

# **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.	<input type="checkbox"/>	<input type="checkbox"/>
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	<input type="checkbox"/>	<input type="checkbox"/>

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

## Todo list

evtl. beispelschema erstellen für zwei klassische transformationsklassen und adapter vs. unified transformation . . . . .	2
Zitat einfügen? Stefan Müller, PCG? ;) . . . . .	2
Muss ich das irgendwie belegen? Ich wüsste nicht, wie/wo ;( . . . . .	4
referenz auf Peschel, evtl weitere eigenschaften wie curl und sicher- stellung der inkompressibilität . . . . .	4
referenz auf gpu-gems-gridbased-zeug, evtl jos stam . . . . .	4
referenz auf thomas steil und GPU gems . . . . .	5
ref auf peschel . . . . .	5
'indirekt' erläutern? transform feedback buffer, scattering über gl_Position und ein-pixel point rendering, siehe DA von Sinje Thiedemann? . . . . .	5
Wo zum Henker soll ich dazu nun wieder ne Referenz herauskra- men? Ich habe leider keine idee, außer zig allgemeine Papers und Übcher zu überfliegen Zeitproblem!... das ist der nachteil, wenn man so viel recherchiert hat, zunächst mal fürs Verständ- nis und oder oder den ganz groben überblick, und nicht für eine umfassende wissenschaftliche Dokumentation: All den Input, der nur als Hintergrundwissen dienen sollte, jetzt noch wieder zu finden... :( Wie kann ich mit diesem allgemeinen Problem umgehen? . . . . .	5
Referenz eines guten Artikels zu GPGPU finden; . . . . .	5
ref auf späteren abschnitt oder OpenCL spec . . . . .	6
hier bin ich mir nicht mal sicher, ob das stimmt; meine Erinnerung ist nur sehr vage, diese Fragestellung wurde in meiner Liter- atur höchstens ganz marginal behandelt; Was tun? Aussage weg lassen, mich tod- suchen, oder diese gewagte These nach bestem Wissen und Gewissen stehen lassen? . . . . .	6
hier specs und refs zu PCIe 2.0 anbringen? eher nicht, oder? . . . . .	8
screenshots? oder lieber erst später,zusammen mit detaillierter er- läuterung? . . . . .	12
continue brainstorming . . . . .	14
überlegen, ob ich aus Interesse nicht noch weiter in die Richtung recherchieren sollte, da ich nach meiner Implementierung erst so richtig beeindruckt von dem Verfahren war (ich habe im In- ternet noch keine Fluid-Demo gefunden, die ebenfalls SPH im- plementiert; ok., ich hab auch nicht gesucht ;) ), und gerne mehr über die Hintergründe verstehen würde... problem, wie immer: Zeitdruck ;( . . . . .	15

## Inhaltsverzeichnis

1 Einleitung . . . . .	1
------------------------	---

1.1	Motivation . . . . .	2
1.1.1	„Unified Rendering Engine“ . . . . .	2
1.1.2	Fluidsimulation . . . . .	4
<b>2</b>	<b>Überblick</b>	<b>7</b>
2.1	Vision . . . . .	7
2.2	Paradigmen . . . . .	7
2.3	Begriffe . . . . .	8
2.4	Schwerpunkte . . . . .	10
2.4.1	Template-Engine . . . . .	10
2.4.2	Performance durch Implementierung auf der GPU mit modernen GPU-Computing-APIs . . . . .	10
2.4.3	Potential der Entwicklung hin zu einer Unified Engine nicht verschenken . . . . .	11
<b>3</b>	<b>Systemarchitektur</b>	<b>11</b>
3.1	Entwicklungsumgebung . . . . .	11
3.2	Klassendiagramm . . . . .	11
3.3	Dependencies . . . . .	11
3.3.1	OpenGL3/4 . . . . .	11
3.3.2	OpenCL 1.0 . . . . .	11
3.3.3	GLFW . . . . .	11
3.3.4	Grandlee . . . . .	11
3.3.5	assimp . . . . .	12
3.3.6	ogl math . . . . .	12
3.3.7	TinyXML . . . . .	12
3.4	Die Buffer-Abstraktion . . . . .	12
3.5	Das WorldObject . . . . .	12
3.5.1	Das SubObject . . . . .	12
3.6	Material . . . . .	12
3.7	Geometry . . . . .	12
3.8	Massively Parallel Program . . . . .	12
3.8.1	Shader . . . . .	12
3.8.2	OpenCLProgram . . . . .	12
3.9	Status der Implementierung am Ende der BA . . . . .	12
<b>4</b>	<b>Simulation</b>	<b>14</b>
4.1	Die visuelle Simulationsdomäne . . . . .	14
4.1.1	Der LightingSimulator . . . . .	14
4.1.2	Die Lighting Simulation Pipeline Stages . . . . .	14
4.1.3	ShaderManager . . . . .	14
4.1.4	genutzte moderne OpenGL- Features . . . . .	14
4.1.4.1	Uniform Buffers . . . . .	14
4.1.4.2	Hardware Tesselation . . . . .	14

