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

Jens Harald Krüger

Lehrstuhl für Computer Grafik und Visualisierung
der Fakultät für Informatik der Technischen Universität München
jens.krueger@in.tum.de

Abstract: 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 (s. Abbildung 1)

Abbildung 1: Eine Auswahl von Fluid-Phänomenen auf PC-Grafikkarten. Alle diese Bilder wurden auf heutiger Standardhardware in Echtzeit sowohl simuliert als auch dargestellt.

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. Abbildung 1). Viele dieser Anwendungen benötigen die Echtzeit-Generierung, -Interaktion und -Darstellung dieser Effekte, nicht nur auf Supercomputern bzw. großen Clustersystemen, sondern auch auf handelsüblicher günstiger PC-Hardware. Auf diesen PC-Systemen wird heutzutage bereits ein Großteil der Darstellungsarbeit – insbesondere von 3D-Applikationen – 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 Moores Law vorhergesagt, führte. Heutzutage übersteigt die Leistungsfähigkeit von handelsüblichen GPUs, wie sie in praktisch jeden Discounter-PC eingebaut sind, die Leistung selbst der modernsten Multicore-CPUs um mehrere Größenordnungen [OLG+07]. Allein schon die bloße Rechenkapazität von GPUs lässt diese Subsysteme als ideale Plattformen für die numerische Simulation erscheinen, aber – insbesondere für die o.g. zeitkritischen Systeme – ist ein weiterer noch wichtigerer Vorteil von Simulationen direkt auf der GPU die Tatsache, dass heutige Bus-Systeme mit der enormen Datenmenge, welche für interaktive Umgebungen benötigt wird, nicht Schritt halten können. Daher erweist es sich als außerordentlich nützlich, nicht nur die reine Darstellungsarbeit der Simulationsergebnisse, sondern auch die Simulation selbst auf die schnellen und hochgradig parallel arbeitenden GPUs zu verlagern und dadurch jeglichen Bus-Transfer zu vermeiden.

Um den Transfer von Algorithmen und Systemen auf die GPU zu vereinheitlichen und zu vereinfachen wird in dieser Arbeit eine mehrschichtige Softwarebibliothek zur effizienten Simulation und Darstellung von Fluiden und davon abgeleiteten Phänomenen auf Standard-PC-Grafikkarten, vorgestellt (s. Abbildung 2).

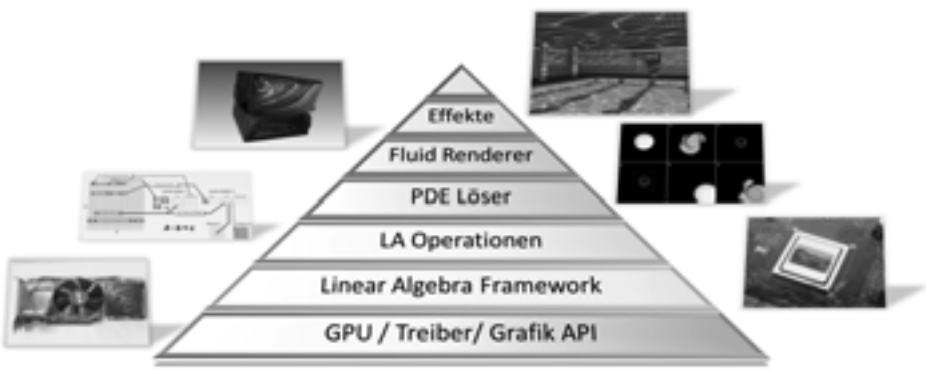


Abbildung 2: Eine Übersicht über die in dieser Dissertation erstellten Softwarekomponenten. Aufbauend auf Standard-Grafik-APIs wurde zunächst in einer GPU-Abstraktionsschicht die Basisfunktionalität für Berechnungen der Linearen Algebra auf GPUs realisiert. Auf dieser Basisschnittstelle wurden zunächst weitere komplexere Operatoren und damit schließlich ein kompletter Fluid-Löser realisiert. Abschließend wurde ein effizientes Darstellungsframework zur interaktiven Visualisierung und Interaktion mit den Fluid-Phänomen geschaffen, welches auch in der Lage ist komplexe Interaktionen der Simulation mit der virtuellen Umgebung zu simulieren.

