

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

Projekt-Inf 3D-Rendering

Dominik Sellenthin

Christian Stegmaier

Gariharan Kanthasamy



Abb. 1. Links: Testergebnisbild nach einem erfolgreichen Durchlauf der OOC-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, die CPU auszunutzen 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 im nächsten Frame 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 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. Hinzu kommt, dass in modernen Anwendungen die zu rendernden Szenen immer weniger statisch und immer mehr dynamisch werden, wie es beispielsweise in Computerspielen der Fall ist. Deshalb wird sich von traditionellen Verfahren mit potenziellen Sichtbarkeitsmengen (PSM) entfernt [4]. Ein großes Problem bei PSM in Kombination mit dynamischen Objekten ist, dass PSM mit festen Sichtbarkeitsmengen arbeitet, die je nach Region bestimmte Objekte enthält. Diese Listen werden üblicherweise in einem Vorverarbeitungsschritt berechnet und müssten so für jedes dynamische Objekt, das eine Region betritt, aktualisiert werden [2]. Um dieser Dynamik gerecht zu werden, wird sich deshalb einer Methode namens *Occlusion Culling* (*to occlude = verdecken, to cull = aussondern, herausfiltern*) bedient. Ziel des Occlusion Cullings ist es, noch vor dem Rendering des nächsten Frames, herauszufinden, welche Objekte im nächsten Frame 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

CPU	OC Frame 0	OC Frame 1	OC Frame 2	OC Frame 3	...	OC Frame n
GPU	-	Render Frame 0	Render Frame 1	Render Frame 2	...	Render Frame n-1
Frame	0	1	2	3	...	n

Abb. 2. Während ein Frame gerendert wird, berechnet der Prozessor parallel zum Rendering bereits, welche Objekte im nächsten Frame zu sehen sind.

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 der GPU übergeben werden, um festzustellen welche Objekte gerendert werden sollen, so dass unnötiger Rechenaufwand 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 mehrere separate nutzbare Kerne, die parallel verwendet werden können, um das Occlusion

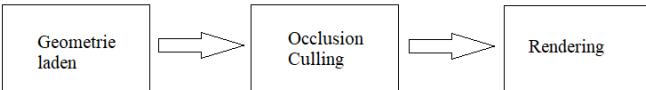


Abb. 3. Verwende größtenteils vorhandene Strukturen, um die verwendete Geometrie zu laden. Anschließend erfolgt das Culling durch die neue OOC-Culling-Methode. Zum Schluss wird die nicht gecullte Geometrie an das Framework zurückgegeben, damit das Ergebnis gerendert wird.

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 mit dem Gallium Ipvmpipe Treiber zurückgegriffen, der mit der bewährten OpenGL API arbeitet.

Implementiert wurde die OOC-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 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 OOC-Methode ermöglicht wird, siehe Abb. 1 linkes Bild.

## 2 RELATED WORK

Diese Arbeit orientiert sich stark an Intels SOC-Framework, in das die in dieser Arbeit entwickelte OOC-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. Hinzu kommen lediglich Veränderungen der Routinen zum Laden der Testszene und die Implementierung des Cullings selbst. Vorhanden sind unter anderem Methoden, die mit *Streaming SIMD Extensions* (SSE) und *Intel Advanced Vector Extensions* (AVX) arbeiten. Außerdem ist noch eine optimierte Variante der AVX-Technik vorhanden, *Masked Software Occlusion Culling* (MOC) [4], die mit einer abgewandelten Form des hierarchischen Z-Buffers (HiZ) [3] arbeitet, der durch einen Software-Rasterisierer berechnet wird.

Hasselgren et al. [4] stellen in ihrer Arbeit einen Algorithmus vor, der durch seinen effizienten HiZ die Performanz signifikant verbessert. Einer der Unterschiede ihres HiZ ist die Aufteilung von Tiefen- und Verdeckungsdaten in zwei separate Ebenen. Zum einen in eine Referenzebene (reference layer), die die aktuellen Tiefenwerte enthält und die für die späteren Tiefentests verwendet wird. Zum anderen in eine Arbeitsebene (working layer), die angibt, welche Bereiche der Referenzebene bereits verdeckt sind und zusätzlich die dazugehörigen Tiefenwerte enthält. Ziel der Aufteilung ist die Reduzierung des Speicherbedarfs. Der zweite Unterschied ist, dass anstatt wie in früheren Arbeiten Pixel als kleinste Einheit zu betrachten, nun *Kacheln* verwendet werden. 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. 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.



