UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Partikelsimulation einer Explosion mit Raytracing unter Beobachtung der Performance

Bachelorarbeit

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

vorgelegt von
Lorenz Wech

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Martin Schumann Institut für Computervisualistik, AG Computergraphik

Koblenz, im April 2012

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)

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Implementation einer Partikelsimulation einer Explosion. Diese soll mit Ray-tracing angezeigt werden. Die Implementation soll mithilfe des OpenCL Standard stattfinden und echtzeitfähig sein. Untersucht wird die Performanz der Kombination dieser Komponenten.

Abstract

This thesis addresses the implementation of a particle simulation of an explosion. The simulation will be displayed via ray tracing in near real time. The implementation makes use of the openCL standard. The focus of research in this thesis is to analyse the performance of this combination of components.

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I
Einleitung

1 Motivation

Die Idee den Prozessor einer Grafikkarte für allgemeine und nicht nur grafische Berechnungen zu verwenden, existiert schon seit mehr als zwanzig Jahren. Die Performanzfähigkeiten der GPU hat viele Forscher und Entwickler fasziniert, die ihre Algorithmen als Grafikberechnungen ausdrückten. Dies war eine sehr monotone Arbeit, die allerdings bereits einige gute Ergebnisse lieferte. So hat die Forschung an General-Purpose Graphic Processor Units (GPGPU) im letzten Jahrzehnt enorme Sprünge gemacht, nicht zuletzt wegen der zunehmenden Verbreitung von GPUs. Diese Entwicklung erhielt einen Boom als am 15. Februar 2007 die Nvidia Corporation CUDA (Compute Unified Device Architecture) veröffentlichte, eine der ersten Programmiersprachen mit denen auch ein Laie direkt auf der GPU programmieren und Code kompilieren konnte. Da CUDA aber nur Grafikkarten von Nvidia Corporation unterstützte, bildete sich bald eine Kollaboration der großen Konzerne um eine allgemeinere Sprache zu definieren: OpenCL war geboren und wurde am 8. Dezember 2008 der Öffentlichkeit vorgestellt.

Meine Motivation war es mit OpenCL ein einfaches Partikelsystem zu programmieren und mithilfe von Raytracing anzeigen zu lassen.

2 Ziel der Arbeit

Ziel dieser Arbeit ist der Aufbau eines Partikelsystems, das verschiedene Verhalten von Partikeln simuliert, darunter auch eine Simulation einer einfachen Explosion. Die Anzeige des Partikelsystems soll über einen Raytracer stattfinden. Beide Systeme sollen zusammen echtzeitfähig sein. Es werden mehrere Tests durchgeführt mit unterschiedlichen Systemeinstellungen. Dabei werden die berechneten Frames per Second gemessen.

3 Stand der Technik

Die OpenCL 1.0 Implementation wurde mit Mac OS X Snow Leopard von Apple veröffentlicht.[kno] Inzwischen hat der Standard am 15. November

Abbildung 1: HighEnd-Beispiel für Partikelsimulation[kne]

2011 die Version 1.2 erreicht und die meisten zeitgenössischen Grafikkarten und Prozessoren von AMD, Intel und Nvidia sind OpenCL-fähig. [khra] Unterstützt wird der Standard in Apple Mac OS, Microsoft Windows und vielen Linux Betriebssystemen.

Teil II

Grundlagen

4 Partikelsimulation

Partikel, das: Teilchen von lateinisch: particulum/particula, Verkleinerungsform von pars = Teil. In der Informatik ist ein Partikel ein Teil eines Partikelsystems. In der Chemie ist ein Partikel die Bezeichnung für ein festes Teilchen eines heterogenen Gemischs wie z.B. Kohlestaub in Rauch oder aufgewirbelter Schlamm unter Wasser.[bri]

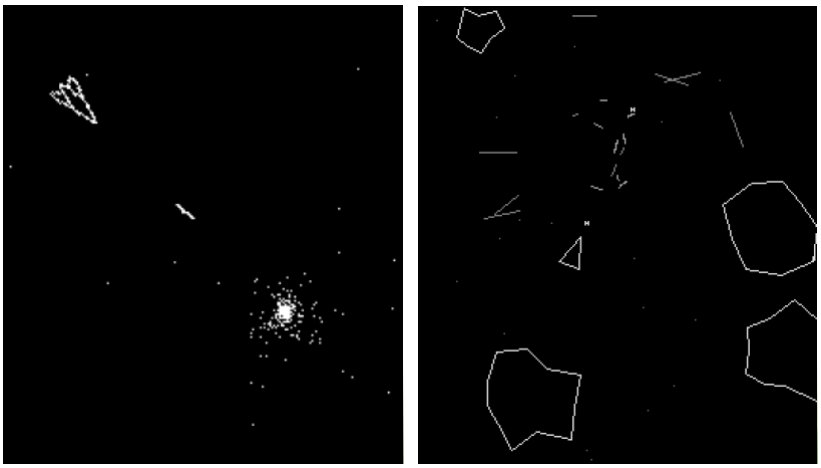
Simulation, die: Eine Computersimulation ist die Darstellung eines mathematischen Models, normalerweise eines in der Realität vorhandenen Systems, in einem Computerprogramm auf einen Datensatz über eine Zeitspanne. Die resultierenden Daten sollen möglichst genau beschreiben, wie sich das reale System verhalten würde, sollte es dieselben Werte wie im Datensatz erhalten.[bri]

"(...)Man unterscheidet zwischen der deterministischen und der stockastischen Simulation. Bei der deterministischen Simulation sind alle von dem Modell beteiligten Größen exakt definiert oder aufgrund mathematischer Zusammenhänge berechenbar. Bei der stockastischen Simulation werden in dem Modell auch zufallsabhängige Größen verwendet, zum Beispiel bei der Monte-Carlo-Methode."[Dam08]

Partikelsimulation, die: nach eigenem Wissen ein Verfahren in dem das Verhalten einer Menge Partikeln über Zeit berechnet wird. Den Partikeln liegen gewisse Eigentschaften inne, wie Ort, Geschwindigkeit oder Masse. Das Verhalten der Partikel soll meist einen physikalischen Vorgang simulieren, wie zum Beispiel Feuer, Rauch, Wasser oder eine Explosion.

4.1 Geschichte

Die Geschichte von Partikelsimulationen ist so alt wie die von Computerspielen. Schon der *Spacewar!*-Prototyp des MIT im Jahre 1961 besaß ein Partikelsystem. Es trat in Aktion, wenn ein Raumschiff gesprengst wurde. Abbildung 2(a) zeigt einen *Spacewar!*-Klon für PC. Ein weiteres Beispiel ist *Asteroids* im Jahrgang 1978 (siehe Abbildung 2(b)), das selbige Asteroiden,



(a) Spacewar!-Klon **(b)** Asteroids!-Klon

Abbildung 2: Frühe Spiele mit Partikelsystemen[gam]

die vom Spieler zerstört werden mussten, verwaltetete. Im Jahre 1983 entwickelte dann William T. Reeves für seine Arbeit *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects* das erste allgemeine Konzept für ein Partikelsystem, das bis heute noch genutzt wird. Eine Anwendung fand es im Film *Star Trek II: The Wrath of Kahn* in dem ein Planet stufenweise mit Flammen, die mit einem Partikelystem simuliert wurden, überzogen wurde, wie Abbildung 3 zeigt.

Heute werden Partikelsysteme in fast jedem Computerspiel und jedem Film verwendet. Entwicklungsumgebungen für Spiele wie UDK oder Unity haben Primitive von Partikelsystem von Anfang an eingebunden. Was als eine Menge von Punkten anfing, geht heute für jede erdenkliche Art von Physiksimulation.

4.2 Aufbau

Ein Partikelsystem nach Reeves[Dam08] besteht aus Partikeln, die sich im 2D- oder 3D-Raum bewegen. Je nach System besitzen sie verschiedene Eigenschaften und Kräfte, auf sie einwirken, wie:

- Ort
- Geschwindigkeit und Richtung
- Farbe

(a) Einschlag **(b)** Flammenwand

Abbildung 3: Star Trek II: The Genesis Effekt[1]

- Transparenz
- Größe
- Form
- Lebenszeit

Die Partikel werden über Emitter mit Initialwerten versehen. Erreicht ein Partikel eine festgelegte Lebenszeit, wird es gelöscht bzw. mit neuen Initialwerten versehen. Die Initialwerte sollten immer zu einem gewissen Teil zufällig erzeugt werden um die Simulation realistischer wirken zu lassen. Das Partikelsystem wird in Zeitschritten aufgefrischt, wie in Abbildung 4 gezeigt wird.
Andere Partikelsysteme, wie zum Beispiel N-Body-Simulationen, nehmen noch weitere Partikeleigenschaften hinzu. Darunter zum Beispiel Masse.

5 Raytracing

Raytracing, das: Technik um realistische Bilder zu rendern. Dabei wird die Physik von Licht als Strahlen zu Hilfe genommen. Es werden Strahlen - für jedes zu renderndes Pixel einer - vom Kamera ausgeschickt, die mit Schnitttests determinieren, welche Objekte getroffen wurden. Weitere Strahlen werden dann versendet für Verschattung, Reflektionen und Refraktionen. Vorteile von Raytracing sind die Simulation

[1] http://www.youtube.com/watch?v=MQFfaCvh2yE

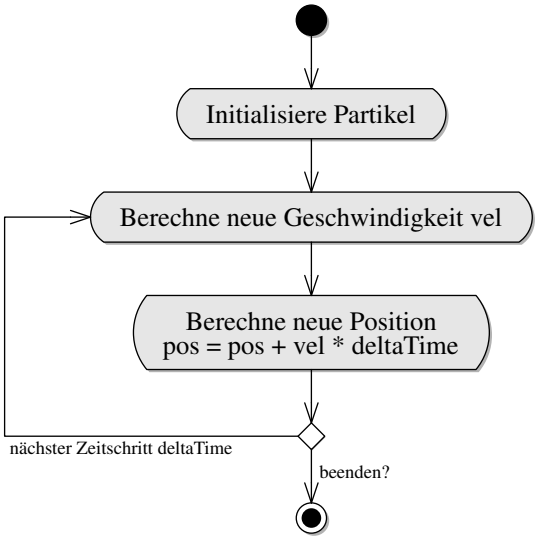


Abbildung 4: Ablaufdiagramm eines Partikelsystems

von natürlichem Licht. Nachteile sind vor allem der mit den Schnitttests verbundenen Rechenaufwand, der durch verschiedene Strukturen verbessert werden kann. Ein Beispiel dafür wären Grids, die den Raum in Blöcke unterteilen. Nun werden zuerst Schnitttests über die Blöcke ausgeführt. Wird ein Block getroffen, werden alle darin enthaltenen Objekte auf Schnittpunkte untersucht.[Bra07][cod]

Raycasting, das: vereinfachtes Raytracing ohne die zusätzlichen Strahlen für Licht, Reflektionen oder Refraktionen.

5.1 Geschichte

Der erste, wirkliche Raytracing Algorithmus wurde 1979 von Turner Whitted entwickelt, der das Raycasting um 3 weitere Rays für Licht, Reflektionen und Refraktionen erweiterte.[Whi79] Eine Weiterentwicklung kam später durch Paul S. Heckbert 1990 der Raytracing bidirektional machte, was auch als Pathtracing bezeichnet wird. [Hec90] Dabei wird der Strahl sowohl vom Beobachter, als auch zum Beobachter verfolgt. Heute wird Raytracing vielfältig verwendet in Werbung, Filmen und inzwischen auch einigen Computerspielen.

