

# 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

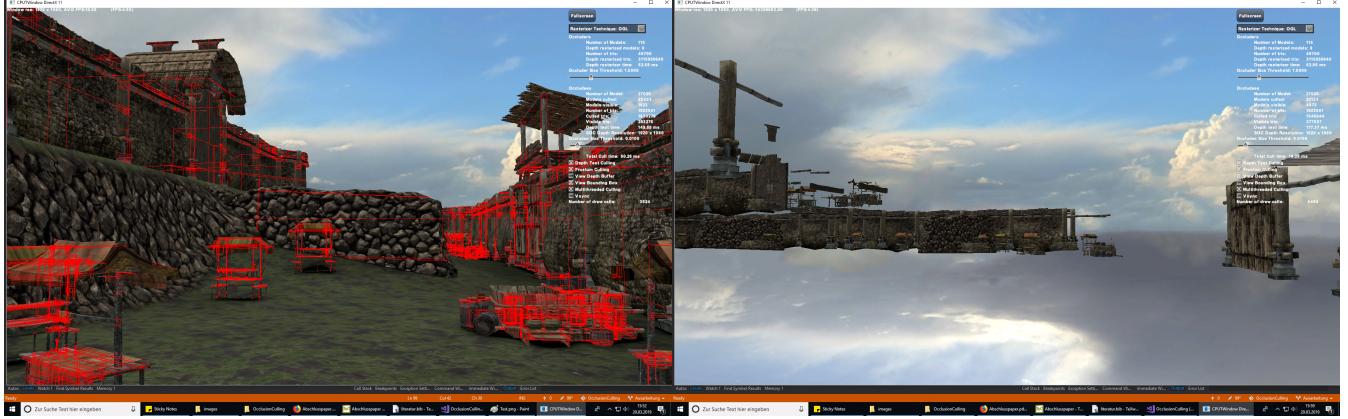


Abb. 1. Links: Testergebnisbild nach einem erfolgreichen Durchlauf der OGL-Methode aus der verwendeten Intel-Testszene. Rote Linien um Objekte stellen die Axis Aligned Bounding Box des jeweiligen Objekts dar. Rechts: Bild aller Objekte, die beim Occlusion Culling als verdeckt klassifiziert wurden und 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. Damit die GPU entlastet wird, wird ein Software-Rasterisierer eingesetzt, so dass ein Teil der Berechnungen in einem Vorverarbeitungsschritt auf die CPU ausgelagert wird. Um die vollständige Rechenleistung moderner Multi-Core CPUs zu gewährleisten, wird in dieser Arbeit ein bereits vorhandener Rasterisierer, Mesa 3D, verwendet, der zusätzlich die OpenGL API zur Verfügung stellt. Ziel des Software-Rasterisierers ist es, komplett parallel zum GPU-Rendering des aktuellen Frames, in zwei Schritten festzustellen, welche Objekte in der kommenden Szene zu sehen sind. Im Wesentlichen gilt es dabei zuerst einen geeigneten Tiefenpuffer zu generieren und anschließend mittels Occlusion Queries alle Objekte zu bestimmen, die noch sichtbar sind (auch Z-Buffering genannt). Das Ergebnis der Occlusion Queries kann nun ohne nennenswerte Latenz verwendet werden, um der GPU mitzuteilen, welche Objekte gerendert werden sollen, so dass unnötiger Rechenaufwand der GPU vermieden wird ( 160).

## 1 EINLEITUNG

Egal, ob im Bereich der wissenschaftlichen Visualisierung oder in anderen 3D-Rendering-Bereichen, heutige Anwendungen haben immer höhere Ansprüche an die Leistung der Engines(?) und der Wunsch nach besserer Performance 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 Rechen-

CPU	OCF0	OCF1	OCF2	...
GPU	DoOtherStuff	RenderF0	RenderF1	...
Schritt	0	1	2	3

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

aufwand der GPU vermieden wird.

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 separate 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 ge-

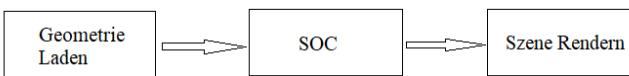


Abb. 3. Verwende größtenteils vorhandene Strukturen, um die verwendete Geometrie zu laden. Anschließend erfolgt das Culling durch die neue OGL-Culling-Methode. Zum Schluss wird die nicht gecullte Geometrie an das Framework zurückgegeben, damit das Ergebnis gerendert wird.

stellt wird [1]. Das Framework eignet sich sehr gut für die Implementierung 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 Testszene 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 Performance 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 Performance 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].

## 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 andere Objekte verdecken. Zum anderen gibt es Occludees. Occludees sind all diejenigen Objekte die potentiell 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 Frustumculling 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. Ist die AABB außerhalb des Sichtfeldes (Frustum) der Kamera, kann das



Abb. 4. 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 ebenfalls nicht sichtbar sein und bereits aus der Menge der zu betrachtenden Objekte genommen werden.

