

# OpenGL-basiertes Software Occlusion Culling zur Beschleunigung des 3D-Renderings großer Datenmengen und komplexer Szenen

Projekt-Inf 3D-Rendering

Dominik Sellenthin

Michael Stegmaier

Gariharan Kanthasamy

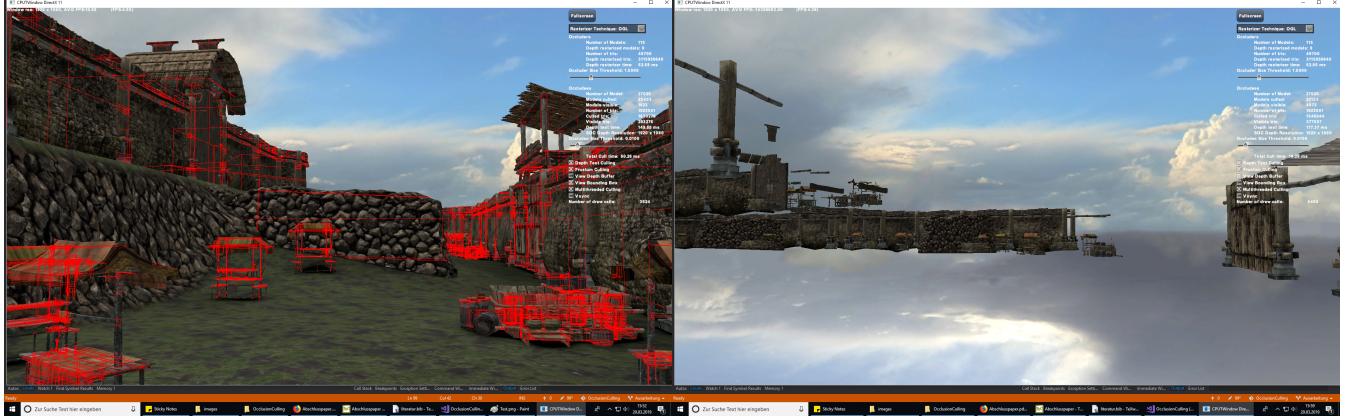


Abb. 1. Links: Testergebnisbild nach einem erfolgreichen Durchlauf der OGL-Methode aus der verwendeten Intel-Testszene. Rote Linien um Objekte stellen die Axis Aligned Bounding Box des jeweiligen Objekts dar. Rechts: Bild aller Objekte, die beim Occlusion Culling als verdeckt klassifiziert wurden und im linken Bild nicht gerendert wurden.

## Kurzbeschreibung—

In Bereichen des 3D-Rendering, sei es in der wissenschaftlichen Visualisierung oder in Computerspielen, ist die Rechenleistung der Grafikkarte schnell an ihrer Grenze, während die CPU kaum in Anspruch genommen wird. Es werden daher Techniken benötigt, die den Aufwand beim Rendern so gering wie möglich halten und gleichzeitig der ständig zunehmenden Dynamik gerecht werden. Eine dieser Techniken ist das Software Occlusion Culling (SOC). Ziel des Occlusion Cullings ist es, komplett parallel zum GPU-Rendering des aktuellen Frames, festzustellen, welche Objekte in der kommenden Szene zu sehen sind und welche Objekte nicht gerendert werden müssen. Damit die GPU entlastet wird, ist es Wünschenswert, die Berechnungen des Occlusion Cullings auf die CPU auszulagern. Um das zu erreichen und um die vollständige Rechenleistung moderner Multi-Core CPUs zu gewährleisten, wird in dieser Arbeit ein bereits vorhandener Software-Rasterizer, Mesa 3D, verwendet, der zusätzlich die OpenGL API zur Verfügung stellt. In dieser Arbeit wird daher in einem von Intel entwickelten SOC-Framework eine auf Mesa-3D basierende Occlusion Culling Technik implementiert und folglich auf ihre Tauglichkeit mit bereits im Framework existierenden Techniken verglichen.

