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

Tabellenverzeichnis

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

[1] Es soll der generische 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.

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 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

[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.

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 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

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

- 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, eine deratige Fluidsimulation in einen Anwendungskontext zu integrieren, der langfristig über den einer Demo hinausgehen soll, hat also einen leicht pionierhaften Beigeschmack.

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

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 beein-

referenz auf gpugems-gridbased-zeug, evtl jos stam

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

flussen 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;

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 Visualisierungmöglichkeit 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 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

ref auf späteren abschnitt oder OpenCL spec

Simulation als auch zur Visulalisierung 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 Fragestellung wurde in meiner Literatur höchstens ganz marginal behandelt; Was tun? Aussage weg lassen, mich todsuchen, oder diese gewagte These nach bestem Wissen 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 [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:

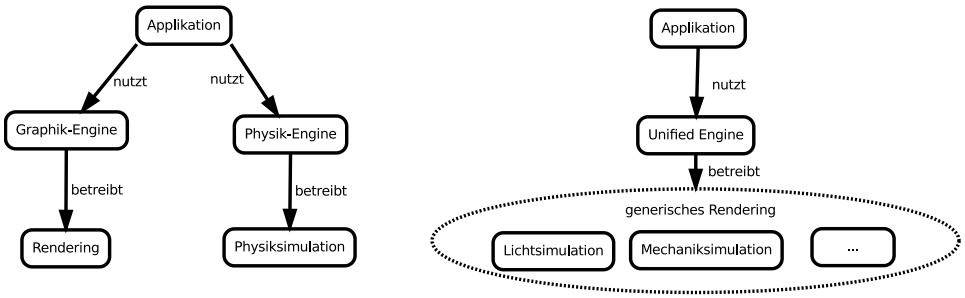

Abbildung 1: Gegenüberstellung von Verwendung und Begrifflichkeiten von klassischen Engines und einer Unified Engine

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 3 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] Zum Schreiben der GPU-Programme habe ich *gedit*[16] mit entsprechenden Syntax-Highlightings verwendet. Für die Source Code-Versionsverwaltung kam *git*[17] zum Einsatz.

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

[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.
[16] http://projects.gnome.org/gedit/
[17] http://git-scm.com/

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).

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 4.1.8.

Template-Engine Da GLSL[18] 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*[19]; 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 **??** 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 4.1.10 detaillierter vorgestellt.

Buffer-Abstraktion Der zentrale Dreh- und Angelpunkt der Unified Engine ist der *Buffer*;

[18] OpenGL Shading Language
[19] www.grantlee.org/

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 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; [20] 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

[20] 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.

- 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.)

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 3.12.

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 4.2.1.6 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. Dieses Kapitel soll die groben Konzepte und Ideen teilweise anhand von Beispielen vorstellen. Nicht jedes Feature, welches erwähnt wird, ist auch tatsächlich vollständig implementiert und getestet. Über den Status der Implementierung zum Zeitpunkt der Abgabe dieser Ausarbeitung gebe ich am Ende dieses Kapitels (Abschnitt 3.15) einen Überblick.

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 Dreiec 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“[21], 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 [22], 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

[21] beziehbar unter http://www.opengl.org/registry/
[22] die GPGPU-Computing einschließt

ist direkt von der Khronos-Homepage[23] beziehbar. Diesen Wrapper habe ich verwendet, da er die Nutzung der API wesentlich eleganter macht.

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[24] 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.

OpenGL Mathematics (GLM) Seit sämtlicher Mathematik-bezogener OpenGL-State inklusive zugehöriger built-in-Variablen und -Funktionen wie `gl_ModelViewProjectionMatrix` oder `ftransform()` in GLSL abgeschafft wurden, führt um eine Bibliothek für Vektor- und Matrix-Algebra kein

[23] http://www.khronos.org/registry/cl/
[24] http://www.glfw.org/

Weg mehr herum. Ich habe mich für *GLM*[25] entschieden, da sie klein und dennoch mächtig ist, und einige Convenience-Functions hat; QT hat ebenfalls eine Mathe-Bibliothek, verwendet jedoch double, also 64bit-Fließkomma-Werte als Basis-Datentyp, was das direkte Übergeben als Array von float-Uniforms an die OpenGL-Shader verhindert. Zwar unterstützen OpenGL und moderne Graphikkarten double nativ, jedoch mit drastisch geringerer Geschwindigkeit, da weniger Recheneinheiten für diesen Datentypen zur Vervügung stehen.
Es sei bemerkt, dass ich persönlich die direkte Verwendung einer C++ - Mathe-Bibiothek mit überladenen Operatoren und eigener Akkumulation von Matrizen wesentlich angenehmer und eleganter finde als das zähe Hantieren mit der C-API zum modifizieren des OpenGL-State mit seinen Matrix-Modes.

Grantlee Die bereits erwähnte Template-Engine; die Syntax entspricht der der Template-Sprache des *Django* web Frameworks[26].

Es lassen sich durch die Applikation Werte an die Template-Engine übergeben, anhand derer dann die Code-Generierung Kontrolliert wird, bzw welche durch durch Einschluss in doppelte geschwungene Klammern direkt eingefügt werden können.
Beipiel:

```
1 vec4 fragmentColor =
2   {% if SHADING_FEATURE_DIFFUSE_TEXTURING %}
3     texture(decalTexture,input.texCoords.xy);
4   {% else %} color;
5   {% endif %}
```

```
1 #define NUM_BITS_PER_KEY_TO_SORT ( {{numBitsPerKey}} )
```

Die Engine stellt einen Vererbungs-Mechanismus bereit:
Auf diese Weise kann z.B. die Datei „particleSimulationTemplate.cl“ verschiedene „Code-Blocks“ definieren wie z.B.

```
1 {% block performSPHCalculations %}
2   {% comment %}
3   the core of the physics simulation:
4   accumulate all relevant values
5   (density, pressure force, viscosity force etc ...)
6   {% endcomment %}
7 {% endblock performSPHCalculations %}
```

[25] http://glm.g-truc.net/
[26] https://www.djangoproject.com/

, die von anderen Datein geerbt und entsprechend angepasst implementiert werden.
„updateDensity.cl“ erbt von dieser Datei durch die Directive {% extends "particleSimulationTemplate.cl" %} und implementiert die Dichte-Berechnungen, ohne dass der umschließende (nicht unerheblich lange und komplexe) Kontrollfluss-Code wiederholt werden muss:

```
1 {% block performSPHCalculations %}
2   if( BELONGS_TO_FLUID(
3       GET_CURRENT_NEIGHBOUR_PARTICLE_OBJECT_ID  ) )
4   {
5     ownDensity +=
6       cObjectGenericFeatures [ GET_CURRENT_NEIGHBOUR_PARTICLE_OBJECT_ID  ].massPerParticle
7       * poly6( ownPosition -  GET_CURRENT_NEIGHBOUR_POS , cSimParams );
8   }
9 {% endblock performSPHCalculations %}
```

Assimp Die *Open Asset Import Library*[27] ermöglicht das Auslesen von Szenen direkt aus .blend-Dateien, dem nativen Datenformat des exzellenten Free Software – 3D- Modellierungsprogramms *Blender*[28]. Somit entfällt der Umständliche Export in ein Zwischenformat.

TinyXML Um zu gewährleisten, dass das System schon in der frühen Entwicklungsphase weitgehend ohne Recompile-benötigenden „Hard-Codes“ konfigurierbar ist, wurde TinyXML verwended, um eine XML-config-Datei zu parsen.

Außerdem haben manche Komponenten der *Boost*-Libraries[29] verwendet.

3.2 Klassendiagramm

Abbildung 2 und 3 zeigen ein vereinfachtes Klassendiagramm von *Flewnit*. Die roten Klassen stellen wichtige Meta-Informationen dar, um Kontrollfluss und Objekt-Erstellung zu delegieren. In den folgenden Unterabschnitten werden die einzelnen Komponenten detaillierter vorgestellt.

3.3 BasicObject und Memory Tracking

Die Bibliothek soll zur Laufzeit kontrolliert herunterfahr- und re-initialisierbar sein, um eine vielseitige und flexible Anwendung zu gewährleisten.

[27] http://assimp.sourceforge.net/
[28] http://www.blender.org/
[29] http://www.boost.org/

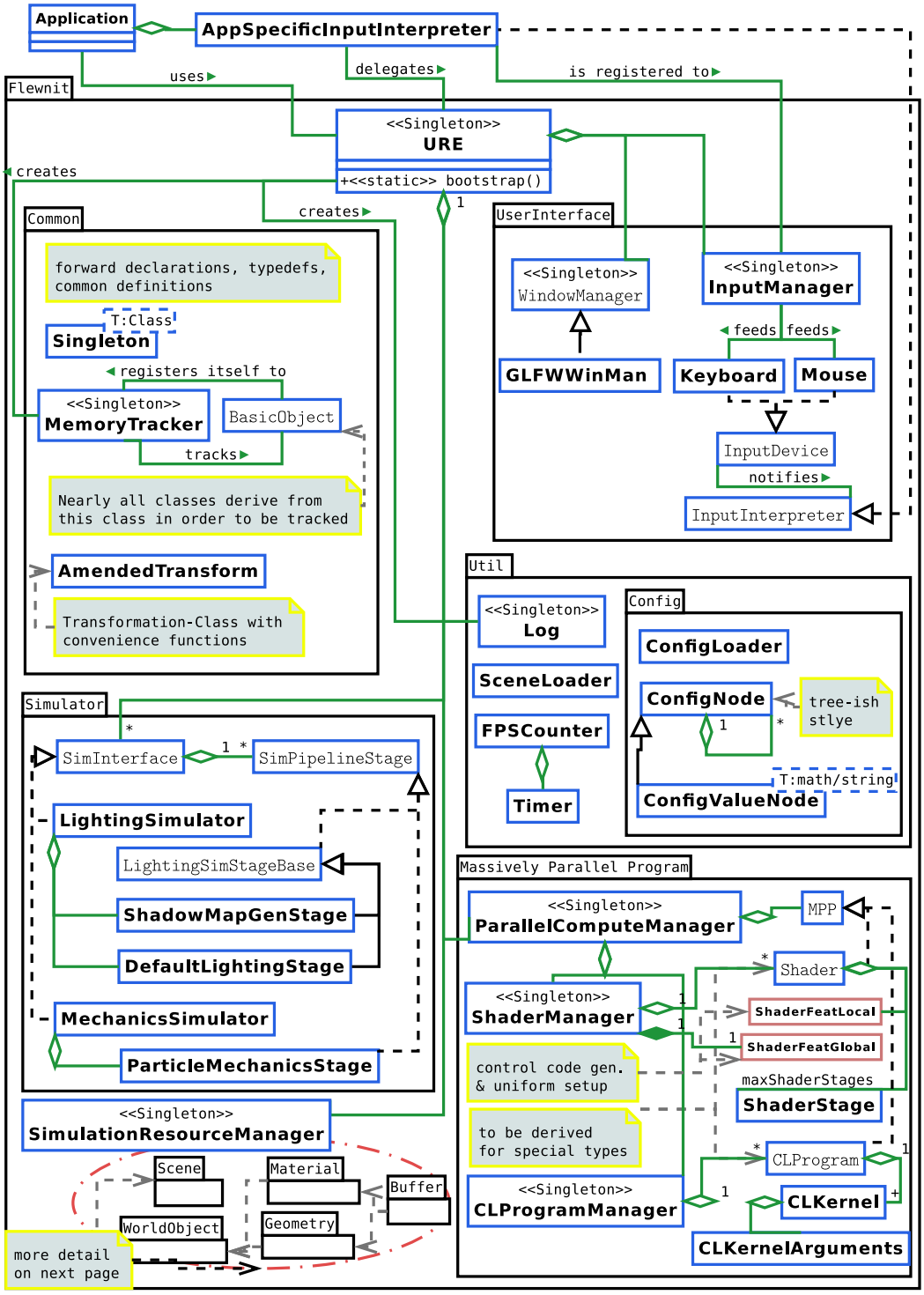

Abbildung 2: Klassendiagramm des Gesamtsystems, Teil 1

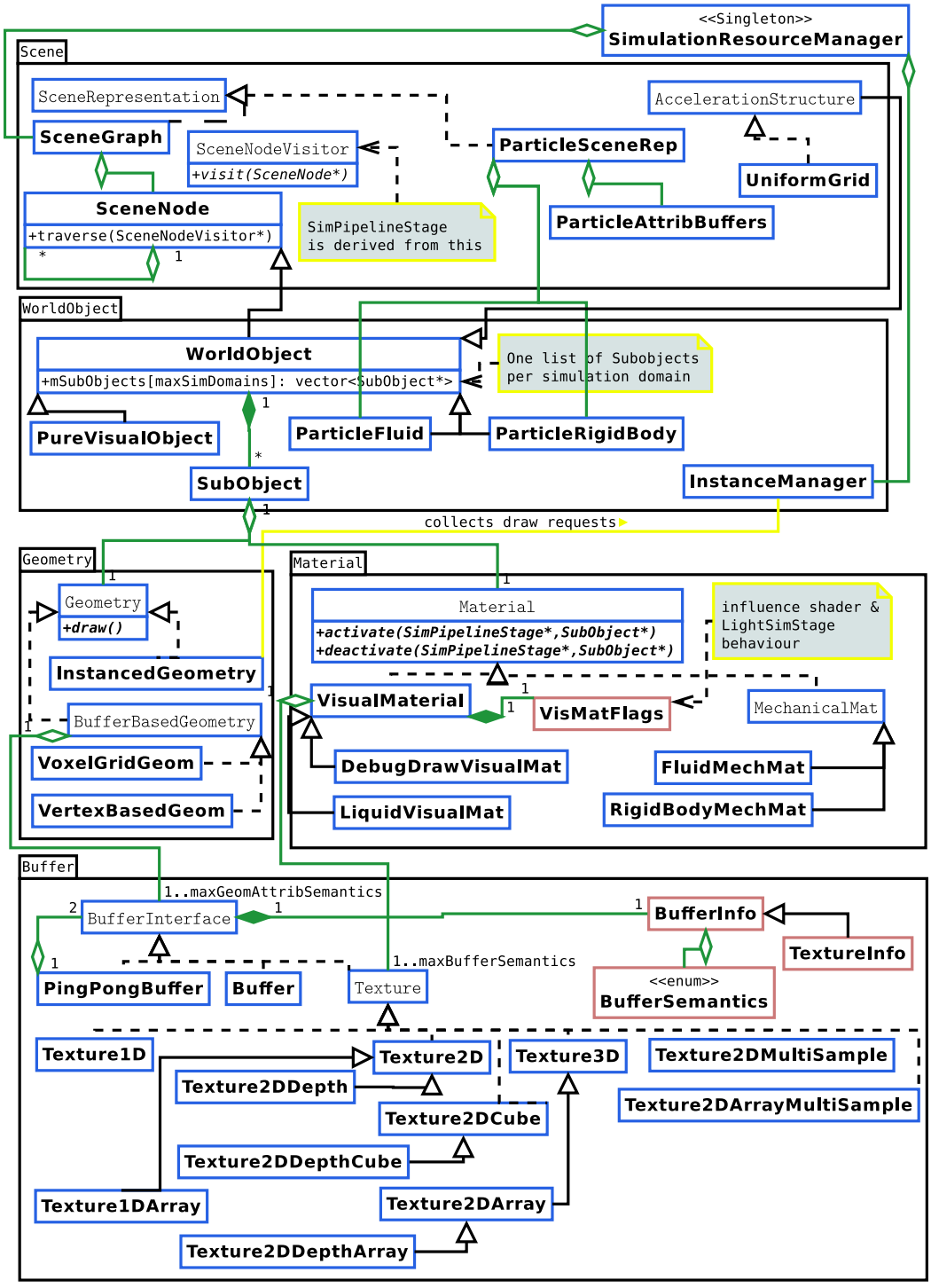

Abbildung 3: Klassendiagramm des Gesamtsystems, Teil 2

Da bei C++ das Speicher-Management dem Programmierer überlassen ist, und man daher vor allem bei massiver Nutzung von Pointern schnell den Überblick verliert,

- welches Objekt welche anderen erschaffen hat
- welche Objekte eine Klasse „besitzt“ und somit für ihre Speicher-Freigabe zuständig ist bzw
- auf welche Objekte eine Klasse nur ein Handle für Verwaltung oder Backtracking hat, ohne für Erschaffung oder Löschung zuständig zu sein

wurde in Anbetracht der oberen Forderung ein sehr simples Meta-Object-System system realisiert, welches Klassen-Namen und Speicherverbrauch eines Objektes feststellt sowie jedem Objekt eine *unique ID* zuweist. Zweck dieses Meta Object System-Systems war es, Speicherlecks durch Nicht-Löschen von Objekten während der Entwicklung zu finden;

Dafür erbt beinahe jede Klasse in *Flewnit* von `BasicObject`. Diese Klasse registriert sich automatisch bei der `MemoryTracker`-Singleton-Instanz, bekommt dort eine ID zugewiesen. Diese ID hat noch keine Anwendung, könnte sich aber in einem Netzwerk-Kontext als nützlich erweisen.