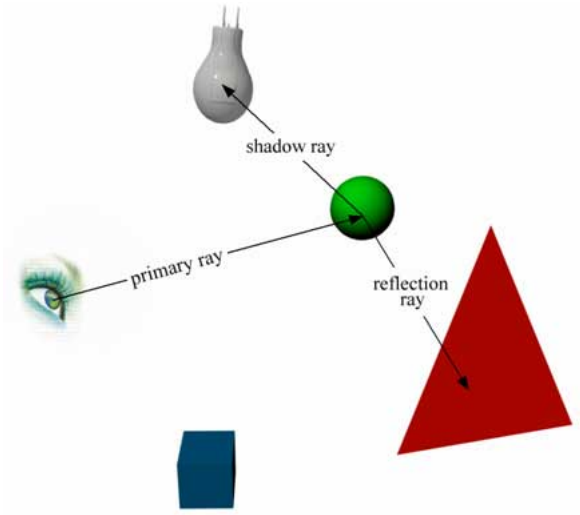


Abbildung 5: Beispielszene eines Raytracers

5.2 Aufbau

Ein Raytracer funktioniert auf einem sehr einfachen Prinzip (siehe Abbildung 5[2]). Für alle Pixel:

1. Ein Hauptstrahl wird von der Beobachterposition verschickt.
2. Schnittpunkttest für getroffene Objekte.
3. Shadow Rays, Reflection Rays und Refraction Rays werden verschickt.
4. Schnittpunkttests für die verschickten Strahlen.

Für die Schnittpunkttests wird zumeist eine Datenstruktur erstellt, die eine bessere Performanz ermöglicht. Beispiele hierfür sind: [Abe09]

- Grids - Die zu rendernde Szene wird in Voxel unterteilt. Es werden nur Objekte in getroffenen Voxel auf Schnittpunkte geprüft.
- Bounding Volume Hierachy - Objekte der Szene werden größeren, schneller zu testenden Objekten eingeschlossen.
- Bounding Interval Hierachy

[2] http://www.pcper.com/reviews/Graphics-Cards/Ray-Tracing-and-Gaming-Quake-4-Ray-Traced-Project

Teil III

OpenCL

OpenCL oder auch Open Computing Language ist der erste offene, gebührenfreie Standard für plattform-übergreifende und parallele Programmierung von modernen Prozessoren die sich in PCs, Servern und mobilen Geräten befinden. Ziel von OpenCL eine große Verbesserung von Performanz und Antwortzeiten von vielen Anwendungen in vielfältigen Kategorien, von Spiel- und Unterhaltungs- bis Wissenschafts- und Medizinsoftware.[khra]

6 Geschichte

Die Entwicklung von OpenCL lässt sich auf Anstöße aus der Softwareindustrie, besonders der Spielebranche zurückführen. Diese forderten immer bessere Rechner mit besserer CPU-Leistung, aber vor allem besserer GPU-Leistung. So entstanden mit der Zeit Grafikchips mit sehr hoher Leistung und schnellem Speicher. Durch die Fokussierung auf die GPU ergab sich aber nur für Applikationen mit Grafikberechnungen, wie 3D-Rendern oder in Spielen ein Vorteil. Andere Programme konnten aber auf die rohe Kraft der GPU nicht zugreifen und so entwickelte sich langsam das Konzept von GPGPU, das vorsah Grafikchips fest als Co-Prozessor auch für andere Berechnungen zu verwenden.[kno]

Was noch fehlte war eine API, mit der man sich der GPU bedienen konnte. NVDIA entwickelte darum die API CUDA, die es ermöglichte Berechnungen auf dem Grafikchip ausführen zu lassen. Da Cuda aber nur auf NVIDIA Grafikchips verfügbar war, machte sich die Forderung nach einer verallgemeinerten API laut. Diese wurde in OpenCL verwirklicht. [nvi]

7 Aufbau[Khrd]

Um den Aufbau von OpenCL verständlich zu beschreiben ist es nötig, ihn in mehrere hierachisch aufgebaute Teilmodelle zu unterteilen.

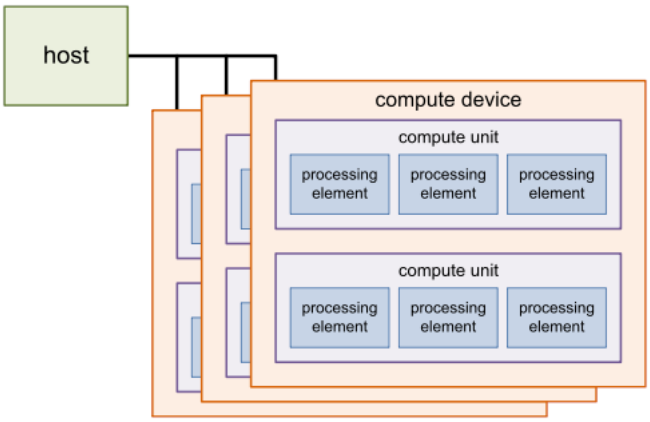


Abbildung 6: The Platform Model[kno]

7.1 Platform Model

Das Platform Model für OpenCL beschreibt die Kommunikation mit der Hardware. Das Modell besteht aus dem Host der mit einem oder mehreren OpenCL Devices verbunden ist, zu sehen in Abbildung 6. Der Host ist hierbei der übergeordnete Prozess, der das OpenCL Programm ausführt und die Verwaltung und Verteilung von Ressourcen an die Devices übernimmt und kontrolliert.

Besonders wichtig ist die Feststellung von:

- Der Platformversion
- Der Version der verschiedenen Devices
- Der unterstützten Version von OpenCL auf den Devices

um die Kompatibilität der OpenCL Anwendung zu garantieren.

OpenCL ist rückwärts kompatibel und versucht standardmäßig die höchst verfügbare Version zu verwenden. Die Grenze hierbei ist die Platformversion. Sie ist die Version die der Host verwendet.

7.2 Execution Model

Die Ausführung eines OpenCL Programms erfolgt in zwei Teilen. Ein Teil sind die Kernels, die auf den Devices ausgeführt werden. Der andere Teil

der Ausführung erfolgt auf dem Host. Er übernimmt die Verwaltung und Ausführung der Kernels und definiert die Systemumgebung - in OpenCL der Context - für sie. Bei der Ausführung eines Kernels wird ein Index Space bestimmt, der von den Devices abhängig ist. Es wird eine Instanz des Kernels an jedem Punkt des Index Space aufgerufen. Diese Instanz ist in OpenCL als Work-item definiert.

Jedes Work-item erhält eine globale ID. Verwaltet werden diese Work-items in Work-groups. Diese erhalten ebenfalls eine globale ID und teilen ihren zugewiesenen Work-items lokale IDs zu. Dieser Prozess dient zur Sequenzierung, denn die Größe der Work-group gibt an, wie viele Work-items gleichzeitig auf einem Device ausgeführt werden können. Der Index Space, in dem sich die Gruppen befinden ist in OpenCL als NDRange definiert, der N-dimensional ist von minimal 1 bis maximal 3.

Dieses Execution Model kann auf eine große Menge von Programmierungsmodellen übertragen werden. Speziell von OpenCL unterstützt sind das Data Parallel Programming Model und das Task Parallel Programming Model.

7.3 Memory Model

Sehr wichtig in OpenCL ist die Verwaltung von Speicherregionen. OpenCL unterteilt diese in 4 Bereiche:

- Globaler Speicher - Vollständiger Schreib- und Lesezugriff von allen Work-items und Work-groups
- Konstanter Speicher - Ein Teil des globalen Speichers, der sich während einer Ausführung eines Kernels nicht ändern kann.
- Lokaler Speicher - Speicher der nur einer bestimmten Work-group und ihren Work-items zugeteilt wird.
- Privater Speicher - Speicher der nur einem bestimmten Work-item zugeteilt wird.

Diese Speicher sind für sich unabhängig, müssen aber zu gewissen Zeiten miteinader interagieren können. Während diesen Zeiten kann der Speicher als unschreibbar und/oder unlesbar markiert werden. Veranschaulicht wird das Modell in Abbildung 7.

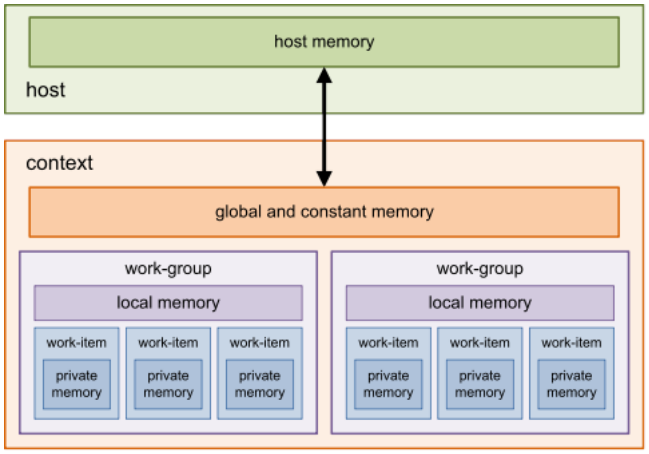


Abbildung 7: The Memory Model[kno]

7.4 Programming Model

Die beiden - von OpenCL unterstützten - Programming Models sind das Data Parallel und Task Parallel Programming Model. Vor allem das Data Parallel Programming Model unterstützt wird.

Das Data Parallel Programming Model definiert sich darüber, das eine Reihe von Befehlen auf mehrere Speicherobjekte angewendet wird. OpenCL verwendet eine weniger strikte Version des Modells, in dem eine Eins-zu-Eins-Zuweisung von Work-item und Speicher nicht benötigt wird.

Das Task Parallel Programming Model lässt mehrere Sets von verschiedenen Befehlen über beliebigen Speicher laufen. Dies kann in OpenCL entweder über den Kernel oder der Aufteilung des Devices in mehrere Teil Devices erreicht werden. Die Aufteilung ist bis jetzt Mehrkern-CPUs vorbehalten und wird über eine Erweiterung geregelt. Mithilfe dieses Models lassen sich auch Anwendungen, die nicht OpenCL geeignet sind, in OpenCL ausführen.

Bei beiden Modellen ist eine gute Kontrolle über den Speicher wichtig. Dazu gibt es mehrere Synchronisationsmöglichkeiten, die später im Framework aufgeführt werden.

8 Framework[Khrd]

Das Framework von OpenCL besteht aus mehreren Teilen:

- Dem OpenCL Platform Layer
- Der OpenCL Runtime
- Dem OpenCL Compiler

Außerdem hat das Framework einen genau festgelegten Methodenaufbau:

- cl_int methodenName(Mischung aus Ein- und Ausgabevariablen)

Der zurückgegebene Integer gibt an, ob die Methode erfolgreich ausgeführt wurde oder ob ein Fehler aufgetreten ist. Dies ist darauf zurückzuführen, dass OpenCL Methoden selbst keine Routinen besitzen, wenn ein Fehler auftritt. Der Fehler wird als int-Wert zurückgegeben und kann innerhalb von OpenCL abgeglichen werden um eine genaue Fehlerkurzbeschreibung zu erhalten. Es ist dem Programmierer selbst überlassen, diesen zu erkennen. Die Eingabevariablen legen die Parameter der Methode fest, während die Ausgabevariablen die Rückgabe der Methode sind. Sofern diese korrekt abgelaufen ist. Die Ausgabevariablen sind auschließlich Zeiger.

Die Ausnahme bilden die clCreateObject()-Methoden, die für die Erstellung von Objekten zuständig sind. Sie geben das erstellte Objekt zurück, anstelle eines Zahlencodes. Für die Fehlererkennung findet sich eine extra Ausgabevariable im Methodenkopf.

Jede OpenCL Applikation wird aus folgenden Komponenten zusammen gesetzt:

- Host: Interagiert mit dem Context durch die OpenCL API.
- Platform: Auf ihr wird die OpenCL Anwendung ausgeführt.
- Device(s): GPUs, CPUs und andere Prozessoren, die OpenCL unterstützen.
- Context(s): Systemumgebung für OpenCL.
- Program(s): Auszuführende OpenCL Funktionen.

