

Effizientes Streamen interaktiver 3D Computer Spiele

In diesem Artikel stellen wir eine Plattform zum Streamen von interaktiven 3D Computer Spielen vor. Der wesentliche Fokus liegt auf der performanten Unterstützung unterschiedlichster Endgeräte. Um die hohen Anforderungen an die Reaktionszeiten zu realisieren, wurden zwei Streaming Methoden entwickelt. Beim Graphics Streaming werden direkt die Graphik Befehle an Endgeräte mit Graphik-Prozessor übertragen, um dort das Bild dem Display angepasst zu rendern. Dadurch ist die Datenrate von der Bildschirmauflösung unabhängig. Um die enge Kopplung zwischen Spiel und Grafik-Kontext aufzubrechen, wurden verschiedene Optimierungen entwickelt, wie intelligentes Caching, Kodierung und Emulation des Graphik-Kontextes. Dadurch konnte die Datenrate um 80% reduziert werden. Alternativ für Endgeräte ohne GPU wird der visuelle Output als Video kodiert übertragen. Um Interaktivität in Echtzeit zu unterstützen, wurde der sonst sehr rechenintensive Videokodierprozess optimiert. Basierend auf zusätzlichen Informationen aus dem Render-Kontext werden direkt Bewegungsvektoren berechnet und Makroblöcke partitioniert. Im Durchschnitt werden dabei Beschleunigungen von rund 25% erzielt.

In this article, we present a streaming platform for interactive 3D Computer gaming. The main focus is to fluently support a wide range of types of end devices. Two streaming methods have been developed in order to fulfill the needed very low response time. Graphics streaming is used to stream the graphics commands directly to end devices with a graphics processor. The image is rendered there optimally for the connected display. In this way the data rate is independent of the screen resolution. In order to open the tight coupling between game and render context, several optimizations have been developed, e.g. intelligent caching, entropy-coding and local emulation of graphic context. These optimizations lead to bit rate savings of 80%. Alternatively, for end devices without GPU, the visual output is streamed encoded as video. In order to achieve interactivity in real-time, the usually demanding encoding process has been optimized. Based on additional information available in the render context, motion vectors and macroblock partitions are calculated directly. On average accelerations around 25% are achieved.

Framework

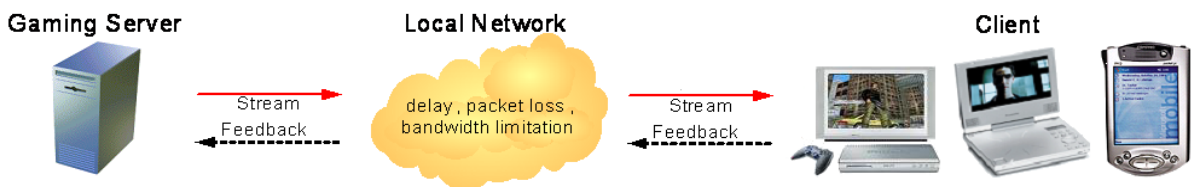


Abbildung 1: Gaming on Demand Streaming Architektur

Das Fraunhofer HHI Institut entwickelt zusammen mit 13 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 etc.

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 [G@L]. Anwendungsbereiche ergeben sich überall dort, wo man einerseits nicht vielfach in teure Computer Hardware investieren möchte und andererseits den Software-Wartungsaufwand gering halten möchte, beispielsweise in Hotels, Internet-Cafés, Altersheimen etc. Durch überladen der Graphik Bibliothek [Stegmaier03] unterstützt die Plattform kommerzielle, handelsübliche Spiele ohne jegliche Anpassung. Client-seitig ist keine lokale Installation nötig, da das Spiel auf dem Server ausgeführt wird. Um viele (kostengünstige) Clients (z.B. im Hotel) gleichzeitig zu bedienen, können mehrere Instanzen beispielsweise in einer Server-Farm parallel ausgeführt werden.