## 1 EINLEITUNG

Heutige Anwendungen haben immer höhere Ansprüche an die Leistung der Engines(?) und der Wunsch nach besserer Performanz und höheren Bildraten (frames per second, FPS) ist groß. Hinzukommt, dass in modernen Anwendungen die zu rendernden Szenen immer weniger statisch und immer mehr dynamisch werden, wie es beispielsweise in Computerspielen der Fall ist. Um dieser Dynamik gerecht zu werden, wird sich von Verfahren mit potenziellen Sichtbarkeitsmengen entfernt [3] und es wird sich einer Methode namens *Occlusion Culling* (*to occlude* = verdecken, *to cull* = aussondern, herausfiltern) bedient. Ziel des Occlusion Cullings ist es, noch vor dem Rendering der nächsten Szene, herauszufinden, welche Objekte in der kommenden Szene sichtbar sind und welche nicht. Im Wesentlichen gilt es dabei zuerst mit einer ausgewählten Teilmenge der Objekte einen geeigneten Tiefenpuffer (Z-Buffer) zu generieren und darauffolgend mit Hilfe von Occlusion Queries alle Objekte zu bestimmen, die noch sichtbar sind (auch Z-Buffering oder Tiefentest genannt). Das Ergebnis der Occlusion Queries kann anschließend ohne nennenswerte Latenz verwendet werden, um der GPU mitzuteilen, welche Objekte gerendert werden sollen, so dass unnötiger Rechenaufwand der GPU vermieden wird.

CPU	OCF0	OCF1	OCF2	...
GPU	DoOtherStuff	RenderFO	RenderF1	...
Schritt	0	1	2	3

Abb. 2. Prozessor berechnet parallel zum Rendering der Grafikkarte, welche Objekte in der nächsten Szene zu sehen sind.

Bei Anwendungen, die ohnehin schon enormen Rechenaufwand benötigen und die maximale Rechenkapazität der Grafikkarte schnell ausreizen, ist es schwierig den zusätzlichen Mehraufwand ebenfalls der Grafikkarte aufzuerlegen. Es bietet sich daher an, den Mehraufwand dem wenig genutzten Prozessor zu übergeben, der dann komplett parallel zur Grafikkarte das Occlusion Culling in einem Vorverarbeitungsschritt durchführen soll, siehe Abb. 2.

Moderne Prozessoren besitzen mittlerweile einige separat nutzbare Kerne, die parallel verwendet werden können, um das Occlusion Culling so effektiv und effizient wie möglich zu implementieren. Damit die gesamte Rechenleistung der Prozessoren auch genutzt wird, wird in dieser Arbeit auf den Software-Rasterisierer Mesa 3D zurückgegriffen, der außerdem mit der bewährten OpenGL API arbeitet.

Implementiert wurde die OGL-Methode in einem Software Occlusion Culling (SOC) Framework, das von Intel frei zur Verfügung gestellt wird [1]. Das Framework eignet sich sehr gut für die Implemen-

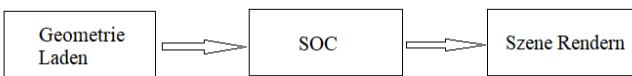


Abb. 3. Verwende größtenteils vorhandene Strukturen, um die verwendete Geometrie zu laden. Anschließend erfolgt das Culling durch die neue OGL-Culling-Methode. Zum Schluss wird die nicht gecullte Geometrie an das Framework zurückgegeben, damit das Ergebnis gerendert wird.

tierung einer neuen SOC-Methode, da es sowohl essentielle Ablaufstrukturen als auch eine ausreichend große und komplexe Testszene zur Verfügung stellt, mit der eine Evaluation der OGL-Methode ermöglicht wird, siehe Abb. 1 links.