Wie bei Qt das `Q_OBJECT`-Makro muss in jede von `BasicObject` erbende Klasse das Makro `FLEWNIT_BASIC_OBJECT_DECLARATIONS` eingetragen werden; Dieses Makro definiert die Implementatoin einer virtuellen Funktion, welche die Meta-Information setzen:

```
1  #define FLEWNIT_BASIC_OBJECT_DECLARATIONS \
2    public:\
3      virtual void initBasicObject() \
4      { \
5        mMemoryFootPrint = (int) sizeof(*this); \
6        mClassName = String(typeid(*this).name()); \
7      } \
8    private:
```

Diese umständliche Implementierung hat die Urasache, dass Typ-Informationen der Blatt-Klasse einer Klassenhierarchie erst dann zur verfügung stehen, wenn alle Konstruktoren der Basis-Klassen zurückgekehrt sind, weshalb der obere Code also nicht im Konstruktor der Basisklasse stehen darf, da der this-Pointer noch nicht die Informationen der Blatt-Klasse enthält. Ferner, um dem Programmierer nicht zuzumuten, `initBasicObject()` in jedem Konstruktor aufzurufen (das führt bei tiefen Klassenhierarchien zu unnötigen ausführungen dieser Funktion, da nur die Info der Blatt-Klasse benötigt wird, außerdem entstünde so eine weitere Vergesslichkeits-Fehlerquelle, die durch das Meta Object System ja gerade *vermieden* werden soll), stellt der Memory Tracker eine Routine `updateMemoryTrackingInfo()` bereit, die aufgerufen

werden sollte, wenn man auf valide Meta-Information zugreifen will. Aufpassen muss man bei tiefen Klassenhierarchien, da hier der Compiler keinen Fehler erzeugt, wenn eine Blatt-Klasse das Makro nicht eingetragen hat, da eine Implementierung der Funktion ja bereits durch eine Oberklasse existiert.

Über bedingte Kompilierung lässt sich diese Meta-Funktionalität deaktivieren. Ein „richtiges“ Meta Object System wie das von Qt wollte ich nicht verwenden, da es für meine Zwecke zu komplex und mächtig ist, und ich nicht mit Kanonen auf Spatzen schießen wollte. Ohnehin gibt es Tools wie Valgrind, mit denen man Speicherlecks finden kann.

Letztendlich ist meine Lösung nicht wirklich elegant (auch wenn ich ohne Meta-Object-Compiler (MOC) keine bessere Lösung gefunden habe) und daher eher als eine Spielerei anzusehen mit dem Zwecke, die C++-Interna zu verinnerlichen und zu jeder Zeit daran erinnert zu werden, Destruktoren zu implementieren und sich über „Besitz- und Verwaltungs-Verhältnisse“ der Klassen Gedanken zu machen.

Der `MemoryTracker` verfolgt auch sämtliche Allokationen der `BufferInterface`↩ -Klasse, also der Basisklasse sämtlicher Buffer-Typen;

3.4 AmendedTransform

Im Zuge der Überlegungen zum Szenegraphen, der Kamera und des visuellen Renderings vieler Objekte per Hardware-Instancing entschied ich mich, die klassische 4x4-Transformationsmatrix zu wrappen und mit convenience functions anzureichern, siehe Listing 1.
Der Zweck dieser Klasse ist es, Transformationsmatrizen anhand „anschaulicher“ Parameter (Position, Richtung, Up-Vektor, Skalierung) zu definieren, und diese Parameter auch nach Akkumulationen, Modifikationen und/oder Animationen zur Verfügung zu haben.
Der Fokus liegt hier klar auf der Bequemlichkeit für den Entwickler, nicht auf der Performanz. Sobald sehr große, dynamisch animierte und tiefe Szenegraphen zum Einsatz kommen, könnte diese Klasse zum Flaschenhals werden. Dann ist vielleicht ein Refactoring vonnöten; Vorerst scheint mir die Implementation jedoch angemessen.

So lässt sich z.B. bequem ein `Camera`-Objekt, welches selber von `WorldObject`↩ erbt, welches wiederum von `SceneNode` abgeleitet ist, an eine andere SceneNode anhängen, und aus der automatisch berechneten globalen Transformation erhält man durch `AmendedTransform::getLookAtMatrix()` direkt die lookAt-Matrix, so dass die Camera-Klasse selber sich nur noch um seine Projektionsmatrix kümmern muss. Auch typische Animationen werden von der Klasse übernommen. Ob dies guter Stil ist, oder doch lieber in die SceneNode-Klasse ausgelagert werden sollte, ist eine Frage, die ich noch nicht abschließend beantworten kann. Es wurde hier dem Paradigma der Datenlokalität gefol-

gt, evtl. auf Kosten der Separation of Concerns. Eine weitere Erörtertung lasse ich aus, da das Refactoring in diesem Falle nicht zu aufwändig wäre;

Es sei bemerkt, dass durch die Menge an Matrizen, die an den Vertex Shader beim Instanced Rendering übergeben werden müssen, wenn man für alle Fälle gewappnet sein will [30], nicht unerheblich ist, da für jede Instanz ein anderes Matrix-Set übergeben werden muss; Die Menge an Daten, die per Uniform-Variable übergeben werden kann, ist begrenzt, daher hängt die maximale Anzahl an Instanzen, die pro Draw-Call gezeichnet werdne können, direkt davon ab, wie viele Daten pro Instanz übergeben werden müssen;
Dieser Umstand hat mich dazu verleitet, die Normal Matrix, die transponierte Inverse der ModelView-Matrix, „einzusparen“: da Normalen und Tangenten durch Interpolation für den Fragment Shader ohnehin normalisiert werden müssen, spielt beim Ergebnis der Transformation eines Vektors [31] nur die Richtung eine Rolle, nicht der Betrag. Die Richtung ändert sich nicht, wenn eine uniforme Skalierung in der Matrix steckt, wenn also der Skalierungsfaktor pro Raumdimension überall gleich ist. Der Translations-Anteil einer Matrix spielt bei Vektoren keine Rolle. Aus diesen Umständen lässt sich folgern, dass man zur Transformation von Vektoren ebenfalls die Model- bzw ModelView- Matrix verwenden kann, und *keine* dedizierte Normal Matrix braucht, solange nur Translation, Rotation und uniforme Skalierung in der Matrix akkumuliert sind.
Eben dies versuche ich, durch die `AmendedTransform`-Klasse zu forcieren, indem nur ein eingeschränkter public-Konstruktor existiert, direkte Initialisierung per Matrix-Konstruktor nur für bestimmte friend-Klassen erlaubt sind, und auch dann diese Matrix validiert wird. Die Einschränkung erscheint mir legitim, da eine Uniform Rendering Engine den Anspruch hat, zumindest im Ansatz „physikalisch basiert“ zu sein; Deshalb sollten sich Skalierungen ohnehin nur selten zur Laufzeit ändern, und dann nur einheitlich. So kann man im Vorfeld die Geometrie so anpassen, dass keine nicht-uniforme Skalierung in der Transformationsmatrix auftaucht.

Eine `AmendedTransform` ist insofern mit der `SceneNode`-Klasse verzahnt, dass falls eine `AmendedTransform`-Instanz als Tranformation einer Scene-Node dient, erstere ein Handle auf die Scene Node bekommt. Auf diese Weise kann die Scene Node immer automatisch informiert werden, wenn sich bei der `AmendedTransform` etwas geändert hat (es müssen dann ggfs. die Kinder der Scene-Node aktualisiert werden, z.B. für eventuelles Updade von Bounding Boxes). Auf diese Weise muss weder die `AmendedTransform` nochmal durch die `SceneNode` gewrappt werden, noch können Seiteneffekte durch direk-

[30] z.B. für Layered Rendering in verschiedene Texturen, z.B in ein Textur-Array zur Generierung mehrerer Spotlight-Shadowmaps mit einem Draw-Call; Hierbei braucht es explizit nur die Model-Matrix, da es pro Layer verschiedene View-Matrices (pro Spotlight eine) gibt, von daher nicht im Vorfeld eine ModelView Matrix akkumuliert werden kann

[31] mit homogener Koordinate 0,im Gegensatz zu Punkten, homog. Koord. 1

te Manipulation der Transformation auftreten, sollte der Programmierer vergessen, anschließend die Scenenode über ihre Transformations-Änderung zu informieren. Die Scene-Node-Kopplung ist rein intern, und braucht den Benutzer der Klasse nicht weiter zu interessieren, außer in der Hinsicht, dass Scene-Node-Manipulation direkt geschieht und die `SceneNode`-Klasse kein eigenes Interface zur manipulation von Tranformationen bereit stellt.

Listing 1: AmenededTransform Klassendefinition, gekürzt

```
1  class AmendedTransform
2  {
3  public:
4    AmendedTransform(
5        const Vector3D& position = Vector3D(0.0f,0.0f,0.0f),
6        //(0,0,-1) is assumed as initial orientation, extress any deviation↩
             in euler angles(radians)
7        const Vector3D& direction = Vector3D(0.0f,0.0f,-1.0f),
8        const Vector3D& upVector = Vector3D(0.0f,1.0f,0.0f),
9        float scale = 1.0f);
10   AmendedTransform(const AmendedTransform& rhs);
11   virtual ~AmendedTransform();
12
13   AmendedTransform operator*(const AmendedTransform& rhs)const;
14   //the assignment operators copy only the transformation values, not the↩
         SceneNode-Related info
15   const AmendedTransform& operator*=(const AmendedTransform& rhs);
16   const AmendedTransform& operator=(const AmendedTransform& rhs);
17
18   //accum: translationMatrix * rotationMatrix * scaleMatrix;
19   const Matrix4x4& getTotalTransform()const;
20   //convenience functions:
21   //unscaled, i.e. orthonormal rotation matrix:
22   Matrix3x3 getRotationMatrix()const;
23   //inverse of (translationMat*rotationMat);
24   Matrix4x4 getLookAtMatrix()const;
25   Matrix4x4 getScaleMatrix()const;
26   Matrix4x4 getInverseScaleMatrix()const;
27   static bool matricesAreEqual(const Matrix4x4& lhs, const Matrix4x4& rhs↩
         );
28   AmendedTransform getInverse()const;
29
30   /*... getter and setter for pos, dir, up omitted */
31
32   //"animation" functions:
33   void moveRelativeToDirection(float forwardBackward, float rightLeft, ↩
         float upDown);
34   //change direction by rotating it angleDegrees degrees around cross(↩
         direction,upVector)
35   void pitchRelativeToDirection(float angleDegrees);
36   //change direction by rotating it angleDegrees degrees around the ↩
         upVector;
37   void yawRelativeToUpVector(float angleDegrees);
38
39 protected:
40   //backtrace pointer to tell the scene node to update itself after its ↩
         transform has been modified directly;
41   //mOwningScenNode->transformChanged(bool global) will be called by any ↩
         setter function of this class if there is an
42   //associated SceneNode;
```

```
43    SceneNode* mOwningSceneNode;
44    //flag needed by Scene node to update itself appropriately
45    bool mIsGlobalTransform;
46
47    friend class SceneNode;
48    void setOwningSceneNode(SceneNode*node, bool isGlobalTransform)
49      {mOwningSceneNode = node; mIsGlobalTransform=isGlobalTransform;}
50
51
52    Vector3D mPosition;
53    Vector3D mDirection;
54    Vector3D mUpVector;
55    float mScale;
56
57    Matrix4x4 mAccumTranslationRotationScaleMatrix;
58
59    //construct the transformation matrix from current pos,dir,up,scale; is↩
          called by any constructor and related setter;
60    //validates members, if possible
61    void setup();
62
63
64    friend class Loader;//the loader may set transformation matrices, as ↩
          they are read directly from Scene descriptions ;)
65    //protected matrix-constructor to omit that the user passes a non-↩
          conforming matrix
66    AmendedTransform(const Matrix4x4& transform);
67
68  };
```

3.5 Die *Unified Rendering Engine* (URE)

Dies ist die Singleton-Klasse, die sämtliche Komponenten besitzt/verwaltet, und über welche direkt oder indirekt alle Daten beschafft oder Funktionalität aufgerufen werden kann, die nach außen hin verfügbar sein soll.

3.6 Das UserInterface-Paket

Das User Interface hat einen abstraktes Window-Manager-Interface, welches von einer Implementation, die *GLFW* verwendet, realisiert wird; Aus der geparsten Config wird die gewünschte OpenGL-Kontext-Version, Auflösung, Multisamples-Count, Fullscreen-Option etc. ausgelesen und ein entsprechendes Fenster mit OpenGL-Kontext erzeugt.

Die Input-Verwaltung ist im Gegensatz zum Fenstermanager nicht vollständig von der verwendeten Input-Bibliothek (ebenfallse GLFW) abstrahiert, da beim Parsen des Inputs die von GLFW definierten Makros direkt verwendet werden, wie z.B `GLFW_KEY_ENTER`. Da die Austauschbarkeit von verwendeten Bibliotheken zwar erwünscht ist, aber ich mich nicht verzetteln wollte, bleibt die Abstraktion des Inputs noch ein langfristiges „TODO“ mit geringerer Priorität.

Eine GUI wurde leider noch nicht implementiert, vor allem aus dem Grund, dass vorhandene GUI-Toolkits wie z.B. *AntTweakBar*[32] Legacy-OpenGL-Routinen verwenden, was die Verwendung eines modernen Core-Profiles ausschließen würde. Da langfristig ein konsistentes Erscheinungsblid angestrebt wird, welches seine GUI vollständig in das OpenGL- Rendering-Fenster integriert hat, kam auch die Verwendung von GUI-Toolkits wie dem von Qt nicht in Frage.
Letztendlich wollte ich schon immer mal eine kleine GUI entwickeln, wozu ich jedoch erwartungsgemäß im Rahmen dieser Arbeit keine Zeit hatte. Die GUI verbleibt also als ein „TODO“ von mittlerer Priorität.

3.7 Das Simulator-Paket

Wie bereits erwähnt, soll jede Simulationsdömane einen eigenen Simulator haben, der eine eigene Pipeline an Simulations-Stages verwaltet. Es kann theoretisch beliebig viele Simulationsdomänen geben. Momentan sind jedoch nur drei über einen enum definiert: Die visuelle, mechanische und akustische Domäne, wobei die akustische Domäne noch nicht in der prototypischen Implementierung vorkommt. Abb. 4 zeigt grob beispielhaft den Ablauf einer Simulation und die verschiedenen Varianten, wie man sich die „Rendering Results“ anderer Stages beschaffen kann:
Entweder aus der Main Loop der Engine heraus oder von der benutzenden Applikation wird `URE::stepSimulation()` aufgerufen; Diese Funktion iteriert in fester Reihenfolge über die vorhandenen Simulatoren und ruft deren `stepSimulation()`-Routine auf; Hierin werden Simulator- spezifische Operationen ausgeführt (Framebuffers des Fensters resetten und Haupt-Viewport setzen beim `LightingSimulator`, CL/GL-shared Buffers für OpenCL akquirieren beim `MechanicsSimulator`...), dann über die einzelnen zugehörigen `SimulationPipelineStage`'s iteriert, widerum deren `stepSimulation()`-Methode aufgerufen. Hier passiert dann das eigentlich *generische Rendering*. So verschieden die Algorithmen sind, die in den einzelnen Stages abgearbeitet werden, so verschieden sind auch die Möglichkeiten, wie einzelne Stages miteinander Daten austauschen: Der generischste Ansatz ist, sich von einer Stage einen Buffer mit einer bestimmten Semantik zu beschaffen, sofern die entsprechende Stage ein solchen bereit stellt. Bei Lichtsimulation, die viel mit Texturen und OpenGL-Framebuffer-Objects (FBOs) umgeht, bietet sich an, sich das `RenderTarget` (die Abtraktion des FBO in *Flewnit*) zu beschaffen, um mit dem gesamten RenderTarget weiter zu arbeiten, oder sich einzelne Texturen zu beschaffen. Es gibt aber auch die relativ indirekte Methode des Datenaustausches, wie es z.B. zwischen der `ParticleMechanicsStage` und der `ParticleLiquidDrawStage` der Fall ist: Die Buffer mit dem physikalischen Attributen sind in der SceneNode-abgeleiteten Klasse `ParticleFluid` enthalten,

