortschritt in der Computergrafik auf dem Weg zur realistischen Darstellung dreidimensionaler, computergenerierter Szenen.

Doch wie kann diese Technologie in Softwaresystemen genutzt werden?

Die Beantwortung dieser Frage ist keineswegs trivial. Einen Ansatz bieten die Treiber der jeweiligen Grafikkarten, die einige vordefinierte Aufteilungsmodi anbieten. Die Aufteilung

der Rechenlast durch den Treiber ist jedoch allenfalls für Heimanwendungen ausreichend, da die dreidimensionale Aufteilung im Vergleich zu professioneller Computergrafik weniger aufwendig ist. Für professionelle Anwendungen müssen jedoch neue Ansätze gefunden werden, damit Vorverarbeitung wie auch Renderpipeline optimal von dieser neuen Technologie profitieren. Die Technologie ist dabei so innovativ, dass nur sehr wenige Beispielansätze zur Verfügung stehen.

Es gibt viele verschiedene Anwendungsgebiete, die von dieser Technologie profitieren können. Zu einem der wahrscheinlich wichtigsten Anwendungsgebiete gehört die medizinische Computergrafik. Dabei werden meist Daten verarbeitet und dargestellt, die durch 3D-Scans wie Magnetresonanztomographie oder Computertomographie erzeugt wurden. Der Anwendungsbereich wird auch als CAS (Computer Assisted Surgery) bezeichnet. Grund für das massive Datenaufkommen sind die geforderte, hohe Auflösung der Darstellung sowie die große Anzahl an dreidimensionalen Punkten. Die hohe Auflösung der Darstellung ist nötig, da basierend auf dem dreidimensionalen Modell Krankheitsdiagnosen erfolgen. Falsche Diagnosen führen zu unnötigen Operationen, unbehandelten (weil nicht erkannten) Krankheiten oder Fehloperationen. Die große Anzahl an Punkten ist dadurch zu erklären, dass diese Scans üblicherweise ein Voxelmodell (Volume Pixel; Pixel mit dreidimensionaler Ausdehnung) erzeugen. Der Vorteil ist ein solides, gefülltes Objekt im Gegensatz zu polygonalen Modellen, wobei sehr genaue Oberflächendetails vorhanden sind. Jeder gescannte Punkt wird abgespeichert und muss darauf folgend dargestellt werden. In Verbindung mit der gewünschten hohen Auflösung entsteht dadurch ein massives Datenaufkommen, was verarbeitet werden muss. Aufgrund dieser genannten Probleme muss bisher nicht nur bei der Aufnahme, sondern auch bei der Darstellung auf sehr leistungsstarke Rechentechnik sowie teure Speziallösungen zurückgegriffen werden. Durch eine innovative Ausnutzung heutiger Standardkomponenten könnte die teure Spezialrechentechnik, zumindest im Bereich der Darstellung, durch Standardrechner mit entsprechender Software ersetzt werden. Dies ermöglicht große Einsparung für Praxen der Chirurgie.

Ein weiterer großer Anwendungsbereich der professionellen Computergrafik ist das Computer Aided Design (CAD). CAD Systeme dienen der Erstellung von Design-Prototypen von Produkten, Automobilen, Kleidung und Schmuck. Mithilfe dieser virtuellen Prototypen werden immense Kosten für Produkt produzierende Firmen gespart, da man dadurch die reale Fertigung von Prototypen umgeht, von denen die meisten nachher nicht zum Einsatz kommen. Dabei ist eine reale Darstellung dieser Objekte nötig. Die Darstellungsverfahren, die dabei zum Einsatz kommen, beinhalten rechentechnisch sehr aufwendige Algorithmen. Somit ist, abhängig von der Komplexität der darzustellenden Objekte, größtenteils eine Visualisierung in Echtzeit nicht möglich. Dabei verwendete Verfahren sind beispielsweise Raytracing und Radiosity. Diese Algorithmen sind jedoch sehr gut parallelisierbar. Somit bietet sich durch die neuen Technologien der Grafikkarte an, diese Algorithmen nicht wie bisher auf der CPU sondern auf der GPU berechnen zu lassen. Dadurch ist entweder ein

höherer Detailgrad oder eine flüssige Animation der Darstellung möglich. Dies ist für die Akquisition dieser Produkte von entscheidendem Vorteil.

Schlussendlich wäre als letztes Anwendungsfeld die Filmindustrie und Digital Content Creation zu nennen. Anhand der neuesten Filme [NVI10] wird das massive Datenaufkommen schnell deutlich. Bisher wurde dazu auf sehr teure Grafikkartenlösung von ATI und NVIDIA (die beiden führenden Grafikhersteller) vertraut. Bei genauerer Analyse ist, wie später auch dargestellt, zu sehen, dass diese Speziallösungen sich weder in Architektur noch in den verwendeten Chips von Standardlösungen unterscheiden. Einziger Unterschied ist ein erweiterter, nativer OpenGL-Befehlssatz. Diese Speziallösungen sind traditionell eine Komposition mehrerer, gleicher Karten zu einem System, was auch mit heutigen Standardgrafikkarten möglich ist. Innovative Softwarelösungen können gewaltige Einsparungen in betreffenden Firmen bewirken.

Diese Erkenntnisse, die durch die Entwicklung einer solchen, innovativen Software entstehen, sollten desweiteren in der Lehre vermittelt werden. Somit kann man den Studenten einen immensen Wissensvorteil in der Wirtschaft und in angesprochenen Bereichen ermöglichen.

Bezüglich dieser Ausführung der Aktualität des Themas ist die Aufgabe dieser Arbeit zu untersuchen, in wie weit diese Technologien praktisch eingesetzt werden können. Desweiteren sollen Messungen dabei helfen, den Vorteil dieser Technologien zu belegen. Dazu sind geeignete Kennwerte der Messungen zu finden, diese vorzustellen und zu erläutern. Durch die große Abhängigkeit der Technologie von der gewählten Hardware ist ebenfalls eine aktuelle Übersicht zu erstellen, welche einen Überblick des heutigen technologischen Umfeldes gewährt.

Da bislang in diesem Bereich noch keine Referenzsoftware gegeben ist muss zur Untersuchung der technologischen Machbarkeit und zur Aufnahme von Messwerten eine eigene Softwarelösung entwickelt werden. Eventuelle Komplikationen und Probleme sind dabei zu notieren.

2. Vorbetrachtungen

Zum Gesamtverständnis der Arbeit sind einige Erläuterungen des Themengebietes notwendig. Dabei sind die Vorbetrachtungen in drei Komplexe gegliedert

Im ersten Abschnitt werden auf computergrafische Grundsätze eingegangen. Dazu gehört der Ablauf der Darstellung dreidimensionaler Szenen. Dabei wird geklärt, an welchen Schritten angesetzt werden kann um eine sinnvolle Aufteilung auf mehrere grafische Verarbeitungseinheiten (GPUs) zu erreichen. Da das Thema durch die aktuelle Entwicklung der Hardware zur grafischen Datenverarbeitung geprägt ist wird folglich kurz auf die historische Entwicklung von Grafikbeschleunigern eingegangen. Anschließend wird der logische und physische Aufbau aktueller Grafikkarten vorgestellt. Abschließend werden die durch Grafikhersteller entwickelten Ansätze zur Grafikkomposition erläutert.

Im zweiten Abschnitt werden Kenngrößen zur Leistungsfähigkeit der Grafikkarte und Probleme verteilter Berechnungen vorgestellt. Diese sind Voraussetzung für die spätere Messwertermittlung.

Im letzten Abschnitt werden grundlegende Techniken beleuchtet, welche die Basis für die Beispielapplikationen (sogenannt „Demos“) bieten. Die Entwicklung der Demos ist nötig um die Möglichkeiten der neuen Technologien im Praxiseinsatz zu testen und um die Messwerte auf den Testsystemen zu ermitteln.

2.1. Grafische Datenverarbeitung und Grafikkarte

Dieser Abschnitt behandelt Grundsätze der grafischen Datenverarbeitung. Desweiteren werden Aufbau sowie Funktionsweise von Grafikbeschleunigungskarten (kurz: Grafikkarten) erläutert.

2.1.1. Übersicht der grafischen Datenverarbeitung

Zum Einstieg in das Themengebiet gilt es zu klären, womit sich die Informatik in der grafischen Datenverarbeitung befasst.

Grafische Datenverarbeitung behandelt die grafische Repräsentation von Informationen. Zu diesen Informationen zählen Punkte eines bestimmten Koordinatensystems sowie deren zugehörige Farbwerte. Diese Farbwerte können festgelegt sein, aus einer Grafik gelesen werden oder aber durch physikalische Berechnungen von Licht berechnet sein. Punkte existieren als ungeordnete Menge (Punktewolken) oder aber eine durch Reihenfolge und Anzahl geordnete Menge (Punktenetze, Dreiecks- oder Polygonale Netze) von Koordinaten. Auf diese Begriffe wird in späteren Absätzen eingegangen.

Um diese Daten grafisch sinnvoll zu präsentieren gibt es verschiedene Technologien. Desweiteren stehen die Daten meist in einem gewissen Kontext. Sie sind Anwendungsgebunden. Die Verarbeitung dieser Daten geschieht elektronisch, mit Hilfe von Computern und dazugehöriger Software. Eine Übersicht in Form einer MindMap soll verdeutlichen, was „grafische Datenverarbeitung“ ist. Angelehnt an den Namen ist die Grafik dreigeteilt.

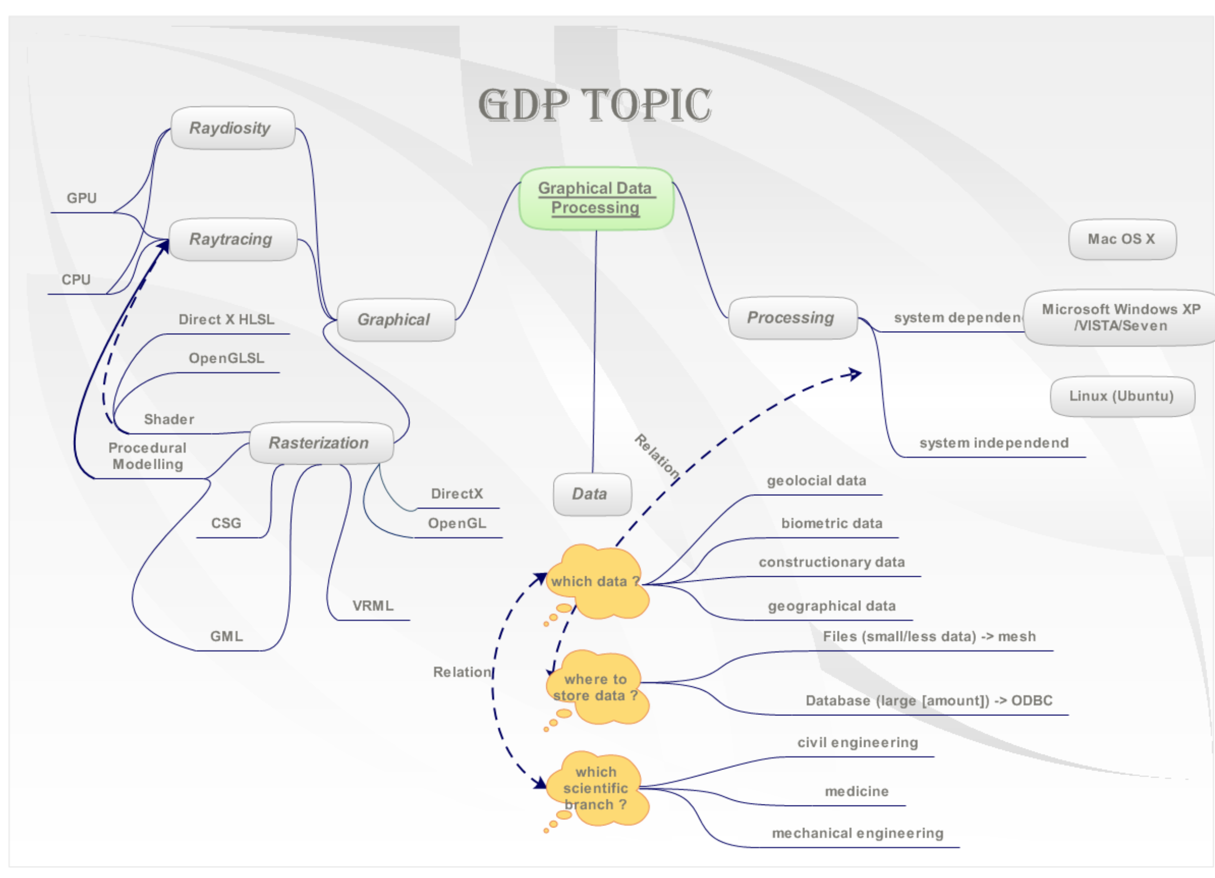


Bild 1 Mind Map Bedeutung "grafische Datenverarbeitung"

Unter „grafisch“ sind die Technologien zu sehen, mit deren Hilfe die Daten aufbereitet werden. Es gibt zwischen bestimmten Technologien eine Verbindung – sie existieren im Kontext mit einer anderen Technologie. Hierbei ist das „Procedural Modelling“ zu nennen. Basis für die Nutzung dieser Technologie sind vorverarbeitete GML-Daten (Geometric Modelling Language). Diese Daten sind mathematische Beschreibungen von Kurven, Linien, Ebenen und anderen geometrischen Primitiven. Procedural Modelling-Systeme können nur mit diesen Objektbeschreibungen umgehen [TUl08] . Die Komposition dieser Daten mit Texturen zu einer Szene wird durch das System erreicht[For01] . Deren Anzeige wird an einen Raytracer geleitet. Auch zwischen anderen Technologien bestehen diese Verbindungen. Aufgrund der Vielzahl der Technologien ist die MindMap nicht komplett.

Bei dem Begriff „Daten“ treten viele Fragen bezüglich des Kontexts auf. Daten stehen sind Anwendungsspezifisch. Auch deren grafische Repräsentation ist im Kontext zu sehen. So haben, je nach Anwendungsfall, einige Daten eines Datensatz in einem CAD-System andere Prioritäten als in CAS-Systemen. Auch deren Aussage ist je nach Anwendungsgebiet verschieden. Ein gutes Beispiel hierfür ist der Einsatz von Farben, deren Aussage und Sinn stark davon abhängen, in welchem Kontext diese stehen.

Schlussendlich werden Daten „verarbeitet“. Die Verarbeitung geschieht in heterogenen Computersystemen auf verschiedenen Komponenten. Die Schnittstelle der verschiedenen

Anwendungsprogramme, die Daten grafisch darstellen, ist das Betriebssystem. Die Abstimmung einer Software ist sowohl Anwendungs- als auch Technologieabhängig. In verschiedenen Wirtschaftbranchen haben sich gewisse Betriebssysteme zum Standard entwickelt. Dies basiert auf das durchschnittliche, informationstechnologische Verständnis der Anwender. Desweiteren sind einige Technologien an bestimmte Betriebssysteme gebunden. Daher sind gewisse computergrafische Systeme nur auf dem jeweiligen Betriebssystem nutzbar, was wiederum den Nutzerkreis einschränkt.

Abschließend ist zu sagen, das die grafische Datenverarbeitung ein komplexes Gebiet ist, mit dem man sich als Entwickler intensiv beschäftigen muss, um konstruktiv, effektiv und Anwender- sowie Anwendungsbezogen zu arbeiten.

2.1.2. Die Renderpipeline

Die Generierung von Ausgabebildern basierend auf dreidimensionalen Objekten, Effekten und Texturen wird in einem Algorithmus zusammengefasst, der allgemein als „Renderpipeline“ bekannt ist.

Einen Überblick des Algorithmus verschafft folgende Skizze.

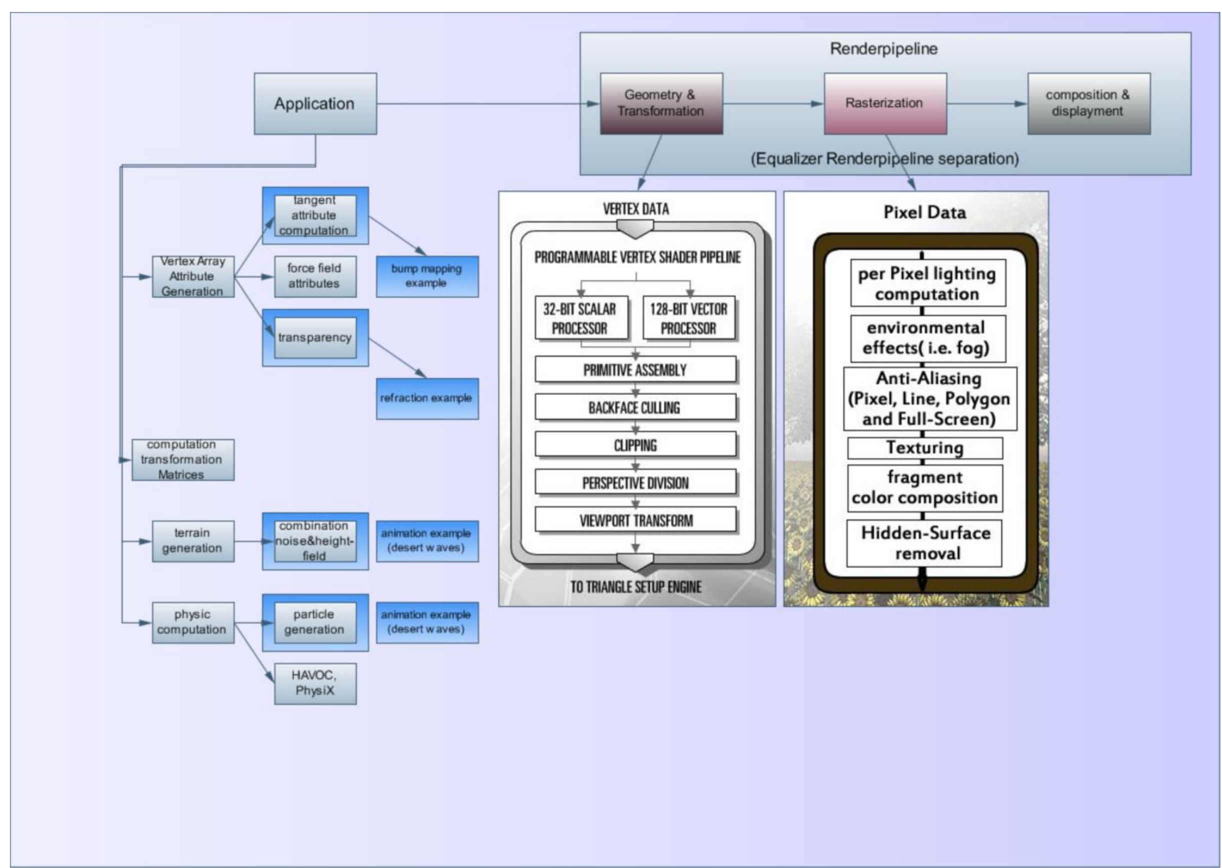


Bild 2 Aufteilung der Renderphasen

Die Erstellung des Ausgabebildes geschieht in vier Phasen.

Die erste Phase der Berechnung findet traditionell auf der CPU statt. Dabei handelt es sich um die Vorverarbeitung durch die eigentliche Anwendung. Dabei werden gewisse Werte der physikalischen Umgebung sowie Parameter darzustellender Objekte vorberechnet. Des Weiteren ist in dieser Phase die Zusammensetzung von Transformationsmatrizen in andere Koordinatensysteme zu finden. Attributparameter für spezielle Effektberechnungen werden generiert. Diese Vorverarbeitungsschritte sind i.d.R. von geringer Komplexität, jedoch von großem Rechenaufwand und werden meist für jedes neue Bild (Frame) wiederholt. Durch neue Compute Shader-Technologie ist es möglich, die Vorverarbeitungsschritte auf die Grafikkarte auszulagern, um einen Geschwindigkeitsvorteil zu erzielen.

Nach der Vorverarbeitung wird die eigentliche Renderpipeline abgearbeitet. Sie besteht heutzutage aus drei Schritten.

In der zweiten Phase werden dreidimensionale Punktkoordinaten (Vertices) zu geometrischen Objekten zusammengefasst. Verschiedene Transformation werden ausgeführt, um Objekte in der Szene zu platzieren, eine virtuelle Kamera (die Szene) zu kreieren und diesen dreidimensionalen Raum auf die, in der Regel, zweidimensionale Bildschirmfläche zu projizieren. Aufgrund dieser Aktionen wird die Phase typischerweise „Geometry and Transformation“ genannt.

Zu Veranschaulichung des Ergebnisses dieses ersten Schrittes folgt ein Bild, in dem das Objekt eine vordefinierte Farbe zugeordnet bekommen hat.

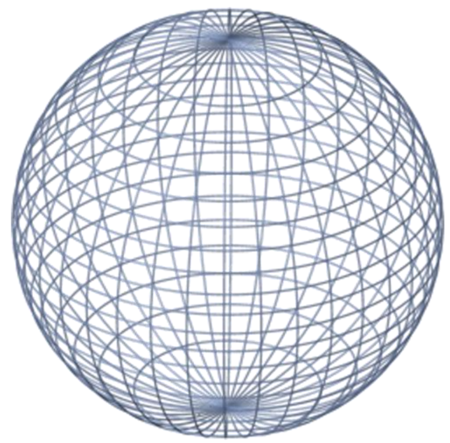


Bild 3 Drahtgittermodel einer polygonalen Kugel [Wikipedia]

In der dritten Phase liegt dann die projizierte Szene vor. In dieser Phase erfolgt die Farbgebung der im dreidimensionalen Raum vorhandenen Pixel, welche auch „Fragmente“ genannt werden. Hierbei wird jedem Pixel entweder ein vordefinierter Farbwert zugeordnet, der mit dem Punkt abgespeichert ist, oder es erfolgt die Auswertung eines ausgewählten Lichtmodells. Desweiteren werden Umgebungseffekte, wie zum Beispiel Nebel, in die Szene

eingefügt. In dieser Phase werden die Objekte mit eventuell abgespeicherten Texturen versehen.

Das Beispiel für das Ergebnis dieser Phase ist aus der offiziellen OpenGL Referenz entnommen.

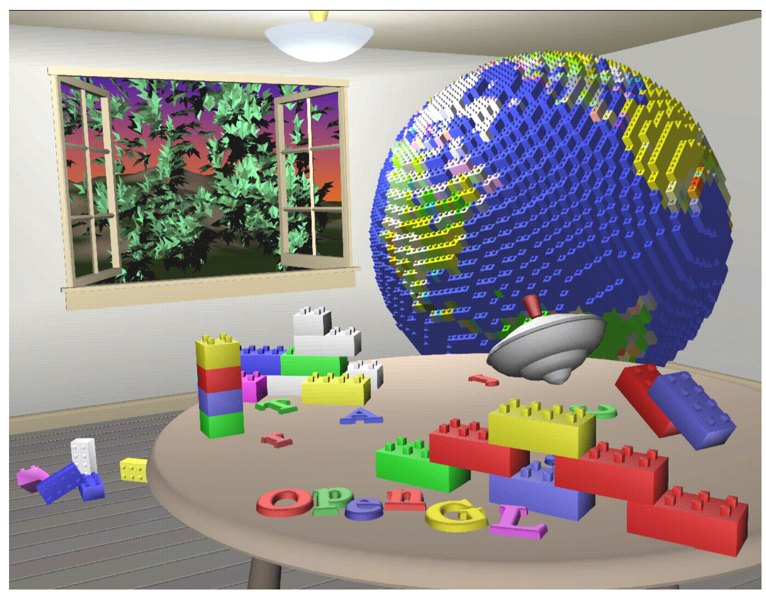


Bild 4 Smooth Shading [OpenGL RedBook Plate 6]

Die jeweiligen Transformationen kann der Entwickler beeinflussen, indem die angesprochenen Phasen und deren Berechnungen mit eigenen Berechnungsalgorithmen ersetzt werden. Die dazu nötige Technologie wird als „Shader Programming“ bezeichnet. Entsprechend der modifizierbaren grafischen Objekte heißen die Programmiereinheiten Vertex-, Geometry- und Fragment Shader. Auf diese Technologie wird in einem späteren Kapitel näher eingegangen.

Die vierte Phase ist aufgrund technologischer Neuerung der letzten Jahre entstanden. In der Phase „Composition and Displayment“ sind eine Reihe von Nachbearbeitungseffekten zu finden. Wichtige Techniken, dabei erwähnenswerte Effekte, sind Multisample- und Coverage Sampled Anti-Aliasing (MSAA und CSAA), Tiefenunschärfe und Shadow Mapping. Grundlegende Technologie bei allen genannten Technologien ist das „Off-Screen Rendering“. Dabei wird die Szene aus mehreren, unterschiedlichen Positionen gerendert und das entstehende Bild in einem Hintergrundspeicher (Pixel Buffer Object oder Fragment Buffer Object) abgelegt. Die unterschiedlichen Bilder werden im letzten Renderzyklus kombiniert und gewichtet zusammengerechnet. Dies wird als „Composing“ (dt. komponieren, lat. compositio = „Zusammenstellung, Zusammensetzung“) bezeichnet. Das finale Bild ist demnach das Ergebnis der gewichteten Zusammensetzung von Einzelbildern. Dieses Bild wird anschließend an den Hauptrenderingkontext gesendet und schlussendlich auf der Anzeigefläche dargestellt.

Als Beispiele für diesen Renderschritt werden folgend die Tiefenunschärfe und das Shadow Mapping präsentiert.

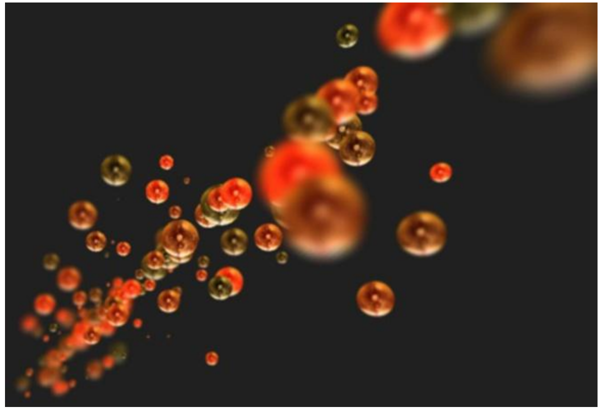


Bild 5 Tiefenunschärfe (Depth of Field) [imageshak.us]

Bild 6 Shadow Mapping [blenderartist.org]

2.1.3. Entwicklung der Grafikbeschleuniger

Die Renderpipeline wird auf der Grafikkarte ausgeführt. Um die Möglichkeiten der heutigen Karten einschätzen und realistische Prognosen für zukünftige Technologien entwickeln zu können ist es unerlässlich, sich mit der rapiden Entwicklung der Hardware von den Anfängen bis in die heutige Zeit zu beschäftigen.

Ausgehend vom Stand der grafischen Rechentechnik seit 1990 ist ein starker Anstieg der Entwicklung der Grafikhardware zu beobachten. Zu dieser Zeit wurden die ersten Zusatzkarten für grafische Anwendungen entwickelt. Diese Karten werden „Grafikbeschleunigerkarte“ oder auch „Grafikkarte“ genannt.

Die Entwicklung seit diesem Zeitpunkt ist in folgendem Diagramm zusammengefasst. Dabei symbolisiert die Höhe der Balken den Innovationsgrad im Vergleich zur Vorgängertechnologie, da einige Technologien sehr viel drastische Konsequenzen und größere Verbesserungen boten als andere. Am oberen Rand des Diagramms sind ungefähre Jahreszahlen für die entsprechenden Entwicklungsschritte genannt, sowie die vorherrschenden Versionen der Grafikschnittstellen. Die wechselnde Anordnung dient nur der Übersichtlichkeit. Am unteren Rand der Grafik sind einige der bekannteren Grafikbeschleuniger der jeweiligen Epoche genannt.

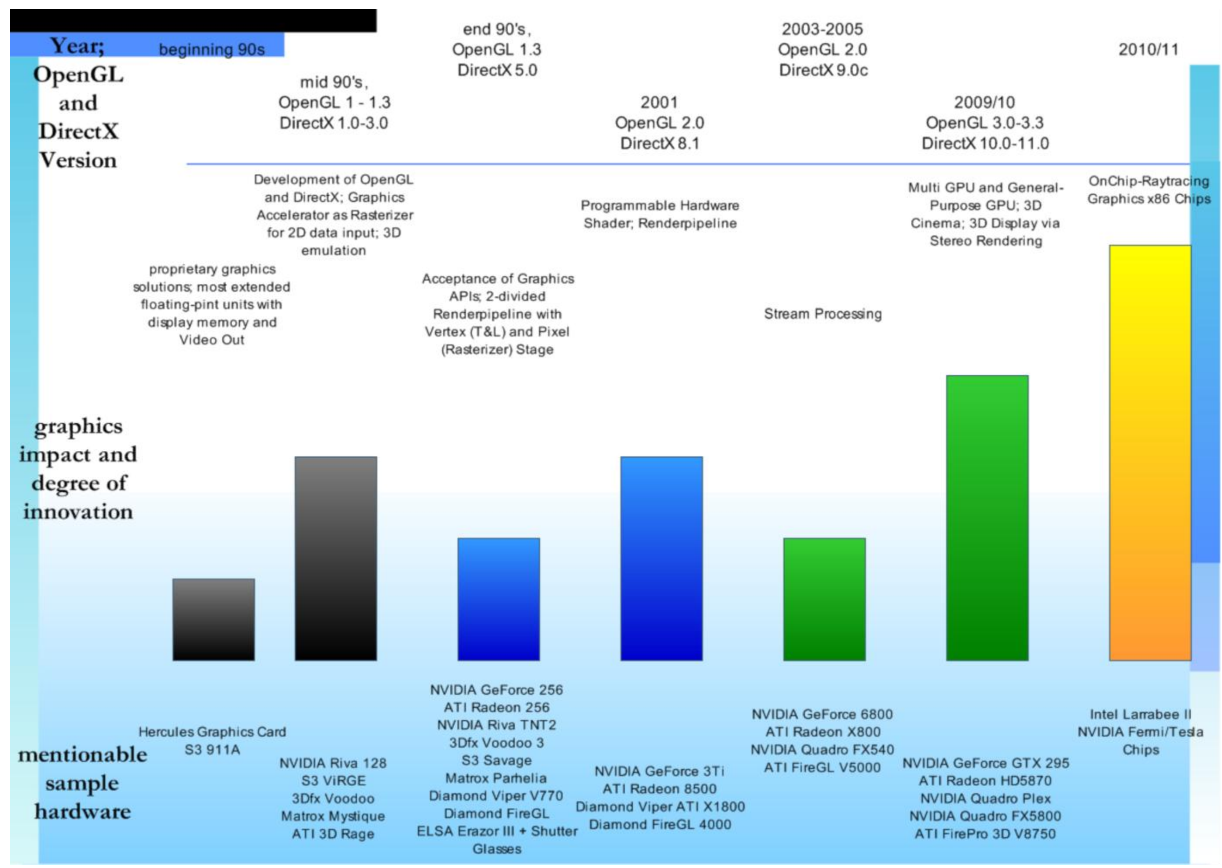


Bild 7 Historische Entwicklung der Grafikbeschleiniger

Momentan(Januar 2010) ist eine Entwicklungsphase mit starken Veränderungen der Grafikhardware erkennbar, wie aus der Grafik ersichtlich ist. Es erscheinen in sehr geringen Abständen, relativ zu den letzten Jahren, neue Grafiktechnologien und damit verbundene Hardwarearchitekturen. Diese könnten das Hauptziel für DCC - die fotorealistische Darstellung computergenerierter, dreidimensionaler Szenen in Echtzeit – in naher Zukunft ermöglichen.

Die Entwicklung kann besonders anhand der Leistungsparameter der Grafikbeschleuniger festgestellt werden. Es ist ersichtlich, dass sich beispielsweise der Datendurchsatz der aktuellen Grafikkarten ungefähr im Dreimonates-Zyklus verdoppelt.

Folgend wird ein Vergleich älterer Grafikkarten (Herstellungsjahr 2006/2007), aktueller Grafikkarten und zukünftiger, bisher schon teilweise bestätigter Grafikkarten präsentiert, die diese Entwicklung verdeutlicht.

	NVIDIA GeForce7600 Go	NVIDIA GeForce 8800GT	NVIDIA GeForce GTX275	ATI Radeon HD4850X2	NVIDIA Fermi	Intel Larrabee	NVIDIA Quadro FX 5800	ATI FirePro 3D V8750
GPUs	1	1	1	2	1	around 32	1	1
ShaderUnits	8 P + 5 V	112	240	800	512		240	800
CLOCK:								
Memory Tech	GDDR2	GDDR3	GDDR3	GDDR3	GDDR5	GDDR5	GDDR3	GDDR5
Core	450 MHz	600 MHz	633 MHz	625 MHz	?	2000 MHz	650 MHz	750 MHz
Memory	350 MHz	900 MHz	567 MHz	993 MHz	?		816 MHz	900 MHz
Shader	450 MHz	1500 MHz	1404 MHz	625 MHz	?		1476 MHz	750 MHz
							(Same Core as GTX285)	
Capacity								
Main Mem	256 Mbyte	512 Mbyte	896 Mbyte	1024 Mbyte			4096 Mbyte	2048 Mbyte
Verr&Tex			198 Mbyte					
Pixelbuffer			64 Mbyte					
Chip Architecture	G73M	G92	GT200b	Stream/RV770	GT300	x86	GT200GL	RV770XT
Price	no distribution anymore	100-140€; distribution expiring	220 €	220 €			3.499 $	1.800 $

Tabelle 1 Grobvergleich ausgewählter Grafikkarten

Es sei erwähnt, dass die speziellen Profi-Grafikkarten (Quadro FX, FirePro) auf den Chips ihrer Desktop-Pendants basieren. Einzige Unterschiede zwischen den aktuellen Topmodellen der Profi-Serie und der Desktop-Serie sind der erweiterte, native OpenGL-Befehlssatz und der relativ große Grafikspeicher der Profikarten. Architektur und Leistung sind bei beiden Serien vergleichbar. Teilweise sind die Desktop-Varianten sogar leistungsfähiger, da diese sehr viel häufiger durch neue Modelle ersetzt werden als die Profi-Serien.

Die Auswahl der Grafikbeschleuniger ist durch mehrere Fakten begründet. Die erste, dritte sowie vierte Karte wurde gewählt, da diese im späteren Vergleich in den Testsystemen zum Einsatz kommen. Grafikkarte Zwei wurde aufgenommen, da es die erste Karte mit Compute Shader-Architektur war, welche durch die CUDA-Technologie unterstützt wurde. Daher wurden nach dieser Karte die Shader-Einheiten nicht mehr nach Vertex-, Geometry- und Fragment Shader unterteilt, da diese alle die gleichen Grundfunktionen ausführen können. Diese Technologie wird bei ATI Technologies unter dem Stream bekannt; zugehörige Shader-Einheiten heißen dabei „Stream Prozessoren". Grafikkarte Fünf und Sechs stellen die die zukünftigen Grafikbeschleuniger dar. Sie werden Mitte 2010 bis Mitte 2011 erscheinen. Grafikkarte Sieben und Acht sind die aktuell leistungsfähigsten Grafikkarten des Profi-Segments. Diese sind besonders durch den erheblichen Preisunterschied bei annähernd gleicher Leistungsfähigkeit zu Grafikkarte Drei und Vier sowie den aktuellen Hochleistungs-Desktopkarten in den Vergleich mit einbezogen.

Abschließend ist erkennbar, dass die rapide aktuelle Entwicklung der Grafikkarten viele neue Technologien ermöglicht. Diese führen zu Anwendungen, die lange Zeit schwer vorstellbar waren. Dies ist beispielhaft an der korrekten physikalischen Berechnung von Luft- und Flüssigkeitsströmungen in Echtzeit für diverse Simulationen und Animationen zu erkennnen.

Dies ist allerdings nur durch die optimale Ausnutzung der Technologien möglich, welche in der Entwicklung umfangreiches Wissen um Architektur und Funktionsweise der Grafikbeschleuniger voraussetzt.

2.1.4. Architektur aktueller Grafikbeschleuniger

Wie im vorigen Abschnitt erläutert bieten aktuelle Grafikkarten eine diverse Zahl an Technologien um dreidimensionale Szenen darzustellen, den Grad an Realismus zu erhöhen, die Rechenlast zu reduzieren und schlussendlich auch aufzuteilen. Um diese Technologien jedoch zu nutzen um neue Anwendungen zu entwickeln ist eine fundierte Kenntnis über den logischen und physischen Aufbau der Grafikbeschleuniger nötig.

Dafür gibt es mehrere Gründe. Zum Einen setzen die effektivsten und stärksten Erweiterungen sehr tief in der Grafikpipeline an. Meist muss das Backend bestehender Anwendungen zur Implementierung der neuen Technologien komplett umgeschrieben werden, da die Software an neue Hardwarearchitekturen angepasst werden muss. Bei Nutzung von Bibliotheken, APIs und Frameworks muss herausgefunden werden, ob deren neue Version die gewünscht Technologie unterstützt. Da Frameworks und APIs meist das Grundgerüst großer Anwendungen bilden, kann die Anwendung nicht auf neue Technologien angepasst werden, wenn ungenügendes Wissen über die Grafikarchitektur vorliegt.

Ein zweiter Grund, warum das Wissen um die Architektur der Grafikkarte wichtig ist, sind die von den Anwendern geforderten Leistungen und grafischen Zusätze. Es ist nötig, zu wissen, ob ein Effekt von der Hardware des Anwenders tragbar ist und ob die Anwendung dies bietet. Das Wissen hat direkte Auswirkungen auf Teile der Softwareentwicklung wie zum Beispiel die Güte einer Aufwandsschätzung für etwaige Erweiterungen. Dieses Wissen ist von Entwicklern wie auch Programmierern nötig, da diese neue Technologien direkt umsetzen.

Zuletzt bildet die Architektur der Grafikkarte ebenfalls die Grundlage für die zahlreichen Softwarearchitekturen von APIs und Frameworks. Ein umfangreiches Wissen über die Grafikkartenarchitektur erhöht das Verständnis der darauf basierenden Schnittstellen.

Folgende Grafik stellt den Aufbau einer aktuellen Grafikkarte dar. Bezogen auf das Thema der Arbeit, die Aufteilung der Renderpipeline auf mehrere Grafikkerne, wurde die Sapphire HD4850 X2 als eine Grafikkarte mit 2 physischen Kernen auf einem PCB (Printed Circuit Board = Platine) zur Darstellung gewählt. Die markierten Bereiche kennzeichnen Kernkomponenten der Grafikkarte.

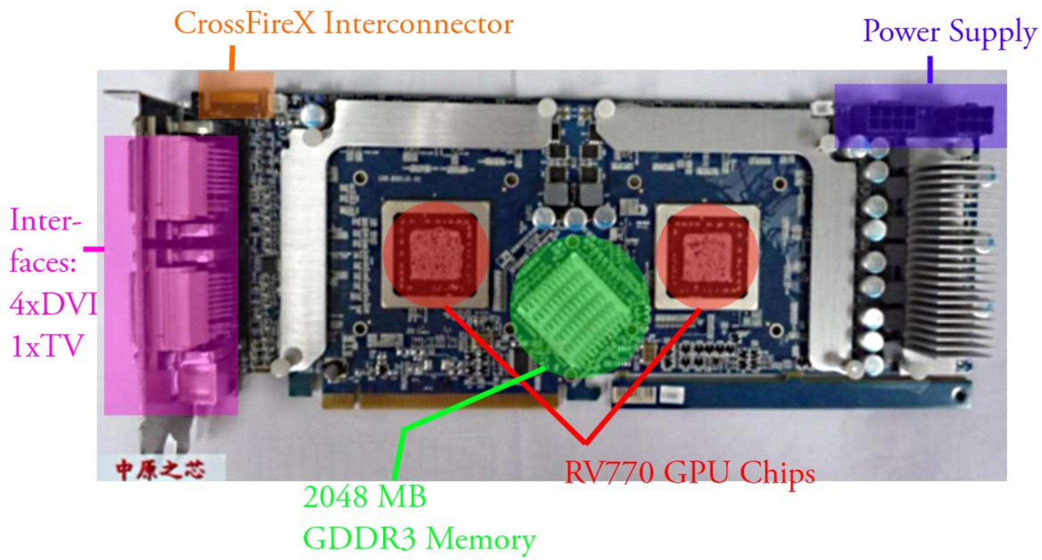


Bild 8 physische Architektur der Grafikkarte

Hierbei sind die RV770-Chips die beiden GPUs der Grafikkarte. Der RV770-Chip ist einer der neuesten Prozessoren. Die neuesten und leistungsfähigsten Grafikchips und die ersten mit nativer DirectX 11-Unterstützung sind Cypress- und Hemlock-Chips . Dieser Chip wird in der ATI Radeon-Reihe HD 5000 genutzt, welche die aktuell größte Leistungsfähigkeit aufweisen. Die beiden Kerne sind intern durch ATI-spezifische Multi-GPU Technologie „CrossFire" verbunden. Es ist möglich, eine zweite dieser Karten in einem System zu installieren um die Rechenlast weiter aufteilen zu können. Die vier Grafikkerne sind untereinander dann mit CrossFireX verbunden. Dazu dient ein kleines, als Brücke eingesetztes Kabel welches am orange-markierten CrossFireX-Interconnector angeschlossen wird. Diese Rechenleistung ist für professionelle grafische Datenverarbeitung sowie für HighDefinition-Rendering auf einer Bildschirmwand (Display Wall) sinnvoll.

Der Grafikkarten- und Softwarehersteller NVIDIA besitzt eine ähnliche Technologie zur Verbindung mehrerer GPUs und Grafikkarten. Dies wird SLI (Scalable Link Interconnector), 3-Way-SLI und Quad-SLI genannt.

Die Grafikkartenhersteller setzen stark auf die Vereinigung mehrerer Grafikchips auf einem PCB um so eine größere Skalierbarkeit für alle Grafikkarten bieten zu können.

Der Grafikchip, unabhängig ob NVIDIA GT200 oder ATI RV770, ist jedoch an sich eine Verknüpfung einer Vielzahl von Grafikkomponenten. Desweiteren ist, ähnlich dem Hauptspeicher der CPU, der Grafikspeicher in unterschiedliche Bereiche aufgeteilt. Diese Aufteilungen werden durch die logische Architektur der Grafikkarte deutlich, worauf die folgende Grafik bezogen ist.

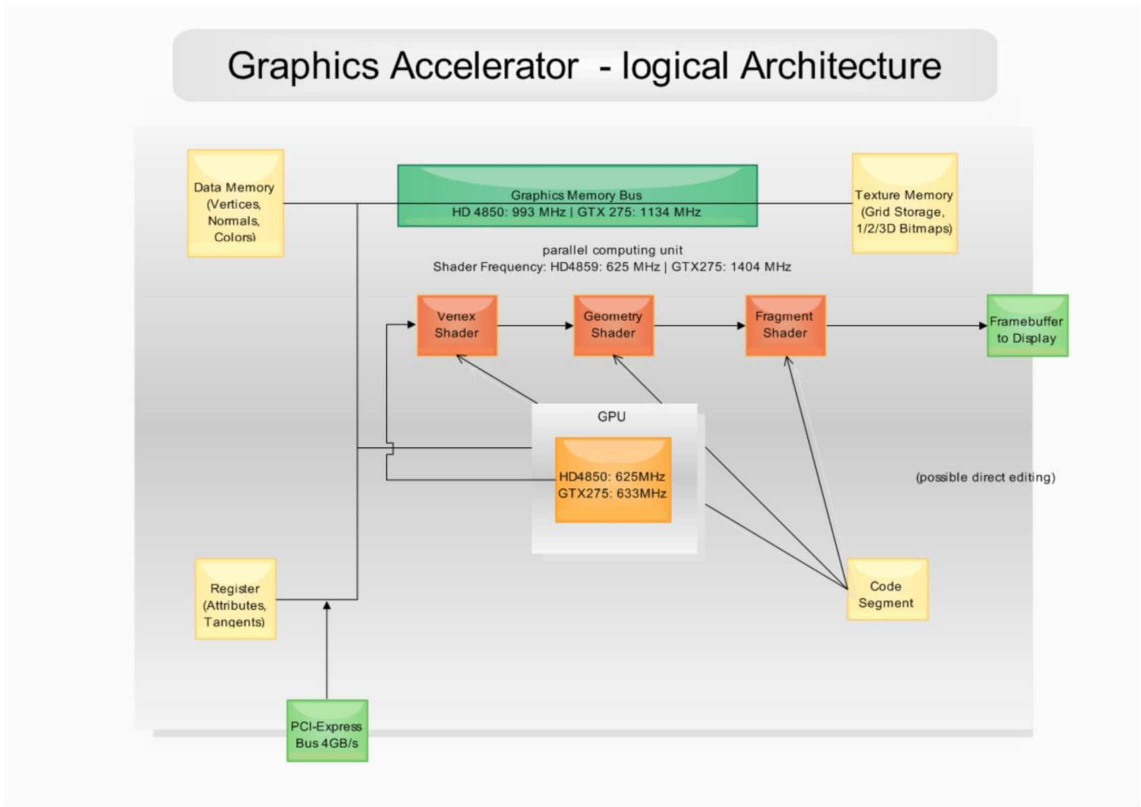


Bild 9 logische Architektur der Grafikkarte

Hierbei ist zu erkennen, dass die Shader-Einheiten, deren Funktionsweise später noch erläutert wird, als ein Verbund vom „Compute Units“ existieren. Diese sind logisch vom eigentlichen Kern getrennt, der für die Transition der Daten durch die Pipeline verantwortlich ist. Jedoch sind die Shader physisch in den Chip integriert. Dabei werden die Shader bei ATI-Chips typisch mit dem Kerntakt des Prozessors angesteuert, während bei NVIDIA-Karten die Shader unabhängig von der GPU mit einem eigenen Takt angesteuert werden.

Desweiteren ist der Speicher logisch in 4 Bereiche eingeteilt. Physisch befinden sich die Register direkt im Grafikkern. Daten-, Textur- und Codespeicher gehören zum Grafikspeicher, in welchem sich ebenfalls mehrere Bildschirm-Zwischenspeicher sowie Speicher für die Hintergrundbildschirme (Framebuffer und Pixelbuffer) befinden.

Zu verarbeitende, dynamische Grafikdaten werden über den PCI-Express Bus an die Grafikkarte gesendet. Statische, nicht ändernde Bild-zu-Bild-Daten sind resident im Datenspeicher verfügbar. Diese Daten werden mit Hilfe des Grafikprozessors durch die Renderpipeline verarbeitet. Dabei werden die in vorigen Absätzen erwähnten Aktionen durch die Shader ausgeführt. Das entstehende Bild wird über den Framebuffer an den Bildschirm weitergegeben.

2.1.5. Prinzipien der Grafikkomposition

Durch die Aufteilung der Renderpipeline wird der Algorithmus auf unabhängigen Komponenten vollständig durchlaufen. Dabei entstehen je GPU Teile des gesamten Ausgangsbildes. Diese Teilbilder müssen demnach zusammengeführt werden, weshalb dieser Schritt in der Renderpipeline im Bereich „Composition and Displayment“ zu finden ist. Bei dieser Grafikkomposition gibt es verschiedene, durch die Grafikkartenhersteller entwickelte Ansätze. Desweiteren treten verschiedene Probleme auf, die die Leistungssteigerung mindern.

Ein erstes Problem stellen Objekte dar, die direkt auf der Schnittkante der Grafikchips liegen. Diese müssen von jeder GPU, die die Objekte in ihrer Renderfläche einschließen, getrennt berechnet werden. Dies führt bei hoch tesselierten Objekten (Objekte mit vielen Vertices) zu einem feststellbaren Leistungsverlust der Darstellung. Dieses Problem kann nicht komplett vermieden werden, da je nach Betrachtungsposition jedes Objekt auf einer Schnittfläche liegen könnte. Das Problem kann jedoch gemindert werden, indem man hauptsächlich gering tesselierte Objekte verwendet. Dies ist jedoch in professionellen Anwendungen nicht möglich. Daher kann durch einen Algorithmus das Objekt vor der Renderpipeline an der Schnittfläche in zwei unabhängige Objekte geteilt werden. Dieser zusätzliche Tesselationsschritt kostet zwar auch Leistung, ist jedoch bei hoch tesselierten, großen Objekten schneller als das doppelte Rendering des Objekts.

Ein weiteres Problem ist ein Phänomen bei der Grafikkomposition, das allgemein unter „Mikrorucklern“ bekannt ist. Mikroruckler treten bei einem Kompositionsverfahren namens AFR auf. Dabei dauert das Rendern von Frames gerader Anzahl länger als das Rendern der ungeraden Frames. Der Grund dafür ist ungewiss bzw. nicht offiziell bekannt. Dadurch benötigt im angesprochenen Modus eine Animation eine größere Rendergeschwindigkeit als in anderen Modi bzw. im Einzelmodus, um flüssig zu wirken. Die Lösung des Problems ist die Aktivierung eines anderen Kompositionsmodus.

Unter den Kompositionsmodi gibt es zwei Standardalgorithmen, die von allen Multi-GPU Systemen unterstützt werden, sowie herstellerabhänge Sondermodi.

Ein erster Kompositionsmodus ist Alternate Frame Rendering (AFR). Dabei werden aufeinanderfolgende Bilder durch unterschiedliche Grafikprozessoren berechnet. Bei einer 2-GPU-Konfiguration wird jeder erste Frame durch GPU 0 berechnet, jeder zweite Frame durch GPU 1. Höhere Konfigurationen skalieren dementsprechend.

Ausgehend von der Tatsache, dass dies der erste entwickelte Kompositionsmodus und daher der Standardmodus ist, treten dabei jedoch verschiedene Probleme auf. Neben den aufgeführten Mikrorucklern werden die Daten des Folgebildes durch die Sprungvorhersageeinheit (Branch Prediction Logic) der CPU vorberechnet. Durch den dynamischen Charakter einer Animation sind die vorausberechneten Werte häufig falsch.

Die Daten müssen bei einer Fehlberechnung vor Aufruf des Folgebildes neuberechnet werden. Dies schränkt die Leistung ein und kann der Grund für Mikroruckler sein. Desweiteren ist AFR aufgrund seiner suboptimalen Skalierbarkeit ein ungünstiger Kompositionsmodus. Die Darstellungsgeschwindigkeit des einzelnen Bildes wird dadurch nicht erhöht. Somit können professionelle Anwendungen mit großem Datenaufkommen pro Bild nicht von diesem Kompositionsmodus profitieren.

Ein weiterer Standardkompositionsmodus ist Split Frame Rendering (SFR). Bei SFR wird die Renderfläche in gleichgroße Bereiche aufgeteilt, deren Anzahl der Anzahl der vorhandenen GPUs entspricht und die dann von den jeweiligen Grafikprozessoren berechnet werden. Am Ende der Berechnung werden die Teile zu einem Bild zusammengefügt.

Dieser Kompositionsmodus entspricht einer logischen Aufteilung. Mikroruckler sind beim SFR-Modus nicht vorhanden. Durch die gute Skalierbarkeit gilt SFR als einer der schnellsten Kompositionsmodi. Daher ist SFR auch ein Kompositionsmodus, der als Standard zu jeder Multi-GPU Konfiguration gehört. Aufgrund dieser positiven Fakten basiert die entwickelte Software auf diesem Kompositionsmodus.

Der erste hier vorgestellte, herstellergebundene Kompositionsmodus ist SuperTiling von ATI Technologies. Beim SuperTiling wird die Renderfläche in mehrere gleichgroße Bereiche aufgeteilt, die ähnlich einem Schachbrett angeordnet ist. Die logisch weißen Flächen werden dabei von GPU 0 berechnet, die logisch schwarzen von GPU 1. Am Ende der Renderpipeline wird komponiert.

Mit diesem Modus wird die harte Schnittfläche der Renderflächen umgangen. Durch die proprietäre Hardwareimplementation wird das Mehrfachrendern verhindert. SuperTiling bietet für sich eine relativ geringe Skalierbarkeit. Durch eine gute Verbindung mit AFR und SFR kann jedoch eine hohe Skalierbarkeit erreicht werden. Schwachpunkte dieses Kompositionsmodus sind die Herstellerbindung und die Komplexität des Algorithmus, was die Softwareimplementation des Algorithmus erschwert.

