Fachhochschule Gießen Friedberg

Bachelorarbeit

Untersuchung zur Performance-Optimierung von 3D-Fahrzeugmodellen im Bereich der fotorealistischen Bild-generierung für Fahrerassistenzsystemtests im Automotive Bereich

vorgelegt von:	Matthias Wagner
geboren am:	05.12.1986
Studiengang:	Medieninformatik
Martrikelnummer	827539
Referent:	Prof. Dr. Cornelius Malerczyk
Korreferent:	Dipl.-Ing. René Nold
Beginn der Arbeit:	01.11.2010
Abgabe der Arbeit:	31.01.2011

Gießen, den 31.01.2011

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe bzw. unerlaubte Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Gießen, den 31.01.2011

Matthias Wagner

Danksagung

Mehrere Personen haben zum Gelingen dieser Arbeit beigetragen, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte. Für die Begleitung während des Studiums möchte ich meinem Referenten an der Hochschule Gießen-Friedberg, Prof. Dr.-Ing. Cornelius Malerczyk danken. Für die Unterstützung bei fachlichen Fragen danke ich meinem Korreferenten Dipl.-Ing. René Nold und den Kollegen bei weltenbauer. Auch möchte ich mich bei meinem Betreuer von Seiten der MBtech-Group, Dipl.-Ing. (Univ.) Florian Schmidt bedanken, der mich bei wissenschaftlichen Fragen unterstützt hat, sowie den Kollegen und studentischen Mitarbeitern vor Ort.

Außerdem danke ich meiner Verlobten Aaltje Schippers für die vielen Stunden des Korrekturlesens und die Ermutigungen während der Erstellung. Danke auch an alle Freunde, die mir stets mit Rat zur Seite stehen und auch in schwierigen Zeiten nicht im Stich lassen. Zuletzt gilt mein dank meiner Familie und ganz besonders meinen Eltern, die mir dieses Studium überhaupt erst ermöglicht haben und mich zu dem gemacht haben, der ich heute bin.

Inhaltsverzeichnis

Glossar

Batch	Ein Array von Vertices, das der Grafikkarte übergeben wird
Draw Call	Ein Renderaufruf der Grafikkarte. Dabei können Objekte mehrere Draw Call verursachen, wie beispielsweise die Berechnung von Schatteneffekten auf der Oberfläche des Objekts
GPU	(Graphics Processing Unit) Berechnungen werden auf die Prozessoren der Grafikkarte ausgelagert, um so die Hauptprozessoren des Rechners (CPU) zu entlasten
Triangle	Eine Dreiecksebene, die durch drei Vertices beschrieben ist.
Vertex	Ein Vertex beschreibt in der Geometrie eine Ecke eines Polygons
Visual-Loop-Komponente	Dient der Darstellung eines möglichst realistischen Testszenarios, das für das Test von kamerabasierten Fahrerassistenzsystemen verwendet wird

Abbildungsverzeichnis

Tabellenverzeichnis

Abstract

In modernen Fahrzeugen leisten Fahrerassistenzsysteme, wie Spurhalteassistenten oder Fußgängererkennung mittlerweile weit mehr, als den Fahrer vor kritischen Verkehrssituationen lediglich zu warnen. Durch frühzeitige Erkennung kann in Gefahrensituationen beispielsweise der Notbremsvorgang oder ein Ausweichmanöver selbständig eingeleitet werden. Fahrerassistenzsysteme werden in ausgiebigen Testfahrten erprobt, diese sind aber unter den selben Bedingungen nur schwer reproduzierbar und mit einem hohen Kosten- und Zeitfaktor versehen.

"Virtuelle Testfahrten" ermöglichen es zusätzlich, die funktionale Absicherung in einer Laborumgebung durchzuführen. Hierzu wird eine virtuelle Welt erstellt, die interaktiv und in Echtzeit reagiert. Bei der Darstellung dieser virtuellen Welten spielt die Performance eine entscheidende Rolle. Das Spannungsfeld liegt zwischen der rechenzeitaufwändigen Erzeugung fotorealistischer Bilder und der Anforderung, diese auf Standard-Hardware in Echtzeit zu liefern. Durch Verwendung einer hohen Anzahl geometrischer Objekte wird die Framerate erheblich beeinträchtigt. Auf Grund dessen müssen geometrische Objekte zwar detailgetreu, aber so Performance einsparend wie möglich gestaltet werden. Ein besonderes Augenmerk liegt auf den Fahrzeugobjekten, da sie im Automotive Kontext besonderer Beachtung bedürfen.

Ziel dieser Bachelorarbeit ist es, die Performance im Bezug auf geometrische Fahrzeugmodelle in einer 3D-Echtzeit-Umgebung zu analysieren und zu verbessern. Es soll untersucht werden, welche Ansätze und Techniken zu einer Erhöhung der Performance angewendet und prototypisch realisiert werden können.

1 Einleitung

Bereits zu Beginn des 20. Jahrhunderts gewann die Elektronik in Fahrzeugen immer mehr an Bedeutung. Die Kurbel zum Starten des Motors wurde durch einen elektronischen Starter, die damals verwendeten Karbidlampen durch elektrisches Licht ersetzt. Später folgten automatische Fensterheber und andere elektronische Bauteile. Die ersten Steuergeräte wurden Ende der 80er Jahre eingeführt und arbeiteten zunächst eigenständig, wurden aber in den 90er Jahren zunehmend miteinander vernetzt. **[Mül07]** Für den Fahrer spürbare Auswirkungen von Steuergeräten sind zum Beispiel die automatische Gurtstraffung oder der elektrische Fensterheber. Um zu gewährleisten, dass Steuergeräte den hohen Belastungen der täglichen Nutzung und den Anforderungen an Sicherheit und Zuverlässigkeit standhalten, ist es erforderlich sie auf Funktionalität zu testen. Gerade bei einem autonomen Eingriff in das Fahrverhalten, wie zum Beispiel der automatischen Einleitung des Bremsvorgangs, müssen Steuergeräte die erforderliche Präzision und Leistung erbringen. Abbildung 1.1 zeigt neben den im Fahrzeug verbauten Steuergeräten die Bussysteme, welche zur Kommunikation der Steuergeräte untereinander dienen.

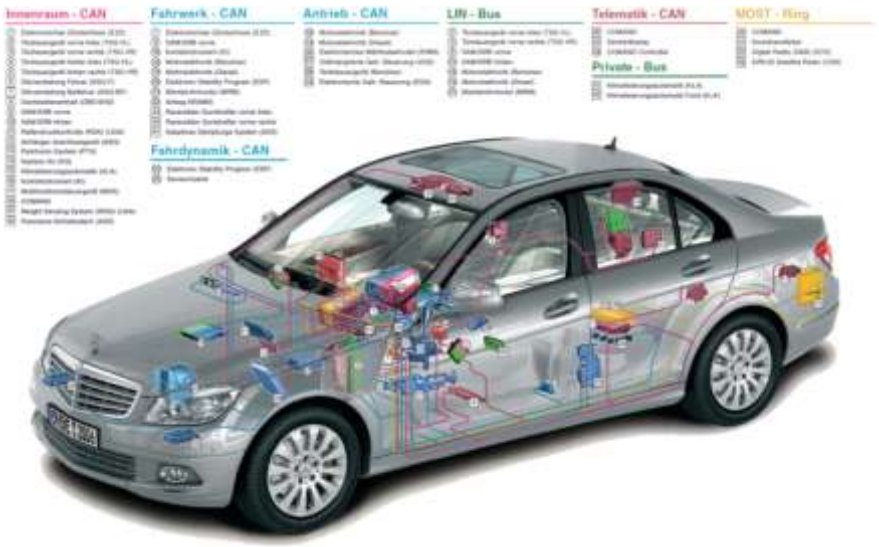


Abbildung 1.1: Gesamtvernetzung C-Klasse (Quelle: Daimler AG).

Moderne Oberklasse-Fahrzeuge können heute bereits eine Anzahl von bis zu 80 Steuergeräten beinhalten **[Mül07]**. Durch die Vernetzung über Bussysteme werden die potentiellen Fehlerquellen um ein Vielfaches erhöht, somit ist das Testen von Steuergeräten und Steuergeräteverbünden für die Nutzung im Straßenbetrieb erforderlich **[Bre04]**. Um die Leistungsfähigkeit von Fahrzeugen in Bezug auf Sicherheit, Komfort und Umweltbelastung zu verbessern, wird der Einsatz von Elektronik auch in Zukunft weiter erheblich zunehmen **[Bre04]**.

Seit einigen Jahren werden von der Automobilindustrie so genannte Fahrerassistenzsysteme (FAS) entwickelt, die den Fahrer in seiner Aufgabe der Fahrzeugführung unterstützen und entlasten sollen **[Bre04]**. Sie übernehmen zunehmend direkt in die Fahraufgabe eingreifende und damit sicherheitskritische Aufgaben. Damit gerade kamerabasierte FAS, beispielsweise Objekt- oder Verkehrszeichenerkennung, Spurhalte- und Totwinkelassistent, effektiv getestet werden können, wird eine Testumgebung benötigt, die der späteren Anwendung der Systeme entspricht. Die zurzeit gängigste Methode zum Testen kamerabasierter FAS ist das Testen durch reale Fahrten auf vorgegebenen Strecken, das jedoch mit einem großen Aufwand verbunden ist. Außerdem lassen sich Gefahrensituationen schlecht bis gar nicht simulieren, da dies mit einem enormen Sicherheitsrisiko verbunden ist. Aus diesem Grund kommen neuartige Techniken aus dem Bereich der 3D-Echtzeitantwendung zum Einsatz. Anstelle der realen Welt werden computergenerierte Bilder den optischen Sensoren der Kameras zur Verfügung gestellt. Die Software zur Generierung dieser Bilder wird im Automotive Bereich als Visual-Loop-Komponente bezeichnet.

1.1 Motivation

Durch die stetige Entwicklung der Computertechnik ist es möglich, immer komplexere virtuelle Welten zu erschaffen. Gerade beim Testen von kamerabasierten FAS besteht ein hoher Anspruch an die Grafik, die durch die Visual-Loop-Komponente generiert wird. Die Funktionen dieser FAS bauen auf Informationen aus dem Umfeld des Fahrzeugs auf **[Bis05]**. Um demzufolge der Wirklichkeit entsprechende Ergebnisse beim Testen von kamerabasierten FAS mit Hilfe von 3D-Echtzeitanwendungen zu erhalten, muss die simulierte virtuelle Welt so realistisch wie möglich beschrieben sein **[Sch10]**. In Abbildung 1.2 ist ein Vergleich einer Aufnahme der Realität mit einer Visual-Loop-Komponente abgebildet.

Abbildung 1.2: Vergleich Visual-Loop-Komponente mit Realität (Quelle: MBtech).

Damit bei einem System von einer Echtzeitanwendung gesprochen werden kann, muss dieses innerhalb einer bestimmten Zeitspanne ein Ergebnis, das aus der Verarbeitung anfallender Daten resultiert, zurückgeben **[Zöb08]**. Im Falle einer 3D-Echtzeitanwendung besteht der Anspruch eine flüssige Darstellung zu gewährleisten. Um eine Simulation als flüssig bezeichnen zu können, muss ein Bild in 40 ms berechnet und angezeigt werden können. Daraus ergibt sich eine Bildwiederholfrequenz von 25 Bildern pro Sekunde. Je mehr 3D-Objekte berechnet werden müssen, desto größer muss die Rechenleistung des Hardwaresystems sein. Es besteht also ein Konflikt zwischen dem Streben nach einem hohen Detailreichtum durch eine hohe Anzahl an 3D-Objekten und der gleichzeitigen flüssigen Darstellung. Demzufolge muss entschieden werden, auf welche Details verzichtet werden kann, um eine komplexe virtuelle Welt in Echtzeit darstellen zu können. Dabei dürfen die Grundformen, Grundfarbwerte und interaktive Funktionalitäten der 3D-Objekte, wie beispielsweise das An- und Abschalten der Scheinwerfer bei 3D-Fahrzeugmodellen, nicht verloren gehen **[Dis03]**.

1.2 Problemstellung

Durch eine Verbesserung der Hardwareleistung lässt sich zwar die Performance geringfügig verbessern, aber da gerade bei 3D-Echtzeitanwendungen Engpässe zumeist durch die Software verursacht werden, ist dies nicht als eine geeignete Lösung anzusehen **[ATH08]**. Ein Performanceengpass im Bereich der 3D-Echtzeitanwendung entsteht durch Arbeitsschritte, die durch die lange Berechnungszeit einen Stau in der Grafikpipeline verursachen.

Herausforderung dieser Arbeit ist es, die Darstellung von 3D-Fahrzeugmodellen einer Visual-Loop-Komponente zu verbessern. Dabei liegt das Augenmerk nicht nur auf der Optimierung der Performance, sondern auch auf dem Erreichen einer realitätsnahen Darstellung der 3D-Fahrzeugmodelle. Zum konzeptionellen Vorgehen soll daher zunächst eine Analyse durchgeführt werden, um genau bestimmen zu können, unter welchen Bedingungen ein Einbruch der Performance auftritt.

Da der Anspruch an eine flüssige Darstellung der Testszenarien besteht, können nicht beliebig viele 3D-Objekte platziert werden. Zur Reduzierung von 3D-Objekten kommen unterschiedliche Techniken zum Einsatz, die unter bestimmten Kriterien angewendet werden. So werden beispielsweise Flächen, deren Normalen nicht in die Richtung der Kamera weisen, mit Hilfe des Back Face Cullings ausgeblendet. 3D-Objekte, die sich nicht im Sichtfeld der Kamera befinden, werden durch das View Frustum Culling erst gerendert, wenn diese in den Sichtkegel der Kamera eintreten. Außerdem lassen sich unter Verwendung von Occlusion Culling 3D-Objekte, die durch Andere verdeckt werden, ebenfalls ausblenden. **[AM99]**. (Abbildung 1.3)

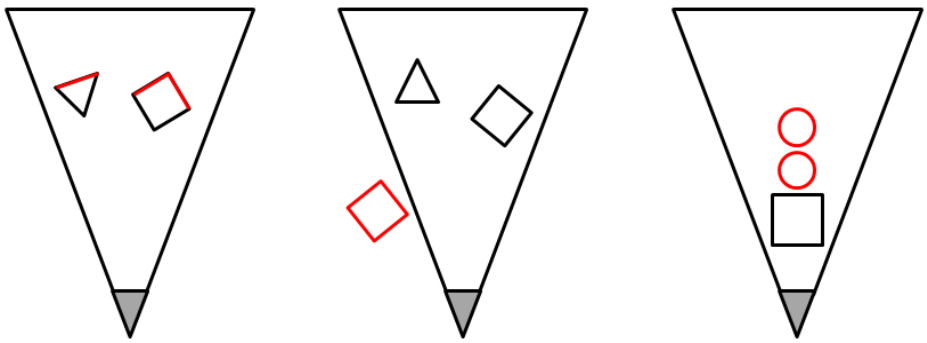


Abbildung 1.3: (von links nach rechts) Back Face Culling, View Frustum Culling und Occlusion Culling. Die rot makierten Flächen werden in der Bildberechnung nicht berücksichtigt.

Eine weitere Methode zur Performanceeinsparung wird als Level-of-Detail bezeichnet. Bei dieser Technik werden unwichtige, sowie nicht sichtbare Informationen ausgeblendet oder vereinfacht. Da das Auge des Betrachters Objekte, die sich weit entfernt befinden, nur noch sehr schwer zu erkennen vermag, kann bei diesen Objekten auf Details verzichtet werden.

1.3 Zielsetzung der Arbeit

Ziel dieser Arbeit ist zunächst eine Performanceanalyse der Visual-Loop-Komponente bzgl. der 3D-Fahrzeugmodelle durchzuführen. Anschließend werden Methoden für ein Konzept aufgestellt, die zu einer Performanceoptimierung und einer Erhöhung der realistischen Darstellung führen. Die unterschiedlichen Ansätze werden gegeneinander abgewogen, um entscheiden zu können, welche in einer prototypischen Implementierung umgesetzt werden können. Schlussendlich wird durch eine Evaluation nachgewiesen, dass eine Performanceoptimierung erreicht wurde. Hierzu wird die im Konzept angewandte Performanceanalyse erneut durchgeführt und die Ergebnisse der Evaluation mit dieser verglichen. Zusammenfassend werden folgende Ziele definiert:

Z1: Verringerung der Performanceeinbrüche
Z2: Verbesserung der realistischen Darstellung

1.4 Aufbau der Arbeit

Innerhalb des Kapitels "Einführung" werden zunächst Motivation, Problemstellung und Zielsetzung genannt. Anschließend wird dem Leser im Kapitel " Fahrerassistenzsysteme, Grafikpipeline und Performancemessungen" ein Einblick in die Grundlagen und den Stand der Technik dieser Arbeit gegeben. In diesem Kapitel wird neben den FAS auch Bezug zu dem aktuellen Stand des Testens mittels Visual-Loop-Komponenten im Automotive Bereich genommen. Anschließend wird die Grafikpipeline erläutert, der sich auch die Visual-Loop-Komponenten bei der Berechnung der Frames bedient. Am Ende des Kapitels wird der Stand der Technik zu Performancemessungen von 3D-Echtzeitanwednungen Bezug genommen. Im Kapitel "Performanceanalyse" wird eine Analyse anhand einer Visual-Loop-Komponente durchgeführt, um diese auf Performanceengpässe zu untersuchen. Auf die Analyse aufbauend wird im Kapitel "Optimierung der 3D-Fahrzeugmodelle" weiterführend ein Konzept erstellt, in dem mögliche Optimierungswege vorgeschlagen werden. Die in dem Konzept vorgestellten Methoden werden gegeneinander abgewogen und entschieden, welche für eine Realisierung geeignet erscheinen. In Kapitel "Implementierung" wird dann die prototypische Implementierung beschrieben. Durch die Evaluation im 6. Kapitel werden die implementierten Methoden durch eine erneute Performanceanalyse überprüft. Zum Abschluss wird im letzten Kapitel die Arbeit zusammengefasst und einen Ausblick auf weiterführende Ansätze gegeben.

1.5 Zusammenfassung der wichtigsten Ergebnisse

Im Rahmen dieser Arbeit wird die Performance einer Visual-Loop-Komponente untersucht und geeignete Methoden zur Verbesserung dieser recherchiert und entwickelt. Um beim Testen von kamerabasierten FAS durch eine Visual-Loop-Komponente Ergebnisse zu erzielen, die denen herkömmlicher Testfahrten so Nah wie möglich sind, muss die Darstellung so realitätsnah wie möglich sein. Dabei gilt, dass die Bilder, die durch die Visual-Loop-Komponente erzeugt werden, mit mindestens 30 Frames pro Sekunde angezeigt werden, da die Kamerasysteme der FAS die Umgebung mit dieser Framerate aufnehmen. Die Framerate dient für die nachfolgende Performanceanalyse als Messwert.

PROVEtech:VL, die Visual-Loop-Komponente der PROVEtech Suite von der MBtech-Group, dient als Testsystem für die Performanceanalyse. Zur Analyse wird NVIDIA's PerfHUD verwendet, das sich durch eine Vielzahl an Analysemöglichkeiten auszeichnet. Durch die zusätzliche Möglichkeit der automatischen Performanceanalyse von PerfHUD kann genau bestimmt werden, welche Objekte einen Engpass verursachen.

Da die Abschnitte der Grafikpipeline voneinander abhängig sind, lässt sich das Konzept der Bottleneck-Analyse für die Performanceanalyse anwenden. Das Ergebnis der Performanceanalyse ergibt, dass die Terrainlogik und die Fahrzeuge die Hauptursache für die massiven Performanceeinbrüche sind. Mit ca. 3250 Draw Calls ist das Terrain das Objekt, das die meisten Draw Calls verursacht. Die Fahrzeuge liegen mit 1652 Draw Calls an zweiter Stelle. Die restlichen Objekte besitzen insgesamt 200 Draw Calls und fallen somit im Vergleich zur Terrainlogik und den Fahrzeugen kaum ins Gewicht.

Da der Performanceengpass, der durch die Terrainlogik verursacht wird auf einem schnell behebbaren Fehler im Skript basiert, wird das Hauptaugenmerk auf die Optimierung der Fahrzeuge gelegt. Zunächst wird eine Recherche über bestehende Methoden durchgeführt. Die Bewertung der Methoden wird nach den Kriterien Implementierungsaufwand, Performancegewinn und Eingriff in das bestehende Programm vorgenommen.

Die Methode, die für die Performanceoptimierung den meisten Nutzen erbringt, ist das Zusammenfügen der einzelnen Objekte, wie z. B. Lichter, Karosserie, Plastikverkleidung, etc., die in einem Fahrzeug enthalten sind. Dabei entstehen jedoch neue Einschränkungen, wie beispielsweise die Tatsache, dass nur noch ein Material pro Fahrzeug verwendet werden kann. Diese Problematik kann allerdings mit Hilfe einer Textur ebenfalls behoben werden. Die zweite Me-

thode, die zu einem großen Performancegewinn führt und dabei keinen Eingriff in die bestehenden Strukturen benötigt, ist die Verwendung eines diskreten LOD-System.

Um die Performance noch weiter zu steigern, kann das Konzept dieser Methoden zu einer neuen Idee zusammengefügt werden. Das daraus entstehende Bonematrix-LOD-System basiert zunächst auf einem diskreten LOD-System, es werden jedoch zusätzlich alle Objekte der letzten LOD-Stufe zu einem Geometrieobjekt zusammengefügt. Durch die Zuweisung von Bones wird die Transformation der einzelnen Fahrzeuge gewährleistet. Somit können alle Objekte der letzten LOD-Stufe als ein Batch an die Grafikkarte geschickt werden. In der Evaluierung der prototypischen Implementierung wird deutlich, dass die Verwendung eines solchen Systems gegenüber des diskreten LOD-Systems in der letzten Stufe die Performance um fast das doppelte erhöht.

Diese Arbeit bietet eine Grundlage Performanceengpässe zu kategorisieren. So kann das Konzept des Bonematrix-LOD-System in Zukunft für eine vollständige Implementierung in eine Visual-Loop-Komponente dienen. Um die Performance von PROVEtech:VL weiter steigern zu können, sollte eine weitere Performanceanalyse durchgeführt werden, da sich der Performanceengpass nun an einer anderen Stelle befinden wird. Anschließend können die restlichen im Konzept vorgestellten Methoden implementiert werden.

2 Fahrerassistenzsysteme, Grafikpipeline und Performancemessungen

Dieses Kapitel befasst sich mit den Grundlagen und dem Stand der Technik dieser Arbeit. Zu Beginn werden FAS in ihrer Funktion erläutert. Anschließend wird auf das Testen von FAS eingegangen und die unterschiedlichen Methoden, die zum Einsatz kommen, vorgestellt. Mit Hilfe von 3D-Echtzeitanwendungen, wie z.B. die Visual-Loop-Komponente, lassen sich FAS in Zukunft, annähernd mit derselben Effizienz, wie unter realen Bedingungen testen **[Sch10]**. Die Grafikpipeline dient dabei als Schnittelement zwischen der 3D-Echzeitanwendung und der Hardware des Computersystems. Abschließend wird auf den Stand der Technik von Performancemessungen im Bezug auf 3D-Echtzeitanwendungen eingegangen.

2.1 Fahrerassistenzsysteme

Der Fahrzeugführer erwartet von FAS, dass sie ihn im Verkehr unterstützen und keine zusätzliche Belastung oder Ablenkung hervorrufen. Es gibt eine Vielzahl von FAS, die den Fahrer in allen Verkehrssituationen unterstützen und ihm ein entspanntes, stress- und unfallfreies Fahren ermöglichen sollen. FAS beispielsweise, die aktiv und ohne situative Initiierung durch den Fahrer in die Fahrdynamik eingreifen und so eine Teilfunktion der Fahrzeugführung selbsttätig bewältigen, werden als autonom bezeichnet. Häufig wird dabei die Längs- oder Querführung des Fahrzeugs direkt übernommen. Darüber hinaus umfasst diese Kategorie die Beeinflussung weiterer Faktoren, wie beispielweise die Verteilung der Radlasten oder der Nick-, Wank-, und Rollmomente. Autonome Systeme zeichnen sich somit durch die Manipulation dieser Faktoren ohne Einflussnahme durch den Fahrer aus. **[Sti05]**

Während autonome Systeme durch direkte Eingriffe zur Erfüllung der Fahraufgabe unmittelbar beitragen, bieten Fahrerinformationssysteme Informationen an, die dem Fahrer für die Ausführung seiner Stelleingriffe nützlich ist, ohne das Fahrverhalten direkt zu beeinflussen. In diese Kategorie fallen beispielsweise Navigationssysteme, die Sensierung und Anzeige von Verkehrszeichen sowie zahlreiche Warnfunktionen. Dabei bedienen sich diese FAS auch neuartiger Techniken, wie Radar- oder Kamerasystemen. Mit dem Spurverlassenswarner bieten bereits

mehrere Fahrzeughersteller ein kamerabasiertes Fahrerinformationssystem als Produkt an. **[Sti05]**

Beispielsweise kann durch die frühzeitige Erkennung eines Fußgängers durch ein kamerabasiertes FAS und die daraus resultierende Reaktion, eine Kollision mit diesem verhindert werden. Diese Systeme übernehmen also sehr sicherheitskritische Funktionen und müssen, damit sie im Straßenverkehr eingesetzt werden können und dürfen, ausgiebig getestet werden, um Fehlfunktionen und Fehlererkennungen (falsch-positiv und falsch-negativ) auszuschließen.

