Information Technology

GPU basierte Verfahren zur interaktiven Simulation und Darstellung von Fluid-Effekten

A GPU Framework for Interactive Simulation and Rendering of Fluid Effects

Dr. Jens Harald Krüger: Scientific Computing and Imaging Institute, University of Utah, South Central Campus Drive, 3616 WEB, 72, 84112 Salt Lake City, USA
Tel: +1-801-587-9906, Fax: +1-801-587-6513, E-Mail: jens@sci.utah.edu
Dr. Jens Harald Krüger wurde am 3. Juli 1975 in Duisburg geboren. Nach Abschluss seines Informatikstudiums an der RWTH-Aachen im Jahre 2002 trat er als Doktorand der neuen Computer Grafik- und Visualisierungsgruppe von Prof. Dr. Rüdiger Westermann an der Technischen Universität München bei. Dort erhielt er im Jahre 2003 als erster europäischer Doktorand den internationalen Forschungspreis, den ATI-Fellowship-Award. Im Jahre 2006 beendete er seine Doktorarbeit und arbeitete dort zunächst als technischer Koordinator des exploraTUM-Projekts und Postdoc-Forscher weiter an der TU-München, bis er im Februar 2008 eine Postdoc-Stelle an der University of Utah annahm. Dort forscht er z.Z. an Echtzeit-Visualisierungstechniken für Petascale-Datensätze auf großen GPU Clustern.

Keywords: computer graphics, graphics hardware, GPGPU, fluid simulation, real time application, secondary effects

Schlagworte: Computer Grafik, Grafikkarten, GPGPU, Fluidsimulation, Echtzeitanwendungen, Sekundre Effekte

MS-ID:	jens@sci.utah.edu	21st April 2008
Heft: 46/3 (2004)		

Abstract

This thesis aims at exploiting graphics processing units (GPUs) for interactive simulation and rendering of fluid effects. I target the GPU for a couple of reasons. First, GPUs present the first commercially successful examples of a class of future computing architectures key to high-performance, cost-effective supercomputing. Second, as the simulation is carried out on the GPU, simulation results are already where they are needed for display—in local GPU memory. This eliminates CPU-GPU data transfer, which is likely to become the bottleneck otherwise.

To be able to implement general techniques of numerical computing on the GPU I have developed a linear algebra framework which allows to abstract from the underlying GPU data structures and algorithms, to focus on the application itself. Based on this framework I have implemented efficient algorithms for solving large systems of linear equations, and I have used these algorithms to solve partial differential equations such as the wave equation or the Navier-Stokes equations on GPUs. In addition to physics-based simulation of fluid phenomena, in this thesis I also present new techniques to simulate optical effects caused by such phenomena; i.e., caustics that can appear whenever light impinges upon reflecting or transmitting material.

Zusammenfassung

Ziel dieser Arbeit ist die Entwicklung von Konzepten und Methoden zur interaktiven visuellen Simulation von Fluid-Phänomenen auf PC-Grafikkarten (GPUs). Für die numerische Simulation auf GPUs wurde eine GPU-Abstraktionsschicht entwickelt, die Operatoren für Lineare Algebra bereitstellt. Damit wurden komplexere Algorithmen, z.B. Löser für große lineare Gleichungssysteme, realisiert und zur effizienten numerischen Lösung von Differentialgleichungen auf der GPU verwendet. Zur Modellierung von Strömungsstrukturen wurden neue Interaktionstechniken entwickelt. Für die Darstellung der dynamischen Phänomene wurden partikel- und texturbasierte Volume-Rendering-Techniken erforscht. Durch das Zusammenspiel mit der Simulation auf der GPU lassen sich realistische 3D-Effekte sehr schnell generieren und visualisieren.