[32] http://www.antisphere.com/Wiki/tools:anttweakbar

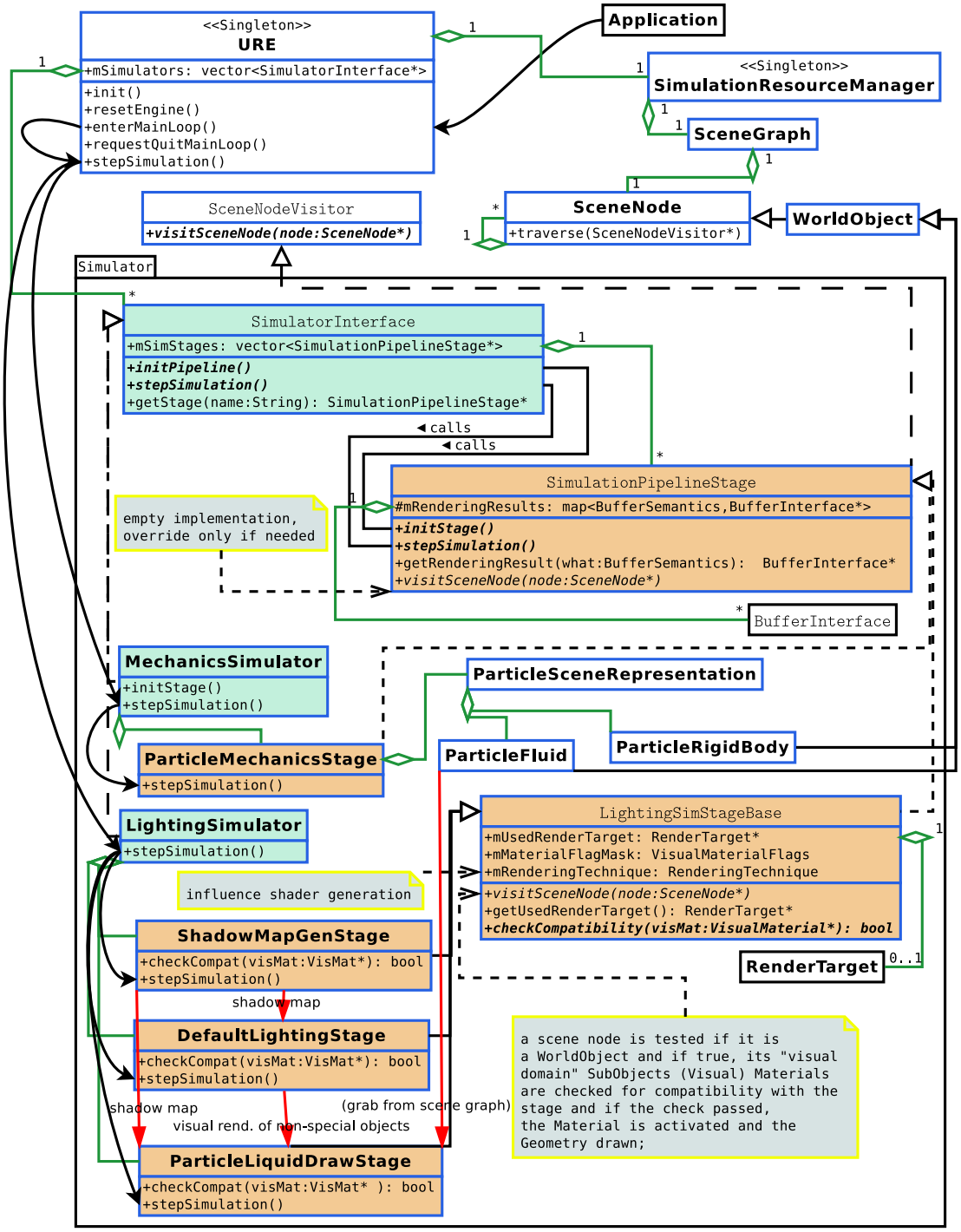

Abbildung 4: Grobes Beispiel von einer Simulation und den zugehörigen Daten-Abhängigkeiten

so dass die aktualisierten Daten über schlichtes Szenegraphen-Traversieren beschafft werden können.
Details zum Ablauf der Simulation(en) werden in Kapitel 4 behandelt.

3.8 Der SimulationResourceManger

Diese Singleton-Klasse besitzt und verwaltet die Assets, als `std::map<String,X*>` für Referenzierung über einen Namen, wobei `X` folgende Klassen betrifft:

- `Material`
- `Geometry`
- `BufferInterface`
- `Texture` [33]
- `InstanceManager`

Außerdem bestitzt die Klasse den `SceneGraph` und stellt einen Handle auf den aktuellen `SkyDome` zur Verfügung, so dass alle Objekte ein konsistentes Environment Mapping betreiben können, sofern dies erwünscht ist.

3.9 Das Scene-Paket

Neben dem klassischen Szene-Graphen, der vor allem im Kontext des visuellen Echtzeit-Rendering verwendet wird, da er Culling und relative Transformationen ermöglicht, erfordern verschiedene Simulationsdomänen und Repräsentationen der beteiligten Simulations-Objekte (z.B. als Dreiecks-Mesh, Komposition von komplexeren Primitiven, Punktwolke oder Voxel-Grid) ggfs. unterschiedliche Szene-Repräsentationen, damit die spezifischen Anforderungen der Simulation als solchen und der Respräsentation der simulierten Objekte optimal erfüllt sind.
Diese Szenen-Repräsentationen unterscheiden sich in der logischen Organisation ihrer Objekte (z.B. Baumstruktur, sortierte oder unsortierte Sammlung, Graph-Struktur, Voxel-Struktur) und in den Meta-Informationen, welche für die einzelnen Objekte notwendig sind.

Beispielsweise stellt die die effiziente Suche nach räumlich benachbarten Objekten eine wichtige Anforderung bei vielen Simulationen (auch bei Lichtsimulation für globale Beleuchtungseffekte) dar. Hierfür bieten sich verschiedene Beschleunigungsstrukturen an. Je nach Simulationsdomäne, Objekt-Repräsentation,verwendetem Simulations-Verfahren, verwendeter Hardware

[33] Da `Texture` von `BufferInterface` erbt, handelt es sich um eine Handle-Sammlung der Untermenge der Texturen aller `BufferInterface`s, um einen bequemeren Zugriff zu ermöglichen

	Szenen-Repräsentation	Beschleunigungsstruktur
Organisation der Elemente	logisch für Zugriff auf Ebene der Anwendungslogik	räumlich für effizienten Zugriff durch einen Simulations-Algorithmus
Zweck von Meta-Informationen	verschieden, je nach Repräsentation und Typ der Simulations-Objekte	(sofern vorhanden) unterschliedlich für verschiedene Algorithmen, z.b. effiziente Traversierung und/oder Nachbarschaftssuche

Tabelle 1: Gegenüberstellung der Zwecke einer generischen Szenen-Repräsentation und einer Beschleunigungsstruktur

und Zielsetzung (Geschwindigkeit, Genauigkeit, Speicherverbrauch...) sind verschiedene Beschleunigungsstrukturen angemessen. Somit besteht keine 1:1-Verbindung zwischen Szenen-Repräsentation und Beschleunigungsstruktur.
Beschleunigungsstruktur und Szenen-Repräsentatioen sind somit nur lose gekoppelt. Um die verschiedenen Zwecke zu verdeutlichen, stellt Tabelle 1 diese gegenüber.

Es gibt also zwei Basis-Klassen, wie in Abb. 3 zu sehen ist: `SceneRepresentation`↩ und `AccelerationStructure`. `AccelerationStructure` erbt von `WorldObject`(siehe Abschnitt 3.10), um optional Debug-Draw-Funktionalität bereit zu stellen. Ansonsten handelt es sich schlicht um abstrakte Basis-Klassen ohne weitere Funktionalität.

Für das gesamte System über den `SimulationResourceManger` verfügbar ist der `SceneGraph`. Dieser enthält die Wurzel-`SceneNode`, welche beliebig viele Kinder derselben Klasse haben kann. Somit wird eine Baumstruktur gebildet. Die SceneNode bildet die Basis-Klasse des `WorldObject` (siehe Abschnitt 3.10).

Es ist ein Mechanismus konzipiert, sowohl Kinder als auch Vorfahr über Änderungen der eigenen Transformation und/oder Bounding Box zu informieren, so dass ggfs. globale Transformationen akkumuliert werden können und/oder die Bounding Box für den eigenen Unterbaum ermittelt, die für Culling-Berechnungen verwendet werden kann. Noch ist dieses Feature nicht in Nutzung und seine Implementation unvollständig und ungetestet, daher wird auf weitere Details verzichtet.

Die klassischen Operationen (Einfügen, Aushängen, rekursives Suchen im Unterbaum nach Namen, Node bewegen, Transformationen akkumulieren) sind jedoch implementiert. Es sei an die enge Kopplung dieser Klasse an `AmendedTransform` (s. Abschnitt **??**) erinnert.

Der Scenegraph kann nach dem Visitor- Design Pattern traversiert wer-

den. Z.B. erbt `SimulationPipelineStage` von `SceneNodeVisitor`, so dass bei Bedarf eine Pipeline-Stage nur `virtual void visitSceneNode(SceneNode* node)` implementieren muss, um Operationen auf oder mit den Objekten des Szenegraphen auszuführen. Momentan „besitzt“ die `SceneNode`-Klasse seine Kinder, was den Vorteil hat, dass die Nodes nicht woanders verwaltet werden müssen, aber den Nachteil, dass ausgehängte Scene Nodes nicht automatisch beim Reset der Engine gelöscht werden. Sollte sich dies später als Problem herausstellen, könnte man die Scene Nodes auch vom `SimulationResourceManger`↩ verwalten lassen. Auch empfiehlt sich langfristig die Möglichkeit zur Erschaffung und Verwaltung mehrerer Szenegraphen. Aus Zeitgründen habe ich es jedoch erstmal bei dieser rudimentären Implementation belassen.

Intern verwaltet die `ParticleMechanicsStage`, die `SimulationPipelineStage`↩ , die im `MechanicsSimulator` für die partikelbasierte Simulation zuständig ist, eine `ParticleSceneRepresentation`. Diese verwaltet

- die „makroskopischen Objekte“, die an dieser Simulation teilnehmen, namentlich (allesamt von `WorldObject` abgeleitete) partikelbasierte Fluide, partikel-repräsentierte Rigid Bodies und statische Kollisions-Meshes [34]

- die einzelnen Attribute-Buffers der Partikel (Position, Dichte, Geschwindigkeit, Beschleunigung usw.)

- Buffers mit Meta-Informationen der „makroskopischen Objekte“, die an OpenCL-Kernels übergeben werden

Ferner stellt die Klasse Factory-Methoden für die makroskopischen Objekte bereit, da diese direkt mit den Partikel-Buffern assoziiert sind, sich also alle beteiligten Objekte die Buffers teilen müssen. Die korrekte Delegierung der Offsets und Sicherstellung der richtigen Größen wird so übernommen. Die `ParticleMechanicsStage` nutzt außerdem ein `UniformGrid` für die Simulation. Details hierzu werden im Verlauf des Kapitels 4.2 beschreiben.

3.10 Das WorldObject

Das `WorldObject` stellt die abstrakte Basisklasse für sämtliche Objekte dar, mit denen eine oder mehrere Simulationen durchgeführt werden sollen[35]. Sie erbt von `SceneNode`. Damit ist sie bequem in den Szenegraphen integrierbar und hat außerdem schon seinen Transformations-Membervariable.

Bei dieser Klasse zeigt sich erstmals die angestrebte Symmetrie zwischen sen Simulationsdömanen: Für jede Simulationsdomäne gibt es eine

[34] diese müssen in einem bestimmten Format vorliegen

[35] Und sei es nur Debug Drawing; Selbst dies kann als eine rudimentäre Form von Lichtsimulation aufgefasst werden.

(möglicherweise leere) Liste an `SubObject`'s. Das `SubObject` stellt eine Abstraktion dessen dar, was in einer klassischen Graphik-Engine ein „SubMesh“ oder in einer klassischen Physik-Engine eine Komponente einer „(Compound) Collision Shape“ sein könnte.
Ein `SubObject` hat immer einen Pointer auf genau ein `Material`, eine `Geometry`↩ und für Backtracking-Zwecke auf sein besitzendes `WorldObject`. Wie der Backtracking-Pointer andeutet, besitzt das ein `WorldObject` jedes seiner `SubObject`↩ s. Die vom `SubObject` genutzte Geometrie und Materialien können jedoch –sofern sinnvoll – von beliebig vielen `SubObject`s gemeinsam genutzt werden.

Es sind verschiedenste Ableitungen von `WorldObject` denkbar, z.B. (standard/particle based/voxel based) Rigid Body, Soft Body, rein visuelles Objekt, Kamera,Lichtquelle, Debug-Draw-Objekte, Cloth, Hair, oder (particle based/voxel based) Fluid.

Welche Objekte im Szenegraph für die Nutzung in einer Simulations-Stage in Frage kommen, lässt sich entweder über eine Vorfilterung bei Objekt-Erstellung (Factory-Pattern oder automatische Registierung bei einer Manager-Klasse im Konstruktor), oder über dynamic_cast<>()'s und anschließendes Auslesen von spezifischer Meta-Information ermitteln, oder aber durch „property flags“, also Bit-Flags, die Features einer Scene Node andeuten. Welche dieser Methoden langfristig die robusteste, flexibelste und eleganteste ist, vermag ich noch nicht zu sagen. Da Typ-Informationen über enums (als Bitflags oder „normale“ Aufzählungstypen) eine gewisse Redundanz hat ggü. der dynamischen Typ-Information, die C++ bereit stellt, und außerdem eine weitere Fehlerquelle für den Programmierer darstellt, sind die Bitflags nicht erste Wahl. Außerdem schränkt die explizite Auflistung von Features womöglich die Erweiterbarkeit des Systems ohne komplette Neu-Kompilierung ein. Andererseits machen sie Programme vielleicht lesbarer, indem bestimmte Kategorien/Features explizit aufgelistet (und im Falle von Bitflags beliebig kombinierbar) sind, und nicht implizit in C++-Typinformationen codiert sind. Ich verwende Aufzählungstypen intensiv in *Flewnit*. Nicht immer sind diese für die Programmlogik nötig. Sie halfen mir jedoch zur Strukturierung und Ideen-Sammlung. Letztere ist wichtig, da bei einem Softwaresystem, welches generische Verwendbarkeit anstrebt, schon weit vor Implementierung konkreter Klassen die Struktur der Basisklassen anhand von abstrahierten (erwarteten) gemeinsamen Features der möglichen abgeleiteten Klassen möglichst exakt bestimmt werden sollte. Je weiter oben in der Klassenhierarchie Design-Fehler auftauchen, desto aufwändiger droht das spätere Refactoring zu werden.

3.11 Material

Der Begriff „Material'", der sich in der Computergraphik eingebürgert hat als Beschreibung von visuellen Eigenschaften einer Oberfläche, erscheint

mir prädestiniert, sich in seiner Bedeutung in einem alltäglichen, nicht computergraphischen Kontext wieder anzunähern: das *Material* in *Flewnit* steht für „sämtliche Eigenschaften von Materie außer ihrer geometrischen Form und internen Repräsentation dieser“.

Abbildung 5: Der Begriff *Material* mit einer physikalisch orientierten Bedeutung: sämtliche Eigenschaften von Materie außer ihrer geometrischen Form. (Bild entnommen aus [APS00])

Würden wir über die entsprechende Rechenleistung verfügen und als Programmierer das physikalische Know-How besitzen (z.b. über Maxwell'sche Gleichungen), bräuchte man dieses Konzept gar nicht weiter zu konkretisieren, und könnte sämtliches Verhalten von Simulations-Objekten durch subatomare Material-Eigenschaften modellieren. Dies wäre eine "wirkliche"Vereinheitlichung der bestehenden Engine-Konzepte, die unserer Realität Rechnung trüge. Da wir jedoch realistisch (im Sinne der Machbarkeit) bleiben wollen jetzt und heute Echtzeit-Fähigkeit, auch auf Kosten von physikalischer Korrektheit/Genauigkeit anstreben, sind Domänen-spezifische Konkretisierungen des Material-Konzeptes notwendig, und wie so viele andere Klassen in diesem System wird die `Material`-Klasse zur abstrakten Oberklasse. Bei einem `SubObject`, welches der visuellen Simulationsdömane angehört, wird implizit erwartet, dass das genutzte `Material` ein `VisualMaterial` ist (mit Texturen, Shader, und verscheiedenen Meta-Informationen, um das visuelle Rendering zu delegieren, siehe Abschnitt 4.1.3), und in der mechanischen Domäne muss es ein `MechanicalMaterial`(mit Eigenschaften wie Masse, Reibung, Elastizität etc.) sein. Diese domänenspezifischen Material-Typen können oder müssen (je nach Typ des nutzenden `WorldObject`s) noch weiter abgeleitet sein.