2.1.1 Testen kamerabasierter FAS

Mit der gestiegenen Komplexität dieser Systeme ändern sich auch die Anforderungen an die bis zur Entwicklung der Serienreife benötigten Test- und Simulationswerkzeuge. Aktuelle und künftige Assistenzsysteme können mit etablierten Methoden oft nur eingeschränkt oder überhaupt nicht erprobt werden **[Boc09]**. Somit wird ein breites Spektrum an unterschiedlichen Tests benötigt, um eine Abdeckung aller eventuellen Fehlerquellen zu ermöglichen.

Für einen funktionalen Test von kamerabasierten FAS müssen optische Informationen über die aktuelle Fahrsituation zur Verfügung gestellt werden. Hierzu stehen gegenwärtig folgende Möglichkeiten zur Verfügung: Reale Testfahrten, Video-Tests, HiL-Tests[1] mittels abstrahierten computergenerierten Fahrsimulationen und Tests mittels computergenerierten fotorealistischen Fahrsimulationen. Die unterschiedlichen Methoden werden in den nachfolgenden Abschnitten beschrieben.

Reale Testfahrten

FAS können während der Fahrt mit einem Fahrzeug getestet werden. Als Testfahrzeuge kommen dafür Prototypen oder Technikträger zum Einsatz. Die Spezialkonstruktionen, die hierzu notwendig sind, sind jedoch sehr teuer und erst in späteren Entwicklungsstadien der Gesamtfahrzeugentwicklung möglich. Zu den Kosten des Einbaus in diese speziellen Testfahrzeuge kommen außerdem weitere, wie beispielsweise Treibstoff oder Personalkosten, hinzu. Davon abgesehen ist eine exakte Reproduzierung der Testfälle nicht möglich, da sich die Umwelt und das Fahrverhalten des Autos permanent ändern. Außerdem lassen sich Gefahrensituation, wie

[1] Hardware-in-the-Loop-Tests

beispielsweise die Erkennung von Fußgängern nicht ohne ein erhebliches Risiko testen. Dennoch können beispielsweise Testsysteme mit 3D Echtzeitanwendungen reale Testfahrten nicht ersetzen, da eine Umgebungssimulation am Prüfstand nicht identisch mit der Kombinatorik und Zufälligkeit einer realen Situation im Fahrzeug ist **[Mül08]**. Nur im Fahrzeug sind sämtliche Umgebungsbedingungen vollständig real und daher können auch einige Einflüsse nur im Fahrzeug vollständig real getestet werden (z. B. mechanische Einflüsse, Störstrahlungen, etc.) **[Mül07]**.

Video-Tests

Neben Testfahrten besteht die Möglichkeit, zuvor aufgezeichnete Videodaten in ein zu testendes Steuergeräte einzuspielen. Dies hat den Vorteil, ein größeres Spektrum an Testfällen abdecken zu können. Darüber hinaus ist bei dieser Methode die Reproduzierbarkeit der Testfälle gegeben. Der mit dieser Methode verbundene Nachteil ist die fehlende Rückkopplung der, von dem FAS ausgelösten Reaktionen auf die Umgebung. Eine durch das FAS ausgelöste Vollbremsung würde somit keine Reaktion auf die Umgebung, in diesem Fall das Bildmaterial, auslösen. Ein Test, ob der Eingriff des Steuergerätes funktioniert ist zwar möglich, ob dieser allerdings korrekt und mit der erforderlichen Präzision erfolgt, bleibt offen.

Test mittels Abstrakter Darstellungen

Da reale Testfahrten und Video-Tests aus den o.g. Gründen nicht das volle Spektrum der zu testenden Funktionalitäten abdecken, wird eine Methode verwendet, bei der kamerabasierte FAS mit Hilfe von HiL-Technik getestet werden. Hierbei wird den Kamerasensoren eine stark abstrahierte computergenerierte Umgebungssimulation eingespielt, wie sie in Abbildung 2.1 beispielhaft dargestellt ist.

Bei der HiL-Technik werden eingebettete System, wie z. B. elektronische Steuergeräte, über I/O-Schnittstellen mit einem Testsystem verbunden. Dieses System simuliert die reale Umgebung des eingebetteten Systems und unterzieht diesem mehrere Tests in einer vorher definierten Reihenfolge. **[Mül07]**

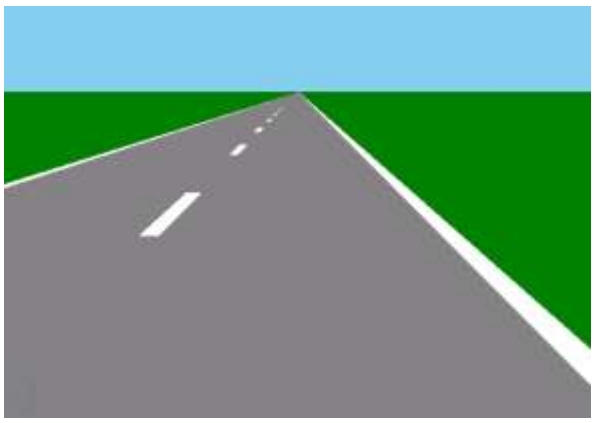


Abbildung 2.1: Einzelbild einer abstrahierten Computergrafik-Simulation für funktionale kamerabasierte Steuergerätetests [Fen09].

Dieses Bild entstammt einer Simulation, welche in **[Fen09]** für den Test eines Spurverlassenswarner entwickelt wurde. Grundlegende Testsituationen, wie zum Beispiel „Spurwechsel", „Überfahren der Seitenlinie" etc., können gezielt erzeugt und getestet werden. Durch das erneute Abspielen der Simulation sind eine Reproduktion und eine grundlegende Fehleranalyse der Algorithmen innerhalb des Steuergeräts möglich. Außerdem lassen sich schnell grundlegender Testparameter, wie bspw. „Spuranzahl" oder „Spurbreite" verändern, wodurch es möglich wird, eine große Anzahl an Testszenarien zu generieren. Die Reaktionen des Steuergeräts auf die Umgebung haben direkte Auswirkungen auf die Simulation. Dennoch besteht der Nachteil, dass diese abstrahierte Simulationsdarstellung nicht der Realität entspricht und somit keine ganzheitliche Testfallabdeckung erzielt werden kann.

Weiterführende Literatur zum Thema Hardware-in-the-Loop-Systeme: **[Sax08] [Har08] [Mül07]**

Visual-Loop-Test

Da die vorgestellten Testmethoden für eine abschließende sicherheitsrelevante Absicherung der Funktionalität nicht ausreichend sind, wurde auf die Technik von derzeitigen 3D-Echtzeitanwendungen, wie sie auch in Computerspielen verwendet werden, zugegriffen. Abbildung 2.2 zeigt den Aufbau der Visual-Loop-Komponente von PROVEtech in Verbindung mit einem HiL-System.

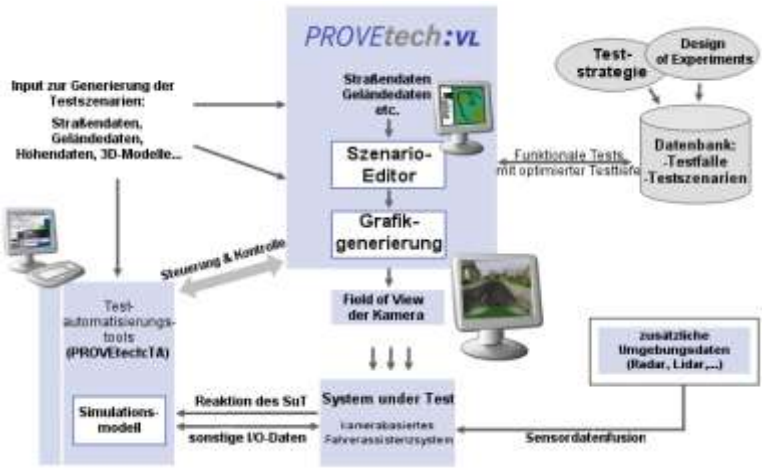


Abbildung 2.2: HiL-Aufbau für funktionale kamerabasierte Steuergerätetests (Quelle: MBtech Group).

Fotorealistische computergenerierte Fahrsimulationen haben den Vorteil, die Testtiefe durch das Hinzufügen von Parametern wie z.B. „Nebel", „Sonnenintensität" etc. zu erhöhen. Darüber hinaus lassen sich gezielt (Gefahren-)Situationen erstellen, die auf Erkennung und korrekte Ausführung der Funktionen des FAS getestet werden können. Außerdem sind Reproduktionen und Variationen des Tests sowie eine Fehleranalyse im Falle einer Nicht-Erkennung einfacher möglich. Im Folgenden wird der Systemaufbau einer Visual-Loop-Komponente, wie sie bei der MBtech Group zum Test kamerabasierter Steuergeräte verwendet wird, beschrieben.

Neben PROVEtech:VL existieren vergleichbare Testsysteme dieser Art, wie bspw. Vires Image Generator v-IG[2], TNO PreScan[3], Oktal Scaner[4], FTronik[5], dSPACE MotionDesk[6], IPG CarMaker[7]. Zu Testzwecken wird eine simulierte Fahrt durch ein computergeneriertes Testszenario, wie es in Abbildung 2.3 beispielhaft dargestellt ist, als Input für das FAS den Videosensoren zur Verfügung gestellt. Die Generierung eines Testszenarios wird im Folgenden beschrieben.

[2] http://www.vires.com/Products_ImageGeneratorSoftware.htm
[3] http://www.tno.nl/prescan
[4] http://www.scaner2.com/php/index.php
[5] http://www.ftronik.de
[6] http://www.dspace.de/de/gmb/home/applicationfields/automotive/ecu_testing.cfm
[7] http://www.ipg.de/CarMaker.609.0.html?&L=2

Abbildung 2.3: Einzelbild einer Computergrafik-Simulation für funktionale kamerabasierte Steuergerätetests (Quelle: MBtech Group).

Die Umgebung lässt sich mit Hilfe eines Szenario-Editors erstellen. Hier lassen sich externe Daten, wie Straßen oder Geoinformationen, bspw. aus Open Street Map[8] und Höhendaten von der NASA SRTM[9] (shuttle radar topography mission), importieren und anzeigen. Lebewesen, Vegetation und sonstige natürliche oder vom Menschen erschaffene Objekte der realen Welt, die dazu beitragen die Umgebung komplexer zu gestallten, können mit Hilfe von 3D-Modellierungssoftware[10] erstellt, anschließend in den Szenario-Editor importiert und dort beliebig positioniert werden. Durch Animationen von Charakteren, z.B. Menschen oder Tieren, gewinnt das Szenario deutlich an Realismus. Die Summe der Straßen-, Gelände- und 3D-Objektdaten definiert ein Szenario. Durch vorher definierte Testfälle, die in einer Datenbank hinterlegt sind, lassen sich Testszenarien zufällig aneinanderreihen. Somit kann ein individueller Testablaufplan mit unterschiedlichen Testfällen generiert werden. Diese Testszenarien werden grafisch aufbereitet und bspw. durch Abfilmen von einem Monitor oder direkte Einspielung als Input in das kamerabasierte FAS übertragen. Das Steuergerät wertet dann durch Bildalgorithmen das empfange Bild aus und führt daraus resultierende Reaktionen aus.

Ein weiterer Vorteil des virtuell in Echtzeit befahrbaren Szenarios gegenüber aufgezeichnetem Videomaterial ist - neben der Individuellen Testfallerstellung - die Möglichkeit, dieses Szenario

[8] http://openstreetmap.org
[9] http://www2.jpl.nasa.gov/srtm
[10] Bspw. Autodesk 3DS Max, Autodesk Maya, Autodesk Softimage, Maxon Cinema4D, Blender etc.

mit Berücksichtigung eventueller Eingriffe in die Fahrdynamik des Fahrzeuges zu befahren. So kann das FAS durch Rückkopplung mit der Visual-Loop-Komponente direkten Einfluss auf die Fahrsituation nehmen. Die Reaktion des Steuergerätes, bspw. bei einer Vollbremsung des Fahrzeugs, kann direkt verfolgt und reproduziert werden.

In Abbildung 2.4 ist die Entwicklung von virtuellen Darstellungen zum Testen von kamerabasierten FAS dargestellt. Die Visual-Loop-Komponente hebt sich mittlerweile deutlich von der abstrahierten Darstellung ab, der Unterschied zu realen Videobild bleibt jedoch sichtbar **[Sch10]**. In Zukunft wird sich die Darstellung der Visual-Loop-Komponente dem realen Videobild mehr und mehr annähren.

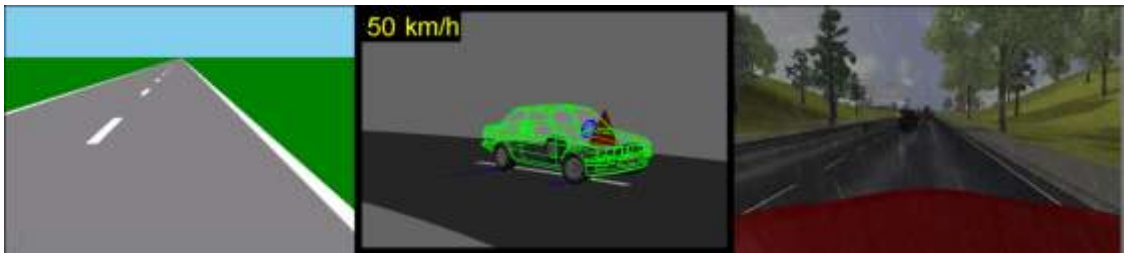


Abbildung 2.4: Entwicklung von virtuellen Darstellungen zum Testen von kamerabasierten FAS: (von links nach rechts): abstrahierte Labordarstellung [Fen09], Virtual-Test-Drive [Str99], Visual-Loop-Komponente (Quelle: MBtech Group).

In Tabelle 2.1sind die unterschiedlichen Testvarianten mit ihren Vor- und Nachteilen gegenüber gestellt. Dabei werden die Testvarianten in den Kategorien Kosten, Testfallabdeckung, Closed Loop Test, Sicherheit, Reproduzierbarkeit und Realität bewertet. Das Testen mittels Testfahrten zeichnet sich durch die unmittelbare Realitätsnähe und eine mäßige Testfallabdeckung aus, jedoch bietet es sonst nur Nachteile. Mit Hilfe von Videodaten wird die Reproduzierbarkeit der Testfälle möglich. Die Vorteile für die Verwendung von abstrakten Darstellungen sind die geringeren Kosten bei der Erstellung, die Sicherheit und die Reproduzierbarkeit. Die Nachteile sind eindeutig der Verlust der Realität durch die Abstrahierung und die nicht vorhandene Testfallabdeckung. Der Test mittels einer Visual-Loop-Komponente bietet die meisten Vorteile. Neben der Reproduzierbarkeit und der verringerten Kosten, kann eine komplette Testfallabdeckung gewährleistet werden. Außerdem wird durch die Visual-Loop-Komponente eine höhere Realitätsnähe, als durch die Verwendung von abstrakten Darstellungen gewährleistet, sowie ein Closed-Loop-Test ermöglicht. Für die Bewertung in der Tabelle werden die Zeichen + (hoch), o (mittel), - (niedrig) verwendet.

Tabelle 2.1 Vergleich ausgewählter Testarten von Assistenzsystemen

Art des Tests	Kosten	Testfallabdeckung	Closed Loop Test	Sicher-heit	Repro-duzierb arkeit	Reali-tät
Testfahrt	-	o	-	-	-	+
Video-Test	o	o	-	o	+	+
Abstrakter Darstellungen	+	-	o	o	+	-
Viusal-Loop-Komponente	+	+	o	o	+	o

2.2 Grafikpipeline

Die Grafikpipeline ist ein Modell, das beschreibt, welche Schritte ein Grafiksystem zum Rendern, also zur Darstellung einer 3D-Szene auf einem Medium, durchführen muss. Die Hauptfunktion der Grafikpipeline besteht darin, dreidimensionale Objekte, Shader, Texturen, Lichtquellen, etc. zu generieren oder zu berechnen. Durch eine virtuelle Kamera wird anschließend die dreidimensionale Szene auf ein zweidimensionales Bild projiziert und angezeigt. Die Grafikpipeline ist somit das grundlegende Werkzeug für Echtzeitrendering **[ATH08]**.

Eine Pipeline besteht aus mehreren Abschnitten, die von Daten durchlaufen werden müssen **[Hen96]**. Dabei können sie nicht in den nächsten Abschnitt der Pipeline gelangen, bevor die Verarbeitung in dem aktuellen Abschnitt beendet ist. Es können jedoch zugleich Daten in anderen Abschnitten enthalten sein. Um die Geschwindigkeit der Datenverarbeitung zu erhöhen, kann die Pipeline in *n* Abschnitte unterteilt werden, wodurch sich mehr Daten in der Pipeline aufhalten können und die Geschwindigkeit um den Faktor *n* erhöht wird. Dennoch ist die Geschwindigkeit, mit der die Daten durch die Pipeline gelangen, durch die Zeit für die Verarbeitung des langsamsten Arbeitsschrittes begrenzt. Dies gilt unabhängig davon, wie schnell die anderen verarbeitet werden. Das Blockieren der Pipeline durch einen Arbeitsschritt wird als Performanceengpass oder engl. Bottleneck bezeichnet. **[Fer04]**

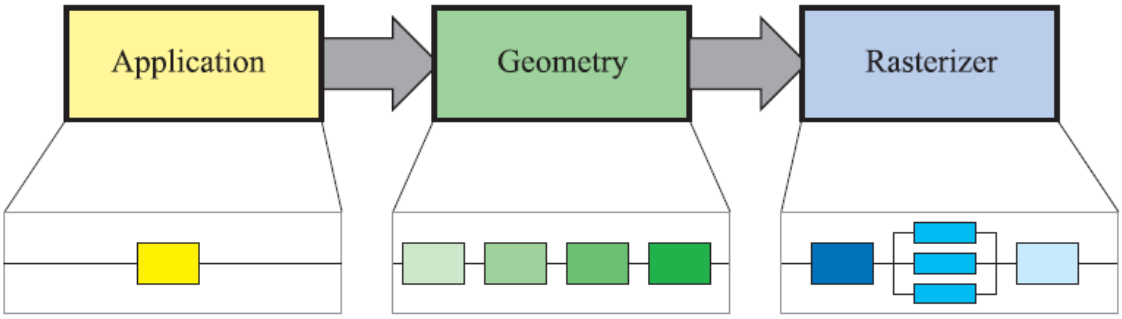


Abbildung 2.5: Die Basis einer Grafikpipeline. [**ATH08**]

Diese Art der Pipelinekonstruktion wird auch im Bereich des Echtzeitrendering angewendet. Die drei konzeptuellen Abschnitte der Grafikpipeline (Applikation, Geometrie, Rasterung) sind in Abbildung 2.5 dargestellt. Zur Ansteuerung der Grafikpipeline stehen sogenannte Grafik-APIs, wie bspw. Direct3D[11] oder OpenGL[12], zu Verfügung. Diese abstrahieren die zugrundeliegende Hardware und nehmen dem Programmierer dadurch viele Aufgaben ab.

2.2.1 Applikation

Wie der Name schon andeutet, ist der erste Abschnitt der Rendering Pipeline bereits in der Anwendung implementiert und wird daher von der CPU verarbeitet. Die CPU besteht häufig aus mehreren Prozessorkernen, die Daten parallel berechnen können und somit die Verarbeitungsgeschwindigkeit wesentlich erhöhen. Eine typische Aufgabe der Applikation ist zu entscheiden, welche Geometrie gerendert werden soll und sie anschließend an den nächsten Pipelineabschnitt weiterzuleiten. **[ATH08]**

2.2.2 Geometrie

Das 3D-Objekt, das von der Applikation übergeben wurde, durchläuft in dem Geometrieabschnitt mehrere Phasen: Objekt- und Kameratransformation, Vertex-Shading, Projektion, Clipping und Screen Mapping. Zunächst werden die unterschiedlichen Koordinatensysteme der 3D-Modelle in das Koordinatensystem der virtuellen Kamera, auch "eye-space" genannt, umgewandelt. Somit befinden sich alle 3D-Modelle im selben Koordinatenraum und verfügen über

[11] http://msdn.microsoft.com/de-de/directx/
[12] http://www.opengl.org/

eine eindeutige Position und Rotation. Um eine realistischen Szene zu erzeugen, ist es erforderlich nicht nur die Form und Position des 3D-Objektes zu bestimmen, sondern es bedarf auch einer realistischen Oberflächenbeschreibung. Ein Chrom-Material zum Beispiel muss neben der Farbdarstellung auch die Umgebung reflektieren und Glanzeffekte von Lichtquellen darstellen können. Materialien und Lichter können dabei durch einfache Farben bis hin zu einer aufwendigen Darstellung von physikalisch korrekten Berechnungen beschrieben sein. Die Oberflächen werden mit Hilfe sogenannter Shader beschrieben. Diese auf mathematischen Algorithmen beruhenden Programme werden von dem Grafikprozessor (GPU)[13] ausgeführt. **[ATH08]**

Nachdem das Vertex-Shading angewandt wurde, werden die 3D-Modelle vom dreidimensionalen in den zweidimensionalen Bildraum der virtuellen Kamera projiziert. Dabei wird die perspektivische Projektion verwendet. Bei dieser Methode werden Objekte, die sich weiter weg von der Kamera befinden, kleiner dargestellt. Um nicht alle Objekte, unabhängig von ihrer Position, auf die Projektionsebene abzubilden, muss ein Bereich (Volumen) definiert werden, in dem sich die Objekte, die gerendert werden sollen, aufhalten. Der Bereich ist, im Falle der Zentralprojektion, ein Pyramidenstumpf und wird begrenzt durch die "Front- und die Back-Clipping-Plane". Alle Objekte, bzw. Objektteile, die sich nicht innerhalb dieses Volumens befinden, werden nicht auf die Bildebene projiziert und beim Renderprozess nicht beachtet **[ATH08]**. Der Vorgang wird als Clipping bezeichnet. Objekte, die sich in dem Sichtvolumen befinden, werden der Screen-Mapping Phase übergeben. Die Koordinaten sind nach wie vor dreidimensional. X- und Y-Koordinaten jedes Objekts werden in das Koordinatensystem des verwendeten Mediums transformiert. Die Z-Koordinate wird im Mapping Prozess nicht beachtet. In Abbildung 2.6 ist die Geometrieabschnitt schematisch dargestellt. **[Ola98]**

[13] Graphics Processing Unit: Berechnungen werden auf die Prozessoren der Grafikkarte ausgelagert, um so die Hauptprozessoren des Rechners (CPU) zu entlasten

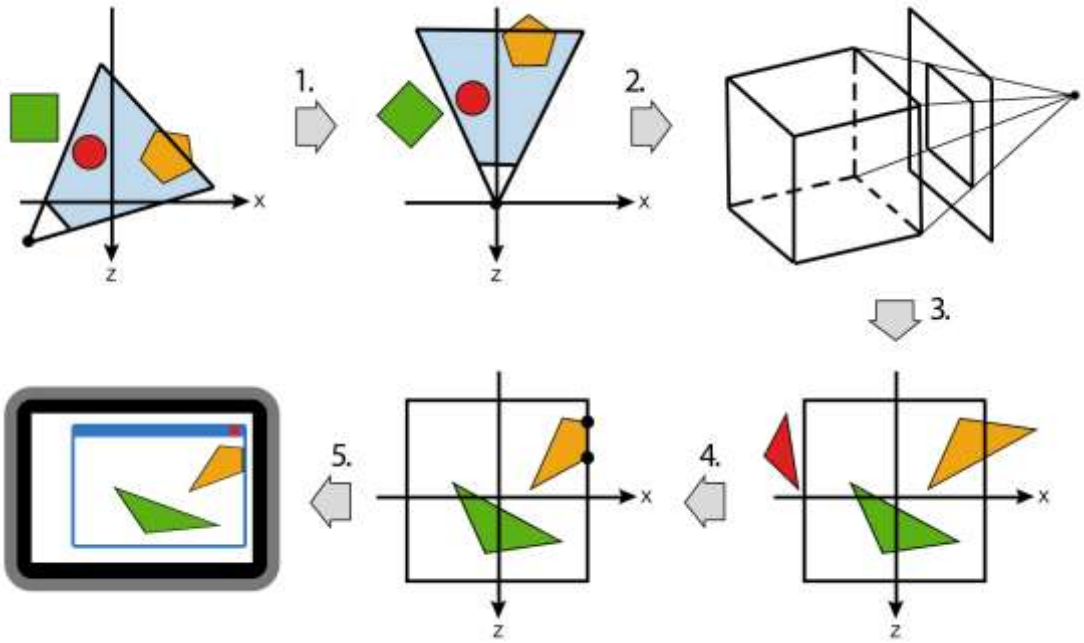


Abbildung 2.6: 1. Kamera-Transformation, 2. Projektion, 3. Clipping, 4. Screen-Mapping, 5. Scan Conversion. Angelehnt an [ATH08].

