



Sicherstellung der Effektivität von CHC++ in Vulkan

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Jakob Pernsteiner

Matrikelnummer 01627767

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Mitwirkung: Dipl.-Ing. Bernhard Kerbl, BSc

Wien, 19. Oktober 2020

Jakob Pernsteiner

Michael Wimmer



Ensuring the Effectiveness of CHC++ in Vulkan

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Jakob Pernsteiner

Registration Number 01627767

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Assistance: Dipl.-Ing. Bernhard Kerbl, BSc

Vienna, 19th October, 2020

Jakob Pernsteiner

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Jakob Pernsteiner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Oktober 2020

Jakob Pernsteiner

Danksagung

Zuerst möchte ich mich bei meinem Betreuer Bernhard Kerbl und verantwortlichen Professor Michael Wimmer für das zahlreiche und hilfreiche Feedback während der Entwicklung dieser Bachelorarbeit bedanken. Durch sie war es mir überhaupt erst möglich mich ausführlich mit dieser interessanten Thematik auseinanderzusetzen.

Vielen Dank auch an meine Eltern für ihre umfangreiche Unterstützung, damit ich auch in Zeiten der Coronakrise nicht meine Motivation verlor.

Zum Schluss auch noch danke an all meine Freunde, die mir immer zugehört haben und mich mit unseren Diskussionen auf neue Ideen brachten.

Acknowledgements

First, I want to thank my supervisor Bernhard Kerbl and responsible professor Michael Wimmer for their plenty and helpful feedback during the development of this thesis. Only because of them, it was possible for me to have a detailed look into this interesting subject.

Also, many thanks to my parents for their extensive support, so that I didn't lose my motivation in this time of the corona-crisis.

At last, I want to thank all my friends, who always listened to me and brought on new ideas through our discussions.

Kurzfassung

Echtzeit-Verdeckungserkennung ist ein wertvolles Werkzeug zur Erhöhung der Performance von Echtzeit-Rendering Anwendungen durch das Erkennen und Entfernen von unsichtbarer Geometrie aus der Rendering Pipeline. Durch neue Rendering Application Programming Interfaces (APIs) wie Vulkan und moderne Hardware können solche Verdeckungserkennungs-Algorithmen noch leistungsstärker werden. Diese Bachelorarbeit versucht die Leistungsfähigkeit von Coherent Hierarchical Culling Revisited (CHC++) unter dieser neuen Umgebung durch verschiedene Optimierungen des Algorithmus sicherzustellen und zu evaluieren. Die Änderungen beinhalten das Zusammenlegen von aufeinanderfolgenden draw-calls und Occlusion Queries in einen einzigen GPU-Queue submit, um den Mehraufwand auf der CPU und GPU zu reduzieren. Zusätzlich wurde die Unterstützung von alpha-überblendeten transparenten Objekten zum Algorithmus hinzugefügt, was die korrekte Verdeckungserkennung und das Darstellen dieser Objekte erlaubt. Der Algorithmus funktioniert gut in Umgebungen mit viel Verdeckung und seine Leistungsfähigkeit bleibt stabil in einem Schlechtesten-Fall Szenario. Aber der Leistungsanstieg der originalen Implementierung konnte nicht repliziert werden, was sich aber auf die Unterschiede in den Rendering APIs und Verbesserung der Hardware zurückführen lässt.

Abstract

Real-time occlusion culling is a valuable tool to increase the performance of real-time rendering applications by detecting and removing invisible geometry from the rendering pipeline. Through new rendering Application Programming Interfaces (APIs) like Vulkan and modern hardware, these culling algorithms can become even more powerful. This thesis tries to ensure and evaluate the performance of Coherent Hierarchical Culling Revisited (CHC++) in this new environment by performing various optimisations to the algorithm. The changes include the batching of consecutive draw-calls and occlusion queries into single GPU-queue submits to reduce the overhead on the CPU and GPU. Additionally, the support for alpha-blended transparent objects was added to the algorithm, which allows for correct culling and rendering of these objects. The algorithm performs great in environments with high occlusion and does not degrade in performance in the worst case scenario. But the high performance increase of the original implementation could not be replicated, which is attributed to the difference in rendering APIs and hardware improvements.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Motivation	2
1.2 Approach	3
2 Related work	5
2.1 Early approaches	5
2.2 Occlusion queries	7
2.3 Hierarchical-Z culling	10
2.4 Culling methods in high-end game engines	13
3 Problems	19
3.1 Vulkan vs. OpenGL	19
3.2 Transparency	22
4 Implementation	23
4.1 Handling command buffers	23
4.2 Reducing queue submits	24
4.3 Handling transparency	27
5 Results and Discussion	29
5.1 Results	29
5.2 Discussion	33
6 Conclusion	39
6.1 Summary	39
6.2 Outlook	39
List of Figures	43

List of Algorithms	45
Acronyms	47
Bibliography	49

Introduction

Performance is critical in real-time rendering applications, therefore a lot of optimizations need to be applied to the rendering pipeline to keep up with the ever-growing demand in visual fidelity. As scenes get bigger and bigger and geometry density increases, more performance is needed. One of the biggest problems is that in a naive rendering pipeline, where every 3D model is rendered every frame, a lot of the rendering power is lost to geometry that is not visible in the final render. Depending on the viewpoint, a lot of the geometry is outside of the view frustum or is completely occluded by other geometry and therefore invisible. Therefore, a lot of performance can be gained in most situations by culling these invisible objects from the rendering pipeline and only render visible geometry. Culling geometry outside the view-frustum, called View-Frustum Culling (VFC), has been studied extensively and efficient algorithms have been established. On the other hand, culling occluded geometry, simply called occlusion culling, has proven to be harder to accomplish as it is more difficult to detect the visibility of objects than to check if something is outside a given volume. Figure 1.1a and figure 1.1b compare which objects are culled when using VFC. In addition, figure 1.1c depicts how many objects can be culled if occlusion culling is used. There are many different approaches for real-time occlusion culling, most of which are designed with older, more synchronous graphics APIs like OpenGL in mind. This thesis presents an implementation of one of these algorithms called Coherent Hierarchical Culling Revisited (CHC++) [MBW08] with the rather new asynchronous graphics API Vulkan, to show how this algorithm performs with today's hardware and API.

The following section details why this thesis tries to adapt CHC++ to work efficiently with Vulkan. The chapters afterwards show what alternative occlusion culling methods are available, what problems were encountered while adapting CHC++, how these were solved and what results were gathered while evaluating.

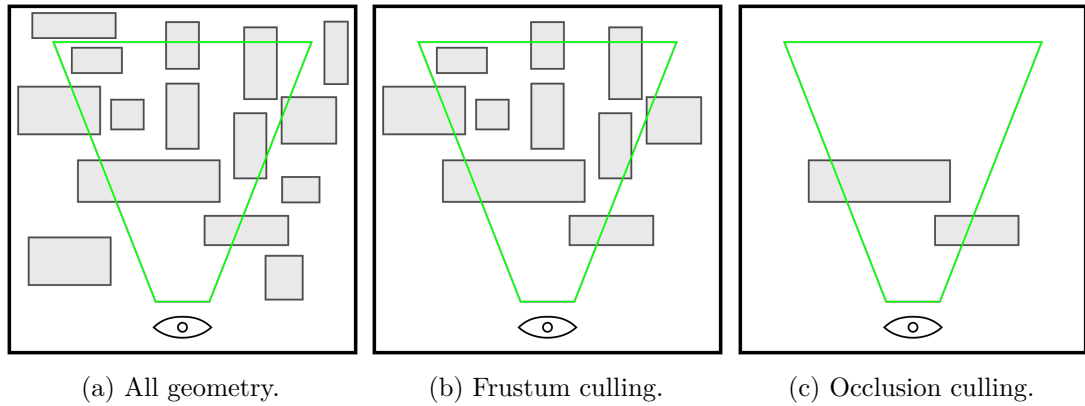


Figure 1.1: A scene of multiple objects and a viewing frustum in different rendering modes. 1.1a shows all the geometry that could be rendered. 1.1b shows what is rendered with frustum culling enabled - only the objects inside the frustum are rendered. 1.1c shows what objects are rendered with occlusion culling enabled - only visible objects are rendered.

1.1 Motivation

In real-time rendering applications it is essential to maintain frames per second (FPS) upwards of 60 FPS and with newer monitors up to 144 and 240 FPS or even more. To reach these numbers, powerful hardware is needed, but also the software needs to be optimised to be able to output the high-fidelity graphics of modern games and visualisations at this rate. Therefore, no computational capacity should be wasted on things that do not contribute to the final render to get the best performance out of the application. In other terms, performance should not depend on overall scene complexity, but on the complexity of the part of a scene that is actually seen by the virtual camera.

To do so, sophisticated culling mechanisms need to be implemented to reduce the overall load on the GPU and CPU. As already mentioned, frustum culling can be achieved rather easily because it can be done analytically, for example by determining if the bounding box of a 3D object is completely outside one of the planes of the viewing frustum. But detecting occlusion—or rather visibility of geometry—is more difficult because complex visibility interactions cannot be calculated analytically. Occluder fusion—i.e., multiple objects combined occlude another object—needs to be taken into account, which can be hard to achieve. So other techniques have to be used like occlusion queries or Hierarchical-Z (Hi-Z) culling, which are detailed in later chapters. Although occlusion detection proves to be harder to do, a lot of research has been done on the topic and many approaches have been published by research institutions and game engine developers.

Also, the adoption of newer asynchronous graphics APIs like Vulkan and DirectX 12 provide more possibilities to improve these visibility detection methods. The biggest performance improvement these new APIs provide is fewer abstractions and command

queues. Lower abstractions means that they give a lot more power to the developer of the graphics application while reducing the amount of work that needs to be done by the graphics driver. Command queues are used to record multiple API commands into a buffer before actually sending them to the GPU. This command buffer is later submitted to be executed asynchronously by the GPU in a single API call to the graphics driver. This enables the batching of multiple draw calls that are then submitted and executed on the GPU independent of the CPU to increase parallelism and to reduce the number of API calls.

This thesis tries to adapt and optimize the CHC++ algorithm to see how well it performs with Vulkan on modern hardware compared to the original implementation provided in conjunction with [BMW09]. CHC++ was originally developed for OpenGL before the release of OpenGL version 3.0, so a lot of improvements to the API happened since then. Also because hardware has improved significantly over the past years, we hoped to see big performance gains by adapting CHC++ into a modern setting. The adapted algorithm still performs great, but the major performance increase of the original implementation could not be replicated in Vulkan. How this was achieved is detailed in the following chapters.

1.2 Approach

At first, a naive re-implementation of the original algorithm presented in [MBW08] using excessive synchronisation to present a more synchronous workflow and to be more true to the original OpenGL implementation was made. This naive implementation was evaluated to find bottlenecks and problems that arise when running CHC++ on Vulkan. The bulk of the problems, detailed in chapter 3, were the excessive synchronisation and the many graphics-queue submits that caused too much overhead. Then the optimised version of the algorithm was implemented by finding proper solutions, which are detailed in chapter 4, to the detected problems. This implementation was further profiled to find additional bottlenecks and adapted to solve these issues.

Results were evaluated on multiple test scenes with high geometric density, some with high potential for occlusion culling and some with less to compare how the implementation performs in different scenarios, which is presented in chapter 5. To get comparable results, walkthroughs of the scenes were recorded and rerun with different rendering modes enabled. Statistics were gathered and plotted to evaluate the results. Additionally, profiling tools like Nvidia Nsight were used for evaluation.

Related work

This chapter presents an overview of different occlusion culling methods. This includes early approaches like cell-based and portal culling techniques. Then it gives an introduction to occlusion queries and how they are used in Coherent Hierarchical Culling (CHC) and CHC++. Later sections show how Hi-Z culling can be used in different ways to detect visibility and what culling mechanism some selected game engine developers use in their graphics pipelines. At the end, it also presents the commercially available Umbra 3 occlusion culling middleware that can be integrated into nearly any rendering pipeline without the cost of implementing your own solution.

2.1 Early approaches

In general, there are two types of occlusion culling called point and area culling. The difference is, that point-based culling calculates the occlusion from a fixed viewpoint, whereas area based culling calculates it for a defined area or volume like regular cells. In comparison, occlusion has to be calculated every frame for point culling because the viewpoint can change every frame. But it is less expensive to calculate as only this one viewpoint needs to be taken into account. A lot of real-time occlusion culling methods use point-based culling, since it can be done by using occlusion queries (section 2.2) or Hi-Z culling (section 2.3) for example, but there are many more options available.

