TU WIEN Informatics

Frequenzanalyse von Verdeckern für die Bestimmung des Sichtbarkeitsgrades von teilweise verdeckten Objekten

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Elias Kristmann
Matrikelnummer 01518693

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Wien, 15. Juli 2022

Elias Kristmann	Michael Wimmer

TU WIEN Informatics

Occluder Frequency Analysis for Evaluating the Level of Visibility of Partly Occluded Objects

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Elias Kristmann
Registration Number 01518693

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Vienna, 15th July, 2022

Elias Kristmann	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Elias Kristmann

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, 15. Juli 2022

Elias Kristmann

Danksagung

Zuallererst möchte ich mich bei meinem Betreuer Bernhard Kerbel und verantwortlichen Professor Michael Wimmer für die Geduld und die Unterstützung während des Entstehungsprozesses meiner Bachelorarbeit bedanken.
Außerdem bedanke ich mich bei meiner Familie und meiner Freundin für die stätige Unterstützung während des gesamten Studiums.

Acknowledgements

First of all, I would like to thank my supervisor Bernhard Kerbel and the responsible professor Michael Wimmer for their patience and support during the process of writing my Bachelor's thesis.
I would also like to thank my family and my girlfriend for their constant support throughout my studies.

Kurzfassung

Um die Rendering-Effizienz von großen und komplexen Szenen zu verbessern, erkennen Verdeckungsalgorithmen Objekte, die von anderen vollständig verdeckt sind und daher nicht gerendert werden müssen (occlusion culling). Diese Methoden verfolgen allerdings oft ein Alles-oder-nichts-Prinzip, da sie die Geometrie entweder vollständig entfernen oder diese in voller Ausführung zeichnen. Bei dieser Vorgehensweise wird eine wichtige Unterkategorie des Sichtbarkeitsproblems nicht berücksichtigt: das Erkennen von Objekten, welche kaum sichtbar sind, da sie von anderen teilweise verdeckt werden und daher weniger detailliert gerendert werden können, ohne wahrnehmbare optische Fehler zu erzeugen. In dieser Arbeit wird der Grad der Sichtbarkeit solcher Objekte bestimmt, indem eine hierarchische Verdeckungskarte berechnet und ihre Struktur anhand der Frequenz der Verdecker analysiert wird. Mit Hilfe dieser Analyse kann ein Parameter berechnet werden , welcher dann den Detailgrad (level of detail (LOD)) dieser Objekte bestimmt. Die in dieser Arbeit angewandte Methode erzielt gute Resultate, selbst in Szenen, wo nur spärliche und räumlich weit verteilte Verdecker vorhanden sind. Sie hat außerdem noch viel Potential für weitere Verbesserungen.

Abstract

To increase rendering efficiency of large and complex scenes, occlusion culling algorithms detect objects which are completely hidden by others and therefore do not need to be rendered. However, these methods often follow an all-or-nothing principle, either culling the geometry entirely or drawing it at full detail. This approach disregards an important subcategory of the visibility problem: detecting objects that are hardly visible because they are partly occluded and which can therefore be rendered at a lower level of detail without generating noticeable artifacts. In this thesis we assess the level of visibility of such objects by computing a hierarchical occlusion map and analysing its structure based on the frequencies of the occluders. This analysis results in a parameter that controls the level of detail (LOD) in which the geometry is rendered. The algorithm performs well even in scenes with sparse occlusion, surpassing the standard hierarchical occlusion map algorithm, with still a lot of potential for even further improvements.

Contents

CHAPTER 1

Introduction

Modern large-scale applications such as video games or walkthrough programs need to render complex scenes consisting of millions of polygons in real-time. Naively drawing every object is impossible because of the immense number of polygons that need to be processed every frame. Therefore, methods to reduce the scenes complexity were developed over the last two decades.
One of the most common ideas is to remove primitives that have no influence on the final image, because they are invisible. The process of finding and removing them is called visibility culling and is one of the oldest problems in the field of computer graphics. In a way, to computer graphics engineers who want to achieve peak performance in their applications, invisible triangles have actually become much more important than visible ones. There are three different approaches that are often combined to achieve optimal results:

- Back-face culling
- Frustum culling
- Occlusion culling

Back-face culling describes the process of identifying and removing polygons that are facing away from the viewpoint. It is easy to implement, integrated into hardware and often already cuts the number of polygons in half.
The second method is called view frustum or simply frustum culling. The view frustum is the part of a scene that is situated within the current field of view of the observing camera. Consequently, frustum culling is removing objects which are entirely outside of the view frustum.
Finally, occlusion culling is the method of removing objects from the rendering pipeline that are completely hidden behind other objects and are therefore not visible [PT02].

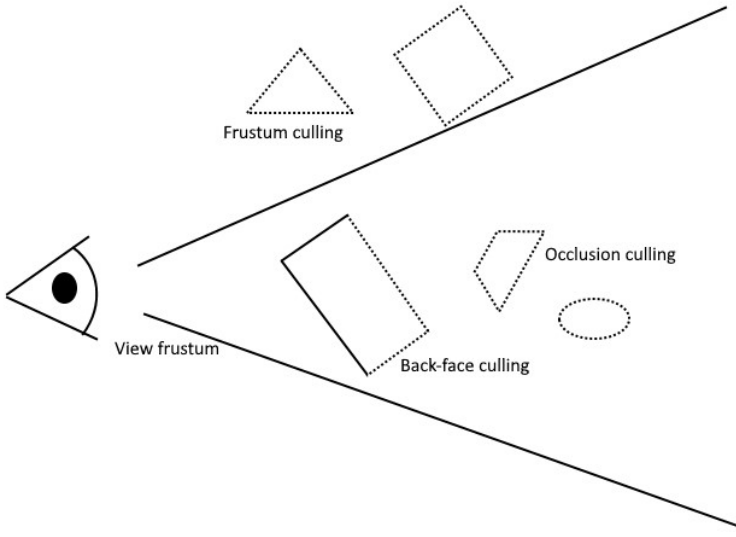

Figure 1.1: Examples of frustum, back-face and occlusion culling. The dotted lines are the geometry that is removed.

Figure 1.1 illustrates the three different culling techniques.

Back-face and frustum culling are trivial, and there exist straightforward implementations for both. On the contrary, occlusion culling in 3D scenes is extremely complex. There is extensive literature on different occlusion culling methods, each coming with their own benefits and disadvantages. However, those methods mostly focus on objects either being completely visible or completely occluded. In the worst case, regarding the rendering efficiency, this leads to an object being drawn at full resolution even if it is only contributing a few visible pixels to the final image. This all-or-nothing approach ignores an important subcategory of the visibility problem which can further improve rendering efficiency: detecting objects that are only partly occluded and which can therefore be rendered at a lower level of detail, without generating any noticeable errors. An example for such a situation can be seen in Figure 1.2, where the leaves and branches of a tree are covering the facades of buildings behind it. Above the first floor, only small regions of the facades are visible, and it is difficult to distinguish any details.

In this thesis we expand on the hierarchical occlusion map culling algorithm by Zhang et al. [ZMHH97]. First, we utilise the presented occlusion culling method to identify all partly occluded objects. In the next step, we introduce a novel concept of an occluder frequency analysis to estimate the level of visibility of the occluded objects. To do so, we utilise the discrete cosine transformation. Finally, we use the results of this transformation to control the level of geometric complexity in which each object is drawn. Our method is an efficient occlusion culling algorithm that can render large and complex scenes in real-time, even if only sparse occlusion is present.

In the following section we will give an overview of the most important occlusion culling methods as well as techniques that reduce the geometric complexity of models. We

Figure 1.2: Real-life example of hardly visible objects: the facades of the buildings are visible through multiple holes in the foliage of the tree and can therefore not be culled completely. However, hardly any details of the facades are perceptible.

then introduce the discrete cosine transformation and explain how we can use it to transform images to the frequency spectrum. Subsequently, we detail our method and implementation of the proposed occlusion algorithm and discuss our results when applying our method to two large and complex scenes. Finally, we consider open problems and possible improvements of our technique.

CHAPTER 2

Related Work

In this chapter we will first present key strategies of occlusion culling algorithms and how they attempt to address the visibility problem. Afterwards, we discuss methods that try to reduce the geometric complexity of a scene, with particular attention to the conditions under which they are applicable. Finally, we introduce the discrete cosine transformation which is fundamental to our proposed solution.

2.1 Visibility culling

In literature, visibility culling algorithms are generally classified into two main categories: from-point and from-region visibility. Whilst having many analogies considering principle concepts, they optimize the visibility calculation for different constraints [PT02].
In the following, we will first detail the differences of those two approaches. Subsequently, we will then concentrate on the most influential from-point algorithms that use hierarchical structures to increase their efficiency.

2.1.1 From-point visibility

From-point visibility algorithms calculate all visible objects for the exact location and perspective of the camera. Therefore, these computations are precise for the current position, but need to be updated whenever the perspective of the camera changes even slightly. Storing this visibility information for every possible location in a scene is impracticable. As a result, the current viewpoint is calculated at run-time, meaning that these methods are applied online.
This has the advantage of requiring minimal to no preprocessing and being more flexible to use, but at the cost of additional computations during execution. If implemented naively, the per-frame costs of the visibility calculations can outweigh the benefits and

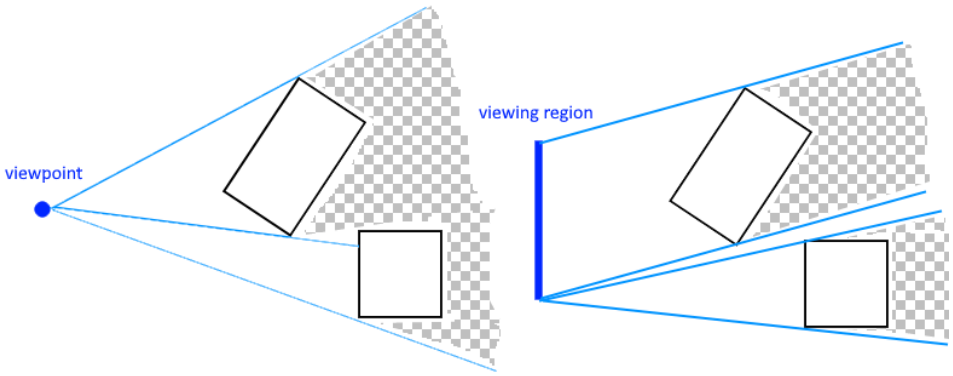

Figure 2.1: Left: from-point visibility. Right: from-region visibility. The light blue lines show the border between visible (everything in white) and invisible (checkerboard pattern) parts of the scene. Since from-region visibility must take multiple viewpoints into account, fewer parts of the scene can be determined as completely occluded. In this example, there is a small region between the boxes that is only visible from a small part of the viewing region but therefore is considered to be visible for the entire region.

actually reduce efficiency. Nevertheless, one advantage of from-point visibility methods is, that hardly any storage costs have to be accounted for, as most results from visibility computation do not need to be stored or can be discarded shortly after the viewpoint changes [Mat10, AMHH+18].