## 2 RELATED WORK

Diese Arbeit orientiert sich stark an Intels SOC-Framework, in das die in dieser Arbeit entwickelte OGL-Methode eingebettet wurde. Da das Framework bereits alle notwendigen Strukturen für schon bestehende Methoden besitzt (siehe Abb. 3), gestaltet sich die Erweiterung um eine weitere Methode größtenteils unkompliziert. Hinzukommen lediglich Veränderungen der Routinen zum Laden der Testszenen und die Implementierung des Cullings selber. Vorhanden sind unter anderem Methoden, die mit *Streaming SIMD Extension* (SSE) und *Intel Advanced Vector Extension* (AVX) arbeiten. Außerdem ist noch eine optimierte Variante der AVX-Technik vorhanden, *Masked Software Occlusion Culling* (MSOC) [3], die mit einer abgewandelten Form des hierarchischen Z-Buffers (HiZ) [2] arbeitet, der durch einen Software-Rasterisierer berechnet wird.

Hasselgren et al. [3] stellen in ihrer Arbeit einen Algorithmus vor, der durch seinen effizienten HiZ die Performanz signifikant verbessert. Anstatt wie in früheren Arbeiten Pixel als kleinste Einheit zu betrachten, werden nun *Kacheln* verwendet. Kacheln sind dabei lediglich eine Zusammenfassung mehrerer aneinanderliegender Pixel. Da AVX2 ermöglicht, 8 SIMD Instruktionen mit 32-Bit Präzision auszuführen, wurde für die Kacheln eine Größe von 32x8 Pixeln gewählt. Indem Bitmasken von rechts und links in die Kacheln geschoben werden, wird am Ende eine Abdeckungsmaske erhalten, die angibt, welche Pixel in einer Kachel verdeckt werden [3]. Während ihr Algorithmus keine 100% Präzision garantiert - *false positives* sind möglich - bewegt sich der Fehler in gleicher Größenordnung wie bei bisherigen Algorithmen. Ein wichtiger Faktor für die Performanz ihres Algorithmus ist die Reihenfolge, in der die Objekte gerendert werden. Die Objekte sollten bestmöglichst von vorne nach hinten gerendert werden. Dadurch werden die wichtigsten Occluder als erstes rasterisiert (dargestellt) und für den Fall, dass die Zeit für Occlusion Culling begrenzt ist, wird trotzdem ein nahezu optimales Ergebnis erzielt [3].

Bei einem Vergleich zwischen einem HiZ-Algorithmus und dem MSOC - bei voller Auflösung (1920\*1080 Pixel) und Single-Core Performanz - geht hervor, dass der MSOC-Algorithmus 2% weniger Dreiecke als der HiZ wegwirft, aber dennoch eine bessere Performanz erzielt. Mit beiden Occlusion Culling Algorithmen kann eine 1,5-7x schnellere Total Frame Time gegenüber Rendering mit nur Frustum-culling erreicht werden. In einem zweiten Test in einer wesentlich komplexeren Szene mit 143k Occluder-Meshes ist die Occlusion Culling Time des MSOC-Algorithmus zeitweise bis zu 10x schneller als die des HiZ-Algorithmus. Die Skalierbarkeit des MSOC ist der des HiZ deutlich überlegen. Mit steigender Größe der Dreiecke steigt auch der Performanzunterschied zwischen dem MSOC und dem HiZ. Im Allgemeinen soll der MSOC-Algorithmus 3x schneller sein und gleichzeitig 98% aller Dreiecke wegwerfen als die bisherigen Algorithmen bei einem nur geringen Memory Overhead.

## 3 SOFTWARE OCCLUSION CULLING

Die Menge der Objekte, die es zu Rendern gilt, wird in zwei Mengen aufgeteilt. Zum einen gibt es die *Occluder*. Occluder sind eine Menge von Objekten, die groß genug sind, dass es wahrscheinlich ist, dass sie



Abb. 4. Links: Menge der Occluder ohne Occludees, die in dieser Kameraeinstellung zu sehen sind, vgl. Abb. 1 linkes Bild. Rechts: Testszene ohne Occluder, es sind ausschließlich Occludees zu sehen, vgl. ebenfalls Abb. 1.

