UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Konzeption und Implementierung eines Unified Rendering Frameworks mit modernen GPU-Computing-APIs

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Markus Schlüter

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dipl.-Inform. Dominik Grüntjens

Koblenz, im Mai 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieser Bachelorarbeit wurde der Frage nachgegangen, inwiefern eine sogenannte „Unified Rendering-Engine", welche verschiedene Simulationsdomänen vereint, einen Mehrwert darstellen kann gegenüber dem klassischen Ansatz, z.B. gesondert sowohl eine Graphik- als auch eine Physik-Engine zu verwenden, die zunächst einmal keinen Bezug zueinander haben;

Hierbei wurde besonderer Wert auf die Verwendung moderner GPU-Computing-APIs gelegt, namentlich auf OpenGL3/4 und OpenCL.
Da bei diesem ganzheitlichen Thema eine vollständige Implementierung einer solchen vereinheitlichten Engine unmöglich war, konnte nur ein Bruchteil der Konzepte implementiert werden;

Dieser Umstand war von Vornherein bekannt, und die Versuchung ist stark, wie in einer Demo die schnelle Realisierung eines Feature-Sets einer konsistenteren, aber zeitaufwändigeren und zunächst karger wirkenden Implementierung vorzuziehen. Dieser Versuchung wurde versucht, nur dort nachzugeben, wo die negativen Auswirkung auf die Konsistenz des Gesamtsystems lokal bleiben, und so nicht „Hacks“ sich irreversibel durch das gesamte System ziehen.

Letztendlich wurden exemplarisch für die Nutzung in der visuellen Simulationsdomäne einige gängige visuelle Effekte einer Graphik-Engine implementiert, wie Shadow Mapping, Normal Mapping, Environment Mapping, Displacement Mapping und dynamisches LOD. Es wurden moderne OpenGL- und Hardware-Features wie Instancing, Uniform Buffers und Hardware-Tesselation verwendet. Schwerpunkt war hier der Einsatz einer Template-Engine, damit

1. Boilerplate-Code in den Shadern vermieden wird und
2. Effekte beliebig (sinnvoll) nach Möglichkeit zur Laufzeit miteinander kombinierbar sind

. Mehr dazu in Kapitel 4.1
In der mechanischen Simulationsdomäne wurde eine partikelbasierte Fluidsimulation mit OpenCL auf Basis von Smoothed Particle Hydrodynamics implementiert. Mehr dazu in Kapitel 4.2.

Das System trägt den Namen „Flewnit“, eine bewusst nicht auf den ersten Blick erkennbar sein sollende[1] Kombination der Worte „Fluid“, in

[1] Es soll der generalistische Ansatz des Frameworks nicht in den Hintergrund gedrängt werden.

Anspielung auf den urspünglichen Zweck einer Bibliothek zur Fluidsimulation und „Unit“, in Anspielung auf „Unity“-„Einheit“. Zufälligerweise ist das Nit auch noch die englische Einheit für die Leuchtdichte, $\frac{Cd}{m^2}$.

1.1 Motivation

Ursprünglich als Arbeit zur Implementierung einer Fluidsimulation geplant, wurde bald ein generischer, eher softwaretechnisch orientierter Ansatz verfolgt, der jedoch die Implementierung einer Fluidsimulation als mittelfristiges Ziel hatte;

1.1.1 „Unified Rendering Engine“

Der Wunsch nach einer „Unified Rendering Engineërwächst aus eigener Erfahrung der Kopplung von Physik- und Graphik-Engines, namentlich der Bullet Physics Library[2] und der OGRE Graphik-Engine[3]. Diese Hochzeit zweier Engines, die jeweils für verschiedene „Simulationsdomänen“ zuständig sind, bringt gewissen Overhead mit sich, da Konzepten wie Geometrie und ihrer Transformationen unterschiedliche Repräsentationen bzw. Klassen zugrunde liegen; Hierdurch wird die gemeinsame Nutzung beider Domänen von Daten wie z.B. Geometrie nahezu unmöglich; Ferner müssen für eine die beiden Engines benutzende Anwendung diese Klassen mit ähnlicher Semantik durch neue Adapterklassen gewrappt werden, um dem Programmier der eigentlichen Anwendungslogik den ständigen Umgang mit verschiedenen Repräsentationen und deren Synchronisation zu ersparen.