Area culling, on the other hand, is harder to compute as the occlusion needs to be calculated for the entire volume of each cell. When determining the visibility from one such cell, other invisible cells need to be invisible from every point inside the current cell to stay conservative. The other way round, every visible cell needs to be visible from at least one point inside of the current cell's volume. Because finding every other cell that is visible from a tested cell is really expensive, this calculation is mostly done in a pre-process and not at runtime. For every tested cell, all the cells that are visible from inside its volume are saved in a list, which allows for fast lookup at runtime. At runtime,

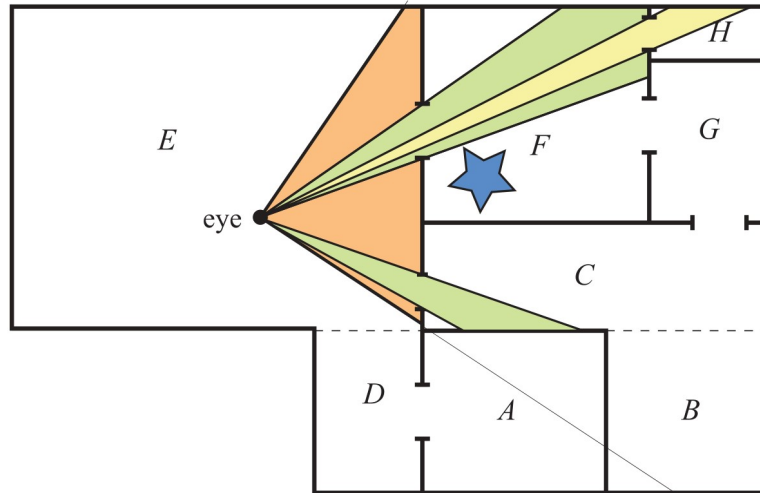


Figure 2.1: Portal culling is depicted. There are multiple cells, denoted by the letters A-H, that are connected by portals. The eye represents the viewpoint, visualising the multiple frustums resulting from the recursive portal culling algorithm. For example, the star is completely occluded, as it is outside of cell F’s frustum. Reprinted from the portal culling chapter in [AM18].

it is checked which cell contains the current viewpoint. Then this cell’s list of visible cells is frustum culled and the geometry of the remaining visible cells is rendered, which makes the occlusion culling fast as only the list needs to be looked up. On the other hand, this will not cull every object that is invisible for the given viewpoint, because the visibility is calculated for the whole cell and not only for the viewpoint. Therefore, everything in a visible cell needs to be rendered, even if some of the geometry is occluded inside the cell.

Another early approach is portal culling, which can mainly be used in indoor scenes or when areas are cut off by huge obstacles but are still connected by corridors, like mountains and tunnels. It works by defining areas—like rooms—and portals—like doors, windows or corridors—and then creating an adjacency graph where neighbouring areas are connected by the portals. At runtime, the view frustum is used to determine which adjacent cells are visible from the viewpoint by checking which portals are inside the frustum. Then the frustum is shrunk to fit into the connecting portals. This is done recursively, starting with the area that contains the viewpoint to all the areas that are connected by the visible portals. An example can be seen in figure 2.1. Now there are different frustums for the visible cells which are then used for frustum culling in the respective cells. This results in a rather simple and fast form of occlusion culling, but can only be used if portals can be placed in the scene. Therefore, this approach is not really suitable for large open-world scenarios, as portals cannot be placed in a meaningful manner. Also, occlusion inside of a cell cannot be determined with this approach as occlusion is only detected between different cells through the portals. But this still

provides a meaningful reduction of rendered geometry in a lot of cases [AM18].

2.2 Occlusion queries

Occlusion queries provide an alternate rendering mode in which the newly rendered geometry is tested against the depth buffer to report the number of visible fragments or to only report if a fragment is visible at all. This is supported in both software and hardware nowadays. Therefore, occluder geometry can be rendered to fill the depth buffer before testing other occludees against the occluder depth. In this case, the occluder and occludee geometry should be simple, so it can be rasterized quickly. Additionally, the testing geometry must be a conservative boundary approximation of the actual mesh. The results of the queries can then be used to determine which tested meshes are occluded and which are not. Figure 2.2 shows the result of an occlusion query where the axis-aligned bounding box (AABB) of a chair is tested against the depth buffer. This chair is visible because much of its bounding box is visible (green). If an object turns out to be occluded, we saved the time of pushing a potentially complex object through the rendering pipeline. If it is not occluded we wasted additional time on the query itself. Although queries are asynchronous—as in the CPU can perform work while the query is processing on the GPU—the latency of when the result is available can be rather long. Therefore, it is desirable to keep the number of queries to a minimum and test multiple objects that are likely occluded together in a single query. Else the time the CPU has to stall till the results are ready gets too long, causing poor performance [AM18].

OpenGL also provides conditional rendering with occlusion queries. This means, that rendering operations can be tied to an occlusion query, where the draw-calls are only executed if the query return that the geometry is visible. Ergo an object is automatically rendered if its occludee geometry is visible. This removes the problem of having to wait for occlusion queries on the CPU, but it could still cause stalls in the graphics pipeline itself. Also, implementations or performance comparisons of this approach could not be found, so it is hard to say how this compares to other approaches.

The following subsections describe different occlusion culling methods utilising occlusion queries. These methods try to interleave rendering with occlusion testing to solve the high latency problem of the test results.

2.2.1 Coherent Hierarchical Culling (CHC)

Bittner et al. proposed a new method called Coherent Hierarchical Culling (CHC) in 2004 which tries to reduce the number of occlusion queries and also eliminate CPU stalls and GPU starvation by interleaving rendering and queries. The algorithm uses a kd-tree for breadth-first front-to-back traversal of the scene geometry. Only leaf nodes and interior nodes that were invisible in previous frames are occlusion queried, visible interior nodes do not need to be queried. Also, previously visible leaf nodes are assumed to stay visible and are therefore rendered immediately and also queried. All the queries are put into a



Figure 2.2: A visualisation of an occlusion query. The green part shows which pixels of the chairs bounding box got through the depth test resulting in the occlusion query to return that the object is visible. The red parts show how many pixels of the bounding box are occluded by other geometry. Would every pixel be red, the object would be fully occluded and the query would return that the object is invisible and therefore does not need to be rendered. The figure was created using RenderDoc.

queue so they can be checked for completion repeatedly. While traversing a tree node, it is checked if the first query already completed. If it did not yet complete, traversal is continued as already described. If it has completed and the checked node is invisible, the whole subtree can be culled. If a node turns out to be visible, its children need to be queried too.

This results in an interleaving of rendering and occlusion queries, where the CPU can do further draw-calls or start occlusion queries while the GPU performs previously issued rendering operations and occlusion queries. This minimises the amount of CPU stalls while also keeping the GPU busy. Also, it reduces the amount of queries needed, as only leaf nodes and previously invisible interior nodes need to be queried. In addition, if an interior node is invisible, the whole subtree can be skipped, further reducing the number of queries.

As a result, this approach improves performance over standard VFC in most cases, but still performs slower in cases where hardly any occlusion is observed and the additional overhead of the queries makes it slower. Furthermore, temporal and spatial coherence can be leveraged even more effectively, which will be explained in the following section [BWPP04].

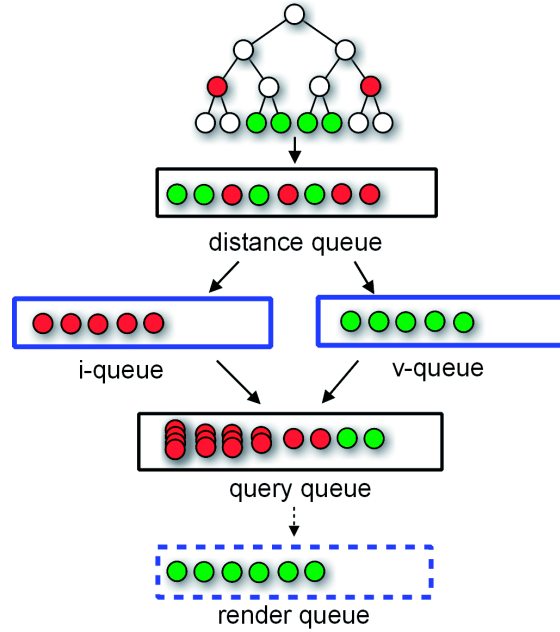


Figure 2.3: The different queues of CHC++ and how they work together. Reprinted from [MBW08]

2.2.2 Coherent Hierarchical Culling revisited (CHC++)

Mattausch and Bittner et al. further developed CHC++, an updated version of CHC. It optimises CHC by further leveraging temporal and spatial coherence, which is done by adaptively predicting the visibility of objects and batching queries depending on how likely some objects will be occluded together. This, in turn, reduces the number of queries and also the number of API calls and state changes, thus speeding up the algorithm greatly in comparison to CHC and others.

The core algorithm remained roughly the same, with some additions and changes. The main additions are the three queues: the i-queue, v-queue and render queue. While traversing the Bounding Volume Hierarchy (BVH) tree, the i-queue is filled with all the previously invisible nodes and the v-queue with the previously visible nodes that are scheduled for an occlusion test. The render-queue is filled with all the previously visible nodes that need to be rendered. In addition, the render-queue can also be used for material sorting to decrease the number of state changes by grouping meshes with similar materials. Then material state changes only need to be done between material groups. The different queues and how they interact with one another can be seen in figure 2.3.

Instead of rendering previously visible nodes immediately, they are added to the render-queue for later rendering to reduce the number of state changes and they are also added to the v-queue if they need to be queried again. It is assumed that previously visible nodes will likely stay visible for some frames, so they do not need to be queried every frame,

thus reducing the number of queries. The amount of frames to wait is temporally jittered, so the re-tests do not fall into a regular pattern and do not cause lag spikes. Previously invisible nodes are added to the i-queue and occlusion queries are only executed after the queue reaches a certain batch size, in order to also minimise state changes. Before the queries are executed, the render-queue is rendered to fill the depth-buffer. Also, multiple objects of the i-queue are combined into a single query using a cost-benefit heuristic, so that objects that are likely invisible again are tested together, further reducing the number of needed occlusion queries. In addition, tight bounds are rendered instead of the bounding box when checking for occlusion. The tight bounds are simply the bounding boxes of the nodes further down the BVH tree, selected by a heuristic. This reduces the amount of falsely visible detected nodes, because the testing geometry fits the actual object more tightly, which improves the occlusion classification.

The reported results show a significant improvement over other tested methods like simple VFC, CHC and Near Optimal Hierarchical Culling [GBK06], outperforming them in all test cases. The main improvement is caused by the reduction in state changes by batching the queries with the aforementioned queues [MBW08].

This thesis evaluates how this method holds up when implemented and optimised for the Vulkan rendering API in comparison to the paper’s implementation in OpenGL.

2.3 Hierarchical-Z culling

Another interesting approach, that has gained more interest in the last few years is Hi-Z culling. This technique is also used in modern graphics hardware to speed up the depth check by reducing the number of z-buffer lookups. Taking z-max culling as an example, the max z-value of all the pixels in a fixed size tile is saved in a separate buffer. Then the min z-value inside that tile of the tested triangle is compared to the max value. If the min value is higher than the max value, the triangle is invisible and per-pixel depth testing is not needed, which spares time [AM18].

A similar approach can be taken to perform such a check manually for occlusion culling before sending all the geometry through the graphics pipeline. The basic idea behind this technique is to first render the depth of the occluders into a depth buffer and then calculate hierarchical MIP levels of the buffer. Each higher level contains the max value of the four corresponding pixels in the previous level to stay conservative. A selection of MIP levels of such a depth buffer can be seen in figure 2.4. Afterwards, the bounding geometry of the occludees that need to be tested is sent to the GPU, which calculates the screen space bounding rectangle. The size of the rectangle is in turn used to calculate the MIP level on which the depth should be looked up. The level is calculated in a way that the screen space rectangle will only overlap up to four of the MIP-levels pixels. Therefore, only the four pixels need to be looked up and compared to the depth of the screen space rectangle to determine occlusion, as can be seen in figure 2.5. There is one problem though, objects that are large on the screen are looked up in higher MIP levels where the depth is more coarse, which results in the occlusion detection being overly

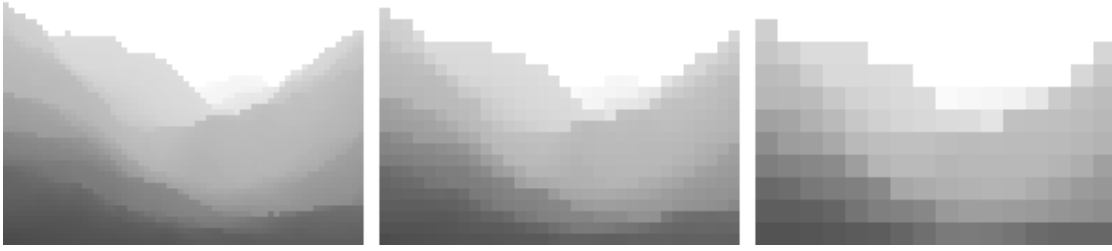


Figure 2.4: Three levels of the generated Hi-Z depth map used for Hi-Z culling. Reprinted from [Dan10].

