Institut für Parallele und Verteilte Systeme / Verteilte Systeme

Masterarbeit Nr. 1

# Entwurf und Analyse von Konzepten zur effizienten Datenübertragung von Grafikrendering-Befehlen auf eingebetteten Systemen

Andrej Eisfeld

**Studiengang:** Informatik

**Prüfer/in:** Prof. Dr. Kurt Rothermel

**Betreuer/in:** Dipl.-Inf. Stephan, Schnitzer
Dipl.-Inf. Simon, Gansel

**Beginn am:** 1. April 2014

**Beendet am:** 30. September 2014

**CR-Nummer:** I.3.2, C.2.4, C4

# Danksagung

Diese Arbeit wäre ohne die Unterstützung folgender Personen nicht entstannden:

Den größten Dank verdienen Stephan Schnitzer und Simon Gansel. Ohne sie gäbe es für mich keine Möglichkeit im Kontext des ARAMiS Projektes bei der Daimler AG an dieser Arbeit schreiben zu dürfen. Vor allem der Betreuung vor Ort durch Simon Gansel sind große inhaltliche Teile der Arbeit zu verdanken.

Der Daimler AG und allen weiteren Beteiligten des ARAMiS Projektes möchte ich außerdem für eine sehr angenehme Arbeitsatmosphäre danken. So danke ich auch Ulrich Krämer für seine fachliche Unterstützung und Joachim Schiele für sehr viele anregende Gespräche.

Gerne möchte ich auch einen besonderen Dank allen Betreuern während des Master- und des Bachelorstudiums aussprechen. So möchte ich Prof. Dr. Achim Karduck für eine wunderschöne Zeit und Bachelor Thesis in Australien danken. Auch Prof. Dr. Rainer Mueller der Hochschule Furtwangen verdanke ich durch sehr viele Praxisprojekte eine hervorragende Ausbildung in der technischen Informatik. Besonders erwähnen möchte ich auch Minh Cuong Tran, meinen Betreuer eines Praktikums bei der IBM Deutschland GmbH.

Auch meiner Familie und meiner Freundin Irina Rutz möchte ich für ihre Unterstützung über das komplette Studium hinweg danken. So hat Irina außerdem durch ein Probelesen, woruas viele Verbesserungsvorschläge folgten, viel zum Schreibstil der Arbeit beigetragen.

# Kurzfassung

Die Interaktion zwischen Mensch und Maschine war seit Anbeginn der Automobilindustrie ein wichtiger Faktor für die Kundenzufriedenheit. Heutzutage erfolgt diese verstärkt unter Einsatz von Anwendungen, die auf dem Kombiinstrument und dem Display des Infotainment Systems angezeigt werden. Werden beide Systeme auf einer Plattform betrieben, so bekommt die Isolation ihrer Anwendungen, auch durch die steigende Zahl dieser, eine immer höhere Bedeutung und kann durch Virtualisierung erzielt werden. Um ein effizientes Rendering von Grafikbefehlen in einer solchen virtualisierten Umgebung zu ermöglichen, entwarf die Daimler AG eine Middleware als Schnittstelle zwischen den Anwendungen und der GPU. Mögliche Optimierungspotentiale dieser, um eine nahezu native Performanz von Grafikanwendungen zu gewährleisten, sind Gegenstand dieser Arbeit.

Dazu wurden sowohl die Middleware, als auch typische OpenGL ES basierte Grafikanwendungen und ihnen zugrundeliegende Grafikbefehle untersucht und klassifiziert. Ausgehend davon wurden Optimierungslösungen vorgeschlagen und unter Hinzunahme verschiedener repräsentativer Szenarien hinsichtlich Performanz evaluiert. Ein weiterer Punkt der Arbeit ist die Skalierbarkeit der OpenGL ES Middleware. Zur Auswertung dieser dienen sowohl homogene, als auch heterogene Szenarien, die mit dem nativen Ausführungsverhalten der Szenarien verglichen. Ein wichtiger Aspekt, der ebenfalls zu behandelt wurde, ist das Management der beschränkten Systemressourcen auf eingebetteten Systemen.

Anhand der ausgewählten Szenarien wird gezeigt, dass die eingesetzten Konzepte sowohl die Performanz einzelner Anwendungen verbessern, als auch die Skalierbarkeit der Middleware insgesamt positiv beeinflussen. Dabei wird bis zu 98% der nativen Framerate erzielt. Dies wird hauptsächlich durch eine Puffer-basierte Asynchronisierung von synchronen Grafikbefehlen und einen effizienten Einsatz von Shared Memories erzielt.

# Inhaltsverzeichnis


**1. Einleitung** **13**

**2. Grundlagen** **17**

2.1. OpenGL ES . . . . . 17
2.2. EGL . . . . . 19
2.3. Virtualisierung . . . . . 21

**3. GLES / EGL Middleware** **23**

3.1. Architektur . . . . . 23
3.2. Kommunikationsprotokoll . . . . . 24
3.3. Shared Memory Ringpuffer . . . . . 24
3.4. Befehlsausführung . . . . . 25

**4. Konzeption & Implementierung** **27**

4.1. Überblick . . . . . 27
4.1.1. Klassifikation von OpenGL ES und EGL Funktionen . . . . . 27
4.1.2. Klassifikation von OpenGL ES und EGL Anwendungen . . . . . 28
4.1.3. Zuordnung von Optimierungen zu Klassen . . . . . 29
4.2. Zero-Copy Ringpuffer . . . . . 30
4.2.1. Kommunikation über Shared Memories . . . . . 30
4.2.2. Anpassung an die GLES / EGL Middleware . . . . . 32
4.2.3. Implementierung . . . . . 32
4.3. Zustandspuffer . . . . . 33
4.3.1. Problem synchroner Befehle . . . . . 33
4.3.2. Asynchrone Zustandsabfragen . . . . . 33
4.3.3. Implementierung . . . . . 34
4.4. Asynchrones eglSwapBuffers . . . . . 35
4.5. Effiziente Fehlerbehandlung . . . . . 36
4.6. Shared Memory Management . . . . . 37
4.6.1. Parameter und Systemannahmen . . . . . 37
4.6.2. Speicherverteilung bei ausreichender Verfügbarkeit . . . . . 38
4.6.3. Speicherverteilung bei begrenzter Verfügbarkeit . . . . . 38
4.6.4. Implementierung . . . . . 41

**5. Evaluation** **43**

5.1. Simulationsaufbau . . . . . 43
5.1.1. Systemaufbau . . . . . 43

5.1.2. Anwendungen . . . 45
5.1.3. Szenarien . . . 47
5.1.4. Messparameter . . . 52
5.2. Leistungsfähigkeit . . . 54
5.2.1. Zero-Copy Kommunikation . . . 54
5.2.2. Asynchronisierung . . . 58
5.3. Skalierbarkeit . . . 60
5.3.1. Shared Memory Management . . . 64
5.4. Überblick über die Ergebnisse . . . 67

**6. Verwandte Arbeiten** **69**
6.1. VMGL . . . 69
6.2. VirtGL . . . 70
6.3. BroadcastGL . . . 71
6.4. ClusterGL . . . 72

**7. Zusammenfassung und Ausblick** **75**

**A. Befehlsreferenz** **77**

**Literaturverzeichnis** **79**

**Glossar** **83**

# Abbildungsverzeichnis


1.1. Touchscreen des Tesla Model S . . . . . 13
1.2. Beispiel einer virtualisierten Lösung . . . . . 14

2.1. OpenGL ES Grafikpipeline . . . . . 18
2.2. Zusammenhang zwischen EGL und GLES . . . . . 19
2.3. SwapBuffers . . . . . 20
2.4. Virtualisierung . . . . . 21

3.1. Architektur der Middleware . . . . . 23
3.2. Forwarding von OpenGL ES Befehlen . . . . . 24
3.3. Aufbau des Shared Memory Kanals . . . . . 25
3.4. Native Befehlsausführung von OpenGL ES . . . . . 25
3.5. Forwarding von OpenGL ES Befehlen . . . . . 26

4.1. Klassen von GLES Anwendungen . . . . . 28
4.2. Mmap in Kernelspace . . . . . 30
4.3. Mmap in Userspace . . . . . 31
4.4. Mmap in Userspace, „zero copy“ Ansatz . . . . . 31
4.5. Zero-Copy Ringpuffer Realisierung . . . . . 32
4.6. Problem synchroner GLES und EGL Befehle . . . . . 33
4.7. Uniform Caching . . . . . 34
4.8. Fehlerbehandlung . . . . . 36
4.9. Shared Memory Management Einheit . . . . . 37
4.10. Beispiel einer Equal-Framerate Verteilung . . . . . 39
4.11. Beispiel einer Optimal-Framerate Verteilung . . . . . 40
4.12. Implementierung Shared Memory Management Einheit . . . . . 41

5.1. Evaluation Setup . . . . . 43
5.2. Screenshots der Szene *build* der Anwendung *glmark2* . . . . . 45
5.3. Screenshot der es2gears Anwendung . . . . . 46
5.4. Screenshot der Anwendung Quake 3 Arena . . . . . 46
5.5. Befehlshistogramm: Szenario 1 . . . . . 47
5.6. Befehlshistogramm: Szenario 2 . . . . . 48
5.7. Befehlshistogramm: Szenario 3 . . . . . 49
5.8. Befehlshistogramm: Szenario 4 . . . . . 50
5.9. Middleware Skalierbarkeit: Heterogenes Szenario . . . . . 50
5.10. Screenshot des heterogenen Szenarios . . . . . 51

5.11. Evaluation Setup . . . . . . . . . . 52
5.12. Zero-Copy: Vergleich von Paketumlaufzeiten . . . . . . . . . . 54
5.13. Zero-Copy: Szenario 1 + Szenario 2 . . . . . . . . . . 55
5.14. Zero-Copy: Performanzvergleich in Quake 3 Arena (Szenario 5) . . . . . . . . . . 56
5.15. Zero-Copy: Transfervolumen von *glDrawArrays(. . . )* . . . . . . . . . . 57
5.16. Zustandspuffer: Szenario 3 + 4 . . . . . . . . . . 59
5.17. Middleware Skalierbarkeit: Applikationsklasse 1 . . . . . . . . . . 60
5.18. Middleware Skalierbarkeit: Applikationsklasse 2 . . . . . . . . . . 61
5.19. Middleware Skalierbarkeit: Applikationsklasse 3 . . . . . . . . . . 62
5.20. Middleware Skalierbarkeit: Applikationsklasse 4 . . . . . . . . . . 63

6.1. Architektur von VMGL . . . . . . . . . . 69
6.2. Architektur von VirtGL . . . . . . . . . . 70
6.3. Architektur von BroadcastGL . . . . . . . . . . 71
6.4. OpenArena über ClusterGL . . . . . . . . . . 72
6.5. ClusterGL-Anwendung mit 5 Anzeigeknoten . . . . . . . . . . 72

# Tabellenverzeichnis

4.1. Klassen von GLES und EGL Funktionen . . . . . . . . . . 27
4.2. Überblick: Optimierungskonzepte . . . . . . . . . . 29
4.3. Überblick: Transportkonzepte . . . . . . . . . . 32
4.4. Shared Memory Management Parameter . . . . . . . . . . 37

5.1. Freescale i.MX6Q . . . . . . . . . . 44
5.2. Positionierung und Auflösung der Automotive Anwendungen . . . . . . . . . . 51
5.3. Zero-Copy: Durchschnittliche Framezeiten bei Szenario 1 + 2 . . . . . . . . . . 56
5.4. Uniform caching: Durchschnittliche Framezeiten . . . . . . . . . . 58
5.5. Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 2 . . . . . . . . . . 61
5.6. Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 3 . . . . . . . . . . 62
5.7. Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 4 . . . . . . . . . . 63
5.8. Evaluation: Optimale Speicherverteilung bei ausreichender Verfügbarkeit . . . . . . . . . . 64
5.9. Evaluation: Gleichverteilung des ermittelten Speicherbedarfs . . . . . . . . . . 65
5.10. Evaluation: Equal-Framerate Speicherverteilung . . . . . . . . . . 65
5.11. Evaluation: Optimal-Framerate Speicherverteilung . . . . . . . . . . 66
5.12. Überblick: Messergebnisse der Leistungsfähigkeit . . . . . . . . . . 67
5.13. Überblick: Messergebnisse der Skalierbarkeit . . . . . . . . . . 67

A.1. GLES 2.0 Befehlsreferenz . . . . . . . . . . 77
A.2. EGL Befehlsreferenz . . . . . . . . . . 78

# Verzeichnis der Algorithmen

4.1. Optimale Zuweisung von Shared Memory Kanälen bei ausreichendem Speicher . . . . 38
4.2. Equal-Framerate Zuweisung von Shared Memory Kanälen . . . . . . . . . . . . . . . . 39
4.3. Optimal-Framerate Zuweisung von Shared Memory Kanälen . . . . . . . . . . . . . . . 40

5.1. Messung der Latenz . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.2. Messung der Paketumlaufzeit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.3. Messung der Framerate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

# 1. Einleitung

> Nichts in der Welt steht einzeln und irgend ein Wirksames muß nicht als Ende, sondern als Anfang betrachtet werden.
>
> (Johann Wolfgang von Goethe)

In der Informationstechnik der Automobilindustrie haben sich in den letzten Jahren zwei Tendenzen herauskristallisiert. Zum einen nimmt die Anzahl der elektronischen Steuergeräte stetig zu [1], sodass es heute üblich ist in Oberklassenfahrzeugen um die 100 Steuergeräte, die untereinander über Bussysteme verbunden sind, zu verbauen. Als weitere Tendenz zeichnet sich heraus, dass analoge Anzeigeinstrumente immer mehr durch Displays verdrängt werden. Eines der besten Beispiele hierfür ist das Display des Tesla Model S [2], bei dem sowohl das Kombiinstrument, als auch die Hauptbedieneinheit (siehe Abbildung 1.1) über digitale Anzeigegeräte umgesetzt sind. Letztere Komponente bezeichnet das Infotainment System des Fahrzeugs.

**Abbildung 1.1.**: Touchscreen des Tesla Model S [3]

Der Einsatz einer stetig steigenden Zahl von Steuergeräten zieht mehrere Konsequenzen mit sich: Neben einem erhöhten Energie- und Bauraumbedarf, entstehen durch die Verwendung mehrerer, teilweise redundanter, Hardwarekomponenten auch höhere Kosten. Ebenfalls nicht zu vernachlässigen ist der daraus

resultierende Kommunikationsaufwand zwischen den einzelnen Geräten. Um dem entgegenzuwirken, ist man heute darauf bestrebt Steuergeräte zu zentralisieren [4]. Die zentralisierten Steuereinheiten können dabei auf Displays, in Form von Anwendungen, visualisiert werden. So kann über das Touchscreen des Tesla Model S u.A. die Scheinwerferbeleuchtung, das Öffnen des Schiebedaches oder auch die Zentralverriegelung gesteuert werden.

Kommen Anwendungen mit verschiedenen Anforderung auf einer zentralen Steuereinheit zum Einsatz, so kann es zum Wettstreit um die Hardwareressourcen kommern. Eine mögliche Lösung: Virtualisierung. Diese ermöglicht den Betrieb mehrerer logisch paralleler Systeme auf einer gemeinsamen Hardwareplattform. Als Beispiel einer solchen virtualisierten Lösung sei Abbildung 1.2 angeführt, die ein Kombiinstrument und ein Navigationssystem darstellt. Darin werden die Applikationen für die Geschwindigkeits- und Drehzahlmesser in einer sog. virtuellen Maschine (VM) isoliert von der Navigationsanwendung in einer anderen VM ausgeführt.

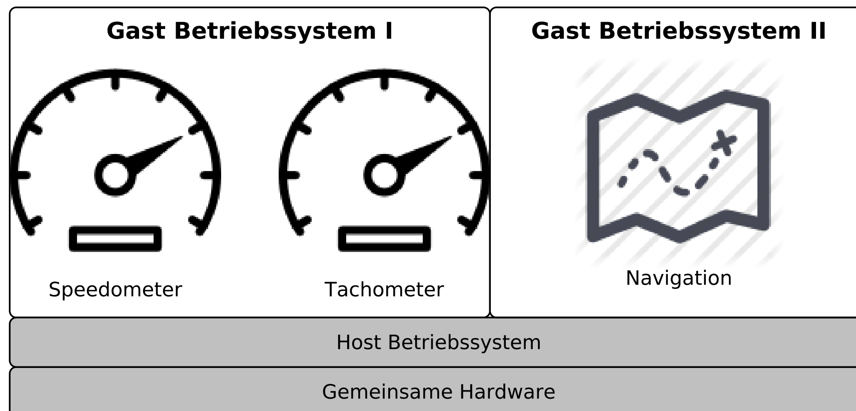



**Abbildung 1.2.**: Beispiel einer virtualisierten Lösung

Der Zugriff der virtualisierten Systeme auf die gemeinsamen Hardwareressourcen, z. B. auf eine Grafikkarte, stellt jedoch eine besondere Herausforderung dar, da die Isolation der Systeme, sowie eine gute Performanz gewährleistet werden müssen. Gerade im Falle der Grafikkarte existiert kein Virtualisierungskonzept, das eine ausreichende Isolation liefert [5]. Daraus folgt, dass der Grafiktreiber in einer einzigen VM ausgeführt werden muss, sodass andere VMs keinen direkten Zugriff auf die GPU haben. Um trotz dieser Limitierung 2D und 3D Rendering auf allen VMs zu ermöglichen, muss die Grafik-Bibliothek abstrahiert und Grafikdaten effizient zwischen verschiedenen VMs übertragen werden.

Hierfür wurde von der Daimler AG eine Middleware entworfen, die den Austausch von Grafikbefehlen zwischen verschiedenen VMs unterstützt. Da das entwickelte System auf eingebetteten Plattformen zum Einsatz kommt und eine Vielzahl von Anwendungen unterstützen soll, muss Aspekten wie der Speicher- und Energieeffizienz, Skalierbarkeit und auch der Leistungsfähigkeit einzelner Anwendung besondere Beachtung geschenkt werden. Im Rahmen der vorliegenden Arbeit sollen daher mögliche Optimierungen dieser Middleware aufgezeigt und evaluiert werden.

## Gliederung

Nach dem einleitenden Kapitel folgt das Kapitel **Grundlagen**, in dem ein Überblick über die eingesetzten Standards zur Entwicklung von Grafikanwendungen gegeben wird. Zusätzlich wird hier das Konzept der Virtualisierung beschrieben.

Das Kapitel **GLES / EGL Middleware** führt in die Middleware Architektur sowie grundlegende Bestandteile dieser ein. Dazu zählen das eingesetzte Kommunikationsprotokoll, sowie die der Kommunikation zugrundeliegende Datenstruktur. Abgerundet wird das Kapitel durch eine detailliertere Beschreibung der Abarbeitung von Grafikbefehlen durch die Middleware.

Das Kapitel **Konzeption & Implementierung** stellt den Kern der Arbeit dar. Darin werden Engpässe bzw. Limitierung der bestehenden Middleware analysiert und konkrete Verbesserungen vorgeführt. Diese zielen auf ein effizientes Übertragen großer Datenmengen und eine asynchrone Anwendungsausführung unter Berücksichtigung begrenzter Ressourcen ab.

Die Optimierungen werden im Anschluss ausgiebig im Kapitel **Evaluation** ausgewertet. Dabei werden die Optimierungen sowohl unter Betracht einzelner Anwendungen unter die Lupe genommen, als auch die Summe dieser in heterogenen und homogenen Szenarien.

Ein Vergleich zu bereits existierenden Lösungsersuchne wird im Kapitel **Verwandte Arbeiten** durchgeführt.

Die Arbeit wird mit dem Kapitel **Zusammenfassung und Ausblick** abgeschlossen. Neben einer kurzen Zusammenfassung werden hier auch Anknüpfungspunkte vorgestellt.

# 2. Grundlagen

Grundlagenforschung betreibe ich dann, wenn ich nicht weiß, was ich tue.

(Wernher Freiherr von Braun)

Die Middleware, deren Optimierungen das Ziel der vorliegenden Arbeit ist, basiert auf einer Menge von Technologien und Standards, die ein effizientes Rendering von Grafikbefehlen ermöglichen sollen. Standards die die Grafikbefehle definieren, werden in den Abschnitten 2.1 und 2.2 eingeführt. Da die Middleware für die Kommunikation zwischen verschiedenen virtuellen Maschinen über einen gemeinsamen Speicher ausgelegt ist, wird auch das Konzept der Virtualisierung in Abschnitt 2.3 beschrieben.

## 2.1. OpenGL ES

Zur Entwicklung von 2D- und 3D Grafikanwendungen existieren eine Menge standardisierter, geräteunabhängiger Grafikbibliotheken, die die Grafikhardware für eine Hardware-beschleunigte Verarbeitung der Anwendungen nutzen. Ein ihnen zugrundeliegender Standard ist OpenGL [6], der von der Khronos Group [7] aktiv entwickelt wird. Dabei ist OpenGL 4.5 [8] die aktuelle Spezifikation des Standards. Auch für eingebettete Systeme existieren OpenGL Standards, die entsprechend als OpenGL ES [9] spezifiziert sind. Diese definieren eine Untermenge der OpenGL Funktionalitäten und sind dahingehend optimiert, dass die entwickelten Grafikanwendungen einen reduzierten Ressourcenverbrauch im Vergleich zu OpenGL basierten Anwendungen aufweisen. In dieser Arbeit wird die Version OpenGL ES 2.0 [10], die konform zu OpenGL 2.0 [11] ist, eingesetzt.

OpenGL ES 2.0, im folgenden GLES genannt, implementiert eine Grafikpipeline mit programmierbaren Shadern, die in Abbildung 2.1 zu sehen ist. Durch einen Aufruf von *glDrawArrays(...)* oder *glDrawElements(...)* wird die Verarbeitung von Vertexdaten durch diese angestoßen. Im Folgenden werden die einzelnen Stufen der Pipeline kurz angeschnitten. Die Beschreibungen sind hierbei [12] entnommen.

**Vertex Arrays / Buffer Objects**

Vertex Arrays speichern pro Vertex Daten. So werden beispielsweise Vertexposition, Farbwerte, Textur Koordinaten u. Ä. in einem Vertex Array definiert. Diese Daten werden im Hauptspeicher auf Clientseite gespeichert und bei jedem Aufruf von *glDrawArrays(...)* oder *glDrawElements(...)* in den Grafikspeicher kopiert. Vertex Buffer Objects (VBOs) können dazu eingesetzt werden um diese Daten im Grafikspeicher zu puffern und dadurch die Rendering Performanz zu verbessern. Um Bereiche im Grafikspeicher zu allozieren wird hierzu die GLES Funktion *glGenBuffers(...)* verwendet. Ein anschließendes hochladen der Vertexdaten in den Cache wird über *glBindBuffer(...)* und *glBufferData(...)* ermöglicht.