Die Aussage „Photorealistische Computergraphik ist die Simulation von Licht“ hat mich wohl auch inspiriert, den Simulationsbegriff allgemeiner aufzufassen und das Begriffspaar "Rendering und Physiksimulation“ zu hinterfragen[4].

Es sei bemerkt, dass weder eine Hypothese bestätigt noch widerlegt werden sollte, geschweige denn überhaupt eine (mir bekannte) Hypothese im Vorfeld existierte; Es sprechen etliche Argumente für eine Vereinheitlichung der Konzepte (geringerer Overhead durch Wegfall der Adapterklassen, evtl. Speicherverbrauch durch z.T. gemeinsame Nutzbarkeit von Daten), aber auch einige dagegen (Komplexität eines Systems, Anzahl an theoretischen Kombinationsmöglichkeiten steigt, viele sind unsinnig und müssen implizit oder explizit ausgeschlossen werden).

[2] http://bulletphysics.org

[3] http://www.ogre3d.org

[4] Auch wenn dieses Framework nicht vornehmlich auf physikalisch basierte, also photorealistische Beleuchtung ausgelegt ist, soll diese aufgrund des generischen Konzepts jedoch integrierbar sein.

Für mich persönlich bringt die Bearbeitung dieser Fragestellung zahlreiche Vorteile; Ich muss ein wenig ausholen:
Schon als Kind war ich begeistert von technischen Geräten, auf denen interaktive Computergraphik möglich war; Sie sprechen sowohl das ästhetische Empfinden an, als auch bieten sie eine immer mächtigere Ergänzungs- und Erweiterungsmöglichkeit zu unserer Realität an; Letztendlich stellten diese Geräte für mich wohl auch immer ein Symbol dafür dar, in wie weit die Menschheit inzwischen fähig ist, den Mikrokosmos zu verstehen und zu nutzen, damit demonstriert, dass sie zumindest die rezeptiven und motorischen Beschränkungen seiner Physiologie überwunden hat.
Die Freude an Schönheit und Technologie findet für mich in der Computergraphik und der sie ermöglichenden Hardware eine Verbindungsmöglichkeit; Die informatische Seite mit seinen Algorithmen als auch die technische Seite mit seinen Schaltungen faszinieren mich gleichermaßen; Auch das „große Ganze"der Realisierung solcher Computergraphischen Systeme, das Engine-Design mit seinen softwaretechnischen Aspekten, interessiert mich.
Ferner wollte ich schon immer "die Welt verstehen", sowohl auf physikalisch-naturwissenschaftlich-technischer, als auch - aufgrund der system-immanenten Beschränkungen unseres Universums - auf metaphysischer Ebene[5];

Und hier schließt sich der Kreis: Sowohl in der Philosophie als auch in der Informatik spielt das Konzept der Abstraktion eine wichtige Rolle; Nichts anderes tut eine „Unified Engine": sie abstrahiert bestehende Konzepte teilgebiets-spezifischer Engines, wie z.B. Graphik und Physik; Ich erhoffe mir, dass mit dieser Abstraktion man in seinem konzeptionellen Denken der der realen Welt ein Stück weit näher kommt; Die verfügbaren Rechenressourcen steigen, die Komplexität von Simulationen ebenfalls; Ob eine semantische Generalisierung von seit Jahrzehnten verwendeten Begriffen wie „Rendering“ und „Physiksimulation“, welche dieser Entwicklung angemessen sein soll, eher hilfreich oder verwirrend ist, kann eine weitere Interessante Frage sein, die ich jedoch nicht weiter empirisch untersucht habe.

Letzendlich verbindet dieses Thema also viele meiner Interessen, welche die gesamte Pipeline eines Virtual-Reality-Systems, vom Konzept einer Engine bis hin zu den Transistoren einer Graphikkarte, auf sämtlichen Abstraktionsstufen betreffen: Es gab mir die Möglichkeit,

- den Mehrwert einer Abstraktion gängiger Konzepte von Computergraphik und Physiksimulation zu erforschen
- die Erfahrungen im Engine-Design zu vertiefen
- die Erfahrungen im (graphischen) Echtzeit-Rendering zu vertiefen

[5] ob der Begriff „Verständnis“ im letzten Falle ganz treffend ist, bleibt Ermessens-Sache

