Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3194

Analyse der Echtzeitfähigkeit und des Ressourcenverbrauchs von OpenGL ES 2.0

Armin Cont

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Stephan Schnitzer
Externer Betreuer:	Dipl.-Inf. Simon Gansel (Daimler AG)
begonnen am:	16. Juni 2011
beendet am:	16. Dezember 2011
CR-Klassifikation:	I.3.4, I.3.6, C.4

Kurzfassung

OpenGL ES 2.0 (Open Graphics Library for Embedded Systems 2.0) ist eine Schnittstelle zur Entwicklung von 2D- und 3D-Computergrafik-Anwendungen. Die Spezifikation von OpenGL ES 2.0 definiert eine Reihe von Befehlen, mit denen Daten zum und vom OpenGL ES-System übermittelt werden können, mit denen das Zeichnen von Grafiken angestoßen werden kann (Rendering) und Einstellungen für das Rendering durchgeführt werden können. Üblicherweise verwenden OpenGL ES-Systeme für das Rendering physische Grafikkarten (GPUs). Keines der heute verfügbaren OpenGL ES-Systeme mit physischer GPU unterstützt aber die Priorisierung von Anwendungen hinsichtlich der Ausführung von OpenGL ES-Befehlen oder Einschränkungen von Anwendungen hinsichtlich der Nutzung von GPU-Ressourcen. Insbesondere bietet OpenGL ES weder einen konfigurierbaren Scheduler noch die Möglichkeit, Echtzeitgarantien für die Ausführung von OpenGL ES-Befehlen zu erfüllen. Ziel dieser Arbeit ist es, zu untersuchen, inwieweit dennoch sichergestellt werden kann, dass Befehle sicherheitskritischer Anwendungen rechtzeitig ausgeführt werden können. Dazu werden relevante Befehle bestimmt, deren Laufzeitverhalten und Ressourcenverbrauch analysiert wird. Außerdem werden spezielle Szenarien untersucht, um festzustellen, inwiefern das Verhalten von OpenGL ES-Systemen die rechtzeitige Ausführung kritischer Befehle verhindern kann. Schließlich werden Untersuchungsmethoden und Metriken für die Prognose des Ressourcenverbrauchs von OpenGL ES-Befehlen und die Ermittlung der dafür notwendigen systemspezifischen Kennzahlen entwickelt. Die Untersuchung werden auf einigen realen OpenGL ES-Systeme durchgeführt. Dabei wird gezeigt, dass insbesondere das Speicherbelegungsverhalten und die Nutzung der Renderpipeline mit Problemen verbunden sind, die der Erfüllung von Echtzeitgarantien im Wege stehen und nicht auf der Ebene von OpenGL ES gelöst werden können.

Abstract

OpenGL ES 2.0 (Open Graphics Library for Embedded Systems 2.0) is an interface for 2D and 3D computer graphics application development. The specification of OpenGL ES 2.0 defines a set of commands which allow to transfer data to and from an OpenGL ES system, to render graphics and to change settings regarding the rendering process. Some OpenGL ES systems make use of a physical GPU in order to perform rendering. None of the OpenGL ES systems available today neither support the prioritization of applications regarding the execution of OpenGL ES commands nor the limitation of applications regarding their utilization of GPU resources. Particularly, OpenGL ES offers neither a configurable scheduler nor the possibility to fulfill any sort of real-time guarantee regarding the execution of OpenGL ES commands. The intention of this work is to explore to what extend it is still possible to guarantee that commands of safety-critical applications can be executed in time. Therefore, relevant commands are identified whose run time behavior and resource consumption will be analyzed. Furthermore, special scenarios are analyzed in order to determine the impact of OpenGL ES system behavior finishing critical commands in time. Finally, examination methods and metrics are developed to predict the resource consumption of OpenGL ES commands and to determine the required system specific characteristics. Then, some real-world OpenGL ES systems are examined. In doing so it can be shown that their system behavior – especially regarding the utilization of GPU memory and the rendering pipeline – is fraught with problems which make it impossible to fulfill real-time guarantees and which cannot be solved on the level of OpenGL ES.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

1.1 Hintergrund dieser Arbeit

Früher wurden 3D-Computergrafik-Anwendungen vor allem für den Einsatz in Desktop-Systemen entwickelt. Dies ändert sich dahingehend, dass solche Anwendungen zunehmend auch für eingebettete Systeme entwickelt werden, wie man sie beispielsweise in Mobiltelefonen, PDAs aber auch in Automobilen findet. OpenGL ES fungiert dabei inzwischen als eine de-facto Standardschnittstelle für die Entwicklung von 3D-Computergrafik-Anwendungen für eingebettete Systeme (vgl. [Cole 2005], Seite 7).

Die Zunahme dieser Anwendungen bringt eine Reihe von neuen Anforderungen an solche Systeme mit sich, insbesondere im Hinblick auf Ausführung von Anwendungen, die um die Ressourcen der eingebetteten GPU kämpfen, und im Hinblick auf die Erfüllung von Echtzeitgarantien für sicherheitskritische Anwendungen. Dies wird nachfolgend an einem konkreten Beispiel aus der Automobilbranche näher erläutert.

Abbildung 1.1: Innenraum der Fahrzeugstudie MB F800 (Quelle: benzs.blogspot.com)

Abbildung 1.1 zeigt den Innenraum des Mercedes-Benz F800. Bei diesem Fahrzeug handelt es sich um eine sogenannte Fahrzeugstudie, d. h. dieses Fahrzeug wird nicht in die Serienproduktion gehen – es lässt aber erahnen, in welche Richtung die künftige Entwicklung gehen könnte.

Hierbei sind insbesondere die beiden in der Abbildung erkennbaren Displays interessant. Große Displays wie das rechte, im oberen Bereich der Mittelkonsole – die sogenannte *Headunit* – sind aus neueren Serienfahrzeugen bekannt. Eine solche Headunit dient zum Beispiel der Darstellung der grafischen Benutzeroberflächen von Autoradios, Navigationssoftware, etc.

Hinter dem Lenkrad befindet sich das sogenannte Kombiinstrument, auf dem beispielsweise die aktuelle Geschwindigkeit, Drehzahl, etc. angezeigt werden. Die Besonderheit an diesem kombiinstrument ist, dass es ohne mechanische Teile auskommt. Instrumente wie Drehzahlmesser, Geschwindigkeitsanzeige, etc. sind hier nur noch virtuell vorhanden – sie werden auf einem Computer-Display gerendert.

In diesem Fahrzeug werden die auf der Headunit und dem Kombiinstrument angezeigten Grafiken auf separaten GPUs erstellt. Prinzipiell könnten die Ausgaben für die Headunit und das Kombiinstrument aber auch auf der selben GPU berechnet werden. Dies hätte Vorteile hinsichtlich Kosten, Platz- und Stromverbrauch.

Dabei ist jedoch zu beachten, dass einige der Anzeigen auf dem Kombiinstrument sicherheitskritisch sind, zum Beispiel die Geschwindigkeitsanzeige. Es darf nicht passieren, dass solche Anzeigen durch Anwendungen der Mittelkonsole behindert werden – wenn der Benutzer beispielsweise die Navigationssoftware auf der Mittelkonsole startet, darf die Geschwindigkeitsanzeige auf dem Kombiinstrument deshalb nicht „ruckeln“, oder „einfrieren“.

Wenn sich die Anwendungen der Mittelkonsole und des Kombiinstruments eine GPU teilen, müssen die sicherheitskritischen Anwendungen gegenüber anderen Anwendungen priorisiert und Echtzeitgarantien für die Ausführung der GPU-Befehle kritischer Anwendungen erfüllt werden können. Im folgenden Abschnitt wird erläutert, was in dieser Arbeit darunter zu verstehen ist.

1.2 Echtzeitgarantien

Unter Echtzeitgarantie wird verstanden, dass sichergestellt ist, dass eine kritische Anwendung durch andere Anwendungen, die die selbe GPU verwenden, nicht derart behindert wird, dass ihre (sicherheitskritische) Anzeige nicht rechtzeitig aktualisiert werden kann. Was *rechtzeitig* bedeutet, hängt dabei von der jeweiligen Anwendung ab.

Wenn nachfolgend von einer Echtzeitgarantie für eine kritische Anwendung gesprochen wird, wird darunter verstanden, dass sichergestellt ist, dass höchstens ein Intervall von einer i-tel Sekunde zwischen dem Zeitpunkt des Abschlusses des Renderings eines Bildes und dem Abschluss des Renderings des darauf folgenden Bildes vergeht. Dies wird nachfolgend an einem konkreten Beispiel erklärt:

Einige sicherheitskritische Anzeigen, wie beispielsweise die Geschwindigkeitsanzeige, müssen regelmäßig aktualisiert werden. Dabei soll dem Fahrer der Eindruck einer kontinuierlichen Bewegung vermittelt werden. Um dies zu erreichen, muss die Anzeige mehrmals pro Sekunde aktualisiert werden, indem ein neues Bild des Instruments gerendert wird.

An dieser Stelle wird beispielhaft davon ausgegangen, dass hierfür ein Mindestwert von 30 Bildern pro Sekunde nicht unterschritten werden darf. Eine naive Festlegung der Anzahl an gerenderten Bildern pro Sekunde reicht dabei aber nicht aus, um eine praktikable Echtzeitgarantie für diese Anwendung zu definieren. In diesem Beispiel könnte diese Bedingung auch dann erfüllt werden, wenn die für die Geschwindigkeitsanzeige zuständige Anwendung $\frac{9}{10}$ Sekunden blockiert würde und in der letzten Zehntel Sekunde 30 Bilder rendern könnte. In dem Fall hätte der Fahrer jedoch nicht den Eindruck einer kontinuierlichen Bewegung.

Um Bilder rendern zu können, müssen Anwendungen Befehle an die GPU übermitteln. Eine der Schnittstellen, die Anwendungen dies ermöglicht, ist OpenGL ES. Die Betrachtung dieser Schnittstelle (in der Version 2.0) steht im Mittelpunkt dieser Arbeit. Eine eingehende Beschreibung der Grundlagen von OpenGL ES 2.0 erfolgt in Kapitel 2. Im nächsten Abschnitt werden einige der grundsätzlichen Probleme dieser Schnittstelle erörtert, die der Erfüllung von Echtzeitgarantien im Wege stehen.

Wenn nicht explizit anders angegeben, ist nachfolgend OpenGL ES in der Version 2.0 gemeint, wenn von OpenGL ES ohne Angabe der Versionsnummer gesprochen wird.

1.3 Probleme heutiger OpenGL ES-Systeme

Keines der heute verfügbaren OpenGL ES-Systeme[1] unterstützt eine Virtualisierung der für die Ausführung der OpenGL ES-Befehle verwendeten GPU. Insbesondere ist es nicht möglich,

- die Ausführung von Befehlen einer bestimmten Anwendung gegenüber anderen Anwendungen zu priorisieren oder
- Einschränkungen hinsichtlich der Nutzung der Ressourcen der verwendeten GPU (zum Beispiel GPU-Speicher) für einzelne Anwendungen festzulegen.

Dies hat zur Folge, dass „bösartige“ oder fehlerhafte Anwendungen einen Großteil der GPU-Ressourcen für sich beanspruchen können. OpenGL ES definiert keinen Mechanismus, um in einem solchen Umfeld für kritische Anwendungen sicherstellen zu können, dass deren Befehle rechtzeitig (oder überhaupt) ausgeführt werden können.

Erschwerend kommt hinzu, dass die von einem OpenGL ES-System verwendete Hardware von der Spezifikation von OpenGL ES als *Blackbox* behandelt wird. Das Verhalten der Hardware bei Ausführung eines bestimmten OpenGL ES-Befehls ist auf Grundlage seiner Spezifikation daher nicht vorhersagbar. OpenGL ES definiert zum Beispiel eine Reihe von Datenobjekten, die vom OpenGL ES-System verwaltet werden. Anwendungen haben die Möglichkeit über OpenGL ES-Befehle Daten in solchen Datenobjekten zu speichern. Die Spezifikation schreibt aber weder vor, wo diese Daten abgelegt werden (zum Beispiel im Hauptspeicher oder im GPU-Speicher), noch wie sie abgelegt werden. Es wäre demnach spezifikationskonform, wenn durch die Übertragung von einem Megabyte Daten in ein solches Datenobjekt fünf Megabyte des im GPU-Speicher verfügbaren Speicherplatzes belegt würden.

[1] Ein OpenGL ES-System ist ein System, das eine Implementierung der OpenGL ES-Schnittstelle bereitstellt für Anwendungen bereitstellt und ihnen ermöglicht, über diese Schnittstelle Bilder zu rendern. Eine genaue Beschreibung der Komponenten eines OpenGL ES-Systems erfolgt in Kapitel 2.2.

Im nächsten Abschnitt wird ein Lösungsansatz beschrieben, durch den es trotz der hier genannten Probleme ermöglicht werden soll, Echtzeitgarantien für kritische Anwendungen zu erfüllen, auch wenn das verwendete OpenGL ES-System zur gleichen Zeit von anderen Anwendungen genutzt wird.

1.4 Ansatz zur Lösung der genannten Probleme

Die Grundidee zur Lösung der oben beschriebenen Probleme basiert darauf, dass Anwendungen ihre Befehle nicht direkt an das OpenGL ES-System übermitteln, sondern an eine Zwischenschicht, die die gleiche Schnittstelle bereitstellt. Diese Zwischenschicht wertet die übermittelten Befehle anhand der übergebenen Parameter und des aktuellen Kontextes aus und entscheidet dann darüber, ob die Befehle sofort oder erst zu einem späteren Zeitpunkt an das eigentliche OpenGL ES-System weitergeleitet werden können, oder ob sie ganz abgelehnt werden müssen. Die Zwischenschicht verfolgt bei dieser Entscheidung das Ziel, sicherzustellen, dass die nächsten Befehle kritischer Anwendungen rechtzeitig ausgeführt werden können. Damit die Zwischenschicht dieser Aufgabe nachkommen kann, müssen jedoch die folgenden beiden Voraussetzungen erfüllt sein:

- Der Ressourcenverbrauch und die Laufzeit der übermittelten Befehle müssen insoweit vorhergesagt werden können, dass zumindest eine Obergrenze dafür garantiert werden kann.
- Das Verhalten des verwendeten OpenGL ES-Systems hinsichtlich der Ausführung der übermittelten Befehle muss bekannt sein.

Der erste Punkt allein reicht nicht aus. Dies soll an einem Beispiel erläutert werden: Angenommen es soll ein Befehl einer unkritischen Anwendung ausgeführt werden, dessen Laufzeit gerade kurz genug ist, um den nächsten Befehl einer kritischen Anwendung rechtzeitig an das OpenGL ES-System übermitteln zu können. Dann dürfte der Befehl der unkritischen Anwendung auf Grundlage des ersten Punktes ausgeführt werden. Falls aber über das System-Verhalten bekannt ist, dass der erste Befehl einer Anwendung stets um eine bestimmte Zeit verzögert wird, wenn er auf den Befehl einer anderen Anwendung folgt, dann dürfte der Befehl der unkritischen Anwendung nicht ausgeführt werden.[2]

Der Wunsch, diese beiden Voraussetzungen erfüllen zu können, ist der Ausgangspunkt für die vorliegende Arbeit. Im folgenden Abschnitt wird ihre Zielsetzung genauer erläutert.

1.5 Zielsetzung dieser Arbeit

Ziel dieser Arbeit ist es, anhand einiger realer OpenGL ES-Systeme zu untersuchen, inwieweit die beiden im vorhergehenden Abschnitt genannten Voraussetzungen für die Implementierung einer Zwischenschicht mit den genannten Aufgaben erfüllt werden können. Konkret setzt sich diese Arbeit die folgenden vier Ziele:

[2] Dies kommt in OpenGL ES-Systemen vor, weil die relevanten GPU-Einstellungen der aufrufenden Anwendung erneut in die GPU übertragen werden müssen, da sich die Einstellungen der zuvor bedienten Anwendung davon unterscheiden können. Ein solcher Vorgang wird auch als *Kontextwechsel* bezeichnet.

- Sämtliche relevanten Befehle von OpenGL ES sind zu bestimmen, deren Ausführung einen negativen Einfluss auf die Erfüllung von Echtzeitgarantien für kritische Anwendungen haben könnte. Ein solcher Einfluss ist nicht für alle von OpenGL ES definierten Befehle zu erwarten. Es existiert beispielsweise ein Befehl, mit dem der Name des Herstellers des verwendeten OpenGL ES-Systems abgefragt werden kann. Es ist nicht anzunehmen, dass durch diesen Befehl die Ressourcen der GPU in irgendeiner Weise belastet werden.
- Für die relevanten OpenGL ES-Befehle ist eine Analyse ihres Laufzeitverhaltens und ihres Ressourcenverbrauchs durchzuführen.
- Spezielle Szenarien sind zu untersuchen, um festzustellen, inwiefern das Verhalten von OpenGL ES-Systemen die rechtzeitige Ausführung kritischer Befehle verhindern kann. Die im letzten Abschnitt beschriebenen zusätzlichen Laufzeitkosten für die Ausführung von Befehlen, die unmittelbar nach dem Befehl einer anderen Anwendung ausgeführt werden, wären ein Beispiel für so ein solches Systemverhalten.
- Untersuchungsmethoden und Metriken sind zu entwickeln, auf deren Grundlage der Ressourcenverbrauch sowie die Laufzeit von OpenGL ES-Befehlen auf konkreten OpenGL ES-Systemen prognostiziert und die dafür notwendigen systemspezifischen Kennzahlen ermittelt werden können. Die entwickelten Untersuchungsmethoden sind auf realen OpenGL ES-Systemen durchzuführen.

Die in dieser Arbeit entwickelten Metriken finden sich in Kapitel 2. Die Untersuchungsmethoden werden in Kapitel 3 beschrieben. Die Anwendung der Untersuchungsmethoden und Metriken auf drei realen OpenGL ES-Systemen erfolgt schließlich in Kapitel 4. Im nächsten Abschnitt wird ein Überblick über relevante, verwandte Arbeiten gegeben.

1.6 Verwandte Arbeiten

[Dwarakinath 2008] beschäftigt sich damit, wie sichergestellt werden kann, dass jede Anwendung, die die GPU nutzt, einen gleichen Anteil an den Rechenzeit-Ressourcen der GPU erhält. Diese Arbeit setzt auf Ebene der GPU-Treiber an. Diese werden um einen Scheduler erweitert, der Befehle verschiedener Anwendungen zwischenspeichert. Er teilt die verfügbare Rechenzeit in gleich große Intervalle auf und weist jeder Anwendung in jedem Intervall einen zeitlichen Anteil an der GPU-Nutzung zu. Falls eine Anwendung ihren Anteil in einem Intervall nicht ausschöpfen kann, erhöht sich ihr Anteil im nächsten Intervall entsprechend (umgekehrt verringert er sich, wenn eine Anwendung im vorhergehenden Intervall die GPU länger nutzt als erlaubt). Befehle einer Anwendung werden nur an die GPU übergeben, wenn deren prognostizierte Laufzeit geringer ist als der ihr zugewiesene Anteil.

Durch dieses Vorgehen soll es ermöglicht werden, dass alle Anwendungen im langen Mittel einen gleichen Anteil an den Rechenzeit-Ressourcen der GPU erhalten. Eine Priorisierung einzelner Anwendungen ist nicht vorgesehen und auch nicht die Erfüllung von Echtzeitgarantien für einzelne Anwendungen.

[Bautin u. a. 2008] erweitert den Ansatz von [Dwarakinath 2008] dahingehend, dass Anwendungen, die um die GPU konkurrieren, auch einen gleichen Anteil am GPU-Speicher erhalten.

[Kato u. a. 2011] verfolgt einen ähnlichen Ansatz wie [Dwarakinath 2008]. Auch hier werden die GPU-Treiber um einen Scheduler erweitert, der zum Ziel hat, die verfügbare GPU-Rechenzeit auf konkurrierende Anwendungen aufzuteilen; im Gegensatz zu [Dwarakinath 2008] ermöglicht dieser Ansatz jedoch die Priorisierung einzelner Anwendungen gegenüber anderen Anwendungen.

Auch dieser Ansatz hat nicht die Erfüllung von Echtzeitgarantien zum Ziel. Auch bei höchster Priorisierung kann nicht ausgeschlossen werden, dass durch die Ausführung von Befehlen niedrig priorisierter Anwendungen die rechtzeitige Ausführung von Befehlen kritischer Anwendungen verhindert wird.

[Grottel u. a. 2009] beschäftigt sich mit der Nutzung des GPU-Speichers durch OpenGL und mit der Datenübertragung vom Hauptspeicher in die GPU und untersucht verschiedene Ansätze, um zu visualisierende Datensätze, die sich häufig ändern, der GPU möglichst effizient zugänglich zu machen. Dabei wird der Einfluss der verschiedenen Ansätze auf das Laufzeitverhalten von OpenGL-Datenübertragungsbefehlen und Zeichenbefehlen analysiert. Da dieses Verhalten auch im Rahmen der vorliegenden Arbeit untersucht werden muss, sind die von [Grottel u. a. 2009] gewonnen Erkenntnisse dazu für die kommenden Untersuchungen relevant.

Im nächsten Kapitel wird ein Überblick über die Schnittstelle OpenGL ES gegeben und es werden die Grundlagen von OpenGL ES-Systemen erörtert, die zum Verständnis der nachfolgenden Kapitel unverzichtbar sind.

2 Grundlagen von OpenGL ES-Systemen

2.1 OpenGL und OpenGL ES

Für GPUs existiert keine Standard-Hardwareschnittstelle wie beispielsweise die IDE- oder SCSI-Schnittstelle für Laufwerke. Es wäre daher sehr schwierig, Computergrafik-Anwendungen zu entwickeln, die eine große Anzahl verschiedener GPUs nutzen können, wenn sie die GPUs über ihre Hardwareschnittstellen ansprechen müssten. Es ist daher unüblich für solche Anwendungen, GPUs direkt über die Hardwareschnittstelle anzusprechen. Stattdessen wird eine High-Level-Grafik-API verwendet, die die Details der jeweiligen Hardware verbirgt. OpenGL ist eine solche API. Sie unterstützt alle großen Betriebssysteme und wird von den meisten GPU-Herstellern für ihre Produkte implementiert (vgl. [Lagar-Cavilla u. a. 2007], Seiten 33–34).

OpenGL ES ist eine Variante von OpenGL, die speziell im Hinblick auf eingebettete GPUs entwickelt wurde (wie sie zum Beispiel in Fahrzeugen, Mobiltelefonen, PDAs, etc. zum Einsatz kommen). Eines der vorrangigen Ziele der Hersteller solcher eingebetteten GPUs ist die Reduktion des Stromverbrauchs. OpenGL ES ermöglicht es ihnen, eine gegenüber Standard-OpenGL erheblich reduzierte API zu unterstützen, was wiederum die Anforderungen an die Hardware verringert und somit hilft, Geräte zu entwickeln, die letztendlich weniger Strom verbrauchen (vgl. [Cole 2005], Seite 7).

OpenGL ES 2.0 ist an (Standard-) OpenGL 2.0 angelehnt. Nach [Munshi u. a. 2008] unterscheidet es sich im Wesentlichen in den folgenden beiden Punkten von OpenGL 2.0:

- Jegliche Redundanz in OpenGL 2.0 wurde in OpenGL ES 2.0 entfernt. Wo es in OpenGL 2.0 mehr als eine Möglichkeit gibt, um die selbe Operation auszuführen, wurde für OpenGL ES 2.0 nur eine der Möglichkeiten übernommen.
- Um spezielle Einschränkungen von eingebetteten GPUs zu berücksichtigen, wurde neue Funktionalität in OpenGL ES 2.0 eingeführt, beispielsweise die Möglichkeit, die Präzision von Fließkommavariablen in Shadern zu spezifizieren (eine geringere Präzision ermöglicht die Nutzung vereinfachter Fließkommaprozessoren, die wiederum einen geringeren Stromverbrauch aufweisen können).

OpenGL ES 2.0 ist also keine echte Untermenge von OpenGL 2.0. Von Version 4.0 auf 4.1 wurde OpenGL jedoch dahingehend erweitert, dass es die gesamte Funktionalität von OpenGL ES 2.0 enthält (ab OpenGL 4.1 ist OpenGL ES 2.0 also eine echte Untermenge von OpenGL).

2.2 Implementierungsvarianten von OpenGL ES-Systemen

Die Spezifikationen von OpenGL ES und Standard-OpenGL schreiben nicht vor, dass in einem konkreten System, das OpenGL unterstützt, diese Unterstützung in eigener Hardware implementiert

sein muss (siehe [Segal u. a. 2010], Seite 2). Es gibt Systeme, die die gesamte Funktionalität von OpenGL in Software realisieren, zum Beispiel Mesa 3D [Paul 2007].[1]

Die Nutzung von Softwareimplementierungen ist jedoch unüblich, da die Verwendung von spezialisierter Hardware die Entwicklung leistungsfähigerer Grafik-Anwendungen ermöglicht. Auf Software-Rendering wird im Notfall zurückgegriffen, wenn die verwendete Hardware eine bestimmte Funktionalität nicht unterstützt [vgl. Lagar-Cavilla u. a. 2007, Seite 34].

Üblicherweise bestehen OpenGL ES-Systeme aus vier Komponenten:

- Eine Implementierung der OpenGL ES-API, die von Anwendungen angesprochen werden kann.
- Einen separaten GPU-Speicher, in dem das OpenGL ES-System seine Daten ablegt. Dieser GPU-Speicher ist im einfachsten Fall ein Bereich des Hauptspeichers, der vom OpenGL ES-System genutzt werden kann. Es gibt aber auch Systeme, die über einen eigenen, physischen Speicher verfügen. Moderne Desktop-Systeme verfügen typischerweise über mehrere Gigabyte eines solchen Speichers mit einer internen Speicherbandbreite von über 64 Gbps [Satish u. a. 2009].
- Eine physische GPU, die idealerweise alle Schritte der Renderpipeline[2] in Hardware unterstützt. Während früher üblicherweise jeder Schritt der Renderpipeline durch eigene Hardware unterstützt wurde, geht seit einigen Jahren der Trend hin zur Verwendung sogenannter *Unified Processors* – programmierbare Prozessoren, die verschiedene Aufgaben erfüllen können und auf denen mehrere Schritte der Renderpipeline berechnet werden (siehe beispielsweise Nvidia 2006, Seiten 20–21).
- Ein Gerätetreiber, der als Schnittstelle zwischen der OpenGL ES-API, dem Betriebssystem sowie der GPU und dem GPU-Speicher fungiert.

2.3 Client-Server-Modell der Befehlsübermittlung

Die Übermittlung von OpenGL ES-Befehlen folgt einem Client-Server-Modell. Wenn eine Anwendung einen OpenGL ES-Befehl aufruft, wird der Befehl an eine Komponente des OpenGL ES-Systems übermittelt, die als *GL-Client* bezeichnet wird. Für jede Anwendung, die OpenGL ES nutzt, existiert eine Instanz dieser Komponente. Sie hat die Möglichkeit, an sie übermittelte Befehle zwischenzuspeichern und erst zu einem späteren Zeitpunkt an den sogenannten *GL-Server* weiterzuleiten. OpenGL ES-Befehle werden erst ausgeführt, nachdem sie an den GL-Server übermittelt wurden.

Der GL-Server kann sich prinzipiell auf einem anderen Rechner befinden als der GL-Client. In eingebetteten Systemen ist dies üblicherweise nicht der Fall (siehe [Munshi u. a. 2008], Seite 16). Dennoch kann die Übermittlung von Befehlen durch den GL-Client verzögert werden. Dies ist abhängig vom konkreten OpenGL ES-System. Sobald der GL-Client einen Befehl zwischengespeichert

[1] Mesa 3D ist seit Version 2.2 keine ausschließliche Softwareimplementierung mehr. Es kann optional auch Hardwarebeschleunigung nutzen, sofern entsprechende Treiber vorhanden sind.

[2] Die Renderpipeline von OpenGL ES besteht aus einer Abfolge von Datenverarbeitungsschritten, durch die die übergebenen Eingangsdaten sukzessive umgewandelt werden, bis am Ende ein Bild entsteht, das auf Computerbildschirmen ausgegeben werden kann. Die Renderpipeline wird in Abschnitt 2.6 detailliert beschrieben.

hat, kann die von der Anwendung aufgerufene Funktion bereits zurückspringen, d. h. es ist prinzipiell möglich, dass ein Rücksprung erfolgt, bevor der entsprechende Befehl vom OpenGL ES-System ausgeführt wurde.

Es macht daher wenig Sinn, die Laufzeit eines OpenGL ES-Befehls zu messen, indem die verstrichene Zeit zwischen dem Aufruf des entsprechenden Befehls und dessen Rücksprung ermittelt wird. OpenGL ES definiert jedoch zwei Befehle, die hier sehr nützlich sind, `glFlush` und `glFinish`:

- Der Aufruf von `glFlush` veranlasst den GL-Client, alle zwischengespeicherten Befehle an den GL-Server weiterzuleiten.
- Der Aufruf von `glFinish` überprüft, ob alle Befehle, die von der aufrufenden Anwendung an das OpenGL ES-System übermittelt wurden, vollständig ausgeführt wurden, und kehrt erst zurück, sobald dies der Fall ist.

Mit Hilfe dieser beiden Befehle kann eine sinnvolle Metrik für die Bestimmung der Laufzeit von OpenGL ES-Befehlen definiert werden: Die Laufzeit eines OpenGL ES-Befehls C ist die Zeit, die zwischen dem Aufruf von C durch die Anwendung A und dem Rücksprung des nächsten Aufrufs von `glFinish` durch A vergeht, wobei die folgenden Bedingungen von A erfüllt werden müssen:

- Vor Aufruf von C ruft A `glFinish` auf.
- Zwischen dem Rücksprung von `glFinish` und dem Aufruf von C ruft A keinen weiteren OpenGL ES-Befehl auf.
- Unmittelbar nach dem Rücksprung von C ruft A `glFlush` auf.
- Unmittelbar nach dem Rücksprung von `glFlush` ruft A `glFinish` auf.

Wenn nachfolgend im Zusammenhang einer OpenGL ES-Anwendung von der „Übermittlung eines OpenGL ES-Befehls an den GL-Server“ gesprochen wird, ist damit gemeint, dass dem Aufruf des entsprechenden Befehls ein Aufruf von `glFlush` folgt.

2.4 Datenübergabe und -verwaltung

Bevor die Renderpipeline eines OpenGL ES-Systems ein Bild rendern kann, muss die OpenGL ES-Anwendung dem System Daten übergeben, aus denen das Bild berechnet werden kann. OpenGL ES unterscheidet fünf Arten solcher Daten:

- Vertexdatenarrays
- Indexdatenarrays
- Vertex-Attribute
- Texturen
- Uniforms

Eine Minimalvoraussetzung, um ein Bild rendern zu können, ist, dass dem OpenGL ES-System von der aufrufenden Anwendung mindestens ein Vertexdatenarray übergeben wurde. Dieses Array muss mindestens ein Element enthalten und die Größe der Elemente muss mindestens ein Byte

betragen. Elemente von Vertexdatenarrays dienen als Input für den Vertexshader-Schritt der Renderpipeline. Im Zuge dieses Schrittes wird vom OpenGL ES-System für jedes Element im Vertexdatenarray eine Instanz eines speziellen Programms ausgeführt, das als *Vertexshader* bezeichnet wird. Da Vertexshader bewusst so entworfen wurden, dass sie relativ geringe Datenmengen verarbeiten, ist die maximale Größe der Elemente von Vertexdatenarrays auf 16 Byte begrenzt (siehe [Bailey 2011], Seite 67). Vertexshader werden in Abschnitt 2.5 näher erläutert.

Die Elemente eines Indexdatenarrays bestimmen beim Rendering, welche Elemente der zu verarbeitenden Vertexdatenarrays berücksichtigt werden sollen und in welcher Reihenfolge sie berücksichtigt werden sollen. Die Übergabe eines Indexdatenarrays ist optional. Wenn keines übergeben wird, werden alle Elemente von Vertexdatenarrays in der Reihenfolge berücksichtigt, in der sie im Array liegen.

Die Übergabe von Vertex-Attributen, Texturen oder Uniforms ist ebenfalls optional. Vertex-Attribute dienen ebenfalls als Input für Vertexshader. Für sie gelten die gleichen Einschränkungen wie für Elemente eines Vertexdatenarrays. Sie werden in Abschnitt 2.6.2.1 näher erläutert. Texturen dienen als Input für Fragmentshader[3] und werden in Abschnitt 2.6.2.4 näher erläutert. Uniforms dienen sowohl als Input für Vertex- als auch für Fragmentshader. Sie erfüllen die Rolle von konstanten Variablen und stehen jeder ausgeführten Instanz dieser Programme zur Verfügung. Ihre Größe ist auf 128 Byte begrenzt.

Vertexdatenarrays liegen im Hauptspeicher und werden von der OpenGL ES-Anwendung verwaltet – vor Anstoßen des Renderings muss dem OpenGL ES-System durch die Anwendung ein Zeiger auf das Array übergeben werden. Alle übrigen Daten werden vom OpenGL ES-System verwaltet, sobald sie ihm übergeben wurden. Nach erfolgter Übergabe kann die OpenGL ES-Anwendung diese Daten freigeben. Optional besteht auch für Vertexdatenarrays die Möglichkeit, sie vom OpenGL ES-System verwalten zu lassen, indem sie in ein spezielles Datenobjekt kopiert werden. Im nächsten Abschnitt werden die verschiedenen Arten von Datenobjekten genauer erläutert.

2.4.1 Datenobjekte

Datenobjekte sind vom OpenGL ES-System verwaltete Ressourcen. OpenGL ES-Anwendungen können Datenobjekte zwar erzeugen und freigeben sowie Daten in diese Datenobjekte übertragen (mit Ausnahme von Renderbuffer-Objekten, siehe Abschnitt 2.4.1.3); das OpenGL ES-System ist jedoch für die Reservierung und Freigabe des dafür nötigen Speicherplatzes zuständig und bestimmt darüber, wo sich Datenobjekte befinden (zum Beispiel im Hauptspeicher oder im GPU-Speicher). Nachdem sie Daten in Datenobjekte übertragen haben, erhalten OpenGL ES-Anwendungen keinen direkten Zugriff mehr darauf. Sie können allerdings erneut Daten in ein Datenobjekt übertragen, wodurch bereits dort befindliche überschrieben werden (außer bei Renderbuffer-Objekten, siehe Abschnitt 2.4.1.3).

OpenGL ES kennt vier Arten von Datenobjekten: Vertexbuffer-Objekte, Texturobjekte, Renderbuffer-Objekte und Framebuffer-Objekte. Diese Datenobjektarten werden in den nächsten Abschnitten genauer beschrieben.

[3] Wie Vertexshader sind auch Fragmentshader spezielle Programme, die im Zuge der Datenverarbeitung durch die Renderpipeline ausgeführt werden. Sie werden in Abschnitt 2.5 näher erläutert.

2.4.1.1 Vertexbuffer-Objekte

In Vertexbuffer-Objekten können Vertex- oder Indexdatenarrays gespeichert werden. Außer einer systemspezifischen Maximalgröße definiert OpenGL ES keine Einschränkungen hinsichtlich ihrer Größe. Eine Besonderheit, die Vertexdaten-Objekte von anderen Datenobjekten unterscheidet, ist die Möglichkeit, einen *Usage Hint* anzugeben. Der Usage Hint dient dazu, dem OpenGL ES-System einen Hinweis zu geben, wie das entsprechende Datenobjekt von der Anwendung verwendet werden wird (vgl. [Munshi und Leech 2010], Seiten 23–24). Es stehen drei Möglichkeiten zur Wahl:

- `GL_STATIC_DRAW`: Der Inhalt des Datenobjekts wird von der Anwendung einmal festgelegt und häufig zum Rendern verwendet.
- `GL_DYNAMIC_DRAW`: Der Inhalt des Datenobjekts wird von der Anwendung häufig verändert und häufig zum Rendern verwendet.
- `GL_STREAM_DRAW`: Der Inhalt des Datenobjekts wird von der Anwendung einmal festgelegt und selten zum Rendern verwendet.

Die Spezifikation von OpenGL ES schreibt nicht vor, wie ein konkretes OpenGL ES-System auf diese Hinweise reagieren soll. Im Gegenzug ist es den OpenGL ES-Anwendungen freigestellt, sich anders zu verhalten, als dies durch den angegeben Usage Hint zu erwarten wäre. Es ist beispielsweise durchaus legal, ein Vertexbuffer-Objekt mit Angabe von `GL_STATIC_DRAW` zu erzeugen und es anschließend häufig zu verändern (siehe auch [Munshi u. a. 2008], Seite 118).

2.4.1.2 Texturobjekte

In Texturobjekten können Texturen gespeichert werden. Texturen dienen als optionaler Input für Fragmentshader (vgl. Abschnitt 2.6.2.4). Die Datenelemente einer Textur – *Texel* genannt – sind als zweidimensionales Array organisiert. Hinsichtlich der Zeilen und Spalten-Anzahl gilt die Einschränkung, dass es sich dabei um eine Zweierpotenz handeln muss. Die Größe eines Texels ist auf vier Byte begrenzt.

2.4.1.3 Renderbuffer-Objekte

Renderbuffer-Objekte unterscheiden sich von den anderen Datenobjekten darin, dass OpenGL ES-Anwendungen keine Daten in sie übertragen können. In Renderbuffer-Objekten werden von der Renderpipeline erzeugte Ergebnisdaten gespeichert.

2.4.1.4 Framebuffer-Objekte

Anders als bei den andern drei Datenobjektarten haben OpenGL ES-Anwendungen keinen Einfluss auf die Größe von Framebuffer-Objekten. In ihnen können die IDs von bis zu drei anderen Datenobjekten gespeichert werden, wobei nur IDs von Textur- und Renderbuffer-Objekten gespeichert werden können. Framebuffer-Objekte fungieren als Zielpunkte für die Ausgabe der Renderpipeline und leiten die Ergebnisdaten an die Datenobjekte weiter, deren IDs sie gespeichert haben.

Die drei Datenobjekte werden einem Framebuffer-Objekt jeweils in einer von drei Rollen zugewiesen: Als sogenannter Colorbuffer, Depthbuffer oder Stencilbuffer. Im Colorbuffer wird das gerenderte Bild gespeichert. Im Depth- und im Stencilbuffer werden Werte gespeichert, die im Rahmen des letzten Schrittes der Renderpipeline dazu genutzt werden können, zu entscheiden, ob und ggf. wie einzelne Ergebniswerte mit den bereits vorhandenen Werten in den drei Datenobjekten des Framebuffer-Objekts kombiniert werden (siehe Abschnitt 2.6.2.5).

2.4.2 Eviction

Wenn gewisse Daten zur Ausführung des Renderings im GPU-Speicher liegen müssen, kann die Situation eintreten, dass ein Bild nicht gerendert werden kann, weil im GPU-Speicher nicht mehr genug Platz vorhanden ist, um die dafür benötigten Daten dort unterzubringen. Um auch in einer solchen Situation Rendering zu ermöglichen, verfügen gewisse OpenGL ES-Systeme über einen sogenannten Eviction-Mechanismus. Unter dem Begriff Eviction versteht man das Auslagern von Daten aus dem GPU-Speicher, um Platz für die Speicherung anderer Daten zu schaffen.

Eviction kann unter ungünstigen Umständen aber auch negative Auswirkungen haben. Die Auslagerung eines Datenobjekts könnte zur Folge haben, dass sich ein nachfolgender Renderingvorgang verlängert. Dies soll an einem Beispiel erläutert werden: Ausgangspunkt ist, dass ein Vertexbuffer-Objekt erzeugt wird, das die Vertexdaten für einen Renderingvorgang speichern soll. Da im GPU-Speicher nicht mehr genug Platz dafür ist, wird durch den Eviction-Mechanismus ein Texturobjekt in den Hauptspeicher ausgelagert. Nun wird der Renderingvorgang gestartet. Wenn im Zuge dieses Vorgangs aber genau diese Textur im GPU-Speicher benötigt wird, muss sie erst wieder dorthin zurückübertragen werden. Um dies zu ermöglichen, wird durch den Eviction-Mechanismus eine andere Textur ausgelagert und anschließend die erste Textur in den GPU-Speicher übertragen.

Nun kann das Rendering fortgesetzt werden. Es ist aber nicht ausgeschlossen, dass im weiteren Verlauf des Renderingvorgangs auch die zweite ausgelagerte Textur benötigt wird. Diese müsste dazu auch wieder in den GPU-Speicher zurückübertragen werden, wofür erst ein drittes Datenobjekt ausgelagert werden müsste, das möglicherweise später auch benötigt wird, ... Durch die Auslagerung eines Datenobjekts kann es unter ungünstigen Umständen bei der späteren Ausführung eines Renderingvorgangs zu einer Kaskade von Evictions kommen.

2.4.3 Speichergranularität, Effektivgröße und Speicherblockgröße

Wie in Kapitel 1.3 dargelegt, ist nicht gewährleistet, dass die Menge an GPU-Speicher, die von einem Datenobjekt belegt wird, der Menge der in diesem Datenobjekt gespeicherten Daten entspricht. Da Datenobjekte ein zentrales Konzept von OpenGL ES darstellen, ist es wichtig, deren Speicherbedarf in einem konkreten OpenGL ES-System analysieren und vorhersagen zu können. Dazu werden hier die Begriffe *Datenobjektgröße* und *Speicherblock* sowie die Speichermetriken *Speichergranularität*, *Effektivgröße* und *Speicherblockgröße* genau definiert, die in den nachfolgenden Untersuchung von großem Nutzen sein werden:

- Die *Datenobjektgröße* bezeichnet die Menge der von einer OpenGL ES-Anwendung in einem Datenobjekt gespeicherten Daten.

- *Speichergranularität*: Wenn der von Datenobjekten belegte GPU-Speicher stets einem ganzzahligen Vielfachen eines bestimmten Wertes entspricht, dann wird dieser Wert nachfolgend als *Speichergranularität* bezeichnet, wenn zusätzlich für Datenobjekte die folgenden Bedingungen erfüllt werden:
  1. Falls die Datenobjektgröße genau einem ganzzahligen Vielfachen der Speichergranularität entspricht, dann entspricht die von ihm belegte Menge an GPU-Speicher exakt der Datenobjektgröße.
  2. Falls die Größe eines Datenobjekts nicht einem ganzzahligen Vielfachen der Speichergranularität entspricht, dann belegt dieses Datenobjekt die Menge an GPU-Speicher, die dem nächstgrößeren ganzzahligen Vielfachen der Speichergranularität entspricht.
  3. Der zusätzlich von einem solchen Datenobjekt belegte Speicherplatz steht nicht mehr für die Speicherung nachfolgender Datenobjekte zur Verfügung.
- Die *Effektivgröße* eines Datenobjekts entspricht exakt der Größe des Datenobjekts, falls seine Größe einem ganzzahligen Vielfachen der Speichergranularität entspricht. Sonst entspricht die Effektivgröße dem nächstgrößeren ganzzahligen Vielfachen der Speichergranularität.
- *Speicherblock*: Wenn die Menge an GPU-Speicher, die von einem Datenobjekt A belegt wird, größer ist als dessen Effektivgröße X, dann wird nachfolgend davon gesprochen, dass A in einem *Speicherblock* der Größe Y abgelegt wird, wenn mindestens die folgenden Bedingungen erfüllt werden:
  1. Der durch die Erzeugung von A belegte GPU-Speicher entspricht exakt Y.
  2. Falls $\exists\, n \in \mathbb{N} : nX \leq Y \wedge (n+1)X > Y$, dann wird kein weiterer GPU-Speicher mehr durch die Erzeugung der nächsten $n-1$ Datenobjekte belegt, wenn deren Effektivgröße jeweils X entspricht.
  3. Falls kein solches n existiert, dann wird für die Erzeugung des nächsten Datenobjekts kein weiterer GPU-Speicher belegt, falls für dessen Effektivgröße Z gilt: $Z \leq Y - X$

  Hinsichtlich der in 2. und 3. erwähnten Datenobjekte wird nachfolgend davon gesprochen, dass diese im selben Speicherblock abgelegt sind wie A.
- Y wird nachfolgend als *Speicherblockgröße* bezeichnet.

Der entscheidende Unterschied zwischen der Ablage von Datenobjekten in Speicherblöcken und einer Speichergranularität, die größer ist als ein Byte, besteht also darin, dass der zusätzlich zur Datenobjektgröße belegte GPU-Speicher bei Speicherblöcken für nachfolgender Datenobjekte zur Verfügung steht.