Ein Schwerpunkt 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 Spiele-Server 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 (siehe **Abbildung 1**). Um die Netzwerk-basierten 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 Spiel-Spaß auswirken. Dies ist je nach Spielart und Genre unterschiedlich stark. In der Praxis hat sich gezeigt, dass im Allgemeinen Verzögerungen von bis zu 100 msec 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 Graphikbefehle des Spiels zum Client übertragen und dort gerendert. Alternativ wird das Bild Server-seitig gerendert, videokodiert, zum Endgerät übertragen und dort dekodiert dargestellt.

Graphics Streaming

Für Endgeräte mit Hardware-seitiger Grafikerunterstützung werden die vom Spiel benutzten Grafikbefehle abgefangen und zum Client übertragen, so dass 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 problemlos auch sehr große Bildschirme Artefakt-frei und ohne größere Verzögerungen angesteuert werden können. Außerdem kann der Transfer der Grafik-Befehle beginnen, sobald das erste Kommando vom Spiel abgesetzt wird, bevor das eigentliche Bild gerendert wurde. Im Gegensatz zum Streamen von Video wird die Latenz um knapp ein Frame verringert.

Die Herausforderung der praktischen Realisierung ist jedoch die starke Koppelung von Spiel und Grafik Bibliothek aufzulösen. Besonders problematisch sind jene Kommandos, die Daten zurückgeben, da hier die Netzwerk-Transfer-Zeit 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 Grafik Bibliothek zugreifen:

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

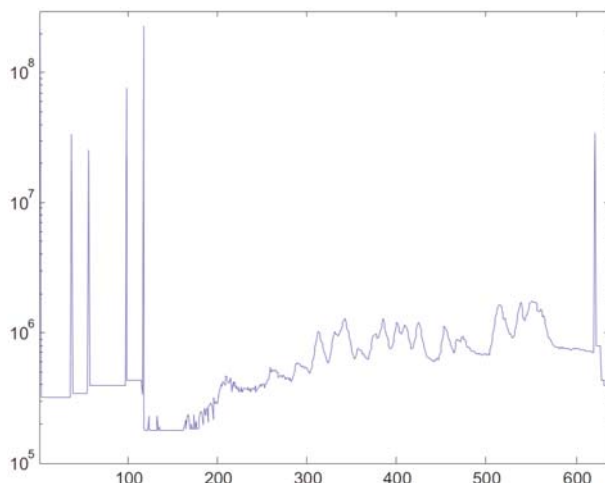


Abbildung 2: Bitrate in bits/frame für PenguinRacer. Hohe Bitraten korrespondieren zu Wechseln der Szene bzw. des Levels.

In **Abbildung 2** ist ein exemplarisches Datenaufkommen von Grafikbefehlen inkl. Argumenten pro Frame dargestellt, das zwischen Applikation und Grafik-Bibliothek ausgetauscht wird. Es ist deutlich zu sehen, dass beträchtliche Datenmengen zu transferieren sind, speziell zu Beginn neuer Szenen durch das Nachladen von Texturen.

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 [Eisert08].

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 etc. nur selten ändern, reicht es, um das wiederholte Übertragen von identischen Daten zu vermeiden, Server-seitig die bereits geladenen Daten zu cachen und nur Änderungen zu übermitteln. Um Client-seitige Änderungen zu detektieren und dem Server zu signalisieren, werden auch Client-seitig die Daten als Kopie der aktuellen Daten vorrätig gehalten. Server-seitig wird nun der Speicher aktualisiert bevor ein *Draw()*-Befehl ausgeführt wird. Auf diese Weise werden Vertex-, Normal-, Textur- etc. Arrays gecached.