- mich mit Physiksimulation (genauer: Simulation von Mechanik) zu beschäftigen, konkret mit Fluidsimulation
- mich in OpenGL 3 und 4 einzuarbeiten, drastisch entschlackten Versionen der Graphik-API, deren gesäuberte Struktur die Graphikprogrammierung wesentlich generischer macht und somit die Abstraktion erleichtert
- mich in OpenCL einzuarbeiten, den ersten offenen Standard für GPGPU[6]
- mich intensiver mit Graphikkarten-Hardware, der zu Zeit komplexesten und leistungsfähigsten Consumer-Hardware zu beschäftigen, aus purem Interesse und um die OpenCL-Implementierung effizienter zu gestalten

Die vielseitigen didaktischen Aspekte hatten bei dieser Themenwahl also ein größeres Gewicht als der Forschungsaspekt, was dem Ziel einer Bachelorarbeit angemessen ist.

1.1.2 Fluidsimulation

Warum ich mich bei der exemplarischen Implementierung einer mechanischen Simulationsdomäne für eine partikelbasierte Fluidsimulation entschieden habe, hat viele Gründe:

Wohingegen Rigid Body-Simulation in aktuellen Virtual-Reality-Anwendungen wie Computerspielen schon eine recht große Verbreitung erreicht hat, sucht man eine komplexere physikalisch basierte Fluidsimulation, die über eine 2,5 D- Heightfield-Implementation hinausgehen, noch vergebens; Das Ziel, eine deratige Fluidsimulation in einen Anwendungskontext zu integrieren, der langfristig über den einer Demo hinausgehen soll, hat also einen leicht pionierhaften Beigeschmack.

Mir ging es um eine Simulation, welche die Option eriner möglichst breiten Integration in die virtuelle Welt bietet; Wohingegen sich Grid-basierte Verfahren aufgrund Möglichkeit zur Visualisierung per Ray-Casting sehr gut zur Simulation von Gasen eignen, sind Partikel-basierte Verfahren eher für Liquide geeignet, da bei Grid-basierten Verfahren die Volumen-Erhaltung eines Liquids durch zu Instabilität und physikalischer Inplausibilität neigenden Level-Set-Berechnungen sichergestellt werden muss. Liquide beeinflussen aufgrund ihrer Dichte Objekte ihrer Umgebung im Alltag mechanisch stärker als Gase; Aufgrund dieser erhöhten gegenseitigen Beeinflussung von Fluid und Umgebung bevorzugte ich das Verfahren, welches Liquide besser simuliert.

[6]General Purpose Graphics Processing Unit- Computing, die Nutzung der auf massiver Paralleltät beruhenden Rechenleistung von Graphikkarten in nicht explizit Graphikrelevanten Kontexten

Die Partikel-Domäne bietet außerdem eine theoretisch unendlich große Simulationsdomäne, wohingegen in der Grid-Domäne der Simulationsbereich auf das Gebiet beschränkt ist, welches explizit durch Voxel repräsentiert ist. Ferner lassen sich relativ einfach auch Rigid Bodies durch einen partikelbasierten Simlator simulieren, indem eine Repräsentation der Geometrie des Rigid Bodies als Partikel gewählt wird;

Bei grid-basierten Verfahren lässt sich die Simulations-Domäne z.B. als Sammlung von 3D-Texturen oder nach einem bestimmten Schema organisierten 2D-Texturen repräsentieren; Hiermit wird die Simulation auf der GPU mithilfe von Graphik-APIs wie OpenGL ohne weiteres möglich, und wurde auch schon erfolgreich implementiert . Mein Anliegen war jedoch, explizit die Features moderner Graphikhardware und sie nutzender GPGPU-APIs wie Nvidia CUDA, Microsoft's DirectCompute oder OpenCL zu verwenden; Die Partikel-Domäne stellt damit eine größere Abgrenzung zu gewohnten Workflows auf der GPU dar. Vor allem die *Scattered Writes*, die Graphik-Apis nicht oder nur sehr indirekt ermöglichen, und die von so vielen Algorithmen benötigt werden, sollten zum Einsatz kommen dürfen.