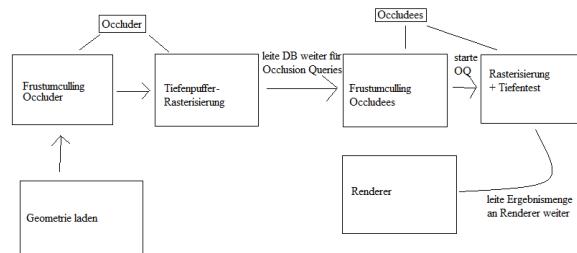


Abb. 5. SOC-Ablauf: Nach Laden der Occludermenge wird im ersten Schritt durch Rasterisieren der Occluder der Tiefenpuffer generiert. Im zweiten Schritt werden die Occlusion Queries gestartet, die durch Rasterisierung und Tiefentests bestimmen, welche Objekte der Occludee-Menge sichtbar sind und welche nicht. Die Ergebnismenge wird an den Renderer weitergeleitet.

andere Objekte verdecken. Zum anderen gibt es *Occludees*. Occludees sind all diejenigen Objekte die potenziell von Occludern verdeckt werden (das heißt, sie beinhalten ebenfalls alle Occluder). Sowohl Occluder als auch Occludees liegen dabei in zwei Formen vor. Einmal als Netz, bestehend aus Punkten (Mesh), das zur exakten Darstellung des Objekts in der gerenderten Szene dient und einmal in Form einer *Axis Aligned Bounding Box* (AABB), die sowohl zum Frustum-culling als auch zum (Tiefen-)Rasterisieren verwendet wird. AABBs eignen sich wegen ihrer einfachen geometrischen Form sehr gut, um erste grobe Tests durchzuführen, ob ein Objekt überhaupt von der Kamera gesehen werden kann (Frustumculling) und dementsprechend für die folgende Rasterisierung beim Occlusion Culling in Frage kommt.

Occlusion Culling besteht im Wesentlichen aus zwei Schritten. Als erstes wird der Tiefenpuffer auf Basis einer Occludermenge beschrieben. Diese Occluder werden in einem ersten Renderingdurchlauf rasterisiert, jedoch ohne die Objekte tatsächlich zu zeichnen, und der Tiefenpuffer wird entsprechend der Occludermenge beschrieben.

Schritt zwei besteht darin Occlusion Queries durchzuführen. Bei den Occlusion Queries werden die Bounding Boxen aller Occludees gegen den im vorherigen Schritt erstellten Tiefenpuffer getestet und es wird geprüft, ob die Occludees den Tiefentest bestehen oder nicht, sprich, ob die Occludees von einem Occluder komplett verdeckt werden oder (teilweise) sichtbar sind.

Unmittelbar vor jedem dieser beiden Schritte wird zusätzlich noch Frustumculling durchgeführt, um die Menge der zu testenden Objekte bereits im Vorfeld einzuschränken und somit weiter an Performanz zu gewinnen. Der grundsätzliche Ablauf in jedem Frame sieht also entsprechend Abb. 5 aus.

### 3.1 Frustumculling

Frustumculling lässt nur Objekte passieren, die sich im Sichtbereich der Kamera befinden und kann somit je nach Kameraposition einen großen Teil der Objekte aus der Menge der zu rendernden Objekte herausnehmen. Dafür wird die AABB des Objekts gegen jede der sechs Ebenen des Frustums getestet, ob sich die AABB *vollständig* außerhalb einer dieser Ebenen befindet. Ist das der Fall, kann das

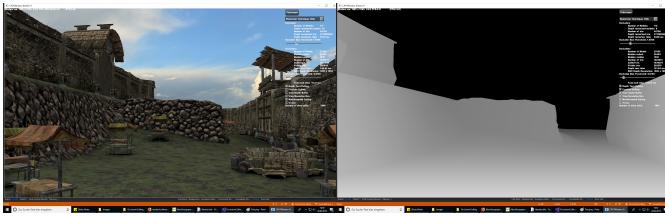


Abb. 6. Links: Testbild der Szene. Rechts: Der dazugehörige Tiefenpuffer, der in jedem Frame einmal berechnet wird und später für die Tiefentests der Occlusion Queries verwendet wird.