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.

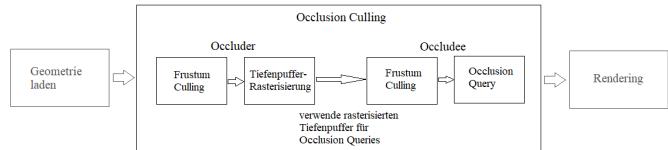


Abb. 5. SOC-Ablauf: Nach Laden der Occludermenge wird im ersten Schritt durch Rasterisieren der Occluder der Tiefenpuffer generiert. Im zweiten Schritt werden die Occlusion Queries gestartet, die durch Rasterisierung und Tiefentests bestimmen, welche Objekte der Occludee-Menge sichtbar sind und welche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

Bei einem Vergleich zwischen einem HiZ-Algorithmus und dem MOC - bei voller Auflösung (1920x1080 Pixel) und Single-Core Performanz - geht hervor, dass der MOC-Algorithmus 2% weniger Dreiecke als der HiZ verwirft, aber dennoch eine bessere Performanz erzielt [4]. 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 MOC-Algorithmus zeitweise bis zu 10x schneller als die des HiZ-Algorithmus [4]. Die Skalierbarkeit des MOC ist der des HiZ deutlich überlegen. Mit steigender Größe der Dreiecke wächst auch der Performanzunterschied zwischen dem MOC und dem HiZ. Im Allgemeinen ist der MOC-Algorithmus durchschnittlich 3x schneller als bisherige Algorithmen und bei einem nur geringeren Memory Overhead können 98% aller Dreiecke verworfen werden [4].

## 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 potenziell von Occludern verdeckt werden (das heißt, sie beinhalten ebenfalls alle Occluder), siehe Abb. 4.

Sowohl Occluder als auch Occludees liegen dabei in zwei Formen vor. Einmal als Netz, bestehend aus Punkten, 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 (Frustum Culling) und dementsprechend für die folgende Rasterisierung beim Occlusion Culling in Frage kommt.

Occlusion Culling besteht im Wesentlichen aus zwei Schritten. Die Occluder werden als erstes in einem Renderingdurchlauf rasterisiert, jedoch ohne die Objekte tatsächlich zu zeichnen, und der Tiefenpuffer wird entsprechend der Occludermenge gefüllt. Schritt zwei besteht darin, Occlusion Queries anzustoßen. Bei den Occlusion Queries werden