Ein letzter ATI-spezifischer Kompositionsmodus ist Super Anti-Aliasing (SuperAA). Die Hintergrundidee besteht darin, durch mehrerer GPUs eine hohe Kantenglättung zu erreichen, um die Ausmaße der Digitalisierung der Anzeige zu mindern. Durch den Standard der TFT-Displays, die ausschließlich mit digitalen Signalen angesteuert werden können, wird der Effekt vergrößert, dass schräge Kanten der Darstellung in eine Treppenform übergehen. Die Technik, diesen Effekt durch gewichtete Verrechnung mehrerer, versetzter Bilder zu minimieren, wird als Kantenglättung oder auch Anti-Aliasing bezeichnet. Dabei gibt die Anzahl (2-fach, 4-fach etc.) an, wie viele gewichtete Einzelbilder zur Komposition verwendetet werden. Der höchste Faktor für Einzel-GPUs ist momentan 16. Der Vorgang der Kantenglättung reduziert drastisch die Darstellungsgeschwindigkeit, da mit jeder Stufe die doppelte Anzahl an Rendervorgängen benötigt wird. Bei der Anwendung des SuperAA kann demzufolge das aktuelle Bild mehrfach von mehreren Karten gerendert werden. Der maximale Anti-Aliasing-Faktor ist dabei multiplizierbar mit der Anzahl der GPUs. Dies

ermöglicht eine nahezu natürliche Darstellung bei entsprechender Anzahl von Grafikprozessoren.

Jedoch besteht in diesem Fakt der Nachteil der Technologie. Die Skalierbarkeit ist eingeschränkt, da ab einem gewissen Faktor keine optischen Unterschiede erkennbar sind. Die Verbindung mit AFR oder SFR ist ebenfalls relativ kompliziert, da die Prozessoren, die die gleiche SFR-Fläche rendern, möglichst absolut synchron arbeiten müssen. Auf der einen Seite bedingt dies Konfigurationen, bei denen mit nativen CrossFire-Lösungen (zwei GPUs auf einem PCB) gearbeitet werden muss. Auf deren anderen Seite verstößt dies gegen das CrossFire-Prinzip, bei dem die Zusammenarbeit von heterogenen ATI-Karten (Grafikkarten mit verschiedenen Chips und Taktungen) gewährleistet sein muss.

Zur Veranschaulichung der bisher vorgestellten Modi folgt eine Skizze der Kompositionsmodi.

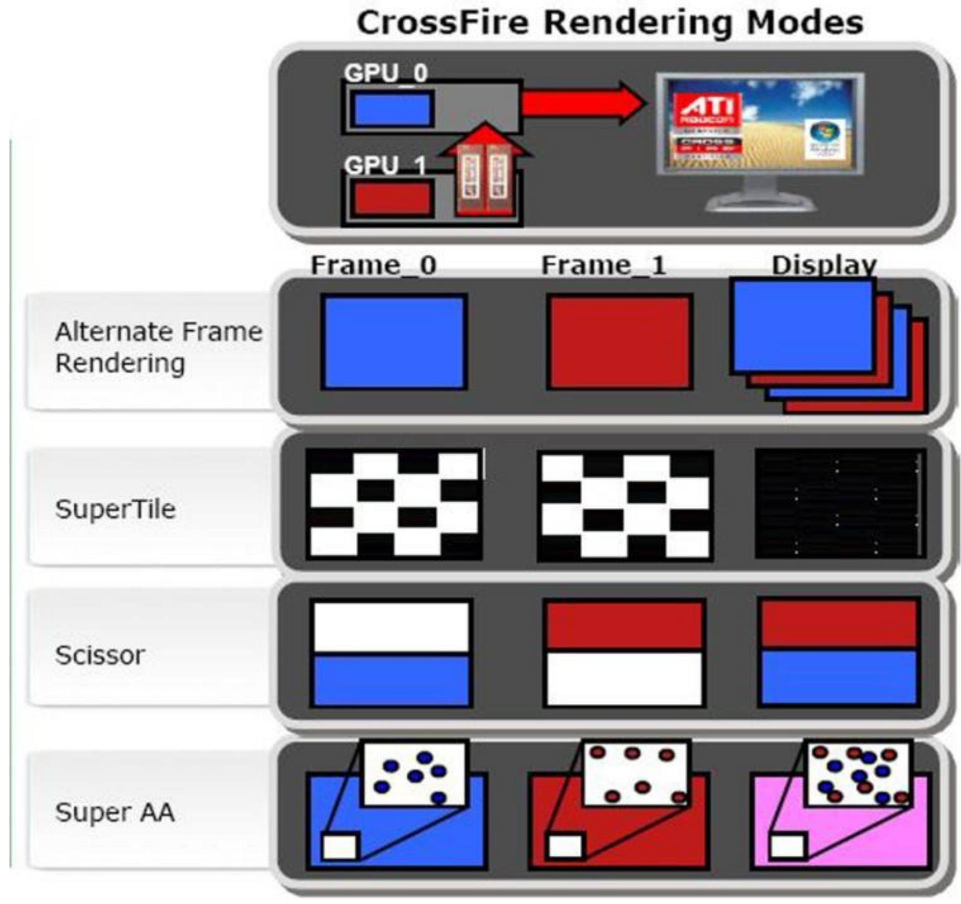


Bild 10 ATI Technologies Kompositionsmodi [ATI Technologies]

Parallel zu den ATI-spezifischen Kompositionsmodi existieren von NVIDIA ebenfalls herstellerspezifische Kompositionslösungen. Der erste hierbei zu erläuternde Modus ist Multi View. Dabei wurde die Entwicklung auf die Ansteuerung von mehreren Monitoren für eine Anwendung konzentriert. Die Renderfläche und das damit zusammenhängende

Sichtfeld werden dabei je Karte um eine Bildschirmdimension erweitert. Damit es möglich, durch Anschluss einer gewissen Anzahl von Bildschirmen und Karten, ein 360° Sichtfeld zu erzeugen – den virtuellen Raum komplett in die Realität zu übertragen. Dabei steht diese Idee in Konkurrenz zur Entwicklung von 3D-Monitoren.

Obwohl diese Lösung logisch gut skalierbar ist scheitern hohe Anzahlen von Grafikkarten im Multi View-Modus an der physischen Realisierbarkeit, da für jede Karte 2 Monitore vorgesehen sind. Der Modus ist eine Lösung für große, komplett durch Monitore verdeckte Wände. AMD hat in Verbindung mit ATI eine Konkurrenzlösung zum Multi View-Modus, genannt EyeFinity, entwickelt. Diese ist jedoch nur in Konfiguration mit AMD Board, AMD Phenom-Prozessor und ATI Radeon Grafikkarte einsetzbar.

Die folgende Grafik verdeutlicht an einem kleinen Beispiel Multi View.

Bild 11 NVIDIA Multi View-Modus [NVIDIA]

Ein weiterer Zusatzmodus von NVIDIA ist das Load Balancing (LB). LB ist eine Erweiterung von SFR. Es basiert auf der Idee, ein System aus heterogenen NVIDIA-Chips optimal zu betreiben. Dabei wird die Renderfläche der leistungsstärkeren GPU um die Leistungsdifferenz, basierend auf der Kerntaktfrequenz, zur leistungsschwächeren GPU erweitert. Load Balancing optimiert den SFR-Modus. Als Erweiterung von SFR erbt dieser Kompositionsmodus die Vorteile der Skalierbarkeit und gleicht eine kleine Schwäche des Standardmodus aus.

Folgende Grafik zeigt die Funktionsweise von Split Frame Rendering Load Balancing.

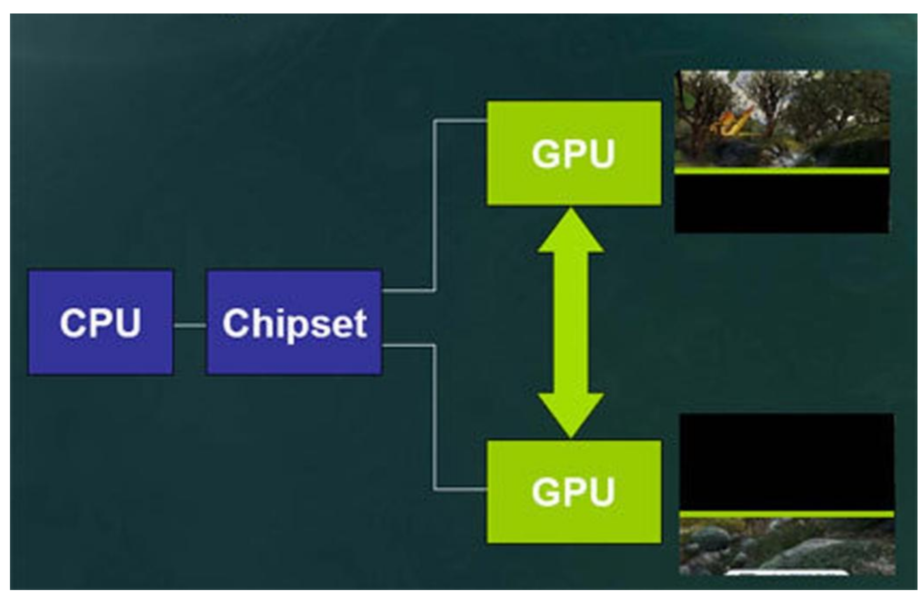


Bild 12 NVIDIA Load Balancing SFR-Modus [NVIDIA]

2.2. Kennwerte zur Leistungsfähigkeit von Grafikkarten

Eine wissenschaftliche Untersuchung zur Optimierung eines Prozesses bedingt Wissen um die Randbedingungen des Prozesses. Diese wurden im vorigen Abschnitt erläutert. Um die Güte der Optimierung zu bestimmen bedingt es Messwerte rationeller Kenngrößen des Prozesses. Diese werden folgend untersucht.

Da der Prozess der Aufteilung der Renderpipeline, wie bereits erläutert, durch Hardware und Software gleichermaßen bestimmt wird, ist es nötig ebenfalls Kenngrößen beider Seiten zu untersuchen. Aus der Untersuchung der Hardwarekenngrößen in Bezug auf die Komponenten der späteren Testsysteme können Leistungsprognosen abgeleitet werden.

Daraus resultierend werden folgend zuerst bestimmte Hardwarekenngrößen untersucht. Zu diesen zählen hauptsächlich die PCI-Express Schnittstelle zur Grafikkarte, Grafikspeichergröße, Grafikspeicherbandbreite und Grafikprozessoranzahl sowie Shaderanzahl. Daraufhin werden die softwareseitig messbaren Kenngrößen wie Ladezeiten der einzelnen Teilprozesse sowie die Bildwiederholfrequenz beleuchtet. Schlussendlich wird eine Übersicht der Hardwareparameter in Bezug auf spezielle Grafikbeschleuniger vorgestellt. Eine Auswertung der Parameter in Bezug auf die Komponenten der Testsysteme wird darauffolgend zu einer Leistungsprognose der Geschwindigkeit basierend auf die Hardware führen.

2.2.1. Schnittstelle PCI-Express

Computergrafische Modelle werden vor dem Rendern von der CPU in den Hauptspeicher geladen und zur Weiterverarbeitung vorbereitet. Die computergrafische Interpretation der Pixel und die Komposition einer Szene finden auf der Grafikkarte statt. Beide Komponentenpaare - CPU und Hauptspeicher sowie GPU und Grafikspeicher - sind mit schnellen Bussystemen ausgestattet, mit welchen die Daten intern übertragen werden. Hierbei zu nennen wären der Front Side Bus (AMD Phenom, Intel Core 2 Duo, Intel Core 2 Quad) und der QuickPath Interconnect Bus (Intel Core i7). Grafikkarte und Grafikspeicher bilden ein Streamsystem. Der Bus zwischen beiden besitzt keinen Namen, jedoch wird üblicherweise die Speichertaktfrequenz bei Grafikprozessoren mit angegeben. Dabei wird über den QPI-Bus Datendurchsatz von bis zu 25,6 GB/s erreicht. Ein Streamsystem erreicht (Beispiel AMD/ATI HD4870 X2) einen Datendurchsatz von bis zu 230 GB/s.

Jedoch werden die zu verarbeitenden Daten von der CPU an die GPU über einen weiteren Bus weitergegeben. Dabei hat sich während der letzten Jahre PCI-Express als Standardbus zur Verbindung von CPU und GPU (über den Memory Controller Hub) etabliert.

PCI-Express ist wie folgt definiert:

"PCI-Express („Peripheral Component Interconnect Express“, abgekürzt: PCIe oder PCI-E) ist ein Erweiterungsstandard zur Verbindung von Peripheriegeräten mit dem Chipsatz eines Hauptprozessors. [...]" [Wik10]

Der PCI-Express Bus ist eine serielle Punkt-zu-Punkt-Verbindung von Peripheriegeräten untereinander sowie zum Chipsatz (MCH). Dabei gilt jede Verbindung als eine Bahn. Diese Bahn hat im PCI-Express 1.0 Standard eine Geschwindigkeit von 250 MB/s. Aktueller Standard zur Verbindung von MCH und Grafikkarte ist PCI-Express 2.0 mit einem Datendurchsatz je Bahn von 500 MB/s. Durch Bündelung mehrerer Bahnen zu einer Verbindung zwischen zwei Geräten wird der Datendurchsatz entsprechend erhöht. Die Anzahl der Bahnen wird durch den Faktor des PCI-Express Anschlusses symbolisiert. So hat eine aktuelle Grafikkarte einen PCI-Express x16 2.0 Anschluss. Dies bedeutet einen Datendurchsatz von 8000 MB/s (rund 7,8 GB/s).

Entsprechende Datendurchsätze des PCI-Express Anschlusses können folgendem Bild entnommen werden.

Datenrate PCI-Express
(theoretisch)

	PCIe 1.0	PCIe 2.0	PCIe 3.0
1x	250 MB/s	500 MB/s	1000 MB/s
2x	500 MB/s	1000 MB/s	2000 MB/s
4x	1000 MB/s	2000 MB/s	4000 MB/s
8x	2000 MB/s	4000 MB/s	8000 MB/s
16x	4000 MB/s	8000 MB/s	16000 MB/s
32x	8000 MB/s	16000 MB/s	32000 MB/s

Bild 13 Übersicht PCI-Express Bandbreiten [Wikipedia]

Im Vergleich zu den Datendurchsätzen von QPI oder dem Streamingsystem ist erkennbar, dass in einem computergrafischen System die Übertragung von Daten von CPU an GPU über den PCI-Express Bus der „Flaschenhals“ der Anwendung sein wird.

Daher ist in der Entwicklung darauf zu achten, die Anzahl der dynamisch ändernden Daten während des Renderns möglichst gering zu halten. Der Großteil der Daten sollte in Form von Arrays und Listen im Grafikspeicher resident sein, sodass diese nur noch durch Matrixtransformationen verändert werden müssen. Die Übertragung einer 4x4-Transformationsmatrix über den PCI-Express Bus ist dabei zeitlich unerheblich und ändert nahezu nichts an der Darstellungsgeschwindigkeit der Anwendung.

2.2.2. Grafikspeichergröße

Der Grafikspeicher ist ein spezielles DDR-Speichermodul für Grafikkarten. Analog zu DDR-RAM Modulen existieren GDDR-Module in verschiedene Versionen. Die Kapazität von Grafikspeicher wird in Megabyte oder Gigabyte angegeben.

Momentan erhältliche Grafikkarten haben Grafik-DDR-Module in den Versionen GDDR2, GDDR3 und GDDR5 auf den Grafikbeschleunigern verbaut. Dabei ist GDDR2 die Basis mit Standard-Double Data Rate-Verfahren. GDDR3-Module werden im Dual Channel-Modus betrieben, welches ebenfalls als Hauptspeichertechnologie bekannt ist. GDDR5-Module sind auf Grafikkarten mit den neuesten Chips (ATI RV770, RV790 und Cypress sowie NVIDIA Fermi GF100) verbaut und werden im Quad Pumped-Modus betrieben. Dies hat Auswirkungen auf die später erläuterte Grafikspeicherbandbreite.

Für Grafikspeicher gibt es zwei mögliche Einteilungen. Die erste Einteilung basiert auf einem grafischen Ansatz. Dabei ist der vorhandene Grafikspeicher, wie schon bei der logischen Grafikarchitektur aufgeführt, in Code-, Data-, Pixel und Texture Memory unterteilt. Der Grafikspeicher ist in jenem Fall dafür vorgesehen, entsprechende grafische Daten resident auf der Grafikkarte zu speichern. Dadurch wird eine ständige Übertragung großer Datenmengen über den PCI-Express Bus vermieden und die Animation beschleunigt.

Zum Speichern entsprechender Daten stehen Speicherallokationskommandos der Grafikschnittstelle zur Verfügung. Dabei werden Grafikspeicherobjekte (Buffer Objects) angelegt und die im Hauptspeicher vorhandenen Daten zu Beginn des Renderings auf die Grafikkarte übertragen. Dieser Vorgang wird als Mapping bezeichnet. Es existieren Vertex-, Texture-, Pixel- und Frame Buffer Objects. Dabei werden auf Pixel- und Frame Buffer Objects die gleichen Operationen ausgeführt. Sie befinden sich daher im gleichen Speicherbereich.

Das Wissen über diese Unterteilung ist für eine optimale Zusammensetzung der Szene nötig. Der Grafikspeicher, der insgesamt bei aktuellen Karten zwischen 1024 und 4096 Megabyte groß ist, kann keine Einzelobjekte dieser Größe (unabhängig vom Typ) komplett abspeichern. Daher ist eine Simulation mit einem ausgewogenen Verhältnis zwischen Vertex- und Bildinformationen sehr viel performanter als eine Simulation mit überdurchschnittlich hoch tesselierten Objekten.

Eine weitere Möglichkeit der Einteilung bezieht sich auf die General Purpose Graphics Processing Unit (GPGPU) - Berechnungen. Hierbei besteht der Grafikspeicher aus globalem, gemeinsamem und lokalem Speicher. Globaler Speicher ist der Texturspeicher der Grafikkarte, der große, von jeder Shadereinheit nutzbare Daten abspeichert. Hierbei wird der Zugriff durch blockierende Schreibsperren geregelt. Der Lesezugriff ist nicht limitiert. Gemeinsam genutzte Speicher sind Teile des Pixelspeichers. Dieser besteht exklusiv für jede

Recheneinheit und dient zum Austausch von Daten zwischen Shadern einer gemeinsamen Recheneinheit. Im lokalen Speicher sind Variablen, welche innerhalb eines Shader angelegt werden. Diese sind außerhalb des Shader weder sichtbar noch lesbar.

2.2.3. Grafikspeicherbandbreite

Die Grafikspeicherbandbreite beschreibt die Geschwindigkeit, mit der Grafikprozessor und Shader auf den Grafikspeicher zugreifen, und wird in Megabyte pro Sekunde (MB/s) oder Gigabyte pro Sekunde (GB/s) angegeben. Ein Synonym für die Bandbreite ist der Datendurchsatz.

Die Grafikspeicherbandbreite entspricht dem Faktor von Speichertaktfrequenz und der Speicherbusbreite.

$$x\,[MHz] \cdot y\,[bit] = x \cdot\ 1000000\ \cdot\ \left[\frac{1}{s}\right] \cdot y \cdot \frac{1}{8}\,[Byte]$$

$$= \frac{x \cdot y \cdot 1000000}{8}\ \cdot\ \left[\frac{Byte}{s}\right] = \frac{x \cdot y}{8} \cdot \left[\frac{MByte}{s}\right]$$

Formel 1: Berechnung Bandbreite

Die Speicherbusbreite wird von der Anzahl der Buskanäle und der Anzahl an Bit bestimmt, die durch einen Kanal adressiert werden können. Grafikprozessoren besitzen einen 32 Bit Adressbus und zwischen 8 und 16 Kanälen. Daraus resultierende Speicherbusbreiten liegen zwischen 256 Bit und 512 Bit.

Die Speichertaktfrequenz wird von der Grundtaktfrequenz und der Art des Grafikspeichermoduls bestimmt, da diese unterschiedliche Speichertechnologien besitzen. Die Grundtaktfrequenz ist zwischen den Grafikkarten sehr unterschiedlich. Die nutzbaren Technologien sind der Dual Channel Modus (Verdopplung der Frequenz gegenüber GDDR2) und Quad-Pumped (Vervierfachung der Frequenz gegenüber GDDR2).

2.2.4. GPU Anzahl

Der Auswirkungen der GPU Anzahl ist der Fokus der Untersuchung der Arbeit. Der Parameter gibt an, wie viele grafische Prozessoren am Rendern beteiligt sind. Diese können sich auf einem oder mehreren, unterschiedlichen Karten befinden und besitzen jeder seinen eigenen Grafikspeicher.

Theoretisch sollte sich die Bildwiederholfrequenz mit der Anzahl der GPUs linear erhöhen. Damit verbundene Technologien wurden im Abschnitt „Grafikkomposition“ bereits erläutert.

2.2.5. Shadereinheiten und parallele Recheneinheiten

Shadereinheiten sind in Vertex-, Geometry- und Pixelshader unterteilt. Bedingt durch den Fakt, dass mehr Fragmente als Vertices in einem Rendering existieren ist die Anzahl an Fragment Shader größer als die Anzahl an Vertex Shader. Die Anzahl an Geometry Shader liegt typischerweise zwischen beiden Werten.

Shader sind kleine, programmierbare Mikroprozessoren auf der Grafikkarte, die Datentransformationen an Vektoren und Farben durchführen um bestimmte Effekte zu erzielen.

Shader werden heutzutage durch Hochsprachen wie „C for Graphics“ (Cg), „Graphics Library Shading Language“ (GLSL) und „High Level Shading Language“ (HLSL) programmiert. Diese Sprachen besitzen einheitliche, C-ähnliche Operationen welche direkt im Grafikchip implementiert sind. Der Shadercode wird auf dem Hauptprozessor entwickelt und zur Laufzeit des Programms in den Codespeicher der Grafikkarte geladen. Daraufhin wird der Shader kompiliert und in einem Shaderprogramm auf der Grafikkarte gelinkt. Es existieren Operationen zur Fehlerabfrage, welche zur sicheren Programmierung erforderlich sind.

Der Begriff „parallele Recheneinheit“ bezeichnet eine Kombination aus Vertex-, Geometry- und Pixelshader, die als Compute Shader allgemeine Operationen ausführen können. Diese Recheneinheit sind parallel arbeitende Einheiten, welche Ähnlichkeiten zu nativen Multiprozessoren aufweisen. Daher bieten sich Grafikkarten als schnelle Prozessoren für die Verarbeitung von parallelisierbaren Prozessen an.

Die praktischen Auswirkungen vom Einsatz der Shader zur Berechnung und Darstellung computergrafischer Prozesse ist weitgehend ungeklärt. Theoretisch skaliert die Bildwiederholrate linear mit der Anzahl der Shader und Recheneinheiten.

2.2.6. Ladezeiten

Eine softwareseitige Kenngröße für das Rendern einer Szene ist die Ladezeit. Dabei ist die Ladezeit in drei Phasen eingeteilt.

Die erste Phase des Ladens ist die anwendungsseitige Vorbereitung der Szene. Hierbei werden Grafikumgebungen, sogenannte Kontexte, erstellt, Modeldaten aus Dateien gelesen,

Bilder für Texturen und Höhenkarten gelesen und vorbereitet sowie die nötigen Bibliotheken, APIs und Frameworks für den Einsatz der verschiedenen Technologien zur Verfügung gestellt. Diese Phase ist in der professionellen grafischen Datenverarbeitung die kurzweiligste, da die Daten nur in Hauptspeicher geladen und vorbereitet werden. Der Vorgang wird komplett vom Hauptprozessor ausgeführt. Resultate dieser Phase sind vorbereitete grafische Daten im Hauptspeicher. Der Vorgang kann jedoch durch Laden von Daten auf verteilten Rechennetzen oder langsamen Massenspeichern verzögert werden.

Die zweite Phase ist die Übertragung der im Hauptspeicher verfügbaren Daten auf die Grafikkarte über den PCI-Express Bus, um so viele Datensätze wie möglich resident im Grafikspeicher zu lagern. Desweiteren werden die übertragenen Objekte tesseliert. Ebenso werden Texturen mit Texturkoordinaten für das spätere Mapping auf die Objekte versehen und komprimiert. Je nach Anzahl und Größe der Datensätze kann dies einen großen Anteil an der Vorverarbeitungszeit beanspruchen. Ein weiterer Nachteil ist die unzureichende Möglichkeit, durch Bildschirmausgaben zu signalisieren, welche Daten momentan verarbeitet werden bzw. dass Daten verarbeitet werden. Einzige Möglichkeit ist die Messung der Zeit vom Start der Übertragung bzw. dem Ende von Phase Eins bis zum ersten Beginn des ersten Durchlaufs der Renderpipeline.

Die dritte Phase ist das Rendern an sich. Dabei wird die Zeit gemessen, die benötigt wird, um ein Ausgabebild zu erzeugen. Da dies bei Animationen in der Regel mehrmals in der Sekunde geschieht wird dazu eine andere Kenngröße verwendet. Bei hochdetaillierten 3D-Szenen mit diversen aufwendigen Bilderzeugungsverfahren wie Raytracing und Radiosity kann diese Phase jedoch mehrere Minuten oder Stunden benötigen.

2.2.7. Bildwiederholrate

Die Bildwiederholrate gibt die Anzahl der Ausgabebilder auf das Darstellungsmedium (Bildschirm, Display Wall, CAVE etc.) pro Sekunde an. Die Maßeinheit ist „Frames per Second“ (fps).

Dieser Parameter beschreibt, wie im vorigen Absatz angedeutet, indirekt die Verweildauer der Renderpipeline pro Bild in Phase Drei der Ladezeiten. Der Parameter wird durch das 3D-Datenvolumen, hardwaretechnische Kenngrößen, Darstellungsauflösung und verwendetes Betriebssystem bestimmt und begrenzt. Dabei regeln Microsoft Windows-basierte Betriebssysteme das Rendern während der Laufzeit auf die ehemalige Standard-Darstellungsfrequenz für CRT-Monitore von 60 Hertz (entsprechend 60 fps) ab. Dieses Verhalten ist über die Entwicklung der Windows Betriebssysteme geblieben. Bei Linux-Distributionen und MacOS gibt es dieses Limit nicht.

Die automatische Aufnahme und Abspeicherung der Bildwiederholfrequenz ist kompliziert. Der Ansatz und dazugehörige Prozess ist simpel. Erste Möglichkeit besteht darin, die Zeit zu bestimmen, die das Darstellen des aktuellen Frames benötigt und darauf folgend den Reziproken dieser Messung zu nehmen. Dabei werden jedoch sehr starke Sprünge deutlich. Ein weiterer Ansatz wäre die Zeitnahme als Hintergrundprozess und die parallele Erhöhung der Bildanzahl. Dieser Ansatz hat jedoch Nachteile durch Verdrängung niedrigpriorisierter Hintergrundprozesse auf der CPU und die Möglichkeit, nur ganze Zahlen messen zu können. Dadurch verliert die Messung, besonders bei geringen Bildwiederholfrequenzen, an wichtigen Details.

Ein generelles Problem der Messwertermittlung ist die Anzahl der Messwerte. Bei einem fünf Minuten langen Rendering mit durchschnittlich 35 fps entstehen 10500 Messwerte, wobei die Anzahl bei höheren Bildwiederholraten rapide ansteigt. Diese Anzahl an Messwerten erschwert die Auswertung. Eine geeignete Lösung für dieses Problem ist Anwendungsabhängig.

2.2.8. Leistungssteigerungsprognosen

Basierend auf den hardwaretechnischen Kenngrößen wurde eine Übersicht über die wichtigsten Grafikbeschleuniger mit den jeweiligen Kenngrößen ermittelt. Die Auswahlkriterien wurden im Abschnitt 2.1.3 „Entwicklung der Grafikbeschleuniger“ bereits erläutert. Diese Tabelle wurde um Angaben der Speicherbandbreite ergänzt, um einen kompletten Überblick über alle erläuterten Kennwerte zu bieten.

	NVIDIA GeForce7600 Go	NVIDIA GeForce 8800GT	NVIDIA GeForce GTX275	ATI Radeon HD4850X2	NVIDIA Fermi	Intel Larrabee	NVIDIA Quadro FX 5800	ATI FirePro 3D V8750
GPUs	1	1	1	2	4	around 32	1	1
ShaderUnits	8 P + 5 V	112	240	800	512	16	240	800
CLOCK:								
			Memory: GDDR2 - 1x; GDDR3 - 2x; GDDR5 - 4x					
Memory Tech	GDDR3	GDDR3	GDDR3	GDDR3	GDDR5	GDDR5	GDDR3	GDDR5
Core	450 MHz	600 MHz	633 MHz	625 MHz	?	2000 MHz	650 MHz	750 MHz
Memory	400 MHz	900 MHz	567 MHz	993 MHz	?	?	816 MHz	900 MHz
Shader	450 MHz	1500 MHz	1404 MHz	625 MHz	?	?	1476 MHz (Same Core as GTX285)	750 MHz
Capacity								
Main Mem	256 MB	512 MB	896 MB	1024 MB	1536 MB	?	4096 MB	2048 MB
Vert&Tex	54 MB		198 MB	-				
Pixelbuffer	16 MB		64 MB	64 MB				
Bandwidth	12,8 GB/s	57,6 GB/s	127 GB/s	127 GB/s	?	?	102 GB/s	115,2 GB/s
Chip Architecture	G73M	G92	GT200b	Stream/RV770	GF100	x86	GT200GL	RV770XT
Price	no distribution anymore	100-140€; distribution expiring	220 €	200 €			3.499 $	1.800 $

Tabelle 2 Leistungsfähigkeit aktueller Grafikchips

Auf Basis dieser Werte lassen sich für die Testsysteme (Grafikbeschleuniger 1,3 und 4) Leistungsprognosen im Verhältnis zu einander ermitteln. Dies bildet einen theoretischen Erwartungswert für die späteren Messungen und dient zur objektiveren Einschätzung.

Die Aufstellungen über die Leistungsprognosen sind im Anhang zu finden.

2.2.9. Echtzeitfähigkeit, Messfehler und Interprozesskommunikation

Echtzeitfähigkeit einer Anwendung ist eine graduelle Eigenschaft. Diese wird an Grenzwerten bestimmter Kennwerte der Anwendung festgelegt. In der grafischen Datenverarbeitung ist der bestimmende Kennwert die Bildwiederholrate der Darstellung. Dabei wird allgemein eine flüssige Darstellung von Animationen ab einer Bildwiederholrate von 25 fps angenommen. Dies bedeutet die Darstellung in Echtzeit.

Die Messung der in vorigen Absätzen besprochenen Kennwerte unterliegt Messfehlern. Diese sind in systematische und zufällige Messfehler unterteilt. Systematische Messfehler sind durch die Umgebung und das System der Messung bedingt. Diese können durch sorgfältige Analyse des Messsystems und der Anwendung minimiert werden. Zufällige Messfehler sind minimale Abweichungen der Messung. Wiederholungen der Messungen besitzen dabei immer den gleichen Fehler, deren Abweichung jedoch in Vorzeichen und Betrag streut.

In der speziellen Messung von Leistungsunterschieden in einer verteilten Umgebung können systematische Messungenauigkeiten durch ungenügende Analyse und Interprozess-Kommunikation (IPC) bedingt sein. IPC beschreibt die Interaktion paralleler Prozesse. Dabei unterscheidet man verwandte und nicht verwandte Prozesse.

Zur Regelung der Kommunikation zwischen jenen Prozessen existieren verschiedene Ansätze [Ste10] . Zur Kommunikation verwandter Prozesse werden lokale IPCs verwendet. Dabei teilen sich verwandte Prozesse einen gemeinsamen Kommunikationsraum (Informatik: Speicherraum), über welchen die Informationen ausgetauscht werden. Dieser ist den beteiligten Prozessen bekannt. Auszutauschende Informationen werden unter gemeinsamer Zugriffsverwaltung im Paket, genannt Chunk, auf das Kommunikationsmedium geschrieben bzw. davon gelesen. Kommunikation nicht verwandter Prozesse wird durch globale IPCs gelöst. Dabei werden Informationen zwischen verschiedenen Rechnern oder Rechennetzen (Hosts) ausgetauscht. Die Kommunikation kann durch Signale oder öffentlichen Speicherraum (beispielsweise Dateien) geregelt sein.

Für zwei interagierende, verwandte Prozesse ist das „Sender-Empfänger Problem“ zu lösen. Dabei werden Werte durch einen Prozess generiert und diese durch einen zweiten Prozess

verarbeitet. Dessen Ergebnisse können wiederrum vom ersten Prozess aufgenommen werden. Ein Beispiel dieses Problems in der Applikation ist die Kommunikation eines Grafikkerns mit seinem dazugehörigen CPU-Thread. Dieser gibt 3D-Daten an die GPU weiter, welche durch die Renderpipeline ein Ausgabebild generiert. Dies wird vom CPU-Thread gelesen und an die Darstellungs-GPU weitergegeben. Die Lösung dieses Problems gehört zu Kategorie der lokalen IPCs. Dabei kann eine „unnamed pipe“ genutzt werden. Dies ist ein durch den „Vaterprozesses“ geschützter Speicherbereich des Hauptspeichers.

In einem großen System mit mehreren Sendern und mehreren Empfängern, beispielsweise einem Rechennetz oder Cluster, muss das „Erzeuger-Verbraucher Problem“ gelöst werden. Dabei werden Daten von mehreren Komponenten oder Clients erzeugt. Verarbeitungseinheiten greifen anschließend auf Teile der Datensätze oder auf komplette Datensätze zu. Sie „verbrauchen“ Informationen. Zur Lösung des Problems werden globale IPCs genutzt. Dazu gehören beispielsweise Sockets oder Streams.

2.3. Grundlegende Techniken der Beispielanwendungen

Im vorigen Abschnitt wurde auf Kenngrößen der Messwertermittlung zur rationalen Bewertung der Güte der Optimierung eingegangen. Messwerte werden durch Abspeichern dieser Kennwerte (Logging) während der Ausführung eines oder mehrerer Beispielprogramme ermittelt.

Für repräsentative Messwerte müssen grafische Anwendungen und Demos genutzt werden, die parallelisierbare, aufteilbare Prozesse beinhalten und deren Algorithmen ebenfalls in einer der in Kapitel 2 genannten, professionellen Anwendungsgebiete Anwendung findet. Dabei gibt es einige Beispielprogramme deren anspruchsvolle grafische Berechnungen auf der CPU verarbeitet werden. Es steht eine Vielzahl an zu verbessernden Referenzsystemen und Algorithmen zu Verfügung. Jedoch sind die zur Aufteilung benötigten Technologien derart Innovativ, dass keine Beispielapplikation zur Messwertermittlung im aufgeteilten Modus existieren, weshalb eine eigene Rendersoftware entwickelt wurde. Die Algorithmen der darin enthaltenen Technikdemos werden folgend vorgestellt.

Dabei werden im ersten Absatz die geplanten Technikdemos vorgestellt. Darauf die daraus resultierenden, vorzustellenden Techniken sind:

- Bump Mapping
- Partikeleffekte
- Terrainkonstruktion
- Objektmodellierung
- Shaderprogrammierung
- Compute Shader und Physikengines

2.3.1. Beispielanwendungen

Bezüglich der Auswahl einer Referenzanwendung, welche vollständig auf der CPU berechnet wird, ist ein hochaufgelöstes Mental Ray-Rendering einer Tastatur mit der Software Autodesk Image Studio 2008 32-Bit gewählt worden. Das dabei zu rendernde Objekt ist das Produkt einer Diplomarbeit von Frau Rica Bünning (Diplomdesignerin, Hochschule Wismar).

Desweiteren wird zur Ermittlung der grafischen Leistungsfähigkeit durch OpenCL die Bildwiederholfrequenz bei einer Animation der Julia-Menge mit der Software „GPU Caps Viewer 0.3.8“ notiert.

Eine erste Demonstrationsanwendung zur Ermittlung der Komplexität einer Entwicklung eines Rendersystems ist die Animation eines Würfels mit der Oberflächenstruktur des Logos der Hochschule Wismar. Die hierbei verwendetet Technik zur Erschaffung der Oberflächenstruktur aus einer Textur wird Bump Mapping genannt. Desweiteren wird für den Effekt der Einsatz von Shadern benötigt.

Weitere Demos befassen sich mit der Darstellung von medizinischen Objekten, die mit Hilfe eines CTs aufgenommen wurden. Dafür ist Grundkenntnis im Umgang mit soliden Objekten und deren Datenformaten notwendig. Desweiteren dienen die Demos als Beispiel für das Rendern großer, hoch tesselierter Objekte.

Das erste CAD Demo, bei welchem die Darstellung auf mehrere GPUs aufgeteilt wird und dessen Ergebnisse komponiert werden, ist die Visualisierung der Freiheitsstatue. Es gelten hierbei die gleichen Randbedingungen wie bei den medizinischen Demos.

Eine letzte, große Demonstration zeigt eine Pyramide in der Wüste. Dabei werden die Staubpartikel und deren Geschwindigkeit durch ein selbstentwickeltes physikalisches Partikelsystem berechnet. Die Geschwindigkeit ist auf Tastendruck modifizierbar. Für die Darstellung dieser Demo ist Wissen über Partikelsysteme, Terrainkonstruktion aus Bildern, polygonale Objektmodellierung und Compute Shadern in Verbindung mit Physikengines nötig.

2.3.2. Bump Mapping

Bump Mapping ist eine Technik mit der feine Oberflächenstrukturen auf polygonale Objekte projiziert werden können. Dabei wird die Lichtberechnung des Objektes durch Verrechnung der Flächennormale mit dem dazugehörigen, in einem Bild gespeicherten Wert modifiziert.

Polygonale Objekte bestehen aus Eckpunkten und dazugehörigen Parametern. Diese Parameter sind zum Beispiel:

- Texturkoordinate
- Flächennormale
- Oberflächenfarbe

Dabei wird ein Objekt in der Renderpipline häufig durch Auswertung einer Formel eines bestimmten Lichtmodells koloriert. Standardlichtmodelle der heutigen Grafikschnittstellen sind das Ambiente-, Lambert- und Phong-Modell. Dabei setzt sich der Farbanteil einer jeden Grundfarbe (Rot, Grün und Blau) durch Anteile der Umgebungsbeleuchtung sowie eines diffusen und spiegelnden Anteils eines einfallenden Lichtstrahls zusammen [Her08] . Dabei geht mit der Flächennormale die Höhe eines bestimmten Punktes in die Berechnung ein. Im

Standardmodus werden jedoch die Flächennormalen in Einheitsvektoren umgewandelt, da die Höhe eines Punktes einer Fläche relativ zur Fläche an sich gleichbleibend ist.

Durch Modifikation gezielte Modifikation des Normalvektors kann jedoch der Effekt hervorgerufen werden, ein Punkt liege über seiner einschließenden Fläche. Dies würde in der Visualisierung als erhöhter Punkt erscheinen.

Zur Realisierung des Effektes muss die Länge des Normalvektors mit dem, in einer Textur als Höhenkarte, abgespeicherten Differenzwertes verrechnet werden. Dies ist eine Operation des Vertex-Shaders, auf den im folgenden Absatz eingegangen wird. Ein dabei auftretendes Problem ist, dass der Normalvektor nicht absolut sondern relativ zu seiner Fläche existiert. Jedoch finden Veränderungen in den Shadern grundsätzlich im globalen Welt-Koordinatensystem (World Coordinate Space). Daher muss eine Transformation der aktuellen Koordinaten vorgenommen werden. Die Modifikation der jeweiligen Flächennormale an einem Punkt wird im Koordinatensystem des jeweiligen Objekts vorgenommen (Object Coordinate Space).

Folgende Grafik illustriert die Koordinatensysteme.

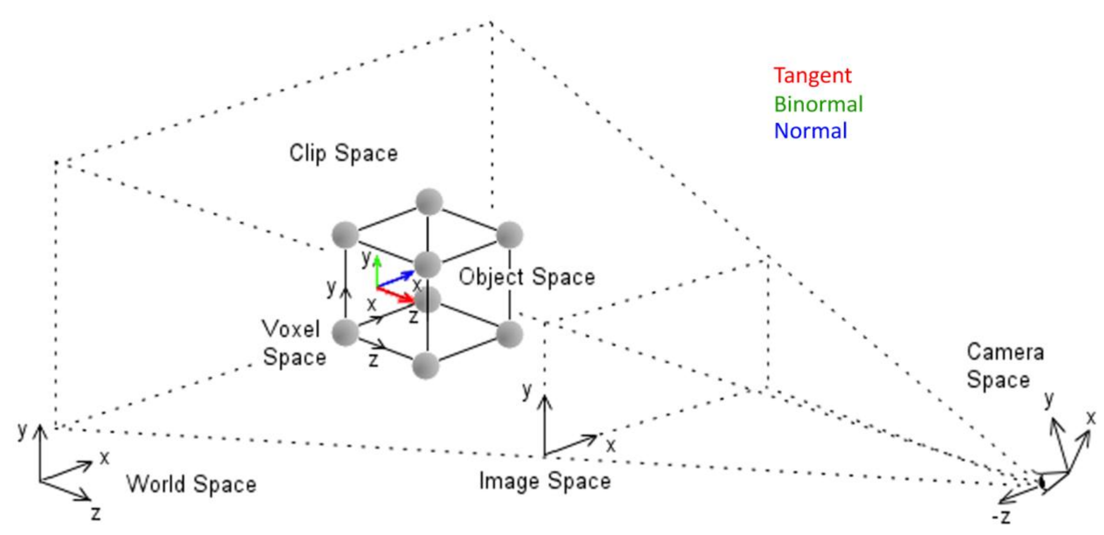


Bild 14 Übersicht der verschiedenen Koordinatensysteme

Zur Korrekten Modifikation einer Normale muss eine Matrixtransformation des jeweiligen Punkts in ein anderes Koordinatensystem vorgenommen werden. Da sich die Matrix für jeden Punkt unterscheidet kostet dieser Vorgang in der Vorverarbeitung eines zu rendernden Bildes viel Zeit. Aufgrund der komplizierten Berechnung der Tangente in einem Punkt an beliebig geformten Objekten wird die Berechnung bislang häufig von der CPU übernommen. Die Bi-Normale ist die der transponierte Normalvektor. Die Transformationsmatrix in das Objektkoordinatensystem setzt sich folgendermaßen zusammen:

T – Tangente

B – Bi-Normale

N - Normale

$$M = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

Formel 2: Zusammensetzung Tangent Space-Matrix

Folgend wird Punkt des Objekts mit der dazugehörigen Matrix multipliziert um in das neue Koordinatensystem überführt zu werden. Daraufhin wird der entsprechende Differenzwert der Höhe aus der dazugehörigen Textur ausgelesen und mit dem Z-Wert des jeweiligen Punktes addiert. Dadurch entsteht bei der Auswertung des Lichtmodells der Eindruck eines höher liegenden Punktes.

Bild 15 Auswirkung von Bump Mapping

Bump Mapping kann in folgenden, visualisierten Gebieten genutzt werden:

- Erzeugung von Oberflächen aus Kartenmaterial
- Visualisierung von Mauerwerk
- Erhöhung des Realitätsgrades eine Objektes durch hinzufügen von Unebenheiten

Bild 16 Bump Map eines Mauerwerks [imageshak.us]

Bild 17 Erhöhung der Platizität von Objekten durch "Kratzer" (Bump Mapping) [NVIDIA]

2.3.3. Shader-Programmierung

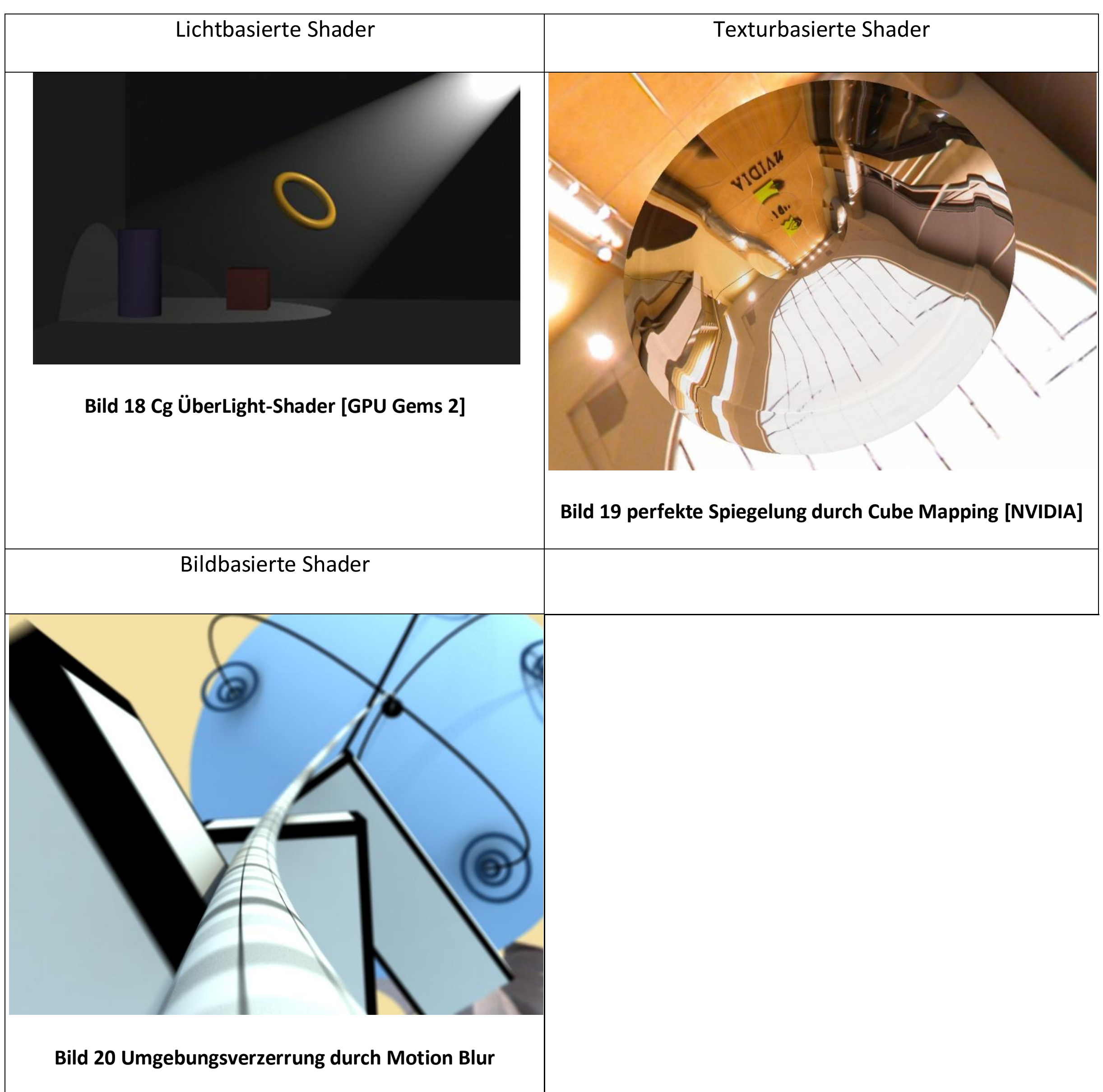
In vorigen Abschnitten wurden häufig auf die hardwaretechnische Funktion von Shadern eingegangen. In diesem Absatz wird die Entwicklung von Shadern und deren Funktion für die Visualisierung genauer untersucht.

Viele Shader-basierte Effekte modifizieren die Lichtberechnung für die grafische Ausgabe. Beispiele dieser Gruppe von Effekte sind Per-Pixel-Lighting, Texture Lighting und ÜberLight. Eine weitere Gruppe ist die das texturbasierte Rendern, bei dem die Ausgabe des Fragment Shaders in eine Textur gespeichert und in einem weiteren Renderschritt ausgewertet wird. Dabei können Farb- wie auch Tiefeninformationen der Objekte in Texturen abgelegt. Vertreter dieser Gruppe sind viele Arten der Schattengenerierung (z.Bsp. Shadow Mapping), Cube Mapping zur Darstellung vollkommen spiegelnder Objekte und Environment Mapping. Eine letzte zu nennende Gruppe von Effekten sind bildbasierte Effekte, bei denen das Ausgabebild samt Tiefeninformationen abgespeichert und mehrfach gerendert wird. In diese Gruppe fallen Motion Blur, Anti-Aliasing und Depth-of-Field Effekte. Dabei orientiert sich die Einteilung der Gruppen nach der Art des Speichers, in dem die Informationen abgelegt werden (Daten-, Textur- und Pixelspeicher).

Effekte werden durch Modifikation bestehender Berechnungen erzielt. Es ist möglich, verschiedene Shader und Funktionen im Grafikspeicher vorrätig zu haben, jedoch kann nur eine Hauptfunktion pro Shaderprogramm existieren.

In den Shadern werden hauptsächlich die, durch die Grafikschnittstelle zur Verfügung gestellten Variablen für Effekte benutzt. Es ist zur Erstellung neuer Effekte ebenfalls möglich selbstdefinierte Variablen festzulegen. Diese Variablen werden in drei Kategorien eingeteilt: globale Variablen (uniform variables), welche für ein Objekt nicht verändert werden, lokale Variablen (attribute variables), die wie Normale und Farbe für jeden Vertex festgelegt sein müssen, und Brückenvariablen (varying variables). Brückenvariablen haben über alle drei Shaderstufen den gleichen Namen. Sie werden im Vertex-Shader erstellt und je Vertex mit einem Wert belegt. Dieser wird von der Rasterisierungseinheit des Grafikprozessors beim Übergang in einen anderen Shader (zum Beispiel von Vertex- in Fragment Shader) zwischen zwei aufeinanderfolgenden Vertices interpoliert. Ein grafisches Beispiel der Interpolation ist ein Farbübergang von einem roten zu einem grünen Vertex.

Es folgen abschließend noch einige Grafiken mit Beispielen der angesprochenen Effekte, nach Effektgruppe geordnet. Im Anhang sind Skizzen zum Aufbau der einzelnen Shader zu finden.



Lichtbasierte Shader	Texturbasierte Shader
Bild 18 Cg ÜberLight-Shader [GPU Gems 2]	**Bild 19 perfekte Spiegelung durch Cube Mapping [NVIDIA]**
Bildbasierte Shader	
Bild 20 Umgebungsverzerrung durch Motion Blur	

2.3.4. Compute Shader, GPGPU und Physikberechnungen

Möglichst realistisch wirkende Darstellung von bewegenden Objekten in grafischen Simulationen wird durch physikalische Berechnungen erreicht. Dabei ist der Abstraktions-grad zwischen Realität und Simulation ausschlaggebend für Bildwiederholfrequenz der Simulation.

Physikalische Berechnungen gehören zur Vorverarbeitung eines Frames. Dabei werden die Parameter der Objekttransformationsmatrizen durch Umrechnungen physikalischer Größen

(Druck, Kraft etc.) generiert. Die physikalischen Größen variieren je nach physikalischem Phänomen, welches ein Objekt beeinflusst.

Jene physikalischen Berechnungen werden hauptsächlich auf der CPU berechnet. Dies kann Anwendungsspezifisch oder mit Hilfe einer Physik-API erreicht werden. Bekannte Physikengines sind dabei PhysX und HavocFX, deren neueste Versionen die Berechnungen auf dem Grafikkern ausführen. Die Berechnungen werden parallel zur Renderpipeline ausgeführt. Dies führt zu einem Geschwindigkeitszuwachs in der Physikberechnung. Die dabei fehlende Synchronisation von Bild und Werteberechnung führt jedoch zu Datenmanagement-Overhead. Desweiteren wird durch die Physikengine die Hälfte der zur Verfügung stehenden Grafikressourcen genutzt, demnach die Hälfte der Shader in einer Single-GPU-Konfiguration bzw. eine gesamte GPU in einer Multi-GPU-Konfiguration [NVI09] . Daher ist schlussendlich mit Leistungseinbußen bei der Bildwiederholfrequenz zu rechnen.