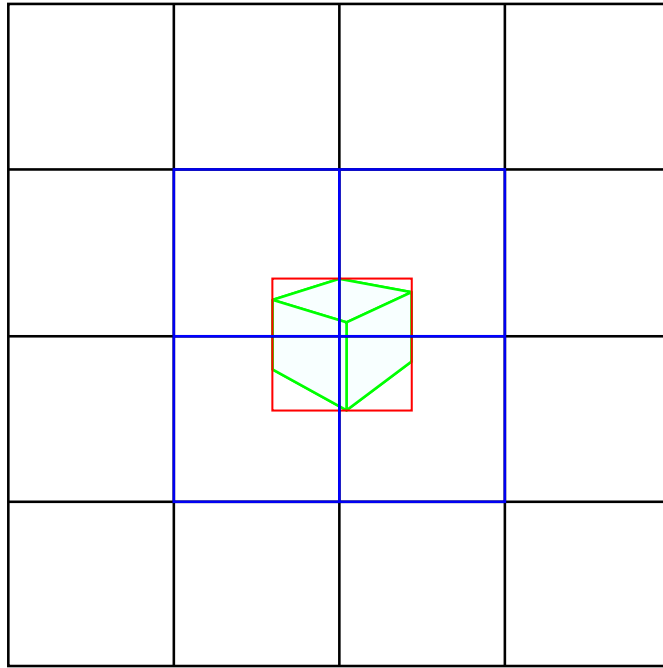


Figure 2.5: Shows the pixels of the lookup MIP level of the Hi-Z hierarchy. The AABB of the object is shown in green. Red shows the screen-space bounding box of the AABB and blue shows the texels that need to be looked up in the MIP level to check for occlusion.

conservative. But because large objects are more likely to be visible either way and a better and slower occlusion test of such an object could be slower than rendering, being overly conservative becomes less of an issue.

This idea is not new: in 1993 Greene et al. presented a paper in which they use an object space octree and a Hi-Z depth map to check if octree cells are occluded. Then the previously visible cells are immediately rendered in the next frame to pre-fill the z-buffer to leverage temporal coherence [GKM93]. Since then graphics hardware and APIs have made major progress, which makes this kind of approach feasible to implement

for demanding real-time rendering applications.

In a blog post, Daniel Rákos shows how this technique can be implemented. First, he renders the occluders into a framebuffer and then generates the Hi-Z depth map through multiple fullscreen quad passes. This map is then used in conjunction with a geometry shader that only emits primitives if the object passes the occlusion test, eliminating the need to wait for the results on the CPU [Dan10].

Nick Darnell also provides an implementation in DirectX where he renders the occluders into a downsized depth buffer and performs the Hi-Z build on that. Then he uses a compute pass on the objects' bounding spheres to calculate the screen-space size and perform the occlusion tests. Then he reads the results back to the CPU [Nic10].

In addition, similar approaches are presented in blog posts by Stephen Hill and Daniel Collin [SD11] and Kotas Anagnostou [Kot17], showing that the Hi-Z approach can be implemented in many different ways.

There are also several other open-source implementations of Hi-Z culling available, some are included in the blog posts that were already mentioned. Another implementation was provided by Nvidia. This project is publicly available and includes a test scene with different modes of culling and drawing [Nvi20]. The culling methods include frustum, Hi-Z and raster culling. Raster culling uses a geometry shader to generate and render bounding boxes and if a fragment passed the depth test, the object is marked as visible in a buffer. They also compare how simple CPU readback rendering (copying the data back to the main memory and drawing visible objects again) compares to using multi draw indirect (MDI) and the Nvidia command list (NVCmdLst). With the last two approaches, they do not need to synchronise the GPU and CPU at all, as the data stays on the GPU. With MDI they manipulate the `GL_DRAW_INDIRECT_BUFFER` to only render visible objects after culling. NVCmdLst operates on a binary token stream that represents GL commands which are then executed by the driver to reduce CPU load on API calls. This way a lot of state changes can be performed cheaply as they are handled by the driver directly, removing the overhead of doing multiple API calls. This is somewhat similar to the MDI approach but NVCmdLst also includes a terminate token that can be used to terminate indirect draws. With MDI, empty draw commands are still issued which costs GPU time. This can be eliminated with the terminate token [Nvi20].

2.3.1 Masked Software Occlusion Culling

Intel also took up the idea of using a depth map to cull objects but took a different approach. In a talk at Game Developer Conference (GDC) 2013, they presented Software Occlusion Culling. This method uses software rasterization of the occluders to the depth buffer using the CPU. To do this performantly, they tile the depth map so that each tile can be efficiently rasterized in few instructions with the Single Instruction, Multiple Data (SIMD) instruction set. Then it performs a depth test of the AABBs of the scene objects. If a visible pixel is detected, the object is immediately rendered and if not, every

pixel of the AABB has to be checked which costs more performance [Int13]. In 2016 they published a paper called "Masked Software Occlusion Culling" which is based on Software Occlusion Culling. There they do not store the actual depth per tile pixel, but only store a bitmask and two float depth values for the zeros (Z0) and ones (Z1) of the mask per tile. The Z0 depth is either set to the clear value or to the depth of a triangle that covers the whole tile. The Z1 depth is the merged depth of other contributing triangles. This can then be used for fast "hierarchical" Z occlusion culling on the CPU as multiple tiles can be checked simultaneously as more data fits into the SIMD registers. In comparison to standard Hi-Z culling, it is a bit more conservative but still performs better than some other Hi-Z implementations [HAAM16].

2.4 Culling methods in high-end game engines

This section presents occlusion culling techniques that are currently used by some of the most well-known game engines and game development companies. Culling methods vary widely, some use the middleware Umbra 3 and others use highly optimised algorithms that perform culling on single triangles in multiple stages. The following subsections present an overview of the methods used in the widely used game engines Unity, Unreal Engine 4 and CryEngine. In addition, it presents the well-optimised approach used in the Frostbite engine and the occlusion culling middleware Umbra 3, which is used in many triple-A game titles.

2.4.1 Unity

In current versions, Unity includes the middleware occlusion culling library Umbra 3 as its occlusion culling system. This system is abstracted behind a simple-to-use interface, where the user can tweak settings on the size of the largest occluder and the smallest holes that are present in the occluders. Then the occlusion information can be baked into an acceleration structure, used by Umbra at runtime, to perform culling. The pre-process baking step performs voxelisation on the scenes and determines which parts are connected and places portals there. This information is then used at runtime to perform visibility detection. A more detailed explanation is provided in the Umbra 3 section 2.4.5.

This system is easy to use as occluders and occludees only need to be tagged correctly in the engine. After that, the occlusion data just needs to be baked, although the parameters need tweaking depending on the scene for best performance. But the baking process can take a really long time depending on the parameters and the complexity of the scene. Unity also provides some debug views, as Umbra can be overly conservative, which can cause errors with some geometry and parameter settings [Uni19, Hou13].

2.4.2 Unreal Engine 4

Unreal Engine 4 (UE4) provides several occlusion culling methods to choose from out of the box. These include standard distance and view frustum culling but also precomputed

visibility and dynamic occlusion culling. The first and default dynamic occlusion culling technique is hardware occlusion queries, which was already detailed above. The difference is that the results are read back a frame later because of the latency of the queries. But this can cause objects to pop in on fast camera movement as they can become visible a frame too late. They also include Hi-Z culling, which is stated to be more conservative but works the same as occlusion queries in terms of frame latency. For mobile, they also provide software occlusion queries that rasterize a defined LOD level of the geometry on the CPU to perform culling. In addition, they include round-robin occlusion which is used for VR: It calculates occlusion only for one eye alternating each frame. Results are also used a frame later which can cause errors as stated previously [Gam20].

2.4.3 CryEngine

Crytek called their occlusion culling method Coverage Buffer. Their occlusion culling system uses the last frame's depth buffer and down-samples and reprojects it on the CPU. Then the AABBs of the occludees are rasterized on the CPU for z-testing. Their approach is not conservative as it can happen that it culls objects that are actually visible. On the other hand, they achieve better performance this way [Sco15].

2.4.4 Frostbite

EA put a lot of work into optimising Frostbite's culling mechanisms for the current console generation and PC. Graham Wihlidal presented their graphics pipeline at the GDC 2016 [Gra16]. The engine is mainly optimised for the AMD GCN architecture and different culling mechanics are performed, some of which even per triangle. AMD open-sourced this solution as GeometryFX as a part of AMD GPUOpen.

In Frostbite, rendering is performed in batches of meshes—meshes that have the same shader and vertex strides—which are sent to the GPU in a single MDI buffer. Then coarse culling is performed on clusters of 256 triangles. These clusters and their bounding cones are calculated in a pre-process step. The optimal bounding cones are found by projecting the normals of each of the clusters 256 triangles onto a unit sphere. Then the minimum enclosing circle of all the points on the sphere is found. The angle of the cone is the diameter of this circle and the cone normal is projected back to the Cartesian coordinates. There the clusters' bounding cone is used for backface culling by comparing its direction and angle to the view direction, the bounding sphere is used for frustum culling and the screen space bounding box is used for Hi-Z culling. Then the indirect draws are compacted and zero size draws are removed for better performance [US15]. Afterwards, per triangle culling is performed on the remaining geometry. First backface culling is performed. Then the min and max extents of the bounding box are used for small triangle culling. If two min/maxed edges fall onto the same pixel edge, the triangle gets discarded, as the triangle is between the pixel centres and would not be rasterized. Then simple frustum culling with four planes is performed again to further reduce the number of triangles. At last, a Hi-Z depth map is used for occlusion culling again. They

generate the depth map with the help of the HTILE metadata available on the GCN hardware or via software rasterization or a depth pre-pass on PC. Although this culling system is rather complicated it saves a good share of the rendering cost as they could cull nearly up to 80% of the triangles in their test scene [Gra16].

2.4.5 Umbra 3

Umbra 3 is an occlusion culling middleware that can be integrated into an existing game engine. It is built to operate on arbitrary polygon soups and uses a pre-process to generate the occlusion culling data structures and a runtime stage to perform the actual culling [3D18]. Because it is cheaper to license than to implement it yourself and because of its great performance, it is used by many AAA development studios and publishers, some of which are Activision, Bethesda, Capcom USA, Square Enix, CD Project Red [BS], the Unity game engine [Uni19] and more. Umbra provides a scalable approach for fast and conservative occlusion culling of any given triangle soup [Gam].

The preprocess

In the pre-processing stage, the polygon soup is split into a regular axis-aligned grid, where each cell is further voxelised. This can be seen in figure 2.6. Each voxel is then classified as solid or empty by flood filling the cell. Portals are defined at the faces of the cell with adjacent empty voxels. This is in turn used to create a connected cell and portal graph of the voxelised scene. Static objects are also voxelised and assigned to a cell to make culling easier later on.

Because this results in a lot of portals, the graph is further simplified by merging portal cells that do not contribute much to occlusion. This is done iteratively until a certain occlusion score threshold or memory threshold is reached. But this representation is still too complex for runtime traversal, so another simplified version is created called the view-tree. To do so, adjacent voxels with the same classification are grouped together. Also, regions where the camera will never be, like high up or below the map, can be collapsed into single cells. A top-down view of such a graph can be seen in figure 2.7.

Because this pre-process takes a lot of computational power and memory, the generation is split into separate tiles. These tiles can be calculated independently from each other in parallel and on distributed systems. In addition, not every tile needs to be recalculated if something is changed, further improving generation performance for incremental updates. Therefore this system can also be used in huge open world scenarios [3D18].