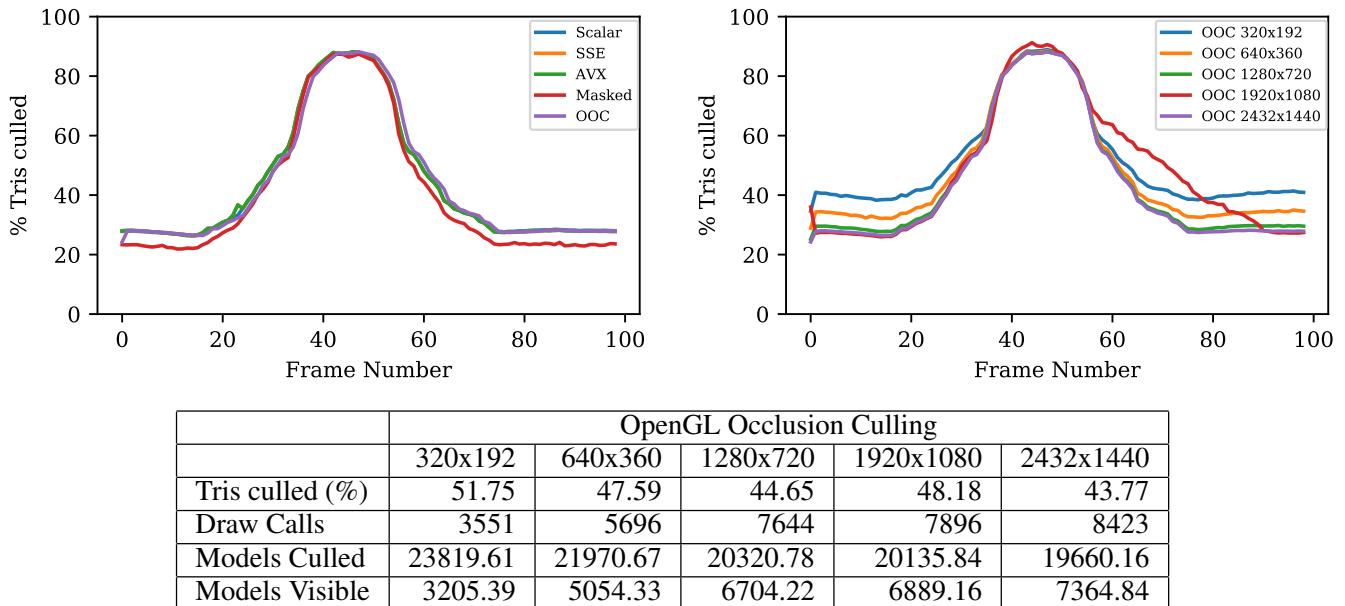


Abb. 6. Im linken Diagramm ist die Prozentangabe der verworfenen Dreiecke mit den fünf verschiedenen Methoden des SOC-Frameworks bei einer Auflösung des Tiefenpuffers von 1920x1080 zu sehen. Im rechten Diagramm wird die Anzahl der verworfenen Dreiecke der OOC-Methode mit fünf verschiedenen Auflösungen des Tiefenpuffers verglichen. Interessant zu beobachten ist der Verlauf für die Auflösung 1920x1080, da er im abfallenden Bereich ungefähr linear und nicht annähernd wie  $\frac{1}{x}$  verläuft. Tabelle: Die gezeigten Werte sind jeweils Mittelwerte über die gesamte Kamerafahrt. Die Anzahl der Draw Calls ist bei 1920x1080 ebenfalls etwas außerhalb des Musters, vergleiche zum Beispiel mit 640x360.

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 Frustum Culling 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 entsprechend Abb. 5 aus.

### 3.1 Frustum Culling

Frustum Culling 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 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 200%.

Anmerkung zum Frustum Culling: Während die Performance bei Einsatz von Frustum Culling 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.

### 3.2 Tiefenpuffer-Rasterisierung

Alle Occluder, die nach dem Frustum Culling als sichtbar markiert sind, werden in diesem Schritt verwendet, um einen Tiefenpuffer zu generieren. Dazu werden alle sichtbaren Objekte als eine Occluder-menge „gerendert“. Es wird lediglich ein Tiefenpuffer gefüllt, ohne die Objekte tatsächlich zu zeichnen, das heißt, der Farbpuffer wird nicht befüllt. Das Rendering rasterisiert die Occluder, das heißt, die Occluder werden in den Bildraum (screen space) transformiert und befüllt anschließend den Tiefenpuffer, indem an den Stellen auf dem

Bildschirm, an denen sich das Objekt befindet, die Tiefenwerte des Objekts gespeichert werden, siehe Abb. 7.



Abb. 7. 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. Der schwarze Balken unterhalb des Tiefenpuffers entsteht durch die Differenz zwischen Fensterauflösung 1920x1200 und Tiefenpuffer-Auflösung 1920x1080.

### 3.3 Occlusion Queries

An dieser Stelle sei erwähnt, dass die gesamte Occludeemenge zu Beginn des Programms einmal auf die GPU hochgeladen wurde 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 wird 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 die Tiefenwerte der rasterisierten Fragmente des Occludees kleiner als die des Tiefenpuffers sind. Die Occlusion Query testet also, ob mindestens ein Fragment des Occludees bei einem potenziellen Renderdurchlauf sichtbar wäre. In diesem Fall liefert die Occlusion Query „true“ zurück, andernfalls „false“. Es existiert auch die Möglichkeit, die Occlusion Query die genaue Anzahl an sichtbaren Fragmenten zurückzugeben zu lassen und ein Occludee erst dann als sichtbar zu klassifizieren, falls die Anzahl an sichtbaren Fragmenten einen gewissen Schwellwert überschreitet. Allerdings