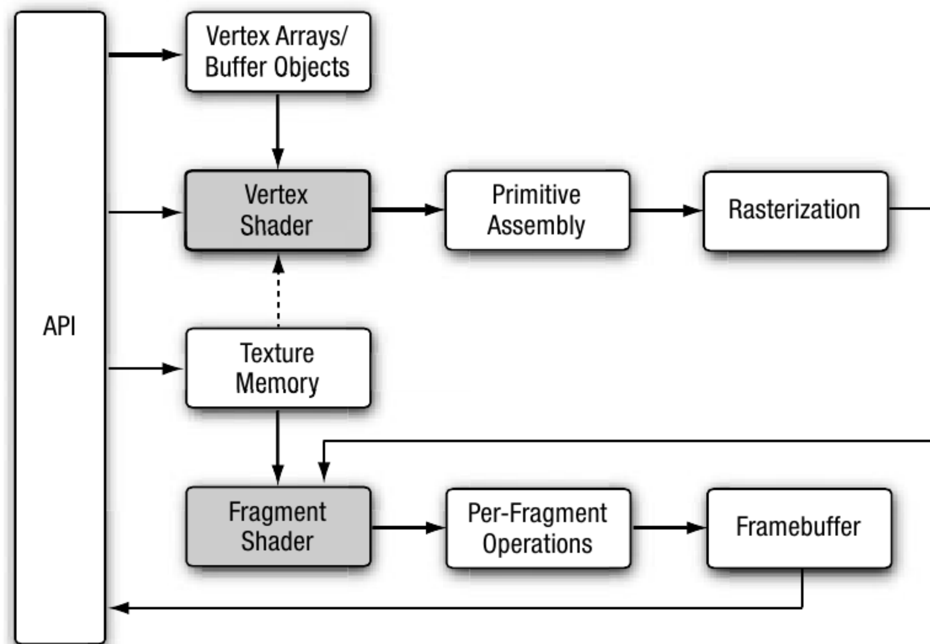



**Abbildung 2.1.**: OpenGL ES Grafikpipeline [12]

**Vertex Shader**

Der Vertex Shader ist eine programmierbare Methode die auf jeden Vertex aus einer Liste von Vertices angewendet wird. Typische Anwendungsfälle beinhalten das Transformieren dieser mit Hilfe einer Matrix, das Berechnen einer Vertex-Farbe oder auch das Transformieren von Texturkoordinaten. Der Vertex Shader benötigt als Eingabe Attribute (Zusatzdaten zu jedem Vertex), Uniforms (Konstante Daten die vom Vertex Shader verwendet werden sollen) und ein Shader Programm, das als Quellcode vorliegt. Als Ausgabe werden sog. variierende Variablen erzeugt, die als Eingabe für die Stufe „Primitive Assembly" und den Fragment Shader dienen.

**Primitive Assembly**

In diesem Schritt werden aus der Ausgabe des Vertex Shaders Primitive erstellt, die mit *glDrawArrays(. . . )* und *glDrawElements(. . . )* gezeichnet werden können. Beschrieben wird ein Primitiv durch eine Menge von Vertices, sowie zusätzliche Informationen wie beispielsweise Farbwerte. Primitive in GLES können entweder Dreiecke, Linien oder auch Punkte darstellen.

**Rasterization**

Rasterisierung ist der Prozess, der aus Primitiven Fragmente erstellt. Jedes Fragment stellt einen Pixel auf dem Bildschirm dar und wird durch seine Pixelkoordinaten identifiziert.

**Fragment Shader**

Der Fragment Shader ist eine programmierbare Methode, die auf jedes Fragment aus der Rasterisierungs

Stufe angewandt wird. Der Shader hat die Aufgabe die erhaltenen Fragmente zu verändern. So können Fragmente beispielsweise gelöscht oder ihr Farbwert geändert werden.

### Per-Fragment Operations

In dieser Stufe werden weitere Funktionen auf jedes Fragment angewandt. Als Ergebnis werden Fragmente entweder abgelehnt oder inklusive Farb-, Tiefen- und Stencilwert in einen Framebuffer geschrieben.

### Framebuffer

Der Framebuffer bezeichnet eine Speicherregion, dessen Inhalt auf dem Bildschirm angezeigt werden kann. Dieser wird jedoch nicht von OpenGL ES zur Verfügung gestellt, sondern von EGL [13]. Nähere Beschreibung zu EGL und Framebuffern wird deshalb im nächsten Kapitel näher eingegangen.

## 2.2. EGL

GLES Kommandos benötigen einen Kontext und eine Oberfläche zum Zeichnen. Die GLES API ist jedoch nicht in der Lage einen Kontext zu erstellen oder einen Kontext mit einem Fenstersystem zu verknüpfen. Für diesen Zweck wurden Programmierschnittstellen wie z.B. EGL entwickelt. Somit bildet EGL die Brücke zwischen GLES und dem Bildschirm Puffer oder einem zugrundeliegenden Fenstersystem (beispielsweise X11 [14] oder Wayland [15]). Auf Fenstersysteme wird hier nicht näher eingegangen.

Abbildung 2.2 zeigt den Zusammenhang zwischen GLES, EGL und einer Anwendung. Letztere nutzen eine GLES Bibliothek um in einen Framebuffer zu zeichnen (*Hintergrundpuffer*). Hat die Anwendung das Zeichnen abgeschlossen, so nutzt sie EGL um den Framebuffer freizugeben und den Inhalt auf den Bildschirm zu zeichnen.

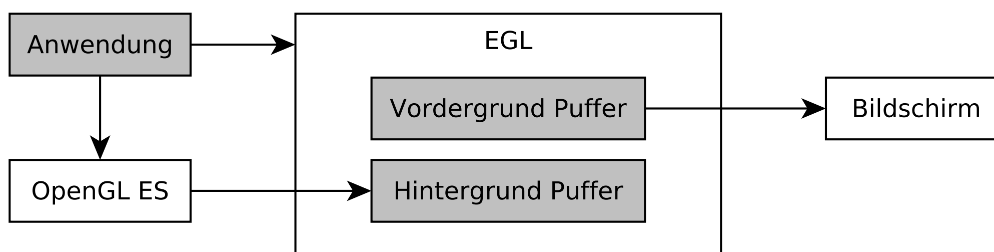



**Abbildung 2.2.**: Zusammenhang zwischen EGL und GLES

Die Freigabe des gezeichneten Framebuffers erfolgt über die Funktion *eglSwapBuffers(. . . )*. Diese Bewirkt einen Wechsel zwischen Vorder- und Hintergrundpuffer wie in Abbildung 2.3 zu sehen ist. Anschließend ist die Anwendung in der Lage in den neuen Hintergrund Puffer zu zeichnen, während der alte Frame auf den Bildschirm übertragen wird. Diese Technik wird auch *Double Buffering* genannt.

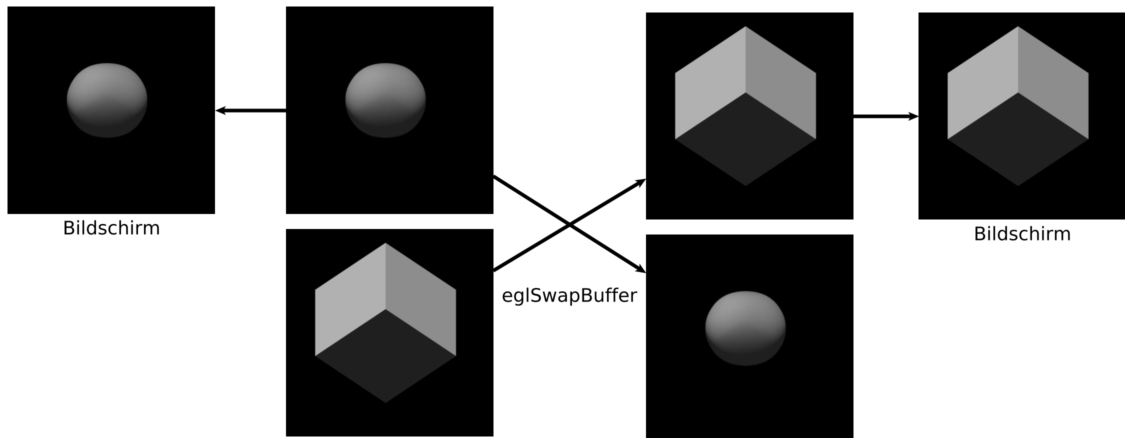



**Abbildung 2.3.**: SwapBuffers

Zu den weiteren Funktionen die in EGL definiert sind, zählen z.B.:

- Suche nach zur Verfügung stehenden Bildschirmen ⟶ *eglGetDisplay(. . . )*
- Erstellung einer Fensterfläche ⟶ *eglCreateWindowSurface(. . . )*
- Erstellung eines Anzeigekontextes ⟶ *eglCreateContext(. . . )*
- Verknüpfung von Kontext und Fensterfläche ⟶ *eglMakeCurrent(. . . )*

## 2.3. Virtualisierung

Virtualisierung stellt eine Technik zur Abstraktion von Systemressourcen [4] dar. Diese Abstraktion ermöglicht das Aufteilen von Ressourcen unter mehreren Laufzeitsystemen (*Virtuellen Maschinen (VM)*). Eine Hypervisor-basierte Virtualisierungslösung kann Abbildung 2.4 entnommen werden. Als Hypervisor wird hierbei eine Softwarekomponente bezeichnet, die die Umgebung für virtuelle Maschinen zur Verfügung stellt [4]. Somit ist der Hypervisor dafür verantwortlich, virtuelle Güter (Hardwareressourcen) den virtualisierten Systemen zuzuordnen. So werden in Abbildung 2.4 der Arbeitsspeicher und 3 der 4 CPU-Kerne unter den 3 Laufzeitsystemen (OS1, OS2, OS3) aufgeteilt. Aus dem ersteren resultiert, dass jedes System seinen eigenen isolierten Speicherbereich erhält und somit vor Zugriffen anderer Laufzeitsysteme geschützt ist.

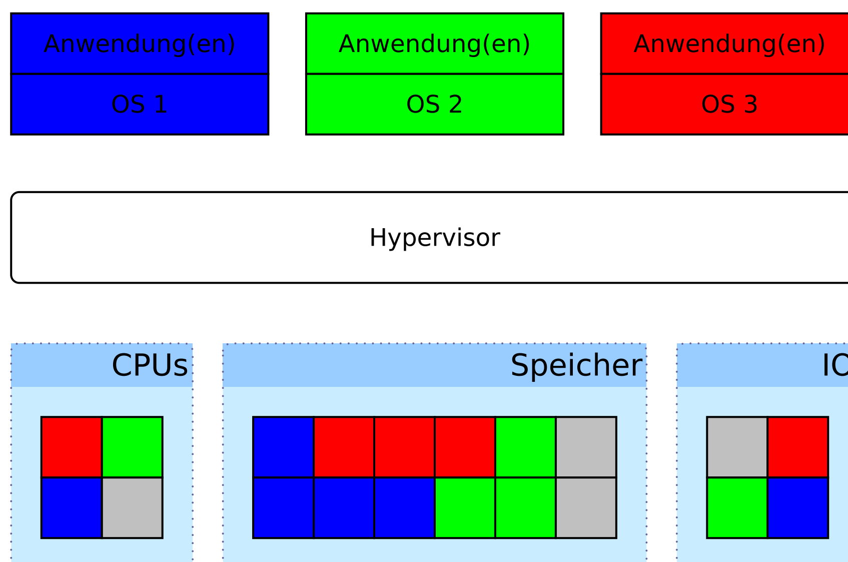



**Abbildung 2.4.**: Virtualisierung mittels Hypervisor [4]

Zur Kommunikation zwischen den virtuellen Maschinen existieren verschiedene Mechanismen [16]. Eine dieser ist der Einsatz von Shared Memories.

# 3. GLES / EGL Middleware

Doch Forschung strebt und ringt,
ermüdend nie, Nach dem Gesetz,
dem Grund, Warum und Wie.

(Johann Wolfgang von Goethe)

Dieses Kapitel beschreibt die Middleware für das Forwarding von OpenGL ES (GLES) und EGL Befehlen und stellt einen Vergleich mit der herkömmlichen, direkten Ausführung von Grafikbefehlen auf.

## 3.1. Architektur

Die Abbildung 3.1 zeigt die Architektur der Middleware für GLES und EGL Befehle auf.

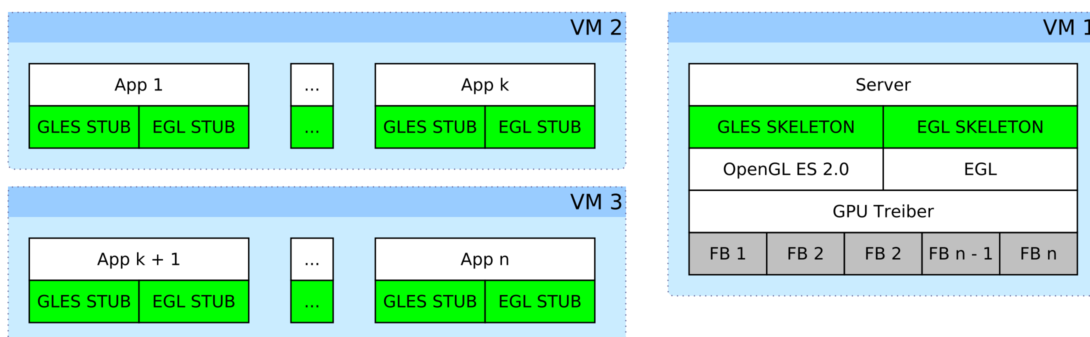



**Abbildung 3.1.**: Architektur der Middleware

GLES und EGL basierte Grafikanwendungen kommunizieren über Stub-Funktionen mit den entsprechenden Skeleton-Funktionen auf dem Server. Dabei erfolgt die Datenübertragung über einen gemeinsamen Speicher zwischen diesen (im folgenden Shared Memory genannt). Die einzelnen Anwendungen sind auf einer oder mehreren virtuellen Maschinen verteilt, wobei nur die VM des Servers einen direkten Zugriff auf die Grafikkarte besitzt. Der Server ist somit in der Lage aufgerufene Befehle Hardware-beschleunigt abzuarbeiten. Er verwaltet außerdem die Framebuffer in die gezeichnet werden kann.

Auf den Aufbau des Shared Memories, sowie das zum Einsatz kommende Kommunikationsprotokoll wird in Abschnitt 3.2 bzw. 3.3 näher eingegangen. Wie GLES und EGL Kommandos konkret durch die Middleware propagieren wird anschließend in Abschnitt 3.4 aufgezeigt.

## 3.2. Kommunikationsprotokoll

Die Kommunikation zwischen GLES-Stub und GLES-Skeleton erfolgt über ein einfaches Protokoll, das in Abbildung 3.2 zu sehen ist. Zunächst wird ein Funktionsidentifikator (FN_ID) übermittelt. Anhand dessen, ermittelt die Serveranwendung welcher GLES-Skeleton aufzurufen ist. Anschließend schreibt die Anwendung die benötigten Funktionsparameter, die vom Skeleton ausgelesen werden, in das Shared Memory Diese Parameter können entweder einzelne OpenGL Datentypen sein oder aus einer Menge davon bestehen. Im zweiten Fall wird dafür auch ein Wert übertragen, der angibt wie groß die zu empfangende Datenmenge ist (Size).

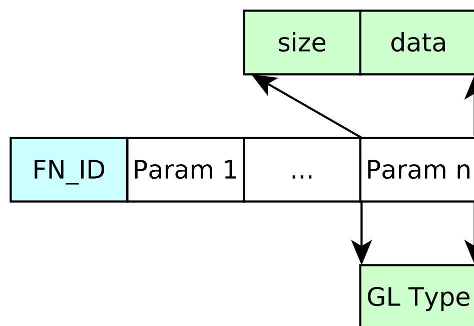



**Abbildung 3.2.**: Forwarding von OpenGL ES Befehlen

Für den Befehl *glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)* sieht das Übertragungsprotokoll beispielsweise wie folgt aus und verschickt 20 Byte an Daten:

```
| 50 | GLclampf | GLclampf | GLclampf | GLclampf |
```

## 3.3. Shared Memory Ringpuffer

Shared Memory Kommunikationskanäle werden als Ringpuffer realisiert. Die Vergabe von Shared Memory Kanälen an Grafikanwendungen ist Aufgabe des Servers. Dieser bestimmt auch ihre Größe (derzeit statisch). Dabei ist jeder Kanal unidirektional und somit werden für die bidirektionale Kommunikation zwei Kanäle pro Anwendung angelegt. Der Aufbau eines solchen Kanals ist in Abbildung 3.3 beschrieben. So besitzt jeder Kanal einen festen Bereich für Metainformationen wie die aktuelle Position des Schreib- und Lesezeigers und die für die Kommunikation zur Verfügung stehende Kanalgröße, die wie folgt definiert wird:

$$size(Kanal) = size(SharedMemory) - size(Kanalinformation) \tag{3.1}$$

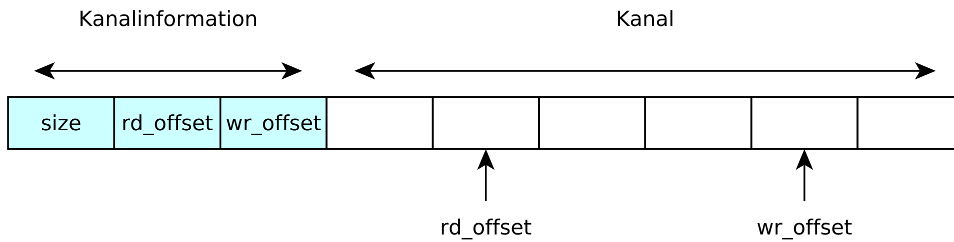



**Abbildung 3.3.:** Aufbau des Shared Memory Kanals

## 3.4. Befehlsausführung

Grafikanwendungen nutzen die API von GLES (oder anderen Grafikbibliotheken) um in einem Fenster zu zeichnen. Sobald die Anwendung im Userspace eine GLES Funktion aufruft, schreibt diese die dazugehörenden Befehle in den Puffer des Grafiktreibers im Kernelspace. Letzterer interpretiert die Befehle aus dem Puffer und steuert die Grafikkarte zur Verarbeitung an. Die beschriebene Abarabeitung ist in Abbildung 3.4 veranschaulicht.

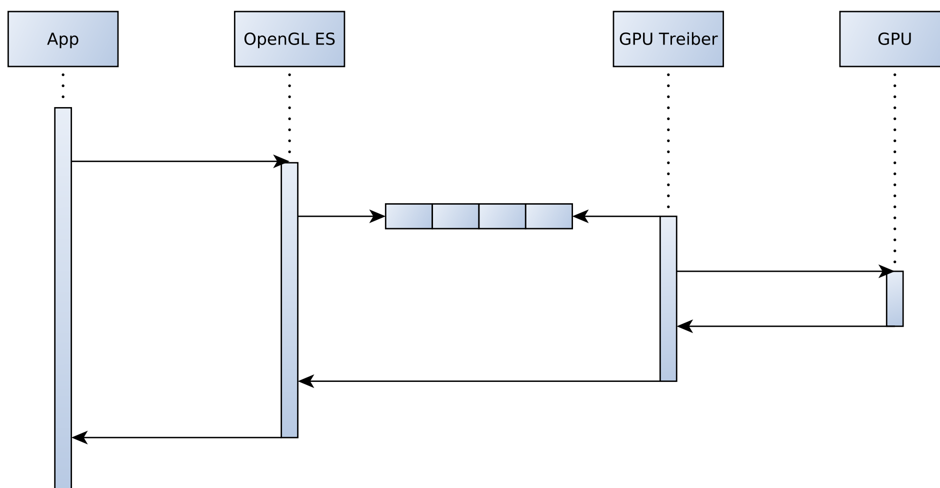



**Abbildung 3.4.:** Native Befehlsausführung von OpenGL ES

Besitzt man auf der anderen Seite eine Plattform ohne direkten Zugang zu einer Grafikkarte, so müssen die Befehle mit Hilfe einer Middleware weitergeleitet und auf einem entfernten Rechner ausgeführt werden (Remote Procedure Call). In unserem Fall leiten Gastsysteme ihre Befehle an den Host weiter, der diese verwertet. Abbildung 3.5 zeigt den Weg, den ein synchroner GLES Befehl zurücklegen muss. Anwendungen auf dem Gastsystem werden Stubs von GLES Funktionen zur Verfügung gestellt. Diese schreiben den Befehl in einen Shared Memory Ringpuffer, der auf Serverseite von dem dazugehörenden Skeleton ausgelesen wird. Der Skeleton führt anschließend die geforderte GLES Funktion lokal aus und generiert bei Bedarf eine passende Antwort für die Client Anwendung.

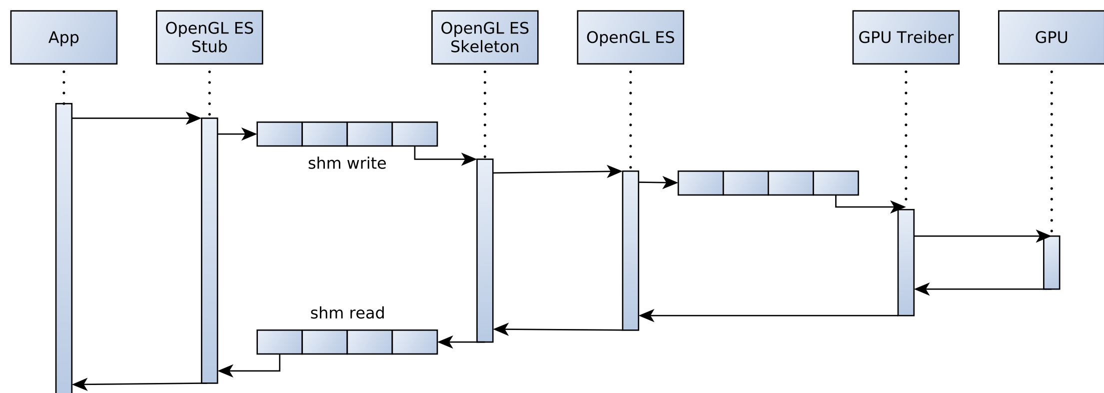



**Abbildung 3.5.**: Forwarding von OpenGL ES Befehlen

Wie der obigen Abbildung entnommen werden kann, erhält man im Vergleich zur nativen Befehlsausführung durch die eingeführten Zwischenschritte eine erhöhte Latenz:

$$L_{forwarding} = timediff(GLES_SKELETON, GLES_STUB) \quad (3.2)$$

# 4. Konzeption & Implementierung

Der Fortgang der wissenschaftlichen Entwicklung ist im Endeffekt eine ständige Flucht vor dem Staunen.

(Albert Einstein)

In diesem Kapitel werden verschieden Konzepte zur effizienten Übertragung von Daten zwischen verschiedenen VMs in eingebetteten Systemen vorgestellt. Dazu wird zunächst eine Klassifikation von GLES / EGL Funktionen und Anwendungen durchgeführt und den Optimierungskonzepten zugeordnet. Darauf folgen Konzepte die ein effizientes Übertragen großer Datenmengen und unter Hinzunahme von Puffern eine Asynchronisierung synchroner Funktionen ermöglichen sollen. Abschließend werden Strategien zur Verwaltung von begrenztem Shared Memory behandelt.

## 4.1. Überblick

### 4.1.1. Klassifikation von OpenGL ES und EGL Funktionen

Zunächst wird hier ein wichtiger Aspekt, der bei der Optimierung zu berücksichtigen ist, erläutert. Dabei handelt es sich um das Ausführungsverhalten von Grafikbefehlen. So können die Funktionen der GLES Bibliothek synchron oder auch asynchron abgearbeitet werden. Aus Sicht der Middleware, zählen Funktionen die einen Rückgabewert des Datentyps *void* besitzen zu den asynchronen und die mit allen anderen Rückgabetypen zu den synchronen Funktionen. Tabelle 4.1 klassifiziert beispielhaft einige ausgewählte Funktionen nach dieser Eigenschaft. Zu bemerken ist hier, dass in GLES / EGL lediglich die Funktionen *eglSwapBuffers(...)* und natürlich alle *glGet*(...)* Befehle synchron sind.

| | **Synchron** | **Asynchron** |
|---|---|---|
| **Konstant** | GLboolean eglSwapBuffers(...)<br>GLint glGetUniformLocation(...) | void glActiveTexture(...)<br>void glBindTexture(...) |
| **Variabel** | void glGetShaderSource(...) | void glVertexAttribPointer(...)<br>void glUniformMatrix4f(...) |

**Tabelle 4.1.**: Klassen von GLES und EGL Funktionen

Im Hinblick auf das Forwarding von diesen Befehlen spielt ebenfalls das Kommuniktationsaufkommen eine wichtige Rolle. Während die meisten Funktionen eine konstante Datenmenge als Parameter übertragen müssen, beispielsweise einen Enum-Wert bei *void glEnable(GLenum cap)*, existieren auch Befehle,

die variable Datenmengen auf die Serverseite bewegen müssen. Ein Beispiel hierfür ist die Methode *glVertexAttribPointer(...)*, deren Funktionsrumpf im Folgenden gezeigt wird.

```
void glVertexAttribPointer(GLuint index,  GLint size,
                           GLenum type,  GLboolean normalized,
                           GLsizei stride,  const
                           GLvoid * pointer);
```

Damit kann eine variabel lange Liste von Vertex Attributen spezifiziert und an den Server übermittelt werden.