4.1.4.3	Instancing . . . . .	14
4.2	Die mechanische Simulationsdomäne . . . . .	15
4.2.1	Fluidsimulation . . . . .	15
4.2.1.1	Grundlagen . . . . .	15
4.2.1.1 .1	Die Navier-Stokes-Gleichungen . . . . .	15
4.2.1.1 .2	Grid-basierte vs. Partikelbasierte Simulation . . . . .	15
4.2.1.1 .2.1	Die zwei Sichtweisen: Lagrange vs. Euler . . . . .	15
4.2.1.1 .3	Smoothed Particle Hydrodynamics . . . . .	15
4.2.1.2	Verwandte Arbeiten . . . . .	15
4.2.1.3	Umfang . . . . .	16
4.2.1.4	Algorithmen . . . . .	16
4.2.1.4 .1	Work Efficient Parallel Prefix Sum . . . . .	16
4.2.1.4 .2	Parallel Radix Sort und Stream Compaction . . . . .	16
4.2.1.4 .3	Ablauf . . . . .	16
<b>5</b>	<b>Ergebnisse</b>	<b>17</b>
<b>6</b>	<b>Ausblick</b>	<b>18</b>
<b>7</b>	<b>Fazit</b>	<b>19</b>

# 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-Tessellation 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<sup>1</sup> Kombination der Worte „Fluid“, in

---

<sup>1</sup>Es soll der generalistische Ansatz des Frameworks nicht in den Hintergrund gedrängt werden.

Anspielung auf den ursprünglichen 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 Engine“ erwächst aus eigener Erfahrung der Kopplung von Physik- und Graphik-Engines, namentlich der Bullet Physics Library<sup>2</sup> und der OGRE Graphik-Engine<sup>3</sup>. 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 Programmierer der eigentlichen Anwendungslogik den ständigen Umgang mit verschiedenen Repräsentationen und deren Synchronisation zu ersparen.

evtl.  
beispielschema  
erstellen für zwei  
klassische trans-  
formationsklassen  
und adapter vs.  
unified transfor-  
mation

Zitat einfügen?  
Stefan Müller,  
PCG? ;)

Die Aussage „Photorealistische Computergraphik ist die Simulation von Licht“ hat mich wohl auch inspiriert, den Simulationsbegriff allgemeiner aufzufassen und das Begriffspaar "Rendering und Physiksimation" zu hinterfragen<sup>4</sup>.

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

<sup>2</sup><http://bulletphysics.org>

<sup>3</sup><http://www.ogre3d.org>

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



Kombinationsmöglichkeiten steigt, viele sind unsinnig und müssen implizit oder explizit ausgeschlossen werden).

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<sup>5</sup>;

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 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

---

<sup>5</sup> ob der Begriff „Verständnis“ im letzten Falle ganz treffend ist, bleibt Ermessens-Sache

- meine Erfahrung im Engine-Design zu vertiefen
- meine Erfahrungen im (graphischen) Echtzeit-Rendering zu vertiefen
- 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<sup>6</sup>
- 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:

Muss ich das irgendwie belegen? Ich wüsste nicht, wie/wo ;(

referenz auf Peschel, evtl weitere eigenschaften wie curl und sicherstellung der inkompressibilität

referenz auf gpu-gems-gridbased-zeug, evtl jos stam

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 derartige 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 einer 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 beein-

<sup>6</sup>General Purpose Graphics Processing Unit- Computing, die Nutzung der auf massiver Parallelität beruhenden Rechenleistung von Graphikkarten in nicht explizit Graphik-relevanten Kontexten

flüssen 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.

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 Simulator simulieren, indem eine Repräsentation der Geometrie des Rigid Bodies als Partikel gewählt wird;

referenz auf  
thomas steil und  
GPU gems

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.

ref auf peschel

'indirekt' erläutern? transform feedback buffer, scattering über gl\_Position und ein-pixel point rendering, siehe DA von Sinje Thiedemann?