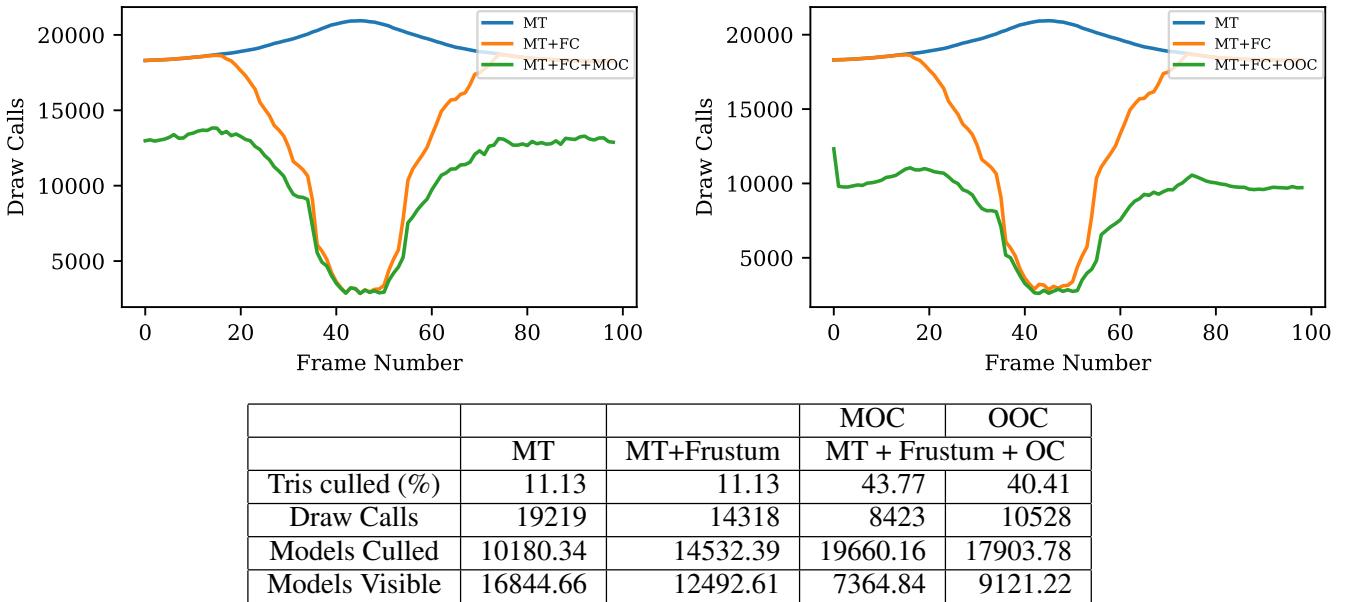


Abb. 8. Vergleich Anzahl Draw Calls von Multi-Threading (MT), MT + Frustum Culling (FC) und MT + FC + Occlusion Culling (OC). Anzahl Draw Calls geht mit Verlauf der Kamerafahrt mit FC und OC deutlich runter. Ist OC aktiviert, sind die Maximalwerte am Anfang und Schluss deutlich niedriger und durch die Konservativität sind die Werte seitens des MOC höher als bei OOC.



Abb. 9. 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.

wurde sich gegen eine solche Implementierung entschieden, da die Wartezeiten der Occlusion Queries viel zu hoch sind. 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

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 48.700 Dreiecken und 27.025 Occludees mit  $\approx 1.923.000$  Dreiecken [1]. Sofern nicht anders angegeben, wurden sämtliche Ergebnisse mit einer Kamerafahrt, wie sie in Abb. 9 angedeutet ist, über 100 Frames erstellt.

Der Tiefenpuffer ist standardmäßig auf eine Auflösung von 1920x1080 Pixel gesetzt und außerdem sind Multithreading, Frustum Culling und Occlusion Culling eingeschaltet. 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 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 i7-4770, einer NVIDIA Geforce GTX 960 und 16 GB RAM verwendet.

Zuerst wurde die OOC-Methode auf den qualitativen Aspekt untersucht, das heißt, ist das Ergebnis korrekt und wie „gut“ ist das Ergebnis im Hinblick auf die Anzahl der verworfenen Dreiecke. Abb.