Objekt als nicht sichtbar markiert werden und wird im weiteren Verlauf nicht weiter betrachtet. Dieser Zwischenschritt ist zwar optional, ist aber den Rechenaufwand wert, denn er bringt eine enorme Leistungssteigerung von teilweise über 100%.

**(Optional! Vielleicht an anderer Stelle?)** Anmerkung zum Frustum-culling: Während die Performance des Frustum Cullings sehr konstant bleibt, sind bei anderen Occlusion Culling Algorithmen größere Schwankungen erkennbar, da ein großer Occluder im Vordergrund potenziell alle Occludees hinter ihm überdecken kann und damit die Berechnung stark vereinfacht [3].

### 3.2 Tiefenpuffer Rasterisierung

Alle Occluder, die nach dem Frustumculling als sichtbar markiert sind, werden in diesem Schritt verwendet, um einen Tiefenpuffer zu generieren. Dazu werden alle sichtbaren Objekte als eine Occludermenge „gerendert“ (es wird lediglich ein Tiefenpuffer erzeugt ohne die Objekte tatsächlich zu zeichnen). Das Rendering rasterisiert die Occluder, das heißt, die Occluder werden auf den Bildschirm (screen space) transformiert und erzeugt anschließend den Tiefenpuffer, indem an den Stellen auf dem Bildschirm, an denen sich das Objekt befindet, die Tiefenwerte des Objekts gespeichert werden, siehe Abb. 6.

### 3.3 Occlusion Queries

An dieser Stelle sei erwähnt, dass die gesamte Occludeemenge zu Beginn des Programms einmal hochgeladen wurde (wohin?) und im Programm selber nur via Indexlisten zugegriffen wird, so dass zusätzliche Ladezeiten vermieden werden. Die Realisierung der Occlusion Queries gestaltet sich durch die von der OpenGL API zur Verfügung gestellten *glQuery* sehr unkompliziert. Zu Anfang werden für alle Occludees, die innerhalb des Frustums sind, eine Occlusion Query gestartet. Die Occludees werden wie die Occluder rasterisiert, allerdings wird in diesem Schritt der Tiefenpuffer nicht überschrieben, sondern es wird ausschließlich getestet, ob der Tiefenwert des Occludees kleiner als der des Tiefenpuffers ist. Die Occlusion Query testet also, ob ein Teil des Occludees bei einem potenziellen Renderdurchlauf sichtbar wäre. In diesem Fall liefert die Occlusion Query „true“ zurück, andernfalls „false“. Nachdem alle Occlusion Queries ihre Berechnungen beendet haben, werden zum Schluss alle Ergebnisse abgefragt und an das Framework für den weiteren Renderingverlauf weitergeleitet.

## 4 ERGEBNISSE

Probleme mit Occlusion Queries (lange Wartezeit bei Ergebnis holen), meiste Zeit in OQ → Performanceverlust

Die dokumentierten Ergebnisse wurden mit dem SOC-Framework von Intel berechnet und verwenden die dortige Testszene einer Burg beziehungsweise eines Marktplatzes, siehe Abb. 1. Die Burg umfasst 115 Occluder mit 48700 Dreiecken und 27025 Occludees mit  $\approx 1923000$  Dreiecken. Sofern nicht anders angeben, wurden sämtliche Ergebnisse mit einer Kamerafahrt, wie sie in Abb. 7 angedeutet ist, über 100 Frames erstellt. Der Tiefenpuffer ist standardmäßig auf eine Auflösung von 1920x1080 Pixel gesetzt. Die Kamera startet mit Blick auf die Burg aus einiger Entfernung und fliegt auf die Burg zu. An der Burg angekommen dreht sich die Kamera in Richtung Boden und entfernt sich anschließend wieder von der Burg. Die Anzahl der Objekte



Abb. 7. Links: Startposition der Kamerafahrt. Mitte: nähert sich in einem Kreisbogen der Brücke an. Rechts: Die Kamera verlässt die Szene in gleichem Bogen, wie sie sich angenähert hat.