2.5 Programmobjekte

Während des Renderings führt die Renderpipeline von OpenGL ES 2.0 zwei Arten von speziellen Programmen aus, Vertexshader und Fragmentshader.[4] Diese Programme müssen von den OpenGL ES-Anwendungen bereitgestellt werden. Dies kann entweder in Form von Quellcode geschehen

[4] Die Erläuterung der genauen Rolle, die die beiden Shader-Arten in der Renderpipeline spielen, erfolgt in Abschnitt 2.6.2.1 für Vertexshader und in Abschnitt 2.6.2.4 für Fragmentshader.

(der in Form von Null-terminierten Zeichenketten übergeben wird) oder in vorkompilierter Form als sogenannte *Shader Binaries*. Shader werden dabei in einem Dialekt der Sprache C geschrieben, der als *OpenGL ES Shading Language* bezeichnet wird [Simpson und Kessenich 2009].

Falls die Shader in Form von Quellcode an das OpenGL ES-System übergeben werden, müssen diese zuerst kompiliert werden, um Shader Binaries zu erzeugen. Die Spezifikation von OpenGL ES schreibt zwingend vor, dass OpenGL ES-Systeme einen Compiler für Shader-Quellcode vorhalten müssen.

Sobald ein Shader Binary von einem Vertexshader und eines von einem Fragmentshader vorliegt, müssen diese zu einem sogenannten Programmobjekt verbunden (gelinkt) werden. Erst sobald ein solches Programmobjekt erzeugt und in der Renderpipeline installiert wurde, kann die Datenverarbeitung durch die Renderpipeline angestoßen werden. Rendering ohne ein Programmobjekt ist in OpenGL ES 2.0 nicht möglich.

2.6 Renderpipeline

2.6.1 Überblick

Durch die Renderpipeline werden die in Abschnitt 2.4 beschriebenen Inputdaten sukzessive in fünf aufeinanderfolgenden Schritten in die Pixel des gerenderten Bildes umgewandelt. Die Ausgabe eines Schrittes fungiert dabei als Eingabe des nächsten Schrittes. In den nächsten fünf Abschnitten werden die einzelnen Schritte der Renderpipeline von OpenGL ES 2.0 eingehender erläutert.

2.6.2 Pipelineschritte

2.6.2.1 Vertexshader

Der Input für den Vertexshader-Schritt besteht aus dem Vertexdatenarray, das dem OpenGL ES-System zuvor für das Rendering übergeben worden ist. Dabei wird für jedes Element des Arrays eine Instanz des installierten Vertexshaders ausgeführt. Jede Instanz erhält mindestens ein Vertexattribut als Input. Dies entspricht dem Element des Vertexdatenarrays, für das die entsprechende Instanz ausgeführt wird.

Zusätzlich kann der Input der Vertexshader-Instanz noch weitere Vertexattribute umfassen, die entweder Elementen anderer Vertexarrays entsprechen oder von der OpenGL ES-Anwendung direkt als Vertexattribut übergeben wurden (siehe auch die Diskussion zu Beginn von Abschnitt 2.4). Außerdem kann der Input noch eine Reihe von Uniform-Variablen umfassen.

Die minimale Ausgabe eines Vertexshaders ist ein Vertex. Ein Vertex ist eine Variable, die eine Position in 2D-Gerätekoordinaten repräsentiert. Zusätzlich kann ein Vertexshader aber über weitere Ausgabevariablen verfügen, die sogenannten *Varyings*, die zusätzlich zu den Vertices durch die Renderpipeline bis zum Fragmentshader-Schritt weitergereicht werden.

Traditionell dienten Vertexshader dazu, Vertices, die in 3D-Weltkoordinaten der Anwendung vorlagen, in 2D-Gerätekoordinaten zu transformieren [Liu u. a. 2007]. In OpenGL ES 2.0 sind Vertexshader nicht mehr darauf beschränkt. Der Input, der ihnen übergeben wird, muss nicht zwingend

Vertices enthalten (auch wenn die Inputdaten aus historischen Gründen als „Vertexdaten“ bezeichnet werden) und seine Ausgabe ist nicht auf Vertices beschränkt.

2.6.2.2 Primitiven-Erzeugung

In diesem Schritt werden aus Vertices sogenannte *Primitiven* erzeugt. Dies sind die grundlegenden zweidimensionalen, geometrischen Figuren, die in den nächsten Schritten der Renderpipeline weiterverarbeitet werden. OpenGL ES kennt drei Arten von Primitiven: Punkte, Linien und Dreiecke. Pro Rendervorgang wird nur eine Art von Primitiven erzeugt. Pro Vertex wird ein Punkt-Primitiv erzeugt. Wenn Linien-Primitive erzeugt werden sollen, wird aus zwei Vertices eine Linie erzeugt, wobei die Vertices die beiden Endpunkte der Linie definieren. Im Falle von Dreiecks-Primitiven wird aus drei Vertices ein Dreieck erzeugt, wobei die Vertices die Eckpukte des erzeugten Dreiecks definieren. Ein und derselbe Vertex kann dabei für die Erzeugung mehrerer Primitive verwendet werden (wenn Indexdaten verwendet werden und der Index des entsprechenden Vertex mehrfach im Indexdatenarray vorkommt).

2.6.2.3 Rasterisierung

Im Rasterisierungsschritt werden für jedes Primitiv diejenigen Pixel im zu rendernden Bild bestimmt, die vom jeweiligen Primitiv berührt werden. Für jedes dieser Pixel wird ein sogenanntes Fragment erzeugt, das im nächsten Schritt der Renderpipeline weiterverarbeitet wird. Falls das OpenGL ES-System *Multisampling* verwendet, wird jeder Pixel des zu rendernden Bildes in mehrere Subpixel unterteilt, die sogenannten Samples. In diesem Fall werden durch die Rasterisierung nicht die Pixel bestimmt, die durch das verarbeitete Primitiv berührt werden, sondern die Samples – es wird dann für jedes berührte Sample ein Fragment erzeugt. Im fertigen Bild werden die erzeugten Fragmente wieder zu ganzen Pixeln kombiniert. Dieses Vorgehen verfolgt das Ziel, sogenannte Aliasing-Artefakte zu reduzieren (siehe [Munshi u. a. 2008], Seite 234).

2.6.2.4 Fragmentshader

Im Fragmentshader-Schritt wird für jedes Fragment eine Instanz des installierten Fragmentshaders ausgeführt. Aufgabe der ausgeführten Fragmentshader-Instanzen ist es, dem von ihnen verarbeiteten Fragment einen Farbwert zuzuweisen. Traditionell dienten Fragmentshader dazu, Fragmenten einen Farbwert zuzuweisen, indem Texel aus Texturobjekten ausgelesen wurden [Liu u. a. 2007]. Auch in OpenGL ES 2.0 können Fragmentshader Texel auslesen, müssen es aber nicht.

Zur Berechnung des Farbwerts ihres Fragments können Fragmentshader-Instanzen auch anderen Input nutzen. Neben der Position des Fragments im zu rendernden Bild stehen für den Input von Fragmentshadern auch Uniforms und Varyings zur Verfügung. Varyings sind optionale Ausgabevariablen von Vertexshadern, die durch die Renderpipeline an Fragmentshader weitergereicht werden – dies geschieht aber nur im Fall von Punkt-Primitiven ohne Veränderung der in den Varyings gespeicherten Werte. Falls ein Fragment aus Linien- oder Dreiecks-Primitiven erzeugt wurde, hängen die Werte der ihm übergebenen Varyings von den Ausgaben von zwei bzw. drei Vertexshadern ab und werden zwischen diesen Ausgaben interpoliert.

2.6.2.5 Fragmentverarbeitung

Ziel des letzten Schritts der Renderpipeline ist es, für jedes Fragment zu entscheiden, ob es verworfen oder in das zu rendernde Bild integriert wird. Dazu wird das Fragment einer Reihe von optionalen Tests unterzogen:

- Falls der Pixel, dem das Fragment zugeordnet ist, momentan verdeckt ist (zum Beispiel durch das Fenster einer anderen Anwendung), wird das Fragment verworfen. Wenn nicht in den sichtbaren Framebuffer gerendert wird, sondern in ein von der OpenGL ES-Anwendung erzeugtes Framebuffer-Objekt, entfällt dieser Test.

- Über dem zu rendernden Bild kann von der OpenGL ES-Anwendung ein sogenanntes Scissor-Rechteck definiert sein. Wenn der Pixel, dem das Fragment zugeordnet ist, außerhalb dieses Rechtecks liegt, wird das Fragment verworfen. Fall kein Scissor-Rechteck definiert ist, entfällt dieser Test.

- Im optionalen Stencil-Test wird anhand der Werte im Stencilbuffer entschieden, ob das Fragment verworfen werden muss oder nicht.

- Ähnlich wie im Stencil-Test wird im ebenfalls optionalen Depthtest anhand der Werte im Depthbuffer entschieden, ob das Fragment verworfen werden muss oder nicht.

Falls das Fragment nicht verworfen wurde, wird dessen Farbwert mit dem aktuellen Farbwert des ihm zugewiesenen Pixels kombiniert. Die Art der Kombination wird dabei von der OpenGL ES-Anwendung bestimmt (standardmäßig wird der im Pixel gespeicherte Farbwert ignoriert).

Optional wird der Farbwert des Fragments anschließend im Rahmen von Dithering nochmals durch das OpenGL ES-System verändert, bevor der Farbwert endgültig in den aktuellen Framebuffer übertragen wird. Abhängig von den Einstellungen, die die OpenGL ES-Anwendung zuvor gesetzt hat, wird dann das dem Fragment zugewiesene Pixel im Colorbuffer-, Stencilbuffer- oder Depthbuffer-Datenobjekt überschrieben.

2.6.3 Gepuffertes und ungepuffertes Rendering

Die Attribute *gepuffert* und *ungepuffert* beziehen sich auf die Art, wie die von der Renderpipeline verarbeiteten Vertex- bzw. Indexdaten gespeichert sind. Falls diese Daten in Vertexdaten-Objekten gespeichert sind, spricht man von gepuffertem Rendering. Falls sie in Vertex- bzw. Indexdatenarrays außerhalb des OpenGL ES-Systems gespeichert sind, spricht man von ungepuffertem Rendering. Der Vorteil von gepuffertem Rendering ist, dass die benötigten Vertex- bzw. Indexdaten nicht im Zuge des Renderings in das OpenGL ES-System übertragen werden müssen, sondern sich bereits dort befinden. Dies kann unter Umständen das Rendering gegenüber dem ungepufferten Fall beschleunigen.

2.7 Erweiterungen von OpenGL ES

Viele Hersteller von OpenGL ES-Systemen und auch das Khronos Konsortium[5] selbst haben Erweiterungen zum OpenGL ES-Standard veröffentlicht. Solche Erweiterungen werden online dokumentiert (siehe [Munshi und Leech 2010], Seite 170). Ein Überblick über die derzeit verfügbaren Erweiterungen findet sich auf der Internetseite der *Khronos API Registry* (siehe [The Khronos Group]).

Die Implementierung von OpenGL ES-Funktionalität, die nur im Rahmen solcher Erweiterungen spezifiziert worden ist, ist optional, das heißt man kann nicht sicher davon ausgehen, dass die Funktionalität einer Erweiterung von einem bestimmten OpenGL ES-System tatsächlich unterstützt wird. Bei zwei Arten von Erweiterungen ist die Wahrscheinlichkeit einer Unterstützung jedoch hoch:

- **EXT-Erweiterungen**: Es handelt sich dabei um Erweiterungen, die von mehreren Herstellern in ihren Produkten unterstützt werden.
- **OES-Erweiterungen**: Solche Erweiterungen sind häufig ehemalige EXT-Erweiterungen, die vom Khronos Konsortium für eine künftige Übernahme in den Kernstandard von OpenGL ES vorgesehen wurden. Auch bei solchen Erweiterungen kann mit einer breiten Unterstützung in Produkten verschiedener Hersteller gerechnet werden. Sie erfüllen somit die gleiche Funktion wie die ARB-Erweiterungen von Standard-OpenGL (vgl. [Segal u. a. 2010], Seite 453).

Eine Übersicht über die derzeit in der *Khronos API Registry* aufgeführten Erweiterungen für OpenGL ES 2.0 findet sich im Anhang in Kapitel 6.2.

[5] Das Khronos Konsortium ist die Institution, die unter anderem für die Weiterentwicklung des OpenGL- und OpenGL ES-Standards verantwortlich ist (vgl. [Trevett 2010]).

3 Methodisches Vorgehen

In diesem Kapitel wird erklärt, welche Problemfelder und Fragestellungen im Hinblick auf den in Kapitel 1 beschriebenen Hintergrund dieser Arbeit untersucht werden. Außerdem werden die gewählten Ansätze zur Beantwortung der Fragen beschrieben und die OpenGL ES-Befehle aufgelistet, die für die jeweiligen Problemfelder relevant sind.

Wie in Kapitel 1.2 erläutert, hängt die Erfüllbarkeit von Echtzeitgarantien für OpenGL ES-Befehle sowohl davon ab, dass deren Ausführung durch den GL-Server rechtzeitig beginnen kann, als auch davon, dass deren Ausführung auch rechtzeitig abgeschlossen werden kann. Dies wiederum ist abhängig von ihrer jeweiligen Laufzeit. Wenn sich die aufgrund irgendwelcher Faktoren verlängert, besteht die Gefahr, dass trotz ihres rechtzeitigen Aufrufs eine gegebene Echtzeitgarantie nicht erfüllt wird.

Aus Sicht von OpenGL ES 2.0 lassen sich drei Ressourcen unterscheiden, durch die die Ausführbarkeit oder die Laufzeit von OpenGL ES-Befehlen negativ beeinflusst werden können: GPU-Speicher, Bandbreite (für die Übertragung von Daten zwischen Hauptspeicher und GPU-Speicher) und die in Kapitel 2 beschriebene Renderpipeline selbst.

Die Verhalten von OpenGL ES-Systemen bei der Belegung des GPU-Speichers ist im Hinblick auf die Erfüllung von Echtzeitgarantien für die Ausführung von OpenGL ES-Befehlen insbesondere aus den folgenden beiden Gründen relevant:

- Im Zuge der Ausführung von Draw-Befehlen werden durch die Renderpipeline (unter anderem) zwei Arten von Daten verarbeitet: Vertexdaten und Texturen (vgl. Kapitel 2.4). Solche Daten werden in Datenobjekten gespeichert, die im GPU-Speicher abgelegt werden.[1] Wenn im GPU-Speicher jedoch nicht mehr genügend Platz zur Ablage benötigter Daten vorhanden ist, kann dies die Ausführung von Draw-Befehlen kritischer Anwendungen verhindern. Um gewährleisten zu können, dass ein solcher Fall nicht eintritt, muss verhindert werden können, dass der Speicherplatz zu knapp wird. Dafür wiederum muss vorhergesagt werden können, wie viel GPU-Speicher durch die Ausführung von OpenGL ES-Befehlen belegt wird, um notfalls die Ausführung solcher Befehle zu verweigern.
- Falls eine OpenGL ES-System über einen Eviction-Mechanismus verfügt, dann kann sich die Laufzeit eines Draw-Befehls verlängern, wenn es im Zuge seiner Ausführung zu einer Eviction-Kaskade kommt (vgl. Kapitel 2.4.2). Da über OpenGL ES nicht gesteuert werden kann, welche Datenobjekte von Eviction betroffen sind, lässt sich nicht vorhersagen, ob und ggf. um welchen Betrag sich die Laufzeit eines Draw-Befehls dadurch verlängert. Um gewährleisten zu können, dass ein solcher Fall nicht eintritt, muss ebenfalls verhindert werden, dass der Speicherplatz im GPU-Speicher so knapp wird, dass das OpenGL ES-System auf den Eviction-Mechanismus zurückgreifen muss. Dafür muss ebenfalls der Speicherbedarf von OpenGL ES-Befehlen vorhergesagt werden können.

Die Beschreibung der Untersuchungsmethoden zum Verhalten von OpenGL ES-Systems hinsichtlich der Speicherbelegung erfolgt in Abschnitt 3.1.

[1] Die für das Rendering benötigten Vertexdaten können von OpenGL ES-Programmen auch im Hauptspeicher abgelegt werden, vgl. Kapitel 2.6.3. Eine solche Möglichkeit besteht für Texturen jedoch nicht.

Das Verhalten des OpenGL ES-Systems hinsichtlich der Datenübertragung ist für den Fall relevant, wenn sich für die Ausführung eines kritischen Draw-Befehls bestimmte Daten im GPU-Speicher befinden müssen, die erst dorthin übertragen werden müssen (dies gilt sowohl für gepufferte als auch für ungepufferte Draw-Befehle). Um sicherstellen zu können, dass die Daten rechtzeitig im GPU-Speicher liegen, muss die Laufzeit des entsprechenden Datenübertragungsbefehls vorhergesagt werden können.

In dem Zusammenhang ist muss auch bekannt sein, wie sich das OpenGL ES-System hinsichtlich konkurrierender Datenübertragungsbefehle verhält. Wenn beispielsweise nach Beginn einer kritischen Datenübertragung eine Datenübertragungsbefehl einer unkritischen Anwendung übermittelt wird, muss dieser Befehl verzögert werden, wenn bekannt ist, dass das OpenGL ES-System konkurrierende Datenübertragungsbefehle nicht sequentiell ausführt, sondern die verfügbare Bandbreite auf die beiden Befehle aufteilt. In dem Fall könnte sich die Laufzeit des kritischen Befehls verlängern.

Es ist ebenfalls von Interesse, ob sich Datenübertragung und Rendering gegenseitig ausschließen. Wenn dies nicht der Fall ist, könnte der Draw-Befehl einer unkritischen Anwendung möglicherweise noch ausgeführt werden, wenn zur gleichen Zeit die Übertragung von Daten durchgeführt werden kann, die von einem nachfolgenden Draw-Befehl einer kritischen Anwendungen benötigt werden. Wenn dies nicht möglich ist, muss die Ausführung des Draw-Befehls der unkritischen Anwendung verzögert werden. Die Beschreibung der Untersuchungsmethoden zum Verhalten von OpenGL ES-Systems hinsichtlich der Datenübertragung erfolgt in Abschnitt 3.2.

Die Laufzeit von Draw-Befehlen selbst ist für die Erfüllung von Echtzeitgarantien für das Rendering von zentraler Bedeutung. Daneben muss aber auch bekannt sein, ob das OpenGL ES-System konkurrierende Draw-Befehle nebenläufig ausführen kann oder nicht. Wenn es dazu in der Lage ist, muss damit gerechnet werden, dass sich die Laufzeiten der nebenläufig ausgeführten Draw-Befehle erhöhen.

Es muss allerdings damit gerechnet werden, dass sich aufgrund der Verwendung von Shadern nicht allgemein vorhersagen lässt, wie lange die Ausführung eines Draw-Befehls dauern wird. Falls dies der Fall ist, ist es wichtig zu wissen, ob laufende Draw-Befehle unkritischer Anwendungen notfalls abgebrochen werden können, um die rechtzeitige Ausführung kritischer Draw-Befehle sicherstellen zu können. Die Beschreibung der Untersuchungsmethoden zum Verhalten von OpenGL ES-Systems hinsichtlich der Nutzung der Renderpipeline erfolgt in Abschnitt 3.3.

Von manchen OpenGL ES-Systemen ist bekannt, dass sich die Laufzeit von OpenGL ES-Befehlen aufgrund von Kontextwechseln erhöhen kann, wenn sie unmittelbar nach einem OpenGL ES-Befehl einer anderen Anwendung ausgeführt werden. Zur Erfüllung von Echtzeitgarantien ist es notwendig, diese zusätzlichen Laufzeitkosten zu kennen und zu berücksichtigen. Die Beschreibung der Untersuchungsmethoden zum Verhalten von OpenGL ES-Systems hinsichtlich Kontextwechseln und zur Bestimmung ihrer Kosten erfolgt in Abschnitt 3.4.

Nach der Beschreibung der jeweiligen Untersuchungsmethoden eines Problemfelds werden diejenigen OpenGL ES-Befehle aufgeführt, die für die Untersuchungen relevant sind. OpenGL ES 2.0 definiert insgesamt 142 Befehle. Es müssen dabei aber nicht alle Untersuchungen für jeden relevante Befehl durchgeführt werden. Dies hat im Einzelfall unterschiedliche Gründe:

- Es gibt eine Reihe von Befehlen, von denen angenommen werden kann, dass sie weder einen Einfluss auf die Ressourcen der GPU haben, noch selbst durch den Zustand dieser Ressourcen beeinflusst werden. So definiert OpenGL ES zum Beispiel die Möglichkeit, den Namen des Herstellers der verwendeten OpenGL ES-Implementierung abzufragen. Es erscheint sehr unwahrscheinlich, dass das OpenGL ES-System dafür GPU-Speicher alloziert, Daten vom Hauptspeicher in den GPU-Speicher überträgt oder etwas rendert.
- Viele Befehle weisen eine sehr ähnliche Funktionalität auf. So definiert OpenGL ES 2.0 beispielsweise sechs Befehle, durch die Daten beliebiger Größe[2] im Zuge eines einzigen Befehlsaufrufs vom Hauptspeicher in die GPU übertragen werden können. Es ist anzunehmen, dass sich ein OpenGL ES-System hinsichtlich der Datenübertragung dieser Befehle nicht unterschiedlich verhalten wird. Die Untersuchungen werden in dieser Arbeit bei solchen Gruppen von Befehlen daher nur für einzelne Befehle tatsächlich durchgeführt.
- Eine Reihe von Befehlen kann schon aufgrund ihrer Definition nur einen sehr begrenzten Einfluss auf die Ressourcen der GPU haben. Durch den Aufruf eines Vertreters der Gruppe der `glVertexAttrib`-Befehle können zum Beispiel höchstens 16 Byte an die GPU übertragen werden. Auf die Durchführung der Untersuchungen für solche Befehle wird im Rahmen dieser Arbeit ebenfalls verzichtet.

Im folgenden Abschnitt wird erläutert, warum die Untersuchung der Speicherbelegung im Hinblick auf den Hintergrund der Arbeit wichtig ist. Anschließend werden die genauen Fragestellungen für diese Untersuchung sowie die Ansätze zu deren Beantwortung erörtert und die dafür relevanten OpenGL ES-Befehle vorgestellt.

3.1 Speicherbelegung

3.1.1 Motivation für die Untersuchung der Speicherbelegung

Wie bereits zu Beginn von Kapitel 3 umrissen, muss genau bekannt sein, wann und wie das über OpenGL ES verwendete System Daten im GPU-Speicher ablegt (und ggf. daraus entfernt), um Echtzeitgarantien für die Ausführung von Draw-Befehlen erfüllen zu können.

Die Spezifikation von OpenGL ES 2.0 definiert den Fehlercode `GL_OUT_OF_MEMORY`. Dieser Fehlercode wird von `glGetError` zurückgeliefert, nachdem ein Befehl aufgrund mangelnden Speicherplatzes nicht erfolgreich ausgeführt werden konnte (siehe [Munshi und Leech 2010], Seite 15). Da aber OpenGL ES 2.0 gleichzeitig keine Vorschriften darüber macht, ob und unter welchen Umständen Daten in einem eventuell vorhandenen GPU-Speicher abgelegt werden, ist nicht sicher, dass dieser Fehlercode erzeugt wird, sobald im GPU-Speicher nicht mehr genug Platz vorhanden ist, um weitere Daten darin abzulegen. Es steht OpenGL-Systemen frei, in diesem Fall Daten im Hauptspeicher abzulegen oder Daten aus dem GPU-Speicher auszulagern, um Platz zu schaffen (dieser Auslagerungsvorgang wird in der Literatur auch als *Eviction* bezeichnet, siehe auch Kapitel 2.4.2).

[2] Die Spezifikation von OpenGL ES definiert selbst keine Obergrenze dafür, sie erlaubt aber, dass konkrete Implementierungen von OpenGL ES eine solche Obergrenze festlegen, zum Beispiel die maximale Größe von Texturobjekten (siehe [Munshi und Leech 2010], Seite 152).

Von GPUs der Hersteller Nvidia und ATI ist bekannt, dass es tatsächlich zu Eviction kommen kann (siehe [Stroyan 2009] für Nvidia und [Blackmer u. a. 2009] für ATI). Dadurch kann sich die Laufzeit von OpenGL-Befehlen verlängern, die Datenobjekte im GPU-Speicher erzeugen, wenn im zuge ihrer Ausführung erst durch Auslagerung bereits vorhandener Datenobjekte Platz für die neu zu erzeugenden Datenobjekte geschaffen werden muss. Unter bestimmten Umständen kann sich auch die Laufzeit von Draw-Befehlen verlängern, wenn dabei Datenobjekte verarbeitet werden müssen, die zuvor ausgelagert wurden (vgl. dazu die Diskussion zur Eviction-Kaskade in Kapitel 2.4.2).

Derzeit existieren keine Erweiterungen für OpenGL ES, die die Möglichkeit bieten, das Eviction-Verhalten zu beeinflussen (um beispielsweise zu steuern, welche Datenobjekte ausgelagert werden oder um bestimmte, kritische Datenobjekte davon auszunehmen). Sowohl Nvidia als auch ATI warnen ausdrücklich davor, OpenGL-Anwendungen zu entwickeln, die den GPU-Speicher in hohem Maße mit Daten füllen und haben daher für ihre GPUs OpenGL-Erweiterungen veröffentlicht, die es zumindest ermöglichen, den Füllstand des GPU-Speichers zur Laufzeit abzufragen, um eine solche Situation zu vermeiden (siehe ebenfalls [Stroyan 2009] für Nvidia-GPUs und [Blackmer u. a. 2009] für ATI-GPUs).

Die einzige Möglichkeit, Eviction zu vermeiden, besteht also darin, zu verhindern, dass der GPU-Speicher so stark belegt wird, dass benötigte Datenobjekte nicht mehr ohne Eviction erzeugt werden können. Dazu muss aber vorhergesagt werden können, wie stark die Belegung des GPU-Speichers durch die Ausführung eines OpenGL ES-Befehls ansteigen wird – und da dies durch OpenGL ES nicht festgelegt ist, muss das Belegungsverhalten eines konkreten OpenGL-Systems gesondert untersucht werden. Die dazu entwickelten Untersuchungsmethoden werden im nächsten Abschnitt beschrieben.

3.1.2 Untersuchungsmethoden zur Speicherbelegung

3.1.2.1 Ablage von Datenobjekten im GPU-Speicher

Da OpenGL ES 2.0 nicht festlegt, ob und wie Daten in einem eventuell vorhandenen GPU-Speicher abgelegt werden, bleibt dies den Herstellern von OpenGL ES-Systemen überlassen. Für die nachfolgenden Untersuchungen muss aber bekannt sein, wie dieses Verhalten in einem konkreten OpenGL ES-System implementiert wurde. Dabei muss insbesondere die Frage beantwortet werden, ob überhaupt über OpenGL ES erzeugte Datenobjekte im GPU-Speicher abgelegt werden. Falls dies prinzipiell möglich ist, muss geklärt werden, wie sichergestellt beziehungsweise gesteuert werden kann, dass ein bestimmtes Datenobjekt tatsächlich im GPU-Speicher abgelegt wird.

Für Vertexbuffer-Objekte können beispielsweise spezielle Parameter angegeben werden, *Usage Hints* genannt, die dem OpenGL ES-System einen Hinweis geben, wie diese Objekte durch die erzeugende Anwendung künftig genutzt werden (vgl. Kapitel 2.4.1). Es steht dem System frei, diese Parameter zu ignorieren, es könnte sie aber auch dazu nutzen, darüber zu entscheiden, ob Datenobjekte im GPU-Speicher oder im Hauptspeicher abgelegt werden.

Um zu überprüfen, inwieweit die einzelnen Parameter von Datenobjekten einen Einfluss darauf haben, ob deren Ablage im GPU-Speicher oder im Hauptspeicher erfolgt, wird ein OpenGL ES-Programm ausgeführt, das Datenobjekte verschiedener Größe mit allen Kombinationen der rele-

vanten Parameter erzeugt und überprüft, wo sie abgelegt werden. Diese Untersuchung erfolgt in Kapitel 4.2.1. Im nächsten Abschnitt wird die Untersuchung beschrieben, mit der festgestellt wird, ob der Speicherbedarf von Datenobjekten im GPU-Speicher von der tatsächlichen Menge der in ihnen gespeicherten Daten abweicht.

3.1.2.2 Speicherbedarf von Datenobjekten

Im Idealfall entspricht die Menge des von einem Datenobjekt belegten GPU-Speichers immer exakt der in den Datenobjekten gespeicherten Datenmenge. Die tatsächliche Menge belegten Speicherplatzes kann davon aber abweichen. Dies könnte der Fall sein,

- wenn zusätzlich zu den eigentlichen Nutzdaten der Datenobjekte auch bestimmte Metadaten[3] im GPU-Speicher abgelegt werden oder
- wenn mehrere Datenobjekte zusammen in größeren Speicherblöcken abgelegt werden oder
- wenn der GPU-Speicher eine Speichergranularität aufweist, die größer ist als ein Byte (siehe Kapitel 2.4.3 für eine Definintion von Speicherblock und Speichergranularität).

Um festzustellen, ob es zu Abweichungen zwischen belegtem GPU-Speicher und Größe der Datenobjekte kommt, wird ein OpenGL ES-Programm ausgeführt, das wie folgt vorgeht: Im leeren GPU-Speicher wird jeweils ein Datenobjekt angelegt und die Menge des dadurch belegten Speicherplatzes ermittelt. Dies wird für Datenobjekte unterschiedlicher Größe wiederholt, um festzustellen, für welche Datenobjektgrößen die Menge des belegten GPU-Speichers von der Datenobjektgröße abweicht.

Sofern es immer zu Abweichungen kommt, und die Größe dieser Abweichungen stets gleich groß ist, deutet dies auf die zusätzliche Speicherung von Metadaten hin. Falls die Größe der Abweichungen schwankt und für bestimmte Datenobjektgrößen gar keine Abweichung auftritt, deutet dies darauf hin, dass Datenobjekte in Speicherblöcken abgelegt werden oder dass der GPU-Speicher eine Granularität aufweist, die größer ist als ein Byte.

Um in diesem Fall unterscheiden zu können, um welchen Effekt es sich handelt, wird das im ersten Punkt beschriebene Vorgehen wiederholt, wobei für jede betrachtete Datenobjektgröße nicht nur ein einziges Datenobjekt im leeren GPU-Speicher angelegt wird sondern zusätzlich noch weitere. Wenn sich dabei zeigt, dass der zusätzlich zur Datenobjektgröße des ersten angelegten Datenobjekts belegte GPU-Speicher für die Ablage weiterer Datenobjekte genutzt wird, handelt es sich offensichtlich um einen Speicherblockeffekt.

Die in diesem Abschnitt beschriebene Untersuchung des Speicherbedarfs von Datenobjekten im GPU-Speicher erfolgt in Kapitel 4.2.2. Im nächsten Abschnitt wird die Untersuchung beschrieben, mit der die Größe eines Speicherblocks ermittelt wird.

3.1.2.3 Bestimmung der Speicherblockgröße

Wenn ein Datenobjekt angelegt werden soll, das vom OpenGL ES-System in einem größeren Speicherblock abgelegt werden wird, muss die Speicherblockgröße bekannt sein, um vorhersagen

[3] Metadaten sind Daten, die vom OpenGL ES-System zusätzlich zu den eigentlichen Nutzdaten der Datenobjekte gespeichert werden. Dies könnten beispielsweise Typ oder Größe eines Datenobjekts sein.

zu können, wie der freie GPU-Speicher durch die Erzeugung des Datenobjektes verringert wird. Wenn die Speicherblockgröße bekannt ist, dann kann von folgenden Annahmen ausgegangen werden:

- Falls ein neuer Speicherblock alloziert werden muss, dann wird sich der freie GPU-Speicher genau um den Wert der Speicherblockgröße verringern.
- Falls das neue Datenobjekt hingegen in einem bereits allozierten Speicherblock abgelegt werden kann, dann wird sich der freie GPU-Speicher nicht verringern.

Für die Untersuchung der Speicherblockgröße stehen die folgenden Fragen im Mittelpunkt:

- Wie groß ist ein Speicherblock, das heißt wieviel GPU-Speicher wird durch die Allozierung eines Speicherblocks belegt?
- Gibt es nur eine einzige Speicherblockgröße? Falls nein: Wovon hängt die Speicherblockgröße ab? In Frage kommen die Größe des Datenobjekts, durch dessen Erzeugung ein Speicherblock alloziert wird, oder die aktuelle Belegung des GPU-Speichers.

Um diese Fragen zu beantworten, wird ein OpenGL ES-Programm ausgeführt, das das folgende Vorgehen verfolgt: Im zu Beginn noch leeren GPU-Speicher werden sukzessive Datenobjekte gleicher Größe angelegt, bis der GPU-Speicher gefüllt ist. Dabei wird nach jedem Anlegen eines Datenobjekts die Menge des aktuell belegten GPU-Speichers ermittelt. Dieses Vorgehen wird für unterschiedliche Datenobjektgrößen wiederholt (wobei nur solche Datenobjektgrößen zu berücksichtigen sind, für die in der vorherigen Untersuchung festgestellt wurde, dass es zur Ablage in Speicherblöcken kommt). Es steht zu erwarten, dass dieses Vorgehen für jede berücksichtigte Datenobjektgröße zu einem der folgenden Ergebnisse führen wird:

- Die Menge des belegten Speichers steigt nicht nach jeder Objekterzeugung an, sondern immer nur dann, wenn seit dem letzten Anstieg der Speicherbelegung eine bestimmte Menge an weiteren Daten im GPU-Speicher angelegt worden ist.
- Es kommt bei jeder Objekterzeugung zu einem Anstieg der Speicherbelegung, wobei dieser Anstieg mindestens der Datenobjektgröße entspricht.

Die Größe der gemessenen Anstiege entspricht dann der gesuchten Speicherblockgröße. Durch dieses Vorgehen wird auch aufgedeckt, ob es nur eine einzige Speicherblockgröße gibt und falls nicht, ob die jeweilige Speicherblockgröße von der Größe der erzeugten Datenobjekte oder von der aktuellen Speicherbelegung abhängt.

Die in diesem Abschnitt beschriebene Untersuchung zur Bestimmung der Speicherblockgröße erfolgt in Kapitel 4.2.3. Der nächste Abschnitt beschreibt die Untersuchung, auf welche Weise Datenobjekte in Speicherblöcken abgelegt werden.

3.1.2.4 Belegungsverhalten innerhalb von Speicherblöcken

Um vorhersagen zu können, ob durch die Erzeugung eines Datenobjekts ein neuer Speicherblock alloziert wird, muss bekannt sein, unter welchen Bedingungen Datenobjekte gegebenenfalls in bereits allozierten Speicherblöcken abgelegt werden. Dabei stehen die folgenden Fragen im Mittelpunkt:

- Aus der Definition eines Speicherblocks ergibt sich: Wenn ein bereits allozierter Speicherblock theoretisch noch X Byte Platz für die Ablage weiterer Datenobjekte aufweist, dann wird ein Datenobjekt, das kleiner ist als X, in diesem Speicherblock abgelegt (sofern dessen Größe mit der Größe der bereits im Speicherblock befindlichen Datenobjekte übereinstimmt). Doch was geschieht, falls ein Datenobjekt größer ist als X? Wird es dann teilweise im bereits allozierten Speicherblock abgelegt oder wird es komplett in einem neuen Speicherblock abgelegt?
- Werden nur Datenobjekte gleicher Größe im selben Speicherblock abgelegt?
- Werden Datenobjekte nur dann in bereits allozierten Speicherblöcken abgelegt, wenn sie unmittelbar aufeinander folgend erzeugt werden, das heißt wird ein Datenobjekt nur dann in einem bereits allozierten Speicherblock abgelegt, wenn das unmittelbar zuvor erzeugte Datenobjekt schon im entsprechenden Speicherblock abgelegt worden ist?
- Werden Datenobjekte in fragmentierten Speicherblöcken abgelegt?[4]
- Werden Datenobjekte verschiedener Prozesse in jeweils eigenen Speicherblöcken abgelegt?

Um diese Fragen zu klären, werden OpenGL ES-Programme ausgeführt, deren Vorgehen in den folgenden vier Abschnitten im einzelnen beschrieben wird:

3.1.2.4.1 Aufteilung von Datenobjekten auf mehrere Speicherblöcke

Um zu überprüfen, ob Datenobjekte, die nicht vollständig in den noch freien Platz eines bereits allozierten Speicherblocks passen, auf mehrere Speicherblöcke aufgeteilt werden, oder ob in dem Fall ein neuer Speicherblock alloziert wird, werden nacheinander mehrere Datenobjekte angelegt, deren Größe genau drei Vierteln der Speicherblockgröße entspricht.

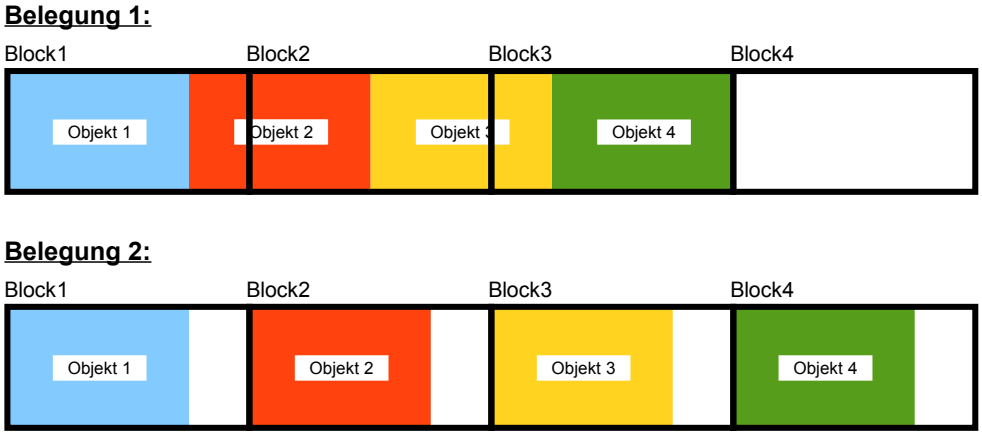

Abbildung 3.1: Ablagemöglichkeiten von vier Datenobjekten mit $\frac{3}{4}$ der Speicherblockgröße

Zeigt sich, dass nach der Erzeugung von vier Datenobjekten drei Speicherblöcke alloziert werden, kann davon ausgegangen werden, dass Datenobjekte auf mehrere Speicherblöcke aufgeteilt wer-

[4] Unter Fragmentierung wird hier verstanden, dass es innerhalb eines Speicherblocks ungenutzten Speicher zwischen benutzten Speicherbereichen gibt.

den (dies entspräche der Belegung 1 in Abbildung 3.1). Wenn stattdessen vier Speicherblöcke alloziert werden, dann ist klar, dass ein neuer Speicherblock alloziert wird, wenn ein Datenobjekt nicht mehr vollständig in einen bereits vorhandenen passt (dies entspräche der Belegung 2 in Abbildung 3.1).

3.1.2.4.2 Ablage von Datenobjekten unterschiedlicher Größe im selben Speicherblock

Zur Überprüfung, ob Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt werden können, werden im leeren GPU-Speicher mehrere Datenobjekte unterschiedlicher Größe angelegt, wobei deren Gesamtgröße der Speicherblockgröße entspricht. Steigt dabei die Speicherbelegung nur um den Wert der Speicherblockgröße an, kann davon ausgegangen werden, dass Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt werden können.

3.1.2.4.3 Nichtsequentielle Ablage von Datenobjekten in Speicherblöcken

Um zu überprüfen, ob Datenobjekte auch dann im selben Speicherblock abgelegt werden können, wenn sie nicht unmittelbar aufeinander folgend erzeugt werden, wird zunächst ein Speicherblock teilweise gefüllt (zum Beispiel durch Erzeugung von zwei Datenobjekten mit jeweils $\frac{3}{8}$ der Speicherblockgröße, so dass der Speicherblock zu $\frac{3}{4}$ gefüllt ist). Anschließend wird ein weiterer Speicherblock komplett gefüllt (zum Beispiel durch Erzeugung von zwei Datenobjekten mit $\frac{1}{2}$ der Speicherblockgröße, so dass diese Datenobjekte nicht im ersten Speicherblock abgelegt werden). Schließlich ein Datenobjekt angelegt, das theoretisch im ursprünglichen Speicherblock Platz finden könnte (in diesem Beispiel also mit $\frac{1}{4}$ der Speicherblockgröße). Kann dabei kein Anstieg der Speicherbelegung festgestellt werden, kann davon ausgegangen werden, dass das betreffende Datenobjekt tatsächlich im ursprünglichen Speicherblock abgelegt wurde.

Dieses Vorgehen wird wiederholt, wobei in jedem Schritt die Anzahl der teilweise gefüllten Speicherblöcke erhöht wird und anschließend überprüft wird, ob alle teilweise gefüllten Speicherblöcke komplett aufgefüllt werden können. Dadurch kann ermittelt werden, ob eine nicht aufeinander folgende Datenobjektablage nur im jeweils zuletzt teilweise gefüllten Speicherblock (beziehungsweise in den N letzten teilweise gefüllten Speicherblöcken) erfolgt.

Voraussetzung für diese Untersuchung ist, dass Datenobjekte nur vollständig in einem Speicherblock abgelegt werden und Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt werden können. Falls dies nicht möglich ist, kann diese Untersuchung nicht durchgeführt werden.

3.1.2.4.4 Ablage von Datenobjekten in fragmentierten Speicherblöcken

Um das Ablageverhalten im Falle fragmentierter Speicherblöcke zu überprüfen, wird zunächst ein Speicherblock mit mehreren Datenobjekten komplett gefüllt. Anschließend wird jedes zweite Datenobjekt wieder freigegeben, so dass innerhalb des Speicherblocks Belegungslücken entstehen.[5] Schließlich werden neue Datenobjekte erzeugt, so dass theoretisch alle Belegungslücken damit gefüllt werden könnten. Kann dabei kein Anstieg der Speicherbelegung festgestellt werden, kann davon ausgegangen werden, dass die neu angelegten Datenobjekte tatsächlich innerhalb des frag-

[5] Die Annahme dahinter ist, dass im leeren GPU-Speicher aufeinander folgend erzeugte Datenobjekte im GPU-Speicher tatsächlich lückenlos hintereinander abgelegt werden.

mentierten Speicherblocks abgelegt worden sind. Dieses Vorgehen wird mehrmals wiederholt, wobei in jedem Schritt die Größe der Datenobjekte halbiert wird, was dazu führt dass sich die Anzahl der Lücken im Speicherblock verdoppelt.

Die in diesem hier beschriebenen Untersuchungen der Ablage von Datenobjekten in Speicherblöcken erfolgt in Kapitel 4.2.4. Im nächsten Abschnitt wird die Untersuchung beschrieben, mit der die Speichergranularität bestimmt wird.

3.1.2.5 Bestimmung der Speichergranularität

Um die Menge des GPU-Speichers vorherzusagen, der durch die Erzeugung eines Datenobjekts belegt werden wird, ist es notwendig den Wert der Speichergranularität zu bestimmen. Dazu wird ein OpenGL ES-Programm ausgeführt, das folgendermaßen vorgeht:

Im leeren GPU-Speicher wird ein Datenobjekt angelegt und die Menge des dadurch belegten GPU-Speichers ermittelt. Dies wird für Datenobjekte zunehmender Größe wiederholt. Ergibt sich dabei ein Treppeneffekt in der Speicherbelegung, das heißt steigt die Menge des belegten GPU-Speichers immer nur bei bestimmten Datenobjektgrößen an und handelt es sich bei diesen Datenobjektgrößen immer um ein ganzzahliges Vielfaches des gleichen Wertes, dann entspricht dieser Wert der Speichergranularität.

Es ist für diese Untersuchung unerheblich, ob pro Messschritt nur ein Datenobjekt oder mehrere erzeugt werden. Dies kann hilfreich sein, wenn Datenobjekte einer bestimmten Größe von der Ablage in Speicherblöcken betroffen sind. In dem Fall ist durch die Erzeugung eines einzelnen Datenobjekts nicht erkennbar, wieviel des belegten GPU-Speichers vom erzeugten Datenobjekt genutzt wird und wieviel für nachfolgende Datenobjekte noch genutzt werden könnte.

Mithin kann auch nicht festgestellt werden ob die Menge des vom Datenobjekt genutzten GPU-Speichers von seiner Größe abweicht. Wenn aber eine große Anzahl solcher Datenobjekte angelegt wird, dann kann aus der Anzahl der dadurch belegten Speicherblöcke darauf geschlossen werden. Wird dies für verschiedene Datenobjektgrößen wiederholt, dann kann die Speichergranularität auf die gleiche Weise bestimmt werden, wie bei der Anlage nur eines Datenobjekts pro Messschritt, das nicht in einem Speicherblock abgelegt wird.