### 4.1.2. Klassifikation von OpenGL ES und EGL Anwendungen

Um vorhersagen zu können, welche Optimierung für eine bestimmte Anwendung voraussichtlich geeignet ist, macht es Sinn OpenGL ES Anwendungen zu klassifizieren. Hierzu wurden für diese Arbeit zwei Kriterien ausgewählt, welche eine Unterteilung von Anwendungen in eine von 4 Klassen ermöglicht:

- Klasse 1: Renderingsaufwand $\gg$ Kommunikationsaufwand
- Klasse 2: Renderingsaufwand $\ll$ Kommunikationsaufwand
- Klasse 3: Renderingsaufwand $\approx$ Kommunikationsaufwand (beide hoch)
- Klasse 4: Renderingsaufwand $\approx$ Kommunikationsaufwand (beide gering)

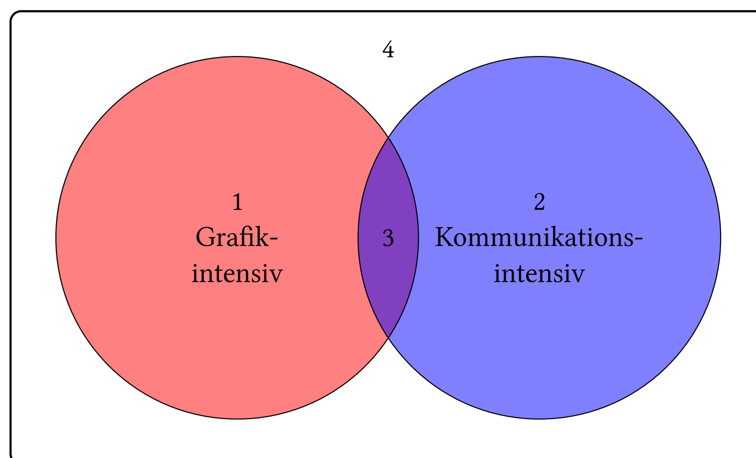



**Abbildung 4.1.:** Klassen von GLES Anwendungen

### 4.1.3. Zuordnung von Optimierungen zu Klassen

Abschnitt 4.2 stellt ein Konzept zur effizienten Übertragung von Daten über einen gemeinsamen Speicher zwischen virtuellen Maschinen vor (**Zero-Copy**). Vor allem wenn größere Datenmengen übertragen werden müssen, kann hier eine Optimierung stattfinden. Somit ist diese Optimierung hauptsächlich für die Klasse der asynchronen Funktionen mit variablen Datenmengen gedacht und kann Anwendungen der Klassen 2-4 optimieren. Da bei der Anwendungsklasse 1 die GPU als Flaschenhals zu sehen ist, ist hier keine relevante Verbesserung der Leistung zu erwarten.

Das zweite Konzept, das in Abschnitt 4.3 vorgestellt wird, zielt darauf ab die Abfrage von Zustandsvariablen über einen **Zustandspuffer** zu optimieren. Diese können über die synchronen GLES Funktionen abgefragt werden. Lediglich die Funktion *glGetError(...)* kann mit dem vorgestellt Konzept nicht umgesetzt werden und Bedarf einer Sonderbehandlung. Das hier vorgestellte Konzept ist zwar anwendungsunabhängig, bringt aber potentiell bei Anwendungen mit hohen Frameraten deutlichere Leistungssteigerungen mit sich.

Des Weiteren wird ein asynchrones **eglSwapBuffers(...)** in Abschnitt 4.4 eingeführt. Dies ist eine selbst definierte Erweiterung des EGL Standards und somit nicht Standard konform. Auch hier sind die Anwendungsklassen 1-4 Ziel der Optimierung.

Wie bereits erwähnt, wird die Fehlerabfrage mittels **glGetError(...)** gesondert behandelt. Abschnitt 4.5 beschreibt diese und zeigt unter welchen Umständen das Konzept eingesetzt werden kann.

Insgesamt ist zu erwähnen, dass die Anwendungsklasse 1 nicht viel Optimierungspotential bietet, da hier die Kommunikation zwischen Client und Server nicht der leistungsbegrenzende Faktor ist. Außerdem ist zu erwähnen, dass bis auf das „Zero-Copy“ alle Konzepte auf eine Asynchronisierung der Funktionen abzielen. Die eben beschriebenen Zuordungen von Konzepten zu Anwendungen, ist in Tabelle 5.12 zusammengefasst.

| Konzept | Funktionsklasse | Anwendungsklasse |
|---|---|---|
| Zero-Copy | asynchron (variabel) | 2-4 |
| Zustandspufferung | synchron | 1-4 |
| eglSwapBuffers | synchron (konstant) | 1-4 |
| glGetError | synchron (konstant) | 1-4 |

**Tabelle 4.2.**: Überblick: Optimierungskonzepte

## 4.2. Zero-Copy Ringpuffer

Die hier erwähnten Konzepte sind teilweise [17] entnommen. Sie zeigen wie Shared Memories eingesetzt werden können um die Kommunikation zwischen mehreren Teilnehmern zu ermöglichen. Anschließend wird näher auf die Anpassung an die GLES / EGL Middleware eingegangen, sowie die konkrete Implementierung beschrieben.

### 4.2.1. Kommunikation über Shared Memories

Auf der Transportschicht kann die inter-VM Kommunikation über einen gemeinsamen Speicher erfolgen wie in Abbildung 4.2 zu sehen ist. Hierbei wird ein Linux Kernel Modul benötigt, welches das Shared Memory in seinem Adressraum (im Kernelspace) abbildet und den Zugriff auf dieses zur Verfügung stellt. Über *read(. . . )* und *write(. . . )* Funktionen des Moduls können Applikationen auf den Speicher lesend und schreibend zugreifen. Das Kernelmodul realisiert hierbei den Schreib- und Lesezugriff über die Funktionen *copy_to_user(. . . )* und *copy_from_user(. . . )*. Dabei wird bei jedem Zugriff auf das Shared Memory ein Kontextwechsel vom Userspace in den Kernelspace verursacht. Wenn nun eine Anwendung Daten in den Speicher schreibt, die von einer anderen ausgelesen werden, werden insgesamt 4 Kontextwechsel und zwei Kopieroperationen für eine Datenübertragung benötigt.

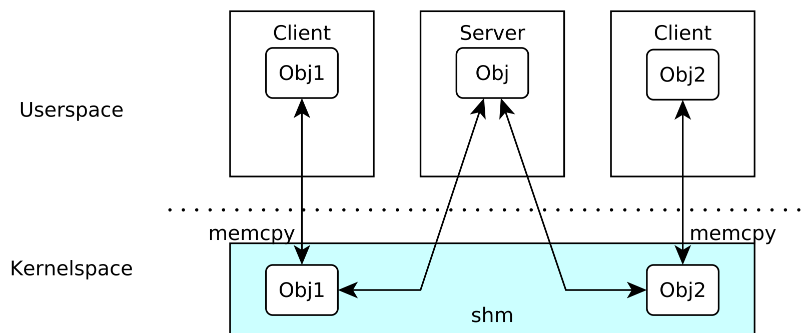



**Abbildung 4.2.**: Mmap in Kernelspace

Abbildung 4.3 zeigt wie eine Verbesserung des oberen Ansatzes erfolgen kann. Bei diesem Ansatz wird das Shared Memory in den Userspace der Anwendungen gemappt. Unter Linux erfolgt dies mit Hilfe der Funktion *remap_pfn_range(. . . )* im Kernel Modul. Nach dem Mapping können Applikation in verschiedenen VMs miteinander kommunizieren indem sie Ihre lokalen Daten mit *memcpy(. . . )* in das Shared Memory oder auf dem selben Weg aus dem Shared Memory in lokale Variablen kopieren. Im Vergleich zum vorherigen Ansatz werden hier die *read(. . . )* und *write(. . . )* Funktionen des Kernel Moduls nicht mehr benötigt, wodurch teure Kontextwechsel umgangen werden können. Diese Methode kommt auch in der Implementierung des Ringpuffers der GLES / EGL Middleware zum Einsatz. Ein bestehender Nachteil ist jedoch, dass das Kopieren von Daten über *memcpy(. . . )* ineffizient und unnötig ist.

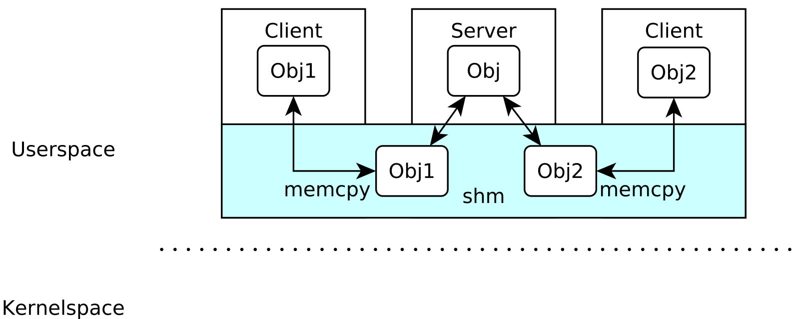



**Abbildung 4.3.:** Mmap in Userspace

Das Problem des Kopierens kann umgangen werden, da jede Applikation, durch das Mapping in den Userspace, direkt auf das Shared Memory zugreifen kann. Ein solcher Ansatz wird auch „zero copy“ Ansatz genannt und ist in 4.4 abgebildet. Im Endeffekt werden hier lokale Variablen direkt im Shared Memory alloziert, sodass Änderungen der Variablen direkt in anderen Applikation sichtbar sind. Dieses Konzept zieht jedoch auch einen Nachteil mit sich: Die Größe der Datenpakete die mit einem read() bzw. write() transportiert werden können ist durch die Größe des Shared Memories beschränkt. Ist das Shared Memory beispielsweise 4 MB groß und es muss ein Datenpaket der Größe 8 MB übertragen werden, können selbstverständlich keine 8 MB im Shared Memory alloziert werden. Somit muss das Shared Memory Segment mindestens so groß wie das größte zu erwartende Datenpaket gewählt werden.

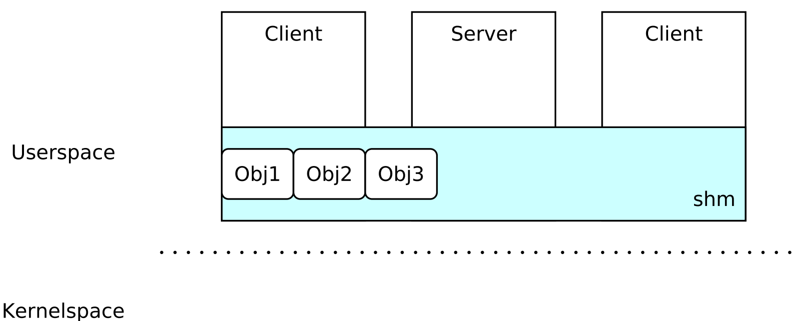



**Abbildung 4.4.:** Mmap in Userspace, „zero copy“ Ansatz

### 4.2.2. Anpassung an die GLES / EGL Middleware

Das Zero-Copy Konzept kann für das Forwarding von GLES und EGL Befehlen nicht wie beschrieben umgesetzt werden, da die Middleware transparent für Entwickler von Grafikanwendungen bleiben sollte. Das Konzept würde in diesem Fall erfordern, dass Entwickler Speicher für die Funktionsparameter manuell vor jedem Befehlsaufruf im Ringpuffer allozieren. Zusätzlich müssten die Funktionsrümpfe abgeändert werden um die Parameter als Referenz übergeben zu können.

Um die Transparenz zu gewährleisten, kann eine Hybridlösung umgesetzt werden. So können Daten auf der Clientseite in den Puffer kopiert und serverseitig ohne *memcpy()* direkt verwertet werden. Tabelle 4.3 gibt einen Überblick über die Transportkonzepte.

| Konzept | Kontextwechsel | Kopiervorgänge | Paketgrößen |
|---|---|---|---|
| Kernelspace mapping | 4 | 4 | $\infty$ |
| Userspace mapping | 0 | 2 | $\infty$ |
| Zero-Copy | 0 | 0 | size(shm) |
| Hybrid-Zero-Copy | 0 | 1 | size(shm) |

**Tabelle 4.3.:** Überblick: Transportkonzepte

### 4.2.3. Implementierung

Eine effiziente Implementierung des Zero-Copy Verfahrens wird durch gezieltes Mapping des Shared Memories auf zwei zusammenhängende Regionen im virtuellen Speicher erzielt. Falls ein Zugriff über die Grenze des ersten Puffers im virtuellen Speicher erfolgt, wird durch dieses Mapping automatisch auf den Anfang des Shared Memories zugegriffen. Sobald sowohl der Schreib- als auch der Lesezeiger auf die zweite Speicherregion zeigen (siehe Abbildung 4.5 rechts), können beide Zeiger um die Länge des Shared Memories reduziert werden, sodass sie anschließend wieder auf die erste Speicherregion zeigen.

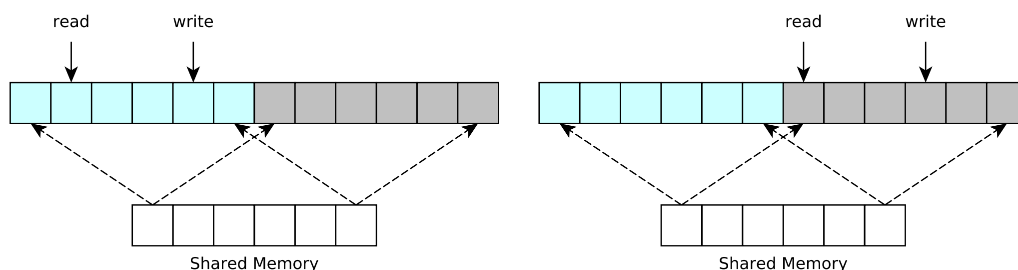



**Abbildung 4.5.:** Zero-Copy Ringpuffer Realisierung

## 4.3. Zustandspuffer

GLES 2.0 definiert eine große Menge an Parametern, die seitens der Anwendung abgefragt und / oder modifiziert werden können um die die GLES Operationen zu beeinflussen [12]. So existieren beispielsweise Funktionen, mit denen Details und Limitierungen der verwendeten GLES Implementierung in Erfahrung gebracht werden können. Aus Sicht der Middleware werden diese Funktionen synchron abgearbeitet, da der Anwendungskontext, der die Parameter verwaltet, serverseitig gespeichert wird.

### 4.3.1. Problem synchroner Befehle

Synchrone GLES Befehle führen zu einer deutlichen Reduzierung der Performanz einer Anwendung. Ein solcher Befehl führt dazu, dass Client und Server sich synchronisieren und der Befehlspuffer somit geleert wird. Erst nach Abschluss des Befehls beginnt die vom Client ausgeführte Anwendung mit dem erneuten Befüllen des Befehlspuffers. Die Latenz zwischen Versandt und Empfang des darauf folgenden Befehls ist dabei der Faktor der sich negativ auf die Leistung auswirkt. Dieses Problem ist auch in Abbildung 4.6 illustriert. Die eben beschriebene Latenz ist hierbei in Form des roten Pfeils gekennzeichnet. Es zeigt, dass der Server nach der Verarbeitung des synchronen Befehls im Leerlauf ist bis das nächste Kommando in den Befehlspuffer geschrieben wird.

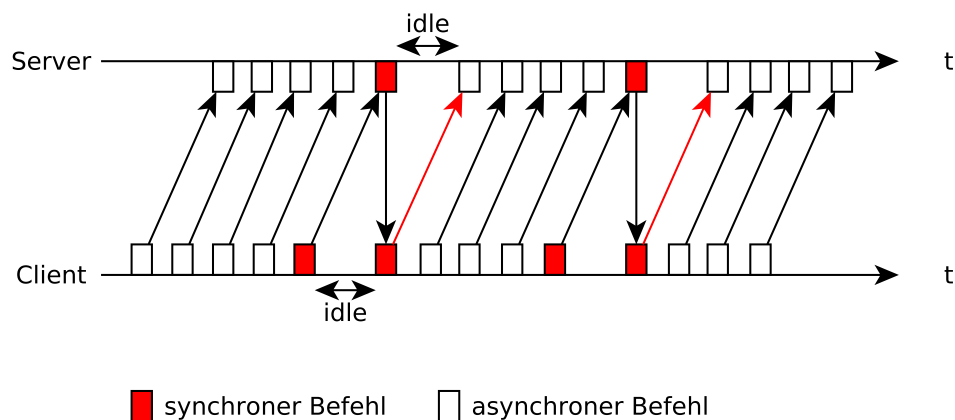



**Abbildung 4.6.**: Problem synchroner GLES und EGL Befehle

### 4.3.2. Asynchrone Zustandsabfragen

Einige synchrone GLES Befehle können durch den Einsatz von Puffern (engl. Cache) asynchronisert werden. Durch diese Optimierung wird nicht nur der Einfluss der Latenz beschwichtigt, es wird außerdem eine Paketumlaufzeit (RTT) eingespart, falls die abzufragende Zustandsvariable clientseitig im Puffer vorhanden ist. Beispiele für asynchronisierbare Funktionen sind u.a.:

- GLint glGetAttribLocation(GLuint program, const GLchar *name);

- GLint glGetUniformLocation(GLuint program, const GLchar *name);

Sowohl die Speicherpositionen von Uniforms als auch die von Attributen werden nach erfolgreichem Linken des Programm Objektes festgelegt. Diese Positionen können anschließend durch obige Funktionen abgefragt und genutzt werden um auf Attribute bzw. Uniforms zuzugreifen (über *glGetUniform* beispielsweise). Diese Position bleibt bis zum nächsten Linken unverändert und kann somit in einem Cache zwischengespeichert werden.

Weiterhin existiert eine Menge statischer Zustandsvariablen, die mit *glGet* Funktionen abgefragt und nach dem selben Prinzip asynchronisiert werden können. Die erwähnten Funktionen lauten wie folgt:

- void glGetBooleanv(GLenum pname, GLboolean * params)
- void glGetFixedv(GLenum pname, GLfixed * params)
- void glGetFloatv(GLenum pname, GLfloat * params)
- void glGetIntegerv(GLenum pname, GLint * params)

### 4.3.3. Implementierung

Zur Konzeptbestätigung wird die Asynchronisierung für Speicherpositionen von Uniforms realisiert. Die Implementierung benutzt eine Hash Funktion, die Parameternamen auf einen Index einer Liste mappt wie in Abbildung 4.7 zu sehen ist. Des Weiteren kommt eine Bitmap zum Einsatz um Kollisionen zu erkennen. Dabei steht die logische '1' für einen belegten Platz in der Liste und die '0' für einen leeren. Beim Aufruf von *glGetUniformLocation(...)* wird geprüft, ob der Wert bereits im Cache zwischengespeichert ist und nur bei einem Cache Miss wird der Wert vom Server angefragt und anschließend in den Cache geschrieben. Ist der Cache groß genug gibt es somit höchstens eine synchrone Kommunikation pro Uniform.

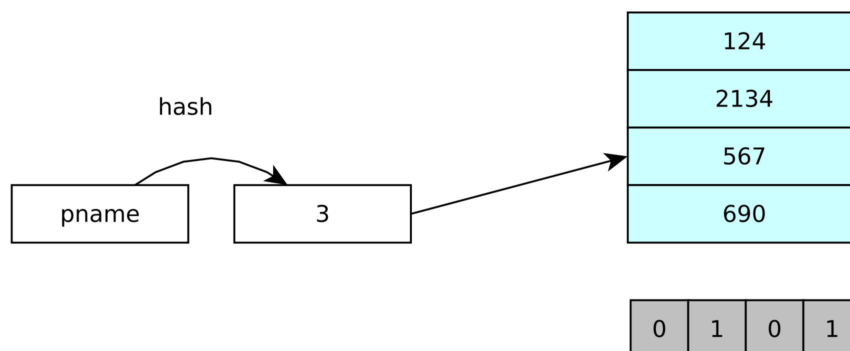



**Abbildung 4.7.:** Uniform Caching

## 4.4. Asynchrones eglSwapBuffers

Am Ende der Rendering-Schleife steht immer ein *eglSwapBuffers(...)* um den Frame auf das Fenster zu zeichnen. Im folgenden wird der Funktionsrumpf gezeigt:

```
EGLBoolean eglSwapBuffers(
          EGLDisplay display,
          EGLSurface surface);
```

Dem obigen Funktionsrumpf kann entnommen werden, dass dieser Befehl synchron ausgeführt wird, da der Client einen Rückgabewert erwartet. Abschnitt 4.3 hat bereits aufgezeigt welche Folgen diese Klasse von GLES Befehlen nach sich ziehen.

Eine Optimierung kann hier erfolgen, wenn der Rückgabewert ignoriert bzw. eine erfolgreiche Befehlsausführung auf Client-Seite angenommen wird. Dadurch wird der Befehl asynchron, womit keine Synchronisierung am Ende des Frames stattfindet. Dies hat jedoch den Nachteil, dass die Semantik von EGL verändert wird, sodass die resultierende Funktion eher folgenden Charakter hat:

```
void eglSwapBuffers(
          EGLDisplay display,
          EGLSurface surface);
```

## 4.5. Effiziente Fehlerbehandlung

Tritt ein Fehler während der Abarbeitung von GLES Funktionen auf, wird ein Error Flag auf einen numerischen Wert gesetzt, der den aufgetretenen Fehler identifiziert. So hat der Fehler „GL_OUT_OF_MEMORY" beispielsweise den Fehlercode 0x0505. Mit der Hilfe der Funktion *glGetError(...)* kann abgefragt werden, ob ein Error Flag gesetzt wurde. Da diese Funktion den Fehlercode als Rückgabewert besitzt, ist es somit ein synchroner Befehl. Im Fall des „GL_OUT_OF_MEMORY" Fehlers, ist der Zustand von GLES undefiniert. In allen anderen Fehlerfällen wird lediglich das Flag gesetzt und der Fehler verursachende Befehl nicht ausgeführt, sodass der Zustand von GLES und der Inhalt des Framebuffers unverändert bleiben.

Typischerweise sind folgende Codekonstrukte bei GLES Anwendungen vorzufinden:

```
if( glGetError() != GL_NO_ERROR ) {
  // Fehlerbehandlung
} else {
  // normale Weiterverarbeitung
}
```

Hier wird der Rückgabewert der Funktion für die Auswahl des Programmzweigs benötigt und kann deshalb nicht asynchron abgefragt werden. Solange die Fehlerabfrage jedoch nicht für Programmverzweigungen verwendet wird, beispielsweise falls sie nur zur Fehlerausgabe oder für einen Programmabbruch verwendet wird, kann diese asynchron mit Hilfe einer Callback Funktion umgesetzt werden. Ein Vergleich eines Programmablaufs mit asynchroner und synchroner Fehlerabfrage ist in Abbildung 4.8 zu sehen. Im asynchronen Fall, kann der Client weitere Befehle an den Server schicken ohne ein blockierendes Empfangen des Rückgabewertes von *glGetError(...)*.