Wie in Abbildung 2 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 [DDCHH88] und LAPACK [ABB+99], 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, wie z.B. die Flachwasser-Differentialgleichung oder die Navier-Stokes-Gleichungen. Somit stehen die Ergebnisse dieser Simulationen vollständig und in einer GPU-freundlichen Datenstruktur ohne zeitraubende Datenkonvertierung oder Bus-Transfer auf der Grafikkarte zur Verfügung und können effizient dargestellt werden.

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. Beispiele für solche Effekte sind Kaustiken, welche sich bilden, wenn Licht auf eine bewegte Wasseroberfläche fällt, oder die Animation von Materialien, welche mit einer simulierten Strömung interagieren, z.B. Rauch, Feuer oder weitere partikelbasierte Phänomene. Unter Verwendung der Techniken und Softwarekomponenten, die in dieser Arbeit entwickelt wurden, können alle diese angesprochenen Effekte in Echtzeit auf günstiger Standardhardware simuliert und dargestellt werden.

Letztlich wird durch die komplette Verlagerung all dieser Simulations- und Darstellungsaufgaben auf die GPU auch die CPU des Systems stark entlastet, so dass dort mehr Resourcen für andere Aufgaben von Echtzeitsystemen zur Verfügung stehen, wie z.B. das Tracking, die Kollisionserkennung, Spieler KI oder die Pfadsuche.

In den nun folgenden Kapiteln wird – im Rahmen der gebotenen Möglichkeiten – jeweils ein knapper Überblick über die o.g. Komponenten und ihre Realisierung gegeben.

2 Moderne Grafikkarten

Die Entwicklungen in der vorliegenden Arbeit sind eng verknüpft mit den enormen GPU-Entwicklungen der letzten Jahre. In kurzer Zeit hat sich die Grafikhardware von mannshohen Mainframe-Systemen zu kleinen günstigen PC-Steckkarten entwickelt. Insbesondere hat sich die inzwischen hochintegrierte GPU vom simplen Framebuffer zur reinen visuellen Ausgabe von Daten, zum leistungsfähigen hochgradig parallelen und nahezu frei programmierbaren Stream-Prozessor gewandelt. Wollen wir daher kurz auf die Funktionsweise aktueller GPUs eingehen.

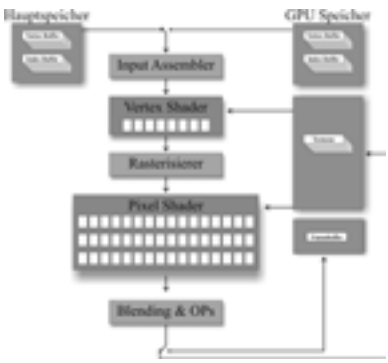


Abbildung 3: Schematischer Aufbau einer aktuellen Grafikkarte. Heutige GPUs arbeiten als Stream-Prozessoren, welche einen Eingabestrom aus Vertex- und Index-Puffern durch verschiedene z.T. hochgradig parallele Stufen führen um schließlich ein Bild auf dem Bildschirm oder im Speicher der Grafikkarte zu generieren.

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 (s. Abbildung 3). Auf der Grafikkarte wer-

den diese Informationen zunächst in der Vertex-Stufe in parallelen Einheiten verarbeitet. Hier können die Objektkoordinaten z.B. gedreht, verschoben oder anderweitig transformiert werden, letztlich werden 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. Da i.A. ein Dreieck viele Pixel auf dem Bildschirm überdeckt, ist davon auszugehen, dass der neu entstandene Fragmentstrom deutlich größer ist als der eingehende Vertexstrom. Daher sind die – in der Pipeline folgenden – Pixelshader-Einheiten auch nochmals stärker parallelisiert als die vorherigen Stufen der Pipeline. 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. Abschließend durchlaufen die Fagmente noch eine finale Stufe, in der z.B. durch den Tiefentest verdeckte Fragmente erkannt und verworfen werden können.