6 zeigt die erzielten Ergebnisse. Im linken Bild ist ein Vergleich aller im SOC-Framework implementierten Occlusion Culling Techniken zu finden. Es ist gut zu sehen, dass die OOC-Methode qualitativ keinerlei Abstriche bei dieser Eigenschaft gegenüber den anderen Techniken machen muss. Lediglich die etwas schlechteren Werte der MOC-Methode stechen heraus, aber das ist nicht verwunderlich, da dies bereits in [4] erläutert wurde. In einem weiteren Test, der nur die OOC-Methode betrifft, wurde untersucht, inwiefern eine geringere Auflösung des Tiefenpuffers Auswirkungen auf die Anzahl der verworfenen Dreiecke hat und somit indirekt ebenfalls Auswirkungen auf die Anzahl der Objekte, die verworfen werden und schließlich auf die Anzahl der Draw Calls, die einen wichtigen Teil der gesamten Performanz ausmachen. Die Ergebnisse sind größtenteils genau so zu erwarten. Je geringer die Auflösung des Tiefenpuffers wird, desto größer und ungenauer wird das Ergebnis, sprich mit fallender Auflösung steigt die Anzahl der Dreiecke, die verworfen werden. Interessant ist das Ergebnis für die Auflösung 1920x1080. Zu Anfang ist der Verlauf ähnlich zu allen anderen, aber etwa ab Frame 60 verringert sich die Anzahl an verworfenen Dreiecken ungefähr linear und nicht wie bei allen anderen ungefähr gemäß  $\frac{1}{x}$ . Ebenfalls wie erwartet ist das Ergebnis für die Anzahl an Objekten, die verworfen werden und die Draw Calls (siehe Abb. 6 Tabelle). Steigt die Anzahl der verworfenen Objekte, sinkt entsprechend die Anzahl der Draw Calls (und darüber hinaus verbessert sich entsprechend die Performanz, später mehr). Lediglich die Anzahl an Draw Calls für 1920x1080 entrinnt dem Muster, da es erwartungsgemäß weniger Draw Calls geben müsste für die Anzahl an verworfenen Dreiecken.

Als nächstes wurden die Auswirkungen verschiedener Culling-Kombinationen auf die Anzahl der verworfenen Dreiecke und vor allem auf die Draw Calls untersucht, siehe Abb. 8. Läuft die OOC-Methode ohne Frustum Culling und ohne das Ausführen der Occlusion Queries, ist die Anzahl an Draw Calls entsprechend hoch, da die Anzahl an sichtbaren Objekten durch die fehlenden Culling-Mechanismen sehr hoch ist, wobei sichtbar im Sinne von „wird gerendert, egal, ob das Objekt tatsächlich in der Szene sichtbar ist“ gemeint ist. Interessant ist der Maximalwert in der Mitte der Kamerafahrt. Wohingegen bei den anderen Kombinationen an dieser Stelle der niedrigste Wert erreicht wird, ist er dort ohne Frustum Culling und ohne Occlusion Queries am höchsten. Besser wird das Ergebnis, wenn zusätzlich zum

