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)

Todo list

Inhaltsverzeichnis

Abbildungsverzeichnis

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

- Boilerplate-Code in den Shadern vermieden wird und
- 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 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.

evtl. beispielschema erstellen für zwei klassischee transformationsklassen und adapter vs. unified transformation

Zitat einfügen? Stefan Müller, PCG? ;)

Die Aussage „Photorealistische Computergraphik ist die Simulation von Licht“ hat mich wohl auch inspiriert, den Simulationsbegriff allgemeiner

[1] Es soll der generische Ansatz des Frameworks nicht in den Hintergrund gedrängt werden.
[2] http://bulletphysics.org
[3] http://www.ogre3d.org

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).

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

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

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

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
- 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:

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

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,

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

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.

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

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

referenz auf gpugems-gridbasedzeug, evtl jos stam

referenz auf thomas steil und GPU gems

ref auf peschel

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

Wo zum Henker soll ich dazu nun wieder ne Referenz herauskramen? 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 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 zu finden... :(Wie kann ich mit diesem allgemeinen Problem umgehen?

Referenz eines guten Artikels zu GPGPU finden;

ref auf späteren abschnitt oder OpenCL spec

hier bin ich mir nicht mal sicher, ob das stimmt; meine Erinnerung ist nur sehr vage, diese Fragestellung wurde in meiner Literatur höchstens ganz marginal behandelt; Was tun?

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;

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

- 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 Simulationsdomänen 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:

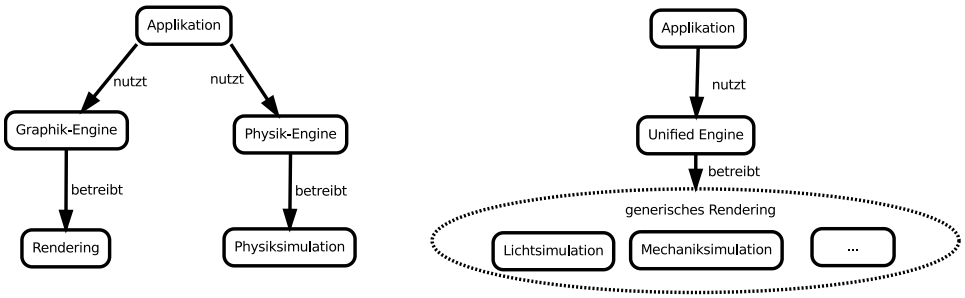

Abbildung 1: Gegenüberstellung von Verwendung und Begrifflichkeiten von klassischen Engines und einer Unified Engine

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 genommen Licht auch ein physikalisches Phänomen ist, und somit

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

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.

GPU Computing Für gewöhnlich bekannt als Begriff, der GPGPU-Computing beschreibt in Abgrenzung zur Verwendung der GPU zur Berechnung von Bildern mittels Graphik-APIs, soll auch dieser Begriff im Folgenden einen Oberbegriff darstellen für beliebige Berechnungen, die auf der GPU ausgeführt werden, gleichgültig ob in einem GPGPU- oder einem Graphik-bezogenen Kontext; Die Abgrenzung der Domänen „Graphik“ und „General Purpose“ ist nicht zwingend anhand der benutzten APIs festzustellen; So kann man z.B. mit einer GPGPU-API Ray Tracing für visuelles Rendering betreiben, oder aber General Purpose-Berechnungen mit Graphik-APIs anstellen. [10]

Beleg? Wie war das mit Folding@Home?

Da im Computer jede Operation eine Form von Berechnung darstellt, erscheint die Verwendung von *GPU Computing* als Oberbegriff für legitim; GPU Computing unterteilt sich dann in *Graphik-Programmierung* (für gewöhnlich, aber nicht zwingend durch Verwendung von Graphik-APIs wie OpenGL oder Direct3D) und *GPGPU* (für gewöhnlich, aber nicht zwingend durch Verwendung von GPGPU-APIs wie CUDA, DirectCompute oder OpenCL).

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

2.4 Schwerpunkte

Die Entwicklung eines solchen *Unified Frameworks*[11] umfasst sehr viele Aspekte, und einige werden wohl in dieser Ausarbeitung keine Erwähnung finden. Um die didaktischen Ziele von Seite 4 nicht aus den Augen zu verlieren, wurden folgende Schwerpunkte gesetzt:

[10] Vor der Zeit der *Compute Unified Device Architecture (CUDA)*-Devices und der gleichnamigen GPGPU- Programmier-Umgebung von Nvidia, welche mit dem G80-Chipsatz bzw. der GeForce 8800 GTX 2006 ihren Anfang nahm, war der „Missbrauch“ von Graphik-APIs für den Endbenutzer die einzige Möglichkeit, die massive Rechenleistung der Graphikkarte für generische Zwecke zu nutzen.