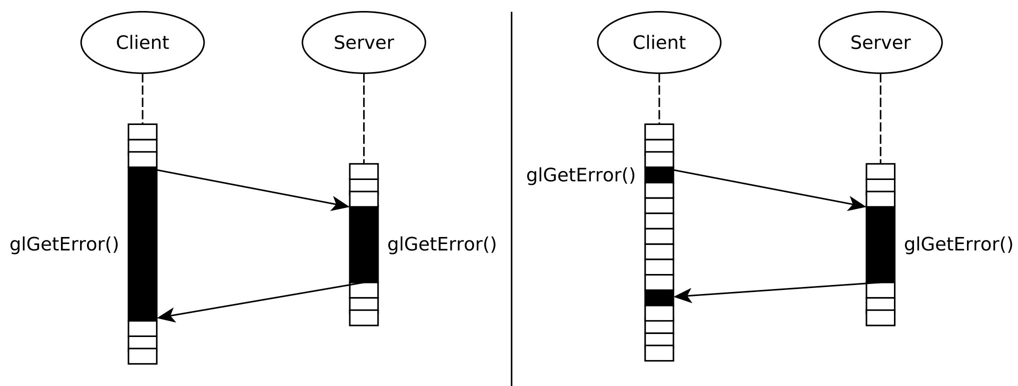



**Abbildung 4.8.**: Blockierender (links) und nicht-blockierende Fehlerabfrage (rechts)

## 4.6. Shared Memory Management

Wenn man jeder Anwendung unbegrenzt viel Shared Memory Speicher zur Kommunikation zuordnen könnte, so könnte jede von ihnen effizient Grafikbefehle übertragen. Jedoch ist der Speicher auf einem eingebettetem System begrenzt, womit eine Verwaltungseinheit im Server benötigt wird, die jeder Anwendungen „ausreichend" viel Speicher zur Kommunikation zur Verfügung stellt.

### 4.6.1. Parameter und Systemannahmen

Abbildung 4.9 zeigt einige Parameter die die Speicherverwaltung nutzen kann, um Entscheidung über die Speicherzuweisung zu treffen. So kann die aktuelle Framerate jeder Anwendung oder auch die aktuelle Speichergröße der Shared Memory Kanäle in Betracht gezogen werden. Sicherlich auch ein wichtiger Faktor ist die Frage, ob die Anwendung VBOs verwendet. Auch Anwendungsprioritäten können hier mit Berücksichtigt werden.

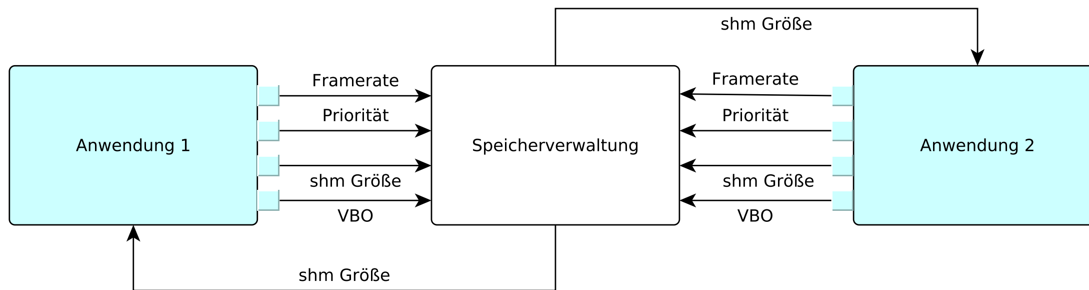



**Abbildung 4.9.**: Shared Memory Management Einheit

Tabelle 4.4 listet alle möglichen statischen und dynamischen Parameter auf:

| Statisch | Dynamisch |
|---|---|
| Ziel Framerate | Aktuelle Framerate |
| Verfügbarer Speicher | Aktuelle Speichergröße |
| VBO Einsatz | Anzahl Anwendungen |
| Anwendungspriorität | Framerate (letzte Messung) |

**Tabelle 4.4.**: Shared Memory Management Parameter

Bei den nachfolgenden Strategien zur Speicherverteilung werden folgende Annahmen getroffen:

1. Die optimale Ringpuffer Größe für eine Anwendung ist:

   (4.1) $RB_OPT_i : shm_size_i >= communication_load_per_frame_i, i \in Apps$

   Somit muss die Datenmenge pro Frame in den Puffer passen.

2. Eine Anwendung wird optimal ausgeführt, wenn eine optimale Puffergröße vorliegt:

   (4.2) $FPS_OPT_i : FPS_i | RB_OPT_i, i \in Apps$

3. Eine Menge von Anwendungen wird optimal ausgeführt, wenn jede Anwendung optimal ausgeführt wird:

   (4.3) $\forall i \in Apps : (1 - \frac{FPS_i}{FPS_OPT_i}) \leq \epsilon$

   Wobei $\epsilon$ ein Schwellenwert in % ist

### 4.6.2. Speicherverteilung bei ausreichender Verfügbarkeit

Ist für die aktuelle Menge an Anwendungen ausreichend Shared Memory verfügbar, so kann unter Berücksichtigung des zu übertragenden Datenvolumens pro Frame eine optimale Verteilung des Speichers an die Anwendungen erfolgen. In diesem Fall ist die Bedingung 1 bzw. die Gleichung 4.1 erfüllt.

**Algorithmus 4.1** Optimale Zuweisung von Shared Memory Kanälen bei ausreichendem Speicher

```
1: procedure assignMemory(communicationLoadLastFrame, currentMemorySize)
2:     if ( communicationLoadLastFrame < minShmSize ) then
3:         newShmSize = minShmSize
4:     else if ( communicationLoadLastFrame > maxShmSize ) then
5:         newShmSize = maxShmSize
6:     else
7:         newShmSize = communicationLoadLastFrame
8:     end if
9: end procedure
```

Der Parameter *communicationLoadLastFrame*, also die übertragene Datenmenge im letzten Frame, kann hierbei nach jedem *eglSwapBuffers(…)* ermittelt werden. Bei Bedarf kann außerdem hier eine Unter- sowie eine Obergrenze für die Größe des Shared Memory Kanals definiert werden.

### 4.6.3. Speicherverteilung bei begrenzter Verfügbarkeit

Ist nun auf der anderen Seite das Shared Memory eine begrenzte Ressource, kann die optimale Ringpuffer Größe (Gleichung 4.1) für jede Anwendung nicht garantiert werden. Somit muss das Shared Memory unter den Anwendungen so aufgeteilt werden, dass möglichst viele der Anwendungen Gleichung 4.3 erfüllen.

### Equal-Framerate Strategie

Eine mögliche Strategie der Speicherzuweisung versucht gleiche Frameraten bei allen Anwendungen zu erzielen. Diese Strategie wird durch Algorithmus 4.2 beschrieben. Hierbei werden die Anwendungen mit der geringsten und höchsten Framerate ermittelt (*winner* und *looser*). Anschließend wird eine definierte Menge an Speicher (*SHM_CHUNK*) der stärkeren Applikation entzogen und der schwächeren zur Verfügung gestellt.

**Algorithmus 4.2** Equal-Framerate Zuweisung von Shared Memory Kanälen

```
1: procedure ASSIGNMEMORY(Apps)
2:     for all app ∈ Apps do
3:         winner = min(averageFPS_winner, averageFPS_app)
4:         looser = max(averageFPS_looser, averageFPS_app)
5:     end for
6:     newShmSize[looser]− = SHM_CHUNK
7:     newShmSize[winner]+ = SHM_CHUNK
8: end procedure
```

Der Algorithmus konvergiert sobald

- jede Anwendung die gleiche Framerate besitzt oder
- die Anwendungen mit den höchsten Frameraten ihren kompletten Speicher an schwächere Anwendungen abgegeben haben

Die Abbildung 4.10 zeigt 3 Anwendungen mit einer linearen Abhängigkeit der Framerate von der Shared Memory Größe. Die Abbildung links zeigt die Anwendungen mit unterschiedlichen Frameraten und unterschiedlichem Verhältnis zwischen Shared Memory und dem benötigten Datenvolumen pro Frame (S/L). Der Algorithmus beendet, sobald die Frameraten bei allen Anwendungen gleich ist (rechtes Bild). Folglich versucht der Algorithmus alle Anwendungen entlang der FPS-Achse zu verschieben.

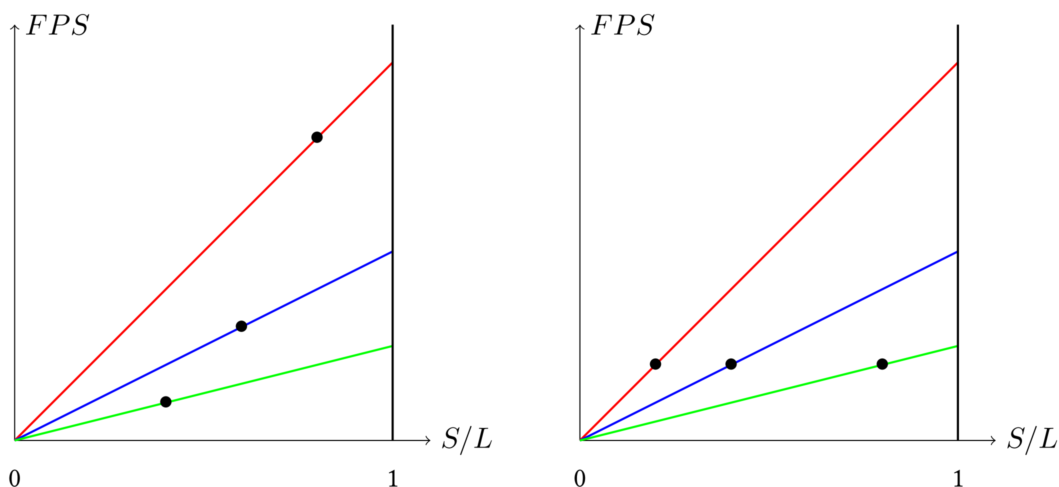



**Abbildung 4.10.**: Beispiel einer Equal-Framerate Verteilung

### Optimal-Framerate Strategie

Eine Verbesserung der *Equal-Framerate* Strategie kann erfolgen, wenn ein lineares Verhältnis zwischen der Framerate einer Anwendung und dem optimalen Shared Memory Kanal angenommen wird. So kann aus der Gleichung 4.1 folgende Funktion definiert werden:

$$FPS_{i+1} = FPS_i * \frac{shm_size_i}{communcation_load_per_frame_i} \tag{4.4}$$

Diese Funktion wird nun zur optimierten Speicherverteilung im Algorithmus 4.3 umgesetzt.

**Algorithmus 4.3** Optimal-Framerate Zuweisung von Shared Memory Kanälen

```
1: procedure ASSIGNMEMORY(Apps)
2:     for all app ∈ Apps do
3:         S_L_app = shmSize_app / communicationLoadPerFrame_app
4:         winner = min(S_L_winner, S_L_app)
5:         looser = max(S_L_looser, S_L_app)
6:     end for
7:     newShmSize[looser]− = SHM_CHUNK
8:     newShmSize[winner]+ = SHM_CHUNK
9: end procedure
```

Wie die nachfolgende Abbildung zeigt, versucht der Algorithmus alle Anwendungen entlang der x-Achse (S/L-Achse) nach rechts zu verschieben um so das Optimum zu erreichen. Der Algorithmus endet, sobald das Verhältnis zwischen Shared Memory und dem benötigten Datenvolumen pro Frame (S/L) bei allen Anwendungen gleich ist (rechtes Bild).

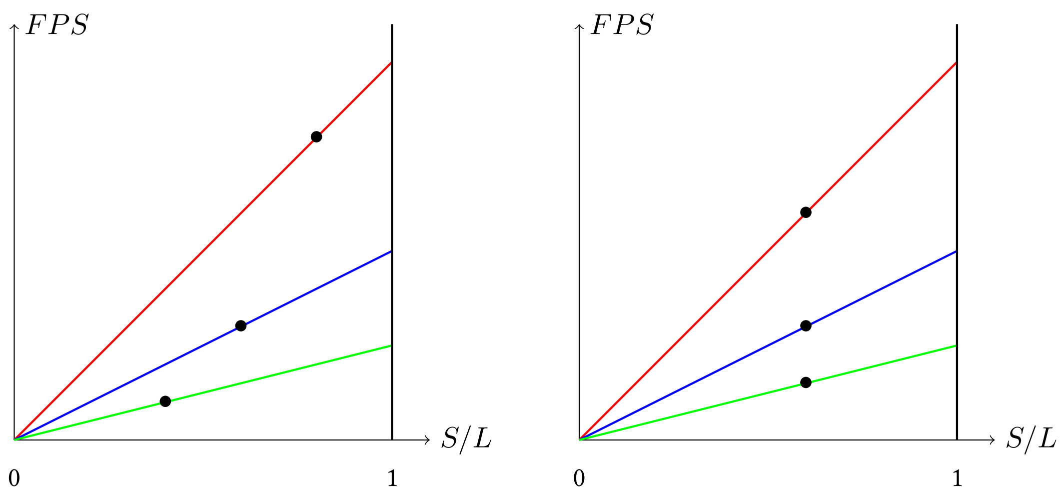



**Abbildung 4.11.:** Beispiel einer Optimal-Framerate Verteilung

### 4.6.4. Implementierung

Die Shared Memory Management Einheit ist als Zustandsmaschine implementiert. Die einzelnen Zustände sind in Abbildung 4.12 dargestellt.

1. Hier werden die zwei Anwendungen bestimmt (*winner & looser*), die einen Speicherblock abgeben bzw. empfangen sollen. Die Auswahl der Anwendungen erfolgt nach dem *Equal-Framerate* oder dem *Optimal-Framerate* Verfahren.
2. Die *looser* Anwendung gibt einen Teil ihres Speichers frei.
3. Die *winner* Anwendung alloziert einen neuen, um SHM_CHUNK vergrößerten, Kanal.
4. In diesem Zustand wird bestimmt, ob der Algorithmus konvergiert hat. Es wird also überprüft, ob alle Anwendungen ihr Optimum erreicht haben, oder ob die Speicherverteilung weiter fortgesetzt werden soll.

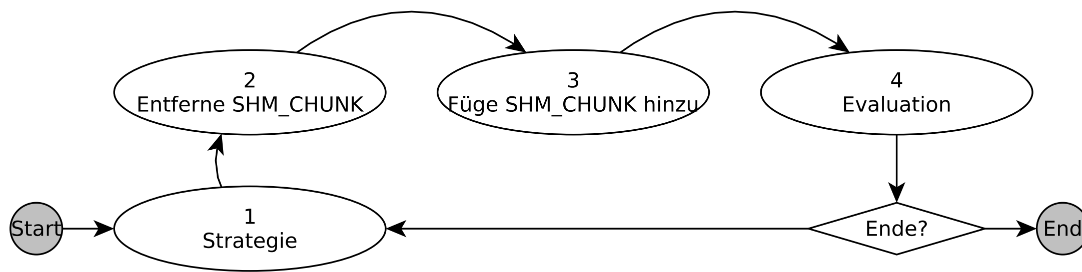



**Abbildung 4.12.**: Implementierung Shared Memory Management Einheit

# 5. Evaluation

Die Praxis sollte das Ergebnis des
Nachdenkens sein, nicht umgekehrt.

(Hermann Hesse)

In diesem Kapitel werden die vorgestellten Optimierungskonzepte evaluiert. Dazu wird zunächst der Systemaufbau, die verwendeten Anwendungen, sowie die zu messenden Parameter samt Messverfahren in Abschnitt 5.1 festgehalten. Außerdem werden hier Szenarien definiert, die dann in Abschnitt 5.2 verwendet werden, um die Auswirkungen einzelner Konzepte zu bewerten. Untersuchungen zur Skalierbarkeit der optimierten Middleware werden in Kapitel 5.3 durchgeführt. Dabei wird zusätzlich das Shared Memory Management unter Einsatz der verschiedenen Strategien bewertet.

## 5.1. Simulationsaufbau

### 5.1.1. Systemaufbau

Das Setup für die nachfolgende Evaluation ist in Abb. 5.1 illustriert. Es kommen hierbei 3 VMs zum Einsatz. Während auf der Host-VM der Server läuft, werden auf den Client VMs die Benchmark Applikationen (siehe Abschnitt 5.1.2) ausgeführt. Diese tauschen Grafikkommandos über einen dedizierten Shared Memory Kanal aus. Der Server bildet dabei die Komponente, die ein Hardware-beschleunigtes Rendering unter Einsatz einer GPU ermöglicht.

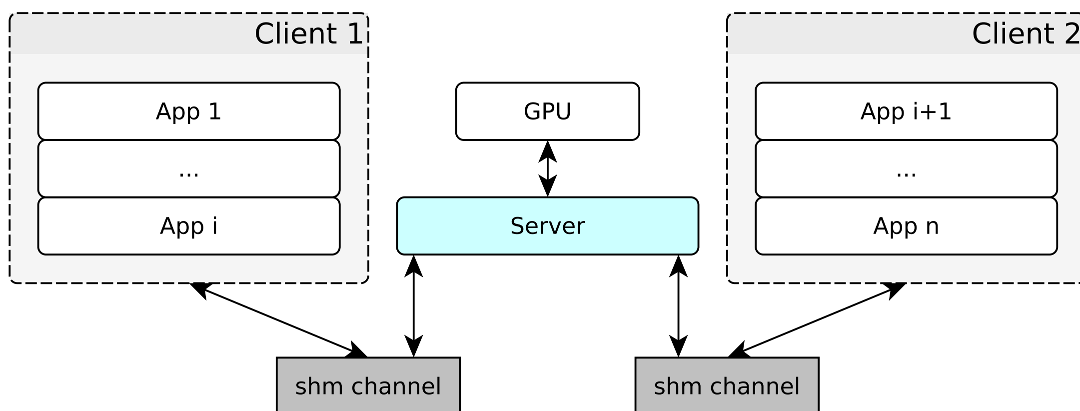



**Abbildung 5.1.**: Evaluation Setup

Der Systemaufbau wird auf der i.MX6Q Plattform von Freescale implementiert. Als Host Betriebssystem kommt hierbei ein ELinOS Linux zum Einsatz. Tabelle 5.1 zeigt die technischen Details dieser Plattform.

| | |
|---|---|
| **CPU** | 4x ARM® Cortex™-A9 mit bis zu 1.2 GHz |
| **GPU 3D** | Vivante GC2000<br>200Mtri/s 1000Mpxl/s, OpenGL ES 2.0 & Halti[a], CL EP[b] |
| **GPU 2D (Vector Graphics)** | Vivante GC355<br>300Mpxl/s, OpenVG 1.1 |
| **GPU 2D (Composition)** | Vivante GC320<br>600Mpxl/s, BLIT |
| **Arbeitsspeicher** | 2x32 LP-DDR2, 1x64 DDR3 / LV-DDR3 |

**Tabelle 5.1.**: Freescale i.MX6Q [18]

[a] Halti ist der Codename für OpenGL ES 3.0
[b] OpenCL Embedded Profile

### 5.1.2. Anwendungen

Die Anwendungen, die in Abbildung 5.1 in den virtuellen Maschinen zur Evaluation gestartet werden, werden in diesem Abschnitt vorgestellt.

**glmark2**

Glmark2 [19] bezeichnet eine hoch konfigurierbare Benchmark Anwendung für OpenGL ES 2.0. Hierfür stellt glmark2 einen Mix aus verschiedenen Szenen zur Verfügung, um verschiedene Aspekte von OpenGL ES 2.0 zu testen. Dabei liefert jede Szene zusätzliche Optionen um gewisse Aspekte der Ausführung zu modifizieren. So gibt es Szenen bei denen über diesen Weg bestimmt werden kann, ob VBOs verwendet werden sollen.

Näher untersucht und eingesetzt wird in dieser Arbeit die Szene *build*. Diese Szene bietet die Möglichkeit, über Optionen verschiedenen Modelle zu rendern und erlaubt je nach Bedarf ebenfalls die Verwendung von VBOs. Abbildung 5.2 zeigt zwei Modelle die über diese Option ausgewählt werden können. Das *horse* Modell ist ein vergleichsweise kleines Modell und besteht aus 3582 Vertices. Im Vergleich dazu besteht das *buddha* Modell aus 543,652 Vertices.

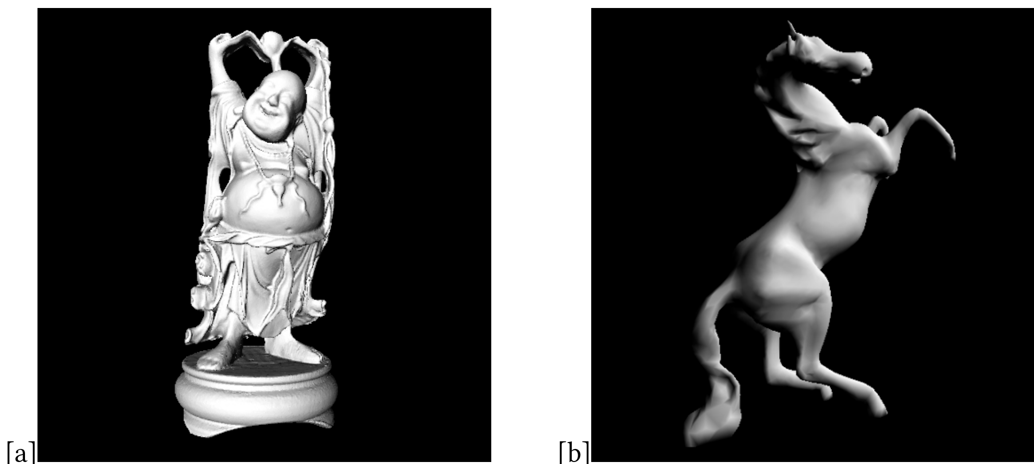



**Abbildung 5.2.**: Screenshots der Szene *build* der Anwendung *glmark2* mit dem Modell *buddha* (a) und dem Modell *horse* (b)

In dieser Arbeit wird die Version 2011.10 von *glmark2* verwendet [20].

**es2gears**

Bei *es2gears* [21] handelt es sich um eine simple Benchmark Anwendung, die aus einer Portierung der *glxgears* [22] Anwendung auf GLES2 entstand. Charakteristisch für *es2gears* ist, dass die Zahnräder aus einer geringen Menge an Vertices bestehen und keine Texturen zum Einsatz kommen. Somit ist die Anwendung nicht für Stresstests geeignet. Wie Abbildung 5.3 zeigt, visualisiert die Anwendung 3 rotierende Zahnräder.

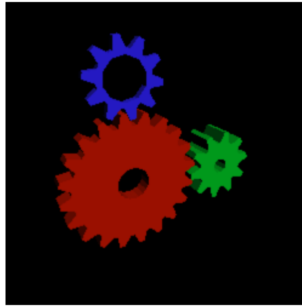


**Abbildung 5.3.:** Screenshot der es2gears Anwendung

**Quake 3 Arena**
Quake 3 Arena [23] bezeichnet ein Computerspiel, das 1999 veröffentlicht wurde. Quake 3 bietet dem Spieler die Möglichkeit Ausschnitte des Spiels aufzunehmen und anschließend die Aufnahme als Benchmark zu verwenden (*timedemo* [24]). Diese Aufnahmen werden ungebremst abgespielt und am Ende die erzielte Framerate ausgegeben.

Eine Anleitung nach der das Spiel in dieser Arbeit gebaut wurde, ist in [25] zu finden. Abbildung 5.4 zeigt hier einen Ausschnitt aus dieser Anwendung.

**Abbildung 5.4.:** Screenshot der Anwendung Quake 3 Arena

### 5.1.3. Szenarien

Aus den vorgestellten Anwendungen wird hier eine Menge von Szenarien und die erwarteten Ergebnisse der Evaluierung definiert.