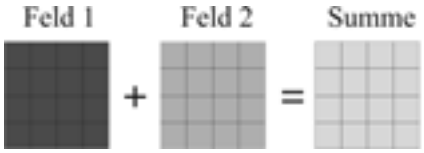
Im weiteren Verlauf dieses Textes wird nun erläutert, wie durch geschickte Konfiguration dieser Pipeline und insbesondere der programmierbaren Pixelshader-Einheiten die Grafikkarte dazu gebracht wird, anstatt Bilder von einer Szene zu generieren, zunächst LA-Operationen auszuführen und schließlich Lösungen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

3 Lineare Algebra

Wegen des knappen Rahmens dieses Textes kann im Folgenden nur auf die Realisation 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 [Krü06].

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 Pixelshader-Einheiten auf Grafikkarten am stärksten parallelisiert und somit am leistungsfähigsten. Gleichzeitig erlauben diese Einheiten einen effizienten Zugriff auf die Texturen. Texturen sind im Prinzip große Datenfelder. Betrachtet man nun nur eine bestimmte Klasse, die sog. 2D-Texturen, so kann man sich diese vereinfachend als auf der GPU gespeicherte Bilder vorstellen, welche im Normalfall auf die Oberfläche von 3D-Objekten gelegt werden, um so eine komplexere Struktur zu modellieren. Da GPUs auch erlauben, die Ausgabe der gesamten Grafikpipeline so umzuleiten, dass das Ergebnis statt auf dem Bildschirm wieder in eine Textur geschrieben wird, eignen sich diese Texturen hervorragend als Speicherort für Vektoren und die programmierbaren Pixelshader als Berechnungseinheiten.

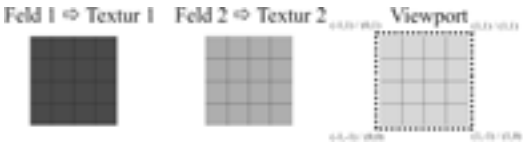
Wie können 2D-Texturen am effizientesten mit den Daten bestückt und die GPU-Pipeline am besten konfiguriert werden um z.B. eine Vektor/Vektor-Addition durchzuführen? Betrachten wir dazu zunächst eine einfache Implementierung dieses Problems auf der CPU (s. Abbildung 4).




```
for (int x = 0; x < N; x++) {
   for (int y = 0; y < N; y++) {
     Summe[x][y] = Feld1[x][y] + Feld2[x][y].
   }
}
```

Abbildung 4: Schema einer komponentenweisen Addition zweier Datenfelder. Links das Layout der Daten, rechts der notwendige C-Code.

In diesem Beispiel wurden die beiden Vektoren direkt als 2D-Felder repräsentiert um die GPU-Portierung auf 2D-Texuren besser verstehen zu können. Im Prinzip kann man sich die Vektoren aber immer noch als "aufgewickelte" 1D-Datenstruktur vorstellen. Einen Algorithmus, wie in Abbildung 4 gezeigt, auf der GPU auszuführen ist nahezu noch einfacher als auf der CPU. Dazu wird der Schleifenrumpf in den Pixelshader geladen und der Rasterisierer so konfiguriert, dass er für jedes Element der Schleife ein Fragment generiert und damit den Pixelshader einmal aufruft. Abbildung 5 zeigt das notwendige Setup. Hierzu wird ein spezielles Viereck in die Grafikpipeline geschickt. Die komplette Vertextransformationsstufe wird deaktiviert, so dass das Viereck ohne Veränderung direkt zum Rasterisierer gelangt. Die Geometrie des Vierecks wurde 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 (s. Abbildung 5).




```
void PixelShader(in  float2  c0    : TEXCOORD0,
                 out float4  Summe : COLOR0) {

  Summe = tex2D(textur1, c0) + tex2D(textur2, c0).
}
```

Abbildung 5: Schema einer komponentenweise Addition zweier – in Texturen gespeicherter – Datenfelder auf einer GPU. Links das Layout der Daten, rechts der notwendige Shader-Code in der C-ähnlichen GPU-Sprache HLSL.