Ein Ansatz zur Veränderung dieser Verarbeitung ist die Rückkehr zur synchronen, sequentiellen Abarbeitung der Physikberechnung, welche jedoch ebenfalls auf der GPU ausgeführt. Dies wird durch die Technologie der General Purpose-GPU ermöglicht. Dabei kann die GPU zur allgemeinen Berechnungen auch vor der eigentlichen Renderpipeline genutzt werden. Hierbei werden, wie in weiteren Kapiteln erläutert, durch die GPGPU-APIs Ansätze des High Performance Computing für die Architektur verwendet.

Zur Visualisierung der unterschiedlichen Ansätze dienen folgende Grafiken.

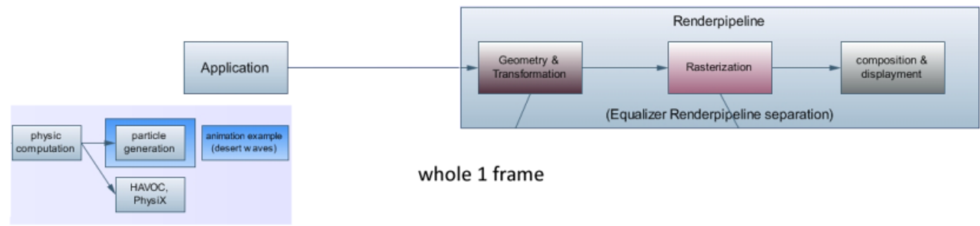


Bild 21 Physikberechnung auf der CPU

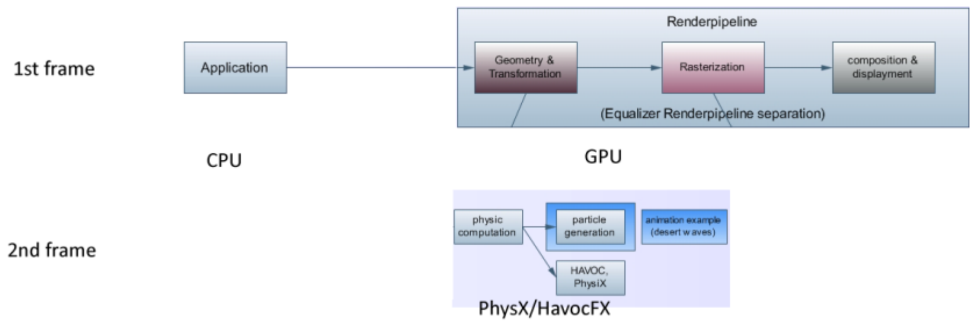


Bild 22 HavocFX und PhysX

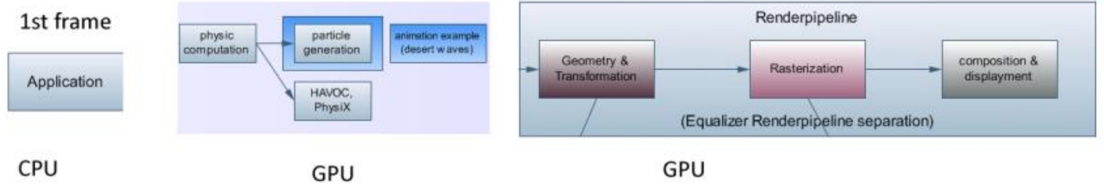


Bild 23 Compute Shader und GPGPU Technologie

2.3.5. Partikeleffekte

Partikeleffekte dienen zur Visualisierung einer großen Menge – relativ zur Anzahl der Hauptobjekte der Szene - kleiner geometrischer, homogener Objekte. Sie werden in Interaktion mit der Umgebung durch physikalische Kräfte bewegt oder deformiert.

Beispiele für Partikeleffekte sind Simulation von Wasserströmungen, Staub- und Feuereffekte sowie Simulationen von astronomischen Objekten.

Partikelsysteme bestehen aus verschiedenen Objekten und Aktoren. Dazu gehört ein Basispartikel des Systems, welches in vielfacher Ausführung generiert wird. Parameter, die ein Partikel charakterisieren sind:

- Radius
- Farbe bzw. Textur
- Form
- Anfangsgeschwindigkeit

Desweiteren werden ein oder mehrere Emitter des Partikels benötigt. Emitter kontrollieren die Generierung neuer Partikel über der Laufzeit der Anwendung.

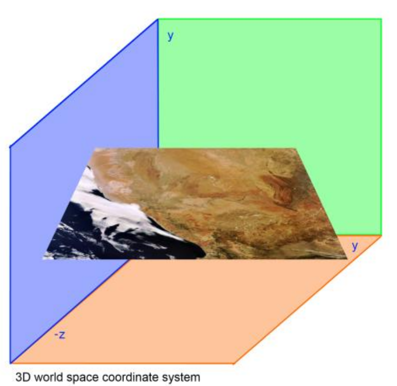
Zur Bestimmung der, auf die Partikel der Szene wirkenden Kraft wird häufig eine Physikengine implementiert. Dessen Funktionsweise wurde im vorigen Abschnitt erläutert.

2.3.6. Terrainkonstruktion

Der Aufbau eines Terrains ist essentieller Bestandteil von professionellen Animationssystemen sowie bei der wissenschaftlichen Simulation von Umgebungen, die von Menschen nicht zugänglich sind. Darunter zählen beispielsweise polare Eiswüsten, Vulkane und Unterwasserregionen sowie Oberflächen astronomischer Objekte wie Monde, Planeten und Sterne. Es ist daher eine sehr wichtige Technik um schwer vorstellbare Lebensbedingungen der Allgemeinheit verständlich zu präsentieren.

Analog zu anderen Objekten besteht dabei Terrain in der grafischen Datenverarbeitung aus dreidimensionalen Punktkoordinaten. Terrain wird mit polygonalen Netzen modelliert. Dabei wird eine ebene Fläche eines Grundmuster vorausgesetzt, dessen Oberflächenpunkte durch abgespeicherte Differenzen angeglichen werden.

Jene Differenzen werden in Höhenkarten abgespeichert. Dabei handelt es sich um ein, auf das allgemeine Farbmuster des Basisbildes angepasste, Graustufenbild einer fotografisch aufgenommenen Umgebung. Dies sind häufig Satellitenbilder und unterliegen dementsprechend den Eigenschaften der orthogonalen Projektion. Bei der Verarbeitung wird eine Koordinatentransformation durchgeführt, da sich das zweidimensionale Koordinatensystem der Karte in der Regel vom dreidimensionalen Koordinatensystem der Szene unterscheidet.



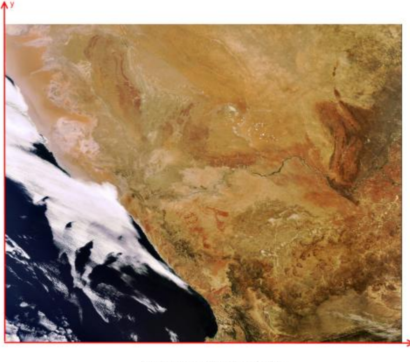


Bild 24 lokales 3D-Koordinatensystem und 2D-Bildkoordinatensystem

Aus den Informationen des Graustufenbildes werden anschließend die Grauwertinformationen als Höhendifferenzen interpretiert und als z-Wert im Kartenkoordinatensystem abgespeichert. Die Auflösung des Bildes muss dabei mit der Größe des Basisrechtecks übereinstimmen. Daher werden durch hochaufgelöste Grauwertbilder entsprechend viele Vertices kreiert, was zu Lasten der Performance geht. Daher ist es üblich das Graustufenbild mit einer vergleichsweise geringen Auflösung zu versehen und den

Detailgrad durch eine höher aufgelöste Farbtextur zu erzwingen. Die zu Grunde liegende Technik ist das Supersampling (Überabtastung).

Große Kartenteile mindern die Bildwiederholfrequenz erheblich. Deshalb ist für geografische Anwendungen ein Datenmanagementsystem zum Nachladen von Karteninformationen zu empfehlen. Professionelle Beispielapplikationen in diesem Bereich sind Google Maps sowie Bing Maps.

2.3.7. Objektmodellierung und Dateiformate

Eine Basis dreidimensionaler Szenen sind darin enthaltene Objekte. Jene Objekte bestehen aus Eckpunkten, sogenannten Vertices. Jedoch existieren zwei verschiedene Ansätze, aus den Eckpunktdaten dreidimensionale Objekte zu modellieren. Diese stehen dabei jedoch nicht in Konkurrenz, da jeder Ansatz für bestimmte Anwendungsgebiete geeignet ist.

Es wird in der Computergrafik zwischen Punktewolken und Dreiecksnetzen unterschieden.

Punktewolken sind solide Objekte die aus einer ungeordneten Menge von Punkten bestehen welche überwiegend in einer gewissen Region liegen. Die Punkte besitzen keine signifikante Relation zu einander. Dabei besteht jeder Punkt eines Objekts einer Punktewolke aus Vertices und der dazugehörigen Farbe. Durch Punktewolken können gefüllte Objekte dargestellt werden. Daher wird diese Art der Objektmodellierung häufig in der medizinischen Computergrafik eingesetzt, bei dem solide Objekte wie Knochen, Muskeln und anderes Gewebe dargestellt werden.

Ein weiteres Einsatzgebiet dieser Modellierungsart ist die Speicherung von Objekten für Rapid Prototyping and Manufacturing Prozesse, bei dem reale Objekte aus dreidimensionalen Modellen generiert werden. Derartige Prozesse sind Stereolithografie, das Spritzgussverfahren und 3D-Drucktechniken. Die Erhebung dieser Daten geschieht durch CAD-Modelle, Stereokamera-Aufnahmetechniken sowie Laserscans.

Nachteil für die Nutzung in weiteren Bereichen der grafischen Datenverarbeitung ist die Abwesenheit von Flächennormalen und die umständliche, nachträgliche automatische Generierung. Flächennormalen von Objekten werden bei dreidimensionalen Darstellungen zur Auswertung von Lichtmodellen und Anwendung gewisser Shader benötigt. Durch Konvertierung eines Modells in ein Objekt aus polygonalen Netzen können Flächennormalen generiert werden. Dabei gehen jedoch die Füllpunkte verloren.

Beispieldateiformate für Punktwolken-basierte Objekte sind „.stl“ (Stereo Lithography), und „.vrml“ (Virtual Reality Modeling Language).

Objekte aus Dreiecksnetzen bestehen, im Gegensatz zu Punktewolken, aus eine Menge geordneter Punkte. Dabei besteht vorerst jeder Vertex aus den dreidimensionalen Koordinaten, optional einer Farbe und eines Index in einer Zählreihenfolge. Dabei ist bei der

Generierung der nach Indexzahl geordneten Liste an Vertices die Aufbaurichtung entscheidend. Dabei kann die Indexierung der Punkte im- oder gegen den Uhrzeigersinn vorgenommen werden. Durch den Anordnung der Punkte entsteht ein, idealerweise geschlossenes, Dreiecksnetz. Dies kann ebenfalls als Oberflächennetz oder auch polygonales Netz bezeichnet werden. Durch die Entstehung von Oberflächen ist die Generierung einer Flächennormale möglich. Hierbei ist besonders auf die Indexierungsrichtung zu achten, da sonst die Normale falsch gerichtet sein kann und dadurch Vorderseite und Hinterseite der Fläche vertauscht werden. Die zugeordnete Normale eines Punktes entsteht aus der Mittelwertbildung aller Normalen der ihn angrenzenden Flächen. Die Bildung der Oberfläche erfolgt durch Triangulation. Weitere Informationen zum Aufbau von Dreiecksnetzen können aus jedem Grundlagenbuch der grafischen Datenverarbeitung bezogen werden [Her081] .

Durch die vorhandenen Normalen bieten sich Oberflächenobjekte für Darstellungen an, bei denen der Fokus auf besonderer Realitätsnähe liegt. Dies wird durch Auswertung verschiedene Lichtmodelle und Effekte erreicht, wobei die Normalen der Punkte benötigt werden. Typische Einsatzgebiete sind daher Renderings von CAD-Modellen zu Akquisitionszwecken, die professionelle Animationstechnik sowie die Visualisierung komplexer wissenschaftlicher Prozesse zur Förderung des Allgemeinverständnisses.

Eine Voraussetzung zur Bildung polygonaler Netze ist die Eliminierung von Punkten ohne Oberflächenzugehörigkeit. Deshalb können durch polygonale Netze keine gefüllten Objekte dargestellt werden.

Polygonale Netze werden bei der Darstellung auf eine Ansammlung an Elementarflächen – Dreiecken – reduziert. Vorteil von Dreiecken zur Darstellung sind Eindeutigkeit der Fläche (ein Dreieck kann nicht in sich geknickt sein) und der Fakt, dass bei der Teilung von Dreiecken wiederum Dreiecke mit halber Fläche entstehen. Dieser Teilungsalgorithmus ist in die Grafikkarte implementiert und heißt „Tesselation“.

Bekannte Datenformate für polygonale Objekte sind:

- .dfx (Digital Effects)
- .3ds (3Ds Max proprietäres Dateiformat)
- .IGES (Initial Graphics Exchange Specification)
- .kml (Keywhole Markup Language)
- .ive (OpenSceneGraph Object)
- .lwo (LightWave Object)
- .dae (Digital Assests Exchange – COLLADA)
- .ply (Polygonial File Format)

3. Lösungsvarianten zur Entwicklung der Rendersoftware

Im vorigen Kapitel wurden Grundsätze von grafischen Darstellungsverfahren beleuchtet. Es wurde ein Einblick in die heutige Grafiktechnik gegeben, welche die Basis der Arbeit bildet. Es wurde erörtert, warum es nötig ist, eine eigene Applikation zu entwickeln um die Untersuchung erfolgreich abzuschließen. Schlussendlich wurden die Demonstrationsapplikationen und deren Technologien vorgestellt, die als Maßstab für die Messungen anzusehen sind.

Im aktuellen Kapitel werden programmiertechnische Lösungsansätze vorgestellt, welche in den Themenbereich der Arbeit einzuordnen sind. Dabei handelt es sich um mögliche Zusammensetzungen von Frameworks, Application Programming Interfaces (APIs) und Grafikbibliotheken. Daher werden die zu Verfügung stehenden Techniken im Zusammenhang vorgestellt.

Einen Überblick der anvisierten Zielgruppen ist als Fallstudie im Anhang zu finden.

Das Kapitel beginnt mit der Vorstellung einer hardwarenahen Basislösung mit geringem Abstraktionsniveau. Darauf folgend werden einige GPGPU-APIs vorgestellt, mit denen die Physikengine realisiert werden kann. Darauf folgend wird das Abstraktionsniveau erhöht und mögliche Großschnittstellen und Frameworks zu Realisierung vorgestellt. Dabei werden sowohl kommerzielle als auch frei verfügbare Lösungen in Betracht gezogen. Begonnen wird demnach mit dem kommerziellen Rapidmind Framework. Desweiteren werden zwei Darstellungsframework, OpenSceneGraph und die Visualization Library präsentiert.

Das Kapitel endet mit einem Überblick an die Herangehensweise der Messwertermittlung. Dabei werden die Entwicklungswerkzeuge, IDEs (Integrated Developer Environments) sowie die Test-Referenzsysteme präsentiert.

3.1. Basislösung

Erster Ansatz bei der Entwicklung ist die Untersuchung einer Basislösung. Dabei wird komplett auf schon vorhandenes Wissen des Studiums zurückgegriffen. Es wird eine Kombination aus relativ simplen Teillösungen in Betracht gezogen und nur auf Basisbibliotheken zurück gegriffen.

Dieser Ansatz bietet sowohl Vor- als auch Nachteile. Durch den Aufbau auf schon vorhandenes Wissen, dass durch geschulte Lehrkräfte erlernt wurde, wird die Komplexität der Applikation auf einem geringen Maß gehalten. Die Software bleibt jederzeit überschaubar und beherrschbar. Desweiteren ist eine Hilfe der jeweiligen Professoren bei Komplikationen während der Entwicklung teilweise möglich. Ein weiterer, entscheidender Vorteil der Basislösung wird die Verarbeitungsgeschwindigkeit und der niedrige Speicherplatzbedarf sein. Durch gezielte Auswahl kleiner Erweiterungen werden nur die nötigsten Komponenten geladen, die zur Lösung der Aufgabe erforderlich sind. Außerdem verbrauchen grundlegende Bibliotheken wenig Haupt- und Festplattenspeicher.

Nachteil einer Basislösung ist die einfach prognostizierbare, geringe Skalierbarkeit der Applikation. Mit steigenden Anforderungen und Datenaufkommen gelangen Basislösungen erfahrungsgemäß an ihre Grenzen. Das „Programmieren im Großen“ ist nicht gegeben. Der wirtschaftliche Nutzen hält sich genauso in Grenzen wie wissenschaftliche Innovations- und Nutzbarkeitsgrad. Es wird sich daher im eine Speziallösung für das Problem handeln, auf welche nicht oder nur in geringem Maße aufgesetzt werden kann.

Die offene Basisbibliothek zur Generierung dreidimensionaler Szenen ist OpenGL. Deren Konkurrenzprodukt, Microsoft DirectX, kommt aus verschiedenen, im Absatz erläuterten Mitteln nicht in Betracht. Zur Realisierung der Effekte wird das auf OpenGL-aufbauende, jedoch in der Vorlesung bislang nicht gelehrte GLSL in Betracht gezogen. Sofern dies nach der Analyse nicht ausreicht könnte Alternativ Cg eingesetzt werden. Für die Berechnung komplexer mathematischer Problem wie z.Bsp. mehrfach verkettete Matrixmultiplikationen, die zu erwarten sind, wird die mathematische Bibliothek MathLib eingebunden. Da Microsoft nach dem Jahr 2003 die weitergehende native Unterstützung für OpenGL eingestellt hat muss eine Erweiterungsbibliothek genutzt werden. Dazu stehen GLee, Mesa3D sowie GLew zur Auswahl. Die nach kurzer Analyse aussichtsreichste Bibliothek GLEW wird daher näher vorgestellt.

3.1.1. OpenGL

OpenGL ist eine OpenSource-Grafikschnittstelle zur Entwicklung grafischer Datenverarbeitungssysteme in System- sowie Anwendungssoftware. Die Schnittstelle wird von der Khronos Group entwickelt, welcher Firmen aus den Bereichen Betriebssysteme, CAD, medizinische Datenverarbeitung sowie ein Großteil der auf dem Markt befindlichen Grafikkartenhersteller angehört. OpenGL steht momentan in der Version 3.2 zu Verfügung und ist damit mit der Microsoft Schnittstelle DirectX in der Version 10.1 vergleichbar. Dabei werden Basisfunktionen zur Erstellung zwei- und dreidimensionaler Systeme, die Nutzung von Grafikspeicherobjekte sowie eine Anbindung zur OpenGL SL angeboten.

OpenGL wird von Linux' KDE und Qt sowie von der MacOS X Oberfläche Cocoa zur Inszenierung dreidimensionaler grafischer Oberflächen genutzt. Daher wird durch in den genannten Betriebssystemen die aktuelle OpenGL-Version nativ zur Verfügung gestellt. Microsoft war bis 2003 Mitglied im OpenGL Architecture Review Board, welches bis dato die Grafikschnittstelle weiterentwickelte. Seit dem Austritt werden neuere OpenGL-Versionen nicht mehr nativ von Microsoft Windows-basierten Betriebssystemen unterstützt. Die seither nativ unterstützte Version stagniert bei 1.3. Neue Versionen sind nur durch Installation von Erweiterungsbibliotheken nutzbar. Mit Windows 7 wird kein nativer OpenGL-Kernel mehr angeboten. Die OpenGL-Version 1.3 kann durch optionale Installation des ungefähr 1,5 GB großen Windows 7 Software Development Kit hinzugefügt werden.

OpenGL arbeitet als Statusmaschine. Jede Umgebungsvariable und Funktion ist mit einem Standardwert belegt. Veränderungen werden durch Funktionsaufrufe mit entsprechenden Variablen erwirkt, welche den aktuellen Zustand verändern. Dies ist ein gegensätzliches Konzept zu DirectX, bei dem ein bestimmter Effekt durch die ausschließliche Belegung aller damit verbundenen Zustände erfolgt. DirectX wirkt häufig kompliziert und unsicher, da es bei teilweiser Initialisierung eines Effekts zu einem unbestimmten Zustand führt.

Gründe für die Wahl von OpenGL zur Realisierung der Applikation sind der offene Code, durch den eine große Community zur Unterstützung bereitsteht. Ein weiteres Argument ist die mögliche Portierbarkeit auf andere Betriebssysteme. Bei Vermeidung proprietärer C++ Funktionen in der Applikation ist eine einwandfreie Portierung möglich. Dies ist für die Weiterentwicklung wichtig, da wissenschaftliche Simulationen häufig auf Linux-basierten Betriebssystemen entworfen und ausgeführt werden. Desweiteren ist die gute Unterstützung Grafikkarten-spezifischer Funktionen durch native Funktionserweiterungen gegeben. Diese werden in gleichem Maße von ATI wie auch NVIDIA freigegeben. Da die zu untersuchende Technologie sehr neu ist werden die OpenGL-Extensions Grundvoraussetzung bei der Entwicklung sein. Desweiteren ist ein Grundwissen von OpenGL nötig, da die meisten höheren Grafikframeworks und APIs auf OpenGL aufbauen und dadurch erweitert werden können.

Die OpenGL Architektur ist an den Ablauf der allgemeinen Grafikpipeline angelehnt. Die Architektur ist zum Vergleich mit der Grafik aus Kapitel 2 in folgendem Bild dargestellt.

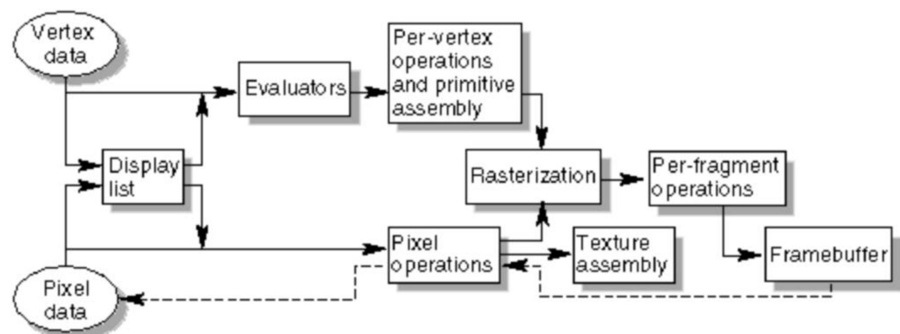


Bild 25 Architektur OpenGL Renderpipeline [OpenGL Red Book]

3.1.2. OpenGL SL

OpenGL SL ist die nativ von OpenGL unterstützte Shader-Programmiersprache. Sie wird von OpenGL 3.2 in der aktuellen Version 1.5 genutzt. Das unterstützte Shader Model ist 4.1. Darin enthalten sind alle spezifischen Funktionen zur Nutzung von Vertex-, Geometry- und Fragment Shader. OpenGL SL war in der Vergangenheit eine häufig genutzte Schnittstelle für Hochleistungsgrafikverfahren wie Radiosity und Raytracing. So wurde 2006 der erste komplett auf der Grafikkarte implementierte Raytracer basierend auf OpenGL SL 2.1 entwickelt. Dieser gebraucht den Grafikspeicher zur Ablage von Zwischenwertender Strahlenverfolgung. Durch die komplizierte Implementierung dieses Verfahrens besteht jedoch auch weiterhin Forschungsbedarf in diesem Bereich.

Alle benötigten Funktionen zur Realisierung der gewünschten Effekte werden durch GLSL zur Verfügung gestellt.

Es gibt jedoch einige Nachteile, die die Entwicklung speziell in GLSL (zusätzlich zu den allgemeinen Komplikationen von Shaderentwicklung aus Kapitel 1.3.3) erschweren. In OpenGL wird das Standardrendern durch eine fest Verarbeitungspipeline durchgeführt, in der eine Vielzahl von Berechnungen zu Darstellung der Standardeffekte abgespeichert ist. Bei Einsatz von GLSL wird diese feste Verarbeitungspipeline komplett durch verfügbare Shader ersetzt. Es müssen demzufolge nicht nur Shader für die Spezialeffekte geschrieben sondern gleichzeitig ebenfalls die gewünschten Standardfunktionen manuell im Shadercode implementiert werden. Durch die große Anzahl an Standardeffekten ist dies ein nicht zu unterschätzendes Kriterium bei Aufwandsabschätzung, Planung und Entwicklung.

3.1.3. MathLib

MathLib ist eine C++ Bibliothek mit Strukturen, Klassen und Funktionen zur Verarbeitung großer mathematischer Datensätze. Zu den zur Verfügung gestellten Klassen gehören:

- Vektoren
- 3x3 Matrizen
- 4x4 Matrizen
- Komplexe Zahlen
- Quaternionen

Die Bibliothek ist durch die simple Handhabung erste Wahl der mathematischen Bibliotheken für die Entwicklung grafischer Datenverarbeitungssysteme. Dabei wird häufig die Verkettung der Multiplikation von 4x4 Matrizen verwendet, da alle grafischen Transformationen auf diesen basieren.

Zur Verwendung müssen Umwandlungsroutinen vom MathLib-Format für Matrizen in das OpenGL-Format vorgenommen werden.

3.1.4. GLEW

GLEW steht für „Graphics Library Extension Wrangler“ und ist eine Erweiterungsbibliothek für OpenGL. Sie steht für die Betriebssysteme Windows, Linux und MacOS X zur Verfügung und erweitert den nutzbaren OpenGL Befehlssatz um OpenGL Funktionen neuerer Versionen. Da OpenGL in Linux und MacOS ständig in der neuesten Version im Kernel vorliegt ist die Nutzung in diesen Betriebssystemen nicht nötig. Hauptaugenmerk der GLEW liegt in der Erweiterung des Befehlssatzes für Windows-Betriebssysteme.

GLEW liegt in der aktuellen Version 1.5.2 vor. Zu Beginn der Entwicklung des Rendersystems war Version 1.5.1 verfügbar.

Der Gebrauch der GLEW ist notwendig um typische Erweiterungen wie Speicherobjekte, Multisampling und Ähnliche nutzen zu können.

3.2. OpenCL

OpenCL (Computation Library) ist eine OpenSource Bibliothek zur Berechnung allgemeiner Operationen auf der Grafikkarte. Die Bibliothek gilt dabei als Zusatzbibliothek für OpenGL für Shader Model 5.0 zur Verwendung der Compute Shader-Technologie. Es wird ebenfalls von der Khronos Group herausgegeben und entwickelt. OpenCL steht dabei in den Versionen 0.9 und 1.0 zu Verfügung. Während die Version 0.9 nur CPU-Kernels verarbeiten kann ist mit Version 1.0 die Berechnung auf CPU wie auch GPU möglich

Analog zu OpenGL wird zur Verwendung der Bibliothek ein Kontext (Berechnungsumgebung) angelegt. Dabei wird überprüft, welche Plattform mit jeweiliger Konfiguration zur Verfügung steht. Daraufhin werden Daten in eine, der Architektur angepassten, Form vorbereitet und ein Compute Shader-Code in Form einer „.cl"-Datei geladen. Aufgrund der Möglichkeit, OpenCL auf CPU wie auch auf GPU zu berechnen, wird der Code kompiliert und unterschiedliche Objektdateien generiert. In einem Windowssystem ist dies eine Objektdatei für CPU-Code und ein herstellerabhängiges Format für GPU Code. Dies sind „.il"-Dateien (Intermediate Language) pro Kernel für ATI Grafikkarten und „.ptx"-Dateien für NVIDIA Grafikkarten. Aus den Objektdaten wird ein Kernel generiert. Daraufhin werden die Daten mit dem jeweiligen Kernel ausgeführt und deren Ergebnisse abgespeichert.

Aufgrund der bereits analysierten Shaderarchitektur wird durch die Abarbeitung großer, parallelisierbarer Prozesse auf der Grafikkarte ein starker Leistungszuwachs erzielt.

Die Verarbeitung des Kernels ist herstellerabhängig. NIVIDA Karten basieren architektonisch auf einem kaskadierten Grid an Compute Shader Einheiten. Dies erleichtert die Konvertierung bisheriger, C-basierter Grid Computing-Programme. ATI Karten verwenden hingegen die Stream Technologie. Dabei werden die Eingangsdatensätze als ein zusammengesetzter Datenstrom betrachtet, der durch eine entsprechende Anzahl von parallelen Recheneinheiten ohne spezielles Layout verarbeitet werden. Dabei handelt es sich um einen Ansatz aus dem High Performance Computing-Bereich.

Eine Architekturskizze der Bibliothek befindet sich im Anhang. Desweiteren sind zur Programmierung die aktuelle OpenCL Spezifikation von Khronos, Einstiegs- sowie Expertenbeispiele von den offiziellen Developer-Webseiten der Grafikhersteller sowie die bei der Installation der Bibliothek mitgelieferten Herstellerspezifikationen von nutzen. Im Expertenabschnitt der Webseiten befinden sich ebenfalls Richtlinien zur Codeoptimierung.

Der größte Nachteil bei der Entwicklung von OpenCL-basierten Applikationen ist der fehlende Compiler bzw. eine Compiling Rule für die Standard-IDE Visual Studio. Häufige Typkonvertierungen sowie herstellerabhängige Kernelfunktionen erschweren die Entwicklung von größeren Compute Shader Applikationen.

3.3. CUDA

CUDA (Compute Unified Device Architecture) ist das proprietäre Pendant zu OpenCL. Dieser proprietäre Vorgänger wurde im Jahr 2007 von NVIDIA entwickelt und dient zur generellen Berechnung von Algorithmen auf der Grafikkarte.

Die Realisierung einer Physikengine für die ausgewählten Demos durch CUDA wäre eine proprietäre Lösung. Das Programm könnte bei exklusiver CUDA Programmierung nur auf Rechnern mit einer NVIDIA-Karte ausgeführt werden. Deshalb wäre Kombinationslösung eine gute Variante.

Es gibt einige Vorteile von CUDA gegenüber OpenCL. Aufgrund des Alters der Technologie besteht eine umfangreiche Entwickler-Community und etwaige Implementationsprobleme zu lösen. Desweiteren sind umfangreiche Beispielapplikationen zur Weiterentwicklung verfügbar. Ein großer Vorteil ist die Übersichtlichkeit der Funktionen der Schnittstelle. Dabei ist vor allem die Erstellung einer CUDA Umgebung im Vergleich zu OpenCL sehr viel simpler, da durch den Fokus auf NVIDIA Plattformen hierbei eine komplizierte Teilung nach verschiedenen Herstellerarchitekturen umgangen wird. Desweitern ist es möglich, dass durch diesen Architekturfokus der Leistungszuwachs auf NIVIDA Karten durch CUDA sehr viel größer ist als bei der Anwendung von OpenCL, obwohl dies nicht zweifelsfrei bewiesen ist.

Durch die einheitliche Architektur ist der Umgang mit dem Grafikkarten-spezifischen Code einfacher, da NVIDIA Chips einen relativ einheitlichen Umgang mit Fließkommazahlen- und Operationen aufweisen.

Ein Nachteil, der jedoch auch bei OpenCL genannt wurde, ist die auch für CUDA fehlende IDE. Es wird von NVIDIA offiziell eine Compile Rule für CUDA vertrieben, welche Fehler der Warnstufe 1 ebenfalls anzeigt. Jedoch ist zur fehlerfreien Programmierung dieser neuen Technologie und zum Debuggen eine IDE mit verfügbarem Trace-Modus nötig. Diese könnte im zweiten oder dritten Quartal 2010 durch das Visual Studio Framework NEXUS veröffentlicht werden. Bis dahin ist die Programmierung komplexerer Kernels ohne Basisbeispiel fehleranfällig.

Zum genaueren Verständnis der Schnittstelle und zum Vergleich mit OpenCL befindet sich im Anhang eine Architekturskizze der API. Weiterführende Literatur und notwendiges Lehrmaterial für CUDA wird von der Entwicklerseite von NVIDIA direkt angeboten.

3.4. Brook

Brook ist eine Stream Processor-Programmiersprache, welche von der Standford University Graphics Group entwickelt wurde. Dabei werden, analog zu OpenCL und CUDA, generelle Berechnungen auf dem Grafikprozessor ausgeführt. Compiler und Laufzeitbibliothek zur Realisierung von Brook-Applikationen werden von der Standford University als „BrookGPU“ angeboten.

Brook ist dabei eine ANSI C-basierte Programmiersprache. Brook-Kernels können als Shader-Programme durch verschiedene Grafikbibliotheken wie OpenGL 1.3 oder DirectX 9.0 genutzt werden. Ein großer Vorteil von Brook ist sowohl die Betriebssystemunabhängigkeit als auch Herstellerunabhängigkeit bezüglich der Grafikbeschleuniger. Somit können jene Kernels auf beliebigen Betriebssystemen ohne spezielle Grafikkonfiguration genutzt werden.

Durch fehlende Bindung an Grafikhersteller oder Entwicklungsgruppen gibt es jedoch ebenfalls Nachteile bei der Entwicklung mit Brook. Im Gegensatz zu CUDA oder OpenCL wird das nötige Multithreading-CPU-Backend durch keine spezielle Schnittstelle verwaltet. Es bedarf daher bei der Implementation von Brook mit Hilfe von OpenGL-Shadern einer selbstdefinierten Lösung des Backends. Desweiteren fehlt die, durch eine spezielle Schnittstelle bereitgestellte, Multi-GPU Unterstützung. Brook ist daher in der OpenSource-Basisversion nicht zur Untersuchung in Multi-GPU-Umgebungen geeignet.

AMD/ATI vertreibt mit Version 1.4 des FireStreams eine an ATI angepasste Version namens Brook+. Dabei wird das sowohl das Problem der Multi-GPU-Unterstützung als auch des Multithreading-Backend durch den ATI FireStream gelöst. Jedoch ist unklar, ob der FireStream mit den nötigen Erweiterungen auch auf ATI-Desktopkarten verfügbar ist. Bis Version 1.4 waren Stream SDK und FireStream mit gleichen Funktionen ausgestattet. Jedoch wurde mit Stream SDK 2.0 beta4 die Unterstützung von Brook+ für Desktopkarten aus dem Stream-Kern entfernt und durch OpenCL ersetzt.

3.5. CAL

CAL ist eine weitere, durch das ATI Stream SDK zur Verfügung gestellte GPGPU-Programmiersprache. Dabei handelt es sich um eine „Zwischensprache“ (Intermediate Language) von Hardware und Software. Als Intermediate Language wird allgemein eine Sprache abstrakter Rechner zur Analyse von Programmen beschrieben[Jes06] . Demnach ist CAL eine Assembler-basierte Programmiersprache, entwickelt für den speziellen Befehlssatz von ATI Shader.

Vorteil von CAL ist die größte Abarbeitungsgeschwindigkeit aller Shader-Sprachen. Die Entwicklung von CAL beruht auf frühen Ansätzen von Shader-Programmierung, bei der jene Mikrocontroller durch speziell für den jeweiligen Chip entwickelte Assembler-Programme gesteuert wurden.

Jene Basis ist demzufolge auch der große Nachteil von CAL. Assembler-Programme haben die generell längste Entwicklungszeit zur Lösung der gleichen Aufgabe im Vergleich zu Hochsprachen. Die Sicherstellung der Stabilität der Programme auf der jeweiligen Hardware ist dabei eines der Hauptziele, da fehlerhafte Assembler-Programme zu gravierenden Störungen des Systems führen können. Desweiteren ist etwaige Portabilität, sogar unter ATI-Karten, fraglich.

Aufgrund der Probleme von Aufwand und Komplexität in der Entwicklung mit CAL ist dies kein praktikabler Lösungsansatz zur Untersuchung der Aufteilung von Rechenlast in einer Multi-GPU Umgebung. Führ eine Ermittlung der größten Leistungsfähigkeit von Grafikkarten ist CAL in zukünftigen Messungen für einen Grafikchip eine langfristige Option.

3.6. Chromium

Chromium ist ein System zum interaktiven Rendern dreidimensionaler Szenen auf Clustern und Grafik-Workstations. Parallele Render-Algorithmen wie beispielsweise sort-first (SFR) oder sort-last (Depth Frame Rendering; DFR) sind im System enthalten[Hum02] . Desweiteren erlaubt Chromium eine interaktive Manipulation von OpenGL-Kommandos. Das System kann unter Microsoft Windows sowie Linux und IRIX genutzt werden.

Chromium ist zur Aufteilung der Renderpipeline in einer verteilten Umgebung entwickelt worden. Dabei wird die zu rendernde Datenmenge auf mehrere, über ein Netzwerk verbundenen, Rechner aufgeteilt. Ein Rechner des Netzwerks ist dabei der verteilende Grafik-Server, welcher gerenderte Frames mit SFR oder DFR komponiert. Ein großer Vorteil ist die Aufteilung der Gesamtdatenmenge durch das System, sodass nur Teile der Szene über ein Netzwerk übertragen werden müssen. Dadurch entsteht eine gut skalierbare Lösung für Grafikkomposition.

Chromium kann für diese Arbeit nicht verwendet werden, da der Ansatz auf ein Netzwerk von Grafik-Clustern beruht, wobei die Rechner des Clusters nur eine GPU besitzen. In dieser Arbeit wird jedoch nach einer anwendbaren Lösung zur Grafikkomposition in einer Umgebung gesucht, in der ein (optional mehrere) Rechner mehrere Grafikprozessoren besitzt, welche die Berechnung durchführen.

3.7. Rapidmind

Rapidmind ist ein C++ - Software Framework zur automatischen Aufteilung von Rechenlast auf vorhandene Komponenten wie Grafikkarte und Multicore CPUs. Durch Rapidmind werden die zu berechnenden Daten in ein dreidimensionales Grid gepackt. Basierend auf auszuführender Operation und zu verarbeitenden Datentyp werden die Daten an die dafür bestpassende Komponente verteilt.

Rapidmind ist ein Framework für die Programmiersprachen C und C++. Durch das Framework unterstützte Komponenten sind aktuelle x86-basierte Hauptprozessoren, Grafikkarten mit mindestens Shader Model 2.0 (Vertex- und Frament Shader) sowie der Cell Broadband Engine (CellBE). Die CellBE ist ein von IBM, Sony und Toshiba entwickelter Hochleistungsprozessor [Pao07] , der in der Spielekonsole "Playstation 3" verbaut ist. Zur Lastverteilung auf die Grafikkarte werden OpenGL-Shader verwendet. Die zu Verfügung stehenden Eingangsdaten werden dabei in 1 Byte-Gleitkomma-Listen gepackt und als Vertex, Normalen und Farblisten sowie weiteren Attributlisten an OpenGL übergeben. Die Anzahl der Eingangslisten hängt von der maximal unterstützen Anzahl an Shaderattributen der Grafikkarte ab. Die Ausgangsdaten werden variabel in 1-, 2- oder 3-dimensionale Texturen gepackt, die nach dem Berechnungsvorgang durch die API auslesbar sind.

Das Rapidmind Framework wird von der gleichnamigen Firma mit Sitz in Waterloo, Ontario (USA) entwickelt und vertrieben. Es handelt sich dabei um eine proprietäre Plattform. Die Firma ist eine Ausgründung der University of Waterloo.

August 2009 wurde Rapidmind von Intel aufgekauft. Die Software ist Teil der Ct Technology [Rei09] und wird unter dieser Technologie weiterentwickelt und vertrieben.

Folgend sind zwei Architekturskizzen des Frameworks dargestellt.

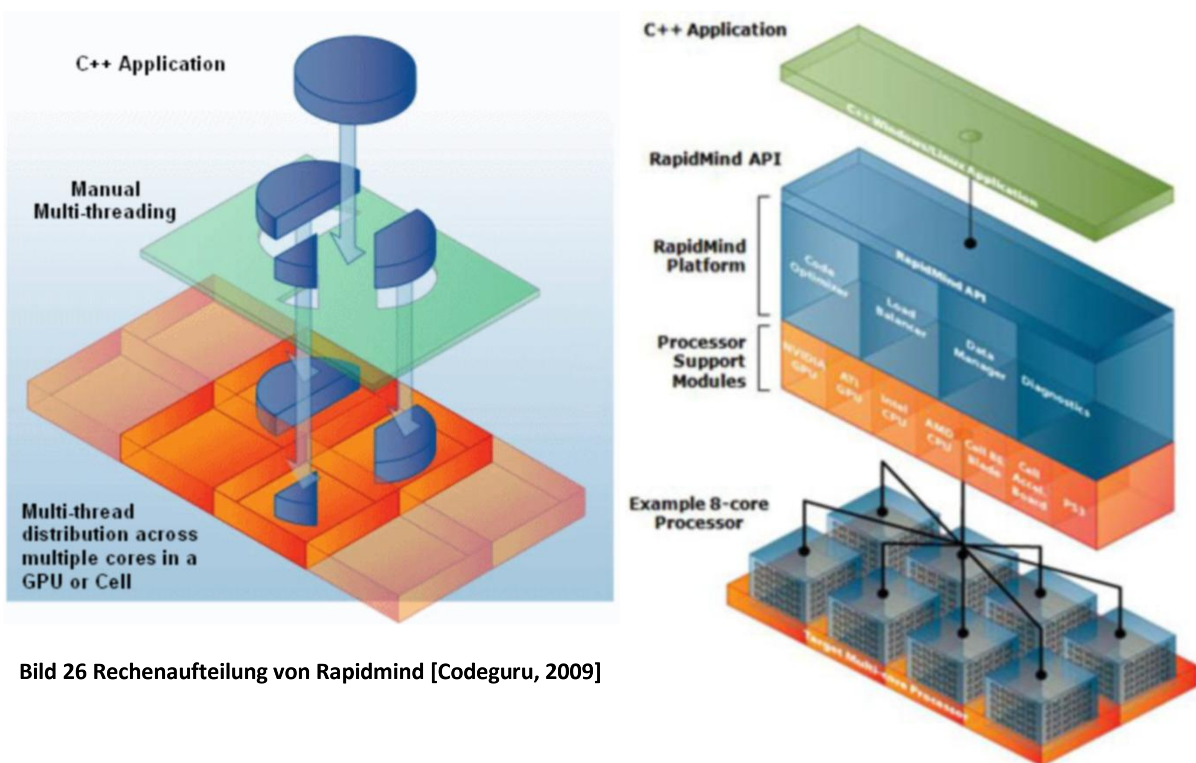


Bild 26 Rechenaufteilung von Rapidmind [Codeguru, 2009]

Bild 27 Rapidmind Architektur [www.redcanary.ca, 2009]

3.8. Open Scene Graph

Die Wahl der Basislösung zwingt in vielen Bereichen zu Beschränkungen. Dies ist besonders wichtig bei der Darstellung und Aufbereitung grafischer Modeldaten. Da die OpenGL im Kern keine Funktionen zum laden externer Datensätze aus 3D-Modelldateien bietet wäre die die Auswahl an grafischen Modellen sehr beschränkt. In Anbetracht der knappen Entwicklungszeit wäre mit der Basislösung höchstens eine Menge an primitiven grafischen Objekten realisierbar. Dies hat mit realen Anwendungen wenig gemein.

Zur Aufhebung dieser Beschränkung kann der Einsatz von Darstellungsframeworks Abhilfe schaffen. Das bekannteste, C++ basierende Darstellungsframework ist dabei Open Scene Graph (OSG).

Open Scene Graph ist ein offenes, objektorientiertes Darstellungsframework. Es nutzt als Backend OpenGL. Dabei werden die einzelnen Objekte, Texturen und Effekte im Hintergrund als Baumstruktur, basierend auf der Graphentheorie, abgebildet. Dabei ist die Szene an sich das Wurzelelement, von welchem verschiedene Zweige zu den enthaltenen Objekten führen. Auch diese können wiederum Gruppen von anderen Objekten sein [Osf03] .

Die Programmierung und Speicherung des Graphen geschieht mit Hilfe von Spezialfeldern, sogenannten Reference Pointer. Diese Zeiger stellen zum Einen ein Objekt dar, dementsprechend den Zeiger auf die Instanz einer Klasse. Die Klassen können aus der Kernbibliothek von OSG oder aus einem seiner Derivate für spezielle Effekte entnommen sein. Zum anderen ist jeder Reference Pointer Teil einer von mehreren untereinander kaskadierten, doppelt verketteten Listen. Diese spezielle Ausprägung der Datenstruktur benutzt eine Modifikation der Vektorklasse der Standard Template Library.

Ein Hauptvorteil von OSG ist, neben der effektiven Datenverwaltung und der Aufteilung der Applikation in Teilgebiete, der relativ große Umfang an nativ mitgelieferten Plug-Ins zum laden und speichern von Standard-Dateiformaten der Computergrafik. Dabei umfasst das Plug-In Verzeichnis einen Großteil der in Kapitel 2 Abschnitt 3 genannten Datenformate, sowohl für Punktewolken als auch für polygonale Netze. Dabei werden jedoch bei der Darstellung Punktwolkenobjekte in polygonale Netze umgewandelt.

Desweiteren existieren viele, umfangreiche, auf OSG basierende oder OSG erweiternde Bibliotheken, welche bei der Weiterentwicklung des Programms zum Einsatz kommen könnten. Daher eignet sich OSG sowohl für kleine Szenen von Technikdemos als auch für das „Programmieren im Großen“.

Für die Erstellung der Demos sollte OSG nur als Darstellungsframework eingesetzt werden. Die Mechanismen des Backends zur Aufteilung sind sehr komplex und sollten daher getrennt

implementiert werden. Daher muss eine geeignete Verbindung gefunden werden. Architekturskizzen können im Anhang gefunden.

Beispielanwendungen, welche auf OSG basieren, sind das VENUS Projekt der Europäischen Union, VitualPlanetBuilder als Basis für militärische Flugsimulatoren und ossimPlanet dreidimensionales Geo-Visualisierungstool.

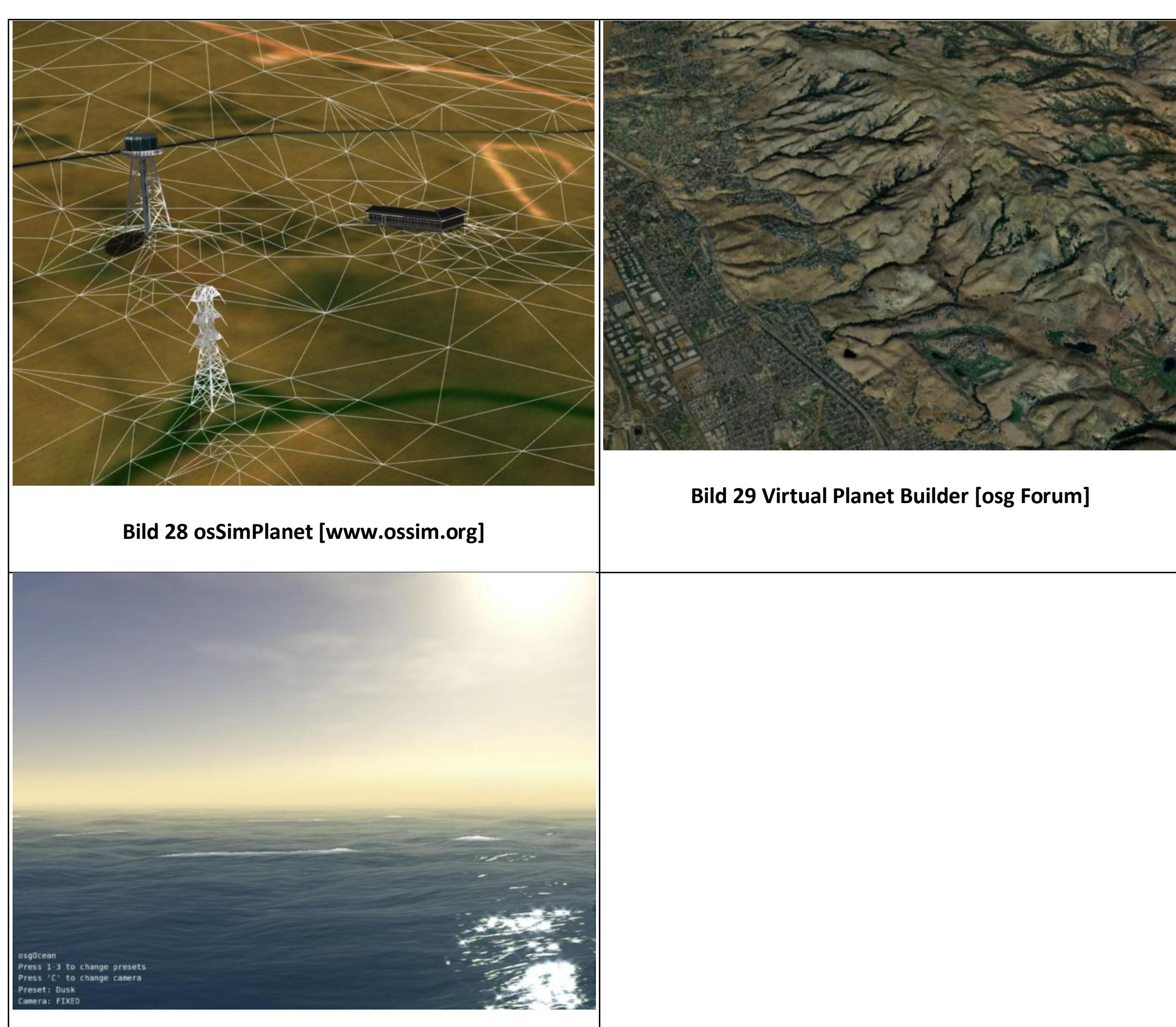


Bild 28 osSimPlanet [www.ossim.org]

Bild 29 Virtual Planet Builder [osg Forum]

Bild 30 osgOcean des VENUS-Projekts [code.google.com/p/osgocean]

3.9. Visualization Library

Die Visualization Library (VL) ist eine API ähnlich OSG, die zur Darstellung dreidimensionaler Szenen genutzt wird. VL wird in ähnlicher Weise wie OSG genutzt, sowohl generell als auch im möglichen Ansatz dieser Arbeit. Die Daten werden ebenfalls in objektorientierter Weise aufbereitet und in einer doppelt verketteten Liste gespeichert. Wie OSG so ist auch VL modular aufgebaut, mit einer Kernbibliothek und weiteren Erweiterungsbibliotheken.

Jedoch werden schon dabei Unterschiede deutlich. In der VL bestehen die Erweiterungen nicht aus Spezialeffekten. Alle Effekte beruhen auf der gleichen Technik und sind daher in der Kernbibliothek gespeichert. Die Erweiterungsbibliotheken dienen zur Entwicklung von Anwendungen für bestimmte Anwendungsgebiete wie z.Bsp. „vlMolecule“ für Molekularphysik oder „vlVolume“ für die Darstellung solide Objekte aus Punktwolke-Daten.

Eine weiterer, innovativer Architekturvorteil sind sogenannte „Applets“. Dies sind kleine Teilprogramme die bestimmend für die eigentliche Applikation sind. Grundgedanke ist, generelle Funktionen wie der Aufbau eines Renderkontext, die Einstellungen der virtuellen Kamera und Randbedingungen wie beispielsweise Multisampling-Rate des Anti-Aliasing in einem generellen Programmteil zu laden. Darauf folgend wird durch eine Auswahl das bestimmte Applet geladen, in welchem die spezifischen Informationen wie Objekte, Lichtverhältnisse und Umgebungseffekte gespeichert sind. Damit ist es möglich viele grundverschiedene Anwendungen in ein Programm zu inkludieren, ohne durch die Entwicklung eines Managementsystems dafür Zeit zu verlieren.

Die Architektur entspricht bietet eine Basis für das gewünschte Endergebnis, da hierbei Anwendungen aus unterschiedlichen Gebieten als Demos im einem Programm integriert werden können.

Folgend einige Beispiele der Darstellungsmöglichkeiten durch VL. Jene Beispiele sind offiziell auf der Visualization Library-Webseite auffindbar.