**Szenario 1: Buddha ohne VBO – Auflösung: 514x540**
Das erste Szenario nutzt *glmark2* und das sehr großes Modell *buddha*. Hierbei kommen keine VBOs zum Einsatz und somit werden große Mengen an Vertexdaten pro Frame über die EGL / GLES Middleware an den Server übertragen. Einordnen lässt sich dieses Szenario aus diesem Grund in die Applikationsklasse 3.

Nachfolgend ist der Startbefehl, sowie ein Histogramm über die GLES Befehle über 1000 Frames abgebildet. Es ist zu erkennen, dass die Funktion *glGetUniformLocation* nicht genutzt wird, woraus zu folgern ist, dass der Zustandspuffer hier keine Auswirkungen auf die Leistung hat. Dadurch, dass auch die Funktion *glBindBuffer* (VBOs) nicht zum Einsatz kommt, ist hier jedoch ein starkes Optimierungspotential mit dem Zero-Copy Verfahren zu erwarten.

```
./glmark2-es2-forwarding  -b build:use-vbo=false:model=buddha
```

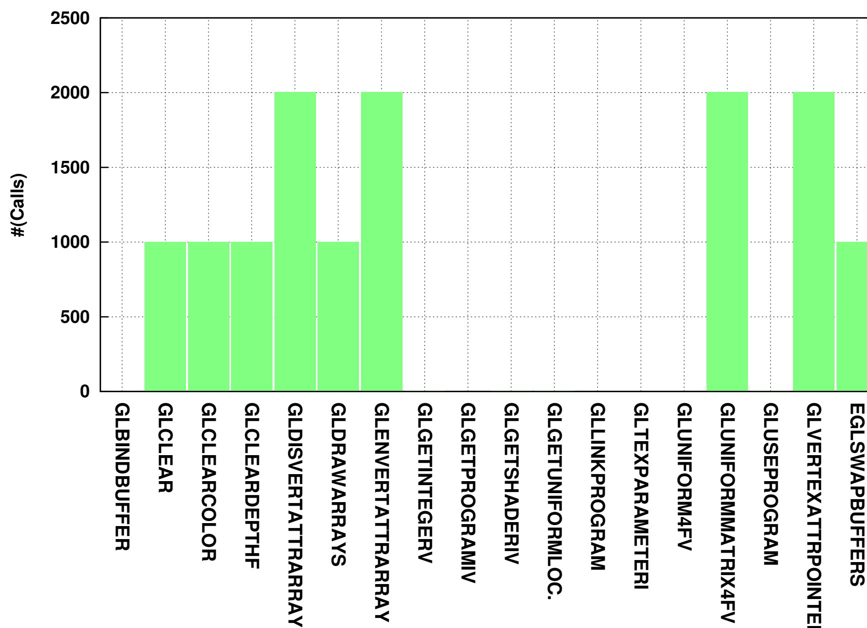



**Abbildung 5.5.**: Befehlshistogramm: Szenario 1

**Szenario 2: Buddha mit VBO – Auflösung: 514x540**
Auch das zweite Szenario verwendet das große *buddha* Modell, setzt jedoch VBOs ein um Vertexdaten serverseitig zu puffern, sodass die zu übertragende Datenmenge pro Frame sehr stark reduziert wird. Dieses

Szenario kann man der Applikationsklasse 1 zuweisen, da verhältnismäßig weniger Kommunikations- als Renderingsaufwand vorliegt.

Abbildung 5.6 zeigt ein Histogramm über die in diesem Szenario verwendeten GLES Funktionen. Dabei ist zu erkennen, dass im Vergleich zu Szenario 1, die Funktion *glBindBuffer* zum Einsatz kommt. Auch hier wird *glGetUniformLocation* nicht verwendet. Somit beide Optimierungen in diesem Szenario nicht einsetzbar. Jedoch ist zu erwarten, dass das Szenario bereits optimal ausgeführt werden kann, da hier die Grafikkarte als Flaschenhals fungiert. Das Szenario wird wie folgt gestartet:

```
./glmark2-es2-forwarding  -b build:use-vbo=true:model=buddha
```

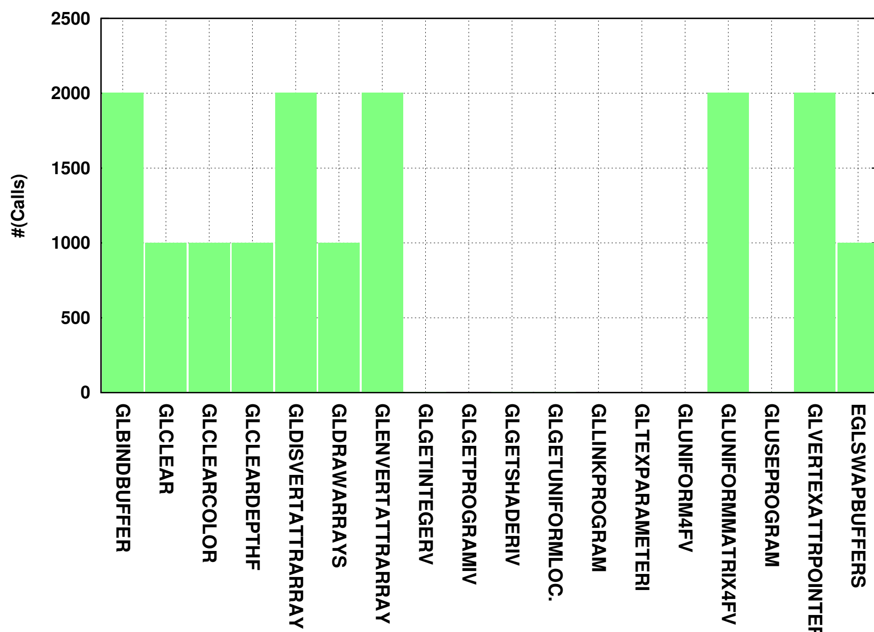



**Abbildung 5.6.:** Befehlshistogramm: Szenario 2

### Szenario 3: Horse ohne VBO – Auflösung: 514x540

Als nächstes wird *glmark2* mit einem kleineren Modell (*horse*) verwendet. Somit hat die Grafikkarte im Vergleich zu den bisherigen Szenarien weniger Arbeit. Dadurch, dass hier keine VBOs verwendet werden, wird verhältnismäßig mehr kommuniziert als gerendert. Damit fällt dieses Szenario in die Anwendungsklasse 2.

Auch hier wird ein Histogramm (siehe Abbildung 5.7) verwendet, um zu bestimmen, welche Optimierungen sinnvoll eingesetzt werden können. In der Abbildung ist zu erkennen, dass Zustandspuffer einen positiven Effekt auf die Framerate liefern sollten, da das Szenario *glGetUniformLocation* Funktionen nutzt. Auch das Zero-Copy Verfahren wird voraussichtlich Erfolge erzielen können (keine Verwendung von *glBindBuffer*).

Das Szenario kann wie folgt aufgerufen werden:

```
./glmark2-es2-forwarding  -b build:use-vbo=false:model=horse
```

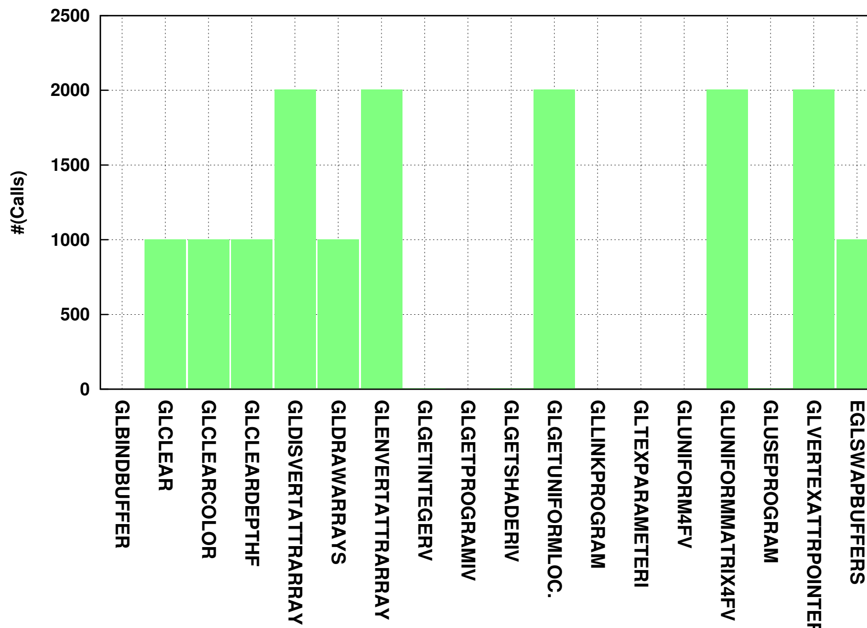



**Abbildung 5.7.:** Befehlshistogramm: Szenario 3

**Szenario 4: Horse mit VBO – Auflösung: 514x540**
Um die Anwendungsklasse 4 abzudecken, kann *glmark2* mit dem kleinen Modell *horse* und unter Hinzunahme von VBOs gestartet werden. Das dazugehörige Befehlshistogramm ist in Abbildung 5.8 abgebildet. Hier ist zu erwarten, dass das Konzept des Zustandspuffers, deutliche Gewinne erzielt.

Gestartet wird das Szenario über die Kommandozeile mit folgendem Befehl:

```
./glmark2-es2-forwarding  -b build:vbo=true:model=horse
```

**Szenario 5: Quake 3 timedemo – Auflösung: 1024x768**
Als ein äußerst realistisches Szenario wird abschließend Quake 3 Arena gewählt. Diese Anwendung lässt sich am ehesten in die Applikationsklasse 3 einordnen und somit ist eine Optimierung der Framerate mit dem Zero-Copy Verfahren zu erwarten. Das Spiel wird mit einer Auflösung von 1024x768 ausgeführt und die *timedemo* wird in der Quake 3 Konsole wie folgt aufgerufen:

```
> timedemo 1
> demo four
```

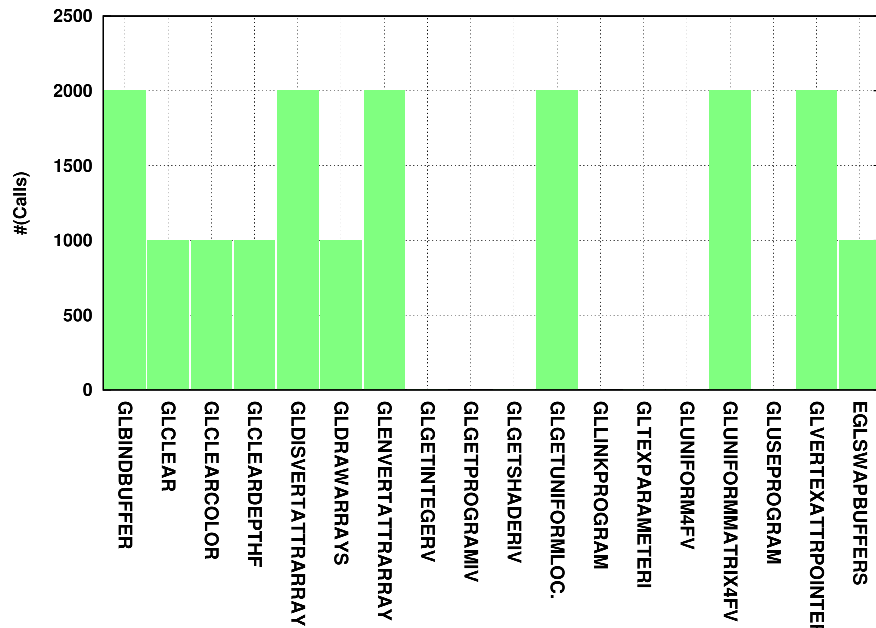



**Abbildung 5.8.:** Befehlshistogramm: Szenario 4

### Szenarien zur Skalierbarkeit

Um die Skalierbarkeit der Middleware zu bewerten werden mehrere Instanzen der Szenarien 1-4 verwendet. Es werden somit 4 homogene Szenarien verwendet. Dabei ist zu erwarten, dass im Vergleich zur nativen Ausführung der Szenarien, die Frameraten sich mit steigender Anzahl von Applikationen einander angleichen.

### Automotive Szenario

Um abschließend auch die Qualität der Shared Memory Verwaltungsstrategien bewerten zu können, wird ein weiteres heterogenes Szenario definiert (das *automotive szenario*). Dabei kommen 12 Anwendungen zum Einsatz, deren Platzierung in Abbildung 5.10 zu sehen ist.

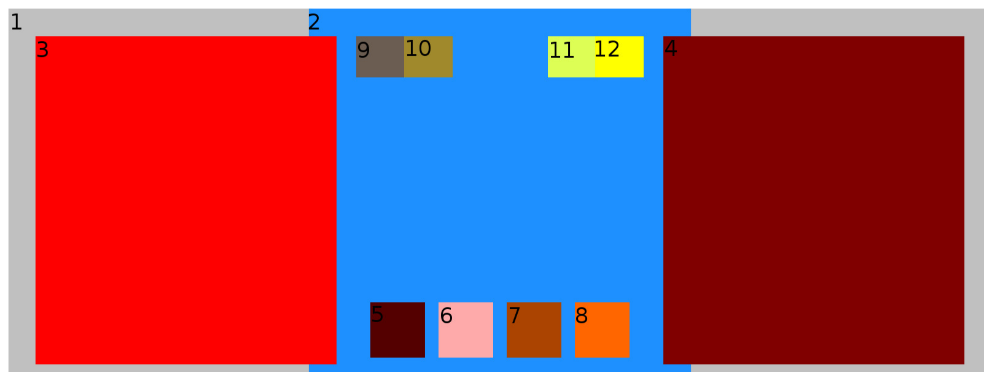



**Abbildung 5.9.:** Middleware Skalierbarkeit: Heterogenes Szenario

Tabelle 5.2 gibt das Offset sowie die Auflösung der einzelnen Anwendungen wieder. Dabei bezeichnet die Auflösung einer Anwendung, die Breite und Höhe des Anwendungsfensters. Das Offset, ausgehend vom linken unteren Bildschirmrand, gibt den Abstand zum Ursprung des Bildschirms an.

| # Anwendung | **Bezeichnung** | **Offset** | **Auflösung** |
|---|---|---|---|
| 1 | glmark2 | 0x0 | 1440x540 |
| 2 | Quake 3 | 440x0 | 560x540 |
| 3 | glmark2 | 40x20 | 440x480 |
| 4 | glmark2 | 960x20 | 440x480 |
| 5 | es2gears | 530x30 | 80x80 |
| 6 | es2gears | 630x30 | 80x80 |
| 7 | es2gears | 730x30 | 80x80 |
| 8 | es2gears | 830x30 | 80x80 |
| 9 | es2gears | 510x580 | 70x60 |
| 10 | es2gears | 580x580 | 70x60 |
| 11 | es2gears | 790x580 | 70x60 |
| 12 | es2gears | 860x580 | 70x60 |

**Tabelle 5.2.**: Positionierung und Auflösung der Automotive Anwendungen

Abschließend zeigt folgende Abbildung die eingesetzten Anwendungen auf dem Display.

**Abbildung 5.10.**: Screenshot des heterogenen Szenarios

### 5.1.4. Messparameter

Der folgende Abschnitt stellt die zu messenden Parameter samt Messmethodik vor.

**Latenz**

Für das Ermitteln der Latenz werden die Zeiten zwischen Funktionsaufruf auf der Clientseite und dem Empfang der Funktionsparameter auf Serverseite voneinander abgezogen (siehe Abbildung 5.11).

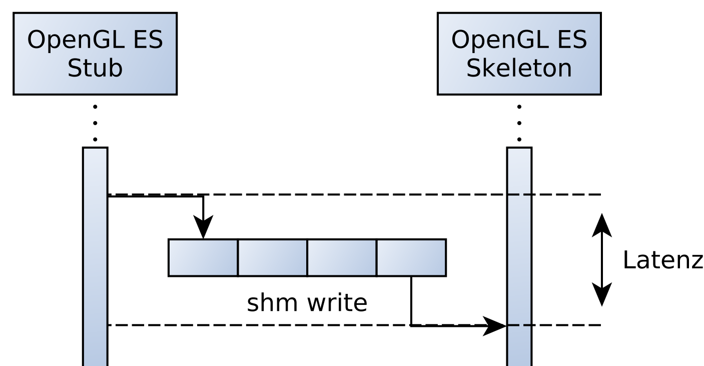



**Abbildung 5.11.**: Evaluation Setup

Nachfolgender Pseudocode zeigt wie die Messung konkret für die Funktion *glDrawArrays(...)* durchgeführt wird. Dabei stoppt der Stub die Zeit vor der Parameterübertragung und übermittelt diesen Wert nach der Übertragung an den Server. Der Skeleton auf Serverseite stoppt nach dem Parameterempfang die Zeit und empfängt anschließend den Zeitwert des Stubs um die Differenz zu bilden.

**Algorithmus 5.1** Messung der Latenz

```
procedure glDrawElementsSkeleton
    rcvParameters()
    currentTime = clock()
    previousTime = rcvClientClock()
    latency = currentTime - previousTime
    glDrawElements()
end procedure

procedure glDrawElementsStub
    currentTime = clock()
    sendParameters()
    sendClientClock(currentTime)
end procedure
```

**Paketumlaufzeit**

Ähnlich wie die Latenz wird auch die Paketumlaufzeit (RTT) bestimmt. Dazu wird in der Stubfunktion vor der Übertragung der Funktionsparameter und nach dem Empfang des Rückgabewertes vom Skeleton die Zeit gestoppt. Anschließend wird, wie Algorithmus 5.2 aufzeigt, aus diesen Werten die Differenz gebildet.

**Algorithmus 5.2** Messung der Paketumlaufzeit

```
procedure glGetErrorSkeleton
    rcvParameters()
    glGetError()
    sendResult()
end procedure

procedure glGetErrorStub
    currentTime = clock()
    sendParameters()
    rcvResult()
    previousTime = clock()
end procedure
```

### Framerate

Zum Messen der Framerate wird die Zeit zwischen zwei konsekutiven *eglSwapBuffers(...)* Funktionen verwendet. Alle Messergebnisse werden in den Einheiten FPS (Frames pro Sekunde) oder der Framezeit (in Sekunden) notiert. Das ermitteln der Framezeit wird in Algorithmus 5.3 gezeigt. Daraus wird die FPS wie folgt berechnet:

$$FPS = 1/Framezeit \tag{5.1}$$

**Algorithmus 5.3** Messung der Framerate

```
procedure eglSwapBuffersSkeleton
    rcvParameters()
    eglSwapBuffers()
    currentTime = clock()
    frameTime = currentTime - previousTime
    previousTime = currentTime
end procedure
```

### Shared Memory Größe

Die Breite der Shared Memory Kanäle wird statisch zugewiesen und der Einfluss dieses Wertes auf die anderen Parameter ermittelt.

### Arbeitsspeicher

Der benötigte Arbeitsspeicher wird mit Hilfe des Tools *top* überwacht.

## 5.2. Leistungsfähigkeit

### 5.2.1. Zero-Copy Kommunikation

#### Latenz

Wie bei der Konzeptbeschreibung bereits erwähnt, wird eine Optimierung der Übertragungszeit mit dem Zero-Copy Verfahren erwartet. Eine Messung der Paketumlaufzeit (siehe Abbildung 5.12) bestätigt diese Vermutung. Man sieht hier zum einen, dass die Übertragungszeit deutlich zunimmt sobald Datenmengen der Größenordnung des Kanals transportiert werden müssen, zum anderen ist aber auch zu erkennen, dass bei großen Datenmengen das Zero-Copy Verfahren etwa doppelt so schnell Daten transportiert (0.022 vs 0.014 Sekunden RTT bei 2 MB Datenpaketen).

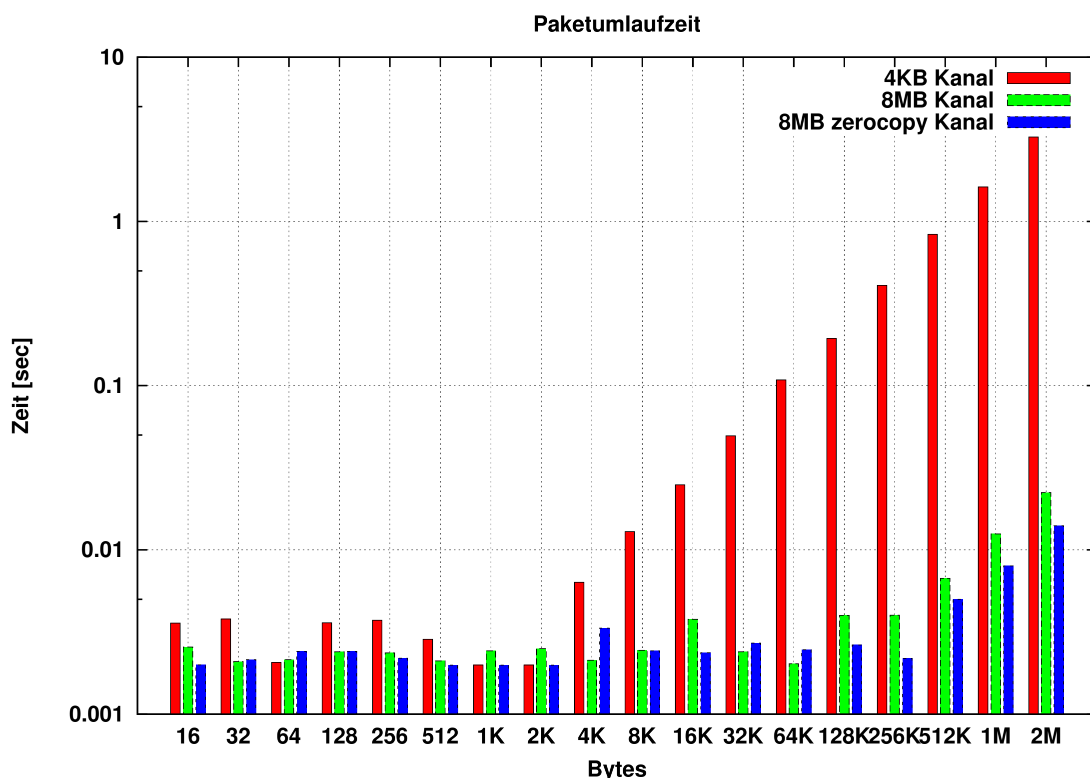
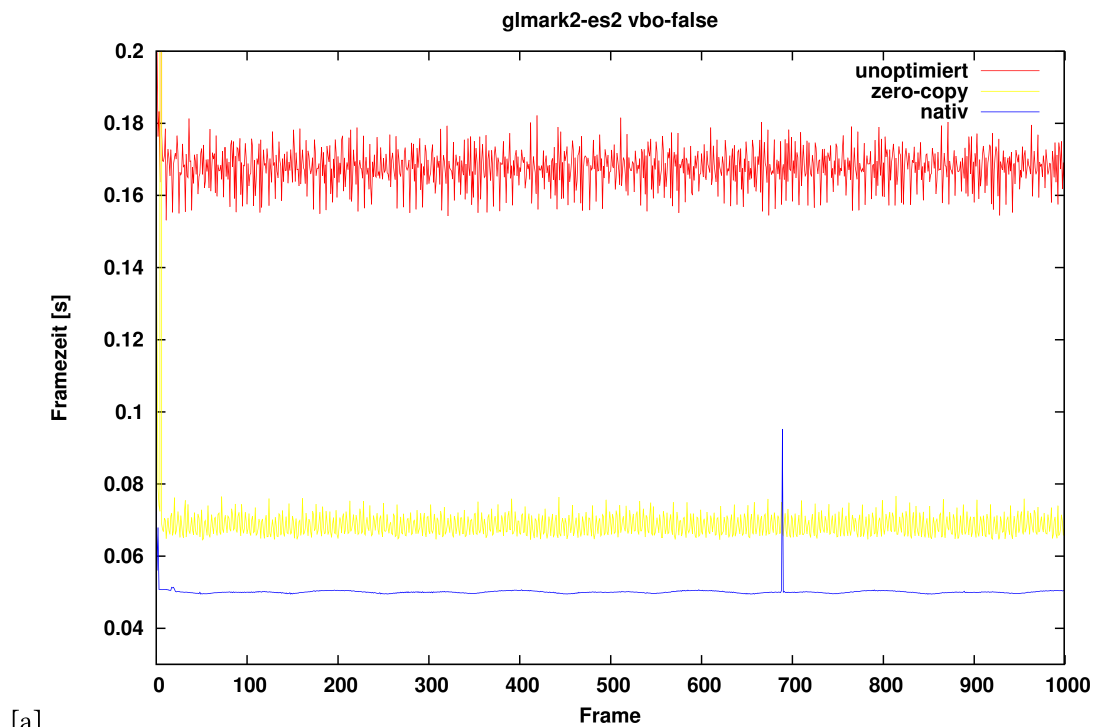