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

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[12] und Eclipse[13] in Kombination mit dem Cross-Platform Build-System CMake[14] gewechselt. Mir war es sehr wichtig, ein System zu entwickeln, welches vollständig für die 64 Bit-Prozessor-Architektur compilet ist, 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. [15]

refernenz finden, bin mir da eher unsicher

explizite Danksagung?

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; Vor allem die Wahl des Fenster-Managers, der Input- Bibliothek und der Mathematik-Bibliothek musste deshalb mit Bedacht getroffen werden; Mehr dazu in Kapitel 3.1.

Nutzung moderner OpenGL-Features Mit OpenGL Version 3 erfuhr die offene Graphik-API eine gründliche Reinigung: Viele nicht mehr zeitgemäße Features wurden für deprecated erklärt und durch einige neue, meist generischere Features ersetzt; Das Resultat ist eine schlankere API mit mehr Verantwortung für den Programmierer über die Rendering-Pipeline; Man muss mehr „selbst machen", hat aber auch mehr Macht. Diese Neuerung spielt dem Konzept einer Unified Engine geradezu in die Hände, da nun fast die gesamte vordefinierte Semantik wie `gl_FrontMaterial.shininess`, `gl_LightSource[i].diffuse`, `gl_NormalMatrix` oder `gl_MultiTexCoord2` entfernt wurde, und man jetzt fast alle benötigten Werte entweder als generische Vertex-Attributes übergibt oder selber explizit als Uniform - Variablen definiert und setzt; Diese neue „Freiheit der Semantik“ war eine große Inspiration, den Gedanken einer Unified Engine zu fassen, wo es nun keinen Anlass mehr gab, OpenGL-State-Variablen implizit um zu interpretieren um bestimmte Effekte zu realisieren (Beispiel: Shadow-Map-Lookup-Matrix in eine der Textur-Matrizen laden).

[12] http://www.ubuntu.com/
[13] http://www.eclipse.org/
[14] http://www.cmake.org/
[15] 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.

Es sollte mindestens ein OpenGL Kontext im Core Profile der Version 3.3 benutzt werden; Das Core Profile stellt sicher, dass Routinen und Flags, die in der Spezifikation als deprecated gekennzeichnet sind, einen Fehler produzieren; Auf diese Weise wird die Programmierung mit nur der „modernen“ Untermenge der OpenGL-API forciert; Optional sollte ein OpenGL 4 Kontext erstellt werden können, sofern unterstützende Hardware existiert und dies vom Benutzer erwünscht ist; Wo es sinnvoll und angebracht war, sollten moderne Features von OpenGL verwendet werden; Mehr dazu in Kapitel 4.1.4.

Template-Engine Da GLSL[16] und OpenCL C, die verwendeten Sprachen, mit denen die GPU programmiert werden kann, nicht objekt-orientiert sind, (zumindest OpenGL 3 und OpenCL 1.0) keine Mechanismen zum Überladen von Funktionen haben, und noch nicht einmal mal eine
#include-Direktive existiert, tendieren diese GPU-Programme, die sich einen erheblichen Anteil an ihrem Code untereinander teilen, zur Code-Vervielfachung in den einzelnen Quelldateien; Dies schränkt die Wartbarkeit und bequeme Änderbarkeit und zuweilen auch die Lesbarkeit enorm ein;
Abhilfe schafft hierbei die Nutzung einer String-Template-Engine namens *Grantlee*[17]; Mit ihrer Hilfe lassen sich zur Laufzeit in Abhängigkeit von aktuellen Parametern angepasste GPU-Programme generieren, die nur den aktuell nötigen Code enthalten; Somit sind die generierten Programme lesbarer als wenn man sie über klassische bedingte Komplilierung (mit #ifdef FEATURE_XY ... #endif-Direktiven) geschrieben hätte; Weitere Einzeiheiten sind in 3.8 zu finden;

Implemetation und Kombination gängiger visueller Effekte Um dem softwaretechnischen Unterbau schließlich etwas Leben einzuhauchen, wurden einige visuelle Effekte unter Nutzung von OpenGL3/4 implementiert; Dank der Template-Engine lassen sich alle Effekte – sofern sinnvoll – Zur Laufzeit in beliebiger Kombination hinzu- oder abschalten. Die Effekte werden in Kapitel 4.1.5 detaillierter vorgestellt.