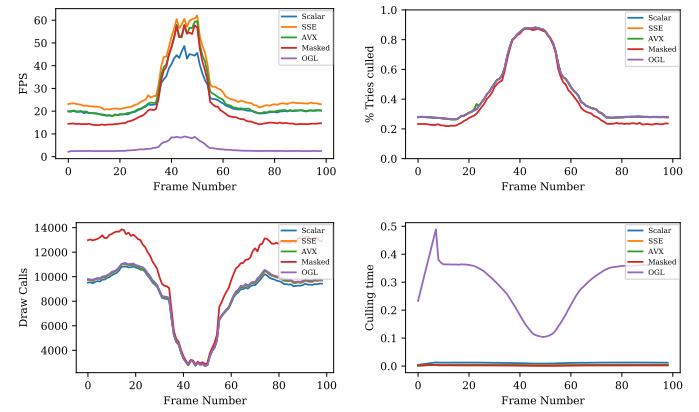


Abb. 8. Vergleich der 5 verschiedenen Varianten

im Sichtfeld ist am Anfang sehr hoch, sinkt in der Mitte der Kamerafahrt ab und steigt am Ende wieder auf den Startwert. Als Hardware wurde ein Rechner mit einem Intel Core i9-7900X, einer NVIDIA Titan Xp und 64 GB RAM verwendet.

In Figure 8 ist die Leistung der 4 verschiedenen Occlusion Culling Methoden im Intel-Framework im Vergleich zu unserer eigenen Implementierung (OGL) zu sehen. Die Durchschnittswerte sind in Table 1 zu sehen. Die Anzahl der gecullten Elemente und die Anzahl der Drawcalls ist bei allen Methoden sehr ähnlich. Nur das Masked Occlusion Culling cultt  $\approx 3,5\%$  weniger und führt deswegen mehr Drawcalls aus.

In Figure 9 wurde unser Programm mit fünf unterschiedlichen Auflösungen des Tiefenbuffers ausgeführt. Je niedriger die Auflösung ist, desto höhere Frameraten können erreicht werden. Betrachtet man die Werte aus Table 2 stellt man fest, dass durch die niedrige Auflösung beim Culling und beim Tiefentest Zeit gespart werden kann.

- Evaluation 1: alle 5, Kamerafahrt 1
- Evaluation 2: alle 5, Kamerafahrt 2
- Evaluation 3: alle 5, DB\_Very\_Large vs DB\_Small
- Evaluation 4: OGL, alle 5 Auflösungen
- Evaluation 5: OGL, mit und ohne Frustum culling
- Evaluation 6: alle 5 + OGL ohne Mesa
- Evaluation 7: OGL + SSE, MT / MT + Frustum Culling / MT + Frustum Culling + Depth Test Culling
- Evaluation 8: OGL + SSE, Depth Test Tasks 5 / 20 / 100

	Scalar	SSE	AVX	Masked	OGL
FPS	24.21	29.30	26.21	22.12	3.63
Tries culled (%)	43.97	43.88	43.85	40.41	43.81
Drawcalls	8299	8442	8471	10529	8514
Culling t (ms)	11.30	4.24	3.73	1.99	292.60
Depth Test t (ms)	6.90	1.59	1.72	1.45	228.38
Rasterizer t (ms)	4.36	2.65	2.01	2.01	64.21