Einen etwas schalen Beigeschmack haben die Routinen:

```
1 virtual void activate(
2     SimulationPipelineStage* currentStage,
3     SubObject* currentUsingSuboject) throw(SimulatorException){}
4 //undoing stuff, like re-enable depth test etc.
5 virtual void deactivate(SimulationPipelineStage* currentStage,
6     SubObject* currentUsingSuboject) throw(SimulatorException){}
```

Sie sind dem Statemachine-basierten visuellen Rendering mit OpenGL zu verdanken und müssen daher von den `VisualMaterial`-Klassen implementiert werden. Um der Möglichkeit Rechnung zu tragen, dass ähnlich Statebasierte Mechanismen an anderer Stelle als beim OpenGL-Rendering vorkommen sollten, sind diese Routinen in die Oberklasse gelangt.

3.12 Die Buffer-Abstraktion

Logischerweise müsste nun eigentlich – der Top-Down-Präsentation der einzelnen Pakete und Klassen entsprechend – die `Geometry`-Klasse vorgestellt werden. Um letztere Klasse jedoch besser zu verstehen, ist die Kenntnis der Buffer-Abstraktion vorteilhaft. Die `Geometry` wird in 3.13 beschrieben.

Wie auf Seite 12 erwähnt, abstrahiert die `BufferInterface`-Klasse sämtliche Buffer-bezogene Funktionalität, namentlich für Host-Buffer, OpenCL-Buffer, verschiedene OpenGL-Buffertypen und letztendlich verschiedene Texturtypen (eine Teilmenge davon ist OpenCL-interop- kompatibel bzw. als reine OpenCL-Textur erschaffbar). Es ergibt sich ein wahrer Moloch, der für eine relativ einheitliche Handhabung zu abstrahieren ist:

- verschiedene APIs (OpenGL C-Api „vs." OpenCL C++-API)
- verschiedene Verwendungen und Verfügbarkeiten schon allein von non-Textur-Buffern (generisch, Vertex Index/Vertex Attribute Buffer, Uniform Buffer, Render Buffer), siehe Tabelle 2
- verschiedene OpenGL-Texturtypen mit nur bedingt kombinierbaren und nur teilweise zu OpenCL kompatiblen erweiterten Features, siehe Tabelle 3
- verschiedene Datenformat/Channel-Layout-Deskriptoren, siehe Listing 7

Es hat schon sehr viel Zeit gekostet, sich aus den Spezifikations-Dokumenten ([Khr10], [Khr09]) die möglichen Permutationen, Zusammenhänge, Entsprechungen und unterstützten Operationen zu erarbeiten (siehe Tabellen 2 und 3).

Die Ergebnisse dieser Tabellen flossen in die Entscheidung ein, welche konkreten Buffer- und Textur-Klassen mit welchen verfügbaren Features durch Ableitung vom `BufferInterface` implementiert wurden. Die Konstruktor-Parameter sollten so spezifisch zugeschnitten sein, dass der Benutzer der Klassen möglichst wenig invalide Kombinationen angeben kann. Außerdem sollte er sich nicht um all die Makros wie `GL_RGBA32UI` bzw `CL_RGBA` und `CL_UNSIGNED_INT32` kümmern müssen [36]. Es soll ein Buffer mit einem einzi-

[36] Was in OpenCL getrennt ist und somit „nur" 4 Channeltypen + 3 Bitgrößen * (3 non-normalized + 2 normalized)= theoretisch 19 makros ergibt, braucht bei OpenGL 4 Channeltypen * 3 Bitgrößen * (3 non-normalized + 2 normalized)= theoretisch 60 makros; Erstens stört die Asymmetrie, zweitens das „Zusammenklatschen" eigentlich unabhängiger Parameter in ein Makro

gen Konstruktor-Call völlständig definiert, falls erwünscht auf dem Host-Memory und – sofern unterstützt – in den gewünschten API-Contexten allokiert bzw registiert werden.

Hierfür mussten einige enums und Meta-Info-Klassen/Structures definiert werden:

enum Type Beschreibt String-, boolsche, Skalar-, Vektor- und Matrix-Datentypen mit verschiedener Genauigkeit. Findet in *Flewnit* u.a. auch beim Config-Parsen Verwendung.

enum ContextType und ContextTypeFlags Mithilfe dieser Datentypen lässt sich spezifizieren, in welchem Kontexten (Host, OpenGL, OpenCL) man einen Buffer erstellt/allokiert/registriert haben will.

enum BufferSemantics Eine sehr wichtige Auflistung konkreter Verwendungszwecke des Buffers, siehe Listing 5 im Anhang A. Dieser Aufzählungstyp ist eine große Erleichterung beim Hantieren mit OpenGL Vertex Buffer Objects (VBO's) [37] und Frame Buffer Objects (FBO's) [38], da hier zur Assoziation von VBO-Vertex Attribute Buffern mit Input-Variablen im Vertex Shader bzw ans FBO gehängte Texturen mit Output-Variablen im Fragment Shader ein Index angegeben werden muss. Wenn wir als Index die numerische Entsprechung der Buffer-Semantik verwenden, ist nicht nur die Lesbarkeit des Codes erhöht, es ist auch automatisch sichergestellt, dass Indizes keine Konflikte versursachen!

enum GPU_DataType Listet nur die von der GPU / OpenGL nativ unterstützten Datentypen auf, ohne Präzisions- und Normalisierungs-Information:

```
1 enum GPU_DataType
2 {
3   GPU_DATA_TYPE_FLOAT,
4   GPU_DATA_TYPE_INT,
5   GPU_DATA_TYPE_UINT
6 };
```

enum GLBufferType Listet die vom Buffer-Interface abstrahierten OpenGL-non-Textur-Buffertypen auf. Aus Zeitgründen und Erwartung, dass sie außerhalb der `RenderTarget`-Klasse nicht benötigt werden, werden z.B. Render Buffers nicht abstrahiert.

```
1 enum GLBufferType
2 {
```

[37] abstrahiert von `Flewnit::VertexBasedGeometry`, s. 3.13

[38] abstrahiert von `Flewnit::RenderTarget`

```
  NO_GL_BUFFER_TYPE,
  VERTEX_ATTRIBUTE_BUFFER_TYPE,
  VERTEX_INDEX_BUFFER_TYPE,
  UNIFORM_BUFFER_TYPE
};
```

struct BufferElementInfo Beschreibt – sofern erwünscht bzw nötig – das Channel-Layout,die Datentypen, die Bit-Genauigkeit und die etwaige Normalisierung eines Buffers mit Channels, z.B. einer Textur oder einem Vertex Attribute Buffer, siehe Listing 7 im Anhang A. Auf diese Weise spart man dem Benutzer die verschiedenen „zusammengesklatschten“, asymmetrischen GL/CL-Makros; Die Structure validiert ihre Daten. Diese werden später für die internen CL/GL-API-Aufrufe in die entsprechenden CL/GL-Makros transformiert, siehe Listing 8 im Anhang A.

struct GLImageFormat OpenCL definiert folgende Structure:

```
typedef struct _cl_image_format {
    cl_channel_order        image_channel_order;
    cl_channel_type         image_channel_data_type;
} cl_image_format;
```

Um das interne Handling der CL/GL-Makros etwas symmetrischer zu machen (man verliert leicht den Überblick), habe ich ein ähnliches Makro definiert, welches aufgrund der Kommentare im Anhang A, Listing 6 zu finden ist.[39]

class BufferInfo Dies ist die Klasse, die sowohl als „Construction Info“ den Konstruktor-Code delegiert, als auch für den Benutzer später Meta-Information bereitstellt. Es sei hier nur der Standard-Konstruktor angegeben:

```
explicit BufferInfo(String name,
  ContextTypeFlags usageContexts,
  BufferSemantics bufferSemantics,
  Type elementType,
  cl_GLuint numElements,
  const BufferElementInfo& elementInfo,
  GLBufferType glBufferType = NO_GL_BUFFER_TYPE,
  ContextType mappedToCPUContext = NO_CONTEXT_TYPE);
```

Ich muss zugeben, dass `elementType` und `elementInfo` redundant sind, außerdem diese beiden Parameter eigentlich bei generischen Buffern,

[39] Die Kommentare geben weiteren Aufschluss über die Asymmetrie der einzelnen APIs: obwohl sie beide von der Khronos Group spezifiziert wurden und explizit Interoperabilität ermöglichen, stiften viele mehr oder minder kleine Asymmetrien Verwirrung (zumindest bei mir); Diese Umstände trugen zu der Entscheidung bei, das `BufferInterface` zu entwickeln, trotz des erheblichen Zeitaufwands.

wo die Typ-Information zumindest für die GPU Computing APIs keine Rolle spielt, fehl am Platze sind. Mangelndem Überblick in der Anfangsphase der Implementierung über die verschiedenen Arten, Buffers in CL und GL zu nutzen, sind diese Design Flaws zuzuschreiben. Ein Refactoring mit spezieller zugeschnittenen Konstruktoren ist geplant.

class TextureInfo So wie `Texture` von `BufferInterface` erbt, erbt auch `TextureInfo` von `BufferInfo`. Sie stellt die gleiche Funktionalität für die (abstrakte) `Texture`-Klasse dar wie `BufferInfo` für `BufferInterface`. Ein Unterschied besteht jedoch darin, dass `TextureInfo` eigentlich nur dazu dient, die kombinierte Information aus den Parametern der „eingeschränkten" Konstruktoren der konkreten Textur-Klassen (`Texture2D` etc.) und der inhärent klassenspezifischen Eigenschaften an die `Texture`-Oberklasse zu übergeben. Damit hat `TextureInfo` eher eine Textur-interne Funktion bei der Konstruktion. Das tut jedoch seiner Funktion als späterer Meta-Daten-Lieferant keinen Abbruch. Auch hier reicht der Standard-Konstruktor für eine Idee dieser Klasse:

```
1  explicit TextureInfo(
2    const BufferInfo& buffi,
3    cl_GLuint dimensionality,
4    Vector3Dui dimensionExtends,
5    GLenum textureTarget,
6    bool isDepthTexture = false,
7    bool isMipMapped = false,
8    bool isRectangleTex = false,
9    bool isCubeTex = false,
10   GLint numMultiSamples = 1,
11   GLint numArrayLayers = 1
12   );
```

Es sei bemerkt, dass die Tiefen-Texturen sich abgesehen von den automatisch bestimmten entsprechenden OpenGL-Makros in ihrer Implementation nur geringfügig von ihren „Farbtextur"- Oberklassen unterscheiden, nämlich in der Implementation der `virtual void setupDefaultSamplerParameters()=0;`-Methode, die in `Texture` definiert ist. Hier wird z.B. der Compare-Mode und die Compare-Function für korrekten Lookup solcher Texturen als Shadow Map über einen Shadow-Sampler in GLSL gesetzt.

Man darf sich fragen, warum ich jede noch so sonderbar anmutende Spezial-Textur unterstützen will. Dies hat den Grund, dass ich langfristig etliche Rendering Features unterstützen will, welche die meisten dieser Spezialtexturen benötigen, und die verbleibenden paar ungenutzten Texturen noch zur Vollständigkeit mit zu implementieren, verursacht dank Vererbung nur einen minimalen Mehraufwand:

1. Layered Rendering in `Texture2DDepthCube` zur Generierung von Point Light Shadow Maps in nur einem Rendering Pass

		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	?	✓	?

Tabelle 2: Verschiedene Buffertypen und ihre Verfügbarkeit in verschiedenen Kontexten
Legende:
✓ → nativ unterstützt; o → kompatibel; x → nicht unterstützt;
? → Unterstützung abhängig von weiteren Parametern, s. Tabelle 3;

	CL interop	MipMap	Depth	Array	Rectangle
Texture1D	x	✓	x	✓	x
Texture2D	✓	✓	✓	✓	✓
Texture2DCube	o	✓	✓	o	x
Texture2DMultiSample	x	x	x	✓	x
Texture3D	✓	✓	x	x	x

Tabelle 3: Verschiedene Texturtypen und ihre Kompatibiltät zu bestimmten Features
Legende:
✓ → unterstützt; o → In OpenCL unterstützt, aber nicht vom Framework; o → nur in OpenGL 4 unterstützt, daher wegen Kompat. zu GL 3 nicht vom Framework unterstützt; x → nicht unterstützt;

2. Layered Rendering in `Texture2DCube` zur Generierung von dynamischen Environment Maps

3. Layered Rendering in `Texture2DDepthArray` zur Generierung von vielen Spot Light Shadow Maps in nur einem Rendering Pass

4. Deferred Shading mit Multisampling: Benötigt `Texture2DMultiSample`↩ -G-Buffers

Zu weiten Teilen sind der Shadercode und die nötigen Framework-Klassen schon für diese Features implementiert, es muss „nur“ noch einiges ergänzt und aufgeräumt, alles zusammengefügt und debuggt werden. Da dieses „Nur“ wirklich äußerst ironisch gemeint ist, habe ich gar nicht erst versucht, die Implementierung dieser Features zu beenden. Es muss vorerst

reichen, dass sie konzeptionell in der Framework-Struktur angelegt sind.

Alle Operationen auf den Buffern/Texturen durch den Benutzer werden dann über das `BufferInterface` getätigt, siehe das exemplarische Listing 2. Fast sämtlicher Kontrollfluss (Synchronisation,Validierung, Auswahl des richtigen Kontextes etc.) befindet sich in der Implementation der (wohlgemerkt großteils *nicht* virtuellen) Methoden von `BufferInterface`. Nur die finale Bürde der verschiedenen API-Calls als Ergebnis des vorangegangenen Kontrollflusses wird von protected virtuellen Funktionen der abgeleiteten Klassen übernommen, wie im folgenden Unterabschnitt beschrieben.

Listing 2: Operationen auf dem BufferInterface, Ausschnitt

```
1  //The binding-machanism is only relevant for OpenGL
2  void bind()throw(BufferException);
3  void setData(const void* data, ContextTypeFlags where)throw(↩
       BufferException);
4  //if both CL and GL are enabled, then the buffer is shared and the ↩
       implementation
5  //will decide which API will be used for the read/write;
6  void copyFromHostToGPU()throw(BufferException);
7  void readBack()throw(BufferException);
```

3.12.1 Die API-bezogenen internen Buffer-Operationen

Zur Vermeidung von Boilerplate-Code sind die Signaturen für die GPU Computing API-bezogenen Buffer-Operationen ausgelagert in eine Datei namens „BufferVirtualSignatures.h“. Um sowohl in der abstrakten Basisklasse als auch in den abgeleiteten Klassen included werden zu können, lässt sich die Abstraktheit der Definitionen über das Makro `FLEWNIT_PURE_VIRTUAL`↩ steuern:

Listing 3: API- und Buffertyp-abhängige Operationen auf Buffern – Definitionen

```
1  #ifdef FLEWNIT_PURE_VIRTUAL
2  # define PURENESS_TAG =0
3  #else
4  # define PURENESS_TAG
5  #endif
6
7    virtual void generateGL() PURENESS_TAG;
8    virtual void generateCL() PURENESS_TAG;
9    virtual void generateCLGL() PURENESS_TAG;
10
11   //there is no pendant to bind() in OpenCL, as the API needs no buffer ↩
         binding mechanism;
12   //instead, buffers are passed as arguments to kernels
13   virtual void bindGL() PURENESS_TAG;
14   //there is no pendant to alloc() in OpenCL, as Buffer generation and ↩
         allocation are coupled within
15   //the same API call
```

```
16    virtual void allocGL()PURENESS_TAG;
17
18    virtual void writeGL(const void* data) PURENESS_TAG;
19    virtual void writeCL(const void* data) PURENESS_TAG;
20    virtual void readGL(void* data) PURENESS_TAG;
21    virtual void readCL(void* data) PURENESS_TAG;
22    virtual void copyGLFrom(GraphicsBufferHandle bufferToCopyContentsFrom) ↩
          PURENESS_TAG;
23    virtual void copyCLFrom(ComputeBufferHandle & bufferToCopyContentsFrom)↩
           PURENESS_TAG;
24    //because the C++ Wrapper for OpenCL is used, deletion takes place ↩
          automatically when th cl::Buffer is destroyed
25    virtual void freeGL() PURENESS_TAG;
26
27  //  //not implemented yet as not needed at the moment: mapping routines:
28  //  virtual void* mapGLToHost() PURENESS_TAG;
29  //  virtual void* mapCLToHost() PURENESS_TAG;
30  //  virtual void* unmapGL() PURENESS_TAG;
31  //  virtual void* unmapCL() PURENESS_TAG;
32
33  #undef PURENESS_TAG
```

So included `BufferInterface` diese Datei innerhalb ihrer Klassendefinition über

```
1  # define FLEWNIT_PURE_VIRTUAL
2  # include "BufferVirtualSignatures.h"
3  # undef FLEWNIT_PURE_VIRTUAL
```

Die anderen Klassen definieren das Makro nicht vor dem Includen.