1 Einführung

In den letzten Jahren zeigte sich ein stetig wachsender Bedarf an realistischen natürlichen Effekten in virtuellen Umgebungen, Computerspielen, "Infotainment" und wissenschaftlichen Anwendungen (s. Abb. 2). Viele dieser Anwendungen benötigen die Echtzeit-Generierung, -Interaktion und -Darstellung dieser Effekte auf handelsüblicher günstiger PC-Hardware. Auf diesen PC-Systemen wird heutzutage bereits ein Großteil der Darstellungsarbeit von den Grafikkarten, den sog. Graphics-Processing-Units (GPUs), erledigt. Um mit dem rasch wachsenden Aufwand zur Darstellung von 3D-Umgebungen und insbesondere von Computerspielen mithalten zu können wurden diese GPUs in den letzten Jahren massiv beschleunigt, was zu einer dreifach schnelleren Leistungssteigerung, als vom Moore's Law vorhergesagt, führte. Heutzutage übersteigt die Leistungsfähigkeit von handelsüblichen GPUs die Leistung selbst der modernsten Multicore-CPUs um mehrere Größenordnungen. Um den Transfer von Algorithmen auf die GPU zu vereinheitlichen und zu vereinfachen wurde in dieser Arbeit eine mehrschichtige Softwarebibliothek zur effizienten Simulation und Darstellung von Fluiden und davon abgeleiteten Phänomenen entwickelt.

Wie in Abbildung 1 zu erkennen, fußt die Arbeit auf zwei Schichten, welche wiederum direkt auf der Grafik-API aufsetzen, dem Framework zur Linearen Algebra auf GPUs und den höheren LA-Operationen. In diesen Komponenten werden, ähnlich der Aufteilung in BLAS und LAPACK, Bibliotheken komplexerer Operatoren zur Lösung großer linearer Gleichungssysteme, aufbauend auf einem einfachen Basissystem, realisiert. Diese Operatoren wiederum bilden die Grundlage für höhere Funktionen zur numerischen Lösung komplexer gewöhnlicher und partieller Differentialgleichungen. Somit stehen die Ergebnisse dieser Simulationen vollständig und in einer GPU-freundlichen Datenstruktur ohne zeitraubende Datenkonvertierung oder Bus-Transfer auf der Grafikkarte zur Verfügung.

Zusätzlich zu diesen reinen Simulationssystemen werden in dieser Arbeit auch neue Wege präsentiert um das Fluid realistisch und effizient anzuzeigen und um die komplexen Interaktionen mit der Umgebung zu simulieren und darzustellen.

2 Grafikhardware

Um ein Bild einer dreidimensionalen Szene auf einer Grafikkarte zu generieren wird die Geometrie zunächst in eine Menge von Dreiecken unterteilt. Diese Dreiecke wiederum werden in ihre drei Eckpunkte zerlegt. Die Eckpunkte werden dann in zwei Listen, den sog. Index- und Vertex-Puffern gespeichert. Diese Listen dienen als Eingabe in die GPU-Pipeline. Auf der Grafikkarte werden diese Informationen zunächst in der Vertex-Stufe in parallelen Einheiten verarbeitet. In dieser werden letztlich die Koordinaten aller Dreiecke vom 3D-Raum in den 2D-Raum des Bildschirms projiziert und an die nächste Pipelinestufe – den Rasterisierer – weitergegeben. Dieser berechnet für jedes Dreieck, welche Pixel auf dem Bildschirm überdeckt werden und generiert somit einen neuen Datenstrom – den sog. Fragmentstrom. Dieser neue Datenstrom wird an die Pixelshader-Einheiten weitergeleitet. Auf aktuellen Grafikkarten kommen über 100 dieser Einheiten zum Einsatz. In dieser Pipeline-Stufe wird für jedes Fragment eine Farbe berechnet. Hierzu wurden vom Rasterisierer jedem Fragment durch Interpolation Daten aus den Dreieckspunkten zugeordnet. Beispiele hierfür sind Farben, Beleuchtungsinformationen und insbesondere auch sog. Texturkoordinaten. Diese Koordinaten werden als Indizes in große mehrdimensionale Arrays – sog. Texturen – verwendet.

Im weiteren Verlauf dieses Textes wird nun erläutert, wie durch geschickte Konfiguration dieser Pipeline die Grafikkarte dazu gebracht wird, anstatt Bilder von einer Szene zu generieren, zunächst LA-Operationen auszuführen und schließlich Lösungen für partielle Differentialgleichungen zu berechnen.