Des Weiteren wird das Warten auf jegliches Feedback vom Client umgangen. Dafür wird der Zustand der Client-seitigen Grafikkarte simuliert um Server-seitig nicht auf Antworten des Clients angewiesen zu sein. Beispielsweise werden alle statischen Eigenschaften der Grafikkarte einmal erfragt, Server-seitig gespeichert und werden fort an lokal beantwortet. Des Weiteren lassen sich die Rückgabewerte vieler Kommandos, beispielsweise von *glGetError()*, einfach vorhersagen, so dass Server-seitig weiter gerechnet werden kann, ohne die Antwort des Clients abzuwarten. Andere Grafik Zustände ändern sich häufiger, lassen sich in Software aber durch Simulation des Client-Zustandes mitführen. Beispielsweise bewirken Kommandos wie *glRotate()*, *glLoadMatrix()*, *glPushMatrix()*, etc. Änderungen an der ModelView bzw. Projection Matrix. Da entsprechende Befehle nicht so häufig auftreten, ist der zusätzliche Overhead vernachlässigbar. Ein weiterer Vorteil des lokalen Simulieren des Client-Zustandes ist es, dass einige Kommandos gar nicht zum Client übertragen werden müssen. Da Computer Spiele 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 Video Codec per 4x4 Integer Transformation kodiert. 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, beispielsweise *glTexCoord()*, *glNormal()*, *glVertex()*. Die Datenrate lässt sich weiter verringern, indem die Argumente quantisiert werden, beispielsweise Doubles als Floats und Integers als Shorts übertragen werden.

Da im Allgemeinen aufeinander folgende Frames ähnliche Inhalte aufweisen, weil Objekte auf ähnliche Weise gerendert werden, weisen die Befehls-Sequenzen sich wiederholende Strukturen auf. Diese temporale Korrelation lässt sich zur weiteren Kompression ausnutzen. Die Methode ist dem Videokodierkonzept ähnlich, Frames im Predictive-Modus oder Intra-Modus zu übertragen. Server-seitig werden codierte Befehls-Sequenzen als Referenz gespeichert, um bei folgenden P-Frames auf Teile davon per Offset und Länge zu verweisen – ähnlich dem Bewegungsvektoren in der Video Kodierung. Größere Befehlsfolgen können dadurch signalisiert werden, ohne sie erneut senden zu müssen. Um effizient identische Teilabschnitte zu finden, wird ausgenutzt, dass Spiele meist auf Baum-artigen Weise rendern (Szenen-Graph) und die Befehle bedeutungsvoll sind. Somit wird die Suche auf nur sehr wenige Befehle wie *glPushMatrix()* oder *glBegin()* reduziert, die das Rendern neuer Objekte einleiten.