Wie in Abbildung 5 zu erkennen, ist der Shader-Code dem Schleifenrumpf sehr ähnlich. Lediglich die Feldadressierung wird statt direkt durch Indizes über vom Rasterisierer generierte Texturkoordinaten ausgeführt. Wichtig für die Berechnung ist, dass auf der GPU der kurze HLSL-Shader Kernel in vielen Shadereinheiten parallel ausgefürt 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 schnell Matrix/Vektor-Operationen ableiten, z.B. indem man eine Matrix/Vektor-Multiplikation als eine Sequenz von aufsummierten Vektor/Vektor- Multiplikationen auffasst. (Für genauere Details sei der interessierte Leser nochmals auf die vollständige Arbeit [Krü06] verwiesen). Dieses Set von Daten und Operationen wurde in einer C++ - Klassenhierarchie soweit gekapselt, dass man als Program-

mierer 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 aus einfachen Datentypen lassen sich einfach komplexere Algorithmen implementieren. Ein konjugierter Gradienten-Löser ist mittels der GPU-LA-Bibliothek in weniger als zehn Zeilen C++ - Code realisiert [Krü06, KW03]. Dieser Löser wiederum wird im Folgenden verwendet um partielle Differentialgleichungen (PDEs) schnell und elegant auf der GPU zu berechnen.

4 Simulation von Fluid-Phänomenen

Im Folgenden wird anhand eines einfachen Beispieles – 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 [Krü06].

Um das Verhalten einer Wasseroberfläche zu beschreiben betrachten wir zunächst das physikalische Modell einer zweidimensionalen Membran (s. Gleichung 1).

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{1}$$

In Gleichung 1 wird – gegeben die Wellengeschwindigkeit c – die Höhe der Wasseroberfläche zu einem Zeitpunkt t an einer Position x, y im Parameter u berechnet. Zur numerischen Lösung dieser PDE wird die Oberfläche zunächst auf einem regulären Gitter diskretisiert und die partiellen Ableitungen durch finite Differenzen approximiert. Somit wird Gleichung 1 in Gleichung 2 überführt, und für jeden Gitterpunkt i, j gilt:

$$\frac{u_{i,j}^{t+1} - 2u_{i,j}^{t} + u_{i,j}^{t-1}}{\Delta t^2} = c^2 \cdot \left(\frac{u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t}}{h^2} \right) \tag{2}$$

Auflösung nach der Unbekannten $u_{i,j}^{t+1}$ liefert schließlich die explizite Formulierung 3.

$$\begin{gathered} u_{i,j}^{t+1} = \alpha \cdot \left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} \right) + (2 - 4\alpha) \cdot u_{i,j}^{t} - u_{i,j}^{t-1} \\ \text{mit} \\ \alpha = \frac{\Delta t^2 \cdot c^2}{h^2} \end{gathered} \tag{3}$$

Diese Gleichung lässt sich zwar durch einfaches Einsetzen der Randbedingungen berechnen, hat jedoch den entscheidenden Nachteil, dass die Stabilität der Simulation stark von

der Wahl des Zeitintervalls und der Wellengeschwindigkeit c abhängig ist. Durch Umformung mittels des Crank-Nicholson-Schemas $u^t_{i,j} := 0.5 \cdot \left(u^{t+1}_{i,j} + u^t_{i,j}\right)$ lässt sich Gleichung 3 in die bedingungslos stabile Gleichung 4 umformen.

$$c\,(4\alpha+1) \cdot u^{t+1}_{ij} - \alpha \cdot \left(u^{t+1}_{i+1j} + u^{t+1}_{i-1j} + u^{t+1}_{ij+1} + u^{t+1}_{ij-1}\right) = c^t_{ij}$$
$$\textbf{mit}$$
$$c^t_{ij} = \alpha \cdot \left(u^t_{i+1j} + u^t_{i-1j} + u^t_{ij+1} + u^t_{ij-1}\right) + (2-4\alpha) \cdot u^t_{ij} - u^{t-1}_{ij} \textbf{ und } \alpha = \tfrac{\Delta t^2 \cdot c^2}{h^2} \tag{4}$$

Zur Anwendung der LA-Operationen auf Gleichung 4 kann diese in die Matrix/Vektor-Schreibweise umgeformt werden (s. Gleichung 5 für ein Beispiel eines 3×3 Gitters).