2.1.2 From-region visibility

From-region methods subdivide the scene into a spatial data structure with regions that have similar visibility characteristics. For each of those regions a potentially visible set (PVS) of objects is calculated and stored at a preprocessing step. At run-time, the PVS at the position of the observer is retrieved. The advantage of such methods is the minimal computational cost once the potentially visible set is created. However, visibility calculations for regions are generally more complex and less precise for an individual viewpoint, an example can be seen in Figure 2.1. Furthermore, since the visibility is precalculated, the application is limited to static scenes. In addition, the storage costs of the PVS for large scenes can be vast if not dealt with explicitly [Mat10, AMHH+18].

2.1.3 Hierarchical occlusion culling

In computer graphics, hierarchical structures are often used to combine multiple similar problems so that they can then be solved with a divide-and-conquer approach, allowing

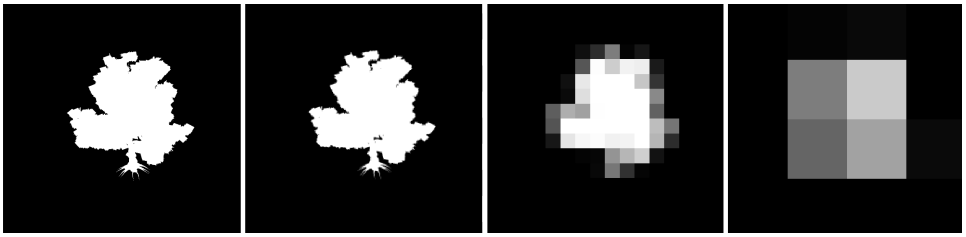

Figure 2.2: Different level of an occlusion map picturing a chestnut-tree [McG17]. The levels of the occlusion map are ascending from left to right .

us to compute numerous small tasks efficiently. In this section we will detail occlusion culling algorithms which use hierarchical data structures to improve the performance of visibility calculations.
In the paper [ZMHH97] Zhang et al. describe an algorithm for a software-based hierarchical occlusion map test. In the first step, a set of potential occluders has to be selected, which are then rendered to a texture with a resolution equal to a fourth of the screen dimension. This forms the lowest level of the occlusion map, also called level 0. The background is drawn in black with an opacity value of 0, while the occluders are drawn completely white with an opacity value of 1. In the next step, this texture is down-sampled, halving the dimensions at each level until the size of the final level is reduced to one 4×4 block. To do so, Zhang et al. propose to average 2×2 blocks of pixels of the higher occlusion map level into a single pixel of the lower level. Figure 2.2 shows an example for an occlusion map construction with a tree object as an occluder. This occlusion map is then used to calculate visibility in two steps, an overlap test and a depth comparison.

For the former, Zhang et al. calculate a bounding box of an object they want to test visibility for. This bounding box is then transformed into screen-space, after which an axis-aligned bounding rectangle (AABR) is calculated containing the entire object, as can be seen in Figure 2.3. They then choose a level of the occlusion map, so that the AABR is roughly the same size as a single pixel of the map. This has the advantage that an occludee's AABR covers at most four pixels of the occlusion map. They then check the opacity value at each of those pixels that overlap with the rectangle. If all opacity values are completely white (opacity value of 1), we know that the object is overlapped by the occluders. If not, some of the pixels in a lower level of the occlusion map are not completely opaque, and therefore the object cannot be entirely occluded and it has to be rendered.
Since the average operation is used at the construction of the occlusion map, it is possible to retrieve the number of pixels that are not covered by occluders with the opacity value. This provides a method to perform so-called approximate visibility culling: we can define a threshold of how many pixels of an occludee are allowed to be visible for it to still be considered occluded. In scenes with sparse occlusion this can improve the effectiveness of the algorithm. However, the occlusion culling is no longer conservative and might introduce noticeable artifacts.
In the next step, we have to compare the depth of the object to the occluders: if it is

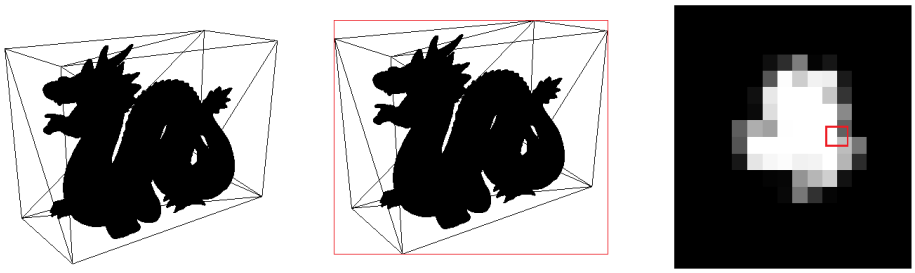

Figure 2.3: Left: Stanford dragon model [Sta] with bounding box. Middle: screen-space axis-aligned bounding rectangle in red. Right: overlap test of the red AABR against the occlusion map. The level of the occlusion map is chosen so that the AABR covers at most four pixels.

behind the occluders, we can cull it, otherwise it has to be rendered.
For the depth comparison Zhang et al. describe a depth estimation buffer that works like a software z-buffer. Nowadays, it has become standard for graphic cards to include a hardware z-buffer, making the method outdated. To check if an overlapped object is also behind the occluders, we can simply compare the minimum depth value of the axis-aligned bounding rectangle to the depth of all pixels of the occluders that are overlapped by it. If the depth is smaller than those of the occluders, we know that the object is invisible.

There are a few similar methods to hierarchical occlusion maps: Greene presents in [Gre95] and [GKM93] occlusion culling with a hierarchical Z test. In this method, a hierarchical Z pyramid is constructed instead of an occlusion map and the scene is spatially structured in an octree. The octree is constructed by first encapsulating the entire model in a cube. If the polygon count inside this cube is higher than a certain threshold, the cube is subdivided into eight smaller cubes. As long as the threshold is not reached, this subdivision repeats itself recursively. For creating the hierarchical Z pyramid, the concept of rendering the occluders and downsampling is quite similar to the hierarchical occlusion maps. However, the initial values that are drawn to the texture are the outputs of the z-buffer, which corresponds to the depth information of the objects. Additionally, when sampling 4×4 blocks, the lowest value is chosen at each step, instead of interpolating between them. This allows conservative overlap tests and removes the need of a depth test, but loses the ability to perform approximate visibility culling. Instead of bounding boxes, the algorithm checks the cubes of the precomputed octree to calculate visibility. If a cube is completely occluded, all the geometry that is inside it is occluded as well. This has the advantage, that children of a tree node can be culled if the parent is detected as being occluded.
Hardware occlusion queries were introduced in the late 90s and are still prominent techniques used in game and rendering engines. They too share many conceptual similarities with occlusion map and hi-Z test methods: In the first step, designated occluders are rendered to the depth buffer. But instead of sub-sampling as in the previous

methods, every pixel contained within the axis-aligned bounding rectangle of a potential visible object is tested against the full resolution. These so-called queries are processed in parallel on the GPU to efficiently determine the visibility. The issue of this technique comes from the communication between the CPU and GPU: the query is issued by the CPU after which it has to wait for the result, staling the rendering pipeline until the GPU has finished calculating the query. At the same time, the GPU could run out of queries to process, meaning that it has to wait for a new CPU input. Factoring in that communication between CPU and GPU is expensive, a naive implementation could even reduce the efficiency of an application. To still maintain real-time performance, means of reducing the number of queries as well as decoupling the CPU and GPU from each other as much as possible are required [Per20, BWPP04].
Bittner et al. propose an algorithm that utilises a spatial data structure in the form of a kd-tree and the property of visibility coherence to address those issues. Additionally, while the CPU waits for the queries to be executed, it begins to render objects that were visible in the previous frame until the queries results are available. In the worst case, if all objects that were started to be rendered are invisible, no performance was lost since the CPU was waiting for the GPU anyway. However, every object that was already rendered and that is still detected to be visible is already drawn, thus using the waiting time effectively. In scenes with reasonably temporal coherence, the number of objects that are rendered in vain is relatively small. Nonetheless, this optimization should be considered with caution since in scenes with relatively small but complex geometry it can yield a negative influence on performance. In their paper Bittner also briefly mentions approximate visibility culling: Since occlusion queries are able to determine the exact number of pixels that are visible, the above mentioned method of defining a threshold of how many pixels are allowed to be visible is applicable [BWPP04].
In a follow-up paper this algorithm was refined, improving the coherence between the GPU and CPU working together as well as further reducing the number of queries that have to be issued thanks to more precise bounding volumes [MBW08].

2.2 Polygon reduction methods

A different strategy to improve rendering performance is to reduce the geometrical complexity (number of polygons) of objects in the scene and replace them with simpler approximations. However, this comes at the cost of reducing the visual quality and potentially introducing optical errors to the final image. For this reason, we only want to reduce the geometric complexity for objects where the change is hardly noticeable.
In this section we will give an overview of the three most common polygon reduction methods: image-based impostors, level of detail (LOD) and real-time tessellation. Further, we pay special attention to how these methods decide when to lower the visual quality of an object.

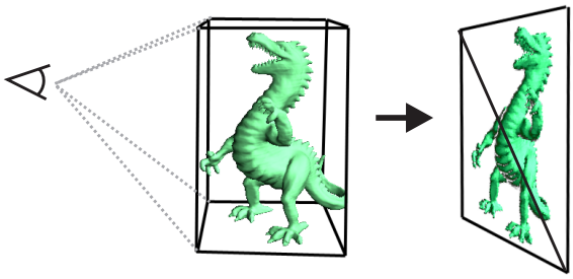

Figure 2.4: Example of a 3D dinosaur object and its planar impostor. Reprinted from [Jes05].

2.2.1 Impostors

There are a few different definitions in the field of computer graphics for impostors and when an object is considered to be one. For this thesis, we will go by the definition from Maciel in their paper [MS95]: "Impostors are image-based entities used as alternative representations for 3D scene parts for accelerating their rendering process." In this context, the term "image-based" describes methods that use a set of images projected on 2D geometry to represent all or some parts of a scene. Figure 2.4 shows an example for an image-based entity.
To generate an impostor, an object usually has to be rendered at least once. The resulting image as well as some basic geometric information is then stored and used to replace the object with its 2D representation with as few artifacts as possible. Special attention should be paid to the following to avoid potential errors in the final image:

- If the required output resolution is higher than the resolution of the impostor, the image-based representation needs to be extrapolated to fill the required space. This can lead to a pixelated appearance and loss of details. To prevent this, the resolution of an impostor should boast similar quality to the image output of the object it replaces.
- With a change of the viewpoint, far away objects seem to move slower than closer objects. This phenomenon is called the parallax effect and can make previously occluded parts of an object visible. Since impostor are only 2D representations without depth, their appearance is unchanged by this phenomenon compared to the original geometric model, resulting in visual artifacts. One way to reduce this kind of error is to generate impostors which approximate the three dimensional geometry. However, this greatly increases the complexity and the benefits are strongly dependent on the structure of the models [SDB97].
- In dynamic scenes, depth information of the original object has to be stored to be able to calculate the visibility for impostors. Moreover, for simple 2D representations