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

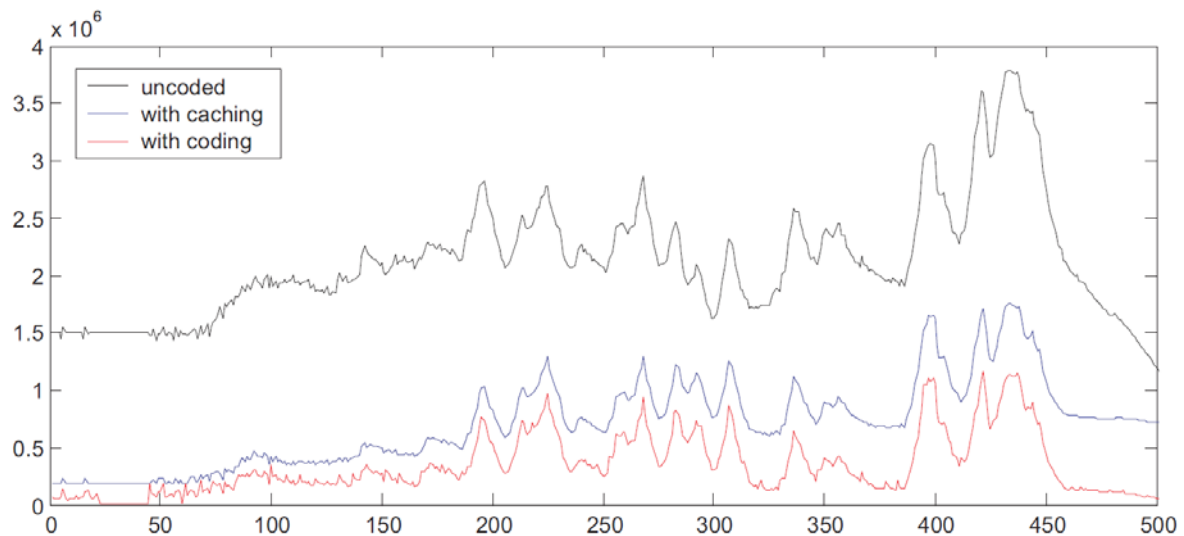


Abbildung 3: Vergleich der optimierten Datenraten (Bits pro Frame)

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 Video Format wird H.264/AVC verwendet, das sehr hohe Kompressionsraten ermöglicht. Da die typischerweise sehr rechenintensive Video-Kodierung parallel zum Spiel abläuft, wurden verschiedene Optimierungen entwickelt, um die Rechenkomplexität zu verringern. Im Wesentlichen basieren sie darauf, Informationen aus dem Grafik-Kontext des Spiels mit einzubeziehen. Um beispielsweise bereits den Render-Prozess die resultierenden Bilder optimal auf das Client-seitige Display angepasst generieren zu lassen, werden Grafik-Befehle, wie `glViewport()` bzw. `SetViewport()`, modifiziert an die Grafik-Bibliothek weitergegeben. Auf diese Weise entstehen weder Skalierungs-Artefakte noch zusätzliche Verzögerungen.

Die Skybox/Skydome genannte Rendering Technik kommt in Computerspielen oft zum Einsatz. Hier wird bei jedem Szenen-Update als erstes eine passende Textur als statischer Szenen-Hintergrund gerendert der sich je nach Perspektive immer als Ganzes verschiebt, beispielsweise Himmel, Berge, Panorama-Ansichten etc. Dies geschieht bei deaktiviertem z-Buffer, was dazu führt, dass eindeutig erkannt werden kann, welche Pixel zur Skybox gehören (siehe **Abbildung 7**). Diese zusätzliche Information lässt sich bei der H.264/AVC Kodierung nutzen [Fechteler09]. Makroblock Partitionen, welche komplett im Skybox Bereich liegen, werden nicht weiter partitioniert, da auf Grund der homogenen Bewegung des Skybox Bereiches 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-/Projektionsmatrizen¹ des Skybox-Bereiches eindeutig bestimmen. Mit dieser Information lassen sich für jede Pixelposition zusammen mit dem entsprechenden z-buffer Wert die Koordinaten im 3D Szenen-Raum ermitteln, aus denen wiederum unter Nutzung der vorigen Transformations-/Projektionsmatrizen die Pixelposition

¹ D3DTS_PROJECTION, D3DTS_VIEW und D3DTS_WORLD für DirectX, GL_MODELVIEW_MATRIX und GL_PROJECTION_MATRIX für OpenGL

desselben Objekts im vorherigen Frame berechnet wird. Der durch Differenzbildung berechnete Bewegungsvektor wird direkt H.264/AVC kodiert, wodurch die sehr rechenintensive generische Bewegungsvektorsuche komplett entfällt (siehe **Abbildung 7**).

Im Gegensatz zu [Cheng04], 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 berechnen. Dadurch entfallen Multiplikationen und Inversionen von Matrizen ebenso wie die Normalisierung der Koordinaten und Differenzbildung. Die Berechnung eines Bewegungsvektors für eine Pixelposition reduziert sich somit auf eine 3x4 Matrix Multiplikation mit einem 4-Vektor.