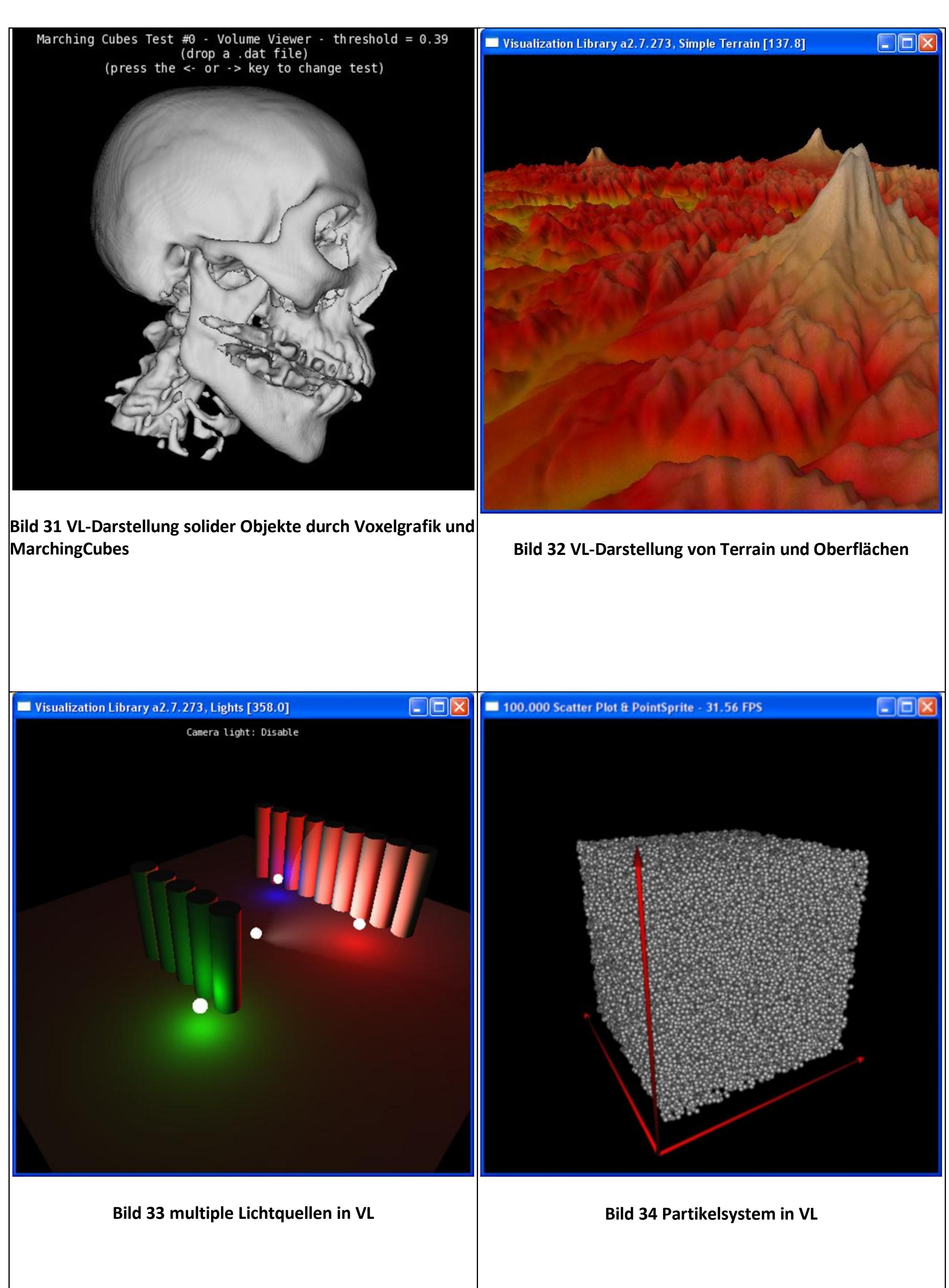


Bild 31 VL-Darstellung solider Objekte durch Voxelgrafik und MarchingCubes

Bild 32 VL-Darstellung von Terrain und Oberflächen

Bild 33 multiple Lichtquellen in VL

Bild 34 Partikelsystem in VL

Im Anhang befinden sich Architekturskizzen der API enthalten.

3.10. Equalizer

Zur Aufteilung der Renderfäche stehen generell nur die herstellerspezifischen Treiber oder die proprietären OpenGL Extensions zur Verfügung. Die Entwicklung der OpenGL Extensions basiert auf langjähriger Forschung in dem Bereich der Grafikaufteilung. Eines der führenden Firmen im Bereich dieser Forschung und Mitglied der Khronos Group ist Silicon Graphics International Corp. (SGI). SGI entwickelte und vertrieb langfristig eine eigene Linux-Distribution (IRIX), welche auf grafische Datenverarbeitung spezialisiert war[Sil09] . Desweiteren wurden von SGI in der Vergangenheit mehrere Lösungsansätze zur Aufteilung der Renderpipeline in Form von OpenGL-Zusatz-APIs veröffentlich. Dazu zählen der OpenGL Performer sowie das Multipipe Software Development Kit[SGI] . Die Ausgründung von Eyescale Software durch einen SGI Mitarbeiter führte diese Arbeit fort. Dadurch entstand 2007 der Equalizer als Software zur Aufteilung der Renderpipeline in Multi-GPU Umgebungen[Ste08] .

Equalizer ist ein Framework für Entwicklung und Vertrieb von parallelen OpenGL Applikationen. Es erlaubt der Anwendung von mehreren Grafikkarten, Prozessoren und Computern zu profitieren und skaliert Renderleistung, visuelle Qualität und Ausgabegröße. Eine Equalizer-basierte Anwendung läuft im Originalzustand auf jedem grafischen Darstellungssystem, vom einzelnen Desktoprechner bis zu großen Grafikclustern, Multi-GPU Systemen und Virtual Reality Systemen. Viele kommerzielle- sowie OpenSource-Anwendung in einer Vielzahl von Branchen nutzen den Equalizer wegen seiner Flexibilität und Skalierbarkeit.

Equalizer bietet dabei ein einfach anpassbares System zur Konfiguration und Skalierung. Dies wird ein weiteren Kapiteln ausführlich erläutert.

Durch die Charakteristik des Equalizers und dem integrierten Polygonial File Format-Reader bietet es sich als Ideallösung des Backend-Frameworks für die Anwendung an. Dazu wird ein großes Entwicklungspotential und eine Erweiterbarkeit der Anwendung in der Zukunft sowie dessen Skalierbarkeit geboten.

3.10.1. Architektur

Die Architektur des Equalizers ist ein mehrstufiges System, bei dem aufeinander aufbauende Komponenten vom jeweiligen Vater Element initialisiert werden.

Die zur Verfügung stehenden Elemente sind (in folgender Reihenfolge):

- Node
- Pipe
- Window
- Channel
- Canvas
- Segment
- Layout
- View
- Observers
- Compounds

Eine umfangreiche Beschreibung der Elemente kann in [Eil09] Kapitel 3 nachgelesen werden. Es folgt eine kurze Beschreibung der wichtigsten Elemente.

Eine Node ist ein physikalisches Rechensystem, welches am Rechennetzwerk angeschlossen ist. Dieses wird durch Session-Protokolle gesteuert und durch die IP-Adresse identifiziert. An einem der Computer muss das Darstellungssystem angeschlossen sein.

Eine Pipe ist eine GPU eines Rechners. Dabei werden die grafischen Daten wie die Nachricht eines Sender-Empfänger-Problems betrachtet, wobei die Daten unter wechselseitigem Ausschluss von dem Grafikkern verarbeitet werden. Dabei müssen die GPUs eines Rechners synchronisiert werden. Dabei wird jede GPU durch einen exklusiven Thread der Node gesteuert.

Ein Window ist ein logisches Fenster, welches von einer oder mehreren Grafikkarten verwaltet werden kann. Dabei ist ein Fenster an einen OpenGL-Kontext gebunden und damit der Empfänger der verarbeiteten Informationen der zugeordneten GPUs.

Ein Channel stellt den virtuellen Kanal dar, in dem die OpenGL Kommandos eines Fensters gespeichert werden. Zu einem Fenster können mehrere, untereinander austauschbare Kanäle gehören.

Ein Fenster wird am Ende auf ein oder mehrere Bildflächen, sogenannten Canvases, projiziert. Dabei kann es sich bei einer Bildfläche um einen Bildschirm, eine Wand oder mehrere Bildschirmwände handeln. Die Wände werden üblicherweise durch einen ein Virtual Reality Projector-System namens CAVE (Cave Automatic Virtual Environment) beleuchtet. Eine Bildschirmwand besteht wiederum aus mehreren Bildschirmen, den Segments, welche wiederum in Layouts und diese in unterschiedliche Views eingeteilt werden können.

Die Ausgaben der durch die unterschiedlichen physischen Komponenten gerenderten Objekte unterliegen den in Kapitel Eins sowie weiteren, folgend vorgestellten Kompositionsverfahren. Die Ausgaben dieser Kompositionsverfahren werden zu den korrespondierenden Anzeigeflächen übertragen.

Nach dem kurzen Überblick über die Equalizer-Elemente wird folgend auf die Softwarearchitektur eingegangen. Das dazugehörige Diagramm kann [Eil09] Seite 21 entnommen werden.

Eine Equalizer-Applikation ist, logisch oder physisch, aufgeteilt. Dabei wird vorerst die Server-Applikation auf dem Rechner gestartet, der im Verlauf der Gesamtanwendung die Bilder komponiert und darstellt. Durch die angegebene Datei entsteht eine Server-Konfiguration, in der die angeschlossenen Render-Client-Systeme enthalten sind. Daraufhin werden die Render-Clients gestartet, welche mit dem Server verbunden werden. Eine Session zwischen Server und Render-Clients wird aufgebaut. Es folgt der Informationsaustausch und Abgleich der Konfigurationen zwischen Server und Clients. Darauf folgend wird das Rendersystem aus Pipes, Fenstern, Kanälen und Visualisierungsoberflächen initialisiert. Diese Initialdaten werden vom System gesichert. Nachdem die Renderumgebung fertig gestellt ist wird das eigentliche Rendern gestartet. Dabei werden, basierend auf den Berechnung und Kommandos des aktuellen Kanals, die aktuellen Bilddaten zusammengefasst und durch die Renderpipeline verarbeitet. Nach Abschluss der Verarbeitung der Framedaten werden die Einzelbilder komponiert und dargestellt. Interaktionsmöglichkeiten sind in den Initialdaten festgelegt und werden in den Framedaten verarbeitet, wodurch die Interaktion mit vielen Eingabegeräten möglich ist.

3.10.2. Erweiterte Kompositionsalgorithmen

Zusätzlich zu den allgemeinen Kompositionsalgorithmen bietet das Equalizer-Framework weitere Bildkompositionsalgorithmen an, welche folgend vorgestellt werden. Quelle der Informationen ist [Eil09] Kapitel 2.

Beim DB- oder auch Sort-Last Compound Algorithmus werden die zu rendernden Daten als Datensätze einer Datenbank betrachtet. Dabei kann diese Stuktur logisch oder physisch bestehen. Datensätze gleicher oder verschiedener Größe werden, je nach Leistung der Clients und Grafikkarten, verteilt und von den Clients gerendert. Die Komposition findet durch Auswerten der Tiefenspeicher-Werte statt, wobei ein zusätzlicher Renderzyklus, jedoch nur auf dem Darstellungsrechner, nötig ist. Dieser Ansatz ist durch das High Performance Computing entstanden, wobei große Datenmengen nach dem Divide-and-Conquer Prinzip aufgeteilt werden und dann durch Sortieralgorithmen wie zum Beispiel Merge Sort wieder zusammengefügt werden.

DB Compounds erreichen eine gute Skalierbarkeit in Renderings, bei denen nicht die Shader-Effekte sondern die Anzahl der Vertices die Bildwiederholfrequenz einschränken. Dabei profitiert eine Darstellung von Punktwolken-Objekten besonders gut im Vergleich zu anderen Kompositionsalgorithmen, da durch den Tiefenvergleichstest nur die momentan

sichtbaren Punkte dargestellt werden. Desweiteren können Leistungszuwächse durch Kombination mit anderen Kompositionsalgorithmen erzielt werden.

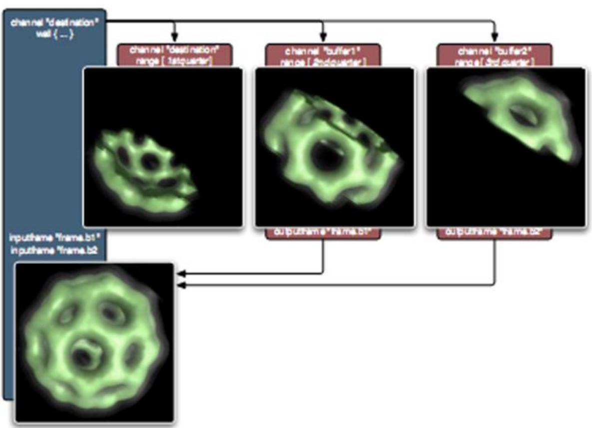


Bild 35 Equalizer DB Compound (sort last-Algorithmus) [Eil09]

Eine weitere, nicht von Standardkompositionsverfahren abgeleitete Methode ist das Stereo Compound. Hierbei wird die Szene aus verschiedenen Augenpositionen gerendert und durch den Server als ein Stereobild komponiert. Durch Equalizer werden Modi wie „Active Stereo“, „Anaglyphic Stereo“ und „Auto-Stereo Display“ unterstützt. Die Anzahl der partizipierenden Render-Clients wird durch die Anzahl der Renderdurchläufe pro Bild begrenzt. Dies sind zwei oder acht (Active Stereo ist vierfach gepuffert) Durchläufe pro Bild. Höhere Skalierbarkeit kann durch Kombination mit Standardkompositionsverfahren pro Augenpositionsbild erreicht werden.

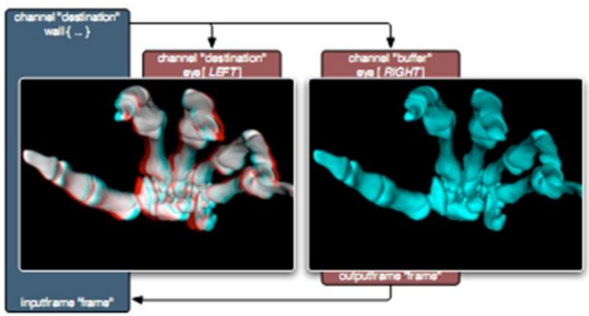


Bild 36 Equalizer Stereo Compound [Eil09]

Mit Equalizer Version 0.9.1 ist der Kompositionsmodus des Subpixel Compound implementiert worden. Dabei wird, ähnlich wie AFR, die grafische Szene mit veränderter Augenposition gerendert. Die, durch die Komposition entstehende, gewichtete Überlagerung von Pixel führt zu einer geglätteten Darstellung. Dadurch können, in Verbindung mit der Auswertung des Tiefenspeichers, Mehrfach-Render-Effekte wie Anti-Aliasing (AFR), aber auch Tiefenunschärfe und Motion Blur beschleunigt werden, welche dabei auf bis zu einen Renderdurchgang reduziert werden können.

3.11. Messwertermittlung

In Abschnitt 2.2 wurden die theoretischen Grundlagen und die Notwendigkeit einer Messwertermittlung zur Untersuchung bereits erläutert. Folgend werden Lösungswege aufgezeigt, wie die Messwertermittlung in die vorgestellten Lösungsvarianten einfließen kann. Dabei ist die Messwertermittlung stark von der jeweiligen Variante abhängig, da bei höheren Varianten das Abstraktionsniveau steigt. Dabei wird die Hardwarenähe der Messung verringert. Desweiteren sind die Auswirkungen systematischer Messfehler je nach Ansatz zu analysieren, deren Auswirkungen abzuwägen und Korrekturansätze zu finden. Messbare Werte sind die Ladezeiten von Phase Eins und Zwei sowie die Bildwiederholrate.

Das allgemeine Problem der Messwertermittlung in der Anwendung ist die Interprozesskommunikation und damit die Verwaltung geteilter Ressourcen, in welche die Messdaten eingetragen werden. Die Anwendung wird durch parallele Prozesse bestimmt. Die grafischen Daten werden parallel von den eingesetzten GPUs berechnet. Desweiteren existiert dabei häufig ein äquivalenter CPU-Thread, welche die berechneten Teilbilder aufnimmt und diese an einen Darstellungskontext weiterreicht. Demnach wird folgend geklärt, in wieweit die verschiedenen Lösungsansätze jene Probleme lösen.

Als erstes ist hierbei die Basislösung zu analysieren. Sie bietet das geringste Abstraktionsniveau und die dadurch bedingte größte Hardwarenähe. Jeglicher Ladevorgang von Objekten, Effekten und Umgebungsvariablen kann nachvollzogen, abgespeichert und gemessen werden. Daher ist die Messung von Ladezeitphase Eins vollkommen möglich. Ladephase Zwei bietet wenige Möglichkeiten zur Aufnahme von Messwerten. Da jedoch in der Basisvariante alle Schreibzugriffe eigens initialisiert werden, und dies nicht durch nebenläufige Prozesse gelöst ist, kann auch in dieser Phase ein Großteil (etwa 90%) der Operationen gespeichert werden. Lediglich das Laden der Shader und deren Variablen kann nur ansatzweise erfasst werden. Durch die Verteilung von CPU- und GPU-Operationen über die Teilklassen entsteht ein zeitlich stark ausgedehnter Übergang zwischen Ladephase Eins und Zwei. Die beiden Phasen sind daher schwer differenzierbar. Systematische Messfehler werden daher durch die Festlegung von Start und Ende der Ladephasen bestimmt. Die Auswirkungen der Messfehler hängen von der Schärfe der Trennung der Phasen ab. Messfehler können durch saubere Codetrennung minimiert werden. Dadurch erhöht sich die Anzahl unabhängiger Klassen. Der Aufwand steigt dabei exponentiell zur Trennungsstufe an. Für jede Trennungsstufe wird eine Klasse dabei in zwei neue Klassen aufgespalten. Dabei werden sukzessiv CPU- und GPU-Informationen getrennt. Dies kann im „worst case“ (bezüglich des Aufwands) in die Erstellung einer neuen API enden. Durch die eigenständige Programmierung der Renderschleife ist die Framerate jedoch sehr genau messbar.

Für die Physikberechnung auf der Grafikkarte stehen CUDA und OpenCL zur Verfügung. Beide verfügen intern über ein Profiling- und Logging-System, bei dem jegliche Aktion abgespeichert werden kann. Interne Datenstrukturen beider Systeme verfügen über Zeitmesser. Jegliche Aktionen in einer dieser Systeme kann einwandfrei geloggt und gemessen werden. Durch die klare Trennung von Code zur Umgebungserstellung und der eigentlichen Berechnung bestehen klare Start- und Endpunkte der Messung. Das Logging wird von parallelen Prozessen initiiert. Hierbei entsteht ein Problem im Zusammenhang mit der Interprozesskommunikation, wie in Kapitel 2 Abschnitt 2 erläutert. Dabei ist die geöffnete Logdatei als gemeinsamer Speicher der Kommunikationsraum der Prozesse. Zur übersichtlichen, sicheren Abspeicherung der Messwerte müssen Wege gefunden werden, die asynchrone, parallele Verarbeitung an gewissen Stellen zu synchronisieren. Dadurch kann jeder Thread eines GPU-Multiprozessors die Informationen als geschlossenen Chunk in das Log-Medium schreiben kann. Desweiteren muss beim Logging gekennzeichnet werden, welcher Mikroprozessor mit der dazugehörigen GPU den jeweiligen Chunk erstellt hat. Somit können im Anschluss Aussagen über Laufzeit und Verhalten der GPUs getroffen werden und im Fehlerfall die fehlerproduzierende GPU isoliert werden. Durch die Verwaltung der Verbindung von CPUs und GPUs durch niedrigpriorisierte Threads können ebenfalls geringfüge systematische Messfehler auftreten.

Die Darstellungsframeworks OSG und VL bieten keine integrierten Messwerkzeuge oder Klassen zum Messen von Laufzeiten an. Dabei werden jedoch durch diese Frameworks die Daten für die jeweilige GPU geladen. Bei der Wahl einer der Lösungen muss daher ein System zur Messwertermittlung geschaffen werden. Das Darstellungsframework wird jedoch vom Backend gesteuert. Durch die Lösung der IPC im Backend muss dies in der Darstellung nicht berücksichtigt werden. Es ist denkbar, den Speicherplatz für Messwerte über das Backend an das Darstellungsframework zu übertragen. Somit werden Parallelisierungs-probleme extern gelöst und das Darstellungsframework bleibt intern von der IPC unberührt.

Der Equalizer bietet sich als das synchronisierende Backend-Framework an. Daher wird in einer möglichen Kombinationslösung das jeweilige Darstellungsframework vom Equalizer gesteuert. Der Equalizer bietet intern Hilfsklassen für Interprozesskommunikation durch Pipes (lokal, GPUs), sowie Sockets (global, Rechennetze) an. Es fehlt jedoch an einer Implementation generischer Nachrichtenwarteschlangen. Durch den Charakter des hochgradig verteilten Systems des Equalizers ist es sinnvoll, bei der Software auf Sockets oder RPCs zurück zu greifen. Die Ladezeiten der Phasen Eins und Zwei werden dabei in den per-Frame-Klassen gemessen, in welchen das jeweilige Darstellungsframework eingebettet ist. Die Bildwiederholfrequenz bezieht sich jedoch auf das schlussendlich ausgegebene Bild der angeschlossenen Darstellungsfläche. Diese sollte daher vom Server abgespeichert werden. Bei der automatischen Abspeicherung gibt es die in Kapitel 2 analysierten Probleme. Daher sollten Mittelwerte der Bildwiederholfrequenz über eine gewisse Zeit bestimmt und abgespeichert werden. Ein ausgewogenes Verhältnis zwischen Simulationslänge und Logging-Umfang kann bei der Auswahl der Lösung getroffen werden. Eine weitere Möglichkeit der Messwertermittlung der Bildwiederholfrequenz ist die Nutzung

des internen Statistiksystems des Equalizers. Dabei wird die Anzahl der genutzten GPUs, die Ladezeit der jeweiligen Ladephasen sowie die Bildwiederholfrequenz optional auf den Bildschirm ausgegeben. Dies würde zu einer manuellen Messwertermittlung führen, wobei jedoch die Messzeitpunkte variabel bestimmt werden können.

3.12. Tools

Die verwendeten Werkzeuge der Arbeit können in vier Kategorien eingeteilt werden.

- Demo-Software
- Mess-Software
- Entwicklungssoftware (IDE)
- Content-Creation Systeme

Die erste Kategorie bildet dabei Software, welche anspruchsvolle grafische Inhalte darstellt. Hierzu gehören professionelle CAD-Systeme, CAS-Systeme etc. sowie Software, welche anspruchsvolle grafische Demonstrationen präsentiert. Dessen Leistungen werden soweit möglich gemessen und notiert. Diese Systeme profitieren nur von Multi-CPU Architekturen oder GPGPU-Technologien auf einer GPU.

Dazu wurden genutzt:

- Autodesk Alias Image Studio 2008 32-Bit
- GPU Caps Viewer 32-Bit

Zum Vergleich werden Messwerte für das Rendering dieser Softwarelösungen mit dem Leistungszuwachs der entwickelten Software verglichen.

In Kategorie Zwei sind Softwarelösungen zur Zeitmessung, CPU- und GPU-Auslastung zu finden. Durch anspruchsvolle Berechnungen sollten jeweilige Softwarelösungen ebenfalls über Temperaturanzeigen der jeweiligen Komponenten verfügen.

Als Mess-Software kamen zum Einsatz:

- GPU Caps Viewer 32-Bit
- GPU-Z 0.3.8 32-Bit
- Windows Task Manager
- CPU-Z 1.53 64-Bit

Die Messwerte der jeweiligen Ladezeiten und Bildwiederholfrequenzen sind im entwickelten System integriert.

Kategorie Drei sind Systeme zur Entwicklung der unterschiedlichen Softwarekomponenten des entwickelten Systems.

Dabei kamen zum Einsatz:

- Microsoft Visual Studio 2008 SP1
- CUDA Compiler Rule

- Typhoon Labs Shader Designer 1.6.0.0 2009 Fix
- ATI RenderMonkey
- NVIDIA FXComposer

Die letzte nennenswerte Kategorie sind Programme, mit denen darzustellende Objekte, Bilder, Höhenkarten und Texturen entworfen werden.

- Autodesk 3DS Max 2010 64-Bit Student Edition
- Autodesk Alias Image Studio 2008 32-Bit
- Normal Map Creator
- Seamless Texture Creator
- Paint .NET
- Adobe Photoshop CS 3

3.13. Vorstellung der Referenzsysteme

Folgend werden die drei Referenzsysteme präsentiert, die zur Messwertermittlung genutzt wurden. Dabei wurde jede Demonstrationsanwendung auf den möglichen Systemen gestartet und deren Laufzeit und Framerate gemessen.

Referenzsystem 1:

Laptop

Prozessor	Intel T2250 Core-2 Duo
Prozessoranzahl	2
CPU-Taktfrequenz	1,73 GHz
Hauptspeicherkapazität	3 GB
Speicherbandbreite	32 Bit
Grafikkarte	NVIDIA GeForce 7600 Go
Grafikchip	G76
GPUs	1
GPU-Takt	450 MHz
Grafikspeicher	400 MHz
Grafikspeichertakt	GDDR2 256 MB
Grafikspeicherbandbreite	12,8 GB/s
Shadereinheiten	8 Vertex- + 5 Fragment Shader
Shadertakt	450 MHz

Prometheus

Prozessor	Intel Core i7 920
Prozessoranzahl	8

CPU-Taktfrequenz	2,66 GHz
Hauptspeicherkapazität	4 GB
Speicherbandbreite	64 Bit
Grafikkarte	Sapphire ATI Radeon HD 4850 X2
Grafikchip	RV770
GPUs	2
GPU-Takt	625 MHz
Grafikspeicher	993 MHz
Grafikspeichertakt	GDDR3 1024 MB
Grafikspeicherbandbreite	127 GB/s
Shadereinheiten	800 Streamproc. (Compute Shader)
Shadertakt	625 MHz

Pandora

Prozessor	Intel Core i7 920
Prozessoranzahl	8
CPU-Taktfrequenz	2,66 GHz
Hauptspeicherkapazität	8 GB
Speicherbandbreite	64 Bit
Grafikkarte	2x NVIDIA GTX 275
Grafikchip	GT200b
GPUs	2x 1
GPU-Takt	633 MHz
Grafikspeicher	567 MHz
Grafikspeichertakt	GDDR3 896 MB
Grafikspeicherbandbreite	127 GB/s

Shadereinheiten	240 Compute Shader CUDA Cores
Shadertakt	1404 MHz

4. Entwurf ausgewählter Lösungsvarianten

Nach eingehender Analyse der Lösungsmöglichkeiten wurde entschieden, die Basislösung zu entwickeln. Aufgrund mehrerer, nachfolgender Gründe wurde jedoch daraufhin ein neues Rendersystem mit der Kombination aus Equalizer und Open Scene Graph entwickelt, welches allen momentanen Anforderungen standhält und dabei für zukünftige Entwicklungen erweiterbar ist. Die Physikengine der Kombination, Projektintern als „eqOSG“ bezeichnet, wurde mit drei Systemen entwickelt. In der 32-Bit Version wird die Physik ausschließlich von der CPU berechnet. In der 64-Bit Variante entscheidet das System selbstständig aufgrund vorhandener Technologien, ob die Physik über CUDA, OpenCL oder auf der CPU berechnet wird. Gründe für die Teilung in zwei verschiedene Versionen werden ebenfalls erläutert.

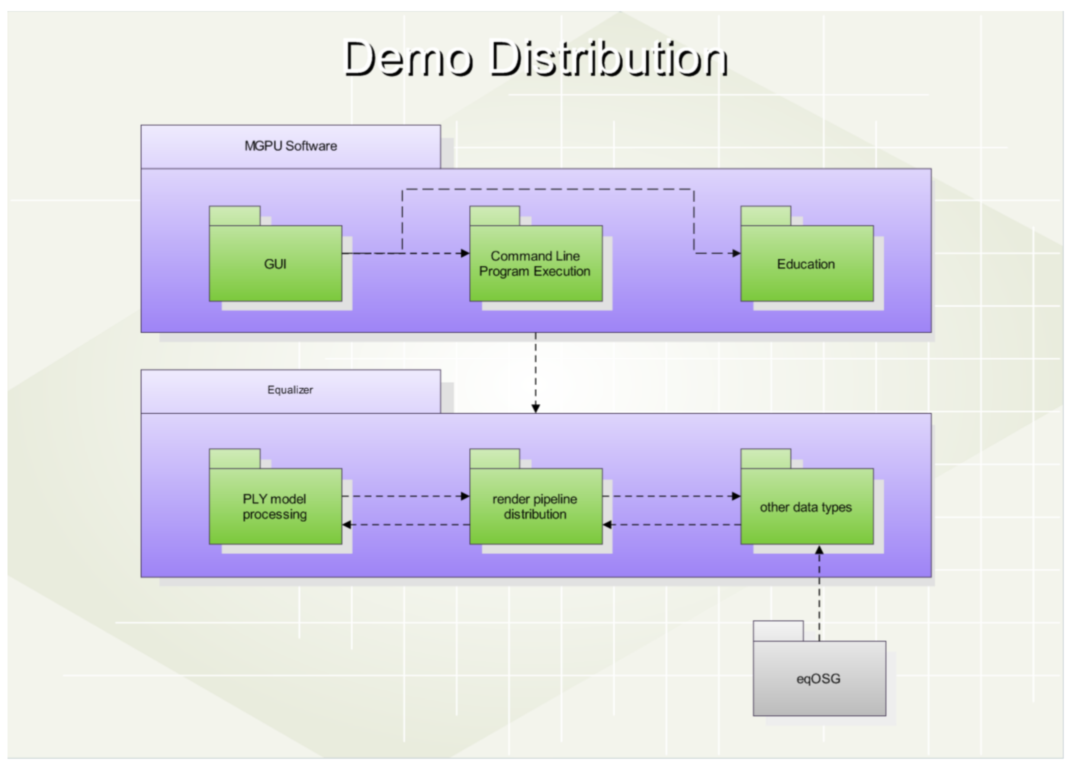
Aufgrund der Portabilität wird das Rendersystem von der Oberfläche getrennt. Zur Benutzerfreundlichkeit und Integration wird eine generelle Überarchitektur entwickelt, von der aus die Rendersysteme gestartet werden.

Im Anhang befinden sich Modelle der gesamten entwickelten Software. Damit ist erkennbar, welche Prototypen realisiert wurden, um die Entscheidung zu treffen, welche Frameworks tatsächlich zur Realisierung genutzt werden können.

4.1. Generelle Architektur

Zur Kombination der unterschiedlichen Rendersysteme mit unterschiedlichen Ansätzen, Architekturen und Bibliotheken ist die Software in Teilprogramme aufgeteilt. Dabei soll für jede angebotene Sotware-Version, unabhängig von Technologie und Architektur, eine einheitliche grafische Benutzeroberfläche geboten werden. Diese Entscheidung basiert auf den anvisierten Nutzerkreis der Software. Ärzte, Designer und Animateure haben in der Regel keine oder eine nur grundlegende Ausbildung in Computertechnik. Der PC wird als Arbeitsgerät angesehen. Dabei sollte die Software einfach installierbar und nutzbar sein. Das Oberflächendesign kann unter Umständen auch schlicht gehalten sein, eine grafische Pberfläche sollte jedoch vorhanden sein. In Hinblick auf Erweiterung sollten gewissen Abstraktionen von Einstellungsoptionen implementiert werden um die Software relativ intuitiv im Vergleich zur anderen Informationssystemen gestaltet sein. Dennoch sollte es die Möglichkeit geben, durch fortgeschrittene Modi neuere Technologien nutzbar zu machen und durch erweiterte Optionen jederzeit Vollzugriff auf die nicht-abstrahierten Werte, wie zum Beispiel Shaderparameter, zu erhalten.

Aufgrund dieser Abstraktion ist die Software in unterschiedliche Pakete aufgeteilt, welche folgend präsentiert werden.



In der Basislösung ist ebenfalls die Demo des animierten Würfels mit Oberflächenstruktur des HS-Wismar-Logos integriert, wobei dessen Entwurf im Folgenden präsentiert wird.

4.2. Basislösung

Das Ziel der ersten Beispielanwendung ist die Untersuchung der Möglichkeiten durch Basistechnologien. Dabei sollte Wert auf Erweiterbarkeit und Modularisierung gelegt werden. Desweiteren ist eine Lösung für die Aufteilung der Renderpipeline durch OpenGL Erweiterungen zu entwickeln.

Zur Modularisierung wird werden die erforderlichen Technologien und notwendigen OpenGL Funktionen analysiert. Dabei wird der Bottom-Up-Entwicklungsansatz gewählt. Es folgt eine Darstellung des Klassenentwurfs, welcher darauf folgend erläutert wird.

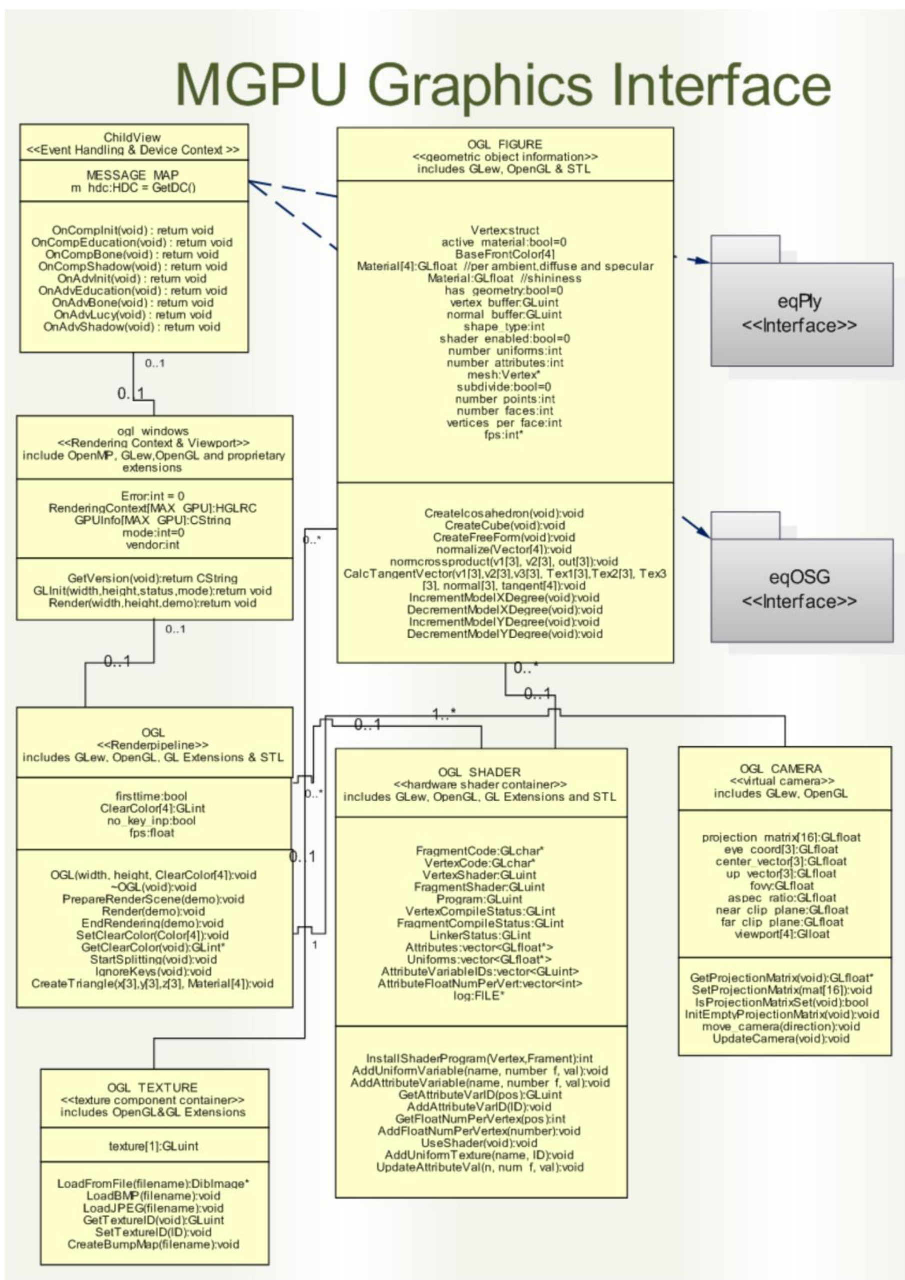


Bild 37 Klassendiagramm des Basisentwurfs

OpenGL bietet Verbindungsstrukturen- und Funktionen zu den jeweiligen Betriebssystemen an. Um eine etwaige Portabilität zu gewährleisten wird vorerst eine Klasse erstellt, die jene Verbindungs- und Basisfunktionen zum Betriebssystem beinhaltet.

Aufgrund der Wahl des Betriebssystems auf Microsoft Windows wird demzufolge vorerst nur eine Klasse für Windows-Kontexte erstellt. In jener werden anschließen Renderkontext, Pixelformatdeskriptor etc. zur Erstellung der OpenGL-Umgebung eingebunden. Dabei ist auf die Software-Architekturwechsel von OpenGL zu achten.

OpenGL bietet bis Version 1.3 nur eine feste Renderpipeline angeboten. Seit Version 2.1 besteht die Möglichkeit, die feste Renderpipeline durch programmierbare Shader teilweise

zu ersetzen. Die Architektur bleib die gleiche. Nach einer langen Ruhephase in der Weiterentwicklung von OpenGL wurde eine neue Softwarearchitektur entwickelt. Diese wurde mit Version 3.0 im August 2008 veröffentlicht. Problem bei der Entwicklung einer neuen Architektur für OpenGL ist, dass eine Übergangsarchitektur existieren muss, da die gesamte professionelle Grafikindustrie auf OpenGL setzt. Es benötigt viel Zeit, alle Produkte auf eine neue Architektur zu portieren. Dies wird dadurch erschwert, dass die Kunden mit der neuen Version ausgestattet werden müssen, da mit einem Update der Architektur ältere Versionen nicht mehr unterstützt werden. So besitzt die Version 3.0 zwei Architekturen. Die erste ist ein Kompatibilitätsmodus, der weiterhin auf eine feste Renderpipeline mit austauschbaren Modulen setzt. Der Funktionssatz ist auf Version 2.1 beschränkt. Desweiteren besteht eine vorwärtskompatible, neue Architektur die neuen Funktionen der Geometry Shader freischaltet und neuen Effekte ermöglicht. Jener Modus kann auch nach einem Update der OpenGL-Treiber auf den jeweiligen Systemen genutzt werden. Version 3.1, welche im März 2009 erschien, erweitert den Befehlssatz der neuen Version, führt den „Core Mode“ ein, welcher den vorwärtskompatiblen Modus ersetzt, und unterstützt die Shading Language 1.40. Die neueste Version von OpenGL ist 3.2, herausgegeben im Dezember 2009. Neue Features sind dabei die Shading Language 1.50, neue Funktionen zur Texturkompression, mehrfach gerenderte Texturen und ein neuer „Seamless CubeMap Filtering“-Algorithmus.

Weitere Funktionen der Verbindungsklasse sind die Erkennung der maximal unterstützten OpenGL-Version sowie das Auslesen der genauen Werte für Texturspeicher, Datenspeicher und die maximale Größe des Pixelspeichers.

Die Aufteilung der Renderpipeline wird durch Initialisierung von Szenen mit unterschiedlichen Ausmaßen erreicht. Hierbei stehen spezielle OpenGL-Funktions-Erweiterungen zur Verfügung, welche ab Version 3.1 genutzt werden können. Demnach wird bei Start eines Renderings zuerst die verfügbare OpenGL-Version ermittelt. Liegt eine ältere OpenGL-Version vor, so wird je nach Anwendung entschieden, ob abgebrochen wird oder das Rendering mit weniger Effekten erfolgt. Bei verfügbarer 3.1-Version wird ein Hauptkontext und, je nach Funktion, pro GPU ein Hintergrundkontext erstellt. Dazu wird pro GPU eine Szene mit den jeweiligen Ausmaßen zur Rendern kreiert.

Durch die Erstellung steht ein Renderkontext zur Verfügung. Die Steuerung der Unteraufgaben zur Darstellung wird durch die Szene des Kontexts durchgeführt. Bezüglich der Namenskonvention werden Klassen mit Betriebssystem-spezifischen Funktionen Namen mit Kleinbuchstaben geben, Betriebssystem-unabhängige Klassen erhalten Namen in Großbuchstaben. Da die Szene die Oberklasse von OpenGL ist wird diese mit „OGL“ bezeichnet. Desweiteren stellt die Szene gleichzeitig die Renderpipeline dar. Dabei wird in der Szene die Renderschleife implementiert, ebenso wie der EventHandler für Funktionseingaben. Nötige Variablen sind Hintergrundfarbe der Szene, eine globale Variable zur Anzeige der Initialisierung und, als markanter Punkt der Untersuchung der Renderpipeline, die gemessene Bildwiederholrate. Zur Interaktion mit anderen Klassen

müssen Funktionen zur Initialisierung der Parameter, zur Vorbereitung, zum Start und zur Beendigung der Renderpipeline existieren. Desweiteren werden Funktionen einer Animation implementiert, bei der der dargestellte Würfel durch sukzessive Tesselation in Einzelteile zerfällt.

Eine Szene besteht in weiterer Unterteilung aus einer virtuellen Kamera und den darin enthaltenen Objekten.

Die virtuelle Kamera bezeichnet dabei den sichtbaren Teil der Szene. Sie besitzt eine vordere und hintere Begrenzungsebene, sogenannte Clipping-Ebenen. Modifikationen dieser Ebene ermöglichen es, innerhalb von Objekte zu sehen, ohne diese aufzuteilen (Slice). Die hintere Clipping-Ebene gibt den Horizont an, ab welchem keine Objekte mehr erkennbar sind. Durch Benutzung der Clipping-Ebenen werden Objekte der Szene von Berechnung und Anzeige exkludiert, welche nicht sichtbar sind. Weitere Parameter der virtuellen Szene unterscheiden sich je nach verwendeter Projektionsart. Orthogonale Projektionen imitieren einen unendlich entfernten Betrachter bezüglich der Szene. Perspektivische Projektionen entsprechen der alltäglichen Sicht der Welt, weshalb diese verwendet wird. Für diese Projektionsart ist die Speicherung des Öffnungswinkels der Kamera nötig. Eine weitere Angabe ist das Größenverhältnis zwischen Breite und Höhe der betrachteten Position. Dieses Größenverhältnis ist beim Menschen naturgemäß 1. Weitere abgespeicherte Werte der virtuellen Kamera sind:

- Augenposition
- Sichtzentrum
- Normalvektor der Kamera

Zur Aufteilung der Pipeline muss jede GPU eine Szene besitzen. Aus Einfachheit und Übersichtlichkeit wird jeweils die gleiche Szene, jedoch mit unterschiedlichen Sichtbereichen (Viewports) genutzt. Diese Sichtbereiche stellen den Bereich dar, der durch die jeweilige GPU gerendert wird. Die Werte des Sichtbereichs werden beim Erstellen der Kamera durch die Szene festgelegt.

In der Szene enthaltene grafische Objekte unterliegen der Objektmodellierung, welche in Kapitel 2 erläutert wurde. Dabei wird die dafür enthaltene Klasse vorerst nur für Objekte aus polygonalen Netzen entworfen. Jedes Objekt besitzt eine Liste aus Vertices, Normalvektoren und optionalen Farben. Zur Anordnung im polygonalen Netz können Punkte einer Fläche hintereinander abgespeichert werden. Die Indizierung ergibt aus der Reihenfolge in der Liste, jedoch besteht der Nachteil, dass dabei Punkte unnötig mehrfach abgespeichert werden. Dies ist nicht optimal, jedoch wird in einigen grafischen Objekt-Dateiformaten die Indizierung ebenfalls so gelöst. Eine weitere Möglichkeit ist die Erstellung einer zusätzlichen Indexliste, wobei für nacheinander zu zeichnende Punkte der Index der Punkte in der Punkteliste abgespeichert wird. Hierbei wird eine Mehrfachspeicherung von Punkten verhindert. Neben diesen generellen Informationen für grafische Objekte kann ein Zeichen abgespeichert werden, welches den Typ der Instanz der Klasse kennzeichnet. Somit werden

Funktionen implementiert um unkomplizierte Darstellungen einfacher grafische Primitiva zu ermöglichen.

Desweiteren werden Effekte durch Shader für einzelne Figuren ermöglicht sowie die Texturierung dieser Figuren.

Für Shader müssen die Dateinamen der jeweiligen Shader-Dateien abgespeichert werden. Desweiteren besitzen Shader mögliche Zusatzvariablen, welche je nach Art gesichert werden müssen. Zum Setzen der Variablen sowie zum Laden, Ein- und Ausschalten des Shaders müssen Member-Funktionen vorhanden sein.

Bezüglich einer Textur wird der Dateiname des Bildes sowie dessen physische Repräsentation als Textur (DIB-Sektion) in Form von Klassenvariablen hinzugefügt. Desweiteren benötigt es Funktionen, die, je nach Bild-Dateityp, die Datei laden, dessen optional komprimierten Daten auslesen und diese als Textur abspeichern.

Zusätzlich müssen Oberflächenfunktionen kreiert werden, um eine mögliche Verbindung zu weiteren Rendersystemen vorzubereiten. Damit werden etwaige Technologie- und API-Wechsel erleichtert.

Die Aufteilung der Szene ist auf der Ebene der Grundeinstellungen der Szene zu finden. Diese Szenen-spezifischen Einstellungen mit Hilfe der Kernerweiterungsfunktion werden daher in die Betriebssystem-spezifische Klasse „ogl_windows" integriert. Dabei wird bei Existenz mehrere GPUs auf dem Nutzersystem je GPU eine OGL-Szene mit gleichen Daten aber unterschiedlichen Viewport initialisiert.

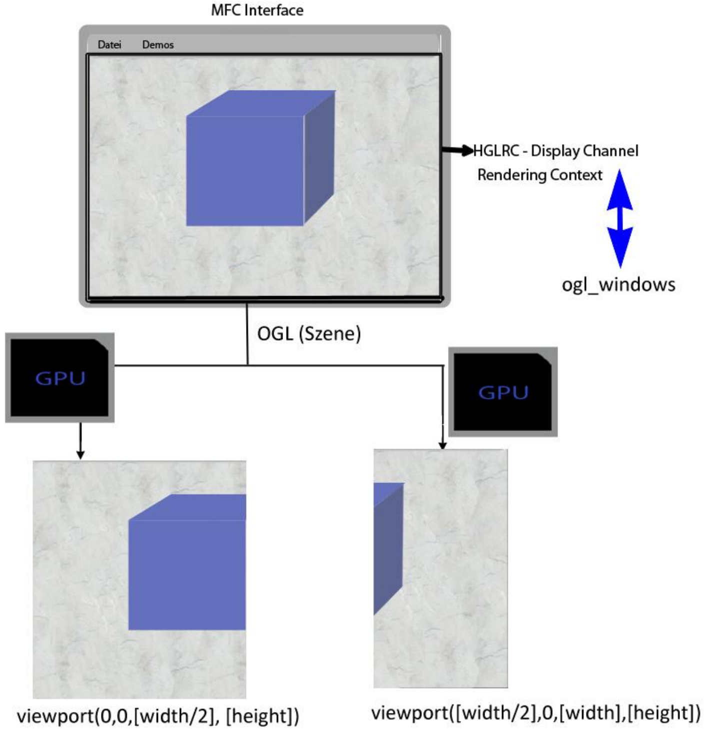


Bild 38 Visualisierung der Aufteilung im Klassenkontext des Basisentwurfs

4.3. Equalizer und Open Scene Graph

Nach Implementation der Basislösung sind erste Analysen bezüglich folgender Kriterien bekannt:

- Flexibilität
- Portierbarkeit
- Stabilität
- Aufwand

Die Flexibilität der Anwendung ist größtenteils zufriedenstellend. Neue Technologien wie z.Bsp. Multi-Texturing sind damit durchaus möglich. Eine Messung der Bildwiederholrate kann ebenfalls durchgeführt werden. Neue Shader können mit Hilfe neuer OpenGL-Kontexte sowie der OGL_SHADER-Klasse implementiert werden. Im Bereich der Objektmodellierung ist die Anwendung jedoch sehr starr. Es gibt keine Schnittstelle um grafische Objekte-Daten aus Dateien zu erstellen. Desweiteren ist die Anwendung auf die Nutzung polygonaler Netze limitiert. Generierung von Freiformen ist sehr aufwendig und mit dem entwickelten, starren Entwurf nicht möglich. Einzig die Generierung von grafischen Primitiva ist gut gelöst.

Die Portabilität ist durch spezielle Verbundklassen sichergestellt. Der Großteil des Entwurfs ist Betriebssystem-unabhängig. Hierfür konnten jedoch keine abschließenden Tests unter Ubuntu oder MacOS X durchgeführt werden.

Durch die Anzahl an Code-Zeilen ist die Verwaltung sowie Sicherstellung der Stabilität der Anwendung nicht gewährleistet.

Die Analyse des Aufwands ergibt eine Realisierungszeit von 10 Wochen für eine Darstellung eines simplen, texturierten Würfels mit Bump Mapping-Shader. Aufgrund dieser Zahl ergibt eine grobe Hochrechnung für die Implementation der NVIDIA-Erweiterung sowie der Erstellung einer Demo mit realistischer Komplexität einen Arbeitsaufwand von ungefähr 54 Monaten (entsprechend 4,5 Jahren). Demnach ist der Ansatz der Basislösung zur Untersuchung, Entwicklung und Messwertermittlung weder für die Arbeitszeit der Thesis noch für eine wirtschaftliche Nutzung der Ressourcen geeignet. Der Ansatz der Basislösung wird zur Weiterarbeit verworfen und im Folgenden nur als grafische Oberfläche zum Aufruf anderer, flexiblerer Anwendungsansätze genutzt.

Zur Aufteilung der Renderpipeline wird aus Effizienzgründen eine Kombination aus Backend- und Frontend-Framework gewählt. Für das Backend werden APIs oder Frameworks benötigt, die über den vollen Befehlssatz verfügen und mit deren Hilfe es möglich ist, OpenGL-Kommandos zu verteilen. Dabei umfasste die Auswahl:

- Equalizer
- Rapidmind
- Chromium

Da Chromium durch die fehlende Multi-GPU Unterstützung keine Lösung zur der eigentliche Aufgabe der Arbeit bietet, wird es nicht genutzt. Die Fähigkeiten von Chromium sind in ähnlichem Ansatz in den beiden Konkurrenzsystemen ebenfalls vorhanden.

Rapidmind wurde aus wirtschaftlichen Gesichtspunkten nicht gewählt. Obwohl es die ideale Lösung zur Aufteilung bietet, so ist die Anschaffung angesichts der Konkurrenz nicht nötig.

Der Equalizer vereinigt die Multi-GPU Möglichkeiten von Rapidmind mit der Flexibilität und Cluster-Skalierbarkeit von Chromium. Der gute Support des Entwicklers und die funktionier-ende Zusammenarbeit mit Eyescale während der Spezifikationsphase ist ein weiterer positiver Punkt für den Equalizer. Aufgrund dessen wurde der Equalizer als Backend Framework gewählt.

Darauf aufbauend wurden die Möglichkeiten des Zusammenspiels des Equalizers mit den verfügbaren Oberflächenframeworks untersucht. Eine prototypische Integration der Visualization Library war auch mit Zusammenarbeit von Eyescale erfolglos und scheiterte an der Komplexität des internen Aufbaus des Equalizers. Die Entscheidung für Open Scene Graph hatte eine längere Einarbeitungszeit in das Oberflächenframework zur Folge, jedoch existiert eine funktionierende Verbindung beider Frameworks, was schließlich zur Entscheidung für OSG führte. Durch die Verbindung von Equalizer und Open Scene Graph wird das Projekt als „eqOSG“ bezeichnet.

In Folge der Entwicklung wurde die Integration der Physikengine notwendig. Hierbei war folgende Auswahl an Bibliotheken und APIs gegeben:

- OpenCL
- Brook/Brook+
- CAL
- CUDA

Aufgrund des für die Arbeit nicht tragbaren Aufwands der Implemention wird CAL als mögliche Sprache zur Realisierung der Physikengine verworfen.