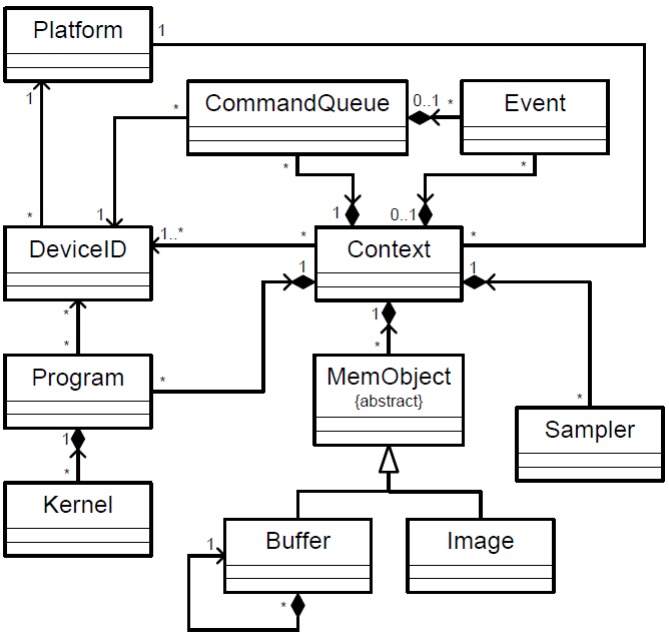


Abbildung 8: OpenCL UML Klassendiagram[Khrd]

- Program Object: Ein mit OpenCL-Compiler kompiliertes Program.
- Command-queue: Befehlswarteschlangen für Kernel und Speicher Objects.
- Kernel: Bestimmte Funktion eines Programs die auf einem Device ausgeführt werden kann.
- Kernel Object: Kapselobjekt für einen Kernel, der Parameterzuweisung zulässt.
- Memory Object: Um Speicherzugriffe zu vereinfachen. Weist auf Region im globalen Speicher.
- Event Object: Beinhaltet die Status einer Anweisung.

Die Assoziationen zwischen den Objekten können der Abbildung 8 entnommen werden.

8.1 OpenCL Objects

Jedes OpenCL Object hat neben seinen speziellen Methoden auch einige andere, die mit allen anderen Objekten übereinstimmen. Object ist im fol-

genden ein Platzhalter für die spezifischen Objekte, wie Contexte, Programs oder Devices:

- cl_object clCreateObject(Objekteigenschaften, Fehlercodes) - Erstellt ein OpenCL Object. Einige Objects haben verschiedene Erstellungsmethoden.
- cl_int clGetObjectInfo(Informationsparameter, Ausgabevariablen) - Gibt Informationen über das Object zurück. Einige Objects haben auch hier mehrere Methoden für verschiedene Arten von Informationen.
- cl_int clRetainObject(cl_object) - Inkrementiert den Reference Count des Objects. Der Reference Count gibt an, ob das Object noch in Verwendung ist. Dies ist hilfreich bei der Arbeit mit verschiedenen Bibliotheken um eine vorzeitige Freigabe zu verhindern. clCreateObject() ruft clRetainObject() einmal auf.
- cl_int ReleaseObject(cl_object) - Dekrementiert den Reference Count des Objects. Ist dieser 0 so wird das Object und all seine untergeordneten Objekte freigegeben zum löschen.

Spezialfälle bilden hier Platforms und Devices. Sie haben keine Create-Methode, da sie Hardware sind, weswegen sie eine GetIDs-Methode besitzen um sie referenzieren zu können. Platform besitzt außerdem weder eine Retain- eine Releasemethode. Device kann diese nur benutzen, wenn es in Subdevices unterteilt wird.

8.2 OpenCL Platform Layer

Diese Schicht ist für die Erkennung und Konfiguration der OpenCL Platform und ihrer OpenCL Devices zuständig. Es werden mit Methoden Informationen über die Devices abgerufen. Diese können dann in einem OpenCL Context verwendet werden.

8.2.1 Platforms

Eine Platform besteht aus dem Host und einem oder mehreren Device(s),die eine Ausführung von OpenCL Applikationen zulassen. Zu Beginn muss man alle möglichen Platformen ermitteln mit denen OpenCL arbeiten kann.

- cl_int clGetPlatformIDs(cl_uint num_entries,
  cl_platform_id *platforms,
  cl_uint *num_platforms)

num_entries gibt die Anzahl an maximal zu findenden Platforms, *platforms* die Referenznummern auf gefundene Platforms und *num_platforms* die Anzahl an verfügbaren Platforms an. Fehler können auftreten durch falsche Parameterangabe oder Speicherknappheit.

Mit der Methode:

- cl_int clGetPlatformInfo(cl_platform_id platform,
  cl_platform_info param_name,
  size_t param_value_size,
  void *param_value,
  size_t *param_value_size_ret)

lassen sich Information über eine Platform abfragen. *platform* ist die abzufragende Platform, *param_name* die abzufragende Information, darunter:

- Die höchste verfügbare OpenCL Version
- Den Namen der Platform
- Den Namen des Herstellers
- und welche Erweiterungen für OpenCL auf dieser Platform verfügbar sind.

Die Rückgabe erfolgt über **param_value*, der die Speichergröße, die mit *param_value_size* angegeben wird, annimmt. **param_value_size_ret* gibt die wirkliche Speichergröße von **param_value* zurück.
Diese Informationen dienen der Auswahl, welche Platform man nutzen möchte.

- cl_int clUnloadPlatformCompiler(cl_platform_id platform)

Hiermit werden alle Ressourcen von *platform* freigeben, analog zu clReleaseObject(...)-Methoden.

8.2.2 Devices

Ein Device ist eine Menge an Compute Units auf die zusammen zugegriffen werden kann. Eine Compute Unit kann eine Work-group abarbeiten. Normalerweise sind dies GPUs, Mehrkern CPUs oder andere Prozessoren. Um eine Liste von verwendbaren Devices zu erhalten, wird

- cl_int clGetDeviceIDs(cl_platform_id platform,
  cl_platform_id platform,
  cl_device_type device_type,
  cl_uint num_entries,
  cl_device_id *devices,
  cl_uint *num_devices)

verwendet. *platform* ist die zu befragende Platform und *num_entries* legt die maximale zurückgebare Anzahl an Devices an. Mit *device_type* kann bereits ein eingrenzendes Kriterium angegeben werden, welche Devices gelistet werden sollen. Darunter:

- CPUs,
- GPUs,
- einige spezielle Devices.

**devices* und **num_devices* geben analog wie clGetPlatformIDs(...) die Devices und die verfügbare Zahl an Devices, die den Kriterien entsprechen. Unter den Fehlercodes sind falsche Parameterangaben, fehlender Speicher oder dass keine Devices gefunden wurden.

Um nun endgültig die Devices auszuwählen die für die OpenCL Anwendung verwendet werden sollen, können mit

- cl_int clGetDeviceInfo(cl_device_id device,
  cl_device_info param_name,
  size_t param_value_size,
  void *param_value,
  size_t param_value_size_ret)

verschiedene Informationen über das, mit *device* angegebene, Device eingeholt werden. Die Methode arbeitet analog wie clGetPlatformInfo(...). Zu den einholbaren Informationen gehören:

- Typ
- Speichergrößen verschiedener Art
- Vorgezogenen Variablengrößen
- Rechenkapazitäten
- Verfügbare Erweiterungen, z.B. double precision 64-bit
- Namen des Devices

Fehlercodes zeigen das Übliche an: falsche Parameterangaben oder fehlender Speicher zur Informationsausgabe.

8.2.3 Contexts

Die Auswahl der verfügbaren Ressourcen ist nun abgeschlossen. Im nächsten Schritt wird die Systemumgebung erstellt in der später die OpenCL Anwendung ausgeführt werden soll. In OpenCL erhält die Systemumgebung den Namen Context. Er verwaltet die Command-queues, Speicherobjekte, Programme und Kernels, die später auf den ausgewählten Devices ausgeführt werden.

- cl_context clCreateContext(const cl_context_properties *properties,
  cl_uint num_devices,
  const cl_device_id *devices,
  void(CL_CALLBACK *pfn_notify)
  (const char *errinfo,
  const void *private_info,
  size_t cb,
  void *user_data),
  void *user_data,
  cl_int *errcode_ret)

**properties* ist eine Liste von Eigenschaften des Contexts, spezifisch welche Platform verwendet wird und ob eine Synchronisation zwischen OpenCL und anderen APIs vom Benutzer vorgenommen werden muss. *num_devices* und **devices* geben Anzahl und IDs der zu verwendenden Devices an. Devices, die mehr als einmal in dieser Liste auftauchen, werden ignoriert. Eine Callback-Funktion *pfn_notify* kann übergeben werden. Diese wird von OpenCL verwendet um Fehlerinformationen des Contexts während dessen

Laufzeit zurückzugeben. **user_data* sind von Benutzer eingebene Information die von *pfn_notify* verwendet werden. **errcode_ret* ist der Fehlercode.

Ein Context kann auch mit clCreateContextFromType(...) erstellt werden. Im Unterschied zu clCreateContext(...) wird in dieser Methode clGetDeviceIDs(...) aufgerufen, weswegen statt einer Liste von Devices nur der Devicetyp angegeben werden muss. Ansonsten sind beide Methode analog zueinander. Wenn man diese Methode verwendet, mag es sich als nützlich erweisen

- cl_int clGetContextInfo(cl_context context,
  cl_context_info param_name,
  size_t param_value_size,
  void *param_value,
  size_t *param_value_size_ret)

aufzurufen um Informationen über den Context zu erhalten. Diese werden in **param_value* zurückgegeben. Darunter:

- Den Reference Count
- Anzahl der inbegriffenen Devices
- Liste der inbegriffenen Devices
- Eigenschaften des Contexts

Der Reference Count kann mit clRetainContext(...) und clReleaseContext(...) verändert werden.

8.3 OpenCL Runtime

Hier werden die Aufrufe der OpenCL API gelistet die auf Klassen, wie Command-queues, Memory Objects, Programs und Kernels zugreifen.

8.3.1 Programs

Programs in OpenCL bestehen aus einer Reihe von Kernels, die als Funktionen fungieren. Sie werden über __kernel identifiziert. Neben den Kernels können auch Hilfsfunktionen und feste Datenangaben in einem Pro-

gram enthalten sein, die von den Kernels benutzt werden können. Programs lassen sich separat mithilfe eines OpenCL C Compilers kompilieren. Programs werden in Program Objects eingebunden. Diese müssen an einen bestimmten Context gebunden sein. Sie besitzen Quelltext in reiner Form oder als Binary. Außerdem besitzen sie den Quelltext in der zuletzt erfolgreich, kompilierten Form. Zusätzlich wissen sie, wie viele Kernel Objects ihren Code nutzen.

Es gibt mehrere Wege Program Objects zu erzeugen:

- cl_program clCreateProgramWithSource(...)

Mit dieser Methode wird der Klartext an das Program Object übergeben. Da er allerdings als ein Feld von Zeigern verwendet, müssen sowohl die gesamte Zeilenzahl des Textes und die spezifische Länge jeder Zeile übertragen werden.

- cl_program clCreateProgramWithBinary(...)

Hier wird statt des Quelltextes die Binary verwendet. Dazu müssen auch alle Devices angegeben werden, die das Program verwenden sollen. Für jedes Device gibt es eine eigene Binary. Diese enthält für das Device spezifischen Code und/oder Anweisungen zur Implementation für das Device. Als Ausgabeparameter gibt es deshalb einen Status für jedes Device. Dieser gibt an, ob das Program erfolgreich auf das Device geladen wurde.

Als letzte Methode gibt es:

- cl_program clCreateProgramWithBuiltInKernels(...)

In dieser Methode wird kein externer Quelltext, sondern sogenannte Built-In Kernels, verwendet. Diese existieren bereits festverdrahtet oder als Firmware auf den Devices. Darum wird eine Liste von Devices gebraucht, die genutzt werden sollen. Eine zweite Liste führt die Namen der Kernels, die mit dem Program verbunden werden sollen.