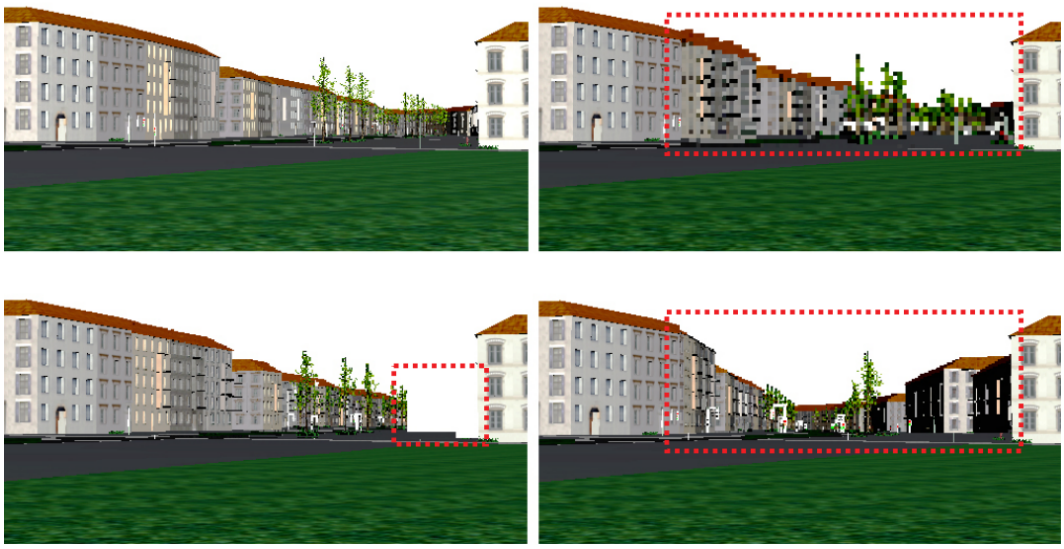

Figure 2.5: Possible image artifacts caused by 2D impostors. Top left: ground truth. Top right: low sampling resolution causing pixelation. Bottom left: impostor generation from different viewpoint causes perspective errors. Bottom right: parallax errors yield wrong perspective and scene integration. Reprinted from [Jes05].

> it is impossible to display visibility correctly if one part of an impostor is in front of an object and another part is occluded by that same object. One solution to this is to store the depth information for every pixel of the impostor so that a z-test is possible [Sch97].

Examples for these artifacts were well illustrated by Jeschke et al. and can be seen in Figure 2.5 [Jes05, JWP05].

The impostor generation can be dynamic at run-time or static in a preprocessing step. The former has the advantage of requiring less storage, being generally easier to implement, as well as allowing for a more flexible and dynamic scene. The trade-offs are additional computations during the execution of the application, as the impostor creation requires the rendering of the original object. As a consequence, to reduce the number of impostor generations, most methods allow impostors to diverge slightly from the appearance of the object they are replacing, providing that the difference contains only small and acceptable visual errors. Therefore, efficiency gains are tightly coupled to the extent to which the temporal coherence of a scene can be exploited to reduce the rate at which impostors need to be replaced [Jes05, JWP05].
As long as an impostor does not need to be updated, it is called valid. The factors that influence validity of an impostor are dependent on the particular method that is used. This threshold can be seen as an upper bound for the error that is introduced in the final

image when replacing the original object. The regions in which an impostor is valid is called view cell and most commonly depends on the distance (impostors should not be too close to the viewpoint) and the change of viewing angle between the observer and the impostor (because of parallax errors). For these reasons, the view cell region is often cone shaped.
The efficiency gains expected from using impostors depends on the soundness of the selection of which geometric objects should be replaced by their image-based counterparts. Besides the consideration of image quality, it is a trade-off between rendering speed and memory requirements.
One of the simplest approaches is to replace all objects that are far away, since they mostly occupy only a few pixels of the final image which makes potential artifacts less noticeable. Furthermore, distant objects need to be updated less often because of the parallax effect described earlier. Methods that expand on these observations are detailed in [AL99] and [WM03].
Jeschke et al. propose in [JWSP05] an algorithm which combines multiple objects into a single impostor to reduce their size as well as guaranteeing a desired frame rate. Among other improvements, this is achieved by applying occlusion culling to reduce the number of objects before placing the impostor dynamically. However, occlusion culling is only used to completely remove objects that are invisible, ignoring the subset of hardly visible objects. Without factoring in the screen-space contribution, objects that are partly occluded, with only a few pixels visible in the final image, are still rendered as expensive geometrical models.
So-called perceptually-based rendering techniques expand on the fact that the visual system of humans is not perfect, allowing methods to deliberately trick our perception to achieve more efficient rendering algorithms. There has been some research regarding the perception of partly occluded objects that are in motion. It was shown that the movement of grids or patterns is difficult to notice when only visible through small holes in an occluder [AM82]. In their paper [SLCO+04], Sayer et al. build on this observation and propose to replace partly occluded objects with impostors, since the potential parallax error is less likely to be noticeable.

2.2.2 Level of detail

The time it takes to render an object depends strongly on how many polygons it contains. As a consequence, reducing the number of polygons can greatly speedup the rendering process. Level of detail (LOD) methods store for each object multiple models with different number of polygons, an example can be seen in Figure 2.6. These LOD are generated either in a preprocessing step, offline or are designed manually in advance. At run-time, for each visible object its LOD model, that is detailed sufficiently enough to not produce any visual errors, is rendered. The selection criteria for which LOD to choose are typically the same as with impostor rendering: the distance between the viewpoint and the object as well as the screen-size [HD04]. It is important to note that the parallax error is less of an issue for LOD compared to images based representations, since even at

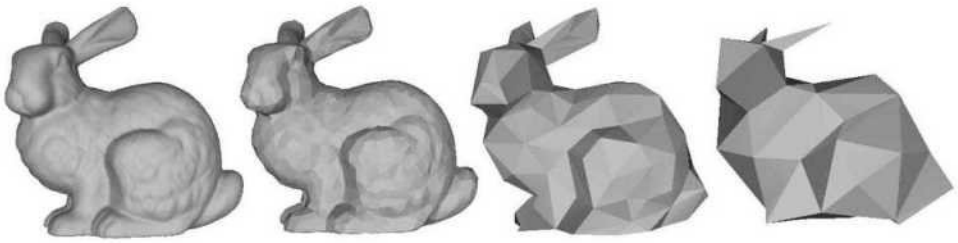

Figure 2.6: Different levels of detail of the Stanford bunny [Sta]. From left to right: 69451, 2502, 251, and 76 polygons. The models with fewer polygons are faster to render, but can cause a visible error in the final image.

the lowest LOD the object is usually still three dimensional.
Although under-researched, there has been exploration regarding object visibility as an supplementary input for LOD selection. The idea is, that if only a few pixels of an object are visible, most details are occluded anyway. Especially in wide open scenes with sparse occlusion this can increase the effectiveness of the overall LOD methods.
The term hardly visible set (HVS) was first introduced in [ASVNB00] by Andújar et al. for describing objects which are only partly occluded. In their method, this set is computed in a preprocessing step: First, the visibility with exact occluders is calculated. Afterwards the scene is rendered a second time, but with enlarged occluders. Objects that were visible at the first rendering step but are occluded at the second pass form the HVS. Andújar et al. propose to group the objects contained in this set on how many pixels they are contributing to the final image. If the contribution is below a user-specified threshold, the object can be culled or rendered with a low LOD.
In [ESSS01], Birkholz expand on this idea with the addition of view-dependent LOD. This technique allows to render different parts of a model at different LOD. To do so, usually a very low-poly model is rendered first before refining it continuously at each step. Whilst this method is not new in itself, the combination with visibility culling results is a novel approach. In their application, Birkholz first renders the lowest detailed version of the occluders to the z-buffer. Then, for each occludee the low-poly model is rendered. At each refinement step the visibility of the geometry that would be added is tested against the previously generated z-buffer with the help of occlusion queries. If a refinement is determined to be visible, it is executed, otherwise it is ignored. As a result, only the visible parts of an object are becoming more and more exact and geometrically complex while the occluded parts remain simple approximations. However, this process is inefficient and increases rendering time, since it was designed to optimally render a scene using a fixed number of polygons and not as an improvement to efficiency.
As with impostors, there have been some attempts that exploit the shortcomings of our visual system for an increase in rendering performance by decreasing the LOD of hardly visible objects. In [DBD+07], Drettakis et al. present a perceptual rendering pipeline. In their method, they first divide the scene into planes which are perpendicular to the

viewing direction. In the next step, these planes are combined to compute so-called threshold maps [RPG99]. These threshold maps contain information on how each plane is masked by all the other planes' geometry and shadows and predict the maximum change in illumination that is unnoticeable to the human eye for each pixel. These threshold maps are then used to determine an acceptable LOD for each object which ensures that the difference in contrast when replacing the current LOD is imperceptible. While there are a few tricks described in the paper to speed up this process, the basic idea is to render both the current LOD and best possible LOD and then compare the difference in luminance with the threshold for each pixel. Depending on the number of pixels that are above the threshold, we then either increase, decrease or keep the current LOD. The effectiveness of this method in reducing the LOD while keeping the visual quality was then tested in an user study, where it achieved an average success rate of 65.5% in the test scenarios. However, while their paper shows promising results regarding the image quality, the algorithm introduces a lot of additional computational overhead, which only makes it beneficial in scenes with sufficiently high visual masking.

2.2.3 Real-time tessellation

In contrast to impostors and LOD, real-time tessellation methods allow to increase the polygon count of objects during run-time on the GPU, enabling them to dynamically add details and smooth out surfaces and silhouettes. This is done by strategically placing new vertices and forming additional edges derived from the already existing geometry. Therefore, this makes it possible to send only simple polygons to the GPU which are then refined into complex models. As mentioned before when talking about hardware occlusion queries, communication between CPU and GPU is expensive, so only having to pass the reduced geometry can increase the performance significantly.
The number of new edges generated is controlled by the tessellation level or sometimes called the tessellation factor. A higher level of tessellation results in a tighter and more fine-meshed object. In general, the distance to an object is used as an heuristic, where farther away objects are rendered at a lower level. In this section we will present two key techniques to calculate these new edges and vertices dependent on the input mesh.
The first is called Phong tessellation and prevails because of its simplicity while still achieving high visual quality. To find a new vertex, first its barycentric coordinates in the target polygon are calculated. In the next step, for each of the polygons normals a tangent plane is constructed on which the vertex is projected. Simply interpolating those projected points yields the final position. Figure 2.7 visualises this process. The biggest advantages of the algorithm is that no neighbourhood information of the polygon is needed and that a single rendering pass is sufficient for constructing all additional vertices'. Moreover, since vertices and normals are usually shared between adjacent batches, a continuous surface is guaranteed [BA08].
Point-normal triangles, often abbreviated to PN triangles, follow a very similar principle of interpolating the normals and vertices. First, a set of ten control points is derived from the vertices' position and normals, which are evenly spread across the triangle. These

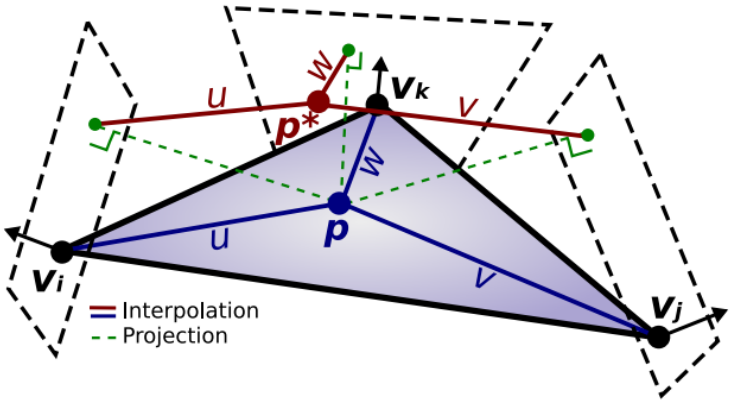

Figure 2.7: Schematic construction of a new vertex with Phong tessellation. The blue point labeled with p is the trilinear interpolation, which is then projected (green dotted lines and points) to each of the tangent planes (dotted rectangles). The red point is the interpolation of those projected points and the final position. Reprinted from [BA08].

control points are then interpolated to form a Bézier surface, smoothing the edges of the object [VPBM01].

swapping one LOD for a lower one.