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, siehe Abb. 4. Schritt zwei besteht darin Occlusion Queries durchzuführen. Bei den Occlusion Queries werden die Bounding Boxes 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 Performance zu gewinnen. Der grundsätzliche Ablauf in jedem Frame sieht also wie folgt aus:

Occluder Frustumculling → Tiefenpuffer Rasterisierung  
→ Occludee Frustumculling → Occlusion Queries  
→ Rendern aller passierten Occludees

### 3.1 Frustumculling

Frustum Culling rendert nur Objekte die sich im Sichtbereich der Kamera befinden und kann somit je nach Kameraposition einen Teil der Objekte cullen.

Vorteil von OC gegenüber FC (**ggf bei Ergebnisse**): Während die Performance des Frustum Cullings sehr konstant bleibt, sind bei den beiden 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.

### 3.2 Tiefenpuffer Rasterisierung

### 3.3 Occlusion Queries

## 4 ERGEBNISSE

Daten zur Testszene

Als Basiswert für die Performance des Masked Occlusion Algorithmus (MOC) wurde ein normales Rendering ohne Occlusion Culling, aber mit Frustum Culling verwendet. Als alternativer Algorithmus wird der „Hierarchical Z Buffer algorithm“ (HiZ) evaluiert. Alle Messungen wurden mit voller Auflösung (1920\*1080 Pixel) ausgeführt, außerdem wurde nur die Single-Core Performance betrachtet. Der MOC Algorithmus ist im Vergleich zum HiZ Algorithmus etwas vorsichtiger und cultt 2% weniger Dreiecke, erreicht allerdings trotzdem eine bessere Performance. Bei einem ersten Test mit einer kleinen Kamerafahrt im Intel Occlusion Culling Framework wurde die Framezeit, also die Zeit für die Berechnung eines Frames gemessen. Szene enthält 49k Dreieckige Occluder meshes.

Mit den Occlusion Culling Algorithmen kann eine 1,5-7x schnellere Total Frame Time erreicht werden. Sind die Berechnungen sehr einfach und können sehr viele Objekte gecullt werden, ist die Performance von HiZ und MOC sehr gut und fast identisch.

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

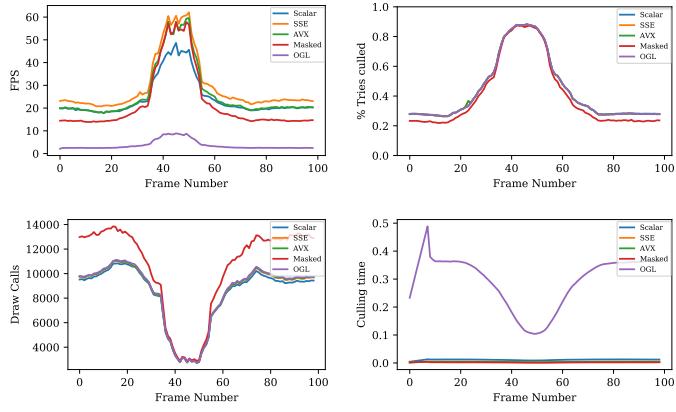


Abb. 5. Vergleich der 5 verschiedenen Varianten

	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

Je komplexer die Berechnungen sind und je genauer alle Objekte auf potenzielles culling überprüft werden müssen, desto besser schneidet der MOC Algorithmus ab. In einzelnen Frames erreicht er so teilweise eine etwas mehr als doppelt so schneller Total Frame Time. In einem zweiten Test wurde eine wesentlich komplexere Szene mit 143k Occluder meshes für die Messung der Daten benutzt. Die Occlusion Culling Time des MOC Algorithmus ist ca. 10x so schnell wie die des HiZ Algorithmus. Die Autoren halten fest, dass eine 10-fache Beschleunigung durch ihren Algorithmus vermutlich nicht immer angenommen werden kann, ihr Algorithmus jedoch sehr robust gegenüber komplexen Occluder Strukturen ist. Um die Skalierbarkeit des MOC Algorithmus zu testen, wurden 32k zufällige Occluder Dreiecke mit unterschiedlicher Größe erzeugt. Während die Performance bei einer Größe der Dreiecke von 10x10 praktisch identisch ist, gewinnt der MOC Algorithmus im Vergleich zum HiZ Algorithmus mit steigender Größe der Dreiecke immer mehr an Abstand. Als Endergebnis halten sie fest, dass ihr Algorithmus 3x schneller ist als die bisherigen Algorithmen und gleichzeitig nur einen geringen Memory Overhead hat. Mit ihrem Algorithmus können 98% aller Dreiecke gecullt werden.