2.2.3 Rasterung

Das Ziel der Rasterung besteht darin, die Farben für jedes Pixel[14], die das 3D-Objekt auf dem Bildschirm einnimmt, zu berechnen. Der Scan-Line-Algorithmus durchläuft jede Pixelzeile und sucht nach den Schnittpunkten der aktuellen Scanline mit dem Polygon. In der Geometriepipeline wird jedem Vertex ein Farbwert zugewiesen. Dieser Wert wird nun mit Hilfe des Beleuchtungsmodells, das im Pixel-Shading definiert ist, auf die Pixel interpoliert. Anschließend werden alle Pixel gezeichnet, die zwischen zwei Schnittpunkten liegen und sich im Inneren des Polygons befinden. **[Bli96]**

Im letzten Schritt der Rasterung werden die einzelnen 3D-Objekte durch den Framebuffer zusammengefügt. Dieser teilt sich in zwei Ebenen. Im Backbuffer werden die berechneten 3D-Objekte zusammengefügt und zusätzliche Tests durchgeführt. Der Frontbuffer verweist auf das nächste fertig zusammengesetzte Frame im Backbuffer und übergibt dieses an das Ausga-

[14] Abkürzung für ein Bildpunkt

bemedium. Im Backbuffer werden diverse Tests für die Darstellung von 3D-Objekten durchgeführt. Der Alpha-Buffer-Test überprüft, ob ein Pixel eine definierte Transparenzbedingung erfüllt. Ist diese nicht erfüllt, wird das Pixel nicht auf dem Bildschirm dargestellt. **[ATH08]**

Wenn ein Polygon von einem anderen verdeckt wird, muss entschieden werden, welches Polygon auf dem Bildschirm gezeichnet wird. Mit Hilfe des Z-Buffer-Tests werden die Abstände eines jeden Polygons zu der Position der Kamera ermittelt **[Yoo06]**. Um das Polygon, das sich am nächsten zu der Kamera befindet, zu ermitteln, wird der Z-Buffer zunächst mit dem größten möglichen Tiefenwert initialisiert. Für jedes Polygon, das sich näher an der Kamera befindet wird der diesbezügliche Tiefenwert des Z-Buffers mit dem des Polygons überschrieben. Daraus ergibt sich eine Rangfolge, in der die Polygone gezeichnet werden. Jedoch können bei dem Z-Buffer-Test auch Fehler auftreten. Abbildung 2.7 zeigt den ungewollten Effekt, wenn sich die Z-Werte der Pixel zweier sehr dicht hintereinanderliegender Polygone durch Rundungsfehler überschneiden. Dieses Phänomen wird oft als Z-Fighting bezeichnet **[Bie08]**.

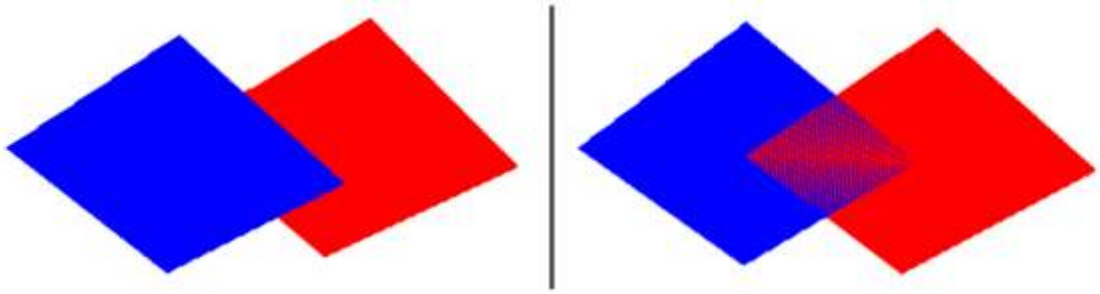


Abbildung 2.7: Z-Phänomen des Z-Fighting [Bie08].

2.3 Grafikengine

Im Bereich der 3D-Echtzeitanwendungen werden Grafikengines als Schnittstelle zwischen Entwickler und Grafikprozessor (GPU) eingesetzt. Diese verbergen die Komplexität und Infrastruktur der GPU und bieten dem Entwickler aber einfache Werkzeuge für die Erstellung von grafischen Inhalten oder der Programmierung von Skripten. Abbildung 2.8 zeigt die Oberfläche des Editors von Unity3D[15], der im weiteren Verlauf der Arbeit verwendet wird.

Neben dieser Grafikengine, die auch für diese Arbeit verwendet wird, gibt es die CryENGINE[16], Unreal Engine[17], OGRE[18], Unigine[19] um, nur die häufigsten Grafikengines zu nennen.

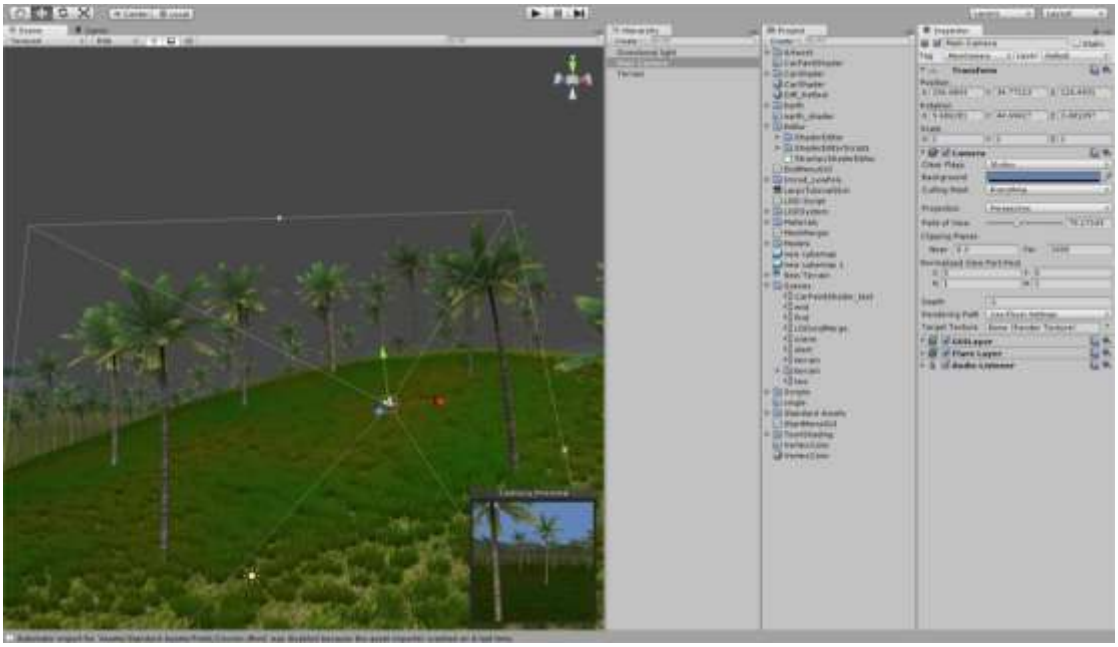


Abbildung 2.8: Editoroberfläche von Unity3D.

[15] http://unity3D.com
[16] http://www.crytek.com/cryengine
[17] http://www.unreal.com/
[18] http://www.ogre3d.org/
[19] http://unigine.com/

2.4 Shaderprogrammierung

Wie in der Grafikpipeline bereits beschrieben, sind Shader Softwaremodule, die hauptsächlich dazu benutzt werden, Rendereffekte auf der GPU zu berechnen. Traditionell wird zwischen zwei Arten unterschieden, der Vertex- und Pixel-Shader. Diese Programme beschreiben die Eigenschaften eines jeden Vertex oder Pixel **[ATH08]**. Die Shaderprogrammierung im Detail zu beschreiben geht über den Rahmen dieser Arbeit hinaus. Es gibt jedoch bereits einiges an Fachliteratur zu diesem Thema, wie z. B. in **[Fer03]**, **[Eng02]**, **[McG04]**. Dennoch soll das Thema der Schaderprogrammierung zum besseren Verständnis in diesem Unterkapitel kurz erläutert werden.

Shader benötigen eine Grafikkarte, die die nötigen Shaderfunktionalitäten bereitstellt. Je höher die unterstützte Shaderversion, desto umfangreicher sind die Möglichkeiten, Oberflächenstrukturen zu beschreiben. Eine Abwärtskompatibilität der Shaderversionen ist nicht gegeben. Bei preiswerten Grafikchips werden die Shadereinheiten häufig weggelassen, wodurch Shader mit Hilfe der CPU berechnet werden müssen, was zu einer deutlich längeren Berechnungszeit führt. Shader sind flexibel einzusetzen, da sie vor dem eigentlichen Renderaufruf aktiviert und danach deaktiviert werden. Sie können Daten vertexweise oder pixelweise schneller ändern, als es durch Software möglich wäre. Durch speziell entwickelte Sprachen (Cg[20], GLSL[21], HLSL[22]), werden die Shader ausgeführt und zur Laufzeit der 3D-Anwendung vom Grafikkartentreiber in einen für die Grafikkarte verständlichen Maschinencode übersetzt. Jeder programmierte Shader besitzt zweit Arten von Inputs: uniform Inputs, mit Werten, die über den Draw Call konstant bestehen bleiben und varying inputs, die für jeden Vertex oder Pixel unterschiedlich sein können. **[Eng02]**

Seit DirectX 10 wurde die Shader-Architektur um einen dritten Shader-Typ, den Geometry-Shader, erweitert. Dieser Shader erhält die vom Vertex-Shader ausgegeben Polygondaten und bearbeitet diese, z.B. durch Hinzufügen von neuer Geometrie. Der Vertex-Shader hingegen kann nur bestehende Geometrie verändern. Durch das Hinzufügen einer Displacement-Map lassen sich wesentlich komplexere 3D-Objekte schaffen. Bei dieser Technik wird das Model mit Hilfe

[20] C for Graphics
[21] OpenGL Shading Language
[22] High Level Shading Language

einer Textur neu trianguliert. So können wesentlich detailreichere 3D-Modelle dargestellt werden. **[Hir04]**

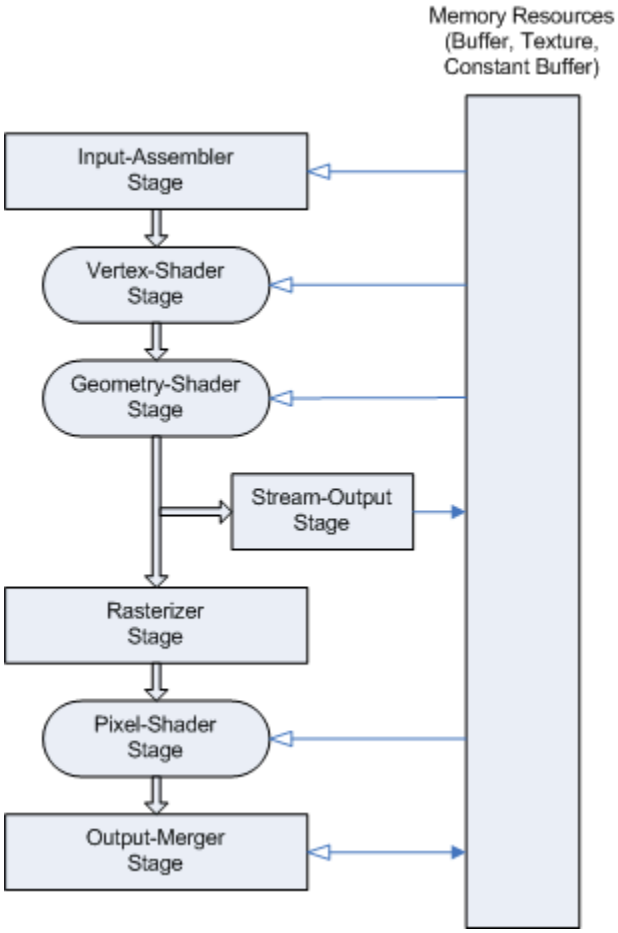


Abbildung 2.9: Diagramm des Datenflusses in der Shaderpipeline [Mic10] .

In Abbildung 2.9 ist die Shaderpipeline, wie sie in der Grafikkarte implementiert ist, dargestellt. Die Vertices durchlaufen dabei die gesamte Shaderpipeline, vom Input-Assembler, der die für die Übertragung von der CPU zur Grafikkarte sorgt, bis hin zum Output-Merger, in dem alle berechneten 3D-Modell zu einem fertigen Frame zusammenfügt werden. Dazwischen werden die unterschiedlichen Shaderprogramme, wie der Vertex-Shader, Geometry-Shader und Pixel-Shader, ausgeführt.

Bei der Erstellung der 3D-Modelle muss auch das spätere Shaderverhalten in Betracht gezogen werden. So können Fehler in der Geometriestruktur auch zu Anzeigefehlern durch den Shader

führen. Eine ausführliche Sammlung zur Optimierung der Geometrie für die Shaderverwendung findet sich in **[Chu04]**.

2.5 GPU Datenübertagung

Wenn eine große Anzahl an Geometrie (Millionen von Vertices) berechnet werden soll, kann es in zwei Abschnitten der GPU zu Performanceengpässen kommen: in der Bus Übertragung und dem Vertex-Cache. Die Leistungsfähigkeit von beiden Abschnitten hängt mit dem Datenfluss von der Anwendung zur Grafikpipeline ab. (vgl. Abbildung 2.10)

2.5.1 Video Bus

In modernen Computern wird die Grafikkarte, die die GPU enthält, mit dem I/O Bus (Ein- und Ausgangsschnittstelle) durch eine PCI-Express (PCI-E) Schnittstelle verbunden. Die theoretische Bandbreite für das Lesen und Schreiben dieses Bus beträgt 4 GB/s. Für einen Fall mit 32 Byte per Vertex bedeutet dies einen Maximalwert von 125 Millionen Vertices pro Sekunde, bzw. 4,2 Millionen Vertices pro Frame bei 30 fps. Es muss jedoch beachtet werden, dass durch Fehler bei der Übertragung die tatsächliche Datenrate weniger als 4 GB/s betragen kann. **[Lou07]**

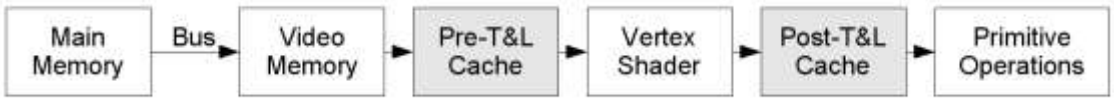


Abbildung 2.10: Speicherabschnitte und Vertex Caches [Lou07]**.**

2.5.2 Vertex Caches

Die Vertex-Koordinaten werden vom Videospeicher der Grafikkarte zur GPU übertragen. Die Übertragungszeit pro Vertex hängt von dem Datentyp der Vertex-Eigenschaften ab.

Ein Cache beschreibt eine Methode, die Inhalte, die bei der Berechnung bereits vorlagen, für den nächsten Zugriff schneller zur Verfügung zu stellen. Es gibt zwei Caches[23] in gegenwärtigen GPUs: Ein Cache zwischen dem Videospeicher und dem Vertex-Shader, genannt "Pre-Transform-Cache" und ein kleinerer Cache nach dem Vertex-Shader, genannt "Post-Transform-Cache". Der "Pre-Tansform-Cache" beinhaltet vier Kilobyte und der Post-Transform-Cache kann drei oder mehr Vertices, die bereits die Vertex-Shader Information erhalten haben, beinhalten. Neuartige Grafikkarten können bereits ein viel größeren "Pre-Transform-Cache" besitzen. **[Lou07]**

2.5.3 Batches und Draw Calls

Ein Objekt besteht aus mehreren Vertices. Diese werden als Array von der CPU an die Grafikkarte übergeben. Das Array wird auch als Batch bezeichnet. Ein Quader beispielsweise besteht aus acht Vertices, diese bilden zusammen einen Batch.

In der Grafikpipeline werden 3D-Objekte von einem Verarbeitungsschritt in den Nächsten weitergeleitet. Die Applikation übergibt dem Grafikkartenspeicher über den I/O-Bus einen Batch von Vertices. Der Batch wird anschließend der Geometriephase übergeben. Um die im Shader definierte Oberflächenbeschreibung darstellen zu können, werden je nach Anforderungen an den Shader pro Batch mehrere Berechnungsaufrufe benötigt, die als Draw Call (DC) bezeichnet werden. Ein DC wird immer dann erzeugt, wenn die Grafik-API den Renderaufruf *glDrawElements* an die GPU schickt. Je komplexer die Oberflächenbeschreibungen werden, desto mehr DCs werden benötigt. Beispielsweise werden durch die Verwendung von Echtzeitschatteneffekten zusätzliche DCs verursacht. **[Wlo03]**

Wenn ein Batch der GPU übergeben wird, benötigt die CPU Berechnungszeit um die Informationen an die GPU zu senden. Der Speicher der Grafikkarte ist jetzt mit den Informationen des Batches gefüllt und die GPU beginnt diese zu berechnen. In der Zeit in der die Grafikkarte den Batch verarbeitet, kann kein weiterer Batch verarbeitet werden. Wenn die Grafikkarte schneller berechnet, als die CPU Informationen sendet, kommt es zu einem Performanceengpass, der durch die CPU verursacht wird. Performanceengpässe können somit zu jeder Zeit während der

[23] Ein Cache bezeichnet eine Methode, die Inhalte bei erneutem Laden schneller aufruft

Entwicklung der Software auftreten. Es sollten deshalb Performancemessungen in regelmäßigen Abständen durchgeführt werden. **[Wlo03]**

2.6 Stand der Technik: Performancemessungen

Das Ziel von Performancemessungen in der Informatik besteht darin, Informationen über ein System zu sammeln, mit denen es möglich ist, Aussagen über die Leistung dieses Systems zu treffen **[Sto09]**. Performance wird als allgemeiner Ausdruck für die Leistung eines Systems definiert, das aus Hardware und Software oder auch nur aus einem einzelnen Algorithmus bestehen kann. Die möglichen Parameter, die zur Performancebestimmung herangezogen werden können, sind sehr unterschiedlich. Oft wird der Datendurchsatz als Schlüsselparameter gewählt, d.h. als Maß für Performance ist die Menge der Daten relevant, die in einer bestimmten Zeit verarbeitet werden **[Sto09]**. Für die Performanceanalyse dieser Arbeit wird die Framerate als Maß für die Performance gewählt, da sie genau diese Bedingung erfüllt. Für das Testen kamerabasierter Fahrerassistenzsysteme darf eine Framerate von 30 fps nicht unterschritten werden, da die Kamerasysteme mit 30 Bildern pro Sekunde ihre Umgebung wahrnehmen.

Sobald Abhängigkeiten in einer Abfolge von Aktionen bestehen und somit eine Korrelation von verschiedenen Prozessen und Komponenten existiert, können Engpässe entstehen. Da jedes System anders aufgebaut ist und die Performanceengpässe somit immer an unterschiedlichen Stellen auftreten können muss die Performanceanalyse für jeden Anwendungsfall neu angepasst werden **[Wit08]**. Als Bottleneck-Analyse wird der Ansatz bezeichnet, der für Identifizierung von Engpässen in dynamischen Systemen bzw. dynamischen Prozessabläufen eingesetzt werden kann. Die Bottleneck-Analyse, hat sich dabei als wichtiger analytischer Schritt zur Prozessoptimierung in vielen Bereichen herauskristallisiert, wie z.B. bei der Beachtung von Rechtzeitigkeit und Gleichzeitigkeit bei der Entiwcklung von Systemsoftware. Durch die gesammelten Informationen lassen sich Entscheidungen über den Optimierungsansatz treffen.

In **[Wlo03]** und **[Fer04]** wird das Vorgehen für eine Performanceanalyse im Bereich der 3D-Echtzeitanwendung beschrieben. Die Lokalisierung des Engpasses umfasst ca. 50% der Performanceoptimierung **[Fer04]**. Wie in Kapitel 2.3 beschrieben, basiert der Renderprozess für ein Bild auf einer Pipelinearchitektur mit drei konzeptionellen Phasen: Applikation, Geometrie, Rasterung. Da die Abschnitte der Grafikpipeline voneinander abhängig sind, lässt sich die Bottleneck-Analyse für die Performanceanalyse anwenden.

Bei der Bottleneck-Analyse werden mehrere Testdurchläufe gestartet, bei denen jeweils genau ein Wert verändert wird. Anschließend wird verfolgt, ob sich die Framerate signifikant ändert. Wenn sich eine Änderung einstellt, kann dies als Performanceengpass bezeichnet werden. Die Phase, die am längsten zur Berechnung benötigt sowie die Kommunikation zwischen den einzelnen Phasen, werden immer einen Performanceengpass verursachen. Diese Phase gibt somit die Geschwindigkeit, mit der die Daten die Pipeline durchlaufen, vor. **[McG04]**

Wenn die Geschwindigkeit dieser Phase nicht mehr weiter optimiert werden kann, können die anderen Phasen der Grafikkarte zu ihrem vollen Potential ausgereizt werden. Die Performance wird dabei nicht verschlechtert, da die Geschwindigkeit der langsamsten Phase nicht überschritten wird, sondern die Berechnungszeiten können dazu genutzt werden die Qualität des Bildes zu verbessern. **[ATH08]**

Wird beispielsweise die Auflösung der Texturen oder alle Objekte mit demselben Shader versehen und sich dabei eine große Veränderung in der Framerate einstellt, wird durch diese ein Performanceengpass verursacht **[Wlo03]**. Dieser kann anschließend, durch die während der Performanceanalyse gewonnen Informationen, behoben werden. Wenn jedoch nach der Behebung des Engpasses die gewünschte Performancesteigerung ausbleibt, muss die Lokalisierung des Engpasses wiederholt werden **[ATH08]**. Zu Beachten ist, dass sich ein Engpass nach der Optimierung wieder an derselben Stelle befinden kann. Auch kann durch die Behebung eines Engpasses an einer anderen Stelle ein neuer Engpass entstehen **[Fer04]**.

Neben den Engpässen, die durch die Software verursacht werden, können Performanceengpässe auch durch die Hardware entstehen. Je nachdem, wie groß die Rechenleistung des Grafikprozessors und die Speicherkapazität der Grafikkarte ausfällt, werden die Berechnungszeiten beschleunigt oder verlangsamt. Dabei gilt, je neuer und schneller die Grafikkarte ist, desto größer ist die Performancesteigerung. **[ATH08]**

Die aus der Analyse gewonnen Informationen können dann zur Behebung der Performanceengpässe benutzt werden. Dafür werden bereits existierende Methoden verwendet, bzw. auch neue Methoden entwickelt werden. Jedoch ist zu beachten, dass durch die Entwicklung von neuen Methoden neue Performanceengpässe an anderen Stellen entstehen können **[Fer04]**.

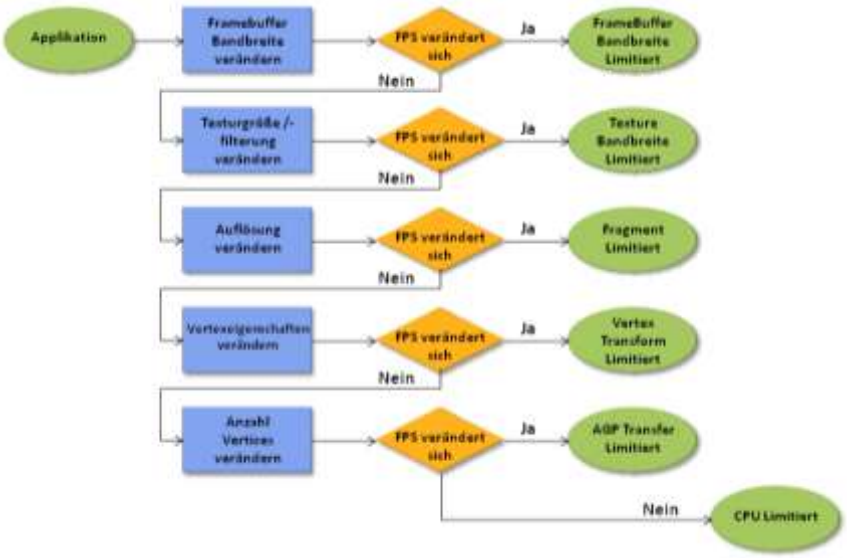


Abbildung 2.11: Engpass Flussdiagramm: Wenn sich eine Änderung in der Framerate einstellt, kann dieser Abschnitt als Performanceengpass bezeichnet werden. Angelehnt an [Fer04].