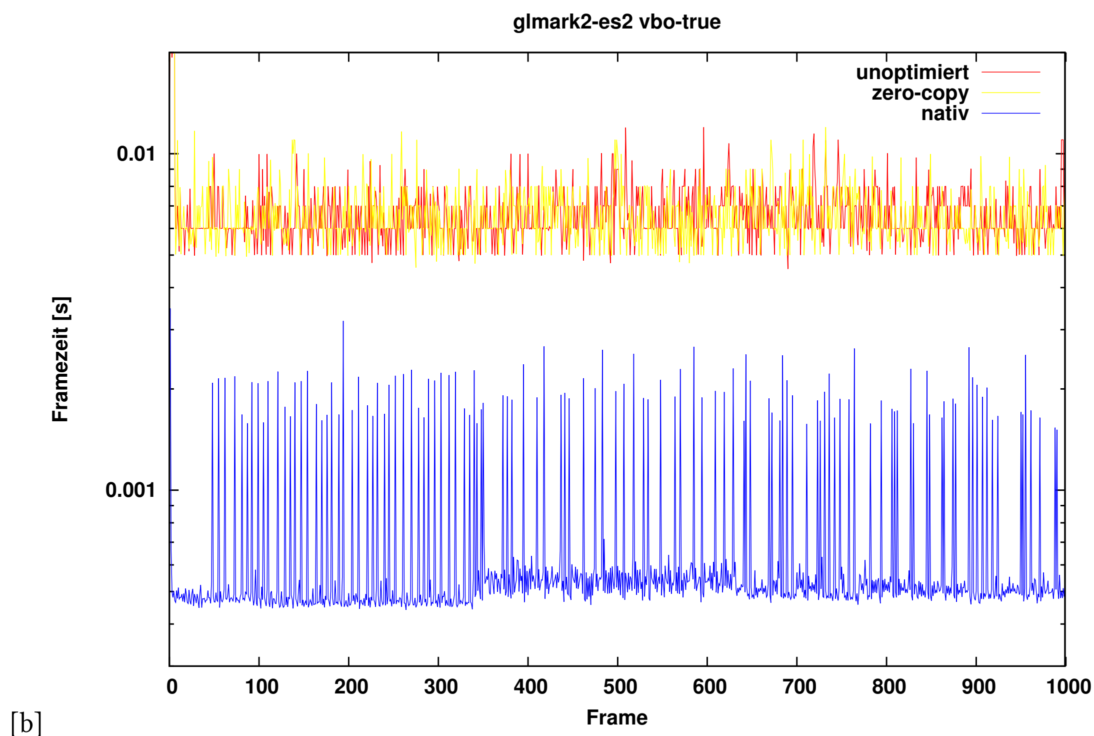

**Abbildung 5.12.**: Zero-Copy: Vergleich von Paketumlaufzeiten

#### Framerate

Die Auswirkungen des Zero-Copy Verfahrens auf die Framerate wird mit Hilfe der Szenarien 1, 2 und 5 evaluiert. Das Verfahren wurde hierbei für die Funktionen *glDrawArrays* und *glDrawElements* implementiert. Die Ergebnisse des *glmark2* Benchmarks sind in Abbildung 5.13 zu sehen. In diesen Diagrammen sind die jeweiligen Framezeiten der ersten 1000 Frames auf der x-Achse abgebildet. Dabei ist zu erkennen, dass auf der einen Seite eine deutliche Reduzierung der Framezeit (y-Achse) im Fall *a)* erfolgt ist. Auf der anderen Seite (Fall *b)*) ist jedoch keine Optimierung erkenntlich.




[a]




[b]

**Abbildung 5.13.:** a) glmark2 Modell *buddha* ohne Vertex Buffer Objects (Szenario 1)
b) glmark2 Modell *buddha* mit Vertex Buffer Objects (Szenario 2)

Tabelle 5.3 zeigt aus Vollständigkeitsgründen die Durchschnittswerte aus der vorhergehenden Messung. Diese zeigen eine Optimierung von etwa 240% beim Szenario das keine VBOs einsetzt.

| | **glmark2 ohne VBOs** | **glmark2 mit VBOs** |
|---|---|---|
| **nativ** | 0.050069 (FPS=19.97) | 0.0060062 (FPS=166.49) |
| **unoptimiert** | 0.167706 (FPS=5.96) | 0.0071727 (FPS=157.39) |
| **zero-copy** | 0.068566 (FPS=14.58) | 0.0072329 (FPS=156.74) |

**Tabelle 5.3.:** Zero-Copy: Durchschnittliche Framezeiten bei Szenario 1 + 2

Auch im Fall der Benchmark-Anwendung Quake 3 ist ebenfalls eine deutliche Verbesserung erkenntlich. Unter Hinzunahme der Zero-Copy Optimierung kann eine nahezu native Performanz erzielt werden (siehe Abbildung 5.14). So beträgt hier die native Framerate 27.7 FPS im Vergleich zu 27.4 FPS im Fall „Forwarding zero-copy“. Im Vergleich zur unoptimierten Lösung mit einem 8MB großen Ringpuffer, kann hier eine Verbesserung der Framerate um 9,6% festgestellt werden.

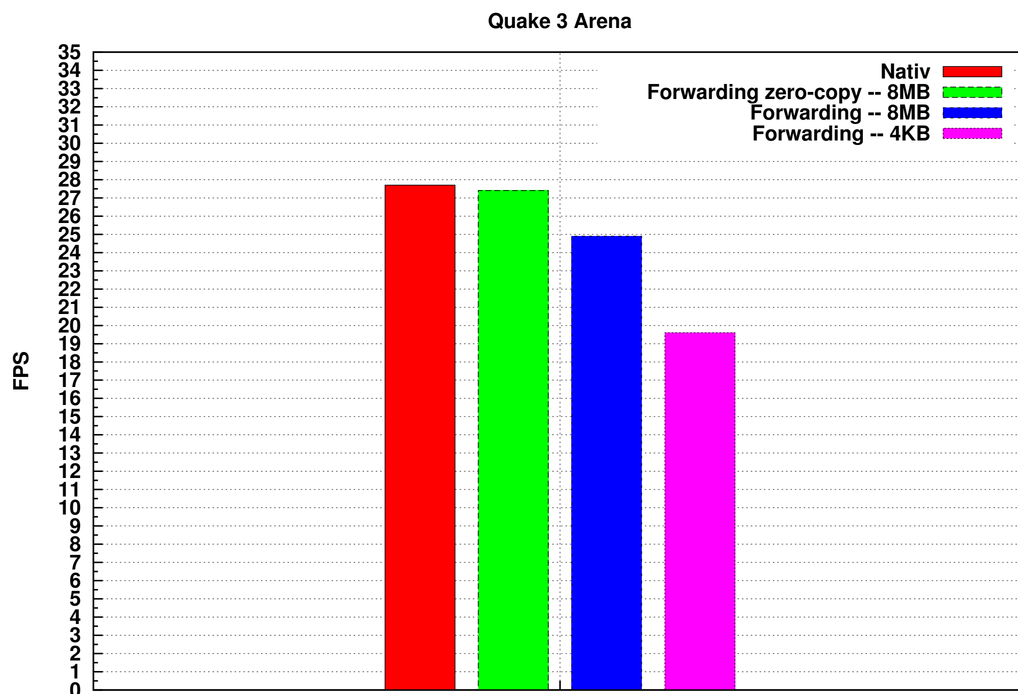



**Abbildung 5.14.:** Zero-Copy: Performanzvergleich in Quake 3 Arena (Szenario 5)

Die Erklärung für die ermittelten Ergebnisse bei *glmark2* mit dem Modell *buddha* sind in Abbildung 5.15 verdeutlicht. Während beim ersten Szenario keine *Vertex Buffer Objects* (vbo=false) verwendet werden, kommen diese beim Zweiten zum Einsatz. Die Konsequenz ist, dass bei jedem Aufruf von *glDrawArrays* im ersten Fall wesentlich mehr Vertexdaten übertragen werden müssen als im zweiten

Fall. In der zweiten Messung werden diese zu Beginn dem Server übermittelt und auf Serverseite im Grafikspeicher gepuffert. Dadurch werden in diesem Fall bei der einen Messung lediglich 96 Bytes und bei der anderen Messung etwa 7 MByte an Vertexdaten pro Frame übertragen (Vergleich Abbildung 5.15), wodurch das Zero-Copy Verfahren viel performanter wird.

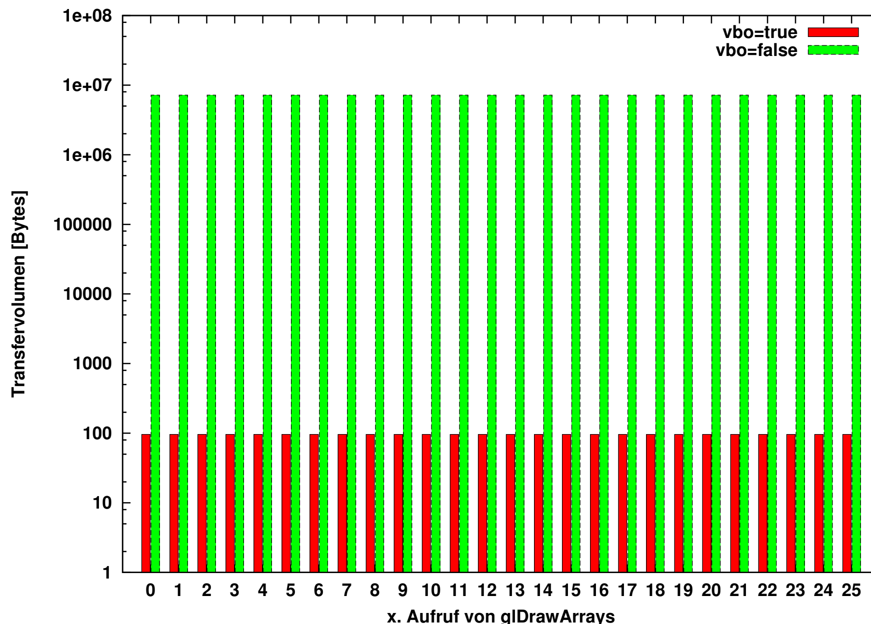



**Abbildung 5.15.**: Zero-Copy: Transfervolumen von *glDrawArrays(...)*

### Shared Memory Größe

Aus der Abbildung 5.15 ist zu schließen, dass im Szenario 1 keine Kommunikation über das Zero-Copy Verfahren erfolgen kann, falls für die Kommunikation weniger als 7MB zur Verfügung stehen. Somit ist der Nachteil des Verfahrens, dass die Größe des Shared Memory Kanals an die zu übertragende Datenmenge eines *glDrawArrays* Befehls angepasst werden muss:

$$size(shm_channel) > max(\#Bytes(glDrawArrays)) \quad (5.2)$$

### Arbeitsspeicher

Aufgrund der aktuellen Implementierung erhöht sich bei dieser Optimierung der Arbeitsspeicherbedarf um die Größe des Zero-Copy Shared Memories, da die bisherige Implementierung einen zusätzlichen Kanal neben dem bisherigen einführt.

### Fazit

Es konnte gezeigt werden, dass die Optimierung die Übertragungszeit von großen Datenpaketen deutlich reduzieren kann. Dadurch wird insbesondere bei kommunikationsintensiven GLES Anwendungen die Framerate verbessert, jedoch nimmt man auf Grund der aktuellen Implementierung einen erhöhten Arbeitsspeicherbedarf auf Client- und Serverseite in Kauf.

### 5.2.2. Asynchronisierung

**Latenz**
Die Asynchronisierung der Anwendungen hat keinen direkten Einfluss auf die Latenz. Jedoch verschwindet der Einfluss der Latenz vollständig im Falle einer komplett asynchronen Ausführung der Anwendung (Vergleich Spalte "glmark2 mit VBOs‘"der Tabelle 5.4).

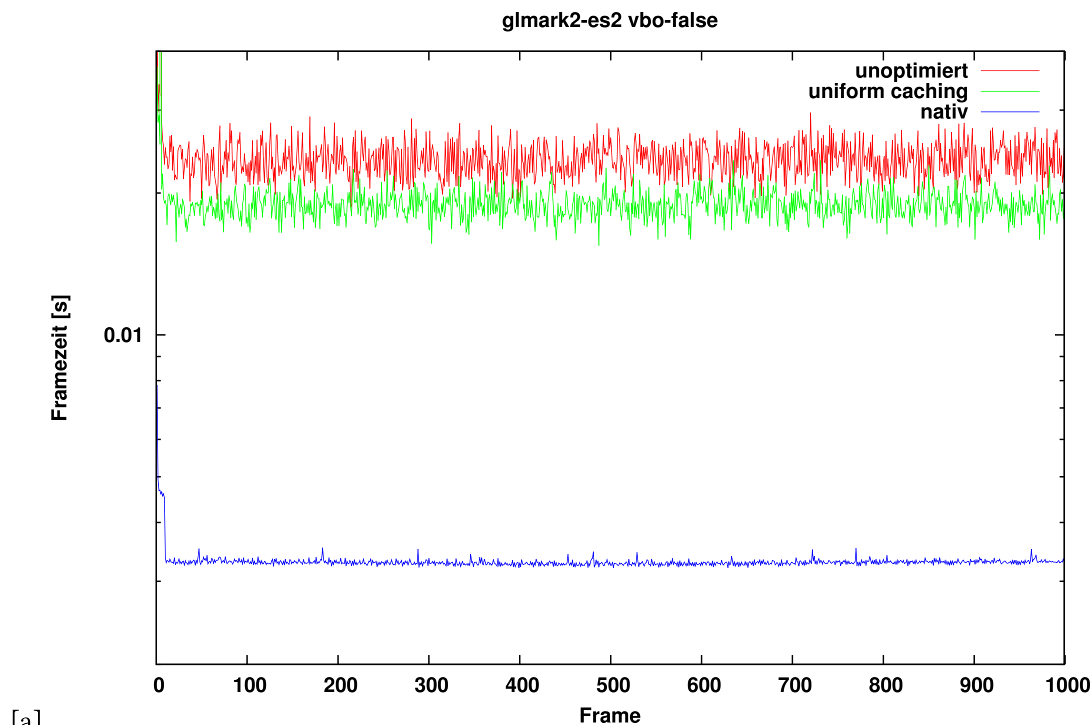
**Framerate**
Die Zustandspufferung (im folgenden *uniform caching*) wird bei dieser Auswertung für die Funktion *glGetUniformLocation* implementiert und mit Hilfe der Szenarien 3 und 4 evaluiert. Dadurch, dass der Zustandspuffer im Falle eines „Cache Hits“ eine Paketumlaufzeit einspart und durch die Asynchronisierung die Leerlaufzeit auf Serverseite reduziert wird hier ein deutlicher Performanzgewinn erzielt. Dies zeigt Abbildung 5.16 und die dort ermittelten Durchschnittswerte in Tabelle 5.4. So wird beim Szenario 3 eine Verbesserung um 27,8% der Framerate und bei Szenario 4 sogar eine Optimierung um 158% erreicht.

| | **glmark2 ohne VBOs** | **glmark2 mit VBOs** |
|---|---|---|
| **nativ** | 0.003360 (FPS=297.58) | 0.0006762 (FPS=1478.85) |
| **unoptimiert** | 0.024108 (FPS=41.48) | 0.0071727 (FPS=139.41) |
| **uniform caching** | 0.019475 (FPS=51.35) | 0.0027799 (FPS=359.73) |
| **asynchron** | 0.015755 (FPS=63.47) | 0.0007719 (FPS=1295.43) |

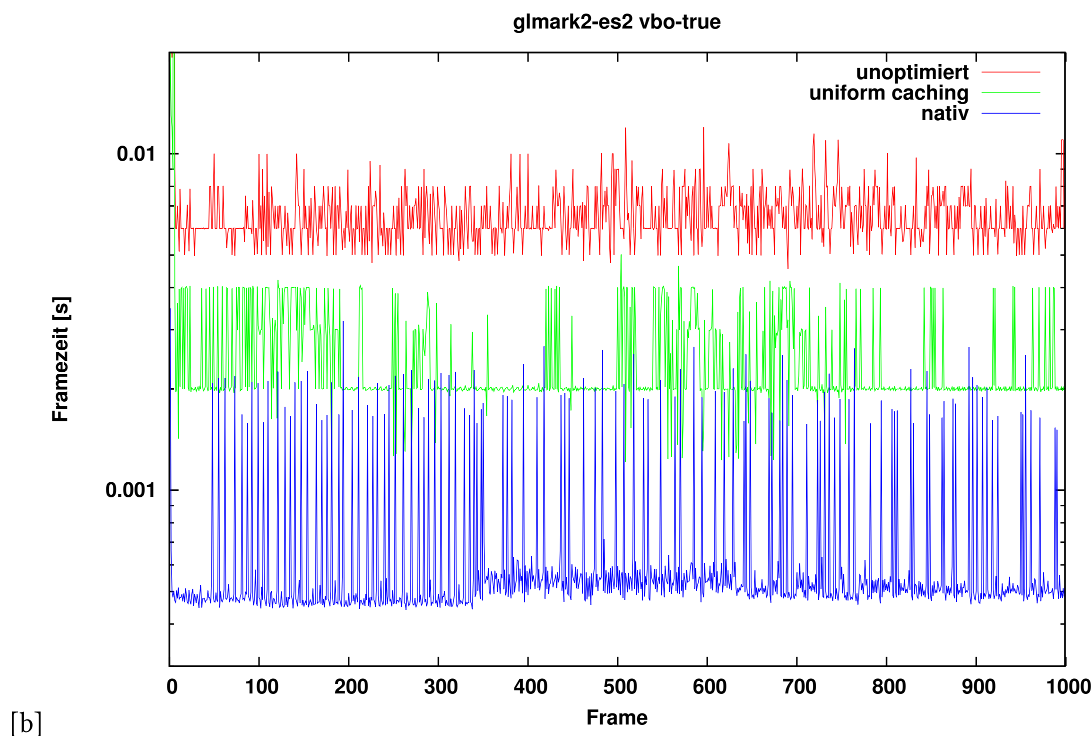
**Tabelle 5.4.**: Uniform caching: Durchschnittliche Framezeiten bei Szenario 3 + 4

Aktiviert man außerdem noch das *asynchrone eglSwapBuffers* wird die Anwendung komplett asynchron. Wirft man beispielsweise einen Blick auf die Ergebnisse mit VBOs, stellt man fest, dass durch diesen Ansatz pro Sekunde etwa 3000 synchrone Befehle (1500**eglSwapBuffers* + 1500**glGetUniformLocation*) asynchron ausgeführt werden und somit extrem viel Latenz eingespart werden kann.

**Fazit**
Die Ergebnisse in diesem Abschnitt haben deutlich gemacht, dass eine asynchrone GLES Anwendung viel performanter ist. Während der Zustandspuffer semantikerhaltend ist, wird durch das *asynchrone eglSwapBuffers* die Semantik von OpenGL ES verändert. Ein weiterer Nachteil der hier anzuführen ist, ist die Tatsache, dass eine komplett asynchrone Anwendungsausführung zu Problemen bei Echtzeitanwendungen führen kann. Denkt man beispielsweise an eine Geschwindigkeitsanzeige und wählt einen viel zu großen Shared Memory Puffer, so kann es passieren, dass der Frame mit dem aktuellen Geschwindigkeitswert stark verspätet gezeichnet wird da der Server alle vorhergehenden Frames im Puffer zunächst zeichnen muss.




[a]




[b]

**Abbildung 5.16.:** a) glmark2 Modell *horse* ohne Vertex Buffer Objects (Szenario 3)
b) glmark2 Modell *horse* mit Vertex Buffer Objects (Szenario 4)

## 5.3. Skalierbarkeit

Im Folgenden wird die Skalierbarkeit der Forwarding Middleware für die einzelnen Klassen von GLES Applikationen untersucht. Abschließend wird ein Mix dieser Klassen, realisiert durch ein Automotive Szenario, evaluiert.

### Skalierbarkeit optimaler Anwendungen

Zunächst wird hier das Szenario 1 untersucht. Dabei werden mehrere Instanzen von glmark2 mit dem Modell *buddha* gleichzeitig auf den zwei Gastsystemen gestartet. Das Ergebnis dieser Untersuchung ist in Abbildung 5.17 zu sehen. Aus diesem Diagramm lässt sich schließen, dass das Skalierungsverhalten der Middleware bei dieser Klasse von Anwendungen ähnlich der nativen Ausführung ist. Dies liegt daran, dass hier die Grafikkarte der Engpass ist und die Kommunikation dadurch kaum Einfluss auf die Performanz hat. Wie auch schon die Untersuchung der Zero-Copy Optimierung gezeigt hat, ist das Verfahren in dieser Anwendungsklasse nicht gewinbringend. Jedoch zeigt die Abbildung 5.17, dass die Middleware in diesem Fall bereits optimal ist (zwischen 93% und 98% der nativen Leistung in Abbildung 5.17).