2.3 Discrete Cosine Transformation

Utilizing the discrete cosine transformation (DCT), any signal can be represented as a weighted sum of basic cosine functions with varying orthogonal frequencies. This is also applicable for two dimensional signals such as images. Figure 2.8 shows a matrix containing every 2D function needed to depict all possible 8×8 block of pixels. The frequency of the function in the top left of the matrix is zero and represents the mean amplitude of the input, also called discrete cosine coefficient. The intensity of vertical frequencies increases the further we move to the right and that of the horizontal frequencies the further we move to the bottom.
Transforming an 8×8 pixel block of an image to the frequency domain with the DCT results in an 8×8 matrix containing the weights that indicate the influence each corresponding cosine function has on the appearance of the image. The higher the absolute value of a coefficient, the greater is the influence of the corresponding frequency. Input signals with amplitudes ranging from $[0, P]$, where P is the highest possible value, are often shifted to the range of $[-\frac{P}{2}, \frac{P}{2}]$ before the transformation. This has the advantage of the resulting coefficients being centred around 0, where a negative coefficient means the inverse of a frequency is present. This matrix will be called DCT coefficient matrix for the remainder of this thesis [ANR74].

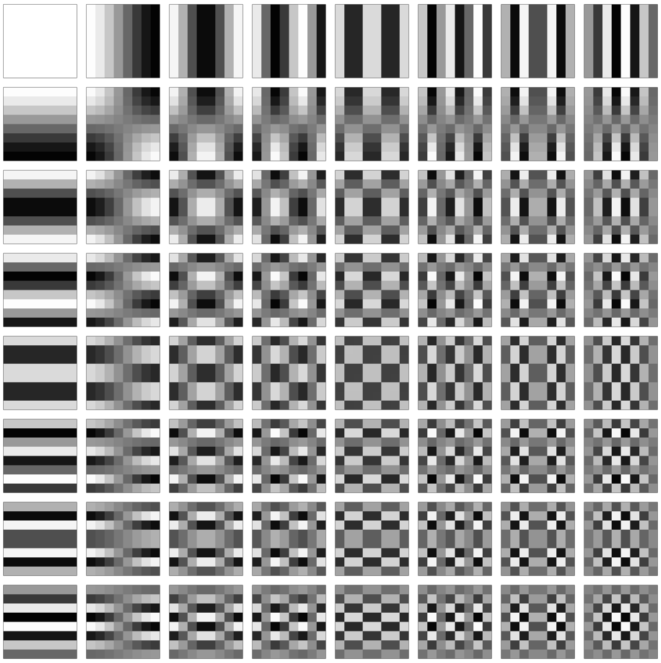

Figure 2.8: The 2D 8×8 discrete cosine transformation's basis functions [Wik21]. The frequency of the function in the top left is zero. The intensity of vertical frequencies increases the further we move to the right and that of the horizontal frequencies the further we move to the bottom.

The DCT is widely used in different fields of computer science: In image compression methods, combined with the observation that high frequencies are less important for the quality of an image, the DCT is used to reduce the storage size of images. To do so, an image is first split into 8×8 blocks of pixels. In theory, an arbitrary block size can be chosen, but 8×8 has proven to be a good balance between simplicity of transformation and achieving a high level of compression. After the transformation, for each block a quantization matrix is multiplied with the DCT coefficient matrix which prioritises lower frequencies and eliminates the high frequencies of images. This results in a sparse coefficient matrix, which combined with other data compression algorithms, makes it possible to store images extremely compact with generating only minimal errors [Wal92].

Another application for the DCT is the detection and removal of noise in images. For this purpose, at first the level of noise has to be estimated. In their paper [KYFI16], Katase et al. divide an image into multiple smaller patches and then analyses the high frequencies of each patch respectively. This information is then used to calculate an overall noise approximation which can then be subtracted from the image. This method illustrates well how the DCT can be used to obtain valuable information and to examine the content of images.

CHAPTER 3

Method

We present an occlusion culling algorithm with hierarchical occlusion map and occluder frequency analysis. The core algorithm is derived from the paper [ZMHH97] by Zhan et al. but extends the method by a novel concept of an occluder frequency analysis. The idea of this addition is to reduce the level of detail of objects which are not significant to the quality of the final image, because they are partly occluded by other objects. Our technique is based on the observation that for the same degree of occlusion, objects that are only visible through small holes can be rendered with less geometric complexity than if they are visible through a single large hole. Figure 3.1 visualises this idea.
Subsequently, we introduce a method that estimates the structure of the occluders and provides a parameter defining the level of visibility. To do so, we apply the discrete cosine transformation and analyse the resulting coefficient matrix. This parameter can then be used to reduce the geometric complexity of hardly visible objects.
The method is designed to take advantage of the parallel processing power of the GPU

Figure 3.1: Left: original armadillo model [Sta] Middle: occluder with large holes Right: occluder with small holes. Even though the number of pixel that are visible of the armadillo is almost equal, fine details are less noticeable through small holes.

to achieve real-time rendering. After loading and preprocessing the scene, the algorithm performs the following steps in each frame:

1. Occluder selection
2. Construction of the occlusion map
3. Overlap test
4. Frequency analysis

In the following sections we will detail the individual tasks of each step and explain the reasons for our design choices.

3.1 Preprocessing

Our approach is designed to require very limited preprocessing and makes no assumptions about the structure of the scene. This makes it versatile and easy to integrate into existing applications.
When loading a scene, we create a bounding box for each mesh. In theory, arbitrary bounding shapes can be used. Although spheres have the benefit of requiring less storage and are easier to construct, the advantage of rectangular shapes are the simpler calculations as well as a better integration of DCT that will become apparent in Section 3.5. It is important that the bounding volume is not too conservative, as this would greatly reduce the effectiveness of the culling process.
For calculating the box, we simply choose the minimum x, y, and z coordinates of a mesh as the lower left corner and the maximum as the upper right. The remaining vertices can be derived implicitly. While these calculations are processed on the CPU, all calculated bounding boxes are stored in a single buffer and are passed to the GPU.

3.2 Occluder selection

The selection of good occluders has the highest impact on the performance of our culling algorithm. Both selecting too few and too many objects can result in a very small number of occludees. Consequently, the visibility algorithm can then only remove a very limited number of polygons, therefore hardly improving rendering performance.
An optimal occluder selection is highly dependent on the scene and should be adjusted accordingly. Our test scenes consist of objects that are similar in size and fairly equally distributed, for details we refer to Chapter 4. This allows us to keep the implementation as simple as possible: we define an object as an occluder solely if it is close enough to the viewpoint.
As a first step, we calculate the distance of the closest point from an object's bounding box to the camera and test if it is below a user-defined threshold. Every object selected

this way is marked as occluder. All the other objects are forming the set of potentially visible set, for which we have to perform the visibility testing in the following steps.
Our application's occluder selection is computed on the GPU, making it extremely efficient. As a consequence, we are able to perform the selection online, avoiding to build an occluder database. Another advantage of this selection is that we do not have to perform a depth test as in the original hierarchical occlusion map algorithm, because no object can be closer to the viewpoint than the occluders.

3.3 Construction of the occlusion Map

For creating the occlusion map, every object determined as occluder in the previous step is rendered at full intensity to a texture without a depth buffer and only a single colour channel. The tessellation level or LOD of the occludees should be set to its maximum, since the so-created curved and convex surfaces may occlude objects that would otherwise be visible. Any lightning or shadow calculations are ignored at this stage. The texture is then downsampled, averaging 2×2 blocks of pixels at every step, until the texture's size is only a single pixel. In the source paper, Zhan et al. render the occluders into a texture with only a fourth of the size of the screen resolution. We assume that this is due to reducing the number of calculations, since the map is built on the CPU and the computation is quite costly. In contrast, we build our map hardware accelerated and can therefore afford rendering at full resolution.
We note that it is important to set the texture parameters to return the nearest pixel when performing a texel lookup instead of interpolating between neighbours, as this would yield errors in the following Section 3.4 and Section 3.5.
We observed one problem when implementing this method which is occurring when a single pixel is not occluded in an otherwise completely occluded region. In high levels of the occlusion map, the influence of a single pixel is becoming so small, that the opacity value is eventually rounded to full intensity due to machine precision. This can result in objects being culled although a very small number of pixels is visible, a detailed explanation can be seen in Figure 3.2. Since this has only very limited impact on the results of this thesis, we did not take any special precautions, but point to possible solutions in Chapter 5.

3.4 Overlap test

In this step we determine for every object in the PVS if it overlaps with the occluders. This is done using the same method as Zhan et al. and is described in detail in Section 2.1.3. First, we transform each vertex of an object's bounding box to clip space. We then compute the two points at minimum and maximum x and y coordinates of the transformed vertices, which implicitly form an axis-aligned bounding rectangle. Following, we shift the AABR coordinates from the clip space range of [-1, 1,] to texture coordinates of [0, 1]. In the next step, we calculate the level of the occlusion map where the AABR has roughly

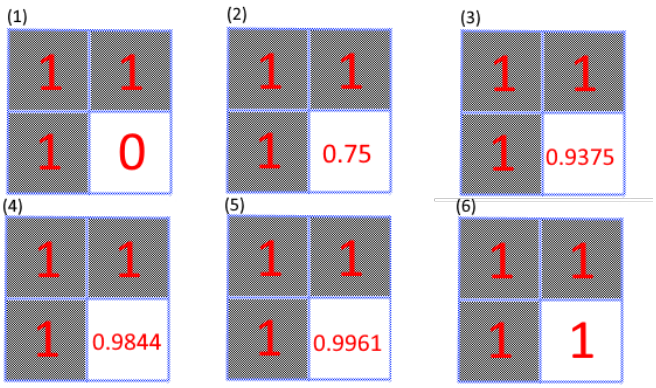

Figure 3.2: A single non-occluded pixel with an opacity value of 0 is surrounded by occluded pixels with an opacity value of 1 (image 1). For each level of the occlusion map, an average of four pixels is calculated. With three occluded pixels and a single non-occluded one, this results in $\frac{1+1+1+0}{4} = 0.75$ for the first level (image 1 to 2). This is repeated for the second level (image 2 to 3) with $\frac{1+1+1+0.75}{4} = 0.9375$, and continues until the final level of the occlusion map is computed. In the case that a single pixel was not occluded in the lowest level of the occlusion map, the occlusion value is quickly converging towards 1, until machine precision reaches its limit and the value is actually rounded to 1, marking it incorrectly as completely occluded (image 6).

the size of a pixel and therefore covers at most four pixels. Since the occlusion map is constructed by averaging 2×2 blocks, its dimensions are reduced at every level by half. As a consequence, we can simply take the logarithm of the largest side relative to the screen size of our AABR to result in our level of occlusion map (LOM). In Equation 3.1, X and Y are the dimensions of the viewport, l and h the length and height of the AABR.

$$LOM = \lceil \log_2(\max(Xl, Yh)) \rceil \tag{3.1}$$

For every corner of the AABR we check the opacity of the occlusion map at the calculated LOM. If the value for all points is above a user-defined threshold, we can cull the object. If any corner is below the threshold, we descend one level in the occlusion map and check every sub-pixel within the AABR. This is done twice at most, because the number of pixels we potentially have to test grows by the power of two. If any of the sub-pixels are still below the threshold, we know that the object is not completely occluded.
For an early termination, we also check if the opacity values of every corner of the AABR equals zero. In this case, the object is not occluded at all, and we simply set the object's tessellation level to the maximum. Otherwise, we perform a frequency analysis to determine their level of visibility.
Algorithm 3.1 outlines the overlap test. The outer for-loop is redundant in the actual implementation because we are processing every bounding box in parallel in its own thread.