Dazu ein Beispiel: Die Speicherblockgröße in einem System betrage vier Megabyte und die Speichergranularität 128 Byte. Würden dann pro Messschritt 2^{16} Datenobjekte angelegt, so würden für alle Datenobjektgrößen bis einschließlich 128 Byte genau zwei Speicherblöcke belegt ($2^{16} * 128$ Byte = 8 MB). Ab einer Datenobjektgröße von 129 Byte bis einschließlich 256 Byte würden vier Speicherblöcke belegt, von 257–384 Byte sechs Speicherblöcke, ... Die Menge der belegten Speicherblöcke würde sich also immer nach Überschreiten einer Datenobjektgröße erhöhen, die einem ganzzahligen Vielfachen von 128 Byte entspricht.

Die in diesem Abschnitt beschriebene Untersuchung zur Bestimmung der Speichergranularität erfolgt in Kapitel 4.2.5. In den vergangenen Abschnitten wurden alle untersuchten Fragestellungen hinsichtlich der Speicherbelegung und die Ansätze zu deren Beantwortung detailliert erörtert. Im nächsten Abschnitt werden sämtliche OpenGL ES-Befehle aufgelistet, die einen Einfluss auf die Belegung des GPU-Speichers haben können, und es wird dargelegt, für welche dieser Befehle die Untersuchungen zur Speicherbelegung in Kapitel 4 durchgeführt werden.

3.1.3 Relevante Befehle für die Speicherbelegung

Für die Belegung des GPU-Speichers sind vier Gruppen von OpenGL ES-Befehlen relevant: Befehle zur Erzeugung von Datenobjekten (siehe Abschnitt 3.1.3.1), Befehle zur Freigabe von Datenobjekten (siehe Abschnitt 3.1.3.2), Befehle zur Erzeugung und Verwendung von Programmobjekten (siehe Abschnitt 3.1.3.3) und Draw-Befehle mit ungepufferten Vertexdaten (siehe 3.1.3.4). Ein umfassender Überblick über alle von OpenGL ES 2.0 definierten Befehle mit einer kurzen Beschreibung findet sich im Anhang in Kapitel 6.1.

3.1.3.1 Befehle zur Erzeugung von Datenobjekten

OpenGL ES 2.0 definiert die folgenden sieben Befehle, durch die Datenobjekte erzeugt werden können (siehe Kapitel 2.4.1 für eine Beschreibung der verschiedenen Datenobjekte von OpenGL ES):

- `glBufferData` zur Erzeugung von Vertexbuffer-Objekten,
- `glRenderbufferStorage` zur Erzeugung von Renderbuffer-Objekten,
- `glBindFramebuffer` zur Erzeugung von Framebuffer-Objekten sowie
- `glTexImage2D`, `glCompressedTexImage2D`, `glCopyTexImage2D` und `glGenerateMipmap` zur Erzeugung von Texturobjekten.

Diese sieben Befehle sind im Anhang in Kapitel 6.1.1 ausführlicher beschrieben. Von ihnen werden im Rahmen dieser Arbeit nur für den Befehl `glBufferData` die Untersuchungen zur Speicherbelegung durchgeführt. Die Wahl fällt aus den folgenden Gründen speziell auf diesen Befehl:

- `glBufferData` ist unter allen Befehlen zur Erzeugung von Datenobjekten der einzige, für den zusätzlich zur Größe des Datenobjekts auch *Usage Hints* angegeben werden können, durch die das Verhalten des OpenGL ES-Systems hinsichtlich der Speicherbelegung beeinflusst werden könnte. Eine solche Einflussmöglichkeit besteht für alle anderen Datenobjekte nicht (vgl. Kapitel 2.4.1).
- Außerdem bestehen keine Einschränkungen hinsichtlich der Größe der durch `glBufferData` erzeugten Vertexbuffer-Objekte (mit Ausnahme einer systemspezifischen Maximalgröße, was aber für alle Datenobjekte gilt). Texturobjekte sind auf Größen beschränkt, die sich als Produkt aus zwei Zweierpotenzen ausdrücken lassen (siehe [Munshi und Leech 2010], Seite 67); Framebuffer-Objekte haben stets die gleiche Größe.
- Da durch alle sieben Befehle essentiell der gleiche Vorgang im Hinblick auf den GPU-Speicher durchgeführt wird – es wird eine bestimmte Menge GPU-Speicher belegt – ist anzunehmen, dass sich das OpenGL ES-System hinsichtlich des GPU-Speichers jeweils gleich verhalten wird.

3.1.3.2 Befehle zur Freigabe von Datenobjekten

Zur Freigabe von Datenobjekten definiert OpenGL ES 2.0 die folgenden vier Befehle:

- `glDeleteBuffers` zur Freigabe von Vertexbuffer-Objekten,
- `glDeleteRenderbuffers` zur Freigabe von Renderbuffer-Objekten,

- `glDeleteTextures` zur Freigabe von Texturobjekten und
- `glDeleteFramebuffers` zur Freigabe von Framebuffer-Objekten.

Diese vier Befehle sind im Anhang in Kapitel 6.1.2 ausführlicher beschrieben. Sie haben insofern einen erheblichen Einfluss auf die Speicherbelegung, als sich durch ihren Aufruf die Menge des verfügbaren GPU-Speichers erhöhen kann. Da durch einen solchen Vorgang jedoch keine *negative* Auswirkungen im Hinblick auf Echtzeitgarantien zu erwarten sind (der verfügbare Speicherplatz verringert sich nicht) werden diese Befehle bei Bedarf verwendet aber nicht weitergehend untersucht.

3.1.3.3 Relevante Befehle zu Programmobjekten

Von den im Anhang in Kapitel 6.1.11 näher beschriebenen Befehlen zu Programmobjekten, können die folgenden beiden die Speicherbelegung beeinflussen:

- `glLinkProgram`, der aus einem Vertex- und einem Fragmentshader ein ausführbares Programmobjekt erzeugt, das für nachfolgende Renderoperationen genutzt werden kann, und
- `glUseProgram`, der ein erzeugtes Programmobjekt im aktuellen Kontext installiert, so dass es tatsächlich für nachfolgende Renderoperationen genutzt wird.

Da Shaderprogramme innerhalb der GPU ausgeführt werden, erscheint es möglich, dass Programmobjekte innerhalb des GPU-Speichers abgelegt werden. Wenn dies der Fall ist, könnten sie prinzipiell bereits nach Ausführung von `glLinkProgram` in den GPU-Speicher übertragen werden. Sie könnten aber auch erst dann übertragen werden, wenn sie explizit durch Aufruf von `glUseProgram` zur Verwendung ausgewählt werden.

Die beiden hier aufgeführten Befehle könnten also für die Untersuchungen zur Speicherbelegung relevant sein, insbesondere da bereits GPUs existieren, bei denen die Größe von Programmobjekten nicht begrenzt ist, zum Beispiel die bereits im Jahr 2006 auf den Markt gebrachte Nvidia GeForce 8800 (siehe [Nvidia 2006], Seite 40).

Typischerweise sind Programmobjekte jedoch sehr klein (siehe [Hill u. a. 2008], Seite 1), was für Datenobjekte – mit Ausnahme von Framebuffer-Objekten – nicht gilt. Aus diesem Grund wurden die Untersuchungen zur Speicherbelegung nicht für Programmobjekte durchgeführt, sondern für Datenobjekte.

3.1.3.4 Draw-Befehle

Beide von OpenGL ES 2.0 definierten Draw-Befehle (`glDrawArrays` und `glDrawElements`) können dazu genutzt werden, eine Renderoperation mit ungepufferten Vertex- bzw. Indexdaten anzustoßen (siehe Kapitel 2.6.3 für eine Diskussion der Unterschiede von gepuffertem und ungepuffertem Rendering und Kapitel 6.1.8 für eine ausführlichere Beschreibung der beiden Draw-Befehle).

Diese beiden Befehle sind tatsächlich nur im ungepufferten Fall für die Untersuchung der Speicherbelegung relevant. In diesem Fall liegen die im Zuge ihrer Ausführung zu verarbeitenden Vertex-

und ggf. Indexdaten im Hauptspeicher und müssen erst in die GPU übertragen werden. Es ist dabei prinzipiell möglich, dass diese Daten teilweise oder sogar komplett im GPU-Speicher zwischengespeichert werden müssen, bevor das eigentliche Rendering durchgeführt werden kann (frühere Untersuchungsergebnisse deuten darauf hin, dass dies für manche GPUs tatsächlich nicht der Fall ist, vgl. [Grottel u. a. 2009], Seite 71). Dadurch kann es zu Eviction kommen.

Da anzunehmen ist, dass die Übertragung ungepufferter Vertexdaten in den GPU-Speicher im Prinzip der Übertragung von Vertexdaten durch `glBufferData` entspricht, wird der Einfluss ungepufferter Draw-Befehle im Zusammenhang mit der Untersuchung der Speicherbelegung nicht gesondert untersucht. Diese Befehle werden aber im Zusammenhang mit der Untersuchung der Datenübertragung in Kapitel 4.3.4 näher behandelt.

Dort wird überprüft, ob die Übertragung ungepufferter Vertexdaten zeitgleich zum Rendering durchgeführt wird. Für GPUs, bei denen dies der Fall ist, kann davon ausgegangen werden, dass ungepufferte Draw-Befehle tatsächlich keinen Einfluss auf die Speicherbelegung haben, während im gegenteiligen Fall davon ausgegangen werden muss.

Im nun folgenden Abschnitt wird erläutert, warum die Untersuchung der Datenübertragung im Hinblick auf den Hintergrund der Arbeit sehr wichtig ist. Anschließend werden die genauen Fragestellungen für diese Untersuchung sowie die Ansätze zu deren Beantwortung erörtert und die dafür relevanten OpenGL ES-Befehle vorgestellt.

3.2 Datenübertragung

3.2.1 Motivation für die Untersuchung der Datenübertragung

Wie zu Beginn von Kapitel 3 dargelegt, ist es im Hinblick auf die Erfüllung von Echtzeitgarantien wichtig, die Laufzeit von Datenübertragungsbefehlen vorhersagen zu können. Um die Laufzeit von Befehlen vorhersagen zu können, die Daten übertragen (vom Hauptspeicher in den GPU-Speicher, in umgekehrter Richtung und innerhalb des GPU-Speichers), muss neben der zu übertragenden Datenmenge und einem eventuell anfallenden Laufzeit-Overhead auch die Datenübertragungsrate des verwendeten Systems bekannt sein (das heißt welche Datenmenge pro Zeitintervall übertragen wird).

Das allein reicht aber nicht aus, um die Laufzeit in allen Fällen zuverlässig vorhersagen zu können. Dafür muss – insbesondere im Hinblick auf den Hintergrund der Arbeit – auch bekannt sein, wie sich das verwendete System gegenüber konkurrierenden Datenübertragungsbefehlen verhält, (das heißt gegenüber Datenübertragungsbefehlen, die von verschiedenen Anwendungen an den GL-Server übermittelt werden und sich zeitlich überschneiden).

Außerdem muss bekannt sein, ob das verwendete System in der Lage ist, Datenübertragungen durchzuführen, während gleichzeitig ein Draw-Befehl ausgeführt wird. Einen Sonderfall stellt hierbei die Datenübertragung im Rahmen der Ausführung eines ungepufferten Draw-Befehls dar, da hier die Ausführung des Draw-Befehls von den zu übertragenden Daten direkt abhängt (siehe Kapitel 2.6.3 für eine Diskussion der Unterschiede zwischen gepufferter und ungepufferter Ausführung von Draw-Befehlen). Dies muss daher gesondert untersucht werden.

Im folgenden Abschnitt werden die genauen Fragestellungen näher erläutert, die im Zuge der Untersuchung der Datenübertragung beantwortet werden müssen, und dabei auch die Ansätze zu deren Beantwortung beschrieben. Anschließend werden die für diese Untersuchung relevanten OpenGL ES-Befehle aufgelistet.

3.2.2 Untersuchungsmethoden zur Datenübertragung

3.2.2.1 Bestimmung von Datenübertragungsrate und -laufzeit

Hinsichtlich der Datenübertragungsrate sind die folgenden Fragestellungen zu beantworten:

- Ist die Laufzeit von Datenübertragungsbefehlen immer proportional zur übertragenen Datenmenge, oder hängt sie auch von anderen Faktoren ab? Als Faktoren kommen die Menge des aktuell belegten GPU-Speichers oder die Anzahl bereits im GPU-Speicher vorhandener Datenobjekte in Frage.
- Wie groß ist der Laufzeit-Overhead für die Ausführung eines Datenübertragungsbefehls, der unabhängig von der übertragenen Datenmenge zur Laufzeit hinzukommt? Ist dieser Overhead konstant oder ändert er sich abhängig vom Belegungsgrad des GPU-Speichers oder abhängig von der Anzahl bereits vorhandener Datenobjekte?
- Falls der Belegungsgrad des GPU-Speichers oder die Menge bereits vorhandener Datenobjekte einen Einfluss auf die Laufzeiten hat, hängt dieser Einfluss dann davon ab, ob die Belegung des GPU-Speichers beziehungsweise die Erzeugung der Datenobjekte vom Kontext eines anderen Prozesses erfolgt ist?

Zur Beantwortung dieser Fragen wird ein OpenGL ES-Programm ausgeführt, das folgendes Vorgehen verfolgt: Zunächst wird das Verhalten von Datenübertragungsbefehlen[6] bei leerem GPU-Speicher untersucht. Dazu wird jeweils ein Datenobjekt angelegt, die Laufzeit für die Datenübertragung gemessen und das Datenobjekt anschließend wieder entfernt. Dies wird mehrmals wiederholt, wobei die Größe des erzeugten Datenobjekts stets erhöht wird. Diese erste Untersuchung liefert die Referenzwerte für die nachfolgenden Untersuchungen.

Anschließend wird die Frage geklärt, ob die Laufzeit der Datenübertragung vom aktuellen Belegungsgrad des GPU-Speichers abhängt. Dazu wird der GPU-Speicher vorab zu einem bestimmten Prozentsatz belegt und anschließend die gleichen Messungen wie in der ersten Untersuchung durchgeführt. Dies wird mit steigenden Speicherbelegungsgraden wiederholt. Diese Untersuchung wird in zwei Varianten durchgeführt: In der einen Variante wird der GPU-Speicher über den selben Kontext belegt, mit dem danach auch die Laufzeitmessungen durchgeführt werden. In der anderen Variante wird der Speicher über einen Kontext belegt, der von einem anderen Prozess erzeugt wurde.

Die Frage, ob die Laufzeit von Datenübertragungsbefehlen von der Anzahl der bereits erzeugten Datenobjekte beeinflusst wird, wird durch eine weitere Untersuchung geklärt. Dabei wird vorab eine gewisse Menge sehr kleiner Datenobjekte erzeugt, so dass der Speicherbelegungsgrad immer noch sehr niedrig ist. Dann werden die gleichen Messungen wie in der ersten Untersuchung durchgeführt. Wie bei der zweiten Untersuchung werden auch hier zwei Varianten durchgeführt, wo-

[6] Siehe Abschnitt 3.2.3 für eine Auflistung der dafür relevanten Befehle.

bei in der einen die kleinen Datenobjekte über den selben Kontext erzeugt werden, mit dem auch die Messungen durchgeführt werden, während in der anderen Variante die Datenobjekte über den Kontext eines anderen Prozesses erzeugt werden.

Der Laufzeit-Overhead für die Ausführung von Datenübertragungsbefehlen wird durch die Messung der Laufzeiten für die Übertragung von sehr kleinen Datenmengen bestimmt. Die Annahme dahinter ist folgende: Bei einer Datenübertragungsrate von mehreren Gigabyte pro Sekunde ist die Übertragungszeit von wenigen Byte großen Datenmengen praktisch nicht mehr messbar (die Zeit zur Übertragung von einem Kilobyte Daten würde bei einer Bandbreite von mehr als einem Gigabyte pro Sekunde weniger als eine Mikrosekunde betragen). Wenn die gemessenen Laufzeiten für immer kleinere Datenmengen aber nicht gegen Null gehen sondern gegen einen anderen Wert, so entspricht dieser Wert dem zu bestimmenden Overhead.

Die in diesem Abschnitt beschriebenen Untersuchungen zur Laufzeit von Datenübertragungsbefehlen erfolgen in Kapitel 4.3.1. Im nächsten Abschnitt wird die Untersuchung des Verhaltens des OpenGL ES-Systems bei konkurrierenden Datenübertragungen beschrieben.

3.2.2.2 Konkurrierende Datenübertragungen

Grundsätzlich gibt es zwei Möglichkeiten, wie ein konkretes OpenGL ES-System Datenübertragungen durchführt, die sich zeitlich überschneiden:

- Die verfügbare Bandbreite wird auf mehrere Datenübertragungsbefehle aufgeteilt, so dass kein Datenübertragungsbefehl verzögert wird, bis ein anderer vollständig ausgeführt wurde. Dadurch verlängert sich aus Sicht der Client-Programme die Laufzeit aller betroffenen Datenübertragungsbefehle.
- Die Datenübertragungsbefehle werden sequentiell ausgeführt, das heißt, solange ein Datenübertragungsbefehl noch nicht vollständig ausgeführt worden ist, wird der Ausführungsbeginn anderer Datenübertragungsbefehle verzögert. Dadurch verlängert sich aus Sicht der Client-Programme nur deren Laufzeit.

Um die Laufzeit von Datenübertragungsbefehlen in einem konkreten System vorhersagen zu können, muss also geklärt werden, welche dieser beiden Möglichkeiten durch das System umgesetzt wird. Dazu werden zwei OpenGL ES-Programme ausgeführt, wobei die Ausführung des einen Programms durch das andere Programm bestimmt wird. Zur Vereinfachung der Diskussion darüber, wird daher das bestimmende Programm nachfolgend als *Masterprogramm* bezeichnet und das bestimmte als *Slaveprogramm*. Die beiden Programme gehen wie folgt vor:

Zuerst wird das Masterprogramm gestartet. Dieses misst – als Vorbereitung für die eigentliche Untersuchung – die Laufzeiten des zu untersuchenden Datenübertragungsbefehls für unterschiedliche zu übertragende Datenmengen. Die für diese Untersuchung relevanten Datenübertragungsbefehle werden in Kapitel 3.2.3 aufgeführt – durch das hier beschriebene Vorgehen wird jeweils nur einer dieser Befehle untersucht. Dabei ist darauf zu achten, dass keine anderen Programme Datenübertragungen durchführen, während das Masterprogramm diese Messungen vornimmt. Die dabei ermittelten Laufzeiten dienen dann später als Referenz- und Vergleichswerte.

Nachdem das Masterprogramm seine Referenzmessung beendet hat, startet es das Slaveprogramm und geht in einen Wartezustand über. Das Slaveprogramm führt nun die gleiche Referenzmes-

sung durch wie das Masterprogramm. Die von den beiden Programmen ermittelten Referenzwerte müssen übereinstimmen, um sicherzustellen, dass das Slaveprogramm vom OpenGL ES-System gleich behandelt wird wie das Masterprogramm. Das ist eine wesentliche Voraussetzung, um die später von den beiden Programmen ermittelten Laufzeiten sinnvoll miteinander vergleichen zu können.

Nach Durchführung der Referenzmessung signalisiert das Slaveprogramm dem Masterprogramm die Bereitschaft, fortzufahren, woraufhin das Masterprogramm den Wartezustand verlässt. Nun übermitteln die beiden Programme zeitgleich einen Datenübertragungsbefehl an den GL-Server und messen die Laufzeit des übermittelten Befehls (beide Programme lassen dabei jeweils die gleiche Menge an Daten übertragen). Diese Messungen werden für alle Datenmengen wiederholt, für die zuvor auch eine Referenzmessung durchgeführt wurde.

Anschließend wird der gesamte Vorgang wiederholt, aber mit folgender Änderung des Ablaufs: Die Übermittlung des Datenübertragungsbefehls durch das Masterprogramm wird gegenüber der Übermittlung durch das Slaveprogramm verzögert, und zwar um die Hälfte der gemessenen Referenzlaufzeit. Es ist anzunehmen, dass dadurch die Ausführung des Datenübertragungsbefehls des Slaveprogramms bereits zur Hälfte abgeschlossen ist, sobald die Übermittlung des Datenübertragungsbefehls des Masterprogramms erfolgt.

Schließlich werden die von den beiden Programmen gemessenen Laufzeiten miteinander verglichen, um darauf schließen zu können, ob konkurrierende Datenübertragungsbefehle vom untersuchten OpenGL ES-System sequentiell ausgeführt werden oder nicht. Dabei wird von den beiden folgenden Annahmen ausgegangen:

- Wenn die beiden Datenübertragungsbefehle sequentiell abgearbeitet werden, dann wird bei deren gleichzeitiger Übermittlung entweder der Datenübertragungsbefehl des Master- oder der des Slaveprogramms etwa doppelt so lange laufen wie bei der Referenzmessung, während die Laufzeit beim jeweils anderen Programm der Laufzeit bei der Referenzmessung gleicht. Bei verzögertem Masterprogramm wird stets dessen Laufzeit eineinhalb mal so groß sein wie bei der Referenzmessung.

- Wird hingegen die Bandbreite gleichmäßig auf die beiden übermittelten Datenübertragungsbefehle aufgeteilt, so werden die Laufzeiten von Master- und Slaveprogramm etwa gleich groß sein. Wenn die beiden Programme ihre Befehle gleichzeitig übermitteln, wird sich die Laufzeit der Datenübertragungsbefehle gegenüber der Referenzmessung etwa verdoppeln. Wird die Übermittlung des Befehls beim Masterprogramm um die Hälfte der Referenzzeit verzögert, so wird die Laufzeit der Datenübertragungsbefehle etwa eineinhalb mal so groß sein wie bei der Referenzmessung.

Die in diesem Abschnitt beschriebene Untersuchung zur Laufzeit von konkurrierenden Datenübertragungsbefehlen erfolgt in Kapitel 4.3.2. Im nächsten Abschnitt wird beschrieben, wie untersucht wird, ob eine Datenübertragung durchgeführt werden kann, während zur gleichen Zeit ein Draw-Befehl ausgeführt wird.

3.2.2.3 Nebenläufige Ausführung von Datenübertragung und Rendering

Im Hinblick auf die Untersuchung zur nebenläufigen Ausführung von Datenübertragungs- und Draw-Befehlen stehen die folgenden beiden Fragen im Mittelpunkt:

- Kann eine Datenübertragung durchgeführt werden, während gleichzeitig ein gepufferter Draw-Befehl ausgeführt wird? Gepufferte Draw-Befehle sind hier von besonderem Interesse, da sich die von ihnen verarbeiteten Daten bereits in Datenobjekten befinden. Dadurch ist für die Ausführung von gepufferten Draw-Befehlen keine eigene Datenübertragung notwendig (sofern sich die Datenobjekte im GPU-Speicher befinden, wovon hier ausgegangen wird).
- Wie behindern sich Datenübertragungen und gepufferte Draw-Befehle gegenseitig, wenn sie nebenläufig ausgeführt werden können? Werden sie sequentiell abgearbeitet, oder werden Datenübertragungen oder Draw-Befehle unterbrochen, sobald ein Befehl der jeweils anderen Art an den GL-Server übermittelt wird?

Um diese Fragen zu beantworten, werden auch für diese Untersuchung zwei OpenGL ES-Programme ausgeführt, die ähnlich vorgehen, wie die beiden im vorhergehenden Abschnitt beschriebenen. Aber in dieser Untersuchung übermitteln nicht beide Programme Datenübertragungsbefehle, sondern nur eines der beiden. Das andere übermittelt einen Draw-Befehl an den GL-Server.

Der Ablauf bleibt ansonsten unverändert: Zunächst führt das Masterprogramm seine Referenzmessung durch, startet anschließend das Slaveprogramm und geht in einen Wartezustand über, bis das Slaveprogramm seine Referenzmessung durchgeführt hat. Anschließend messen die beiden Programme die Laufzeiten des von ihnen übertragenen Befehls, wobei die Übertragung der beiden Befehle in einem Fall gleichzeitig erfolgt, während im anderen Fall das Masterprogramm um die halbe vom Slaveprogramm gemessene Referenzlaufzeit verzögert wird.

Falls die Laufzeiten der konkurrierenden Befehle mit den Referenzwerten übereinstimmen, so ist das untersuchte System offensichtlich in der Lage, Datenübertragungen durchzuführen, während gleichzeitig ein gepufferter Draw-Befehl ausgeführt wird. Falls nicht, wird durch die Messungen mit verzögertem Masterprogramm sicher geklärt, ob es zu Unterbrechungen kommt.

Zur Ausführung des Draw-Befehls werden Minimalshader verwendet wie in [Munshi u. a. 2008] auf Seite 22 beschrieben (Listing und ausführliche Diskussion von Minimalshadern erfolgen in Kapitel 4.1.6). Durch die Verwendung von Minimalshadern soll sichergestellt werden, dass im Zuge der Ausführung des Draw-Befehls nicht unbeabsichtigt eine Funktion der Renderpipeline genutzt wird, die die Interoperabilität von Draw und Datenübertragung verhindert, während die nebenläufige Ausführung eines Draw- und eines Datenübertragungsbefehls prinzipiell möglich wäre.

Sofern die Untersuchung mit Minimalshadern zeigt, dass dies tatsächlich der Fall ist, kann gesondert untersucht werden, ob die nebenläufige Ausführung auch bei Verwendung komplexerer Shader möglich ist. Wenn sie hingegen bereits mit Minimalshadern nicht möglich ist, erübrigt sich jede weitergehende Untersuchung.

Die in diesem Abschnitt beschriebene Untersuchung zur nebenläufigen Ausführung von Datenübertragungs- und gepufferten Draw-Befehlen erfolgt in Kapitel 4.3.3. Im nächsten Abschnitt wird die Untersuchung der Datenübertragung von ungepufferten Draw-Befehlen beschrieben.

3.2.2.4 Datenübertragung ungepufferter Draw-Befehle

Ungepufferte Draw-Befehle stellen insofern einen Sonderfall dar, dass sich zumindest ein Teil der von ihnen verarbeiteten Daten zum Zeitpunkt ihrer Übermittlung an den GL-Server noch nicht im GPU-Speicher befindet. Diese Daten müssen also im Zuge der Ausführung des Draw-Befehls an die GPU übertragen werden.

Frühere Untersuchungen haben gezeigt, dass zumindest manche GPUs in diesem Fall die Datenübertragung parallel zur Ausführung der Draw-Befehle durchführen können (vgl. [Grottel u. a. 2009], Seite 71). Diese Fähigkeit muss also für ein konkretes System explizit überprüft werden; denn auch wenn die vorhergehende Untersuchung zeigt, dass eine nebenläufige Ausführung von Datenübertragungs- und Draw-Befehlen nicht möglich ist, könnte dies im Falle von ungepufferten Draw-Befehlen tatsächlich anders aussehen. Dazu ist folgende Frage zu beantworten:

- Entspricht die Laufzeit für einen ungepufferten Draw-Befehl der Summe der Laufzeiten für die Datenübertragung und die Ausführung des Draw-Befehls im gepufferten Fall? Falls dies so ist, muss davon ausgegangen werden, dass eine nebenläufige Ausführung von Datenübertragung und Rendering auch im Fall der Ausführung von ungepufferten Draw-Befehlen nicht möglich ist. Wenn die gemessenen Laufzeiten der ungepufferten Draw-Befehle die Summe der beiden Laufzeiten für Datenübertragung und Rendering im gepufferten Fall deutlich unterschreiten, dann muss vom Gegenteil ausgegangen werden.

Zur Klärung dieser Frage wird ein OpenGL ES-Programm ausgeführt, das wie folgt vorgeht: Noch vor der Ausführung des ungepufferten Draw-Befehls werden in einem ersten Schritt für dessen Vertexdaten die folgenden Referenzwerte ermittelt:

- Die Laufzeit des Befehls zur Übertragung der Vertexdaten in den GPU-Speicher.
- Die Laufzeit des gepufferten Draw-Befehls für diese Vertexdaten (die sich in dem Fall bereits vor Übermittlung des Draw-Befehls im GPU-Speicher befinden, wodurch keine Datenübertragung notwendig ist).
- Die Dauer für die Kopie der Vertexdaten innerhalb des Hauptspeichers.

Der letzte Punkt hat folgenden Hintergrund: Laut Spezifikation von OpenGL ES 2.0 darf eine Veränderung der vom Draw-Befehl verwendeten Daten, die nach dessen Rücksprung erfolgt, keine Auswirkungen auf das Ergebnis des Draw-Befehls haben (siehe [Munshi und Leech 2010], Seite 5). Der Rücksprung darf aber bereits erfolgen, bevor der Draw-Befehl auf der GPU vollständig abgearbeitet worden ist.

Wenn ein Rücksprung erfolgt, bevor die Daten in die GPU übertragen worden sein können (feststellbar durch Vergleich der bis zum Rücksprung verstrichenen Zeit mit der Referenzlaufzeit für die Datenübertragung), dann liegt die Annahme nahe, dass das OpenGL ES-System vor dem Rücksprung des Draw-Befehls eine Kopie der Daten im Hauptspeicher anlegt.[7]

Nach den Referenzwerten wird die Laufzeit des ungepufferten Draw-Befehls ermittelt und diese mit den Referenzwerten verglichen. Dabei wird von folgenden Annahmen ausgegangen:

[7] Theoretisch wäre auf den Testsystemen auch eine Kopie auf Festplatte möglich. Dies erscheint jedoch eher unwahrscheinlich, zumal eine Kopie im Hauptspeicher um Größenordnungen schneller durchgeführt werden kann.

- Wenn die Laufzeit des Draw-Befehls deutlich geringer ist als die Summe der Referenzwerte für Datenübertragung und gepufferten Draw-Befehl, dann ist das untersuchte System offenkundig in der Lage, Daten in die GPU zu übertragen, während der ungepufferte Draw-Befehl auf der GPU ausgeführt wird.
- Entspricht die Laufzeit des ungepufferten Draw-Befehls hingegen der Summe der Referenzwerte (ggf. zuzüglich der Zeit für die Datenkopie im Hauptspeicher), dann kann davon ausgegangen werden, dass auch im ungepufferten Draw-Fall die eigentliche Abarbeitung des Draw-Befehls in der GPU und die notwendigen Datenübertragungen nicht zur gleichen Zeit stattfinden.

In den vergangenen Abschnitten wurden alle untersuchten Fragestellungen hinsichtlich der Datenübertragung und die Ansätze zu deren Beantwortung detailliert erörtert. Im nächsten Abschnitt werden sämtliche OpenGL ES-Befehle aufgelistet, die für die hier beschriebene Untersuchung relevant sind, und es wird dargelegt, für welche dieser Befehle die Untersuchungen zur Datenübertragung in Kapitel 4 durchgeführt werden.

3.2.3 Relevante Befehle für die Datenübertragung

Für die Untersuchung der Datenübertragung sind die folgenden X Gruppen von Befehlen relevant: Befehle zur Erzeugung von Datenobjekten (siehe Abschnitt 3.2.3.1), Befehle zur teilweisen Aktualisierung von Datenobjekten (siehe Abschnit 3.2.3.2), Draw-Befehle (siehe Abschnitt 3.2.3.3, Befehle zur Programmverwaltung (siehe Abschnitt 3.2.3.4), Befehle zur Festlegung konstanten Shaderinputs (siehe Abschnitt 3.2.3.5) und alle sonstigen Datenübertragungsbefehle (siehe Abschnitt 3.2.3.6). Ein umfassender Überblick über alle von OpenGL ES 2.0 definierten Befehle mit einer kurzen Beschreibung findet sich im Anhang in Kapitel 6.1.

3.2.3.1 Befehle zur Erzeugung von Datenobjekten

Von den sieben von OpenGL ES 2.0 definierten Befehlen zur Erzeugung von Datenobjekten sind nur die folgenden fünf für die Datenübertragung relevant (da durch die beiden anderen Befehlen keine Daten übertragen werden können):

- `glBufferData` zur Erzeugung von Vertexbuffer-Objekten sowie
- `glTexImage2D`, `glCompressedTexImage2D`, `glCopyTexImage2D` und `glGenerateMipmap` zur Erzeugung von Texturobjekten.

Ob es durch `glGenerateMipmap` tatsächlich zu einer Datenübertragung kommt, ist abhängig von der konkreten Implementierung des OpenGL ES-Systems. Wenn die Ausgangstextur zur Erzeugung der Mipmaps im GPU-Speicher liegt und die Berechnung der Mipmaps innerhalb der GPU stattfindet ist keine Datenübertragung notwendig. Die anderen hier aufgeführten Befehle zur Erzeugung von Datenobjekten sind deshalb relevant, weil ihnen auch ein Zeiger auf Daten im Hauptspeicher übergeben werden kann, mit denen die betreffenden Datenobjekte gefüllt werden sollen. Falls hierbei kein Nullpointer übergeben wird, führen diese Befehle tatsächlich eine Datenübertragung durch. Außerdem dienen diese Befehle auch dazu, den kompletten Inhalt bereits bestehender Datenobjekte zu überschreiben. Auch in diesem Fall wird eine Datenübertragung durchgeführt.

Von den hier aufgeführten fünf Befehlen werden im Rahmen dieser Arbeit nur für den Befehl `glBufferData` die Untersuchungen zur Datenübertragung durchgeführt. Die Wahl fällt aus den gleichen Gründen speziell auf diesen Befehl, die auch für seine Wahl im Rahmen der Untersuchung der Speicherbelegung ausschlaggebend waren (siehe Abschnitt 3.1.3.1).

3.2.3.2 Befehle zur teilweisen Aktualisierung von Datenobjekten

OpenGL ES 2.0 definiert vier Befehle zur teilweisen Aktualisierung von Datenobjekten:

- `glBufferSubData` zur teilweisen Aktualisierung von Vertexbuffer-Objekten sowie
- `glTexSubImage2D`, `glCompressedTexSubImage2D` und `glCopyTexSubImage2D` zur teilweisen Aktualisierung von Textur-Objekten.

Diese vier Befehle entsprechen im Prinzip den jeweiligen Befehlen ohne *Sub* im Namen, allerdings mit dem Unterschied, dass durch sie keine neuen Datenobjekte erzeugt werden können, sondern nur Daten in bereits bestehende Datenobjekte übertragen werden können. Dafür kann optional nur ein Teil der Daten im jeweiligen Zielobjekt überschrieben werden (statt dem gesamten Inhalt). Da mit `glBufferData` bereits ein Datenübertragungsbefehl für die Untersuchung ausgewählt wurde, der zur Übertragung beliebiger Datenmengen verwendet werden kann, wurde auf eine gesonderte Untersuchung dieser vier Befehle verzichtet.

3.2.3.3 Draw-Befehle

Wie auch für die in Abschnitt 3.1.3.4 behandelte Untersuchung zur Speicherbelegung sind die beiden von OpenGL ES 2.0 definierten Draw-Befehle (`glDrawArrays` und `glDrawElements`) nur im ungepufferten Fall für die Untersuchung der Datenübertragung relevant, da hierbei die für das Rendering benötigten Vertex- bzw. Inputdaten noch nicht an das OpenGL ES-System übertragen wurden.

Aufgrund der Ergebnisse von [Grottel u. a. 2009] sind die ungepufferten Draw-Befehle im Hinblick auf die Untersuchung zur Datenübertragung sogar von besonderem Interesse. Die hierzu in den Abschnitten 3.2.2.3 und 3.2.2.4 beschriebenen Untersuchungen werden im Rahmen dieser Arbeit für `glDrawArrays` durchgeführt. Der einzige Unterschied zu `glDrawElements` besteht darin, dass für `glDrawElements` zusätzliche Indexdaten übergeben werden können, die dafür sorgen, dass nicht die gesamte Menge der übergebenen Vertexdaten für das Rendering verwendet werden. Im Gegensatz dazu werden bei `glDrawArrays` sämtliche übergebenen Vertexdaten verwendet (und somit im ungepufferten Fall auch übertragen). Dies dürfte jedoch keine Auswirkung auf die Verarbeitung der Vertexdaten durch die Renderpipeline haben.

3.2.3.4 Befehle zur Programmverwaltung

Von den Befehlen zur Programmverwaltung sind die selben beiden für die Untersuchung der Datenübertragung relevant, die auch für die Untersuchung der Speicherbelegung relevant sind:

- `glLinkProgram`, durch den aus einem Vertex- und einem Fragmentshader ein ausführbares Programmobjekt erzeugt wird und

- `glUseProgram`, durch den ein erzeugtes Programmobjekt im aktuellen Kontext installiert wird.

Falls Programmobjekte im GPU-Speicher abgelegt werden, müssen sie dorthin übertragen werden. Dies Übertragung kann im Zuge der Ausführung von `glLinkProgram` oder `glUseProgram` erfolgen. Die (mögliche) Übertragung von Programmobjekten durch diese beiden Befehle wurde im Rahmen dieser Arbeit jedoch nicht gesondert untersucht, und zwar aus den gleichen Gründen, aus denen diese beiden Befehle auch nicht bei der Untersuchung der Speicherbelegung berücksichtigt wurden (siehe Abschnitt 3.1.3.3).

3.2.3.5 Befehle zur Festlegung konstanten Shaderinputs

OpenGL ES 2.0 definiert insgesamt 27 Befehle, mit denen konstante Werte für den Shaderinput festgelegt werden können. Aus Sicht von Shaderprogrammen existieren aber nur zwei Typen von konstantem Input, nämlich Vertexattribut- und Uniform-Variablen. 19 der 27 Befehle dienen dazu, die Werte von Uniform-Variablen festzulegen, und die übrigen acht dienen dazu, die Werte von Vertexattribut-Variablen festzulegen.

Die einzelnen Befehle, die jeweils für eine der beiden Arten von Variablen „zuständig“ sind, unterscheiden sich nicht in ihrer grundsätzlichen Funktion sondern nur hinsichtlich der Parameter, mit denen die Werte der Variablen spezifiziert werden (so umfassen Vertexattribut-Variablen 16 Byte – diese 16 Byte können beispielsweise in einer Befehlsvariante durch Übergabe von vier Integern à vier Byte spezifiziert werden und in einer anderen Variante durch Übergabe eines Byte-Arrays mit 16 Elementen).

Da im Fall von Uniform-Variablen höchstens 128 Byte durch die Ausführung eines Befehls übertragen werden können und im Fall von Vertexattribut-Variablen sogar nur 16 Byte, wurden die Untersuchungen zur Datenübertragung im Rahmen dieser Arbeit nicht für die Befehle zur Festlegung des konstanten Shaderinputs durchgeführt (zumal bei den Untersuchungen mit dem Datenübertragungsbefehl `glBufferData` auf den in den hierbei verwendeten Testsystemen hunderte Megabyte durch einen Aufruf übertragen werden können, siehe Kapitel 4).

3.2.3.6 Sonstige Datenübertragungbefehle

Der Befehl `glReadPixels` dient dazu, Daten aus dem aktuellen Framebuffer in den Hauptspeicher zu kopieren. Daher ist dieser Befehl für die Untersuchung zur Datenübertragung relevant. Im Rahmen dieser Arbeit wurde auf eine gesonderte Untersuchung dieses Befehls jedoch verzichtet.

Im nun folgenden Abschnitt wird erläutert, warum die Untersuchung der Pipelinenutzung im Hinblick auf den Hintergrund der Arbeit sehr wichtig ist. Anschließend werden die genauen Fragestellungen für diese Untersuchung sowie die Ansätze zu deren Beantwortung erörtert und die dafür relevanten OpenGL ES-Befehle vorgestellt.

3.3 Pipelinenutzung

3.3.1 Motivation für die Untersuchung der Pipelinenutzung

Wie bereits in Kapitel 2 dargelegt, wird die Datenverarbeitung durch die Renderpipeline in OpenGL ES 2.0 durch Aufruf eines der beiden Draw-Befehle (`glDrawArrays` und `glDrawElements`) angestoßen. Eine zwingende Voraussetzung zur Erfüllung von Echtzeitgarantien für kritische Anwendungen ist die Möglichkeit, zumindest eine Obergrenze für die Laufzeit dieser Draw-Befehle angeben zu können. Dies ist bei OpenGL ES 2.0 für beliebige Shader und beliebigen Input jedoch nicht möglich, ohne die Draw-Befehle zuvor mit exakt den gleichen Inputdaten und sonstigen Einstellungen ausgeführt zu haben. Dafür gibt es zwei Gründe:

- Im Zuge der Ausführung von Draw-Befehlen werden auf der GPU Shader-Programme ausgeführt, und zwar jeweils mindestens eine Instanz eines Vertexshader- und eines Fragmentshader-Programms (vgl. Kapitel 2.6). Allgemein lässt sich deren Laufzeit nicht vorhersagen, ohne das Halteproblem für diese Programme zu lösen.

  Zwar erlaubt die Spezifikation der Shading Language von OpenGL ES 2.0 (d. h. der C-Dialekt, in dem Shader-Programme geschrieben werden) gewisse Einschränkungen hinsichtlich solcher Programme. Dadurch kann die Vorhersage ihrer Laufzeit erleichtert werden, zum Beispiel wenn Schleifen so eingeschränkt werden, dass die Anzahl der Schleifeniterationen zur Kompilierzeit ermittelt werden kann und Endlosschleifen ausgeschlossen sind. Ob solche Einschränkungen für ein konkretes System gelten, bleibt aber den Systemherstellern überlassen. Die Spezifikation der Shading Language erlaubt sogar explizit Endlosschleifen (siehe [Simpson und Kessenich 2009], Seite 57). Tatsächlich existieren mittlerweile GPUs, die keine Einschränkungen mehr hinsichtlich der Verwendung von Schleifen in Shaderprogrammen definieren, zum Beispiel die ARM-GPUs Mali-55, Mali-200 und Mali-400 MP (siehe [ARM 2009], Seite 37). Auf solchen GPUs kann ein Shaderprogramm unbegrenzt lange laufen.

- Selbst wenn die Laufzeit von Shader-Programmen genau vorhergesagt werden kann, gibt es ein weiteres Problem, das die Vorhersage der Laufzeit eines Draw-Befehls sehr erschwert: Für die Vorhersage reicht es nicht aus, nur die Laufzeit der Shader-Programme zu kennen; es muss auch bekannt sein, wieviele Instanzen der einzelnen Shader-Programme im Zuge der Ausführung eines Draw-Befehls ausgeführt werden. Während die Anzahl der ausgeführten Vertexshader-Instanzen anhand der Menge der zu verarbeitenden Vertices genau vorhergesagt werden kann, gilt dies für die Anzahl der ausgeführten Fragmentshader-Instanzen nicht.[8] Diese Anzahl ist abhängig von den konkreten Werten der einzelnen Vertices und steht erst fest, nachdem die Rasterisierung vollständig abgeschlossen wurde, also erst während der Ausführung eines Draw-Befehls.

Es stellt sich also die Frage, inwieweit ein konkretes OpenGL ES-System ermöglicht, dass konkurrierende Anwendungen die Renderpipeline gleichzeitig nutzen können oder ob ein einzelner, langlaufender Draw-Befehl einer Anwendung die Renderpipeline komplett für alle anderen Anwendun-

[8] Die einzige Ausnahme davon ist, wenn im Zuge eines Draw-Befehls weder Linien noch Dreiecke sondern nur Punkte gerendert werden. Dann stimmt die Anzahl der ausgeführten Vertexshader-Instanzen mit der der ausgeführten Fragmentshader-Instanzen überein. Der häufigste Anwendungsfall für Draw-Befehle in 3D-Anwendungen besteht allerdings darin, Dreiecke zu rendern (siehe [Munshi u. a. 2008], Seite 128).

gen blockieren kann. Falls eine nebenläufige Ausführung mehrerer Draw-Befehle möglich ist, sind Echtzeitgarantien für Draw-Befehle kritischer OpenGL ES-Anwendungen nicht zwingend ausgeschlossen, auch wenn von einer anderen Anwendung lang laufende Draw-Befehle an das OpenGL ES-System übermittelt werden:

Sofern eine Obergrenze für die Laufzeit von Draw-Befehlen kritischer Anwendungen bekannt ist[9] und bei der nebenläufigen Ausführung von Draw-Befehlen die Ressourcen der Renderpipeline zu gleichen Teilen auf die konkurrierenden Befehle aufgeteilt werden, kann von einer Verdopplung dieser Obergrenze ausgegangen werden; die mangelnde Vorhersagbarkeit der Laufzeit von Draw-Befehlen anderer Anwendungen stünde dann der Erfüllung von Echtzeitgarantien für kritische Anwendungen nicht mehr prinzipiell im Wege.

Alternativ sind solche Garantien trotz mangelnder Vorhersagbarkeit der Laufzeit von Draw-Befehlen möglich, wenn ein konkretes OpenGL ES-System die Unterbrechung oder zumindest den Abbruch der Ausführung eines Draw-Befehls ermöglicht. Sofern der Abbruchvorgang auf einem solchen System hinreichend schnell durchzuführen ist, kann ein Draw-Befehl, der eine kritische Anwendung behindert, im Notfall abgebrochen werden, um die rechtzeitige Ausführung der kritischen Befehle zu ermöglichen.