In Anhang A sind exemplarisch Implementationen dieser Funktionen gelistet, einerseits die der konkreten `Buffer`-Klasse (Listing 9), die alle non-Textur-Buffer zusammengefasst abstrahiert, andererseits die der konkreten `Texture2D`-Klasse (Listing 10 und 11), die von der abstrakten `Texture`-Klasse erbt und selbst die Basisklasse für `Texture2DDepth` und `Texture1DArray` bildet, vgl. Klassendiagramm (Abb. 3). Die meisten Signaturen aus „BufferVirtualSignatures.h“ werden schon von `Texture` implementiert, da nicht jede API-Routine Textur-spezifisch ist.
Ich muss gestehen, dass ich diese Funktionen noch nicht systematisch getestet habe, daher Fehlerfreiheit nicht garantiert ist. Es sei nochmals betont, dass der Schwerpunkt bei einer möglichst generischen Konzeption lag, nicht bei einer vollständigen Implementation. Es ist viel „schlafender“ Code im System, der noch getestet, integriert, weiter entwickelt, ergänzt werden will.

3.12.2 PingPongBuffer

Auch die für viele Algorithmen aufgrund ihrer inhärenten Struktur oder aufgrund der Beschränkungen bei der Synchronisation parallelen Codes

nötige „Ping-Pong“-Funktionalität [40] ist durch das `BufferInterface` abstrahiert: Der `PingPongbuffer` verwaltet intern zwei andere `BufferInterface`-Objekte und leitet die Methoden-Aufrufe, die durch das `BufferInterface` definiert sind, an den aktuell aktiven Buffer weiter. Der Rollentausch geschieht dann über `PingPongBuffer::toggleBuffers()`. Bei komplexeren Algorithmen, wei denen viele Toggles passieren, kann man schnell den Überblick verlieren, welcher Buffer jetzt gerade aktiv ist; Besonders kritisch ist dieser Umstand dort, wo ein OpenGL Buffer-Handle gebunden ist, oder ein OpenCL-Buffer-Handle als Argument für einen OpenCL-Kernel gesetzt ist. Diese Bindings lassen sich nicht von einem `toggleBuffers()` beeindrucken, da die `BufferInterface`-Klasse selbst nicht weiß, in in welche Hierarchie an verschiedensten Binding-Points seine verwalteten Handles gerade integriert sind oder nicht. Um dem Programmierer die Bürde zu nehmen, neben den Toggles auch noch ständig die CL/GL-Bindings zu ändern, wurden Mechanismen implementiert, die diese Operationen automatisch tätigen, nämlich bisher in `VertexBasedGeometry`↩ um Attribute und Index Buffer Bindings konsitent zu halten (s. Abschnitt 3.13.2) und in `CLKernelArgument` (s. Abschnitt 3.14.3.1), um die Kernel-Arguments nicht „von Hand“ immer neu setzen zu müssen.

3.13 Geometry

Die `Geometry`-Klasse bildet die abstrakte Basisklasse verschiedenster geometrischer Reräsentationen.
Beispiele solcher Repäsentationen sind:

- Vertex-basiert: Punkte, Linien(-Züge), Dreicke, Quads und beliebige Polygone lassen sich theoretisch aus ihnen konstruieren [41]
- Voxel-basiert
- Freiform-Flächen-basiert: NURBS, Bézier-Patches etc.
- Primitiv-Basiert: das Objekt ist aufgebaut aus Primitiven die Quader, Kugel, Kapseln, Zylindern etc.; Diese Repräsentation wird gerne für Collision Shapes bei der mechanischen Simulation von Rigid Bodies verwendet.

Die Repräsentationen Überschneiden sich in ihren Kategorisierungen und Eigenschaften, eine sinnvolle Strukturierung ist also weder trivial noch womöglich eindeutig. Die API-Spezifischen Beschränkungen erleichtern diese Aufgabe nicht gerade, wenn man das Effiziente Mapping auf eine API im Hinterkopf behalten muss. Ich habe mir nicht weiter den Kopf zerbrochen,

[40] also die Verwendung von zwei Buffern, einem zum Lesen, einem zum Schreiben, und die Rollen werden nach jedem Schritt getauscht

[41] theoretisch deshalb, weil z.B. OpenGL maximal Dreiecks-Primitive unterstützt, im Falle von Tessellation noch Quads, aber das ist buchstäblich ein anderes Kapitel

auch, weil mir spezifisches Wissen z.B. über Primitiv-basierte Rigid Body-Simulation fehlt, und habe mich pragmatisch in der exemplarischen Implementierung auf die beiden Basis-Repräsentationen beschränkt, die bei aktuellem visuellen Echzeitrendering und interaktiver Fluidsimulation die wichtigste Rolle spielen: Vertex- und Voxel-basiert. Hierbei sind auch API-spezifische Überlegungen mit eingeflossen. Man muss abwägen zwischen der konzeptionell saubersten und umfassendsten Lösung und der effizienten Realisierbarkeit mit verfügbaren Hard- und Software-Architekturen; im Zweifelsfalle siegt das Konzept, was die effiziente Realisierbarkeit begünstigt.

Die Basisklasse hat die Methode

```
1 virtual void draw(
2   unsigned int numInstances=1,
3   GeometryRepresentation desiredGeomRep = DEFAULT_GEOMETRY_REPRESENTATION↩
        ) = 0;
```

`numInstances` findet beim Hardware-Instancing Verwendung (s. Abschnitt 4.1.8.3), `desiredGeomRep` bietet die Möglichkeit – sofern kompatibel – für den aktuellen Draw-Call eine andere Repräsentation zu wählen. So lassen sich z.B. gezielt bestimmte Dreiecks-Meshes als Linen oder Punkte zeichnen, ohne dass andere Objekte beeinflusst würden, wie es bei einer globalen Einstellung der Fall wäre (z.B. über `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`).

3.13.1 BufferBasedGeometry

Diese Klasse ist ebenfalls noch abstrakt; Sie ist die Basis für alle Klassen, deren geometrische Repräsentation ein 1:1-Verhältnis zwischen all ihren Attributen aufweist, also keine komplexen Datenstrukturen benötigt. Abgeleitet hiervon sind `VertexBasedGeometry` (siehe Abschnitt 3.13.2) und `VoxelGridGeometry`↩ (nicht implementiert).

`BufferBasedGeometry` hat ein Array von `BufferInterface`-Pointern als Member, welches über den `BufferSemantics`-Aufzählungstyp indiziert werden kann:

```
1 BufferInterface* mAttributeBuffers[↩
    __NUM_VALID_GEOMETRY_ATTRIBUTE_SEMANTICS__];
```

Diese Buffers können über

```
1 virtual void setAttributeBuffer(BufferInterface* buffi);
```

gesetzt werden. Die Position in `mAttributeBuffers` wird von der Semantik von `buffi` bestimmt. Die Methode ist virtuell, da sie von `VertexBasedGeometry`↩

für das Setzen des entsprechenden OpenGL-States überschrieben werden muss.

Zur Information, falls man mit dem System eine Voxel-basierte Simulation plant:
`VoxelGridGeometry` könnte als konkrete Attribute-Buffers entweder 3D-Texturen oder einen generischen Buffer haben. Hier tut sich ein Problem auf: Das Schreiben in 3D-Texturen ist zwar als OpenCL-Extension verfügbar, wird aber zumindest noch nicht von Nvidia-Treibern unterstützt, evtl. noch nicht einmal nativ von der Hardware (hierüber habe ich keine Informationen gefunden). Folglich würde man einen generischen Buffer verwenden wollen. Dieser lässt sich jedoch nicht mehr direkt per OpenGL via Ray-Casting visualisieren. Abhilfe könnte schaffen, dass man eine Textur allokiert, diese jedoch an OpenCL als klassischen Buffer übergibt. Derartige Mechanismen stellt OpenCL jedoch nicht bereit. In CUDA ist es zumindest möglich, einen generischen Buffer als Textur zu binden, darüber, ob es umgekehrt auch geht, habe ich keine Information [42]. 3D-Image Writes sind auch in CUDA noch nicht möglich. Man scheint sich also immer noch mit Work-Arounds behelfen zu müssen, wie 3D-Texturen in 2D-Texturen zu encoden oder pro Frame von einem Buffer in eine 3D-Textur zu kopieren.
Auch wenn das beschriebene Phänomen meine partikelbasierte Fluidsimulation nicht betrifft, finde ich es doch bemerkenswert, dass bei all den Fortschritten, die Hardware und APIs gemacht haben, teils immer noch Beschränkungen wie diese existieren.

3.13.2 VertexBasedGeometry

Dies Klasse abstrahiert das OpenGL Vertex Buffer Object(VBO).

Wie in den Textur- und Buffer-Klassen selbst, wird auch hier die `BufferInfo`↩ -Klasse genutzt, um die relevanten Parameter (Datentyp und dessen Bit-Count, Anzahl Channel-Komponenten, Normalisierungs-Flags, BufferSemantics als Attribute Index, s. Seite 34) der VBO-bezogenen API-Calls zu bestimmen .

Die Klasse hat zusätzlich zu den Attribute Buffers einen Pointer, der optional als Vertex Index Buffer gesetzt werden kann. Dann wird der OpenGL-Draw Call über `glDrawElementsInstanced(..)` getätigt. Zusätzlich kann man optional ein Intervall angeben, welcher Bereich im Index Buffer genutzt werden soll. Dies ist z.B. sinnvoll, wenn mehrere logische Objekte mit verschiedenen Materials in einem Buffer liegen, wie es z.B. bei einer Fluidsimulation der Fall ist, in welcher mehrere Fluide und Rigid Bodies sich ein Attribute Buffer Set teilen. So kann man kontrollieren, was im Buffer gezeich-

[42] Texturen liegen intern so im Speicher, dass eine gute 2D-räumliche Lokalität gewährleistet ist, z.B. über eine Z-Order-Curve (siehe [**?**]). Auf diese Weise werden Cache Misses und benötigte Bandbreite reduziert. Dieses Layout is non-disclosed, entsprechend ist es unwahrscheinlich, dass man in CUDA Texturen als generische Buffer binden kann.

net werden soll und was nicht. In diesem Fall wird `glDrawElementsInstancedBaseVertex(..)` für den OpenGL- Draw Call genutzt.

Ist kein Index Buffer gesetzt, werden die Vertices über `glDrawArraysInstanced(..)` „direkt“ gezeichnet, also wird z.B. angenommen, dass drei hintereinander liegende Vertices ein Dreick bilden.

Um bei Ping Pong- Buffers vor dem Draw Call sicher zu gehen, dass der aktive Buffer gebunden ist, werden in der `draw()`-Routine, falls eine Flag anzeigt, dass Ping Pong Buffers unter den Attribute Buffers sind, diese neu ans VBO gebunden, falls der aktive Buffer seit dem letzten Draw call gewechselt hat.

3.13.3 InstancedGeometry

Dies Klasse stellt eine reine „Dummy-Geometrie“ dar. in Ihrer `draw()`-Routine teilt sie ihrem zugehörigen `InstanceManager` nur mit, dass diese Instanz beim nächsten Instanced Rendering-Draw Call ebenfalls gezeichnet werden will. Für Details sei auf Abschnitt 4.1.8.3 verwiesen.

3.14 Das MPP-Paket

Dieses Paket stellt dem *Flewnit*-System alle Funktionaltät zur Vefügung, die mit GPU-Programmen zu tun hat, außer der OpenGL-Kontext-Erstellung, welche vom `WindowManager` übernommen wird.

3.14.1 MPP und ParallelComputeManager

Das `MPP`, kurz für „Massively Parallel Program“, ist die Basis-Klasse für `Shader` und `CLProgram`.

```
1  class MPP
2  : public SimulationObject [43]
3  {
4    FLEWNIT_BASIC_OBJECT_DECLARATIONS;
5  public:
6    MPP(String name, SimulationDomain sd);
7    virtual ~MPP();
8    virtual void build()=0;
9  protected:
10   virtual void setupTemplateContext(TemplateContextMap& contextMap)=0;
11   virtual void validate()throw(SimulatorException)=0;
12   //for later debugging of the final code of a stage:
13   void writeToDisk(String sourceCode, Path where);
14 };
```

[43] SimulationObject ist schlicht eine Klasse, die Namen und Simulations-Domäne speichert, damit diese nicht überall neu definiert und Getter implementiert werden müssen.

Die `ParallelComputeManager`- Singleton-Klasse erstellt einen OpenCL-Kontext, der für CL/GL-Interoperabilität mit dem (zuvor vom `WindowManager` erstellten) OpenGL-Kontext assoziiert wird.
Dann werden über OpenCL-Queries die Members einer `ParallelComputeDeviceInfo`↩ -Instanz gesetzt, so dass System-global alle relevanten/interessanten Hardware Features bequem verfügbar sind.

Die Klasse besitzt die OpenCL-C++-Objekte, die mit dem Kontext assoziiert sind: Den `cl::Context` selbst, das assoziierte `cl::Device` (also für gewöhnlich die Repräsentation der GPU des Rechners), und die assoziierte `cl`↩ `::CommandQueue`, in die die einzelnen Kernel Launches, Buffer/Image Reads/Writes/Copies, Synchronisations-Punkte etc. eingefügt werden, und stellt diese über Getter-Funktionen dem System zur Verfügung.

Der OpenCL-C++-Wrapper implementiert Error-Handling inklusive dem Werfen von `cl::Exceptions`. Bei der OpenGL-C-API muss man Errors aber nach wie vor explizit abfragen. Dieses - mit zusätzlich formatiertem Konsolen-Output - übernimmt void `checkCLGLErrors()` (checkt auch den letzten CL-Error, einfach zur Vollständigkeit, auch wenn es nicht nötig ist).
Um bei der Entwicklung des Systems sofort auf OpenGL-Errors aufmerksam gemacht zu werden, ist folgendes Makro definiert:

```
1 #ifdef _DEBUG
2   //Macro to permanently check for errors in debug mode
3 # define GUARD(expression) \
4   expression; \
5   ParallelComputeManager::getInstancePtr()->checkCLGLErrors()
6 #else
7 # define GUARD(expression) expression
8 #endif
```

Sämtliche OpenGL-Calls sind irgendwie durch dieses Makro eingeschlossen. Der etwaige Performance-Overhead spielt durch bedingte Kompilierung im Release-Build keine Rolle.

Da auf die Klasse sehr häufig zugegriffen wir, ist das Shortcut-Makro
#define PARA_COMP_MANAGER ParallelComputeManager::getInstancePtr()
für bequemere Refernzierung der Singleton-Instanz im Code definiert.

`ParallelComputeManager` besitzt sämtliche `MPP`-Instanzen, die sich bei ihr über ihren Konstruktor automatisch registrieren, und ist für deren Löschung beim Engine-Reset zuständig.

Alle `BufferInterface`-Instanzen, die sowohl eine OpenCL- als auch eine OpenGL- Repräsentation haben, registrieren sich automatisch bei dieser Instanz, so dass die Akquisition von Shared Buffers für einen bestimmten Kontext ganz einfach geschehen kann:

```
1 void ParallelComputeManager::acquireSharedBuffersForCompute()
2 {
3   //skip if not necessary;
4   if(computeIsInControl()) return;
5   //force to wait for all GL commands to complete
6   barrierGraphics();[44]
7   mLastCLError = mCommandQueue.enqueueAcquireGLObjects(& ↩
        mRegisteredCLGLSharedBuffers);
8   mCLhasAcquiredSharedObjects = true;
9 }
```

void acquireSharedBuffersForGraphics() ist ähnlich implementiert.

Ob Buffer-Operationen mit dem Host-Code synchronisiert werden soll (also ob API-Calls wie cl::CommandQueue::enqueueWriteImage(..) bis zum Beenden der Buffer-Operation nicht zurückkehren sollen) oder nicht, lässt sich global über void setBlockAfterEnqueue(cl_bool val) und cl_bool getBlockAfterEnqueue↩ ()const setzen bzw abfragen (vgl. Listing 9 ff.). Letztendlich hat sich jedoch herausgestellt, dass Blocking bei Write-Operationen nicht nötig ist, weil ich (zumindest bei Kernel Launches) intensiv von cl::Events Gebrauch mache, auf von denen man Listen als Parameter an cl::CommandQueue::enqueueXY()↩ übergeben kann, und dieser Command wird frühstens dann ausgeführt, wenn das alle Commands, die mit der Event-Liste assoziiert sind, vollständig ausgeführt sind. Blocking von Buffer-Operationen ist also nur dann nötig, wenn man nach einem Read-Back sofort im Host-Code die ausgelesenen Werte benötigt. Ich hätte mich beim Design noch mehr auf den Event-Waiting-Mechanismus fokussieren sollen, der zur Zeit vom Bufferinterface↩ noch nicht unterstützt ist. Ich wollte das BufferInterface so schlicht wie möglich halten. Momentan kommt deshalb die Synchronisation von Buffer-Operationen etwas holprig daher; Man kann den Design Flaw umgehen, indem man vor einer Buffer-Operation PARA_COMP_MANAGER->getCommandQueue↩ ().enqueueWaitForEvents(eventVector) aufruft, ein Refactoring ist dennoch in Planung.