Buffer-Abstraktion Der zentrale Dreh- und Angelpunkt der Unified Engine ist der *Buffer*;
Ist „Buffer“ für gewöhnlich die Bezeichnung eines allokierten Speicherbereiches zur Nutzung durch ein Programm, verkompliziert sich diese simple Sicht auf einen Buffer durch die GPU Computing- APIs, erstens, weil man zwischen Host- und Device- Memory unterscheiden muss, zweitens, weil die GPU Computing- APIs den Buffern verschiedene Semantiken zuschreiben (generischer Buffer, Vertex Attribute

[16] OpenGL Shading Language
[17] www.grantlee.org/

Buffer, Vertex Index Buffer, Uniform Buffer, Transform Feedback Buffer, Texture Buffer, Render Buffer, verschiedene Texturtypen etc.), die je nach API zwischen den einzelnen Verwendungs-Kontexten (Host, OpenGL, OpenCL) kompatibel zueinander sind oder eben auch nicht; [18] Die einzelnen Buffertypen haben trotz ihrer semantischen Unterschiede und verschiedensten zugehörigen API-Routinen einige konzeptionelle Gemeinsamkeiten:
Die meisten Buffertypen haben irgendeine Form von folgenden assoziierten Operationen:

- Allokation von Speicher
- Freigabe von Speicher
- Schreiben
- Lesen
- Kopieren
- Mappen von device-Memory zu Host-Memory
- Spezifikation von Semantik (insb. bei OpenGL Buffers)
- Spezifikation von internen Datentypen, ggfs. Channel-Layout etc.
- Synchronisieren vor dem nächsten Zugriff
- Acquirering für einen Kontext zur Nutzung von API- Interoperabilität

Ebenfalls ist eine Menge Meta-Information vielen Buffertypen gemeinsam:

- Name
- Buffergröße
- Buffertyp
- Buffer-Semantik
- interne Datentypen
- weitere Buffertyp-spezifische Meta-Informationen
- Information der beteiligten Kontexte (z.B. nur Host-Memory, reiner OpenGL-Buffer, CL/GL-interop-Buffer mit oder ohne assoziierten Host memory etc.)

[18] Mit CUDA ist es z.B ohne weiteres möglich, einen beliebigen GPU-Buffer an eine Textur-Einheit zu binden und entsprechend wie eine Textur zu samplen, und umgekehrt; OpenCL erlaubt dies nicht; Hier muss man sich im Vorfeld entscheiden, ob man einen „normalen“ Buffer oder eine Textur haben will.

Je nach Buffertyp und assoziierter API unterscheiden sich die Routinen, um diese Operationen auszuführen bzw. diese Meta-Informationen auszulesen, teilweise erheblich; Um dem Benutzer der Unified Engine die Bürde der API-spezifischen Operationen abzunehmen, und möglichst viel Meta-Information geschlossen zur Verfügung zu stellen, bietet sich an, ein vereinheitlichtes Interface bereit zu stellen, von dem konkrete Buffertypen abgeleitet werden können, die die entsprechenden Operationen implementieren; Hieraus entstanden eine Reihe von Buffer-Und Texturklassen, welche zugehörige Meta-Informationen enthalten. Eine detaillierte Beschreibung findet sich in Abschnitt 3.9.

Effiziente Verwendung von OpenCL Bei der OpenCL-Implementierung lag der Schwerpunkt sowohl bei der algorithmischen Effizienz als auch bei hardwarespezifischen Optimierungen; Es werden Parameter wie z.B. die Größe des lokalen Speichers einer OpenCL Compute Unit abgefragt, und in Verbund mit den benutzerdefinierten Simulations-Parametern die Konstanten im OpenCL-Code mit der Template Engine und Workload-Parameter und Buffergrößen auf Seiten der Applikation entsprechend gesetzt; Für Details sei auf 4.2.1.5 verwiesen;

3 Systemarchitektur

Dieses Kapitel soll ein Gefühl für die Komponenten von *Flewnit* und ihre Zusammenhänge vermitteln. Die Komponenten „in Aktion“ werden im Detail im Verlauf des Kapitels 4 beschrieben.

Das System wurde in C++ als (wahlweise statisch oder dynamisch zu linkende) Bibliothek implementiert. Der Code der GPU-Programme ist in GLSL bzw. OpenCL C verfasst.