Runtime occlusion culling

For the real-time part, the cells of a tile are processed in a front-to-back breadth-first fashion. There the portals are software-rasterized into a one-bit coverage buffer, where each set bit represents empty space of the cell that is visible through the portal. Then all the screen space bounds of the cell's static objects are tested against the coverage. If the

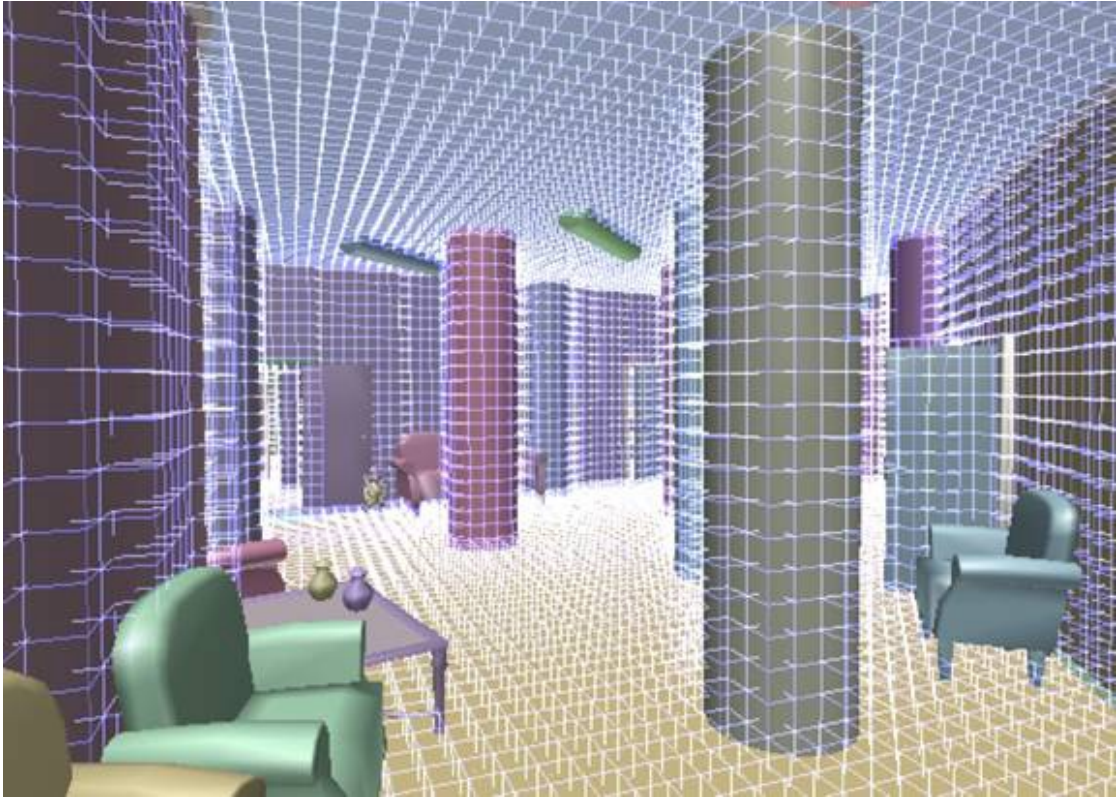


Figure 2.6: The voxelisation of the occluder geometry used in Umbra 3. Reprinted from [3D18]

bounding rectangle overlaps a set bit, it is visible. Because static objects got assigned to the cells in the pre-process, this can be done easily.

Because dynamic objects cannot be assigned to cells easily at this point, another solution is used. While traversing, a low-resolution depth buffer is generated with the cell's bounds. This buffer is then used to cull any dynamic object in the scene [3D18].

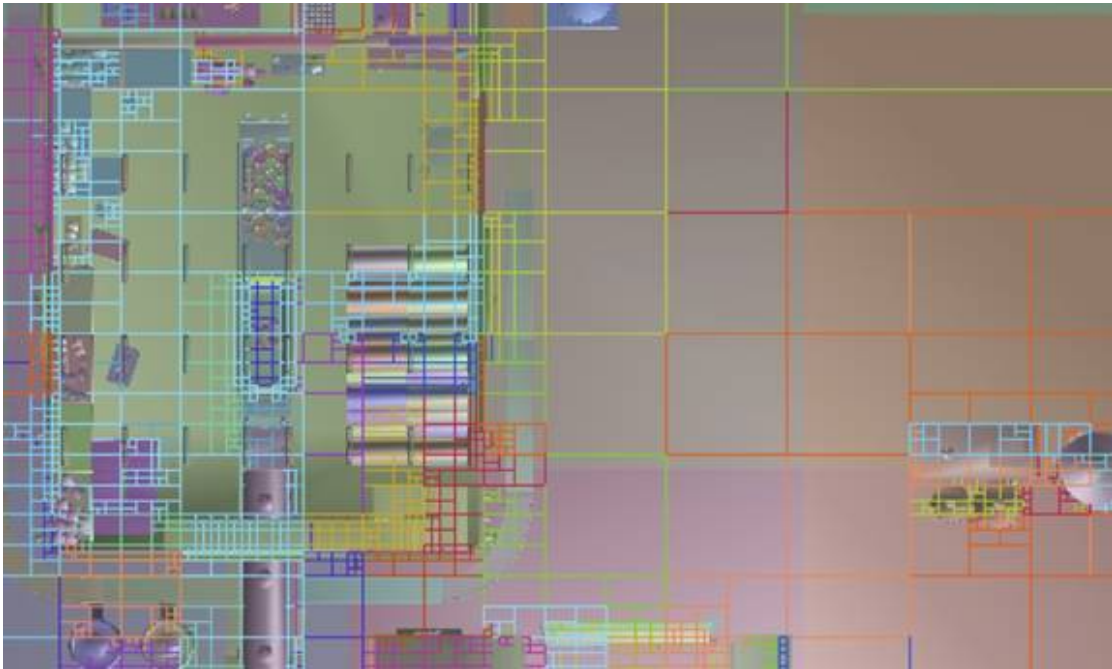


Figure 2.7: The optimised cells generated in the Umbra 3 pre-process. Reprinted from [3D18]

Problems

Re-implementing CHC++ for a newer hardware and graphics API—in our case the Vulkan API—comes with some challenges that need overcoming to achieve similar results as in the original study [MBW08]. Most notably is the difference between OpenGL, which was used in the original paper, and Vulkan, as these two APIs are rather different in terms of draw call submission, state changes and synchronisation. This especially becomes a problem because of the dynamic nature of the CHC++ algorithm, as it repeatedly has to switch between rendering and occlusion queries. Also, some benefit of using occlusion culling in comparison to no culling is likely lost because the improved design and low-overhead nature of the Vulkan API, which means that more optimisations are needed to get better performance out of CHC++. In addition, transparency was not addressed in the original paper, therefore a new solution had to be found to correctly render transparent and alpha-blended geometry while still being able to cull said geometry.

The following sections detail the problems that were encountered while implementing and analysing the behaviours of CHC++ in Vulkan on modern hardware. Solutions to these problems are presented in the Implementation chapter afterwards.

3.1 Vulkan vs. OpenGL

Although OpenGL and Vulkan have some things in common, setting the current state and performing work on the GPU is handled rather differently. In OpenGL, a global state called the OpenGL context is used to define certain rendering parameters which can be changed at will through different OpenGL commands. For example, this can be changing a setting of the fixed function states, binding a shader or texture, or changing blending behaviour. Each of these API calls needs to go through the graphics driver to be validated and executed, which costs time. Because a lot of such state changes are often needed to get the context into a desirable configuration, a significant overhead is introduced. The

same goes for draw commands, which can be executed any time—provided the state is correct.

Vulkan on the other hand uses a more object-oriented approach without the need of such a global state. Therefore, objects that represent a collection of configurations are used. As an example, one of these objects is the Pipeline State Object (PSO), which is an object containing the configurations of the shaders and its needed data bindings, the viewport, the rasterizer and more. Therefore only one API call is needed to change the whole pipeline, whereas OpenGL needs multiple. This reduces the amount of needed API calls immensely, which results in less overhead and better performance in most cases. Because Vulkan uses these objects to define states, nearly everything has to be specified explicitly. This can make developing Vulkan applications difficult as most things that are implicitly specified in OpenGL need to be taken care of in detail. This leads to a lot of boilerplate code needed to perform simple tasks, which can be daunting at first and also lead to errors more easily. On the other hand, this provides the developer with the most flexibility to achieve optimal performance.

Vulkan also has a different concept for performing operations on the GPU. Most operations are not simply started by a call to some API function but instead need to be recorded in a so-called command buffer. A command buffer then needs to be submitted to a queue for execution on the GPU. These command buffers are created on the CPU and can also be used multiple times without the need to re-record them. Therefore only a single submission to the GPU is needed to perform a multitude of pipeline state changes, resource bindings, compute passes, synchronisation, draw calls and so on. This vastly increases the amount of work that can be done in a single API call, reducing the overhead to a minimum. Although the driver executes most OpenGL commands asynchronously, Vulkan enables the developer to decide when a huge chunk of work—a command buffer—should be executed asynchronously on the GPU. This makes it easier to schedule parallel computations on both the CPU and GPU, but also introduces the need for explicit synchronisation on both sides.

Therein lie some of the major problems that were encountered while implementing CHC++ for Vulkan, which will be discussed in the following subsections.

3.1.1 Command buffers

Due to the fact that command buffers aggregate multiple draw calls into a single submit, draw calls become rather cheap in comparison to OpenGL where each draw call has to go through the driver separately. Unless something like MDI is used, which can be used to write multiple draw parameters into a single buffer which only need one draw call to execute.

CHC++ is dynamic by nature because rendering and occlusion queries are interleaved to reduce both CPU stalls and GPU starvation. This is achieved by issuing queries only after enough nodes are accumulated. Then the renderqueue is drawn and the queries are issued. This can be implemented straightforwardly in OpenGL, simply by issuing draw

calls and queries. But this becomes a problem when you need to think about command buffers, as you need a varying amount of them which need to be allocated.

Therefore, command buffer management needs to be taken care of. Because the number of submissions vary between frames, the amount of needed command buffers also varies. One could either allocate command buffers at runtime, which will cost time or allocate a fixed amount of buffers and reuse them when needed. Allocating at runtime will likely introduce more overhead, which could cause lag spikes in cases where a lot of them need to be allocated. In turn, a fixed amount will increase memory usage and could cause CPU stalls when all buffers are in flight (in computation on the GPU) and the CPU needs to wait on the completion of one of them.

3.1.2 Submission overhead

As previously stated, command buffer submits come with an overhead both on the CPU and GPU side, therefore submissions should be kept to a minimum. Because of that, it needs to be decided what should be recorded into a single command buffer and what needs to be split up into multiple. CHC++ relies on work being executed on the GPU while the CPU is doing traversal of the BVH, so it can submit previously visible queries while waiting on the already submitted queries to finish.

Therefore, multiple submits *need* to happen in order for that to be possible, else the CPU would just stall until all the queries are completed. One could simply submit at each state change from rendering to queries, but that would result in many submissions causing poorer performance because of the overhead. Or different draws and queries could be batched into single submits reducing the number of submissions. What makes this difficult is that at multiple stages in the algorithm either geometry needs to be rendered or occlusion queries need to be issued. The problem therein is, that it is not known in advance when the switch from rendering to occlusion queries happens, as additional queries could be issued when another query finishes in the middle of the BVH traversal. For example, when a multiquery—a single occlusion query for multiple BVH nodes—fails, occlusion queries for each of the nodes in the multiquery need to be issued, but it is not known when the next batch of queries are issued, making it hard to batch. The same problem occurs when issuing v-queue queries while waiting for other queries to complete. A proper balance needs to be found to ensure that the overhead is kept to a minimum and the CHC++ operation is not hindered.

3.1.3 Synchronisation

In the case of CHC++ in OpenGL, synchronisation is not a problem, as the draw calls and framebuffer accesses are implicitly synchronised. In Vulkan however, everything needs to be specified explicitly and synchronisation is not an exception. Command buffer submission needs to be synchronised on the GPU and CPU because they are executed asynchronously. Semaphores can be used to synchronise different submits on the GPU to establish a before-after relationship between certain pipeline stages. The state of a

semaphore can also be checked on the CPU side. Additionally, because command buffers cannot be re-recorded while they are being executed on the GPU, fences are needed to synchronise them on the CPU side to know when they have finished executing. A fence is signalled when the command buffer execution has completed, which enables the re-recording of said buffer. Pipeline barriers can be used for synchronisation inside of a command buffer, as commands are not necessarily executed in the recorded order.