Algorithm 3.1: Overlap test

```
1  Function isVisible(aabr):
2      for object in PVS do
3          LOM = ⌈log2(max(Xl, Yh))⌉;
4          for corner in object.aabr do
5              visibility = textureLod(occlusionMap, corner.xy, LOM);
6              if (visibility ≳ threshold) ∨ checkSubPixels() then
7                  corner.occluded = true;
8              end
9          end
10         if all corners occluded then
11             object.instanceCount = 0;
12             return;
13         end
14         if all corners equal 0 then
15             object.tessellation_level = MAX_TESSELLATION;
16             return;
17         end
18         doFrequencyAnalysis();
19     end
```

3.5 Frequency analysis

In regions where the overlap test determines objects to be visible, we want to calculate a parameter measuring their level of visibility. This parameter is then used to reduce the geometrical complexity of partly occluded models to increase their rendering efficiency. Simply calculating the number of visible pixels does not fully exploit the imperfections of our visual perception or the spatial information available from the hierarchical occlusion map. To recall, Figure 3.1 shows that with the same number of visible pixels, small details of an object are far less noticeable if we can perceive them only through multiple small holes in the occluders, instead of a single large one. Accordingly, we can much further reduce a model's geometric complexity without introducing apparent artifacts if it is only visible through small holes.
To analyse the structure of the occluders, we utilise the 2D DCT. Its equation can be seen in Equation 3.2 and is used to transform the regions overlapped by the AABR of the occlusion map to the frequency spectrum. If the resulting coefficient matrix has low values in the low frequency sections, we know that the analysed block must be fluctuating frequently between high and low opacity values. This is reminiscent of occluders with many small holes, as can be seen in Figure 3.3.

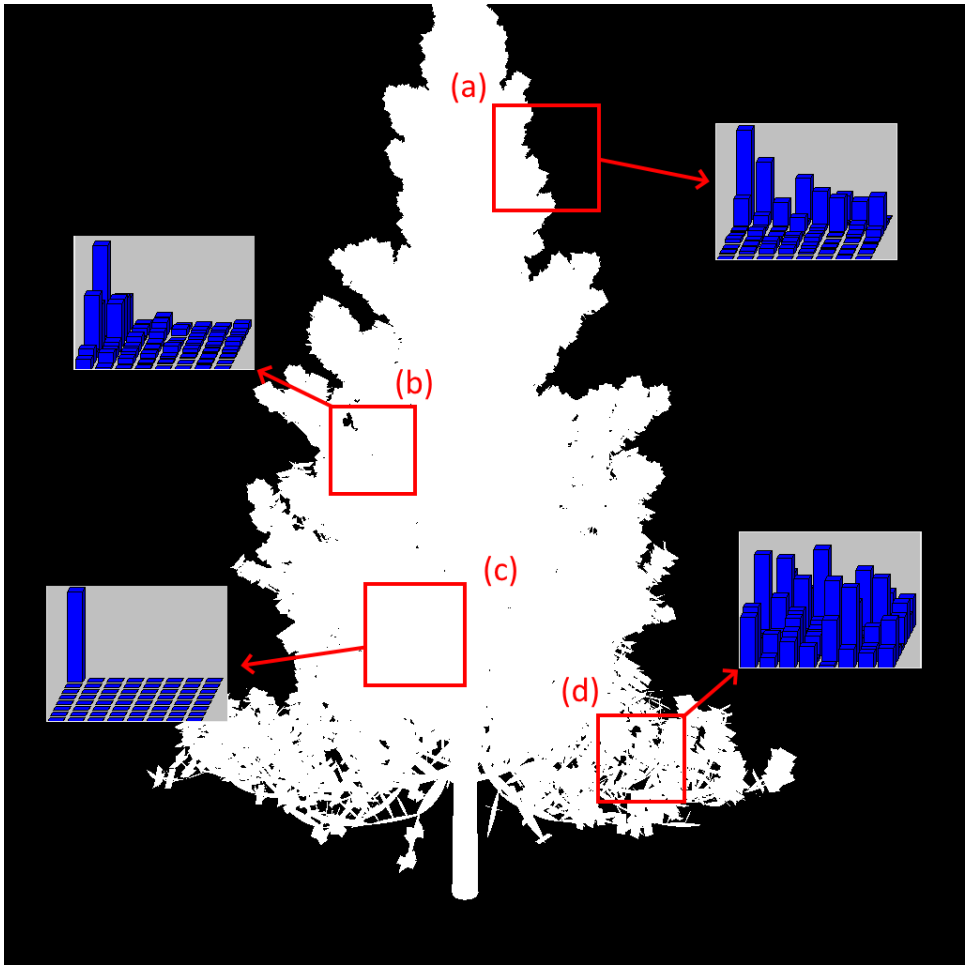

Figure 3.3: Schematic depiction of the coefficient matrices of the pixels within the red boxes when transformed by the DCT. The graphs are illustrating the resulting coefficient matrices with a bar for each coefficient. The higher a bar, the greater is the value of the coefficient and therefore the influence of the corresponding frequency on the image. a) Abrupt cut from occluded to non-occluded results in a lot of vertical frequencies. b) A single large hole does not include many high frequencies. c) Completely occluded regions result in a single bar at the coefficient for the frequency of 0 d) Frequent changes in the occlusion map result in many high frequencies. DCT coefficient graphics were created with [BGH+05].

$$F(u,v) = \alpha(N)\alpha(M) \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f(i,j) \cos(\frac{\pi u}{2N}(2i+1) \cos(\frac{\pi v}{2M}(2j+i)$$
$$\alpha(x) = \begin{cases} \frac{1}{\sqrt{x}}, & \text{for } x \geq 0 \\ \sqrt{\frac{2}{x}}, & \text{otherwise} \end{cases} \tag{3.2}$$

One challenge arises from the fact that the AABR of objects at the computed LOM covers at most four pixels. This implies that its largest dimension must be smaller than the size of a pixel, and larger than half a pixel. However, for the DCT to yield meaningful results, inputs with a size between 8×8 or 16×16 blocks of pixels are ideal. For this purpose, we again take advantage of the fact that the number of pixels in the occlusion map decreases by the power of 2 at each level. If we descend four levels in the occlusion map, we have at the minimum $2^4 \cdot 0.5 = 8$ pixels and at the maximum $2^4 \cdot 1.0 = 16$ pixels in one dimension, and equal or less in the other. This block can then be transformed with the DCT, ensuring its size is a good compromise between keeping computational costs manageable and still accomplishing meaningful results.
The simplest idea would be to calculate the entries of the coefficient matrix for all frequencies. However, the fastest implementations of the 2D DCT still has a complexity of $O(n^2 \log n)$. Even though the algorithm can be parallelised well on the GPU, the costs of calculating the DCT for every object in the PVS could still outweigh the benefits. Therefore, we suggest two simplifications to increase efficiency of the DCT: firstly, we calculate the coefficients only for frequencies that are important for computing the visibility parameter. Secondly, we check the number of high frequencies by calculating the number of low frequencies. This is possible because the frequencies are indirectly proportional, so that small occurrences of low frequencies imply high occurrences of high frequencies and vice versa.

In Equation 3.3 we present a simple parameter for describing the level of visibility (LV) that takes the average intensity of the pixel block (the first coefficient) and subtracts the absolute value of the highest frequencies in every direction as well as their combination. This results only in high values for inputs with a sufficiently high level of occlusion as well as a high occluder frequency. Figure 3.4 shows the steps required to calculate the LV.

$$LV = F(0,0) - (|F(1,0)| + |F(0,1)| + |F(1,1)|) \tag{3.3}$$

To benefit from the level of visibility estimation, the application needs to implement a method that can reduce the complexity of models dynamically in the scene, the most common being impostor, LOD and tessellation. Depending on which technique is implemented, the visibility parameter has to be computed slightly differently. While impostor techniques require a threshold of when to replace an object or update the image representation, LOD and tessellation enable a more continuous control of the level of complexity.

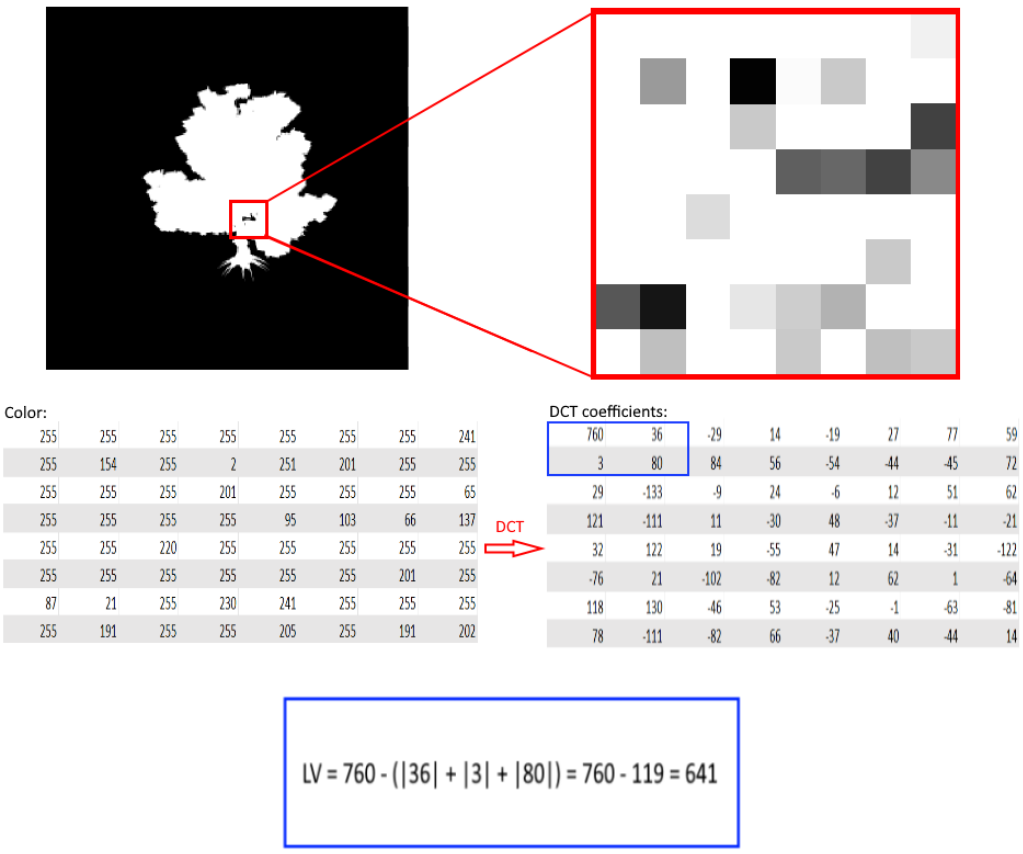

Figure 3.4: Computation of the level of visibility for an AABR. Top left: occlusion map with a tree occluder. Highlighted in red is the AABR of the object for which we want to test the level of visibility for. Top right: Closeup of the 8×8 block of the occlusion map contained within the bounding box. Bottom left: the opacity values of the 8×8 block. The values were multiplied by 255 to conform to standard image colour values and to easier compare their size. Bottom right: discrete cosine transformation coefficients after shifting the values to the range [-128,127] and transforming them with the DCT. The values within the blue rectangle are input for the level of visibility parameter. The LV for a completely occluded 8×8 block is 1024 and for a non-occluded -1024, so this example has $\frac{3}{4}$ of the maximum LV, meaning that the object is fairly occluded.