3.14.2 Shader und ShaderManager

Die Shader-Klasse wrappt erwartungsgemäß die OpenGL-Shader-Funktionalität, stellt außerdem Convenience Functions zum Setzen von Uniform Variablen bereit, deren Kontrollfluss z.T. von den ShaderFeaturesLocal und den ShaderFeaturesGlobal↩ abhängt. Das MPP-Interfaces ist implementiert, jedoch noch ableitbar. Nur die neue Routine virtual void use(SubObject* so)throw(SimulatorException)=0; ist rein abstrakt.

In wiefern das Shader-Interface und seine Implementatioen geschickt

[44] barrierGraphics() ist schlicht ein Wrapper für GUARD(glFinish()), so wie barrierCompute() als mCommandQueue.enqueueBarrier() implementiert ist.

und generisch gewählt sind oder nicht, konnte leider empirisch noch nicht umfassend ermittelt werden, da bisher der Shader-Template-Code aller abgeleiteten Shader-Klassen auf eine einzige Sammlung von Dateien zurück gehen, nämlich der mit dem suggestiven Namen „GenericLightingUberShader“; diese realisiert sowohl das klassische visuelle Rendering mit den in Abschnitt 4.1.8 vorgestellten Features, als auch die Shadow Map Generation und die Debug-Draw-Funktionalität.
Die Wahre Probe für die `Shader`-Klasse wird jedoch die, wenn eine Datei-Sammlung und mit ihr eine (Multi-Pass)-Shaderstruktur, die nicht viel mit dem „GenericLightingUbershader“ gemeinsam hat, zur Code-Basis wird. Inwiefern die Convenience Functions dann noch sinnvoll sind, (nicht-virtuell) in der Shader-Oberklasse definiert zu sein, vermag ich schwer abzuschätzen. Ob trotz der signifikanten Unterschiede diese Klasse gut abstrahiert ist und so allgemein nutzbar bleibt, wird sich erst dann herausstellen, wenn die `ParticleLiquidDrawStage` (s. Abb. 4) implementiert ist. Aus Zeitmangel konnte ich das Fluid noch nicht angemessen visualisieren, das `LiquidVisualMat`↩ ist momentan noch – entgegen dem Klassendiagramm in Abb. 3 – „hacky“ von `DegugDrawVisualMat` abgeleitet.
In Anbetracht eines solchen nicht unerheblichen Vollständigkeits-Mangels bei der Implementierung sei hier explizit auf Abschnitt 3.15 verwiesen, der den Status der Implementierung zur Zeit der Abgabe der Ausarbeitung skizziert.

Die `ShaderManager`-Singleton-Klasse hat zum Ziel, so viele visuelle Rendering-Techniken und -Effekte in beliebigen (sinnvoller) sinnvollen Permutationen zu ermöglichen. Um nicht hunderte Shader von Hand anlegen zu müssen, die vor Boilerplate-Code überlaufen, wird wie gesagt die *Grantlee*-Template-Engine verwendet, um aus relativ wenigen Dateien angepasste Shader zu generieren.

Sie ist dafür zuständig, immer wenn sich das „Rendering Szenario“ ändert, also eine neue `LightinSimStageBase` im `LightingSimulator` aktiv wird, anhand von diversen Parametern allen `VisualMaterial`s einen zum Material, dem Szenario und den globalen Shader-Features kompatiblen `Shader` zuzuweisen. Dieser wird entweder zur Laufzeit per *Grantlee* generiert, oder wenn schon ein Shader mit benötigten Features existiert, aus einer Hash-Map beschafft, wo der Hash-Wert der Shader-Features als Schlüssel dient. Das Sequenz-Diagramm in Abb. 6 skizziert den Ablauf.

Außerdem lassen sich einige Features abfragen, die ansonsten nur indirekt in den Feature-Structures encoded sind, wie z.B. ob die aktuelle Stage einen Framebuffer braucht oder nicht.

Die Parameter sind:

ShaderFeaturesGlobal Parameter, die global per Config gesetzt wurden und ohne Reset der Engine nicht mehr zu ändern sind, und aufgrund

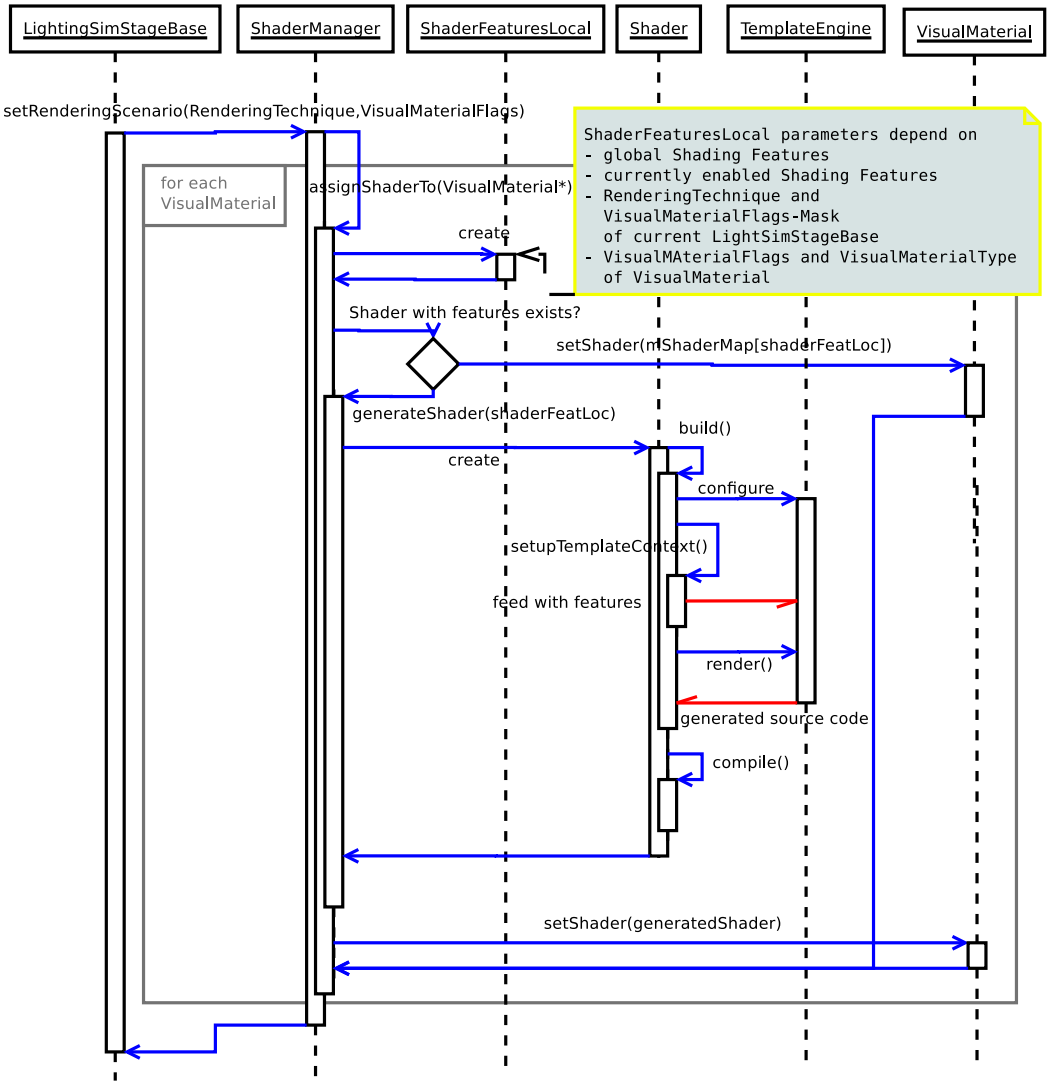

Abbildung 6: Sequenz-Diagramm der Shader-Generierung

von Buffergrößen, Kompatibilitäten und/oder der schlichten Konsitenz des visuellen Renderings sich zwischen den einzelnen Shadern nicht unterscheiden dürfen; dazu zählen:

- die maximale Anzahl an Lichtquellen
- die maximale Anzahl an Lichtquellen, die Schatten werfen (Shadow Caster),
- die maximale Anzahl an Instanzen, die pro Draw Call per Hardware Instancing gezeichnet werden kann
- falls man Deferred Rendering betreibt: Der Textur-Typ des G-Buffers (`Texture2D`, `Texture2DMultisample` etc.),

- falls man Deferred Rendering betreibt: Der MultiSample-Count des G-Buffers, falls er aus MultiSample-Texturen besteht
- das

```
enum LightSourcesLightingFeature
{
  LIGHT_SOURCES_LIGHTING_FEATURE_NONE           =0,
  LIGHT_SOURCES_LIGHTING_FEATURE_ONE_SPOT_LIGHT     =1,
  LIGHT_SOURCES_LIGHTING_FEATURE_ONE_POINT_LIGHT      =2,
  LIGHT_SOURCES_LIGHTING_FEATURE_ALL_POINT_LIGHTS     =3,
  LIGHT_SOURCES_LIGHTING_FEATURE_ALL_SPOT_LIGHTS      =4,
  LIGHT_SOURCES_LIGHTING_FEATURE_ALL_POINT_OR_SPOT_LIGHTS =5,
  __NUM_LIGHT_SOURCES_LIGHTING_FEATURES__         =6
};
```

  Dieser Aufzählungstyp delegiert bei der Shader-Generierung per Template-Engine, welche Typen und welche Anzahl an Lichtquellen vom Shader unterstützt werden sollen.

- das

```
enum LightSourcesShadowFeature
{
  LIGHT_SOURCES_SHADOW_FEATURE_NONE     =0,
  LIGHT_SOURCES_SHADOW_FEATURE_ONE_SPOT_LIGHT =1,
  LIGHT_SOURCES_SHADOW_FEATURE_ONE_POINT_LIGHT  =2,
  LIGHT_SOURCES_SHADOW_FEATURE_ALL_SPOT_LIGHTS  =3,
  __NUM_LIGHT_SOURCES_SHADOW_FEATURES__   =4
};
```

  Es gibt an, ob und wenn ja, welche Art Shadow Mapping vonstatten gehen soll.

- das

```
enum ShadowTechnique
{
  SHADOW_TECHNIQUE_NONE   =0,
  SHADOW_TECHNIQUE_DEFAULT  =1,
  SHADOW_TECHNIQUE_PCFSS    =2,
  __NUM_SHADOW_TECHNIQUES__ =3
};
```

  , welches bestimmt, nach welchem Algorithmus Shadowmapping vonstatten gehen soll

diverse ShadingFeatures Diese Flags geben an, welche Shading-Features „aktiv“ sind: Die `ShadingFeatures` des letztendlich generierten Shaders hängen von den Shading Features des Materials selbst ab, aber auch von denen der aktuell aktiven `LightinSimStageBase` (eine `ShadowMapGenerationStage`↩ hat z.B. gar keine „Shading“-Features) und von den aktuell global

aktivierten Features; Welche Features global aktiv sind, kann vom Benutzer bestimmt werden, hängt aber z.T. auch von der verwendeten OpenGL-Version ab; Tessellation kann z.B. erst ab OpenGL 4.0 verwendet werden.

```
enum ShadingFeatures
  {
  SHADING_FEATURE_NONE        =1<<0,
  SHADING_FEATURE_DIRECT_LIGHTING   =1<<1,
  //global lighting via layered depth images or stuff... just a ↩
      brainstroming, wont be implemented
  SHADING_FEATURE_GLOBAL_LIGHTING   =1<<2,
  SHADING_FEATURE_DIFFUSE_TEXTURING   =1<<3,
  SHADING_FEATURE_DETAIL_TEXTURING  =1<<4,
  SHADING_FEATURE_NORMAL_MAPPING    =1<<5,
  SHADING_FEATURE_CUBE_MAPPING    =1<<6,
  SHADING_FEATURE_AMBIENT_OCCLUSION =1<<7,
  SHADING_FEATURE_TESSELATION     =1<<8,
  __NUM_SHADING_FEATURES__      =9
};
```

VisualMaterialType Der Typ bestimmt für gewöhnlich die Klasse des konkreten Shaders (erinnere: `Shader` ist eine abstrakte Basisklasse):

```
enum VisualMaterialType
{
  VISUAL_MATERIAL_TYPE_NONE         =0,
  VISUAL_MATERIAL_TYPE_DEFAULT_LIGHTING     =1,
  VISUAL_MATERIAL_TYPE_SKYDOME_RENDERING    =2,
  VISUAL_MATERIAL_TYPE_DEBUG_DRAW_ONLY    =3, //just set a color ↩
      value or something
  VISUAL_MATERIAL_TYPE_GAS_RENDERING      =4,
  VISUAL_MATERIAL_TYPE_LIQUID_RENDERING   =5,
  __NUM_VISUAL_MATERIAL_TYPES__       =6
};
```

VisualMaterialFlags Diese Klasse trägt nicht nur zur Delegation der Shader-Generierung bei, sie gibt auch bei einem `VisualMaterial` bestimmte Eigenschaften an, welche auf Kompatibilität mit den Flags der aktuellen `LightingSimStageBase` getestet werden können. Auf diese Weise können inkompatible Objekte masikert werden. Der Konstruktor reicht aus für eine Idee:

```
VisualMaterialFlags(
  bool castsShadows = true,
  bool isTransparent = false,
  bool isShadable = true,
  bool isDynamicCubeMapRenderable = true,
  bool isInstanced=false,
  bool isCustomMaterial=false);
```

3.14.2.1 Probleme

Es sei noch ein nicht so unwichtiges Wort verloren über den Versuch, viele Effekte und Techniken in beliebiger Permutation zu kombinieren: Da dank der Template-Engine nun die Möglichkeit besteht, in Vererbungs-Hierachien zu denken und somit objektorientierte Ansätze ins Shader-Design einfließen zu lassen, zeigt sich umso krasser, wie schwer es ist, Effekte und Techniken sauber und modular zu kombinieren, wenn man obendrein noch maximale Performance haben will.
Viele Features in einem Shader bringen bei Integration neue Einschränkungen mit sich, die besonderer Behandlung bedürfen. Ein Beispiel ist die Kombination von Tessellation mit Shadow Mapping und/oder Layered Rendering: Hier tut sich das Problem auf, dass wir zur Shadowmap-Generierung aus Sicht der Lichtquelle rendern (die View-Matrix ist aus Position und Richtung der *Lichtquelle* konstruiert), wir aber für Level-Of-Detail- Berechnungen zur Bestimmung des Tessellation-Levels Vertex-Positionen im *Betrachter*-Viewspace benötigen! Dies hat den Grund, dass das LOD eines jeden Patches sowohl in der Shadow Map als auch im finalen Rendering gleich sein müssen! Wenn dies nämlich nicht der Fall ist, ist die Geometrie, die dem Shadow-Map-Lookup zugrunde liegt, eine andere als die des finalen Renderings; somit werden sprichwörtlich Äpfel mit Birnen verglichen, und es kann zu Artefakten kommen, z.B. Selbstverschattung, wo keine sein sollte (wenn durch Tessellation z.B. eine Furche generiert wird und die Lichtquelle weiter weg von der Geometrie ist als die Betrachter-Kamera, s. Abb. 7).