In the case of CHC++ it needs to be guaranteed that rendering has finished and all depth values have been written before the occlusion queries are executed. Also, the next rendering step needs to wait for the queries to finish, as the depth buffer cannot be written while the queries are executing. Therefore, depending on how the command buffers are used, semaphores and pipeline barriers need to be utilised to ensure proper ordering on the GPU side. In addition, command buffer submits need to use fences in order to ensure that they are only re-recorded if they have already completed execution.

3.2 Transparency

Transparency and alpha masked geometry was not part of either CHC and CHC++, but is vital for modern games and rendering in general. Transparent objects are a bit of a special case for occlusion culling because they need to be handled a bit differently than solid geometry. Foremost, transparent or alpha masked objects also need to be culled the same as other objects, as they can be occluded like others. But transparent objects especially are not allowed to occlude other geometry and therefore need to be taken out of the depth buffer rasterization that is needed for the occlusion queries. Alpha masked objects can still be used as occluders, if only the silhouette could be rendered into the depth buffer and not the geometry itself, which could cause false positives while culling.

Additionally, transparent objects need to be alpha blended correctly to get the right resulting colour. The easiest way for that is to render them back-to-front after everything else has already been rendered. Therefore, rendering for these objects needs to be postponed and cannot happen while the render-queue is rendered. This also falls in line with not rendering to the depth buffer for culling, solving one problem while solving another.

Implementation

The application was developed in C++ version 17 and Vulkan SDK version 1.2.131.2. At first, a naive re-implementation was made with excessive synchronisation on the CPU to see where problems with Vulkan arise. Step by step these problems, as detailed in chapter 3, were found and solutions were integrated. Because of the dynamic nature of CHC++, some of these problems were hard to find. Therefore, profiling of the application via Nvidia Nsight and statistics exported from the application were vital to finding them.

A lot of the algorithm is identical to the original algorithm with the additions to solve the problems in conjunction with Vulkan. A Surface Area Heuristics - Bounding Volume Hierarchy (SAH-BVH) of the axis-aligned bounding boxes of the scene's objects was generated at runtime, but without tight bounds for the leaf-nodes, as this would only provide a minor performance enhancement. The scenes were loaded from glTF files. Heuristics used by the original CHC++ algorithm—mainly the generation of multiqueries—stayed the same, as they should not have changed by much. The following sections detail the solutions to the problems described in chapter 3.

4.1 Handling command buffers

As stated in sections 3.1 and 3.1.1 it is not trivial to port an algorithm from OpenGL to Vulkan due to their major differences in handling draw calls. Command buffers need to be handled accordingly because it is not known in advance how many are needed for each frame in CHC++. As a simple solution, this implementation uses a ring-buffer of command buffers with a fixed size. This means, whenever the last command buffer of the ring-buffer was used, the first one is used again the next time, so no new command buffers need to be allocated at runtime. Looking at the pseudocode of algorithm 4.1 at line 22 the function *StartCommandBuffer()* simply retrieves a new command buffer from the ring-buffer and starts recording draw commands.

For this thesis, this is a sufficient solution, but it also has some drawbacks if implemented in a bigger project. The main problem is having the right amount of buffers at hand to never run out while rendering, because if all the command buffers are in use at the same time, the CPU needs to wait for one to finish. This could probably be solved by dynamically allocating additional command buffers when more are needed. This would improve performance if the time needed to allocate a new buffer is less than the time waited on the completion of a buffer. This project just uses a sufficiently sized ring-buffer, so this case never occurs.

4.1.1 Synchronisation

With multiple command buffer submits per frame, proper synchronisation is needed. Therefore, the command buffer ring-buffer does not only store command buffers but also a fence and semaphore primitive for each command buffer. This enables synchronisation before we start recording on the CPU using the buffers fence, stalling if the command buffer is already in execution. Although this should not happen with a sufficiently sized amount of command buffer, this needs to be done to avoid undefined behaviour. In addition, the semaphores allow us to easily achieve synchronisation on the GPU side, as we can simply retrieve the semaphore of the previous submit from the ring-buffer. This makes it possible to effortlessly chain multiple queue submits with proper synchronisation.

4.2 Reducing queue submits

Looking back to section 3.1.2, submission overhead can become a problem when too many submissions are issued to the queue. While implementing, it became apparent that this is the main issue that needs to be solved in order for CHC++ to reach acceptable performance under Vulkan. The reason is that without proper batching, a lot of small submissions need to be made, that suffer from overhead that is bigger than the actual work being done on the GPU. Therefore, a lot of work went into finding out where the drawing commands and occlusion queries can be batched into single submits as much as possible.

4.2.1 Batching draws and queries

Looking at the original CHC++ algorithm, it can be observed that the renderqueue is always rendered before one or multiple occlusion queries are issued. This leads to the first most obvious choice to create a batch. This can be seen in the function *IssueMultiQueries()* at line 26 in algorithm 4.1. There the command buffer is started in the beginning, then the draw commands of the renderqueue and the query commands of the occlusion queries are recorded. At last, the command buffer is submitted. A pipeline barrier is recorded between the draw and query commands to ensure that all draw commands finished writing the depth buffer before the queries are executed. This allows for the execution of a multitude of draw and query commands with a single queue submit.

Algorithm 4.1: Opti-CHC++ — Traversal

```

1 Function Traverse():
2   DistanceQueue.push(Root);
3   while !DistanceQueue.Empty() // !QueryQueue.Empty() do
4     // Handle Queries
5     CheckQueries();
6     // Actual Traversal
7     if !DistanceQueue.Empty() then
8       N = DistanceQueue.Pop();
9       if InsideViewFrustum(N) then
10        if IntersectsNearPlane(N) then
11          N.IsVisible=True;
12          PullUpVisibility(N);
13          TraverseNode(N);
14        else
15          if !WasVisible(N) then
16            QueryPreviouslyInvisibleNode(N);
17          else
18            if N.IsLeaf() && QueryReasonable then
19              vQueue.Push(N);
20              TraverseNode(N);
21      // Issue queries, if nothing else to do
22      if DistanceQueue.Empty() && QueryQueue.Empty() then
23        IssueMultiQueries();
24  if !RenderQueue.Empty() // !vQueue.Empty() then
25    StartCommandBuffer();
26    RenderRenderQueue();
27    while !vQueue.Empty() do
28      IssueQuery(vQueue.Pop());
29    SubmitCommandBuffer();
30  while !QueryQueue.Empty() do
31    HandleOcclusionQuery(QueryQueue.Pop());
32  RenderTransparentMeshes();

```

This is also used in the function *CheckQueries()* (line 1 in algorithm 4.2) where v-queue nodes are recorded to a single command buffer, as long as the first query has not yet finished, which would otherwise cause a lot of submits. Although this eliminates most of the small submits, they can still happen when only a single node of the v-queue is queried and then the first query finishes for example. But it could be observed that this is an adequate solution nevertheless, as the edge cases do not happen too often to cause a significant performance impact.

Algorithm 4.2: Opti-CHC++ — Functions 1

```

1 Function CheckQueries():
2   while !QueryQueue.Empty() && (DistanceQueue.Empty() //
   FirstResultAvailable) do
3     while !FirstResultAvailable && !vQueue.Empty() do
4       if !CommandBufferStarted then
5         StartCommandBuffer();
6         RenderRenderQueue();
7       IssueQuery(vQueue.Pop());
8       CheckFirstQueryAvailability();
9       if CommandBufferStarted then
10        SubmitCommandBuffer();
11    HandleOcclusionQuery(QueryQueue.Pop());

```

It was also observed that the *if* at line 19 in algorithm 4.1 caused a lot of small submits with its original *if* condition that only contained a check if the distance-queue was empty. This simply happened too often, which was not a problem in OpenGL. Now with the added check—if the query-queue is also empty—this happens far less often, which in turn reduces the amount of submits.

4.2.2 Batching failed multiqueries

Because multiqueries test multiple nodes in a single query to reduce the number of queries, all of them need to be re-tested when one such multiquery fails. In the original implementation, these nodes are tested immediately when the failed query is checked. In scenarios, where not much is rendered and queried, this does not become much of a problem as this does not happen all too often, because the multiquery creation heuristic performs well. There the higher submission count overhead does not degrade performance by much. However, the more queries are issued, the higher the amount of failed queries. This would mean the number of submits would scale directly with the number of failed queries, which causes significant performance penalties in low-occlusion and high-density viewpoints of a scene. Because the queries are checked while traversing, this could happen at any point, making batching harder. Therefore another solution had to be found.

The solution we found is to accumulate all the nodes of the failed multiqueries and issue their occlusion queries at a later point. For this, another queue—the c-queue—was introduced. Then, whenever we encounter a failed multiquery, we add its nodes to the c-queue (line 16 in algorithm 4.3). Now whenever multiqueries are issued, the queries of the c-queue nodes are also issued afterwards. This effectively reduced the submits to a minimum, even if a lot of the multiqueries fail. Also delaying their checks can yield better culling results, as more of the visible geometry has been drawn at that point.

4.3 Handling transparency

As stated in section 3.2, transparency rendering was not handled by the original algorithm and was therefore integrated into this implementation. Two separate queues were added, one for alpha-blended and one for alpha-masked geometry. Now, each time a transparent object is added to the renderqueue, it is actually added to either the alpha-blended or alpha-masked queue. Because the BVH tree is already traversed front-to-back, the objects in the two alpha queues just need to be rendered in reverse order at the end of the traversal to achieve back-to-front sorting.

First of all, this enables the correct rendering of transparent objects with correct blending because of the back-to-front order. Additionally, the transparent objects are also correctly culled if they are out of the viewing frustum or occluded, as this approach does not interfere with the general culling algorithm, i.e., the transparent objects are not handled differently while culling. Also, because the transparent meshes are never added to the actual renderqueue, they are not present in the depth buffer while checking for occlusion. This elegantly solves the problem that transparent objects should not be considered as occluders.

Algorithm 4.3: Opti-CHC++ — Functions 2

```
1 Function TraverseNode (Node):
2   if Node.IsLeaf() then
3     if Node.IsOpaque() then
4       | RenderQueue.Push(Node);
5     else
6       | TransparentQueue.Push(Node);
7   else
8     | DistanceQueue.Push(Node.Children);
9 Function PullUpVisibility (Node):
10  while !Node.IsVisible do
11    | Node.IsVisible = true;
12    | Node = Node.Parent;
13 Function HandleOcclusionQuery (Query):
14  if Query.Visible then
15    if Query.NodeCount > 1 then
16      | // handle failed multiquery
17      | cQueue.AddAll(Query.Nodes);
18    else
19      | // handle failed single node query
20      | PullUpVisibility (Query.FirstNode());
21      | TraverseNode (Query.FirstNode());
22  else
23    | Query.SetNodesInvisible();
24 Function QueryPreviouslyInvisibleNode (Node):
25  iQueue.Push(Node);
26  // Issue queries when batch size is reached
27  if iQueue.Size() > batchSize then
28    | IssueMultiQueries();
29 Function IssueMultiQueries():
30  StartCommandBuffer();
31  RenderRenderQueue();
32  while !iQueue.Empty() do
33    | MQ = GetNextMultiQuery (iQueue);
34    | IssueQuery (iQueue);
35    | iQueue.PopNodes(MQ.Nodes);
36  while !cQueue.Empty() do
37    | IssueQuery (cQueue.Pop());
38  SubmitCommandBuffer();
```

Results and Discussion

The main goal of this thesis was to adapt CHC++ to a modern rendering API—Vulkan in our case—and evaluate how the results compare to the previous implementation and how it performs on today’s hardware.

The project was tested on a Windows 10 x64 PC with an Intel i5 6600k CPU overclocked to 4.5 GHz, 16 GB of RAM and an Asus Strix - Nvidia GTX 1070 GPU at default clock rates. As a main test scene we used the Neu Rungholt Minecraft map by kesha, because it contains a lot of buildings which provide a lot of occlusion. The map was exported via mineways into an OBJ format, in a way that every chunk and material are in a separate mesh to get a more fine-grained subdivision of the map. Splitting the world into smaller pieces allows for more fine-grained occlusion culling than it would have if only split by material. The OBJ file was then converted to glTF using the NodeJS tool *obj2gltf*. Additionally, the Lumberyard Bistro [Lum17] was used for testing, as it provides high and low occlusion areas with a lot of transparent objects. The sub-models were imported into Blender and then exported as a single glTF file. Texture size was halved to not overflow GPU memory and materials were manually edited in the glTF file to have correct transparency.

A walkthrough was recorded for each scene, which was subsequently replayed using different rendering methods. Statistics were recorded at every frame while replaying. GPU times were recorded using Vulkan Timestamp queries for accurate measurements. Generated graphs and the Nvidia Nsight System profiling tool were used for evaluation.