Abbildung 2.11 zeigt die Stellen, an denen Engpässe, die zu jedem Zeitpunkt während der Berechnung des Bildes, in der Grafikpipeline auftreten können. Die folgenden Abschnitte in der Grafikpipeline sind anfällig für Performanceengpässe: Applikation, Framebuffer, Textur, Fragment, Vertexeigenschaft, Vertex Anzahl, CPU. **[Fer04]** Für diese Performanceengpässe existieren unterschiedliche Möglichkeiten diese zu beheben. Einige Ansätze zur Performanceoptimierung finden sich in **[ATH08]**, **[Fer04]**, **[Wlo03]** und **[Rig02]**.

Zur Untersuchung von Performanceengpässen stehen für 3D-Echtzeitanwendungen eine Vielzahl an Tools zur Verfügung. Neben NVIDIA's PerfHUD[24] existieren weitere Performanceanalysetools, wie z.B. PIX[25], gDEBugger[26], ATI's GPU PerfStudio[27], Apple's OpenGLProfiler[28].

Zum Beispiel bietet NVIDIA's PerfHUD die Möglichkeit ein breites Spektrum an Daten, wie z. B. DCs, GPU- und CPU-Berechnungszeit, GPU-Benutzung, etc., über Diagramme zu visualisieren. So können auch einzelne DCs auf Berechnungszeit untersucht werden und exakt bestimmt wer-

[24] http://developer.nvidia.com/de
[25] http://msdn.microsoft.com/en-us/library/ee417062%28v=vs.85%29.aspx
[26] http://www.gremedy.com/
[27] http://developer.amd.com/gpu/PerfStudio/Pages/default.aspx
[28] http://developer.apple.com/graphicsimaging/opengl/profiler_image.html

den, welche Performanceengpässe verursachen. **[NVI08]** Die Abbildung 2.12 zeigt die Verwendung von PerfHUD in der Entwicklung des Spieltitels "Unreal Tournament 3" von Epic Games[29].

Abbildung 2.12: Performanceanalyse mit PerfHUD von "Unreal Tournament 3" (Quelle: Epic Games).

[29] www.epicgames.com

2.7 Zusammenfassung

Wie gezeigt, ist das Testen kamerabasierter FAS, da diese auch in die Fahrzeugführung eingreifen können, für die Serienfertigung zwingend erforderlich. Heutzutage werden kamerabasierte FAS durch vier unterschiedliche Verfahren getestet.

- Testfahrten
- Videotestmaterial
- Test mittels abstrakter Darstellungen
- Visual-Loop-Test

Jeder einzelne Test gewährleistet nicht die volle Testabdeckung, sondern es wird vielmehr eine Kombination dieser Verfahren benötigt. Beispielsweise können die Auswirkungen der FAS auf den Fahrer im Labor schwierig bis gar nicht simuliert werden, da sich dieser nicht in derselben Situation befindet, wie auf einer echten Teststrecke. Jedoch können Testfahrten nie unter den selben Bedingungen reproduzierbar. Der Visual-Loop-Test hat durch die stetige Verbesserung der realistischen Darstellung ein großes Potential in Zukunft eine weitreichende Testfallabdeckung zu gewährleisten.

Damit annähernd realistische Tests möglich werden, muss die Visual-Loop-Komponente dreidimensionale Objekte in Echtzeit darstellen können. Dabei greift die Software für die Darstellung auf die Pipeline der Grafikkarte zu. Die Daten werden von der Applikation an die Grafikkarte geschickt, dort verarbeitet und anschließend auf einem Medium, das von dem FAS ausgewertet wird, ausgegeben. Die Daten durchlaufen somit drei unterschiedliche Abschnitte.

- Applikation
- Geometrie
- Rasterung

In diesen Abschnitten kann es immer wieder zu Performanceengpässen führen. Deshalb ist eine regelmäßige Performanceanalyse bereits während der Entwicklung von Nöten. Hierzu bietet der heutige Stand der Technik Software, wie z. B. NVIDIA's PerfHUD, die Informationen über das untersuchte System bereit stellen. Dabei wird für das genaue Vorgehen die Bottleneck-Analyse verwendet, die sich für den Einsatz von 3D-Echtzeitanwendungen etabliert hat. So kann exakt bestimmt werden, an welchen Stellen Performanceengpässe auftreten, um diese dann gezielt zu beheben.

3 Performanceanalyse

Das Testen von kamerabasierten FAS ist für die Serienfertigung ein essentieller Faktor. Die Methoden, die hierzu zur Verfügung stehen, wurden im Kapitel 2.1.1 beschrieben. Vor diesem Hintergrund werden in diesem Kapitel zunächst die Anforderungen an das Konzept aufgestellt. Diese sollen im Konzept berücksichtigt und umgesetzt werden. Anschließend wird mit Hilfe einer Software zur Analyse von 3D-Echtzeitanwendungen eine Visual-Loop-Komponente hinsichtlich dieser Anforderungen untersucht. Es gilt mögliche Performanceeinbrüche zu lokalisieren und die Ergebnisse der Analyse zu klassifizieren und auszuwerten. Aufgrund dieser werden dann Methoden entwickeln, die die Anforderung an das Konzept und letztendlich auch die im Einführungskapitel definierten Ziele erfüllen.

3.1 Aus der Zielsetzung abgeleitete Anforderungen an das Konzept

Nachfolgend werden die Ziele (Z) und die daraus abgleitenden Anforderungen (A) an das Konzept vorgestellt. Damit exakt bestimmt werden kann, unter welchen Kriterien ein Performanceeinbruch entsteht, soll eine Performanceanalyse durchgeführt werden. Dazu muss zunächst eine Lokalisierung der Engpässe erfolgen. Nachdem diese lokalisiert und klassifiziert wurden, werden Methoden recherchiert und entwickelt, die zur einer signifikanten Steigerung der Performance führen. Dabei ist darauf zu achten, dass kein Funktionsverlust durch die Optimierung eintritt. Die Erfüllung der Anforderung A1.3 soll durch die flüssige Darstellung eines Beispiel-Szenarios „Verkehrsstau“ mit 1000 gleichzeitig sichtbaren Fahrzeugen nachgewiesen werden.

Z1: Optimierung der Performance insbesondere durch Anpassung der virtuellen Fahrzeuge

- → A1.1: Lokalisierung der Engpässe
- → A1.2: kein Funktionsverlust
- → A1.3: deutlich messbare Steigerung der Performance

Durch die Anforderung einer realistischen Darstellung der Umgebung an die Visual-Loop-Komponente und die Verwendung dieser im Automotive Bereich, muss dem Realitätsgrad von 3D-Fahrzeugen besondere Beachtung zu Teil werden. So soll neben der Optimierung der Performance auch eine Verbesserung der realistischen Darstellung erzielt werden. Die Anforderung

an dieses Ziel ist das Verhindern von erneuten Performanceengpässen durch die Implementierung und ein gesteigerter Realitätsgrad, der durch Fachkräfte bewertet wird.

Z2: Verbesserung der realistischen Darstellung von Fahrzeugen

- → A2.1: keine Verursachung von Performanceengpässen
- → A2.2: durch Fachkräfte bewerteter Anstieg des Realitätsgrades

Allerdings ist anzumerken, dass die definierten Anforderungen der Ziele Z1 und Z2 im Widerspruch zueinander stehen. Je detaillierter die 3D-Fahrzeugmodelle gestaltet werden, desto größer wird der Performanceverlust pro 3D-Modell werden. Die Performanceoptimierung darf die Darstellung der Fahrzeuge nicht so weit verschlechtern, dass es für das Fahrerassistenzsystem nicht mehr als solches erkannt werden kann. Im Gegenteil soll der Realitätsgrad weiter erhöht werden, um FAS umfassender Testen zu können. Hier ist ein geeigneter Kompromiss zwischen Z1 und Z2 zu finden und zu begründen.

3.2 Durchführung der Performanceanalyse

Für den folgenden Verlauf der Performanceanalyse fiel die Wahl auf NVIDIA's PerfHUD. Das Tool stellt ein breites Spektrum an Möglichkeiten zur Performanceuntersuchung zur Verfügung, wie z.B. der "Frame Profiler", mit dessen Hilfe Frames automatisch auf Engpasse untersucht werden können. Außerdem bietet PerfHUD, die Möglichkeit mit Hilfe der manuellen Analysemöglichkeiten eine gezielte Bottleneck-Analyse, wie sie in Kapitel 2.6 beschrieben ist, durchzuführen.

PerfHUD wird zusammen mit der zu untersuchenden Software gestartet. Bereits zu Beginn werden dem Benutzer über das "Performance Dashboard" mehrere Graphen angezeigt. (siehe Abbildung 3.1) Die Graphen beinhalten Informationen, wie z.B. DC-Anzahl, Framerate, etc., die sich von Frame zu Frame aktualisieren. Über ein vorher definiertes Tastaturkürzel lässt sich das Menü aktivieren.

Durch die Aktivierung des Menüs können die vorhandenen Graphen den Bedürfnissen angepasst oder neue Graphen hinzugefügt werden. Außerdem lassen sich über eine Menüleiste am unteren Rand die verschiedenen Methoden zur Performancemessung anwählen.

Abbildung 3.1: Analyse von PROVEtech:VL mit Hilfe von NVIDIA's PerfHUD: Das Performance Dashboard bietet die Möglichkeit Performanceveränderungen in Echtzeit zu analysieren.

Durch den Einsatz des Frame Profilers lässt sich exakt bestimmen, an welcher Stelle sich ein Engpass befindet und wie dieser konkret definiert ist. Der Frame Profiler ist das stärkste und effektivste Werkzeug um Engpässe zu lokalisieren. Durch mehrere Analyseschritte wird der Frame untersucht und anschließend Informationen über jeden einzelnen DC ausgegeben.

Für die Performanceanalyse wird die Simulationssoftware PROVEtech:VL von der MBtech Group verwendet. Damit die Engpässe exakt lokalisiert werden können, werden mehrere Testszenarien mit typischen Anwendungsfällen der Software erstellt. Da bei unterschiedlich leistungsfähiger Hardware verschiedene Ergebnisse vorliegen, muss darauf geachtet werden, dass sich das Testsystem nicht verändert und möglichst konstant bleibt. Zur Analyse wird ein Computersystem mit einem Intel Core2 3,2 Ghz, 8 GB Arbeitsspeicher und einer NVIDIA GTX 280 Grafikkarte verwendet.

In den folgenden Unterkapiteln werden unterschiedliche Testszenen erstellt und auf Performanceengpässe untersucht. Dabei wird das Testszenario in immer kleinere Testszenarien aufgespalten um schlussendlich genau bestimmen zu können, wo sich die Performanceengpässe befinden.

3.2.1 Testszenario

Für die Performanceanalyse wird ein Testszenario erstellt, das beim Testen von kamerabasierten FAS zum Einsatz kommen könnte. 3D-Objekte werden in PROVEtech:VL in verschiedene Kategorien, wie z. B. Fahrzeuge, Vegetation, Verkehr, etc. eingeteilt. Aus diesen Objektkategorien wird jeweils und mindestens ein 3D-Objekt ausgewählt, dass als Repräsentant für diese Kategorie steht. Bei der Platzierung der Objekte muss darauf geachtet werden, das diese sich im Sichtfeld der Kamera befinden, damit sie nicht durch das Kamerafrustum vom Renderprozess und somit von der Performanceanalyse ausgeschlossen werden. Für das Szenario wird eine Verkehrssituation inszeniert, wie sie täglich im normalen Straßenverkehr vorkommen kann. Um zu gewährleisten, dass keine Fehler in der Straßenbeschreibung existieren, wird auf eine mehrfach getestete XODR-Datei[30], die auf der Entwicklerseite[31] heruntergeladen werden kann, zurückgegriffen. Außerdem wird darauf geachtet, dass die Objektanzahl mit dem Vorkommen in der Realität vergleichbar ist.

Das Testszenario wird zusammen mit PerfHUD gestartet. Zunächst kann durch eine manuelle Analyse der Graphen das Performanceverhalten des Szenarios in Echtzeit betrachtet werden. Durch das Rotieren der Kamera wird sichtbar, dass die DC-Anzahl in unterschiedlichen Blickwinkeln der Testszene enorm variiert. Dies ist zunächst nicht verwunderlich, da die DC-Anzahl niedriger ausfällt, wenn sich weniger Objekte im Kamerablickwinkel befinden, weil sie durch das Kamerafrustum nicht in den Renderprozess einbezogen werden. In einem zweiten Schritt wird die Kamera so positioniert, dass sich alle Objekte im Sichtbereich befinden. Die DC-Anzahl steigt bei diesem Blickwinkel bis auf 3000 dpf (DCs pro Frame). Bei dieser Einstellung werden alle Objekte, die nicht durch das Level-of-Detail-System ausgeblendet werden, angezeigt. Dennoch ist die DC-Anzahl im Vergleich zu den 3D-Objekten (ca. 350), die platziert wurden, sehr hoch. Durch die manuelle Performanceanalyse kann jedoch nicht bestimmt werden, aus welchem Grund eine solch hohe DC-Anzahl zustande kommt.

Die in Kapitel 2.6 vorgestellte Methode, wird nun auch für das Testszenario angewandt. Dabei stehen dem Benutzer in PerfHUD mehrere manuelle Funktionalitäten zur Verfügung. Im Folgenden werden alle durchgeführten Tests aufgeführt und die Ergebnisse beschrieben.

[30] XODR beschreibt die Straßenlogik und basiert auf der Auszeichnungssprache XML
[31] http://www.opendrive.org/download.htm

Durch die Möglichkeit die Texturen sämtlicher Objekte durch Texturen mit der Größe 2x2 Pixel auszutauschen kann untersucht werden, ob der Performanceengpass durch die verwendeten Texturen verursacht wird. Die Framerate verändert sich durch diese Einstellung um ca. 0,3 fps. Aus diesem Grund können die Texturen als Performanceengpass ausgeschlossen werden.

Die Bildschirmauflösung kann ebenfalls zu einem Performanceengpass führen. Je größer die Ausgabe der Bildauflösung, desto mehr Pixel müssen berechnet werden und desto größer ist die Berechnungsdauer. Die Veränderung der Bildschirmauflösung bringt einen Performancegewinn von ca. 0,2 fps.

Wie in Kapitel 2.3.2 beschrieben, können einzelnen Vertices eine begrenzte Anzahl an Eigenschaften vergeben werden. Beispielsweise lassen sich Farbwerte oder die Ausrichtung der Normalen in den Vertex Eigenschaften speichern. Auch durch das Weglassen sämtlicher Vertexinformationen, die Position ausgeschlossen, verändert sich die Framerate nicht. Dennoch kann die Übertragungszeit der Vertices von der CPU zur Grafikkarte bereits einen Performanceengpass verursachen.

Die Reduzierung jedes Geometrieobjektes auf ein Triangle ermöglicht eine Analyse, bei der keine Berechnungskosten durch zahlreiche Vertices anfallen. Unter Verwendung dieser Einstellung sinkt die Framerate um ca. 0,5 fps. Aus dem Ergebnis lässt sich ableiten, dass der Performanceengpass nicht durch die Anzahl der Vertices verursacht wird, was im Umkehrschluss eine größere Vertex-Anzahl bei der Erstellung der Modelle zulässt.

Die Berechnungsdauer die der Alpha- und Z-Depth-Test verursachen, kann durch eine komplexe Szene einen Performanceengpass verursachen. Hier kann beispielsweise eine Verkürzung der Berechnungsdauer durch Berechnung der Tiefenkomplexität von vorne nach hinten erzielt werden. Durch diese Änderung fallen mehr Objekte durch den Z-Depth-Test und es werden weniger Objekte in den Framebuffer geschrieben. Durch das Anzeigen der Tiefenkomplexität der Szene kann ermittelt werden, ob es in der Szene zu Verdeckung von Geometrie kommt. Je länger dabei die Berechnungszeit auf der Grafikkarte ist, desto größer ist der Performanceengpass, der durch die Verdeckung entsteht. Da bei der verwendeten Grafikengine nicht auf den interenen Renderablauf zugegriffen werden kann, können in diesem Bereich keine Änderungen vorgenommen werden.

Bei allen manuellen Tests hat sich keine große Veränderung in der Framerate ergeben, was im Umkehrschluss auf einen Performanceengpass durch die CPU schließen lässt. In der folgenden Tabelle sind die Ergebnisse der manuellen Analyse zusammengefasst.

Tabelle 3.1 Ergebnis der manuellen Performanceanalyse

Art des Tests	Texturen	Bildschirmauflösung	Vertex Eigen-schaften	Ver-tex-Anzahl	Framebuffer	CPU
Änderung der Framerate	-0,3 fps	-0,2 fps	0 fps	ca. -0,5fps	---	---
Performance-engpass	Nein	Nein	Nein	Nein	---	Ja

Nach der ersten manuellen Analyse der Graphen wird mit Hilfe des Frame-Profilers die Szene automatisch auf Performanceengpässe analysiert. Mit der Möglichkeit die Performanceanalyse in einer CSV-Datei zu speichern, können die Ergebnisse gespeichert und anschließend in einem Diagramm formatiert werden.

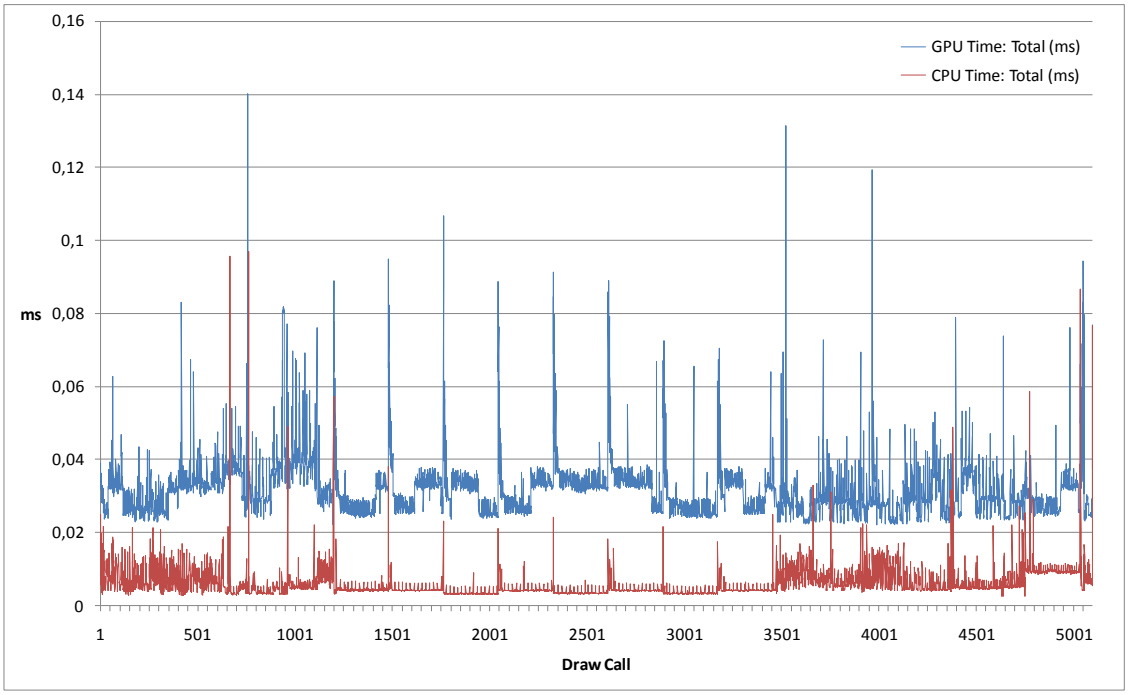


Abbildung 3.2: Analyse des Testszenarios mit Darstellung der GPU- und CPU-Leistung.

Deutlich auffallend ist die große DC-Anzahl, wie sie auch schon in der manuellen Performance-analyse zu sehen war. In Abbildung 3.2 ist die Analyse des Testszenarios für einen Frame dargestellt. Die Abszisse steht für die Anzahl der DrawCalls, während die Ordinate für die Berechnungsdauer steht. Durch die Analyse des Frame Profilers kann nun genau bestimmt werden, welches Objekt, wie viele DCs verursacht und wie lange diese berechnet werden. Dadurch fällt

deutlich auf, dass die Terrainlogik mehr DCs verursacht als angedacht. Neben dem Terrain fällt auf, dass die Fahrzeuge einen großen Anteil der Berechnungsdauer benötigen. Alle weiteren Objekte der Szene fallen von der Berechnungsdauer so niedrig aus, dass diese von der Berechnungsdauer im Vergleich zu dem Terrain und den Fahrzeugen nicht ins Gewicht fallen. Der Performanceengpass wird also eindeutig durch die Terrainlogik und die Fahrzeuge verursacht.

3.2.2 Terrain

Aus der Analyse des "Frame Profilers" geht hervor, dass das Terrain eine größere DC-Anzahl verursacht, als das Terrain aus Unity3D, auf welchem dieses basiert. Das Terrain verwendet ein sichtabhängiges LOD-System, bei dem Teile des Terrains, je weiter sie sich von der Kamera entfernt befinden, an Detailreichtum verlieren (näheres zu sichtabhängigem LOD findet sich in Kapitel 4.3.4).

Da auch die Möglichkeit bestehen soll, Vertiefung in das Terrain einzufügen, wurde die Höhe des Terrain, durch Änderung der Heightmap[32], auf einen fest definierten Wert angehoben. Je heller ein Pixel in der Heightmap definiert ist, desto höher wird das Terrain angehoben, je dunkler, desto tiefer wird das Terrain abgesenkt. Zwei Ränder des Terrains bleiben jedoch bei der Anhebung der Heightmap auf der ursprünglichen Höhe des Terrains. Dabei wird fälschlicherweise eine Verbindung von dem Grund des Terrains auf den Wert, den die Height-Map vorgibt, gezogen und mit Geometrie gefüllt (vgl. Abbildung 3.3).

Die Terraingeometrie ist in viele Teilobjekte gegliedert, die jeweils als Batch der GPU übergeben werden müssen. Dieser Vorgang benötigt bereits einen enormen Rechenaufwand. Da das Terrain auch Schatten empfängt, werden pro Terrainabschnitt mehrere DCs verursacht. Durch den Fehler in der Terrainlogik werden anstatt insgesamt 9 DC für das gesamte Terrain unnötiger Weise ca. 3250 verursacht. Die Berechnungsdauer pro Frame beträgt ~82,3 ms. Der Fehler in der Terrainlogik ist auf einen falsch eingestellten Wert zurückzuführen. Die Behebung dieses Problems soll nicht Teil der Arbeit sein.

[32] Eine Heightmap ist eine Textur, die über die Farbwerte Höheninformationen in der Terrainlogik definiert.

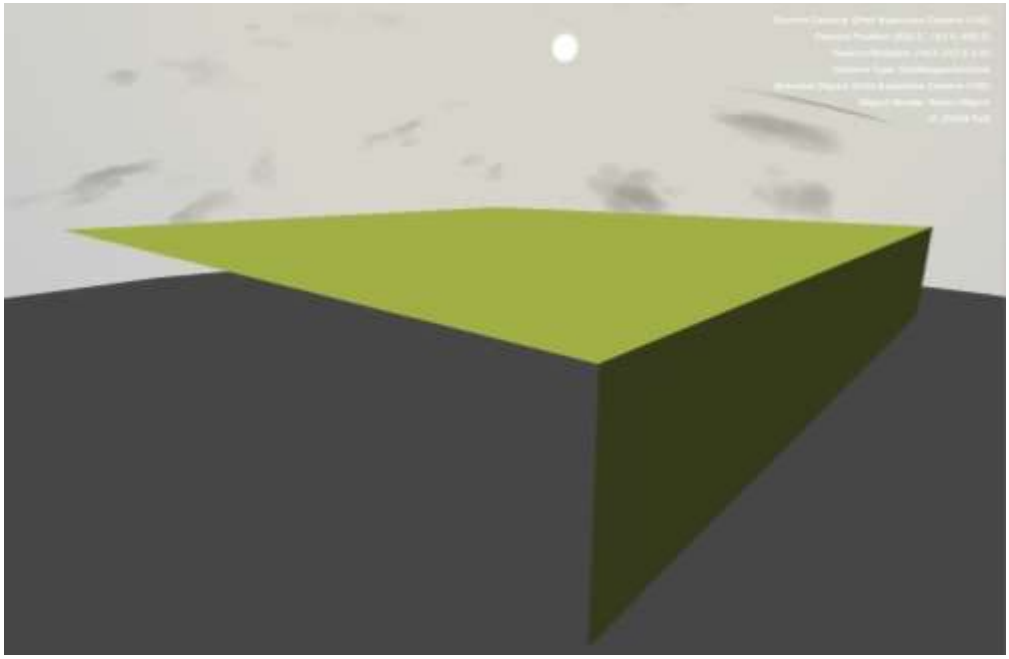


Abbildung 3.3: Fehler in der Terrainlogik (Quelle MBtech).