3 Lineare Algebra

Hier soll nur auf die Realisierung von Vektoren und deren Operationen auf der Grafikkarte eingegangen werden. Für eine ausführliche Beschreibung der komplexeren LA-Objekte, wie z.B. dünnbesetze Matrizen, sei der interessierte Leser auf den vollständigen Text der Dissertation verwiesen [1].

Um einen Algorithmus auf die GPU zu portieren, müssen zunächst zwei Fragen beantwortet werden: erstens, wie die Daten gespeichert werden sollen, und zweitens, wie die Berechnungen auf diesen Daten auszuführen sind. Wie bereits oben erwähnt, sind die Shader-Einheiten auf Grafikkarten hochgradig parallelisiert und somit sehr leistungsfähig. Gleichzeitig erlauben diese Einheiten einen effizienten Zugriff auf die Texturen. Diese sind im Prinzip große Datenfelder, welche man sich vereinfacht als auf der GPU gespeicherte Bilder vorstellen kann. Da GPUs erlauben, die Ausgabe der Grafikpipeline in eine Textur umzuleiten, statt auf dem Bildschirm anzuzeigen, eignen sich diese hervorragend

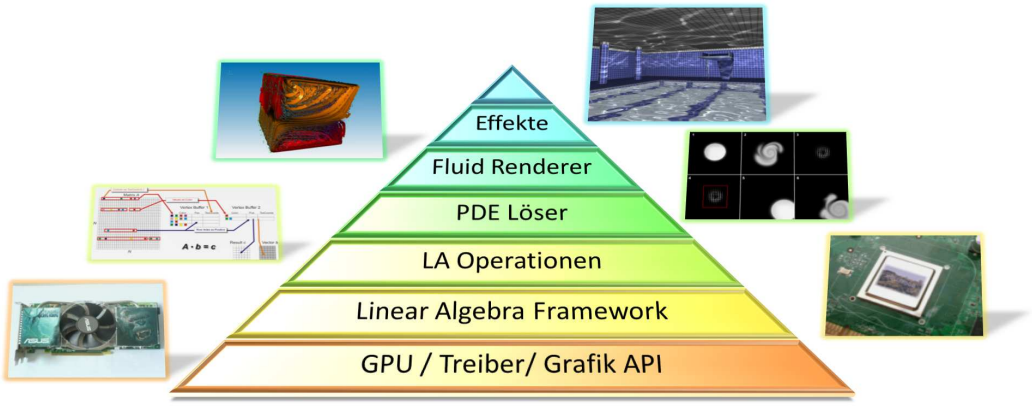

Bild 1: Eine Übersicht über die in dieser Dissertation erstellten Softwarekomponenten.

als Lese/Schreib-Speicherort für Vektoren und Matrizen. Zum Verständnis einer GPU Realisierung betrachten wir im Folgenden das Beispiel einer Vektor/Vektor-Addition.

Dazu werden die beiden als Felder A und B repräsentierten Vektoren direkt an die GPU als Texturen weitergeleitet. Auf einer CPU würde die Vektoraddition durch eine einfache Schleife mit der Berechnungsvorschrift $C[i] = A[i] + B[i]$ realisiert. Auf der GPU wird genau die gleiche Vorschrift in den Pixel-Shader geladen und der Rasterisierer so konfiguriert, dass er für jedes Element der Schleife ein Fragment generiert und damit den Pixelshader einmal aufruft. Hierzu wird ein einziges Viereck in die Grafikpipeline geschickt. Die komplette Vertextransformationsstufe wird deaktiviert, so dass das Viereck ohne Veränderung direkt zum Rasterisierer gelangt. Die Geometrie des Vierecks wird so gewählt, dass es genau den virtuellen Bildschirm (den sog. Viewport) überdeckt, und der Bildschirm wiederum ist so konfiguriert, dass er genau die Größe des Zielvektors hat.