Tabelle 1. Vergleich der 5 verschiedenen Varianten

# Projekt-Inf 3D-Rendering: Software Occlusion Culling

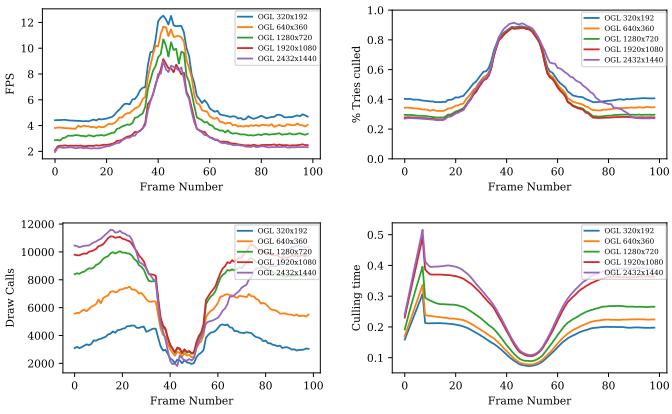


Abb. 9. OGL mit unterschiedlicher Auflösung des Tiefenbuffers

	320x192	640x360	1280x720	1920x1080	2432x1440
FPS	6.03	5.35	4.53	3.59	
Tries culled (%)	51.88	47.65	44.69	43.81	
Draw Calls	3640	5759	7725	8515	
Culling t (ms)	171.02	188.8	223.72	295.06	
Depth Test t (ms)	108.86	125.27	160.27	231.08	
Rasterizer t (ms)	62.15	63.53	63.46	63.97	

Tabelle 2. OGL mit unterschiedlicher Auflösung des Tiefenbuffers

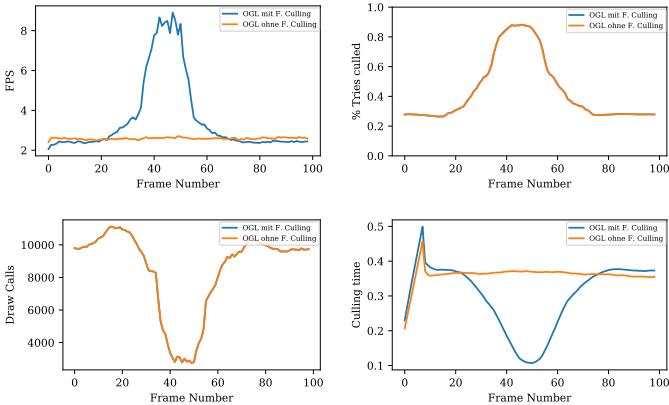


Abb. 10. OGL mit und ohne Frustum Culling

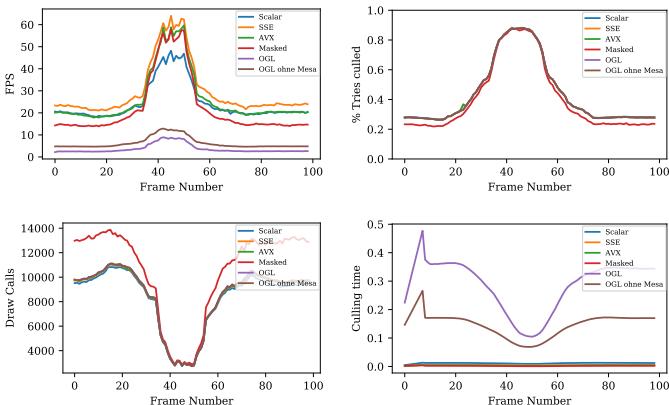


Abb. 11. Vergleich der 5 verschiedenen Varianten + OGL ohne OS Mesa

## 5 FAZIT

Das ist ein Testsatz für das Fazit.

## 6 FUTURE WORK

Occludee zu Paketen schnüren und als Paket hochladen. Ziel ist es, weniger Occlusion Queries bei gleichem Ergebnis zu generieren und dadurch an Performance zu gewinnen. Auswahl der Occludee, die zu einem Paket zusammengeschnürt werden ist dabei von großer Wichtigkeit und entscheidend für das Ergebnis, sowohl im Hinblick auf die Performance, als auch auf die Korrektheit des Ergebnisses (werden alle Occludee angezeigt, die sichtbar sind oder fehlen offensichtliche Objekte in der Szene).

## LITERATUR

- [1] C. Chandrasekaran, D. McNabb, K. Kiefer, M. Faconnneau, and F. Giesen. Software occlusion culling. <https://software.intel.com/en-us/articles/software-occlusion-culling>, 2013-2016.
- [2] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM, 1993.
- [3] J. Häggström, M. Andersson, and T. Akenine-Möller. Masked Software Occlusion Culling. In U. Assarsson and W. Hunt, editors, *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, 2016.