Die auslaufende Unterstützung von Brook+ als GPGPU-Sprache für ATI-Karten durch das Stream SDK erschwert die Entwicklung auf ATI-Karten. Eine Unterstützung der Sprache durch die Chips ist zwar gegeben, jedoch ist der Vertrieb von Compiler, Debugger und Profiler sowie der IDE für Brook+ von AMD/ATI komplett eingestellt worden. Daher wird diese Lösung vorerst nicht weiter verfolgt.

Durch die große Community und die Leistungsfähigkeit von CUDA als auch der Hersteller-Unabhängigkeit wird für die Physikengine eine Kombination aus CUDA und OpenCL verwendet.

4.3.1. Server-Client-Paradigma

Aufgrund der Nutzung des Equalizers und diverser anderer, noch zu erläuternden Gründen muss bei der Entwicklung des neuen Systems ein grundverschiedener Ansatz gewählt werden. Hierbei handelt es sich um das Server-Client-Paradigma.

Durch das Paradigma wird ein Softwaresystem als eine Einheit eines Datenverteilers (Server) und den verarbeitenden Komponenten (Clients) gesehen. Es bietet sich als Modell der Structured Analyse and Design Technique an, bei der Software eine Menge von Daten ist, welche durch ein System von Aktoren verändert werden. Die Benutzung des Server-Client-Prinzips ist dabei nicht nur auf physisch verteilte Systeme von Rechnern anwendbar. Server und Clients können ebenso verschiedene Komponenten eines Rechners oder verschiedene Arten von Softwaresystemen und deren Schnittstellen sein. Prägnantes Merkmal des Prinzips ist die Anwendung auf heterogene Systeme.

In Hinblick auf den Einsatz in eqOSG wird dabei das verarbeitende Rechnernetz als heterogenes System betrachtet. Dabei ist die Einheit von Memory Controller Hub, Hauptspeicher und CPU der Server. Dieser enthält alle Daten, die von der Festplatte in den RAM geladen werden. Software-intern wird entschieden, welche Komponente des Systems daraufhin die Daten verarbeitet. Dabei sind die verschiedenen Grafikprozessoren die Clients. Schlussendlich werden die verarbeiteten Daten an Ausgabeclients wie beispielsweise den Bildschirm weitergeleitet.

Der Equalizer basiert in seiner Grundform auf dem angesprochenen Prinzip, da durch den Equalizer die Daten nicht nur von einem Rechner sondern ebenfalls über ein Netzwerk durch mehrere Rechner verarbeitet werden kann. Dabei agiert ein Rechensystem in der Gesamtheit als Server während Workstations, welche über ein Netzwerk adressiert werden, als Clients arbeiten.

Desweiteren bietet sich das Server-Client-Paradigma heutzutage allgemein für zunehmend heterogene Systeme an. Während in vergangenen Jahren die CPU die herausragende Komponente für jegliche Art von Datenverarbeitung war haben Fortschritte in der Chip-Herstellungstechnologie sowie in der Entwicklung Komponenten-spezifischer Softwareschnittstellen dazu geführt, dass sich zur Leistungssteigerung die Verteilung von Rechenlast auf spezifische Komponenten je nach Anwendung anbietet. Dabei werden grafische Informationen, ob nun durch Computergrafik, Bildverarbeitung oder Videoverarbeitung, durch den hochgradig parallelisierbaren Charakter am besten auf der

GPU ausgeführt, während die Lösung von Differentiation und Integration schneller durch einen Audioprozessor durchgeführt. Zur idealen Verteilung der Rechenlast bietet sich daher an, die CPU als Server-Komponente mit Managementaufgaben zu verwenden, welche die Verbindung zu jeder Teilkomponente darstellt. Die spezialisierten Teilkomponenten sind dabei die Verarbeitungszellen in Form von Clients.

Durch Anpassung und Einsatz des Server-Client-Prinzips kann die entwickelte Software auch für zukünftige Aufgaben mit etwaigen Anforderungserweiterungen aus anderen Bereichen genutzt werden.

Folgende Skizze erläutert die lokale und globale Server-Client-Architektur.

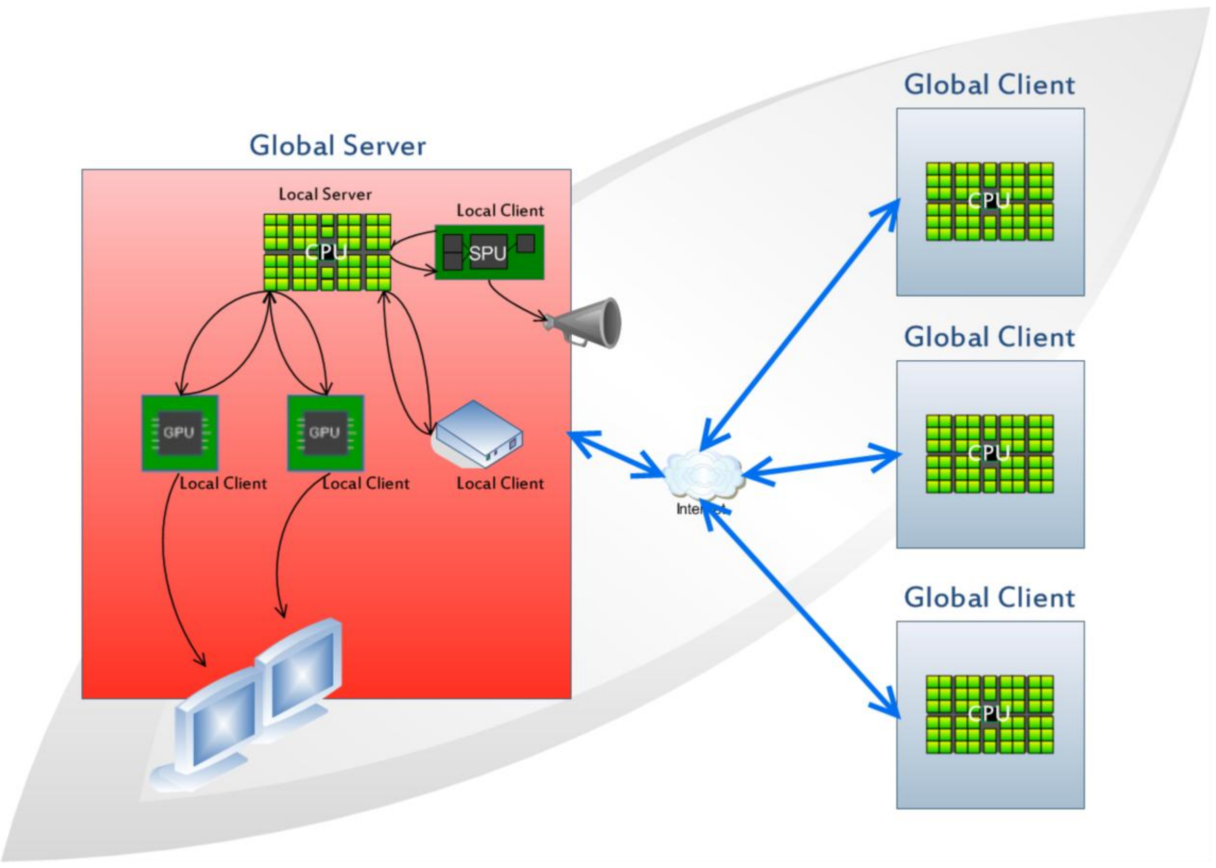


Bild 39 Verteilte Server-Client-Architektur des Equalizer-basierten Entwurfs

4.3.2. Ansatz

Die Software eqOSG verteilt Rechenlast in zwei unterschiedlichen Bereichen und Phasen. Zum Einen werden auf globaler Ebene die Daten durch eine festgelegte Equalizer-Konfiguration verteilt. Vorerst wird dabei die Verteilung der Rechenlast auf einem lokalen System in einer Multi-GPU Umgebung fokussiert. Hierbei wird ebenfalls der verwendete

Kompositionsalgorithmus festgelegt. Diese Aufteilung bezieht sich auf die Komposition der Ergebnisse der Rechenpipeline der einzelnen GPUs.

Eine weitere Verteilung bezieht sich auf die physikalische Berechnung der Windgeschwindigkeiten der Staubpartikel in der Szene. Dabei wird die Berechnung je Bild vor dem Einsatz der Renderpipeline auf den Grafikprozessoren ausgeführt. Die Verteilung geschieht, wenn die Unterstützung einer GPGPU-Technologie vorhanden ist, durch die jeweilige GPGPU-Schnittstelle/Bibliothek. Es wurde sich bezüglich der GPGPU-Technologien für eine Kombination aus CUDA und OpenCL entschieden. Jedoch sollte die geschaffene, neue Physikengine soweit flexibel sein, dass jederzeit die Integration weiterer Technologien, wie zum Beispiel CAL, vorgenommen werden kann. Sollte keine GPGPU-Technologie vorhanden sein wird die Physik weiterhin durch den Hauptprozessor berechnet. Hierbei wird mit Leistungseinbußen gerechnet.

Zur Berechnung der Physik werden eine bestimmte Umgebungstemperatur sowie eine bestimmte Luftfeuchtigkeit an den Emittern der Partikelsysteme als Eingabeparameter vorausgesetzt. Durch einen vierstufigen Kernel wird dabei die resultierende Windgeschwindigkeit am jeweiligen Punkt berechnet.

Architektur und Ablauf der Software sind in Klassen-, Übersichts- sowie Aktivitäts-diagrammen im Anhang zu finden. Es folgt ein Komponentendiagramm zu Darstellung des Zusammenspiels der verschiedenen APIs, Bibliotheken und Frameworks.

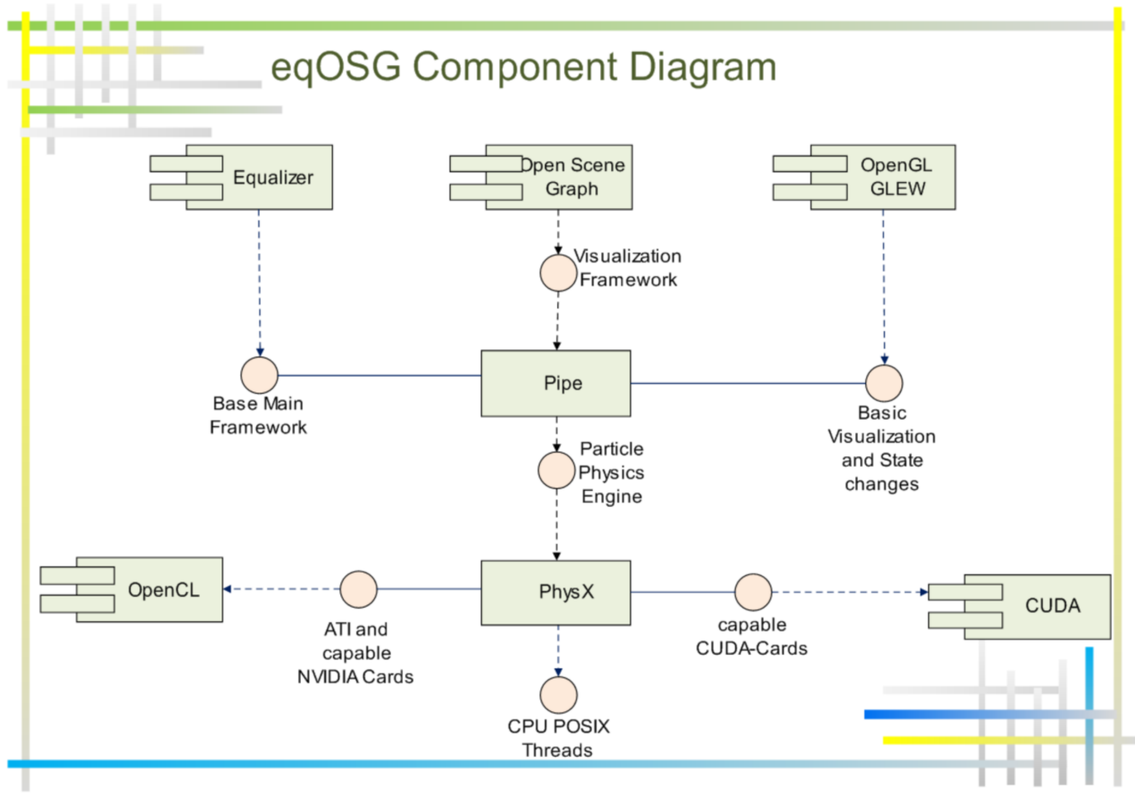


Bild 40 Komponentendiagramm des Equalizer-basierten Softwareentwurfs (eqOSG)

5. Realisierung gewählter Lösungen

In Kapitel vier wurden gewählte Lösungsvarianten der analysierten Möglichkeiten präsentiert. In diesem Abschnitt wird nun detailliert auf Eckpunkte der Implementation jener Lösungen eingegangen.

Anlehnend an Kapitel vier wird dabei ebenfalls mit der Basislösung begonnen. Hierbei wird zu Beginn präsentiert, wie sich die Grafikpipeline mit Hilfe von OpenGL-Extensions in einer Multi-GPU Umgebung aufteilen lässt. Jene Erweiterungen bilden die Basis aller Aufteilungslösungen. Weitere Aufteilungen durch Frameworks oder APIs basieren direkt (Equalizer) oder indirekt (OpenCL und CUDA) auf jenen Erweiterungen. Desweiteren wird Anwendung und Shader zur Realisierung des Bump Mappings erläutert.

Aufgrund des fehlenden Debuggers für CUDA und des dazu noch fehlenden Compilers für OpenCL wurde ein Testsystem zur Sicherstellung der korrekten Funktionsweise dieser Technologien entwickelt. Auf dieses Testsystem und die damit verbundenen Rückschlüsse zur Implementation der Physikengine wird in einem weiteren Abschnitt des Kapitels eingegangen.

Schlussendlich wird eqOSG als Rahmenlösung zur Visualisierung verschiedener Demos präsentiert. Dabei wird zuerst auf die Konfigurationsdatei für eine Multi-GPU Umgebung eingegangen, welche Verteilung sowie Komposition beschreibt. Darauf folgend wird die Nachrichtenwarteschlange zur Nutzer-Software-Interaktion beschrieben. Da es sich bei eqOSG um ein Rahmenwerk für verschiedene Demos handelt wird ebenfalls auf die Demonstrationsauswahl eingegangen. Ein weiterer Absatz behandelt die Terraingenerierung durch Höhenkarte und Textur. Die abschließenden zwei Absätze behandeln die Erkennung der verfügbaren Parallelisierungstechnologie (GPGPU) sowie letztendlich die genutzte Physikberechnung und das realisierte Partikelfeld.

5.1. Basislösung

Neben der nachhaltig verwendeten grafischen MFC-Oberfläche der Basislösung ist auf zwei entwickelte Teile der Basislösung genauer einzugehen.

Dabei handelt es sich in erster Hinsicht um die Aufteilung der Renderpipeline durch angesprochene Erweiterung. Dies ist als Referenzlösung für derartige Implementationen zu sehen, insbesondere da es bisher noch keine lauffähige Implementation der ATI-Erweiterung existiert. Daher konnte bei der Realisierung nur auf das Material der herausgegebenen Spezifikation zurückgegriffen werden. Die Spezifikationen beider Erweiterungsfunktionen sind im Anhang zu finden.

Zur Erläuterung der Komplexität des damit verbundenen Umdenkens in Entwurf und Entwicklung der Lösungen werden ebenfalls Teile der Implementation des Bump Mappings für einen Würfel mit HS Wismar-Logo dargestellt.

5.1.1. Ansatz der herstellerabhängigen Aufteilung der Renderpipeline

Die manuelle Aufteilung der Grafikkarte wird durch Implementation neuer, proprietärer OpenGL-Erweiterungen erreicht.

Dazu muss im Vorfeld überprüft werden, ob eine der Aufteilungsfunktionen zu Verfügung steht. Desweiteren wird dabei auch erkannt, welche herstellerspezifische Funktion genutzt wird.

```
//Check for AMD/ATI – MultiGPU Extension
if(OpenGLExtensionSupported("WGL_AMD_gpu_association"))
{
	//continue with AMD Architecture Extension
}
else if(OpenGLExtensionSupported("WGL_NV_gpu_affinity"))
{
	//continue with NVIDIA Architecture Extension
}
```

In den folgenden Abschnitten wird die die AMD/ATI-Erweiterung durch Code dargestellt, da diese implementiert wurde. Die Unterscheide des NVIDIA-Pendants werden dabei kurz erläutert.

Basierend auf der erfolgreichen Erkennung einer der Technologien ist es nötig, die Funktionen der jeweiligen Erweiterung zur Verfügung zu stellen. Diese Funktionen sind bei Vorhandensein der Erweiterung nativ auf der Grafikkarte im Codesegment des

Grafikspeichers abgelegt. Um diese nutzen zu können müssen Zeiger auf die jeweilige Funktion kreiert und die Funktion damit gebunden werden.

Dazu wird im ersten Schritt ein Prototyp der späteren Funktion deklariert. Hierbei wird die Parameterliste festgelegt. Darauffolgend wird die Prototypdeklaration mit dem eigentlichen Funktionsnamen verbunden. Im letzten Schritt wird die Adresse der Funktion aus dem Grafikspeicher extrahiert und mit dem Zeiger der vorher festgelegten Prototypdeklaration verbunden.

Bezüglich der AMD/ATI-Erweiterung werden Funktionen benötigt um Grafikkarteninformationen zu extrahieren, OpenGL-Kontexte für die jeweilige GPU anzulegen, GPU-gebundene Kontexte freizugeben sowie erstellte Bilder der Grafikpipeline anzuzeigen.

```
//declare function prototypes

#ifdef _WIN32
typedef UINT (APIENTRY * PFNWGLGETGPUIDSAMDPROC)(UINT maxCount, UINT *ids);
typedef INT (APIENTRY * PFNWGLGETGPUINFOAMDPROC)(UINT id, INT property, GLenum dataType,
UINT size, void *data);
typedef UINT (APIENTRY * PFNWGLGETCONTEXTGPUIDAMDPROC)(HGLRC hglrc);
typedef HGLRC (APIENTRY * PFNWGLCREATEASSOCIATEDCONTEXTAMDPROC)(UINT id);
typedef HGLRC (APIENTRY * PFNWGLCREATEASSOCIATEDCONTEXTATTRIBSAMDPROC)(UINT id, HGLRC
hShareContext, const int *attribList);
typedef BOOL (APIENTRY * PFNWGLDELETEASSOCIATEDCONTEXTAMDPROC)(HGLRC hglrc);
typedef BOOL (APIENTRY * PFNWGLMAKEASSOCIATEDCONTEXTCURRENTAMDPROC)(HGLRC hglrc);
typedef HGLRC (APIENTRY * PFNWGLGETCURRENTASSOCIATEDCONTEXTAMDPROC)(void);
typedef VOID (APIENTRY * PFNWGLBLITCONTEXTFRAMEBUFFERAMDPROC)(HGLRC dstCtx, GLint srcX0,
GLint srcY0, GLint srcX1, GLint srcY1, GLint dstX0, GLint dstY0, GLint dstX1, GLint
dstY1, GLbitfield mask, GLenum filter);
#endif
```

```
//initialize AMD multicore procedures and bind prototypes to function names
PFNWGLGETGPUIDSAMDPROCwglGetGPUIDsAMD        = NULL;
PFNWGLGETGPUINFOAMDPROC       wglGetGPUInfoAMD       = NULL;
PFNWGLGETCONTEXTGPUIDAMDPROC  wglGetContextGPUIDAMD = NULL;
PFNWGLCREATEASSOCIATEDCONTEXTAMDPROC wglCreateAssociatedContextAMD = NULL;
PFNWGLCREATEASSOCIATEDCONTEXTATTRIBSAMDPROC wglCreateAssociatedContextAttribsAMD = NULL;
PFNWGLDELETEASSOCIATEDCONTEXTAMDPROC wglDeleteAssociatedContextAMD= NULL;
PFNWGLMAKEASSOCIATEDCONTEXTCURRENTAMDPROC    wglMakeAssociatedContextCurrentAMD = NULL;
PFNWGLGETCURRENTASSOCIATEDCONTEXTAMDPROC    wglGetCurrentAssociatedContextAMD = NULL;
PFNWGLBLITCONTEXTFRAMEBUFFERAMDPROC  wglBlitContextFramebufferAMD= NULL;
```

```
//bind function addresses to function pointer

wglGetGPUIDsAMD = (PFNWGLGETGPUIDSAMDPROC) wglGetProcAddress("wglGetGPUIDsAMD");

wglGetGPUInfoAMD = (PFNWGLGETGPUINFOAMDPROC) wglGetProcAddress("wglGetGPUInfoAMD");

wglGetContextGPUIDAMD = (PFNWGLGETCONTEXTGPUIDAMDPROC)
wglGetProcAddress("wglGetContextGPUIDAMD");

wglCreateAssociatedContextAMD = (PFNWGLCREATEASSOCIATEDCONTEXTAMDPROC)
wglGetProcAddress("wglCreateAssociatedContextAMD");

wglCreateAssociatedContextAttribsAMD = (PFNWGLCREATEASSOCIATEDCONTEXTATTRIBSAMDPROC)
wglGetProcAddress("wglCreateAssociatedContextAttribsAMD");

wglDeleteAssociatedContextAMD = (PFNWGLDELETEASSOCIATEDCONTEXTAMDPROC)
wglGetProcAddress("wglDeleteAssociatedContextAMD");

wglMakeAssociatedContextCurrentAMD = (PFNWGLMAKEASSOCIATEDCONTEXTCURRENTAMDPROC)
wglGetProcAddress("wglMakeAssociatedContextCurrentAMD");

wglGetCurrentAssociatedContextAMD = (PFNWGLGETCURRENTASSOCIATEDCONTEXTAMDPROC)
wglGetProcAddress("wglGetCurrentAssociatedContextAMD");

wglBlitContextFramebufferAMD = (PFNWGLBLITCONTEXTFRAMEBUFFERAMDPROC)
wglGetProcAddress("wglBlitContextFramebufferAMD");
```

Durch das unterschiedliche Konzept von NVIDIA (siehe Modellskizze) werden Funktionen zum extrahieren der GPU Informationen, erstellen deines Gerätekontextes mit Zugehörigkeitsmaske, wechseln der Zugehörigkeitsmaske sowie zum löschen des Gerätekontextes benötigt.

Die dafür nötigen Funktionen sind wie folgt festgelegt:

```
BOOL wglEnumGpusNV(UINT iGpuIndex, HGPUNV *phGpu);
BOOL wglEnumGpuDevicesNV(HGPUNV hGpu, UINT iDeviceIndex,
PGPU_DEVICE lpGpuDevice);
HDC wglCreateAffinityDCNV(const HGPUNV *phGpuList);
BOOL wglEnumGpusFromAffinityDCNV(HDC hAffinityDC, UINT iGpuIndex,HGPUNV *hGpu);
BOOL wglDeleteDCNV(HDC hdc);
```

Nachdem die Funktionen zur Verfügung stehen müssen die Ausmaße der zu zeichnenden Bereiche festgelegt werden. Dabei unterscheiden sind NVIDIA und ATI nicht. Der Unterschied ist, dass die Bereiche bei NVIDIA für eine virtuelle Zeichenfläche auf dem Geräte- und Renderkontext verwendet werden, während die Ausmaße bei ATI auf einen real existierenden Renderkontext pro GPU angewendet werden.

Zur Aufteilung der Zeichenfläche wird auf die dazugehörige Skizze verwiesen. Die genaue Anzahl der Teilungen lässt sich mit dieser Formel bestimmen:

$$n = Number\ of\ Cores$$

$$height\ cores = \mathbb{N}(\sqrt{n}) \qquad (1.1)$$

$$width\ cores = \mathbb{N}\left(\frac{height\ cores}{n}\right) \qquad (1.2)$$

Formel 3 Berechnung der Aufteilungsanzahl der GPUs

Dabei ist die Formel insofern assoziativ, dass man die Anzahl der Breiten- und Höhenkerne in der Formel vertauschen kann. Jedoch bietet die dargestellte Formel ein wohlgeformtes Erscheinungsbild (siehe Skizze). Die Formel ist nur auf gerade Anzahl von Formeln ausgelegt. Eine ungerade Anzahl (zum Beispiel „3“) würde einen brachliegenden Kern bedeuten. Auch wenn ungerade Anzahl von Kernen eher ungewöhnlich sind und auch „non-power-of-two“ Anzahlen nur sehr selten anzutreffen sind, so sollte die Möglichkeit nicht außer Acht gelassen werden. Eine mögliche Lösung für den übrigen Kern bei ungeraden Anzahlen wäre eine zusätzliche Anti-Aliasing-Stufe sowie die Auslagerung anderer Nachbearbeitungseffekte auf den Kern.

Darauf folgend werden die Renderkontexte (bei AMD/ATI) bzw. der Gerätekontext und Renderkontext mit Zugehörigkeitsmaske (NVIDIA) erstellt.

Dies wird hierbei anhand der AMD/ATI Funktionen verdeutlicht.

```
HGLRC MainContext;
//define a maximum number of supported GPUs
#define MAX_GPU 8
int gpuids[MAX_GPU];
HGLRC gpu_rc[MAX_GPU];
int gpu_num = wglGetGPUIDsAMD(MAX_GPU,gpuids);
for(int i=0;i<gpu_num;i++)
{
	//for each available GPU: create a rendering context
	gpu_rc[i]=wglCreateAssociatedContextAMD(gpuids[i]);
}
```

Danach sind die Zeichenflächen vorhanden und die Renderschleife kann beginnen. Dabei werden in jedem Renderdurchlauf die Renderkontexte durchlaufen und nacheinander aktualisiert. Nach jedem Kontext wird der geschriebene Inhalt auf die Gesamtzeichenfläche gebracht.

```
//Main rendering Loop
while(!(GetAsyncKeyState(VK_ESCAPE)))
{
	//iterate over the field of available GPUs
	for(int i=0;i<gpu_num;i++)
	{
		wglMakeAssociatedContextCurrentAMD(gpu_rc[i]);
		//Call the function that renders the scene
		Render();
		//Blit present context to global main context to display
		wglBlitContextFramebufferAMD(MainContext, 0,0, width, height,
		GPU_Rect[i].x, GPU_Rect[i].y, GPU_Rect[i].cx, GPU_Rect[i].cy, NULL, NULL);
	}
}
```

5.1.2. Bump Mapping

Die Technologie des Bump Mappings bezieht sich auf ein jeweiliges, dargestelltes Objekt. Daher beginnt die Implementation in der OGL_FIGURE-Klasse. Dabei kann bei der Erstellung eines Objekts nicht auf die vordefinierten OpenGL Utility Toolbox zurückgegriffen werden, da der Zugriff auf die einzelnen Oberflächen-Objektdaten nicht möglich ist. Diese werden jedoch zur Generierung der Oberflächentangenten und zur Erstellung der TBN-Matrix benötigt (siehe Kapitel 2 Abschnitt 3 Abs. 2). Folglich muss das Objekt, in diesem Fall der Würfel, manuell generiert werden.

Es folgt der Code-Ausschnitt mit der Koordinatengenerierung. Die Punktdaten haben dabei folgende Syntax:

Punkt: { Vertex_x, Vertex_y, Vertex_z, Textur_s, Textur_t, Normale_x, Normale_y, Normale_z, Tangente_x, Tangente_y, Tangente_z, Orientierung }

```
Vertex g_cube[24] =
{
// Positive Z Face.
{ -1.0f, -1.0f,  1.0f,  0.0f, 0.0f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f, -1.0f,  1.0f,  1.0f, 0.0f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f,  1.0f,  1.0f, 1.0f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f,  1.0f,  1.0f,  0.0f, 1.0f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f, 0.0f, 0.0f},

// Negative Z Face.
{  1.0f, -1.0f, -1.0f,  0.0f, 0.0f,  0.0f, 0.0f, -1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f, -1.0f, -1.0f,  1.0f, 0.0f,  0.0f, 0.0f, -1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f,  1.0f, -1.0f,  1.0f, 1.0f,  0.0f, 0.0f, -1.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f, -1.0f,  0.0f, 1.0f,  0.0f, 0.0f, -1.0f,  0.0f, 0.0f, 0.0f, 0.0f},

// Positive Y Face.
{ -1.0f,  1.0f,  1.0f,  0.0f, 0.0f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f,  1.0f,  1.0f, 0.0f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f, -1.0f,  1.0f, 1.0f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f,  1.0f, -1.0f,  0.0f, 1.0f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},

// Negative Y Face.
{ -1.0f, -1.0f, -1.0f,  0.0f, 0.0f,  0.0f, -1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f, -1.0f, -1.0f,  1.0f, 0.0f,  0.0f, -1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f, -1.0f,  1.0f,  1.0f, 1.0f,  0.0f, -1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f, -1.0f,  1.0f,  0.0f, 1.0f,  0.0f, -1.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},

// Positive X Face.
{  1.0f, -1.0f,  1.0f,  0.0f, 0.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f, -1.0f, -1.0f,  1.0f, 0.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f, -1.0f,  1.0f, 1.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{  1.0f,  1.0f,  1.0f,  0.0f, 1.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},

// Negative X Face.
{ -1.0f, -1.0f, -1.0f,  0.0f, 0.0f,  -1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f, -1.0f,  1.0f,  1.0f, 0.0f,  -1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f,  1.0f,  1.0f,  1.0f, 1.0f,  -1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f},
{ -1.0f,  1.0f, -1.0f,  0.0f, 1.0f,  -1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.0f, 0.0f}
};
```

Die Flächennormalen können anhand der Vertices ebenfalls berechnet werden. Eine Funktion zur Generierung der Normalen für beliebige polygonale Flächen wurde implementiert.

Anhand der Vertices, Texturkoordinaten und Flächennormalen können folgend die Tangenten berechnet werden. Die dazu gehörige Funktion wird nachfolgend präsentiert. Dabei kommen zur Berechnung Operationen der MathLib zum Einsatz.

```
void CalcTangentVector(const float pos1[3], const float pos2[3],
                       const float pos3[3], const float texCoord1[2],
                       const float texCoord2[2], const float texCoord3[2],
                       const float normal[3], float tangent[4])
{
    // Given the 3 vertices (position and texture coordinates) of a triangle
    // calculate and return the triangle's tangent vector.

    // Create 2 vectors in object space.
    Vector3 edge1(pos2[0] - pos1[0], pos2[1] - pos1[1], pos2[2] - pos1[2]);
    Vector3 edge2(pos3[0] - pos1[0], pos3[1] - pos1[1], pos3[2] - pos1[2]);

    edge1.normalize();
    edge2.normalize();

    // Create 2 vectors in tangent (texture) space that point in the same
    // direction as edge1 and edge2 (in object space).
    Vector2 texEdge1(texCoord2[0] - texCoord1[0], texCoord2[1] - texCoord1[1]);
    Vector2 texEdge2(texCoord3[0] - texCoord1[0], texCoord3[1] - texCoord1[1]);

    texEdge1.normalize();
    texEdge2.normalize();

    Vector3 t;
    Vector3 b;
    Vector3 n(normal[0], normal[1], normal[2]);

    float det = (texEdge1.x * texEdge2.y) - (texEdge1.y * texEdge2.x);

    if (Math::closeEnough(det, 0.0f))
    {
        t.set(1.0f, 0.0f, 0.0f);
        b.set(0.0f, 1.0f, 0.0f);
    }
    else
    {
        det = 1.0f / det;

        t.x = (texEdge2.y * edge1.x - texEdge1.y * edge2.x) * det;
        t.y = (texEdge2.y * edge1.y - texEdge1.y * edge2.y) * det;
        t.z = (texEdge2.y * edge1.z - texEdge1.y * edge2.z) * det;

        b.x = (-texEdge2.x * edge1.x + texEdge1.x * edge2.x) * det;
        b.y = (-texEdge2.x * edge1.y + texEdge1.x * edge2.y) * det;
        b.z = (-texEdge2.x * edge1.z + texEdge1.x * edge2.z) * det;

        t.normalize();
        b.normalize();
    }
```

```
    // Calculate the handedness of the local tangent space.
    // The bitangent vector is the cross product between the triangle face
    // normal vector and the calculated tangent vector. The resulting bitangent
    // vector should be the same as the bitangent vector calculated from the
    // set of linear equations above. If they point in different directions
    // then we need to invert the cross product calculated bitangent vector. We
    // store this scalar multiplier in the tangent vector's 'w' component so
    // that the correct bitangent vector can be generated in the normal mapping
    // shader's vertex shader.

    Vector3 bitangent = Vector3::cross(n, t);
    float handedness = (Vector3::dot(bitangent, b) < 0.0f) ? -1.0f : 1.0f;

    tangent[0] = t.x;
    tangent[1] = t.y;
    tangent[2] = t.z;
    tangent[3] = handedness;
}
```

Die Daten werden in den dafür vorgesehenen Plätzen des Vertex Arrays abgespeichert. Anschließend wird das Array im Grafikspeicher als Vertex Array Object (VBO) gespeichert.

```
glGenBuffers(1,&this->vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, this->vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(this->mesh[0])*(this->number_points), this->mesh,
GL_STATIC_DRAW);
```

Daraufhin wird ein neuer Shader kreiert und dessen Code, Umgebungsvariablen und die Textur geladen. Hierbei sind die erstellten Tangenten Attribute-Variablen des Shaders, der Index des Texture Buffer Objects (TBO) eine Uniform-Variable, ebenso die Höhendifferenz zwischen Oberfläche und maximaler Erhebung. Jene Funktion zur Erstellung des Shaders wird folgend als Auszug präsentiert.

```
int OGL_FIGURE::SetShaderProg(unsigned int ShaderChoice, const char* VertexProg, const
char* FragmentProg)
{
	this->shader=new OGL_SHADER();
	this->shader->InstallShaderProgramm(VertexProg, FragmentProg);

	switch(ShaderChoice)
	{
	case 0:
		//TemperatureShader
	case 1:
		//BumpShader - SnakeLeather
	case 2:
		{
		//Demo 1 - Cube with HS logo emboss splitting into triangles OnClick
		tangents=(GLfloat*)malloc(sizeof(GLfloat)*(this->number_points)*4);
		for(int j=0; j<(this->number_points*4); j+=4)
		{
			tangents[j]=this->mesh[j].tangent[0];
			tangents[j+1]=this->mesh[j].tangent[1];
			tangents[j+2]=this->mesh[j].tangent[2];
			tangents[j+3]=this->mesh[j].tangent[3];
		}
		this->shader->AddAttributeVariable("tangent",4,tangents);
		GLfloat bsize=16.0f;
		OGL_TEXTURE* logo= new OGL_TEXTURE();
		logo->LoadBMP("logo_normal.bmp");
		glBindTexture(GL_TEXTURE_2D, logo->GetTextureID());
		glActiveTexture(logo->GetTextureID());

		//Add Function to Shader "AddUniformTexture"
		this->shader->AddUniformTexture("height", 0);
		this->shader->AddUniformVariable("BumpSize",1,&bsize);
		//this->EnableShaderProgram();
		break;
		}
	}
	return 1;
}
```

Nach der Erstellung des Shaders müssen die Attribute-Variablen in Form von Feldzeigern an den Shader übergeben werden. Da sich diese Variablen mit dem Renderzyklus ändern können geschieht dies in der Zeichnen-Funktion.

```
//Display Model
//Use Memory Array Objects
glClientActiveTexture(GL_TEXTURE0);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, this->vertex_buffer);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vertex), this->mesh[0].pos);
glClientActiveTexture(GL_TEXTURE0);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, GL_FLOAT, sizeof(Vertex), this->mesh[0].texcoord);*/
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, sizeof(Vertex), this->mesh[0].normal);
```

```
//Start building polygonial model
glBegin(GL_TRIANGLES);
//loop over all triangles of the model
for(int m=0; m<(this->number_points); m++)
{
	//Set Normal Vector
	glNormal3fv(this->mesh[m].normal);
	//Set Texture Coordinates
	glTexCoord2fv(this->mesh[m].texcoord);
	//if shader is activated …
	if((this->shader_enabled==true)&&(this->shader_used==true))
	{
		this->shader->UseShader();
		int number_per_vertex; GLuint ID; GLfloat* data;
		//Loop over all vailable attributes
		for(k=0; k<(this->shader->Attributes.size()); k++)
		{
			//use tangents always as first attribute
			if(k==0)
			{
				this->TransformTangents(k);
			}
			glEnableVertexAttribArray(this->shader->GetAttributeVariableID(k));
		//determine the number of float values of a specific attribute
		number_per_vertex=this->shader->GetAttributeFloatNumbersPerVertex(k);
			ID=this->shader->GetAttributeVariableID(k);
			data=this->shader->Attributes.at(k);
			//Build up Vertex Attribute based on the number of float values it
			//contains
			switch(number_per_vertex)
			{
			case 1:
				{
					glVertexAttrib1fv(ID,&data[m]);
					break;
				}
			case 2:
				{
					glVertexAttrib2fv(ID,&data[m*2]);
					break;
				}
			case 3:
				{
					glVertexAttrib3fv(ID,&data[m*3]);
					break;
				}
			default:
				{
					glVertexAttrib4fv(ID,&data[m*4]);
					break;
				}
			}
		}
	}
	//Call of actual vertex position ends the attribute process of one vertex
	glVertex3fv(this->mesh[m].pos);
}
glEnd();
//Disable the Memory Buffer Objects
glDisableClientState(GL_NORMAL_ARRAY);

glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Einzig die Erstellung eines Shader sollte hierbei noch Erwähnung finden. Dabei wird der Shader-Quellcode aus einer Textdatei in eine Zeichenkette gelesen. Dieser, in der Zeichenkette enthaltene String, wird durch den Compiler auf syntaktische und funktionale Richtigkeit überprüft. Darauf folgend werden Vertex- und Fragment Shader einem Shader-Programm zugeordnet. Dieses wird durch den Linker in ein ausführbares Programm auf der Grafikkarte überführt. Bei dieser manuellen Kompilierung ist auf eine präzise Fehlerabfrage zu achten, da unsachgemäße Handhabung der Shader zum Absturz des Programms führen kann.

```
int OGL_SHADER::InstallShaderProgramm(const char* Vertex, const char* Fragment)
{
	this->FragmentCode=(GLchar*)malloc((strlen(Fragment)+2)*sizeof(GLchar));
	this->VertexCode=(GLchar*)malloc((strlen(Vertex)+2)*sizeof(GLchar));
	strncpy_s(this->FragmentCode, strlen(this->FragmentCode), Fragment,
strlen(Fragment));
	strncpy_s(this->VertexCode, strlen(this->VertexCode), Vertex, strlen(Vertex));

	//load source code out of the files
	FILE* vertexfile;
	FILE* fragmentfile;
	char* file1; char* file2;
	long len1, len2;

	char* bad="}";

	fopen_s(&vertexfile, Vertex, "r");
	fseek(vertexfile, 0, SEEK_END);
	len1=ftell(vertexfile);
	fseek(vertexfile, 0, SEEK_SET);
	fopen_s(&fragmentfile, Fragment, "r");
	fseek(fragmentfile, 0, SEEK_END);
	len2=ftell(fragmentfile);
	fseek(fragmentfile, 0, SEEK_SET);

	file1=(char*)malloc(sizeof(char)*len1);
	file2=(char*)malloc(sizeof(char)*len2);

	unsigned int read1=(unsigned int)fread(file1, sizeof(char), len1, vertexfile);
	unsigned int read2=(unsigned int)fread(file2, sizeof(char), len2, fragmentfile);

	if(file1[read1-1]!=*bad)
	{
		file1[read1-1]='\0';
	}
	else
	{
		file1[read1]='\0';
	}
	if(file2[read2-1]!=*bad)
	{
		file2[read2-1]='\0';
	}
	else
	{
		file2[read2]='\0';
	}

	fclose(vertexfile);
	fclose(fragmentfile);
	//data loaded

	const char * vv = file1;
	const char * ff = file2;

	this->VertexShader=glCreateShader(GL_VERTEX_SHADER);
	this->FragmentShader=glCreateShader(GL_FRAGMENT_SHADER);

	glShaderSource(this->VertexShader, 1, &vv, NULL);
	glShaderSource(this->FragmentShader, 1, &ff, NULL);

	free(file1); free(file2);

	int infologlen;
	glCompileShader(this->VertexShader);
	printOpenGLError();
	glGetShaderiv(this->VertexShader, GL_COMPILE_STATUS, &(this-
>VertexCompileStatus));
```

```
        fprintf_s(this->log, "%d",this->VertexCompileStatus);
        fseek(this->log, 0, SEEK_END);
        char* InfoLog1;
        glGetShaderiv(this->VertexShader, GL_INFO_LOG_LENGTH, &infologlen);
        if(infologlen>0)
        {
                InfoLog1=(char*)malloc(infologlen*sizeof(char));
                glGetShaderInfoLog(this->VertexShader, infologlen, NULL, InfoLog1);
                fprintf_s(this->log, "%s\n\n", InfoLog1);
                fseek(this->log, 0, SEEK_END);
        }
        printShaderInfoLog(this->VertexShader);

        glCompileShader(this->FragmentShader);
        printOpenGLError();
        glGetShaderiv(this->FragmentShader, GL_COMPILE_STATUS, &(this-
>FragmentCompileStatus));
        fprintf_s(this->log, "%d",this->FragmentCompileStatus);
        fseek(this->log, 0, SEEK_END);
        char* InfoLog2;
        glGetShaderiv(this->FragmentShader, GL_INFO_LOG_LENGTH, &infologlen);
        if(infologlen>0)
        {
                InfoLog2=(char*)malloc(infologlen*sizeof(char));
                glGetShaderInfoLog(this->FragmentShader, infologlen, NULL, InfoLog2);
                fprintf_s(this->log, "%s\n\n", InfoLog2);
                fseek(this->log, 0, SEEK_END);
        }
        printShaderInfoLog(this->FragmentShader);

        if(!(this->VertexCompileStatus) || !(this->FragmentCompileStatus))
                return 0;

        this->Programm=glCreateProgram();
        glAttachShader(this->Programm, this->VertexShader);
        glAttachShader(this->Programm, this->FragmentShader);

        glLinkProgram(this->Programm);
        printOpenGLError();
        glGetProgramiv(this->Programm, GL_LINK_STATUS, &(this->LinkerStatus));
        fprintf_s(this->log, "%d",this->LinkerStatus);
        fseek(this->log, 0, SEEK_END);
        char* InfoLog3;
        glGetProgramiv(this->Programm, GL_INFO_LOG_LENGTH, &infologlen);
        if(infologlen>0)
        {
                InfoLog3=(char*)malloc(infologlen*sizeof(char));
                glGetProgramInfoLog(this->Programm, infologlen, NULL, InfoLog3);
                fprintf_s(this->log, "%s\n\n", InfoLog3);
                fseek(this->log, 0, SEEK_END);
        }
        printProgramInfoLog(this->Programm);

        if(!(this->LinkerStatus))
                return 0;

        glUseProgram(this->Programm);

        free(InfoLog1); free(InfoLog2); free(InfoLog3);
        fclose(this->log);
        return 1;
}
```

Zum Gesamtverständnis fehlen nun einzig noch die Erklärungen für Vertex- und Fragment Shader. Dabei sei darauf hingewiesen, dass der Vertex Shader die Eckpunkt-Daten wie Position, Normale und Texturkoordinaten den jeweiligen Shader-Variablen übergibt. Diese werden durch den Rasterisierer interpoliert. Die interpolierten Werte stehen dann im Fragment Shader zur Auswertung des Lichtmodells (Per-Pixel Lighting) mit angepassten lokalen Höhen im Objektkoordinatensystem zur Verfügung. Durch die Lichtberechnung werden die Fragmente des Bildes gefärbt. Das entstehende Bild wird dann auf den Bildschirm ausgegeben.

Vertex Shader:

```
varying vec3 EyeDirection;
varying vec3 LightDirection;

attribute vec4 tangent;

void main()
{
	//Vector in direction to eye point
	EyeDirection = vec3(gl_ModelViewMatrix*gl_Vertex);
	//Build original vertex position
	gl_Position = ftransform();
	//Build Vertex Coordinates
	gl_TexCoord[0] = gl_MultiTexCoord0;

	// 3 times 3-column vector for TBN Matrix
	vec3 n = normalize(gl_NormalMatrix*gl_Normal);
	vec3 t = gl_NormalMatrix*tangent.xyz;
	vec3 b = cross(n,t);

	vec3 v;
	//Matrix Multiplication with TBN Matrix for direction vertex-light in T- space
	v.x = dot((gl_LightSource[0].position.xyz-EyeDirection),t);
	v.y = dot((gl_LightSource[0].position.xyz-EyeDirection),b);
	v.z = dot((gl_LightSource[0].position.xyz-EyeDirection),n);
	LightDirection = normalize(v);

	//new vector from vertex point to eye point in tangent space
	v.x = dot(-EyeDirection,t);
	v.y = dot(-EyeDirection,b);
	v.z = dot(-EyeDirection,n);
	EyeDirection = normalize(v);
}
```

Fragment Shader:

```
varying vec3 LightDirection;
varying vec3 EyeDirection;
uniform float BumpSize;
uniform sampler2D height;
vec4 ComputeColors(float diffuse, float specular);

void main()
{
	vec2 p=vec2(texture2D(height, gl_TexCoord[0].st));
	//compute occasion and height of a “bump” (height difference)
	float d, f;
	d = p.x*p.x + p.y*p.y;
	f=1.0 / sqrt(d+1.0);
	//if there is no Bump: height elevation is one, else there is a difference
	if(d>BumpSize)
	{
		p = vec2(0.0);
		f = 1.0;
	}
	//set new normal vector with scaled height
	vec3 normDelta = vec3(p.x, p.y, 1.0)*f;
	//precompute inputs for lighting equation for diffuse, reflected and specular part
	float diffuse = max(dot(normDelta, LightDirection), 0.0);
	vec3 R = reflect(LightDirection, normDelta);
	float specular = max(dot(EyeDirection,R),0.0);
	gl_FragColor = ComputeColors(diffuse, specular);
}
vec4 ComputeColors(float diffuse, float specular)
{
	vec4 LitColor;
	specular = pow(specular, 0.3*gl_FrontMaterial.shininess);
	vec4 ambient_color = gl_FrontLightProduct[0].ambient;
	vec4 diffuse_color = diffuse*(gl_FrontLightProduct[0].diffuse);
	vec4 specular_color = specular*(gl_FrontLightProduct[0].specular);
	LitColor =
gl_FrontLightModelProduct.sceneColor+ambient_color+diffuse_color+specular_color;
	return LitColor;
}
```

Als Höhenkarte für die Basis des Bump Mappings wird das Logo der Hochschule Wismar verwendet. Es handelt sich dabei um eine vom Normal Map Generator umgewandelte Version des Logos. Dabei kennzeichnen Flächen mit einem hohen Rotanteil steigende Kanten und Flächen mit wenig Rotanteil sinkende Kanten des Bildes. Demzufolge kann eine Bump Map durch das Kantenbild einer Grafik erzeugt werden.

Bild 41 Originalbild Logo HS Wismar EuI [HS Wismar]

Bild 42 Bump Map Logo HS Wismar EuI

5.2. Testsystem Physikberechnung

Aufgrund des bereits erörterten Problems, dass zum Zeitpunkt der Entwicklung diverse Debugging-, Kompilier- und Entwicklungswerkzeuge fehlten musste für die verteilte, GPU-basierte Physikengine ein Weg gefunden werden, eine generelle, möglichst einheitliche Schnittstelle für verschiedene Technologien zu schaffen. Diese Schnittstelle soll durch Funktionen ermitteln, welche Technologie zur Verfügung steht und die performanteste davon wählen. Durch dieses mehrstufige Ausschlussverfahren kann ebenfalls sichergestellt werden, dass bei fehlender GPU-Technologie auch weiterhin auf einer ähnlichen Architektur wie im GPU-Teil die Physik vom Hauptprozessor berechnet wird.

Ein Test eines solch kritischen, schwierig analysierenden Systems kann nicht in der schlussendlichen Laufumgebung geschehen. Dabei existieren zu viele Variablen. Für einen aussagekräftigen Test sollte das jeweilige System aus dem Gesamtprojekt herausgezogen werden. Dabei sinkt die Komplexität und die möglichen Fehlerquellen werden minimiert.

Als Basis wurde der Multi-GPU Ansatz von CUDA verfeinert und der einstufige Kernel in einen vierstufigen Kernel nach den im Entwurf erläuterten Formeln umgestaltet worden. Im Anschluss wurden OpenCL-Schnittstelle sowie OpenCL-Kernel an dieses Schema angepasst.

5.2.1. x64-Portierung

Die Programmierung von CUDA wie auch OpenCL basiert auf Software Development Kits (SDKs) für die jeweilige Umgebung der Entwicklungsplattform. Dies birgt einige Nachteile in der Entwicklung portabler GPGPU-Applikationen.

Für Plattformen mit NVIDIA Grafikchips wird die Unterstützung von CUDA, OpenCL und DirectCompute durch das NVIDIA Computing SDK gewährleistet. Auf AMD/ATI Plattformen erfolgt die Unterstützung durch das ATI Stream SDK. Durch das größere OpenSource-Engagement werden neuere OpenCL-Funktionen hauptsächlich von seitens AMD/ATI veröffentlich und folgend von NVIDIA angepasst. Durch die unterschiedliche Unterstützung von OpenCL bei den Grafikchip-Herstellern ergeben sich Probleme in der Portabilität. Funktionsdeklarationen werden auf ATI-Karten teilweise als „veraltet“ gekennzeichnet, wobei die alte Syntax währenddessen von NVIDIAs Treibern unterstützt wird. Dies erschwert die herstellerunabhängige Entwicklung.

Desweiteren ist nur jeweils eine Version auf dem Entwicklungsrechner installierbar. Bei einem 64-Bit Betriebssystem des Entwicklungsrechners ist die Installation der 64-Bit Version

des SDKs sinnvoll. Bei dieser sind dann jedoch keine Entwicklungsbibliotheken für 32-Bit mitgeliefert (Ausnahme: Emulationsmodus). Daher müssten für eine OpenCL-Applikation pro unterstütztes Betriebssystem zwei getrennte Betriebssystemversionen, 32- und 64-Bit, installiert sein. Demnach muss das Projekt in 32- und 64-Bit Version geteilt werden. Dies wurde für eqOSG zu erst im Testsystem und darauf folgend im Endnutzersystem durchgeführt.

Die Reihenfolge zur Einstellung einer 64-Bit Kompilierregel ist als Anhang auf der Entwicklungs-DVD zu sehen.

In den Projekteinstellungen werden spezielle Makros für 32- und/oder 64-Bit geschaffen. Ein manueller, plattformunabhängiger Bibliotheksimport sieht folgendermaßen aus.

```
#ifdef _WIN32
	#ifdef _DEBUG
		//32-Bit Debug OpenCL Utility Library
		#pragma comment(lib,“oclUtils32D“)
		//32-Bit Debug OpenCL Shell Logging and Profiling Library
		#pragma comment(lib,“shrUtils32D”)
		//32-Bit Debug OpenCL Library
		#pragma comment(lib,“OpenCL”)
	#endif
	#ifdef NDEBUG
		//32-Bit Debug OpenCL Utility Library
		#pragma comment(lib,”oclUtils32”)
		//32-Bit Debug OpenCL Shell Logging and Profiling Library
		#pragma comment(lib,”shrUtils32”)
		//32-Bit Debug OpenCL Library
		#pragma comment(lib,”OpenCL”)
	#endif
#elif _X64
	#ifdef _DEBUG
		//32-Bit Debug OpenCL Utility Library
		#pragma comment(lib,“oclUtils64D“)
		//32-Bit Debug OpenCL Shell Logging and Profiling Library
		#pragma comment(lib,“shrUtils64D”)
		//32-Bit Debug OpenCL Library
		#pragma comment(lib,“OpenCL”)
	#endif
	#ifdef NDEBUG
		//64-Bit Debug OpenCL Utility Library
		#pragma comment(lib,”oclUtils64”)
		//64-Bit Debug OpenCL Shell Logging and Profiling Library
		#pragma comment(lib,”shrUtils64”)
		//64-Bit Debug OpenCL Library
		#pragma comment(lib,”OpenCL”)
	#endif
#endif
```

Aufgrund der Betriebssystemabhängigkeit der GPGPU-Technologien werden diese nur in der 64 Bit-Version genutzt. Die 32-Bit Version berechnet die Physik auf der CPU.

5.2.2. Testprogramm

Im Testsystem wird die Physikengine mit einem definierten Datensatz aufgerufen. Dabei wird die gleiche Physikengine wie im darauf folgenden Endnutzersystem angewendet.

Beim Testsystem handelt es sich dabei um einen BlackBox-Test. Es bestehen Eingangswerte in Form einer mittleren Umgebungstemperatur und Luftfeuchtigkeit. Darauf aufbauend werden zweidimensionale Felder von Werten erzeugt, die mit einem zufälligen Wert in einem bestimmten Bereich um den Mittelwert pendeln. Diese werden durch ein unbekanntes System verarbeitet. Das System ist unbekannt, da durch die etlichen Verzweigungen und den fehlenden Debugger nie bestimmt werden kann, welche Verarbeitungssequenz schlussendlich ausgeführt wird. Durch eine manuelle Berechnung des korrespondierenden Mittelwertes der Windgeschwindigkeit existiert ein Referenzpunkt für die Ausgabe. Daher ist ein gewisser Bereich als Erwartungswert für das endgültige Ausgabefeld gegeben. Durch den Vergleich von Erwartungswert und Ausgabewerten kann beurteilt werden, ob das System funktioniert.

Bei der Ausführung werden alle Logging-Optionen der Schnittstellen sowie ein eigenes Logsystem für die Ausgabewerte erstellt.

Es folgt der Code der ersten Version des Testprogramms.