Im folgenden Abschnitt werden daher zwei Untersuchungen beschrieben, die die Fragen nach nebenläufiger Ausführbarkeit und Abbrechbarkeit von Draw-Befehlen für ein konkretes OpenGL ES-System klären.

3.3.2 Untersuchungsmethoden zur Pipelinenutzung

3.3.2.1 Ausführung konkurrierender Draw-Befehle

Um zu klären, ob konkurrierende Draw-Befehle nebenläufig ausgeführt werden können, werden zwei OpenGL ES-Programme ausgeführt, wobei die Ausführung des einen Programms durch das andere Programm bestimmt wird. Zur Vereinfachung der Diskussion darüber, wird das bestimmende Programm nachfolgend als *Masterprogramm* bezeichnet und das bestimmte als *Slaveprogramm*. Die beiden Programme gehen wie folgt vor:

In einem ersten Schritt führt das Masterprogramm eine Reihe von Draw-Befehlen aus, wobei dadurch jeweils eine unterschiedlich Anzahl an Dreiecken gerendert wird. Dabei wird die Laufzeit der Draw-Befehle gemessen. Die Ergebnisse dieser Messung fungieren als Referenzwerte für die nachfolgende Untersuchung. Für diese Untersuchung werden Minimalshader verwendet (wie in Kapitel 4.1.6 beschrieben), und zwar aus den gleichen Gründen, die auch in den bisher beschriebenen Untersuchungen für die Wahl von Minimalshadern ausschlaggebend waren (vgl. Abschnitt 3.2.2.2).

Nach Durchführung der Referenzmessungen startet das Masterprogramm das Slaveprogramm und geht in einen Wartezustand über. Das Slaveprogramm führt nun die gleichen Referenzmessungen durch wie das Masterprogramm. Wie in der in Abschnitt 3.2.2.2 beschriebenen Untersuchung dient dies dazu, sicherzustellen, dass die beiden Programme vom OpenGL ES-System gleich behandelt werden.

[9] Diese Obergrenze kann vor dem Einsatz der betreffenden Programme im Automobil ermittelt werden.

Nach Abschluss der Referenzmessung signalisiert das Slaveprogramm dem Masterprogramm die Bereitschaft, fortzufahren, woraufhin das Masterprogramm den Wartezustand verlässt und die eigentliche Untersuchung anstößt. Dabei übermitteln beide Programme die gleichen Draw-Befehle, die auch bei der Referenzmessung übermittelt wurden, wobei immer zuerst der Draw-Befehl des Masterprogramms an den GL-Server übermittelt wird und unmittelbar darauf der des Slaveprogramms. Es muss dabei sichergestellt sein, dass der Draw-Befehl des Slaveprogramms übermittelt wird, bevor der Befehl des Masterprogramms vollständig ausgeführt worden ist.

Falls die Laufzeiten der Draw-Befehle von Master- und Slaveprogramm übereinstimmen, muss davon ausgegangen werden, dass das bei der Untersuchung verwendete OpenGL ES-System in der Lage ist, konkurrierende Draw-Befehle nebenläufig auszuführen. Falls hingegen die vom Slaveprogramm gemessenen Laufzeiten deutlich länger sind als die vom Masterprogramm gemessenen, und letztere mit den Referenzlaufzeiten übereinstimmen, muss davon ausgegangen werden, dass die Ausführung der Draw-Befehle des Slaveprogramms verzögert wurde, bis die Ausführung des Draw-Befehls des Masterprogramms abgeschlossen war und eine nebenläufige Ausführung konkurrierender Draw-Befehle nicht möglich ist.

Die hier beschriebene Untersuchung zur Ausführung konkurrierender Draw-Befehle wird in Kapitel 4.4.1 durchgeführt. Im nächsten Abschnitt wird die Untersuchung zur Abbrechbarkeit von Draw-Befehlen beschrieben.

3.3.2.2 Abbrechbarkeit von Draw-Befehlen

OpenGL ES 2.0 definiert keinen Befehl, um laufende Draw-Anweisungen abzubrechen. Dennoch existieren GPUs, die dazu in der Lage sind, und zwar solche, die unter Windows-Betriebssystemen ab Windows Vista eingesetzt werden können. Die GPUs müssen das sogenannte *Windows Display Driver Model* (WDDM) unterstützen, damit sie unter diesen Betriebssystemen genutzt werden können. Die Abbrechbarkeit von Befehlen, die die GPU über einen längeren Zeitraum blockieren,[10] ist eine der zwingenden, von WDDM gesetzten Anforderungen an solche GPUs [Microsoft 2006].

Es stellt sich die Frage, welche Folgen ein solcher Abbruch für OpenGL ES-Programme hat. Unter den genannten Windows-Betriebssystemen sind diese Folgen von der jeweiligen GPU abhängig. Sobald die Ausführung eines Befehls das erlaubte Zeitintervall überschreitet, wird der GPU-Treiber durch das Betriebssystem aufgefordert, die Ausführung des laufenden Befehls abzubrechen und einen anderen Befehl auszuführen (dieser Vorgang wird als *preempt operation* bezeichnet). Falls dies gelingt, werden andere OpenGL ES-Anwendungen nicht negativ davon betroffen. Falls dies fehlschlägt, wird ein Hardware-Reset der GPU durchgeführt und der GPU-Treiber durch das Betriebssystem neu gestartet – davon sind dann alle OpenGL ES-Programme betroffen, da deren Kontexte dadurch ungültig werden [MSDN 2009].

Für die Echtzeitfähigkeit kritischer OpenGL ES-Anwendungen ist es entscheidend, ob durch den Abbruch eines laufenden Draw-Befehls die Kontexte aller OpenGL ES-Programme ungültig werden. Das dadurch erneut notwendige Setup des Kontextes und aller benötigten Datenobjekte könnte dazu führen, dass der nächste Draw-Befehl eines kritischen OpenGL ES-Programms nicht mehr rechtzeitig ausgeführt werden kann.

[10] Dieser Zeitraum kann vom Benutzer oder von Anwendungen verändert werden.

Zur Untersuchung des Abbruchverhaltens werden zwei OpenGL ES-Programme ausgeführt, wobei eines der beiden einen Draw-Befehl an den GL-Server übermittelt, der länger läuft als erlaubt, so dass ein Abbruch provoziert wird. Anschließend wird überprüft, ob die andere Anwendung noch in der Lage ist, über den bisherigen Kontext OpenGL ES-Befehle ausführen zu lassen.

Die hier beschriebene Untersuchung zur Ausführung konkurrierender Draw-Befehle wird in Kapitel 4.4.2 durchgeführt. Im nächsten Abschnitt werden die OpenGL ES-Befehle aufgeführt, die für die Untersuchung der Pipelinenutzung relevant sind.

3.3.3 Relevante Befehle für die Pipelinenutzung

Wie bereits in Abschnitt 3.3 dargelegt, sind die beiden von OpenGL ES 2.0 definierten Draw-Befehele (`glDrawArrays` und `glDrawElements`) für die Untersuchung der Pipelinenutzung relevant, da diese Befehle die Datenverarbeitung durch die Renderpipeline anstoßen. Für die Untersuchungen in Kapitel 4.4.1 und 4.4.2 wird `glDrawArrays` verwendet. Der einzige Unterschied zu `glDrawElements` besteht in der Art, wie die von der Renderpipeline zu verarbeitenden Vertexdaten spezifiziert werden (siehe Abschnitt 3.2.3.3), die Datenverarbeitung selbst bleibt davon unberührt.

Neben den Draw-Befehlen sind auch die in Abschnitt 3.2.3 aufgeführten Datenübertragungsbefehle relevant, da es Systeme geben kann, auf denen die Renderpipeline blockiert ist, solange eine Datenübertragung stattfindet (vgl. [Dwarakinath 2008, Seite 17). Die Untersuchung, inwieweit sich Draw- und Datenübertragungsbefehle gegenseitig behindern, wurde bereits in Abschnitt 3.2.2.3 beschrieben.

Im folgenden Abschnitt wird erläutert, warum die Berücksichtigung von Kontextwechseln im Hinblick auf den Hintergrund der Arbeit relevant ist. Danach werden die Untersuchungen beschrieben, mit denen Kosten für Kontextwechsel ermittelt werden können, und erörtert, welche OpenGL ES-Befehle für diese Untersuchungen relevant sind.

3.4 Kontextwechsel

3.4.1 Motivation für die Untersuchung von Kontextwechseln

Bevor der GL-Server einen OpenGL ES-Befehl ausführt, muss er sicherstellen, dass dieser Befehl gegen den korrekten Kontext ausgeführt wird, das heißt, dass für die Ausführung dieses Befehls sämtliche Einstellungen gelten, die von der übermittelnden Anwendung im aktuellen Kontext gesetzt wurden. Wenn zuvor ein Befehl einer anderen Anwendung ausgeführt worden ist, gelten zunächst noch die Einstellungen ihres Kontextes. Da diese nicht zwingend genau den Einstellungen der Anwendung entsprechen, die den neuen Befehl übermittelt, müssen in diesem Fall die Einstellungen ihres Kontextes übernommen werden, bevor der Befehl tatsächlich ausgeführt wird. Diese Übernahme von Einstellungen eines Kontextes wird als *Kontextwechsel* bezeichnet.

Ein solcher Kontextwechsel ist mit gewissen Kosten verbunden, das heißt aus Sicht einer OpenGL-Anwendung verlängert sich die Laufzeit eines an den GL-Server übermittelten Befehls, wenn zur

Ausführung dieses Befehls erst ein Kontextwechsel durchgeführt werden muss. [Dwarakinath 2008] beziffert die Kosten für einen Kontextwechsel auf einem System mit ATI r200 GPU mit 3 µs. Das Whitepaper zur Fermi-Architektur für Nvidia GPUs feiert die Reduktion der Kosten für einen Kontextwechsel gegenüber GPUs der vorhergehenden Generation um den Faktor 10 auf unter 25 µs (siehe [Nvidia 2009], Seite 18). Offenbar unterscheiden sich also die Kosten für Kontextwechsel zwischen GPUs verschiedener Hersteller sehr stark (Faktor 80 zwischen ATI r200 und Nvidia GPUs der Vor-Fermi-Generation).

Da zur Erfüllung von Echtzeitgarantien eine Obergrenze für die Laufzeit von OpenGL ES-Befehlen angegeben werden können muss, ist es sehr wichtig, die zusätzlichen Kosten durch einen Kontextwechsel zu kennen, denn dadurch erhöht sich diese Obergrenze. Im folgenden Abschnitt wird daher eine Untersuchung beschrieben, mit der sich die Kosten für Kontextwechsel ermitteln lassen.

3.4.2 Untersuchung der Kosten von Kontextwechseln

Prinzipiell könnte es nach Ausführung jedes Befehls zu einem Kontextwechsel kommen, wenn anschließend ein Befehl gegen einen anderen Kontext ausgeführt wird. Bei manchen Befehlen wäre es nicht verwunderlich, wenn das OpenGL ES-System auf einen Kontextwechsel verzichten würde, zum Beispiel bei Ausführung eines Befehls, der weder den GPU-Speicher noch die Renderpipeline berührt, was beispielsweise auf den Get-Befehl zur Abfrage des Namens des GPU-Herstellers zutrifft.[11]

Bei der Ausführung eines Draw-Befehls kann davon ausgegangen werden, dass ein Kontextwechsel durchgeführt wird. Daher werden zur Ermittlung der Kosten eines Kontextwechsels zwei OpenGL ES-Programme ausgeführt (im Folgenden wie in vorherigen Abschnitten als Master- und Slaveprogramm bezeichnet), die Draw-Befehle im gegenseitigen Wechsel ausführen (wobei das Slaveprogramm den ersten Draw-Befehl an den GL-Server übermittelt). Durch dieses Vorgehen wird auf Seiten des Masterprogramms für jeden Draw-Befehl ein Kontextwechsel erzwungen. Das Masterprogramm misst dabei die Laufzeit seiner Draw-Befehle. Die dabei auftretende Abweichung von der Referenzmessung entspricht den Kosten eines Kontextwechsels.

Dieses Vorgehen allein reicht aber nicht aus, um in jedem Fall die Kosten eines Kontextwechsels bei der Prognose der Laufzeit von OpenGL ES-Befehlen korrekt berücksichtigen zu können. Es kann – zumindest bei gewissen OpenGL ES-Implementierungen – zu Situationen kommen, die zu höheren Kosten für einen Kontextwechsel führen: Für den Linux-Treiber der ATI r200-GPU ist bekannt, dass die Userspace-Komponente[12] des Treibers eine vollständige Sicht auf das aktuelle Layout des GPU-Speichers[13] unterhält, die bei Bedarf aktualisiert wird (siehe [Dwarakinath 2008], Seite 7). Es ist nicht auszuschließen, dass auch auf anderen Systemen für jedes einzelne OpenGL ES-Programm eine solche Sicht unterhalten wird.

[11] Es kann natürlich konkrete OpenGL ES-Implementierungen geben, bei denen dies nicht der Fall ist; dies erscheint jedoch höchst unwahrscheinlich.

[12] Der Treiber für eine r200-GPU besteht aus zwei Komponenten, die sogenannte Userspace- und die Kernelspace-Komponente. Eine Instanz der Kernelspace-Komponente wird vom Kernel des Betriebssystems ausgeführt. Zusätzlich wird jeweils eine Instanz der Userspace-Komponente für jedes OpenGL ES-Programm erzeugt.

[13] Es handelt sich dabei um Informationen darüber, welche Abschnitte des GPU-Speichers gerade belegt oder frei sind.

Sofern dies bei einem konkreten System der Fall ist und die Aktualisierungen der Sichten nicht ohnehin bei jedem Kontextwechsel erfolgen, muss damit gerechnet werden, dass sich einzelne Kontextwechsel verteuern (dann, wenn eine solche Aktualisierung durchgeführt wird). Es ist nicht auszuschließen, dass nur solche OpenGL ES-Befehle davon betroffen sind, für deren Ausführung aktuelle Informationen zum Layout des GPU-Speichers benötigt werden – das sind alle Befehle, die die GPU-Speicherbelegung verändern können.[14].

Zur Überprüfung dieser Möglichkeit werden zwei OpenGL ES-Programme ausgeführt, die ganz ähnlich vorgehen, wie die beiden oben beschriebenen. In diesem Fall übermittelt das Slaveprogramm jedoch keinen Draw-Befehl, sondern einen Befehl, der die GPU-Speicherbelegung ändert. Das Masterprogramm wird in zwei Varianten ausgeführt: Die erste entspricht dem oben beschriebenen Masterprogramm. Dadurch wird festgestellt, ob durch die Änderung der Speicherbelegung die Kosten des „regulären“ Kontextwechsels erhöht werden. Anschließend wird die zweite Variante des Masterprogramms ausgeführt. Hierbei übermittelt das Masterprogramm statt eines Draw-Befehls ebenfalls einen Befehl, der die GPU-Speicherbelegung ändert, um festzustellen ob die Kosten für diesen Befehl erhöht sind.

Schließlich wird das Masterprogramm in der zweiten Variante zusammen mit dem ursprünglichen Slaveprogramm ausgeführt, um ausschließen zu können, dass Kontextwechsel für Befehle, die die GPU-Speicherbelegung ändern, generell erhöht sind – nach Ausführung aller in diesem Abschnitt beschriebenen Untersuchungen sind also die Kosten für die folgenden vier Fälle bekannt:

- Ein Draw-Befehl des Masterprogramms folgt einem Draw-Befehl des Slaveprogramms.
- Ein Draw-Befehl des Masterprogramms folgt einem Befehl des Slaveprogramms, der die GPU-Speicherbelegung ändert.
- Ein Befehl des Masterprogramms, der die GPU-Speicherbelegung ändert, folgt einem Draw-Befehl des Slaveprogramms.
- Ein Befehl des Masterprogramms, der die GPU-Speicherbelegung ändert, folgt einem Befehl des Slaveprogramms, der die GPU-Speicherbelegung ändert.

Dadurch wird geklärt, ob die Änderung der GPU-Speicherbelegung erhöhte Kosten verursacht und falls ja, ob diese erhöhten Kosten erst dann zum Tragen kommen, sobald ein weiterer Befehl ausgeführt wird, der die GPU-Speicherbelegung ändert.

3.4.3 Relevante Befehle für die Untersuchung von Kontextwechseln

Wie bereits zu Beginn von Abschnitt 3.4.2 dargelegt, kann es bei der Ausführung jedes OpenGL ES-Befehls zu einem Kontextwechsel kommen, wenn zuvor der Befehl eines anderen Programms ausgeführt wurde, zum Beispiel bei Draw-Befehlen. Für die hier beschriebene Untersuchung wird `glDrawArrays` verwendet. Es ist davon auszugehen, dass der andere von OpenGL ES 2.0 definierte Draw-Befehl (`glDrawElements`) gleiche Ergebnisse liefert (siehe Abschnitt 3.2.3.3).

Relevant für die Untersuchung der erhöhten Kosten von Kontextwechseln bei Änderung der Speicherbelegung sind grundsätzlich alle Befehle, die auch für die Untersuchung der Speicherbele-

[14] Diese Befehle sind in Abschnitt 3.1.3 aufgeführt.

gung relevant sind (siehe Abschnitt 3.1.3). Hier wird – wie in den vorhergehenden Untersuchungen auch – `glBufferData` verwendet, um ein Datenobjekt zu erzeugen.

Die in diesem Abschnitt beschriebene Untersuchung wird in Kapitel 4.5 durchgeführt. Im nächsten Kapitel sind die Durchführung aller Untersuchungen zu Speicherbelegung, Datenübertragung, Pipelinenutzung und Kontextwechseln sowie deren Ergebnisse dokumentiert.

4 Untersuchungen

Die im vorherigen Kapitel beschriebenen Untersuchungen wurden auf drei verschiedenen Testsystemen durchgeführt. Die dabei ausgeführten OpenGL ES-Programme und die dadurch gewonnenen Ergebnisse sind in den Abschnitten 4.2 bis 4.5 dokumentiert.

In Abschnitt 4.1 werden zunächst die technischen Details der Untersuchungen beschrieben, die nicht auf eine einzelne Untersuchung beschränkt sind. Dazu gehören die Beschreibungen der drei Testsysteme, der Verfahren für Laufzeit- und Speicherplatzmessungen, der Umgang mit Fehlern des GL-Systems, das Verfahren zur Interprozesskommunikation und die Beschreibung der für die Untersuchungen verwendeten Shaderprogramme.

4.1 Technische Details der Untersuchungen

4.1.1 Testsysteme

Komponente	**„Nvidia Quadro 2000D“**	**„ATI FirePro V4800“**	**„ATI FirePro V5900“**
Prozessor	Intel Core i7 920	Intel Pentium E2140	Intel Core 2 Q8400
Rechenkerne	4	2	4
Hauptspeicher	6 GB DDR3	8 GB DDR2	4 GB DDR2
GPU	Nvidia Quadro 2000D	ATI FirePro V4800	ATI FirePro V5900
GPU-Speicher	1 GB	1 GB	2 GB
Treiberversion	275.65	6.14.10.10225	8.01.01.1134
PCI-Slot	PCIe 2 x16 (max. 8 GB/s)	PCIe x8 (max. 2 GB/s)	PCIe 2 x8 (max. 4 GB/s)

Tabelle 4.1: Daten der verwendeten Testsysteme

Tabelle 4.1 zeigt die Daten der drei verwendeten Testsysteme im Überblick. Als Betriebssystem kommt auf allen drei Systemen Microsoft Windows 7 Professional x64 mit installiertem Service Pack 1 zum Einsatz, und zwar mit deaktivierter virtueller Speicherverwaltung und Verwendung von Minimalgrafik für die grafische Benutzeroberfläche (d. h. deaktivierte Aero-Oberfläche und keine visuellen Designs). Die Frequenz des Thread-Schedulers des Betriebssystems wird auf den höchstmöglichen Wert eingestellt. Dadurch erreichen Funktionen zum Thread-Timing, wie zum Beispiel der Sleep-Befehl, eine Genauigkeit von einer Millisekunde.

4.1.2 Durchführung von Laufzeitmessungen

Zur Durchführung von Laufzeitmessungen kommt die in Kapitel 2.3 beschriebene Laufzeitmetrik für OpenGL ES-Befehle zum Einsatz, d. h. unmittelbar vor Abfrage des ersten Zeitstempels wird

`glFinish` aufgerufen, und dem Aufruf des zu messenden Befehls folgt unmittelbar ein Aufruf von `glFlush` und `glFinish`, bevor der zweite Zeitstempel abgefragt wird. Die Laufzeit wird dann durch die Bildung der Differenz der beiden Zeitstempel berechnet.

Für die Abfrage von Zeitstempeln wird auf den drei Testsystemen eine *High-Performance-Clock* mit mikrosekunden-genauer Auflösung verwendet wie in [Walbourn 2005] beschrieben. Diese Uhr liefert für alle Prozesse des Systems identische Zeitstempel, wenn sie innerhalb der selben Mikrosekunde abgefragt werden.

Zwischen der Abfrage der Zeitstempel und dem Aufruf des zu messenden Befehls kann es zu Verzerrungen der Zeitmessung kommen, zum Beispiel wenn ein Prozesskontextwechsel durchgeführt wird. Außerdem können die Laufzeiten des zu messenden Befehls selbst schwanken. Die Messung einer einzelnen Ausführung des zu messenden Befehls reicht daher nicht aus. Deshalb wird ein Befehl im Zuge der Laufzeitmessung viele Male ausgeführt. Die Laufzeitmessung eines Befehls besteht also aus vielen Einzelmessungen.

Algorithmus 4.1 zeigt beispielhaft die Messung der Laufzeit des Befehls `glBufferData`:

Algorithmus 4.1 Messung der Laufzeit von `glBufferData`

```
1 HPCClock c;   // für Zugriff auf Mikrosekunden-Uhr
2 glFinish();   // alle noch nicht abgearbeiteten OpenGL ES-Befehle vollständig ausführen
3 for (unsigned int n = 0; n < iterations; n++)
4 {
5     c.start();              // ersten Zeitstempel speichern
6     glBufferData();         // zu messender Befehl
7     glFlush();              // Befehl sofort an GL-Server übermitteln
8     glFinish();             // Befehl vollständig ausführen
9     results[n] = c.stop(); // Differenz zwischen erstem und aktuellem Zeitstempel bilden
10 }
```

4.1.3 Durchführung von Speicherplatzmessungen

Die Spezifikation von OpenGL ES macht keine Vorschriften darüber, wie Daten in einem eventuell vorhandenen GPU-Speicher abgelegt werden und wie ein solcher Speicher aufgeteilt sein muss. Es findet sich nur eine einzige Erwähnung eines solchen Speichers und die einzige Angabe, die an der Stelle dazu gemacht wird, ist der Hinweis, dass Vertexbuffer-Objekte eine Möglichkeit darstellen, diesen Speicher zu nutzen (siehe [Munshi und Leech 2010], Seite 22).

Sowohl Nvidia als auch ATI haben für ihre Produkte jeweils eine Erweiterung von OpenGL veröffentlicht, über die abgefragt werden kann, wie viel freier GPU-Speicher für solche Datenobjekte zur Verfügung steht (vergleiche [Stroyan 2009] für die Erweiterung von Nvidia und [Blackmer u. a. 2009] für die Erweiterung von ATI). Diese Erweiterungen werden für die Untersuchungen dieser Arbeit verwendet. Die Details, wie und welche Informationen zum GPU-Speicher zur Verfügung gestellt werden, unterscheiden sich jedoch zwischen Nvidia und ATI.

4.1.3.1 Informationen der Nvidia-Erweiterung

Durch die Nvidia-Erweiterung werden OpenGL-Anwendungen folgende Informationen zum aktuellen Zustand des GPU-Speichers zur Verfügung gestellt:

- Die Menge des insgesamt vorhandenen GPU-Speichers (in kB) – dies entspricht der Gesamtmenge des auf der Hardware vorhandenen GPU-Speichers abzüglich der Allozierungen, die die Hardware im Zuge ihrer Initialisierung selbst durchführt.
- Die Gesamtmenge des momentan für OpenGL-Anwendungen verfügbaren GPU-Speichers (in kB).
- Die Anzahl an Datenobjekten, die seit der Initialisierung der GPU vom GPU-Speicher in den Hauptspeicher ausgelagert wurden (der sogenannte *eviction count*). Es ist dabei nicht klar, ob diese Datenobjekte vollständig oder nur teilweise ausgelagert wurden, oder ob sie immer noch ausgelagert sind.
- Die Gesamtmenge an Daten, die seit Initialisierung der GPU in den Hauptspeicher ausgelagert wurden (in kB).

Die Nvidia-Erweiterung vermittelt keinerlei Informationen über eine Aufteilung des GPU-Speichers (z. B. für verschiedene Arten von Datenobjekten). Insbesondere werden keine Informationen über die momentane Fragmentierung des GPU-Speichers zur Verfügung gestellt, die die Nutzung des gesamten GPU-Speichers verhindern kann.

4.1.3.2 Informationen der ATI-Erweiterung

Die ATI-Erweiterung vermittelt dem Benutzer ein anderes Modell des GPU-Speichers. Sie unterscheidet drei verschiedene Arten von GPU-Speicher: Speicher für Vertexbuffer-Objekte, für Texturen und für Renderbuffer-Objekte. Für jeden dieser drei Speicher können die folgenden Informationen zu ihrem aktuellen Zustand abgefragt werden:

- Die Gesamtmenge des freien, *regulären* Speichers, der für die entsprechende Art von Datenobjekten zur Verfügung steht (in kB).
- Die Größe des größten derzeit verfügbaren, zusammenhängenden Speicherabschnitts (in kB).
- Die Gesamtmenge des freien Hilfsspeichers (*auxiliary memory*) für die entsprechende Art von Datenobjekten (in kB).
- Die Größe des größten derzeit verfügbaren, zusammenhängenden Speicherabschnitts im Hilfsspeicher (in kB).

Bei manchen GPUs können diese drei Speicherarten auch zusammenfallen, d. h. das Anlegen eines Datenobjekts belastet nicht nur den Speicher für diese Datenobjektart, sondern gleichzeitig auch die Speicher der beiden anderen Datenobjektarten. Über die Natur des Hilfsspeichers werden in der Spezifikation der Erweiterung keine Angaben gemacht, d. h. es ist nicht klar, ob es sich dabei um Speicher handelt, der auf der Grafikkarte physisch vorhanden ist, oder ob es sich dabei um Hauptspeicher handelt.

Es wurde daher untersucht, wie dies für die im Rahmen dieser Arbeit verwendeten ATI-Grafikkarten implementiert wurde. Dabei konnte folgendes Verhalten beobachtet werden:

- Vor Beginn der Untersuchungen war im „ATI FirePro V4800“-System ein Gigabyte Hauptspeicher vorhanden. Zu diesem Zeitpunkt standen OpenGL-Anwendungen 500 MB Hilfsspeicher zur Verfügung. Nach der Erweiterung des Hauptspeichers auf acht Gigabyte standen zwei Gigabyte Hilfsspeicher zur Verfügung. Dies legt die Vermutung nahe, dass es sich bei dem von der ATI-Erweiterung angezeigten Hilfsspeicher tatsächlich um Hauptspeicher handelt, zumal sich an der Menge des auf der Grafikkarte verbauten GPU-Speichers nichts geändert hatte.
- Der freie Speicherplatz des regulären Speichers für Vertexbuffer-, Textur- und Renderbuffer-Objekte stimmt auf den beiden ATI-Systemen überein. Das bedeutet, wenn beispielsweise ein Vertexbuffer-Objekt erzeugt wird, sinkt nicht nur die Menge des verfügbaren Speichers für Vertexbuffer-Objekte, sondern gleichzeitig auch die Menge des verfügbaren Speichers für Textur- und Renderbuffer-Objekte, und zwar stets um den gleichen Betrag. Auf gleiche Weise stimmt auch der freie Speicherplatz in den drei Hilfsspeichern überein.

4.1.3.3 Definition leerer GPU-Speicher

Alle drei Testsysteme haben gemeinsam, dass OpenGL-Anwendungen nie über einen komplett leeren GPU-Speicher verfügen können. Dies hat zwei Gründe:

- Noch bevor eine OpenGL-Anwendung gestartet werden kann, wird durch das System selbst im Zuge seiner Initialisierung eine gewisse Menge an GPU-Speicher belegt.
- Im Zuge der Erzeugung eines GL-Kontextes werden eine Reihe sogenannter statischer Datenobjekte im GPU-Speicher angelegt. Es handelt sich dabei um die Datenobjekte, die den ersten Framebuffer des erzeugten Kontextes bilden. Die Erzeugung dieser Datenobjekte kann nicht vermieden werden und sie können auch nicht zerstört werden, solange ihr Kontext existiert.

Wenn nachfolgend von einem *leeren* GPU-Speicher gesprochen wird, dann ist ein Zustand gemeint, bei dem neben diesen unvermeidlichen Datenobjekten keine weiteren Datenobjekte im GPU-Speicher vorhanden sind.

4.1.4 Umgang mit Fehlern des GL-Systems

OpenGL ES 2.0 definiert den Befehl `glGetError`, mit dem der aktuelle Fehlercode abgefragt werden kann. Der aktuelle Fehlercode ist 0, wenn seit dem letzten Aufruf von `glGetError` im GL-System kein Fehler aufgetreten ist. Falls es sich um den ersten Aufruf von `glGetError` handelt, wird 0 zurückgeliefert, falls seit Erzeugung des Kontextes kein Fehler im GL-System aufgetreten ist. In den nachfolgenden Untersuchungen wird der aktuelle Fehlercode immer am Ende der dafür ausgeführten OpenGL ES-Programme abgefragt. Wenn er ungleich 0 ist, werden die von den Programmen ermittelten Ergebnisse nicht berücksichtigt (und stattdessen das Programm erneut ausgeführt).

4.1.5 Interprozesskommunikation

Für manche Untersuchungen ist die Ausführung von mehreren OpenGL ES-Programmen notwendig, die sich gegenseitig abstimmen müssen. Diese Programme interagieren mit Hilfe von sogenannten *Event-Objekten* des Betriebssystems [MSDN 2011]. Solche Objekte befinden sich entweder im Zustand *signalled* oder *not signalled*. Standardmäßig befinden sie sich im Zustand *not signalled*. Der Übergang in den Zustand *signalled* kann von Prozessen veranlasst werden.

Prozesse können außerdem in einen Wartezustand übergehen, der von einem bestimmten Event-Objekt abhängig ist. Sobald dieses Event-Objekt in den Zustand *signalled* übergeht, verlässt ein solcher Prozess den Wartezustand, woraufhin das Event-Objekt automatisch in den Zustand *not signalled* wechselt. Sofern sich das Objekt bereits im Zustand *signalled* befindet, wenn ein Prozess in einen davon abhängigen Wartezustand übergeht, verlässt der Prozess den Wartezustand sofort wieder und das Objekt wechselt in den Zustand *not signalled*.

Die Zeit, die vom Setzen des *signalled*-Zustands bis zur Fortsetzung eines darauf wartenden Prozesses vergeht, bewegt sich auf den Testsystemen im ein- bis zweistelligen Mikrosekundenbereich. Ausreißer sind aber nicht auszuschließen. Daher werden in den folgenden Untersuchungen die aktuellen Zeitstempel von allen beteiligten Programmen protokolliert, wenn Event-Objekte zu deren zeitlichen Synchronisation eingesetzt werden.

4.1.6 Minimalshader

Wie in Kapitel 2.6 beschrieben, werden im Zuge der Ausführung von Draw-Befehlen Vertex- und Fragmentshader ausgeführt. Damit ein Draw-Befehl ausgeführt werden kann muss vom aufrufenden OpenGL ES-Programm ein Vertex- und ein Fragmentshader bereitgestellt werden. Ein Rendering ohne Ausführung von Vertex- und Fragmentshadern ist in OpenGL ES 2.0 nicht möglich.

Algorithmus 4.2 Minimaler Vertexshader

```
1 attribute vec4 input;
2 void main()
3 {
4     gl_Position = input; // Input unverändert an Ausgabe weiterreichen
5 }
```

Zur Ausführung von Draw-Befehlen werden im Rahmen der folgenden Untersuchungen bei Bedarf Minimalshader verwendet wie in [Munshi u. a. 2008] auf Seite 22 beschrieben. Algorithmus 4.2 zeigt den Code eines minimalen Vertexshaders. Zeile 1 spezifiziert den Input des Vertexshaders. Es handelt sich dabei um eine Variable bestehend aus vier Fließkommawerten, die die Position des Vertex beschreiben, für den der Vertexshader ausgeführt wird.

Diese Variable wird im Zuge der Ausführung eines Draw-Befehls auf den Wert eines Elements der übergebenen Vertexdaten gesetzt (es wird jeweils eine Instanz des Vertexshaders für jedes Element der übergebenen Vertexdaten ausgeführt).[1] Dieser Wert wird in Zeile 4 unverändert an die built-in

[1] Siehe auch Kapitel 2.4.1 ab Seite 20 für die Rolle von Vertexdaten als Input für Vertexshader.

Variable `gl_Position` weitergereicht. Sie fungiert als minimale Ausgabe eines Vertexshaders (es können zusätzliche Ausgabevariablen definiert werden). Ihr muss zwingend ein Wert zugewiesen werden (siehe [Simpson und Kessenich 2009], Seite 59).

Algorithmus 4.3 Minimaler Fragmentshader

```
1 precision mediump float;
2 void main()
3 {
4     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // Ausgabe auf Farbwert für "rot" setzen
5 }
```

Algorithmus 4.3 zeigt den Code eines minimalen Fragmentshaders. Zeile 1 definiert die gewünschte Präzision für Fließkommaoperationen. Diese Angabe wird von der Spezifikation der Shading Language zwingend vorgeschrieben (siehe [Simpson und Kessenich 2009], Seite 36). Ähnlich wie auch bei Vertexshadern, fungiert hier eine built-in Variable als minimale Ausgabe und muss zwingend auf einen Wert gesetzt werden. Dies geschieht in Zeile 4 (`gl_FragColor` bestimmt den RGBA-Farbwert des vom Shader verarbeiteten Fragments, der hier auf „rot“ gesetzt wird – jeder andere Farbwert wäre ebenfalls akzeptabel).

4.2 Speicherbelegung

4.2.1 Ablage von Datenobjekten im GPU-Speicher

4.2.1.1 Durchführung der Untersuchung

Um festzustellen, unter welchen Bedingungen neu erzeugte Datenobjekte im GPU-Speicher oder im Hauptspeicher abgelegt werden, wird – wie in Kapitel 3.1.2.1 beschrieben – ein OpenGL ES-Programm ausgeführt, das Datenobjekte verschiedener Größen erzeugt und dabei die Informationen zur aktuellen Speicherbelegung abfragt. Dabei wird `glBufferData` zur Erzeugung der Datenobjekte verwendet (siehe Kapitel 3.1.3).

Algorithmus 4.4 zeigt das für diese Untersuchung ausgeführte OpenGL ES-Programm. Die Schleife von Zeile 9 – 30 iteriert über die übergebenen Datenobjektgrößen. Die eigentliche Messung für eine bestimmte Größe findet im Schleifenrumpf statt.

In Zeile 12 werden die aktuellen Informationen zum GPU-Speicher abgefragt. Anschließend wird in den Zeilen 15–18 ein neues Datenobjekt erzeugt, bevor in Zeile 22 erneut die aktuellen Informationen zum GPU-Speicher abgefragt werden. Der Aufruf von `glFinish` in Zeile 19 stellt sicher, dass das Datenobjekt erzeugt ist, bevor fortgefahren wird (vgl. Kapitel 2.3). In Zeile 28 wird das Datenobjekt wieder freigegeben. Auch hier wird anschließen `glFinish` aufgerufen, um sicherzustellen, dass das Datenobjekt bereits entfernt wurde, bevor die nächste Schleifeniteration ausgeführt wird.

Das in Algorithmus 4.4 gezeigte Programm wurde für unterschiedliche Datenobjektgrößen ausgeführt. In einem ersten Lauf wurden Datenobjektgrößen von einem Kilobyte bis zehn Megabyte untersucht, wobei in jedem Messschritt die Datenobjektgröße um ein Kilobyte erhöht wurde. In einem zweiten Lauf wurde, beginnend bei zehn Megabyte, die Datenobjektgröße in jedem Schritt

um ein Megabyte erhöht, bis die größtmögliche Datenobjektgröße erreicht war (512 MB beim „Nvidia Quadro 2000D“-System und 256 MB bei den beiden ATI-Systemen). Die beiden Läufe wurden für jeden möglichen Usage Hint durchgeführt (siehe Kapitel 2.4.1 zu Usage Hints). Im folgenden Abschnitt werden die Ergebnisse dieser Läufe zusammengefasst.

Algorithmus 4.4 Ablage von Datenobjekten in Haupt- oder GPU-Speicher

```
1  void testDataObjectPlacement(
2      unsigned int minSize,      // minimale Datenobjektgröße
3      unsigned int maxSize,      // maximale Datenobjektgröße
4      unsigned int stepSize,     // Zunahme der Datenobjektgröße zwischen Messungen
5      GLenum usageHint,          // Usage Hint
6      GLbyte *data)              // zu übertragende Daten
7  {
8      // Messreihe durchführen
9      for (unsigned int size = minSize; size <= maxSize; size += stepSize)
10     {
11         // Aktuelle Informationen zum GPU-Speicher abfragen
12         GPUMemoryInformation miBefore = getGPUMemInfo();
13
14         // Datenobjekt erstellen
15         GLuint id;
16         glGenBuffers(1, &id);
17         glBindBuffer(GL_ARRAY_BUFFER, id);
18         glBufferData(GL_ARRAY_BUFFER, size, data, usageHint);
19         glFinish();
20
21         // Aktuelle Informationen zum GPU-Speicher abfragen
22         GPUMemoryInformation miAfter = getGPUMemInfo();
23
24         // Ergebnisse für aktuelle Datenobjektgröße speichern
25         writeToResults(size, miBefore, miAfter, glGetError());
26
27         // Datenobjekt freigeben
28         glDeleteBuffers(1, &id);
29         glFinish();
30     }
31 }
```

4.2.1.2 Ergebnisse

Usage Hint	„Nvidia Quadro 2000D“	„ATI FirePro V4800“	„ATI FirePro V5900“
GL_STATIC_DRAW	GPU-Speicher	GPU-Speicher	GPU-Speicher
GL_DYNAMIC_DRAW	GPU-Speicher	Hauptspeicher	Hauptspeicher
GL_STREAM_DRAW	GPU-Speicher	Hauptspeicher	Hauptspeicher

Tabelle 4.2: Ablageorte von Datenobjekten in Abhängigkeit des Usage Hints

Die Größe des erzeugten Datenobjekts hat auf keinem der drei Testsysteme einen Einfluss darauf, ob das Datenobjekt im GPU-Speicher oder im Hauptspeicher abgelegt wird, im Gegensatz zum

übergebenen Usage Hint. Tabelle 4.2 zeigt dessen Einfluss auf den einzelnen Testsystemen im Überblick. Auf dem „Nvidia Quadro 2000D“-System werden Datenobjekte unabhängig vom gewählten Usage Hint im GPU-Speicher abgelegt. Auf den beiden ATI-Systemen werden sie nur bei der Wahl von `GL_STATIC_DRAW` im GPU-Speicher abgelegt. Ansonsten werden sie im Hilfsspeicher abgelegt (das heißt im Hauptspeicher, vgl. Abschnitt 4.1.3). Im nächsten Abschnitt erfolgt die Untersuchung, inwieweit der Speicherbedarf von Datenobjekten von ihrer Größe abweicht.

4.2.2 Speicherbedarf von Datenobjekten

4.2.2.1 Durchführung der Untersuchung

Algorithmus 4.5 Untersuchung des Speicherbedarfs von Datenobjekten.

```
1  void testMemoryConsumption(
2      unsigned int stepSize,     // Größenunterschied der Datenobjekte zwischen Messschritten
3      unsigned int maxSize,      // maximale Größe eines Datenobjekts
4      unsigned int numBuffers,   // Anzahl anzulegender Datenobjekte pro Messschritt
5      GLbyte *data,              // zu übertragende Daten
6      GLenum bufUsageHint)       // buffer usage hint
7  {
8      for (unsigned int size = stepSize; size <= maxSize; size += stepSize)
9      {
10         // freien Speicher vor Datenobjekterzeugung abfragen
11         GPUMemoryInformation miBefore = getGPUMemInfo();
12
13         // Datenobjekte anlegen
14         GLuint bufferIDs[numBuffers];
15         glGenBuffers(numBuffers, bufferIDs);
16         for (unsigned int n = 0; n < numBuffers; n++)
17         {
18             glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[n]);
19             glBufferData(GL_ARRAY_BUFFER, size, data, bufUsageHint);
20             glFinish();
21         }
22
23         // freien Speicher nach Datenobjekterzeugung abfragen
24         GPUMemoryInformation miAfter = getGPUMemInfo();
25
26         // Datenobjekte freigeben
27         glBindBuffer(bufTarget, 0);
28         glDeleteBuffers(numBuffers, bufferIDs);
29         glFinish();
30
31         // Daten das aktuellen Messschritts speichern
32         writeToResults(size, miBefore, miAfter, glGetError());
33     }
34 }
```

Um festzustellen, ob der von Datenobjekten belegte GPU-Speicher von der Größe der Datenobjekte abweicht, wird – wie in Kapitel 3.1.2.2 beschrieben – zunächst jeweils ein einzelnes Datenobjekt

einer bestimmten Größe im leeren GPU-Speicher angelegt und anschließend mehrere. Algorithmus 4.5 zeigt den Code des dazu ausgeführten OpenGL ES-Programms.

Die Schleife von Zeile 9–35 iteriert über die verschiedenen zu untersuchenden Datenobjektgrößen. Die Messung für eine bestimmte Datenobjektgröße findet dann im Schleifenrumpf statt. In Zeile 12 wird der verfügbare Speicherplatz abgefragt, bevor ein Datenobjekt erzeugt wird. Die Erzeugung der Datenobjekte erfolgt in den Zeilen 15–22.

Durch den Aufruf von `glFinish` wird sichergestellt, dass die Datenübertragung vollständig abgeschlossen ist und dementsprechend der gesamte von den Datenobjekten benötigte Speicherplatz belegt worden ist, bevor in Zeile 25 der nun verfügbare Speicherplatz abgefragt wird (der Datenübertragungsbefehl könnte bereits zurückspringen, bevor die Datenübertragung vollständig abgeschlossen ist, vgl. Kapitel 2.3).

In den Zeilen 27–30 werden die zuvor erzeugten Datenobjekte wieder freigegeben um für die nachfolgenden Messungen (mit der nächsten Datenobjektgröße) die ursprüngliche Speicherbelegungssituation wiederherzustellen, bevor schließlich in Zeile 34 die Ergebnisse für die aktuelle Datenobjektgröße gespeichert werden.

Das in Algorithmus 4.5 gezeigte Programm wurde in einem ersten Lauf für Datenobjektgrößen von einem Kilobyte bis zu zehn Megabyte ausgeführt und anschließend der dabei erkannte Trend stichprobenartig für die Größenbereiche zwischen 20–21 MB, 50–51 MB, 100–101 MB, 150–151 MB, 255–256 MB und 511–512 MB überprüft (der Abschnitt von 511–512 MB wurde dabei nur auf dem „Nvidia Quadro 2000D“-System durchgeführt, da die maximale Datenobjektgröße auf den beiden ATI-Systemen nur 256 MB beträgt).

Dabei wurden jeweils ein, zwei, drei und vier Datenobjekte in einem Messschritt erzeugt (wobei die Messungen für die Erzeugung mehrerer Datenobjekte bei den letzten beiden Größenbereichen ausgelassen wurden, wenn die Gesamtgröße der erzeugten Datenobjekte den verfügbaren GPU-Speicher überstieg – davon waren das „Nvidia Quadro 2000D“- und das „ATI FirePro V4800“-System betroffen, da auf diesen Systemen nur ein Gigabyte GPU-Speicher vorhanden ist). Zwischen den einzelnen Messschritten wurde die Größe der Datenobjekte jeweils um ein Kilobyte erhöht.

Zur Erzeugung der Datenobjekte wurde `glBufferData` verwendet, wobei die einzelnen Messungen für jeden möglichen Usage Hint durchgeführt wurden (siehe Kapitel 3.1.3 zur Wahl dieses Befehls). Die dabei ermittelten Ergebnisse werden im folgenden Abschnitt zusammengefasst.