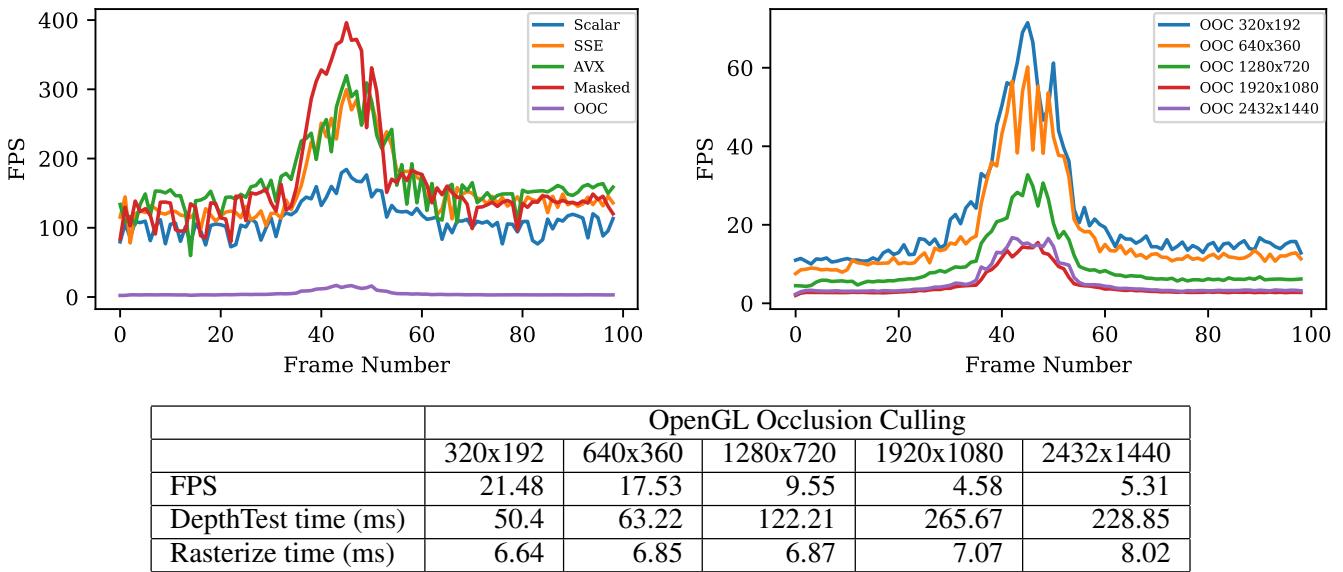


Abb. 10. Links: FPS-Vergleich aller SOC-Methoden im Intel-Framework bei Full-HD Auflösung, aktiviertem MT, FC und TTC. OOC schneidet mit nur einem Bruchteil der FPS von MOC oder SSE deutlich am schlechtesten ab. Rechts: Kamerafahrt mit unterschiedlichen Tiefenpuffergrößen. Mit niedriger werdenden Auflösungen sinken die Tiefentestzeiten ab (siehe Tabelle) und die FPS steigen erwartungsgemäß an, mit einer Ausnahme bei der 2432x1440 Auflösung, die trotz ihrer erhöhten Puffergröße, etwas besser abschneidet als die Full-HD Auflösung.

Multithreading das Frustum Culling aktiviert wird. Je näher die Kamera sich der Burg nähert, desto höher ist die Anzahl verworfener Objekte und die Menge an Draw Calls fällt entsprechend auf ihren Tiefstwert und steigt anschließend wieder mit zunehmender Distanz der Kamera zur Burg. Sind sowohl Frustum Culling als auch Occlusion Queries aktiviert, ist der Verlauf im Prinzip der Gleiche, mit dem Unterschied, dass die Menge an Draw Calls zu Anfang und zum Schluss deutlich niedriger ist. Sind es ohne Occlusion Queries zu Anfang rund 17000 Objekte, die gerendert werden müssen, so sind es mit Occlusion Queries in etwa 6000-7000 weniger, womit Occlusion Queries einen enormen Renderingaufwand einsparen. An dieser Stelle wird ein Unterschied zu Intels MOC deutlich. Zwar sind die Verläufe für die Kombinationen „MT“ und „MT + FC“ bei beiden gleich, so kommt bei der Einstellung „MT + FC + OC“ die Konservativität des MOC ins Spiel und zwar in Form von mehr Draw Calls, verursacht durch weniger verworfene Objekte. Dadurch, dass die verschiedenen Methoden nur für den Tiefentest benutzt werden, sind die Werte ohne den Tiefentest unabhängig von der Methode und erst durch das Aktivieren der Tiefentests lassen sich Unterschiede feststellen.

Als letztes wurde der wohl wichtigste Aspekt untersucht, die Performanz im Vergleich mit anderen SOC-Methoden im Intel-Framework. Trotz der guten Werte, die die OOC-Methode in den Tests davor erzielt, schneidet sie im Leistungsstest sehr schlecht ab. Getestet wurden alle Methoden mit aktiviertem Multithreading, Frustum Culling und Tiefentest Culling. Abb. 10 zeigt den Leistungsvergleich mit den anderen SOC-Methoden. Es ist gut zu sehen, dass die FPS bei allen Methoden im Bereich circa zwischen Frame 35 und 55 stark angestiegen sind. In diesem Bereich werden mehr Objekte verworfen und die Anzahl an Draw Calls verringert sich, was wiederum mehr FPS zur Folge hat. Ebenfalls deutlich zu erkennen ist, dass die OOC-Methode viel weniger FPS während der gesamten Fahrt als alle anderen hat. Wohingegen sich der Rest zu Beginn und gegen Ende um die  $120 \pm 20$  FPS befindet, ist die OOC-Methode bei circa 3 FPS anzutreffen. Ein ähnliches Bild ergibt die Analyse der Spitzenwerte. Der Maximalwert bei OOC ist bei knapp 16 FPS, während sich die anderen Methoden bei circa 280-380 FPS aufzuhalten (lediglich die Scalar-Methode befindet sich nicht in diesem Spitzenbereich). Diese schlechten FPS-Werte sind nicht verwunderlich, wenn die Zeiten für das Rasterisieren und die Tiefentests in der Tabelle aus Abb. 10 betrachtet werden. So hat die OOC-Methode bei einer Auflösung von