3.1 Dependencies

Zunächst sollen die verwendeten Third-Party-Bibliotheken kurz vorgestellt werden:

OpenGL3/4 Die schon mehrfach erwähnten modernen Versionen der *Open Graphics Library*, der offenen API der Khronos Group zur hardwarebeschleunigten Graphik-Programmierung auf Basis der Dreie Rasterisierung. Um die Programmierung ohne Legacy-Routinen nicht erst zur Laufzeit über einen OpenCL-Error durch Verwendung eines Core-Profiles zu erzwingen, gibt es einen OpenGL- Header namens „gl3.h“[19], der in Kombination mit der entsprechenden Präprozessor-Definition `#define GL3_PROTOTYPES 1` schon zur Compile-Zeit nur die non-deprecated Routinen zur Verfügung stellt.

evtl treffenderen Ausdruck finden: Scanline-basiert oder was auch immer

OpenCL 1.0 Die *Open Computing Language*, erste Version der noch jungen API für massiv parallele Programmierung [20], wie OpenGL von der Khronos Group verwaltet; sie stellt den ersten offenen Standard für GPGPU dar, d.h., die Verwendung der API ist nicht mehr an eine bestimmte Hardware (wie bei Nvidia CUDA) oder ein bestimmtes Betriebssystem (wie Microsofts DirectCompute) gebunden.

Zur Zeit der Implementierung waren noch keine Non-Developer-Treiber für OpenCL 1.1 verfügbar, außerdem gab es kein Feature dieser Version, welches ich dringend benötigt hätte. Deshalb habe ich die Version 1.0 verwendet.

Es gibt einen C++ -Wrapper der C-API, welcher stark auf C++-Templates basiert und in einer einzigen Headerdatei implementiert ist. Dieser ist direkt von der Khronos-Homepage[21] beziehbar. Diesen Wrapper habe ich verwendet, da er die Nutzung der API wesentlich eleganter macht.

[19] beziehbar unter http://www.opengl.org/registry/
[20] die GPGPU-Computing einschließt
[21] http://www.khronos.org/registry/cl/

GLFW 2.7 Wie auf Seite 11 angedeutet, waren mir folgende Dinge wichtig, damit die Einsetzbarkeit des Frameworks in professionelleren Kontexten nicht schon im Vorfeld verbaut ist:

- Option auf Fullscreen
- Option auf Multisampling
- Die Möglichkeit der Erstellung eines OpenGL-Kontextes einer frei wählbaren Version mit Option zwischen Core- und Compatibility-Profile
- Option auf „*Mouse Grab*“, so dass man wie in einem Computerspiel mit ausgeblendetem Mauszeiger nur durch Bewegung der Maus ohne Bildschirm-/Fenster-Grenzen die virtuelle Kamera rotieren kann;
- „Input events“, d.h. Aktualisierungen von Benutzereingaben sollen häufig und mit minimaler Latenz geschehen, außerdem so unabhängig wie möglich von der Framerate sein; Nach möglichkeit sollten Input-Updates zumindest äktiv abfragbarßein (im Gegensatz zum passiven Warten darauf, dass von der Input-Library eine Callback-Funktion aufgerufen wird)
- Es soll volle Kontrolle über die "Render-Loop"geben, so dass man nicht den Kontrollfluss an eine Funktion übergibt, die womöglich nie zurückkehrt und weiteren Kontrollfluss durch das Benutzerprogramm nur über Callback-Funktionen ermöglicht (wie `glutEnterMainLoop()` beim in die Jahre gekommenen *GLUT*). Ein derartiges Konstrukt ist einer Engine nicht würdig und verhindert womöglich sauberes Herunterfahren und Neu-Initialisierung, wie es z.B. beim Wechseln einer Szene oder eines fundamentalen globalen Settings nötig sein könnte.

GLFW[22] in der Version 2.7, die zum Zeitpunkt der Implementation aktuellste stabile Version, erfüllt diese Forderungen, und findet damit in *Flewnit* Einsatz sowohl im Fenster- als auch im Input-Manager. Die Timing-Funktionalität wird ebenfalls von GLFW übernommen.