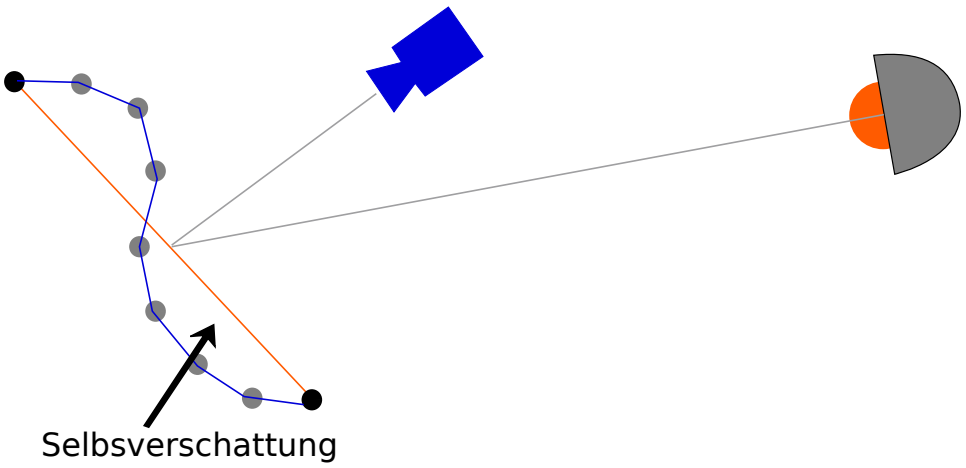

Abbildung 7: Artefakte bei naiver Tessellation-Level-Berechnung mit Camera-Viewspace bei der Shadow Map Generation

Um diese Artefakte zu vermeiden, muss für die LOD-Berechnung im Tesselation Control Shader *immer* die Viewspace-Position der Vertices aus

Sicht der Betrachter-Kamera zur Verfügung stehen. Bei der Shadow-Map-Generierung und/oder bei Layered Rendering (in Cube Maps und/oder Texture Arrays) werden aber für alles weitere, wie Displacement Mapping im Tessellation Evaluation Shader oder Beleuchtung im Fragment Shader, Viewspace-Positionen aus *Lichtquellen*-Sicht bzw *Worldspace*-Positionen durch die einzelnen Shader Stages geschleift.[45] Somit sind die klassischen Interface-Varying-Veriablen schon belegt. Man muss also eine eine neue Variable benutzen, um die Betrachter-Kamera-View Space-transformierte Vertex-Position vom Vertex Shader an den Tessellation Control Shader zu übergeben. Da die built-in-Variable gl_Position für OpenGL erst direkt vor der Fragment Shader-Stage für die Rasterisierung benötigt wird, kann man sie zuvor einfach zum Durchschleifen beliebiger Werte zweckentfremden. Somit ergibt sich folgender Fragment Shader Template-Code:

```
1  // step 2: calculate the gl_Position, if necessary
2  {%if not layeredRendering and not SHADING_FEATURE_TESSELATION %}
3    //default case:
4    //we need projected transform for rasterization because no tessCtrl/↩
         TessEval/geom shader follows the vertex shader;
5    gl_Position = modelViewProjectionMatrix * inVPosition; /*default MVP ↩
         transform*/
6  {% endif %}
7  {%if SHADING_FEATURE_TESSELATION %}
8    {%if RENDERING_TECHNIQUE_SHADOWMAP_GENERATION %}
9      //write the "spectator view space position" to gl_Position for ↩
           tessellation LOD calculations
10     gl_Position = spectatorCamViewMatrix * (modelMatrix * inVPosition);
11   {% else %}
12     {% if worldSpaceTransform %}
13       //write the "own view space position" to gl_Position for ↩
             tessellation LOD calculations;
14       //Hence, even for dynamic cubemap generation, a modelViewMatrix ↩
             must be passed
15       //to the vertex shader for this scenario;
16       //The tessellation LOD calculation is invariant to cam rotation,
17       //hence only the translational part of the view matrix
18       //is relevant, and this part is equal for every six cube map faces;
19       gl_Position = modelViewMatrix * inVPosition;
20     {% endif %}
21   {% endif %}
22 {% endif %}
```

und für den Tesselation Control Shader:

```
1  //In order to determine the tessellation levels,
2  //we need all three vertex positions in spectator view space
3  //of the triangle patch to which the current vertex belongs:
4  {%if RENDERING_TECHNIQUE_SHADOWMAP_GENERATION or worldSpaceTransform %}
5    //every user varyings are either in spectator world space
```

[45] bei Layered Rendering muss man bis zur Geomtry Shader Stage im World Space (nur mit der Model Matrix) rechnen, da im Geometry Shader jedes Layer eine andere Kamera, also eine andere View Matrix hat; Entsprechend kann man keine ModelView Matrix für frühzeitige View Space-Berechnungen akkumulieren!

```
6   //or in the light source- view- or world space;
7   //but we need the VIEW space vertex positions seen from the SPECTATOR ↩
        camera for the tessellation level calculations!
8   //The vertex shader has written this value to gl_Position:
9   vec3 edgeStart = gl_in[ (gl_InvocationID + 1) % 3 ].gl_Position.xyz;
10  vec3 edgeEnd   = gl_in[ (gl_InvocationID + 2) % 3 ].gl_Position.xyz;
11  vec3 ownVert   = gl_in[  gl_InvocationID          ].gl_Position.xyz;
12 {% else %}
13  //no special case, we do just default rendering in spectator- view ↩
        space; So grab the default position value:
14  vec3 edgeStart = input[ (gl_InvocationID + 1) % 3 ].position.xyz;
15  vec3 edgeEnd   = input[ (gl_InvocationID + 2) % 3 ].position.xyz;
16  vec3 ownVert   = input[  gl_InvocationID          ].position.xyz;
17 {% endif %}
```

Das Fazit ist: Modulares Shader Design, der effizienten Code produziert, scheint je nach Feature-Set schwer bis unmöglich. Das Nutzen von Vererbungsmechanismen wird durch die notwendigen Spezial-Behandlungen ebenfalls erschwert. Vor der Integration weiterer Features in diesen Uber Shader muss die Struktur überarbeitet werden. Ich war zur Zeit des Schreibens des Shader-Codes weder mit dem Templating vertraut, noch habe ich alle Spezialfälle antizipieren können. Ich hoffe, dass mit neuen Einsichten und Erfahrungen Les- und Erweiterbarkeit vom Shader Code durch ein Refactoring verbessert können, und dass sich geeignete Wege finden, trotz Problemen wie dem vorgestellten Komplexität durch Vererbung besser zu handhaben.
All diese Probleme betreffen nur den Shader-Code. Insbesondere die Vererbungsmechanismen lassen sich in OpenCL-Code-Templates reibungslos nutzen.

3.14.3 CLProgram und CLProgramManager

`CLProgram` und `CLKernel` wrappen die entsprechende OpenCL-Funktionalität: Das Programm repräsentiert kompletten ausführbaren Code, welches aus einem in OpenCL C geschriebenen Source Code kompiliert wurde. Ein Kernel ist dagegen eine von womöglich mehreren Funktionen innerhalb des Programmms (durch das `__kernel`-Tag gekennzeichnet), welche vom Hostcode per `cl::CommandQueue::enqueueNDRangeKernel()` aufgerufen werden kann. Es besteht also eine $1 : n$-Relation zwischen Programm und Kernel.

Die `CLProgramManager`-Singleton-Klasse gewährleistet einen Zugriff auf sämtliche OpenCL-Programme im System über ihren Namen; Dies kann nützlich sein, wenn ein Kernel auf die Beendigung eines anderen Kernels warten muss, dafür das Event, was mit der letzten Ausführung assoziiert ist, benötigt, aber kein Handle auf die entsprechenen `CLProgram`s vorhanden ist.

`CLProgramManager` besitzt einen `IntermediateResultBuffersManager`, der Buffers bereit stellt, die sich verschiedenen Kernels für ihre Zwischen-Ergebnisse teilen können. Somit wird GPU-Speicher gespart. Jedes `CLProgram`, welches Kernels hat, die Zwischenergebnisse benötigen, kann eine bestimmte Menge und Größe an Buffers requesten. Es ist dann sichergestellt, dass der

Buffer mit entsprechendem Index nach Allokation (geschieht nach Erschaffung sämtlicher `CLProgram`-Instanzen) mindestens die gewünschte Größe hat.

3.14.3.1 CLKernelArguments

Der Umgang mit Kernel-Argumenten, also Parametern, die an die Kernel-Funktionen übergeben werden, wird durch die Klasse `CLKernelArguments` und seine Members deutlich erleichtert: In OpenCL kann man Werte an Kernels nur über ihren Index in der formalen Parameter-Liste übergeben. Diese holprige Methode ist nicht nur im Code wenig aussagekräftig, sie birgt auch die Gefahr, dass wenn Kernel-Signaturen sich ändern, im Host-Code genutzte Indices invalid werden. Wenn diese auch noch verteilt im System auftauchen, wird der Aufwand und die Fehlerwahrscheinlichkeit noch größer, seinen Host-Code anzupassen. Aus diesem Grund erstellt jedes konkrete CLProgram seine eigenen Kernels, und erstellt auch eine Liste an Default-Parametern für jeden Kernel. Auf diese Parameter kann nun bequem sowohl per Index als auch über den Namen des Arguments zugegriffen werden. Typsicherheit ist ebenfalls durch die Template-Klasse `CLValueKernelArgument` gewährleistet, wo bei einem falschen Cast eine Exception geworfen wird. Ferner behebt das `CLBufferKernelArgument` das bereits erwähnte Binding-Problem beim Toggle von PingPong-Buffers.
Ein detaillierteres Klassendiagramm zu den OpenCL-relevanten Klassen, welches zusätzlich die Call-Hierarchie des Build-Prozesses sowie einer Kernel-Invocation skizziert, zeigt Abb. 8.

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

3.15 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

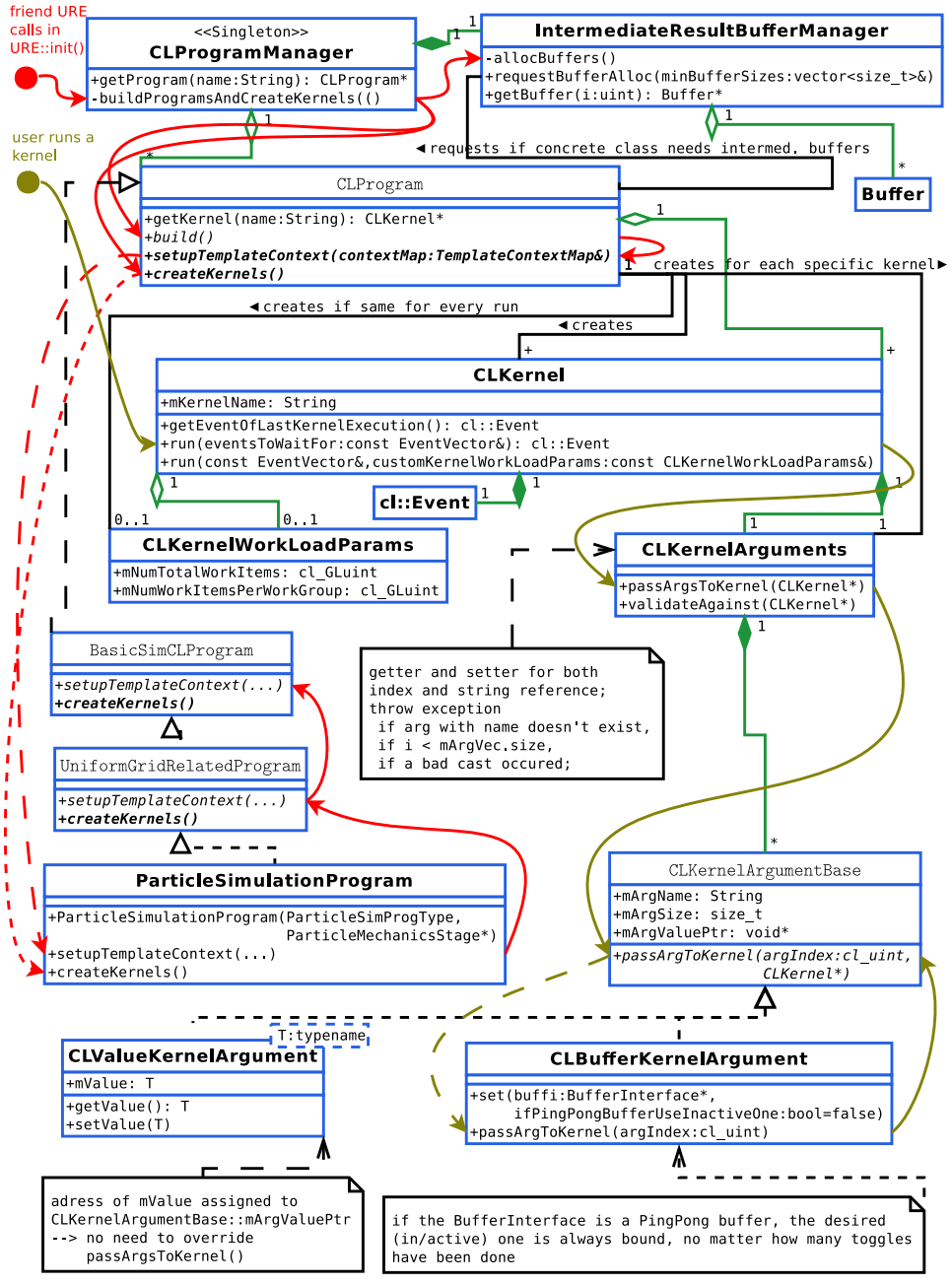

Abbildung 8: Klassendiagramm der OpenCL-relevanten Klassen inkl. Skizze der Call-Hierarchie des Build-Prozesses sowie dem Kernel-Launch

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..

in planung: deferred rendering G-Bufferfill, deferred rendering shade, div. post processing stages

4.1.3 VisualMaterial

4.1.4 Camera

4.1.5 LightSource und LightSourceManager

4.1.6 RenderTarget

4.1.7 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

4.1.8 genutzte moderne OpenGL- Features

4.1.8.1 Hintergrund:Batching

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

referenz zu PCIe-Flaschenhals-stuff

4.1.8.2 Uniform Buffers

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

4.1.8.3 Instancing

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

Listing 4: Transformationsmatrizen-Uniforms im Vertex Shader

```
{% if instancedRendering %}

  struct InstanceTransform
  {
    mat4 modelMatrix;  //needed for layered rendering to be combined with the several lightsource matrices
    mat4 modelViewMatrix; //needed in a non-layered context for calculation of view-space values for lighting calculations
    mat4 modelViewProjectionMatrix; //needed in a non-layered context for gl_Position calculation

    int uniqueInstanceID; //it is not guearnteed that for each "logic" instance, the gl_InstanceID stays the same for every draw call
                          //e.g. because of culling of "previous" instances, the own gl_InstanceID will get smaller

    //no padding, because the offsets will be queried via glGetActiveUniformsiv(...)
  };

  layout(shared) uniform InstanceTransformBuffer
  {
    InstanceTransform instanceTransforms[  {{numMaxInstancesRenderable}} ];
  };
{% else  %}
  uniform mat4 modelMatrix;
  uniform mat4 modelViewMatrix;
  uniform mat4 modelViewProjectionMatrix;
{% endif %}
```

4.1.8.4 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

4.1.9 Ablauf

4.1.10 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

4.2.1.1 Grundlagen

blindtext blindtext blindtext blindtext blindtext blindtext blindtext 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 .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

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

4.2.1.5 Ablauf

initialisierung, und beschreibung der einzelnen phasen...

4.2.1.6 Hardwarespezifische Optimierungen

5 Ergebnisse

6 Fazit

7 Ausblick

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

A Listings zur Buffer-Abstraktion

Listing 5: BufferSemantics

```
1  enum BufferSemantics
2  {
3    // following semantics generally used in
4    //Generic Vertex Attribute Buffers and/or OpenCL (interop) Buffers
5    ///\{
6    POSITION_SEMANTICS,
7    NORMAL_SEMANTICS,
8    TANGENT_SEMANTICS,
9    TEXCOORD_SEMANTICS,
10
11   VELOCITY_SEMANTICS,
12   MASS_SEMANTICS,
13   DENSITY_SEMANTICS,
14   PRESSURE_SEMANTICS,
15   FORCE_SEMANTICS,
16
17   Z_INDEX_SEMANTICS,
18   DIFFUSE_COLOR_SEMANTICS,
19   CUSTOM_SEMANTICS,
20
21   //we need that value to have static-length arrays holding VBO ↩
         maintainance information
22   __NUM_VALID_GEOMETRY_ATTRIBUTE_SEMANTICS__,
23
24   //Semantic of the index buffer from a VBO used via glDrawElements()
25   INDEX_SEMANTICS,
26
27   //for a uniform buffer for matrices of instanced rendering
28   TRANSFORMATION_MATRICES_SEMANTICS,
29   LIGHT_SOURCE_BUFFER_SEMANTICS,
30   ///\}
31
32   //following texture-only semantics; Texture can also have most of the ↩
         above Semantics
33
34   DISPLACEMENT_SEMANTICS, //normal-depth or normal map
35   ENVMAP_SEMANTICS,
36   MATERIAL_ID_SEMANTICS,
37   PRIMITIVE_ID_SEMANTICS,
38   SHADOW_MAP_SEMANTICS,
39   AMBIENT_OCCLUSION_SEMANTICS, //attenuation value as result of AO ↩
         calculations on a depth/position buffer
40   DETAIL_TEXTURE_SEMANTICS, //for stuff like terrain to hide low res ↩
         decal texture
41   NOISE_SEMANTICS,
42   DEPTH_BUFFER_SEMANTICS, //e.g. for memory optimized ambient occlusion ↩
         calculation
43   STENCIL_BUFFER_SEMANTICS,
44   INTERMEDIATE_RENDERING_SEMANTICS,
45   FINAL_RENDERING_SEMANTICS,
```

```
46
47    //actually real amount is one less than this value, but that doesnt ↩
          matter
48    __NUM_TOTAL_SEMANTICS__,
49
50    //indicator for "empty" stuff, e.g. an empty Color Attachment slot in ↩
          an FBO
51    INVALID_SEMANTICS
52  };
```

Listing 6: GLImageFormat-Definition für intern symmetrischeren Umgang mit den Type-/Layout-/Precision-/Normalization- Makros von OpenCL und OpenGL

```
1   struct GLImageFormat
2   {
3     //"internalformat" paramater of glTexImageXD; specifies
4     //  - number of channels
5     //  - number of bits per channel
6     //  - data type of channel values (float, uint, int)
7     //  - if (u)int values shall be normalized;
8     // All of this packed in one enum! Plus, this enum must comply with the↩
            channelOrder
9     // and the channelDataType below! So much invalid combinations are ↩
          possible!
10    // Thus, the values of those enums are determined
11    //valid values: see the big table within the comment of ↩
          BufferElementInfo: GL_RGBA32F .. GL_R8_SNORM;
12    GLint   desiredInternalFormat;
13
14    //"format" paramater of glTexImageXD resp. cl_channel_order:
15    // texelInfo.numChannels == 1 --> GL_RED, 2 --> GL_RG, 4 --> GL_RGBA
16    GLenum channelOrder;
17
18    //"type" parameter of glTexImageXD, i.e cpu data layout;
19    //Compared to OpenCL, the if a (u)int- texel is normalized during ↩
          texture lookup or not is
20    //determined via the "internalformat" param and not together with the ↩
          CPU data type specifier;
21    //Therefore, only the following values are allowed;
22    //  GL_FLOAT     |GL_INT    |GL_UNSIGNED_INT
23    //  GL_HALF_FLOAT |GL_SHORT    |GL_UNSIGNED_SHORT
24    //  ---          |GL_BYTE   |GL_UNSIGNED_BYTE
25    GLenum  channelDataType;
26  };
```

Listing 7: BufferElementInfo

```
1   struct BufferElementInfo
2   {
3     /*
4     In this framework, it is strongly recommended to specify
5     the desired internal layout exactly, i.e. specify
6       1. number of channels: 1,2 or 4;
7          three channel types are not supported
8          by this framework due to alignment reasons;
9       2. internal data format on GPU memory: float,int, unsigned int
10      3. byte size per element: 8,16, or 32
```

```
    4. flag, if  unsigned integer data formats
       should be normalized to float in [0..1]
       resp.if signed integer data formats
       should be normalized to float in [-1..1]

  There is one minor drawback at the moment:
  CPU data must have the same format as the data
  which will reside on the GPU;
  */

  //will be mapped internally to:
  //  the "format" param of glTexImageXD:
  //    GL_RED,   GL_RG,  GL_RGB (forbidden), GL_RGBA resp.
  //  the  cl::ImageFormat.image_channel_order:
  //    CL_R,   CL_RG,  CL_RGB (forbidden), CL_RGBA
  int numChannels;
  GPU_DataType internalGPU_DataType;
  int bitsPerChannel; //8,16 or 32
  bool normalizeIntegralValuesFlag;

  //guard to tell the validate() function that this struct is unused
  //(e.g. for lightsource uniform buffers, generic opencl buffers etc);
  //is set to true in the default constructor,
  //copied in the copy constructor and set to false in the value-passing consturctor
  bool hasNoChanneledElements;

  /*...Constructors, equality and assignment operators omitted...*/

  void validate()const throw (BufferException);
};
```

Listing 8: Mapping von BufferElementInfo-Members zu OpenGL/OpenCL-Makros, exemplarischer Ausschnitt

```
/*...*/

bool normalize =  elementInfo.normalizeIntegralValuesFlag;

switch(elementInfo.internalGPU_DataType)
{
case GPU_DATA_TYPE_UINT :
  switch(elementInfo.bitsPerChannel)
  {
  case 8:
    glImageFormat.channelDataType = GL_UNSIGNED_BYTE;
    if(normalize) clImageFormat.image_channel_data_type = CL_UNORM_INT8;
    else      clImageFormat.image_channel_data_type = CL_UNSIGNED_INT8;
    switch(elementInfo.numChannels)
    {
    case 1:
      elementType = TYPE_UINT8;
      if(normalize) glImageFormat.desiredInternalFormat  = GL_R8;
      else      glImageFormat.desiredInternalFormat  = GL_R8UI;
      break;
    case 2:
      elementType = TYPE_VEC2UI8;
      if(normalize) glImageFormat.desiredInternalFormat  = GL_RG8;
      else      glImageFormat.desiredInternalFormat  = GL_RG8UI;
      break;
```

```
26      case 4:
27        elementType = TYPE_VEC4UI8;
28        if(normalize) glImageFormat.desiredInternalFormat  = GL_RGBA8;
29        else      glImageFormat.desiredInternalFormat  = GL_RGBA8UI;
30        break;
31      default:
32        assert(0&&"should never end here");
33      }
34      break;
35      case 16:
36        glImageFormat.channelDataType = GL_UNSIGNED_SHORT;
37        if(normalize) clImageFormat.image_channel_data_type = ↩
              CL_UNORM_INT16;
38        else      clImageFormat.image_channel_data_type = CL_UNSIGNED_INT16↩
              ;
39
40      switch(elementInfo.numChannels)
41      {
42      case 1:
43        elementType = TYPE_UINT16;
44
45    /*... continuing that way for many many lines...*/
```

Listing 9: API- und Buffertyp-abhängige Operationen auf Buffern – Implementationen durch die Buffer-Klasse

```
1  void Buffer::generateGL()
2  {
3    glGenBuffers(1, &mGraphicsBufferHandle);
4  }
5  void Buffer::generateCL()
6  {
7    mComputeBufferHandle = cl::Buffer(
8        PARA_COMP_MANAGER->getCLContext(),
9        //TODO check performance and interface "set-ability" of ↩
             CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY
10       CL_MEM_READ_WRITE,
11       mBufferInfo->bufferSizeInByte,
12       NULL,
13       PARA_COMP_MANAGER->getLastCLErrorPtr()
14   );
15 }
16 void Buffer::generateCLGL()
17 {
18   mComputeBufferHandle = cl::BufferGL(
19       PARA_COMP_MANAGER->getCLContext(),
20       //TODO check performance and interface "set-ability" of ↩
             CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY
21       CL_MEM_READ_WRITE,
22       mGraphicsBufferHandle,
23       PARA_COMP_MANAGER->getLastCLErrorPtr()
24   );
25   PARA_COMP_MANAGER->registerSharedBuffer(mComputeBufferHandle);
26 }
27 void Buffer::bindGL()
28 {
29   glBindBuffer(mGlBufferTargetEnum, mGraphicsBufferHandle);
30 }
31 void Buffer::allocGL()
32 {
```

```
33    glBufferData(
34      //which target?
35      mGlBufferTargetEnum,
36      // size of storage
37      mBufferInfo->bufferSizeInByte,
38      //data will be passed in setData();
39      NULL,
40      //draw static if not modded, dynamic otherwise ;)
41      mContentsAreModifiedFrequently ? GL_DYNAMIC_DRAW : GL_STATIC_DRAW);
42  }
43  void Buffer::writeGL(const void* data)
44  {
45    glBufferSubData(mGlBufferTargetEnum,0,mBufferInfo->bufferSizeInByte,↩
          data);
46  }
47  void Buffer::writeCL(const void* data)
48  {
49    PARA_COMP_MANAGER->getCommandQueue().enqueueWriteBuffer(
50        static_cast<cl::Buffer&>(mComputeBufferHandle),
51        PARA_COMP_MANAGER->getBlockAfterEnqueue(),
52        0,
53        mBufferInfo->bufferSizeInByte,
54        data,
55        0,
56        PARA_COMP_MANAGER->getLastEventPtr()
57    );
58  }
59  void Buffer::readGL(void* data)
60  {
61    glGetBufferSubData(mGlBufferTargetEnum,0,mBufferInfo->bufferSizeInByte,↩
          data);
62  }
63  void Buffer::readCL(void* data)
64  {
65    PARA_COMP_MANAGER->getCommandQueue().enqueueReadBuffer(
66        static_cast<cl::Buffer&>(mComputeBufferHandle),
67        PARA_COMP_MANAGER->getBlockAfterEnqueue(),
68        0,
69        mBufferInfo->bufferSizeInByte,
70        data,
71        0,
72        PARA_COMP_MANAGER->getLastEventPtr()
73    );
74  }
75  void Buffer::copyGLFrom(GraphicsBufferHandle bufferToCopyContentsFrom)
76  {
77    //bind other buffer as read target
78    GUARD(glBindBuffer(mGlBufferTargetEnum, bufferToCopyContentsFrom););
79    //bind own buffer as write target;
80    GUARD(glBindBuffer(GL_COPY_WRITE_BUFFER, mGraphicsBufferHandle););
81
82    GUARD(glCopyBufferSubData(
83        mGlBufferTargetEnum, //operator==() asserts that buffer types are ↩
              equal, so mGlBufferTargetEnum == rhs.mGlBufferTargetEnum
84        GL_COPY_WRITE_BUFFER,
85        0,
86        0,
87        mBufferInfo->bufferSizeInByte);
88    );
89  }
90  void Buffer::copyCLFrom(ComputeBufferHandle& bufferToCopyContentsFrom)
91  {
```

```
92    PARA_COMP_MANAGER->getCommandQueue().enqueueCopyBuffer(
93        static_cast<const cl::Buffer&>(bufferToCopyContentsFrom),
94        static_cast<const cl::Buffer&>(mComputeBufferHandle),
95        0,
96        0,
97        mBufferInfo->bufferSizeInByte,
98        0,
99        PARA_COMP_MANAGER->getLastEventPtr()
100   );
101 }
102 void Buffer::freeGL()
103 {
104   glDeleteBuffers(1, &mGraphicsBufferHandle);
105 }
```

Listing 10: API- und Buffertyp-abhängige Operationen auf Buffern – teilweise Implementationen für alle Textur-Typen durch die abstrakte Texture-Klasse

```
1  //not pure, as all the same for every texture;
2  void Texture::generateGL()
3  {
4    glGenTextures(1, & mGraphicsBufferHandle);
5  }
6  //the two non-symmetrci GL-only routines:
7  //non-pure, as binding is all the same, only the enum is different, and ↩
       that can be looked up in mTexturInfo;
8  void Texture::bindGL()
9  {
10   glBindTexture(mTextureInfoCastPtr->textureTarget, mGraphicsBufferHandle↩
         );
11 }
12 //can be non-pure, as clEnqueueWriteImage is quite generic;
13 void Texture::writeCL(const void* data)
14 {
15   cl::size_t<3> origin; origin[0]=0;origin[1]=0;origin[2]=0;
16   cl::size_t<3> region;
17   region[0] = mTextureInfoCastPtr->dimensionExtends.x;
18   region[1] = mTextureInfoCastPtr->dimensionExtends.y;
19   region[2] = mTextureInfoCastPtr->dimensionExtends.z;
20   PARA_COMP_MANAGER->getCommandQueue().enqueueWriteImage(
21       static_cast<cl::Image&>(mComputeBufferHandle),
22       PARA_COMP_MANAGER->getBlockAfterEnqueue(),
23       origin,
24       region,
25       0,
26       0,
27       //BAAAD haxx due to a bug in the ocl-c++-bindings: a otherwise ↩
             const-param is non-const here,
28       //although it is const in the wrapped c-function 0_o
29       const_cast<void*>( data ),
30       0,
31       PARA_COMP_MANAGER->getLastEventPtr()
32   );
33 }
34 //non-pure, as glGetTexImage is quite generic :); at least one generic GL↩
       function ;(
35 //BUT:
36 //must be overidden with an exception-throw-implementation for certain ↩
       concrete Texture classes
```

```
37 //(namely multisample textures), which dont seem to be writable, copyable↩
       ar readable
38 //must also be overridden by CubeMap class, as the read call must happen ↩
       six times;
39 void Texture::readGL(void* data)
40 {
41   glGetTexImage(mTextureInfoCastPtr->textureTarget,
42       0,
43       mTextureInfoCastPtr->glImageFormat.channelOrder,
44       mTextureInfoCastPtr->glImageFormat.channelDataType,
45       data
46       );listing
47 }
48 //can be non-pure, as clEnqueueReadImage is quite generic;
49 void Texture::readCL(void* data)
50 {
51   cl::size_t<3> origin; origin[0]=0;origin[1]=0;origin[2]=0;
52   cl::size_t<3> region;
53   region[0] = mTextureInfoCastPtr->dimensionExtends.x;
54   region[1] = mTextureInfoCastPtr->dimensionExtends.y;
55   region[2] = mTextureInfoCastPtr->dimensionExtends.z;
56
57   PARA_COMP_MANAGER->getCommandQueue().enqueueReadImage(
58       static_cast<cl::Image&>(mComputeBufferHandle),
59       PARA_COMP_MANAGER->getBlockAfterEnqueue(),
60       origin,
61       region,
62       0,
63       0,
64       data,
65       0,
66       PARA_COMP_MANAGER->getLastEventPtr()
67   );
68 }
69 //as it seems that a generic copying of many texture types can happen in ↩
       an agnostic way
70 //via two helper FBOs, this can be non-pure;
71 //BUT:
72 //must be overidden with an exception-throw-implementation for certain ↩
       concrete Texture classes
73 //(namely multisample textures), which dont seem to be writable, copyable↩
       or readable;
74 //must also be overriden by CubeMap, as there are six color attachments ↩
       to copy; or should one work with
75 //glFrameBufferTextureFace ? well see in the future ;(
76 void Texture::copyGLFrom(GraphicsBufferHandle bufferToCopyContentsFrom)
77 {
78   //TODO implement when FBO class exists;
79   throw(BufferException("Texture::copyGLFrom: sorry, for copying, FBOs ↩
       are needed, and the still must be implemented "));
80 }
81 //can be non-pure, as clEnqueueCopyImage is quite generic;
82 void Texture::copyCLFrom( ComputeBufferHandle & bufferToCopyContentsFrom)
83 {
84   cl::size_t<3> origin; origin[0]=0;origin[1]=0;origin[2]=0;
85
86   cl::size_t<3> region;
87   region[0] = mTextureInfoCastPtr->dimensionExtends.x;
88   region[1] = mTextureInfoCastPtr->dimensionExtends.y;
89   region[2] = mTextureInfoCastPtr->dimensionExtends.z;
90
91   PARA_COMP_MANAGER->getCommandQueue().enqueueCopyImage(
```

```
92        //source
93        static_cast<const cl::Image&>(bufferToCopyContentsFrom),
94        //destination
95        static_cast<const cl::Image&>(mComputeBufferHandle),
96        origin,
97        origin,
98        region,
99        0,
100       PARA_COMP_MANAGER->getLastEventPtr()
101   );
102 }
103 //can be non-pure, as glDeleteTextures() applies to every texture type :)
104 void Texture::freeGL()
105 {
106   glDeleteTextures(1, & mGraphicsBufferHandle);
107 }
```

Listing 11: API- und Buffertyp-abhängige Operationen auf Buffern – restliche Implementationen durch die Texture2D-Klasse

```
1  void Texture2D::generateCLGL()
2  {
3    mComputeBufferHandle = cl::Image2DGL(
4        PARA_COMP_MANAGER->getCLContext(),
5        CL_MEM_READ_WRITE,
6        mTextureInfoCastPtr->textureTarget,
7        0,
8        mGraphicsBufferHandle,
9        PARA_COMP_MANAGER->getLastCLErrorPtr()
10   );
11 }
12 void Texture2D::allocGL()
13 {
14   glTexImage2D(
15       mTextureInfoCastPtr->textureTarget,
16       0,
17       mTextureInfoCastPtr->glImageFormat.desiredInternalFormat,
18       mTextureInfoCastPtr->dimensionExtends.x,
19       mTextureInfoCastPtr->dimensionExtends.y,
20       0,
21       mTextureInfoCastPtr->glImageFormat.channelOrder,
22       mTextureInfoCastPtr->glImageFormat.channelDataType,
23       //dont set data yet, just alloc mem
24       0
25   );
26
27 }
28 void Texture2D::writeGL(const void* data)
29 {
30   glTexSubImage2D(
31       mTextureInfoCastPtr->textureTarget,
32       0,
33       0,0,
34       mTextureInfoCastPtr->dimensionExtends.x,
35       mTextureInfoCastPtr->dimensionExtends.y,
36       mTextureInfoCastPtr->glImageFormat.channelOrder,
37       mTextureInfoCastPtr->glImageFormat.channelDataType,
38       data
39   );
40   if(mTextureInfoCastPtr->isMipMapped)
```

```
41    {
42      glGenerateMipmap( mTextureInfoCastPtr->textureTarget );
43    }
44  }
```

Literatur

[APS00] Michael Ashikhmin, Simon Premoze, and Peter Shirley. A microfacet-based brdf generator, 2000.

[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.

[Khr09] Khronos Group. *The OpenCL Specification*, version: 1.0, document revision: 48 - 10/6/09 edition, 2009.

[Khr10] Khronos Group. *The OpenGL Graphics System: A Specification*, version 4.1 (core profile) - july 25, 2010 edition, 2010.

[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.