4.2.2.2 Ergebnisse

Abbildung 4.1 zeigt die Ergebnisse für die Erzeugung eines einzelnen Datenobjekts pro Messschritt und Datenobjektgrößen bis drei Megabyte. Auf der X-Achse sind die Datenobjektgrößen aufgetragen und auf der Y-Achse die Speicherbelegung.

Auf den beiden ATI-Systemen entspricht die Menge des belegten GPU-Speichers exakt der Datenobjektgröße. Beim „Nvidia Quadro 2000D“-System sind zwei Bereiche zu unterscheiden. Für Datenobjekte kleiner als zwei Megabyte werden stets vier Megabyte GPU-Speicher belegt. Bei Datenobjekten ab zwei Megabyte entspricht die Menge des belegten GPU-Speichers exakt der Datenobjektgröße, falls die Datenobjektgröße ein ganzzahliges Vielfaches von 128 kB ist.

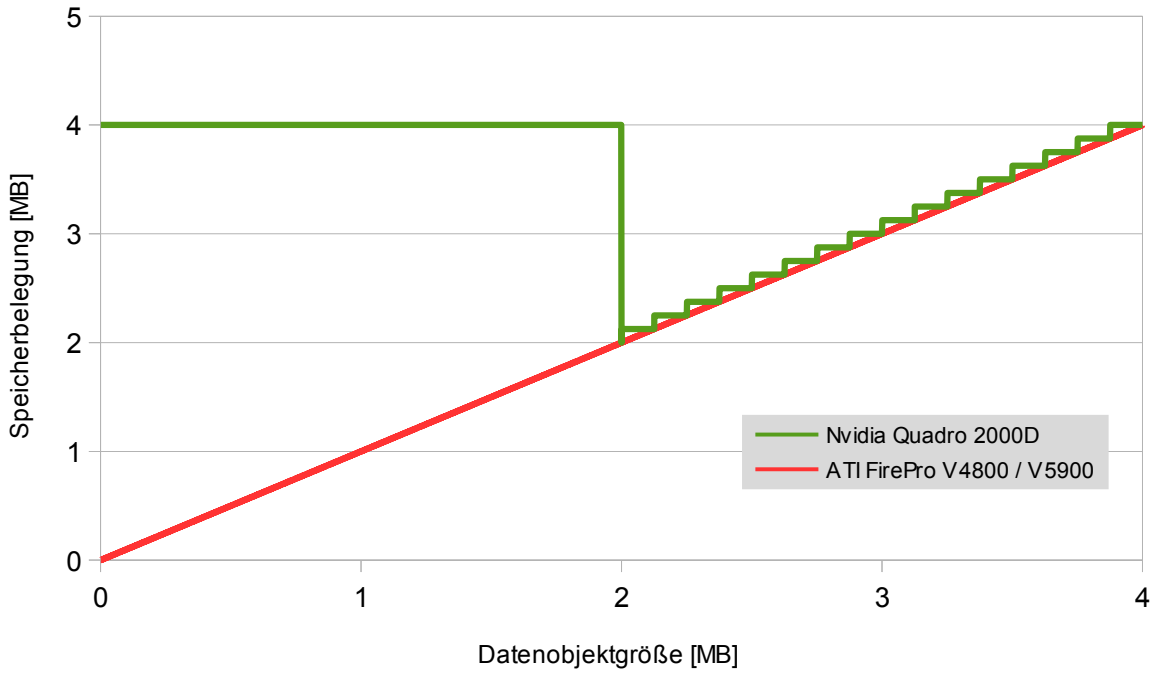

Abbildung 4.1: Speicherbelegung bei Erzeugung eines Vertexbuffer-Objekts

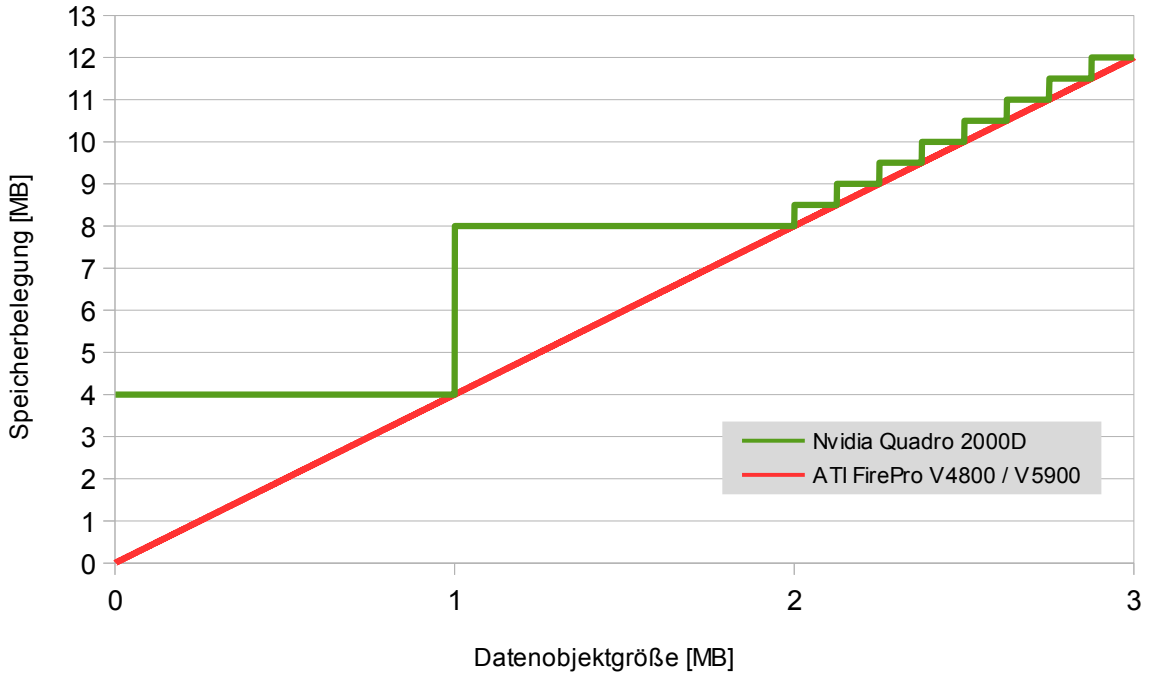

Abbildung 4.2: Speicherbelegung bei Erzeugung von vier Vertexbuffer-Objekten

Betrachtet man die in Abbildung 4.2 gezeigten Ergebnisse für die Erzeugung von vier Datenobjekten pro Messschritt, so deutet dies darauf hin, dass auf dem Nvidia-System Datenobjekte kleiner zwei Megabyte in Speicherblöcken von vier Megabyte Größe abgelegt werden (die Speicherbelegung überschreitet vier Megabyte erst, wenn vier Datenobjekte größer als ein Megabyte erzeugt werden).

Für Datenobjekte ab zwei Megabyte steigt die Speicherbelegung in Intervallen von einem halben Megabyte, was vier mal 128 kB entspricht. Offensichtlich steht der zusätzlich belegte Speicherplatz eines einzelnen Datenobjekts nicht für die Ablage von weiteren Datenobjekten zur Verfügung. Dies deutet darauf hin, dass auf dem Nvidia-System für Datenobjekte ab zwei Megabyte Größe eine Speichergranularität von 128 kB vorliegt (eine genauere Untersuchung der Speichergranularität erfolgt in Abschnitt 4.2.5, zumal die Untersuchung auf dem Nvidia-System aufgrund der Ablage von Datenobjekten in Speicherblöcken ein anderes Vorgehen erfordert).

Die Wahl verschiedener Usage Hints hat auf diese Ergebnisse keinen Einfluss (mit Ausnahme des Ablageortes der Datenobjekte auf den ATI-Systemen, vgl. Abschnitt 4.2.1.2). Die in den beiden Abbildungen nicht dargestellten Ergebnisse für die restlichen berücksichtigten Datenobjektgrößen bestätigen den Trend – die Menge des belegten GPU-Speichers entspricht auf den beiden ATI-Systemen exakt der jeweiligen Datenobjektgröße und beim „Nvidia Quadro 2000D“-System dann, wenn die Datenobjektgröße einem ganzzahligen Vielfachen von 128 kB entspricht. Im nächsten Abschnitt erfolgt die Untersuchung zur Bestimmung der Speicherblockgröße.

4.2.3 Bestimmung der Speicherblockgröße

4.2.3.1 Durchführung der Untersuchung

Zur Bestimmung der Speicherblockgröße werden – wie in Kapitel 3.1.2.3 beschrieben – sukzessive Datenobjekte gleicher Größe im GPU-Speicher erzeugt, wobei nach jeder Erzeugung eines Datenobjekts die aktuelle Speicherbelegung ermittelt wird. Dies wird anschließend für unterschiedliche Datenobjektgrößen wiederholt. Das dazu ausgeführte OpenGL ES-Programm wird in Algorithmus 4.6 gezeigt.

Zunächst wir in Zeile 8 die aktuelle Speicherbelegung ermittelt und gespeichert. Anschließend werden in der Schleife von Zeile 11–20 die eigentlichen Messungen durchgeführt. In Zeile 13 wird ein Datenobjekt erzeugt, und zwar auf die gleiche Weise wie in den Zeilen 15–19 von Algorithmus 4.4. In Zeile 16 wird anschließend die nun aktuelle Speicherbelegung gespeichert.

Dieses Programm wurde stichprobenartig für Datenobjektgrößen von 32 kB bis einem Megabyte ausgeführt, wobei zwischen den Läufen die Datenobjektgröße jeweils verdoppelt wurde. In jedem Lauf wurden so viele Datenobjekte angelegt, dass der GPU-Speicher komplett gefüllt wurde. Dieses Vorgehen wurde für alle Usage Hints wiederholt. Die dabei ermittelten Ergebnisse werden im nächsten Abschnitt zusammengefasst.

Algorithmus 4.6 Messung des Anstiegs der GPU-Speicherbelastung

```
1  void testBlocksize(
2      unsigned int numBuffers, // Anzahl der anzulegenden Datenobjekte
3      unsigned int bufSize,    // Größe der Datenobjekte
4      GLbyte *bufData,         // zu übertragende Daten
5      GLenum bufUsageHint)     // buffer usage hint
6  {
7      // Speicherbelegung abfragen und speichern
8      writeToResults(getGPUMemInfo());
9
10     // Messungen durchführen
11     for (unsigned int k = 1; k < numBuffers; k++)
12     {
13         (...) // Datenobjekt erzeugen
14
15         // Daten das aktuellen Messschritts speichern
16         writeToResults(
17             k,                  // Anzahl bislang angelegter Datenobjekte
18             k*bufSize,          // Menge der bislang übertragenen Daten
19             getGPUMemInfo(),    // Aktuelle Speicherbelegung
20             glGetError());      // Aktueller Fehlercode
21     }
22 }
```

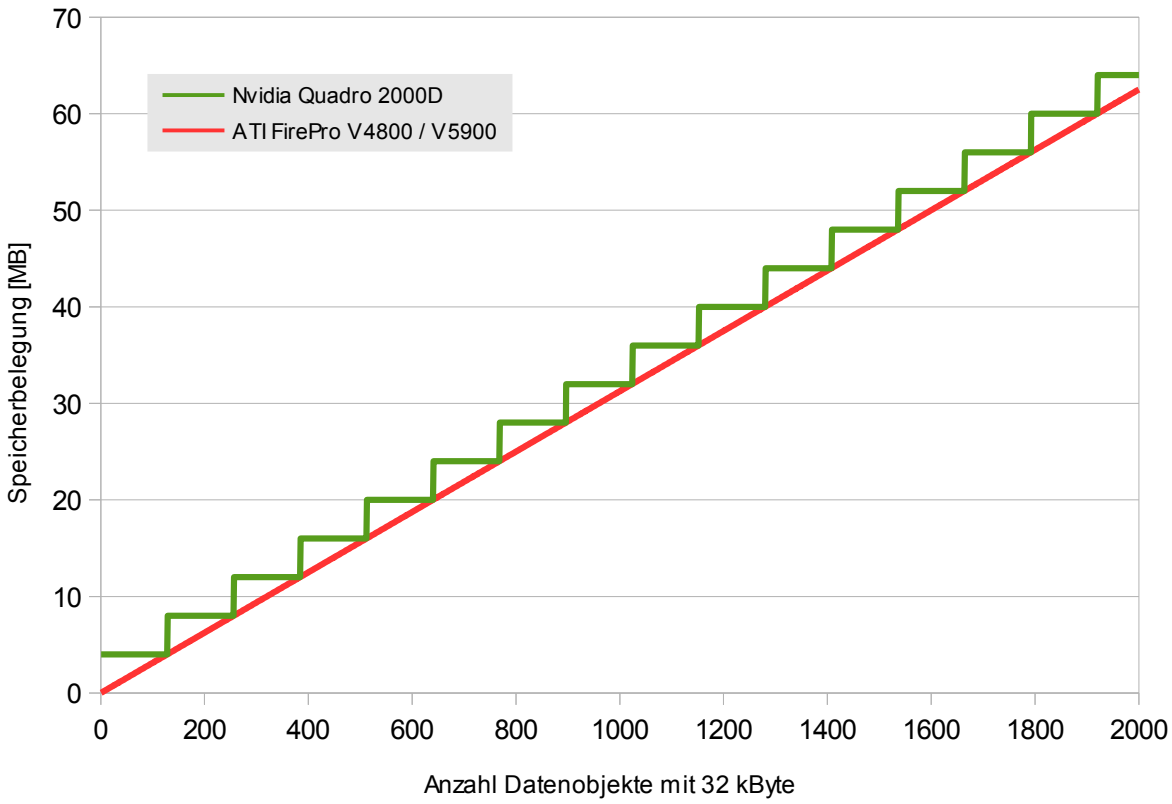

Abbildung 4.3: Anstieg der Speicherbelegung bei Erzeugung von Datenobjekten mit 32 kByte

4.2.3.2 Ergebnisse

Abbildung 4.3 zeigt die Ergebnisse für die Erzeugung von Datenobjekten mit 32 kB. Auf dem „Nvidia Quadro 2000D“-System steigt die Speicherbelegung in Sprüngen von jeweils vier Megabyte an, sobald seit dem letzten Sprung insgesamt mehr als ein ganzzahliges Vielfaches von vier Megabyte an Daten in den GPU-Speicher übertragen wurde. Die Abbildung 4.3 zeigt dabei nur einen Ausschnitt der Ergebnisse – das hier erkennbare Verhalten setzt sich aber so fort, bis der GPU-Speicher komplett gefüllt ist.

Die Verwendung unterschiedlicher Usage Hints ändert nichts an den Ergebnissen (mit Ausnahme des Ablageortes der Datenobjekte auf den ATI-Systemen, vgl. Abschnitt 4.2.1.2). Die Wiederholung des Programmlaufs mit anderen Datenobjektgrößen bringt qualitativ die gleichen Ergebnisse – auch hier steigt die Speicherbelegung auf dem „Nvidia Quadro 2000D“-System in Sprüngen von vier Megabyte an. Die Speicherblockgröße beträgt auf diesem System also stets vier Megabyte. Im nächsten Abschnitt erfolgen die Untersuchungen zum Belegungsverhalten innerhalb von Speicherblöcken.

4.2.4 Belegungsverhalten innerhalb von Speicherblöcken

Zur Ermittlung des Belegungsverhaltens innerhalb von Speicherblöcken werden – wie in Kapitel 3.1.2.4 beschrieben – mehrere Untersuchungen durchgeführt, um die folgenden Fragen zu klären:

- Werden Datenobjekte auf mehrere Speicherblöcke aufgeteilt?
- Werden Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt?
- Ist eine nichtsequentielle Ablage von Datenobjekten in Speicherblöcken möglich?
- Werden Datenobjekte in fragmentierten Speicherblöcken abgelegt?
- Werden Datenobjekte verschiedener Prozesse im selben Speicherblock abgelegt?

Die Durchführung dieser Untersuchungen wird in den nachfolgenden Abschnitten näher erläutert und deren jeweilige Ergebnisse zusammengefasst.

4.2.4.1 Aufteilung von Datenobjekten auf mehrere Speicherblöcke

Anders als in Kapitel 3.1.2.4.1 beschrieben, werden hier nicht Datenobjekte mit $\frac{3}{4}$ der Speicherblockgröße angelegt, sondern Datenobjekte mit $\frac{3}{8}$ der Speicherblockgröße. Dies liegt daran, dass auf dem „Nvidia Quadro 2000D“-System nur Datenobjekte von der Ablage in Speicherblöcken betroffen sind, die kleiner als die Hälfte der Speicherblockgröße sind. Abbildung 4.4 zeigt schematisch die Ablagemöglichkeiten für acht Datenobjekte mit $\frac{3}{8}$ der Speicherblockgröße.

Falls durch die Erzeugung von acht solchen Datenobjekten nur drei Speicherblöcke belegt werden, werden Datenobjekte über mehrere Speicherblöcke verteilt (Belegung 1 in Abbildung 4.4). Wenn stattdessen vier Speicherblöcke belegt werden, wird für die Erzeugung eines Datenobjekts ein neuer Speicherblock begonnen, wenn das Datenobjekt nicht mehr vollständig in einen bereits bestehenden passt (Belegung 2 in Abbildung 4.4). Um festzustellen, welcher Fall auf dem „Nvidia Quadro 2000D“-System vorliegt, wird das in Algorithmus 4.7 gezeigte OpenGL ES-Programm ausgeführt.

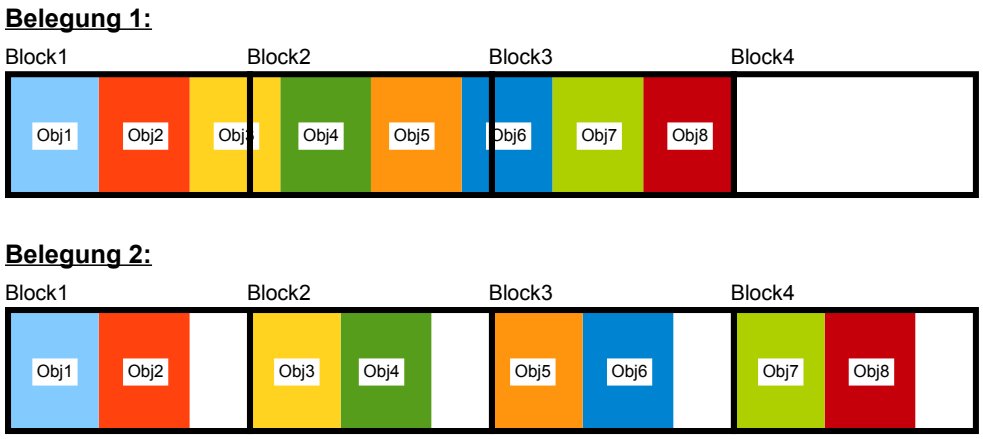

Abbildung 4.4: Ablagemöglichkeiten von acht Datenobjekten mit $\frac{3}{8}$ der Speicherblockgröße

Algorithmus 4.7 Erzeugung von Datenobjekten mit $\frac{3}{8}$ der Speicherblockgröße

```
1  void testThreeEighthBlocksizeObjects(
2      unsigned int blocksize,   // Speicherblockgröße
3      unsigned int iterations,  // Anzahl der Messschritte
4      GLbyte *bufData,          // zu übertragende Daten
5      GLenum bufUsageHint)      // buffer usage hint
6  {
7      // Speicherbelegung abfragen und speichern
8      writeToResults(getGPUMemInfo());
9
10     // Messungen durchführen
11     for (unsigned int n = 0; n < iterations; n++)
12     {
13         // Acht Datenobjekte mit 3/8 der Speicherblockgröße anlegen
14         for (unsigned int k = 0; k < 8; k++)
15         {
16             (...) // Datenobjekt erzeugen
17         }
18
19         // Ergebnis des Messschritts speichern
20         writeToResults(getGPUMemInfo(), glGetError());
21     }
22 }
```

Zunächst wird in Zeile 8 die aktuelle Speicherbelegung ermittelt und gespeichert. Anschließend werden in der Schleife von Zeile 11–21 die eigentlichen Messungen durchgeführt. Dazu werden in der Schleife von Zeile 14–17 jeweils acht Datenobjekte erzeugt. Die Erzeugung erfolgt auf gleiche Weise wie in den Zeilen 15–19 von Algorithmus 4.4, mit dem Unterschied, dass die Größe der Datenobjekte hier $\frac{3}{8}$ der Speicherblockgröße entspricht. In Zeile 20 wird anschließend die nun aktuelle Speicherbelegung gespeichert. Dieses Programm wurde auf dem „Nvidia Quadro 2000D“-System

mit so vielen Iterationen durchgeführt, dass der GPU-Speicher komplett mit Datenobjekten belegt wurde (auf den beiden ATI-Systemen kommt es nicht zur Ablage von Datenobjekten in Speicherblöcken, daher wurde dieses Programm dort nicht ausgeführt).

Dies wurde für alle Usage Hints wiederholt. Dabei wurden in jedem Messschritt vier Speicherblöcke belegt. Das bedeutet, dass auf diesem System die zweite der in Abbildung 4.4 gezeigten Ablagemöglichkeiten zutrifft – Datenobjekte werden nicht auf mehrere Speicherblöcke verteilt.

4.2.4.2 Ablage von Datenobjekten unterschiedlicher Größe im selben Speicherblock

Um zu überprüfen, ob Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt werden können, wird das in Algorithmus 4.8 gezeigte OpenGL ES-Programm ausgeführt.

Algorithmus 4.8 Erzeugung von Datenobjekten unterschiedlicher Größe

```
1  void testVarObjectSizes(
2      unsigned int blocksize,   // Speicherblockgröße
3      unsigned int iterations, // Anzahl durchzuführender Messschritte
4      GLenum usageHint,         // Usage Hint
5      GLbyte *data)             // zu übertragende Daten
6  {
7      // Messreihe durchführen
8      for (unsigned int n = 0; n < iterations; n++)
9      {
10         // aktuelle Informationen zum GPU-Speicher abfragen
11         GPUMemoryInformation miBefore = getGPUMemInfo();
12
13         // Speicherblock füllen
14         unsigned int remainingSpace = blocksize;
15         do
16         {
17             unsigned int size = getRandomSize(remainingSpace);
18             remainingSpace -= size;
19             (...) // Datenobjekt erzeugen
20         }while(remainingSize);
21
22         // aktuelle Informationen zum GPU-Speicher abfragen
23         GPUMemoryInformation miAfter = getGPUMemInfo();
24
25         // Ergebnisse des aktuellen Messschritts speichern
26         writeToResults(miBefore, miAfter, glGetError());
27     }
28 }
```

Zunächst wird in Zeile 11 die aktuelle Speicherbelegung ermittelt. Anschließend werden in der Schleife von Zeile 15–20 sukzessive Datenobjekte zufälliger Größe erzeugt, bis die Gesamtgröße der erzeugten Datenobjekte genau der Speicherblockgröße entspricht. `getRandomSize` liefert dabei eine Datenobjektgröße zurück, die zwischen einem Kilobyte und 1,999 MB liegt, aber den übergebenen Wert (`remainingSpace`) nicht überschreitet, so dass das danach erzeugte Datenobjekt

im Speicherblock noch Platz finden kann (sofern Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt werden). Die Erzeugung des Datenobjekts erfolgt dabei auf die gleiche Weise wie in den Zeilen 15–19 von Algorithmus 4.4. Anschließend wird in Zeile 23 die aktuelle Speicherbelegung ermittelt, bevor die Ergebnisse des Messschritts in Zeile 26 gespeichert werden.

Dieses Programm wurde für alle Usage Hints ausgeführt. Dies hatte keinen Einfluss auf die Ergebnisse – auf dem „Nvidia Quadro 2000D“-System werden Datenobjekte unterschiedlicher Größe im selben Speicherblock abgelegt.

4.2.4.3 Nichtsequentielle Ablage von Datenobjekten in Speicherblöcken

Da Datenobjekte unterschiedlicher Größe auf dem „Nvidia Quadro 2000D“-System im selben Speicherblock abgelegt werden können, kann untersucht werden, ob Datenobjekte auch dann im selben Speicherblock abgelegt werden, wenn sie nicht unmittelbar aufeinander folgend erzeugt werden. Dazu werden – wie in Kapitel 3.1.2.4.3 beschrieben – im leeren GPU-Speicher zunächst mehrere Datenobjekte erzeugt, so dass ein Speicherblock teilweise gefüllt ist. Anschließend wird durch Erzeugung weiterer Datenobjekte ein anderer Speicherblock komplett gefüllt und schließlich ein Datenobjekt erzeugt, das theoretisch im ersten Speicherblock Platz finden könnte, um zu überprüfen, ob es dort abgelegt wird. Anschließend wird dieses Vorgehen für größere Anzahlen teilweise gefüllter Speicherblöcke wiederholt.

Da auf dem „Nvidia Quadro 2000D“-System nur Datenobjekte, die kleiner sind als die halbe Speicherblockgröße, in Speicherblöcken abgelegt werden, muss diese Untersuchung mit anderen Datenobjektgrößen durchgeführt werden als im Beispiel in Kapitel 3.1.2.4.3 beschrieben. Um Speicherblöcke teilweise zu füllen, werden nicht zwei Datenobjekte mit $\frac{3}{8}$ sondern fünf Datenobjekte mit $\frac{3}{16}$ der Speicherblockgröße erzeugt (der Speicherblock ist dann zu $\frac{15}{16}$ gefüllt). Anschließend wird ein Speicherblock durch Erzeugung von vier Datenobjekten mit $\frac{1}{4}$ der Speicherblockgröße komplett gefüllt. Schließlich wird versucht, durch Erzeugung eines Datenobjekts mit $\frac{1}{16}$ der Speicherblockgröße den ersten Speicherblock aufzufüllen.

Das dazu ausgeführte OpenGL ES-Programm wird in Algorithmus 4.9 gezeigt. Die Schleife von Zeile 7 bis 44 iteriert über die Anzahl von teilweise gefüllten Speicherblöcken, die in einem Messschritt angelegt werden. In Zeile 11 werden dann für jede teilweise zu füllende Seite sechs IDs für Datenobjekte erzeugt, plus vier für den am Ende komplett zu füllenden Speicherblock.

In der Schleife von Zeile 14 bis 17 werden dann die teilweise gefüllten Speicherblöcke angelegt, indem für jeden Speicherblock fünf Datenobjekte angelegt werden. In der darauffolgenden Schleife werden die vier Datenobjekte angelegt, um einen Speicherblock komplett zu füllen. Bevor dann in der Schleife von Zeile 29 bis 33 versucht wird, jeden der teilweise gefüllten Speicherblöcke komplett zu füllen, wird in Zeile 26 die aktuelle Speicherbelegung abgefragt.

Nach dem Auffüllversuch wird ebenfalls die Speicherbelegung abgefragt (Zeile 35). Anschließend werden in den Zeilen 38–40 sämtliche erzeugten Datenobjekte wieder freigegeben, um für den nächsten Messschritt die ursprüngliche Speicherbelegung wiederherzustellen. Die Erzeugung einzelner Datenobjekte in den Zeilen 16, 22 und 31 erfolgt auf die gleiche Weise wie in den Zeilen 15–19 von Algorithmus 4.4.

Algorithmus 4.9 Nichtsequentielles Auffüllen von Speicherblöcken

```
1  void testNonSequentialStorage(
2      unsigned int blocksize, // Speicherblockgröße
3      unsigned int maxBlocks, // maximale Anzahl teilweise gefüllter Speicherblöcke
4      GLbyte *data,           // zu übertragende Daten
5      GLenum bufUsageHint)    // buffer usage hint
6  {
7      for (unsigned int n = 1; n <= maxNumPages; n++)
8      {
9          // IDs für die benötigten Datenobjekte erzeugen
10         GLuint bufferIDs[6*n+4];
11         glGenBuffers(6*n+4, bufferIDs);
12
13         // teilweise gefüllte Speicherblöcke anlegen
14         for (unsigned int k = 0; k < 5*n; k++)
15         {
16             (...) // Datenobjekt mit 3/16 der Speicherblockgröße erzeugen
17         }
18
19         // einen komplett gefüllten Speicherblock anlegen
20         for (unsigned int l = 0; l < 4; l++)
21         {
22             (...) // Datenobjekt mit 1/4 der Speicherblockgröße erzeugen
23         }
24
25         // Speicherbelegung vor Auffüllversuch abfragen
26         GPUMemoryInformation miBefore = getGPUMemInfo();
27
28         // Versuch, teilweise gefüllte Speicherblöcke aufzufüllen
29         for (unsigned int m = 0; m < n; m++)
30         {
31             (...) // Datenobjekt mit 1/16 der Speicherblockgröße erzeugen
32         }
33
34         // Speicherbelegung nach Auffüllversuch abfragen
35         GPUMemoryInformation miAfter = getGPUMemInfo();
36
37         // angelegte Datenobjekte freigeben
38         glBindBuffer(bufTarget, 0);
39         glDeleteBuffers(6*n+4, bufferIDs);
40         glFinish();
41
42         // Ergebnisse des Messschritts speichern
43         writeToResults(miBefore, miAfter, glGetError());
44     }
45 }
```

Dieses Programm wurde für alle Usage Hints ausgeführt, wobei im letzten Messschritt so viele Speicherblöcke angelegt wurden, dass der GPU-Speicher komplett gefüllt wurde. Dabei zeigte sich, dass sich die Speicherbelegung auf dem „Nvidia Quadro 2000D“-System zwischen der Messung vor und nach dem Auffüllversuch nicht ändert. Daraus lässt sich schließen, dass auf diesem System Datenobjekte in Speicherblöcken abgelegt werden, auch wenn sie nicht unmittelbar aufeinander folgend erzeugt werden. Usage Hints haben keinen Einfluss auf dieses Verhalten.

4.2.4.4 Ablage von Datenobjekten in fragmentierten Speicherblöcken

Algorithmus 4.10 Ablage von Datenobjekten in fragmentierten Speicherblöcken

```
1  void testFragmentedStorage(
2      unsigned int blocksize,          // Speicherblockgröße
3      unsigned int minObjectSize,      // minimale Größe der zu erzeugenden Datenobjekte
4      GLbyte *bufData,                 // zu übertragende Daten
5      GLenum bufUsageHint)             // buffer usage hint
6  {
7      for (unsigned int size = blocksize/8; size >= minObjectSize; size /= 2)
8      {
9          // IDs für die benötigten Datenobjekte erzeugen
10         unsigned int numObjects = blocksize / size;
11         GLuint bufferIDs[numObjects + numObjects/2];
12         glGenBuffers(numObjects + numObjects/2, bufferIDs);
13
14         // einen Speicherblock mit Datenobjekten füllen
15         for (unsigned int k = 0; k < numObjects; k++)
16         {
17             (...) // Datenobjekt erzeugen
18         }
19
20         // Belegungslücken erzeugen
21         glBindBuffer(bufTarget, 0);
22         for (unsigned int k = 0; k < numObjects; k+=2)
23         {
24             glDeleteBuffers(1, &bufferIDs[k]);
25             glFinish();
26         }
27
28         // Speicherbelegung vor Füllversuch der Lücken abfragen
29         GPUMemoryInformation miBefore = getGPUMemInfo();
30
31         // Versuch, Lücken zu füllen
32         for (unsigned int k = 0; k < numObjects/2; k++)
33         {
34             (...) // Datenobjekt erzeugen
35         }
36
37         // Speicherbelegung nach Füllversuch ermitteln
38         GPUMemoryInformation miAfter = getGPUMemInfo();
39
40         (...) // Erzeugte Datenobjekte freigeben
41
42         // Daten des Messschritts speichern
43         writeToResults(miBefore, miAfter, glGetError());
44     }
45 }
```

Um zu überprüfen, ob Datenobjekte in fragmentierten Speicherblöcken abgelegt werden, wird – wie in Kapitel 3.1.2.4.4 beschrieben – ein Speicherblock zunächst komplett mit Datenobjekten ge-

füllt und anschließend jedes zweite Datenobjekt wieder gelöscht, um den Speicherblock zu fragmentieren.[2] Dazu wird das in Algorithmus 4.10 gezeigte OpenGL ES-Programm ausgeführt.

Die Schleife von Zeile 7 bis 44 iteriert über die zu untersuchende Datenobjektgröße (die in jedem Messschritt halbiert wird, bis die minimale Datenobjektgröße erreicht ist). In den Zeilen 10–12 werden die IDs für die zu erzeugenden Datenobjekte erzeugt. Es werden eineinhalb mal so viele IDs benötigt wie Datenobjekte in einen Speicherblock passen würden (da die Hälfte der Datenobjekte des Speicherblocks gelöscht und dann ein zweites Mal erzeugt werden).

Durch die Schleife von Zeile 15 bis 18 wird ein Speicherblock komplett mit Datenobjekten gefüllt, bevor in den Zeilen 21–26 jedes zweite dieser Datenobjekte gelöscht wird, um den Speicherblock zu fragmentieren. Anschließend wird in den Zeilen 32–35 versucht, diese Lücken aufzufüllen, wobei unmittelbar davor und danach die aktuelle Speicherbelegung ermittelt wird. Schließlich werden alle erzeugten Datenobjekte freigegeben, um für den nächsten Messschritt die ursprüngliche Speicherbelegungssituation wiederherzustellen.

Dieses Programm wurde für alle Usage Hints ausgeführt, bis zu einer minimalen Datenobjektgröße von einem Kilobyte (was 2000 Belegungslücken im Speicherblock entspricht). Dabei zeigte sich, dass sich auf dem „Nvidia Quadro 2000D“-System für alle untersuchten Datenobjektgrößen und alle Usage Hints die Speicherbelegung vor und nach dem Füllversuch nicht ändert. Daraus kann geschlossen werden, dass Datenobjekte auf diesem System in fragmentierten Speicherblöcken abgelegt werden.

Im nächsten Abschnitt erfolgt die Untersuchung zur Bestimmung der Speichergranularität. Da dies nicht nur für das „Nvidia Quadro 2000D“-System relevant ist, wird diese Untersuchung für alle drei Testsysteme durchgeführt.

4.2.5 Bestimmung der Speichergranularität

4.2.5.1 Durchführung der Untersuchung

Hinsichtlich der Bestimmung der Speichergranularität muss als zusätzliche Schwierigkeit beim „Nvidia Quadro 2000D“-System beachtet werden, dass Datenobjekte, die kleiner als zwei Megabyte sind, von der Ablage in Speicherblöcken betroffen sind, wodurch nicht ohne weiteres gemessen werden kann, wieviel GPU-Speicher durch deren Erzeugung belegt wird. Wie in Kapitel 3.1.2.5 beschrieben, wird dieses Problem umgangen, indem eine sehr große Zahl an Datenobjekten erzeugt wird, wodurch auf diesem System durch die Anzahl der belegten Speicherblöcke darauf geschlossen werden kann, welche Speichergranularität vorliegt. Dazu wird das in Algorithmus 4.11 gezeigte OpenGL ES-Programm ausgeführt.

[2] Unter der Fragmentierung eines Speicherblocks wird hier verstanden, dass es innerhalb des Speicherblocks ungenutzten Speicher zwischen benutzten Speicherbereichen gibt. Dies kann auf dem „Nvidia Quadro 2000D“-System nicht überprüft werden. Es wird von der Annahme ausgegangen, dass solche Belegungslücken entstehen, wenn jedes zweite Datenobjekt des Speicherblocks gelöscht wird.

Algorithmus 4.11 Bestimmung der Speichergranularität.

```
1  void testForMemGranularity(
2      unsigned int baseSize,    // Mindestgröße eines Datenobjekts
3      unsigned int testRange,   // maximale zusätzliche Datenobjektgröße
4      unsigned int stepSize,    // Größenunterschied zwischen zwei Messschritten
5      unsigned int numBuffers, // Anzahl anzulegender Datenobjekte pro Messschritt
6      GLbyte *bufData,          // zu übertragende Daten
7      GLenum bufUsageHint)      // buffer usage hint
8  {
9      for (unsigned int size = baseSize; size <= baseSize + testRange; size += stepSize)
10     {
11         (...) // Datenobjekt-IDs erzeugen
12
13         // aktuelle Speicherbelegung ermitteln
14         GPUMemoryInformation miBefore = getGPUMemInfo();
15
16         // Datenobjekte erzeugen
17         for (unsigned int n = 0; n < numBuffers; n++)
18         {
19             (...) // Datenobjekt erzeugen
20         }
21
22         // aktuelle Speicherbelegung ermitteln
23         GPUMemoryInformation miAfter = getGPUMemInfo();
24
25         // Ergebnisse des Messschritts speichern
26         writeToResults(miBefore, miAfter, glGetError());
27
28         (...) // Datenobjekte freigeben
29     }
30 }
```

Die Schleife von Zeile 9 bis 29 iteriert über die zu untersuchenden Datenobjektgrößen. In Zeile 11 werden die IDs der zu erzeugenden Datenobjekte erzeugt (analog zu den Zeilen 10–11 von Algorithmus 4.9). Die Schleife von Zeile 17 bis 20 erzeugt dann die Datenobjekte für den aktuellen Messschritt. Die Speicherbelegung wird dabei unmittelbar vor und nach der Erzeugung der Datenobjekte ermittelt. Nach der Speicherung der Ergebnisse des aktuellen Messschritts werden in Zeile 28 die erzeugten Datenobjekte wieder freigegeben, um für den nächsten Messschritt die ursprüngliche Speicherbelegungssituation wiederherzustellen.

Dieses Programm wurde auf allen drei Testsystemen für Datenobjektgrößen von einem Byte bis vier Kilobyte durchgeführt, wobei in jedem Messschritt die Datenobjektgröße um ein Byte erhöht wurde. Anschließend wurde auf den beiden ATI-Systemen der Trend stichprobenartig für Datenobjektgrößen von 8 MB bis 8 MB + 4 kB, 50 MB bis 50 MB + 4 kB, 150 MB bis 150 MB + 4 kB und 250 MB + 4 kB wiederholt, wobei auch hier die Datenobjektgröße zwischen zwei Messschritten um ein Byte erhöht wurde.

Da die in Abschnitt 4.2.2 ermittelten Ergebnisse darauf hinweisen, dass auf dem „Nvidia Quadro 2000D“-System für Datenobjekte ab zwei Megabyte eine Speichergranularität von 128 kB vorliegt, wurde hier der Messbereich auf vier Megabyte ausgedehnt und die Datenobjektgröße zwischen zwei Messschritten um ein Kilobyte erhöht.

Für dieses System wurde der Trend stichprobenartig für Datenobjektgrößen von 8–12 MB, 50–54 MB, 150–154 MB, 250–254 MB und 500–504 MB überprüft. Dies wurde für alle Usage Hints wiederholt. Im folgenden Abschnitt werden die dabei ermittelten Ergebnisse zusammengefasst.

4.2.5.2 Ergebnisse

Abbildung 4.5 zeigt die Ergebnisse für die Erzeugung von 65.536 Datenobjekten pro Messschritt auf dem „Nvidia Quadro 2000D“-System. Auf der X-Achse sind die Datenobjektgrößen aufgetragen und auf der Y-Achse die Speicherbelegung bzw. die übertragene Datenmenge (dies gilt auch für die Achsen der darauf folgenden drei Abbildungen). Diese Abbildung zeigt einen Ausschnitt der Ergebnisse der Untersuchung für Datenobjektgrößen von einem Byte bis acht Kilobyte.

Die Menge des durch die Erzeugung der Datenobjekte belegten GPU-Speichers stimmt nur dann mit der Gesamtmenge der übertragenen Daten überein, wenn die Datenobjektgrößen einem Vielfachen von 512 Byte entsprechen. Bei Datenobjektgrößen von 1025 bis 1536 Byte ist eine Anomalie zu erkennen. In diesem Bereich wäre eine Speicherbelegung von 96 MB zu erwarten (65536 * 1536 Byte = 96 MB). Tatsächlich werden aber 100 MB belegt, was einem zusätzlichen Speicherblock entspricht.

Dies geschieht, weil Datenobjekte auf dem „Nvidia Quadro 2000D“-System nicht in teilweise belegten Speicherblöcken abgelegt werden, wenn sie nicht vollständig hineinpassen (vgl. Abschnitt 4.2.4.1). Da die Speicherblockgröße auf diesem System kein ganzzahliges Vielfaches von 1536 Byte ist, bleibt ein Teil jedes Speicherblocks unbelegt, und zwar jeweils 1024 Byte (4 MB mod 1536 Byte = 1024). Nachdem 96 MB gefüllt sind, werden noch 16 weitere Datenobjekte erzeugt. Dadurch kommt es zur Reservierung eines weiteren Speicherblocks.

Abbildung 4.6 zeigt die Ergebnisse für die Erzeugung von 65.536 Datenobjekten pro Messschritt auf den beiden ATI-Systemen. Diese Abbildung zeigt einen Ausschnitt der Ergebnisse der Untersuchung für Datenobjektgrößen von einem Byte bis acht Kilobyte. Auf den ATI-Systemen stimmt die Speicherbelegung mit der Gesamtmenge der übertragenen Daten überein, wenn die Datenobjektgrößen einem ganzzahligen Vielfachen von 256 Byte entsprechen; Datenobjektgrößen, die kein Vielfaches von 256 Byte sind, führen zu einer Speicherbelegung, die der von Datenobjekten des jeweils nächsthöheren Vielfachen von 256 Byte entspricht.

Abbildung 4.7 zeigt die Ergebnisse für die Erzeugung von 32 Datenobjekten mit einer Größe von acht bis neun Megabyte auf dem „Nvidia Quadro 2000D“-System. In diesem Größenbereich stimmt die Menge des belegten GPU-Speichers mit der Gesamtmenge der übertragenen Daten überein, wenn die Datenobjektgrößen einem ganzzahligen Vielfachen von 128 kB entsprechen. Datenobjektgrößen, die kein solches Vielfaches sind, führen zu einer Speicherbelegung, die der von Datenobjekten des jeweils nächstgrößeren Vielfachen von 128 kB entsprechen. Dieser Trend bestätigt sich auch für die übrigen untersuchten Größenbereiche. Auf dem „Nvidia Quadro 2000D“-System gibt es also offenbar zwei verschiedene Speichergranularitäten: 512 Byte für Datenobjekte kleiner zwei MB und 128 kB für alle anderen.

Abbildung 4.8 zeigt die Ergebnisse für die Erzeugung von 32 Datenobjekten mit einer Größe von 8 MB bis 8 MB + 2 kB auf den beiden ATI-Systemen. Auch in diesem Größenbereich stimmt die Menge des belegten GPU-Speichers mit der Gesamtmenge der übertragenen Daten überein, wenn die Datenobjektgrößen einem ganzzahligen Vielfachen von 256 Byte entsprechen.