Unsere Ergebnisse wurden mit dem Intel-Occlusion-Culling-Framework berechnet und verwenden die dortige Testszene einer Burg. Die Burg hat 115 Occluder mit 48700 Dreiecken und 27025 Occludee Model mit  $\approx 1923000$  Dreiecken. Unsere Ergebnisse wurden, sofern nicht anders angeben, mit einer Kamerafahrt über 100 Frames erstellt. Der Tiefenbuffer ist standardmäßig auf 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 Richtung Boden und entfernt sich anschließend wieder von der Burg. Die Anzahl der Objekte 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 5 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 Dra-

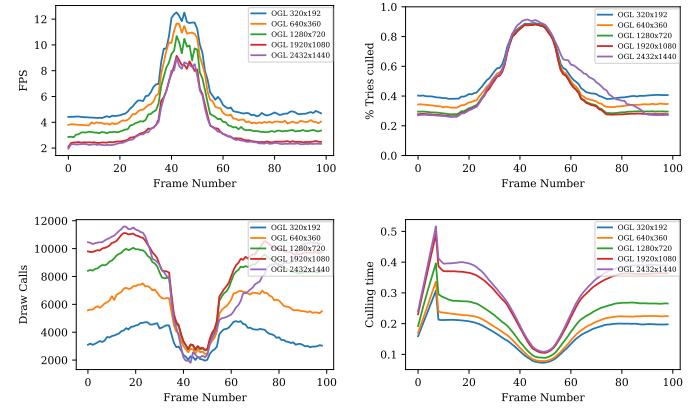


Abb. 6. OGL mit unterschiedlicher Auflösung des Tiefenbuffers

	320x192	640x360	1280x720	1920x1080	2432x1440
FPS	6.03	5.35	4.53	3.59	3.38
Tries culled (%)	51.88	47.65	44.69	43.81	48.75
Draw Calls	3640	5759	7725	8515	7915
Culling t (ms)	171.02	188.8	223.72	295.06	312.00
Depth Test t (ms)	108.86	125.27	160.27	231.08	249.00
Rasterizer t (ms)	62.15	63.53	63.46	63.97	63.00

Tabelle 2. OGL mit unterschiedlicher Auflösung des Tiefenbuffers

walls ist bei allen Methoden sehr ähnlich. Nur das Masked Occlusion Culling cultt  $\approx 3,5\%$  weniger und führt deswegen mehr Drawcalls aus.

In Figure 6 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

## 5 FAZIT

Das ist ein Testsatz für das Fazit.

## LITERATUR

- [1] C. Chandrasekaran, D. McNabb, K. Kiefer, M. Fauconneau, 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. Hasselgren, 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.

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

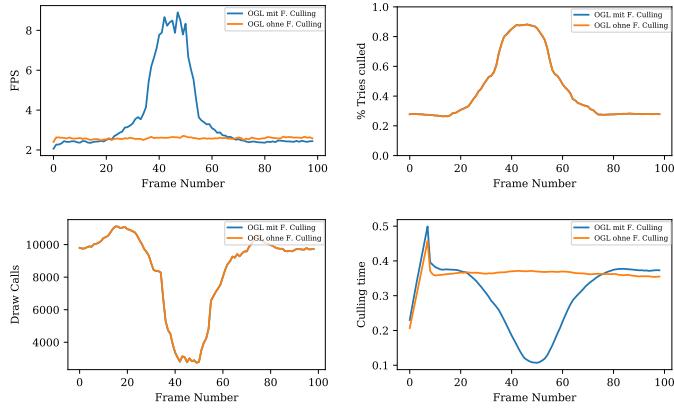


Abb. 7. OGL mit und ohne Frustum Culling

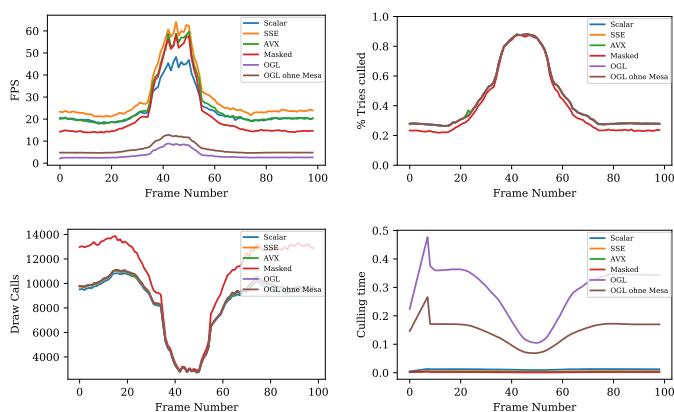


Abb. 8. Vergleich der 5 verschiedenen Varianten + OGL ohne OS Mesa