Die Fluidsimulation stellt einen relativ „seichten" Einstieg in die Welt der GPU-basierten Echtzeit-Physiksimulation dar:

- Es gibt schon zahlreiche Arbeiten zur Fluidsimulation, welche erfolgreich den Spagat zwischen Echtzeitfähigkeit und physikalischer Plausibilität gemeistert haben (siehe Kapitel 4.2.1.2);
- Fluide sind für gewöhnlich ein homogenes Medium, daher eignet sich bei Partikel-Ansatz für die Suche nach Nachbarpartikeln die Beschleunigungsstruktur des Uniform Grid besonders gut, wohingegen sich für Simulation von Festkörpern eher komplexere Beschleunigungsstrukturen wie Oct-Trees, Bounding Volume-Hierarchies oder kD-Trees anbieten, da diese sich besser an die inhomogenen Strukturen, Ausmaßen und Verteilungen der Objekte anpassen. ; Letztere Strukturen lassen sich schwerer auf die GPU mappen, welche als Stream-Prozessor nicht optimal für komplexe Kontrollflüsse und Datenstrukturen geeignet ist.
- Die Partikel lassen sich direkt als OpenGL-Vertices per Point Rendering darstellen, was während der Entwicklungsphase eine einfache Visualisierungmöglichkeit bietet;
- Die gemeinsame Nutzung von Geometrie sowohl zur mechanischen Simulation als auch zur Visulalisierung mit OpenGL ist hiermit ermöglicht. OpenCL ermöglicht diese gemeinsame Nutzung explizit über gemeinsame Buffer-Nutzung.
- Nicht zuletzt ist die Mathematik bei Partikel-Simulation einfacher: Die *„eulersche Sicht"* auf die Simulationsdomäne beim Grid-Ansatz

erfordert einen Advektionsterm, der dank der *„lagrange'schen Sicht"* bei Partikeln wegfällt; Außerdem erfordern Partikelsysteme keine Berechnungen zur Sicherstellung der Inkompressibilität [7] oder Bewahrung des Volumens.

[7] Das heißt nicht, dass die Inkompressibilität automatisch gewährleistet ist; Im Gegenteil hat man öfter mit „Flummi-artigem" Verhalten des Partikel-Fluids zu tun, weil diese eben *nicht* ohne weiteres forcierbar ist;

2 Überblick

2.1 Vision

Die langfristige Vision, die *Flewnit* begleitet, ist die Entwicklung eines interaktiven Paddel-Spiels unter Verwendung dieser Unified Engine mit ausgefeilter Fluid-Mechanik und -Visualisierung, partikelbasierten Rigid Bodies und Dreiecks-Mesh als Repräsentation für statische Kollisions-Geometrie; Spiele, in der große Mengen Fluid, die komplexer simuliert sind als durch Height-Fields [8] einen integrativen Bestandteil der Spielmechanik ausmachen, sind mir nicht bekannt;
Von Dreiecks-Geoemetrie erhoffe ich mir eine genauere Repräsentation zur Kollisionsbehandlung, bei gleichzeitiger Ersparnis vieler Partikel, die sonst z.T große Oberflächen repäsentieren müssten; Ferner könnte die Dreiecksstruktur später zur Simulation nicht-partikelbasierter Rigid Bodies verwendet werden;

2.2 Paradigmen

Vor dem Entwurf eines komplexen Softwaresystems mit einigen Zügen, die in etablierten Systemen keine so große Bedeutung haben, hat es Sinn, sich einige Paradigmen zu überlegen, welchen das System nach Möglichkeit folgen soll, um eine gewisse Konsitenz zu gewährleisten:

- Es wurde beim Entwurf der Unified Engine für jede Simulationsdomäne eine möglichst ähnliche Struktur von Klassen und ihren Beziehungen zueinander angestrebt. Diese Ähnlichkeit spiegelt sich nach Möglichkeit in einer gemeinsamen (manchmal abstrakten) Oberklasse eines jeden Konzeptes wider, wie z.B.:
  - dem Simulations-Objekt als solchem
  - der Geometrie
  - dem Material
  - der Szenen-Repräsentation

  Auf diese Weise soll eine maximale *Symmetrie* zwischen den Domänen hergestellt werden, so dass domänen-bedingte Spezial-Behandlung von Objekten und Workflows minimiert wird;
- Es sollte eine Art Pipeline-Architektur entstehen, wo bestimmte Pipeline-Stages bestimmte Simulations-(Zwischen)- Ergebnisse implemetieren, und ggfs. anderen Stages diese zur Verfügung stellen. Jede Simulationsdomäne hat seine eigene Pipeline; Dennoch können Interdependenzen bestehen;