3.2.3 Fahrzeuge

Neben der Terrainlogik ergibt die Performanceanalyse des Testszenarios einen deutlichen Performanceengpass, der durch die 3D-Fahrzeuge verursacht wird. Die Fahrzeuge verursachen insgesamt 1652 DCs und eine Berechnungsdauer von ~42,7 ms.

Bereits durch das Platzieren von vierzig Fahrzeugen in einer Szene ohne weitere Objekte sinkt die Framerate auf unter 25 fps. Da jedoch beim Testen von kamerabasierten FAS auch Testfälle mit mehreren hundert Fahrzeugen, wie z.B. Stausituationen, getestet werden müssen, ist eine Performanceoptimierung zwingend erforderlich.

Das Fahrzeugmodell "mercedes_01.bundle" aus der Objektbibliothek von PROVEtech:VL dient als Testobjekt für die Fahrzeuganalyse. Das Fahrzeug bietet einen guten Durchschnitt in der Objekt- und Vertex-Anzahl. Damit äußere Faktoren, wie beispielsweise das Terrain, die Analyse nicht verfälschen, wird das Fahrzeug in Unity3D als separates Programm erstellt und kann so mit Hilfe von PerfHUD analysiert werden. Abbildung 3.4 zeigt das Diagramm der Fahrzeuganalyse.

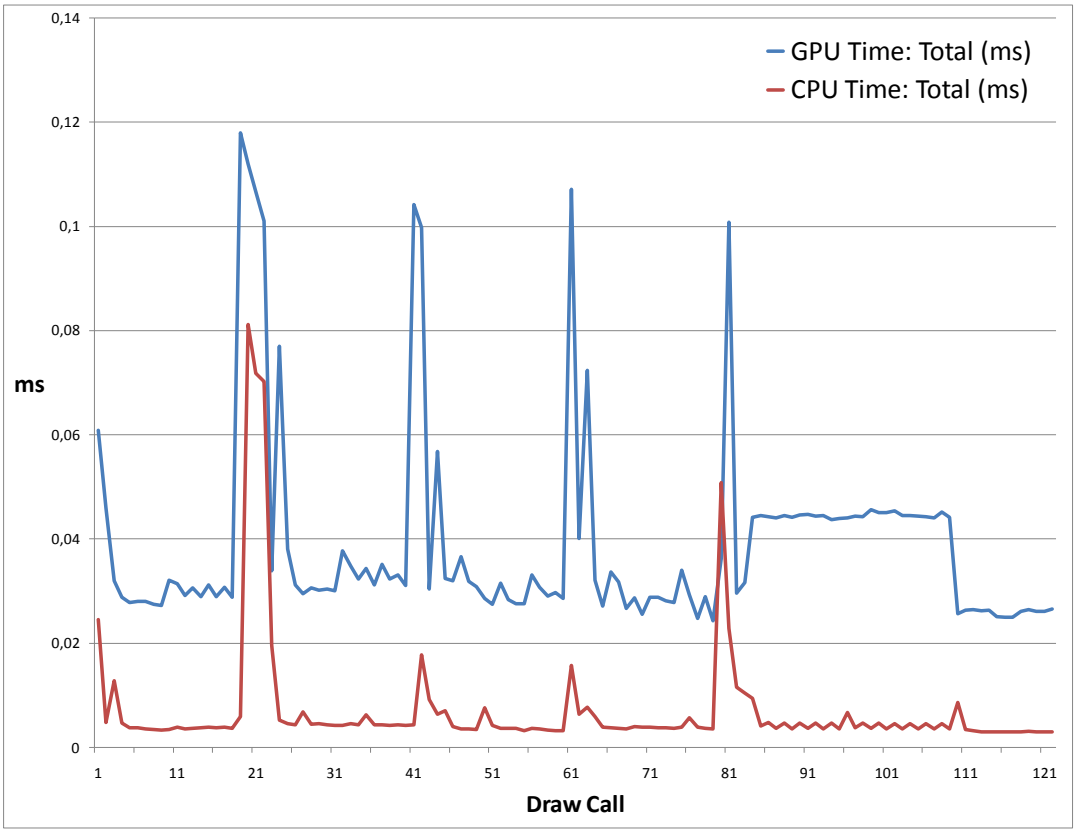


Abbildung 3.4: Analyse von einem Fahrzeug in einer neutralen Testumgebung.

Bei der ersten Betrachtung kann eine Regelmäßigkeit in der Berechnung festgestellt werden. Die Regelmäßigkeit kommt durch mehrere Berechnungsdurchläufe jedes einzelnen Objektes zustande. Zunächst wird der Schattenwurf des Fahrzeugs, anschließend wird das Empfangen der Schatten anderer Objekten auf der Objektoberfläche des Fahrzeugs berechnet. Bei den beiden letzten Durchläufen werden die Objekte gezeichnet und das Beleuchtungsmodell angewendet. Dabei verursacht der Shader, je nach Komplexität, mehrere DCs pro Objekt. In diesem Fall werden durch den Shader zwei DCs verursacht, ein DC für den Farbwert und ein DC für die Darstellung von Lichteffekten auf der Geometrieoberfläche. Die vier hohen Peaks werden durch das Karosserieobjekt des Fahrzeugs verursacht, da dieses im Vergleich zu den anderen Objekten die meisten Vertices besitzt und somit rechenintensiver ist, da eine größere Datenmenge von der GPU berechnet werden muss. Im letzten Berechnungsschritt werden transparente Objekte, wie die Leuchtkugeln (Halos) der Lichter oder die Gläser der Scheinwerfer gerendert. Die Berechnungszeit setzt sich also aus den einzelnen Fahrzeugkomponenten zusammen. Dabei sind einige Objekte, wie beispielsweise die Karosserie rechenintensiver als ein Halo. Durch die Maße

der Objekte summiert sich jedoch die Berechnungszeit und führt somit zu einem Performanceengpass.

Insgesamt werden bei voll eingeschalteten Scheinwerfern 123 DCs benötigt, um das Fahrzeug darzustellen. Die Berechnungszeit auf der Grafikkarte beträgt insgesamt ~4,13 ms pro Fahrzeug.

3.3 Ergebnis der Analyse und Zusammenfassung

Die Performanceanalyse macht deutlich, dass sichtlich Engpässe vorhanden sind. Durch die manuelle Analyse kann festgestellt werden, dass der Performanceengpass nicht von der Grafikkarte, sondern einzig von der Menge an DCs abhängt, die von der CPU zur Berechnung geschickt werden. Dadurch, dass die Grafikkarte die DCs schneller berechnet, als diese von der CPU geschickt werden können, wird ein Performanceengpass verursacht. Mit Hilfe der automatischen Analyse kann genau nachvollzogen werden, welche Objekte diese DCs im Speziellen verursachen. Das Terrain und die Fahrzeuge fallen dabei im Vergleich zu allen anderen Objekten deutlich als Ursache für die Performanceengpässe auf. Damit ist die Anforderung A1.1 (Lokalisierung der Engpässe) erüllt.

Das Terrain verursacht insgesamt ca. 3300 DC und benötigt ~82,3 ms Berechnungsdauer auf der Grafikkarte, die durch einen Fehler bei der Ermittlung der Terrainhöhe entstehen. Dieser Fehler kann durch eine Korrektur im Skript behoben werden.

Die Fahrzeuge hingegen benötigen einer genauen Untersuchung, da der Performanceengpass nicht durch einen Fehler entsteht, sondern bereits im Modellierungskonzept der Fahrzeuge begründet liegt. Zum Beispiel könnten anstatt einem Fahrzeug ca. fünfzig Verkehrsschilder, jedes einzelne bestehend aus zwei DCs, platziert werden, ohne dass sich die Framerate verändern würde. Für den weiteren Verlauf der Arbeit sollen deshalb Methoden recherchiert und entwickelt, die genau diesen Performanceengpass lösen und dabei eine Darstellung von mehreren hundert Fahrzeugen ermöglichen.

4 Optimierung der 3D-Fahrzeugmodelle

Aus Kapitel 3 ist hervorgegangen, dass die 3D-Fahrzeuge einen großen Performanceengpass verursachen. In diesem Kapitel wird zunächst der Stand der Technik von Methoden zur Behebung des Performanceengpasses recherchiert und anschließend ein Konzept für eine Implementierung entwickelt. Dabei sollen nicht nur die Anforderungen A1.2 (kein Funktionsverlust) und A1.3 (deutlich messbare Steigerung der Performance) berücksichtigt werden, sondern auch Möglichkeiten zur Umsetzung gefunden werden.

Für die Reduzierung der DCs gibt es bereits mehrere Ansätze. Diese bewirken meist, dass die Objektanzahl in der Szene reduziert wird. LOD-Systeme sind einfach zu realisierende und effektive Methoden um die DC-Anzahl zu reduzieren. Im Laufe dieses Kapitels soll näher auf den LOD-Ansatz eingegangen und die Vor- und Nachteile der unterschiedlichen LOD-Arten diskutiert werden. Weiterhin kann bereits bei der Erstellung der Fahrzeuge auf die Objektanzahl geachtet werden. Objekte, die nicht transformiert oder verändert werden müssen, sollten immer zu einem Objekt zusammengefasst werden. Wie bereits in Kapitel 3.2 beschrieben können aus der Behebung eines Engpasses neue Probleme entstehen.

4.1 Batch-Reduzierung

Ein Fahrzeug besteht aus mehreren Unterobjekten, die jeweils als einzelne Batches an die Grafikkarte geschickt werden. Durch die hohe Anzahl an Batches pro Fahrzeug entsteht ein Performanceengpass, der maßgeblich durch die CPU verursacht wird. Die CPU benötigt für das Senden eines Batches Rechenleistung, die sich summiert, je mehr Objekte geschickt werden müssen. Eine hohe Anzahl an Batches erhöht somit die Zeit, die benötigt wird einen Frame darzustellen und führt schlussendlich zu einer niedrigeren Framerate.

In der Messung **[Wlo03]** von Matthias Wloka (vgl. Abbildung 4.1) wurden mehrere Millionen Triangles erstellt und in ansteigender Batchgröße aufgeteilt. Je mehr Triangles in einen Batch gelegt werden, desto größer ist die Anzahl an Triangles, die verwendet werden kann. Es fällt auf, dass bis zu einer Anzahl von ca. 130 Triangles pro Batch alle Grafikkarten dieselbe Kurve beschreiben.

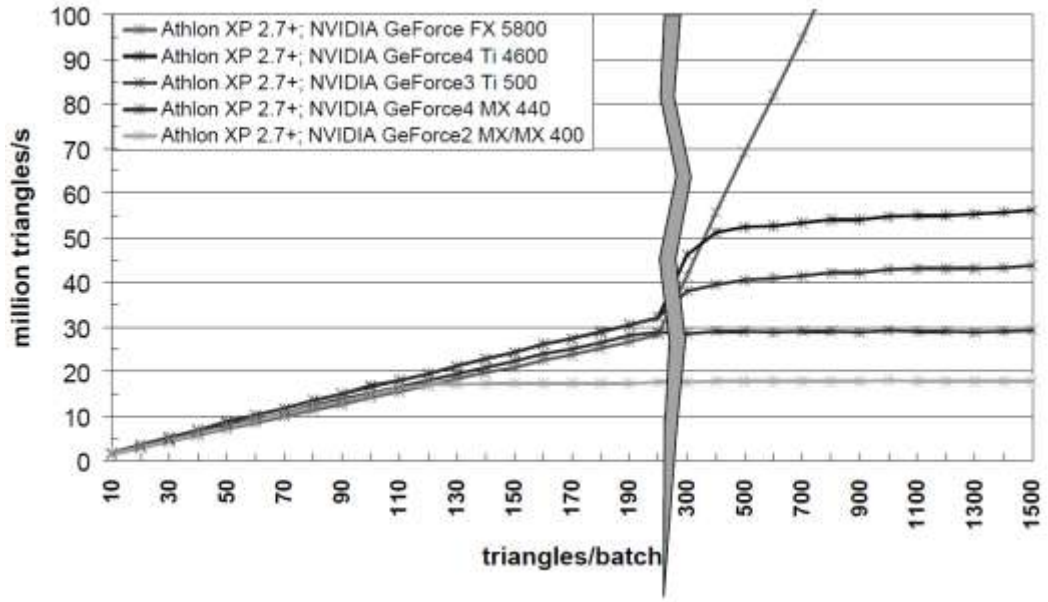


Abbildung 4.1: Messung von Matthias Wloka [Wlo03].

Dieser Schwellenwert entsteht, weil die Dauer für das Senden eines Batches von der CPU höher ist, als die Berechnungsdauer der Grafikkarte für einen Batch. Die Anzahl der Triangles wird daher nicht von der Grafikkarte, sondern hauptsächlich von der CPU beschränkt. Somit hängt die Anzahl der Batches, die pro Frame ausgeführt werden können, zunächst von der Leistung und der Auslastung der CPU und zum anderen von der geforderten Framerate ab. Mit steigender Triangleanzahl pro Batch wird die Berechnungsdauer der Grafikkarte verlängert.

Diese Erkenntnis ist in der Formel von Wloka zusammengefasst:

$$X = \frac{BCU}{F}$$

B bezeichnet die Anzahl an Batches pro Sekunde für eine CPU mit 1 GHz Rechenleistung, *C* ist die GHz Rate der gegenwärtigen CPU und *U* den Prozentsatz der CPU, der für den Batch zur Verfügung steht. Das Produkt dieser Werte teilt sich durch die angesetzte Framerate *F.* Das Ergebnis *X* gibt den Schwellenwert, an dem die Dauer für das Senden der Batches gleich der Berechnung durch die Grafikkarte ist, an. Durch seine Messungen gibt Wloka *B* als eine Konstante von 25.000 Batches pro Sekunde für einen 1 GHz CPU bei 100% Auslastung an. **[Wlo03]**

Beispiel: Batches pro Frame anhand der Testsystemspezifikation, die für die Performanceanalyse zur Verfügung steht.

$$X = \frac{25000 \frac{batches}{GHz} \times 3.7GHz \times 0{,}30\ \%}{30\ fps} = 925\ \frac{batches}{frame}$$

Für die Konfiguration, CPU-Nutzung, und Zielspezifikationen wird der Schwellenwert durch die CPU auf 925 Batches, die pro Frame zur Grafikkarte gesendet werden, festgelegt.

Die Leistung der CPU lässt sich durch Austauschen mit der neusten Hardware erhöhen. Da die Leistung von Grafikprozessoren jedoch in der Entwicklung im Vergleich zu Computerprozessoren deutlich schneller voranschreitet, sollte eher die Triangle Anzahl pro Batch, als die Anzahl der Batches an sich erhöht werden, um die Übertragungsdauer von der CPU möglichst niedrig zu halten. **[Wlo03]**

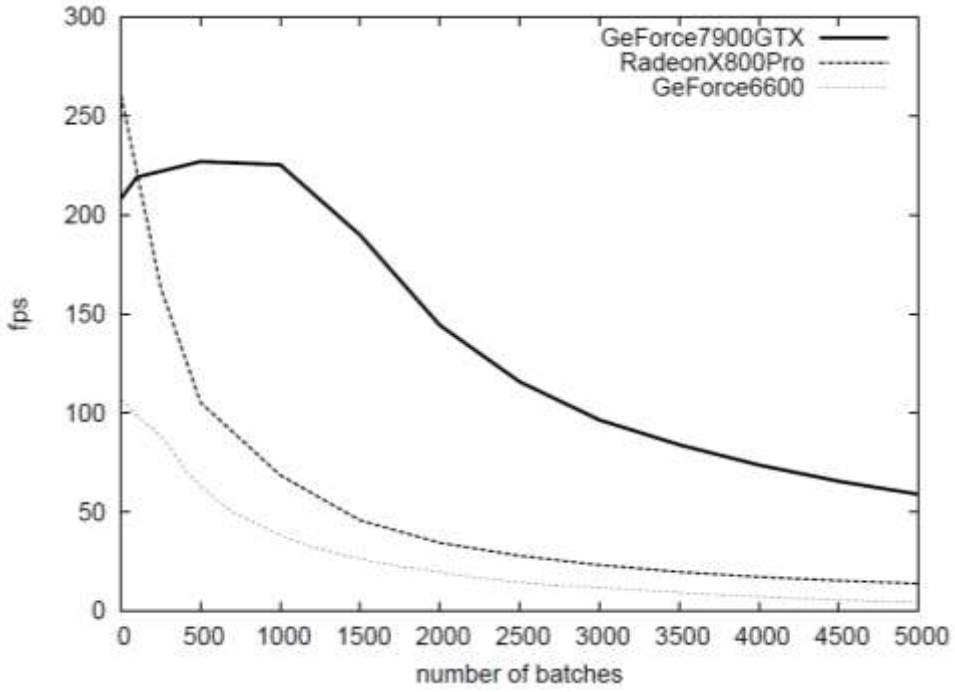


Abbildung 4.2: Rendern von 500,000 Traingles mit einer Aufteilung in unterschiedlich viele Batches [Lou07].

Für Abbildung 4.2 aus **[Lou07]** wurde eine Szene mit insgesamt 500.000 Triangles erstellt und in mehreren Testdurchläufen in unterschiedlich große Batches eingeteilt. Anschließend wurde für jede Batchgröße die Framerate gemessen. Es wird deutlich sichtbar, dass die Verwendung von wenigen Batches eine höhere Framerate ermöglicht. Für die GeForce7900GTX liegt die optimale Batchanzahl zwischen 500 und 1000.

Die Anzahl an Batches pro Frame kann durch Zusammenfügen von 3D-Objekten reduziert werden. Dies kann bereits bei der Erstellung der 3D-Objekte beachtet werden, aber auch durch automatisches Zusammenfügen in der Applikation erzielt werden. Beispielsweise lassen sich Objektinstanzen, d.h. Kopien eines Grundobjektes, zu einem Objekt zusammenfassen und können somit als Batch von der CPU an die Grafikkarte übergeben werden. Dennoch gibt es Ausnahmen, die das Aufteilen in mehrere Batches verhindern. Wenn Batches zusammengefasst werden, muss auch die Texturinformation erhalten bleiben. Diese kann, wenn es sich um unterschiedliche Objekte mit verschiedenen Texturen handelt, zu einer Textur zusammengefasst werden. Das Vorgehen funktioniert jedoch nur dann, wenn sich die Textur auf der Geometrie nicht wiederholt. Außerdem kann durch das automatische Zusammenfügen nicht genutzter Platz auf der Textur entstehen, da diese oft durch automatisierte Vorgänge nicht optimal ausgenutzt wird. Durch die Speicherbegrenzung der Grafikkarte kann die Textur nicht beliebig groß gestaltet werden. So kann durch das Zusammenfügen der Texturen ein Qualitätsverlust verursacht werden. Weiterhin lassen sich Batches, die zusammengefasst wurden, nicht mehr einzeln anwählen, um beispielsweise Transformationen durchführen zu können. 3D-Objekte, die durch den Benutzer anwählbar sind, sollten aus diesem Grund als separate Batches bestehen bleiben.

4.1.1 Zusammenfügen der Fahrzeugobjekte

Aus den Messungen von **[Wlo03]**, **[Lou07]** und der Performanceanalyse kann geschlossen werden, dass die Batchanzahl der Fahrzeuge drastisch reduziert werden muss. Durch die Anforderung A1.3 ist die Darstellung einer großen Anzahl an Fahrzeugen pro Frame gefordert. Diese kann nur durch eine Reduzierung der Batches pro Fahrzeug erreicht werden.

Die derzeitigen 3D-Fahrzeugmodelle bestehen aus ca. 30 Objekten, wie zum Beispiel Lichter, Karosserie, Plastikverkleidung. Diese Unterobjekte lassen sich durch die verwendeten Materialien in unterschiedliche Kategorien einordnen.

- Diffuse, wie z.B. die Plastikverkleidung
- Spiegelnd, wie z.B. die Karosserie oder verchromte Felgen
- Transparent, wie z.B. die Scheiben
- Leuchtend, wie z.B. die Scheinwerfer

Jedes Objekt, das ein anderes Material besitzt wird als separates Objekt gehalten. Außerdem müssen Objekte, die durch die Anwendung verändert werden können, wie bspw. das Rotieren der Räder oder das Anschalten der Scheinwerfer, als einzelne Objekte erstellt werden. So wird durch das Skript pro Fahrzeug eine Mindestanzahl von 11 Unterobjekten benötigt. Diese beinhalten die Karosserie, vier Reifen und sechs Lichter. Es werden zusätzliche DCs durch die Schattenberechnung und die Berechnung von Oberflächenbeschreibungen, wie z.B. Reflektionen verursacht.

Durch Zusammenfügen zu einem 3D-Objekt kann diesem nur noch ein Material zugewiesen werden. Da jedoch ein Fahrzeug mehrere unterschiedliche Oberflächen besitzt, können diese nicht mehr durch mehrere Materialien, wie es bisher gehandhabt wurde, beschrieben werden. Ein speziell für diesen Anwendungsfall geschriebener Shader, kann die Anforderungen an das Material erfüllen. Der Shader muss die Möglichkeit bieten, über Texturen das Oberflächenverhalten zu bestimmen. Über eine Textur können dem Fahrzeug unterschiedliche Farbwerte zugewiesen werden. So kann beispielsweise dem Frontscheinwerfer eine andere Farbe zugewiesen werden, als der Karosserie. Die unterschiedlichen Farbwerte, die das Fahrzeugmodell, bevor die Objekte zusammengefasst wurden, besessen hat, können jetzt mit Hilfe der Textur wieder dargestellt werden.

Durch Berechnung von sogenannten Lightmaps lässt sich die Qualität der Darstellung noch weiter steigern. Dabei werden Licht- und Schatteneffekte mit der Farbtextur verrechnet. Eine solche Lightmap ist in Abbildung 4.3 dargestellt und im Rahmen der Implementierung entstanden.

Abbildung 4.3: Lightmap/Textur: dient dazu die realistische Darstellung des Fahrzeugs zu steigern.

Jedoch stellt sich durch die Verwendung von Texturen ein Funktionsverlust für den Benutzer ein. Dieser kann nun nicht mehr ohne weiteres die Farbe der Karosserie anpassen, da nur noch ein Material existiert und die Farbänderung das gesamte Fahrzeug betreffen würde. Das Hinzufügen einer weiteren Textur kann dieses Problem lösen. Durch diese können Bereiche in der Textur bestimmt werden, die über eine Farbauswahl eingefärbt werden. Dadurch dass eine Textur aus insgesamt vier Kanälen (Rot, Grün, Blau und Alpha) besteht, kann die "Farbauswahlstextur" in den Alphawert der Farbtextur gelegt werden. Über das Multiplizieren mit einem Farbwert, der über einen Dialog ausgewählt werden kann, kann die Farbe des Fahrzeuges wieder angepasst werden.

Durch die Verwendung von Texturen stehen bei der Erstellung der Objekte weitere Möglichkeiten zur Verfügung. Mit Hilfe des sogenannten Normal-Mappings lassen sich Objekte detailierter darstellen. Dabei werden die Details eines hochaufgelösten 3D-Modells auf ein niedrig Aufgelöstes projiziert. Die daraus resultierende Textur stellt Schatteninformationen auf dem niedrig aufgelösten 3D-Modell dar und ermöglicht dadurch das Darstellen von Vertiefungen und Erhöhungen. **[Coh99]** In Abbildung 4.4 ist eine Normalmap beispielhaft dargestellt.

Abbildung 4.4: Das Fahrzeug (links) kann durch Zuweisen der Normalmap (mitte) im Detailreichtum deutlich erhöht werden (rechts).

4.1.2 Schattenobjekt

Jedes Geometrieobjekt besitzt die Eigenschaft Schatten zu werfen oder zu empfangen, die sich bei der Konfiguration an- bzw. abwählen lässt. Durch die Aktivierung der Schatteneffekte wird die DC-Anzahl verdreifacht. Ein Fahrzeug, das aus 20 Objekten besteht, verursacht somit ca. 60 DCs, was zu einem enormen Performanceengpass führt.