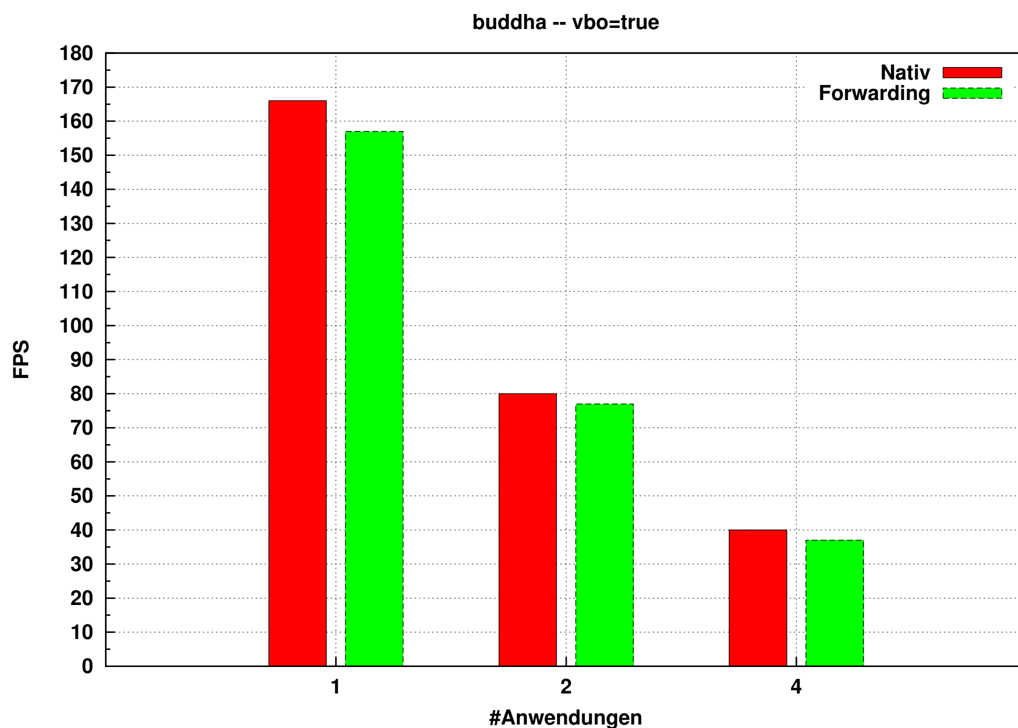



**Abbildung 5.17.:** Middleware Skalierbarkeit: Applikationsklasse 1

### Skalierbarkeit kommunikationsintensiver Anwendungen

Als nächstes wird die Skalierbarkeit unter Einbezug der Applikationsklasse 2 untersucht. Wie Abbildung 5.18 aufzeigt, ist hier die Skalierung anders verglichen mit der vorhergehenden Untersuchung. Die Abbildung und die Tabelle 5.5 veranschaulichen, dass die **native** Framerate mehr als halbiert wird, wenn die Anwendungszahl verdoppelt wird. Mit Blick auf das **Forwarding** ist aber zu sehen, dass die Framerate viel schwächer sinkt und sich der nativen immer weiter annähert. Die Erklärung für das Verhalten ist eine Reduzierung der Leerlaufzeiten des Servers, da mit einer größeren Menge an Anwendungen es immer unwahrscheinlicher wird, dass zu einem Zeitpunkt alle Befehlspuffer leer sind.

| # Anwendungen | **Nativ [FPS]** | **Forwarding opt. [% Nativ]** | **Forwarding [% Nativ]** |
|---|---|---|---|
| 1 | 363 | 41.6 | 6.3 |
| 2 | 160 | 73.0 | 15.4 |
| 4 | 60 | 78.3 | 26.6 |
| 8 | 28 | 85.7 | 53.6 |

**Tabelle 5.5.**: Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 2

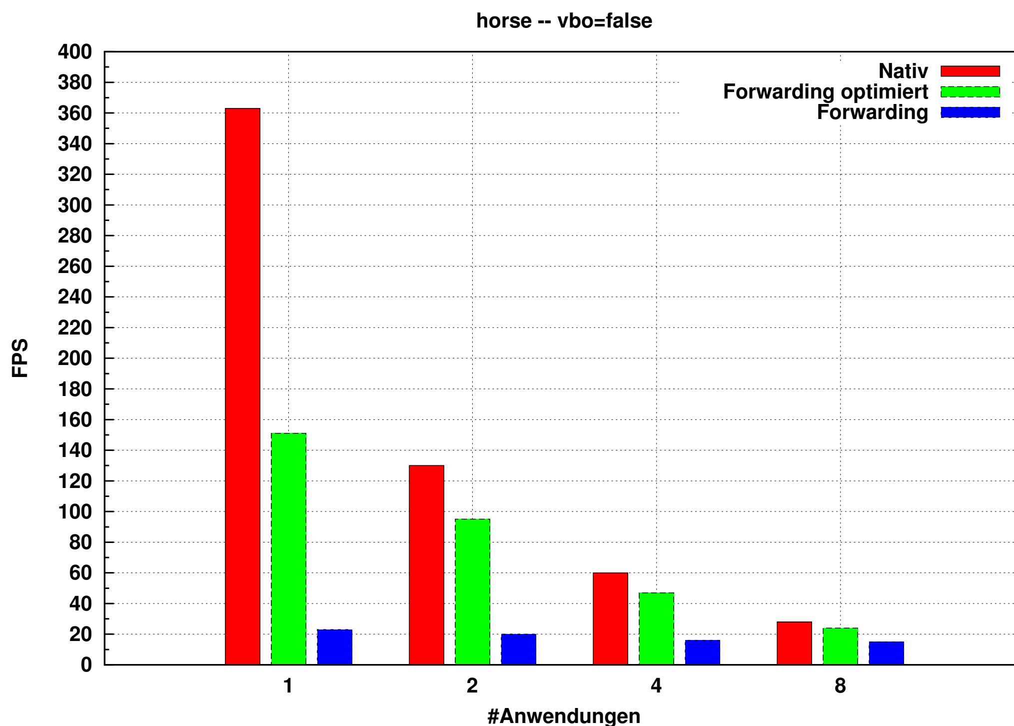



**Abbildung 5.18.**: Middleware Skalierbarkeit: Applikationsklasse 2

### Skalierbarkeit kommunikations- und renderingsintensiver Anwendungen

Bei dieser Messung ist zu erkennen, dass sich die Skalierbarkeit ähnlich zu der nativen Verhält. Das liegt daran, dass in diesem Fall die Synchronität von GLES Funktionen einen geringen Einfluss auf die Leistung hat. Viel eher ist der Kommunikationskanal hier der Flaschenhals und im Vergleich zur nativen Framerate werden hier deshalb auch nur bis zu 80% der nativen Leistung erreicht (siehe Tabelle 5.6).

| # Anwendungen | Nativ [FPS] | Forwarding opt. [% Nativ] | Forwarding [% Nativ] |
|---|---|---|---|
| 1 | 20 | 73.0 | 30.0 |
| 2 | 11 | 80.0 | 32.7 |
| 4 | 5.5 | 76.4 | 34.5 |

**Tabelle 5.6.:** Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 3

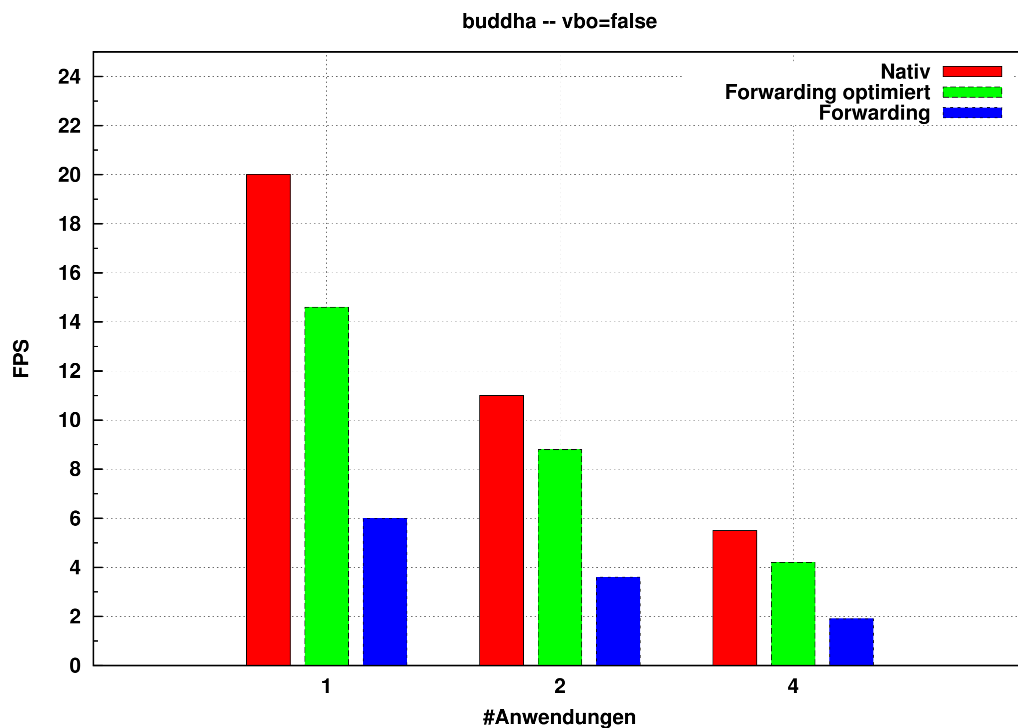



**Abbildung 5.19.:** Middleware Skalierbarkeit: Applikationsklasse 3

### Skalierbarkeit kommunikations- und renderingsschwacher Anwendungen

Die Applikationsklasse 4 ist ein extremes Beispiel dafür, dass der synchrone Funktionen mit zunehmender Anzahl an Anwendungen kompensiert werden können. Wirft man einen Blick auf die nachfolgende Tabelle, so erkennt man, dass etwa 1500 synchrone Befehle pro Sekunde ausgeführt werden, sodass der Server genauso oft auf ein erneutes Befüllen des Befehlspuffers im Fall des Forwardings warten muss. Dies hat zur Folge, dass lediglich 29.6% der nativen Framerate erreicht werden. Werden jedoch 8 Instanzen dieser Anwendung gestartet, so erreicht man 92.2% der nativen Framerate wie der Tabelle entnommen werden kann. Aus Vollständigkeitsgründen visualisiert Abbildung 5.20 die Annäherung der FPS aneinander.

| # Anwendungen | Nativ [FPS] | Forwarding opt. [% Nativ] | Forwarding [% Nativ] |
|---|---|---|---|
| 1 | 1488 | 29.6 | 9.3 |
| 2 | 614 | 45.6 | 21.0 |
| 4 | 240 | 73.3 | 46.3 |
| 8 | 90 | 92.2 | 82.2 |

**Tabelle 5.7.**: Prozentualer Vergleich der Skalierbarkeit: Applikationsklasse 4

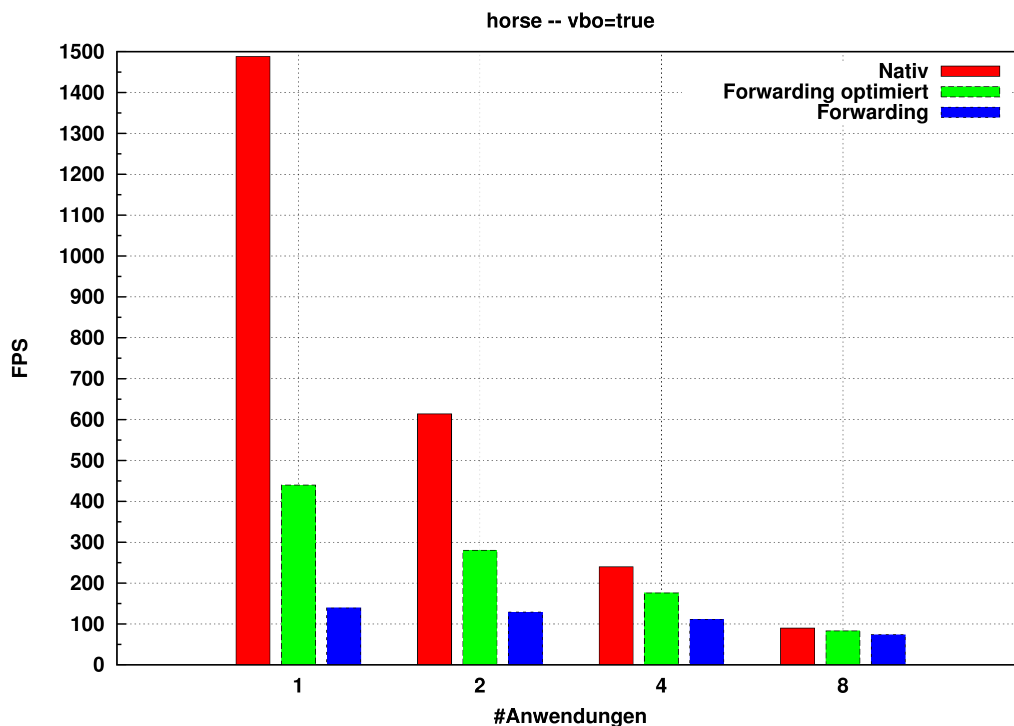



**Abbildung 5.20.**: Middleware Skalierbarkeit: Applikationsklasse 4

### 5.3.1. Shared Memory Management

Um die Speicherverteilung zu evaluieren wird hier das *Automotive Szenario* eingesetzt. Tabelle 5.8 zeigt in der Spalte „Ziel“ die Referenz Framerate, die als optimale Framerate der Anwendungen gesehen wird. Diese Werte wurden in einer Simulation mit ausreichend statischem Speicher ermittelt (1MB pro Anwendung). Alle nachfolgenden Messungen wurden jeweils über 40.000 Frames durchgeführt.

Wird die Strategie zur optimalen Verteilung bei ausreichendem Speicher genutzt, so erhält man die Ergebnisse, die die Spalten 3 und 4 zeigen. Es wird hier eine optimale Framerate erzielt bei deutlich reduziertem Speicherbedarf. So werden in diesem Szenario nur etwa 1.25MB an Shared Memory benötigt und es wird eine durchschnittliche Framerate von 14.0 FPS erzielt (Unter Hinzunahme der Zero-Copy Optimierung sind es hier durchschnittlich 16.4 FPS).

| # Anwendung | **Bezeichnung** | **ShM Größe** | **Forwarding [FPS]** | **Ziel [FPS]** |
|---|---|---|---|---|
| 1 | glmark2 (desktop) | 4KB | 23.7 | 23.7 |
| 2 | Quake 3 | 512KB | 5.8 | 5.8 |
| 3 | glmark2 (horse ohne VBO) | 512KB | 8.5 | 8.4 |
| 4 | glmark2 (buddha mit VBO) | 1KB | 33.7 | 33.7 |
| 5 | es2gears | 32KB | 12.1 | 12.0 |
| 6 | es2gears | 32KB | 12.0 | 12.0 |
| 7 | es2gears | 32KB | 12.0 | 11.9 |
| 8 | es2gears | 32KB | 12.0 | 12.0 |
| 9 | es2gears | 32KB | 12.1 | 12.1 |
| 10 | es2gears | 32KB | 11.9 | 12.0 |
| 11 | es2gears | 32KB | 12.0 | 12.0 |
| 12 | es2gears | 32KB | 12.0 | 12.0 |

**Tabelle 5.8.**: Evaluation: Optimale Speicherverteilung bei ausreichender Verfügbarkeit

Wird der ermittelte Speicherbedarf (1.25 MB) gleichmäßig auf alle Anwendungen verteilt (Tabelle 5.9), so erhält man bei gleichem Speicherbedarf eine Framerate von 13.9 FPS. Es ist dabei zu erkennen, dass Anwendungen denen nun weniger Speicher zur Verfügung steht, etwa 5-10 % an Leistung verlieren. Somit wird deutlich, dass eine Speicherverwaltungseinheit durchaus Sinn macht.

#### Equal-Framerate Strategie

Um die Speicherverteilungsstrategien bei begrenztem Shared Memory zu evaluieren, wird das vorhandene Shared Memory auf 120KB reduziert. Die Ergebnisse der Simulation mit der Equal-Framerate Strategie unter dieser Bedingung sind in Tabelle 5.10 dargestellt.

| # Anwendung | Bezeichnung | ShM Größe | Forwarding [FPS] | Nativ [FPS |
|---|---|---|---|---|
| 1 | glmark2 (desktop) | 100KB | 23.7 | 23.7 |
| 2 | Quake 3 | 100KB | 5.6 | 5.8 |
| 3 | glmark2 (horse ohne VBO) | 100KB | 7.7 | 8.4 |
| 4 | glmark2 (buddha mit VBO) | 100KB | 33.7 | 33.7 |
| 5 | es2gears | 100KB | 12.0 | 12.0 |
| 6 | es2gears | 100KB | 12.0 | 12.0 |
| 7 | es2gears | 100KB | 12.0 | 11.9 |
| 8 | es2gears | 100KB | 12.0 | 12.0 |
| 9 | es2gears | 100KB | 12.1 | 12.1 |
| 10 | es2gears | 100KB | 12.0 | 12.0 |
| 11 | es2gears | 100KB | 11.9 | 12.0 |
| 12 | es2gears | 100KB | 12.0 | 12.0 |

**Tabelle 5.9.**: Evaluation: Gleichverteilung des ermittelten Speicherbedarfs

Die Tabelle 5.10 zeigt, dass mit dieser Strategie alle Anwendungen mit höherer Framerate ihren Speicher den schwächeren Anwendungen (hier Quake 3 und glmark2-horse) abgegeben haben. Dadurch haben sich die Frameraten der Anwendungen einander angenähert. Die einzige Ausnahme bildet hierbei die Anwendung 4. Diese hat eine optimale Kanalgröße (1KB) und dadurch ,dass im Schnitt alle Anwendungen eine geringere Framerate haben, kann Anwendung 4 mehr GPU Zeit in Anspruch nehmen. Die durchschnittliche Framerate in diesem Szenario beträgt 13.4 FPS.

| # Anwendung | Bezeichnung | ShM Größe | Forwarding [FPS] | Nativ [FPS |
|---|---|---|---|---|
| 1 | glmark2 (desktop) | 1KB | 25.7 | 23.7 |
| 2 | Quake 3 | 64KB | 6.1 | 5.8 |
| 3 | glmark2 (horse ohne VBO) | 46KB | 6.8 | 8.4 |
| 4 | glmark2 (buddha mit VBO) | 1KB | 48.4 | 33.7 |
| 5 | es2gears | 1KB | 9.2 | 12.0 |
| 6 | es2gears | 1KB | 9.2 | 12.0 |
| 7 | es2gears | 1KB | 9.3 | 11.9 |
| 8 | es2gears | 1KB | 9.3 | 12.0 |
| 9 | es2gears | 1KB | 9.3 | 12.1 |
| 10 | es2gears | 1KB | 9.2 | 12.0 |
| 11 | es2gears | 1KB | 9.3 | 12.0 |
| 12 | es2gears | 1KB | 9.2 | 12.0 |

**Tabelle 5.10.**: Evaluation: Equal-Framerate Speicherverteilung

**Optimal-Framerate Strategie**

Tabelle 5.11 zeigt die Simulationsergebnisse der Optimal-Framerate Speicherverteilung. Es ist zu erkennen, dass die Anwendungen 5-12 im Vergleich zur vorherigen Simulation nicht verhungern, da ihnen nicht der komplette Speicher entzogen wird. Des weiteren zeigt sich, dass der Anwendung 3 mehr Speicher zugewiesen wird als Anwendung 1, da hier ein größerer Gewinn erwartet wird. Im Durchschnitt liegt die Framerate bei dieser Speicherverteilungsstrategie bei 14.6 FPS und somit etwa 10% über dem Wert der Equal-Framerate Strategie. Außerdem können 10 der 12 Anwendungen ihre optimale Framerate erzielen.

| # Anwendung | **Bezeichnung** | **ShM Größe** | **Forwarding [FPS]** | **Nativ [FPS** |
|---|---|---|---|---|
| 1 | glmark2 (desktop) | 1KB | 23.7 | 23.7 |
| 2 | Quake 3 | 46KB | 5.0 | 5.8 |
| 3 | glmark2 (horse ohne VBO) | 56KB | 6.5 | 8.4 |
| 4 | glmark2 (buddha mit VBO) | 1KB | 41.5 | 33.7 |
| 5 | es2gears | 2KB | 12.2 | 12.0 |
| 6 | es2gears | 2KB | 12.2 | 12.0 |
| 7 | es2gears | 2KB | 12.2 | 11.9 |
| 8 | es2gears | 2KB | 12.2 | 12.0 |
| 9 | es2gears | 2KB | 12.1 | 12.1 |
| 10 | es2gears | 2KB | 12.1 | 12.0 |
| 11 | es2gears | 2KB | 12.1 | 12.0 |
| 12 | es2gears | 2KB | 12.1 | 12.0 |

**Tabelle 5.11.**: Evaluation: Optimal-Framerate Speicherverteilung

## 5.4. Überblick über die Ergebnisse

Es konnte gezeigt werden, dass unter bestimmten Umständen die vorgestellten Optimierungskonzepte hinzugeschalten werden können um die Framerate einzelner oder einer Menge von Applikationen zu verbessern. Einflüsse der Optimierungen auf einzelne Anwendungen sind in der Tabelle 5.12 zusammengefasst. So ist hier verdeutlicht, dass das Zero-Copy Verfahren bei kommunikationsintensiven Grafikanwendungen sehr gute Ergebnisse erzielt (Vergleich Szenario 1 + 5). Der Grund hierfür ist eine stark verkürzte Übertragungszeit der Daten. Werden geringe Mengen an Daten übertragen (Szenario 2) so kann über diesen Ansatz keine Optimierung erfolgen. Nutzen Anwendungen hingegen Zustandsabfragen, so kann die Framerate beträchtlich verbessert werden. Die Szenarien 3 + 4 in der Tabelle veranschaulichen diese Tatsache.

| Szenario | Seite(n) | Zero-Copy | Zustandspuffer | asynch. eglSwapBuffers |
|---|---|---|---|---|
| 1 | 55 | 244.6% | | |
| 2 | 55 | | | |
| 3 | 59 | | 27.8% | 53.0% |
| 4 | 59 | | 158.0% | 929.2% |
| 5 | 56 | 9.6% | | |

**Tabelle 5.12.**: Überblick: Messergebnisse der Leistungsfähigkeit

Hinsichtlich der Skalierbarkeit der GLES / EGL Middleware unter Einsatz einer homogenen Menge an Anwendungen, konnte bei den Szenarien 1 - 4 festgestellt werden, dass eine zunehmende Zahl von Anwendungen, die Frameraten nicht drastisch verschlechtert. Viel eher konnte gezeigt werden, dass sich die Leistung immer mehr an die Referenz (die selbe Anwendungsmenge ausgeführt ohne Middleware) annähert. Dieser Vergleich der Framerate ist in der Spalte „FPS vs Referenz“ der Tabelle 5.13 dargestellt. Die letzte Zeile zeigt ein heterogenes Szenario, bei dem die durchschnittliche Abweichung von der optimalen Framerate bei **begrenztem Shared Memory** eingetragen ist. So zeigt sich, dass durch ein effizientes Shared Memory Management, mit der *Optimal-Framerate* Strategie, im Durchschnitt 97% der optimalen Framerate erreicht werden konnte.

| Szenario | Seite(n) | FPS vs Referenz | Equal-Framerate | Optimal-Framerate |
|---|---|---|---|---|
| 1 | 62 | 76.4% | | |
| 2 | 60 | 98% | | |
| 3 | 61 | 85.7% | | |
| 4 | 63 | 92.2% | | |
| Automotive | 64-66 | | 83.15% | 97% |

**Tabelle 5.13.**: Überblick: Messergebnisse der Skalierbarkeit

# 6. Verwandte Arbeiten

Wer sich über Kritik ärgert, gibt zu,
dass sie verdient war.

(Tacitus)

In den folgenden Abschnitten werden Verwandte Lösungen vorgestellt. Allen gemein ist die Tatsache, dass sie auf OpenGL basieren.

## 6.1. VMGL

VMGL [26] ist eine OpenGL Virtualisierungslösung und ermöglicht Hardware-beschleunigtes Rendering für Applikationen die in einer VM ausgeführt werden. Dazu implementiert VMGL einen Stub auf Hostseite und ersetzt die OpenGL Bibliothek auf dem Gastsystem. Diese leitet OpenGL Kommandos an den Stub auf dem Host (siehe Abbildung 6.1). Dies erfolgt mit Hilfe des WireGL [27] Protokolls und baut somit auf einer verbindungsorientierten Kommunikation, über TCP beispielsweise, zwischen Client und Server auf. Daraus kann geschlossen werden, dass im Vergleich zu einer Shared Memory basierten Lösung, ein wesentlich geringerer Durchsatz erzielt wird.