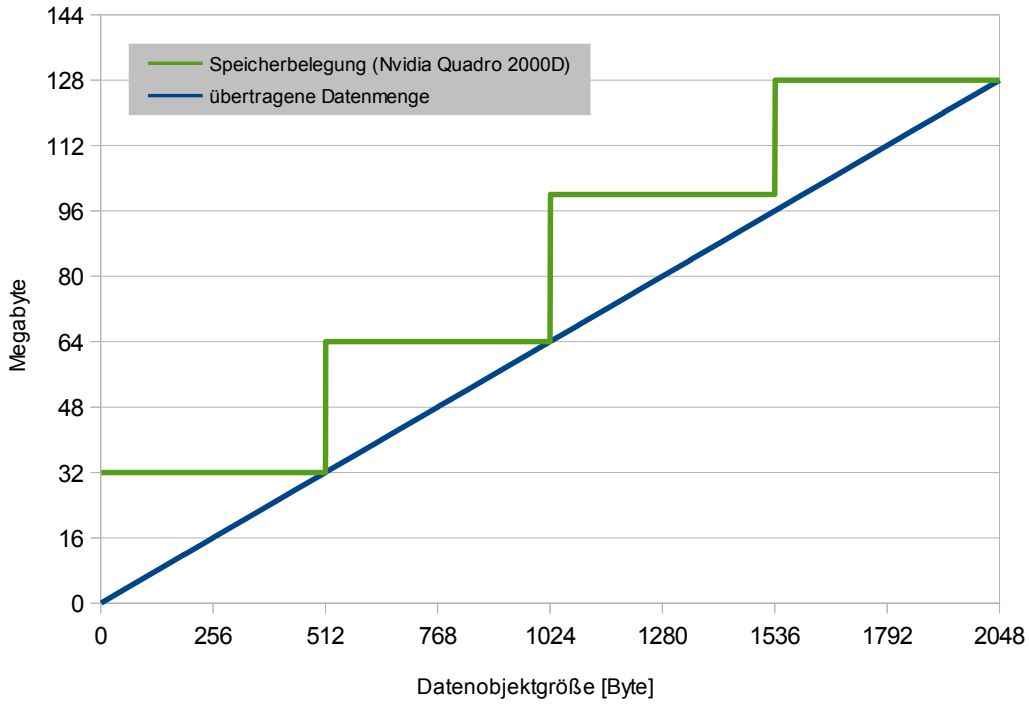

Abbildung 4.5: Speicherbelegung durch Erzeugung von 65.536 Datenobjekten (Nvidia-System)

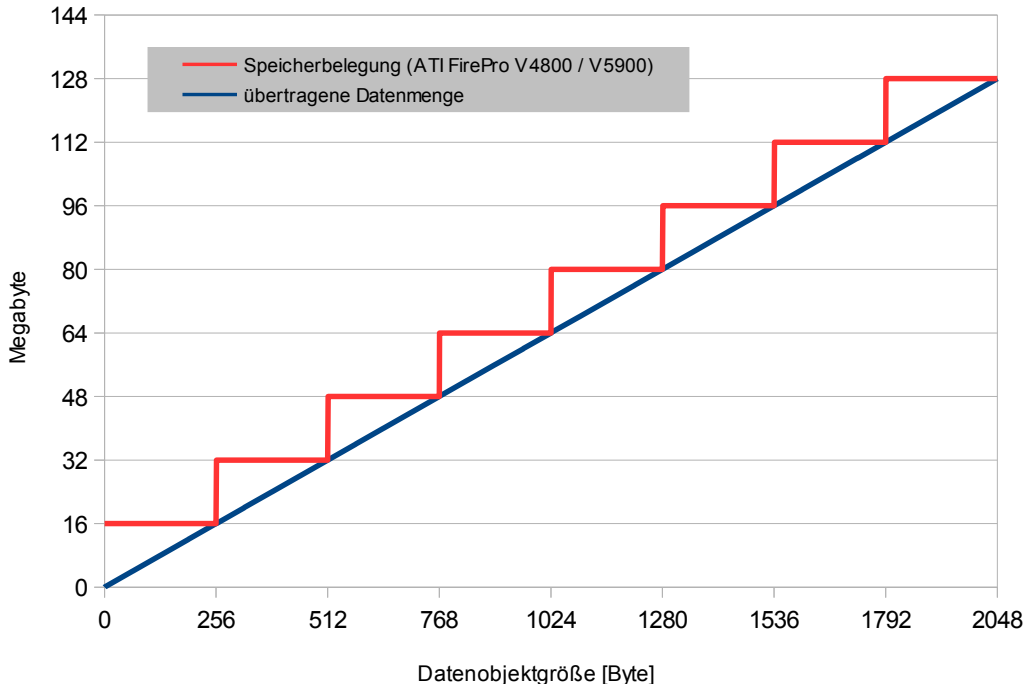

Abbildung 4.6: Speicherbelegung durch Erzeugung von 65.536 Datenobjekten (ATI-Systeme)

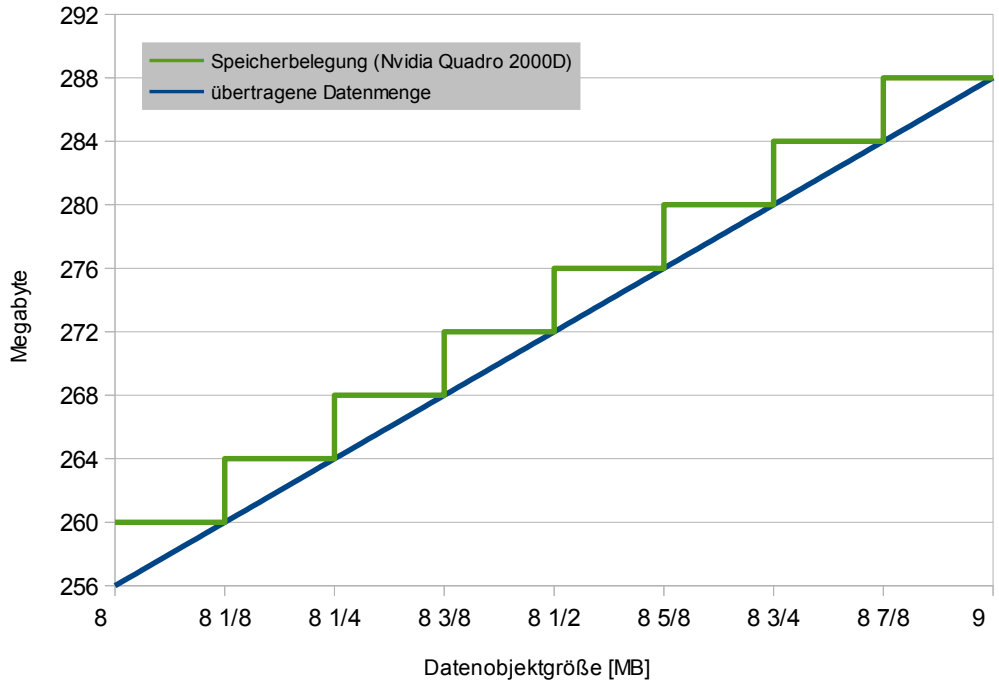

Abbildung 4.7: Speicherbelegung durch Erzeugung von 32 Datenobjekten (Nvidia-System)

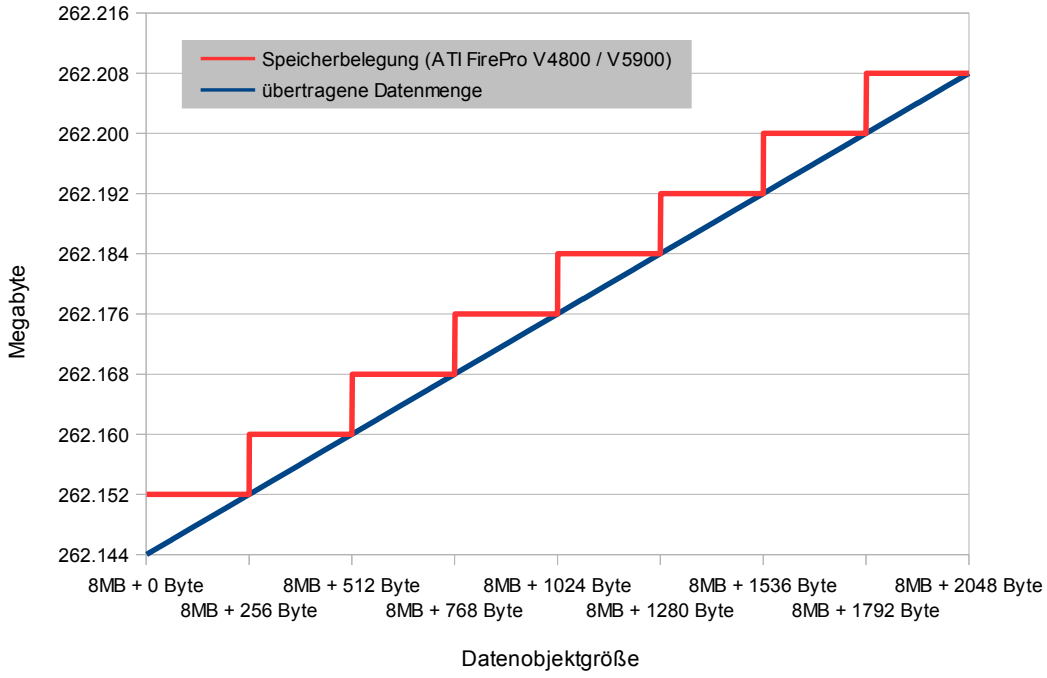

Abbildung 4.8: Speicherbelegung durch Erzeugung von 32 Datenobjekten (ATI-Systeme)

Datenobjektgrößen, die kein solches Vielfaches sind, führen zu einer Speicherbelegung, die der von Datenobjekten des jeweils nächstgrößeren Vielfachen von 256 Byte entsprechen. Dieser Trend bestätigt sich auch für die übrigen untersuchten Größenbereiche. Auf den beiden ATI-Systemen gibt es also offenbar nur eine Speichergranularität von 256 Byte, unabhängig von der Datenobjektgröße.

In keiner der Untersuchungen hatte der Usage Hint einen Einfluss auf die Speichergranularität (auch wenn auf den ATI-Systemen Datenobjekte bei entsprechenden Usage Hints im Hilfsspeicher abgelegt werden, vgl. Abschnitt 4.2.1.2). Im nächsten Abschnitt werden die Ergebnisse der Untersuchungen zur Speicherbelegung zusammengefasst.

4.2.6 Fazit Speicherbelegung

Wie zu Beginn von Kapitel 3.1 dargelegt wurde, kann das Auftreten von Eviction dazu führen, dass sich die Laufzeit von OpenGL ES-Befehlen erhöht. Das kann wiederum die Erfüllung von Echtzeitgarantien signifikant beeinträchtigen, da über OpenGL ES nicht gesteuert werden kann, welche Datenobjekte von Eviction betroffen sind – dadurch kann es auch bei der Ausführung eines Draw-Befehls einer kritischen Anwendung zu einer Eviction-Kaskade kommen, die die Laufzeit dieses Befehls derart verlängern könnte, dass keine ausreichend kleine Obergrenze für dessen Laufzeit mehr garantiert werden kann. Wenn aus diesem Grund das Auftreten von Eviction vermieden werden soll, muss vorhergesagt werden können, wie viel GPU-Speicher durch die Ausführung eines bestimmten OpenGL ES-Befehls belegt wird.

Die Ergebnisse der Untersuchungen zur Speicherbelegung zeigen aber, dass auf den Testsystemen nur für wenige Datenobjektgrößen die Menge des belegten GPU-Speichers exakt mit der Größe des erzeugten Datenobjekts übereinstimmt. Dies ist auf die Speichergranularität zurückzuführen, die auf allen drei Systemen größer ist als ein Byte. Auf dem „Nvidia Quadro 2000D“-System kommt erschwerend hinzu, dass manche Datenobjekte in Speicherblöcken abgelegt werden, was ebenfalls dafür sorgt, dass für solche Datenobjekte die Menge des belegten GPU-Speichers von ihrer Größe abweicht.

Sofern dieses spezielle Verhalten berücksichtigt wird und die Kennzahlen zu Speichergranularität und Speicherblockgröße bekannt sind, kann zumindest eine Obergrenze für den Speicherbedarf eines Datenobjekts angegeben werden: Sofern es sich um ein Datenobjekt handelt, das in Speicherblöcken abgelegt wird, entspricht diese Obergrenze der Speicherblockgröße. Falls es sich um ein anderes Datenobjekt handelt entspricht die Obergrenze der Effektivgröße dieses Datenobjekts (in diesem Fall entspricht die Obergrenze auch der Untergrenze für die Speicherbelegung).

Um eine genauere Obergrenze für Datenobjekten angeben zu können, die in Speicherblöcken abgelegt werden, ist es notwendig den Inhalt der einzelnen Speicherblöcke zu verfolgen. Sobald mehrere Speicherblöcke im GPU-Speicher reserviert wurden, die nicht vollständig mit Datenobjekten gefüllt sind, ist es mit Bordmitteln von OpenGL ES aber nicht möglich, vorherzusagen, in welchem Speicherblock ein neu erzeugtes Datenobjekt abgelegt wird (sofern es in mehreren Platz finden würde). In einer solchen Situation ist für nachfolgend erzeugte Datenobjekte nicht mehr in jedem Fall vorhersagbar, ob ein neuer Speicherblock reserviert werden wird oder nicht (d. h. es gilt wieder die Obergrenze der Speicherblockgröße).

Hinsichtlich der Echtzeitfähigkeit von OpenGL ES 2.0 führt aber die fehlende Möglichkeit, nachzuverfolgen, wo genau im GPU-Speicher Datenobjekte abgelegt werden, zu einem weitaus größeren Problem. [Stroyan 2009] erklärt, dass die Fragmentierung des GPU-Speichers, dazu führen kann, dass Datenobjekte unter Umständen nicht im GPU-Speicher abgelegt werden können, obwohl deren Größe kleiner ist als die Gesamtmenge an freiem GPU-Speicher. In diesem Fall kann es also durch die Erzeugung eines Datenobjekts zu Eviction kommen, selbst wenn der GPU-Speicher noch nicht vollständig belegt ist. Ohne die Möglichkeit, festzustellen, welche Bereiche des GPU-Speichers belegt sind, kann dies nicht sicher vorhergesehen werden.

Dieses Problem kann auf den ATI-Systemen umgangen werden, da hier die Größe des größten zusammenhängenden Speicherbereichs abgefragt werden kann [Blackmer u. a. 2009]. Sofern auf einem solchen System ein Datenobjekt erzeugt werden soll, dessen Effektivgröße kleiner ist als dieser Speicherbereich, kann davon ausgegangen werden, dass das betreffende Datenobjekt erzeugt werden kann, ohne dass es zu Eviction kommt. Auf den beiden ATI-Systemen kann also sichergestellt werden, dass die Echtzeitfähigkeit von OpenGL ES durch die Speicherbelegung nicht eingeschränkt wird.[3]

Auf Systemen, die nicht über einen Eviction-Mechanismus verfügen, kann es geschehen, dass Draw-Befehle kritischer Anwendungen nicht mehr ausgeführt werden können. Dies ist dann der Fall, wenn für diese Befehle bestimmte Daten im GPU-Speicher liegen müssen, dort aber nicht mehr ausreichend Platz dafür ist. Um dies zu verhindern, muss sichergestellt werden, dass für die Zwecke der kritischen Anwendungen immer genug freier Speicherplatz im GPU-Speicher vorhanden ist, d. h. dass andere Anwendungen nicht zu viele Daten im GPU-Speicher ablegen.

Dazu muss aber auch auf solchen Systemen vorhergesagt werden können, wie viel Speicherplatz durch die Erzeugung von Datenobjekten belegt wird. Sofern die dafür notwendigen Informationen nicht anderweitig in Erfahrung gebracht werden können (zum Beispiel von den Herstellern dieser Systeme), müssen diese Informationen durch die in Kapitel 3.1 beschriebenen Untersuchungen ermittelt werden. Eine notwendige Voraussetzung dafür ist aber, dass zumindest der zu einem bestimmten Zeitpunkt vorhandene, freie Speicherplatz des GPU-Speichers abgefragt werden kann. OpenGL ES 2.0 selbst bietet dafür aber keinen Mechanismus. Sofern keine Erweiterung dafür verfügbar ist (wie zum Beispiel bei den in Abschnitt 4.1.1 beschriebenen Systemen) ist dies auf der Ebene von OpenGL ES nicht möglich. Ein möglicher Lösungsansatz dafür, außerhalb von OpenGL ES, wird im Ausblick in Kapitel 5 skizziert. In den nächsten Abschnitten erfolgen die Untersuchungen zur Datenübertragung.

[3] Die Annahme dahinter ist, dass sich der größte zusammenhängende Speicherblock bei der Erzeugung eines Datenobjekts höchstens um dessen Effektivgröße verringert, zumal auf den beiden ATI-Systemen kein anderes Verhalten beobachtet werden konnte.

4.3 Datenübertragung

4.3.1 Bestimmung von Datenübertragungsrate und -laufzeit

4.3.1.1 Durchführung der Untersuchung

Algorithmus 4.12 Laufzeit von Datenübertragungsbefehlen

```
1  void testDataTransferTimes(
2      unsigned int minSize,      // minimale Datenobjektgröße
3      unsigned int maxSize,      // maximale Datenobjektgröße
4      unsigned int stepSize,     // Zunahme der Datenobjektgröße zwischen Messungen
5      unsigned int iterations,   // Anzahl der Einzelmessungen pro Datenobjektgröße
6      GLbyte *data)              // zu übertragende Daten
7  {
8      // Messreihe durchführen
9      long long results[iterations];
10     for (unsigned int size = minSize; size <= maxSize; size += stepSize)
11     {
12         // Datenobjekt erstellen
13         GLuint id;
14         glGenBuffers(1, &id);
15         glBindBuffer(GL_ARRAY_BUFFER, id);
16         glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
17         glFinish();
18
19         // Messung durchführen
20         HPCClock c;
21         for (unsigned int n = 0; n < iterations; n++)
22         {
23             c.start();
24             glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
25             glFlush();
26             glFinish();
27             results[n] = c.stop();
28         }
29
30         // Ergebnisse für aktuelle Datenobjektgröße speichern
31         writeToResults(size, results, glGetError());
32
33         // Datenobjekt freigeben
34         glDeleteBuffers(1, &id);
35         glFinish();
36     }
37 }
```

Um die Laufzeit von Datenübertragungsbefehlen zu bestimmen werden – wie in Kapitel 3.2.2 beschreiben – zunächst im leeren GPU-Speicher einzelne Datenobjekte zunehmender Größe mit `glBufferData`[4] erzeugt und die dessen Laufzeit gemessen. Dieses Vorgehen wird anschließend wiederholt, wobei der anfängliche Speicherbelegungsgrad und die Anzahl bereits vorhandener Datenobjekte erhöht wird.

[4] Siehe Kapitel 3.2.3.1 zur Wahl dieses Befehls.

Algorithmus 4.12 zeigt das dafür ausgeführt OpenGL ES-Programm. Die Schleife von Zeile 10 bis Zeile 35 enthält dabei den Code der Laufzeitmessung für eine Datenobjektgröße. In den Zeilen 13 bis 16 wird ein neues Datenobjekt angelegt. Der Parameter `GL_STATIC_DRAW` stellt für die verwendeten Testsysteme sicher, dass das Datenobjekt tatsächlich im GPU-Speicher angelegt wird (vgl. Kapitel 4.2.1).

Die eigentliche Laufzeitmessung wird in den Zeilen 23 bis 27 durchgeführt. Durch die Befehle `glFlush` und `glFinish` wird sichergestellt, dass der Datenübertragungsbefehl sofort an den GL-Server übertragen wird und die Datenübertragung vollständig abgeschlossen ist, bevor die Laufzeitmessung beendet wird (vgl. Kapitel 2.3).

Das zuvor angelegte Datenobjekt wird in Zeile 34 wieder freigegeben, um für die nachfolgenden Messungen (mit der nächstgrößeren zu übertragenden Datenmenge) die ursprüngliche Speicherbelegungssituation wiederherzustellen.

Diese Programm wurde für Datenobjektgrößen von einem Megabyte bis zur größten, auf dem jeweiligen System unterstützten Datenobjektgröße durchgeführt (512 MB auf dem „Nvidia Quadro 2000D“-System und 256 MB auf den beiden anderen Systemen). Zwischen den Messschritten wurde die Datenobjektgröße um jeweils ein Megabyte erhöht. Dies wurde mehrmals wiederholt, wobei der Speicherbelegungsgrad dabei jeweils um zehn Prozent bis zu einem Maximum von 90 Prozent erhöht wurde. Die Messung wurde stets dann abgebrochen, wenn es aufgrund der Datenobjektgröße zur Auslagerung von Datenobjekten in den Hauptspeicher kam (durch den sogenannten Eviction-Mechanismus, siehe auch Kapitel 3.1). Anschließend wurde die ursprüngliche Simulation ebenfalls mehrmals wiederholt, wobei die Anzahl der vorab vorhandenen Datenobjekte dabei jeweils um 10.000 bis zu einem Maximum von einer halben Million Datenobjekten erhöht wurde.

4.3.1.2 Ergebnisse

Abbildung 4.9 zeigt die Ergebnisse, die durch die Ausführung der Simulation bei leerem GPU-Speicher auf den drei Testsystemen ermittelt wurden.[5] Auf der X-Achse sind die verschiedenen Datenobjektgrößen aufgetragen und auf der Y-Achse die Laufzeit des Datenübertragungsbefehls.

Die dargestellten Messpunkte zeigen jeweils die mittlere Laufzeit des Datenübertragungsbefehls für die jeweilige Datenobjektgröße. Das Diagramm zeigt zur besseren Lesbarkeit nur einen Teil der ermittelten Messpunkte. Die ebenfalls dargestellten Trendlinien bleiben davon unberührt. Die Fehlerbalken über den einzelnen Messpunkten zeigen die durchschnittliche Abweichung der einzelnen Messwerte vom hier gezeigten Mittelwert.

Auf allen drei untersuchten Systemen steigt die Laufzeit von `glBufferData` proportional zur übertragenen Datenmenge. Die Varianzen der gemessenen Laufzeiten steigen bei allen drei Systemen mit zunehmender Datenobjektgröße an, überschreiten bei den beiden ATI-Systemen aber nie 15%. Anders beim Nvidia-System: Die Varianz für das größte Datenobjekt erreicht hier bei 512 MB großen Datenobjekten fast 46%.

Die Durchführung der Simulation bei unterschiedlich hohem Speicherbelegungsgrad lieferte gleiche Ergebnisse, ebenso die Simulation bei unterschiedlicher Anzahl bereits vorhandener Daten-

[5] Es ist dabei zu beachten, dass Anwendungen auf den Testsystemen über OpenGL ES nie auf einen vollständig leeren GPU-Speicher zugreifen können (siehe Kapitel 4.1.3.3).

objekte. Ob der Speicherbelegungsgrad oder die Anzahl vorhandener Datenobjekte dabei über den selben OpenGL-Kontext oder den Kontext eines anderen Prozesses erhöht wurde, hatte keinen Einfluss auf die Ergebnisse.

Bei der Laufzeitmessung für immer kleinere Datenobjekte zeigt sich, dass die Laufzeit des Datenübertragungsbefehls auf keinem der Testsysteme gegen Null geht. Die gemessenen Laufzeiten sinken auf dem „Nvidia Quadro 2000D“-System nie unter 49 µs, auf dem „ATI FirePro V5900“-System nie unter 51 µs und auf dem „ATI FirePro V4800“-System nie unter 56 µs. Für die Ausführung der Datenübertragung fällt also bei allen drei Systemen ein gewisser Overhead an, unabhängig davon, wie gering die zu übertragenden Datenmengen sind.

Auch für kleine Datenobjekte wurden die Simulationen zur Überprüfung des Einflusses der aktuellen Speicherbelegung und der Anzahl der bereits vorhandenen Datenobjekte durchgeführt. Abbildung 4.10 zeigt die Laufzeit von `glBufferData` für die Übertragung von 1kB Vertexdaten bei unterschiedlicher Anzahl von Datenobjekten im GPU-Speicher. Auf der X-Achse ist die Anzahl der Datenobjekte aufgetragen und auf der Y-Achse Laufzeiten des Datenübertragungsbefehls.

Hierbei zeigte sich, dass die zuvor ermittelten Minimallaufzeiten unabhängig von der aktuellen Speicherbelegung auftreten. Auf den beiden ATI-Systemen hat auch die Anzahl der bereits vorhandenen Datenobjekte keinen erkennbaren Einfluss darauf. Beim Nvidia-System steigt hingegen die minimale Laufzeit sprunghaft auf 61 µs an, sobald mindestens 150.000 Datenobjekte im GPU-Speicher vorhanden sind. Auch hier hat es keinen Einfluss auf die Ergebnisse, ob der Speicherbelegungsgrad oder die Anzahl vorhandener Datenobjekte über den selben OpenGL-Kontext oder den Kontext eines anderen Prozesses erhöht wird. Im nächsten Abschnitt erfolgt die Untersuchung des Verhaltens hinsichtlich der Ausführung von konkurrierenden Datenübertragungsbefehlen.

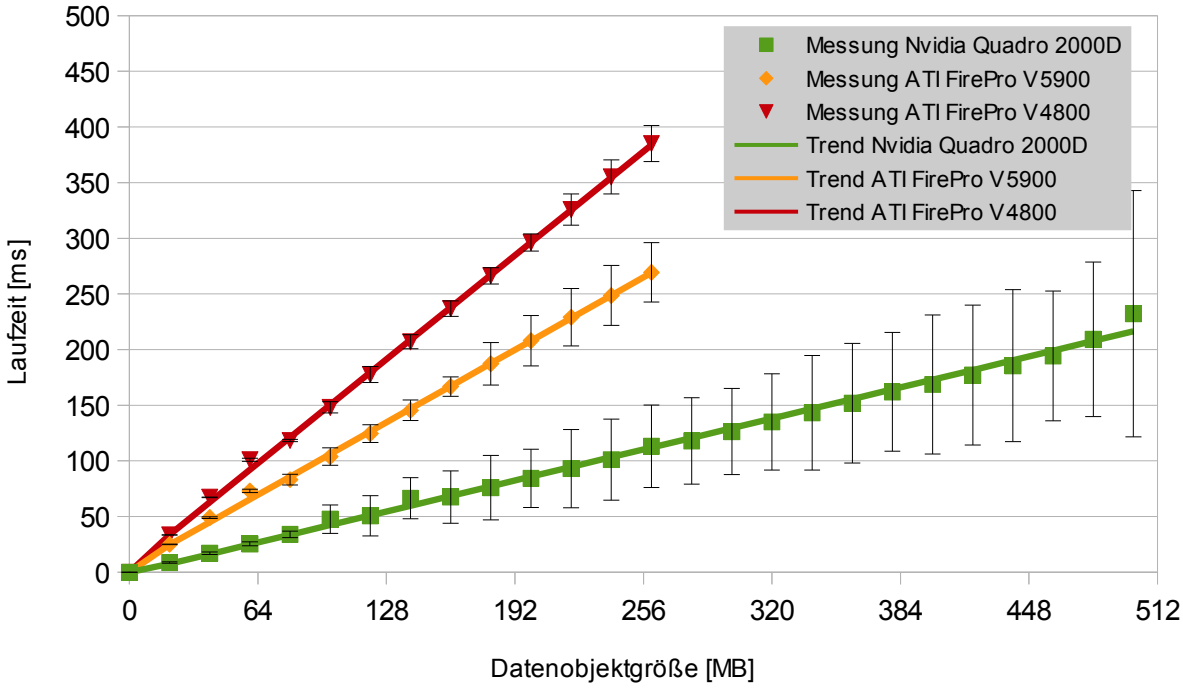

Abbildung 4.9: Datenübertragungszeiten bei leerem GPU-Speicher

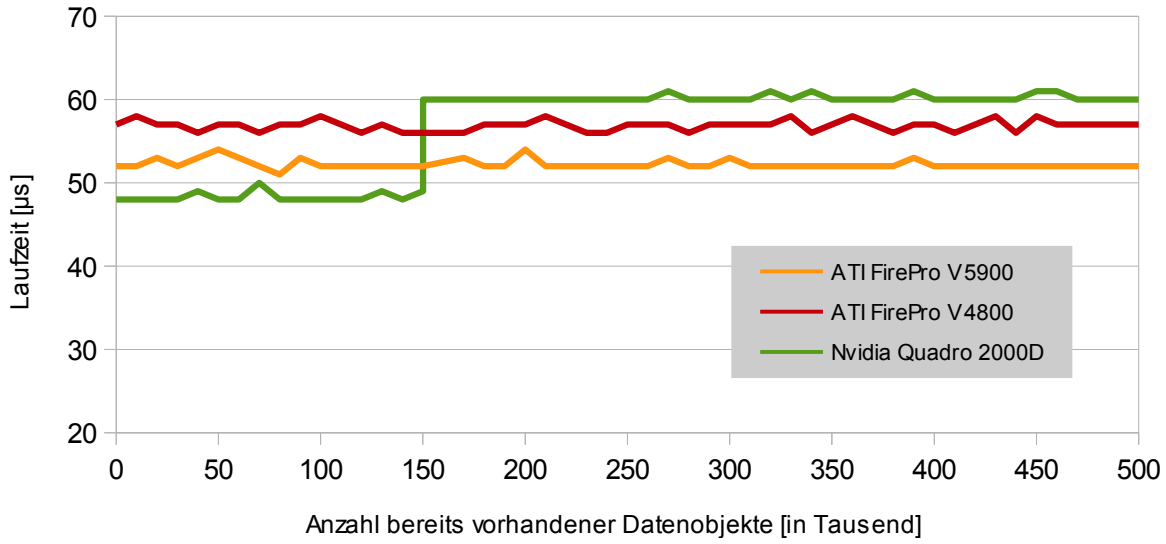

Abbildung 4.10: Laufzeit von `glBufferData` für die Übertragung von 1kB Vertexbuffer-Daten, wenn bereits andere Datenobjekte im GPU-Speicher vorhanden sind

4.3.2 Konkurrierende Datenübertragungen

4.3.2.1 Durchführung der Untersuchung

Um das Verhalten des GL-Servers bei konkurrierenden Datenübertragungsbefehlen zu ermitteln, werden – wie in Kapitel 3.2.2.2 beschrieben – zwei OpenGL ES-Programme ausgeführt, die als Master- und als Slaveprogramm bezeichnet werden. Diese Programme übermitteln gleichzeitig einen Datenübertragungsbefehl an den GL-Server und messen dessen Laufzeit. Anschließend wird dieser Vorgang wiederholt, wobei die Übermittlung des Datenübertragungsbefehl beim Masterprogramm solange verzögert wird, bis die Ausführung des vom Slaveprogramm übermittelten Befehls zur Hälfte abgeschlossen ist.

Algorithmus 4.13 zeigt den Code des Masterprogramms. Das Datenobjekt für die Datenübertragung wird in Zeile 7 auf die gleiche Weise angelegt wie in Algorithmus 4.12, so dass für alle Testsysteme sichergestellt ist, dass das Datenobjekt tatsächlich im GPU-Speicher erzeugt wird. In den Zeilen 10–19 erfolgt die Referenzmessung der Laufzeit des Datenübertragungsbefehls und in den Zeilen 22–23 wird das Slaveprogramm gestartet und darauf gewartet, dass es Bereitschaft signalisiert.

Die eigentliche Messung bei konkurrierender Datenübertragung erfolgt in den Zeilen 27–37. Dabei wird in Zeile 29 der Beginn der Datenübertragung signalisiert (woraufhin das Slaveprogramm seinen Datenübertragungsbefehl an den GL-Server übermittelt) und in Zeile 30 die Ausführung des Masterprogramms bei Bedarf für die Hälfte der durchschnittlichen Referenzlaufzeit angehalten.

Algorithmus 4.13 Laufzeit konkurrierender Datenübertragungsbefehle (Masterprogramm)

```
1  void testCompetingDataTransferMaster(
2      unsigned int size,          // zu übertragende Datenmenge
3      unsigned int iterations,    // Anzahl der Einzelmessungen pro Datenobjektgröße
4      GLbyte *data,               // zu übertragende Daten
5      bool delayMaster)           // Übermittlung des Datenübertragungsbefehls verzögern?
6  {
7      (...) // Datenobjekt erstellen
8
9      // Referenzmessung durchführen
10     HPCClock c;
11     long long refresults[iterations];
12     for (unsigned int n = 0; n < iterations; n++)
13     {
14         c.start();
15         glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
16         glFlush();
17         glFinish();
18         refresults[n] = c.stop();
19     }
20
21     // Slaveprogramm starten und warten, bis es Bereitschaft signalisiert
22     spawnSlave();
23     waitForSlaveEvent();
24
25     // Messung mit konkurrierender Datenübertragung durchführen
26     long long results[iterations], timestamps[iterations];
27     for (unsigned int n = 0; n < iterations; n++)
28     {
29         signalMasterEvent();
30         if (delayMaster) Sleep(average(refresults)/2);
31         timestamps[n] = c.start();
32         glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
33         glFlush();
34         glFinish();
35         results[n] = c.stop();
36         waitForSlaveEvent();
37     }
38
39     // Messergebnisse speichern und Datenobjekt freigeben
40     writeToResults(size, refresults, results, glGetError());
41     glDeleteBuffers(1, &bufferID);
42 }
```

In Zeile 31, unmittelbar vor Übermittlung des Datenübertragungsbefehls an den GL-Server, wird der aktuelle Zeitstempel gespeichert (um später durch den Vergleich mit den Zeitstempeln des Slaveprogramms sicherstellen zu können, dass die Zeitpunkte der Übermittlung der beiden Befehle an den GL-Server zeitlich nicht zu stark auseinander lagen, vgl. Abschnitt 4.1.5). Nach Ausführung der Datenübertragung wartet das Masterprogramm in Zeile 36 darauf, dass das Slaveprogramm den Abschluss seiner Datenübertragung signalisiert.

Algorithmus 4.14 Laufzeit konkurrierender Datenübertragungsbefehle (Slaveprogramm)

```
1  void testCompetingDataTransferSlave(
2      unsigned int size,          // zu übertragende Datenmenge
3      unsigned int iterations,    // Anzahl der Einzelmessungen
4      GLbyte *data)               // zu übertragende Daten
5  {
6      (...) // Datenobjekt erstellen
7
8      (...) // Referenzmessung durchführen
9
10     // Bereitschaft signalisieren
11     setSlaveEvent();
12
13     // Messung mit konkurrierender Datenübertragung durchführen
14     long long results[iterations], timestamps[iterations];
15     for (unsigned int n = 0; n < iterations; n++)
16     {
17         waitForMasterEvent();
18         timestamps[n] = c.start();
19         glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
20         glFlush();
21         glFinish();
22         results[n] = c.stop();
23         signalSlaveEvent();
24     }
25
26     (...) // Messergebnisse speichern und Datenobjekt freigeben
27 }
```

Der in Algorithmus 4.14 in Auszügen gezeigte Code des Slaveprogramms entspricht weitestgehend dem Code des Masterprogramms. Die Erstellung des Datenobjekts (Zeile 6) und die Referenzmessung (Zeile 8) erfolgen genauso wie im Masterprogramm (eine Referenzmessung wird auch beim Slaveprogramm durchgeführt, um sicherzustellen, dass sich die Laufzeiten der Datenübertragung zwischen den beiden Programmen im nicht-konkurrierenden Fall nicht unterscheiden – wäre dies nicht der Fall, dann wäre ein Vergleich der Laufzeiten im konkurrierenden Fall nicht sinnvoll). Die Speicherung der Messergebnisse und Freigabe des Datenobjekts in Zeile 26 entsprechen ebenfalls denen des Masterprogramms.

Die Bereitschaft des Slaveprogramms wird in Zeile 11 signalisiert. Auf dieses Signal wird in Zeile 23 des Masterprogramms gewartet. In gleicher Weise korrespondieren die Zeilen 17 und 23 des Slaveprogramms mit den Zeilen 29 und 36 des Masterprogramms. Durch diese Signale synchronisieren sich die beiden Programme so, dass ihre Datenübertragungsbefehle entweder gleichzeitig an den GL-Server übermittelt werden oder das Masterprogramm seinen übermittelt, wenn die Datenübertragung des Slaveprogramms zur Hälfte abgeschlossen ist.

4.3.2.2 Ergebnisse

Abbildung 4.11 zeigt die Ergebnisse der Untersuchung auf dem „Nvidia Quadro 2000D“-System für die Übertragung von 200MB Vertexbuffer-Daten. Auf der Y-Achse ist die Laufzeit des Datenüber-

tragungsbefehls aufgetragen. Die linke Balkengruppe zeigt die Ergebnisse für die Simulation ohne Verzögerung des Masterprogramms, die rechte Balkengruppe die Ergebnisse, wenn die Übermittlung des Datenübertragungsbefehls an den GL-Server durch das Masterprogramm um die Hälfte der Referenzlaufzeit verzögert wird (in diesem Fall um 42 ms).

Der linke Balken innerhalb einer Gruppe zeigt das Mittel der gemessenen Referenzlaufzeiten. Dieses wird durch die in den Algorithmen 4.13 und 4.14 beschriebene Simulation sowohl vom Master- als auch vom Slaveprogramm ermittelt – die mittleren Laufzeiten stimmen dabei in allen Simulationsläufen stets überein (die Abweichung zwischen der mittleren Laufzeit beim Master- und beim Slaveprogramm liegt hierbei jeweils unter 2%), weshalb die Referenzzeit in dieser Abbildung nur einmal aufgeführt wird. Der mittlere Balken zeigt das Mittel der vom Masterprogramm gemessenen Laufzeiten und der rechte Balken das des Slaveprogramms. Die Fehlerbalken zeigen die durchschnittliche Abweichung der einzelnen Messwerte vom hier gezeigten Mittelwert.

Es zeigt sich, dass die Laufzeiten im Master- und Slaveprogramm übereinstimmen, und zwar in beiden Szenarien (mit und ohne Verzögerung im Masterprogramm). Das bedeutet, dass Datenübertragungsbefehle, die von verschiedenen Prozessen an den GL-Server übermittelt werden, nicht sequentiell abgearbeitet werden. Sobald dem GL-Server mehr als ein Datenübertragungsbefehl zur Abarbeitung vorliegt, wird die verfügbare Bandbreite auf die Ausführung aller Befehle verteilt (wodurch sich die Laufzeiten dieser Befehle entsprechend erhöhen).

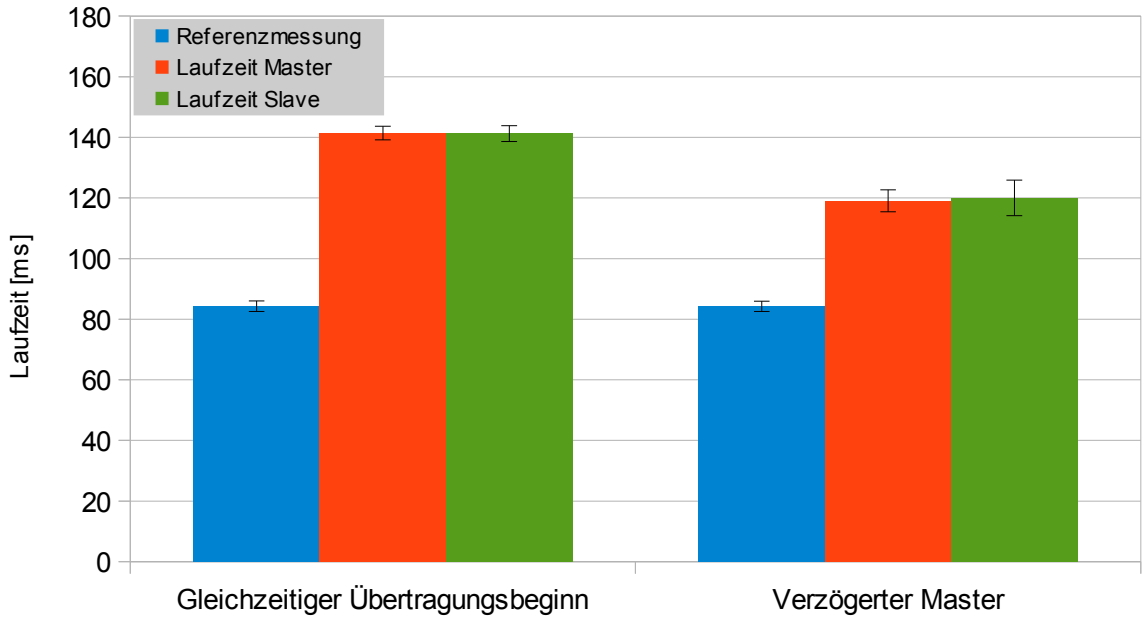

Abbildung 4.11: Laufzeiten konkurrierender Datenübertragungsbefehle

Die Simulation wurde für unterschiedliche zu übertragende Datenmengen wiederholt und auch auf den beiden ATI-Systemen durchgeführt. Auch dort zeigte sich, dass konkurrierende Datenübertragungsbefehle vom OpenGL ES-System nicht sequentiell abgearbeitet werden, sondern dass die verfügbare Bandbreite stattdessen auf die einzelnen Befehle aufgeteilt wird.

Bei diesen Simulationen ist allerdings zu beachten, dass die zu übertragenden Datenmengen nicht zu klein gewählt werden dürfen. Spätestens wenn die zu übertragende Datenmenge so klein ist, dass die Laufzeit des Datenübertragungsbefehls auf den Testsystemen den Bereich um eine Millisekunde erreicht, kann es aufgrund von ungünstigen Prozesskontextwechseln oder der Ungenauigkeit des Sleep-Befehls dazu kommen, dass der Datenübertragungsbefehl eines Programms erst übermittelt wird, nachdem der Befehl des anderen Programms bereits vollständig abgearbeitet worden ist (vgl. Abschnitt 4.1.1 zur Frequenz des Thread-Schedulers und der Genauigkeit des Sleep-Befehls). Es handelt sich dann nicht mehr um konkurrierende Befehle und das hier gewählte Vorgehen macht keinen Sinn mehr.

4.3.3 Nebenläufige Ausführung von Datenübertragung und Rendering

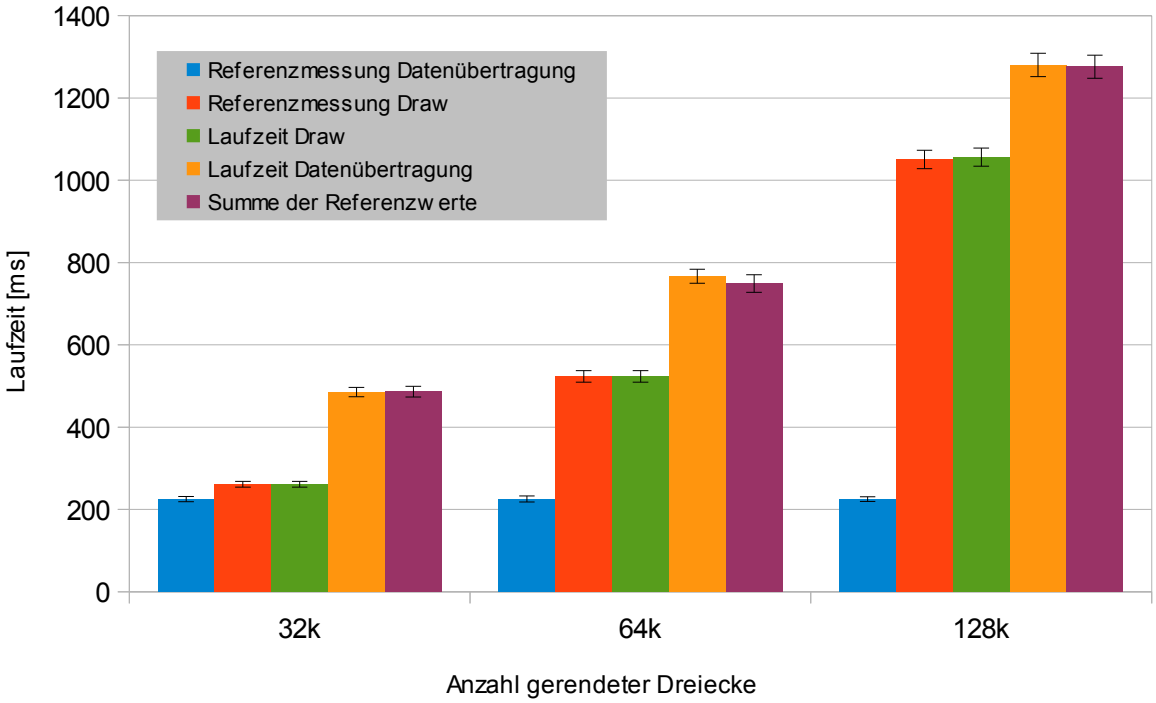

Abbildung 4.12: Laufzeiten bei Datenübertragung und gleichzeitigem Draw

Auch zur Untersuchung der nebenläufigen Ausführung von Datenübertragungs- und Draw-Befehlen werden – wie in Kapitel 3.2.2.3 beschrieben – zwei OpenGL ES-Programme ausgeführt, die als Master- und Slaveprogramm bezeichnet werden. Der Aufbau dieser Programme entspricht dem im letzten Abschnitt erläuterten (siehe die Algorithmen 4.13 und 4.14).

Lediglich das Masterprogramm unterscheidet sich insofern, dass für diese Untersuchung für die Referenz- und die konkurrierende Laufzeitmessung in den Zeilen 15 und 32 kein Datenübertragungsbefehl sondern ein Draw-Befehl an den GL-Server übermittelt wird.

Abbildung 4.12 zeigt die Laufzeiten der sich zeitlich überschneidenden Draw- und Datenübertragungsbefehle auf dem „Nvidia Quadro 2000D“-System. Auf der X-Achse ist die Anzahl der im Zuge des Draw-Befehls gerenderten Dreiecke aufgetragen und auf der Y-Achse die Laufzeit in Millisekunden. Die Balken zeigen jeweils das Mittel der gemessenen Laufzeiten und die darüber dargestellten Fehlerbalken die durchschnittliche Abweichung der Messwerte vom gezeigten Mittelwert.

Für diese Simulation wurde die vom Slaveprogramm übertragene Datenmenge nicht variiert. Es wurde stets die größtmögliche Datenmenge übertragen (für Vertexbuffer-Objekte sind dies 512 MB beim Nvidia-System und 256 MB bei den beiden ATI-Systemen). Variiert wurde die Menge der zu rendernden Dreiecke beim Masterprogramm. Die Simulation wurde pro Anzahl Dreiecke jeweils in zwei Varianten durchgeführt. In der einen erfolgte die Übermittlung des Draw- und des Datenübertragungsbefehls zeitgleich, in der anderen wurde die Übermittlung des Draw-Befehls um 150 ms verzögert (d. h. zum Zeitpunkt der Übermittlung des Draw-Befehls war die Datenübertragung bereits weit fortgeschritten, aber noch nicht abgeschlossen). Das Ergebnis war aber immer das gleiche:

Die Laufzeit des Draw-Befehls entspricht immer der Referenzlaufzeit und die Laufzeit des Datenübertragungsbefehls entspricht immer in etwa der Summe der Referenzlaufzeiten von Draw- und Datenübertragungsbefehl. Dass dies auch bei deutlicher Verzögerung der Übermittlung des Draw-Befehls der Fall ist, deutet darauf hin, dass der GL-Server die Ausführung des Datenübertragungsbefehls unterbricht, sobald ihm ein Draw-Befehl übermittelt wird, und sie fortsetzt, sobald der Draw-Befehl vollständig ausgeführt worden ist.