Da der Schatten der Fahrzeuge nur die Umrisse des Fahrzeugs darstellt, kann die Verwendung eines Schattenobjektes zur Reduzierung der DC-Anzahl verwendet werden. Das Schattenobjekt besitzt dieselbe Form wie das Fahrzeug, besteht dabei aber aus weit weniger Polygonen. Der Schattenwurf der Fahrzeugmodelle wird deaktiviert und ab jetzt durch das Schattenobjekt berechnet. Somit verringert sich die DC-Anzahl auf ca. 40 DCs, da das Fahrzeug weiterhin Schatten empfangen soll.

Weiterführende Literatur zu Schattenberechnung in 3D-Echtzeitanwendungen findet sich in **[McG04]**.

4.2 Fahrzeuglichter

Farbe und Intensität der Lichter können über das Fahrzeug-Skript angepasst werden. Dabei wird das Material des Lichtglanzobjekts (Halo), je nachdem welchen Wert der Benutzer für das Fahrzeug festlegt, in der Intensität der Leuchtkraft erhöht. Diese Implementierung ist für die Berechnung in Echtzeit sehr kostenintensiv. Für jedes Licht, das interaktiv anwählbar sein soll, muss ein Halo-Objekt platziert werden. Insgesamt werden durch das Skript sechs Lichtobjekte (Frontlichter, Rücklichter, Bremslichter, Rückfahrscheinwerfer, zwei Fahrtrichtungsanzeiger) verlangt. Die jeweiligen Halo-Objekte werden an die vorher festgelegt Transformpunkte platziert und erhalten den Farbwert, des zugehörigen Lichtes.

Durch das Zusammenfügen der verschiedenen Objekte, wie in Kapitel 4.1 beschrieben, kann der Farbwert aus den Lichtern nicht mehr ausgelesen werden, da das Fahrzeug nur noch aus einem Material besteht und Farbwerte über eine Textur definiert werden.

4.2.1 Halo-Objekt

Derzeit ist das Leuchten der Fahrzeuglichter durch sogenannte Halo-Objekte realisiert. Diese sind mit einem speziellen Shader versehen, der den Glüheffekt darstellt. Die Intensität des Glühens lässt sich über einen Regler im Material, die Farbe über eine Farbauswahl einstellen.

Da die Halo-Objekte aus Geometrie bestehen und somit als zusätzlicher Batch der Grafikkarte übergeben werden, ist die Verwendung nur bedingt sinnvoll. Für die Performanceoptimierung können Batches durch das Ausblenden nicht aktivierter Halo-Objekte reduziert werden. Außerdem kann durch Zusammenfügen der jeweiligen Lichtpaare (rechter/linker Scheinwerfer) die Batchanzahl noch mal um die Hälfte verringert werden.

Bei der Rotation mit der Kamera um das Fahrzeug wird ein Fehler sichtbar, der durch die Geometrieüberschneidung der Halo-Objekte mit der Motorhaube verursacht wird (vgl. Abbildung 4.5). Diese Effekte fallen deutlich auf und wirken unrealistisch.

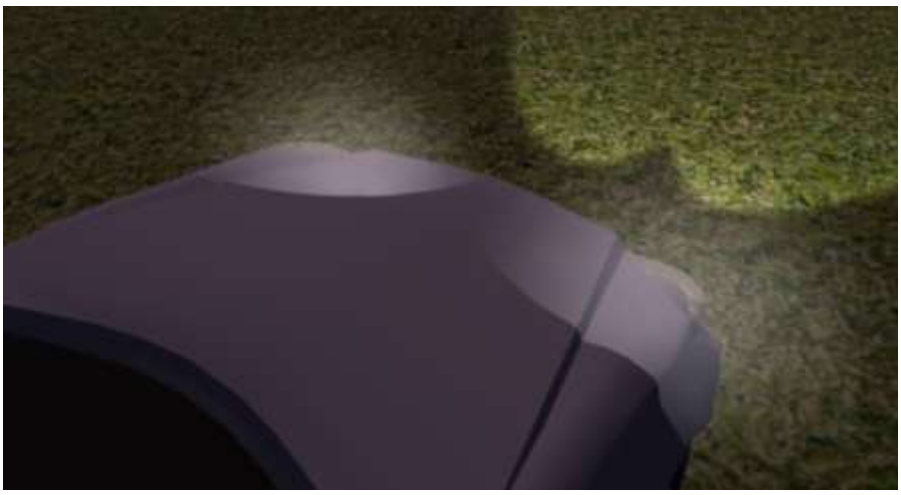


Abbildung 4.5: Fehler in der Darstellung des Halos (Quelle: MBtech).

4.2.2 Post-Processing-Effekt

Die Leuchteffekte können jedoch auch auf anderem Wege erzeugt werden. In der Rasterung der Grafikpipeline lassen sich durch den Pixelshader Effekte, wie z. B. Bewegungsunschärfe, Glüheffekte oder Tiefenunschärfe ohne großen Implementierungs- und Rechenaufwand erzeugen. Für die Realisierung der Fahrzeuglichter bietet sich ein Glüheffekt an.

Der erste Schritt um einen Glüheffekt zu rendern, ist zu definieren, welche Objekte oder Teilobjekte glühen sollen. Es können ganze Geometrieteile als Quelle des Glühens verwendet oder aber mit Hilfe von Texturdaten nur bestimmte Stellen der Geometrie bestimmt werden. Die Textur wird dabei als Maske benutzt, die definiert an welchen Stellen das Objekt glühen soll. **[Jam04]**

Die Maske kann als separate Textur gehalten oder auch in den Alphakanal des Farbwertes des Objektes eingebunden werden. Wenn nun der Glüheffekt gerendert werden soll, wird der Alphawert mit der normalen Farbtextur multipliziert. An den Stellen, wo der Alphawert 0 ist, wird kein Glühen dargestellt, ist jedoch der Wert größer 0, steigt auch die Intensität des Glühens. Nachdem die Stellen an denen, das Objekt glühen soll, ausgewählt sind, wird diese Textur weichgezeichnet. Durch den Weichzeichner wird die Bildschärfe herabgesetzt. Die weichgezeichnete Texturmaske wird nun mit der Farbtextur multipliziert und somit der Glüheffekt geschaffen. Für jede Iteration des Weichzeichners wird ein DC benötigt. **[Jam04]**

Abbildung 4.6: Post Processing Effekt: der Glüheffekt wird nachdem das Bild berechnet wurde nachträglich eingefügt.

Da der Glüheffekt einmal pro Frame und nicht pro Fahrzeug berechnet wird, verringert sich die DC-Anzahl D um $n + i$ wobei n die Anzahl der sichtbaren Halo-Objekte repräsentiert und i die Anzahl der Iterationen des Weichzeichners. Außerdem bleiben die Fehler bei der Geometrieüberschneidung durch die Halo-Objekte aus, da keine Geometrie gezeichnet wird, sondern der gerenderte Frame angepasst wird. Abbildung 4.6 zeigt ein Beispiel für die Verwendung eines Post-Processing-Effekts für die Fahrzeuge von PROVEtech:VL.

4.3 Level of Detail

Eine weitere Möglichkeit die DC-Anzahl aber auch die Vertex-Anzahl zu reduzieren, ist die Verwendung von Level-of-Detail-Systemen (kurz LOD). Ein Objekt, das sich in der Ferne befindet, benötigt einen geringeren Detailgrad, als ein Objekt, das sich direkt vor der Kamera befindet, da es weniger Pixel auf dem Bildschirm einnimmt. Mit Hilfe von LOD-Systemen werden Objekte unter bestimmten Auswahlkriterien in ihrem Detailgrad reduziert oder erst dann angezeigt, wenn der Benutzer diese auch wirklich sieht. Durch die hohe Anzahl an Polygonen pro Fahrzeug bietet sich die Implementierung eines LOD-Systems an. Da im Laufe der Zeit unterschiedliche LOD-Varianten entwickelt wurden, muss anhand von Kriterien entschieden werden, welches System sich für die Implementierung eignet.

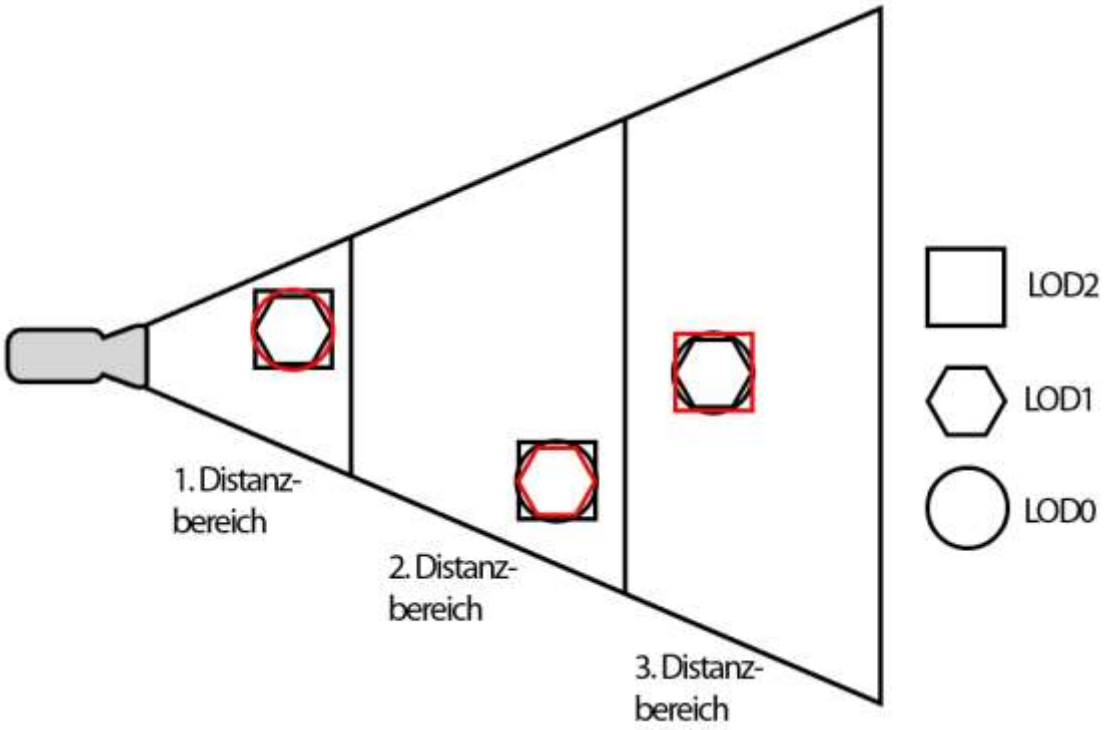


Abbildung 4.7: Diskretes LOD-System mit drei unterschiedlichen Distanzbereichen, denen jeweils eine LOD-Stufe zugeordnet ist. Angelehnt an [Feu10].

Die Grundidee eines LOD-Systems besteht immer in der Vereinfachung von Objekten, sei es durch Austauschen durch ein niedriger aufgelöstes Objekt oder durch Simplifikation der Objektgeometrie in Echtzeit. Im Allgemeinen besteht ein LOD Algorithmus aus drei Abschnitten: Erstellung, Selektion und Austausch. In dem Abschnitt der LOD Erstellung werden die verschiedenen Darstellungen eines 3D-Objekts in unterschiedliche Detailgrade unterteilt. Dabei können die Detailgrade der Objekte in einer separaten 3D-Software oder durch bestimmte Algorithmen automatisch generiert werden. Der Auswahlmechanismus (Selektion) wählt anschließend basierend auf Auswahlkriterien, wie bspw. der Abstand zur Kamera oder der Größe des Objektes auf dem Bild, die geforderte Detailstufe aus. Abschließend muss der Wechsel (Austausch) von einer Detailstufe zur Nächsten gewährleistet werden. In Abbildung 4.7 ist ein solches LOD-System schematisch dargestellt. Dabei wurden drei LOD mit unterschiedlicher Form erstellt. Der Auswahlmechanismus selektiert die LOD nach der Distanz zur Kamera. Der Austausch der LOD tritt jeweils bei dem Übergang des LOD in einen anderen Distanzbereich der Kamera auf. **[Feu10]**

Während der Fokus bei dieser Methode in der Auswahl unter verschiedenen geometrischen Darstellungen liegt, kann die Idee des LOD auch auf andere Aspekte des Modells verwendet werden. 3D-Objekte niedriger Detailstufen können einfachere Shader und Texturen von minderer Qualität benutzen, wodurch weitere Speicherbelegung und Rechenzeit gespart werden kann. Shader können sich selbst, abhängig von der Entfernung, der Wichtigkeit oder anderen Faktoren vereinfachen.

Weiterführende Information zu dem Thema Level of Detail kann in **[Lue03]** gefunden werden.

4.3.1 Auswahlkriterien von LODs

Bevor ein Objekt durch eine niedriger aufgelöste Variante ausgetauscht wird, muss entschieden werden, wie wichtig dieses für eine realistische Darstellung des Gesamtbildes ist. Diese Relevanz wird anhand von unterschiedlichen Auswahlkriterien getroffen.

Die distanzbasierte Selektion, also der Abstand zwischen dem Objekt und dem Blickpunkt des Betrachters, ist die mit am häufigsten benutzte Variante für eine LOD-Implementierung. Für jedes einzelne LOD wird ein Distanzbereich ermittelt, in dem es aktiv ist. Somit entsteht eine Datenstruktur mit den verschiedenen LOD Objekten mit den zugewiesenen Distanzen. Die detaillierteste Version hat einen Bereich von null bis zu einem benutzerdefinierten Wert von r1, was bedeutet, dass dieses LOD angezeigt wird, wenn die Distanz von der Kamera bis zum Objekt kleiner als *r1* ist. Dem nächsten LOD wird ein Bereich von *r1* bis *r2* zugewiesen, wobei $r2 > r1$. Ist die Distanz zum Objekt größer oder gleich r1 und kleiner als *r2*, wird dieses LOD genutzt. Dies ist beliebig erweiterbar. **[Lue03]**

Alternativ zur Distanz kann die LOD Stufe anhand der Größe gewählt werden, bzw. der Größe des Abbildes eines Objekts auf dem Bildschirm. Denn je weiter weg sich ein Modell befindet, desto weniger Platz nimmt dessen Projektion im Bild ein. Anstatt den verschiedenen Distanzen bestimmte Werte zuzuweisen, legt man nun die Größe oder die Anzahl an Pixel im Bild für jede LOD Stufe fest. Die größenbasierte Auswahl vermeidet einige Probleme der distanzbasierten Selektion, da die Projektionsfläche unabhängig von der Auflösung des Bildschirms oder den Proportionen eines Modells ist. Mit dieser Methode wird immer das ganze Objekt betrachtet und nicht nur ein beliebiger Punkt in der Geometrie. Dieser Vorteil beansprucht allerdings einen aufwändigeren Prozess, da das Objekt zuerst in Bildschirmkoordinaten umgerechnet werden muss. Hierbei wird nicht das Originalmodell genommen, sondern ein vereinfachtes Bounding Volumen, meist eine Box oder Kugel, um Rechenzeit zu sparen. Bei rotierenden dünnen Objek-

ten, wie ein Messer, kann sich die Größe sehr schnell verändern, was einen unerwünschten und schnellen Austausch mehrerer LOD Stufen zur Folge hat, weshalb gerne eine Bounding Kugel genommen wird, deren Projektionsfläche aus einer Distanz immer gleich aussieht. **[Lue03]**

Manche Objekte sind für das Testen von kamerabasierten FAS besonders entscheidend. Beispielsweise tragen die Fahrzeugmodelle und die Charaktere, wie z. B. Fußgänger, bei der Objekterkennung eine besondere Rolle, hingegen die Häuser oder Vegetationsobjekte eine eher untergeordnete Rolle. Somit stellt die prioritätsbasierte Auswahl eine dritte Möglichkeit dar, LOD Stufen zu bestimmen. Die Prioritätswerte wird vorzeitig vergeben, sodass zur Laufzeit wichtige Elemente gar nicht und nebensächliche je nach Priorität vereinfacht werden. Diese Methode lässt sich auch mit anderen LOD Systemen verbinden.

4.3.2 Diskretes LOD

Das Grundprinzip des diskreten LODs wurde bereits 1976 von James Clark vorgestellt und wird bis heute in den 3D-Echtzeitanwendungen verwendet **[Lue03]**. Bei dieser Technik werden, abhängig von der Entfernung vom Auge des Betrachters, komplexe Objekte durch nacheinander folgend einfachere (in Geometrie und/oder Shader) Repräsentationen ausgetauscht. Hierbei werden die verschiedenen Detailstufen vorberechnet und in einer speziellen Datenstruktur gespeichert. Zur Laufzeit wird dann durch Auswahlkriterien entschieden, welcher Level tatsächlich dargestellt wird. In Abbildung 4.7 ist ein Beispiel für die unterschiedlichen Level dargestellt.

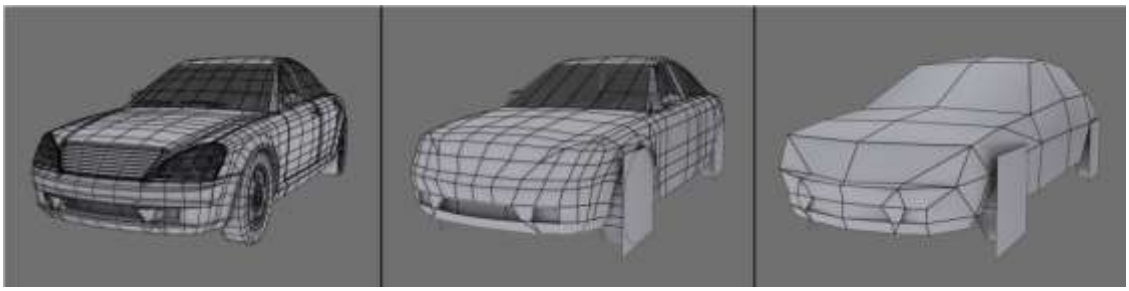


Abbildung 4.8: Diskretes LOD mit drei Detailstufen.

Die Vorteile dieses Systems sind die einfache Erstellung und Implementierung in die Applikation. Jedoch hat dieses System auch Nachteile. Zum Einen kostet es einen nicht unerheblichen Aufwand von ein und demselben Objekt mehrere unterschiedlich aufgelöste Modelle zu erstellen. Zum Anderen kann der Übergang zwischen zwei Levels bei einer zu großen Unterscheidung der Geometrie zu offensichtlich geschehen **[BeBr06]**. Der Effekt, der dabei auftritt, wird als

"popping" bezeichnet. Es gibt unterschiedliche Methoden um das "popping" zu verhindern oder zu reduzieren. Durch die Verwendung von mehreren Levels lässt sich die Veränderung der Modelle möglichst gering halten. Somit gibt es keinen harten Übergang zwischen den Level. Eine andere Methode ist das Überblenden von zwei Levels. Dabei wird die eine Geometrie langsam ausgeblendet, wobei die andere langsam eingeblendet wird. Dies gewährleistet einen weichen Übergang.

4.3.3 Kontinuierliches LOD

Das Kontinuierliche LOD unterscheidet sich von dem diskreten Ansatz durch einen Simplifikationsalgorithmus der das 3D-Objekt kontinuierlich im Detailreichtum verringert. Anders als bei dem diskreten Ansatz müssen keine individuellen LODs während der Entwicklung erstellt werden.

Der Hauptvorteil des kontinuierlichen LODs ist die bessere Granularität, also der Verlauf an Übergängen zwischen den Detailstufen gegenüber dem diskreten LOD. Die Detailstufen müssen nicht im Voraus festgelegt und erstellt werden und sind deswegen auch nicht auf diese beschränkt. Anstatt drei festgelegten Detailstufen wird eine Vielzahl an Schritten, die das 3D-Objekt kontinuierlich in Details verringert oder erhöht, angelegt. Durch die Reduzierung der Polygone können andere Objekte detaillierter gestaltet werden.

Doch das kontinuierliche LOD zieht auch diverse Nachteile mit sich. Es besteht nicht die Möglichkeit die Bereiche zu bestimmen, an denen Details verringert oder erhöht werden, da jedes Objekt anders gestaltet ist. Durch die unterschiedlichen Formen, aus denen Fahrzeugen bestehen können, ist die globale Verwendung eines kontinuierlichen LODs für die Implementierung nicht geeignet. Durch die automatische Simplifikation können Fehler in der Geometrie entstehen, die von Fahrzeug zu Fahrzeug nicht abschätzbar sind. Gerade bei Fahrzeugen, die einer quadratischen Form gleichen, wie bspw. Stadtbusse oder Lastkraftwagen, verursacht die Simplifikation irreparable Fehler in der Geometrie. (vgl. Abbildung 4.7)

Weitere Literatur zu kontinuerlichem LOD findet sich in **[Cho97]**, **[Lue03] [EIS00]**.

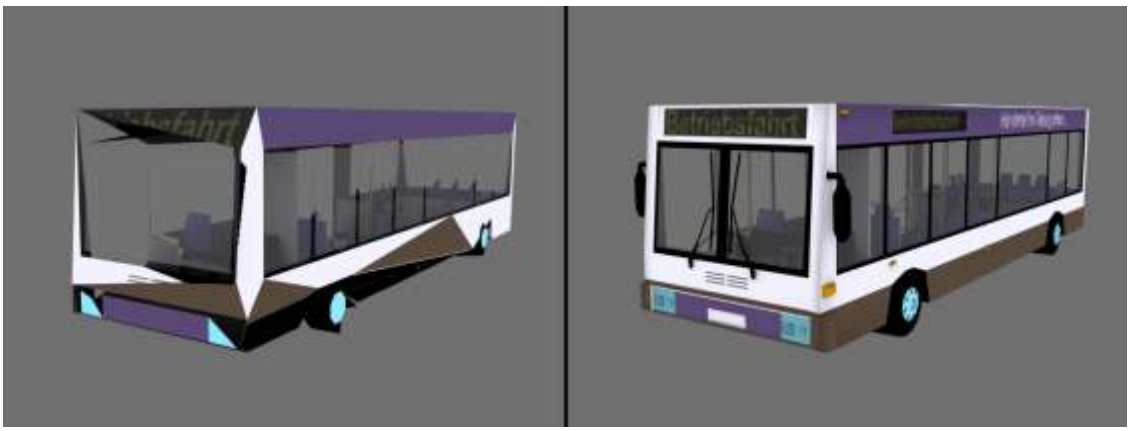


Abbildung 4.9: Fehler durch die Simplifikation der Fahrzeuge.

4.3.4 Sichtabhängiges LOD

Das sichtabhängige LOD kann auch als eine Erweiterung des kontinuierlichen LOD verstanden werden. Ähnlich wie bei diesem LOD wird auch beim sichtabhängigen die Detailstufe des 3D-Objektes zur Laufzeit berechnet. Ausschlaggebend für die lokale Detailstufe ist jedoch der Blickwinkel der virtuellen Kamera. Elemente, die sich in der Nähe des Betrachters und in dessen Sichtbereich befinden werden, sowie die Umrisse des Objektes werden detaillierter dargestellt. Ein typisches Beispiel für ein sichtabhängiges LOD ist die auch in Visual-Loop-Komponenten vorkommende Terrainlogik. Da sich ein Teil des Terrains immer in unmittelbarer Nähe des Betrachters befindet, werden auch die Bereiche die sich nicht im Blickwinkel des Betrachters befinden hoch aufgelöst. Es ist wichtig darauf zu achten, dass die Details für den Betrachter nicht verloren gehen. **[Xia96]**

4.3.5 Reflektion LOD

Die Reflektionen, die auf der Oberfläche des Fahrzeuges sichtbar sind, werden in jedem Frame neu berechnet. Für die Darstellung wird eine Cubemap verwendet, die sich aus sechs Bildern zusammensetzt. Im Fahrzeug befindet sich eine Virtuelle Kamera, die pro Frame die Szene in diese sechs Bilder rechnet und sie in die Cubemap schreibt. Dieser Vorgang ist sehr aufwendig, da die Umgebung des Fahrzeugs sechs Mal gerendert wird. Wenn nun mehrere Fahrzeuge nebeneinander stehen und sich in der Oberfläche spiegeln wird der Rechenaufwand potenziert.

Mit Hilfe eines Reflektions-LOD (RLOD) kann die Berechnung der Cubemap beschleunigt werden. Jedes 3D-Objekt in der Szene erhält ein RLOD-Objekt. Dieses wird in der Reflektionsberechnung anstatt des normalen Modells verwendet. RLOD-Objekte bestehen lediglich aus wenigen Polygonen und besitzen eine sehr geringe Texturauflösung, wodurch sie nicht rechenintensiv sind. Außerdem lässt sich durch das Zusammenfügen aller RLOD-Objekte zu einem Batch und einer Textur die Berechung auf ein Minimum zurücksetzen. Gerade für Testszenarien mit großen Städten lässt sich ein solches RLOD-System anwenden, da die Häuserfassade durch ein Polygon mit der Texturprojektion des Hauses ersetzt werden kann.