Abbildung 4: Originalausgabe des DirectX Spiels Total Overdose (links) und die extrahierte Szeneninformation (rechts)

Auf die gleiche Weise lassen sich die Bewegungsvektoren für einen Großteil der restlichen Szenen direkt berechnen [Fechteler10]. Dazu werden während des Renderns die Transformations-/Projektionsmatrizen aller gerenderten Objekte abgefangen und mit den entsprechenden Matrizen desselben Objekts im vorangegangenen Frame in Beziehung gesetzt (siehe **Abbildung 7**). 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 vorangegangenen 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 Szene-Hintergrund (siehe **Abbildung 4**). Durch Einsammeln dieser Matrizen in Listen sortiert nach Objekt-Anzahl, lässt sich die Szenen-Matrix extrahieren (siehe **Abbildung 5**). In komplexeren Fällen können weitere Information zur Lösung des Korrespondenzproblems herangezogen werden, beispielsweise 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 meist-genutzte benutzt werden.

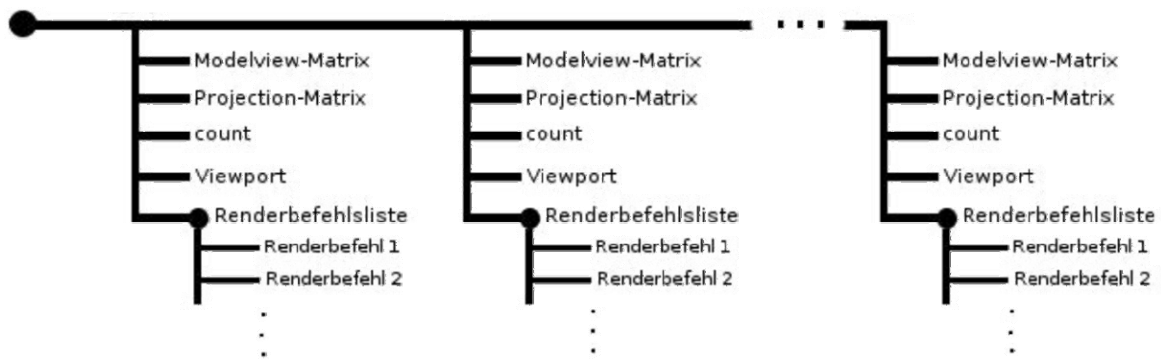


Abbildung 5: Liste eingesamelter Matrizen der Render-Befehle

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 benötigt wird, und die fixed-function-pipeline von DirectX9 dafür keine Möglichkeiten bietet, wird mit einem einfachen Shader Programm die Tiefenkarte erzeugt. In **Abbildung 6** 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 msec 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. In **Abbildung 6** ist zu sehen, dass das Rendern der Tiefenkarte im Schnitt 0.1 msec und somit das gesamte Tiefenkarten-Capturen im Mittel rund 3 msec dauert, wohl bemerkt bei doppelter Datenmenge.

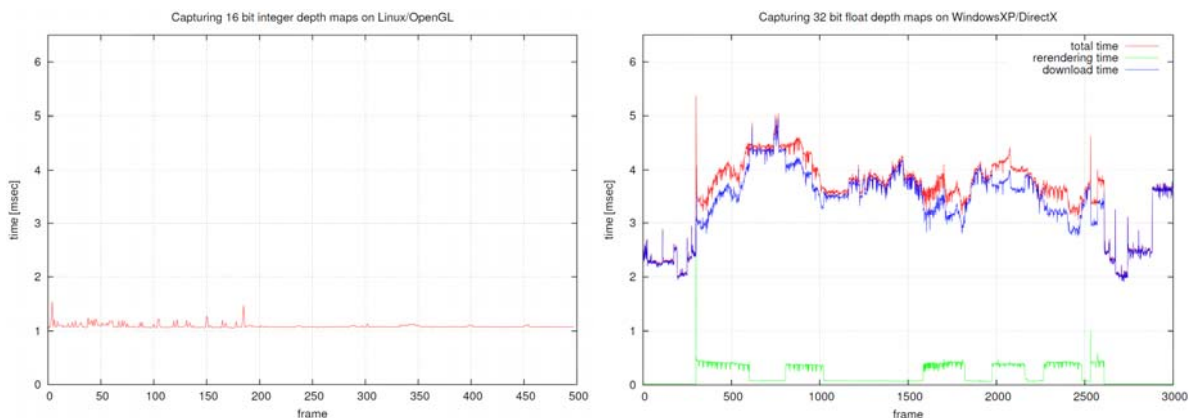


Abbildung 6: Typische Auslese-Zeiten der Tiefenkarte.