GLM leichte, aber doch recht maechtige mathe-bibliothek

Grantlee die string template engine die CL und GL code erzeugt

assimp

boost filesystem

TinyXML - möglichst hohe Konfiguriertbarkeit ohne ständigen recompile: parsing von XML config file

[22] http://www.glfw.org/

3.2 Klassendiagramm

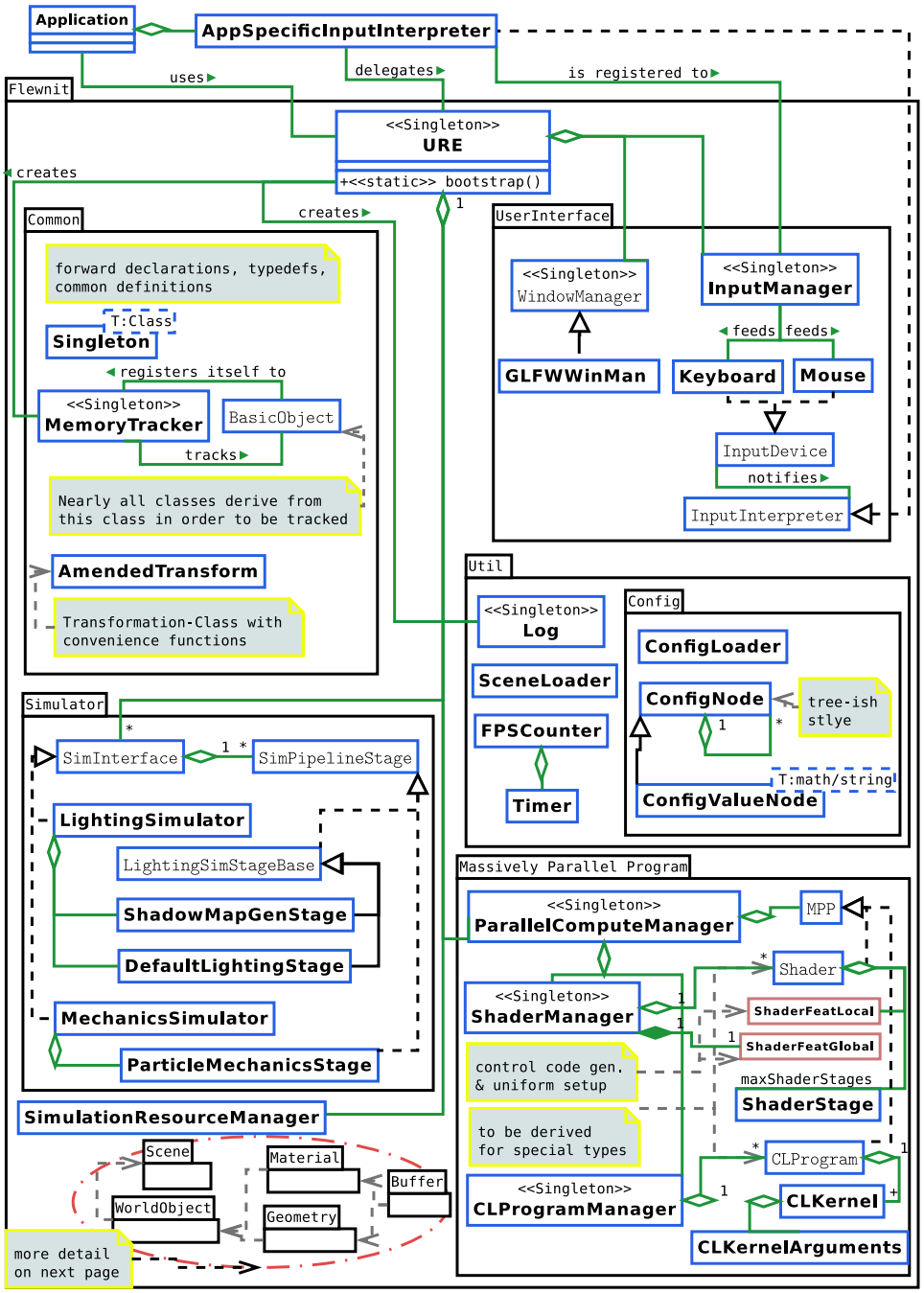

Abbildung 2: Klassendiagramm des Gesamtsystems, Teil 1

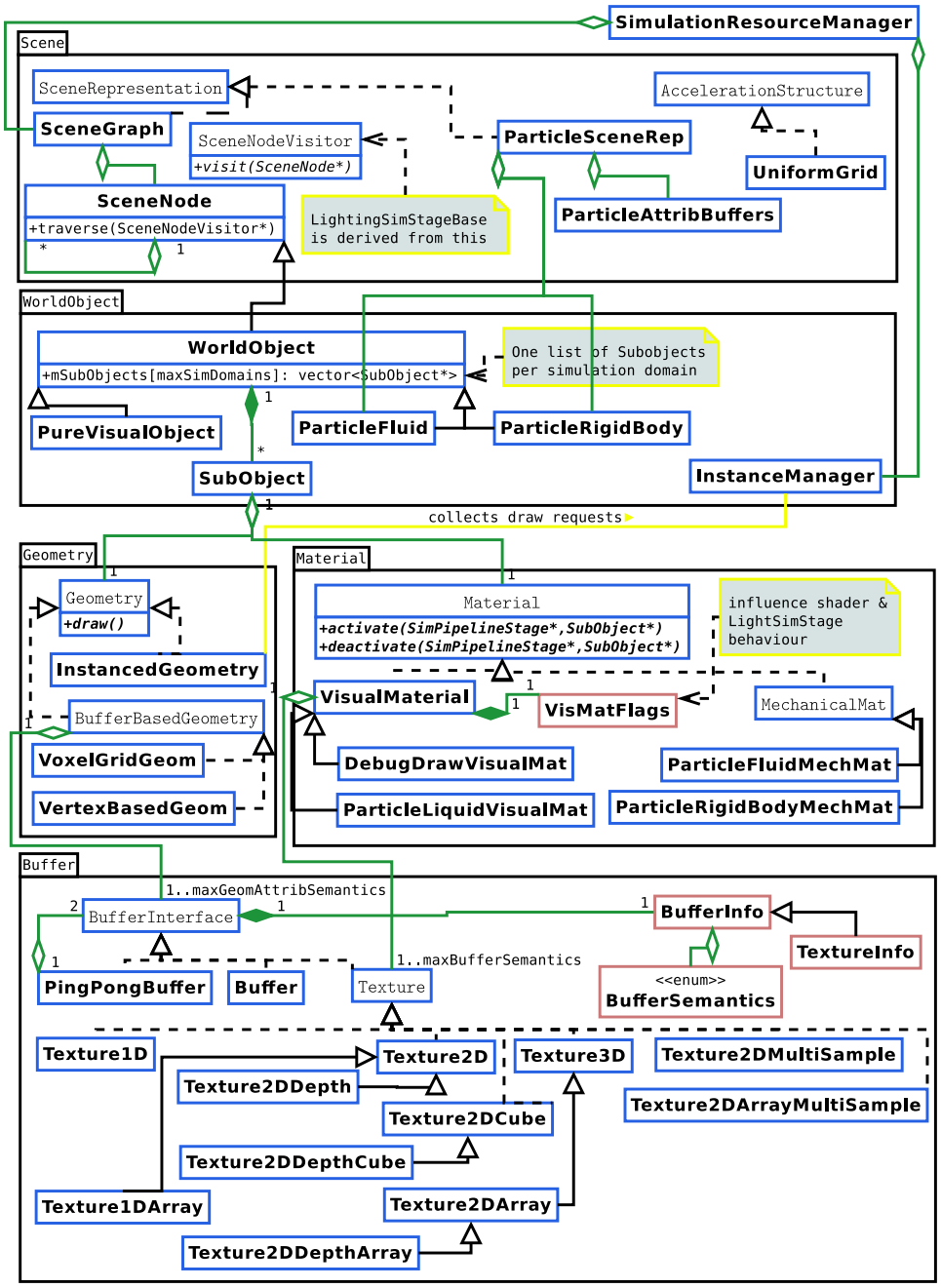

Abbildung 3: Klassendiagramm des Gesamtsystems, Teil 2

3.3 BasicObject und Memory Tracking

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)

3.4 Die *Unified Rendering Engine*

URE blubb

3.5 Die Simulator-Klassen

3.6 Die SimulationPipelineStages

shdow map gen, direct lighting, rendering features particlemechanics stage; in planung: deferred rendering G-Bufferfill, deferred rendering shade, div. post processing stages

3.7 Die Manager-Klassen

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

evtl. andere reihenfolge

3.8 Die Template-Engine

exemplarischer code schnipsel, refernz su shadermanager und CLProgram-Manager, erklärung wie man templat contex setzt, vererbung etc;

3.9 Die Buffer-Abstraktion

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

3.10 Das WorldObject

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

3.10.1 Das SubObject

3.11 Material

was stellt welches material in welcher Domain dar?

3.12 Geometry

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

		Context		
		Host	OpenGL	OpenCL
generic Buffer		✓	x	✓
OpenGL Buffers	Vertex Attribute Buffer	o	✓	o
	Vertex Index Buffer	o	✓	o
	Uniform Buffer	o	✓	o
	Render Buffer	x	✓	✓
Textures	1D Texture	o	✓	x
	2D Texture	o	✓	✓
	3D Texture	o	✓	✓
	Special Texture	?	✓	?