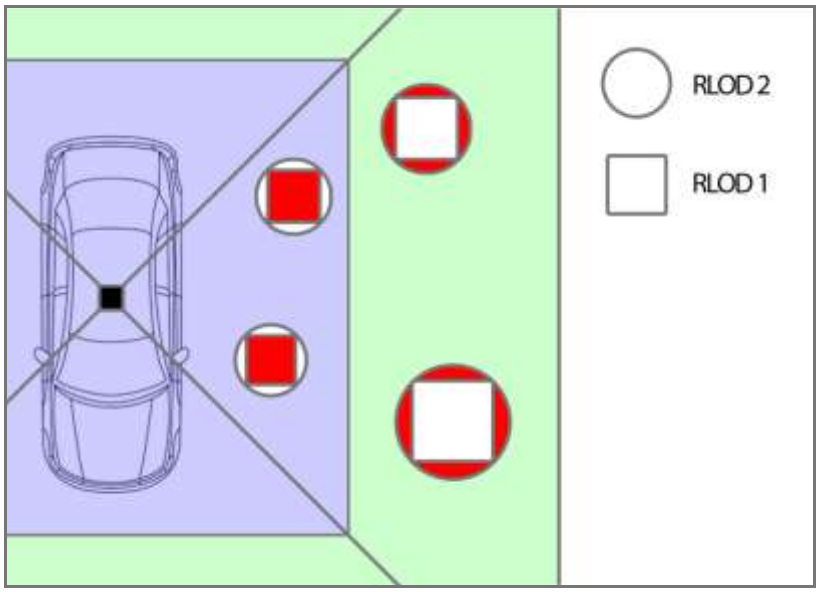


Abbildung 4.10: RLOD: die rot markierten Objekte werden zur Laufzeit von einer Reflektionskamera in die Cubemap gerendert.

In Abbildung 4.10 ist ein solches RLOD-System schematisch dargestellt. Die rot markierten Objekte werden zur Laufzeit von einer Reflektionskamera in die Cubemap gerendert. Dabei werden, je nachdem in welchem Distanzbereich sich die Objekte befinden, die dementsprechende RLOD angezeigt. Das "RLOD1"-Objekt wird nur im blauen, das "RLOD2"-Objekt nur im grünen Distanzbereich angezeigt. Wenn sich das Objekt in keinem der beiden Distanzbereiche befindet, wird es nicht in der Reflektion angezeigt. Die RLOD-Objekte werden nur von der Reflektionskamera gerendert und sind für die Hauptkameras nicht sichtbar.

Die Implementierung eines solchen Systems bedarf einer frühzeitigen Planung und einem großen Aufwand in der Entwicklung. Die RLOD-Objekte müssen bei der Erstellung der Objekte zusätzlich modelliert werden.

4.3.6 Shader LOD

Durch die Verwendung eines Shader LOD (SLOD) können rechenintensive Eigenschaften unter bestimmten Auswahlkriterien ausgeschaltet oder verändert werden. Beispielsweise werden beim Shader der Fahrzeuge die Reflektionen ab einem bestimmten Abstand zur Kamera weggelassen, um somit die Erstellung der Cubemap und den damit verbundenen Renderaufwand nicht berechnen zu müssen.

Weiterführende Erläuterungen zu Shader LODs können in **[Pel05]** nachgelesen werden.

4.4 Bonematrix-LOD-System

Durch das Zusammenfügen der 3D-Objekte innerhalb der Fahrzeuge lässt sich bereits eine hohe Anzahl an Batches einsparen. Zusätzlich kann mit Hilfe eines LOD-Systems die Batchanzahl in der Entfernung verringert werden. Wenn jedes Fahrzeug mit einem Batch gerendert wird, wird dennoch für die Darstellung von beispielsweise eintausend Fahrzeugen die gleiche Anzahl an Batches benötigt.

Die Idee ist, alle Fahrzeuge ab einer bestimmten Entfernung zu einem Batch, d. h. zu einem Geometrieobjekt, zusammenzufassen. Der Performancegewinn der dadurch erzielt ist enorm, ist jedoch mit einigen Problemen verbunden. Beispielsweise lassen sich die einzelnen zusammengefügten Fahrzeuge nicht mehr separat transformieren. Jedoch können mit Hilfe des SkinnedMeshRenderers von Unity3D die Vertices über Bones transformiert werden. Ein Bone bezeichnet eine Transformationsmatrix, die die Position der Vertices beeinflussen kann. Die Stärke mit der die Vertices durch die Bones verschoben werden, lässt sich in den Vertex-Eigenschaften speichern. In diesem Anwendungsfall sollten sie die Vertices zu 100% verschieben. So kann durch das Hinzufügen eines Bones und der entsprechenden Informationen in den Vertices das Fahrzeug wieder transformiert werden.

Sobald ein Fahrzeug der Szene hinzugefügt wird, wird auch dem globalen Fahrzeugobjekt des LOD-Systems eine niedrig aufgelöste Version des Fahrzeugs hinzugefügt. Je nachdem, welchen Abstand die Kamera zu dem Fahrzeug besitzt, wird die "Bone-Version" des Fahrzeugs sichtbar und das derzeitige LOD-Objekt ausgeblendet. Die Position an der ein LOD in die "Bone-Version" übergeht, soll als Schwellenposition bezeichnet werden.

Da die "Bone-Versionen" der Fahrzeuge nur aus einem Geometrieobjekt bestehen, muss für die Implementierung ein globales Material benutzt werden, das diesem bei der Erstellung neu zugewiesen wird. Da die unterschiedlichen Oberflächen des Fahrzeugs verschiedene Eigenschaften besitzen, wurde durch das Zusammenfügen der Fahrzeugobjekte die Verwendung von Texturen notwendig. Da die Fahrzeuge sich untereinander in der Form und Ausstattung unterscheiden, muss eine Lösung gefunden werden, alle Fahrzeugtexturen in einer einzigen Textur zu vereinen. Die Verwendung eines Textureatlas ermöglicht genau dies. Dabei bekommt jedes Fahrzeug einen lokalen Abschnitt der globalen Textur, in der die Textur des Fahrzeuges abgelegt wird, eindeutig zugewiesen. So ist gewährleistet, dass jedes Fahrzeug die ursprüngliche Textur behält, aber dennoch ein Material mit einer Textur verwendet wird. In **[NVI04]** ist das Konzept eines solchen Textureatlases näher erläutert.

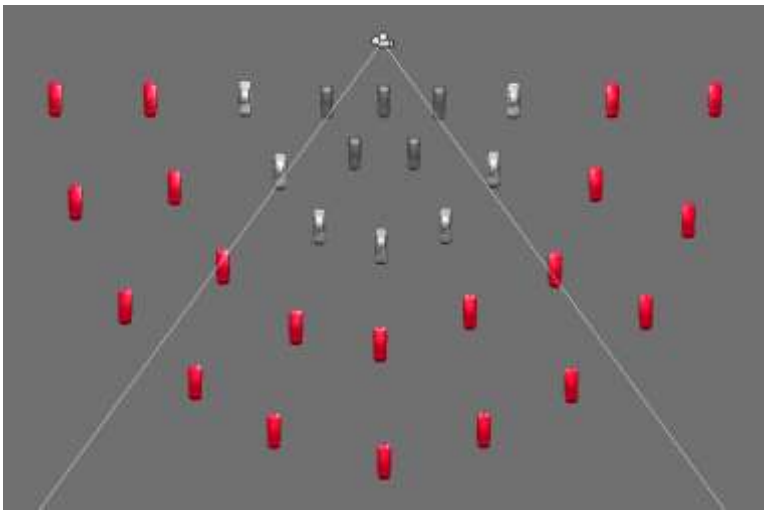


Abbildung 4.11: Bone-Array-System: die rot markierten Fahrzeuge werden zu einem globalen Objekt zusammengefasst.

Ein weiteres Problem ist das Ein- und Ausblendung einzelner Fahrzeuge, die sich in der Bone-LOD-Stufe befinden. Da die Objekte nicht mehr als separate Objekte existieren, können sie auch nicht mehr einzeln deaktiviert werden, wie es bei einem normalen diskreten LOD-System der Fall ist. Durch das View-Frustum-Culling werden Objekte, die sich nicht im Sichtfeld der Kamera befinden, nicht angezeigt. Diese Funktion kann als Lösung für das Problem dienen. Es wird eine Position definiert, die nie in das Sichtfeld der Kamera gelangt. Wenn die Kamera rotiert wird, muss sich somit auch diese Position abhängig von dem Kamerablickwinkel mit rotieren. Hier werden die Fahrzeuge über die Bones abgelegt und bei Bedarf von dort an den Schwellenpunkt platziert. Jedem Bone innerhalb des SkinnedMeshRenderers wird ebenfalls die Objekt-ID des

Fahrzeugs zugewiesen, um die Objektzugehörigkeit zu gewährleisten. Durch die Verwendung dieses Systems wird die Batch-Anzahl der Fahrzeuge der Bonematrix-LOD-Stufe auf genau einen Batch reduziert.

4.5 Auswertung und Zusammenfassung der Methoden

In Kapitel 3.2.3 wurde erläutert, warum es nötig ist, die Anzahl an Batches pro Fahrzeugobjekt zu reduzieren. Hierzu wurden bereits einige Methoden vorgestellt und diskutiert. Das Zusammenfügen der Fahrzeugobjekte bietet eine erste Basis bei der Reduzierung der Batches. Hinzu kommt die Verwendung eines diskreten LOD-Systems, das sich durch eine einfache Implementierung und hohe Effizienz auszahlt. Ab einer bestimmten Entfernung wird die Reifendrehung nicht mehr als solche erkannt, wodurch das Fahrzeug in der Ferne zu einem Batch zusammengefasst werden kann. Dies spart bereits 4 Batches für ein Fahrzeug, das sich in dem letzten LOD befindet. Das Zusammenfügen beider Techniken zu einem Konzept kann die Performance weiterhin steigern. Durch das Rendern aller Fahrzeuge der Bone-LOD-Stufe in einem Batch erhöht sich die Möglichkeit der zu platzierenden Fahrzeuge um das Hundertfache.

Die in diesem Kapitel vorgestellten Methoden werden nachfolgend nach ihrem Performancegewinn sortiert und bewertet. Anschließend wird abgewogen, welche für eine Implementierung im Rahmen der Arbeit in Frage kommen. In Tabelle 5.1 sind die Methoden nach den Kriterien "Performancegewinn", "Implementierungsaufwand" und "Eingriff in das Programm" bewertet.

Tabelle 4.1 Bewertung der Methoden für die Implementierung.

Methoden	Perfromancegewinn	Implementierungsaufwand	Eingriff in das Programm
Zusammenfassen der Fahrzeugobjekte	+	o	+
diskretes LOD-System	+	+	o
kontinuirliches LOD-System	-	o	o
RLOD-System	o	-	-
Shader-LOD	-	+	+
Post-Processing-Effekt	o	-	-
Bonematrix-LOD-System	+	o	o

Das Zusammenfügen der Fahrzeugobjekte, das diskrete LOD-System und das Shader-LOD erreichen im Vergleich zu den anderen Methoden den größten Nutzen für die Performancesteigerung. Auch das Bonematrix-LOD-System sticht mit einem hohen Performancegewinn heraus. Alle anderen Methoden bringen zwar einen Performancegewinn, sind aber vom Verhältnis Implementierungsaufwand zu Performancegewinn für eine Implementierung im Rahmen dieser Arbeit nicht geeignet. Für den Post-Processing-Effekt und das RLOD-System müsste beispielsweise jedes vorhandene Objekt neu angepasst und mit den veränderten Werten exportiert werden. Das erfordert einen enormen zeitlichen Aufwand, der nicht im Verhältnis zum Performancegewinn steht.

5 Implementierung

Die aus dem Konzept hervorgegangenen Methoden werden in der Implementierung prototypisch in einer Visual-Loop-Komponente umgesetzt. Das Vorgehen und die damit verbundenen Problematiken werden in diesem Kapitel diskutiert. Zunächst wird ein Fahrzeug aufgrund der im Konzept vorgestellten Methode erstellt und angepasst. Danach wird ein diskretes LOD-System in das vorhandene System integriert. Schlussendlich wird das in Kapitel 4.4 vorgestellten Konzepts des Bonematrix-System in einer Testumgebung prototypisch umgesetzt.

5.1 Optimierung der Fahrzeuge

Wie in Kapitel 3.2.3 beschrieben, müssen die Fahrzeuge in der Objekt-Anzahl reduziert werden. Für die Implementierung wird deshalb ein Fahrzeug unter Berücksichtigung der Performanceanalyse erstellt. Dabei werden auch die Modellierungsmethoden aus **[And11]** verwendet. Zunächst werden mit einem Linienwerkzeug die Konturen des Fahrzeugs bestimmt. Für die Modellierung wurde das Boxmodelling-Verfahren angewandt, bei dem ein Quader durch Unterteilung der Polygone im Detailreichtum erhöht wird. Aus Performancegründen wurde bisher auf die Modellierung des Fahrzeuginnenraums verzichtet. Durch das Zusammenfügen der Fahrzeugobjekte wird die Verwendung von Texturen nötig. Über diese kann dem Innenraum eine andere Farbe als der Karosserie zugewiesen werden. Das Zusammenfügen der Fahrzeugobjekte erlaubt auch eine erhöhte Vertex-Anzahl pro Fahrzeug. Somit kann nicht nur der Innenraum komplett ausmodelliert, sondern auch der Karosserie ein größerer Detailreichtum hinzugefügt werden. Außerdem können Details, wie beispielsweise Scheibenwischer, ausmodelliert werden. Insgesamt besteht das neumodellierte Fahrzeug aus 14445 Triangles, was im Vergleich zu den alten Fahrzeugen einen Zuwachs von ca. 2000 Triangle bedeutet.

Das Zusammenfügen der Fahrzeuge beschränkt, wie in Kapitel 4.1.1 beschrieben, die Verwendung auf ein Material pro Fahrzeug. Deshalb werden die Farbwerte über eine Textur hinzugefügt. Jedoch wird durch die Verwendung einer Textur nicht nur die Möglichkeit gegeben, die ursprünglichen Farbwerte wieder herzustellen, sondern auch zusätzliche Details hinzuzufügen. Dabei lässt sich beispielsweise das sogenannte Lightbaking verwenden, bei dem Licht- und Schatteneffekte in die Textur gerechnet werden. Damit das neue Fahrzeugmaterial sämtliche

Oberflächenbeschreibungen aller vorherigen Materialien beinhaltet, wird ein genau für diesen Anwendungsfall angepasster Shader geschrieben werden.

Abbildung 5.1: Fahrzeug: Besteht insgesamt aus 14.443 Triangles und 6 Batches.

Durch Zusammenfügen der Objekte muss auch das Fahrzeugskript angepasst werden. Dies ist zwingend erforderlich, da bei den alten Fahrzeugen beispielsweise die Farbe der Halos aus den Scheinwerferobjekten ausgelesen wurde. Nun kann die Farbe direkt in den Halo-Objekten gesetzt und über das Skript angepasst werden.

5.2 Shader

In Kapitel 4.1.1 wurde beschrieben, welche Eigenschaften die Materialien der unterschiedlichen Fahrzeugobjekte besitzen. Die Karosserie muss beispielsweise die Umgebung reflektieren und Glanzeffekte auf der Oberfläche gewährleisten. Weiterhin gibt es Plastikmaterialien, die weder reflektieren noch glänzen sollen. Jedes dieser Materialien soll nun durch die Verwendung eines

Shader ermöglicht werden. Es besteht die Option, die Materialinformationen über Texturen festzulegen. Da jedoch die Verwendung von Texturen Speicherplatz in der Grafikkarte benötigt können nicht beliebige viele Texturen pro Objekt genutzt werden. Da für die Beschreibung der Stellen an denen das Fahrzeug reflektieren oder glänzen soll jeweils lediglich ein Graustufenbild erforderlich ist, können diese in den jeweiligen Farbkanälen einer Textur gespeichert werden. Das heißt, dass beispielsweise die Stellen für die Reflektion in den Rot-Kanal und die Glanzeffekte in den Grün-Kanal gespeichert und anschließend vom Shader ausgelesen werden.

Um sich auch diese Textur sparen zu können, werden diese Graustufeninformationen direkt in die Vertices geschrieben. Mit Hilfe des Shaders kann auf diese Informationen zugegriffen werden. Allerdings wird durch die Speicherung der Farbinformationen in den Vertices eine detailierte Oberflächenbeschreibung nicht mehr möglich, da diese nur an den Vertexpositionen definiert und anschließend über jedes Triangle interpoliert werden.

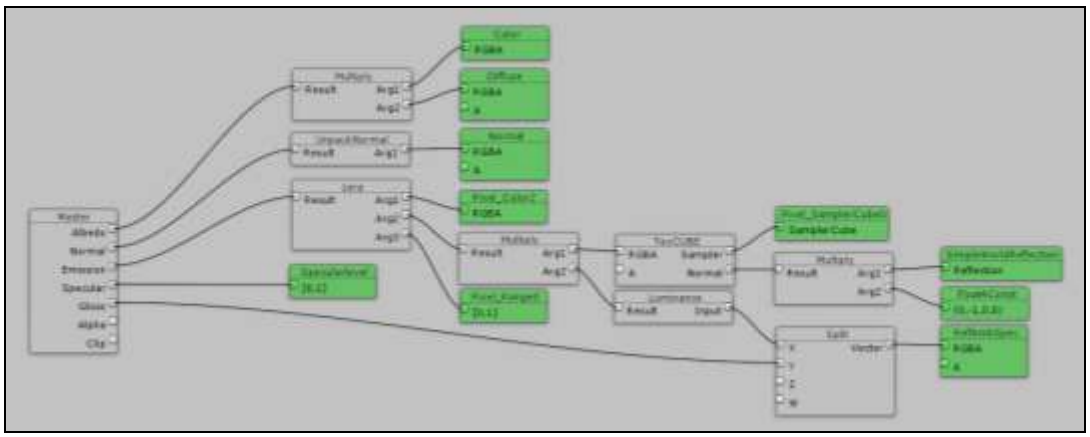


Abbildung 5.2: Shader Knotendiagramm: die grünen Quader markieren die Eingabewerte.

Unity3D bietet für die Programmierung von Shadern eine eigene Programmiersprache, die sich Shaderlab nennt. Die Sprache ist einfach zu verstehen und bietet viele Möglichkeiten, die aber nicht an die Möglichkeiten von den Grundsprachen wie HSL oder CG reichen. Jedoch kann über die Einbindung von CG-Programmen der Funktionsumfang erweitert werden. Für die Implementierung wurde ein Fahrzeugshader geschrieben, der eine Farbtextur und NormalMap unterstützt. Außerdem lassen sich Reflektion und Glanzeffekte aus den Farbinformationen der Vertices auslesen. **[Fer03]** bietet für die Programmierung eines solchen Shaders eine gute Basis. Der Shader bietet somit alle Funktionalitäten der vorherigen Materialien und erweitert diese noch um die Möglichkeit Normalmaps darstellen zu können. In Abbildung 5.2 ist der geschriebene

Shader in einem Knotendiagramm dargestellt. Dabei kennzeichnen die grün markierten Quader die Eingabewerte des Shaders. Dies können beispielsweise Zahlen oder Texturen sein. Die Eingabewerte können dann über mathematische Operationen, wie zum Beispiel Multiplizieren, Addieren oder Subtrahieren, miteinander verrechnet werden. Schlussendlich wird der Shader kompiliert und einem Material zugewiesen.

5.3 LOD-System

Um nicht nur die Batch-Anzahl zu senken, sondern auch die Vertex-Anzahl zu reduzieren, wird im Rahmen der Implementierung ein LOD-System realisiert. Das LOD-System wird als separates Skript gehalten, das jedem 3D-Objekt zugewiesen werden kann, damit so wenig wie möglich in der bestehenden Programmstruktur geändert werden muss. Für die Implementierung bietet sich somit ein diskretes LOD-System an. Das Skript wird dabei so variabel wie möglich gehalten, so dass nicht nur die Fahrzeuge mit diesem ausgestattet werden können, sondern dieses auch allen anderen Objekten hinzugefügt werden kann. In dem Skript lassen sich beliebig viele LOD-Stufen anlegen. Diese werden dann, je nachdem wie weit das Objekt von der Kamera entfernt ist, ausgetauscht.

Neben dem LOD-System muss auch das Fahrzeugskript auf die neuen Veränderungen angepasst werden. So muss beispielsweise beachtet werden, dass sich die Reifen der niedrig aufgelösten 3D-Objekten nicht mehr drehen können. Die einzelnen LODs des Fahrzeugs werden dem Skript hinzugefügt und jedem ein Distanzbereich zugewiesen, in dem es angezeigt werden soll. Dann wird zur Laufzeit die Distanz von Fahrzeug zur Kamera geprüft und anhand dessen das jeweilige LOD eingeblendet.

5.4 Bonematrix-LOD-System

Zusätzlich zu dem diskreten LOD-System wird das in Kapitel 4.4 vorgestellte Konzept eines Bonematrix-LOD-Systems in einer Testumgebung prototypisch umgesetzt. Hierfür wird neben dem LOD-Skript ein weiteres Skript benötigt, das die LOD der letzten Stufe zusammenfügt. Das sogenannte MeshMerger-Skript fasst beim Start der Visual-Loop-Komponente alle Objekte der letzten LOD-Stufe zu einem Geometrieobjekt zusammen. Dabei wird jedem Objekt ein Bone zugewiesen, damit dieses weiterhin transformierbar bleibt. Jedem Bone wird die Nummer des Objekts eindeutig zugewiesen, damit das eindeutige Aufrufen der Bones gewährleistet bleibt.

Die Position an der sich Bone-Versionen platziert werden, wenn sie durch das LOD-System nicht verlangt werden, wird im Prototypen zunächst direkt hinter die Kamera platziert. Wenn das System in die Visual-Loop-Komponente integriert wird, kann diese Position direkt unterhalb des Terrains gewählt werden, da diese Position niemals in den Blick der Simulationskamera kommt. Wenn nun eine Bone-Version durch das LOD-Skript verlangt wird, ruft dieses die Methode setBone() des MeshMerger Skripts auf und übergibt die Schwellenposition und die Nummer des Objekts. Die Methode sucht dann den Bone mit der angegeben Nummer und setzt diesen an die Schwellenposition. So ist das Ein- und Ausschalten der Bone-Versionen gewährleistet.

5.5 Zusammenfassung

Für die Implementierung werden, die in Kapitel 4.5 ausgewerteten Methoden, verwendet.

Zunächst werden die Fahrzeuge in der Objektanzahl reduziert, wodurch sich weitere Probleme ergeben, wie die Tatsache, dass nur noch ein Material pro Fahrzeug verwendet werden kann. Um nun alle Funktionalitäten der vorherigen Materialien zu gewährleisten muss ein neuer Shader entwickelt werden, der diese in sich vereint. Die unterschiedlichen Oberflächenbeschreibungen des Fahrzeugs werden über die Farbwerte in den Vertices definiert. Beispielsweise gibt der Rot-Kanal, dieses Farbwertes in den Vertices die Stärke der Reflektion, die später zu dem Farbwert des gesamten Fahrzeuges addiert wird, an.

Zusätzlich zu dem Zusammenfügen der Fahrzeugobjekte wird ein diskretes LOD-System implementiert. Dafür werden bei der Erstellung des Fahrzeug insgesamt drei Detailstufen erstellt. Je nachdem wie weit sich die Kamera von dem Fahrzeug entfernt befindet wird die dementsprechende LOD-Stufe angezeigt.

Diese beiden Methoden lassen sich nun in einem Bonematrix-LOD-System zu einem Objekt zusammenfassen. Dabei wird jedem Fahrzeug ein Bone zugewiesen, damit die Fahrzeuge nach dem Zusammenfügen zu einem Geometrieobjekt noch transformierbar bleiben.

Durch die Implementierung ergibt sich kein Funktionsverlust (A1.2). Die Qualität der Fahrzeuge ist durch die Bewertung von Fachkräften deutlich verbesser worden (A.2.2) und eine damit verbundene potentielle Verursachung von Performanceengpässe durch die Fahrzeuge (A2.1) bleibt aus.

6 Evaluation und Ergebnisse

Für die Evaluation der Fahrzeuge wird ein leeres Testszenario erstellt. Das Terrain wird dabei wieder auf die ursprüngliche Höhe gesetzt, damit das Testergebnis durch den Terrainfehler nicht verfälscht wird. Es werden insgesamt drei Tests durchgeführt, die dann miteinander verglichen und ausgewertet werden. Zunächst werden auf den ersten 100 m² des Terrains insgesamt zweihundert Fahrzeuge, die derzeit in PROVEtech:VL enthalten sind, platziert. Außerdem wird eine neue Kamera erstellt. Diese wird an eine feste Position platziert, so dass alle Fahrzeuge in Sichtweite sind, aber die Kamera sich dennoch wenige Meter von den ersten Fahrzeugen entfernt befindet. Anschließend wird mit Hilfe des Frame Profilers von PerfHUD die Testszene analysiert. Dieser Vorgang wird ebenfalls für die neuerstellten Fahrzeuge, die durch die alten Modelle ausgetauscht werde, durchgeführt. Anschließend werden diese Fahrzeuge mit dem angehängten LOD-Skript analysiert. In Abbildung 6.1 sind die drei Testergebnisse nach der DC-Anzahl dargestellt.