[8] s. Kapitel 4.2.1.2 für mehr Informationen zu Height-Field-basierter Fluidsimulation

Diesen Interdependenzen wird durch eine Konzept-spezifische Verwaltung durch verschiedene Singleton-Manager-Klassen genüge getan; Ein und dasselbe Objekt kann von verschiedenen Managern in unterschiedlichem Zusammenhang verwaltet werden; Mehr dazu in Kapitel 3;

- Es sollen langfristig so viele Features (Visualisierungstechniken und -effekte, Simulationstechniken) wie möglich miteinander kombinierbar sein, sofern die Kombination nicht unsinnig ist;
- Es soll so viel wie möglich auf der GPU berechnet werden, um die massive Paralleltät auszunutzen, und um nicht durch Buffer-Transfers, die die Aufteilung von Algorithmen in CPU- und GPU- Code meist mit sich bringen, auf den Bandbreiten- und Latenz- Flaschenhals der PCI-Express-Schnittstelle zu stoßen;
- Es soll immer das Potential gewahrt werden, dass aus dem Framework — außerhalb des Rahmens dieser Bachelorarbeit — tatsächlich noch eine Art *Unified Engine* entstehen kann; Somit sind „schnelle Hacks“, also unsaubere Programmier-Weisen, die mit geringstem Programmier-Aufwand ein bestimmtes Feature implementieren, überall dort unbedingt zu vermeiden, wo sie die konsistente Gesamstruktur des Systems zu bedrohen scheinen.

2.3 Begriffe

Im Zuge der angestrebten Vereinheitlichung der verschiedenen Simulation müssen wir auch einige Begriffe verallgemeinern, welche in ihrer jahrzentelangen Tradition in der Terminologie der Computergraphik eine spezifische Bedeutung erhalten haben; Zur besserern Einordnung stellt Abbildung 1 ein grobes Schema dar, welches die klassische Verwendung verschiedener Engines und die einer Unified Engine gegenüber stellt:

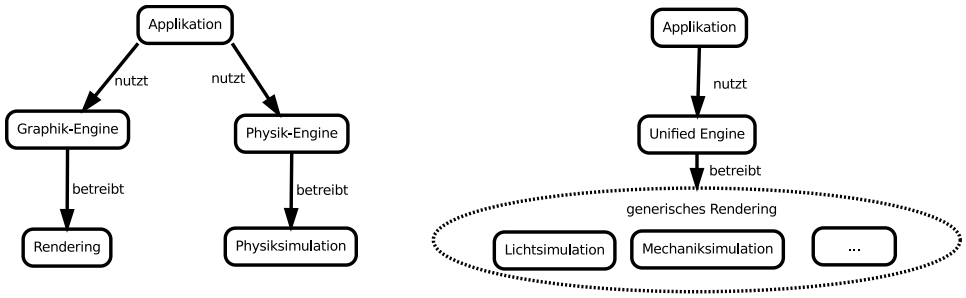

Abbildung 1: Gegenüberstellung von Verwendung und Begrifflichkeiten von klassischen Engines und einer Unified Engine

Rendering Im Wiktionary [**?**] wird das Verb *to render* u.a. umschrieben als:

> „(transitive, computer graphics) To transform digital information in the form received from a repository into a display on a computer screen, or for other presentation to the user.
> “

Es geht also um die Transformation einer formalen Beschreibung in eine für einen menschlichen Benutzer wahrnehmbare Form. Diese muss entgegen der gewöhnlichen Verwendung des Begriffes nicht zwingend visuell, sondern kann z.B. auch akustischer oder haptischer Natur sein, übertragen durch Lautsprecher oder Force-Feedback-Devices.

Verallgemeinern wir den Begriff *Rendering* weiter, gemäß der Übersetzung der Verb-Form als *erbringen*, *machen* [**?**], und in Anlehnung an seine Ethymologie,

> „From Old French *rendre* (“to render, to make”)“ [...] [**?**]

bietet sich eine freie Übersetzung als *Erzeugung eines Zustandes beliebiger Natur* an;
Unter diese generische (Um)-Deutung des Begriffes fällt nun auch *die Ausführung beliebig gearteter Simulation*.