5.1 Results

The new optimised CHC++ algorithm titled Opti-CHC++ shows significant performance gains in high occlusion environments over VFC and is on par or slightly worse in low occlusion environments. This can be seen in the frame time comparison of the New

Rungholt walkthrough in figure 5.1. The unoptimised CHC++ performs nearly as good as the optimised version in most cases but exhibits huge spikes in frame time when a lot of queue submits happen. Opti-CHC++ also performs great in the worst-case scenario here, where the camera looks down at the whole city and most geometry is not occluded. This can be seen in the graph after frame 3500, where the camera is gradually rising upwards. In this case, it can also be observed that the unoptimised version of CHC++ starts to perform consistently worse than VFC and even no-culling. Additionally, framerate is more stable when compared to VFC and the unoptimised version. Opti-CHC++ stays far over 144 FPS until the worst-case happens, whereas VFC fluctuates repeatedly because of the much higher overdraw.



Figure 5.1: CPU Frame time comparison of Opti-CHC++ vs. Unoptimised-CHC++ vs. Frustum Culling vs. No-Culling in the Neu Rungholt walkthrough.

On average, Opti-CHC++ reaches around 497 FPS and VFC reaches around 200 FPS in the Neu Rungholt walkthrough. This shows that Opti-CHC++ is nearly 2.5 times faster than VFC in this test scene. In addition, Opti-CHC++ does not fall below 91 FPS, whereas the FPS of VFC fall as low as 45. Opti-CHC++ was also around 1.6 times faster on average in this scene than the unoptimised version.

In comparison, the walkthrough of the Lumberyard Bistro depicted in figure 5.2 does not perform as well. It only performs better from frame 400 to 800, where the entire interior

of the Bistro is occluded. In the rest of the walkthrough, almost no geometry is occluded. Although Opti-CHC++ does not perform better than VFC in this case, it does not perform significantly worse in this scenario. This means that the overhead caused by the occlusion checks does not degrade performance by much and Opti-CHC++ is therefore a viable occlusion culling solution. In huge parts of this walkthrough the unoptimised version of CHC++ performs significantly worse than without culling, because of the huge overhead of the many submits and excessive synchronisation.

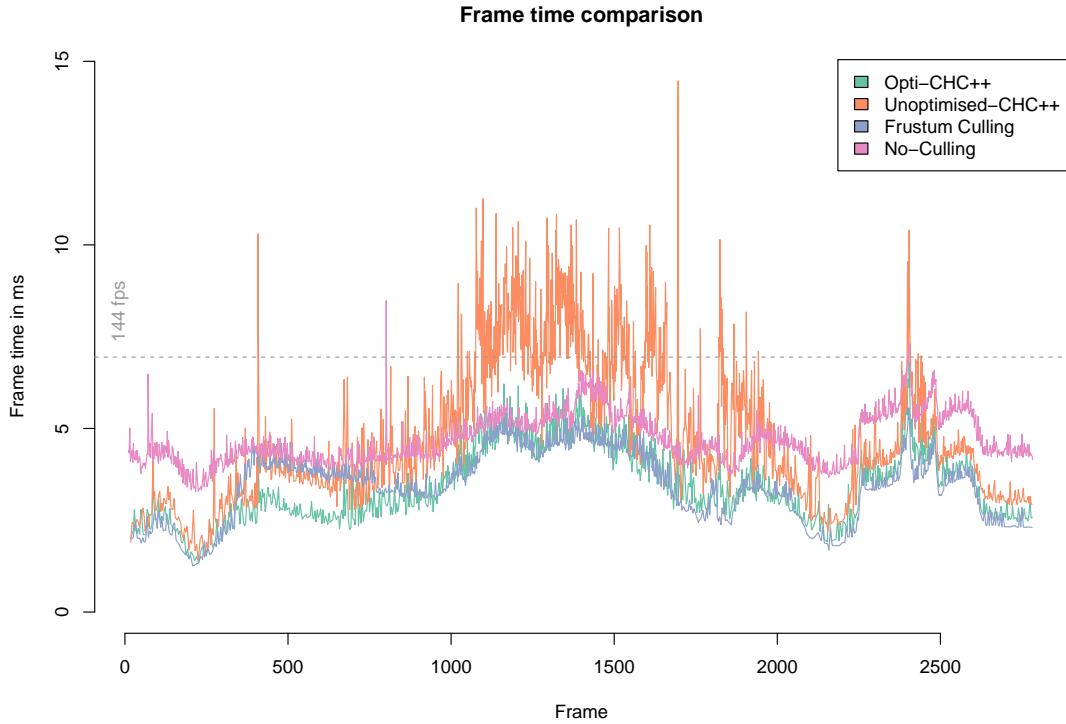


Figure 5.2: CPU Frame time comparison of Opti-CHC++ vs. Frustum Culling vs. No-Culling in the Lumberyard Bistro walkthrough.

Also, all the transparent objects like glasses, bottles and foliage are correctly rendered and culled in this scene because of our transparency handling addition to the algorithm, as can be seen in figure 5.3.

Opti-CHC++ and VFC both reach the the same average FPS of 295 in the Bistro. This shows that there is not much benefit from using this occlusion culling method if the scene does not contain much occlusion. On the other hand, this further solidifies the argument that Opti-CHC++ does not degrade in performance in such cases. In this scene, Opti-CHC++ is also around 1.3 times faster than the unoptimised version on average.

Looking at GPU frame timing of the Rungholt walkthrough in figure 5.4, it can be

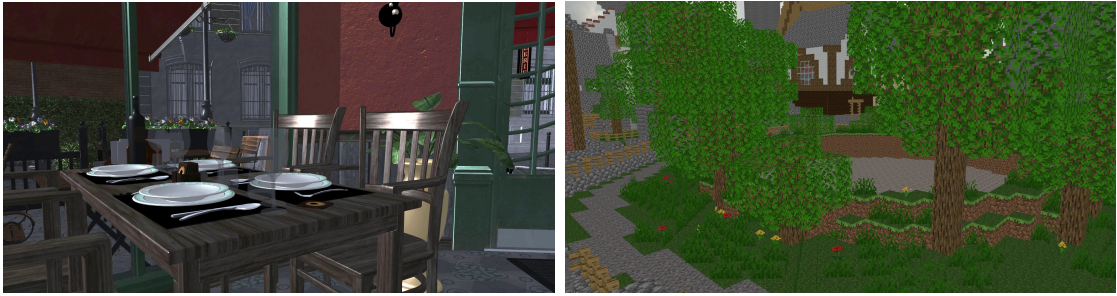


Figure 5.3: The correct alpha blending of the glasses and window panes in the Lumberyard Bistro scene can be seen on the left. On the right, the correct alpha cutouts of the leaves in the Neu Rungholt scene can be seen. There the correct blending of the water and the window panes in the background can be observed.

observed that most of the time is used by rendering and only a tiny amount is used by the queries itself. Towards the end of the graph, the gap between the rendering and the overall frame-time widens, showing the amount of additional overhead introduced by multiple queue submits. Figure 5.7a and 5.7b also show how much the number of queries and therefore submits increase in this timespan as more of the scene becomes visible. This and the profiling done in Nvidia Nsight Systems in figure 5.5 shows that the GPU is nearly fully utilised while rendering a frame. This means that the graphics card is optimally used and hardly any starvation happens.

But looking at the frame-timing on the CPU side, it can be seen that the majority of the time is wasted on waiting on the completion of occlusion queries as can be seen in figure 5.6. This means that the CPU is stalling for most of the frame, which is not desirable, as this time could be used to perform other tasks on the same thread. This could be attributed to the significant performance increase of today's hardware and because work is submitted less often to the GPU, increasing the latency of the occlusion queries. But because most modern rendering engines operate multi-threaded, this is less of a problem as other threads are not stalled. In addition, it can be noticed that the queue submission time has been brought to a minimum by various optimizations discussed in chapter 4.

Additionally, VFC renders around 8.1 million triangles on average, Opti-CHC++ 1.2 million and an optimal culling solution 450.000 in the Neu Rungholt scene as can be seen in figure 5.8. This shows that Opti-CHC++ provides approximately a 6.8 times reduction of rendered triangles over VFC in this scene. But it also shows that it still renders approximately 2.6 times more triangles than an optimal culling solution. The reason for that is that visible objects are assumed to stay visible for some frames and visibility changes are ignored in this time period, resulting in objects that are wrongly classified as visible. But because this is an integral part of the CHC++ algorithm to improve performance, a different approach is likely necessary to get the number of rendered triangles to optimal levels.

However, in the Lumberyard Bistro scene VFC renders around 1.2 million triangles

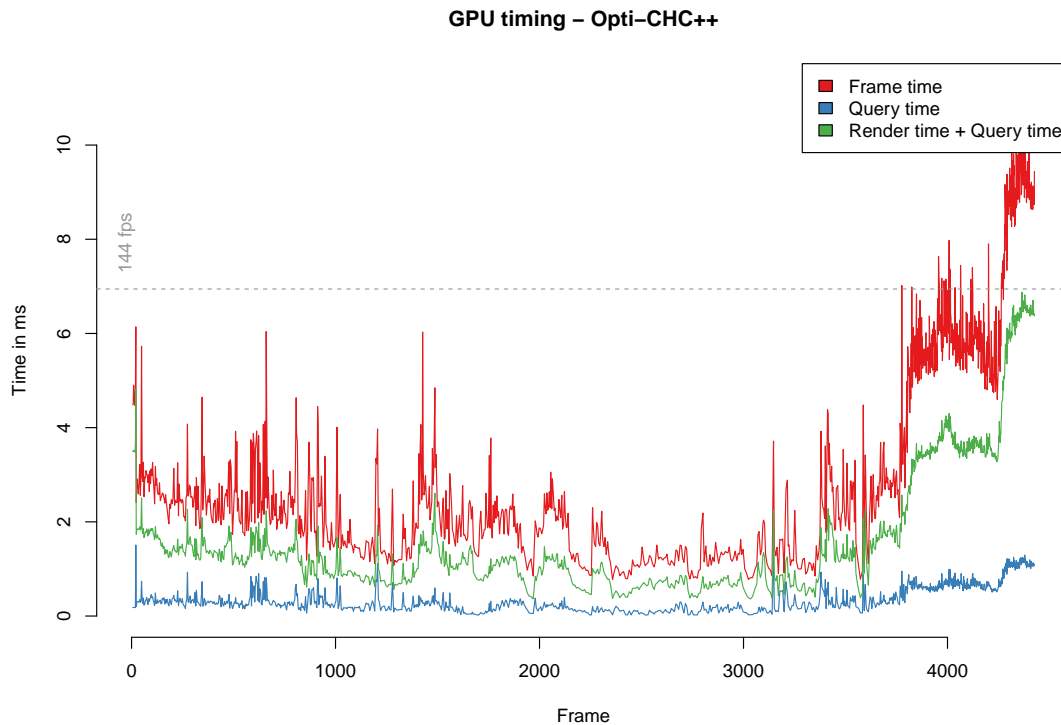


Figure 5.4: GPU Frame timing of the Neu Rungholt walkthrough. Shows that most of the time is used by rendering and overhead when a lot of submits happen.

on average, Opti-CHC++ 625.000 and with optimal culling 470.000 which can be seen in figure 5.9. This only results in an approximate 2 times improvement from VFC to Opti-CHC++, likely due to low occlusion in this scene. Opti-CHC++ still renders around 1.3 times more triangles than the optimal solutions, likely because of the same reason as in the Rungholt scene. But it shows that triangle numbers and performance can still be improved in high and low occlusion scenes.

5.2 Discussion

Although the optimised CHC++ algorithm performs well in Vulkan, the performance difference between CHC++ and VFC found in the original paper could not be replicated. Looking at figure 5.10 it can be seen that the difference is a lot bigger than with Opti-CHC++ which can be seen in figure 5.1 and 5.2. Also, the original implementation performed consistently better, never falling back to the performance levels of VFC.