Zur weiteren Beschleunigung lassen sich GPU-basierte Optimierungen integrieren. Da beispielsweise der Framebuffer im RGB 4:4:4 Format vorliegt, aber der H.264/AVC Kodierer YCbCr 4:2:0 als Input erwartet, findet eine Farbkonvertierung inklusive Unterabtastung statt. Durch effiziente Integration auf die GPU (ähnlich wie [Rijsselbergen05]) 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, so dass pro 4x4 Makroblock Partition nur ein Tiefen-Pixel generiert

wird. In Versuchen hat sich gezeigt, dass die Transferzeit von 1.1 msec für reguläre Tiefenkarten auf rund 0.12 msec für unterabgetastete Tiefenkarten reduziert wurde.

Alternativ können die Berechnungen der Bewegungsvektoren auch direkt auf der GPU ausgeführt werden. Experimente haben gezeigt, dass die 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, beispielsweise bei Aufdeckungen, 2D-Projektionen (z.B. Punktestand, Tachometer etc.) oder Objektveränderung (z.B. Bewegung von Figuren). 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.

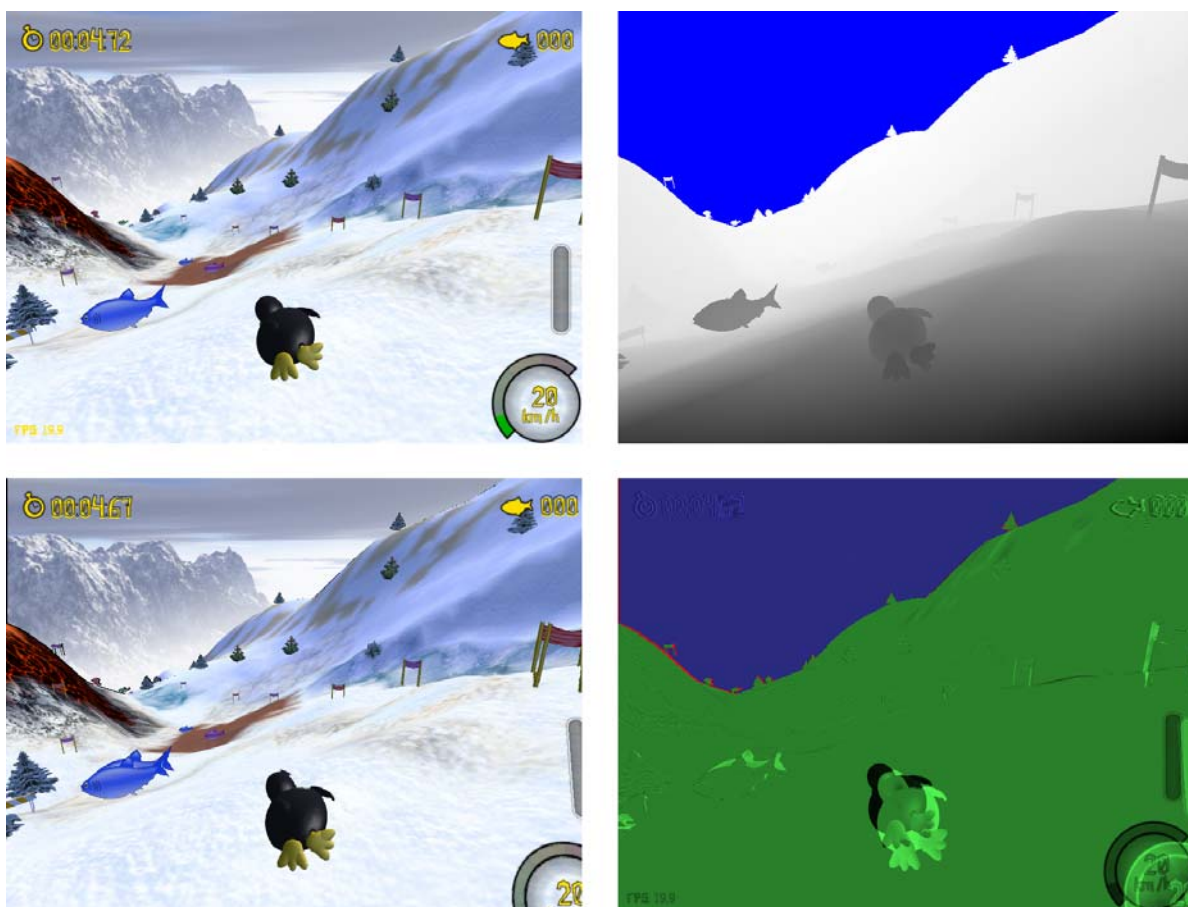


Abbildung 7: Prädiktion basierend auf Render-Kontext für das OpenGL Spiel Extreme Tux Racer – oben links: Original Framebuffer, oben rechts: Korrespondierender z-buffer mit blau gefärbten Skyboxbereich, unten links: Pixel-weise Prädiktion aus vorangegangenem Frame, unten rechts: Differenzbild zwischen original Framebuffer und Prädiktion, Abweichungen vom mittleren grün/blau bedeuten größere Fehler, im roten Bereich ist keine Prädiktion verfügbar

Da neben der Datenrate bei vorgegebener PSNR Bildqualität die Kodierzeit die Echtzeitfähigkeiten bestimmen, haben wir in unseren Experimenten mit verschiedenen Quantisierungsstufen getestet und sowohl die Datenrate als auch die Kodierzeiten für eine PSNR Bildqualität von 35 dB interpoliert. In **Abbildung 8** wird für die beiden

Spiele Total Overdose (DirectX) und Extreme Tux Racer (OpenGL) bei SVGA (800x600) Auflösung der Einfluss der Optimierungen gezeigt. Man sieht deutlich, dass die Datenrate nur minimal zunimmt, während hingegen die Kodier-Zeit drastisch abnimmt.