Die beiden ATI-Systeme zeigen bei Ausführung dieser Simulation das gleiche Verhalten. Auch hier werden Datenübertragungsbefehle durch konkurrierende Draw-Befehle unterbrochen, während die Ausführung von Draw-Befehlen durch Datenübertragungsbefehle anderer Prozesse nicht messbar beeinflusst wird.

4.3.4 Datenübertragung ungepufferter Draw-Befehle

4.3.4.1 Durchführung der Untersuchung

Um zu überprüfen, ob bei der Ausführung eines ungepufferten Draw-Befehls die benötigten Vertexdaten in die GPU übertragen werden können, während gleichzeitig das Rendering durchgeführt wird, wird – wie in Kapitel 3.2.2.4 beschrieben – die Laufzeit des ungepufferten Draw-Befehls mit der Datenübertragungszeit für diese Vertexdaten und mit der Laufzeit des gepufferten Renderings verglichen. Dazu wird das in Algorithmus 4.15 gezeigte Programm ausgeführt.

Der Code für das Anlegen des Vertexbuffer-Objekts im GPU-Speicher und die Messung der Referenzlaufzeit für die Datenübertragung vom Hauptspeicher in dieses Datenobjekt entspricht dem von Algorithmus 4.13. In den Zeilen 9–11 wird festgelegt, dass dieses Vertexbuffer-Objekt für nachfolgende Draw-Befehle genutzt werden soll (d. h. nachfolgende Draw-Befehle sind gepufferte Draw-Befehle, für die keine eigene Datenübertragung notwendig ist).

Die Messung der Referenzlaufzeit des Draw-Befehls ohne Datenübertragung erfolgt dann in den Zeilen 14–21. Die Ermittlung der Referenzlaufzeit für die Kopie der Vertexdaten im Hauptspeicher erfolgt in den Zeilen 24–30.[6]

[6] Die Größe des Vertexdatenarrays in Bytes entspricht dem Produkt aus der Anzahl der Vertices und der Größe eines Vertex in Bytes. In diesem Fall wird ein Vertex durch zwei GLfloat-Variablen repräsentiert (für die X- und Y-Koordinate des entsprechenden Vertex).

Algorithmus 4.15 Datenübertragung ungepufferter Draw-Befehle

```
1  void testDataTransferInUnbufferedDraw(
2      unsigned int iterations,   // Anzahl der Einzelmessungen
3      unsigned int numVertices, // Anzahl der zu rendernden Vertices
4      GLbyte *data)             // Vertexdaten
5  {
6      (...) // Erzeugung Datenobjekt + Referenzmessung Datenübertragungszeit
7
8      // Angelegtes Datenobjekt als Input für Draw-Befehle festlegen
9      glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
10     glEnableVertexAttribArray(0);
11     glFinish();
12
13     // Referenzmessung Draw-Laufzeit
14     for (unsigned int n = 0; n < iterations; n++)
15     {
16         c.start();
17         glDrawArrays(GL_POINTS, 0, numVertices);
18         glFlush();
19         glFinish();
20         refResultsDraw[n] = c.stop();
21     }
22
23     // Referenzmessung Laufzeit Datenkopie im Hauptspeicher
24     GLfloat tmp[numVertices*sizeof(2*GLfloat)];
25     for (unsigned int n = 0; n < iterations; n++)
26     {
27         c.start();
28         memcpy(tmp, data, numVertices*sizeof(2*GLfloat));
29         refResultsMemCopy[n] = c.stop();
30     }
31
32     // Die im Hauptspeicher liegenden Vertexdaten als Draw-Input festlegen
33     glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, data);
34     glEnableVertexAttribArray(0);
35     glFinish();
36
37     // Ungepufferte Draw-Laufzeiten messen
38     for (unsigned int n = 0; n < iterations; n++)
39     {
40         c.start();
41         glDrawArrays(GL_POINTS, 0, ud->numberOfPoints);
42         returnTimes[n] = c.getTime();
43         glFlush();
44         glFinish();
45         results[n] = c.stop();
46     }
47
48     // Ergebnisse speichern
49     writeToResults(numVertices, refResultsDataTransfer, refResultsDraw, refResultsMemCopy,
50         returnTimes, results, glGetError());
51 }
```

Die Zeilen 33–35 legen fest, dass nachfolgende Draw-Befehle das Vertexdatenarray im Hauptspeicher als Input nutzen sollen (d. h. nachfolgende Draw-Befehle werden ungepuffert ausgeführt).

Die Laufzeitmessung für den ungepufferten Draw-Befehl findet dann in den Zeilen 38–46 statt. Dabei werden die Rücksprungzeiten des Draw-Befehls gesondert vermerkt (Zeile 42). Die Ergebnisse der Referenzmessungen, der Laufzeitmessung des ungepufferten Draw-Befehls und dessen Rücksprungzeiten werden schließlich in den Zeilen 49–50 gespeichert.

Auch in dieser Simulation werden im Zuge der Ausführung der Draw-Befehle die beiden Minimalshader verwendet, die in Abschnitt 4.1.6 beschrieben werden.

4.3.4.2 Ergebnisse

Die Abbildungen 4.13 und 4.14 zeigen die Ergebnisse der Simulation auf dem „Nvidia Quadro 2000D“- und dem „ATI FirePro V4800“-System. Auf den X-Achsen ist die Anzahl der gerenderten Vertices aufgetragen und auf den Y-Achsen die Laufzeit in Millisekunden. Die Balken zeigen jeweils das Mittel der gemessenen Laufzeiten und die darüber dargestellten Fehlerbalken die durchschnittliche Abweichung der Messwerte vom gezeigten Mittelwert.

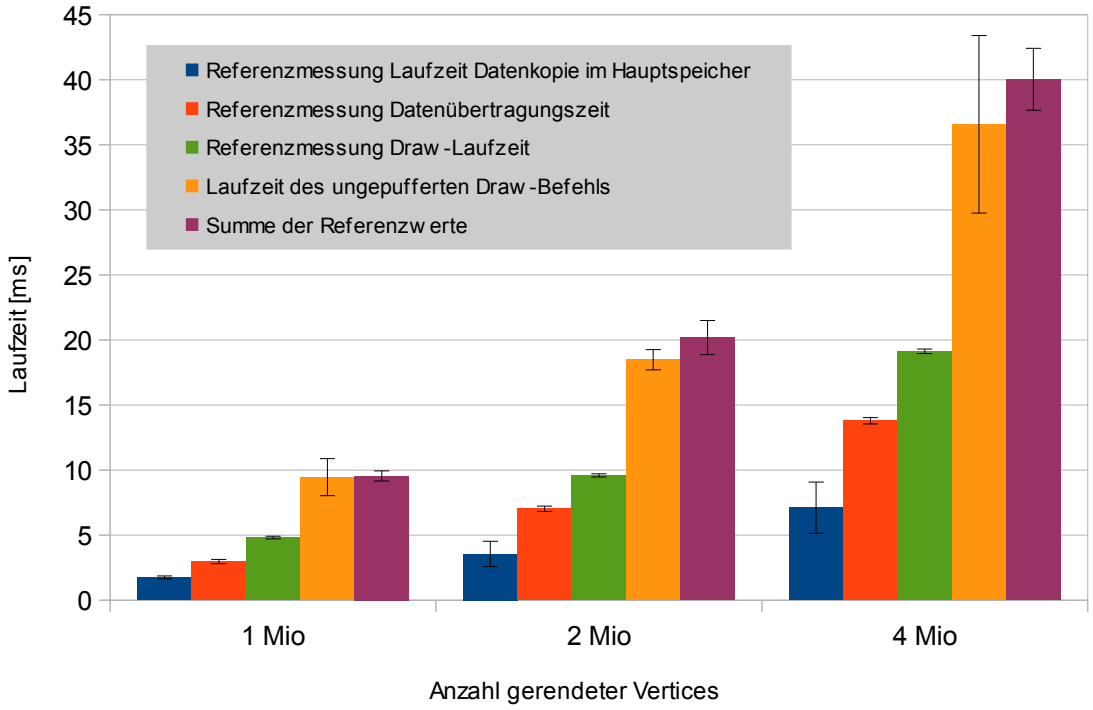

Abbildung 4.13: Laufzeiten ungepufferter Draw-Befehle („Nvidia Quadro 2000D“-System)

Auf diesem System sind die ermittelten Laufzeiten des ungepufferten Draw-Befehls deutlich größer als die Summe der Referenzwerte für die Datenübertragung und die Laufzeit des gepufferten Draw-Befehls. Da gleichzeitig die Rücksprungzeiten des ungepufferten Draw-Befehls unter der Referenzlaufzeit für die Datenübertragung liegen, muss davon ausgegangen werden, dass das OpenGL ES-System eine Kopie der Vertexdaten im Hauptspeicher anlegt (sonst wäre ein so früher Rücksprung

nicht möglich). Tatsächlich liegen die Rücksprungzeiten auch über der Referenzlaufzeit für die Datenkopie im Hauptspeicher.

Dass die Laufzeiten des ungepufferten Draw-Befehls dennoch kleiner sind als die Summe aller drei Referenzwerte, könnte vom OpenGL ES-System zum Beispiel dadurch erreicht werden, dass die Kopie im Hauptspeicher in einzelnen Tranchen durchgeführt wird. Nach dem Kopieren der ersten Tranche könnte deren Übertragung in die GPU erfolgen, während bereits die nächste Tranche im Hauptspeicher kopiert wird. Ob tatsächlich auf diese Weise eine Datenkopie im Hauptspeicher angelegt wird, konnte auf dem Testsystem jedoch nicht abschließend geklärt werden.

Da die Laufzeiten des ungepufferten Draw-Befehls größer sind als die Summe der Referenzlaufzeiten von Datenübertragung und gepufferten Draw, ist das „Nvidia Quadro 2000D“-System offensichtlich auch im Falle eines ungepufferten Draw-Befehls nicht in der Lage, Datenübertragung und Rendering nebenläufig durchzuführen.

Anders sehen die in Abbildung 4.14 gezeigten Ergebnisse für das „ATI FirePro V4800“-System aus: Hier entsprechen die Laufzeiten des ungepufferten Draw-Befehls der ermittelten Referenzlaufzeit für die Datenübertragung (wobei die mittlere Laufzeit des Draw-Befehls unwesentlich größer ist als die der Datenübertragung allein).

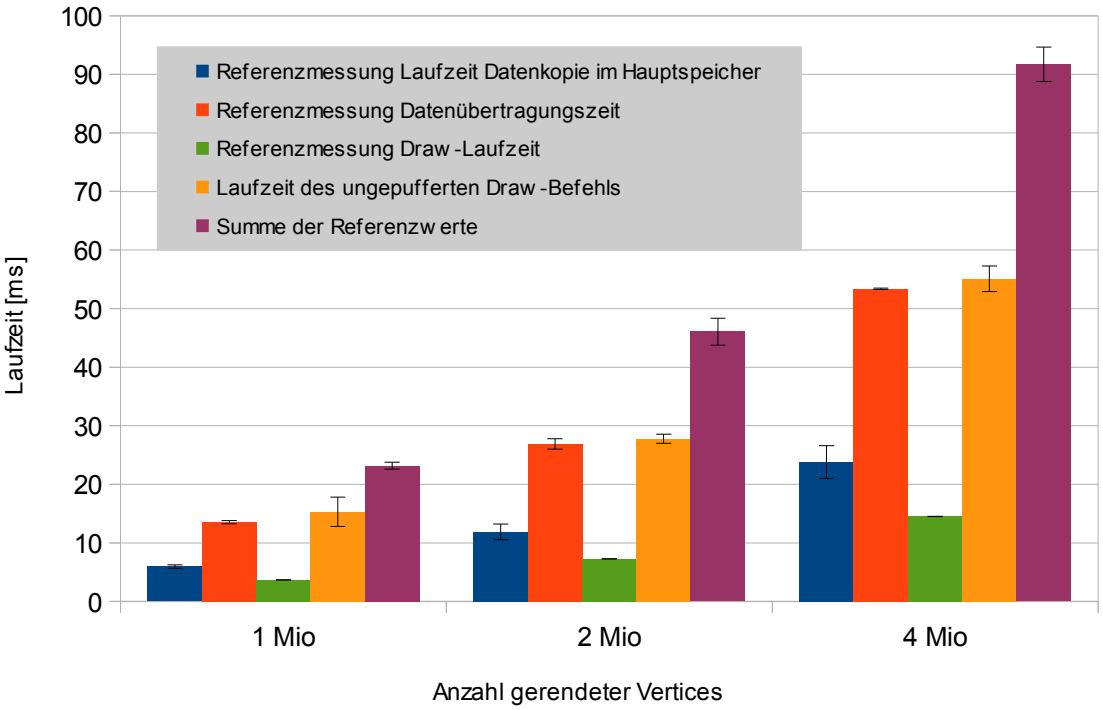

Abbildung 4.14: Laufzeiten ungepufferter Draw-Befehle („ATI FirePro V4800“-System)

Offensichtlich ist dieses System im Gegensatz zum „Nvidia Quadro 2000D“-System in der Lage, bei der Ausführung von ungepufferten Draw-Befehlen Datenübertragung und Rendering nebenläufig durchzuführen. Die Ausführung der Simulation auf dem „ATI FirePro V5900“-System erbrachte das gleiche Ergebnis: Auch dieses System ist offensichtlich dazu in der Lage.

4.3.5 Fazit Datenübertragung

Wie zu Beginn von Kapitel 3.2 dargelegt, ist es im Hinblick auf die Erfüllung von Echtzeitgarantien für die Befehle kritischer Anwendungen notwendig, zumindest eine Obergrenze für die Laufzeit von Datenübertragungsbefehlen garantieren zu können und zu wissen, inwieweit sich konkurrierende Datenübertragungen sowie Rendering und Datenübertragung gegenseitig beeinträchtigen.

Auf den drei untersuchten Testsystemen kann die Laufzeit von Datenübertragungsbefehlen vorhergesagt werden wenngleich die Ergebnisse der Laufzeitmessungen eine gewisse Varianz aufweisen. Je kleiner die übertragenen Datenmengen sind, umso kleiner sind diese Varianzen. Auf den beiden ATI-Systemen liegen sie sogar für die größten Datenobjekte im Mittel unter 15%.

Allerdings muss ein Scheduler mit dem Ziel, Echtzeitgarantien für Befehle kritischer Anwendungen zu erfüllen, den Einfluss von Draw-Befehlen und konkurrierenden Datenübertragungen berücksichtigen. Datenübertragungsbefehle werden auf den drei Testsystemen nicht sequentiell abgearbeitet. Stattdessen werden sie nebenläufig ausgeführt, wobei die verfügbare Bandbreite auf die konkurrierenden Datenübertragungsbefehle aufgeteilt wird. Sofern die Ausführbarkeit eines Draw-Befehls einer kritischen Anwendung davon abhängt, dass zuvor bestimmte Daten in den GPU-Speicher übertragen werden, können konkurrierende Datenübertragungsbefehle anderer Anwendungen dafür sorgen, dass sich die Laufzeit des Datenübertragungsbefehls der kritischen Anwendung derart verlängert, dass der davon abhängige Draw-Befehl nicht mehr rechtzeitig ausgeführt werden kann.

Um die Laufzeit von kritischen Datenübertragungen nicht zu beeinflussen, muss der Scheduler ggf. die Übermittlung von Datenübertragungsbefehlen anderer Anwendungen an den GL-Server verzögern, bis die kritische Datenübertragung abgeschlossen ist. Gleiches gilt für Draw-Befehle anderer Anwendungen, durch deren Ausführung laufende Datenübertragungen unterbrochen werden. Umgekehrt haben Datenübertragungen anderer Anwendungen auf den drei Testsystemen keinen messbaren Einfluss auf die Laufzeit von Draw-Befehlen. Es ist aber zu beachten, dass Datenübertragungen, die durch die Ausführung eines Draw-Befehls unterbrochen wurden, fortgesetzt werden, sobald der Draw-Befehl ausgeführt ist.

Die Laufzeit von ungepufferten Draw-Befehlen – die wie Datenübertragungsbefehle wirken – kann vorhergesagt werden. Zusätzlich zur Datenübertragungszeit der dabei verwendeten Vertexdaten hängt sie jedoch von weiteren Faktoren ab, nämlich

- davon, ob bei der Ausführung eines ungepufferten Draw-Befehls die Datenübertragung und das Rendering nebenläufig ausgeführt werden (wie bei den beiden ATI-Systemen) oder nicht (wie beim Nvidia-System), und
- von der Laufzeit des gepufferten Renderings (siehe Kapitel 2.6.3).

Falls eine nebenläufige Durchführung von Datenübertragung und Rendering erfolgt, dann gleicht die Laufzeit dem Maximum aus der Laufzeit des gepufferten Renderings und der Datenübertragung. Falls Datenübertragung und Rendering nicht nebenläufig durchgeführt werden, dann gleicht sie der Laufzeitensumme von gepuffertem Rendering und Datenübertragung. Während die Datenübertragungszeit vorhergesagt werden kann, stellt jedoch die Laufzeit des gepufferten Renderings ein Problem dar – siehe die Untersuchungen zu diesem Thema in den nächsten Abschnitten.

4.4 Pipelinenutzung

4.4.1 Ausführung konkurrierender Draw-Befehle

4.4.1.1 Durchführung der Untersuchung

Um zu klären, ob konkurrierende Draw-Befehle nebenläufig ausgeführt werden können, werden – wie in Kapitel 3.3.2.1 beschrieben – zwei OpenGL ES-Programme ausgeführt, die als Master- und Slaveprogramm bezeichnet werden. Beide Programme übermitteln gleiche Draw-Befehle an den GL-Server, wobei immer zuerst der Befehl des Masterprogramms und unmittelbar darauf der des Slaveprogramms übermittelt wird. Dies wird für unterschiedliche Anzahlen zu rendernder Dreiecke wiederholt.

Algorithmus 4.16 Untersuchung konkurrierender Draw-Befehle (Masterprogramm).

```
1  void testCompetingDrawMaster(
2      unsigned int numTriangles, // Anzahl zu rendernder Dreiecke
3      unsigned int iterations)   // Anzahl Iterationen für die Laufzeitmessung
4  {
5      // Referenzmessung
6      HPCClock c;
7      long long results[iterations];
8      for (unsigned int n = 0; n < iterations; n++)
9      {
10         c.start();
11         glDrawArrays(GL_POINTS, 0, numTriangles * 3);
12         glFlush();
13         glFinish();
14         writeToResults(c.stop(), glGetError());
15     }
16
17     // Slaveprogramm starten und warten, bis es bereit ist
18     spawnSlave();
19     WaitForSlaveEvent();
20
21     // Messung mit konkurrierenden Draw-Befehlen durchführen
22     for (unsigned int n = 0; n < iterations; n++)
23     {
24         long long timestamp_start = c.start();
25         glDrawArrays(GL_TRIANGLES, 0, numTriangles * 3);
26         glFlush();
27         SetMasterEvent();
28         glFinish();
29         long long timestamp_stop = c.stop();
30         writeToResults(timestamp_start, timestamp_stop, glGetError());
31         WaitForSlaveEvent();
32     }
33 }
```

Algorithmus 4.16 zeigt den Code des Masterprogramms. In den Zeilen 6–15 wird die Referenzmessung der Laufzeit des Draw-Befehls durchgeführt. Anschließend wird in Zeile 18 das Slavepro-

gramm gestartet und in Zeile 19 darauf gewartet, dass das Slaveprogramm die Bereitschaft signalisiert, fortzufahren. Die Messung für konkurrierenden Draw-Befehle erfolgt dann in den Zeilen 22–32.

Dabei werden in Zeile 24 der Zeitstempel unmittelbar vor Übermittlung des Draw-Befehls gespeichert, um bei der anschließenden Auswertung der Ergebnisse sicherstellen zu können, dass die Übermittlung der Draw-Befehle von Master- und Slaveprogramm zeitlich nicht so lange auseinanderlag, dass es sich nicht mehr um konkurrierende Draw-Befehle handelt (die Speicherung des korrespondierenden Zeitstempels des Slaveprogramms ist in Algorithmus 4.17 in Zeile 13 zu sehen).

Das Signal für das Slaveprogramm, seinen Draw-Befehl an den GL-Server zu übermitteln wird in Zeile 27 gesetzt, unmittelbar nach Ausführung von `glFlush`. Dieser Befehl stellt sicher, dass der Draw-Befehl des Masterprogramms vor dem des Slaveprogramms an den GL-Server übermittelt wird und auch vor diesem ausgeführt wird (vgl. Kapitel 2.3). In Zeile 31 wird darauf gewartet, dass das Slaveprogramm den Abschluss eines Draw-Befehls signalisiert, bevor die nächste Iteration der Laufzeitmessung begonnen wird.

Algorithmus 4.17 Untersuchung konkurrierender Draw-Befehle (Slaveprogramm).

```
1  void testCompetingDrawSlave(
2      unsigned int numTriangles, // Anzahl zu rendernder Dreiecke
3      unsigned int iterations)   // Anzahl Iterationen für die Laufzeitmessung
4  {
5      (...) // Referenzmessung
6
7      SetSlaveEvent(); // Bereitschaft signalisieren
8
9      // Messung mit konkurrierenden Draw-Befehlen durchführen
10     for (unsigned int n = 0; n < iterations; n++)
11     {
12         WaitForMasterEvent();
13         long long timestamp_start = c.start();
14         glDrawArrays(GL_TRIANGLES, 0, numTriangles * 3);
15         glFlush();
16         glFinish();
17         long long timestamp_stop = c.stop();
18         writeToResults(timestamp_start, timestamp_stop, glGetError());
19         SetSlaveEvent();
20     }
21 }
```

Algorithmus 4.17 zeigt den Code des Slaveprogramms. In Zeile 5 wird zunächst die gleiche Referenzmessung durchgeführt wie im Masterprogramm. Dies dient dazu, sicherstellen zu können, dass das Slaveprogramm vom OpenGL ES-System nicht anders behandelt wird als das Masterprogramm (vgl. Kapitel 3.3.2.1). Nach Abschluss der Referenzmessung wird in Zeile 7 die Bereitschaft signalisiert, fortzufahren. Die Messung mit konkurrierenden Draw-Befehlen erfolgt dann in den Zeilen 10–20. In Zeile 12 wartet das Slaveprogramm darauf, dass vom Masterprogramm signalisiert wird, dass der Draw-Befehl an den GL-Server übermittelt werden kann. Unmittelbar davor wird auch hier der aktuelle Zeitstempel gespeichert, um später sicherstellen zu können, dass tatsächlich die

Laufzeiten konkurrierender Draw-Befehle gemessen wurden (vgl. die Diskussion zu Zeile 24 des Masterprogramms). Nach Abschluss des Draw-Befehls wird in Zeile 19 die Bereitschaft signalisiert, die nächste Iteration der Laufzeitmessung durchzuführen.

Diese Programme wurden für unterschiedliche Mengen an zu rendernden Dreiecken durchgeführt. Dabei wurden die in Abschnitt 4.1.6 beschriebenen Minimalshader verwendet. In einem ersten Lauf wurden 128 Dreiecke gerendert, was auf allen drei Testsystemen trotz Minimalshader lange genug dauert, damit der Draw-Befehl des Slaveprogramms rechtzeitig an den GL-Server übermittelt werden kann (um sicherzustellen, dass die beiden Draw-Befehle tatsächlich um die Renderpipeline konkurrieren). In jedem weiteren Lauf wurde die Anzahl der zu rendernden Dreiecke verdoppelt bis zu einem Maximum von 262.144 Dreiecken (was selbst auf dem schnellsten System etwa 2,1 Sekunden dauert).

Die Simulationen wurden anschließend in einer zweiten Variante durchgeführt, bei der das Slaveprogramm nicht in einem eigenen Prozess, sondern in einem Thread des Masterprogramms ausgeführt wurde. Dies diente dem Zweck, auszuschließen, dass eine nebenläufige Ausführung von Draw-Befehlen nur deshalb nicht festgestellt werden kann, weil die konkurrierenden Draw-Befehle von unterschiedlichen Prozessen übermittelt wurden. Im folgenden Abschnitt werden die Ergebnisse der Simulationen zusammengefasst.

4.4.1.2 Ergebnisse

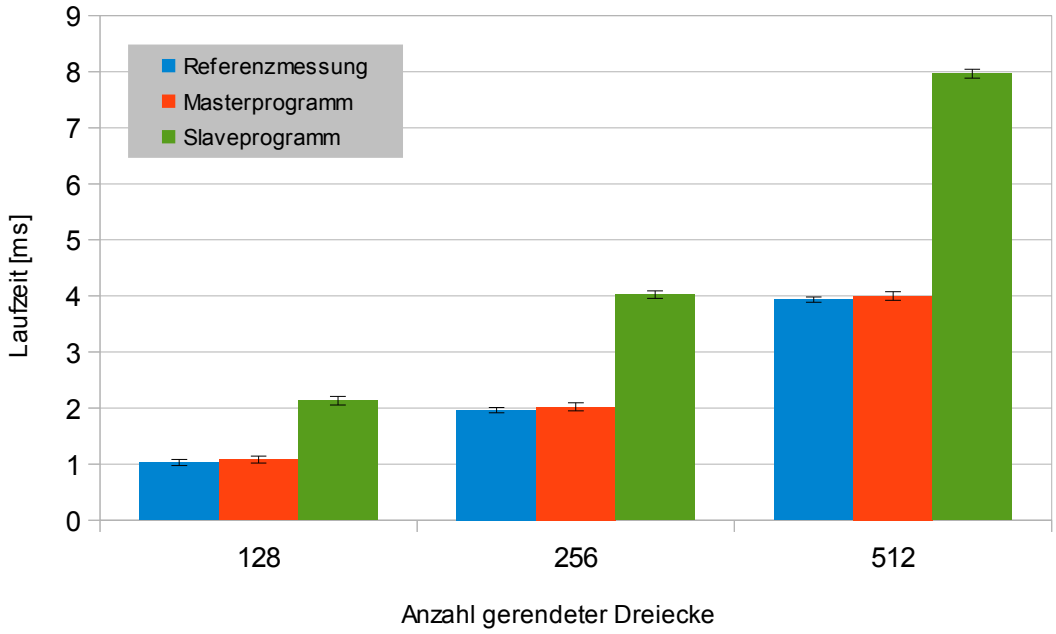

Abbildung 4.15: Laufzeiten konkurrierender Draw-Befehle

Abbildung 4.15 zeigt die auf dem „Nvidia Quadro 2000D“-System ermittelten Ergebnisse für das Rendering von 128 bis 512 Dreiecken. Auf der X-Achse ist die Anzahl der gerenderten Dreiecke

aufgetragen und auf der Y-Achse die Laufzeiten der Draw-Befehle. Die einzelnen Balken zeigen jeweils das Mittel der gemessenen Laufzeiten. Die darüber eingezeichneten Fehlerbalken zeigen die durchschnittliche Abweichung der Messwerte vom dargestellten Mittelwert. Dabei kann festgestellt werden, dass diese Abweichungen abnehmen, je mehr Dreiecke gerendert werden (bei 128 Dreiecken liegen sie noch bei etwa 5%, bei 262.144 Dreiecken unter 1%).

Die Simulationen zeigen auf allen drei Testsystemen in beiden Varianten (Slaveprogramm als eigener Prozess oder als Thread des Masterprogramms) das gleiche Ergebnis: Die Laufzeit der Draw-Befehle des Slaveprogramms ist etwa doppelt so groß wie die des Masterprogramms. Die Laufzeit der Draw-Befehle des Masterprogramms entspricht etwa der Laufzeit der Referenzmessung.[7]

Die Ausführung des zuerst übermittelten Draw-Befehls des Masterprogramms wird offenbar nicht durch den Draw-Befehl des Slaveprogramms unterbrochen; die Ausführung der Draw-Befehle des Slaveprogramms werden offensichtlich verzögert, bis die vorhergehenden Draw-Befehle des Masterprogramms vollständig abgearbeitet sind. Eine nebenläufige Ausführung der Draw-Befehle findet auf den drei Testsystemen nicht statt. Im nächsten Abschnitt wird die Abbrechbarkeit von Draw-Befehlen untersucht.

4.4.2 Abbrechbarkeit von Draw-Befehlen

Zur Untersuchung der Abbrechbarkeit von Draw-Befehlen werden – wie in Kapitel 3.3.2.2 beschrieben – zwei OpenGL ES-Programme ausgeführt. Das erste der beiden Programme erzeugt einen OpenGL ES-Kontext, legt ein Datenobjekt an, rendert ein Bild und wartet anschließend auf eine Benutzereingabe, um ein weiteres Bild zu rendern. Sobald das erste Programm auf die Eingabe wartet, wird das zweite Programm gestartet. Dieses übermittelt einen langlaufenden Draw-Befehl an den GL-Server (acht Millionen Dreiecke als Inputdaten – das Rendering würde auf allen drei Testsystemen länger als eine Minute dauern). Dadurch greift der in [MSDN 2009] beschriebene TDR-Mechanismus.

Das führt dazu, dass das OpenGL ES-Programm beendet wird, das den langlaufenden Draw-Befehl übermittelt. Anschließend wird auf dem andern OpenGL ES-Programm die Benutzereingabe durchgeführt, woraufhin das Programm versucht, ein weiteres Bild zu rendern. Dies schlägt auf allen drei Testsystemen fehl – der OpenGL ES-Kontext ist nicht mehr gültig und das angelegte Datenobjekt auch nach einer Kontext-Neuerzeugung nicht mehr verfügbar. Auf den Testsystemen wird also im Zuge des TDR-Mechanismus ein Hardware-Reset der GPU durchgeführt. Da dies nur der Fall ist, wenn zuvor ein Abbruch des laufenden Draw-Befehls im Zuge der in Kapitel 3.3.2.2 beschriebenen *preempt operation* fehlschlägt, muss davon ausgegangen werden, dass die untersuchten Systeme nicht in der Lage sind, einen laufenden Draw-Befehl ohne Hardware-Reset abzubrechen.

[7] Die in Abbildung 4.15 erkennbare Abweichung der Laufzeit des Draw-Befehls beim Masterprogramm vom Referenzwert entspricht den Kosten für einen Kontextwechsel auf dem Nvidia-System (vgl. Abschnitt 4.5). Ein solcher Kontextwechsel wird notwendig, weil das Master- und das Slaveprogramm im Wechsel Befehle an den GL-Server übermitteln.

4.4.3 Fazit Pipelinenutzung

Um Echtzeitgarantien für kritische Anwendungen erfüllen zu können, ist es notwendig, zumindest eine Obergrenze für die Laufzeit von Draw-Befehlen angeben zu können. Wie zu Beginn von Kapitel 3.3 dargelegt, ist es aber nicht möglich, die Laufzeit eines Draw-Befehls vorherzusagen, wenn der betreffende Befehl nicht bereits mindestens einmal mit gleichen Daten und Einstellungen ausgeführt worden ist.

Daher wird in Abschnitt 4.4.1 untersucht, ob konkurrierende Draw-Befehle nebenläufig ausgeführt werden können, um – bei bekannter Laufzeit von kritischen Draw-Befehlen – eine Obergrenze für deren Laufzeit angeben zu können, auch wenn ein potentiell langlaufender Befehl einer anderen Anwendung in der Renderpipeline ausgeführt wird. Auf den drei hier untersuchten Systemen konnte jedoch keine nebenläufige Ausführung von Draw-Befehlen festgestellt werden. Ein langlaufender Draw-Befehl einer anderen Anwendung könnte daher die Renderpipeline so lange blockieren, dass eine rechtzeitige Ausführung des nächsten Draw-Befehls einer kritischen Anwendung nicht mehr möglich ist.

Die in Abschnitt 4.4.2 durchgeführte Untersuchung zeigte außerdem, dass der Abbruch eines laufenden Draw-Befehls nicht möglich ist, ohne dass ein Hardware-Reset der GPU durchgeführt wird. Damit kritische Anwendungen nach einem solchen Reset weiterarbeiten können, müssen sie erst einen neuen OpenGL ES-Kontext erzeugen und alle benötigten Datenobjekte neu anlegen. Selbst wenn dies nicht zu lange dauern würde, um den nächsten Draw-Befehl der kritischen Anwendung rechtzeitig ausführen zu können, stellt ein solcher Abbruchvorgang auf den betrachteten Systemen höchstwahrscheinlich keine Möglichkeit dar, um zufriedenstellende Frameraten für kritische Anwendungen gewährleisten zu können – zumal allen diesen Systemen beobachtet werden konnte, dass der Hardware-Reset selbst mehrere Sekunden in Anspruch nimmt.

Aufgrund der streng sequentiellen Abarbeitung von Draw-Befehlen sowie der mangelnden Vorhersagbarkeit ihrer Laufzeit und ihrer mangelnden Abbrechbarkeit ist auf den drei untersuchten Systemen auf der Ebene von OpenGL ES keine Erfüllung von Echtzeitgarantien hinsichtlich der Pipelinenutzung möglich. Im Ausblick in Kapitel 5 wird ein möglicher Lösungsansatz für dieses Problem außerhalb von OpenGL ES skizziert.

4.5 Kontextwechsel

4.5.1 Vorgehen

Um die Kosten eines Kontextwechsels zu bestimmen, werden – wie in Kapitel 3.4.2 beschrieben – zwei OpenGL ES-Programme ausgeführt, die als Master- und Slaveprogramm bezeichnet werden. Diese beiden Programme übermitteln entweder einen Draw-Befehl an den GL-Server oder erzeugen ein neues Datenobjekt, wobei immer zuerst das Slaveprogramm seine Aktion durchführt und das Masterprogramm seine, sobald die Aktion des Slaveprogramms vollständig abgeschlossen worden ist. Dadurch wird auf Seiten des Masterprogramms ein Kontextwechsel erzwungen. Durch Vergleich der beim Masterprogramm gemessenen Laufzeiten mit zuvor ermittelten Referenzwerten können die Kosten eines Kontextwechsels ermittelt werden.

Algorithmus 4.18 zeigt das Masterprogramm. In den Zeilen 8–21 wird die Referenzmessung ohne Kontextwechsel durchgeführt. Falls dabei in Zeile 16 ein Datenobjekt erzeugt wird, wird es in Zeile 20 wieder freigegeben. In Zeile 24 wird das Slaveprogramm gestartet und anschließend in Zeile 25 darauf gewartet, dass es die Bereitschaft signalisiert, fortzufahren.

Die Laufzeitmessung mit erzwungenen Kontextwechseln wird in der Schleife von Zeile 28 bis 33 durchgeführt. Dabei wird dem Slaveprogramm in Zeile 30 signalisiert, seine Aktion durchzuführen und anschließend in Zeile 31 darauf gewartet, dass das Slaveprogramm den Abschluss der Aktion signalisiert. Anschließend wird die Zeit für die Ausführung der Aktion des Masterprogramms gemessen und das Messergebnis gespeichert. Dies erfolgt auf die gleiche Weise wie bei der Referenzmessung (in den Zeilen 8–21).

Algorithmus 4.18 Bestimmung der Kosten von Kontextwechseln (Masterprogramm).

```
1  void ctxSwitchMaster(
2      unsigned int numVertices,      // Anzahl Vertices
3      unsigned int iterations,       // Anzahl der Einzelmessungen
4      GLbyte *data,                  // zu rendernde / übertragende Daten
5      bool drawCommand)              // Übermittlung eines Draw-Befehls?
6  {
7      // Referenzmessung
8      HPCClock c;
9      long long results[iterations];
10     for (unsigned int n = 0; n < iterations; n++)
11     {
12         c.start();
13         if (drawCommand)
14           glDrawArrays(GL_POINTS, 0, numVertices);
15         else
16           (...) // Datenobjekt erzeugen
17         glFlush();
18         glFinish();
19         writeToResults(c.stop(), glGetError());
20         (...) // Datenobjekt freigeben, falls zuvor erzeugt
21     }
22
23     // Slaveprogramm starten und warten, bis es bereit ist
24     spawnSlave();
25     WaitForSlaveEvent();
26
27     // Messung mit erzwungenen Kontextwechseln durchführen
28     for (unsigned int n = 0; n < iterations; n++)
29     {
30         SetMasterEvent();
31         WaitForSlaveEvent();
32         (...) // analog Referenzmessung
33     }
34 }
```

Algorithmus 4.19 zeigt das Slaveprogramm. In Zeile 6 wird die Bereitschaft signalisiert, fortzufahren. Die Durchführung der jeweiligen Aktion des Slaveprogramms erfolgt in der Schleife von Zeile

9 bis 19. Dazu wird in Zeile 11 auf das Signal des Masterprogramms gewartet. Nach erfolgter Ausführung des übermittelten Befehls wird dies in Zeile 18 dem Masterprogramm signalisiert. Falls in Zeile 15 ein Datenobjekt erzeugt wird, wird es in Zeile 17 wieder freigegeben.

Algorithmus 4.19 Bestimmung der Kosten von Kontextwechseln (Slaveprogramm).

```
1  void ctxSwitchSlave(
2      unsigned int numVertices,      // Anzahl Vertices
3      GLbyte *data,                  // zu rendernde / übertragende Daten
4      bool drawCommand)              // Übermittlung eines Draw-Befehls?
5  {
6      SetSlaveEvent(); // Bereitschaft signalisieren
7
8      // Befehle nach Aufforderung durch Masterprogramm übermitteln
9      for (unsigned int n = 0; n < iterations; n++)
10     {
11         WaitForMasterEvent();
12         if (drawCommand)
13           glDrawArrays(GL_POINTS, 0, numVertices);
14         else
15           (...) // Datenobjekt erzeugen
16         glFinish();
17         (...) // Datenobjekt freigeben, falls zuvor erzeugt
18         SetSlaveEvent();
19     }
20 }
```

4.5.2 Ergebnisse

Aktion von Master	Aktion von Slave	„Nvidia Quadro 2000D“	„ATI FirePro V4800“	„ATI FirePro V5900“
Draw-Befehl	Draw-Befehl	61	14	5
Draw-Befehl	Neues DO[8]	"	"	"
Neues DO	Draw-Befehl	"	"	"
Neues DO	Neues DO	234	184	126

Tabelle 4.3: Mittlere Laufzeitunterschiede bei erzwungenen Kontextwechseln (in µs)

Tabelle 4.3 zeigt die ermittelten Kosten für Kontextwechsel auf den drei Testsystemen. Nach Erzeugung eines neuen Datenobjekts erhöhen sich die Kosten für einen Kontextwechsel. Diese erhöhten Kosten treten auf allen drei Testsystemen auf, wenn vom Masterprogramm anschließend ein neues Datenobjekt erzeugt wird.

[8] DO steht für Datenobjekt.

4.5.3 Fazit Kontextwechsel

Von den vier betrachteten Problemfeldern stellen die Kosten von Kontextwechseln die geringste Schwierigkeit hinsichtlich der Erfüllung von Echtzeitgarantien dar. Sie müssen bei der Angabe von Obergrenzen für die Laufzeit von OpenGL ES-Befehlen berücksichtigt werden, auch wenn ihr Einflusspotential auf den betrachteten Testsystemen sehr gering ist (im Mikrosekundenbereich) – auf anderen als den untersuchten Systemen kann der Einfluss von Kontextwechseln deutlich schwerer wiegen (vgl. Abschnitt 3.4). Erfreulicherweise lassen sich diese Kosten mit Bordmitteln von OpenGL ES ermitteln. Durch die Kosten von Kontextwechseln wird die Erfüllung von Echtzeitgarantien also nicht signifikant beeinträchtigt.

5 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Ergebnisse wiedergegeben und die Probleme hinsichtlich der Erfüllung von Echtzeitgarantien für kritische OpenGL ES-Anwendungen, die auf Ebene von OpenGL ES nicht gelöst werden können, zusammengefasst. Anschließend werden mögliche Lösungsansätze für diese Probleme skizziert.

5.1 Wenig Probleme bei Kontextwechsel und Datenübertragung

Von den betrachteten Problembereichen bringen der Kontextwechsel und die Datenübertragung die geringsten Schwierigkeiten mit sich, wenn es darum geht, Echtzeitgarantien für die Ausführung kritischer OpenGL ES-Programme zu gewährleisten. Sowohl für Kontextwechsel als auch für die Datenübertragung lassen sich auf der Ebene von OpenGL ES Kennzahlen zum Verhalten der verwendeten Systeme ermitteln, mit deren Hilfe sich die Einflüsse dieser beiden Problembereiche beherrschen lassen. Lediglich die Datenübertragung im Rahmen ungepufferter Draw-Befehle bringt zusätzlich die gleichen Schwierigkeiten mit sich wie die Pipelinenutzung, da ein ungepufferter Draw-Befehl nicht nur Daten in das OpenGL ES-System überträgt, sondern auch die Datenverarbeitung durch die Renderpipeline anstößt.

5.2 Schwer beherrschbare Risiken bei der Pipelinenutzung

Das größte Problem hinsichtlich der Pipelinenutzung stellt die mangelnde Vorhersagbarkeit der Laufzeit von Draw-Befehlen dar, insbesondere auf solchen Systemen, bei denen – wie bei den drei untersuchten Testsystemen – Draw-Befehle weder nebenläufig ausgeführt noch ohne einen Hardware-Reset abgebrochen werden können. Dieses Problem hat zwei Ursachen:

- Wenn keine Einschränkungen hinsichtlich der ausgeführten Shader definiert werden, kann deren Laufzeit nicht allgemein vorhergesagt werden, ohne das Halteproblem für diese Programme zu lösen – sogar Endlosschleifen sind bei solchen Shadern möglich (siehe [Simpson und Kessenich 2009], Seite 57).
- Die Anzahl der ausgeführten Instanzen von Fragmentshadern lässt sich vor der Ausführung eines Draw-Befehls nicht vorhersagen, wenn der entsprechende Draw-Befehl nicht schon einmal mit gleichem Input und gleichen Einstellungen ausgeführt worden ist – diese Anzahl wird erst während dessen Ausführung festgelegt (nach Abschluss der Rasterisierung, siehe Kapitel 2.6).

Dies führt letztendlich zu der Situation, dass prinzipiell die Ausführung jedes Draw-Befehls die Renderpipeline so lange blockieren könnte, dass die rechtzeitige Ausführung des nächsten Draw-Befehls einer kritischen Anwendung verhindert wird.

5.3 Risiken wegen mangelnder Speicherbelegungsinformation

Um garantieren zu können, dass aufgrund der Speicherbelegung die rechtzeitige Ausführung eines Draw-Befehls einer kritischen Anwendung nicht verhindert wird, müssen mindestens die folgenden beiden Bedingungen erfüllt sein:

- Es kann eine Obergrenze für die Menge an GPU-Speicher garantiert werden, die durch die Erzeugung eines Datenobjekts belegt wird (ggf. durch Berücksichtigung der Größe des Datenobjekts und der Kennzahlen zu Speichergranularität und Speicherblockgröße).
- Es stehen ausreichend Informationen hinsichtlich des aktuellen Layouts des GPU-Speichers zur Verfügung, so dass jederzeit bestimmt werden kann, wieviel GPU-Speicher durch ein einzelnes Datenobjekt noch belegt werden kann.

Die erste Bedingung kann mit Hilfe der in Kapitel 3.1 beschriebenen Untersuchungen erfüllt werden, wenn es zumindest im Rahmen der Untersuchungen möglich ist, die Menge des aktuell verfügbaren GPU-Speichers abzufragen. Letztendlich ist es aber unerheblich, wie die Informationen zur Erfüllung dieser Bedingung ermittelt werden. Die zweite Bedingung stellt aber eine Fähigkeit dar, über die ein OpenGL ES-System zur Laufzeit verfügen muss. Dies ist zum Beispiel bei dem in dieser Arbeit untersuchten „Nvidia Quadro 2000D“-System nicht der Fall (siehe Kapitel 4.2.6).

Es ist sehr wahrscheinlich, dass auch auf anderen Systemen diese Bedingungen nicht erfüllt werden können, zumal OpenGL ES 2.0 es weder ermöglicht, den aktuellen freien Speicherplatz noch das aktuelle Layout des GPU-Speichers abzufragen. Sofern durch ein System keine Erweiterung unterstützt wird, durch die die benötigten Informationen zur Speicherbelegung abgefragt werden können, kann auf Ebene von OpenGL ES 2.0 auf solchen Systemen nicht garantiert werden, dass die rechtzeitige Ausführung kritischer Draw-Befehle durch die Speicherbelegung nicht verhindert wird.