Zu besseren Abgrenzung kann man von *generischen Rendering* und dem klassischen *visuellen Rendering* sprechen; Dies soll im weiteren Verlauf dieser Arbeit der Fall sein.
Eine *Unified Engine* (s.u.) betreibt also *generisches Rendering*.

Unified Engine Alternativ-Bezeichnung: *Unified Rendering Engine*;
Eine *Unified Engine* betreibt *generisches Rendering*, indem sie bestimmte Aspekte einer *Welt*[9] simuliert. Darunter kann das klassische (visuelle) Rendering fallen, aber auch die Simulation von Geräuschen und von Mechanik, und beliebige weitere Domänen; Die Domänen sollen dabei durch Abstraktion gemeinsamer Eigenschaften so ähnlich wie möglich organisiert sein;

Simulation Das Begriffspaar *Rendering* und *Physiksimulation* ist im Kontext dieser versuchten Vereinheitlichung nicht mehr angemessen; Stattdessen sollten wir den Simulations-Gedanken aufgreifen und anstelle von *Rendering* lieber von *Licht-Simulation* sprechen; Auf diese Weise werden Missverständliche abwechselnde Verwendung vom Begriff *Rendering* vermieden;
Der Begriff der *Physiksimulation* ist auch nicht ganz sauber, da streng

[9] Diese Welt muss dabei nicht zwingend unserer Realität ähneln oder entsprechen.

genommen Licht auch ein physikalisches Phänomen ist, und somit vom Begriff eingeschlossen wird, statt sich abzugrenzen; Es bietet sich die alternative und genauere Bezeichnung *Mechanik-Simulation* an; Die Quantenmechanik vor Augen (der Name spricht für sich) und damit den Umstand, dass auch Photonen an mechanischen Vorgängen teilnehmen, ist zwar selbst dies keine saubere Abgrenzung, aber auf dem angestrebten Niveau einer plausiblen (im Kontrast zur „korrekten“) Echtzeit-Simulation, welche mit der Newton'schen Physik auskommen wird, ist diese Abgrenzung klar genug.

Somit ist die erste Symmetrie zwischen den Simulationsdomänen durch eine Anpassung der Terminologie bewerkstelligt.

2.4 Schwerpunkte

Die Entwicklung eines solchen *Unified Frameworks*[10] umfasst sehr viele Aspekte, und einige werden wohl in dieser Ausarbeitung keine Erwähnung finden. Um die didaktischen Ziele von Seite 3 nicht aus den Augen zu verlieren, wurden folgende Schwerpunkte gesetzt:

2.4.1 Entwicklungs-Umgebung

Nach Anfängen unter Windows 7 und Visual Studio 2010, gab es bald Probleme beim Compilen von Dependencies auf 64 Bit; Womöglich wäre es eine Frage der Geduld gewesen, jedoch habe ich dann zu Ubuntu Linux[11] und Eclipse[12] in Kombination mit dem Cross-Platform Build-System CMake[13] gewechselt. Mir war es sehr wichtig, ein natives 64-Bit-System zu entwickeln, da viele Register der heutigen 64Bit-Prozessoren sonst ungenutzt bleiben. Das Paket-Management der Linux-basierten Betriebsysteme und die konsequente Implementierung fast aller Programme in 64 Bit erleichtern das Einrichten der Dependencies ungemein; Schon alleine dafür hat sich der Umstieg gelohnt.[14]

[10] Von Engine möchte ich Kontext der Implementierung im Rahmen dieser Bachelorarbeit noch nicht sprechen, da dieser Begriff eine viel zu große Vollständigkeit der Implementation suggeriert.

[11] http://www.ubuntu.com/

[12] http://www.eclipse.org/

[13] http://www.cmake.org/

[14] Einen sehr sehr großen Dank möchte ich an dieser Stelle Lubosz Sarnecki aussprechen, der mich mit meiner zuvor sehr eingeschränkten Linux-Erfahrung unermüdlich mit Profi-Support bei der fortgeschrittenen Customization des Betriebsystems versorgt hat; Ohne ihn wäre mir dieser schnelle, weitgehend reibungslose Umstieg nicht gelungen.