```
#include "stdafx.h"
#include "PhysiX.h"
#include "Log.h"

#define GSIZE 32

int _tmain(int argc, _TCHAR* argv[])
{
	CLog myLog;
	myLog.SetLogFile("MT_Test.txt");
	myLog.OpenLog();
	int i,j;
	PhysiX* myPhysics = new PhysiX();
	std::vector<float> velocity;
	std::vector<float> wind_direction;
	myPhysics->ComputeAirVelocityMps(GSIZE,GSIZE,45.0,0.15,&velocity,&wind_direction);
	int n=GSIZE*GSIZE;

	std::string Ausgabe;
	for(i=0;i<GSIZE;i++)
	{
		for(j=0;j<GSIZE;j++)
		{
			Ausgabe="";
			char me[12];
			sprintf(me,"%f ",velocity[(i*GSIZE)+j]);
			Ausgabe.append(me);
			myLog.PumpMessage(Ausgabe);
		}
		Ausgabe="\n";
		myLog.PumpMessage(Ausgabe);
	}

	Ausgabe="\n\n";
	myLog.PumpMessage(Ausgabe);
	for(i=0;i<GSIZE;i++)
	{
		for(j=0;j<GSIZE;j++)
		{
			Ausgabe="";
			char me[12];
			sprintf(me,"%f ",wind_direction[(i*GSIZE)+j]);
			Ausgabe.append(me);
			myLog.PumpMessage(Ausgabe);
		}
		Ausgabe="\n";
		myLog.PumpMessage(Ausgabe);
	}
```

```
        //free memory space
        velocity.clear();
        wind_direction.clear();
        Ausgabe="Size of resting memory (velocity): ";
        char me[128];
        sprintf(me,"%d",(int)(sizeof(velocity)));
        Ausgabe.append(me);
        Ausgabe.append("\nSize of resting memory (wind_direction): ");
        sprintf(me,"%d",(int)(sizeof(wind_direction)));
        Ausgabe.append(me);
        myLog.PumpMessage(Ausgabe);
        myLog.WriteLog();
        myLog.CloseLog();
        //free(velocity);
        return 0;
}
```

Die ersten Testläufe haben gezeigt, dass eine solch komplexe Berechnung auf der GPU ohne Debugger und Trace-Modus nicht sicher und stabil zur Ausführung gebracht werden kann. Ergebnisse sind bei OpenCL:

- INAN0-Feld für Windgeschwindigkeiten (Zahlenbereichsüberlauf oder fehlerhafte Datentypen während Berechnung)

Berechnungen mit CUDA ergeben ein Null-Werte-Feld.

Als Reaktion auf diese Ergebnisse wird die Windberechnung vereinfach. Die neue Berechnung sieht einen Basiswert für die Geschwindigkeit vor. Ausgehend von diesem Wert wird ein Feld aus zufälligen Windgeschwindigkeiten um den Basiswert generiert. Auf den Grafikkernen wird der jeweilige Wert mit einem Multiplikator versehen, der in der Endnutzeranwendung durch Tastendruck manipuliert werden kann. Das Endergebnis des Windgeschwindigkeitsfeldes wird darauf folgend zurückgegeben.

Nach erfolgreichen Testläufen ist die Physikengine in genau der gleichen Form im Endsystem übernommen worden. Hierbei treten jedoch Speicherfehler aufgrund von Fehlern in Zeigerarithmetik und IPC auf. Zur Sicherstellung der Stabilität über einen längeren Zeitraum wird das Testsystem verfeinert. Durch die Iteration der Berechnung über 120 Schritte konnten alle Speicherfehler beseitigt werden. Zur Referenzmessungen wurde die Abarbeitungszeit über die 120 Iterationen gemessen.

Folgend werden Änderungen zur neuen Testversion aufgezeigt.

```
[…]
                float mod=1.0f;
                CTimer_Count myTime;
                myTime.StartTimer();
                CLog myLog;
                myLog.SetLogFile("MT_Test.txt");
                myLog.OpenLog();
                int i,j;
                PhysiX* myPhysics = new PhysiX();
                std::vector<float> velocity;
```

```
                for(int k=0;k<120;k++)
                {
                        velocity.clear();
                        myPhysics->ComputeAirVelocityMps(GSIZE,GSIZE,20.65,mod,&velocity);
                        mod+=0.1;
                }
                int n=GSIZE*GSIZE;
[…]
                //free memory space
                velocity.clear();

                Ausgabe="Size of resting memory (velocity): ";
                char me[156];
                sprintf(me,"%d",(int)(sizeof(velocity)));
                Ausgabe.append(me);

                myTime.EndTimer();
                Ausgabe="\nComputation Time: ";
                sprintf(me,"%f \n",myTime.GetMilliSeconds());
                Ausgabe.append(me);
                myLog.PumpMessage(Ausgabe);
                myLog.WriteLog();
                myLog.CloseLog();
                velocity.clear();
[…]
```

5.2.3. Schlussfolgerung

Die Ergebnisse des Testsystems lassen einige Rückschlüsse auf die allgemeine Entwicklung mit GPGPU-Schnittstellen sowie die weitere Entwicklung und Realisierung des Endnutzersystems eqOSG zu.

Zur allgemeinen Entwicklung mit GPGPU-Schnittstellen ist festzustellen, dass größere Berechnungsalgorithmen nur mit Einbußen in der Stabilität realisierbar sind. Dies ist durch den unterschiedlichen Befehlssatz der Compute Shader-Programmiersprachen begründet.

Es treten massive Unterschiede in folgenden Bereichen auf:

- einfache- und doppelte Fließkommazahlendarstellung
- Typkonvertierungen
- Automatisch verwendete Typen
- Unterstützung von ausdrücklicher einfacher Fließkommadarstellung (z.Bsp. 0.1f)
- Spezielle, Chip-spezifische Funktionen
- Softwarearchitektur

Eine Compiler-Regel fängt syntaktische Fehler ab. Jedoch fehlt bei jenen hochgradig parallelen Systemen ein echter Debugger mit Trace-Modus, um Fehler einzugrenzen. Ein Testsystem ist bei Fehlen einer geeigneten Entwicklungsumgebung für GPGPU-Technologien unbedingt nötig, lässt jedoch auch nur Kernel mit geringer Komplexität zu. Mit einer geeigneten IDE sind komplexere Kernels sowie mehrstufige Kernels in einer Datei möglich. Dies reduziert den Ladeaufwand neuer Kernels durch die Reduzierung des Festplattenzugriffs. Desweiteren ist es in einem Team von Programmierern sinnvoll, den zuständigen Programmierer für die anderen Shader mit solider Kenntnis von

Mikrocontrollern an den Kernels arbeiten zu lassen. Abgesehen von oben genannten Punkten sind große Ähnlichkeiten mit Vertex- und Fragment Shader-Programmen in GLSL zu erkennen.

Die gemessenen Referenzzeiten zur Abarbeitung sind im dafür vorgesehenen Kapitel dargestellt. Aufgrund der Messwerte ist zu schlussfolgern dass es einen deutlichen Geschwindigkeitszuwachs durch die Nutzung der GPGPU-Technologien geben wird. In Hinblick auf die großen Unterschiede der OpenCL-Messungen mit globalem und lokalem Kontext können etwaige Abweichungen von dieser Prognose bezüglich CUDA an dem ständig lokalen CUDA-Kontext liegen, welcher mit jedem Durchgang neu initialisiert wird. Desweiteren muss für das Endnutzersystem jegliches direktes Logging in Datei unterbunden werden, da dies die Performance aus übertragungstechnischen Gründen drastisch reduziert.

5.3. Equalizer und Open Scene Graph

Durch die gewonnenen Erkenntnisse aus Basislösung und Testsystem wird in diesem Kapitel die Implementation des Hauptprogramms in kleinen Teilen vorgestellt.

Dieses System besteht aus drei wichtigen Kernpunkten. Zum einen wird die Applikation durch das Backend-Framework „Equalizer“ gesteuert. Wichtige, im Weiteren vorgestellte Eckpunkte dabei sind:

- Aufbau einer Konfiguration für eine Anwendung
- Nachrichtenwarteschlange für Nutzereingabe-Ereignisse

Durch das Backend wird das Open Scene Graph-Frontend gesteuert. In diesem sind die diversen Demos enthalten. Es ermöglicht außerdem das Laden von 3D-Modellen aus Dateien. In diesem Kapitel wird tiefer auf folgende Eckpunkte der OSG-Implementation eingegangen:

- Anwendungsauswahl
- Terraingenerierung

In einer Schnittstellenklasse zwischen Back- und Frontend befindet sich die Physikengine. Mit deren Hilfe wird das Staubpartikel-Feld in der Pyramidendemo gesteuert. Kernpunkte dieses Teilsystems, welche präsentiert werden, sind:

- Ermittlung der verfügbaren GPGPU-Technologie
- Physikberechnung

Desweiteren werden Teile des Codes vorgestellt, in welchem die Messwertermittlung erfolgt.

Für eine einfach nutzbare Windows-Anwendung ist die Erstellung einer Installationsdatei unabdingbar. Hierzu müssen Randbedingungen der Architektur (32- und 64-Bit) sowie damit verbundene Technologien beachtet werden. Die Kreation eines Windows-Installers für verschiedene Plattformen wird demnach ebenfalls erläutert.

5.3.1. Equalizer – Konfigurationsdateien

Eine Equalizer-Konfigurationsdatei enthält die Parameter zur Verteilung, Abspeicherung und Komposition der Einzelbilder der GPUs. Desweiteren kann in der Konfigurationsdatei die Auflösung des komponierten Bildes hinterlegt werden.

Der allgemeine Aufbau ist in [Eil09] Appendix A beschrieben. Folgend wird die Konfiguration anhand eines Beispiels für eine Berechnung auf zwei Grafikkarten mit SFR load balancing erläutert, bei welcher die Daten in Frame Buffer Objects gespeichert werden. Die Bilddaten werden durch 16-Bit Fließkommazahlen im RGBA-Modus repräsentiert.

```
#Equalizer 1.0 ascii

# dual pipe, two-to-one sort-first config for a dual-GPU workstation

server
{
    config
    {
        appNode
        {
            pipe
            {
                window
                {
                    viewport [ 0 0 1 1 ]
                    channel
                    {
                              name "channel"
                     }
                }
            }
            pipe
            {
                device 0
                window
                {
                    attributes
                    {
                              hint_drawable FBO
                              planes_color  RGBA16F
                    }
                    channel
                    {
                              name "buffer1"
                    }
                }
            }
            pipe
            {
                device 1
                window
                {
                    attributes
                    {
                              hint_drawable FBO
                              planes_color  RGBA16F
                    }
                    channel
                    {
                              name "buffer2"
                    }
                }
            }
        }
```

Hierbei handelt es sich um die Deklaration der Eingangsparameter. Die Daten werden durch einen Rechner (appNode) verarbeitet. Dieser besitzt drei virtuelle Pipes. Dabei wird die erste Pipe für die Ausgabe genutzt. Als Eingabe dienen die beiden physisch existierenden Pipes Zwei und Drei (namentlich „buffer 1“ und „buffer 2“) die installierten GPUs sind. Jede Pipe besitzt ein Fenster, welches gerendert werden soll. Dabei sind in den Attributen die Parameter zur Abspeicherung für Hintergrund-Speichertechnologie (hier FBOs) und Pixelrepräsentation (hier 16-Bit RGBA float-Werte) hinterlegt. Zur Adressierung bei der Komposition wird jeder Kanal dabei mit einem Namen versehen. Im Ausgabefenster sind die Dimensionen des Ausgabefensters hinterlegt. Diese können dabei normiert (Werte 0..1) oder in absoluter Form (reale Auflösung in Pixeln) vorliegen.

Es folgt der Teil zur Festlegung der Komposition.

```
        compound
        {
            channel "channel"
            load_equalizer { mode 2D }

            wall
            {
                bottom_left  [ -.32 -.20 -.75 ]
                bottom_right [  .32 -.20 -.75 ]
                top_left     [ -.32  .20 -.75 ]
            }

            compound
            {
                channel "buffer1"
                viewport [ 0 0 .5 1 ]
                outputframe { }
            }
            compound
            {
                channel "buffer2"
                viewport [ .5 0 .5 1 ]
                outputframe { }
            }
            inputframe { name "frame.buffer1" }
            inputframe { name "frame.buffer2" }
        }
    }
}
```

Bei der Komposition wird zuerst der Kanal angegeben, in welchem das komponierte Bild erscheinen soll. Daraufhin wird der Load Balancer konfiguriert. Dabei ist „mode 2D“ für den Sort-First SFR-Algorithmus vorgesehen. Mit „wall“ werden etwaige Fensterverschiebungs-Korrekturen durchgeführt. Daraufhin wird für jeden Eingabeframe ein Kompositionskanal vorgesehen. Der Eingabekanal wird anhand des vorher vergebenen Namens adressiert und mit „viewport“ der Teil der Bildinformationen des Eingabekanals festgelegt. Schlussendlich wird dieser Fensterteil als Output Frame der jeweiligen GPU markiert. Im Anschluss werden diese Frames im Ausgabekanal komponiert.

5.3.2. Nachrichtenwarteschlange

Die Nachrichtenwarteschlange für Nutzerinteraktionsereignisse ist in einer dafür vorgesehenen Funktion in der Config-Klasse des Equalizers zu finden. Dabei wird zwischen jenen verschiedenen Aktionen unterschieden:

- Taste loslassen
- Taste drücken
- Mausbewegung

Weitere Ereignisse sind in der „event.h“ des Equalizers auffindbar.

In dieser Datei existieren ebenfalls systemunabhängige Deklarationen für Spezialtasten von Tastaturen. Die Codes sind an X11-Systeme angelehnt. Für Interaktionen durch

Standardtasten müssen die jeweiligen ASCII-Dezimalcodes verwendet werden. Dies wird folgend illustriert.

```
//if "S" (capital) is pressed -> alternate statistics displayment
case 115:
        use_statistics = !(use_statistics);
        eventHandled = true;
        break;

//if "s" (non-capital) is pressed -> alternate statistics displayment
case 83:
        use_statistics = !(use_statistics);
        eventHandled = true;
        break;

//if "u" (non-capital) is pressed -> alternate status of "self- //moving" simulation
case 85:
        move_simulation = !(move_simulation);
        eventHandled = true;
        break;

//if "U" (capital) is pressed -> alternate status of "self-moving" //simulation
case 117:
        move_simulation = !(move_simulation);
        eventHandled = true;
        break;

//if "k" (non-capital) is pressed -> a press on "+" or "-" modifies //velocity
(alternating status)
case 75:
        //modify_velocity = !(modify_velocity);
        increment_velocity=false;
        decrement_velocity=false;
        eventHandled = true;
        break;

//if "K" (capital) is pressed -> a press on "+" or "-" modifies //velocity (alternating
status)
case 107:
        //modify_velocity = !(modify_velocity);
        increment_velocity=false;
        decrement_velocity=false;
        eventHandled = true;
        break;
```

5.3.3. Anwendungsauswahl

Die Auswahl, welche Anwendung (Demo) gestartet werden soll, wird über die Konsolenparameter an das Programm weitergegeben. Dabei werden die Parameter durch die TCLAP-Bibliothek geparst. Dafür gibt es eine Modus-Variable in der InitData- wie auch FrameData-Klasse, welche den jeweiligen Wert des entsprechenden Demos enthält.

Dem Wert wird darauf folgend erst in der Pipe-Klasse die entsprechende Demo zugewiesen. Dafür wird in den Initialisierungsfunktion die Variable aus den Initialisierungsdaten gelesen und intern abgespeichert. Dies sieht wie folgt aus.

```
bool Pipe::configInit( const uint32_t initID )
{
    if( !eq::Pipe::configInit( initID ) )
        return false;

    // Get InitData and map FrameData
    Config* config = static_cast<Config*>( getConfig() );
    const InitData& initData = config->getInitData();
    const uint32_t frameDataID = initData.getFrameDataID();
    // Determine use of a model file
    mUsesModel = initData.loadModel();
    // Get the model file
    mModelFile = initData.modelFileName();
    // Determine use of a special shader
        mUsesShader = initData.loadShader();
    // Get the shader file
        mShaderFile = initData.shaderFileName();
    // Determine the useof a modus
        mUsesModus = initData.loadModus();
    // Determine the used modus
        mMode = initData.Modus();
    const bool mapped = config->mapObject( &mFrameData, frameDataID );
    EQASSERT( mapped );

    // Enable 8x MSAA
        osg::DisplaySettings::instance()->setNumMultiSamples(8);

    // set the scene to render
    mViewer->setSceneData(createSceneGraph().get());
    return mapped;
}
```

Das Mapping von Wert geschieht in Erstellungs- und Update-Routine des Szenengraphens.

```
osg::ref_ptr<osg::Node> Pipe::createSceneGraph()
{
        //If a patricular modus is activated, switch between the modes and load the
appropriate applet.
        //If this is not the case, use the predefined behavior
        if(this->mUsesModus)
        {
                switch(this->mMode)
                {
                case 1:
                        {
                                //Pyramid Terrain Applet Demo
                                std::cout << "Loading Pyramid Demo" << std::endl;
                                PyramidApplet* Pyramid= new PyramidApplet(this-
>mMode,&(this->mRotateMatrix));
                                this->Applet=(void*)Pyramid;
                                std::string Output;
                                char tmbuffer[128];
                                _strtime_s(tmbuffer,128);
                                Output="Begin Mid-Processing Measurement: ";
                                Output.append(tmbuffer);
                                Output.append(" Modus: Pyramide (Animation/Cinematic)");
                                Output.append("\n");
                                midtime->StartTimer();
                                midpart->PumpMessage(Output);
                                return (Pyramid->createSceneGraph());
                                break;
                        }
                case 3:
                        {

                                // one emitter - one particle system
                                //[…]
                                //Mid-Time measurement
                                break;
                        }
                default:
                        {

                                //[…]
                                break;
                        }
                }
        }
```

Somit kann zwischen den einzelnen Demos unterschieden werden. Desweiteren ist es damit möglich die unterschiedlichen Anwendungsfälle in einem Programm zu bedienen.

5.3.4. Terraingenerierung

Die Implementierung des Terrains geschieht im Programm mit Hilfe von Open Scene Graph. Dabei ist dies ein Teil des Pyramiden-Applets. Grundidee ist dabei die Erstellung einer planaren Fläche. Ein Graustufenbild wird geladen und dient als Höhenstruktur. Nachfolgend werden an den Eckpunkten der Oberfläche Höhenmodifikationen entsprechend den Grauwerten der Textur getätigt. Schlussendlich wird das Terrain mit einem überabgetasteten Farbbild überzogen um den Eindruck einer gleichförmigen Oberfläche zu erzeugen.

```
osg::ref_ptr<osg::MatrixTransform> PyramidApplet::CreateTerrain(std::string HeightFile,
std::string ColorFile)
{
		//Ground Geometry Generation

		// Heightfield Generation based upon Heightmap
		// -------------------- Begin -----------------
		osg::Texture2D* HeightTexture = new osg::Texture2D;
		HeightTexture->setDataVariance(osg::Object::DYNAMIC);
		std::cout << "Load Heightmap Image." << std::endl;
		// Load BlackWhite-Picture as heightfield
		osg::Image* HeightImage = osgDB::readImageFile(HeightFile);
		// print out information
		std::cout << "Loading of Heightmap Image done. " << HeightImage-
>getFileName() << " Bytesize: " << HeightImage->getImageSizeInBytes() << " Extents: " <<
HeightImage->s() << "x" << HeightImage->t() << std::endl;

		unsigned int height = HeightImage->t();
		unsigned int width = HeightImage->s();

		// position the Heightfield in the scene
		osg::ref_ptr<osgTerrain::Locator> TLoc = new osgTerrain::Locator;
		TLoc->setCoordinateSystemType(osgTerrain::Locator::PROJECTED);
		TLoc->setTransformAsExtents(0.0,0.0,double(width*factor),
double(height*factor));

		osg::ref_ptr<osg::HeightField> heightField = new osg::HeightField;
		std::cout << "Heaghtfield created. Allocating Memory for Heightfield." <<
std::endl;
		heightField->allocate(width, height);
		std::cout << "Heightfield Memory allocated. Setting seet point for
Heightmap." << std::endl;
		heightField->setOrigin(osg::Vec3(0.0f,0.0f,0.0f));
		std::cout << "Seet Point set. Setting X-Interval." << std::endl;
		heightField->setXInterval(factor);
		std::cout << "X-Interval set. Setting Y-Interval." << std::endl;
		heightField->setYInterval(factor);
		std::cout << "Y-Interval set. Setting Values for Heightfield" << std::endl;
```

```
		// set Height values of the corresponding image in the plane
		int over_zero=0;
		for(int x=0;x<width;x++)
		{
			over_zero=0;
			for(int y=0;y<height;y++)
			{
				float value; int int_val;
				value=int_val=*(HeightImage->data(x,y));
				if(value!=0.0f)
					over_zero++;
				heightField->setHeight(x,y,value);
			}
		}

		std::cout << "Heightfield dimensions: " << heightField->getNumRows() << "x"
<< heightField->getNumColumns() << std::endl;

		osg::ref_ptr<osgTerrain::HeightFieldLayer> HFL = new
osgTerrain::HeightFieldLayer(heightField.get());
		HFL->setLocator(TLoc.get());

		osg::ref_ptr<osgTerrain::GeometryTechnique> GeomTech = new
osgTerrain::GeometryTechnique;
		osg::ref_ptr<osgTerrain::TerrainTile> MapTile = new
osgTerrain::TerrainTile;
		MapTile->setElevationLayer(HFL.get());
		MapTile->setTerrainTechnique(GeomTech.get());

		osg::ref_ptr<osgTerrain::Terrain> Terrain = new osgTerrain::Terrain;
		Terrain->setSampleRatio(1.0f);
		Terrain->addChild(MapTile.get());

		//Center the ground
		osg::Matrix matrix;
		osg::ref_ptr<osg::MatrixTransform> TerrainTranslation = new
osg::MatrixTransform;
		matrix.makeTranslate(osg::Vec3f(-(width*(factor/2.0f)),-
(height*(factor/2.0f)),0.0f));
		osg::Matrix rot;
		rot.makeRotate(-osg::PI_2,osg::Vec3f(1.0f,0.0f,0.0f));
		TerrainTranslation->setMatrix( matrix*rot );
		TerrainTranslation->setDataVariance( osg::Object::STATIC );
		TerrainTranslation->addChild(Terrain.get());
		// -------------------- End -------------------

		//Terrain Generation on ground
		// ----------------- Begin --------------------

		//osg::StateSet* TerrainTex = ground_group->getStateSet();

		osg::StateSet* TerrainTex = Terrain->getOrCreateStateSet();

		std::cout << "Load Colored Texture." << std::endl;
		//osg::Image* ColorImage = osgDB::readImageFile("..\\pictures\\small.bmp");
		osg::Image* ColorImage = osgDB::readImageFile(ColorFile);
		std::cout << "Loading of Colored Texture done." << std::endl;
		if(!ColorImage)
		{
			std::cout << "No colored texture available or texture damaged." <<
endl;
		}
		else
		{
			osg::ref_ptr<osgTerrain::ImageLayer> ColorLayer = new
osgTerrain::ImageLayer(ColorImage);
			MapTile->setColorLayer(0,ColorLayer.get());
		}
		//----------------- End -----------------------

		return TerrainTranslation;
}
```

5.3.5. Ermittlung der Parallelisierungstechnologie

Die parallele Verarbeitung der Physik geschieht in der selbstentworfenen PhysiX-Schnittstelle. In dessen Headerdatei sind die verschiedenen Technologien aufgeführt. Durch ein weiteres Feld wird bestimmt, ob eine jeweilige Technologie verwendet werden darf oder nicht.

```
//----------------------//
//----------------------//
//   MODE DEFINITIONS   //
//----------------------//
//----------------------//
#define VALID_VAL_NUM          3
#define CPU                    0
#define OPEN_CL                1
#define CUDA                   2
#define BROOK                  3
#define CAL_TEC                4
#define SHADER_TEX             5
#define NOT_DETERMINED         6

//----------------------//
// MGPU DEBUG VARIABLES //
//----------------------//
#define USE_CUDA               0
#define USE_OPENCL             1
#define USE_CPU                1
```

Bei Aufruf der Physikengine zur Bestimmung der Windgeschwindigkeit wird durch eine Funktion getestet, welche Technologie genutzt wird. Dabei werden Kontexte der jeweiligen Technologien erschaffen und die Anzahl der verfügbaren Kerne bestimmt. Ist die Anzahl der verfügbaren GPU-Kerne Eins oder größer, so kann diese verwendet werden. Dabei gibt es, je nach Leistungsklasse, eine bestimmte Hierarchie.

1. **CAL**
2. **CUDA**
3. **Brook**
4. **OpenCL**
5. **Shader**
6. **CPU**

Von diesen Technologien sind bisher nur CUDA, OpenCL und die CPU-Lösung implementiert. Im Folgenden wird die Auswahlfunktion vorgestellt.

```
// computes the best-fitting technology used to compute the physical values
void PhysiX::ChooseMode(void)
{
        //Check first for CUDA
        int CUDA_GPU_COUNT=0;
        cudaError_t cud_err, cud_err2;
        cud_err=cudaGetDeviceCount(&CUDA_GPU_COUNT);
        float* airflow_vmps;
        cud_err2=cudaMalloc((void**)&airflow_vmps, 32 * sizeof(float));
        if((!cud_err)&&(CUDA_GPU_COUNT>0)&&(USE_CUDA)&&(!cud_err2))
        {
                cudaFree(airflow_vmps);
                this->mode=CUDA;
                printf("CUDA chosen.\n");
                //std::cout << "CUDA chosen." << std::endl;
        }
        else
        {
                //Check if OpenCL is supported
                cl_int err;
                cl_context Context=clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL,
                NULL, &err);

                // Find out how many GPU's to compute on all available GPUs
                size_t nDeviceBytes;
                err = clGetContextInfo(Context, CL_CONTEXT_DEVICES, 0, NULL,
                &nDeviceBytes);
                shrCheckError(err, CL_SUCCESS);
                int number = (cl_uint)nDeviceBytes/sizeof(cl_device_id);
                clReleaseContext(Context);
                if(shrCheckError(err, CL_SUCCESS))
                if((number>0)&&(USE_OPENCL)&&(!err))
                {
                        // create the OpenCL context on available GPU devices
                        PresentContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                        NULL, NULL, &err);
                        shrCheckError(err, CL_SUCCESS);
                        shrLog(LOGBOTH, 0, "clCreateContextFromType\n\n");

                        // Find out how many GPU's to compute on all available GPUs
                        size_t nDeviceBytes;
                        GPU_N = 0;              //ciDeviceCount = GPU_N

                        err = clGetContextInfo(PresentContext, CL_CONTEXT_DEVICES, 0, NULL,
                        &nDeviceBytes);
                        shrCheckError(err, CL_SUCCESS);
                        GPU_N = (cl_uint)nDeviceBytes/sizeof(cl_device_id);
                        PresentProgram=OpenCLKernelCode("simpleMultiGPU.cl",
                        PresentContext);
                        this->mode=OPEN_CL;
                        //std::cout << "OpenCL chosen." << std::endl;
                        printf("OpenCL chosen.\n");
                }
                else
                {
                        //CPU Computation
                        this->mode=CPU;
                        //std::cout << "Computation done by CPU." << std::endl;
                        printf("Computation done by CPU.\n");
                }
        }
}
```

Auf Basis des abgespeicherten Modus wird im Folgenden die jeweilige Berechnungsroutine der GPGPU-Schnittstelle aufgerufen.

5.3.6. Physikberechnung

Die Physikberechnung wird durch PhysiX durchgeführt. Wie im vorigen Absatz beschrieben wird nach Festlegung der Technologie die jeweilige Funktion der GPGPU-Schnittstelle aufgerufen. Dabei wurden die API-Eintrittsfunktionen von OpenCL an CUDA angepasst, sodass die Funktionen in jeder API genau gleich ausgeführt werden. Zur Berechnung ist jedoch nicht nur die API sondern auch die Kernel-Datei nötig. Diese multipliziert das vorgegebene Geschwindigkeitsfeld mit dem vom Benutzer manipulierbaren Faktor. API wie auch Kernel wurden erst nach erfolgreichen Berechnungen im Testsystem in das Endnutzersystem übernommen.

Im Folgenden wird die API- sowie Kernelfunktion zur Berechnung dargestellt. Dabei werden zuerst Host- und Gerätespeicher für die Variablen angelegt. Danach werden die Daten vom Host zum Gerät gesendet. Es wird der nötige Kernel geladen und dessen Übergabeparameter festgelegt. Daraufhin wird der Kernel aufgerufen. Die Ergebnisse werden im Gerätespeicher der Ausgabe-Variable abgelegt und nach der Berechnung vom Gerät zum Host gesendet. Nach Abschluss der Funktion befinden sich die berechneten Werte im Array der API und werden an PhysiX und von da zur Applikation verteilt.

Kernel:

```
//---------------------------------------------------------------------------------//
//---------------------------------------------------------------------------------//
//      K E R N E L   1 - P R E C O M P U T E   A B S O L U T E   I N P U T S      //
//---------------------------------------------------------------------------------//
//---------------------------------------------------------------------------------//

__kernel void velocity_computation(__global float *d_Input1, float d_Input2, __global
float *d_Result1, int N){

        //Shared constant variables
        //__shared__ const float Tgrad = 0.0065;
        //const float Tgrad = 0.0065;


        //texturals for variables
        //d_Input1 = standard velocity field
        //d_Input2 = velocity factor
        //d_Result1 = final output velocity field

    const int tid = get_global_id(0);

    d_Result1[tid]=d_Input1[tid]*d_Input2;
}
```

API:

```
 [...]
        // Find out how many GPU's to compute on all available GPUs
        size_t nDeviceBytes;
        GPU_N = 0;            //ciDeviceCount = GPU_N
        err = clGetContextInfo(PresentContext, CL_CONTEXT_DEVICES, 0, NULL, &nDeviceBytes);
        shrCheckError(err, CL_SUCCESS);
        GPU_N = (cl_uint)nDeviceBytes/sizeof(cl_device_id);
[...]

static CUT_THREADPROC solveDevices(TGPUplan_cl* Device)
{
        //compute Grid size
        //int  GRID_SIZE = BLOCKS[plan->device] * THREADS[plan->device];
        cl_uint BLOCKS=Device->MGPU_Dev.blocks;
        size_t THREADS=Device->MGPU_Dev.used_threads;
        int GRID_SIZE = (int) (BLOCKS * THREADS);
        int N = Device->DataNumber;
        cl_command_queue Q = Device->commandQueue;
        cl_kernel K = Device->presentKernel;
        double comp_time=0;

        //----------------------------------------------------------------------------//
        //----------------------------------------------------------------------------//
        //                    K E R N E L   1   C O N F I G U R A T I O N             //
        //                                         -                                  //
        //          P R E C O M P U T I N G   A B S O L U T E   I N P U T S           //
        //----------------------------------------------------------------------------//
        //----------------------------------------------------------------------------//

        // Variable Declaration
        //----------------------------
        //Allocate memory First Kernel
        //----------------------------
        // Inputs
        clMallocHostInputSafe(&(Device->h_velocity), N*sizeof(float), (void**)&(Device-
>h_data_velocity));
        clMallocInputSafe(&(Device->d_velocity), N*sizeof(float));
        // Outputs - Switch Memory
        clMallocOutputSafe( &(Device->d_out_Vmps), N*sizeof(float) );

        //---------------------------------------------
        //Copy input data first Kernel from CPU to GPU
        //---------------------------------------------
        //Inputs (no outputs to copy, cause they are results of inputs via Kernel)
        clMemcpyHostToDevice(&Q,&(Device->h_velocity), &(Device->d_velocity),
N*sizeof(float));

        //---------------------------------------------
        //                    Set Kernel Arguments
        //---------------------------------------------
        clCreateNewKernel("velocity_computation",&K);
        ciErrNum |= clSetKernelArg(K, 0, sizeof(cl_mem), &(Device->d_velocity));
        ciErrNum |= clSetKernelArg(K, 1, sizeof(float), &(Device->h_data_factor));
        ciErrNum |= clSetKernelArg(K, 2, sizeof(cl_mem), &(Device->d_out_Vmps));
        ciErrNum |= clSetKernelArg(K, 3, sizeof(int), &N);

    //Perform GPU computations
        ExecuteKernel(&Q,&K,&(Device->GPUExecution),BLOCKS,THREADS);
        comp_time+=executionTime(Device->GPUExecution);

        clMemcpyDeviceToHost(&Q, (void**)&(Device->h_out_Vmps),&(Device->d_out_Vmps),
N*sizeof(float), &(Device->GPUDone));
        clFreePreviousKernel(&K);
        clFree(&(Device->h_velocity));
        clFree(&(Device->d_velocity));

        //Save Total Computation Time
        computation[Device->dev_num]=comp_time;