CHAPTER 4

Implementation and Results

To evaluate our method, we implemented the algorithm in C++ and OpenGL version 4.5. In this chapter, we first summarise the most important aspects of our implementation. Following, we discuss the results of two test scenarios regarding the efficiency of the algorithm and of a small survey concerning the visual quality of the rendered images.

4.1 Implementation

In this section we summarise implementation details that are not specific to our method, but which improve the efficiency and which can be found in most rendering applications.

4.1.1 Indirect rendering

To improve performance, we exploit a technique known as indirect rendering [SWH15]. The difference between indirect and normal rendering is that the parameters of a draw call are retrieved from the GPU instead of the CPU. For this purpose, we have to first load the entire geometry of the scene to a single buffer which we will call the scene buffer. Afterwards, we store a simple struct for every mesh in a draw call buffer that specifies the input variables required for rendering: the vertex count, positions of the first vertex and index in the scene buffer, number of instances that should be rendered as well as the ID of the base instance.
After binding this draw call buffer to the GPU, we can directly manipulate the input parameters when performing visibility culling: If an object is determined to be occluded or outside of the view frustum, we simply set the instance count to zero, preventing it from being rendered.
The advantage of this method is that we only have to issue a single draw call for each shader that is used for rendering the scene. Moreover, we do not have to read back the results from the GPU-side visibility calculations, increasing the performance significantly.

4.1.2 Frustum culling

Since most applications utilise frustum culling, we also included it in our implementation. To recall, frustum culling removes all geometry from the rendering pipeline that is outside of the view frustum. To calculate if an AABR is inside the view frustum, we simply have to check whether both corners are greater 1 or smaller -1 in any direction. If this applies, we know that the AABR is outside of the clip space and therefore the mesh is outside of the view frustum. To cull the mesh, we set its number of instances in the draw call buffer to 0. We compute frustum culling on the GPU and before the occluder selection to already reduce the number of objects we have to apply our algorithm for.

4.1.3 Tessellation

To utilise the level of visibility, we perform Phong tessellation [BA08] over triangle patches because of its simplicity. We define a tessellation level which determines the number of times an edge is split into segments. The maximum tessellation level is bound to 20 and is the highest quality of a model.
The individual level of tessellation for each object depends on their level of visibility that is dynamically computed and described in Section 3.4. In our application the visibility parameter is scaled to the range of our tessellation levels. For this purpose, we simply map the interval of the minimum and maximum of the possible parameter values to the range of the tessellation levels. It is stored in a buffer so that we can set it on the GPU, circumventing a pass through the CPU. As an example, the LV of 641 from Figure 3.4 would result in a tessellation level of 4 in our application.
Special care has to be taken when tessellating so called T-junctions. They can occur when the edge between a flat and a non-flat surface is split and result in a crack in the surface. There are multiple strategies to circumvent this, with some being extremely complex. We implement one of the simplest solutions that is averaging the normals of every mesh of an object, making it impossible for a flat surface to be next to an uneven one. The downside is that sharp corners are smoothed out in the process even if we do not want them to, as can be seen in Figure 4.1 [Dud12, PF05].

4.2 Results

To test the efficiency of our method, we rendered two complex scenes and compared our frame times to hierarchical occlusion map without frequency analysis and only frustum culling. The applications were executed in Windows 10 version 10.0.19042 on an AMD Ryzen 7 2700X, 3700 MHz, 8 kernel processor with 16 GB RAM and a NVIDIA GeForce GTX 1070 Ti GPU.

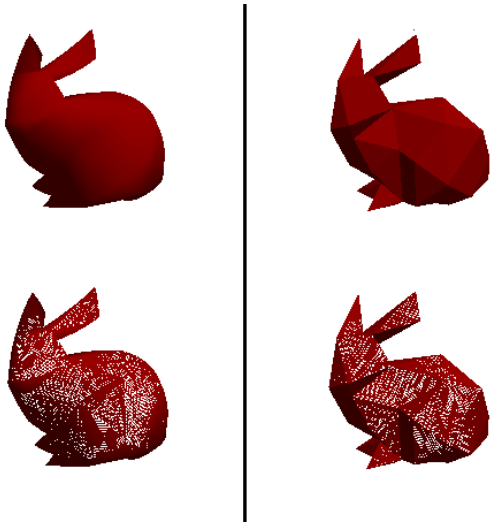

Figure 4.1: Left: tessellation with smoothed normals. Right: tessellation without smoothed normals. The surfaces with smoothed normals are much rounder and plumper than without.

4.2.1 5001 bunnies scene

The first scene consists of 5001 Stanford bunnies [Sta] and is inspired by one of the test scenes from the paper by Bittner et al. [BWPP04]. To utilise the tessellation method, the original model was imported into the software Blender and its mesh was reduced to just 98 triangles. With the maximum tessellation level of 20, the triangle count for a single bunny increases to 58,800. In the next step we copied the object to 5000 random locations within the volume of a relatively small cube, resulting in a tightly packed scene. Each bunny was further given a random rotation on the global x axis and all objects are illuminated by a simple direction light shining from the top. The resulting test scene can be seen in Figure 4.2

An example for the culling and geometry reduction of our application can be seen in Figures 4.3 and4.4. The first pictures the test scene as perceived by the camera. The occluders are drawn in red, the occludees in different shapes of blue. The darker an occludee's colour, the higher is its level of tessellation and therefore its polygon count. We note that hardly any light blue objects are visible and the visible ones are exclusively in places where they can only be seen through small holes between the occluders.
For the second image, the position of the camera that is used for the occlusion culling computations is anchored to the same position as before. To observe what effects the occlusion culling algorithm has, we then move to the backside of the scene looking in the opposite direction as in the previous render. It is easily noticeable, that the bunny objects that would be in the middle of the cube, are culled completely by the hierarchical occlusion map test. Moreover, in contrast to before, we can see that actually numerous

Figure 4.2: First test scene with 5001 densely packed bunnies and 294,058,800 triangles at max tessellation.

objects near the centre are drawn in bright blue, which indicates that they are rendered at a low resolution. Because of this, the scene is reduced to just 132,872,418 triangles which is almost $\frac{1}{9}$ of the original triangle count.

We recorded a walkthrough and replayed it with three different visibility algorithms: only frustum culling, visibility culling with hierarchical occlusion map and our extension of the method. The results can be seen in Figures 4.6 and 4.7.

The camera is first circling around the cube of bunnies so that every object is within the frustum and with only the outermost being detected as occluders. We can see that standard hierarchical occlusion map culling does not deal with this situation very well and even performs worse than frustum culling in the beginning. This is due to the bunnies being small and sparse occluders that to a large extend only occlude parts of an occludee. As a result, the algorithm performs the overlap test for several objects but will not be able to actually cull any, leading to a significant drop in performance.

In contrast, our frequency analysis is still able to remove up to 30 percent of the geometry just by reducing the complexity of the hardly visible models.

Starting from frame 900, the camera is closing in on the scene, increasing the number of polygons outside of the viewing frustum. Subsequently, the efficiency gains of the occlusion culling algorithms are reduced as fewer objects can be removed. On the plus side, we also have to perform less overlap tests, reducing the computational overhead to a minimum.

Figure 4.5 shows the different openGL commands and their GPU run-time costs for our algorithm. The most expensive calls are rendering the scene and creating the occlusion map. The frequency analysis is hardly noticeable in comparison.

Figure 4.3: Front view of the camera. The red bunnies are the occluders, the blue bunnies occludees. The brighter the blue, the lower is the level of visibility and therefore their polygon count.

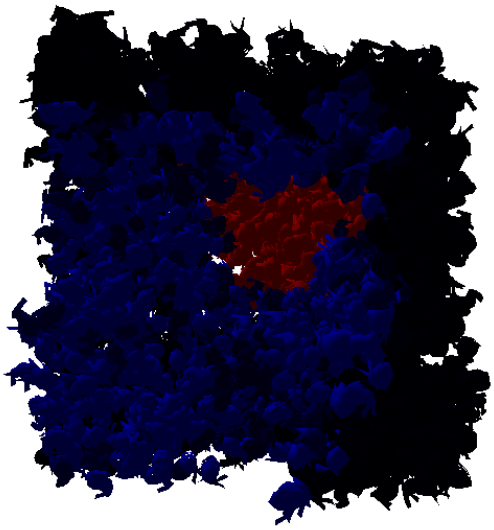

Figure 4.4: View from the back of the scene to see the effects of our algorithm. The hole in the middle is due to bunnies being occluded completely and which were therefore culled. Our algorithm reduces the scene from 294,058,800 to 132,872,418 triangles.

Figure 4.5: Frame captured by Nvidia Nsight Systems from the 5001 bunnies scene. The blue "glMultiDrawElementsIndirect" is the rendering command, the red "glClear" the generation of the occlusion map. The grey part in between these two is the overlap test and frequency analysis. The most expensive operation besides rendering the objects is generating the occlusion map.

4.2.2 10001 bunnies scene

The second test scene uses the same bunny model as the first. Only this time, it is copied 10000 times and placed in a much larger volume to produce a more sparse distribution. In addition to the rotation, each bunny is randomly scaled in size as well. Figure 4.8 shows the entire scene on the left and an example viewpoint from within the bunny cube on the right.

We once more implemented an automated walkthrough to measure the frame time and triangle count of all three methods. The results are presented in the graphs in Figures 4.9 and 4.10. At first, the camera observes almost the entire scene with relatively low occlusion present, resulting in high frame times. It then closes in on the bunny cube and moves along a circle on the inside.

We observe that in this scene with lower geometric density, the occlusion culling algorithms are less effective. However, they still outperform simple frustum culling most of the time. The only exception occurs in the same situation as in the first test scene when frustum culling already removes most of the geometry. This is because our implementation still builds an occlusion map even if there are few objects in the potentially visible set.