2.4.2 Dependencies

Das Endziel einer potenten, modernen Engine sollte auf keinen Fall durch die Wahl suboptimaler Bibliotheken eingeschränkt werden; Andererseits sollten, um Compile- und Link-Zeiten gering zu halten und Konflikte zwischen Bibliotheken zu vermeiden, die Dependencies nicht zu komplex sein; Die Wahl vor allem des Fenster-Managers und der Mathematik-Bibliothek musste deshalb mit Bedacht getroffen werden; Mehr dazu in Kapitel .

2.4.3 Nutzung moderner OpenGL-Features

Uniform Buffers, Tesselation, hardware instancing, verweis auf entpsrechende section;

2.4.4 Implemetation und Kombination gängiger visueller Effekte

-vor allem tesselation, verweise auf entsprechende section

2.4.5 Buffer-Abstraktion

2.4.6 Template-Engine

boilerplate, kombinierbarkeit, nach Möglichkeit lesbarkeit
exemplarischer code schnipsel - Im Zuge des Schwerpunktes auf GPU-Implementierung: grantlee gegen boilerplate,zur generierung schlankerer programme als durch Präprozessordierktiven, $\rightarrow$ einfachere Code- Inspection, verbesserung der lesbarkeit durch generierte, feature-spezifische programme, bessere struktur verwaltung des source codees

2.4.7 Performance duch Implementierung auf der GPU mit modernen GPU-Computing-APIs

auf die massive parallelität eingehen, die sowohl von visualler wie mechanischer domäne genutzt werden kann; Performance-Schwerpunkt, Optimierung, auch hardware-abhängige, erwähnen, Gegenüberstellung zu alten OpenGL-nutzenden GPUPU- Verfahren, die nicht scattern konnten in texturen rendern mussten und auch sonst etliche Nachteile hinnehmen mussten

2.4.8 Effiziente Verwendung von OpenCL

hardware-spezifische bedingte compilings dank grantlee

2.4.9 Weiteres

für vielseitig, flexible anwendung zur Laufzeit sollten keine Speicher-Lecks auftreten damit Funktionalität kontrolliert heruntergefahren und neu initialisiert werden kann; - memory tracking, (erklären, warum nicht tracking mit Valgrind)

Für gemeinsamen zugriff sollten viele Daten für andere Klassen verfügbar sein (Buffer, Rendering Results...); Realisierung über Manager-Singleton-Klassen und Zugriff über Map-Container;

- möglichst hohe Konfiguriertbarkeit ohne ständigen recompile: config file

3 Systemarchitektur

3.1 Entwicklungsumgebung

Linux, CMake, git, Eclipse ; gründe für auswahl: linux: paketmanager für dependencies, unzufriedenheit mit microsofts stiefmütterlicher Behandlung von 64bit- programmen, nicht zuletzt wunsch nach Vertiefung der kenntnisse der linux-welt

cross platform build system, moderne versionsverwaltung,

3.2 Klassendiagramm

3.3 Dependencies

3.3.1 OpenGL3/4

- verwendung von mindestens OpenGL 3/3 core context, um legacy code schon zur compilezeit auszuschließen

3.3.2 OpenCL 1.0

noch keine offenen openCL 1.1 treiber, außerdem keine features davon benötigt; c++-wrapper genutzt, auch von khronos-seiten beziehbar; Gegenüberstellung zu CUDA, vor- und nachteile auflisten, insbesondere das problem ‚dass esk ein 1D-texturen in OpenCL gibt, und man sich entscheiden muss zwischen generischem buffer und Textur, man also nicht hin und her-interpretieren kann wie in CUDA;

3.3.3 GLFW

Fenstermanager + input explizite GL3 core profile creation, einfaches fullscreen, multisampling, keine mainloop, explizites input pulling, mouse grab, alles viel besser als GLUT :)

3.3.4 Grantlee

die string template engine die CL und GL code erzeugt

3.3.5 assimp

3.3.6 ogl math

leichte, aber doch recht maechtige mathe-bibliothek

3.3.7 TinyXML

3.4 Die *Unified Rendering Engine*

URE blubb

3.5 Die Manager-Klassen

blubb2

3.6 Die Buffer-Abstraktion

die bombe, die cpu, ogl und ocl vereint, inclusive ping ponging etc.. fundamentale Klassensammlung fuer den Unified-Aspekt

3.7 Das WorldObject

Basis-Klasse fuer alles was unified simuliert wird: pure viuelle objekt, uniform grid, fluid, rigid body etc..