$$\begin{pmatrix}
4\alpha+1 & -\alpha & & -\alpha & & & & & \\
-\alpha & 4\alpha+1 & -\alpha & & -\alpha & & & & \\
 & -\alpha & 4\alpha+1 & & & -\alpha & & & \\
-\alpha & & & 4\alpha+1 & -\alpha & & -\alpha & & \\
 & -\alpha & & -\alpha & 4\alpha+1 & -\alpha & & -\alpha & \\
 & & -\alpha & & -\alpha & 2-4 & -\alpha & & -\alpha \\
 & & & -\alpha & & & 4\alpha+1 & -\alpha & \\
 & & & & -\alpha & & -\alpha & 4\alpha+1 & -\alpha \\
 & & & & & -\alpha & & -\alpha & 4\alpha+1
\end{pmatrix} \cdot \begin{pmatrix} u^{t+1}_{0,0} \\ u^{t+1}_{1,0} \\ u^{t+1}_{0,1} \\ u^{t+1}_{1,1} \\ u^{t+1}_{0,2} \\ u^{t+1}_{1,2} \\ u^{t+1}_{2,2} \end{pmatrix} = \begin{pmatrix} c^t_{0,0} \\ c^t_{1,0} \\ c^t_{2,0} \\ c^t_{0,1} \\ c^t_{1,1} \\ c^t_{2,1} \\ c^t_{0,2} \\ c^t_{1,2} \\ c^t_{2,2} \end{pmatrix} \tag{5}$$