1920x1080 durchschnittliche Werte von 266ms für Tiefentests und 7ms Rasterisierungszeit. Zum Vergleich, die Werte beim MOC liegen bei ungefähr 0.1ms für die Rasterisierung und 0.39ms für Tiefentests.

Es wurde auch ein Test durchgeführt, der das Leistungsverhalten unterschiedlicher Tiefenpuffer-Auflösungen untersucht. Im rechten Bild von Abb. 10 sind die verschiedenen Verläufe zu sehen. Je geringer die Auflösung wird, desto höher werden die Bildraten, die teilweise fast viermal so hoch sind, wenn zum Beispiel die Mitte der Kamerafahrt betrachtet wird. So kommt die Full-HD Auflösung 1920x1080, im Gegensatz zu den knapp über 21 FPS im Mittel bei der sehr niedrigen Auflösung von 320x192, auf gerade einmal durchschnittlich knapp über 4 FPS. Der Preis für mehr FPS durch eine niedrigere Auflösung ist allerdings ein durchaus schlechteres Ergebnis. Bei geringer Auflösung (z. B. 640x360) ist deutlich zu erkennen, dass einige Objekte fehlen, die bei höheren Auflösungen sichtbar sind. Eine Ausnahme findet sich bei der 2432x1440 Auflösung. Sie schneidet, entgegen dem Trend der niedriger werdenden FPS bei steigender Auflösung, etwas besser ab als die Full-HD Auflösung.

In einem letzten Versuch wird die OOC-Implementierung mit einer äquivalenten Implementierung auf der GPU verglichen. Abb. 11 zeigt den FPS-Verlauf der Kamerafahrt. Die Implementierung auf der GPU liefert durchschnittlich 43 FPS und ist damit in etwa 8x schneller als die OOC-Methode, die im Schnitt nur auf 5.4 FPS kommt.

**Anmerkung** An dieser Stelle ist es auch im Rückblick auf Abb. 8 interessant, anzumerken, dass die Performanz der OOC-Methode am besten ist, wenn sowohl Frustum Culling als auch Tiefentest Culling deaktiviert sind. In der Standard-Kameraeinstellung werden durchschnittlich 91 FPS erreicht. Wird nur Frustum Culling aktiviert, werden im Schnitt 84 FPS erreicht. Je nach Kameraposition erreicht Frustum Culling jedoch stellenweise auch höhere Frameraten. Wird zusätzlich noch das Tiefentest Culling aktiviert, fällt der Wert auf 5 FPS.

**Probleme** Die größten Performanzprobleme werden durch die Occlusion Queries verursacht. Das Programm wurde mit einem Performance Profiler von Visual Studio Enterprise 2017 analysiert, um herauszufinden, an welchen Stellen das Programm viel Zeit verbraucht

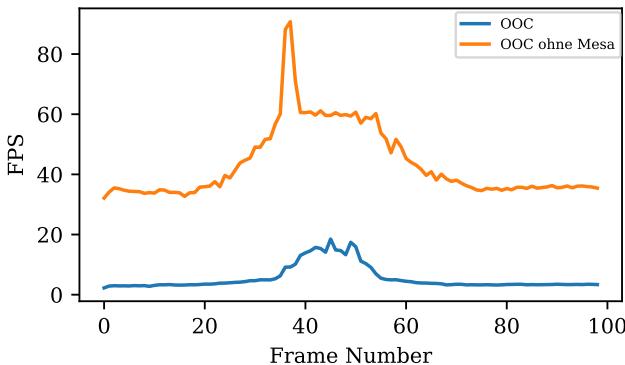


Abb. 11. FPS-Vergleich der OOC-Methode mit einer äquivalenten GPU Implementierung, die mit durchschnittlich 43 FPS gegenüber den 5.4 FPS der OOC-Methode etwa 8x so schnell ist.