Ist ein Program Object erfolgreich erstellt worden, muss es noch kompiliert und verlinkt werden, wie Abbildung 9 zeigt. Sonst kann es nicht verwendet werden. Dies gilt nur für Program Objects die mit Binaries oder Klartext erstellt wurden. Built-In Kernels existieren schon in kompilierter und verlinkter Form.

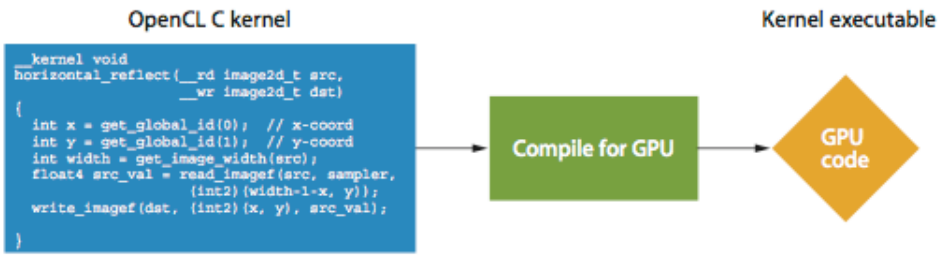


Abbildung 9: Builden eines Programs

- cl_int clBuildProgram(cl_program program,
  cl_uint num_devices,
  const cl_device_id *device_list,
  const char *options,
  const void(CL_CALLBACK *pfn_notify)
  cl_program program,
  void *user_data),
  void *user_data)

Diese Methode kompiliert und verlinkt ein Program Object auf den in **device_list* angegeben Devices. Für spezielle Building Optionen können Eintragungen in **options* vorgenommen werden. Die Routine **pfn_notify* kann - wenn angegeben - Informationen über das Build geben, sobald dieser abgeschlossen ist. Welche Informationen ausgegeben werden sollen, muss in **user_data* spezifiziert werden.

Wenn genauer in den Prozess eingegriffen werden soll, kann man den Vorgang des Buildens in zwei Teile spalten.

- cl_int clCompileProgram(...)

kompiliert ein angegeben Program nur und lässt somit dem Benutzer die Freiheit selbst noch eigene Header dem Programm hinzuzufügen. Optionen für den Compiler sind:

- Zusätzliche Makros
- Behandlung der Präzision von floating points und floating point operations wie sqrt()
- Optimierungsoptionen für schnellere Berechnungen, meist mit Genauigkeitsverlust verbunden.

- Ein- und Auschalten von Warnungen.
- Auswahl, welche OpenCL Version verwendet werden soll.
- Anfragen über Kernel Argumente.

- cl_program clLinkProgram(...)

verlinkt ein Liste von kompilierten Program Objects und gibt diese als ein zusammgefasstes Program Object zurück.

Informationen über Program Objects werden mit

- cl_int clGetProgramInfo(...)
- cl_int clGetProgramBuildInfo(...)

abgefrufen. Abfragbare Informationen über ein Program Object beinhalten:

- Reference Count
- Assoziierter Context
- Assoziierte Devices
- Quelltext oder Binaries
- Namen und Anzahl der verbundenen Kernels

Über den Build kann abgefragt werden, wie der aktuelle Buildstatus ist und welche Optionen beim Builden verwendet wurden. Weiterhin können Informationen über den Binary-Typ und ein Building-Log eingeholt werden.

8.3.2 Command-queue

Command-queues sind Befehlswarteschlangen, die Anweisungen, wie Kernelausführungen oder Speicherzugriffe, abarbeiten und an die spezifizierten Geräte übermitteln. Sie werden erstellt mit

- cl_command_queue clCreateCommandQueue(cl_context,
  cl_device_id device,
  cl_command_queue_properties properties,
  cl_int *errcode_ret)

In *properties* können Einstellungen getroffen werden ob eingereihte Anweisung sequenziell oder nicht sequenziell ausgeführt werden sollen. So können Anweisungen begonnen werden bevor die vorige Anweisung abgeschlossen wurde. Ansonsten lässt sich noch einstellen, ob die Command-queue ihre Anweisungen profilen soll. Profiling gibt zeitliche Informationen über eine Anweisung aus: Wann die Anweisung in die Command-queue eingereiht wurde, wann sie an das Device weitergereicht wurde, wann sie ausgeführt wurde und wann die Ausführung abgeschlossen war.

Informationen über die Command-queue werden mit

- cl_int clGetCommandQueueInfo(...)

abgerufen. Abrufbare Informationen sind sind der assoziierte Context, das Device, der Reference Count und die Eigenschaften der Command-queue. Um eine Command-queue ihre angereihten Anweisungen abarbeiten zu lassen gibt es 2 Methoden:

- cl_int clFlush(...)
- cl_int clFinish(...)

Einziger Unterschied zwischen den beiden Methoden ist, dass clFinish() wartet bis alle Anweisungen ausgeführt wurden. clFlush() garantiert nur, dass alle Anweisungen ausgeführt werden. Deshalb sollte man hier aufpassen und mit Eventwartelisten arbeiten. So können Synchronisierungsfehler vermieden werden.

8.4 Events

Events werden verwendet um den Status von Anweisungen in einer Command-queue abfragen zu können. Ebenso können sie als Synchronisation spunkte dienen. Event Status sind:

- Queued - Anweisung befindet sich in der Warteschlange.
- Submitted - Anweisung wurde an das spezifizierte Device übertragen.
- Running - Anweisung befindet sich in Ausführung.
- Complete - Ausführung der Anweisung wurde abgeschlossen.

Normalerweise werden Events nur von der Command-queue selbst erzeugt, allerdings hat man als Benutzer die Möglichkeit, selbst Events zu erzeugen.

- cl_event clCreateUserEvent(...)

Gibt ein Event zurück. Dessen Status steht auf Submitted. Um den Status eines Events manuell zu ändern wird

- cl_int clSetUserEventStatus(...)

verwendet. Ändert sich der Status, kann mit

- cl_int clSetEventCallback(...)

eine Callback Funktion installiert werden. Diese wird bei einer Änderung zu einem bestimmten Status aufgerufen. Informationen über Events kann, wie bei allen OpenCL Objects mit

- cl_int clGetEventInfo(...)

abgerufen werden. Diese sind: Zu welcher Command-queue und Context das Event gehört, welcher Art die Anweisung ist, welchen Status die Anweisung hat und welchen Reference Count das Events besitzt. Um Events als Synchronisationspunkte zu verwenden kann

- cl_int cl_WaitForEvents(cl_uint num_events,
  const cl_event *event_list)

aufgerufen werden. Der Host muss warten, bis alle gelisteten Events den Status Completed erreicht haben, bevor er weiterarbeiten darf. Ähnliche Methoden sind:

- cl_int clEnqueueMarkerWithWaitList(...)

die einen Marker setzt mit dem alle vorher eingereihten Anweisungen markiert werden. Entweder müssen alle markierten Anweisungen abgeschlossen werden oder alle gelisteten Events den Status Completed erreichen, damit weitere Anweisungen ausgeführt werden können.

- cl_int clEnqueueBarrierWithWaitList(...)

errichtet eine Barriere, die eine Weiterarbeit erst erlaubt, wenn alle vorherigen Anweisungen ausgeführt wurden.

8.5 Memory Objects

OpenCL bietet neben abstrakten Speicherobjekten eine Auswahl zwischen Buffern und Images. Diese teilen sie sich eine Reihe von OpenCL Methoden:

- clRetainMemObject(...)
- clReleaseMemObject(...)
- clGetMemObjectInfo(...)
- clEnqueueUnmapMemObject(...)
- clSetMemObjectDestructorCallback(...)

8.5.1 Buffer

Buffer speichern eine eindimensionale Reihe von Elementen. Diese nehmen bestimmte Datentypen an, wie Integer, float, Vektordaten oder selbstdefinierte Strukturen. Erstellt wird ein Buffer mit:

- cl_mem clCreateBuffer(cl_context context,
  cl_mem_flags flags,
  size_t size,
  void *host_ptr,
  cl_int *errcode_ret)

Dabei wird die Größe des Buffers mit *size* festgelegt. Diese kann nachträglich nicht mehr verändert werden. In den *flags* wird angegeben, welche Schreib-und Leserechte für den Buffer gelten und ob die in **host_ptr* gelisteten Daten in den Buffer eingelesen werden oder nicht. Aus einem Buffer kann mit

- cl_mem clCreateSubBuffer(...)

ein Subbuffer erstellt werden, der auf nur definierte Teile des Buffers zugreifen kann.

Um auf Buffer zugreifen zu können, muss eine Command-queue verwendet werden. In ihr werden Befehle für Schreiben, Lesen, Kopieren, Füllen und Anderes eingereiht. Der allgemeine Aufbau dieser Methoden soll hier an der Lesemethode beschrieben werden.

- cl_int clEnqueueReadBuffer(cl_command_queue command_queue,
  cl_mem buffer,
  cl_bool blocking_read,
  size_t offset,
  size_t size,
  void *ptr,
  cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list,
  cl_event *event)

Es werden die Command-queue und der Buffer angegeben, die verwendet werden sollen. Die Flag *blocking_read* legt fest ob die Anweisung blocking oder non-blocking ist. Gemeint damit ist, ob die Anweisung erst zuende ausgeführt werden muss, bevor weitere Anweisungen bearbeitet werden können oder ob andere Anweisungen bereits angefangen werden können. Diese dürfen nicht auf Speicherregionen zugreifen, die von blocking Anweisungen genutzt werden, da sonst Zugriffsfehler entstehen. *offset* legt fest, an welcher Stelle angefangen werden soll mit schreiben oder lesen. *size* ist Größe der Daten die gelesen/geschrieben werden soll. **ptr* zeigt an die Speicherstelle, die gelesen/beschrieben wird. Als nächstes kommt die Eventwarteliste. Sie legt fest, welche anderen Anweisungen ausgeführt werden müssen, bevor die aktuelle Anweisung ausgeführt werden darf. Sie ist somit ein Synchronisationspunkt für OpenCL. Als letztes Argument kommt **event*, das Event Object, das von der Anweisung selbst erzeugt wird und für eine Eventwarteliste einer anderen Anweisung verwendet werden kann.

- cl_int clEnqueueWriteBuffer(...)
- cl_int clEnqueueReadBufferRect(...)
- cl_int clEnqueueCopyBuffer(...)
- cl_int clEnqueueFillBuffer(...)
- cl_int clEnqueueWriteBufferRect(...)
- cl_int clEnqueueCopyBufferRect(...)

sind weitere Methoden um auf Buffer zuzugreifen. clEnqueueFillBuffer(. . .) schreibt ein generiertes Muster in den Buffer. Die BufferRect-Methoden behandeln den Buffer als eine 2D-, oder 3D-Region in der ein bestimmter Bereich gelesen, geschrieben oder kopiert werden soll.

Ein Spezialfall ist die Methode

- void* clEnqueueMapBuffer(...,
  cl_map_flags map_flags,...
  cl_int *errcoder_ret)

Sie blendet einen Teil des angegeben Buffers in den Host Addressspeicher ein, also den Arbeitsspeicher und gibt einen Zeiger auf die Region zurück. Die Mapflags legen fest, ob die Region zum lesen oder schreiben gedacht ist.

8.5.2 Images

Images werden für 1- bis 3-dimensionale Texturen, Framebuffer oder Bilder verwendet. Die unterstützten Formate können mit

- cl_int clGetSupportedImageFormats(...)

abgerufen werden. Imagetypen sind:

- 1D Image Object
- 1D Image Buffer Object
- 1D Image Array Object

- 2D Image Object
- 2D Image Array Object
- 3D Image Object

Normale Image Objects sind eine Reihe von Elementen, dies gilt auch für das 1D Image Buffer Object. Image Array Objects sind eine eine Reihe von Image Objects. Um ein Image zu erstellen verwendet man

- cl_mem clCreateImage(cl_context context,
  cl_mem_flags flags,
  const cl_image_format *image_format,
  const cl_image_desc *image_desc,
  void *host_ptr,
  cl_int errcode_ret)

Die Flags für Images sind dieselben, wie für Buffer. **image_format* legt die Imageformateigenschaften fest. Formateigenschaften sind, welche Farbkanäle das Image verwendet, wie zum Beispiel RGB, Intensität oder Leuchtdichte. Weiterhin legen die Formateigenschaften fest mit welchem Datentyp die Daten gespeichert sind. Beispiele sind Integer, Shorts oder Floats in verschiedenen Genauigkeitsstufen.