        //Shut down this GPU
        CUT_THREADEND;
}
```

5.3.7. Laufzeitmessung

Bezüglich der Messungen gibt es drei Kennwerte, dessen werde festzustellen sind. Aufgrund der komplexen internen Struktur des Equalizers ist es nicht möglich auf die, durch die Statistik dargestellte, Bildwiederholfrequenz zurück zugreifen. Die Bildwiederholrate wird daher manuell am Start der Anwendung, nach 5- und nach 30 Minuten notiert. Dabei wird nach dem Start die Windgeschwindigkeit modifiziert.

Ladephase Eins ist aufgrund der komplexen Struktur ebenfalls schwer feststellbar. Zum Logging der ersten Ladephase muss am Beginn der Applikation in der main-Funktion ein Timer erstellt werden. Dieser wird über die Klassen Config, InitData und FrameData bis zur Pipe wiedergegeben und da schließlich in der „configInit“-Funktion gestoppt. Dafür muss die Timer-Klasse überall eingebunden sein. Desweiteren gibt es dabei ein Kommunikationsproblem. Die erste GPU muss den Timer stoppen, jedoch wissen die Pipes nichts voneinander. Durch den, im Vergleich zur Phase Zwei und Drei, geringen Einfluss auf die Gesamtzeit kann das Logging für Phase Eins unter jenen Umständen ignoriert werden.

Ladephase Zwei wird bei eqOSG-Anwendungen geloggt. Dabei wird ein Timer in der „createSceneGraph“-Funktion gestartet. Beim ersten Durchgang der „updateSceneGraph“-Funktion, welches der erste Punkt nach Phase Zwei ist, wird der Timer gestoppt und Uhrzeit, Ladezeit sowie aufrufendes Demo in die Log-Datei geschrieben.

5.3.8. Installationsdateien

Zur Erstellung der Installationsdateien unter Windows wird ein neues Setup-Projekt unter Visual Studio angelegt. Es wird die geordnete Dateistruktur des Projekts im Release Build für das jeweilige System im Anwendungsordner übernommen. Zur Erhöhung der Benutzerfreundlichkeit wird eine Verknüpfung auf das Programm der grafischen Oberfläche für den Desktop sowie für das Startmenü erstellt. Zur Wiedererkennung wird diesen Verknüpfungen das vorher erstellte Icon zugewiesen. Desweiteren kann die Installation grafisch an das Programm angepasst werden, indem das weiße Installationsbanner durch ein Logo der Anwendung ersetzt wird.

Zur funktionierenden Erstellung der Installationsdatei sind die entsprechenden dynamischen Bibliotheken für die jeweilige Plattform in den Anwendungsordner zu legen. Dynamische Bibliotheken der falschen Plattform führen zu schweren Ausnahmefehlern beim Starten der installierten Anwendung auf einem anderen PC.

Zur Validierung des Programms werden Testläufe auf unterschiedlichen Systemen durchführt.

5.4. Messwerte

In diesem Abschnitt werden die Messwerte der Demonstrationsanwendungen und Referenz-Applikationen vorgestellt.

Hierbei werden zuerst die Ladezeiten der Phase Drei (Renderzeit) von Image Studio verglichen. Dabei wurde eine Szene durch aufwendige Raytracing-Technik mit einer Auflösung von 5000 x 3750 Pixel und 300 dpi gerendert. Die dabei eingesetzte Mental Ray-Technik ist ein rekursiver Raytracer, welcher komplett auf der CPU durch multiple Threads (ungefähr 400 Threads) berechnet wird. Auf allen Systemen wird die 32-Bit Version genutzt.

Rechner	**Startzeitpunkt**	**Endzeitpunkt**	**Ladezeit**	**Leistung [%]**
Laptop	18:35	Folgetag 19:30	24h 55 min	100 ,00
Pandora	10:21	12:36	2h 15 min	1107,40
Prometheus	00:18	2:35	2h 17 min	1091,24

Als nächstes werden Bildwiederholfrequenzen bei der Darstellung der Julia-Menge durch Compute Shader verglichen. Von der dafür genutzten Software GPU Caps Viewer wird keine Lösung durch Vertex- und Fragment Shader angeboten. Da der G70-Chip des Laptops keine Compute Shader unterstützt gibt es als Vergleich mit dem Laptop keinen Referenzwert. Es werden daher nur Messwerte der neuen NVIDIA- und ATI verglichen. Dabei werden vor allem Unterschiede bei zwischen 2-GPU-1-PCB- und 1-GPU-2-PCB-Konfiguration deutlich.

Rechner	**Min. FPS**	**Avg. FPS**	**Max. FPS**	**Leistung kum. [%]**
Laptop	-	-	-	-
Pandora	13	16	23	100,00
Prometheus	21	30	41	175,76

Folgend werden die Messergebnisse der Bildwiederholfrequenz bei der Darstellung des, durch Laser-Scanner aufgenommenen, Objektmodells der Freiheitsstatue präsentiert. Dabei handelt es sich um eine Demo, bei der die Verarbeitungskomplexität durch die Anzahl an

Vertices entsteht (über 14 Milliarden Eckpunkte). Durch den sehr begrenzten Grafikspeicher des Laptops kann diese Demo nicht auf dem Laptop ausgeführt werden.

Rechner	**Start-FPS**	**Nach 5 min.**	**Nach 30 min.**	**Leistung [%]**
Laptop	-	-	-	-
Pandora	5,45	9,96	10,1	100,00
Prometheus	0,42	-	-	[7,71]

Eine weitere Demo, bei der sich Leistungssteigerungen deutlich bemerkbar machen sollen, ist die Darstellung eines Körper-CTs. Dabei wird die Anzeige in 2 verschiedene Fenster aufgeteilt. Bei Vorhandensein mehrerer GPUs wird jedes Fenster von einer eigenen GPU berechnet. Bei nur einer vorhandenen GPU werden beide Fenster durch die gleiche GPU berechnet.

Rechner	**Start-FPS**	**Nach 5 min.**	**Nach 30 min.**	**Leistung kum. [%]**
Laptop	38,5	24,3	25,2	100,00
Pandora	59,9	55,6	52,3	197,31
Prometheus	60,0	30,7	31,5	135,72

Schlussendlich werden Messwerte der finalen Pyramidendemo miteinander Vergleich. Diese wurde dabei auf verschiedenen Plattformen und mit verschiedenen Technologien gemessen. Sie beinhaltet die Aufteilung der Grafikpipeline durch Equalizer, eine Partikelengine sowie die damit verbundene Compute Shader-Physikengine.

Rechner	**Start-FPS**	**Nach 5 min.**	**Nach 30 min.**	**Leistung kum. [%]**
Laptop 32-Bit CPU	40,4	27,8	[20,3] (28 min)	100,00
Pandora 64-Bit CUDA	10,2	12,9	13,0	45,23
Pandora 64-Bit OpenCL	49,4	50,3	42,5	170,86
Prometheus 64-Bit CPU	60,0	61,3	49,6	204,21
Prometheus 64-Bit OpenCL	-	-	-	-

Desweiteren folgt die Vorstellung der Messwerte von Ladephase Zwei für die Pyramidendemo.

Rechner	**Phase 2 Ladezeit [ms]**	**Leistung kum. [%]**
Laptop 32-Bit CPU	4212	100,00
Pandora 64-Bit CUDA	-	-
Pandora 64-Bit OpenCL	808	521,29
Prometheus 64-Bit CPU	2596	162,25
Prometheus 64-Bit OpenCL	-	-

Es folgen ebenfalls reine Zeitmessungen der GPGPU-Berechnung im Vergleich mit einer reinen CPU-Lösung.

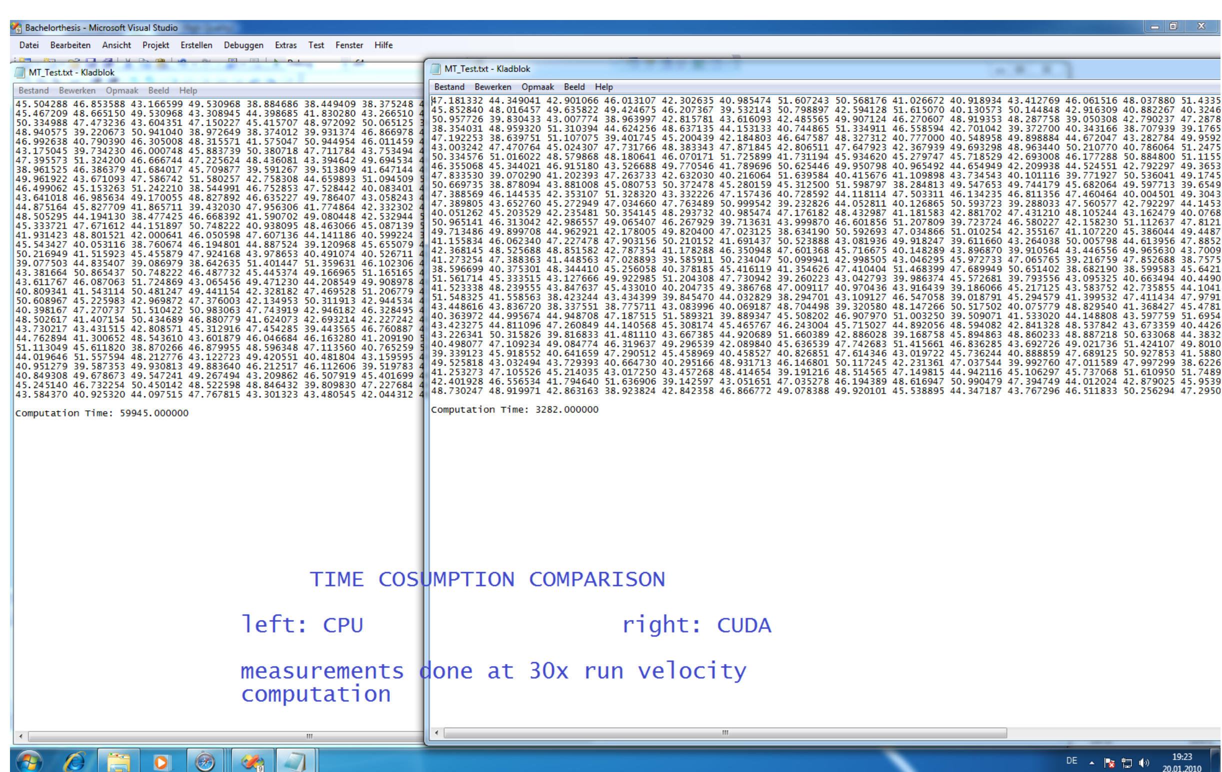


Bild 43 Vergleich der Ladezeiten Phase 3 CPU-CUDA

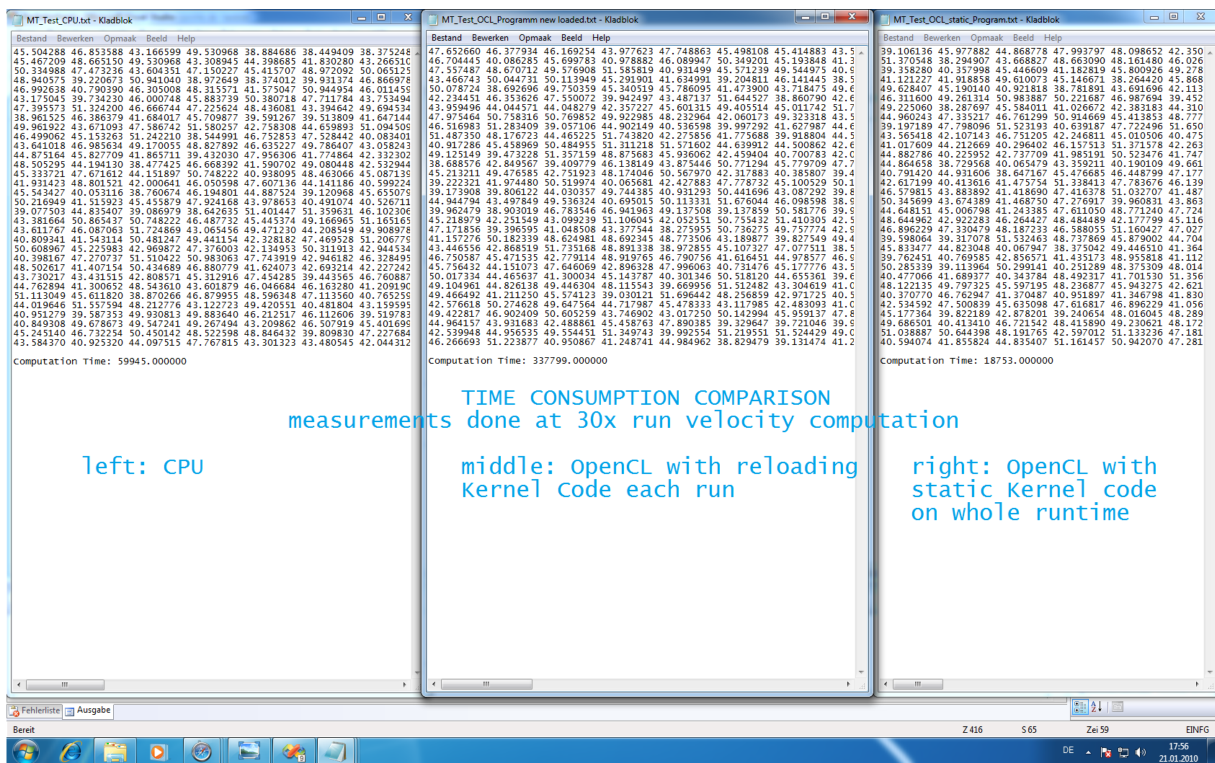


Bild 44 Vergleich der Ladezeiten Phase 3 CPU-OpenCL

6. Bewertung

Im vorigen Kapitel wurde das, zur Ermittlung der Messwerte und Leistungsfähigkeit der Multi-GPU Technologie, entworfene Programm vorgestellt. Desweiteren wurden die Messwerte der einzelnen Demonstrationsanwendungen präsentiert.

Dieses Kapitel wird Bewertung und Rückschlüsse auf drei Themenbereiche der Arbeit eröffnen. Beginnend mit der Bewertung der Messwerte werden mögliche Gründe für die auftretenden Abweichungen von theoretischem Leistungszuwachs und praktischen Messwerten erläutert. Davon ausgehend werden Rückschlüsse für weitere, zukünftige Themenbereiche und Algorithmen gezogen, welche durch Einsatz von mehreren Grafikkernen profitieren können.

Im zweiten Abschnitt werden die gewonnenen Erfahrungen in Bezug auf Ansatz, Entwurf und Realisierung von Projekten mit den genutzten Technologien erläutert. Es werden Erkenntnisse für das Software Engineering in diesen Projekten genannt.

Im letzten Abschnitt wird das entwickelte Programm hinsichtlich des Erfolg und der Ausbaufähigkeit analysiert. Desweiteren erfolgt eine Code-Umfangsanalyse.

6.1. Messergebnisse

Bei der Auswertung der Julia-Menge-Demo ist sichtbar, dass bei umfangreichen, wiederkehrenden Berechnung das 1-PCB Design sehr vorteilhaft ist. Dies kann auf den zentralen Controller-Chip zurückzuführen sein, da bei 2-PCB-Lösungen die Daten von dem Controller der Anzeigekarte verarbeitet werden, die Daten jedoch erst über eine Brücke übertragen werden müssen.

Die Demo der Freiheitsstatue wurde auf Prometheus nicht einwandfrei berechnet. Obwohl die Modelldaten geladen wurden fiel die Bildwiederholrate rapide ab. Dies kann mit dem begrenzten Hauptspeicher von Prometheus in Verbindung gebracht werden. Durch das Betriebssystem werden ungefähr 900 MB Arbeitsspeicher beansprucht. Desweiteren muss für das Mapping der Grafikspeicherobjekte die gleiche Menge an Hauptspeicher zur Verfügung stehen. Dies sind bei Prometheus 2048 MB. Daher bleiben für Benutzerprozesse nur 1148 MB zur Verfügung. Dies ist zur Verarbeitung der Demo der Freiheitsstatue nicht ausreichend. Durch ständige Ein- und Auslagervorgänge auf die Festplatte wird dabei die Demo soweit verlangsamt, dass die Verarbeitung und das System zum erliegen kommt.

Bei der „Body CT“-Demo wird die reine Bildwiederholfrequenz nur durch die Bildkomposition der Anzahl der Grafikkarten bestimmt. Während beim Vergleich von Laptop und Pandora ein Faktor von annähernd 2,0 zu sehen ist trifft dies auf die 1-PCB Variante in Promtheus nicht zu. Dies liegt im Treiber der Grafikkarte begründet, wobei seit Catalyst 8.2 Multi-GPU Systeme als ein Geräte mit unterschiedlichen Speicherbänken anstatt zwei unterschiedlichen Geräte verwaltet wird. Daher kann die Applikation die Grafikkarten nicht getrennt adressieren.

Bezüglich der Hauptdemo der Pyramide sind einige Erklärungen nötig.

Zuerst ist zu sagen, dass, obwohl OpenCL eine offene, durch NVIDIA und ATI abgestimmte Schnittstelle ist, deren Unterstützung unterschiedlich verwaltet wird. Durch schnellere Weiterentwicklung seitens ATI sind bei der Kreation eines OpenCL-Kontexts einige Programmiertechniken als „veraltet“ markiert, welche jedoch vom ebenfalls aktuellen NVIDIA-Treiber akzeptiert werden. Daher kann die Unterstützung von OpenCL sich stark zwischen den Herstellern unterscheiden. Dies Problem wurde jedoch erst nach Abschluss der praktischen Arbeit von AMD/ATI bekannt gegeben. Desweiteren wird OpenCL im ATI Stream SDK 2.0 in Verbindung mit Standard Template Library-Objekten benutzt, während durch das NVIDIA SDK eine prozedurale Vorgehensweise vorgegeben wird. Daher besteht eine Herausforderung in der Schaffung eines gemeinsamen Ansatzes für beide SDKs.

Die Bearbeitung der Physik durch CPU wie auch durch OpenCL liegt auf gleichem Niveau. Gründe dafür können die Einfachheit der Berechnung und die geringe Größe des Feldes sein.

Bei der Übertragung der neuen Daten auf die Grafikkarte entsteht ein gewisser Overhead. Die Leistungsfähigkeit und der Vorsprung an Bildwiederholfrequenz nur dann sichtbare Auswirkungen haben, wenn die eigentliche Bearbeitungszeit der Berechnung auf der CPU unverhältnismäßig größer als die Übertragungszeit der Daten zur GPU ist. Dies wäre bei der ursprünglichen Berechnung im 4-Punkte-Kernel der Fall gewesen. Durch eine fehlende Entwicklungsumgebung können konnte dieser komplexe Kernel nicht genutzt werden. Durch die Einfachheit des neuen Kernels übersteigt der Overhead den Nutzen. Durch die reinen Berechnungsergebnisse lässt sich jedoch prognostizieren, dass bei geeigneter An- und Verbindung der Physikengine mit der Grafikpipeline und bei geeigneter Berechnungsgröße deutliche Geschwindigkeitsvorteile mit Compute Shadern erzielt werden können.

Der Einbruch der Bildwiederholfrequenz bei CUDA ist auf das bereits analysierte Problem zurückzuführen, dass der CUDA-Kontext mit jedem Frame neu initialisiert wird. Dies bedeutet immer neuen Zugriff auf die Festplatte für das laden des Kernels und neues Speicher-Mapping zwischen CPU/Hauptspeicher und GPU/Grafikspeicher. Daher ist der starke Geschwindigkeitsverlust durch den Fall der Bildwiederholrate erklärbar.

Dabei wird die Aktualisierungsrate der Grafikpipeline mit maximal 65 Frames per Second begrenzt. Dieses Phänomen existiert durch die vertikale Synchronisation von Anzeigebildern [gam04] . Vertikale Synchronisation wird bei doppelt gepufferten Anzeigen, der Standard in 3D-Systeme, genutzt. Diese Technik beruht auf der Top-Down-Anzeigecharakteristik von CRT-Monitoren. CRT-Monitore bauen ein Bild mit 30 Halbbildern pro Sekunde (Interlaced) oder maximal 60 Vollbildern pro Sekunde auf. Durch die Anzeigecharakteristik wird VSync eingesetzt, damit keine Überlappungen von Alt- und Neubild entstehen. Die Bilder werden synchronisiert. Dadurch sind höhere Frameraten nicht möglich. Dies kann Nutzung dreifach gepufferte Bilder (Tripple Buffering) anstatt der vertikalen Synchronisation verhindert werden[Wik07] .

Es folgt eine grafische Auswertung der Messwerte.

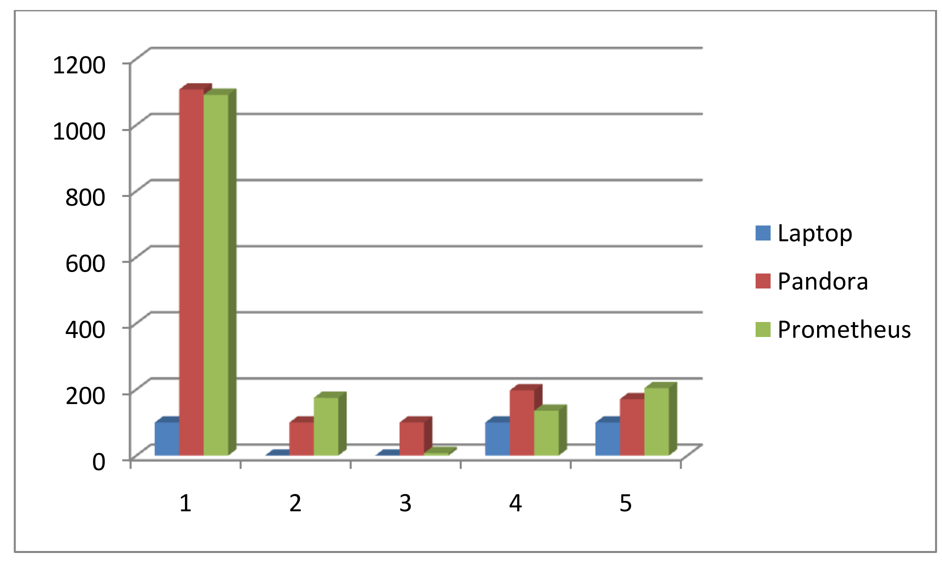


Bild 45 Vergleich der finalen Messwerte

6.2. Software Engineering

Bei der Entwicklung von Anwendungen für den Massenmarkt, welche von einer verteilten Umgebung profitieren sollen, sind einige neue Erkenntnisse und Ansätze zu berücksichtigen.

Während der Planungsphase ist ein Mehraufwand in die Bestandsanalyse zu investieren. Dies kann unter Umständen zu einer nicht unerheblichen Reduzierung des Entwurfs- und Realisierungsaufwands führen. Große grafische Projekte basieren häufig auf einem Framework sowie einer oder mehreren APIs und Bibliotheken. Zu einigen großen Frameworks sind Erweiterungen für Compute Shader in Verbindung mit Multi-GPU Technologie in Entwicklung. Bei einer Möglichkeit der Beschleunigung der Anwendung durch Multi-GPGPU Technologien kann die jeweilige Community unterstützt werden. Durch derartige Zusammenarbeiten kann eine gute, allgemein gültige Lösung mit begrenztem Eigenaufwand erreicht werden.

In der Spezifikationsphase sollte genau festgelegt werden, was mit der neuen Version oder Applikation erreicht bzw. beschleunigt werden sollte. Es existieren viele Möglichkeiten eine Anwendung durch mehrere Grafikkerne zu beschleunigen, wie diese Arbeit belegt. Eine exakte Formulierung der Ziele wird unnötigen Aufwand bei der Entwicklung einer Technologie vermieden, die im Endeffekt jedoch nicht entscheidend für die Anwendung ist. Deshalb ist ein entscheidender Punkt, das „Bottleneck“ der Anwendung zu finden.

Spätestens am Ende der Spezifikationsphase müssen ebenfalls Entscheidungen bezüglich Gruppenzusammensetzung und Aufgabenzuweisungen erfolgen. Dabei besteht die optimale Zusammensetzung nach Analyse des dargelegten Entwicklungsprozesses aus folgender Gruppe:

- **Shader-Programmierer**
  - Solide Kenntnisse mit C
  - Bestenfalls Erfahrung in der Shader-Programmierung
  - Physikalisches Vorwissen oder Interesse wünschenswert
  - Kenntnisse im Bereich Mikrocontroller oder Grafikchips
  - Möglichst stetige oder initiative Person
  - Technikbegeistert
  - Aufgaben: Kernel-Programmierung; Shader-Programmierung
- **Schnittstellen-Programmierer**
  - Fortgeschrittene Kenntnisse der Programmiersprache der Anwendung
  - Vorwissen in verwendeten (geplanten) APIs, wahlweise Front- oder Backend
  - hochgradig teamfähig
  - kommunikativ
  - Grundkenntnisse in Softwarearchitekturen

    - Aufgabe: Verbindung von Frontend, Backend, Verteilungstechnologie, Darstellungsframework und (wenn nötig) Physikengine
- **Software-Architekt**
    - Umfangreiches Wissen in Hardware- wie auch Software-Architekturen
    - leitende Funktion
    - kommunikativ, dominant
    - flexible Denkweise; Fähigkeit, intuitiv auf Modell-Fehlschläge zu reagieren
    - Aufgaben: Entwicklung des Software-Layouts auf Basis verwendeter oder verwendbarer Schnittstellen; System-Modellierung; Regelung der Gruppenkommunikation
- **Grafik-Programmierer**
    - Solides Wissen über grafische Datenverarbeitung
    - Kenntnis über OpenGL, DirectX und deren Derivate
    - Wissen über Grafikpipeline und dreidimensionale Objektformate
    - Erfahrung mit CAD, CAS oder DCC Systemen wünschenswert
    - Kenntnisse in Oberflächenprogrammierung und Design
    - Interesse für Grafik
    - möglichst initiative, freundliche Persönlichkeit; Motivationsfähigkeit erwünscht
    - Aufgaben: Erstellen einer simplen, intuitiven grafischen Oberfläche; Umsetzung der aktuellen Renderpipeline mit gewählten Grafikschnittstelle; Auswahl und Modellierung von Effekte

Während der Entwicklungsphase sollten Verbindungspunkte von Verteilungsklassen, Physikklassen und Renderklassen gefunden werden. Die Effektivität der Aufteilung hängt von den Entscheidungen ab, welche Funktionalität global und welche lokal gelöst werden müssen. Die Software sollte in globale, nicht parallelisierbare Aufgaben (z.Bsp. GUI-EventQueue) und lokal lösbare, parallelisierbare Aufgaben unterteilt werden. Als Basis der Software-Architektur hat sich das Server-Client-Paradigma als guter Ansatz erwiesen.

Bezüglich der Einzelarchitekturen haben sich nach Analyse vorhandener Technologien (siehe Kapitel 3) folgende Ansätze jeweilige Problemgruppen als praktikable Lösungen angeboten.

- Verteilung (Multi-GPU; Multi-CPU; Cluster): Server-Client-Architektur mit globalen Kommunikationsräumen (beispielsweise Sockets)
- Darstellung: Applet-Architektur mit Trennung von genereller Pipeline und speziellen Use Cases
- Physikengine: GPU Compute Units als kaskadierte 1/2/3-dimensionale Felder

Bezüglich der Realisierungsphase ist anzudenken, eine mögliche Physikengine vom Endsystem zu trennen. Dies Reduziert die Komplexität, lenkt den Fokus auf wenige Technologien mit ähnlichem Grundsatz und ein aussagekräftiger Test der Engine ist möglich. Desweiteren ist anzuraten, Schnittstellen mit gleicher Aufgabe auch einer ähnlichen Code-Struktur zu unterwerfen.

6.3. Programm

Das praktische Resultat dieser Arbeit ist ein solides Basisprogramm welches Nutzen aus Multi-GPU Umgebung zieht. Dabei wurde nach eingehende Analyse die bestpassenden Technologien zur Erreichung des Ziels ausgewählt und mit einander verbunden. Dabei wurde eng mit den Grafikherstellern und einem Softwareproduzent für derlei Programme zusammen gearbeitet. Jedoch wurden während der Entwicklung auch Ziele wie Erweiterbarkeit und Übersichtlichkeit verfolgt. Desweiteren wurde versucht, die grundsätzlich existierende Betriebssystemunabhängigkeit beizubehalten.

Durch den stark begrenzten Zeitrahmen waren nur Grundsatz-Demos in der Entwicklung möglich. Diese hatten dabei nicht das Ziel einer fotorealistischen Darstellung und der Visualisierung eines bestimmten Effekts. Jedoch können mit den gewonnenen Kenntnissen fotorealistische Technologien wie Raytracing oder Radiosity effektiv in einer Multi-GPU Umgebung in Zukunft umgesetzt werden.

Die verwendeten Technologien sind stark von der verwendeten Hardware und dem damit verbundenen Befehlssatz abhängig. Deshalb ist eine Arbeit mit den Grafikkartenherstellern und stetige Informationsbeschaffung über die Technologien entscheidend für den Erfolg der Entwicklung.

Bezüglich der Steuerung der Grafikkompositionsmodi existiert seit Ende Januar eine neue NVIDIA-API namens „NVCpl“, welches den direkten Zugriff auf die Kompositionsmodi erlaubt. Durch die späte Herausgabe der API konnte diese nicht mehr in die Software aufgenommen werden. Dies sollte ein Hauptziel der Weiterentwicklung sein. Desweiteren ist auf eine ähnliche Treiber-API von ATI Technologies zu warten.

Bezüglich des Umfangs der Physikberechnung kann festgestellt werden, dass die geplante Physikberechnung der Luftmassen vergleichbar mit allgemeinen, physikalischen Grafiksimulationen ist. Eine umfangreichere Physikberechnung ist aus folgenden Gründen für Demonstrationen nicht möglich gewesen.

- Zeitlicher Aufwand
- Spezialwissen in physikalischen Disziplinen außerhalb der Optik
- Fehlende IDE zur Entwicklung komplexer GPGPU-Kernels
- Fehlender Compute Shader-Debugger
- Fehlende OpenCL Building Rule

Jedoch dienen komplexe astrophysikalische Demos wie beispielsweise „NBody Simulation“ als Leistungstest heutiger Desktop-Grafikkarten. Die genannte Simulation gehört zum Standardrepertoire der Beispielapplikationen, welche mit den GPGPU-SDKs mitgeliefert

werden. Komplexe Galaxie-Simulationen sind als Video auf der Entwickler-DVD beigefügt. Dies zeigt die Möglichkeiten der Echtzeit-Physiksimulation auf heutigen Grafikchips. Standardbeispiele der SDKs sind nur auf Einzel-GPU-Berechnungen aufgebaut, können jedoch bei Wissen der API mit geringem Aufwand auf eine Multi-GPU Plattform portiert werden.

Die Entwicklung brachte 54.286 C++/C- Quellcodezeilen hervor, sowie 268 Kernel-Zeilen, 333 EML-Zeilen (Equalizer Markup Language) und 755 Shader-Quellcodezeilen.

Die Verteilung ist in folgendem Diagramm dargestellt.

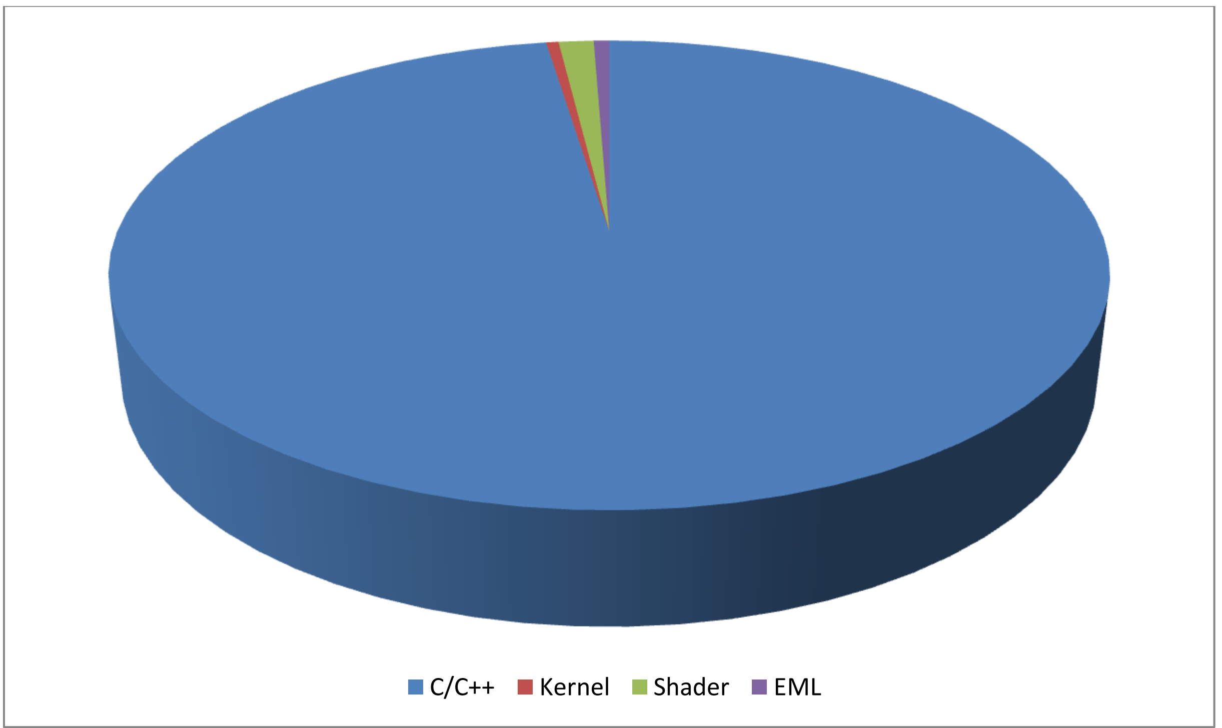


Bild 46 Anteil der Programmiersprachen am Gesamtaufwand

7. Zusammenfassung

Aufgrund der umfangreichen theoretischen und praktischen Abhandlung des Themas wird folgend eine Zusammenfassung der Ergebnisse der Untersuchung und deren Schlussfolgerung dargeboten. Anschließend wird ein Ausblick auf die Weiterentwicklung der Software gegeben. Abschließend folgt ein Technologieausblick der Grafikhardware.

7.1. Ergebnisse und Schlussfolgerungen zur Leistungsuntersuchung

Durch den Einsatz der Technologie lassen, in Bezug der untersuchten Anwendungen, ein Leistungszuwachs von ungefähr 80% erzielen. Der genaue Leistungszuwachs ist stark Anwendungsgebunden.

Bei Anwendungen mit großen Datenmengen ist die Aufteilung der Grafikpipeline nicht effektiv genug, da diese in dem Fall nicht das „Bottleneck" der Anwendung darstellt. Leistungsverluste entstehen bei diesen Anwendungen hauptsächlich durch den begrenzten Speicher der Grafikkarte und dem benötigten, korrespondierenden Mapping-Bereich im Hauptspeicher. Durch diese Limitierung müssen häufig große Datenmengen über den PCI-Express oder von der Festplatte neu geladen werden. Dies ist dabei der größte Kostenfaktor in Bezug auf die Darstellung. Für diese Art der Anwendungen, beispielsweise die medizinische Computergrafik, müssen die Daten vorher logisch aufgeteilt werden, sodass die Daten des Sichtfeldes mit abnehmendem Detailgrad komprimiert verarbeitet werden.