Prinzipiell ist also die GPU Implementierung der CPU Implementierung sehr ähnlich, lediglich die Feldadressierung wird statt direkt durch Indizes über vom Rasterisierer generierte Texturkoordinaten ausgeführt und die Schleifenausführung wird durch die Fragmentgenerierung ersetzt. Wichtig für das Verständnis, warum diese Realisierung schneller ist, ist die Tatsache, dass auf der GPU der kurze Shader Kernel in vielen Shadereinheiten parallel ausgeführt wird und somit deutlich effizienter abgearbeitet wird als die sequenzielle Schleife auf der CPU. Aufbauend auf diesem einfachen Beispiel kann man weitere Vektor/Vektor-Operationen implementieren. Darauf aufbauend lassen sich wiederum schnelle Matrix/Vektor-Operationen ableiten. Im Rahmen der Arbeit wurde ein solches Set von Daten und Operationen in einer C++ - Klassenhierarchie soweit gekapselt, dass man als Programmierer lediglich Skalar-, Vektor- und Matrix-Objekte erstellen muss, und deren Methoden automatisch die nötigen Texturen, Shader und weitere Hilfsdatenstrukturen anlegen und schon in dieser Ebene vollständig von der eigentlichen GPU-Implementierung abstrahieren. Mit diesem Grundgerüst lassen sich schnell komplexere Algorithmen implementieren. Ein konjugierter Gradienten-Löser beispielsweise ist mittels der GPU-LA-Bibliothek in weniger als zehn Zeilen C++ - Code realisiert . Dieser Löser wird wiederum verwendet um partielle Differentialgleichungen (PDEs) schnell und elegant auf der GPU zu berechnen und damit Fluid-Phänomene in Echtzeit zu simulieren.

4 Fluid Simulation

Im Folgenden wird anhand eines einfachen Beispiels – der Flachwasser-Wellengleichung – die prinzipielle Vorgehensweise bei der Lösung von PDEs demonstriert. Mit ähnlichen Methoden ist es auch möglich deutlich komplexere PDEs, wie z.B. die Navier-Stokes-Gleichungen, numerisch zu lösen. Um das Verhalten einer Wasseroberfläche zu beschreiben betrachten wir zunächst das physikalische Modell einer 2D Membran:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Hier wird – gegeben die Wellengeschwindigkeit c – die Höhe der Wasseroberfläche u zu einem Zeitpunkt t an einer Position x, y berechnet. Zur numerischen Lösung dieser PDE wird die Oberfläche zunächst auf einem regulären Gitter diskretisiert und die partiellen Ableitungen durch finite Differenzen approximiert. Die Wellengleichung kann somit folgendermaßen implizit formuliert werden:

$$(4\alpha + 1) \cdot u_{ij}^{t+1} - \alpha \cdot (u_{i+1j}^{t+1} + u_{i-1j}^{t+1} + u_{ij+1}^{t+1} + u_{ij-1}^{t+1}) = F\left(u^t, u^{t-1}\right)$$

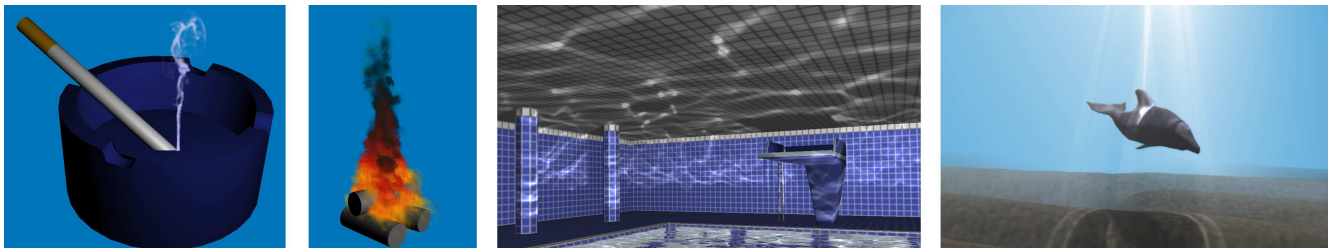

Bild 2: Eine Auswahl von Fluid-Phänomenen auf PC-Grafikkarten. Alle diese Bilder werden auf heutiger Standardhardware in Echtzeit sowohl simuliert als auch dargestellt.