Abbildung 4: Verschiedene Buffertypen und ihre Verfügbarkeit in verschiedenen Kontexten
Legende:
✓ → nativ unterstützt; o → kompatibel; x → nicht unterstützt;
? → Unterstützung abhängig von weiteren Parametern;

3.13 Massively Parallel Program

Basisklasse von Shader und OpenCL Program

3.13.1 Shader

3.13.2 OpenCLProgram

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

weitere klassen/konzepte to go...

3.14 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, SceneNodeVisitor etc..

4.1.1 Der LightingSimulator

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

4.1.2 Die Lighting Simulation Pipeline Stages

baseclass etc... 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 Hintergrund:Batching

PCIe-Bandbreite und -Latenz nicht überlasten durch immediate mode oder andere befehls-serien;

referenz zu PCIe-Flaschenhals-stuff

4.1.4.2 Uniform Buffers

auch von BufferInterface abstrahiert, vorteile auflisten, aber auch stolperfallen: alignment etc) nutzen für transformationsmatrizen beim instancing und für beliebige lichtquekken

4.1.4.3 Instancing

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

4.1.4.4 Hardware Tesselation

basics des GL4- hardware features erwaehnen fuer den geneigten leser, raptor-modell erwaehnen und seinen Aufbereitungsprozess, LOD, displacement mapping erlaeutern

diesen klumbatsch in form bringen, mit bildern anreichern etc pp

4.1.5 Implementierte Effekte

Zunächst zum Begriff "Mapping", der so oft auftaucht: Englisch "map-"Landkarte", "to mapäbbilden"Bedeutet in der Computergrafik meist die Abbildung eines Bildes auf eine Oberfläche nach einem bestimmten Algorithmus;

- Beleuchtung durch beliebig viele Punkt- und Spot-Lichtquellen (also Lichtquellen mit eineme gerichteten Kegel, Scheinwerfer)

- Shadow Mapping: Erzeugen eines bildes aus Tiefenwerten, anschließend vergleich der Tiefenwerte aus Kamerasicht mit denen aus Lichtquellensicht (der ßhadow map", Schattenkarte), pixel im finalen Bild gilt als verdeckt wenn Tiefenwert aus Kamerasicht größer als der entsprechende Pixel in der shadow map, unverdeckt wenn nicht;

- Normal Mapping: Verzerrung der Oberflächen-Normalen (Vektor Sekrecht zur Oberfläche) um relief-artige Geometriedetails zu simulieren, ohne dass tatsächlich diese feine Geometrie in der virtuellen Szene existiert; Dies spart Rechenleistung und Speicher im Vergelcih zu einer Szene, wo all dieses Detail tatächlich in der Geometrie vorhanden wäre; Anschauliches Anwendungsbeispiel: Illusion der feinen Geometrie von Rauhfasertapete auf einem schlichten Quadrat; Nachteil: Die geometrische Illusion brich bei flachen Betrachtungswinkeln ein, die Flachheit der eigentlichen, simplen Geometrie fällt dann auf; Die Information der verzerrten Normalen stammt ebenfall aus einem Bild, der "normal map"; Diesmal werden die Pixelwerte jedoch nicht als Farben oder Tiefenwerte, sondern als Abweichung von der unverzerrten Normalen interpretiert (rot->x-Achse;grün->y-Achse; blau->z-Achse); Da im Computer alles nur Zahlen sind und Semantik erst durch unsere Verwendung und Wahrnehmung erlangen, und da die Graphikkarten so weit flexibel/programmierbar geworden sind, dass man als Programmierer Kontrolle über derartige Üm-Interpretierung"hat, ist dies möglich;

- Environment mapping: Der Trick, perfekt spiegelnde Materialen vorzugaukeln: Es wird in einer "Cube map"nachgeschaut, einer Sammlung von sechs bildern, wo jedes Bild eine Würfelseite repräsentiert; Die Reichtung der Normalen eines Pixels wird umgerechnet in eine Koordinaten, mit der in der Cube Map nachgeschaut wird; Dieser Frabwert fießt dann in die Farbe des Pixels des finalen Bildes ein; Vorteil: Dinge wie lackierte Autokarosserien lassen sich ganz gut vorgaukeln, mit recht geringem Rechenaufwand; Nachteil: Da für gewöhnlich nur in einem statischen Bild nachgeschaut

wird, können dynamische Änderungen der Szene bei der "pseudo-spiegelung"nicht erfasst werden; Ein Objekt, welches sich nache eines autos bewegt, bewegt sich ein seiner SSpiegelung"nicht; Aus solchen Gründen sind in Cube maps oft nur sehr entfernte Dinge dargestellt: Horizong, Himmel, Wolken etc.; Diese Dinge ändern sich in der Realistät ja nicht so schnell, daher fällt der Nachteil beim environment mapping unter dieser Einschränkung nicht mehr so drastisch auf; Der Hintergrund, die orangene Dämmerung, ist gnau diese Cube map, die ich also sowohl für die Pseudo-Spiegelung als auch als "Füllmaterial"dort, wo ich keine Geometrie in der Szene habe, verwende;