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 Visualisierungsmöglichkeit bietet;
- Die gemeinsame Nutzung von Geometrie sowohl zur mechanischen

Wo zum Henker soll ich dazu nun wieder ne Referenz herauskramen? Ich habe leider keine idee, außer zig allgemeine Papers und Bücher zu überfliegen Zeitproblem!... das ist der nachteil, wenn man so viel recherchiert hat, zunächst mal fürs Verständnis und oder oder den ganz groben überblick, und nicht für eine umfassende wissenschaftliche Dokumentation: All den Input, der nur als Hintergrundwissen dienen sollte, jetzt noch wieder

ref auf späteren  
abschnitt oder  
OpenCL spec

Simulation als auch zur Visualisierung mit OpenGL ist hiermit ermöglicht. OpenCL ermöglicht diese gemeinsame Nutzung explizit über gemeinsame Buffer-Nutzung.

hier bin ich mir  
nicht mal sicher,  
ob das stimmt;  
meine Erinnerung  
ist nur sehr vage,  
diese Fragestel-  
lung wurde in  
meiner Literatur  
höchstens ganz  
marginal behan-  
delt; Was tun?  
Aussage weg-  
lassen, mich tod-  
suchen, oder diese  
gewagte These  
nach bestem Wis-  
sen und Gewissen  
stehen lassen?

- 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 <sup>7</sup> oder Bewahrung des Volumens.

---

<sup>7</sup>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<sup>8</sup> einen integrativen Bestandteil der Spielmechanik ausmachen, sind mir nicht bekannt;

Von Dreiecks-Geometrie erhoffe ich mir eine genauere Repräsentation zur Kollisionsbehandlung, bei gleichzeitiger Ersparnis vieler Partikel, die sonst z.T große Oberflächen repräsentieren 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 Konsistenz 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
- Es sollte eine Art Pipeline-Architektur entstehen, wo bestimmte Pipeline-Stages bestimmte Simulations-(Zwischen)-Ergebnisse implementieren, und ggfs. anderen Stages diese zur Verfügung stellen. Jede Simulationsdomäne hat seine eigene Pipeline; Dennoch können Interdependenzen bestehen;  
Diesen Interdependenzen wird durch eine Konzept-spezifische Verwaltung durch verschiedene Singleton-Manager-Klassen genüge getan;

---

<sup>8</sup>s. Kapitel 4.2.1.2 für mehr Informationen zu Height-Field-basierter Fluidsimulation

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 Parallelität 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 Programmieraufwand ein bestimmtes Feature implementieren, überall dort unbedingt zu vermeiden, wo sie die konsistente Gesamtstruktur des Systems zu bedrohen scheinen.

hier specs und refs zu PCIe 2.0 anbringen? eher nicht, oder?