Der Image Deskriptor **image_desc* beschreibt den Aufbau des Image. Dazu gehören:

- Der Image Type
- Höhe, Breite und Tiefe des Bildes
- Anzahl an Bildern, wenn es ein Image Array Object ist
- Beschreibung des Arrays

Image hat dieselben Methoden, wie Buffer für lesen, schreiben, kopieren, füllen und mappen.

- cl_int clEnqueueWriteImage(...)
- cl_int clEnqueueReadImage(...)
- cl_int clEnqueueCopyImage(...)

- cl_int clEnqueueFillImage(...)
- cl_int clEnqueueMapImage(...)

Allerdings besitzen Images neben clGetMemObjectInfo(...) eine eigene GetObject-Methode.

- cl_int clGetImageInfo(...)

Images und Buffer können auch ineinander übertragen werden mit

- cl_int clEnqueueCopyImageToBuffer(...)
- cl_int clEnqueueCopyBufferToImage(...)

8.6 Sampler

Sampler Objects beschreiben, wie Images innerhalb eines Kernels gesampelt werden sollen.

- cl_sampler clCreateSampler(cl_context context,
  cl_bool normalized_coords,
  cl_addressing_mode addressing_mode,
  cl_filter_mode filter_mode,
  cl_int *errcoder_ret)

erstellt einen Sampler. *addressing_mode* gibt an, wie Koordinaten außerhalb des Bildbereichs behandelt werden. *filter_mode* gibt an, ob das Bild mit Manhattan-Abstand oder mit linearer Berechnung gelesen wird.

- cl_int clGetSamplerInfo(...)

gibt Informationen zum Sampler aus.

8.7 Kernels

Ein Kernel ist eine in einem Program deklarierte Methode, der der Bezeichner __kernel vorsteht. Ein Kernel Object kapselt einen spezifischen Kernel, damit ihm Argumente zugewiesen und er ausgeführt werden kann.

- cl_kernel clCreateKernel(cl_program program,
  const char *kernel_name,
  cl_int *ercode_ret)

erstellt ein Kernel Object mit dem Kernel **kernel_name*, der in *program* deklariert wurde. Oder wenn man alle Kernels eines Programs zur Verfügung haben will

- cl_int clCreateKernelsInProgram(cl_program program,
  cl_uint num_kernels,
  cl_kernel *kernels,
  cl_uint *num_kernels_ret)

Um einem Kernel Object nun Argumente zuzuweisen wird

- clSetKernelArg(cl_kernel kernel,
  cl_uint arg_index,
  size_t arg_size,
  const void *arg_value)

verwendet. Mit dieser Methode wird dem Kernel Object *kernel* das *arg_index*-ten Argument der Argumentwert **arg_value* mit der Größe *arg_size* zugewiesen. Um nun den Kernel ausführen zu lassen wird er mit

- cl_int clEnqueueNDRangeKernel(...,
  cl_uint work_dim,
  const size_t *global_work_offset,
  const size_t *global_work_size,
  const size_t *local_work_size,
  ...)

in einer Command-queue eingereiht. Abbildung 10 zeigt die Ausführung eines Kernels. *work_dim* legt eine Dimension von 1 bis maximale Dimension des ausführenden Devices fest. Der *global_work_offset* legt die *work_dim*-Positionen fest, von denen die Ausführung startet. *global_work_size* legt die Anzahl an Kernelaufrufen fest. Diese muss mit *work_dim* korrellieren. *local_work_size* legt die Größe der Work-Groups fest, also wie viele Kernels parallel aufgerufen werden können.

Um einen einzelnen Kernelaufruf zu tätigen wird

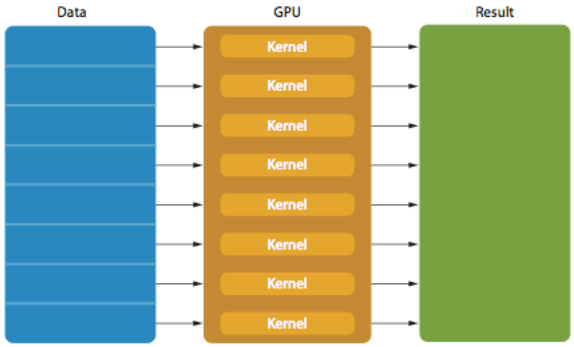


Abbildung 10: Ausführung eines Kernels

- cl_int clEnqueueTask(...)

verwendet. Äquivalent mit clEnqueueNDRangeKernel(...,1,0,1,1,...). Eine spezielle Methode ist

- cl_int clEnqueueNativeKernel(...)

Diese lässt eine C/C++-Methode, die nicht mit dem OpenCL Compiler kompiliert wurde auf dem spezifizierten Device ausführen. Weil sie nicht kompiliert wurde, kann sie auf dafür ausgelegten Devices ausgeführt werden. Dies sind zumeist CPUs. Diese Arbeitsweise sollte allerdings vermieden werden, da sie umständlich ist und wenig Vorteile mit sich bringt. Um gute Parallelität zu erreichen auf jeden Fall häufige Nutzung vermeiden.

8.8 Der OpenCL Compiler

Die letzte Komponente des OpenCL Frameworks ist sein Compiler, der die OpenCL Programmiersprache kompiliert. In diesem Teil wird der Aufbau der Programmiersprache beschrieben.

8.8.1 Unterstützte Datentypen

OpenCL unterstützt mehrer Arten von Datentypen, darunter

- bools - true oder false
- 8-,16-,32- und 64-bit integer, signed und unsigned

- 16- und 32-bit floating-point und falls die Extension verfügbar ist auch 64-bit floating-point.
- size_t ein unsigned 32- oder 64-bit integer, der von sizeof() zurückgegeben wird.
- Zeigerverrechnungsdatentypen
- void

Alle Integer und Floating-points können in n-dimensionaler Vektor-Daten-Form verwendet werden. Andere Built-In Datentypen sind Images, Sampler und Events.

8.8.2 Operatoren

OpenCL verwendet die allgemein üblichen Operatoren, die auch in C verwendet werden:

- Arithmetische: +, -, *, /, ++, –, %
- Relationale: <, <=, >=, >, ==, !=
- Logische: &&, ||
- Bitweise: &, |, ^, ~
- und andere: , , «, », etc.

Für Vector-Daten-Typen sind nur + und - definiert.

8.8.3 Qualifiers

Qualifiers werden verwendet um Daten und Methoden bestimmte Eigenschaften zuzuweisen.

- Für Speicherregionen:(__)global, (__)local, (__)constant, (__)private
- Für Zugriff: (__)read_only, (__)write_only
- Für Funktionen: (__)kernel, __attribute__
- Andere: extern, typedef, static

8.8.4 Built-In Methoden

Weiterhin bietet OpenCL viele Built-In Methoden, die auch wichtig für die Orientierung wärend eines Kernelaufrufs sind.

- uint get_work_dim(): Anzahl an verwendeten Dimensionen.
- size_t get_global_size(uint dimindx): Anzahl an Kernelaufrufen.
- size_t get_global_id(uint dimindx): ID des Aufrufs.
- size_t get_local_size(uint dimindx): Größe der Work-Groups.
- size_t get_local_id(uint dimindx): ID in der Work-Group.
- size_t get_num_groups(uint dimindx): Anzahl an Work-Groups.
- size_t get_group_id(uint dimindx): ID der Work-Group.
- size_t get_global_offset(uint dimindx:): Offset beim Start der Ausführung.

in der dimindx-ten Dimension. Neben diesen wichtigen gibt es viele weitere Methoden verschiedener Art:

- Mathematische
- Integer-spezifische
- Relationale
- Synchronisation
- etc.

9 C++-Wrapper[Khrb]

Die C++-Wrapper API bindet die OpenCL C API in C++ ein und bietet damit einen besseren Klassenaufbau und einige vereinfachte Methodenaufrufe die nun von den deklarierten Objecten selbst aufgerufen werden können. Sie ist allerdings keine komplette Ersetzung der OpenCL C API und ist sehr nahe.

Hier wird der Aufbau einer OpenCL Anwendung kurz im C++-Wrapper dargestellt.

1. Suche verfügbare Platforms mit cl::Platform::get(...).
2. Wähle Platform mit cl::Platform::getInfo(...) aus.
3. Suche verfügbare Devices mit cl::Platform::getDevices(...).
4. Wähle Devices mit cl::Device::getInfo(...).
5. Bilde einen Context aus der bisherigen Auswahl mit cl::Context::Context(...).
6. Erstelle ein OpenCL Program mit cl::Program::Program(...).
7. Builde das Program mit cl::Program::build(...).
8. Erstelle Command-queue(s) mit cl::CommandQueue::CommandQueue(...).
9. Erstelle Buffer und/oder Images und fülle sie mit Daten.
10. Erstelle die ausgewählten Kernels mit cl::Program::Kernel(...).
11. Teile den Kernels die Buffer und constant Daten mit cl::Kernel::setArg(...).
12. Erstelle NDRanges für die Einreihungs-Aufrufe.
13. Lasse die Kernels mit cl::CommandQueue::enqueueNDRangeKernel(...) ausführen.
14. Überwache Events für Speicherkontrolle und Synchronisierung.

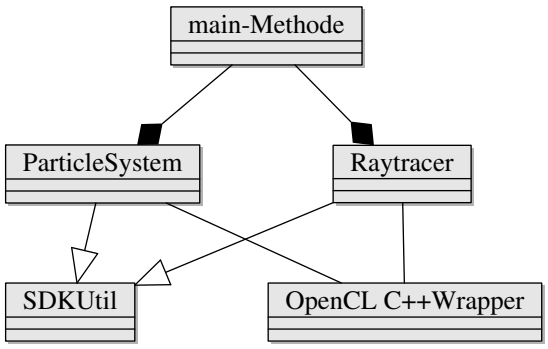


Abbildung 11: Testsystem UML Klassendiagramm

Teil IV

Implementierung

In diesem Segment werden Aufbau und Funktionsweise der Implemtierung erkärt. Das Projekt wurde in Visual Studio 2010 mit C++ und OpenCL programmiert und wird im Verlauf der folgenden Beschreibung als *Particle System* oder als *PS* bezeichnet. Dazu wird ein kurzer Überblick über die in *PS* verwendeten Klassen mithilfe ihrer UML-Klassendiagramme geschaffen. (Siehe Abb. 11, 12 und 13)

10 Aufbau

Die Implementierung besteht aus 3 Teilen:

- Partikelsystem
- Raytracer
- Verwaltungsystem, das OpenGL implementiert

Das Verwaltungssystem besitzt eine Instanz eines Partikelsystems und eines Raytracers und sitzt in der main-Methode. Neben OpenCL wird das Hilfsframework von AMD, das AMD Accelerated Parallel Processing Software Development (APP SDK) verwendet. Das SDK dient als Hilfswerkzeug für:

- Programs einlesen um sie für Programm Objects bereit zu machen.
- Platforms und Devices validieren.
- Informationen über die Anwendung bereitstellen.
- Optionen für CommandLine aufrufe der Anwendung coden.
- Threadklasse für Task Parallel Programme.
- Kann die Fehlercodes von OpenCL in Textform wiedergeben.

Das SDK bevorzugt AMD Produkte, kann aber auch für andere Platformen verwendet werden, was vebesserte Portabilität schafft. Für die grafische Wiedergabe wird OpenGL verwendet, ebenfalls für bessere Portabilität. OpenCL bietet eine Interoperabilität für OpenGL.

Sowohl der Raytracer, als auch das Partikelsystem sind Klassen, die OpenCL über den OpenCL C++-Wrapper verwenden. Beide Klassen sind vom Typ SDKSample, einer in APP SDK definierten Klasse und erben einen Teil ihrer Methoden von dieser Klasse. Nicht alle Methoden werden dabei von *PS* verwendet. Diese sind dann leer oder nur mit einem Rückgabewert definiert. Ursprünglich war angedacht, das Partikelsystem auf der GPU und den Raytracer über die CPU laufen zu lassen. Im aktuellen Stand sind die Umstände vertauscht, mit guten Ergebnissen. Der Aufbau als OpenCL Anwendung ist analog zur Beschreibung im C++ Wrapper.

Die main-Methode initiert zu Anfang ein Objekt der Klasse ParticleSystem und Raytracer. Dann werden werden beide Objekte auf den Stand gebracht um ihren OpenCL-Teil ausführen zu können. Anschließend wird das ParticleSystem im einem Thread parallel zur main-Methode in einer Endlosschleife arbeiten gelassen. Danach werden OpenGL-Einstellungen vorgenommen. In jedem Aufruf der Display-Routine wird vom Raytracer ein aktuelles Bild berechnet und angezeigt.

11 Partikelsystem

In diesem Teil wird das Kernstück des Partikelsystems beschrieben. Die Partikel erhalten zu Anfang:

- eine Position
- eine Geschwindigkeit

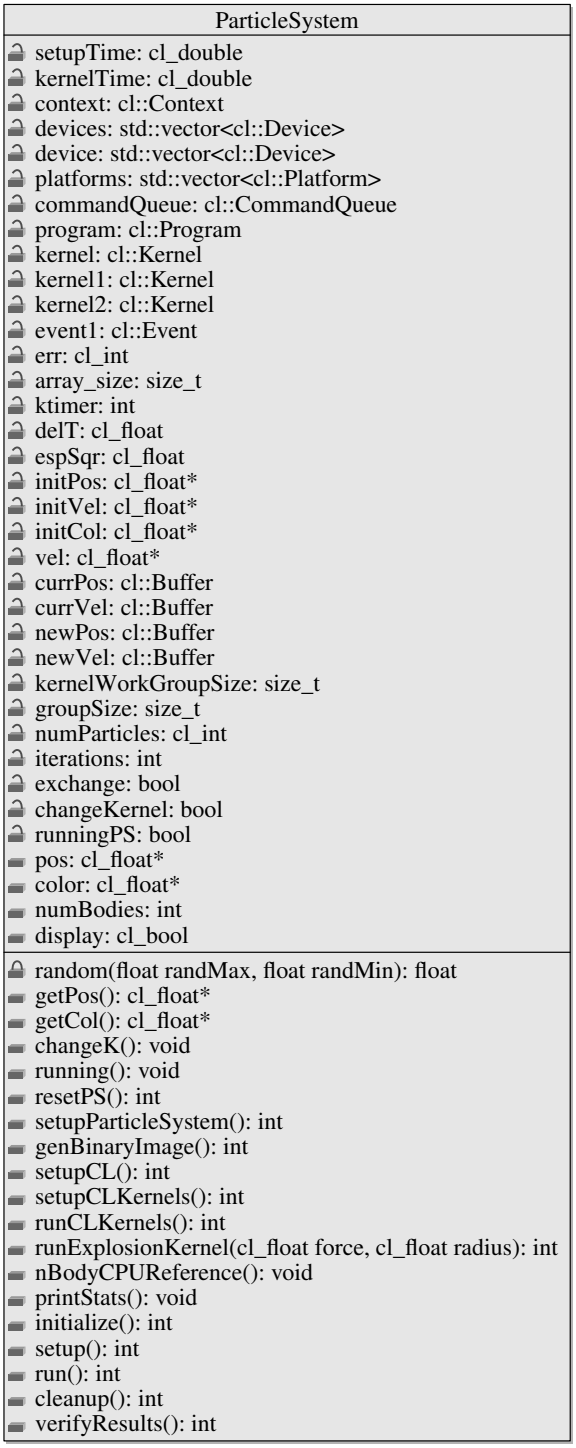


Abbildung 12: Klassendiagramm des Partikelsystems

- eine Farbe
- eine Masse

Die Masse wird als 4.Parameter in der Position eingetragen. Der erste Kernel ist die N-Body-Simulation, entnommen aus dem NBody-Sample des APP SDKs.

Listing 1: N-Body Kernel

```
__kernel
void
particle(
    __global float4* pos ,
    __global float4* vel,
    int numBodies,
    float deltaTime,
    float epsSqr,
    __local float4* localPos,
    __global float4* newPosition,
    __global float4* newVelocity,
        float4 cPos,
        float4 dir)
{
  unsigned int tid = get_local_id(0);
  unsigned int gid = get_global_id(0);
  unsigned int localSize = get_local_size(0);
  unsigned int numTiles = numBodies / localSize;
  float4 myPos = pos[gid];
  float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

  for(int i = 0; i < numTiles; ++i)
  {
    int idx = i * localSize + tid;
    localPos[tid] = pos[idx];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int j = 0; j < localSize; ++j)
    {
      float4 r = localPos[j] - myPos;
      float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
      float invDist = 1.0f / sqrt(distSqr+epsSqr);
      float invDistCube = invDist * invDist * invDist;
      float s = localPos[j].w * invDistCube;
       acc += s * r;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  float4 oldVel = vel[gid];
  oldVel.w = 0.f;
```

```
41
42    float4 newPos = myPos + oldVel * deltaTime +
43      acc * 0.5f * deltaTime * deltaTime;
44    float4 newVel = oldVel + acc * deltaTime;
45
46    float4 dist = myPos - cPos;
47    dist.w = 0.f;
48    float d = dot(dist, dir);
49    if(d > 0)
50    {
51            newVel.w = dot(dist,dist);
52    }
53    else
54    {
55            newVel.w = -1.f;
56    }
57
58    newPos.w = myPos.w;
59    newPosition[gid] = newPos;
60    newVelocity[gid] = newVel;
61  }
```

Argumente für den Kernel sind:

- Aktuelle Position und Geschwindigkeit aller Partikel
- Anzahl Partikel
- Größe des Zeitschritts
- Softening-Faktor
- Zeiger auf allokierten lokalen Speicher
- Neu berechnete Position und Geschwindigkeit
- Position und Blickrichtung des Raytracers

Nach Bestimmung globalen ID-Nummer i des zu bearbeitenden Partikels mit

```
16  unsigned int gid = get_global_id(0);
```

wird von Zeile 22 bis Zeile 37 blockweise über alle Partikel iteriert und die Anziehungskräfte für das bearbeitende Partikel berechnet. Für diesen Vorgang wird der lokal allozierte Speicher verwendet. Um Zugriffsfehler zu vermeiden benötigt Ausführung dieser Bearbeitungstechnik Synchronisierungspunkte, die mit

```
26 barrier(CLK_LOCAL_MEM_FENCE);
```

gesetzt werden. Sie stellen sicher, dass der lokale Speicher erst dann überschrieben werden kann, wenn alle Work-Items die barrier-Methode ausgeführt haben. Für dieses i-te Partikel p_i wird die Beschleunigung acc_i

$$acc_i = \sum_{j=0}^{j=numBodies} (Masse_i * \frac{1}{\sqrt{epsSqr * distance_{ij}}})^3 * distance_{ij}$$

berechnet. Mit dieser wird dann die neue Position pos_i

$$pos_i = pos_i + vel_i * \Delta Time + acc_i * 0.5 * \Delta Time^2$$

und die neue Geschwindigkeit vel_i

$$vel_i = vel_i + acc_i * \Delta Time$$

berechnet und in newPosition und newVelocity an der i-ten Stelle für späteres Auslesen gespeichert. Danach wird noch die Distanz des Partikels zur Blickebene des Raytracers berechnet. Die Blickebene ist die zur Blickrichtung orthogonale Ebene auf der sich die Position des Raytracers befindet.

```
46 float4 dist = myPos − cPos;
47 dist.w = 0.f;
48 float d = dot(dist, dir);
49 if(d > 0)
50 {
51   newVel.w = dot(dist,dist);
52 }
53 else
54 {
55   newVel.w = −1.f;
56 }
```

Ist die Distanz negativ, so befindet sich das Partikel hinter dem Raytracer und muss darum im Raytracer nicht auf Schnittpunkte überprüft werden.

Listing 2: Schwerkraft Kernel

```
1 __kernel
2 void
3 particle2(
4     __global float4* pos ,
5     __global float4* vel,
6     float deltaTime,
7     __global float4* newPosition,
8     __global float4* newVelocity)
```

```
{
  unsigned int gid = get_global_id(0);

  float4 oldPos = pos[gid];
  float4 oldVel = vel[gid];
  oldVel.w = 0.0f;

  oldPos += oldVel * deltaTime;
  oldPos.w = pos[gid].w;
  oldVel.Y -= 98 * deltaTime;

  float4 dist = oldPos - cPos;
  dist.w = 0.f;
  float d = dot(dist, dir);
  if(d > 0)
  {
  oldVel.w = dot(dist,dist);
  }
  else
  {
        oldVel.w = -1.f;
  }
  newPosition[gid] = oldPos;
  newVelocity[gid] = oldVel;
}
```

Der Schwerkraft-Kernel berechnet für jedes Partikel eine aufgefrischte Position bei der ein Schwerkraftsfaktor einbezogen wird. Auch er führt eine Distanzprüfung zur Blickebene durch.

Listing 3: Explosions Kernel

```
__kernel
void
explosion(
        __global float4* pos,
        __global float4* vel,
        float force,
        float radius)
{
  unsigned int gid = get_global_id(0);

  float4 oldVel = vel[gid];
  float4 oldPos = pos[gid];
  float4 exP = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

  exP = oldPos - exP;
  exP.w = 0.0f;
  float distSqr = exP.x * exP.x + exP.y * exP.y + exP.z * exP.z;
```

```
18    distSqr = sqrt(distSqr);
19    if(distSqr <= radius)
20    {
21       exP = exP / distSqr;
22       oldVel = oldVel + exP * force * (distSqr/radius);
23    }
24    vel[gid] = oldVel;
25 }
```

Der Explosions-Kernel simuliert eine Explosion mit einem festgelegten Radius und einer Kraft die auf alle Partikel innerhalb des Radius weg vom Explosionszentrum wirkt. Die Kraft nimmt ab, je weiter das Partikel vom Zentrum entfernt ist.

12 Raytracer

In diesem Teil wird das Kernstück des Raytracers beschrieben. Der Raytracer hat einige Hilfsfunktionen die der Ermittlung der Strahlenrichtung dienen. Hierfür werden Quaternionen zur Hilfe genommmen.

Listing 4: quaternionMult

```
1  float4
2  quaternionMult(
3          float4 q1,
4          float4 q2){
5  float4 r;
6  float3 t;
7
8  float3 v1 = (float3)(q1.x, q1.y, q1.z);
9  float3 v2 = (float3)(q2.x, q2.y, q2.z);
10 float3 c = cross( v1, v2 );
11
12 t = v2 * q1.w + v1 * q2.w + c;
13 r.w = q1.w * q2.w - dot( v1, v2 );
14 r.xyz = t.xyz;
15
16 return r;
17 }
```

quaternionMult multipliziert 2 Quaternionen miteinander. Die mathematische Formel hierfür lautet:

$$q1 \cdot q2 = (w_0, \vec{v_0}) \cdot (w_1, \vec{v_1}) = (w_0 w_1 - \vec{v_0} \circ \vec{v_1}, w_0 \vec{v_1} + w_1 \vec{v_0} + \vec{v_0} \times \vec{v_1})$$

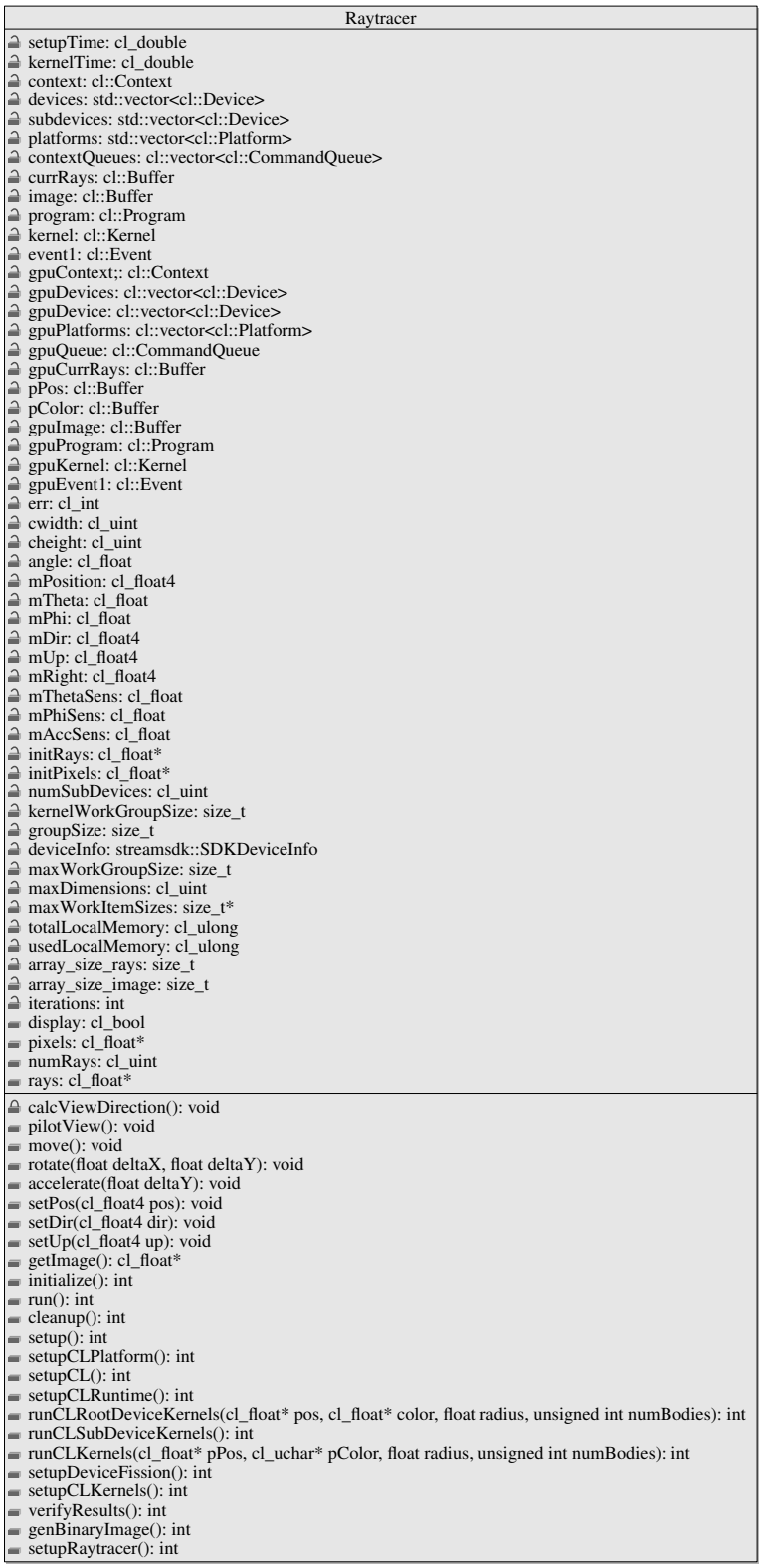


Abbildung 13: Klassendiagramm des Raytracers

Listing 5: quaternionRot

```
18 float4
19 quaternionRot(
20         float4 q,
21         float4 r){
22 float4 rc = r;
23 rc.xyz *= -1;
24 r = quaternionMult(r,q);
25 r = quaternionMult(r,rc);
26 return r;
27 }
```

quaternionRot rotiert einen Vektor $q \in \{\Re^3\}$ um die Achse $\vec{a}$ des Quaterions $r = (w, \vec{a})$. In welche Richtung und um wie viel gedreht wird, legt w fest. Die mathematische Formel hierfür lautet:

$$e = r \cdot q \cdot r^*$$

Bedingungen für korrekte Ausführung dabei sind:

$$r = (\cos(\varphi/2), \sin(\varphi/2) \cdot \vec{v})$$

$$q = (0, \vec{q})$$

$$r^* = (\cos(\varphi/2), -\sin(\varphi/2) \cdot \vec{v})$$

$$e = (0, \vec{e})$$

$\vec{e}$ ist der rotierte Vektor. φ ist der Radiant um den gedreht wird.

Listing 6: computeRay

```
28 float4
29 computeRay(
30         uint x,
31         uint y,
32         uint xSize,
33         uint ySize,
34         float4 dir,
35         float4 up,
36         float4 right,
37         float angle){
38 float xRot = (float) (x - xSize*0.5f) + 0.5f;
39 float yRot = (float) (y - ySize*0.5f) + 0.5f;
40 float axRot = (angle/xSize)*xRot;
41 float ayRot = (angle/ySize)*yRot;
42 float cosxRot;
43 float sinxRot = sincos(axRot, &cosxRot);
44 float cosyRot;
45 float sinyRot = sincos(ayRot, &cosyRot);
```

```
46
47 float4 nup = up;
48 nup.w = cosxRot;
49 nup.xyz *= sinxRot;
50
51 float4 nright = right;
52 nright.w = cosyRot;
53 nright.xyz *= sinyRot;
54
55 float4 ndir = dir;
56 ndir = quaternionRot(ndir, nup);
57 ndir = quaternionRot(ndir, nright);
58 return ndir;
59 }
```

computeRay berechnet den Strahl, mit dem das Work-item rechnen soll. Dafür wird der Blickrichtungsvektor *dir* um einen - über die globalen IDs *x* und *y* - festgelegten Bruchteil des Radianten *angle* jeweils um die Achsen *up* und *right* gedreht. *dir*, *up* und *right* müssen in $\Re^3$ gemeinsam eine orthogonale Basis bilden. Zur schnelleren Berechnung ist gefordert, dass sie alle normiert sind.

Listing 7: raytrace

```
60 __kernel
61 void
62 raytrace(
63     __global float4* pPos,
64     __global float4* pVel,
65     __global float4* pColor,
66     uint numBodies,
67     __global float4* rays,
68     float angle,
69     float radius,
70     float4 cPos,
71     float4 dir,
72     float4 up,
73     float4 right,
74     __local float4* localPos,
75     __local float4* localVel,
76     __global float4* image)
77 {
78     int xPos = get_global_id(0);
79     int yPos = get_global_id(1);
80     int xSize = get_global_size(0);
81     int ySize = get_global_size(1);
82
83     int index = yPos*xSize + xPos;
84     int cidx = -1;
```

```
85
86    float4 ray = computeRay(xPos, yPos, xSize, ySize, dir, up, right, angle);
87    ray.w = 0;
88    rays[index] = ray;
89    for(uint i = 0; i < numBodies; ++i)
90    {
91      if(pVel[i].w == -1.f)
92            {continue;}
93      float4 check = cPos - pPos[i];
94      check.w = 0.0f;
95
96      float b = dot(check, ray);
97      float m = dot(check,check);
98
99      float disk = b*b - m + radius;
100     if(disk >= 0 && range > m)
101     {
102           range = m;
103           cidx = i;
104     }
105   }
106   if(cidx < 0)
107   {
108         image[index] = (float4)(1.0f, 1.0f, 1.0f, 1.0f);
109   }
110   else
111   {
112         image[index] = pColor[cidx];
113   }
114 }
```

Die Argumente für raytrace(...) sind

- Anzahl, Radius zum Quadrat, Position und Farbe aller Partikel
- Welche Partikel überhaupt auf Schnittpunkte getestet werden müssen. (in pVel.w)
- Als Radiant berechneter Öffnungswinkel der Kamera
- Position und Blickrichtung der Kamera
- Up- und Right-Vektoren für die orthogonale Basis der Kamera
- Ausgabebild

Einige Argumente dienten zu Testzwecken:

- Aktuelle Liste der berechneten Strahlen
- Allokierter lokaler Speicher für Positionen und Schnitttestangaben

Der Raytracer funktioniert auf folgende Weise:

1. Ermittle Strahl für Schnittpunkttests mit computeRay(...)
2. Führe einen Kugelschnittpunkttest über alle Partikel aus.[cod]
3. Von den getroffenen Partikeln ermittle das zur Kamera näheste.
4. Trage die Farbe des ermittelten Partikels oder falls keines getroffen wurde weiß in das Ausgabebild ein

Teil V

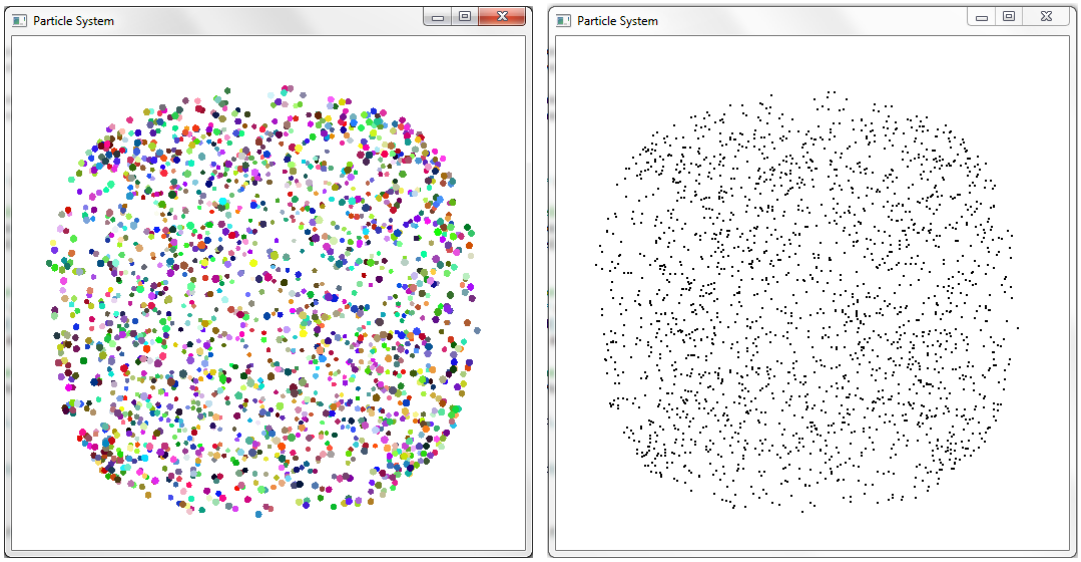
Tests

Die finale Version von *ParticleSystem* bietet zu Anfang mehrere Einstellungsmöglichkeiten:

- Ob CPU oder GPU für die Berechnung von ParticleSystem verwendet werden sollen.
- Work Group Size für ParticleSystem
- Anzahl der Partikel
- Ob CPU oder GPU für die Berechnung des Raytracers verwendet werden sollen.
- Work Group Size für Raytracer
- Öffnungswinkel der Kamera
- Breite des gerenderten Fensters
- Höhe des gerenderten Fensters

Diese Einstellungen sind fest und können erst beim erneuten Starten des Programms neu eingegeben werden. Wichtig für Größe des Fensters und Anzahl der Partikel ist, dass sie ein Vielfaches der Work-group-Größe sind. Ansonsten kommt es zu Argumentenfehlern, da in der letzten Work-Group es ansonsten zu Zugriffsfehlern kommt. Sind diese Einstellungen getätigt, wird OpenGL initialisiert und die Kernels ausgeführt. Die Partikel werden im einem kubischen Bereich in der Mitte der Anzeige gesetzt. Zu Anfang wird das Partikelsystem mit dem Raytracer dargestellt. Nun gibt es mehrere Optionen, die per Tastendruck ausgelöst werden können:

- c: Wechsle zwischen N-Body Kernel und Schwerkraft Kernel für die Partikelverarbeitung.
- t: Pausiere das Partikelsystem, beim nochmaligen Drücken läuft es wieder weiter.
- e: Lasse den Explosions Kernel einmal ausführen.
- r: Stelle das Partikel wieder auf seine Anfangswerte.