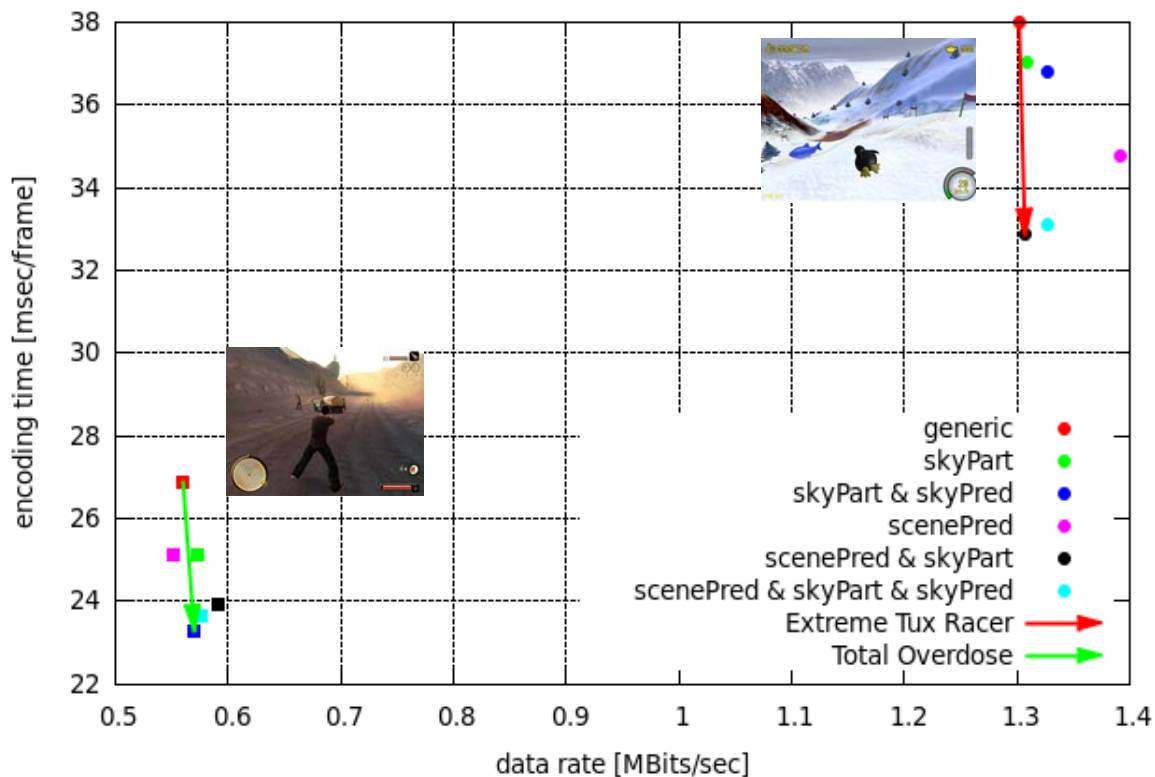


Abbildung 8: Einfluss der Optimierungen auf Datenrate und Kodier-Zeit bei einer Bildqualität von 35dB

Aktueller Status und Ausblick

Das Games@Large System wird momentan in Privathaushalten und öffentlich-zugänglichen Installationen, unter anderem in einem Internet-Café, einem Hotel und einem Altersheim ausgiebig getestet. Neben der reinen Spiel-Performance [Jurgelionis10] wird auch die Einsatztauglichkeit überprüft und ggf. verbessert, beispielsweise Skalierbarkeit, Zugangskontrolle und Benutzerfreundlichkeit.

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

Literatur

[G@L] A. Jurgelionis et al, "Platform for Distributed 3D Gaming", in *International Journal of Computer Games Technology*, Vol. 2009, Article ID 231863

[Eisert08] P. Eisert and P. Fechteler, "Low Delay Streaming of Computer Graphics", in *Proceedings of the 15th International Conference on Image Processing (ICIP2008)*, San Diego, California, USA, 12-15th October 2008, pp. 2704-2707.

[Fechteler09] P. Fechteler and P. Eisert, “**Depth Map Enhanced Macroblock Partitioning for H.264 Video Coding of Computer Graphics Content**”, in *Proceedings of the 16th International Conference on Image Processing (ICIP2009)*, Cairo, Egypt, 7-10th Nov. 2009, pp. 3441-3444.

[Fechteler10] P. Fechteler and P. Eisert, “**Accelerated Video Encoding Using Render Context Information**”, in *Proceedings of the 17th International Conference on Image Processing (ICIP2010)*, Hong Kong, China, 26-29th Sept. 2010.

[Stegmaier03] S. Stegmaier, J. Diepstraten, M. Weiler and T. Ertl, “**Widening the Remote Visualization Bottleneck**”, in *Proceedings of the International Symposium on Image and Signal Processing and Analysis (ISPA2003)*, Rome, Italy, 18-20th Sept. 2003, vol. 1, pp. 74– 179.

[Cheng04] L. Cheng, A. Bhushan, R. Pajarola and Magda El Zarki, “**Real-time 3D Graphics Streaming Using MPEG-4**”, in *Proceedings of IEEE/ACM Workshop on Broadband Wireless Services and Applications (BroadWISE2004)*, 2004.

[Jurgelionis10] A. Jurgelionis et al. “**Testing cross-platform streaming of video games over wired and wireless LANs**”, in *Proceedings of 1st International Workshop on Networking and Games (N&G2010)*, Perth, Australia, 20th – 23rd April 2010.

[Rijsselbergen05] D.V. Rijsselbergen. “**YCoCg(-R) Color Space Conversion on the GPU**”, in 6th UGent-FirW Doctoraatssymposium, 2005.



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



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 Anwendungen.