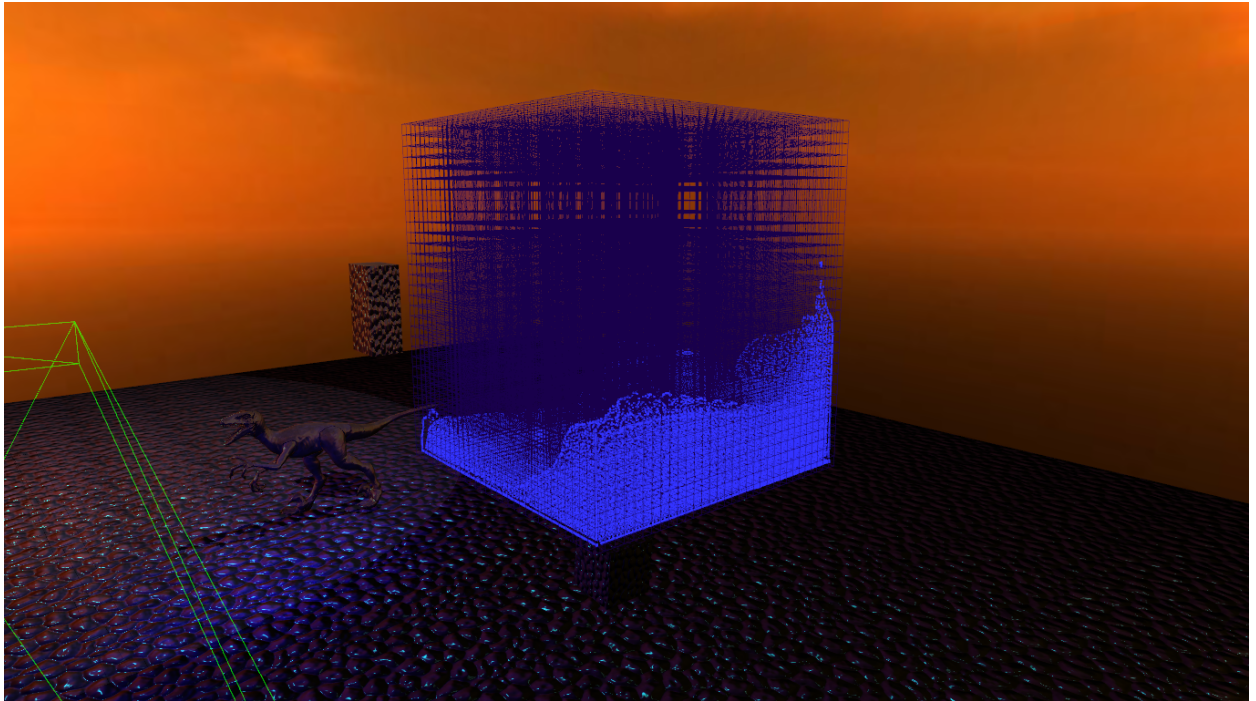
## 2.3 Begriffe

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

**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.



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

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*<sup>9</sup> 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

<sup>9</sup>Diese Welt muss dabei nicht zwingend unserer Realität ähneln oder entsprechen.

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;

erklären, was ich unter Unified Rendering verstehe, was Rendering dadurch fuer eine generalistische Bedeutung bekommt;  
tabelle mit Gegenüberstellung

## 2.4 Schwerpunkte

- entwicklungsumgebungm die die entwicklönung eines konsistenten frameworks erleichtert, moderne tools, nach möglichkeit cross platform (windows vs. linux) - verwendung von bibliotheken, die die entwicklung einer seriösen engine ermöglichen - verwendung von mindestens OpenGL 3/3 core context, um legacy code schon zur compilezeit auszuschließen - realisierung gängiger graphischer effekte, vor allem tessellation, verweise auf entsprechende section - tessellation - nutzung moderner openGL features, Uniform Buffers, Tessellation, hardware instancing, verweis auf entsprechende section; - möglichst hohe konfigurierbarkeit ohne ständigen recompile: config file - buffer abstraktion - memory tracking, (erklären, warum nicht tracking mit Valgrind)

- effiziente Verwendung von OpenCL, hardware- spezifische bedingte compilings dank grantlee

### 2.4.1 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, → einfachere Code- Inspection, verbesserung der lesbarkeit durch generierte, feature-spezifische programme, bessere struktur verwaltung des source codees

### 2.4.2 Performance durch Implementierung auf der GPU mit modernen GPU-Computing-APIs

auf die massive parallelität eingehen, die sowohl von visueller 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



### **2.4.3 Potential der Entwicklung hin zu einer Unified Engine nicht verschenken**

i.e. fenstermanager, input etc sollten wohl überlegt sein, für vielseitig, flexible anwendung zur Laufzeit sollten keine Speicher-Lecks auftreten, damit Funktionalität kontrolliert heruntergefahren und neu initialisiert werden kann; Für gemeinsamen zugriff sollten viele Daten für andere Klassen verfügbar sein (Buffer, Rendering Results...); Realisierung über Manager-Singleton-Klassen und Zugriff über Map-Container;

## **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**

#### **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**

explizite GL3 core profile creation, einfaches fullscreen, multisampling, 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 mächtige mathe-bibliothek

### 3.3.7 TinyXML

## 3.4 Die Buffer-Abstraktion

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

## 3.5 Das WorldObject

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

### 3.5.1 Das SubObject

## 3.6 Material

was stellt welches material in welcher Domain dar?

## 3.7 Geometry

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

## 3.8 Massively Parallel Program

Basisklasse von Shader und OpenCL Program

### 3.8.1 Shader

### 3.8.2 OpenCLProgram

weitere klassen/konzepte to go...

## 3.9 Status der Implementierung am Ende der BA

Features auflisten:

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

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

screenshots?  
oder lieber erst  
später, zusammen  
mit detaillierter  
erläuterung?

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 , abhaengig 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 Tessellation

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

##### 4.1.4.3 Instancing

continue brain-  
storming

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

## 4.2 Die mechanische Simulationsdomäne

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

### 4.2.1 Fluidsimulation

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

##### 4.2.1.1 .2 Grid-basierte vs. Partikelbasierte Simulation

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 blindtext

##### 4.2.1.1 .3 Smoothed Particle Hydrodynamics

ursprünglich aus astronomie blubb blubb

überlegen, ob ich aus Interesse nicht noch weiter in die Richtung recherchieren sollte, da ich nach meiner Implementierung erst so richtig beeindruckt von dem Verfahren war (ich habe im Internet noch keine Fluid-Demo gefunden, die ebenfalls SPH implementiert; ok., ich hab auch nicht gesucht ;) ), und gerne mehr über die Hintergründe verstehen würde... problem, wie immer: Zeitdruck ;(

#### 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 schlicht 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**