Sadly it is not entirely clear what OpenGL version the original implementation used, but it is likely version 3.0 or earlier, as this version was released in the same year as the CHC++ paper. The performance difference of the original CHC++ implementation

5. RESULTS AND DISCUSSION

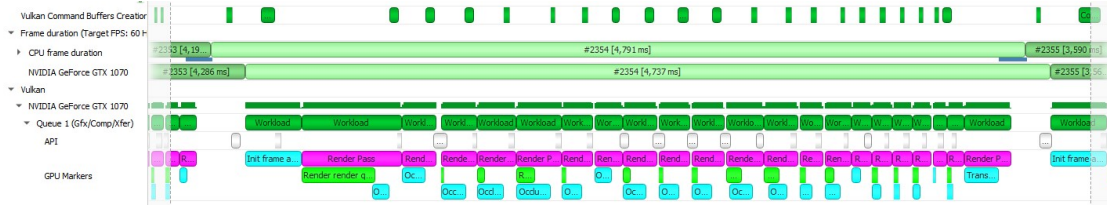


Figure 5.5: Screenshot of Nvidia Nsight System profiler of a frame of the Rungholt scene. The top row shows the time needed on the CPU for the command buffer creation. The next rows show the frame time on the CPU and GPU for this frame. Submits are back-to-back in the *Queue 1* row, showing that the GPU is nearly fully utilised. The *GPU Markers* row show how much is rendered (green) and queried (cyan) per render pass (magenta).

was far greater back then, assumedly because OpenGL API calls were more expensive as a lot of API features of newer versions were not available. This probably made the benefit from reducing state changes and occlusion queries far greater than it is now with Vulkan. Also, the binary that accompanied the original implementation in [BMW09] was tested on newer hardware. Comparing the performance of CHC++ and VFC in this application, CHC++ performed up to ten times faster than VFC, showing similar performance differences than on older hardware used in the original paper. This further proves the point, that the high performance increase of CHC++ could be attributed to the older OpenGL version being used in the original implementation because it still performs similarly. Figures 5.8 and 5.9 also show that that the number of culled triangles approaches the optimum, meaning that the original performance difference is likely not attainable in modern implementations. This is because further reduction in rendered triangles will likely not yield as big of performance improvement. Additionally, a lot of OpenGL extensions that could increase performance in this case were not yet available. Therefore, a new implementation with more sophisticated OpenGL methods—like Zero-Driver Overhead [CGJT14] using MDI—would probably yield similar results to Vulkan in terms of performance difference.

Furthermore, most of the game engines described in section 2.4 use either Umbra 3 or some form of Hi-Z culling and only Unreal Engine 4 uses occlusion queries. This shows that there is a reason why almost all of them do not use occlusion queries, which is most likely the high latency from occlusion query to result. Even UE4 uses the occlusion culling results one frame later as a simple solution to reduce the latency, which in turn introduces unwanted artefacts. Even though CHC++ solves the latency problems of occlusion queries, a simpler more straight-forward algorithm—using Hi-Z culling for example—could prove to yield better results. Simpler algorithms and scheduling could mean that less time is lost to multiple submits and also that more of the geometry can be culled than CHC++ if visibility does not need to be assumed for multiple frames. Different techniques for occlusion detection are presented in chapter 2 and some ideas on other possible approaches are shown in section 6.2.

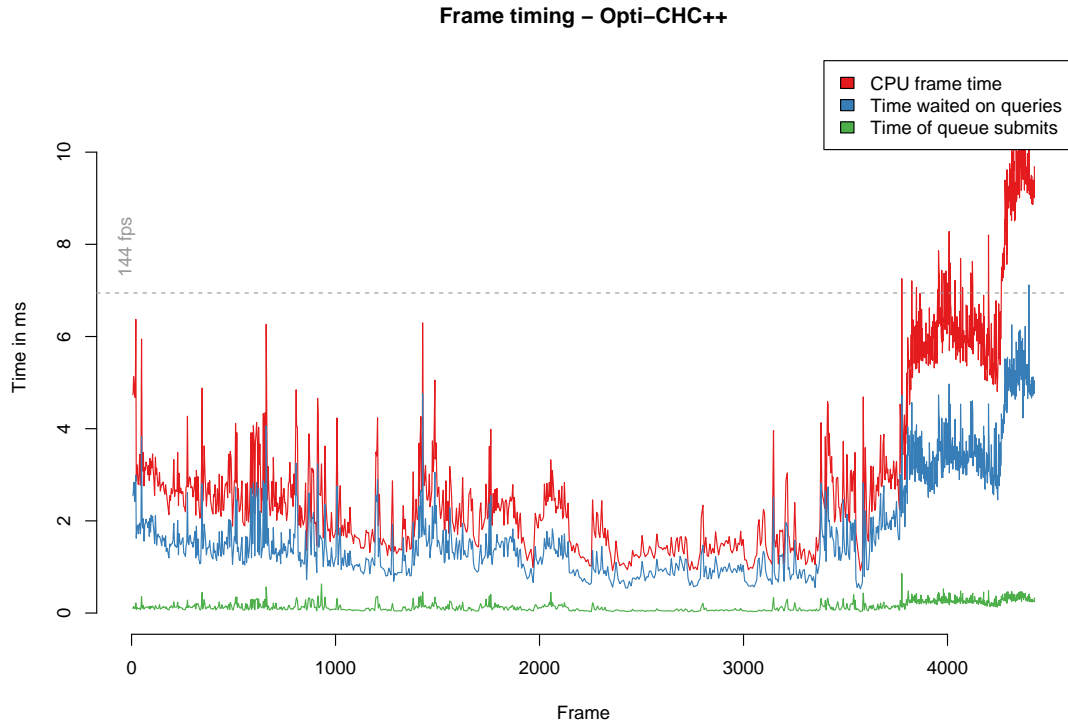
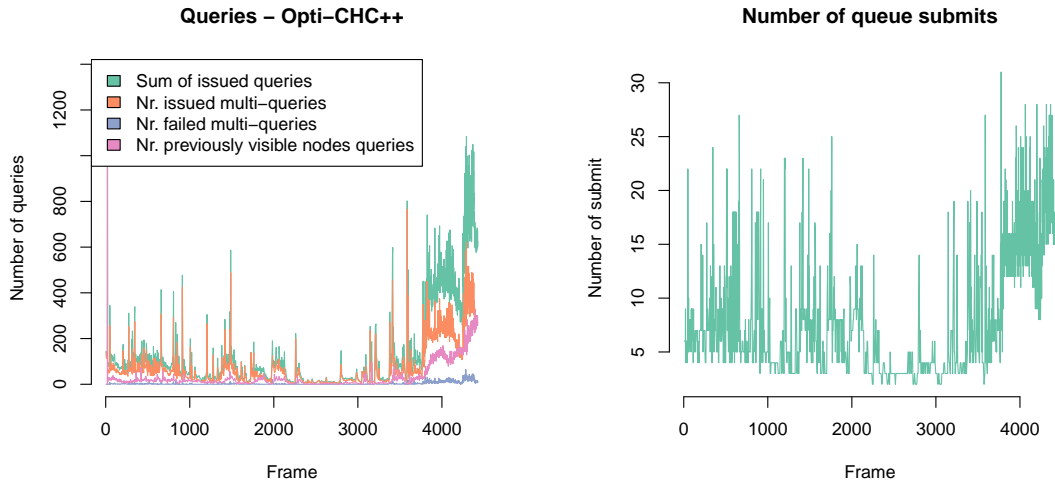


Figure 5.6: CPU frame timing of Opti-CHC++ in the Neu Rungholt scene. Shows that most time is spent on waiting on the completion of occlusion queries and that the queue submission time has been brought to a minimum.

Even though Opti-CHC++ does not achieve the high performance increase over VFC as the original implementation, it still provides a nearly optimal reduction in rendered triangles with an up to 2.5 times performance increase over VFC in high occlusion scenes.



(a) Shows the number of queries that vary widely with the amount of visible geometry. (b) Shows the number of queue submits that increase with the number of queries.

Figure 5.7: Shows the number of queries and queue submits of the Rungholt walkthrough.

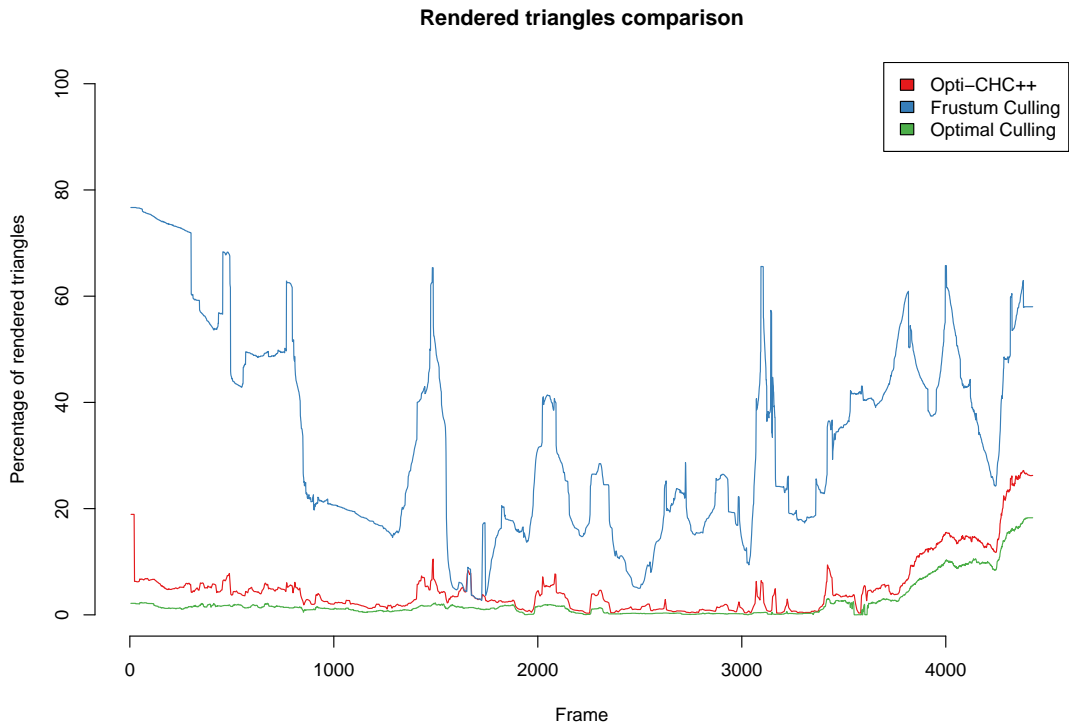


Figure 5.8: Number of rendered triangles in the Neu Rungholt scene.

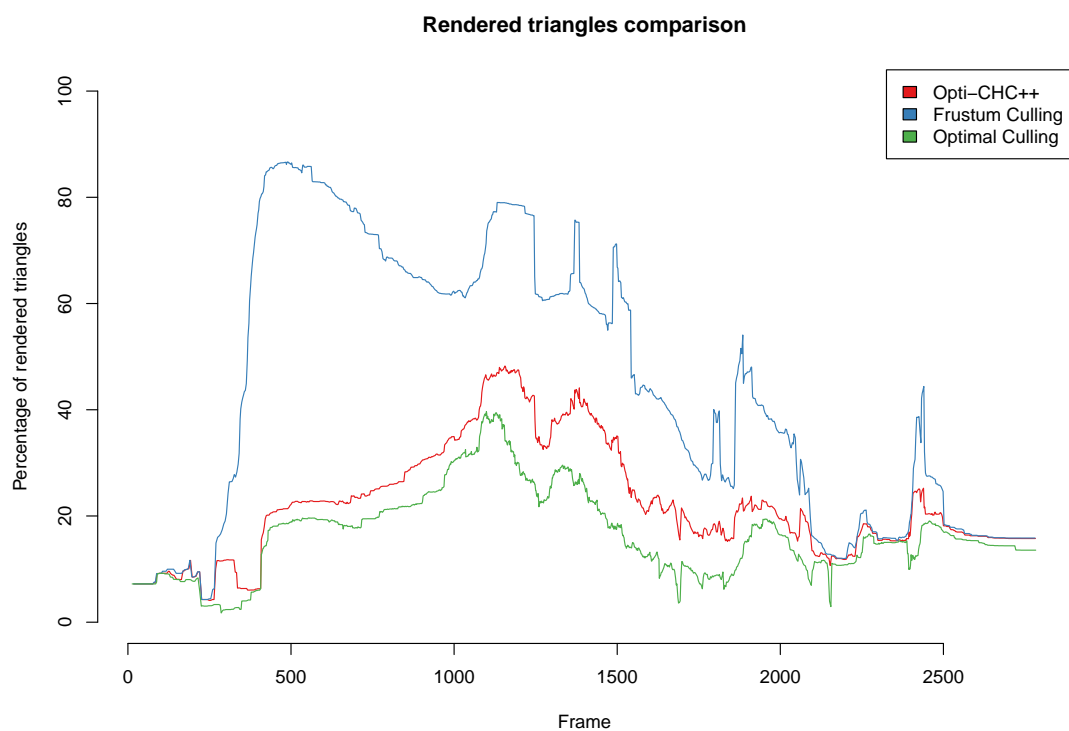


Figure 5.9: Number of rendered triangles in the Lumberyard Bistro scene.