In general, our frequency analysis extension consistently lowers the frame time compared to the original algorithm.

We note that when measuring the frame times, we encountered random spikes that occur more frequently after recording for a while. We think that this is due to the method of recording the frame data as we cannot find any signs of the application stalling at any point and the spikes are only occurring after 1500+ frames. Unfortunately, we were unable to find a solution for this issue, but the overall performance should not be influenced by this.

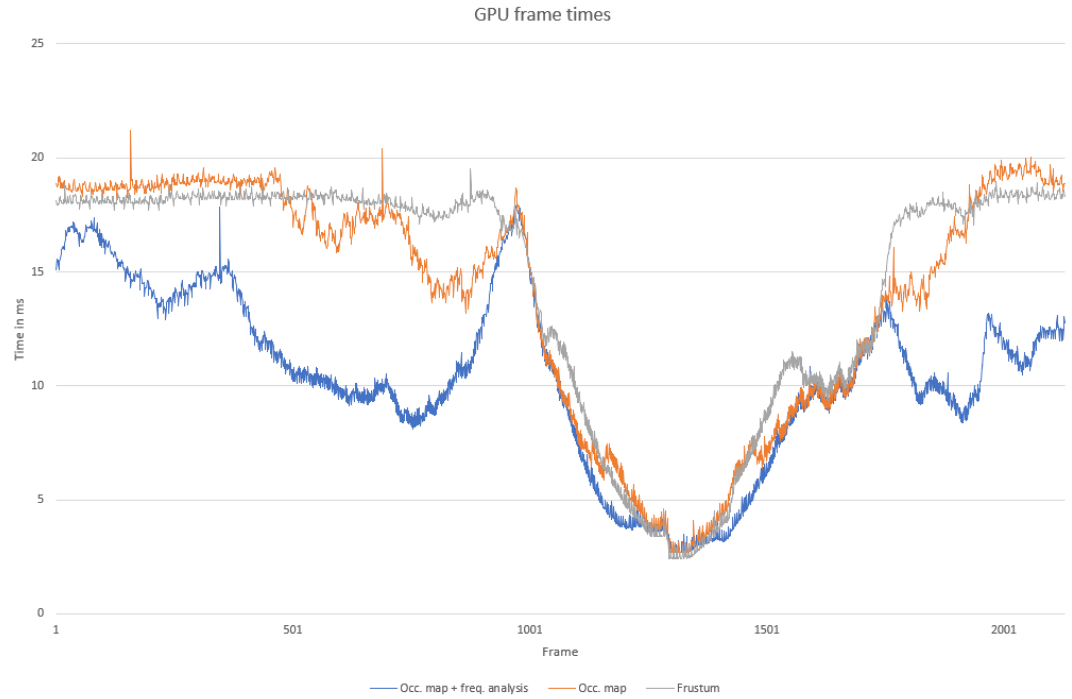

Figure 4.6: Comparison of frame times during the 5001 bunnies scene's walkthrough of occlusion map algorithm with and without frequency analysis and only frustum culling. Our method consistently outperforms the other approaches.

4.2.3 Visual quality survey

The performance gain of our method is only relevant if we can still ensure that the visual quality of the scene is not compromised by reducing the quality at which the hardly visible objects are rendered. However, measuring the impact on the visual quality is challenging, as it depends on human perception and can differ between individual observers [AOS+17].
To still put our method to the test, we conducted a small survey consisting of five people. While this sample size is far from statistical significance, it can at least hint at the potential of our method and ensures that there are no obvious artifacts. As before, we recorded a walkthrough of different scenes, this time consisting of an object behind a plane with different levels of occlusion. An example can be seen in Figures 4.11 and 4.12. We then showed the recordings to the participants, once with the occluder frequency analysis turned on, and once with it turned off.
Especially in scenes with highly occluded objects as well as in scenes with hardly occluded objects we received the feedback that the two methods are indistinguishable from each other. The most noticeable artifact arises in scenes with moderate occlusion and which contain objects whose silhouette shape is heavily affected by the level of tessellation. In this scenario, a noticeable popping effect can occur at the transition between two levels

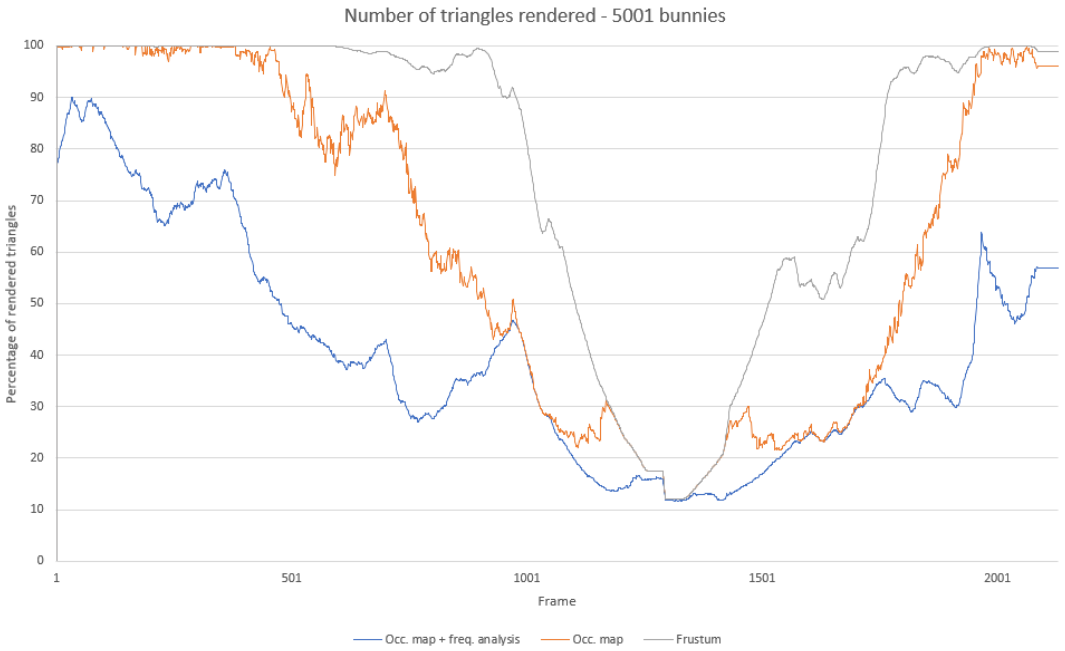

Figure 4.7: Comparison of percentage of number of triangles rendered during the 5001 bunnies scene's walkthrough with 294,058,800 triangles at full tessellation. Our method significantly reduces the amount of rendered geometry.

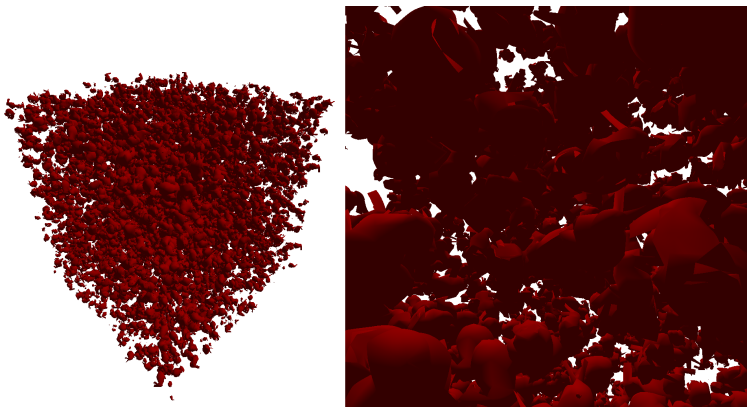

Figure 4.8: Second test scene with 10001 bunnies and 588,058,800 triangles at full tessellation.

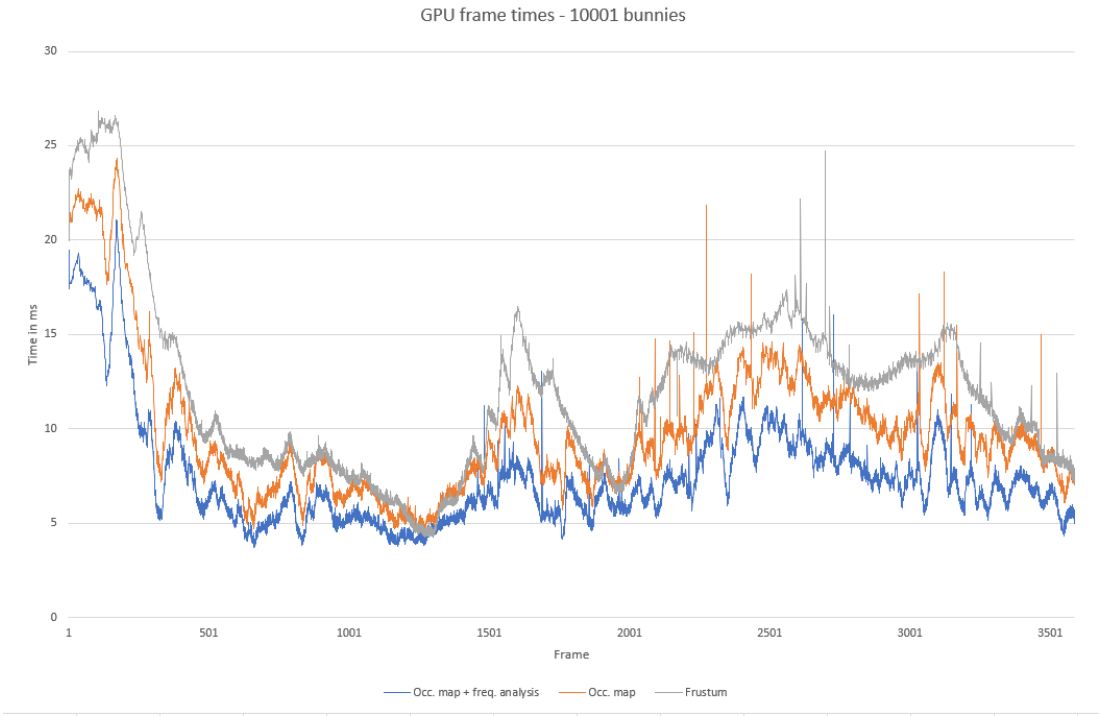

Figure 4.9: Comparison of frame times during the 10001 bunnies scene's walkthrough. Even with a lower geometric density our algorithm performs well compared to the other methods.

of tessellation.

4.2.4 Discussion

The results of our tests show that in terms of rendering efficiency our method consistently outperforms the standard hierarchical occlusion culling algorithm [ZMHH97]. Due to the frequency analysis, this increase in performance is achieved without compromising the visual quality of the scene.
In comparison to the perceptual rendering pipeline [DBD+07], our occluder frequency analysis only introduces minimal overhead in both complexity and fixed run-time cost, as it is built on an already established occlusion culling method. However, our technique is only focusing on the occluder frequency, ignoring the contrast between individual pixels. Therefore, even a combination of both methods might be plausible. In any case, a direct comparison of both algorithms would be an interesting task for a follow-up work.