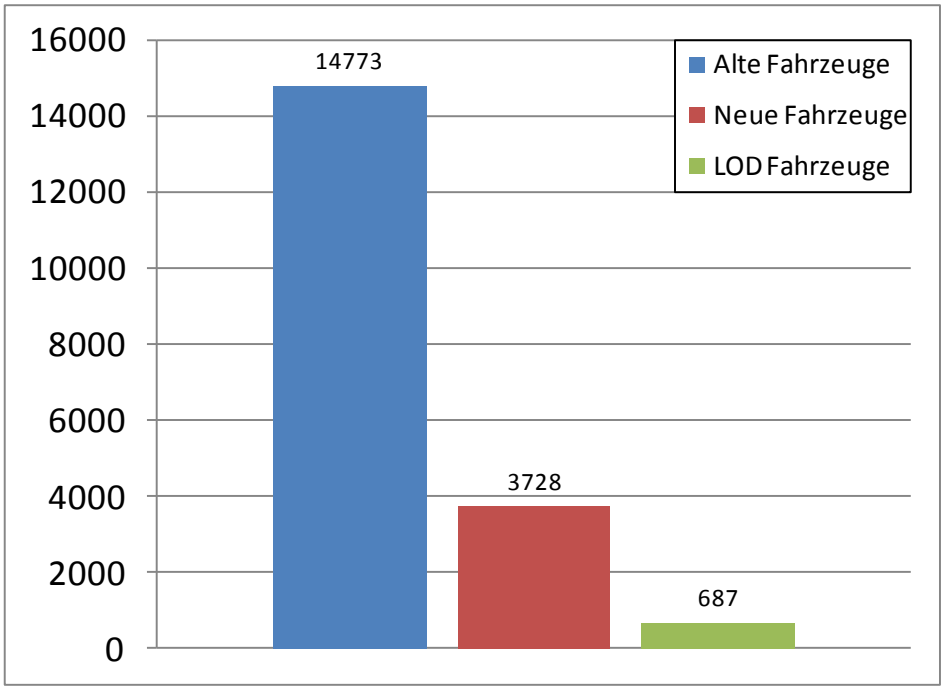


Abbildung 6.1: Versuchsergebnisse: 1. alte Fahrzeuge 2. neue Fahrzeuge 3. LOD Fahrzeuge.

Es wird deutlich, dass die DC-Anzahl erheblich verringert wurde. Durch das LOD-Skript existiert nur noch 1/20 der vorherigen DC-Anzahl. Dies wirkt sich stark auf die Framerate aus. Wurden bisher bei den alten Fahrzeugobjekten insgesamt 7,6 fps erreicht, so kann durch die neuen Fahrzeuge mit zugewiesenem LOD-Skript eine Framerate von 113,4 fps erzielt werden. Durch diese Performancesteigerung ist eine Anzahl von mehreren hundert Fahrzeugen pro Testszenario möglich. Somit ist die Anforderung A1.3 einer deutliche messbare Steigerung der Performance erfüllt.

Im Anhang befinden sich die Ergebnisse der drei Testdurchläufe in Diagrammen formatiert. Aus diesen Diagrammen geht ebenso hervor, dass die Berechnung auf der CPU auch Auswirkungen auf die Framerate haben kann. Die rote Kurve stellt dabei die Berechnungszeit der CPU dar.

Auch wird deutlich, dass die Berechnungsdauer der Grafikkarte bei den neuen Fahrzeugmodellen länger ausfällt. Dies wird durch eine höhere Anzahl an Vertices pro Fahrzeug, da eine größere Menge an Daten verarbeitet werden muss. Die Ergebnisse dieser Analyse bestätigen die Notwendigkeit der Batch-Reduzierung, wie sie in Kapitel 4.1 beschrieben wurde. Durch das Zusammenfügen der Fahrzeuge wird die Performance um das Doppelte verbessert. Die Verwendung eines LOD-Skripts erhöht die Performance insgesamt um das fünfzehnfache.

Für die Evaluierung wird eine Testszene, in der das Bonematrix-LOD-System prototypisch realisiert ist, als eigenständiges Programm erstellt. Diese wird anschließend mit PerfHUD analysiert, wobei lediglich die letzte LOD-Stufe untersucht wird. In der nachfolgenden Tabelle sind die Ergebnisse diese Analyse aufgeführt.

Tabelle 6.1 Performanceanalyse des Bonematrix-LOD-Systems

Art des Tests	GPU-Berechung	Draw Calls
diskretes LOD-System	0,3 ms	30
Bonematrix-LOD-System	0,17 ms	2

Die Berechnungszeit der Grafikkarte beträgt bei dem Bonematrix-LOD-System fast die Hälfte im Vergleich zu dem diskreten-LOD-System, auf der CPU ist Berechnungsdauer fast identisch. Die DC-Anzahl verringert sich bei dem Bonematrix-LOD-System bis auf zwei DCs. Durch die Performanceanalyse wird somit nachgewiesen, dass die vollständige Implementierung eines Bonematrix-LOD-Systems für die Performance des Systems von großem Vorteil ist.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wird die Performance einer Visual-Loop-Komponente untersucht und geeignete Methoden zur Verbesserung dieser recherchiert und entwickelt. Um beim Testen von kamerabasierten FAS durch eine Visual-Loop-Komponente Ergebnisse zu erzielen, die denen herkömmlicher Testfahrten so Nah wie möglich sind, muss die Darstellung so realitätsnah wie möglich sein. Dabei gilt, dass die Bilder, die durch die Visual-Loop-Komponente erzeugt werden, mit mindestens 30 Frames pro Sekunde angezeigt werden, da die Kamerasysteme der FAS die Umgebung mit dieser Framerate aufnehmen. Die Framerate dient für die nachfolgende Performanceanalyse als Messwert.

PROVEtech:VL, die Visual-Loop-Komponente der PROVEtech Suite von der MBtech-Group, dient als Testsystem für die Performanceanalyse. Zur Analyse wird NVIDIA's PerfHUD verwendet, das sich durch eine Vielzahl an Analysemöglichkeiten auszeichnet. Durch die zusätzliche Möglichkeit der automatischen Performanceanalyse von PerfHUD kann genau bestimmt werden, welche Objekte einen Engpass verursachen.

Da die Abschnitte der Grafikpipeline voneinander abhängig sind, lässt sich das Konzept der Bottleneck-Analyse für die Performanceanalyse anwenden. Das Ergebnis der Performanceanalyse ergibt, dass die Terrainlogik und die Fahrzeuge die Hauptursache für die massiven Performanceeinbrüche sind. Mit ca. 3250 DCs ist das Terrain das Objekt, das die meisten DCs verursacht. Die Fahrzeuge liegen mit 1652 DCs an zweiter Stelle. Die restlichen Objekte besitzen insgesamt 200 DCs und fallen somit im Vergleich zur Terrainlogik und den Fahrzeugen kaum ins Gewicht.

Da der Performanceengpass, der durch die Terrainlogik verursacht wird auf einem schnell behebbaren Fehler im Skript basiert, wird das Hauptaugenmerk auf die Optimierung der Fahrzeuge gelegt. Zunächst wird eine Recherche über bestehende Methoden durchgeführt. Die Bewertung der Methoden wird nach den Kriterien Implementierungsaufwand, Performancegewinn und Eingriff in das bestehende Programm vorgenommen.

Die Methode, die für die Performanceoptimierung den meisten Nutzen erbringt, ist das Zusammenfügen der einzelnen Objekte, wie z. B. Lichter, Karosserie, Plastikverkleidung, etc., die in einem Fahrzeug enthalten sind. Dabei entstehen jedoch neue Einschränkungen, wie beispiels-

weise die Tatsache, dass nur noch ein Material pro Fahrzeug verwendet werden kann. Diese Problematik kann allerdings mit Hilfe einer Textur ebenfalls behoben werden. Die zweite Methode, die zu einem großen Performancegewinn führt und dabei keinen Eingriff in die bestehenden Strukturen benötigt, ist die Verwendung eines diskreten LOD-System.

Um die Performance noch weiter zu steigern, kann das Konzept dieser Methoden zu einer neuen Idee zusammengefügt werden. Das daraus entstehende Bonematrix-LOD-System basiert zunächst auf einem diskreten LOD-System, es werden jedoch zusätzlich alle Objekte der letzten LOD-Stufe zu einem Geometrieobjekt zusammengefügt. Durch die Zuweisung von Bones wird die Transformation der einzelnen Fahrzeuge gewährleistet. Somit können alle Objekte der letzten LOD-Stufe als ein Batch an die Grafikkarte geschickt werden. In der Evaluierung der prototypischen Implementierung wird deutlich, dass die Verwendung eines solchen Systems gegenüber des diskreten LOD-Systems in der letzten Stufe die Performance um fast das doppelte erhöht.

Diese Arbeit bietet eine Grundlage Performanceengpässe zu kategorisieren. So kann das Konzept des Bonematrix-LOD-System in Zukunft für eine vollständige Implementierung in eine Visual-Loop-Komponente dienen. Um die Performance von PROVEtech:VL weiter steigern zu können, sollte eine weitere Performanceanalyse durchgeführt werden, da sich der Performanceengpass nun an einer anderen Stelle befinden wird. Anschließend können die restlichen im Konzept vorgestellten Methoden implementiert werden.

8 Literaturverzeichnis

[AM99] Ulf Assarsson, Thomas Möller: Optimized View Frustum Culling Algorithms (Chalmers University of Technology, 1999).

[And11] Andrew Grahan: 3D Automotive Modeling : An Insider's Guide to 3D Car Modeling and Design for Games and Film (Burlington, 2011).

[ATH08] Tomas Akenine-Möller, Eric Haines, Naty Hoffman: Real-Time Rendering (Wellesley, Massachusetts, 2008).

[BeBr06] Michael Bender, Manfred Brill: Computergrafik - Ein anwendungsorientiertes Lehrbuch (Zweibrücken, 2006).

[Bie08] Tim Biedert: Die Rendering-Pipeline (Kaiserslautern, 2008).

[Bis05] Richard Bishop: Intelligent Vehicle Technology and Trends (London, 2005).

[Bli96] Jim Blinn: Jim Blinn's Corner: A Trip Down the Graphics Pipeline (San Francisco, 1996).

[Bre04] Karsten Breuer: Verkehrsflusssimulation zur Entwicklung von Fahrerassistenzsystemen (Aachen, 2004).

[Cho97] Mike Chow: Optimized Geometry Compression for Real-time Rendering (Massachusett, 1997).

[Dis03] Hartwig Distler: Wahrnehmung in Virtuellen Welten (Berlin, 2003).

[ElS00] Jihad El-Sana, Yi-Jen Chiang: External Memory View-Dependent Simplification (Beer-Sheva, Brooklyn, 2000).

[Eng02] Wolfgang Engel: Direct3D ShaderX, Vertex and Pixel Shader Tips and Tricks (Plano, 2002).

[Fen09] Oliver Fendler: Konzeptionierung und prototypische Realisierung einer Simulation von Umgebungsdaten für funktionale Steuergerätetests im Bereich der Fahrerassistenzsysteme (Sindelfingen, 2009).

[Fer03] Randima Fernando, Mark J. Kilgard: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics (Boston, 2003).

[Feu10] Florian Feuerstein: Entwicklung eines datenbankbasierten "Level of Detail" Systems zur Beschleunigung von Echtzeitszenen mit zahlreichen Geometriemodellen (Friedberg, 2010).

[Har08] Nico Hartmann: Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik (Sindelfingen, 2001).

[Hen96] John L. Hennessy, David A. Patterson: Computer Architecture: A Quanitative Approach (Stanford, Berkeley, 1996).

[Lou07] Bavoil Louis: Efficient Multi-Fragment Effects on GPUs (University of Utah, 2007).

[Lue03] David Luebke: Level of Detail for 3D Graphics (San Francisco, 2003).

[Mic10] Microsoft Corporation: MSDN Library: Pipeline Stages (Direct3D 10) (Unterschleißheim, 2010).

[Mül07] Christian Müller: Durchgängige Verwendung von automatisierten Steuergeräte-Verbundtests in der Fahrzeugentwicklung (Dissertation, Universität Karlsruhe, 2007).

[NVI08] NVIDIA Corporation: PerfHUD 6 User Guide (Santa Clara, 2008).

[Ola98] Marc Olano: A Programmable Pipeline for Graphics Hardware (Chapel Hill, 1998).

[Rig02] Guennadi Riguer: Performance Optimization Techniques for ATI Graphics Hardware with DirectX 9.0 (Markham, 2002).

[Sch10] Dennis Schueller: Analyse und Optimierung fotorealistischer Bildgenerierung für funktionale Steuergerätetests (Sindelfingen, 2010).

[Sti05] Christoph Stiller: Fahrerassistenzsysteme mit maschineller Wahrnemung (Karlsruhe, 2005).

[Sto09] Jirk Stolze: Performancetests und Bottleneck-Analyse in Multischichtarchitekturen (Berlin, 2009).

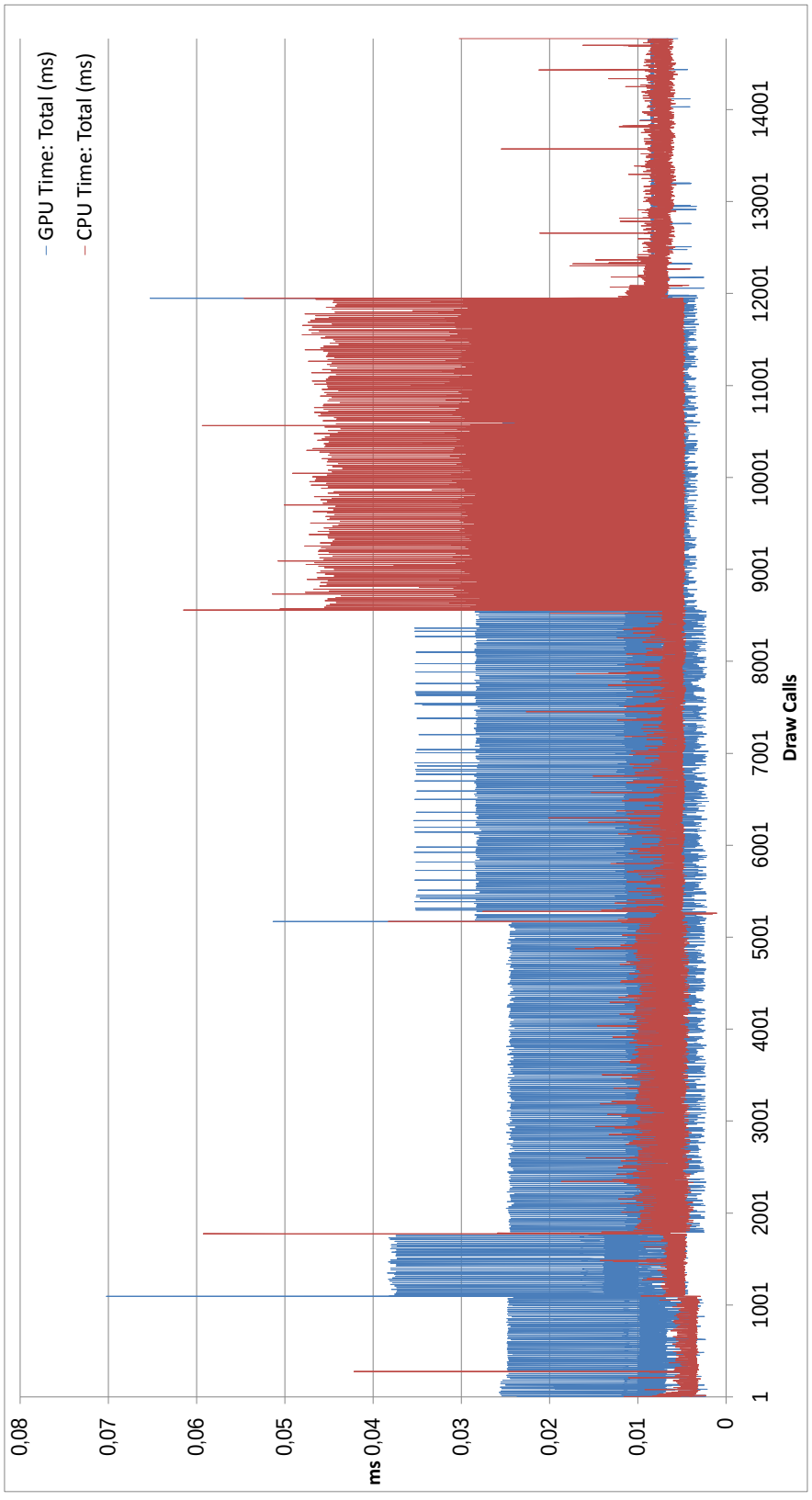
[Wit08] Dr. J. Wittmann: Bottleneck-Analyse (Hamburg, 2008).

[Xia96] Julie C. Xia, Amitabh Varshney: Dynamic View-Dependent Simplification for Polygonal Models (New York, 1996).

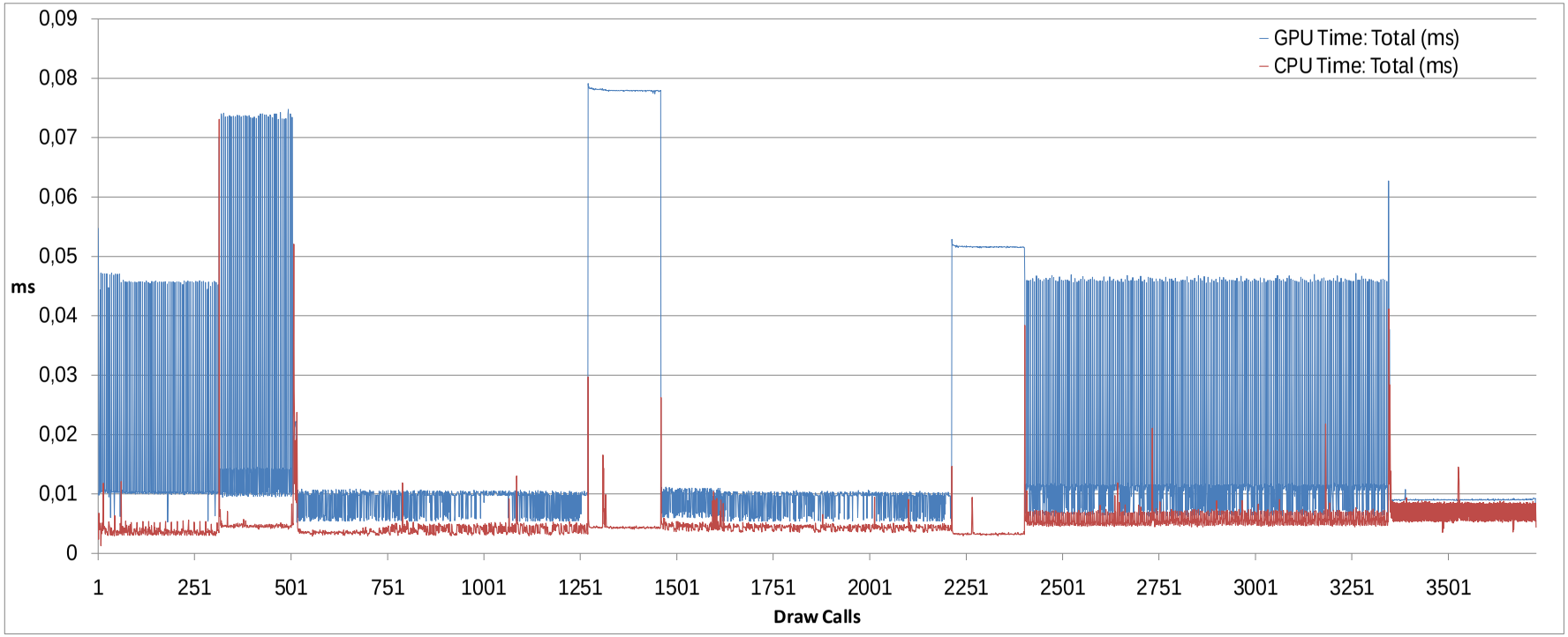
[Yoo06] Sung-Eui Yoon, Christian Lauterbach, Dinesh Manocha: Efficient Multi-Fragment Effects on GPUs (University of Utah, 2006).

[Zöb08] Dieter Zöbel: Echtzeitsysteme, Grundlagen der Planung (Heidelberg, 2008).

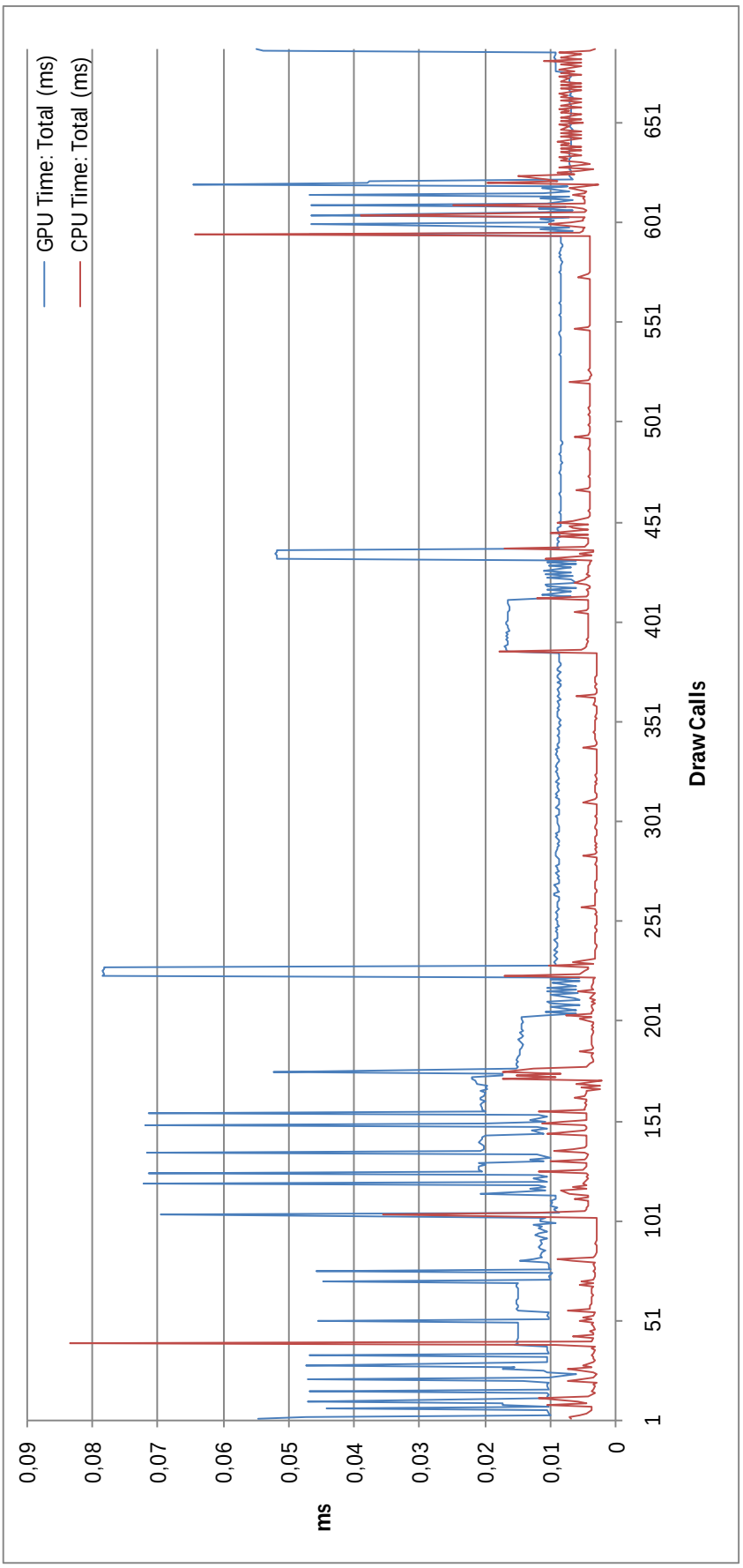
A Anhang



Performanceanalyse von zweihundert alten Fahrzeugen



Performanceanalyse von zweihundert neuen Fahrzeugen



Performanceanalyse von zweihundert neuen Fahrzeugen mit diskreten LOD-System