(a) mit Raytracer **(b)** mit OpenGL

Abbildung 14: Das Testsystem

- o: Wechsle Raytracer und OpenGL Ansicht, in der alle Partikel als Vertices gerendert werden.
- q oder esc: Beende das Programm.

Ist der Raytracer aktiviert, wird wiederholt die Frames/Sekunde Rate in der Konsole ausgegeben.

13 Ergebnisse

Die Performanz Tests wurden mit auf einem Windows 7 32-bit System folgenden Spezifikationen ausgeführt:

- GPU: AMD Cypress
  - Verfügbare Dimensionen: 3
  - Maximale Work Group Size: 256
  - Verfügbarer Globaler Speicher: 1024 MB
  - Verfügbarer Lokaler Speicher: 32 KB
  - Verfügbare Compute Units: 20

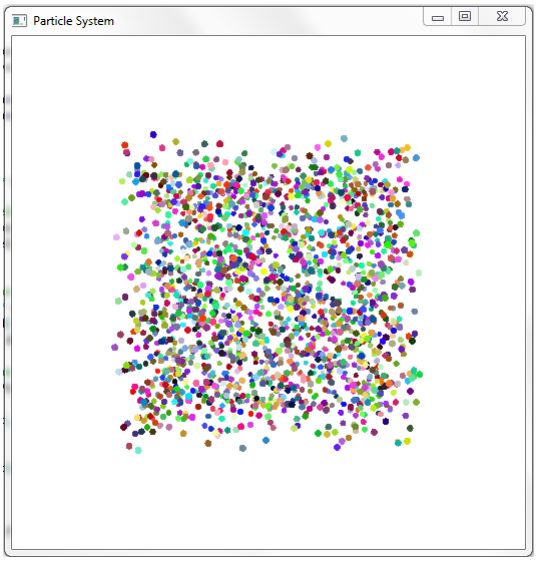


Abbildung 15: Testsystem beim Start

- CPU: Intel Core i7-2600K
  - Verfügbare Dimensionen 3
  - Maximale Work Group Size: 1024
  - Verfügbarer Globaler Speicher: 2048 MB
  - Verfügbarer Lokaler Speicher: 32 KB
  - Verfügbare Compute Units: 8

Die Tests ergaben folgende Frames per Second(Fps)-Messungen:

Particle System	PS GroupSize	Anzahl Partikel	Ray-Tracer	RT GroupSize	Höhe	Breite	~Fps
CPU	256	256	CPU	256	256	256	46.9
CPU	1024	1024	CPU	256	256	256	10.7
CPU	1024	4096	CPU	256	256	256	1.9
CPU	1024	1024	CPU	512	512	512	3.1
CPU	1024	1024	CPU	1024	1024	1024	0.8
CPU	1024	4096	CPU	1024	1024	1024	0.18
CPU	256	256	GPU	256	256	256	88.3
CPU	1024	1024	GPU	256	256	256	30.0
CPU	1024	4096	GPU	256	256	256	5.0
CPU	1024	1024	GPU	256	512	512	18.7
CPU	1024	1024	GPU	256	1024	1024	5.8
CPU	1024	4096	GPU	256	1024	1024	1.8
GPU	256	256	GPU	256	256	256	65.9
GPU	256	1024	GPU	256	256	256	48.1
GPU	256	4096	GPU	256	256	256	24.0
GPU	256	1024	GPU	256	512	512	18.5
GPU	256	1024	GPU	256	1024	1024	5.2
GPU	256	4096	GPU	256	1024	1024	2.4
GPU	256	256	CPU	256	256	256	42.6
GPU	256	1024	CPU	256	256	256	11.8
GPU	256	4096	CPU	256	256	256	3.3
GPU	256	1024	CPU	512	512	512	3.2
GPU	256	1024	CPU	1024	1024	1024	0.8
GPU	256	4096	CPU	1024	1024	1024	0.2

Wie man sehen kann, ist die anfängliche Annahme den Raytracer auf der CPU und die Partikelsimulation auf der GPU zu verwenden nur marginal schneller im Vergleich dazu, wenn das gesamte System auf der CPU ausgeführt wird. Im kleinen Bereich ist es sogar langsamer, aufgrund der Zeit, die CPU und GPU für Kommunikation untereinander brauchen. Man kann also sagen: Solange der Raytracer auf der CPU ausgeführt wird ist es egal, ob die GPU die Berechnung für die Partikelsimulation übernimmt oder nicht.

Die besten Fps-Messungen gab es, wenn der Raytracer über die GPU ausgeführt wurde. Gut zu beobachten ist der Unterschied von GPU zu CPU im Abfall der Fps bei Erhöhung der Partikelanzahl. So fällt bei Verwendung der CPU die Fps-Zahl bei 4-facher Partikelanzahl von 88.3 um 66% auf 30 Fps. Ebenso von 30 auf 5 Fps bei nochmaliger Erhöhung der Partikelanzahl.

Bei Verwendung der GPU fällt die Fps-Zahl von 65.9 auf 48.1 um 27% und bei der nächsten Vervierfachung von 48.1 um 51% auf 24 Fps. In den späteren Tests - mit höheren Einstellungen für den Raytracer - näherten sich die Zahlen stark aneinander an, meist nur 0.2 - 0.6 Fps Unterschied.

Einige weitere Fakten wurden in Vortests beim Programmieren des Testsystems festgestellt:

- Eine blockweise Durchführung der Schnittpunkttests im lokalen Speicher des Raytracers stellte sich als langsamer heraus, als der direkte Zugriff auf das globale Argument. Kernelaufrufe, die schneller fertig waren, mussten immer wieder auf andere Aufrufe warten bis diese auch den aktuell lokal gespeicherten Bereich abgearbeitet hatten.
- Ebenso stellte eine Anwendung von einfachen Sortieralgorithmen wie Bubblesort für den Abstand der Partikel zur Kameraposition als langsamer heraus. Grund dafür ist die Ausführweise von OpenCL von Kernels in Work-groups. Diese macht erst einer neuen Work-group Platz, wenn jeder ihrer Kernelaufrufe abgearbeitet wurde. Somit ist jede Work-group nur so schnell, wie ihr langsamster Kernelaufruf. Man kann davon ausgehen, dass in jeder Work-group sich mindestens ein Strahl befindet, der kein Partikel trifft. Dieser testet dann alle Partikel auf Schnittpunkte und braucht somit die maximale Ausführzeit und somit auch seine zugehörige Work-group. Um die Ausführgeschwindigkeit einer Work-group zu verbessern, muss also die Zahl der Schnittpunkttests im Allgemeinen verringert werden. Dafür würden sich Strukturen wie *Bounding Volume Hierachy* oder Ähnliches anbieten.
- Die Berechnung eines Strahls ist genauso schnell wie das Abrufen eines zuvor abgespeicherten Strahls.
- Der Versuch, die CPU in Subdevices zu unterteilen um jeden Kern einzeln ansprechen zu können, stellte sich als unrentabel heraus. OpenCL ist im Bereich des Data Parallel Programming Models, das im System verwendet wird, selbst fähig genug die Kernelaufrufe effizient zu verteilen.

Teil VI

Fazit

Die für die Bacherlorarbeit entwickelte Testumgebung *ParticleSystem* entspricht den zu Beginn gesetzten Anforderungen. Es beinhaltet:

- Einen Raytracer, der Partikel mit Kugelschnittpunkttests prüft.
- Ein Partikelsystem, das eine vereinfachte Explosion simulieren kann.

Das entwickelte System läuft mit mittleren Einstellungen mit 20 - 30 berechneten Frames pro Sekunde, was für Echtzeit ausreichend, aber kaum akzeptabel im Vergleich zu anderen professionel entwickelten Systemen ist. Es besteht auf jeden Fall noch ein großes Ausbaupotenzial. So könnten für die Partikel, die sich zur Zeit noch auf sehr einfachen Regeln basierend fort bewegen, komplexere Algorithmen verwendet werden um sie noch realistischer wirken zu lassen. Beispiele dafür wären eine Kollisionserkennung, Farbveränderung über Zeit oder Fluidsimulation. Weitere Möglichkeiten wären eine Benutzeroberfläche um die Werte während des Ausführens der Anwendung direkt manipulieren zu können.
Auch eine optimierte Datenstruktur zum organisierten Speichern der Partikel könnte implementiert werden um die Zahl der Schnittpunkttests für den Raytracer zu minimieren.

Für den Raytracer selbst könnten noch Möglichkeiten zur Veränderung der Kameraeinstellungen implementiert werden. Da im Kernel erst der endgültige Strahl aus der Blickrichtung der Kamera berechnet wird, sollte sich dies leicht realisieren lassen. Weitere Ideen wäre ein Ausbau des Raytracers mit Strahlen für Verschattung, Licht, Reflektionen oder Refraktionen.

Abschließend lässt sich sagen, dass OpenCL ein sehr interessanter Standard für GPGPU-Programmierung ist, dessen Weiterentwicklung mit Spannung verfolgt werden kann.

Teil VII

Quellenverzeichnis

Literatur

[Abe09] Oliver Abert. *Augenblick - Ein effizientes Framework für Echtzeit Ray Tracing*. PhD thesis, Fachbereich 4 Informatik, Universität Koblenz, Koblenz, RLP, Okt. 2009.

[amd] Opencl zone, amd. Website, Forum. http://developer.amd.com/zones/OpenCLZone/programming/Pages/default.aspx.

[Bra07] Sebastian Brandt. Entwicklung einer modularen raytracing-engine, Nov. 2007.

[bri] Encyclopedia britannica. Webencyclopedia. www.britannica.com.

[cod] Codermind raytracer tutorial. Artikel. www.codermind.com/articles/Raytracer-in-C++Indtroduction-What-is-ray-tracing.html.

[Dam08] Nina Damasky. Partikelsimulation, Okt. 2008.

[gam] Building a million particle system. Presentation. www.2ld.de/gdc2004/.

[gpg] Gpgpu computing. Artikel. www.planet3dnow.de/vbulletin/showthread.php?t=362621.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *SIGGRAPH*, pages 145–154, 1990.

[khra] Khronos group. Website. http://www.khronos.org/opencl.

[Khrb] Khronos Group. *OpenCL 1.1 C++ Bindings Specifikation*. http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf.

[Khrc] Khronos Group. *OpenCL 1.2 Extensions Specifikation*. http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf.

[Khrd] Khronos Group. *OpenCL 1.2 Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[kne] Physics - particle systems. Presentation. www.kernelz.de/wp-content/uploads/2009/06/particlesystems-helminger.pdf.

[kno] Computing with nvidia's cuda and opencl. Artikel. knol.google.com/k/computing-with-nvidia-s-cuda-and-opencl#.

[nvi] Opencl zone, nvidia. Website, Forum. http://developer.nvidia.com/opencl.

[ocl] Opencl tutorials. Website. http://www.codeproject.com/search.aspx?aidlst=1177&q=*&doctypeid=1.

[Pes06] Yvo Pesek. Simulation von feuer mit hilfe eines partikelsystems, Jan. 2006.

[Rab08] Hanno Rabe. Ray-tracing mit cuda. Diplomarbeit, Institut für Computervisualistik, Universität Koblenz, Koblenz, RLP, Apr. 2008.

[Whi79] Turner Whitted. *An Improved Illumination Model for Shaded Display*. PhD thesis, Bell Laboratories, Holmdel, New Jersey, Okt. 1979. www.cs.drexel.edu/ david/Classes/CS586/Papers/p343-whitted.pdf.