3.7.1 Das SubObject

3.8 Material

was stellt welches material in welcher Domain dar?

3.9 Geometry

Abtract, Buffer based, Vertex based etc.. ein paar konzepte (implementiert/genutzt nur VertexBased)

3.10 Massively Parallel Program

Basisklasse von Shader und OpenCL Program

3.10.1 Shader

3.10.2 OpenCLProgram

weitere klassen/konzepte to go...

3.11 Status der Implementierung am Ende der BA

Features auflisten;

großteils programmierte, aber ungenutzte/ungetestete features erwähnen (Deferred Rendering, Layered Rendering, RenderTarget-Klasse, Partikel-Rigid bodies, verschiedene Fluid-Typen);

überlegte aber nciht programmierte Konzepte/Algorithmen erwähnen (Triangle-Index-Voxelisierung)

schlimmste schnitzer nennen, wie - miese fluid-visualisierung, - unübersichtliche shadertemplates, besser gemacht bei CL- Kernel-Templates, 1. weil struktur hier besser "vererbbar", 2. weil mehr erfahrung mit Template-Engine

4 Simulation

4.1 Die visuelle Simulationsdomäne

Ein paar worte ueber die shading features, wie sie maskiert werden etc..

4.1.1 Der LightingSimulator

Nochmal drauf hinweisen, dass Rendering etwas generisches in diesem Framework ist, und wir leiber von Lichtsimulation sprechen sollten, auch wenn es monetan nicht photrealistisch ist ;)

4.1.2 Die Lighting Simulation Pipeline Stages

shadowmap gen stage, direct lighting stage, was noch in planung is etc..

4.1.3 ShaderManager

generiert mit grantlee, assigned an materials und verwaltet Shader , abhaenging von der aktuellen lighting stage, den registierten Materials, der Erzeugten kontext, den vom user aktivierten rendering features etc pp

4.1.4 genutzte moderne OpenGL- Features

4.1.4.1 Uniform Buffers

auch von BufferInterface abstrahiert, vorteile auflisten, aber auch stolperfallen: alignment etc)

4.1.4.2 Hardware Tesselation

basics des hardware features erwaehnen fuer den geneigten leser, raptormodell erwaehnen und seinen Aufbereitungsprozess, LOD, displacement mapping erlaeutern

4.1.4.3 Instancing

InstanManager, InstangedGeometry vorstellen, konzept, wie es verwaltet wird, erklaeren

4.2 Die mechanische Simulationsdomäne

blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext

4.2.1 Fluidsimulation

blindtext blindtext

4.2.1.1 Grundlagen

blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext

4.2.1.1 .1 Die Navier-Stokes-Gleichungen

Herleitung, Erläuterung blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext

4.2.1.1 .2 Grid-basierte vs. Partikelbasierte Simulation

blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext blindtext

4.2.1.1 .2.1 Die zwei Sichtweisen: Lagrange vs. Euler

blindtext blindtext blindtext blindtext blindtext blindtext

4.2.1.1 .3 Smoothed Particle Hydrodynamics

ursprünglich aus astronomie blubb blubb

4.2.1.2 Verwandte Arbeiten

Referenzen auf Müller03, Thomas Steil, Goswami, GPU gems, Aufzeigen, was ich von wem übernommen habe, was ich selbst modifiziert habe aufgrund von etwaigen Fehlern in den Papers odel weil OpenCL es schlciht nicht zulässt;

4.2.1.3 Umfang

Abgrenzungf zwischen bisher funktionierenden Features, bisher programmierten, aber nicht integrierten und ungetesten Features und TODOs für die zukunft

4.2.1.4 Algorithmen

Verwaltung der Beschleunigungsstruktur ist der Löwenanteil, nicht die physiksimulation, die eher ein Dreizeiler ist;

4.2.1.4 .1 Work Efficient Parallel Prefix Sum

4.2.1.4 .2 Parallel Radix Sort und Stream Compaction

4.2.1.4 .3 Ablauf

initialisierung, und beschreibung der einzelnen phasen...

5 Ergebnisse

6 Ausblick

7 Fazit