Zur Lösung dieser impliziten linearen Gleichung nach u^{t+1} wird sie in eine – für finite Differenzen Schemata typische – bandstrukturierte Matrix und in einen von den letzten Zeitschritten abhängigen Lösungsvektor umgeformt. Das resultierende dünnbesetzte Gleichungssystem kann durch einige wenige Codezeilen mittels der LA-Bibliothek folgendermaßen numerisch gelöst werden:

```
1  // Lösung des GS
2  c->computeRHS(lastSurface, currentSurface);
3  nextSurface = cgSolver->solve();
4  lastSurface->copyVector(currentSurface);
5  currentSurface->copyVector(nextSurface);
6  // Oberfläche stören, z.B. durch Benutzerinteraktion
7  AnwendungExternerKraefte(currentSurface);
8  // Anzeige
9  ZeichneWasser(currentSurface->m_pVectorTexture);
```

Auch ohne genaue Kenntnis der LA-GPU-Funktionen wird an diesem Codefragment deutlich, dass die Berechnung und Datenspeicherung der Wasseroberfläche im GPU-Speicher Funktionsaufrufe wie `ZeichneWasser` ohne CPU-GPU-Datentransfer ermöglicht. Somit steht einer effizienten Echtzeitvisualisierung, die nicht durch aufwändige Bustransfer-Operationen gebremst wird, nichts mehr im Wege.

5 Effekte

Wie in Abbildung 2 zu sehen, lässt sich die Darstellung von Fluiden erheblich realistischer gestalten, wenn anstatt der direkten Visualisierung der Simulationsergebnisse noch eine weitere computergraphische Berechnung auf die Simulation angewendet wird. Das linke Bild in dieser Abbildung zeigt ein Beispiel einer komplexeren Navier-Stokes Simulation auf GPUs. Hier wird das Verhalten eines inkompressiblen Fluids berechnet. In diesem Beispiel wurde über die reine Simulation hinaus noch eine realistische Darstellung mittels eines GPU-basierten Partikelsystems verwendet. Während die bisher dargestellten Anzeigemodalitäten erlauben, Fluid-Phänomene losgelöst von ihrer Umgebung realistisch darzustellen, fehlt zur Integration in interaktive virtuelle Umgebungen noch ein wesentlicher Schritt: die schnelle und realistische Berechnung der komplexen Interaktionen der Simulation mit der umgebenden Szene. Während eine Integration der Szenenparameter und der Benutzerinteraktionen in die Simulation meist relativ einfach über die Randbedingungen erfolgen kann (z.B. Regen auf eine Wasseroberfläche), ist die umgekehrte Abbildung meist sehr komplex (beispielsweise die Interaktion von reflektiertem und refraktiertem Licht an einer Wasseroberfläche oder unter Wasser, s. Abbildung 2). Um auch diese Phänomene in Echtzeit handhaben zu können wurde im Rahmen dieser Arbeit ein GPU-basierter Photo-Mapper realisiert. Dieses System verfolgt die Bewegung einer großen Menge von Photonen innerhalb der Szene und des simulierten Fluids und speichert deren Interaktionen (z.B. Absorptionspositionen von refraktierten und reflektieren Photonen). Beim finalen Anzeigevorgang werden diese Informationen verwendet um eine realistische Beleuchtungssimulation der Szene berechnen zu können.

6 Zusammenfassung

In dieser Arbeit wurde eine generell einsetzbare Bibliothek zur interaktiven Simulation und Darstellung von Fluid-Phänomenen beschrieben. Aufbauend auf einem Grundgerüst wurden neue Methoden zur Modellierung und Darstellung dreidimensionaler Phänomene vorgestellt, die auf Grund ihrer Flexibilität und ihrer Geschwindigkeit bereits erfolgreich in reale Computerspiel-Engines und virtuelle Umgebungen integriert wurden.

Während zu Beginn dieser Arbeit im Jahre 2002 das Potential von Grafikkarten für numerische Berechnung praktisch noch nicht wahrgenommen wurde, waren es nicht zuletzt die hier präsentierten Forschungen und Resultate, welche die beiden weltweit größten Hersteller von Grafikhardware motivieren, seit dem Jahr 2007 spezielle Versionen ihrer Grafikkarten zur numerischen Simulation auf den Markt zu bringen.

Literatur

[1] Jens Krüger: A GPU Framework for Interactive Simulation and Rendering of Fluid Effects *http://mediatum2.ub.tum.de/node?id=604112*