Dieses Gleichungssystem 5 kann jetzt durch einige wenige Codezeilen mittels der GPU-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. mit Regentropfen
7  AnwendungExternerKraefte(currentSurface);
8  // Anzeige
9  ZeichneOberflaeche(currentSurface->m_pVectorTexture);
```

Auch ohne genaue Kenntnis der LA-GPU-Funktionen sollte an diesem Codefragment deutlich werden, dass die Berechnung und Datenspeicherung der Wasseroberfläche im GPU-Speicher Funktionsaufrufe wie `ZeichneOberflaeche` ohne CPU-GPU-Datentransfer ermöglicht. Somit steht einer effizienten Echtzeitvisualisierung, die nicht durch zeitaufwändige Bustransfer-Operationen gebremst wird, nichts mehr im Wege (s. Abbildung 6).

Abbildung 6: Links die direkte Visualisierung der Oberflächentextur, wobei die Wasserhöhe als Grauwert interpretiert wird. Im mittleren Bild wird die Oberfläche als Eingabe in einen Refraktions-Pixelshader verwendet, um eine realistischere Darstellung zu erzielen. Im rechten Bild wird das Höhenfeld schließlich dazu verwendet, ein reguläres Gitter zu positionieren. Auf aktueller Grafikhardware benötigt die Simulation und Darstellung dieser Bilder deutlich weniger als eine Millisekunde pro Frame, ohne dabei die CPU zu belasten.

5 Darstellung und Effekte

Wie schon in Abbildung 6 Mitte gezeigt, lässt sich die Darstellung von Fluiden erheblich realistischer gestalten, wenn anstatt der direkten Visualisierung der Simulationsergebnisse (Abbildung 6 links) noch eine weitere computergraphische Berechnung auf die Simulation angewendet wird. In Abbildung 6 werden dazu durch Berechnung der partiellen Differenzen zunächst die Tangenten der Oberfläche und daraus durch das Kreuzprodukt die Oberflächennormale für jeden Punkt bestimmt. Diese Normale wird im Pixelshader zur Berechnung des Refraktionsstrahls genutzt um damit eine realistische Wasseroberfläche zu generieren.

Abbildung 7: Simulation und Darstellung dieser dreidimensionalen Strömung erfolgen in Echtzeit auf aktueller Grafikhardware.

Abbildung 7 zeigt ein weiteres Beispiel einer komplexeren Simulation auf GPUs. Hier werden die Navier-Stokes-Gleichungen verwendet um das Verhalten eines inkompressiblen Fluids zu beschreiben [Krü06]. Auch in diesem Beispiel wurde über die reine Simulation hinaus noch eine realistische Darstellung mittels eines GPU-basierten Partikelsystems [KKKW05] verwendet.

Während die bisher dargestellten Anzeigenmodalitäten es 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 8). Um auch diese Phänomene in Echtzeit handhaben zu können wurde im Rahmen dieser Arbeit ein GPU-basierter Photo-Mapper realisiert [KBW06, Krü06]. 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 (s. Abbildung 8).

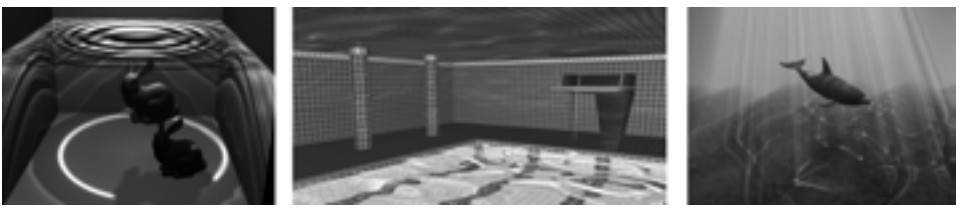


Abbildung 8: Drei Beispiele für interaktive Interaktionen von Licht mit der Szene und der Simulation.

6 Zusammenfassung

In dieser Arbeit wurde eine generell einsetzbare Bibliothek zur interaktiven Simulation und Darstellung von Fluid-Phänomenen beschrieben. Dazu wurde zunächst ein Framework zur Implementierung numerischer Simulationstechniken auf Grafikhardware entwickelt. Dieses Framework verwendet intern effiziente Layouts für verschiedene Typen von Matrizen und Vektoren und kapselt die komplette GPU-Implementierung in C++ - Klassen, so dass diese auch Entwicklern zugänglich gemacht werden können, die nicht mit der GPU vertraut sind. Insgesamt liegt der Fokus der Arbeit darauf, ein allgemeines wiederverwendbares Gerüst zur numerischen Simulation und Darstellung zu schaffen und nicht nur spezielle Effekte auf Grafikhardware zu portieren.

Aufbauend auf diesem 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. Diese Flexibilität und visuelle Qualität bei interaktiven Update-Raten führte dazu, dass sowohl im Jahre 2005 also auch 2006 die im Rahmen dieser Arbeit entwickelte Partikel-Engine den IEEE-Visualization-Contest [SKKW, SKBW] gewann, welcher Teil der “IEEE Visualization”-Konferenz ist – der weltweit größten Konferenz im Bereich der wissenschaftlichen Visualisierung.

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 motivierten, im Jahr 2007 spezielle Versionen ihrer Grafikkarten zur numerischen Simulation auf den Markt zu bringen.

Literatur

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney und D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third. Auflage, 1999.

[DDCHH88] J.J. Dongarra, J. Du Croz, S. Hammarling und R.J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[KBW06] Jens Krüger, Kai Bürger und Rüdiger Westermann. Interactive Screen-Space Accurate Photon Tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering—EGSR)*, Seiten 319–329, June 2006.

[KKKW05] Jens Krüger, Peter Kipfer, Polina Kondratieva und Rüdiger Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6), 11 2005.

[Krü06] Jens Krüger. *A GPU Framework for Interactive Simulation and Rendering of Fluid Effects*. Dissertation, Technische Universität München, http://mediatum2.ub.tum.de/node?id=604112, 2006.

[KW03] Jens Krüger und Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH 2003)*, 22(3):908–916, 2003.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn und Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[SKBW] Jens Schneider, Jens Krüger, Kai Bürger und Rüdiger Westermann. California Streaming, IEEE Visualization contest 2006 first price winning contest entry. http://wwwcg.in.tum.de/Research/Projects/VisContest06.

[SKKW] Jens Schneider, Polina Kondratieva, Jens Krüger und Rüdiger Westermann. All you need is ... – Particles!, IEEE Visualization contest 2005 first price winning contest entry. http://wwwcg.in.tum.de/Research/Projects/VisContest05.

Dr. rer. nat. Jens Harald Krüger wurde am 3. Juli 1975 in Duisburg geboren. Nach Abschluss seines Informatik-Studiums an der RWTH-Aachen im Jahre 2002 trat er als Doktorand der neuen Computer Grafik- und Visualisierungs-Gruppe 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 mit dem Prädikat "summa cum laude" und arbeitet z.Z. als technischer Koordinator des exploraTUM-Projekts und Postdoc-Forscher an der Technischen Universität München.