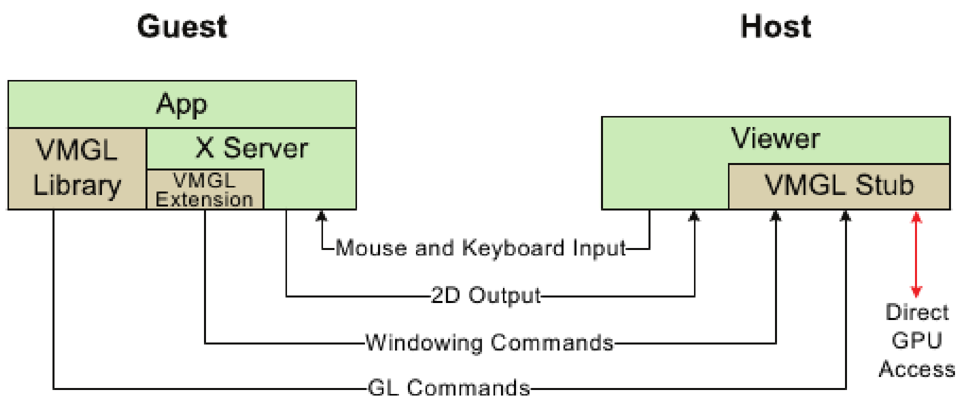



**Abbildung 6.1.**: Architektur von VMGL [26]

## 6.2. VirtGL

Auch VirtGL [28] basiert darauf die OpenGL Schnittstelle zu virtualisieren. Die Architektur (siehe Abbildung 6.2) besteht hauptsächlich aus zwei Komponenten:

- **VirtGL Interface**: Diese Komponente wird als ein Gerätetreiber im Gastsystem realisiert.
- **VirtGL Device**: Diese Komponente wird als ein virtuelles PCI Gerät im Hostsystem implementiert.

Das VirtGL Device im Host Betriebssystem ist für die Durchführung von OpenGL Kommandos zuständig. Dazu werden zwei Memory Mapped IO (MMIO) Regionen definiert und Callback Routinen, die beim Zugriff auf die Regionen aufgerufen werden, implementiert. Während der eine MMIO Bereich zur Kommunikation von OpenGL Kommandos genutzt wird, wird der zweite Bereich zum Austausch von Daten eingesetzt.

Applikationen im Gastsystem nutzen einen Gerätetreiber (VirtGL Interface) um auf die oben erwähnten MMIO Regionen zu schreiben bzw. von ihnen zu lesen. Dabei müssen sie zunächst OpenGL Parameter in den Datenbereich und anschließend einen OpenGL Funktionsidentifier in den Kommandobereich schreiben.

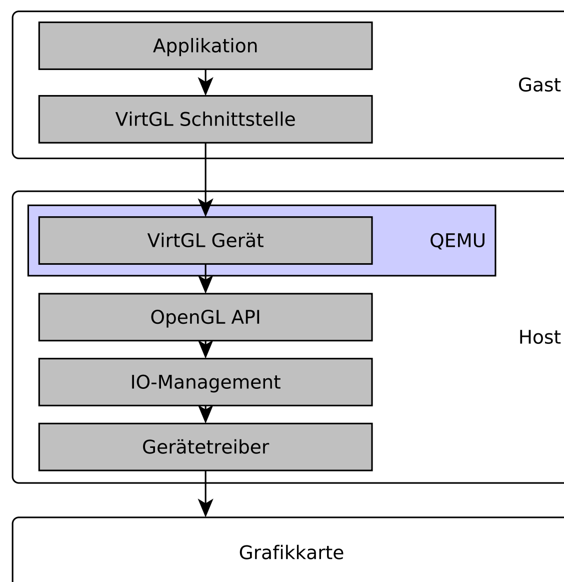



**Abbildung 6.2.**: Architektur von VirtGL [28]

Ein großer Nachteil bei diesem Ansatz ist die Einführung zweier zusätzlicher Schichten. Diese ziehen ine erhöhte Latenz mit sich und müssen bei Änderungen von OpenGL oder QEMU mit gepflegt werden.

## 6.3. BroadcastGL

BroadcastGL [29] nutzt UDP/IP Broadcasts um eine Untermenge von OpenGL Befehlen an mehrere Rendering Knoten zu verteilen. Die Master-Slave Architektur von BroadcastGL ist in Abbildung 6.3 zu sehen. Dabei agiert die Applikation als Master und verschickt Byteströme (serialisierte OpenGL Befehle) an zugehörige Rendering Knoten über UDP/IP Sockets. Dabei wird ein lokaler Puffer der Größe eines UDP Pakets bereitgestellt. Ist dieser gefüllt, wird das Paket über das Netzwerk verschickt. Die unabhängigen Rendering Knoten konvertieren diesen Bytestrom zurück in OpenGL Befehle. Für eventuelle Rückgabewerte besitzt jeder Slave eine TCP/IP Verbindung zu der Anwendung.

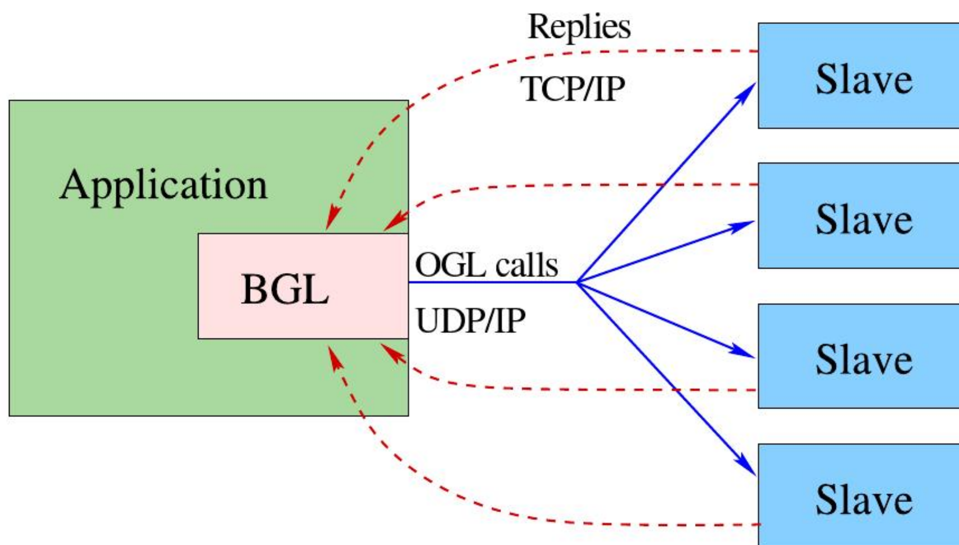



**Abbildung 6.3.**: Architektur von BroadcastGL [29]

Eine in BroadcastGL eingesetzte Optimierung ist das lokale Puffern von einigen Zustandsdaten um Funktionen wie glGetIntegerv, glGetBooleanv usw. zu asynchronisieren. Dies funktioniert nach dem selben Prinzip wie der Zustandspuffer in Kapitel 4.3. So liefert *glGetFloatv(GL_COLOR_CLEAR_VALUE, ¶ms)* beispielsweise aus einem lokalen Puffer die letzten Farbwerte die mit *glClearColor(. . . )* gesetzt wurden.

## 6.4. ClusterGL

Wie in Abbildung 6.4 zu sehen ist, handelt es sich bei ClusterGL [30] um eine Forwarding Lösung zur Ansteuerung von Anzeigewänden. In diesem Beispiel visualisiert ClusterGL die Anwendung OpenArena [31] auf der "Symphony Cluster display wall" [32].

**Abbildung 6.4.**: OpenArena über ClusterGL [30]

Die Architektur des Systems besteht hauptsächlich aus zwei Komponenten: Eine Client Bibliothek die auf dem Anwendungsknoten zum Einsatz kommt und ein Renderer auf den Anzeigeknoten. Die Client Bibliothek ist dafür zuständig OpenGL Befehle abzufangen und sie optimiert an die Anzeigeknoten weiterzuleiten. Die Renderer empfangen diese und führen sie lokal aus. Die Architektur ist beispielhaft für 5 Anzeigeknoten in Abbildung 6.5 illustriert.

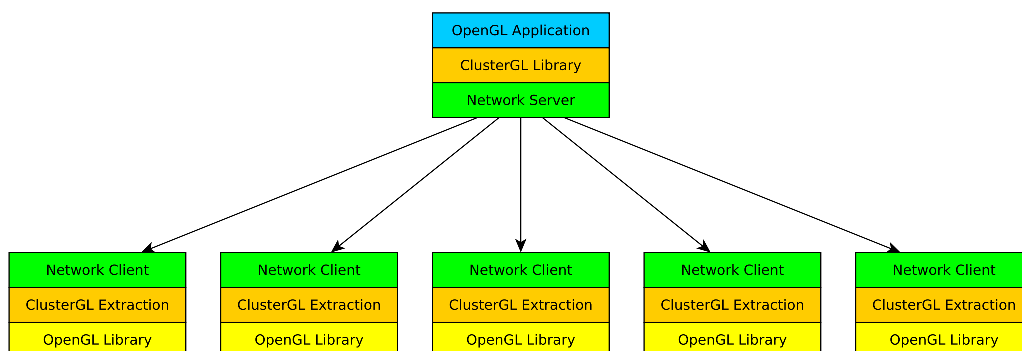



**Abbildung 6.5.**: ClusterGL-Anwendung mit 5 Anzeigeknoten [30]

Bei ClusterGL kommen 3 Optimierungen zum Einsatz:

- Frame Differencing: Hier wird versucht ähnliche Sequenzen von Befehlen in verschiedenen Frames zu erkennen.

- Compression: Hier wird das LZO [33] Verfahren eingesetzt um die zu übertragende Datenmenge zu reduzieren.
- Multi-cast: Der OpenGL Datenstrom wird mit UDP an alle Knoten verteilt.

Diese Optimierungen bringen bei ClusterGL deutliche Leistungsverbesserungen. Im Fall der GLES / EGL Middleware kann lediglich das *Frame Differencing* Optimierungen bewirken. Bei ClusterGL können Paketumlaufzeiten durch Datenkompression eingespart und die Leistung insgesamt stark verbessert werden. Bei Einsatz eines Shared Memories, würde jedoch der Kompressionsalgorithmus bzw. die CPU zum Flaschenhals werden.

# 7. Zusammenfassung und Ausblick

> Was nicht auf einer einzigen Manuskriptseite zusammengefasst werden kann, ist weder durchdacht noch entscheidungsreif.
>
> (Dwight D. Eisenhower)

In dieser Arbeit wurden einige Optimierungskonzepte zur effizienten Übertragung von Grafikbefehlen umgesetzt und ihr Optimierungspotential bewertet. Dazu wurde zunächst die zugrundeliegende Middleware beschrieben, anschließend mehrere Klassen von GLES Anwendungen identifiziert und Beispielanwendungen für die Middleware lauffähig gemacht. Unter Berücksichtigung dieser Applikationsklassen und des Einflusses von synchronen bzw. asynchronen GLES und EGL Funktionen wurden Messungen durchgeführt um die Konzepte zu evaluieren.

Dabei konnte festgestellt werden, dass bei kommunikationsintensiven Anwendungen starke Verbesserungen der Framerate durch Einschalten des Zero-Copy Verfahrens erzielt werden können. Anwendungen, bei denen die Grafikkarte der Flaschenhals ist, wurden als optimal eingestuft, da sich die jeweiligen Frameraten kaum von der der nativen Ausführung der Anwendungen unterschieden hat. Des weiteren konnte gezeigt werden, dass die Asynchronisierung von GLES und EGL Funktionen ebenfalls Verbesserungen mit sich zieht indem Paketumlaufzeiten eingespart und ein Leeren des Befehlspuffers zwischen Client und Server verhindert werden konnte.

Da die Middleware dafür konzipiert ist mehreren Anwendungen einen Zugriff auf die Grafikeinheit zu gewähren, wurden auch Skalierbarkeit-Messungen durchgeführt. Dabei konnte gezeigt werden, dass mit steigender Anzahl von Anwendungen, der Einfluss synchroner Befehle auf die Leistung zunehmend sinkt. Dieser Effekt wird damit begründet, dass serverseitige Leerlaufzeiten seltener auftreten.

Auch die Notwendigkeit einer Shared Memory Management Einheit wurde aufgezeigt und verschiedene Strategien umgesetzt und evaluiert. Unter der Bedingung, dass ausreichend Shared Memory im System vorhanden ist, wurde gezeigt wie eine optimale Verteilung des Speichers an die einzelnen Anwendungen stattfinden kann. Auch für den Fall, dass die Anwendungen mehr Speicher benötigen als zur Verfügung gestellt wird, wurden Strategien diskutiert um trotz des Mangels optimale Frameraten unter allen Anwendungen zu erreichen.

## Ausblick

Die aktuelle Implementierung des Zero-Copy Verfahrens nutzt einen separaten Shared Memory Kanal, der in den Userspace gemappt wird und benötigt daher mehr Arbeitsspeicher. Eine Verbesserung kann

hier erfolgen, wenn die komplette Kommunikation auf die neue Implementierung des Ringpuffers umgestellt wird.

Des Weiteren ist zu empfehlen, die Puffer-basierte Asynchronisierung weiter auszubauen, denn der Zustandspuffer ist derzeit zur Konzeptbestätigung nur für den Befehl *glGetUniformLocation* implementiert. Wie in Kapitel 4.3 beschrieben wurde, können auch weitere GLES Zustände clientseitig zwischengespeichert werden. Dies gilt für alle Zustände die über die Funktionen *glGetIntergv, glGetFloatv* und *glGetBooleanv* abgefragt werden können. Für diese Implementierung wird ein Codegenerator benötigt, der für jeden Parameter in der GLES Spezifikation eine Variable samt Getter und Setter Funktionen erstellt.

Auch das Shared Memory Management kann weiter verbessert werden. Zwar liefert die Annahme einer linearen Abhängigkeit zwischen der Framerate einer Anwendung und der Größe des zugewiesenen Shared Memory Kanals gute Ergebnisse, jedoch können realistischere Abhängigkeiten gewählt werden. So gleicht die Abhängigkeit wahrscheinlich eher einer Funktion mit beschränktem Wachstum mit der optimalen Framerate als obere Schranke. Auch die Annahme bezüglich der optimalen Kanalgröße kann verfeinert werden. So ist ein besseres Memory Management zu erwarten, wenn als optimale Kanalgröße das größte zu übertragende Datenpaket und nicht die übertragene Datenmenge pro Frame gewählt wird.

Ein weiterer offener Punkt ist das *Frame Differencing* aus [30]. Hier sollte untersucht werden, inwiefern das Konzept die Leistung der Middleware verbessern kann.

# A. Befehlsreferenz

Im folgende ist eine Kurzbeschreibung zu allen in der Arbeit erwähnten GLES und EGL Funktionen gegeben.

| Funktion | Beschreibung |
|---|---|
| glActiveTexture | Spezifiziert welche Textureinheit aktiviert werden soll. |
| glBindBuffer | Aktiviert ein VBO. |
| glBindTexture | Aktiviert ein Texturobjekt für eine gesetze Textureinheit. |
| glClear | Setzt alle Werte in dem Farb-, Tiefen- und Stencilpuffer zurück. |
| glClearColor | Setzt den Rücksetzwert für den Farbpuffer. |
| glDisableVertexAttribArray | Deaktiviert den Vertexarray als Input für den Vertexshader verwendet. |
| glDrawArrays | Überträgt zuvor festgelegte Vertexdaten und startet ihre Verarbeitung durch die GLES Renderpipeline. |
| glDrawElements | Ähnlich wie *glDrawArrays*. Überträgt zusätzlich Indexdaten zu den Vertexdaten. |
| glEnable | Setzt einen Zustand. |
| glEnableVertexAttribArray | Legt fest, dass ein Vertexarray als Input für den Vertexshader verwendet. werden soll. |
| glGetError | Liefert Fehlerinformationen. |
| glGetShaderiv | Liefert einen Shaderparameter. |
| glGetUniformLocation | Liefert die Speicherposition eines Uniforms. |
| glUniformMatrix4fv | Setzt einen Uniform Wert. |
| glUseProgram | Installiert ein Programobjekt. |
| glVertexAttribPointer | Definiert ein Array für Vertex Attribute. |

**Tabelle A.1.**: GLES 2.0 Befehlsreferenz

| Funktion | Beschreibung |
| --- | --- |
| eglCreateContext | Erstellt einen Anzeigekontext. |
| eglCreateWindowSurface | Erstellt eine Fensterfläche. |
| eglGetDisplay | Liefert einen Bildschirm. |
| eglMakeCurrent | Verknüpft Anzeigekontext und Fensterfläche. |
| eglSwapBuffers | Tauscht Vorder- und Hintergrundpuffer. |

**Tabelle A.2.:** EGL Befehlsreferenz

# Literaturverzeichnis


[1] Chakraborty, S. ; Lukasiewycz, M. ; Buckl, C. ; Fahmy, S. ; Chang, Naehyuck ; Park, Sangyoung ; Kim, Younghyun ; Leteinturier, P. ; Adlkofer, H.: Embedded systems and software challenges in electric vehicles. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012. – ISSN 1530–1591, S. 424–429 (Zitiert auf Seite 13)

[2] *Model S Features*. http://www.teslamotors.com/de_DE/models/features#/interior (Zitiert auf Seite 13)

[3] *Tochscreen des Tesla Model S*. http://www.blogcdn.com/www.autoblog.com/media/2012/09/38-2012-tesla-model-s-fd-1347336823.jpg (Zitiert auf Seite 13)

[4] Systems, BICCnet Innovationszirkel E.: Relevanz eines Multicore-Ökosystems für künftige Embedded Systems. (2013). http://bicc-net.de/workspace/uploads/subfeatures/downloads/positionspapier_multicore_oekosys-1323952449.pdf (Zitiert auf den Seiten 14 und 21)

[5] Gansel, Simon ; Schnitzer, Stephan ; Dürr, Frank ; Rothermel, Kurt ; Maihöfer, Christian: Towards Virtualization Concepts for Novel Automotive HMI Systems. In: *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, 2013, 193–204 (Zitiert auf Seite 14)

[6] *OpenGL - The Industry Standard for High Performance Graphics*. http://www.opengl.org/ (Zitiert auf Seite 17)

[7] *The Khronos Group Inc.* https://www.khronos.org/ (Zitiert auf Seite 17)

[8] *OpenGL 4.5 Core Profile Specification*. http://www.opengl.org/registry/doc/glspec45.core.pdf (Zitiert auf Seite 17)

[9] *OpenGL ES - The Standard for Embedded Accelerated 3D*. https://www.khronos.org/opengles/ (Zitiert auf Seite 17)

[10] *OpenGL ES 2.0*. https://www.khronos.org/opengles/2_X (Zitiert auf Seite 17)

[11] *OpenGL 2.0 Specification (October 22, 2004)*. http://www.opengl.org/registry/doc/glspec20.20041022.pdf (Zitiert auf Seite 17)

[12] Munshi, Aaftab ; Ginsburg, Dan ; Shreiner, Dave: *The OpenGL ES 2.0 programming guide*. Upper Saddle River, NJ : Addison-Wesley, 2009. – ISBN 9780321502797 0321502795 (Zitiert auf den Seiten 17, 18 und 33)

[13] *EGL - Native Platform Interface*. https://www.khronos.org/egl (Zitiert auf Seite 19)

[14] *X Fenstersystem*. http://www.x.org/wiki/Releases/7.7/ (Zitiert auf Seite 19)

[15] *Wayland.* wayland.freedesktop.org/ (Zitiert auf Seite 19)

[16] Gebhardt, Carl ; Tomlinson, Allan: Challenges for Inter Virtual Machine Communication. 2010. – Forschungsbericht (Zitiert auf Seite 21)

[17] Song, Jia ; Alves-Foss, Jim: Performance Review of Zero Copy Techniqes. In: *International Journal of Computer Science and Security (IJCSS)*, 2012, S. 256 (Zitiert auf Seite 30)

[18] *Freescale i.MX6Q Überblick.* http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q (Zitiert auf Seite 44)

[19] *glmark2: an OpenGL 2.0 and ES 2.0 benchmark.* https://github.com/glmark2/glmark2 (Zitiert auf Seite 45)

[20] *glmark2 version 2011.10.* https://launchpad.net/glmark2/2011.11/2011.10 (Zitiert auf Seite 45)

[21] *es2gears.* http://cgit.freedesktop.org/mesa/demos/tree/src/egl/opengles2/es2gears.c (Zitiert auf Seite 45)

[22] *Glxgears is not a Benchmark.* http://wiki.cchtml.com/index.php/Glxgears_is_not_a_Benchmark (Zitiert auf Seite 45)

[23] *Quake 3: Arena.* http://web.archive.org/web/20101223190257/http://www.idsoftware.com/games/quake/quake3-arena/ (Zitiert auf Seite 46)

[24] *Quake 3 demo system.* http://www.quake3world.com/q3guide/demos.html (Zitiert auf Seite 46)

[25] *Ioquake3.* http://linux-sunxi.org/Ioquake3 (Zitiert auf Seite 46)

[26] Lagar-Cavilla, H. A. ; Tolia, Niraj ; Satyanarayanan, M. ; Lara, Eyal de: VMM-independent Graphics Acceleration. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments.* New York, NY, USA : ACM, 2007 (VEE '07). – ISBN 978–1–59593–630–1, 33–43 (Zitiert auf Seite 69)

[27] Humphreys, Greg ; Eldridge, Matthew ; Buck, Ian ; Stoll, Gordan ; Everett, Matthew ; Hanrahan, Pat: WireGL: A Scalable Graphics System for Clusters. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA : ACM, 2001 (SIGGRAPH '01). – ISBN 1–58113–374–X, 129–140 (Zitiert auf Seite 69)

[28] Weggerle, A. ; Schmitt, T. ; Löw, C. ; Himpel, C. ; Schulthess, P.: *VirtGL- a lean approach to accelerated 3D graphics virtualization* (Zitiert auf Seite 70)

[29] Ilmonen, Tommi ; Reunanen, Markku ; Kontio, Petteri: Broadcast gl: An alternative method for distributing opengl api calls to multiple rendering slaves. In: *In WSCG'2005: The Journal of WSCG, volume 13, Plzen, Czech Republic*, Science Press, 2005, S. 2005 (Zitiert auf Seite 71)

[30] Neal, B. ; Hunkin, P. ; McGregor, A.: Distributed OpenGL Rendering in Network Bandwidth Constrained Environments. In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization.* Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2011 (EG PGV'11). – ISBN 978–3–905674–32–3, 21–29 (Zitiert auf den Seiten 72 und 76)

[31] *OpenArena, a community-produced deathmatch FPS*. openarena.ws (Zitiert auf Seite 72)

[32] *WAIKATO UNIVERSITY: The symphony cluster*. http://symphony.waikato.ac.nz/ (Zitiert auf Seite 72)

[33] *LZO real-time data compression library*. www.oberhumer.com/opensource/lzo (Zitiert auf Seite 73)

Alle URLs wurden zuletzt am 29. September 2014 geprüft.

# Glossar

**BroadcastGL** Viertualisierungslösung für OpenGL. 71

**ClusterGL** OpenGL Forwarding Lösung zur Ansteuerung von Anzeigewänden. 72

**Daimler AG** Hersteller von Nutzfahrzeugen und Personenkraftwagen. 14

**EGL** Schnittstelle zwischen OpenGL ES und dem nativen Fenstersystem. 19

**ELinOS** Linux Distribution von der Sysgo AG. 44

**es2gears** Benchmark Anwendung. 45

**Freescale** Halbleiterhersteller aus Texas. 44

**GLES** OpenGL ES. 17

**glmark2** Benchmark Anwendung. 45

**Khronos** Industriekonsortium, dessen Aufgabenbereich sich über die Erstellung und Verwaltung von offenen Standards Multimedia-Bereich erstreckt. 17

**LZO** Datenkompressionsalgorithmus von Lempel-Ziv-Oberhumer. 73

**MMIO** Memmory Mapped IO. 70

**OpenGL** Programmierschnittstelle zur Entwicklung von Grafikanwendungen. 17

**PCI** Serieller Bus-Standard. 70

**Quake 3** Computerspiel mit integrierter Benchmark Funktion. 46

**RTT** Paketumlaufzeit. 52

**TCP** Verbindungsorientiertes Netzwerkprotokoll. 71

**UDP** Verbindungsloses Netzwerkprotokoll. 71, 73

**VBO** Vertex Buffer Objects. 17

**VirtGL** Viertualisierungslösung für OpenGL. 70

**Wayland** Fenstersystem. 19

**X11** Fenstersystem. 19

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschrift