Effizientes Streamen interaktiver 3D-Computerspiele

Zur performanten Unterstützung unterschiedlichster Endgeräte sind dazu zwei Streaming-Methoden entwickelt worden. Beim Graphics-Streaming werden direkt die Grafikbefehle an Endgeräte mit Grafikprozessor übertragen, um dort das Bild dem Display angepasst zu rendern. Basierend auf zusätzlichen Informationen aus dem Render-Kontext wurde dafür der rechenintensive Videocodierprozess optimiert: Verschiedene in diesem Zusammenhang durchgeführte Optimierungen wie intelligentes Caching, Codierung und Emulation des Grafikkontextes führten zur Reduzierung der Datenrate um 80 %. Alternativ, für Endgeräte ohne GPU, wird der visuelle Output als Video codiert übertragen. Basierend auf zusätzlichen Informationenaus dem Render-Kontext wurde dafür der rechenintensive Videocodierprozess optimiert: im Durchschnitt wurde eine Beschleunigungen von rund 25 % erzielt. Im Artikel wird diese Plattform zum Streamen von interaktiven 3D-Computerspielen vorgestellt.

This article describes a streaming platform for interactive 3D computer gaming. Two streaming methods have been developed to support a wide range of devices. Graphics streaming is used to stream graphics commands directly to the graphics processor incorporated in each end device. The image is then rendered within the processor to match the connected display. Several optimisations have been developed, such as intelligent caching, entropy-coding and local emulation of graphic context, and an optimised encoding process that exploits additional render context information is used to achieve average accelerations of around 25 percent.

Framework

Das Fraunhofer HHI Institut entwickelt zusammen mit dreizehn Partnern im EU-Projekt „Games @Large“ eine Gaming-on-Demand-Plattform. Die Vision des Projekts ist es, ein Framework zu schaffen, das es ermöglicht, gängige 3D-Computerspiele auf beliebigen Endgeräten zu spielen, unabhängig von Ort oder Geräte-Hardware, Betriebssystem usw.

Zur Realisierung wurde ein Streaming-basierter Ansatz gewählt, bei dem Spiele auf Servern ausgeführt werden und lediglich die audio-visuelle Ausgabe zum jeweiligen Endgerät übertragen wird [1]. Anwendungsbereiche ergeben sich überall dort, wo man einerseits nicht vielfach in teure Computer-Hardware investieren will und andererseits den Software-Wartungsaufwand gering halten möchte, zum Beispiel in Hotels, Internet-Cafes, Altersheimen usw. Durch überladen der Graphik-Bibliothek [5] unterstützt die Plattform kommerzielle, handelsübliche Spiele ohne jegliche Anpassung. Clientseitig ist keine lokale Installation nötig, da das Spiel auf dem Server ausgeführt wird. Um viele (kostengünstige) Clients gleichzeitig zu bedienen, können mehrere Instanzen zum Beispiel in einer Server-Farm parallel ausgeführt werden.

Ein Schwerpunk des Projekts zielt darauf ab, möglichst viele verschiedene Typen von Endgeräten mit guter Bildqualität und minimaler Verzögerung zu unterstützen. Der primäre Fokus liegt auf Wireless-Netzwerken, in denen sich die Spieleserver und Clients selbstständig per DLNA konfigurieren. Um interaktives Spielen zu ermöglichen, wurden zwei auf RTP/UDP basierende Streaming-Methoden entwickelt, die je nach Spielart und Endgerät-Charakteristik ausgewählt werden **(Bild 1)**. Um die netzwerkbasierten Verzögerungen gering zu halten, werden QoS-Techniken eingesetzt.

Technische Herausforderung

Die wesentliche Herausforderung der technischen Realisierung ist, die Verzögerung in tolerablen Grenzen zu halten. Untersuchungen für Netzwerkspiele zeigen, wie sich die Reaktionszeit, also die Verzögerung zwischen Eingabe und die darauf folgende Ausgabe, auf den Spielspaß auswirken. Das ist je nach Spielart und Genre unterschiedlich stark. In der Praxis hat sich gezeigt, dass im Allgemeinen Verzögerungen von bis zu 100 ms tolerierbar sind, bevor der Spielspaß merkbar nachlässt.

Um die hohen Anforderungen an die Reaktionszeiten zu realisieren, wurden zwei Streaming-Methoden entwickelt. Beim Graphics-Streaming werden die Grafikbefehle des Spiels zum Client übertragen und dort gerendert. Alternativ wird das Bild serverseitig gerendert, videocodiert, zum Endgerät übertragen und dort decodiert dargestellt.

Graphics-Streaming

Für Endgeräte mit Hardware-seitiger Grafikunterstützung werden die vom Spiel benutzten Grafikbefehle abgefangen und zum Client übertragen, sodass dort das Bild an das Display angepasst gerendert wird. Der Vorteil dieser Technik ist, dass die resultierende Bitrate unabhängig von der Größe des angeschlossenen Displays ist, und somit problem-

Dipl.-Ing. **Philipp Fechteler** beschäftigt sind in der Abteilung Bildsignalverarbeitung am Fraunhofer HHI in Berlin als wissenschaftlicher Mitarbeiter mit 3D-Streaming-Techniken und verschiedenen Computer-Vision-Themen, wie 3D-Rekonstruktion, Augmented-Reality und Motion-Capture

Der Beitrag beruht auf dem Vortrag von P. Fechteler auf der 24. Fachtagung der FKTG (20. Mai 2010) in Hamburg.

Dr.-Ing. **Peter Eisert** ist Professor für Visual Computing an der Humboldt Universität zu Berlin und Leiter der Arbeitsgruppe „Computer Vision und Graphik“ am Fraunhofer HHI. Seine Schwerpunkte sind 3D-Bildanalyse und -synthese, Tracking deformierbarer und starrer Körper, 3D-Gesichtsverarbeitung sowie Augmented-Reality-(AR-)Anwendungen.

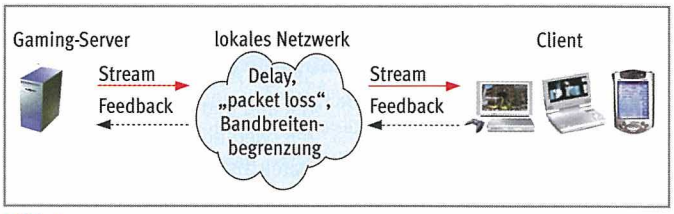

Bild 1. Gaming-on-Demand Streaming-Architektur

los auch sehr große Bildschirme artefaktfrei und ohne größere Verzögerungen angesteuert werden können. Außerdem kann der Transfer der Grafikbefehle beginnen, sobald das erste Kommando vom Spiel abgesetzt wird, bevor das eigentliche Bild gerendert wurde. Im Gegensatz zum Streamen von Video wird durch die Latenz um knapp ein Frame verringert.

Die Herausforderung der praktischen Realisierung ist jedoch, die starke Koppelung von Spiel und Grafikbibliothek aufzulösen. Besonders problematisch sind jene Kommandos, die Daten zurückgeben, da hier die Netzwerk-Transferzeit doppelt einfließt. Versuche zeigten, das für Frames mancher Spiele mehr als 1000 solcher Feedback-Kommandos auftraten. Man kann grob zwischen drei Arten von Zugriffen unterscheiden, wie Spiele-Applikationen auf Daten in der Grafikbibliothek zugreifen:

- Abfrage von statischen Eigenschaften, zum Beispiel die Anzahl der Texturing-Einheiten oder Extensions,
- Lesen/Schreiben von Zustandsinformationen, wie zum Beispiel Projektionsmatrizen und Viewport sowie
- Lesen/Schreiben von Nutzdaten, zum Beispiel Texturen und Vertex-Arrays (in den entsprechenden *Draw*()-Befehlen werden Zeiger auf Speicherbereiche abgegeben, die beim Aufruf über das Netz übertragen werden müssen).

Im **Bild 2** ist ein exemplarisches Datenaufkommen von Grafikbefehlen inklusive Argumenten pro Frame dargestellt, das zwischen Applikation und Grafikbibliothek ausgetauscht wird. Es ist deutlich zu sehen, dass beträchtliche Datenmengen zu transferieren sind, speziell zu Beginn neuer Szenen durch das Nachladen von Texturen.

Das entwickelte Streaming-Protokoll wurde im wesentlichen von *GLX (OpenGL Extension to the X Window System)* abgeleitet und überträgt die Grafikkommandos in einem Grafikbibliothek-unabhängigen Format, sodass zum Beispiel auch DirectX-Spiele auf OpenGL-Clients und vice versa gerendert werden.

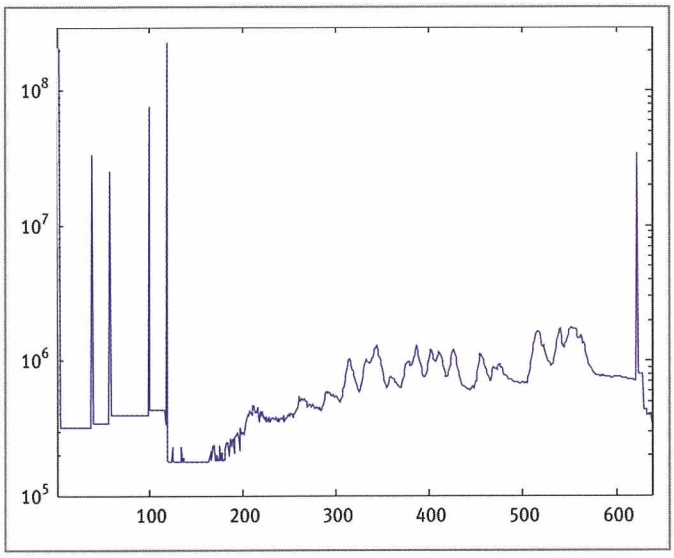

Bild 2. Bitrate in bits/frame für PenguinRacer. Hohe Bitraten korrespondieren zu Wechseln der Szene bzw. des Levels.

Um den korrekten Datenaustausch auch per Netzwerkverbindung ohne größere Verzögerungen zu gewährleisten, und somit die Reaktionszeiten des Spiels in tolerablen Schranken zu halten, wurden verschiedene Optimierungen entwickelt [2].

Einige *Draw*()-Kommandos übergeben Speicheradressen als Argumente, aus denen dann die Daten effizient ausgelesen werden. Da Server und Client nicht über gemeinsamen Speicher verfügen, müssten im Streaming-Ansatz jedes Mal die kompletten Daten übertragen werden. Da sich Texturen, Vertex-Arrays usw. nur selten ändern, reicht es, um das wiederholte Übertragen von identischen Daten zu vermeiden, serverseitig die bereits geladenen Daten zu cachen und nur Änderungen zu übermitteln. Um clientseitige Änderungen zu detektieren und dem Server zu signalisieren, werden auch clientseitig die Daten als Kopie der aktuellen Daten vorrätig gehalten. Serverseitig wird nun der Speicher aktualisiert bevor ein *Draw*()-Befehl ausgeführt wird. Auf diese Weise werden Vertex-, Normal-, Textur- usw. Arrays gecached.

Des Weiteren wird das Warten auf jegliches Feedback vom Client umgangen. Dafür wird der Zustand der clientseitigen Grafikkarte simuliert, um serverseitig nicht auf Antworten des Clients angewiesen zu sein. Zum Beispiel werden alle statischen Eigenschaften der Grafikkarte einmal erfragt, serverseitig gespeichert und fortan lokal beantwortet. Des Weiteren lassen sich die Rückgabewerte vieler Kommandos, zum Beispiel von *glGetError*(), einfach vorhersagen, sodass serverseitig weiter gerechnet werden kann, ohne die Antwort des Clients abzuwarten.

Andere Grafikzustände ändern sich häufiger, lassen sich in Software aber durch Simulation des Client-Zustands mitführen. Zum Beispiel bewirken Kommandos wie *glRotate*(), *glLoadMatrix*(), *glPushMatrix*(), usw. Änderungen an der ModelView- bzw. Projection-Matrix. Da entsprechende Befehle nicht so häufig auftreten, ist der zusätzliche Overhead vernachlässigbar.

Ein weiterer Vorteil des lokalen Simulierens des Client-Zustandes ist es, dass einige Kommandos gar nicht zum Client übertragen werden müssen. Da Computerspiele häufig Zustände einstellen, unabhängig davon wie der aktuelle Zustand ist, können Befehle die keine Änderung hervorrufen einfach ignoriert werden.

Mittels Kompression wird die benötigte Bandbreite der resultierenden Daten gering gehalten, die zum Client transferiert wird. Texturen werden ähnlich dem H.264/AVC-Videocodec per 4x4-Integer-Transformation codiert. Befehle werden durch ein bis zwei Byte-Folgen repräsentiert. Dabei werden häufiger vorkommende Befehle auf kürzere Codes abgebildet, und auch Gruppen von Befehlen werden zu einzelnen Code-Strings zusammengefasst, wenn sie gehäuft in Gruppen auftreten, zum Beispiel *glTexCoord*(), *glNormal*(), *glVertex*(). Die Datenrate lässt sich weiter verringern, indem die Argumente quantisiert und zum Beispiel Doubles als Floats und Integers als Shorts übertragen werden.

Da im Allgemeinen aufeinander folgende Frames ähnliche Inhalte aufweisen, da Objekte auf ähnliche Weise gerendert werden, weisen die Befehlssequenzen sich wiederholende Strukturen auf. Diese temporale Korrelation lässt sich zur weiteren Kompression ausnutzen. Die Methode ist dem Videocodierkonzept ähnlich, Frames im Predictive- oder Intra-Modus zu übertragen. Serverseitig werden codierte Befehlssequenzen als Referenz gespeichert, um bei folgenden P-Frames auf Teile davon per Offset und Länge zu verweisen – ähnlich den Bewegungsvektoren in der Videocodierung. Größere Befehlsfolgen können dadurch signalisiert werden, ohne sie erneut senden zu müssen.

Um effizient identische Teilabschnitte zu finden, wird ausgenutzt, dass Spiele meist auf baumartige Weise rendern (Szenen-Graph) und die Befehle bedeutungsvoll sind. Somit wird die Suche auf nur sehr wenige Befeh-

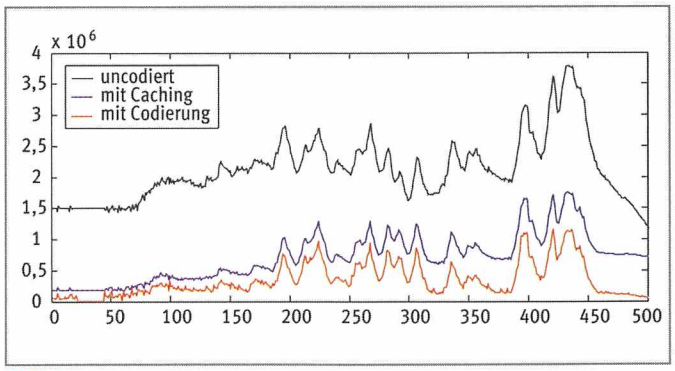

Bild 3. Vergleich der optimierten Datenraten (bits pro Frame)

le wie *glPushMatrix*() oder *glBegin*() reduziert, die das Rendern neuer Objekte einleiten.

Die durch die Optimierungen um bis zu 80 % reduzierte Datenmenge ist im **Bild 3** dargestellt.

Enhanced-Video-Streaming

Als alternativer Streaming-Ansatz wird die Bildinformation als Video übertragen, um Geräte ohne GPU ansprechen zu können oder um für kleine Bildschirmgrößen günstigere Datenraten zu erzielen. Als Videoformat wird H.264/AVC verwendet, das sehr hohe Kompressionsraten ermöglicht. Da die typischerweise sehr rechenintensive Videocodierung parallel zum Spiel abläuft, wurden verschiedene Optimierungen entwickelt, um die Rechenkomplexität zu verringern. Im Wesentlichen basieren sie darauf, Informationen aus dem Grafikkontext des Spiels mit einzubeziehen.

Um zum Beispiel bereits den Render-Prozess die resultierenden Bilder optimal auf das clientseitige Display angepasst generieren zu lassen, werden Grafikbefehle, wie *glViewport*() bzw. *SetViewport*(), modifiziert an die Grafikbibliothek weitergegeben. Auf diese Weise entstehen weder Skalierungsartefakte noch zusätzliche Verzögerungen.

Die Skybox/Skydome genannte Rendering-Technik wird in Computerspielen oft eingesetzt [9]. Hier wird bei jedem Szenen-Update als erstes eine passende Textur als statischer Szenenhintergrund gerendert, der sich je nach Perspektive immer als Ganzes verschiebt, zum Beispiel Himmel, Berge, Panorama-Ansichten usw. Das geschieht bei deaktiviertem z-Buffer, was dazu führt, dass eindeutig erkannt werden kann, welche Pixel zur Skybox gehören **(Bild 4)**. Diese zusätzliche Information lässt sich bei der H.264/AVC-Codierung nutzen [3]. Makroblock-Partitionen, die komplett im Skybox-Bereich liegen, werden nicht weiter partitioniert, da aufgrund der homogenen Bewegung des Skybox-Bereichs ein gemeinsamer Bewegungsvektor die Bewegung hinreichend gut beschreibt. Dadurch entfallen rechenintensive Tests der gängigen H.264/AVC-Encoder.

Des Weiteren lassen sich während des Renderns die entsprechenden Draw-Befehle eindeutig identifizieren, da die Skybox im allgemeinen als erstes und mit deaktiviertem z-Buffer gerendert wird. Somit lassen sich die Transformations-/Projektionsmatrizen (D3DTS_PROJECTION, D3DTS_VIEW und D3DTS_WORLD für DirectX, GL_MODELVIEW_MATRIX und GL_PROJECTION_MATRIX für OpenGL) des Skybox-Bereichs eindeutig bestimmen.

Mit dieser Information lassen sich für jede Pixelposition zusammen mit dem entsprechenden z-Buffer-Wert die Koordinaten im 3D-Szenenraum ermitteln, aus denen wiederum unter Nutzung der vorigen Transformations-/Projektionsmatrizen die Pixelposition desselben Objekts im vorherigen Frame berechnet wird. Der durch Differenzbildung berechnete Bewegungsvektor wird direkt H.264/AVC-codiert, wodurch die sehr rechenintensive generische Bewegungsvektorsuche komplett entfällt (s. Bild 4).

Im Gegensatz zu [6], wo für jeden Bewegungsvektor die GLU-Befehle *gluProject*() und *gluUnProject*() benutzt werden, lassen sich hier große Teile einmal vorab pro Frame für alle Bewegungsvektoren in einer Matrix zusammenfassen. Dadurch entfallen pixelweise Multiplikationen und Inversionen von Matrizen ebenso wie die Normalisierung der Koordinaten und Differenzbildung. Die Berechnung eines Bewegungsvek-

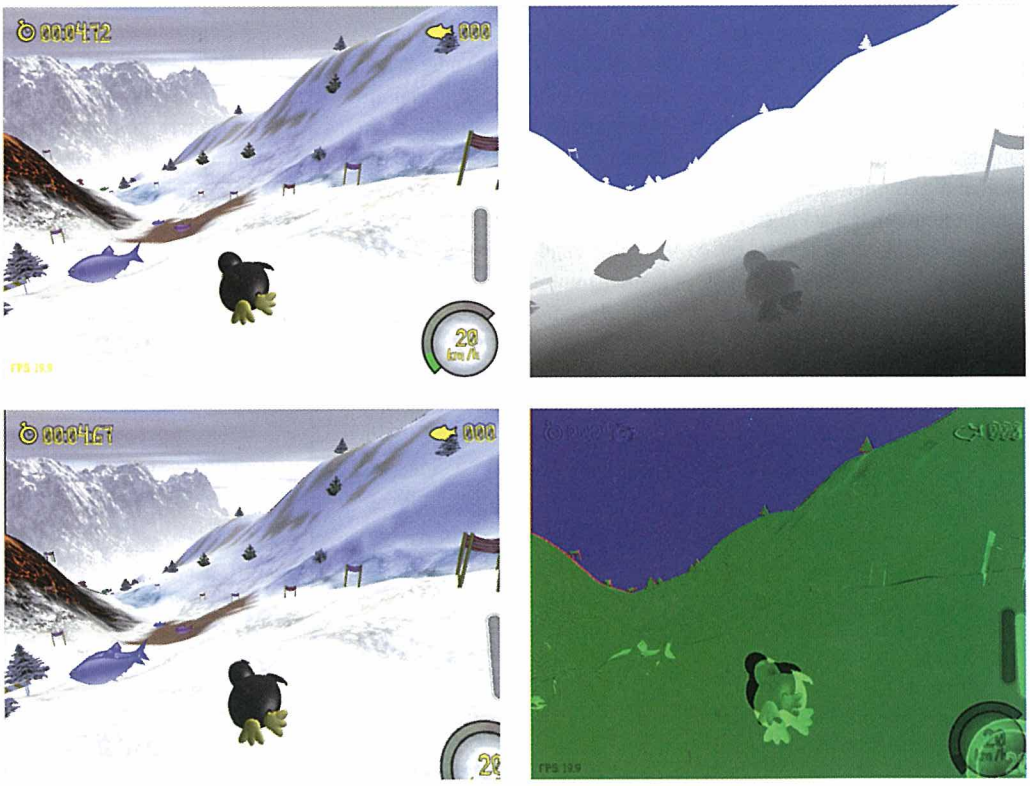

Bild 4. Prädiktion basierend auf Render-Kontext für das OpenGL-Spiel „Extreme Tux Racer" – oben links: Original Framebuffer, oben rechts: Korrespondierender z-Buffer mit blau gefärbten Skyboxbereich, unten links: Pixelweise Prädiktion aus vorangegangenem Frame, unten rechts: Differenzbild zwischen Original Framebuffer und Prädiktion, Abweichungen vom mittleren grün/blau bedeuten größere Fehler, im roten Bereich ist keine Prädiktion verfügbar.

Bild 5. Originalausgabe des DirectX-Spiels „Total Overdose" (links) und die extrahierte Szeneninformation (rechts)

tors für eine Pixelposition reduziert sich somit auf eine 3x4-Matrix-Multiplikation mit einem 4-Vektor.

Auf die gleiche Weise lassen sich die Bewegungsvektoren für einen Großteil der restlichen Szenen direkt berechnen [4]. Dazu werden während des Renderns die Transformations-/Projektionsmatrizen aller gerenderten Objekte abgefangen und mit den entsprechenden Matrizen desselben Objekts im vorangegangen Frame in Beziehung gesetzt (s. Bild 4).

Da pro Szenen-Update viele Objekte mit unterschiedlichen Matrizen gerendert werden und sowohl Reihenfolge als auch Anzahl von Objekten variieren können, gibt es ein Korrespondenzproblem: Welche Matrizen des aktuellen Frames korrespondieren zu welchen Matrizen des vorangegangen Frames.

In Versuchen hat sich gezeigt, dass ein Transformations-Projektionsmatrizenpaar für deutlich mehr Objekte zum Rendern benutzt wird und somit für den Großteil der Szene verantwortlich ist, nämlich den Szenehintergrund **(Bild 5)**. Durch Einsammeln dieser Matrizen in Listen – sortiert nach Objektanzahl – lässt sich die Szenenmatrix extrahieren **(Bild 6)**.

In komplexeren Fällen können weitere Information zur Lösung des Korrespondenzproblems herangezogen werden, zum Beispiel der Typ des *Draw*()-Befehls, die Textur-ID oder Zeiger des Vertex-Puffers. Um weitere Bewegungsvektorkandidaten zu berechnen, können mehrere Matrizen als nur die eine meistgenutzte benutzt werden.

Die benötigten Tiefenkarten können unter OpenGL einfach mit *glReadPixels*() ausgelesen werden. Unter DirectX hat sich die äquivalente Variante als extreme ineffizient erwiesen. Deshalb werden während des Renderns alle *Draw*()-Befehle, die einerseits den z-Buffer modifizieren und andererseits nicht transparent sind (alpha-blending), in einer Liste gesammelt. Bei jedem Szenen-Update wird eine passende Tiefenkarte mit dieser Liste und in ein extra Render-Target gerendert. Da lediglich die Tiefenkarte zusätzlich benötigt wird, und die Fixed-Function-Pipeline von DirectX9 dafür keine Möglichkeiten bietet, wird mit einem einfachen Shader-Programm die Tiefenkarte erzeugt.

Im **Bild 7** sind typische Zeiten für das Auslesen von SVGA-Tiefenkarten dargestellt. Wie zu sehen ist, lässt sich mit OpenGL die Tiefenkarte recht zuverlässig in rund 1,1 ms auslesen. Ein Befehl reicht in OpenGL aus, um die Tiefenkarte in einem gewünschten Format auszulesen, hier 16 bit Integer. Unter DirectX9 sind dafür einige Befehle nötig und die Auswahl des Datenformats ist auf derselben Hardware wesentlich eingeschränkter, hier 32 bit Float. Im Bild 7 ist zu sehen, dass das Rendern der Tiefenkarte im Schnitt 0,1 ms und somit das gesamte Tiefenkarten-Capturen im Mittel rund 3 ms dauert, wohl bemerkt bei doppelter Datenmenge.

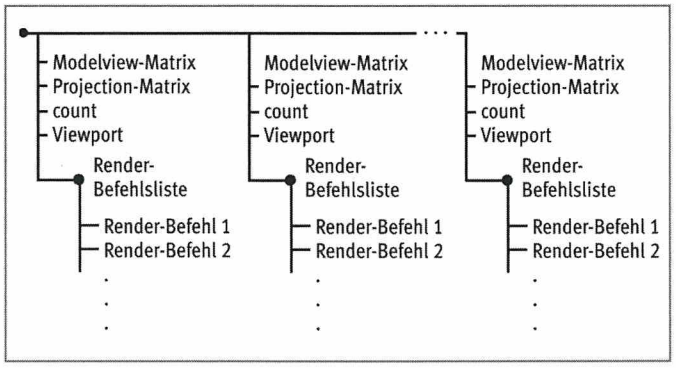

Bild 6. Liste eingesammelter Matrizen der Render-Befehle

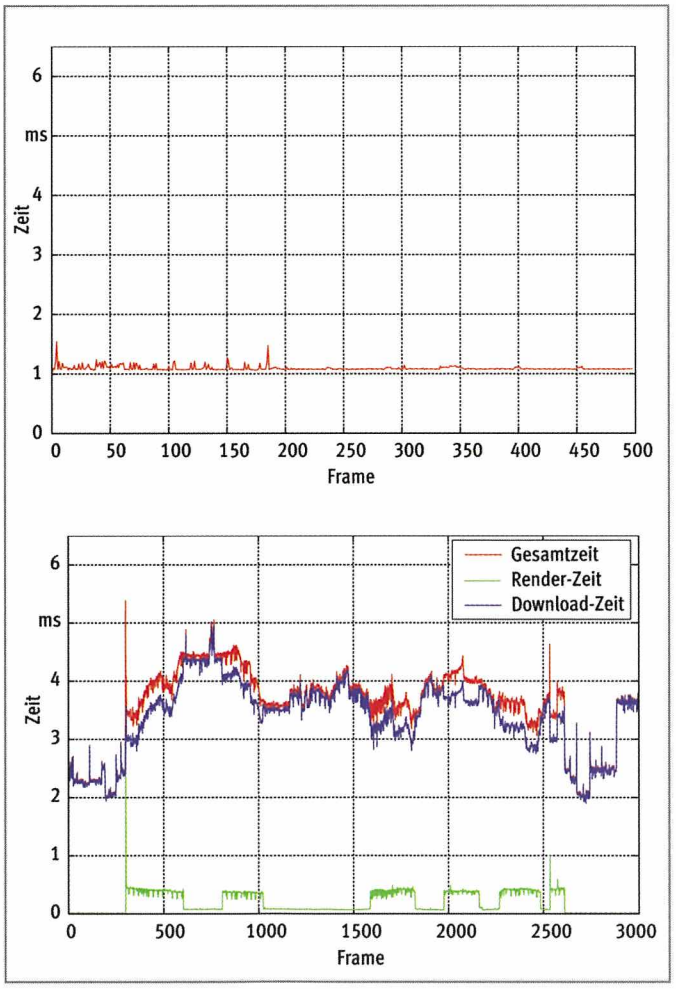

Bild 7. Typische Auslesezeiten der Tiefenkarte

Zur weiteren Beschleunigung wurden GPU-basierte Optimierungen integriert. Da zum Beispiel der Framebuffer im RGB-4:4:4-Format vorliegt, aber der H.264/AVC-Codierer YC_bC_r-4:2:0 als Input erwartet, findet eine Farbkonvertierung inklusive Unterabtastung statt. Durch effiziente Integration auf die GPU (ähnlich wie [8]) wird einerseits die hochgradige Parallel-Verarbeitung ausgenutzt und andererseits die Datenmenge reduziert, die von der Graphikkarte zur CPU transferiert wird.

Da maximal ein Bewegungsvektor pro Macroblock-Partition berechnet wird, ist es ausreichend, eine um den Faktor 16 reduzierte Tiefenkarte für die Berechnungen zu benutzen. Mit GPU-Techniken kann dafür die Tiefenkarte auf der GPU effizient unterabgetastet werden, sodass pro 4x4-Makroblock-Partition nur ein Tiefenpixel generiert wird. In Versuchen hat sich gezeigt, dass die Transferzeit von 1,1 ms für reguläre Tiefenkarten auf rund 0,12 ms für unterabgetastete Tiefenkarten reduziert wird.

Alternativ können die Berechnungen der Bewegungsvektoren auch direkt auf der GPU ausgeführt werden. Experimente haben gezeigt, dass das Ausnutzung der GPU-Parallelität zusammen mit der reduzierten Datenmenge für die Bewegungsvektoren, die nun mehr von der Grafikkarte in den Hauptspeicher geladen werden muss, Beschleunigungen bis zu einem Faktor 10 ergeben.

In manchen Fällen liefert die direkte Prädiktion unzureichende Ergebnisse, zum Beispiel bei Aufdeckungen, 2D-Projektionen (Punktestand, Tachometer usw.) oder Objektveränderung (Bewegung von Figuren oder ähnliches). Per Vergleich der Rate-Distortion-Kosten wird in diesen ungünstigen Fällen auf die generische H.264/AVC-Bewegungsvektorsuche zurückgefallen und somit das Degradieren der Bildqualität verhindert.

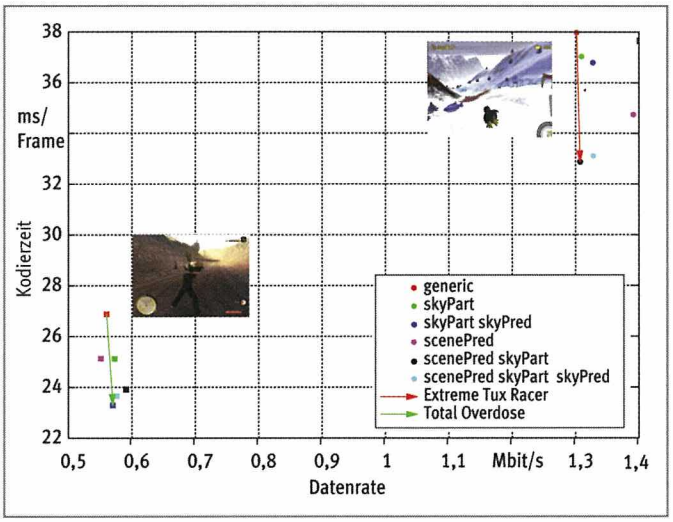

Bild 8. Einfluss der Optimierungen auf Datenrate und Codierzeit bei einer Bildqualität von 35 dB

Da neben der Datenrate bei vorgegebener PSNR-Bildqualität die Codierzeit die Echtzeitfähigkeit bestimmt, wurde in Experimenten mit verschiedenen Quantisierungsstufen getestet und sowohl die Datenrate als auch die Codierzeit für eine PSNR-Bildqualität von 35 dB interpoliert. Im **Bild 8** wird für die beiden Spiele „Total Overdose" (DirectX) und „Extreme Tux Racer" (OpenGL) bei SVGA-Auflösung (800 × 600) der Einfluss der Optimierungen gezeigt. Man sieht deutlich, dass die Datenrate nur minimal zunimmt, während die Codierzeit hingegen drastisch abnimmt.

Aktueller Status und Ausblick

Das „Games@Large"-System wird momentan in Privathaushalten und öffentlich-zugänglichen Installationen, unter anderem in einem Internet-Cafe, einem Hotel und einem Altersheim ausgiebig getestet. Neben der reinen Spiel-Performance [7] wird auch die Einsatztauglichkeit überprüft und gegebenenfalls verbessert, wie zum Beispiel die Skalierbarkeit, die Zugangskontrolle und die Benutzerfreundlichkeit.

Die vorgestellte Plattform zum interaktiven Streamen von 3D-Computerspielen unterstützt vielfältigste Endgeräte. Graphics-Streaming ermöglicht Endgeräten mit GPU durch das direkte Übertragen der Grafikbefehle beliebig große Displays anzuspielen, ohne die Datenrate zu beeinflussen. Mit den vorgestellten Optimierungen konnte die Datenrate um 80 % gesenkt werden, sodass flüssiges Spielen gewährleistet ist. Für den alternativen Ansatz – Video-Streaming zur Unterstützung von Endgeräte ohne GPU – wurden Methoden zur Beschleunigung präsentiert, die unter Ausnutzung des Grafikkontextes im Mittel 25 % der Rechenkomplexität einsparen.

Schrifttum

[1] Jurgelionis, A.; et. al.: Platform for Distributed 3D Gaming. International Journal of Computer Games Technology, Vol. 2009, Article ID 231863.

[2] Eisert, P.; Fechteler, P.: Low Delay Streaming of Computer Graphics. Proceedings of the 15th International Conference on Image Processing (ICIP2008), San Diego, California, USA, 12. bis 15. Oktober 2008, S. 2704 bis 2707.

[3] Fechteler, P.; Eisert, P.: Depth Map Enhanced Macroblock Partitioning for H.264 Video Coding of Computer Graphics Content. Proceedings of the 16th International Conference on Image Processing (ICIP2009), Kairo, Ägypten, 7. bis 10. November 2009, S. 3441 bis 3444.

[4] Fechteler, P.; Eisert, P.: Accelerated Video Encoding Using Render Context Information. Proceedings of the 17th International Conference on Image Processing (ICIP2010), Hong Kong, China, 26. bis 29. September 2010.

[5] Stegmaier, S.; Diepstraten, J.; Weiler, M.; Ertl, T.: Widening the Remote Visualization Bottleneck. Proceedings of the International Symposium on Image and Signal Processing and Analysis (ISPA2003), Rom, Italien, 18. bis 20. September 2003, Vol. 1, S. 74 bis 179.

[6] Cheng, L.; Bhushan, A.; Pajarola, R.; El Zarki, M.: Real-time 3D Graphics Streaming Using MPEG-4. Proceedings of IEEE/ACM Workshop on Broadband Wireless Services and Applications (BroadWISE2004), 2004.

[7] Jurgelionis, A.; et. al.: Testing cross-platform streaming of video games over wired and wireless LANs. Proceedings of 1st International Workshop on Networking and Games (N&G2010), Perth, Australien, 20. bis 23. April 2010.

[8] Rijsselbergen, D. V.: YCoCg(-R) Color Space Conversion on the GPU. 6th UGent-FirW Doctoraatssymposium, 2005.

[9] Dalmau, D. S.-C.: Core Techniques and Algorithms in Game Programming. New Rider Games, September 2003.

VDS: Sapphire V5 mit neuen Effekten für Quantel-Systeme

VDS rüstet das Plug-in-Set Sapphire von GenArts in der Version 5 um 15 neue Effekte auf. Das Upgrade für die Geräte der Q-Serie von Quantel enthält neben einer GPU-Beschleunigung die Funktionen TVDamage, TVChannelChange, Technicolor2Strip, Technicolor3Strip, Swish3D, WipePlasma, WipePointalize, WipeWeave, WipeMoire, DissolveLensFlare, DissolveEdgeRays, DissolveGlint, DissolveGlint Rainbow, DissolveDefocus, DissolveTiles und DissolveDistort.
www.videodesignsoftware.com