und die meiste Zeit hält sich das Programm bei den Occlusion Queries auf. Das ist teilweise auch so zu erwarten, da die Occlusion Queries ein essentieller Teil des Programms sind. Dennoch wird dort zu viel Zeit verbracht, vor allem hinsichtlich der Tatsache, dass der Prozessor nicht mehr als 50% ausgelastet ist und damit nicht seine gesamte Kapazität in Anspruch nimmt. Ein Grund für den großen Zeitverlust wurde bei langen Wartezeiten gesehen, die entstanden sind, nachdem alle OQ gestartet wurden und das Programm auf die Ergebnisse warten musste. Deswegen wurde die Implementierung dahingehend geändert, dass ein zweites Set an Occlusion Queries generiert wurde, das die Ergebnisse der OQs des vorherigen Frames enthält. Durch das zweite Set kann die Waiteroutine weggelassen werden, unter der Annahme, dass im aktuellen Frame alle Ergebnisse des vorherigen Frames vorliegen. Dadurch konnte das Problem mit langen Wartezeiten zwar gelöst werden, allerdings wurden dadurch keine signifikanten Performanceverbesserungen erzielt.

## 5 FAZIT

In dieser Arbeit wurde gezeigt, dass es mit Hilfe von Mesa 3D möglich ist, eine Occlusion Culling Methode zu entwickeln, die parallel zum GPU-Rendering auf dem Prozessor läuft. Die Auswertung hat gezeigt, dass die neu entwickelte OOC-Methode qualitativ, im Sinne von verworfenen Dreiecken beziehungsweise Objekten, mit den bereits vorhandenen SOC-Methoden konkurriert kann. Unter dem Aspekt der Performance kann die OOC-Methode allerdings nicht mit den anderen mithalten.

**Future Work** Ein Thema mit dem sich künftige Arbeiten beschäftigen können, ist eine Überarbeitung der Implementierung für geringere Tiefenpuffer Auflösungen. Bei der derzeitigen Implementierung werden eher zu viele Objekte verworfen als zu viele gerendert werden. Das Gegenteil sollte allerdings der Fall sein, da so gegebenenfalls die Performance, jedoch nicht die Qualität des Ergebnisses vermindert wird. Ähnlich wie es bereits beim MOC angewandt wird.

Zukünftige Arbeiten können sich außerdem damit beschäftigen, Occludees zu Paketen zu schnüren und lediglich eine Occlusion Query mit dem Paket zu starten, anstatt wie bisher für jedes Objekt eine separate Query durchzuführen. Ziel ist es, weniger Occlusion Queries zu generieren, ohne dabei zu viel an Qualität einzubüßen zu müssen und dadurch an Performance zu gewinnen. Die Auswahl der Occludees, 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 Occludees angezeigt, die sichtbar sind, werden zu viele angezeigt oder fehlen Objekte in der Szene). Die Auswahl der Objekte sollte dabei so vonstattengehen, dass nur Occludees zu einem Paket zusammengefasst werden, die sich in einer bestimmten (kleinen) Region befinden,

so dass Pakete vermieden werden, bei denen die Objekte über große Teile der Szene verteilt sind. Wird die Auswahl nicht wie oben beschrieben getroffen, sondern zum Beispiel zufällig, ist es wahrscheinlich, dass im Schnitt zu viele Objekte gerendert werden. Denn liegt nur ein Objekt bei einer Paketgröße von beispielsweise 5 Objekten nicht bei den anderen 4 Objekten und nur dieses Objekt ist in diesem Paket sichtbar, dann werden alle 5 Objekte als sichtbar markiert. Lägen alle 5 Objekte dicht beieinander, ist die Wahrscheinlichkeit geringer, dass dieser Fall eintritt. Mit steigender Größe der Pakete, steigt auch die Gefahr falscher Ergebnisse und infolgedessen auch der Renderingaufwand. Mit einem optimierten Auswahlverfahren können auf die Weise wahrscheinlich deutliche Verbesserungen erzielt werden, sowohl im Performanzbereich als auch bei der Qualität des Ergebnisses.

## 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] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.
- [3] 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.
- [4] 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.