Anwendungen mit einer erforderlichen, großen Zielauflösung bieten den größten Leistungszuwachs. Hierbei wären vor allen Dingen CAD-Systeme zu nennen. Die Aufteilung der Grafikpipeline bedeutet eine Aufteilung des Zielbildes und damit die Reduzierung der Auflösung, die von einer einzelnen Karte geliefert wird. Ein weiterer Grund für das Einsatzgebiet in CAD-Systemen ist die häufige Verwendung von Profi-Grafikkarten mit speziellem Befehlssatz. Diese beinhalten die nötigen OpenGL-Erweiterungsfunktionen zur direkten Aufteilung der Renderfläche.

Grafische Simulationen und die grafische Datenverarbeitung für Animation und Film verbindet ein Fakt: Die Darstellung physikalisch korrekt interagierender Objekte.

Anwendungen dieser Bereiche profitieren am meisten von den neuen Shader-Technologien und den damit verbundenen, parallelen Recheneinheiten. Physikalische Berechnungen auf dem Grafikprozessor ermöglichen durch die hohe Anzahl paralleler Recheneinheiten flüssige Darstellungen komplexer physikalischer Animationen und Bewegungen. Dies skaliert sehr gut mit steigender Grafikprozessorzahl. Um diesen Geschwindigkeitsvorteil zu bemerken müssen jedoch einige Randbedingungen erfüllt sein. Die Integration von GPGPU-Technologie und Renderpipeline ist dabei ein entscheidender Faktor. Desweiteren muss die Komplexität der Berechnung groß genug sein, damit der Vorteil der parallelen Verarbeitung den Kommunikationsoverhead übersteigt.

Durch die Technologie der GPGPU lassen sich große Geschwindigkeitsvorteile für parallelisierbare Operationen erzielen. Dabei wäre für die Zukunft Raytracing und Radiosity zu zählen. Desweiteren wird heutzutage großer Nutzen aus Nachbearbeitungseffekten wie Tiefenunschärfe und Anti-Aliasing gezogen, welche in einer Multi-GPU Umgebung durch Aufteilung der Renderdurchläufe beschleunigt werden können.

7.2. Zukünftige Erweiterung der Software

Trotz einer funktionierenden, installierbaren Version sind einige Technologien in der Entwicklung noch offen.

Zuerst sollte eine gemeinsame Basis in der Erstellung des OpenCL-Kontexts zwischen ATI und NVIDIA gefunden werden. Dabei gibt ATI den Maßstab der Entwicklung vor. Etwaige Quellen der AMD Developer Knowledge Base weisen auf eine Plattformdefinition hin, welche bei ATI im Vorfeld der Kontexterstellung erledigt werden muss. Dies ist bei NVIDIA nicht nötig, trotzdem machbar.

Desweitern sind einige Änderungen und zusätzliche Implementationen in das Equalizer-Framework aufzunehmen. In Zusammenarbeit mit Eyescale sollten Treiber-APIs von ATI und NVIDIA mit dem Equalizer verbunden werden, um etwaige Leistungssteigerungen auch auf Standard-GPUs zur Verfügung zu stellen. Ein weiteres Ziel ist die Implementation der neuen AMD/ATI OpenGL-Erweiterung „wgl_amd_gpu_association“ für weitere Herstellerunabhängigkeit.

Eine Zusammenarbeit mit einem Grafikunternehmen, welches professionelle CAD-Grafikkarten besitzt, kann die Funktionsweise und den Leistungszuwachs durch die Equalizer-Funktionen nachweisen.

Desweiteren sollten Gründe für den unsachgemäßen Abbruch eines Open Scene Graph-Programms bei Nutzung eigener Shader gefunden werden. Ohne selbstgeschriebene Shader bleibt die Auswahl an Effekten sehr begrenzt.

Die Nutzung des Visual Studio-Frameworks „NVIDIA NEXUS“ kann dazu genutzt werden, den originalen Vier-Stufen-Kernel zu debuggen und einzusetzen. Damit können weitere Messwerte bei komplexen Physikberechnungen entnommen und ausgewertet werden.

Für die Nutzung der Anwendungen in der medizinischen Computergrafik müssen Erweiterungen zur dynamischen Datenkomposition- und Komprimierung entwickelt werden. Somit können die massiven Datenaufkommen in der Medizin aufgeteilt werden. Damit wird auch der Nutzen der Aufteilung der Renderpipeline steigen.

Weitere Erweiterungen bestehen in der Implementation komplizierter, bildgebender Verfahren wie Raytracing und Radiosity. Zur Trennung der Anwendungsgebiete kann ein Profiling-System für Nutzer entwickelt werden. Große Datenmengen können ebenfalls in Datenbanken abgespeichert und verwaltet werden. Dies erleichtert ebenfalls die Verarbeitung der Daten, da hierbei die unterschiedlichen Dateiformate durch ein internes Format der Datenbank ersetzt werden kann.

7.3. Technologieausblick

Der momentan schnellste Grafikchip ist ATIs Cypress-Chip der HD5800 Serie. Damit verbundene Demos zeigen korrekte Linsenfokussierung mit Hilfe des Tiefenunschärfe-Effektes. Desweiteren ist die einer der ersten Chips mit integriertem Displacement Mapping. Dabei werden Höhentexturen in neue Objekte umgeformt. Für eine möglichst genaue Lichtberechnung werden die, aus Höhentexturen erstellten Objekte, hoch tesseliert.

Ende des ersten Quartals 2010 erscheint der neue Grafikprozessor GF100/GT300 von NVIDIA, auch als „Fermi“ bekannt. Dieser besitzt vier physische Kerne auf einem PCB, welche intern mit einander verbunden sind. Beispielapplikationen des Fermi-Prozessors sind unter anderem auch Echtzeit-Raytracing Animationen. Diese werden jedoch nur mit Hilfe von drei Fermi-Karten in über 20 fps dargestellt. Desweiteren gibt es viele, verschiedene Raytracing-Einstellungen und Shader-Effekte, welche verschiedene Materialien und Eigenschaften darstellen und demnach auch unterschiedlichen Verarbeitungsgeschwindigkeiten unterliegen. Jedoch ist der GF100 Chip ein Beispiel des allgemeinen Trends zur Echtzeit-Raytracing Grafikkarte.

Jenes Ziel wird ebenfalls von Intel verfolgt. Während der SIGGRAPH 2008 wurde ein erster Prototyp des Larrabee vorgestellt. Dies ist ein x86-basierter Chip mit Befehlssatz- und Architekturanpassungen an die grafische Datenverarbeitung. Ende 2009 wurde der Chip jedoch wegen ungenügender Performance im Grafikbereich nicht auf den Endnutzermarkt gebracht. Er dient zur Entwicklung und wird von Intel an Forschungseinrichtungen verkauft. Die Entwicklung wurde jedoch nicht eingestellt. Sie wird für den Nachfolgechip „Intel Larrabee II“ weitergeführt. Vorteil der x86-Architektur ist die Entwicklung von Shadern mit vollem Befehlsumfang des Hauptprozessors. Damit werden Programmiertechniken wie zum Beispiel Rekursion auf der Grafikkarte mögliches, welches essentiell zum Raytracing gehört.

Literaturverzeichnis

ATI RV3x0 Pixel Shader - Überlight. **[Mit03] Mitchell, Jason L. 2003.** Sand Diego : ATI Research, 2003. SIGGRAPH2003.

[Bos] Bosi, Michele. Visualization Library. [Online] www.visualizationlibrary.com.

[Buc08] Buchty, Rainer und Weiß, Jan-Philipp. 2008. *High-Performance and Hardware-aware Computing.* Karlsruhe : Universitätsverlag Karlsruhe, 2008. 978-3-86644-298-6.

[For01] Cole, Forrester. 2001. Reports from SIGGRAPH. *Reports from SIGGRAPH 2001 - Procedural Modeling.* [Online] August 16, 2001. [Cited: Februar 11, 2010.] www.siggraph.org/conferences/reports/s2001/tech/papers8.html.

Compilation of procedural models. **[TUl08] Ullrich, T. 2008.** Los Angeles, California, USA : Fraunhofer Publica, 2008. WEB3D 2008.

Efficient High Level Shader Development. **[Tar03] Tartachuk, Natalya. 2003.** Europa : ATI Technologies, 2003. GDCE2003.

[Eil09] Eilemann, Stefan. 2009. *Equalizer Programming and User Guide.* Neuchâtel : s.n., 2009.

[Ste08] —. 2008. *Equalizer Whitepaper - OpenGL Scalability for Multi-GPU Systems.* Neuchâtel : Eyescale Software GmbH, 2008.

[Ste] —. Equalizer: Parallel Rendering. [Online] www.equalizergraphics.com.

[Phy08] Forum, Physics. 2008. Why is 60 fps recommended ? *Physics Forum.* [Online] Physics Forum, 8. 1 2008. [Zitat vom: 15. 2 2010.] http://www.physicsforum.com/showthread.php?t=208527.

[gam04] gamedev.net. 2004. [SOLVED] OpenGL 60fps cap on Vista. *gamedev.net - Forum.* [Online] gamedev.net, 29. 8 2004. [Zitat vom: 15. 2 2010.] http://www.gamedev.net/community/forums/topic.asp?topic_id=492589.

GDV Vorlesung - Beleuchtung und Schattierung. **[Her08] Litschke, Herbert. 2008.** Wismar : Hochschule Wismar - Multimediatechnik, 2008.

GDV Vorlesung - Mathematische Modellierung von Objekten. **[Her081] Litschke, Herbert. 2008.** Wismar : Hochschule Wismar, 2008.

[Sil09] Graphics, Silicon. 2009. SGI - Products: Software: IRIX. *sgi accelerating results - IRIX.* [Online] Silicon Graphics International, 2009. [Zitat vom: 2. 13 2010.] http://www.sgi.com/products/software/irix/.

[Hum02] Humphreys, Greg und Houston, Mike. 2002. *Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters.* s.l. : Stanford University, 2002.

[Jes06] Jesshope, C.R. 2006. *uTC: an intermediate language for programming chip multiprocessors.* Amsterdam : Universiteit van Amsterdam, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, 2006.

Multi-GPU Scaling for Large Data Visualization. **[Rug08] Ruge, Thomas und NVIDIA. 2008.** San Jose, California, USA : NVIDIA NVISION 2008, 2008.

[NVI09] NVIDIA. 2009. NVIDIA SLI technology. *nVIDIA SLI ZONE.* [Online] NVIDIA, November 2009. [Zitat vom: 3. 3 2010.] http://de.slizone.com/object/sli_cuda_learn_de.html.

[NVI10] —. 2010. NVIDIA Technology Drives Earth-Shattering Visual Effects in Blockbuster Hit "2012". *nVIDIA.* [Online] NVIDIA, Januar 2010. [Zitat vom: 15. 2 2010.] http://www.nvidia.com/object/io_1258094160555.html.

Open Scene Graph. **[Osf03] Osfield, Robert und Burns, Don. 2003.** s.l. : Open Scene Graph, 2003.

[Osf] Osfield, Robert. Open Scene Graph. *Open Scene Graph.* [Online] www.openscenegraph.org.

[Pao07] Paolo Scarpazzam, Daniele, Villa, Oreste und Petrini, Fabrizio. 2007. Dr. Dobb's | Programming the Cell Prozessor. *High Performance Computing - Programming the Cell Processor.* [Online] techwebb, 9. März 2007. [Zitat vom: 13. 2 2010.] www.ddj.com/dept/64bit/197801624.

Parallel Rendering using OpenGL Multipipe SDK. **[Bha04] Bhaniramka, Praveen. 2004.** Austin, Texas, USA : Silicon Graphics International, 2004. VIS2004.

[Rei09] Reinders, James. 2009. Intel Software Network Blogs. *Rapidmind + Intel.* [Online] Intel Corp., 19. August 2009. [Zitat vom: 13. 2 2010.] resident on CD.

[Ros07] Rost, Randi J. 2007. *OpenGL Shading Language - 2nd ed.* Crawfordsville, Indiana, USA : Addison Wesley, 2007. 0-321-33489-2.

[SGI] SGI. *White Paper - OpenGL Multipipe SDK.* s.l. : Silicon Graphics International.

[Shr09] Shreiner, Dave, et al. 2009. *OpenGL Programming Guide - 6th ed.* Stoughton, Massachusetts, USA : Addison Wesley, 2009. 978-0-321-48100-9.

[Sot06] Sotry, Jon. 2006. *Harnessing the Performance of CrossFireX.* s.l. : ATI Technologies, 2006.

[Ste10] Stenzel, Christian. 2010. Dipl. Ing. *Interprozesskommunikation.* Wismar, 23. 2 2010.

[Typ04] TyphoonLabs. 2004. *OpenGL Shader Designer Manual.* s.l. : TyphoonLabs, 2004.

[Wik10] Wikipedia. PCI-Express. *PCI-Express - Wikipedia.* [Online] Wikipedia. [Zitat vom: 8. 2 2010.]

[Wik07] —. 2007. Vertical Synchronization - Wikipedia. *Wikipedia [EN] - Vertical Synchronization.* [Online] Wikipedia, December 2007. [Zitat vom: 15. 2 2010.] http://en.wikipedia.org/wiki/Vertical_synchronization.

Bildverzeichnis

Bild 1 Mind Map Bedeutung "grafische Datenverarbeitung" 11
Bild 2 Aufteilung der Renderphasen 12
Bild 3 Drahtgittermodel einer polygonalen Kugel [Wikipedia] 13
Bild 4 Smooth Shading [OpenGL RedBook Plate 6] 14
Bild 5 Tiefenunschärfe (Depth of Field) [imageshak.us] 15
Bild 6 Shadow Mapping [blenderartist.org] 15
Bild 7 Historische Entwicklung der Grafikbeschleiniger 16
Bild 8 physische Architektur der Grafikkarte 19
Bild 9 logische Architektur der Grafikkarte 20
Bild 10 ATI Technologies Kompositionsmodi [ATI Technologies] 23
Bild 11 NVIDIA Multi View-Modus [NVIDIA] 24
Bild 12 NVIDIA Load Balancing SFR-Modus [NVIDIA] 25
Bild 13 Übersicht PCI-Express Bandbreiten [Wikipedia] 27
Bild 14 Übersicht der verschiedenen Koordinatensysteme 37
Bild 15 Auswirkung von Bump Mapping 38
Bild 16 Bump Map eines Mauerwerks [imageshak.us] 39
Bild 17 Erhöhung der Platizität von Objekten durch "Kratzer" (Bump Mapping) [NVIDIA] 39
Bild 18 Cg ÜberLight-Shader [GPU Gems 2] 41
Bild 19 perfekte Spiegelung durch Cube Mapping [NVIDIA] 41
Bild 20 Umgebungsverzerrung durch Motion Blur 41
Bild 21 Physikberechnung auf der CPU 42
Bild 22 HavocFX und PhysX 42
Bild 23 Compute Shader und GPGPU Technologie 43
Bild 24 lokales 3D-Koordinatensystem und 2D-Bildkoordinatensystem 44
Bild 25 Architektur OpenGL Renderpipeline [OpenGL Red Book] 50
Bild 26 Rechenaufteilung von Rapidmind [Codeguru, 2009] 58
Bild 27 Rapidmind Architektur [www.redcanary.ca, 2009] 58
Bild 28 osSimPlanet [www.ossim.org] 60
Bild 29 Virtual Planet Builder [osg Forum] 60
Bild 30 osgOcean des VENUS-Projekts [code.google.com/p/osgocean] 60
Bild 31 VL-Darstellung solider Objekte durch Voxelgrafik und MarchingCubes 62
Bild 32 VL-Darstellung von Terrain und Oberflächen 62
Bild 33 multiple Lichtquellen in VL 62
Bild 34 Partikelsystem in VL 62
Bild 35 Equalizer DB Compound (sort last-Algorithmus) [Eil09] 66
Bild 36 Equalizer Stereo Compound [Eil09] 66
Bild 37 Klassendiagramm des Basisentwurfs 78
Bild 38 Visualisierung der Aufteilung im Klassenkontext des Basisentwurfs 81
Bild 39 Verteilte Server-Client-Architektur des Equalizer-basierten Entwurfs 85
Bild 40 Komponentendiagramm des Equalizer-basierten Softwareentwurfs (eqOSG) 86
Bild 41 Originalbild Logo HS Wismar EuI [HS Wismar] 99
Bild 42 Bump Map Logo HS Wismar EuI 99

Bild 43 Vergleich der Ladezeiten Phase 3 CPU-CUDA 120
Bild 44 Vergleich der Ladezeiten Phase 3 CPU-OpenCL 121
Bild 45 Vergleich der finalen Messwerte 124
Bild 46 Anteil der Programmiersprachen am Gesamtaufwand 129
Bild 47 Vertex Shader-Architektur 143
Bild 48 Pixel Shader-Architektur 143
Bild 49 Stream Processor-Architektur 143

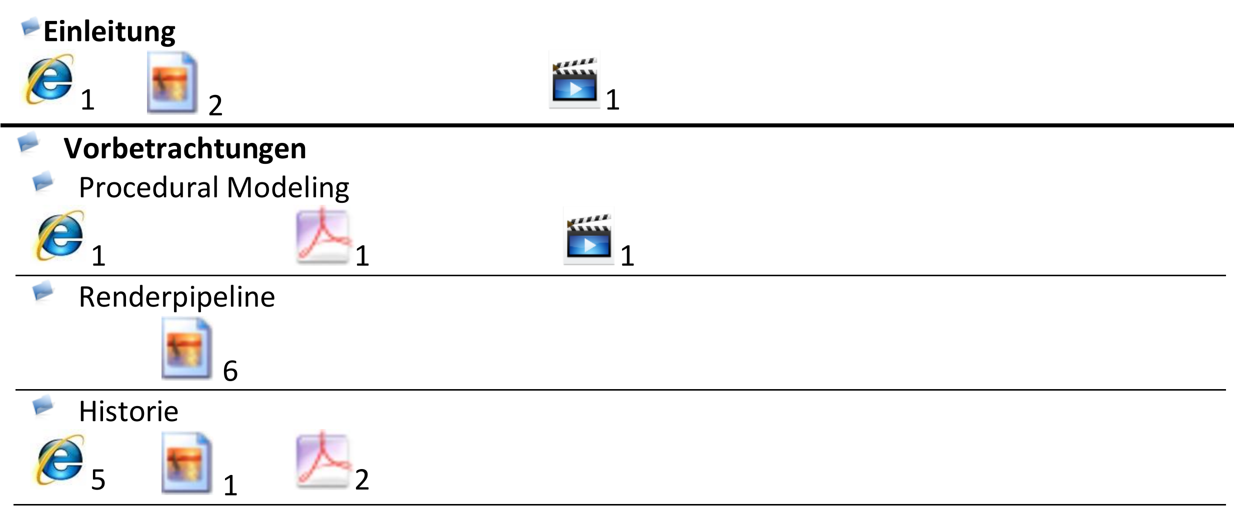
Tabellenverzeichnis

Tabelle 1 Grobvergleich ausgewählter Grafikkarten 17
Tabelle 2 Leistungsfähigkeit aktueller Grafikchips 32

Formelverzeichnis

Formel 1: Berechnung Bandbreite 29
Formel 2: Zusammensetzung Tangent Space-Matrix 38
Formel 3 Berechnung der Aufteilungsanzahl der GPUs 91

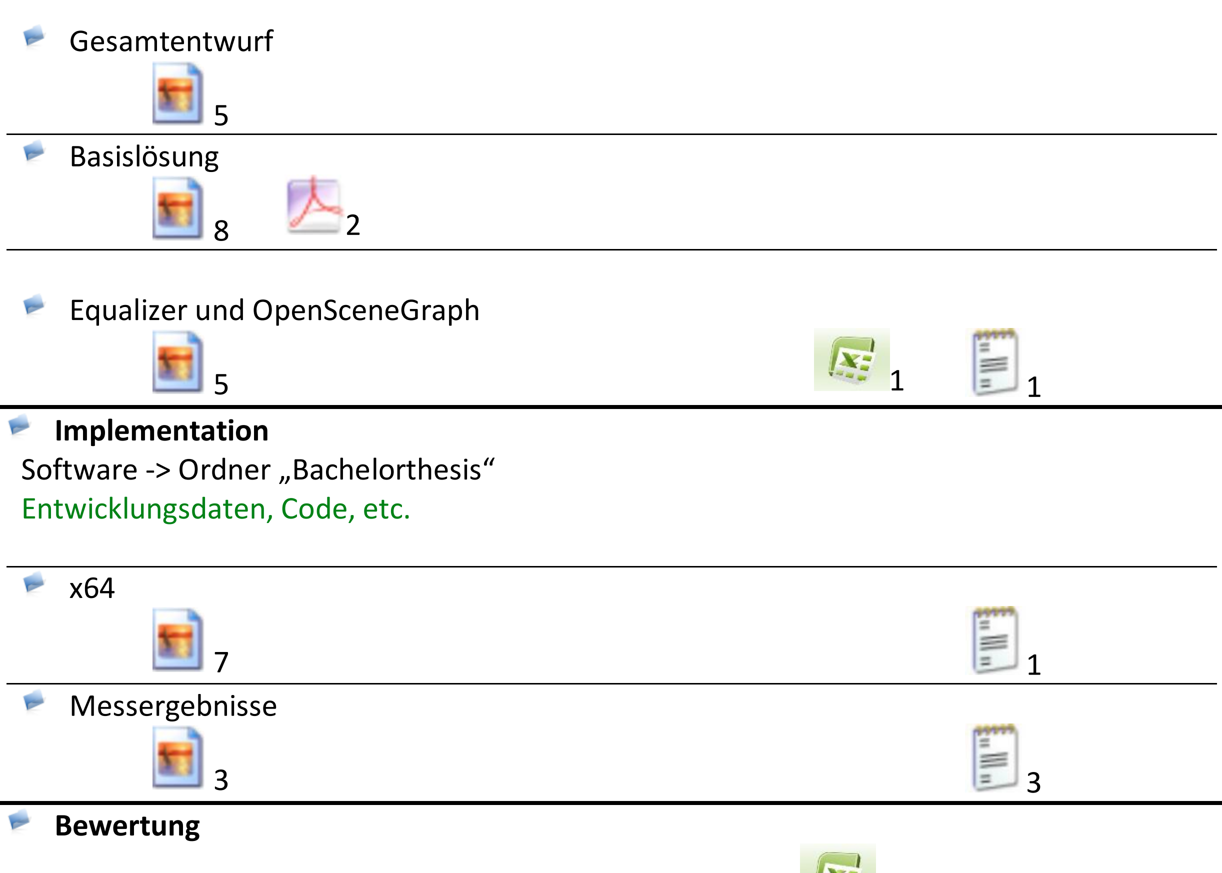
DVD-Verzeichnis



- Architektur
  2
- Prinzipien Grafikkomposition
  5 2 1
- Kennwerte
  1 2
- Beispiel-Apps
  1 13 3 2

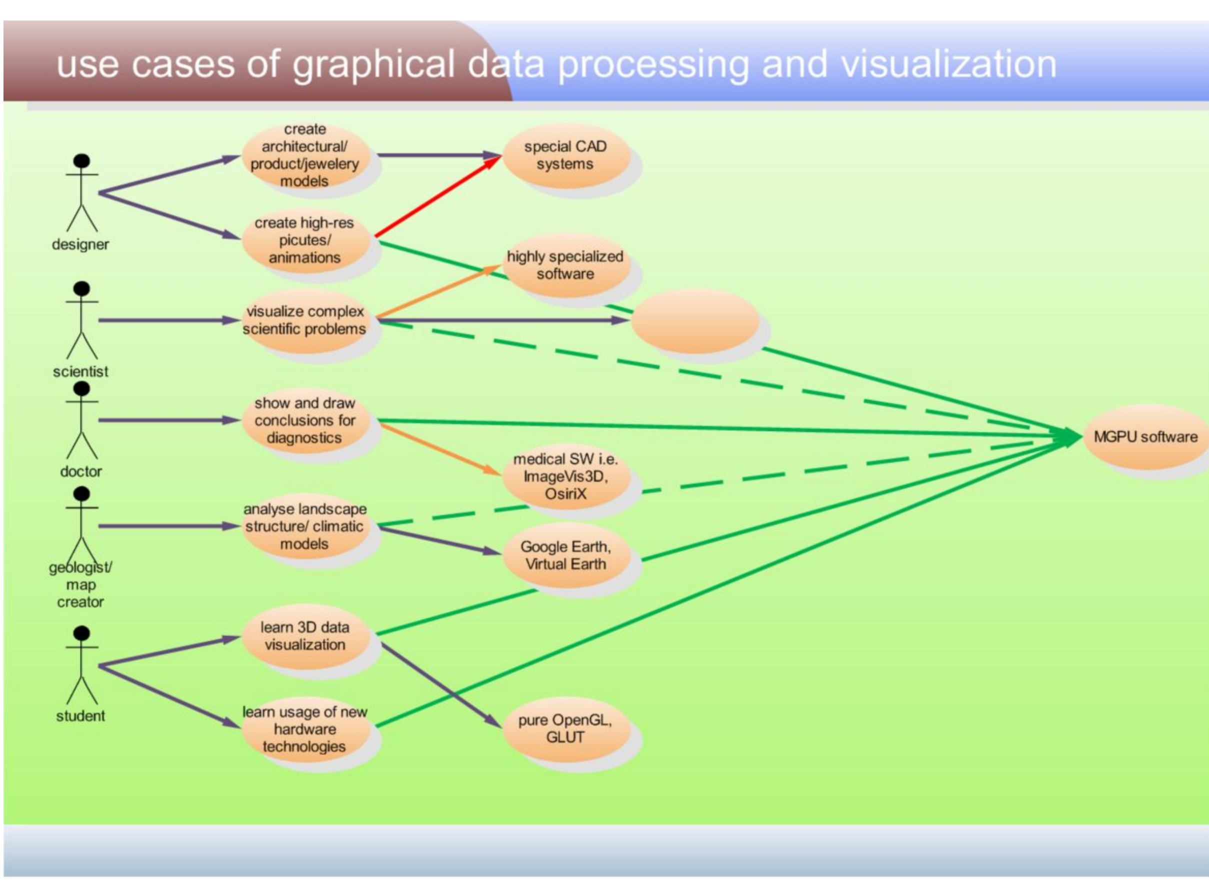
- **Ansätze**
- OpenGL
  1
- OpenCL
  1 2
- CUDA
  3 5
- CAL
  2
- ATI Stream
  1
- Chromium
  1
- Rapidmind
  7 2 1
- OpenSceneGraph
  7 1
- Visualization Library
  11
- Equalizer, SGI Multipipe SDK und SGI Performer
  2 4 5
- Tools
  6 1 1 1

- **Entwurf**


Gesamtentwurf
5
Basislösung
8
2
Equalizer und OpenSceneGraph
5
1
1
Implementation
Software -> Ordner „Bachelorthesis“
Entwicklungsdaten, Code, etc.
x64
7
1
Messergebnisse
3
3
Bewertung
1

Anhang

Allgemeine Fallstudie zur professionellen Nutzung der grafischen Datenverarbeitung



Leistungsprognose-Tabellen

GPU Prozessoren			
OHNE EQUALIZER = verbaute Kerne			
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	50	50
T2	200	100	100
T3	200	100	100
MIT EQUALIZER = unabhängige devices			
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	100	50
T2	100	100	50
T3	200	200	100

CPU Takt	(wichtig für Einlesegeschwindigkeit)		
	Frequenz [GHz]		
T1	1,73		
T2	2,33		
T3	2,33		
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	74,25	74,25
T2	134,68	100	100
T3	134,68	100	100

Grafik-RAM	Gesamt	Pro Kern	
	[Mbyte]	[Mbyte]	
Testsystem 1	256	256	
Testsystem 2	2048	1024	
Testsystem 3	1792	896	
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	25	28
T2	400	100	114
T3	350	87	100

Shadertakt			
	[MHz]		
T1	450		
T2	625		
T3	1404		
<Spalte> vs <	T1	T2	T3
T1	100	72	32
T2	138	100	44
T3	312	224	100

Compute Shader Number			
	cores		
T1	1		
T2	800		
T3	240		
<Spalte> vs <	T1	T2	T3
T1	100	0	0
T2	80000	100	333
T3	24000	30	100

GPU Takt			
	[MHz]		
T1	450		
T2	625		
T3	576		
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	72	78
T2	138	100	108
T3	128	92	100

Speichertakt	[MHz]		
T1	350		
T2	993		
T3	896		
<Spalte> vs <Zeile>	T1	T2	T3
T1	100	35	39
T2	283	100	110
T3	256	90	100

Arbeitsspeicher+GPU Anzahl+Grafik-RAM+GPU Takt					unabhängige Devices	
	T1	T2	T3			
T1	100,00	9,72	1,30			
T2	1012,92	100,00	45,10			
T3	7436,08	230,4864	100,00			

Arbeitsspeicher+GPU Anzahl+Grafik-RAM+GPU Takt					reine Kernanzahl	
	T1	T2	T3			
T1	100,00	4,86	1,66			
T2	2025,84	100,00	90,20			
T3	7436,08	115,26	100,00			

Shader-Architektur

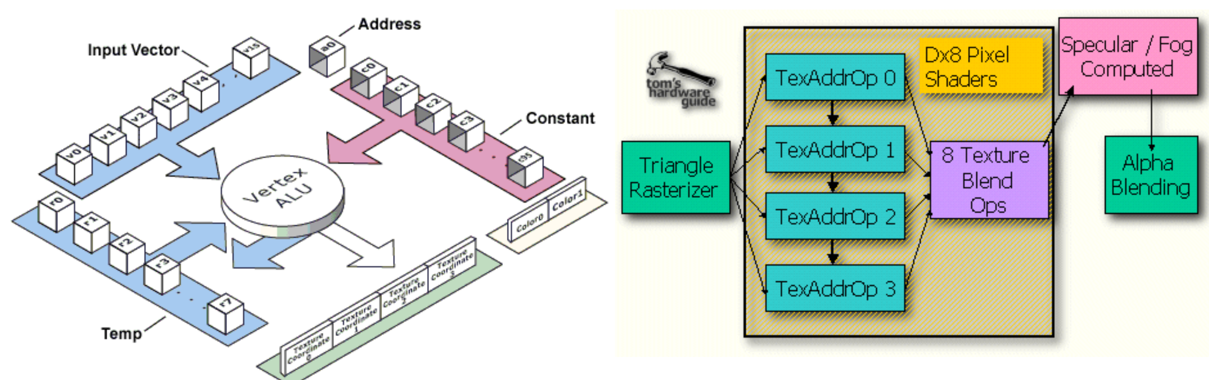


Bild 47 Vertex Shader-Architektur

Bild 48 Pixel Shader-Architektur

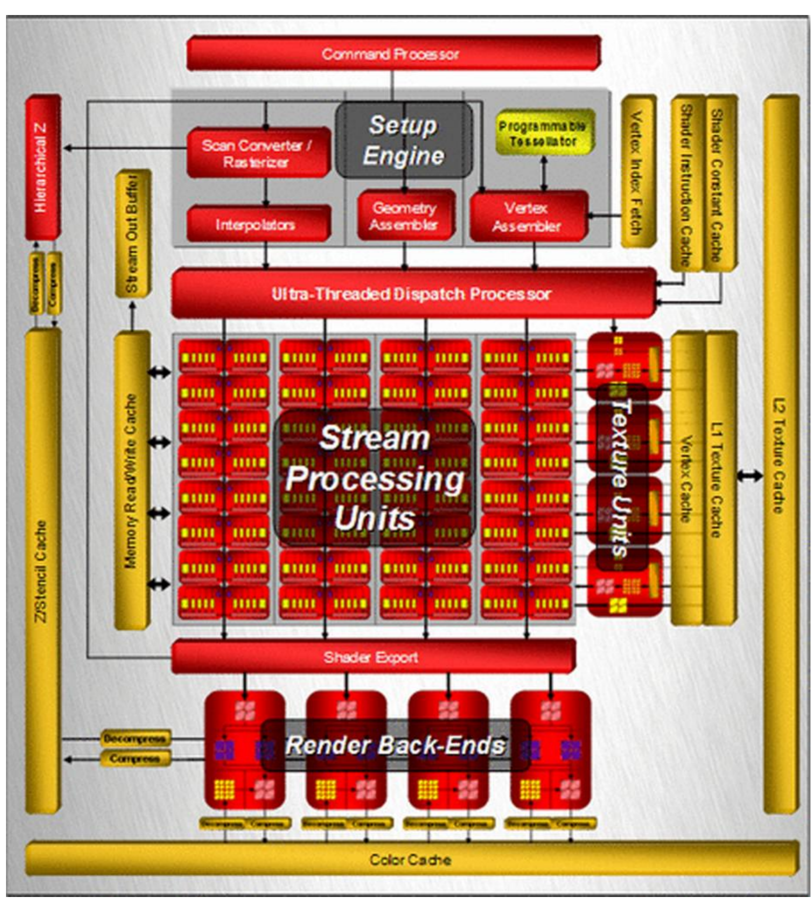
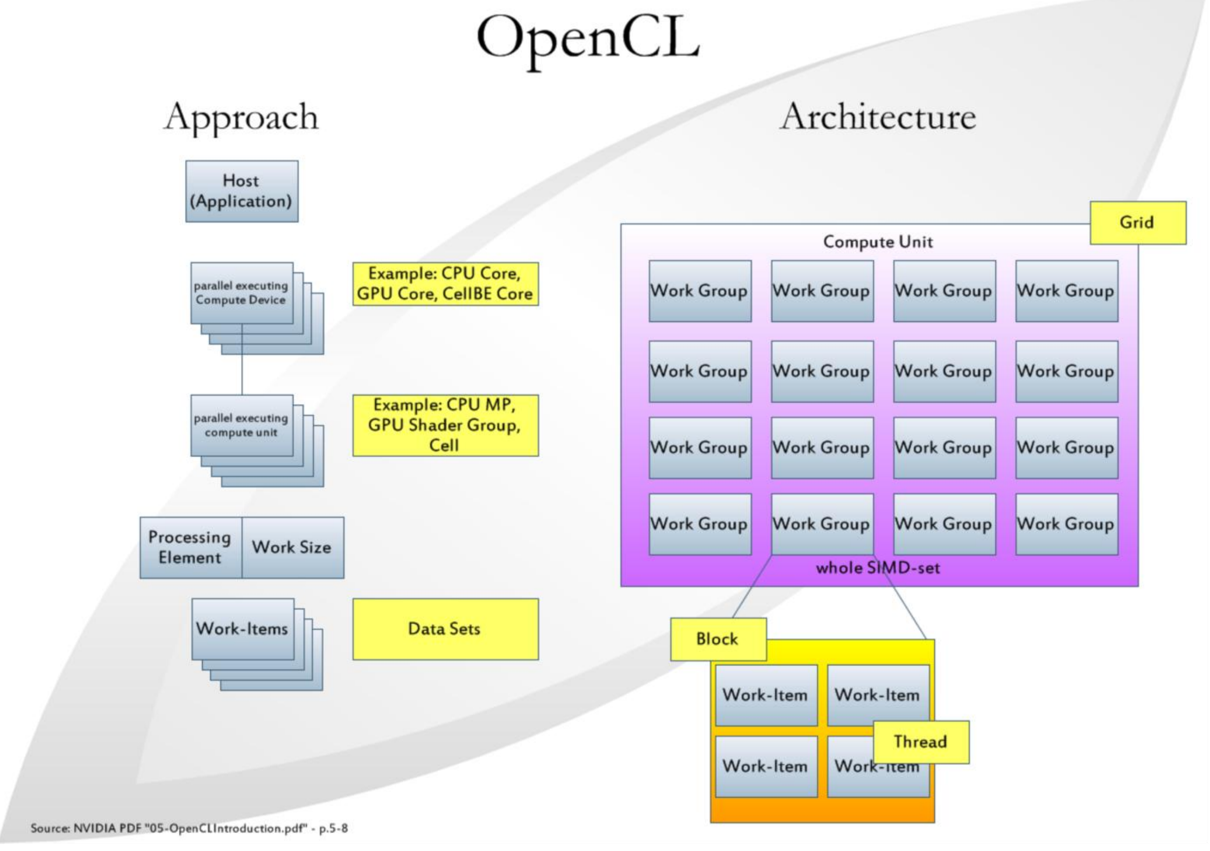
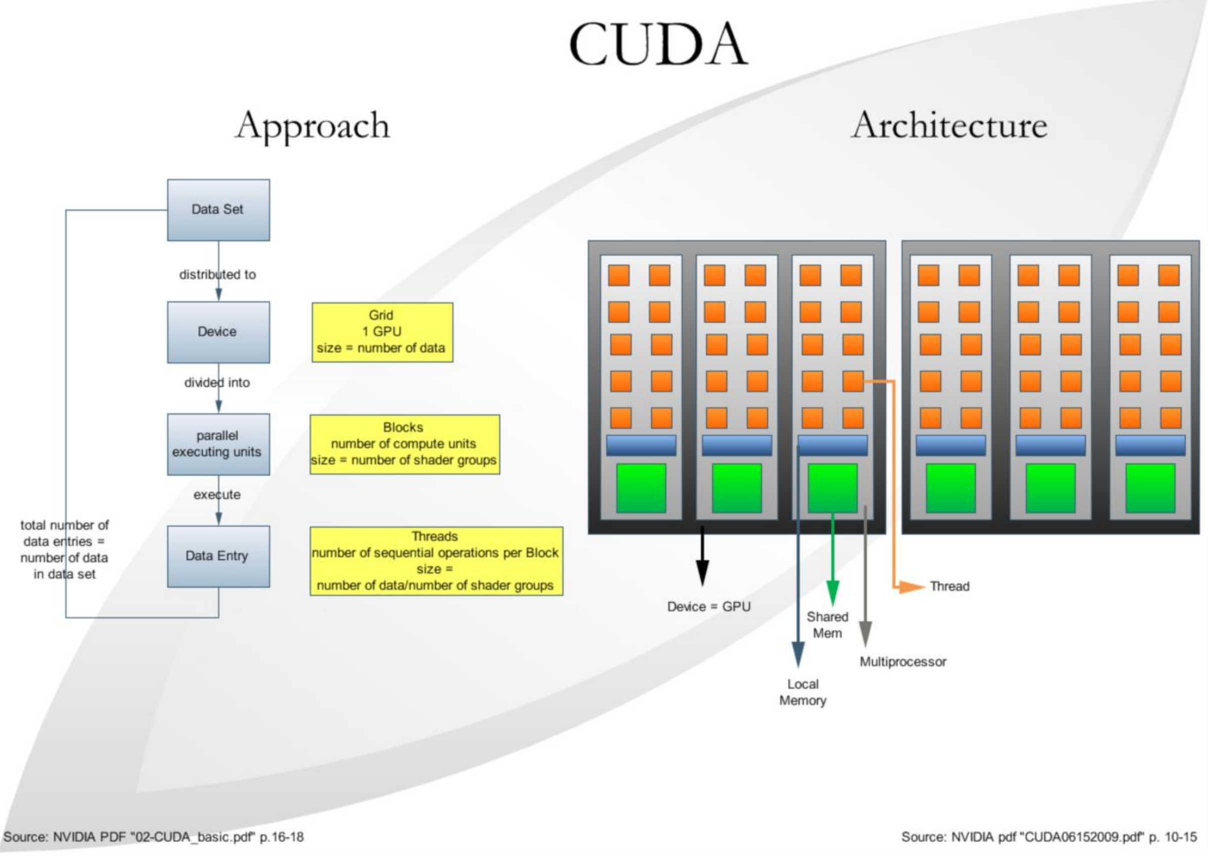


Bild 49 Stream Processor-Architektur

OpenCL Architekturskizze

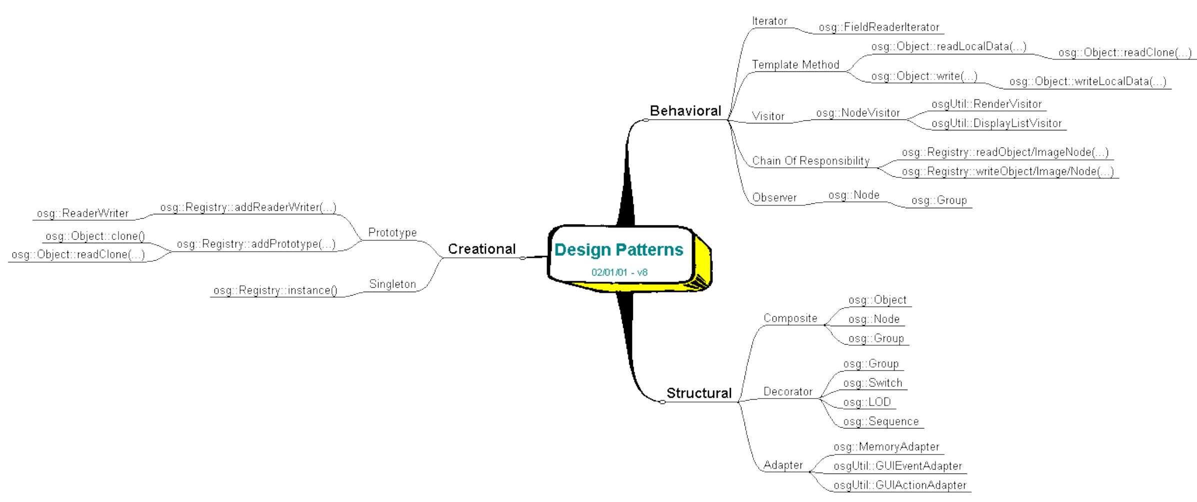


CUDA Architekturskizze



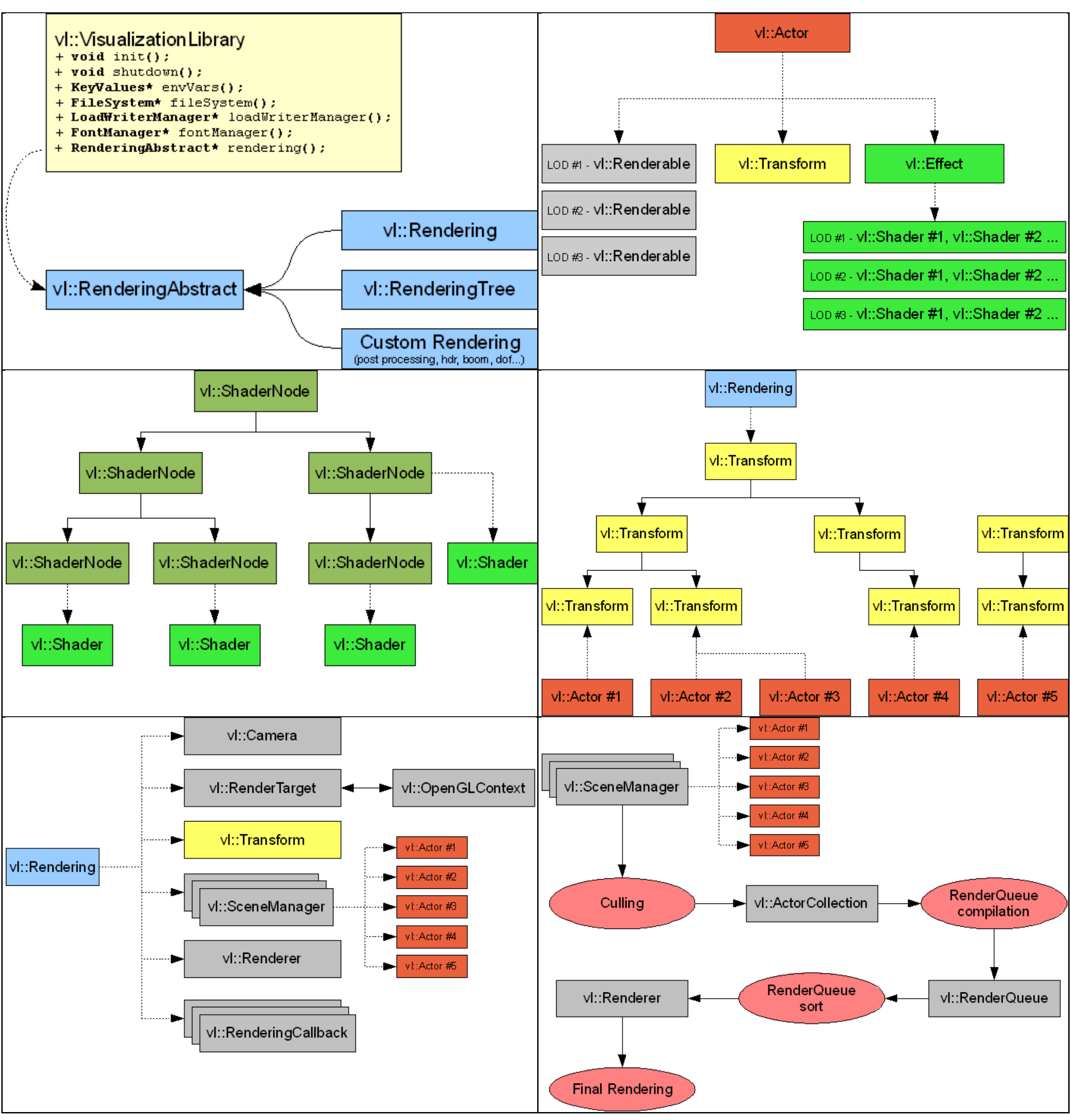
Open Scene Graph Entwurfsskizze

Diese Skizze stammt von der offiziellen OpenSceneGraph-Webseite.

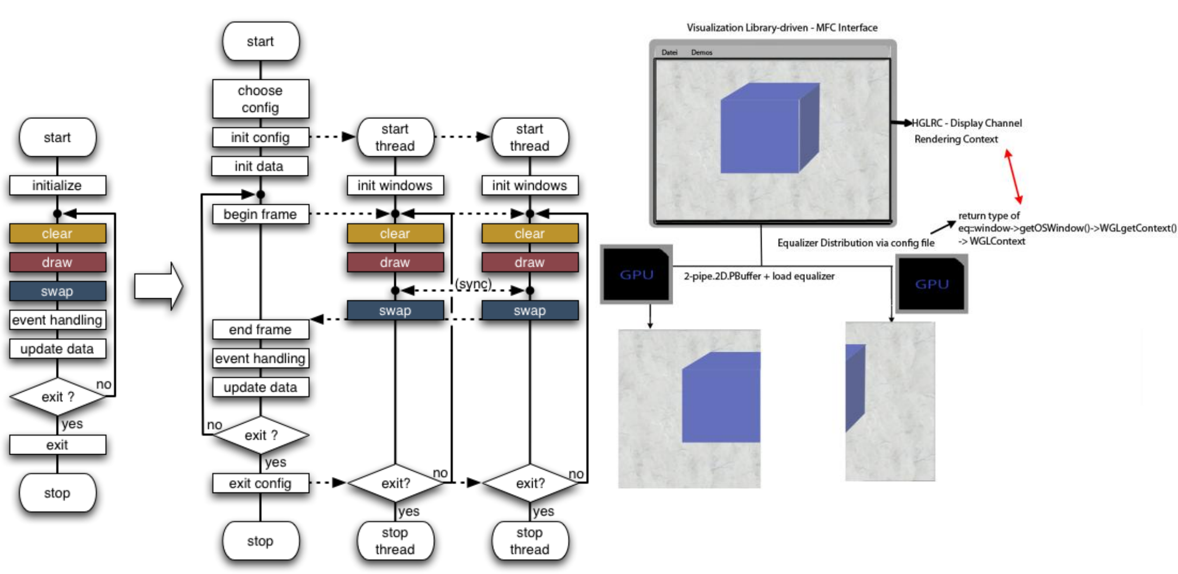


Visualization Library Architekturskizzen

Folgende Skizzen sind der offizellen VL-Webseite entnommen (www.visualizationlibrary.com)

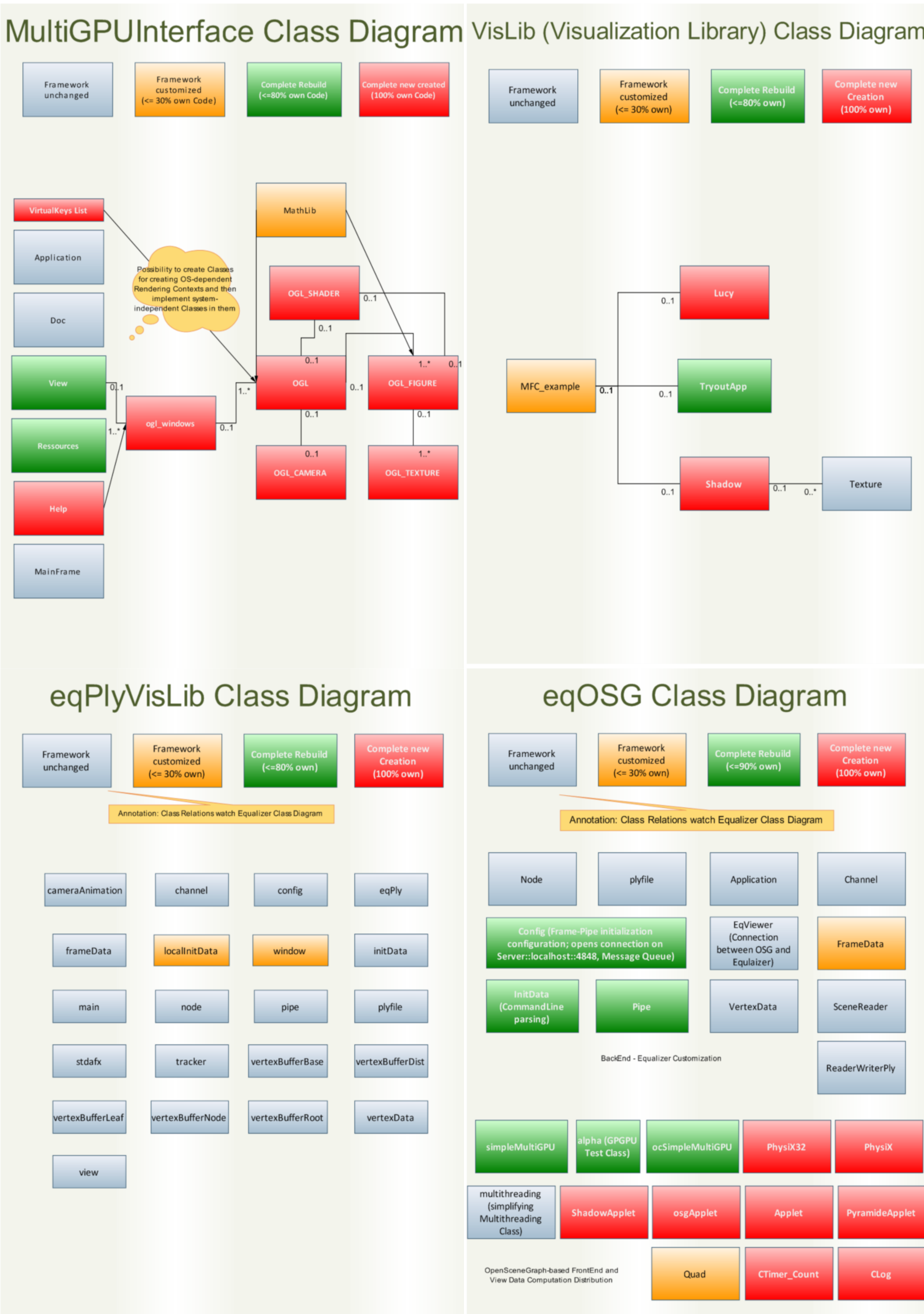


Equalizer-Skizzen

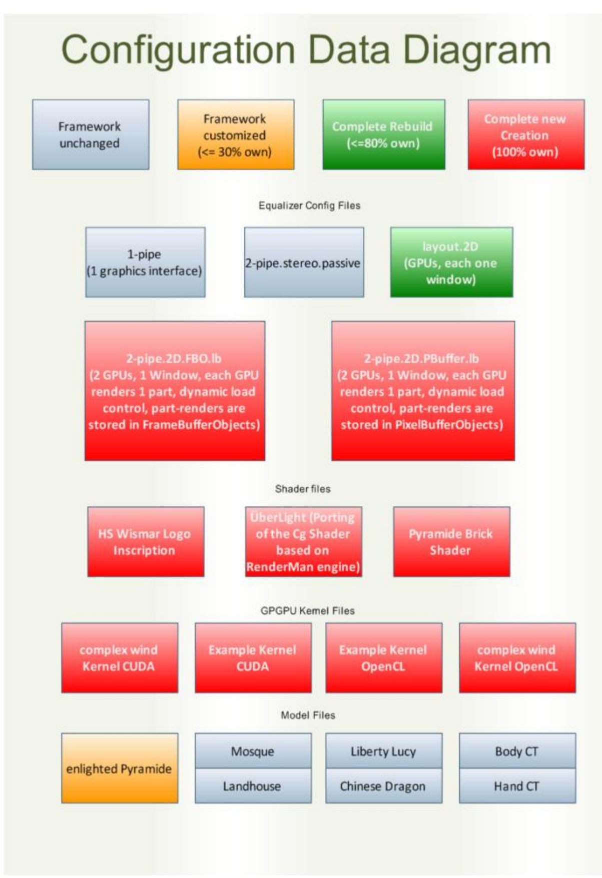


Gesamtaufbau der entwickelten praktischen Lösung

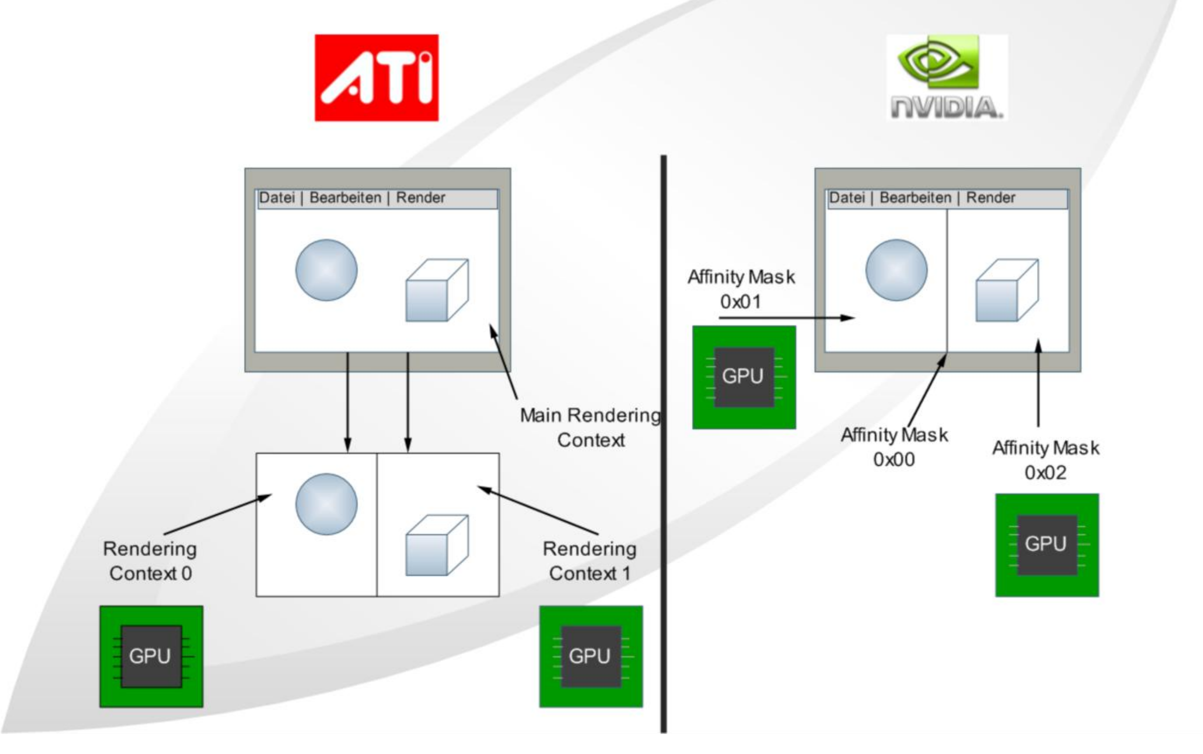
Folgende Skizzen stellen den Gesamtaufbau der Software dar. Dabei steht der Grad der Modifikation im Fokus. Dementsprechend sind die Klassen mit unterschiedlichen Farben markiert, je nach Anpassungsstufe.



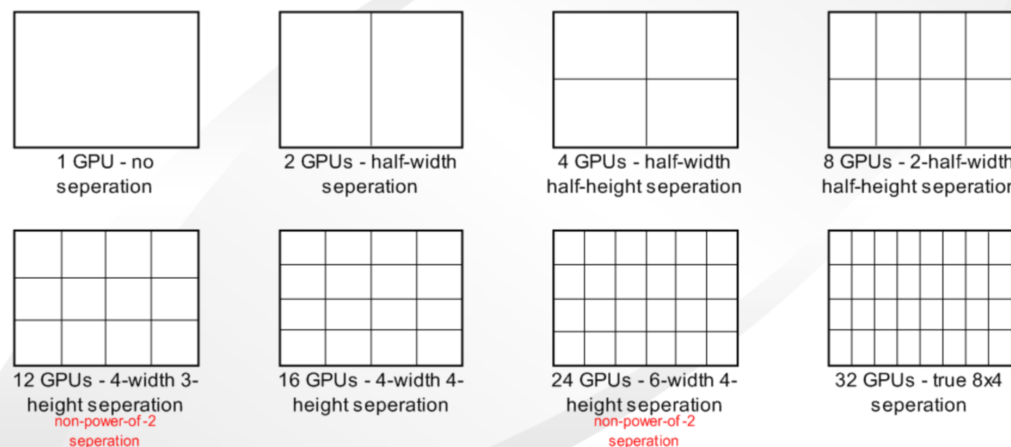
MultiGPUInterface Class Diagram
Framework unchanged
Framework customized (<= 30% own Code)
Complete Rebuild (<=80% own Code)
Complete new created (100% own Code)
VirtualKeys List
Application
Doc
View
Ressources
Help
MainFrame
Possibility to create Classes for creating OS-dependent Rendering Contexts and then implement system-independent Classes in them
ogl_windows
MathLib
OGL_SHADER
OGL
OGL_FIGURE
OGL_CAMERA
OGL_TEXTURE
VisLib (Visualization Library) Class Diagram
Framework unchanged
Framework customized (<= 30% own)
Complete Rebuild (<=80% own)
Complete new Creation (100% own)
MFC_example
Lucy
TryoutApp
Shadow
Texture
eqPlyVisLib Class Diagram
Framework unchanged
Framework customized (<= 30% own)
Complete Rebuild (<=80% own)
Complete new Creation (100% own)
Annotation: Class Relations watch Equalizer Class Diagram
cameraAnimation
channel
config
eqPly
frameData
localInitData
window
initData
main
node
pipe
plyfile
stdafx
tracker
vertexBufferBase
vertexBufferDist
vertexBufferLeaf
vertexBufferNode
vertexBufferRoot
vertexData
view
eqOSG Class Diagram
Framework unchanged
Framework customized (<= 30% own)
Complete Rebuild (<=90% own)
Complete new Creation (100% own)
Annotation: Class Relations watch Equalizer Class Diagram
Node
plyfile
Application
Channel
Config (Frame-Pipe initialization configuration; opens connection on Server::localhost::4848, Message Queue)
EqViewer (Connection between OSG and Equlaizer)
FrameData
InitData (CommandLine parsing)
Pipe
VertexData
SceneReader
BackEnd - Equalizer Customization
ReaderWriterPly
simpleMultiGPU
alpha (GPGPU Test Class)
ocSimpleMultiGPU
PhysiX32
PhysiX
multithreading (simplifying Multithreading Class)
ShadowApplet
osgApplet
Applet
PyramideApplet
OpenSceneGraph-based FrontEnd and View Data Computation Distribution
Quad
CTimer_Count
CLog


Configuration Data Diagram
Framework unchanged
Framework customized (<= 30% own)
Complete Rebuild (<=80% own)
Complete new Creation (100% own)
Equalizer Config Files
1-pipe (1 graphics interface)
2-pipe.stereo.passive
layout.2D (GPUs, each one window)
2-pipe.2D.FBO.lb (2 GPUs, 1 Window, each GPU renders 1 part, dynamic load control, part-renders are stored in FrameBufferObjects)
2-pipe.2D.PBuffer.lb (2 GPUs, 1 Window, each GPU renders 1 part, dynamic load control, part-renders are stored in PixelBufferObjects)
Shader files
HS Wismar Logo Inscription
UberLight (Porting of the Cg Shader based on RenderMan engine)
Pyramide Brick Shader
GPGPU Kernel Files
complex wind Kernel CUDA
Example Kernel CUDA
Example Kernel OpenCL
complex wind Kernel OpenCL
Model Files
enlighted Pyramide
Mosque
Landhouse
Liberty Lucy
Chinese Dragon
Body CT
Hand CT

Skizzen zur Basislösung und manueller Aufteilung der Renderpipeline

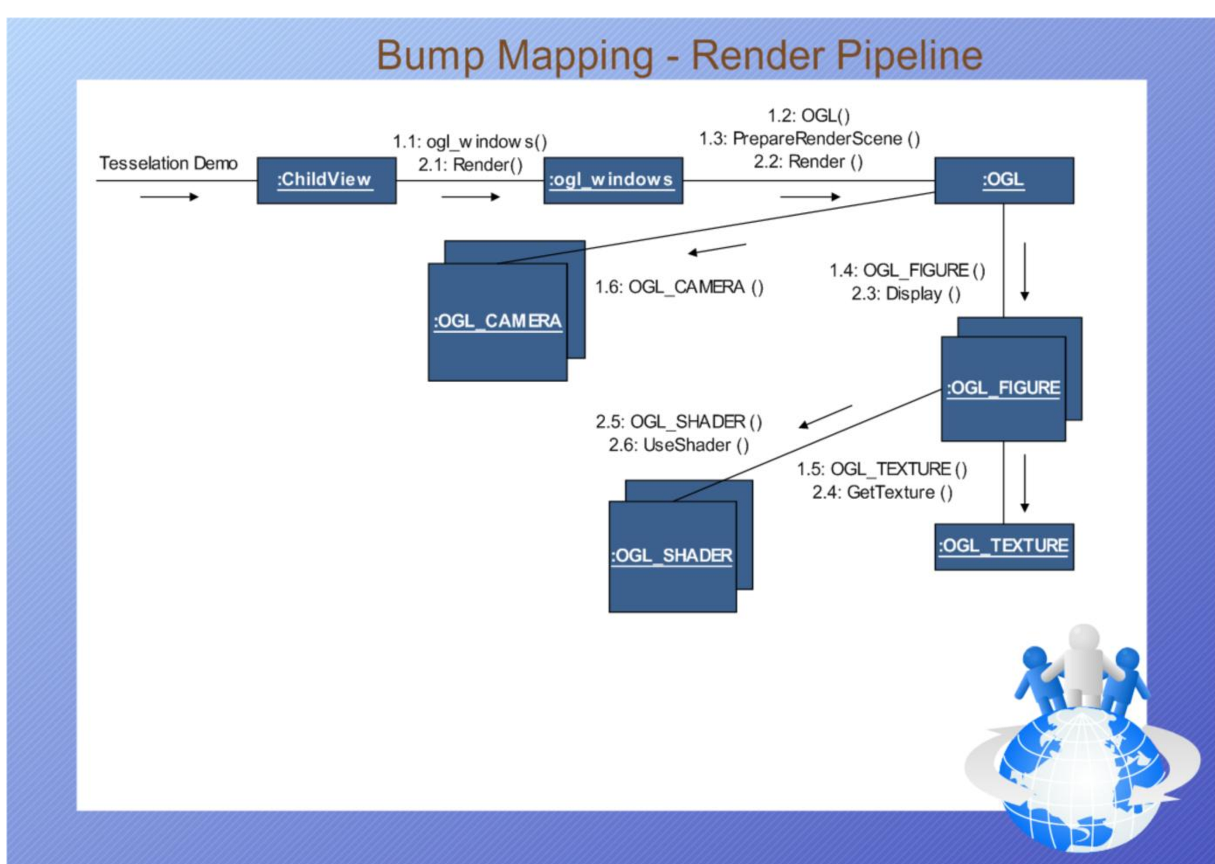


Numeric GPU seperation and distribution

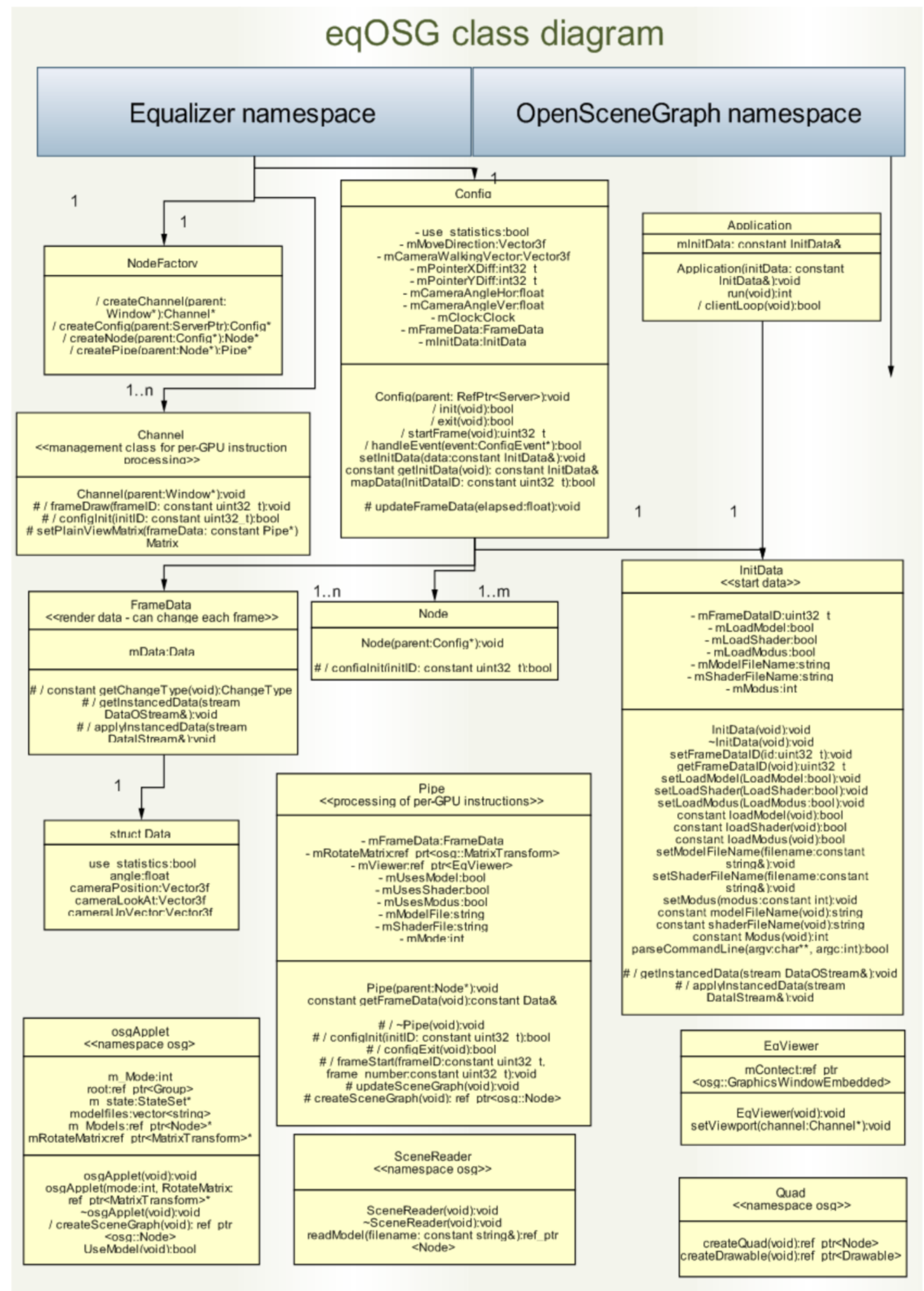


Conclusion:

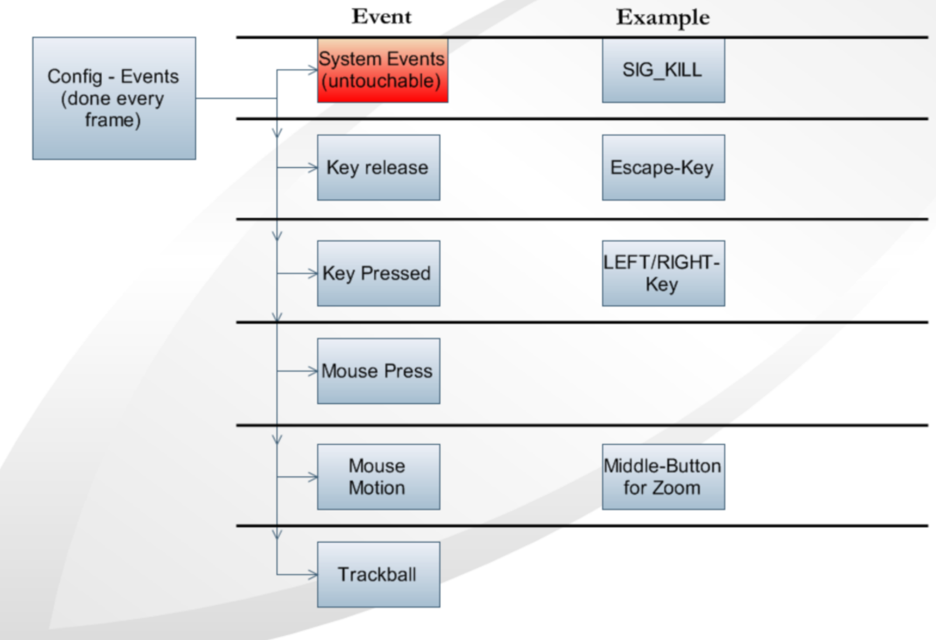
- non-power-of-2 seperation harder to compute and realize
- 12 GPU configuration meets extends for old 4:3 display standard - ideal solution for these screen
- non-power-of-2 seperations are looking more harmonic because width and height of each computation square are the same



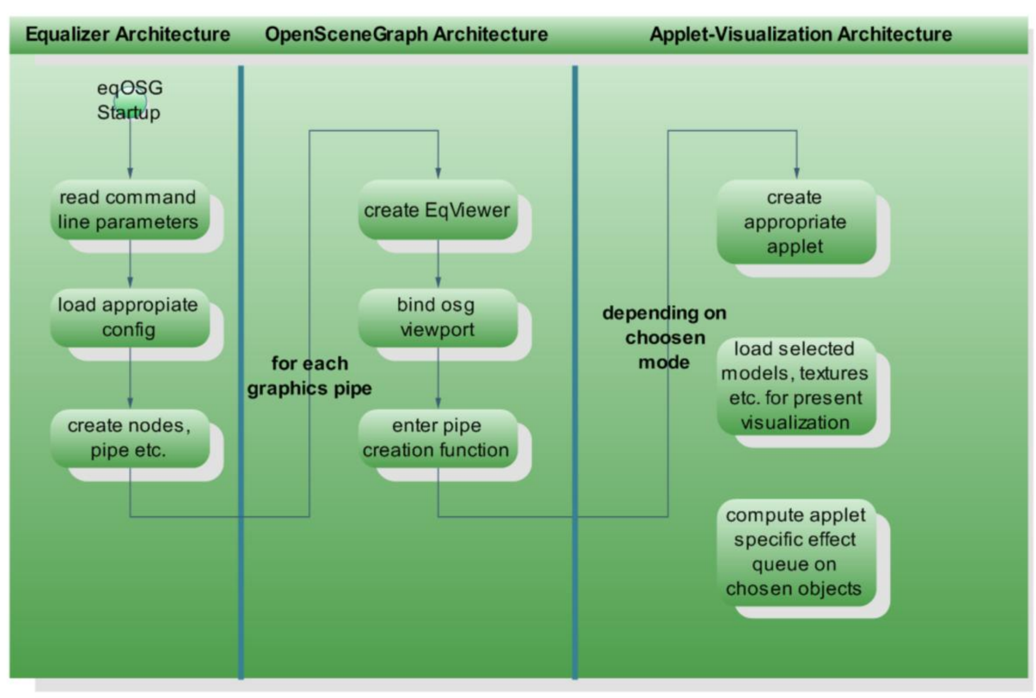
Equalizer- und eqOSG-Diagramme



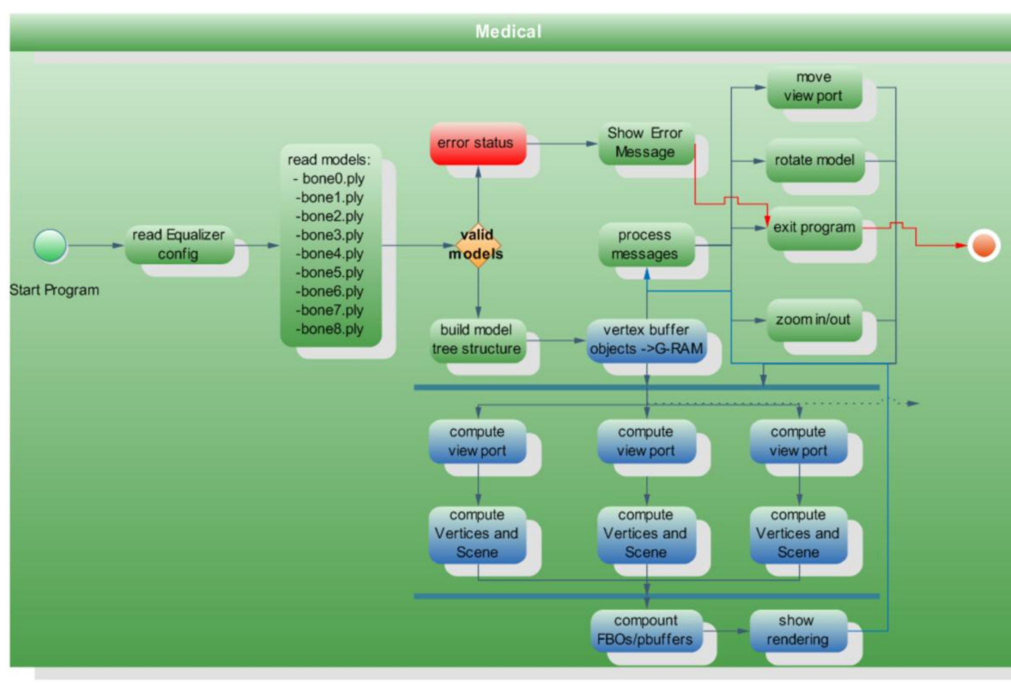
Event Queue in Equalizer



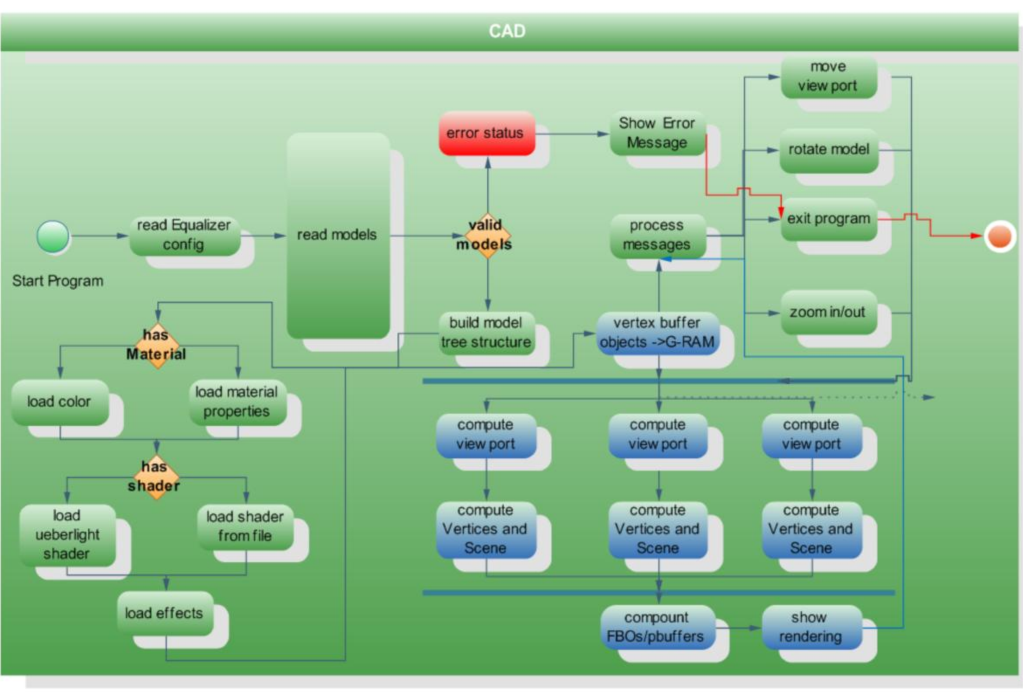
Applet Architecture

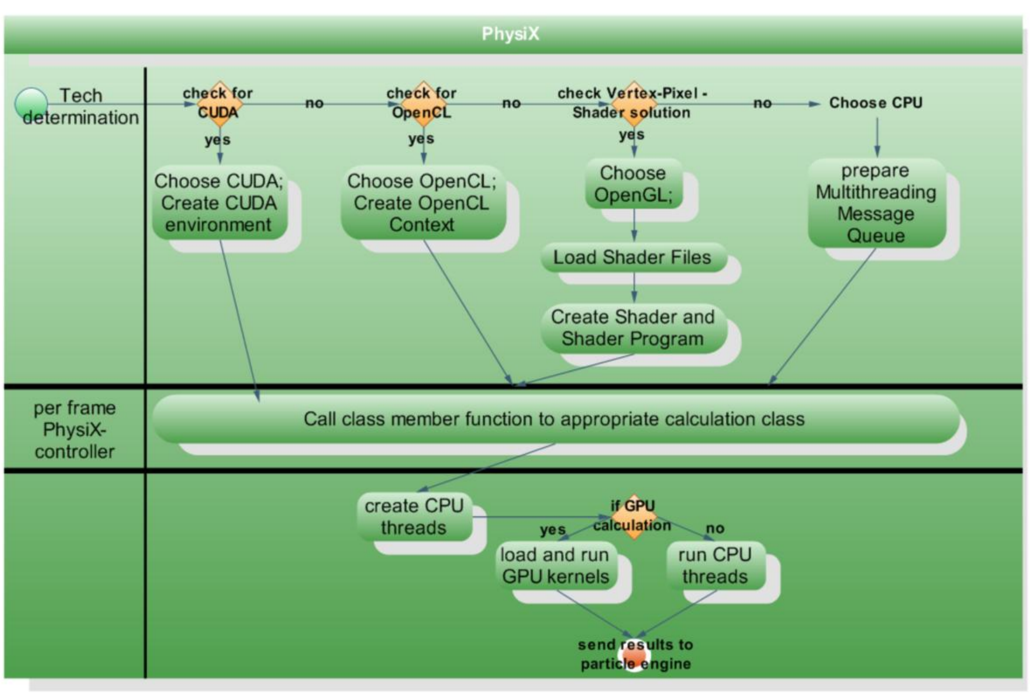


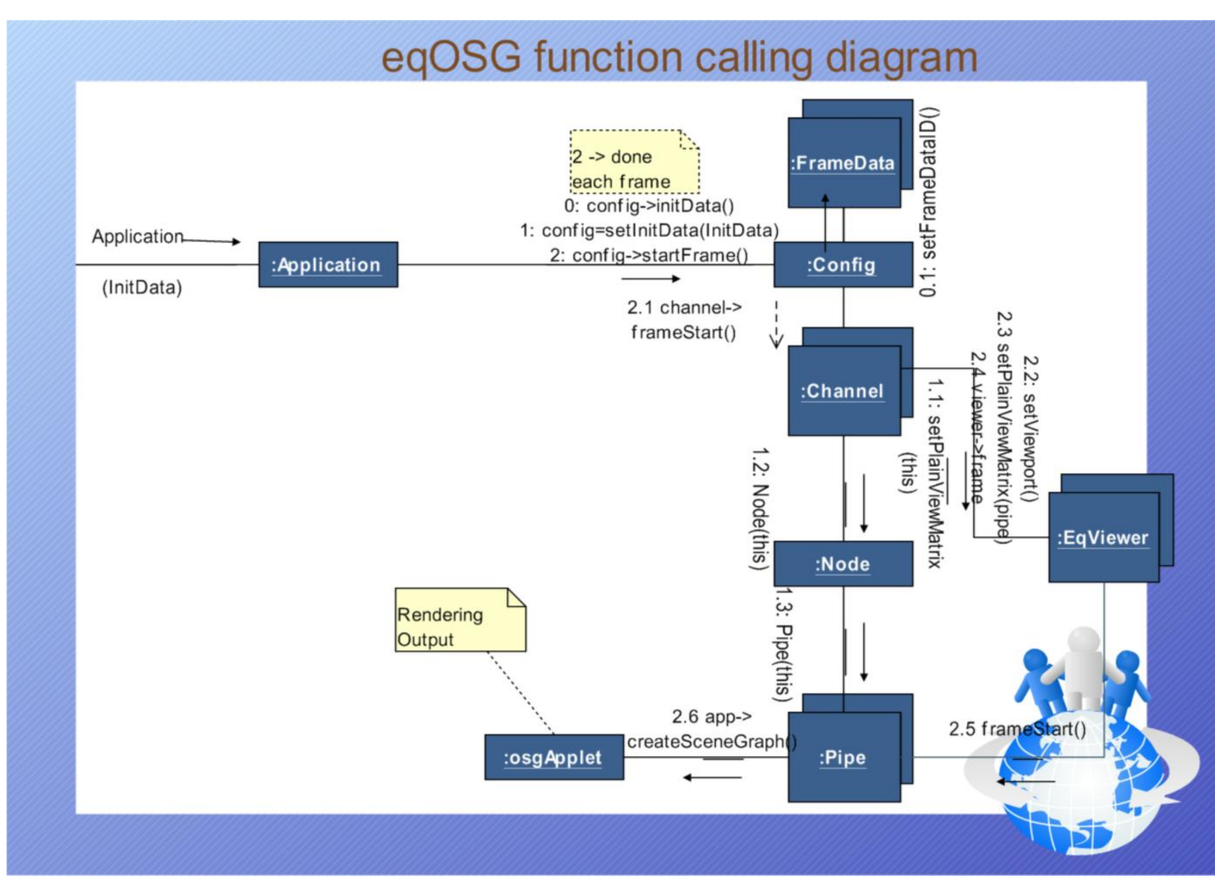
Medical Demo Activities

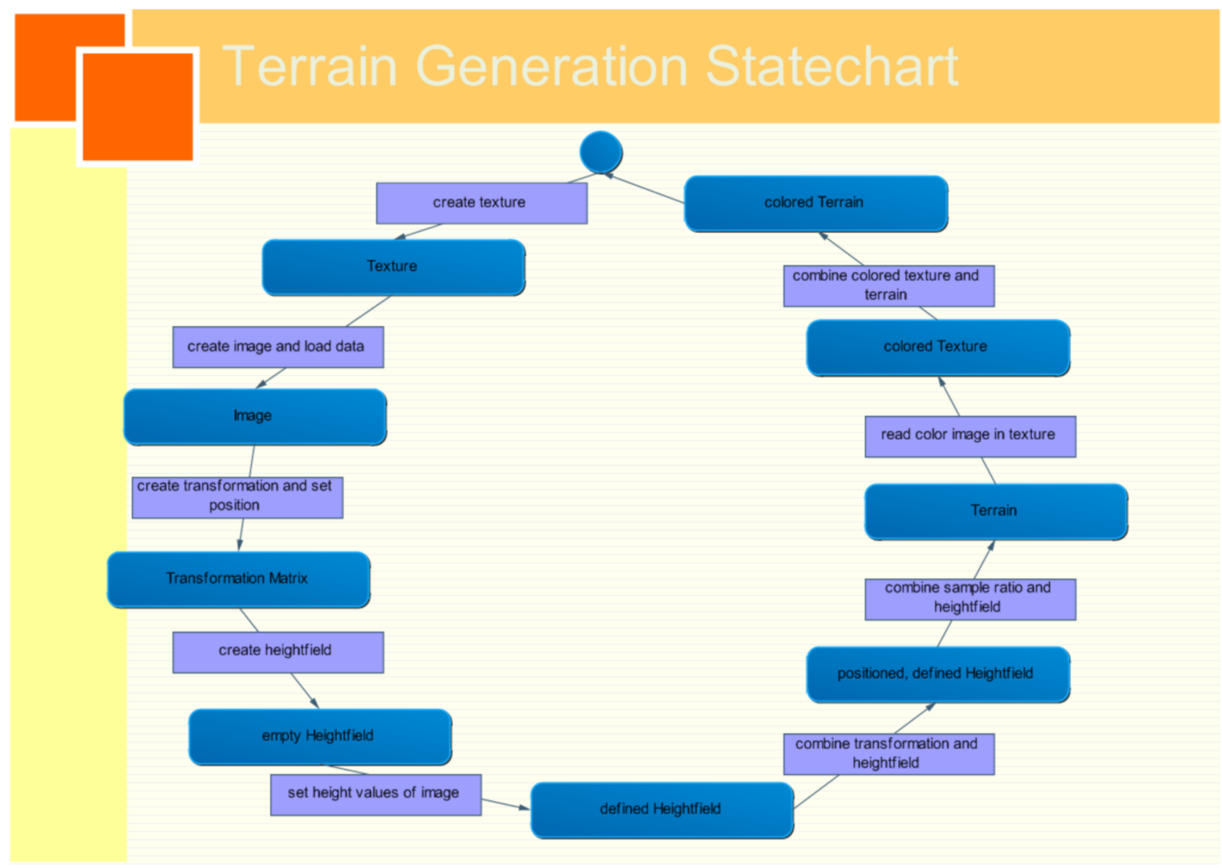


CAD Demo Activities




Physics Engine
PhysiX
Tech determination
check for CUDA
no
check for OpenCL
no
check Vertex-Pixel - Shader solution
no
Choose CPU
yes
yes
yes
Choose CUDA; Create CUDA environment
Choose OpenCL; Create OpenCL Context
Choose OpenGL;
prepare Multithreading Message Queue
Load Shader Files
Create Shader and Shader Program
per frame PhysiX-controller
Call class member function to appropriate calculation class
create CPU threads
if GPU calculation
yes
no
load and run GPU kernels
run CPU threads
send results to particle engine


eqOSG function calling diagram
2 -> done each frame
0: config->initData()
1: config=setInitData(InitData)
2: config->startFrame()
Application
(InitData)
:Application
:FrameData
:Config
0.1: setFrameDataID()
2.1 channel-> frameStart()
:Channel
2.2: setViewport()
2.3 setPlainViewMatrix(pipe)
2.4 viewer->frame
1.1: setPlainViewMatrix (this)
1.2: Node(this)
:Node
:EqViewer
Rendering Output
1.3: Pipe(this)
2.6 app-> createSceneGraph()
:osgApplet
:Pipe
2.5 frameStart()



Konzept Physikengine

Siehe Originalkonzept.

Handbuch zur Software

Siehe Handbuch.