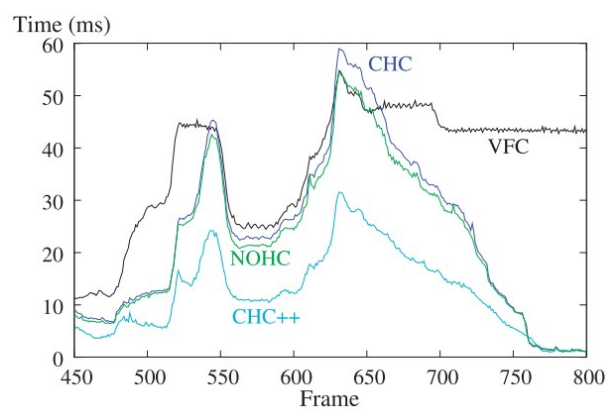


Figure 5.10: Frame timing of the original CHC++ implementation. Reprinted from [MBW08].

Conclusion

6.1 Summary

This thesis provides an adapted implementation of CHC++ that efficiently runs on Vulkan and modern hardware. This could be achieved by batching draw calls and subsequent occlusion queries into single command buffer submits, which reduces the amount of overhead. An additional queue—the c-queue—was introduced to store the nodes of failed multiqueries for later execution. The occlusion queries of these nodes get issued in the same submit as other queries to further reduce the number of command buffer submissions. This allows the GPU to perform a chunk of work while the CPU can traverse the rest of the BVH tree and generate the next command buffer. Proper synchronisation also allows for multiple command buffers to be in execution at the same time. Additionally, proper transparency support was added by storing alpha-masked and alpha-blended nodes into separate render queues that are rendered backwards in a back-to-front order at the end of traversal for correct blending. This also allows for correct culling of transparent geometry by not considering them as occluders but still culling them if occluded. Although the high performance difference between CHC++ and VFC of the original implementation could not be replicated, the algorithm still provides great performance enhancement in scenes with high occlusion and does not degenerate in worst-case scenarios. Our optimised version of CHC++ performed up to 2.5 times better than standard VFC on average in the Neu Rungholt Minecraft map test scene and was as fast as VFC in the Lumberyard Bistro test scene with low occlusion.

6.2 Outlook

Although Opti-CHC++ has been greatly optimised to run on Vulkan and on modern hardware, there are still some things to consider to potentially further improve its

effectiveness. Some of these possible improvement ideas are elaborated in the following subsections.

6.2.1 Multithreading

One of the main characteristics of the algorithm is that it is single-threaded. Therefore, making the algorithm multithreaded could be beneficial. But doing so might prove difficult or even impossible, as it relies on the BVH being traversed breadth-first. So if the BVH tree is just split into subtrees that are traversed on a separate thread, separate render targets for each would also be needed. These would need to be combined in the end somehow. Note that this will result in non-optimal culling as occluders in one rendertarget could occlude objects in another and vice-versa.

A more optimal approach would be to split the viewport into multiple pieces and perform some form of tiled rendering where each tile only considers BVH nodes inside its frustum. In this case, nodes that intersect the inner frustum planes would need to be considered specially by either traversing them multiple times or once at the end of the algorithm. Whether this is a feasible approach and would actually yield more performance would need to be evaluated in more detail.

6.2.2 Make algorithm GPU resident

Another possible improvement could be putting the whole algorithm on the GPU. This entails putting the entire BVH in GPU memory and managing it there. The algorithm would need to run in a compute shader. Here arises the first problem as you would need to perform draw calls and occlusion queries from within the compute shader, which is not possible at the moment. Otherwise a CPU roundtrip is still needed to make the draw and query calls and then compute shader needs to be restarted if it has not finished with the traversal. This reintroduces the need to stall the CPU or perform other work in the meantime. But if this is done in conjunction with indirect drawing, the command buffer can be reused, keeping the CPU load to a minimum. Since the algorithm is already GPU bound, this could also hinder performance instead of increasing it. On the other hand lot of CPU resources will be freed that can be utilised for other tasks.

6.2.3 Replace occlusion queries with another method

The biggest problem with occlusion queries is the high latency between issuing the query and when its results are ready. Therefore, one could try using a different method to detect occlusion like Hi-Z culling. Then a Hi-Z depth buffer needs to be created each time a switch from rendering to queries happens and then the culling needs to be performed in a compute shader. But the Hi-Z generation could be too much additional overhead, as this could happen quite often in case of CHC++. Because of that, changes to the algorithm are likely needed to reduce the amount of Hi-Z recalculations. One could for example postpone the culling until every visible object has been drawn, then only one culling pass is needed for the previously-invisible and v-queue nodes. Depending on how

fast the Hi-Z culling pass is, this could either improve or decrease performance overall, as it still needs to be waited for its results to perform additional draws. Additionally, this could make it more feasibly to put the algorithm on the GPU, as occlusion queries do not need to be issued from the GPU and MDI can be used to eliminate the need for a CPU stall.

List of Figures

1.1	1.1a shows all geometry. 1.1b shows only the geometry that is in the frustum. 1.1c shows only visible geometry.	2
2.1	The different view frustums used by portal culling are shown.	6
2.2	A visualisation of an occlusion query in the Lumberyard Bistro scene. . .	8
2.3	The different queues of CHC++ and how they work together. Reprinted from [MBW08]	9
2.4	Three levels of the generated Hi-Z depth map used for Hi-Z culling. Reprinted from [Dan10].	11
2.5	Shows the pixels of the lookup MIP level of the Hi-Z hierarchy.	11
2.6	The voxelisation of the occluder geometry used in Umbra 3. Reprinted from [3D18]	16
2.7	The optimised cells generated in the Umbra 3 pre-process. Reprinted from [3D18]	17
5.1	CPU Frame time comparison of Opti-CHC++ vs. Unoptimised-CHC++ vs. Frustum Culling vs. No-Culling in the Neu Rungholt walkthrough.	30
5.2	CPU Frame time comparison of Opti-CHC++ vs. Frustum Culling vs. No-Culling in the Lumberyard Bistro walkthrough.	31
5.3	Correct transparency rendering in Opti-CHC++ is shown.	32
5.4	GPU Frame timing of the Neu Rungholt walkthrough. Shows that most of the time is used by rendering and overhead when a lot of submits happen.	33
5.5	Screenshot of Nvidia Nsight System profiler of a frame of the Rungholt scene.	34
5.6	CPU frame timing of Opti-CHC++ in the Neu Rungholt scene.	35
5.7	Shows the number of queries and queue submits of the Rungholt walkthrough.	36
5.8	Number of rendered triangles in the Neu Rungholt scene.	36
5.9	Number of rendered triangles in the Lumberyard Bistro scene.	37
5.10	Frame timing of the original CHC++ implementation. Reprinted from [MBW08].	37

List of Algorithms

4.1	Opti-CHC++ — Traversal	25
4.2	Opti-CHC++ — Functions 1	26
4.3	Opti-CHC++ — Functions 2	28

Acronyms

- AABB** axis-aligned bounding box. 7, 11–14
- API** Application Programming Interface. xi, xiii, 1–3, 9–12, 19, 20, 29, 34
- BVH** Bounding Volume Hierarchy. 9, 10, 21, 27, 39, 40
- CHC** Coherent Hierarchical Culling. 5, 7, 9, 10, 22
- CHC++** Coherent Hierarchical Culling Revisited. xi, xiii, 1, 3, 5, 9, 19–24, 29–34, 37, 39, 40, 43
- FPS** frames per second. 2, 30, 31
- GDC** Game Developer Conference. 12, 14
- Hi-Z** Hierarchical-Z. 2, 5, 10–14, 34, 40, 41, 43
- MDI** multi draw indirect. 12, 14, 20, 34, 41
- NVCmdLst** Nvidia command list. 12
- PSO** Pipeline State Object. 20
- SAH-BVH** Surface Area Heuristics - Bounding Volume Hierarchy. 23
- SIMD** Single Instruction, Multiple Data. 12, 13
- UE4** Unreal Engine 4. 13, 34
- VFC** View-Frustum Culling. 1, 8, 10, 29–35, 39

Bibliography

- [3D18] Umbra 3D. Introduction to Occlusion Culling. Available at <https://medium.com/@Umbra3D/introduction-to-occlusion-culling-3d6cfb195c79>, 2018. Accessed 2020-01-03.
- [AM18] Tomas Akenine-Möller. *Real-Time Rendering*. Crc Press, 2018.
- [BMW09] Jiří Bittner, Oliver Mattausch, and Michael Wimmer. Game-engine-friendly occlusion culling. *SHADERX7: Advanced Rendering Techniques*, W. Engel, Ed, 7:637–653, 2009.
- [BS] Jasin Bushnaief and Umbra Software. Solving Visibility and Streaming in The Witcher 3 : Wild Hunt with Umbra 3.
- [BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. In *Computer Graphics Forum*, volume 23, pages 615–624. Wiley Online Library, 2004.
- [CGJT14] Everitt Cass, Sellers Graham, McDonald John, and Foley Tim. Approaching zero driver overhead, 2014.
- [Dan10] Rákos Daniel. Hierarchical-Z map based occlusion culling. Available at <http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/>, 2010. Accessed 2019-11-19.
- [Gam] Moby Games. Middleware: Umbra 3. Available at <https://www.mobygames.com/game-group/middleware-umbra-3>. Accessed 2020-01-02.
- [Gam20] EPIC Games. Visibility and Occlusion Culling. Available at <https://docs.unrealengine.com/en-US/Engine/Rendering/VisibilityCulling/index.html>, 2020. Accessed 2020-01-03.

- [GBK06] M. Guthe, A. Balázs, and R. Klein. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. *Eurographics Symposium on Rendering 2006*, 2006.
- [GKM93] Ned Greene, Michael Kassy, and Gavin Millery. Hierarchical Z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1993*, 1993.
- [Gra16] Wihlidal Graham. Optimizing the Graphics Pipeline with Compute. Available at https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf, 2016.
- [HAAM16] Jon Hasselgren, Magnus Andersson, and Tomas Akenine-Möller. Masked software occlusion culling. In *High Performance Graphics*, pages 23–31, 2016.
- [Hou13] Kristyna Hougaard. Occlusion Culling in Unity 4.3: The Basics. Available at <https://blogs.unity3d.com/2013/12/02/occlusion-culling-in-unity-4-3-the-basics/>, 2013. Accessed 2020-07-10.
- [Int13] Kuah Kiefer Intel. Software Occlusion Culling. Available at <https://software.intel.com/en-us/articles/software-occlusion-culling>, 2013. Accessed 2020-01-03.
- [Kot17] Anagnostou Kotas. EXPERIMENTS IN GPU-BASED OCCLUSION CULLING. Available at <https://interplayoflight.wordpress.com/2017/11/15/experiments-in-gpu-based-occlusion-culling/>, 2017. Accessed 2019-11-19.
- [Lum17] Amazon Lumberyard. Amazon Lumberyard Bistro, Open Research Content Archive (ORCA), 2017.
- [MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. CHC++: Coherent hierarchical culling revisited. In *Computer Graphics Forum*, 2008.
- [Nic10] Darnell Nick. Hierarchical Z-Buffer Occlusion Culling. Available at <https://www.nickdarnell.com/hierarchical-z-buffer-occlusion-culling/>, 2010. Accessed 2019-11-19.
- [Nvi20] Nvidia. gl occlusion culling. Available at https://github.com/nvpro-samples/gl_occlusion_culling, 2020. Accessed 2020-07-31.
- [Sco15] Fitzgerald Scott. Occlusion - How to Prepare a Level. Available at <https://docs.cryengine.com/display/SDKDOC2/Occlusion+-+How+to+Prepare+a+Level>, 2015. Accessed 2020-01-03.

- [SD11] Hill Stephen and Collin Daniel. Practical, Dynamic Visibility for Games. Available at <https://blog.selfshadow.com/publications/practical-visibility/>, 2011. Accessed 2019-11-19.
- [Uni19] Unity. Occlusion Culling. Available at <https://docs.unity3d.com/Manual/OcclusionCulling.html>, 2019. Accessed 2020-01-03.
- [US15] Haar Ulrich and Aaltonen Sebastian. GPU-Driven Rendering Pipelines. Available at http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf, 2015. Accessed 2019-11-19.