5.4 Mögliche Lösungsansätze

Wie in den vorhergehenden Abschnitten erläutert, stehen der Gewährleistung von Garantien für die rechtzeitige Ausführung von Draw-Befehlen noch die folgenden drei Probleme im Wege:

- Die mangelnde Abrufbarkeit von Information zur aktuellen Speicherbelegung.
- Die mangelnde Vorhersagbarkeit der Anzahl der bei der Abarbeitung eines Draw-Befehls ausgeführten Shader-Instanzen.
- Die mangelnde Garantierbarkeit einer Obergrenze für die Laufzeit von Shader-Instanzen.

In den nächsten Abschnitten werden mögliche Lösungsansätze für diese drei Probleme skizziert.

5.4.1 Nutzung der Treiber-Informationen zum Speicherlayout

[Dwarakinath 2008] berichtet über die ATI r200-GPU, dass auf Ebene ihres GPU-Treibers unter Linux die Informationen zum aktuellen Layout des GPU-Speichers zur Verfügung stehen, d. h. auf dieser Ebene ist bekannt, welche Speicherbereiche innerhalb des GPU-Speichers belegt sind und

welche für die Speicherung weiterer Daten zur Verfügung stehen. Die Virtualisierungsansätze von [Dwarakinath 2008] und [Kato u. a. 2011] setzen auf Ebene der GPU-Treiber an. Eine Möglichkeit zur Erfüllung der in Abschnitt 5.3 genannten Bedingungen könnte darin bestehen, ebenfalls auf dieser Ebene anzusetzen und die Informationen zum Layout des GPU-Speichers zu nutzen. Wenn bekannt ist, welche Speicherbereiche belegt und welche frei sind, kann davon abgeleitet werden, wieviel Speicherplatz in der GPU insgesamt zur Verfügung steht und wieviel durch ein einzelnes Datenobjekt noch belegt werden kann.

5.4.2 Einschränkung von Shadern

Eine Obergrenze für die Laufzeit von Shader-Instanzen könnte zum Beispiel dadurch garantiert werden, dass Beschränkungen hinsichtlich des Kontrollflusses von Shadern eingeführt werden, wie sie in [Simpson und Kessenich 2009] auf den Seiten 108–109 vorgeschlagen werden. Durch diese Einschränkungen könnten Endlosschleifen ausgeschlossen werden und die genaue Anzahl an Schleifeniterationen könnte durch Auswertung des Shader-Quellcodes sicher bestimmt werden.

Somit könnte anhand des Quellcodes eines Shaders die maximale Anzahl an Anweisungen berechnet werden, die durch ihn ausgeführt werden. Nach Bestimmung der maximalen Laufzeitkosten für die verschiedenen Anweisungsarten könnte eine Obergrenze für die Laufzeit eines beliebigen Shaders auf Grundlage seines Quellcodes berechnet werden. Die Verwendung vorkompilierter Shader muss bei diesem Ansatz ausgeschlossen werden.

5.4.3 Trennung von Vertex- und Fragment-Processing

Dem folgenden Ansatz zur Bestimmung der Anzahl ausgeführter Shader-Instanzen liegt die Annahme zu Grunde, dass eine Obergrenze für die Laufzeit von Shader-Instanzen bestimmt werden kann (zum Beispiel wie im vorherigen Abschnitt beschrieben).

Die Anzahl der im Zuge eines Draw-Befehls ausgeführten Instanzen von Vertexshadern kann anhand der übergebenen Vertexdaten vorhergesagt werden (es wird pro definiertem Vertex genau eine Instanz eines Vertexshaders ausgeführt). Die genaue Anzahl der auszuführenden Instanzen von Fragmentshadern steht erst nach Abschluss der Rasterisierung fest (siehe Kapitel 2.6).

Es kann aber eine – reichlich unpraktische – Obergrenze für die Anzahl der ausgeführten Instanzen von Fragmentshadern garantiert werden: Es werden pro gerendertem Primitiv (Dreieck, Linie oder Punkt, siehe Kapitel 2.6.2.2) höchstens so viele Fragmentshader-Instanzen ausgeführt, wie es Pixel im Framebuffer gibt.[1] Diese Obergrenze ist unrealistisch hoch – sie wird nur erreicht, wenn jedes gerenderte Primitiv, den kompletten sichtbaren Bereich überdeckt, also jeden Pixel im Framebuffer.

Der Ansatz zur Bestimmung der Anzahl ausgeführter Fragmentshader-Instanzen basiert nun darauf, dass in einem ersten Schritt das Fragmentshader-Programm der aufrufenden Anwendung durch einen Minimalshader ersetzt wird, für den die Laufzeit bekannt ist. Mit diesem minimalen

[1] Falls das verwendete System Multisampling unterstützt, muss dieser Wert noch mit der Anzahl an Samples pro Pixel multipliziert werden (siehe Kapitel 2.6.2.3).

Fragmentshader und dem ursprünglichen Vertexshader wird der Draw-Befehl nun ausgeführt. Für die Laufzeit dieses Draw-Befehls kann eine Obergrenze garantiert werden, da

- eine Obergrenze für die Laufzeiten der beiden Shaderarten bestimmt werden kann und
- die Anzahl der ausgeführten Vertexshader-Instanzen bekannt ist und
- eine Obergrenze für die Anzahl der ausgeführten Fragmentshader-Instanzen bekannt ist.

Anschließend wird der minimale Fragmentshader durch eine modifizierte Variante ersetzt, die eine geringfügig längere (und ebenfalls vorab bekannte) Laufzeit hat. Damit wird der Draw-Befehl erneut ausgeführt. Eine Obergrenze für die Laufzeit dieses zweiten Draw-Befehls kann auf die gleiche Weise garantiert werden wie beim ersten Draw-Befehl.

Da die Laufzeiten der beiden Fragmentshader vorab bekannt sind, kann durch einen Vergleich der Laufzeiten der beiden Draw-Befehle auf die Anzahl der ausgeführten Fragmentshader-Instanzen geschlossen werden. Mit dieser zusätzlichen Information kann nun eine sinnvollere Obergrenze für die Laufzeit des Draw-Befehls mit dem ursprünglichen Fragmentshader berechnet werden.

5.5 Fazit

In dieser Arbeit wurden diejenigen der 142 von OpenGL ES 2.0 definierten Befehle bestimmt, deren Ausführung einen negativen Einfluss auf die Erfüllung von Echtzeitgarantien für kritische Anwendungen haben könnte, und anschließend deren Laufzeitverhalten und Ressourcenverbrauch analysiert. Dazu wurden Metriken und Untersuchungsmethoden entwickelt, auf deren Grundlage der Ressourcenverbrauch sowie die Laufzeit von OpenGL ES-Befehlen prognostiziert und die dafür notwendigen systemspezifischen Kennzahlen ermittelt werden können.

Diese Untersuchungen wurden auf drei realen OpenGL ES-Systemen durchgeführt, wobei sich zeigte, dass insbesondere das Speicherbelegungsverhalten und die Nutzung der Renderpipeline mit Problemen verbunden sind, die der Erfüllung von Echtzeitgarantien im Wege stehen und nicht auf der Ebene von OpenGL ES gelöst werden können. Für diese Probleme wurden schließlich in diesem Kapitel mögliche Lösungsansätze skizziert.

Es kann an dieser Stelle jedoch keine Aussage dazu gemacht werden, inwieweit die hier skizzierten Lösungsansätze tatsächlich praktikabel sind. Dies zu ergründen, bleibt künftigen Arbeiten vorbehalten. Sofern sich dabei herausstellen sollte, dass sie praktikabel sind, oder sofern andere Lösungen für die zu Beginn von Abschnitt 5.4 aufgeführten Probleme gefunden werden können, dann ist davon auszugehen, dass eine Zwischenschicht, wie sie in Kapitel 1.4 beschrieben wurde, tatsächlich Echtzeitgarantien hinsichtlich der Ausführung von OpenGL ES-Befehlen für sicherheitskritische Anwendungen erfüllen kann.

6 Anhang

6.1 Befehle von OpenGL ES 2.0

6.1.1 Erzeugung von Datenobjekten

OpenGL ES-Befehl	Kurzbeschreibung
`glBufferData`	Dieser Befehl erzeugt ein neues Vertexbuffer-Objekt. Er reserviert den dafür notwendigen Speicherplatz (entsprechend der in den Parametern übergebenen Objektgröße) und füllt das neue Objekt optional mit Daten aus dem Hauptspeicher.
`glRenderbufferStorage`	Dieser Befehl erzeugt ein neues Renderbuffer-Objekt und reserviert den dafür notwendigen Speicherplatz (entsprechend der in den Parametern übergebenen Objektgröße).
`glTexImage2D`	Dieser Befehl erzeugt ein neues Texturobjekt. Er reserviert den dafür notwendigen Speicherplatz (entsprechend der in den Parametern übergebenen Texturgröße) und füllt das neue Objekt optional mit Daten aus dem Hauptspeicher.
`glCompressedTexImage2D`	Analog `glTexImage2D` für komprimierte Texturen.
`glCopyTexImage2D`	Erzeugt ein neues Texturobjekt und füllt es mit Daten aus einem anderen Datenobjekt.
`glGenerateMipmap`	Erzeugt eine vollständige Mipmap-Chain für eine gegebene Textur. Dadurch werden bis zu $\log_2(\max\{w, h\})$ neue Texturobjekte erzeugt, wobei w und h die Breite und Höhe der Ausgangstextur in Texel[1] sind.
`glBindFramebuffer`	Siehe Abschnitt 6.1.5.

Tabelle 6.1: Befehle zur Erzeugung von Datenobjekten

6.1.2 Freigabe von Datenobjekten

OpenGL ES-Befehl	Kurzbeschreibung
`glDeleteBuffers`	Gibt Vertexbuffer-Objekte frei.
`glDeleteRenderbuffers`	Gibt Renderbuffer-Objekte frei.
`glDeleteTextures`	Gibt Texturobjekte frei.
`glDeleteFramebuffers`	Gibt Framebuffer-Objekte frei.

Tabelle 6.2: Befehle zur Freigabe von Datenobjekten

[1] Der Begriff *Texel* bezeichnet einen einzelnen Datenwert einer Textur.

6.1.3 Datenübertragungsbefehle

OpenGL ES-Befehl	Kurzbeschreibung
`glBufferData`	Siehe 6.1.1.
`glBufferSubData`	Überträgt Vertexdaten vom Hauptspeicher in ein Vertexbuffer-Objekt.
`glTexImage2D`	Siehe 6.1.1.
`glTexSubImage2D`	Kopiert Texturdaten vom Hauptspeicher in ein Texturobjekt.
`glCompressedTexImage2D`	Siehe 6.1.1.
`glCompressedTexSubImage2D`	Analog `glTexSubImage2D` für komprimierte Texturdaten.
`glCopyTexImage2D`	Siehe 6.1.1.
`glCopyTexSubImage2D`	Analog `glCopyTexImage2D` mit dem Unterschied, dass nur ein Teil der Zieltextur mit neuen Daten überschrieben wird.
`glReadPixels`	Kopiert einen rechteckigen Bereich des aktuellen Colorbuffers (ein Teil des aktuellen Framebuffers) in den Hauptspeicher.
`glUniform*`	Insgesamt 19 Funktionen, die sich nur hinsichtlich der Parameterübergabe unterscheiden. Sie dienen dem Setzen des Wertes von konstanten Shadervariablen, die entweder Vertex- oder Fragmentshadern zur Verfügung stehen.
`glVertexAttrib*`	Insgesamt acht Funktionen, die sich nur hinsichtlich der Parameterübergabe unterscheiden. Ähnlich den Uniform-Befehlen dienen sie dem Setzen von konstanten Shadervariablen, allerdings nur für Vertexshader.

Tabelle 6.3: Datenübertragungsbefehle

6.1.4 Vertexdatenverwaltung

OpenGL ES-Befehl	Kurzbeschreibung
`glVertexAttribPointer`	Weist dem aktuellen OpenGL ES-Kontext ein Array mit Vertexdaten zu.
`glEnableVertexAttribArray`	Setzt fest, dass ein zugewiesenes Array von Vertexdaten als Input für die Vertexshader genutzt werden soll (statt konstantem Input).
`glDisableVertexAttribArray`	Setzt fest, dass konstanter Input für Vertexshader verwendet werden soll (statt einem zuvor zugewiesenen Array von Vertexdaten).

Tabelle 6.4: Vertexdatenverwaltung

6.1.5 Binding von Datenobjekten

OpenGL ES-Befehl	Kurzbeschreibung
`glBindBuffer`	Erklärt ein Vertexbuffer-Objekt zum aktuellen Vertexbuffer-Objekt, das von allen nachfolgenden Vertexbuffer-Befehlen genutzt wird, oder hebt diese Erklärung auf.
`glBindRenderbuffer`	Erklärt ein Renderbuffer-Objekt zum aktuellen Renderbuffer-Objekt, das von allen nachfolgenden Renderbuffer-Befehlen genutzt wird, oder hebt diese Erklärung auf.
`glBindTexture`	Erklärt ein Texturobjekt zum aktuellen Texturobjekt (für die zuvor mit `glActiveTexture` festgelegte Textureinheit), das von allen nachfolgenden Texturbefehlen genutzt wird, oder hebt diese Erklärung auf.
`glActiveTexture`	Setzt die Textureinheit fest, auf die sich nachfolgende Aufrufe von `glBindTexture` beziehen.
`glBindFramebuffer`	Erklärt ein Framebuffer-Objekt zum aktuellen Framebuffer, der von allen nachfolgenden Framebuffer-Befehlen genutzt wird, oder hebt diese Erklärung auf. Sofern ein noch nicht existierendes Framebuffer-Objekt gebunden wird, wird an dieser Stelle ein neues erzeugt.

Tabelle 6.5: Befehle zum Binding von Datenobjekten

6.1.6 Naming von Datenobjekten

OpenGL ES-Befehl	Kurzbeschreibung
`glGenBuffers`	Liefert ungenutzte Namen (IDs) für Vertexbuffer-Objekte.
`glGenRenderbuffers`	Liefert ungenutzte Namen (IDs) für Renderbuffer-Objekte.
`glGenTextures`	Liefert ungenutzte Namen (IDs) für Texturobjekte.
`glGenFramebuffers`	Liefert ungenutzte Namen (IDs) für Framebuffer-Objekte.

Tabelle 6.6: Naming von Datenobjekten

6.1.7 Zusammensetzung von Framebuffern

OpenGL ES-Befehl	Kurzbeschreibung
`glFramebufferRenderbuffer`	Verbindet ein Renderbuffer-Objekt mit dem aktuellen Framebuffer-Objekt oder hebt diese Verbindung auf.
`glFramebufferTexture2D`	Verbindet ein Texturobjekt mit dem aktuellen Framebuffer-Objekt oder hebt diese Verbindung auf.

Tabelle 6.7: Zusammensetzung von Framebuffern

6.1.8 Draw-Befehle

OpenGL ES-Befehl	Kurzbeschreibung
`glDrawArrays`	Dieser Befehl überträgt die zuvor durch `glVertexAttribPointer` zugewiesenen Vertexdaten an das GL-System (falls diese nicht zuvor gepuffert[2] wurden) und stößt deren Verarbeitung durch die Renderpipeline an.
`glDrawElements`	Analog `glDrawArrays` mit dem Unterschied, dass zusätzlich zu den Vertexdaten auch Indexdaten an das GL-System übertragen werden (falls diese nicht zuvor gepuffert[2] wurden).

Tabelle 6.8: Draw-Befehle

6.1.9 Clearing-Befehle

OpenGL ES-Befehl	Kurzbeschreibung
`glClear`	Setzt alle Pixel im aktuellen Color-, Depth- und/oder Stencilbuffer auf den jeweiligen Clearwert.
`glClearColor`	Setzt den Clearwert für Colorbuffer fest.
`glClearDepthf`	Setzt den Clearwert für Depthbuffer fest.
`glClearStencil`	Setzt den Clearwert für Stencilbuffer fest.

Tabelle 6.9: Clearing-Befehle

6.1.10 Zustandsabfragen

OpenGL ES-Befehl	Kurzbeschreibung
`glCheckFramebufferStatus`	Dient dazu, abzufragen, ob das aktuelle Framebuffer-Objekt als Ziel für Renderoperationen oder Quelle für `glReadPixels` dienen kann oder nicht (d. h. ob alle notwendigen Datenobjekte zugewiesen wurden und diese zusammenpassen).
`glGet*`	Insgesamt 26 Funktionen, die sich nur hinsichtlich der zu übergebenden Parameter unterscheiden. Mit ihnen können aktuelle Zustände oder sonstige Werte abgefragt werden.
`glIs*`	Insgesamt sechs Varianten (für die verschiedenen Arten von Daten- und Programmobjekten), mit denen für einen gegebenen Objektnamen die Art des Objekts abgefragt werden kann.

Tabelle 6.10: Zustandsabfragen

[2] Siehe auch Kapitel 2.6.3 für eine Diskussion der Unterschiede von gepuffertem und ungepuffertem Rendering.

6.1.11 Programmverwaltung

OpenGL ES-Befehl	Kurzbeschreibung
`glCreateShader`	Legt ein leeres Shaderobjekt an.
`glDeleteShader`	Gibt ein Shaderobjekt frei.
`glShaderSource`	Weist einem Shaderobjekt Sourcecode zu.
`glCompileShader`	Kompiliert den zugewiesenen Sourcecode eines Shaderobjekts.
`glCreateProgram`	Legt ein leeres Programmobjekt an.
`glLinkProgram`	Erzeugt aus einem Vertex- und einem Fragmentshader ein ausführbares Programmobjekt, das für nachfolgende Renderoperationen genutzt werden kann.
`glUseProgram`	Installiert ein Programmobjekt im aktuellen Kontext, sofern es zuvor erfolgreich durch `glLinkProgram` erzeugt worden ist. Dieses Programmobjekt wird dann für nachfolgende Renderoperationen verwendet.
`glDeleteProgram`	Gibt ein Programmobjekt frei.
`glAttachShader`	Weist einem Programmobjekt ein Shaderobjekt zu.
`glDetachShader`	Hebt die Zuweisung eines Shaderobjekts zu einem Programmobjekt auf.
`glReleaseShaderCompiler`	Gibt dem GL-Server den Hinweis, dass er eventuell genutzte Ressourcen für den Shadercompiler freigeben kann.
`glShaderBinary`	Lädt vorkompilierten Shadercode in ein Shaderobjekt.

Tabelle 6.11: Befehle zur Programmverwaltung

6.1.12 Befehle zur Änderung von Renderpipeline-Einstellungen

OpenGL ES-Befehl	Kurzbeschreibung
`glTexParameter*`	Insgesamt vier Funktionen, die sich nur hinsichtlich der zu übergebenden Parameter unterscheiden. Sie dienen dazu, die Filter- und Wrappingeinstellungen für Texturen festzulegen.
`glDepthMask`	Setzt fest, ob der aktuelle Depthbuffer schreibgeschützt ist oder nicht.
`glStencilFunc`	Spezifiziert die Bedingung, anhand derer im Stenciltest überprüft wird, ob ein Pixel im Framebuffer durch ein Fragment tatsächlich verändern wird oder nicht.
`glStencilFuncSeparate`	Wie `glStencilFunc` mit dem Unterschied, dass die spezifizierte Bedingung nur für Fragmente mit bestimmter Orientierung gültig ist.
`glStencilOp`	Legt die Operation fest, die im Stencilbuffer abhängig vom Ausgang des Stenciltests durchgeführt wird.
`glStencilOpSeparate`	Wie `glStencilOp` mit dem Unterschied, dass die festgelegte Operation nur für Fragmente mit bestimmter Orientierung durchgeführt wird.

`glDepthFunc`	Ändert die Vergleichsoperation (größer, kleiner, ungleich etc.) für den Depthtest.
`glBlendFunc`	Ändert die Faktoren, mit denen Quell- und Zielfarbwert bei aktiviertem Blending multipliziert werden, bevor aus den beiden Produkten der resultierende Farbwert im Framebuffer berechnet wird.
`glBlendFuncSeparate`	Wie `glBlendFunc` mit dem Unterschied, dass jeweils unterschiedliche Faktoren für die RGB-Komponenten und die Alpha-Komponente angegeben werden können.
`glBlendColor`	Setzt die Werte des konstanten Blendingwerts, der bei `glBlendFunc` und `glBlendFuncSeparate` optional verwendet werden kann.
`glBlendEquation`	Legt den Operator fest, mit dem die beim Blending berechneten Quell- und Zielfaktoren verknüpft werden, um den resultierenden Farbwert im Framebuffer zu berechnen.
`glBlendEquationSeparate`	Wie `glBlendEquation` mit dem Unterschied, dass unterschiedliche Operatoren für die RGB-Komponenten und die Alpha-Komponente gewählt werden können.
`glSampleCoverage`	Legt den Wert fest, der verwendet wird, um eine zusätzliche Bitmaske zu erzeugen, die mit der im Multisampling erzeuten verundet wird, um das Antialiasing zu beeinflussen.
`glLineWidth`	Setzt die Liniendicke fest.
`glFrontFace`	Legt fest, ob Dreiecksnormalen im oder gegen den Uhrzeigersinn bestimmt werden.
`glPolygonOffset`	Legt die Werte für den automatischen Tiefenversatz von Polygonen fest (z.B. um koplanare Polygone ohne Z-Fighting-Artefakte zu rendern).
`glViewport`	Ändert die Viewporteinstellungen.
`glDepthRangef`	Ändert die Einstellungen zur Umrechnung von Z-Werten in Gerätekoordinaten zu Z-Werten in Fensterkoordinaten.
`glColorMask`	Setzt fest, welche Komponenten (RGBA) im aktuellen Colorbuffer überschrieben werden dürfen und welche nicht.
`glStencilMask`	Setzt fest, welche Bits eines Pixels im aktuellen Stencilbuffer schreibgeschützt sind und welche nicht.
`glStencilMaskSparate`	Analog `glStencilMask` mit dem Unterschied, dass verschiedene Bitmasken für Primitives angegeben werden können, abhängig von ihrer Orientierung.
`glCullFace`	Legt die Orientierung von Dreiecken fest, die im Zuge von Culling eliminiert werden sollen.
`glScissor`	Spezifiziert einen rechteckigen Bereich im aktuellen Framebuffer an dem unabhängig von den anderen Clippingoperation geclippt wird (der Bereich außerhalb des Rechtecks ist schreibgeschützt).

Tabelle 6.12: Befehle zur Änderung von Renderpipeline-Einstellungen

6.1.13 Sonstige OpenGL ES-Befehle

OpenGL ES-Befehl	Kurzbeschreibung
`glFlush`	Übermittelt alle vom aufrufenden Programm bisher aufgerufenen OpenGL ES-Befehle an den GL-Server, die noch nicht übermittelt wurden.
`glFinish`	Diese Funktion kehrt zurück, sobald alle bisher aufgerufenen OpenGL ES-Befehle vom GL-Server vollständig abgearbeitet worden sind.
`glValidateProgram`	Führt eine Reihe von Tests aus, um festzustellen, ob das aktuelle Programmobjekt mit dem aktuellen Zustand des Renderkontexts erfolgreich für Renderoperationen ausgeführt werden kann und liefert (implementierungsabhängig) zusätzliche Informationen über vorhandene Performanceprobleme, etc. – Nur für Debugging-Zwecke während der Entwicklung gedacht (siehe [Munshi u. a. 2008], Seite 66).
`glClear`	Setzt alle Pixel im aktuellen Color-, Depth- und/oder Stencilbuffer auf den von `glClearColor` festgesetzten, aktuellen Clearwert.
`glHint`	Erlaubt es, der Implementierung Hinweise zu geben, ob bei der Erzeugung einer Mipmap-Chain höhere Qualität oder höhere Performance angestrebt werden soll. Ob dieser Hinweis überhaupt einen Einfluss hat, ist allerdings implementierungsabhängig.
`glPixelStorei`	Bestimmt das Byte-Alignment von Pixelreihen in Texturen, die vom Hauptspeicher ein- oder in den Hauptspeicher ausgelesen werden. Dieser Wert beeinflusst nicht die Menge der übertragenen Daten und wie diese im GPU-Speicher abgelegt werden (siehe [Munshi u. a. 2008], Seite 187).
`glBindAttribLocation`	Den möglichen Inputvariablen von Vertexshadern sind Nummern zugewiesen (von 0 bis `MAX_ATTRIBS`), der sogenannte *position index*. Durch `glBindAttribLocation` wird ein solcher Index einem symbolischen Variablennamen zugeordnet, der im Quellcode von Vertexshadern verwendet werden kann.

Tabelle 6.13: Sonstige OpenGL ES-Befehle

6.2 Erweiterungen von OpenGL ES 2.0

6.2.1 EXT- und OES-Erweiterungen

Erweiterung	Kurzbeschreibung
OES_byte_coordinates	Ermöglicht die Angabe von Vertex- und Texturkoordinaten als Bytewerte.
OES_compressed_ETC1_RGB8_texture	Unterstützung von Texturen, die im ETC-Format komprimiert sind.
OES_compressed_paletted_texture	Unterstützung von palettierten Texturen.
OES_fixed_point	Unterstützung für Eingabedaten in Festkommaformaten (gedacht für Plattformen mit unzureichender Fließkommaunterstützung).
OES_read_format	Erlaubt die Abfrage, welche zusätzlichen (implementierungsspezifischen) Formate zur Verfügung stehen, um mit glReadPixels Daten aus dem aktuellen Framebuffer in den Hauptspeicher zu kopieren.
OES_EGL_image	Ermöglicht die Erzeugung von Texturen und Renderbuffer, die sich ihren Speicher mit EGLImage-Objekten teilen.
OES_depth24	Ermöglicht die Nutzung von 24bit-tiefen Depthbuffer-Komponenten in Renderbuffer-Objekten.
OES_depth32	Ermöglicht die Nutzung von 32bit-tiefen Depthbuffer-Komponenten in Renderbuffer-Objekten.
OES_element_index_uint	Erlaubt die Übergabe von Indexdaten an glDrawElements im GL_UNSIGNED_INT-Datenformat.
OES_mapbuffer	Ermöglicht, den Inhalt von Vertexbuffer-Objekten in den Hauptspeicher zu mappen (und dort zu modifizieren).
OES_rgb8_rgba8	Ermöglicht 8bit-RGB und -RGBA als zusätzliches Speicherformat für Renderbuffer-Objekte.
OES_stencil1	Ermöglicht 1bit-Stencilkomponenten für Renderbuffer-Objekte.
OES_stencil4	Ermöglicht 4bit-Stencilkomponenten für Renderbuffer-Objekte.
OES_stencil8	Ermöglicht 8bit-Stencilkomponenten für Renderbuffer-Objekte.
OES_texture_3D	Unterstützung von 3D-Texturen.
OES_texture_half_float_linear	Liefert erweiterte Modi für Texturfilterung.
OES_texture_float_linear	Liefert erweiterte Modi für Texturfilterung.
OES_texture_half_float	Unterstützung für Texturformate mit 16bit-Fließkommakomponenten.

`OES_texture_float`	Unterstützung für Texturformate mit 32bit-Fließkommakomponenten.
`OES_texture_npot`	Unterstützung für erweiterte Wrappingmodi für Texturen, deren Ausdehnung keine Potenz von zwei ist.
`OES_vertex_half_float`	Erlaubt die Nutzung von 16bit-Fließkommazahlen als Vertexattributdaten.
`EXT_texture_filter_anisotropic`	Ermöglicht anisotropische Texturfilterung.
`EXT_texture_type_2_10_10_10_REV`	Unterstützung für das 2-10-10-10-Texturformat (zwei Bits für Alpha, jeweils zehn für R, G und B).
`OES_depth_texture`	Unterstützung für Texturen, die Tiefenpufferdaten speichern (sogenannte *depth textures*).
`OES_packed_depth_stencil`	Unterstützung für kombinierte Depth- und Stencilbuffer-Objekte (mit 24 Bits/Pixel für Tiefeninformation und 8 Bits/Pixel für Stencilinformation).
`OES_standard_derivatives`	Unterstützung für die (built-in) Ableitungsfunktionen in Fragmentshadern, die standardmäßig in der OpenGL ES 2.0 Shading Language nicht zur Verfügung stehen.
`OES_vertex_type_10_10_10_2`	Unterstützung des 10-10-10-2-Datenformats für Vertexattributdaten (drei Komponenten zu je zehn Bits und eine zu zwei Bits).
`OES_get_program_binary`	Ermöglicht das Ein- und Auslesen kompilierter Shaderprogramme.
`EXT_texture_compression_dxt1`	Unterstützung von Texturen, die im DXT1-Format komprimiert sind.
`EXT_texture_format_BGRA8888`	Unterstützung für das BGRA8888-Texturformat.
`EXT_discard_framebuffer`	Ermöglicht, den Inhalt des aktuellen Framebuffers für ungültig zu erklären (eröffnet manchen Implementierungen zusätzliche Optimierungensmöglichkeiten).
`EXT_blend_minmax`	Ermöglicht die Nutzung weiterer *blend equations* (zusätzlich zu den von `glBlendEquation` bereits akzeptierten).
`EXT_read_format_bgra`	Zusätzliche Formate für `glReadPixels`.
`EXT_multi_draw_arrays`	Erlaubt die Übergabe mehrerer Geometriedatenarrays beim Aufruf einer Draw-Funktion.
`OES_vertex_array_object`	Unterstützung für Vertexarray-Objekte zur serverseitigen Speicherung von Vertexarray-Zuständen (d. h. zusammengehörige Mengen von konstanten Vertexattributen bzw. Zeiger auf Vertexattributdaten), um schnell zwischen verschiedenen Zuständen wechseln zu können.
`EXT_shader_texture_lod`	Liefert zusätzliche Funktionen für Fragmentshader, durch die der Detailgrad beim Texturzugriff explizit bestimmt werden kann.

EXT_frag_depth	Erlaubt das Setzen des Tiefenwerts eines Fragments aus einem Fragmentshader-Programm heraus.
OES_EGL_image_external	Erlaubt die Nutzung von EGlImages als Ziel für Texturoperationen.
EXT_unpack_subimage	Zusätzliche Parameter für glPixelStorei, die das Befüllen einer serverseitigen Textur mit einem rechteckigen Bereich aus einer clientseitigen Textur erleichtern können.

Tabelle 6.14: EXT- und OES-Erweiterungen

6.2.2 Erweiterungen der AMD Corporation

Erweiterung	Kurzbeschreibung
AMD_compressed_3DC_texture	Unterstützung für Texturen, deren Texel nur aus einer oder zwei Komponenten bestehen, zum Beispiel für *normal* oder *luminance maps*.
AMD_compressed_ATC_texture	Unterstützung für Texturen, die nach dem ATC-Verfahren komprimiert wurden.
AMD_program_binary_Z400	Unterstützung für Shaderbinaries, die für AMDs Z400-Produktfamilie kompiliert wurden.
AMD_performance_monitor	Ermöglicht die Abfrage spezieller Hardwarecounter, die auf manchen AMD-GPUs verfügbar sind.

Tabelle 6.15: Erweiterungen der AMD Corporation

6.2.3 Erweiterungen von Apple Incorporated

Erweiterung	Kurzbeschreibung
APPLE_texture_2D_limited_npot	Unterstützung für erweiterte Wrappingmodi für Texturen, deren Ausdehnung keine Potenz von zwei ist (eingeschränkter als die OES-Erweiterung OES_texture_npot).
APPLE_rgb_422	Liefert ein zusätzliches Texturformat.
APPLE_framebuffer_multisample	Liefert Renderbufferobjekte, die Multisampling unterstützen.
APPLE_texture_format_BGRA8888	Erlaubt das Laden von Texturen im BGRA-Format.
APPLE_texture_max_level	Erlaubt den maximalen Miplevel für einzelne Texturen explizit festzulegen.

Tabelle 6.16: Erweiterungen von Apple Incorporated

6.2.4 Erweiterungen der NVIDIA Corporation

Erweiterung	Kurzbeschreibung
NV_fence	Ermöglicht die Nutzung sogenannter Fence-Befehle, die die Synchronisation zwischen CPU und GPU erleichtern. Durch Fences werden partielle glFinish-Anweisungen möglich (d. h. das Clientprogramm wird nur bis zur Ausführung einer bestimmten Fence-Funktion blockiert und nicht zwingend solange, bis alle an den GL-Server übermittelten Befehle ausgeführt wurden). Außerdem kann der Ausführungszustand von Fence-Befehlen abgefragt werden, um Hinweise darauf zu erhalten, wie viele der aufgerufenen Befehle bereits vom GL-Server ausgeführt worden sind.
NV_coverage_sample	Liefert einen zusätzlichen Algorithmus für das Antialiasing.
NV_depth_nonlinear	Unterstützung für nicht-linearen Tiefenpuffer.
NV_draw_buffers	Ermöglicht Fragmentshadern, mehr als einen Ergebnisfarbwert zurückzuliefern.
NV_fbo_color_attachments	Erlaubt, mehr als einen Colorbuffer mit einem Framebufferobjekt zu verbinden.
NV_read_buffer	Ermöglicht, verschiedene Colorbuffer als Quelle für glReadPixels, glCopyTexImage2D und glCopyTexSubImage2D zu nutzen, statt nur den des aktuellen Framebuffers (Ausnahme: Der Frontcolorbuffer des aktuellen Framebuffers kann nicht als Quelle gewählt werden, wenn ein Backcolorbuffer vorhanden ist).
NV_read_buffer_front	Wie NV_read_buffer, nur dass zusätzlich auch der Frontcolorbuffer des aktuellen Framebuffers als Quelle genutzt werden kann.
NV_read_depth	Ermöglicht es, mit glReadPixels Daten aus dem Depthbuffer auszulesen.
NV_read_stencil	Ermöglicht es, mit glReadPixels Daten aus dem Stencilbuffer auszulesen.
NV_read_depth_stencil	Ermöglicht es, mit glReadPixels Daten aus einem kombinierten Depth-Stencilbuffer auszulesen.
NV_texture_compression_s3tc_update	Erlaubt das Kopieren von Daten aus einer unkomprimierten in eine komprimierte Textur.
NV_texture_npot_2D_mipmap	Liefert Mipmapping-Funktionalität für Texturen deren Ausdehnung keine Potenz von zwei ist.

Tabelle 6.17: Erweiterungen der NVIDIA Corporation

6.2.5 Erweiterungen von Imagination Technologies Limited

Erweiterung	Kurzbeschreibung
IMG_read_format	Liefert weitere Formate für glReadPixels.
IMG_texture_compression_pvrtc	Unterstützung für das PowerVR-Texturkompressionsformat.
IMG_program_binary	Unterstützung für Programmbinaries im SGX_PROGRAM_BINARY_IMG-Format.
IMG_shader_binary	Unterstützung für vorkompilierte Shaderbinaries im SGX_BINARY_IMG-Format.
IMG_multisampled_render_to_texture	Ermöglicht Multisampling beim Rendern in eine Textur, ohne dass anschließend ein automatisches Downsampling durchgeführt wird.

Tabelle 6.18: Erweiterungen von Imagination Technologies Limited

6.2.6 Erweiterungen von Qualcomm Incorporated

Erweiterung	Kurzbeschreibung
QCOM_driver_control	Liefert spezielle Funktion zur Treiberkontrolle, gedacht für Debugging und Profiling während der Anwendungsentwicklung.
QCOM_performance_monitor_global_mode	Erlaubt das Auslesen von globalen Hardwarecountern. Globale Counter reagieren auf Aktionen aller Programme, die die GPU nutzen (nicht nur desjenigen Programms, das die Counter nutzt).
QCOM_writeonly_rendering	Ermöglicht einen sogenannten *write-only* Rendermodus, der für manche Anwendungen einen Performancegewinn mit sich bringen kann.
QCOM_extended_get	Liefert Funktionen, mit denen zusätzliche Informationen über den GL-Zustand abgefragt werden können. Gedacht für Debuggung.
QCOM_extended_get2	Liefert noch weitere Abfragefunktionen für das Debugging.
QCOM_tiled_rendering	Ermöglicht dem Clientprogramm, gezielt einzelne Teile des Framebuffers zum Rendern in schnellem Speicher unterzubringen.
QCOM_alpha_test	Führt den Alphatest von OpenGL ES 1.X für OpenGL ES 2.0 wieder ein.

Tabelle 6.19: Erweiterungen von Qualcomm Incorporated

6.2.7 Erweiterungen des ANGLE-Projekts

Erweiterung	Kurzbeschreibung
ANGLE_framebuffer_blit	Erlaubt das Kopieren von Daten zwischen verschiedenen Framebuffern.
ANGLE_framebuffer_multisample	Liefert Renderbufferobjekte, die Multisampling unterstützen.

Tabelle 6.20: Erweiterungen des ANGLE-Projekts

6.2.8 Erweiterungen von ARM Limited

Erweiterung	Kurzbeschreibung
ARM_mali_shader_binary	Ermöglicht das Laden von Shaderbinaries, die mit dem *Mali ESSL shader compiler* erzeugt wurden.
ARM_rgba8	Ermöglicht die Erzeugung von Renderbufferobjekten, die Daten im RGB8-Format speichern.

Tabelle 6.21: Erweiterungen von ARM Limited

6.2.9 Erweiterungen von DMP Incorporated

Erweiterung	Kurzbeschreibung
DMP_shader_binary	Erlaubt das Laden von Shaderbinaries, die für Chips der Digital Media Professionals Incorporated vorkompiliert wurden.

Tabelle 6.22: Erweiterungen von DMP Incorporated

6.2.10 Erweiterungen der Vivante Corporation

Erweiterung	Kurzbeschreibung
VIV_shader_binary	Erlaubt das Laden von Shaderbinaries, die für Chips der Vivante Corporation vorkompiliert wurden.

Tabelle 6.23: Erweiterungen der Vivante Corporation

Literaturverzeichnis

[ARM 2009] ARM: *Mali GPU OpenGL ES Application Development Guide*. 2009. – URL `http://infocenter.arm.com/help/topic/com.arm.doc.dui0363d/DUI0363D_opengl_es_app_dev_guide.pdf`. – Zugriffsdatum: 05.10.2011

[Bailey 2011] BAILEY, Mike: Using GPU Shaders for Visualization, Part 2. In: *Computer Graphics and Applications, IEEE* 31 (2011), März-April, Nr. 2, S. 67 –73. – ISSN 0272-1716

[Bautin u. a. 2008] BAUTIN, Mikhail ; DWARAKINATH, Ashok ; CHIUEH, Tzi-cker: Graphic engine resource management. In: *Proceedings of SPIE*, 2008

[Blackmer u. a. 2009] BLACKMER, Roy ; STEFANIZZI, Bruno ; WOLF, Andreas ; HART, Evan: *GL_ATI_meminfo OpenGL Extension Specification, Revision 0.2*. 2009. – URL `http://www.opengl.org/registry/specs/ATI/meminfo.txt`. – Zugriffsdatum: 06.07.2011

[Cole 2005] COLE, Phil: OpenGL ES SC – open standard embedded graphics API for safety critical applications. In: *The 24th Digital Avionics Systems Conference, 2005. DASC 2005.* Bd. 2, Oktober-November 2005, S. 8 pp. Vol. 2

[Dwarakinath 2008] DWARAKINATH, Ashok: *A Fair-Share Scheduler for the Graphics Processing Unit*, Stony Brook University, Diplomarbeit, 2008

[Grottel u. a. 2009] GROTTEL, Sebastian ; REINA, Guido ; ERTL, Thomas: Optimized data transfer for time-dependent, GPU-based glyphs. In: *IEEE Pacific Visualization Symposium, 2009. PacificVis '09.*, April 2009, S. 65 –72

[Hill u. a. 2008] HILL, Steve ; ROBART, Mathieu ; TANGUY, Emmanuel: Implementing OpenGL ES 1.1 over OpenGL ES 2.0. In: *International Conference on Consumer Electronics, 2008. ICCE 2008. Digest of Technical Papers.*, Januar 2008, S. 1 –2

[Kato u. a. 2011] KATO, Shinpei ; LAKSHMANAN, Karthik ; RAJKUMAR, Ragunathan R. ; ISHIKAWA, Yutaka: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2011

[Lagar-Cavilla u. a. 2007] LAGAR-CAVILLA, H. A. ; TOLIA, Niraj ; SATYANARAYANAN, Mahadev ; DE LARA, Eyal: VMM-Independent Graphics Acceleration. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments* ACM (Veranst.), 2007, S. 33–43

[Liu u. a. 2007] LIU, Weiguo ; MÜLLER-WITTIG, Wolfgang ; SCHMIDT, Bertil: Performance Predictions for General-Purpose Computation on GPUs. In: *International Conference on Parallel Processing, 2007. ICPP 2007.*, September 2007, S. 50. – ISSN 0190-3918

[The Khronos Group] THE KHRONOS GROUP: *Khronos OpenGL ES API Registry*. – URL `http://www.khronos.org/registry/gles/`. – Zugriffsdatum: 18.07.2011

[Microsoft 2006] MICROSOFT: *Windows Vista Display Driver Model*. 2006. – URL http://msdn.microsoft.com/en-us/library/aa480220.aspx. – Zugriffsdatum: 08.11.2011

[MSDN 2009] MSDN: *Timeout Detection and Recovery of GPUs through WDDM*. 2009. – URL http://msdn.microsoft.com/en-us/windows/hardware/gg487368.aspx. – Zugriffsdatum: 24.10.2011

[MSDN 2011] MSDN: *Event Objects*. 2011. – URL http://msdn.microsoft.com/en-us/library/windows/desktop/ms682655%28v=vs.85%29.aspx. – Zugriffsdatum: 09.11.2011

[Munshi u. a. 2008] MUNSHI, Aaftab ; GINSBURG, Dan ; SHREINER, Dave: *OpenGL(R) ES 2.0 Programming Guide*. 1. Addison-Wesley Professional, 2008. – ISBN 0321502795, 9780321502797

[Munshi und Leech 2010] MUNSHI, Aaftab ; LEECH, Jon: *OpenGL(R) ES Common Profile Specification, Version 2.0.25 (Full Specification)*. 2010. – URL http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf. – Zugriffsdatum: 01.07.2011

[Nvidia 2006] NVIDIA: *NVIDIA GeForce 8800 GPU Architecture Overview*. 2006. – URL http://www.nvidia.de/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf. – Zugriffsdatum: 03.10.2011

[Nvidia 2009] NVIDIA: *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009. – URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. – Zugriffsdatum: 11.10.2011

[Paul 2007] PAUL, Brian: *The Mesa 3D Graphics Library*. 2007. – URL http://www.mesa3d.org/. – Zugriffsdatum: 01.10.2011

[Satish u. a. 2009] SATISH, Nadathur ; SUNDARAM, Narayanan ; KEUTZER, Kurt: Optimizing the use of GPU memory in applications with large data sets. In: *International Conference on High Performance Computing (HiPC), 2009*, Dezember 2009, S. 408 –418

[Segal u. a. 2010] SEGAL, Mark ; AKELEY, Kurt ; FRAZIER, Chris ; LEECH, Jon ; BROWN, Pat: *The OpenGL(R) Graphics System: A Specification, Version 4.1 (Core Profile)*. 2010. – URL http://www.opengl.org/registry/doc/glspec41.core.20100725.pdf. – Zugriffsdatum: 01.07.2011

[Simpson und Kessenich 2009] SIMPSON, Robert J. ; KESSENICH, John: *The OpenGL(R) ES Shading Language, Version 1.00, Revision 17*. 2009. – URL http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf. – Zugriffsdatum: 05.07.2011

[Stroyan 2009] STROYAN, Howard: *GL_NVX_gpu_memory_info OpenGL Extension Specification, Revision 1.3*. 2009. – URL http://developer.download.nvidia.com/opengl/specs/GL_NVX_gpu_memory_info.txt. – Zugriffsdatum: 06.07.2011

[Trevett 2010] TREVETT, Neil: *Khronos Group Overview*. 2010. – URL http://www.webcitation.org/5znTVOLcr. – Zugriffsdatum: 29.06.2011

[Walbourn 2005] WALBOURN, Chuck: *Game Timing and Multicore Processors*. 2005. – URL http://msdn.microsoft.com/en-us/library/windows/desktop/ee417693%28v=vs.85%29.aspx. – Zugriffsdatum: 09.11.2011

Alle hier aufgeführten URLs wurden letztmalig am 15. Dezember 2011 überprüft. Das bei den einzelnen Literatureinträgen jeweils vermerkte Zugriffsdatum bezieht sich auf den Tag des ersten Zugriffs. Bei veränderlichen Internetseiten beziehen sich sämtliche Zitate auf die Version des Tages, an dem der Erstzugriff stattfand.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den 16. Dezember 2011,

Armin Cont