- Tesselation: Wie Bei Normal Mapping soll der wahrgenommene Detailgrad der Geometrie erhöht werden; Jedoch erzeugt die Tesselation „echte“ Geometrie, in Abhängigkeit von der Entwerfnung eines Objekte zur Betracher-Kamera; Somit wird dort Geometrie erzeugt, wo sie nötig für den Detailgrad des aktuellen Bildes ist, und dort eingespart, wo sie momentan unnötig ist; Diese Technik hat nicht die Nachteile des Normal mappings; Jedoch Ist durch die Reine Erzeugung von Geometrie noch nicht viel gewonnen; Sinn bekommt diese neue Geometrie wert dann, wenn sich auch wirklich mehr Datail mit ihr darstellen lässt; Erreicht wird dies durch eine sogenannte Displacement Map (frei Übersetzt "Verschiebungs-Karte", ein Bild, in dem Tiefenwerte der Hoch-detaillierten Geometrie gepeichert sind). Die neu erzeugte Geometrie wird also entlang der Normalen um den Betrag verschoben, wie in der Displacement Map eingetragen ist; Somit entsteht ein "tatsächliches Relief", im gegensatz zum Vorgegaukelten Relief beim Normal Mapping; MEhr Details erspare ich dir, z.b. Warum man Normal Mapping trotzdem immer noch für die Beleuchtung braucht, trotz der Tesselation und dem Displacement mapping;

Anmerkung: Weil ich Tesselation so toll finde, habe ich mir das Velociraptor-3D-Modell aus dem Internet besorgt; Dieses hatte 5 Milloonen Dreiecke; Ich habe es mit einem Programm (was ich nicht selbst geschrieben habe, davon versteh ich leider noch viel zu wenig) herunterrechnen lassen, so dass ein vereinfachtes Modell mit etwa 11000 Dreicken entstand, also ein etwa zweitausend mal simpleres Modell. Mit einem anderen Programm habe ich dann die Geometrie des komplexen Modells auf die des simplen Modells projeziert, die detailgrad-bedingte Distanz zwischen den Geometrien in ein Bild geschrieben; Dieses Bild ist die Displacement map für die Tesselation izur Darstellung in meinem eigenen Programm; Somit kann ich nun den Dinosaurier beinahe so detailliert darstellen, wir er im Originalmodell vorliegt, jedoch mit viel höheren Bildwiederholungsraten;

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

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 ;(

4.2.1.1 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

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...

4.2.1.5 Hardwarespezifische Optimierungen

5 Ergebnisse

6 Ausblick

7 Fazit

[Sta99] [Sta03] [BMF07] [MCG03] [GSSP10] [vdLGS07] [Pes09] [Ste07] [Ebe04] [LG08] [HSO07]

Literatur

[BMF07] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: Siggraph 2007 course notes. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 1–81, 2007.

[Ebe04] David H. Eberly. *Game Physics*. Interactive 3D Technology. Morgan Kaufman, 2004.

[GSSP10] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 55–64. Eurographics Association, 2010.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39. Addison-Wesley Professional, 2007.

[LG08] Scott Le Grand. Broad-Phase Collision Detection with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 32. Addison-Wesley Professional, 2008.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-Based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159. Eurographics Association, 2003.

[Pes09] Franz Peschel. Simulation und Visualisierung von Fluiden in einer Echtzeitanwendung mit Hilfe der GPU. Studienarbeit, Universität Koblenz-Landau, apr 2009.

[Sta99] Jos Stam. Stable fluids. pages 121–128, 1999.

[Sta03] Jos Stam. Real-time fluid dynamics for games, 2003.

[Ste07] Thomas Steil. Efficient Methods for Computational Fluid Dynamics and Interactions. Diplomarbeit, Universität Koblenz-Landau, dec 2007.

[vdLGS07] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen Space Fluid Rendering with Curvature Flow. In Eric Haines, Morgan McGuire, Daniel G. Aliaga, Manuel M. Oliveira, and Stephen N. Spencer, editors, *SI3D*, pages 91–98. ACM, 20092007.