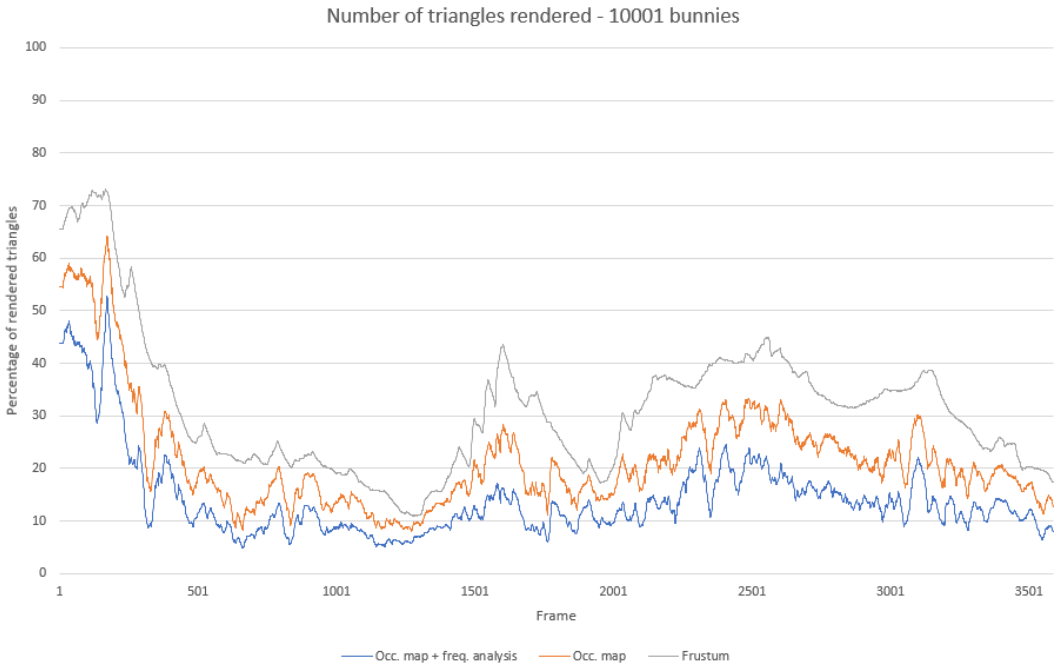

Figure 4.10: Comparison of percentage of number of triangles rendered during the 10001 bunnies scene's walkthrough with 588,058,800 triangles at full tessellation. Although less effective with a lower geometric density, the number of triangles is still reduced noticeably.

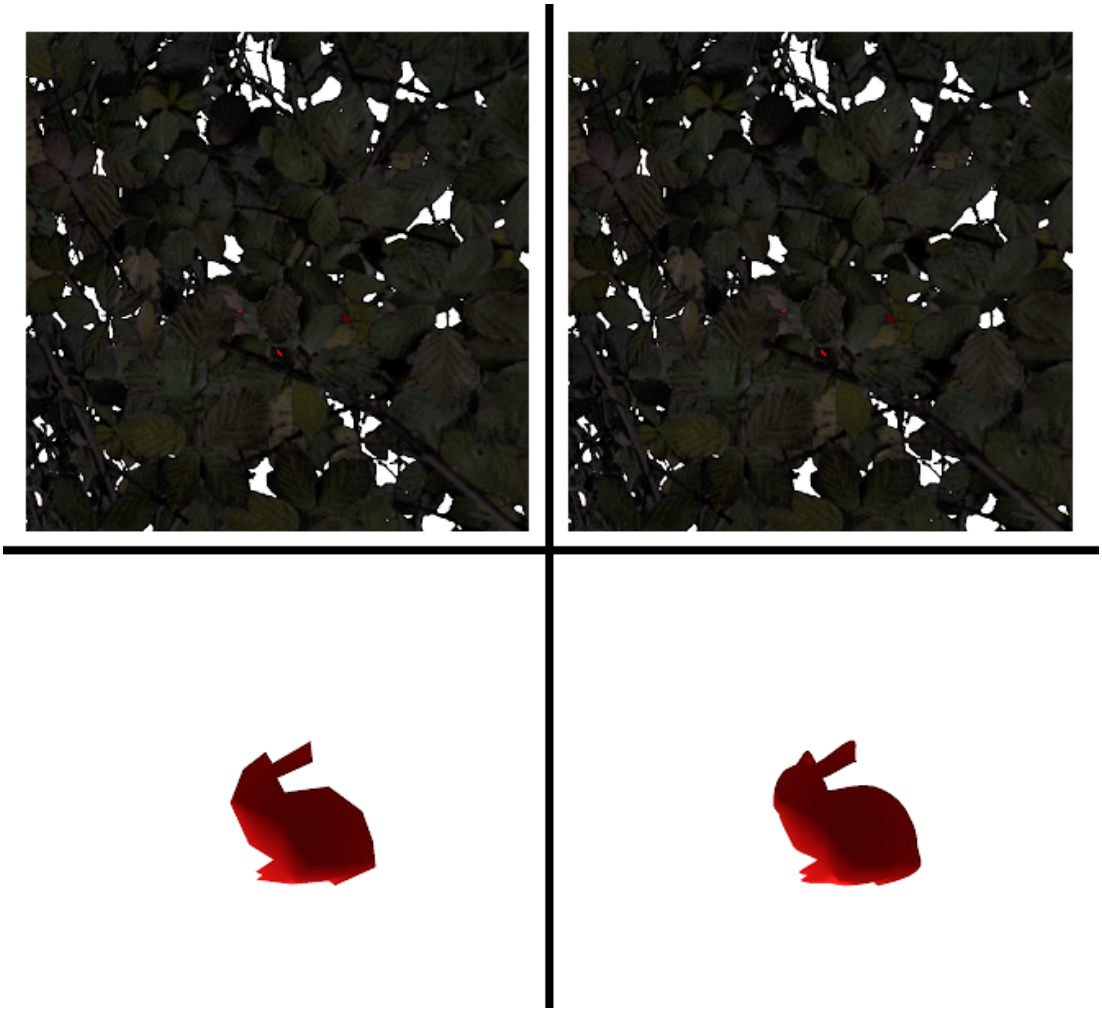

Figure 4.11: Comparison of visual quality with occluder frequency analysis turned on and off for the bunny model [Sta] which is occluded by foliage. Top left: occluder frequency analysis turned on. Top right: occluder frequency analysis turned off. Bottom left: occluder frequency analysis turned on and removed occluders. Bottom right: occluder frequency analysis turned off and removed occluders. Since the bunny is almost completely occluded, our method renders the bunny at a lower LOD. However, it is only really possible to tell the difference if we turn off the occluders (remove the foliage).

Figure 4.12: Comparison of visual quality with occluder frequency analysis turned on (left) and off (right) for lion head model [McG17] and low occlusion.

CHAPTER 5

Conclusion and future work

We have presented a novel concept of occluder frequency analysis that improves the effectiveness of visibility culling when rendering large scenes with occluders containing small holes. We showed that with the help of the discrete cosine transformation, we can gain more insight into the structure of occluders and adjust the occludees' quality accordingly. Because we perform frustum and hierarchical occlusion culling beforehand, this analysis is only applied to a finite number of objects. Therefore, our approach introduces only very limited computational costs in the worst case, while otherwise improving rendering efficiency.
We believe that there is a lot of potential in further investigating methods to improve rendering when dealing with hardly visible objects. Currently, resources addressing this problem are very limited in quantity and many papers that try to solve the general visibility problem regard it only in a short side note, if discussed at all.

5.1 Future work

While we showed the potential of the occluder frequency analysis, there is still room for improvement. In the following subsections we point out problems that have emerged during our implementation and that require further research. Additionally, we will discuss possible further refinements to the algorithm that could be addressed in a follow-up work.

5.1.1 Round off errors

As mentioned in Section 3.3, when creating the occlusion map some inaccuracies may be introduced by roundoff errors. One solution could be to use a higher precision data type to store the opacity values. However, this will only increase the level at which the issue occurs and not solve it completely. Another idea would be to raise a flag for each pixel

where a sub-pixel is not occluded. If only conservative occlusion culling is implemented, we do not have to interpolate at every step and only store if all sub-pixels are occluded.

5.1.2 Popping artifacts between LOD

The popping artifacts that were visible in the visual quality survey form Section 4.2.3 are due to our very simplistic implementation of the tessellation algorithm. Since popping is a common problem in LOD methods, there is plenty of literature on how to reduce this visual error such as unpopping [GW07]. Integrating such a refinement could further improve the results of our method.

5.1.3 Over-conservative bounding volumes

Another challenge arises from scenes where objects have shapes that cannot be tightly enclosed with a bounding box, as illustrated in Figure 5.1. To still achieve good results with our algorithm, the meshes should be split to produce more accurate bounding volumes. However, this can greatly inflate the number of bounding boxes necessary, reducing the efficiency. Therefore, if this is not applicable, a non-conservative bounding box that does only include the core geometry could greatly increase effectiveness. The same applies for objects where the bounding box would have extremely disproportional dimensions. It should be taken into consideration that everything outside the bounding box will be ignored during the visibility calculations.

5.1.4 Fast discrete cosine transformation

Since the discrete cosine transformation is widely used in the field of computer vision, a lot of research is conducted to further improve its efficiency. Whilst our implementation is sufficient for the application, additional speedup when computing the DCT can free the GPU to compute different tasks in rendering engines.
Especially video compression methods require a fast DCT. Our implementation has a higher tolerance for errors since we do not need to revert the transformation. Therefore, replacing the exact DCT with an approximation as presented in [LSSB20] could be very promising.

5.1.5 Improve GPU parallelisation

With GPUs becoming more and more powerful, the parallelisation of our program could be further improved. Every coefficient of the DCT could be calculated by its own thread, further reducing the overhead of our method and allowing us to analyse all frequencies of the occlusion map. As a consequence, the visibility parameter could then be better adjusted to achieve an even more precise adaption of the occludees' quality.

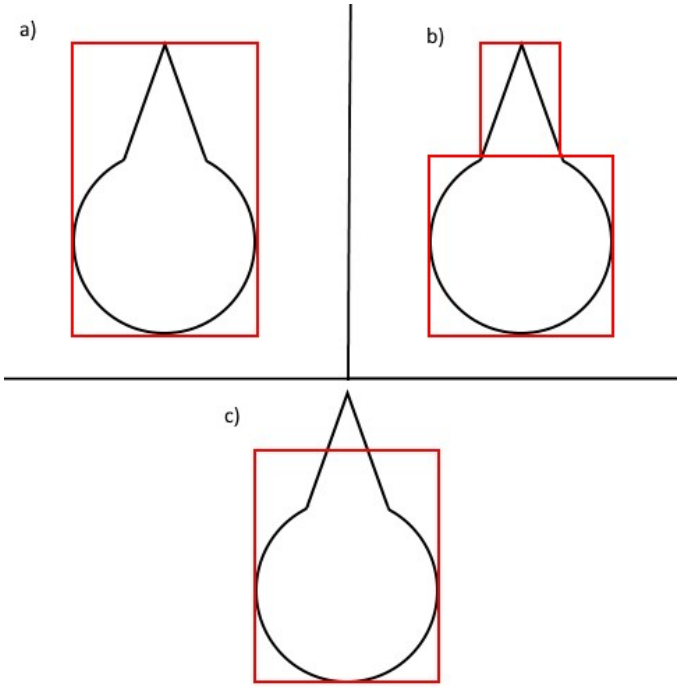

Figure 5.1: a)The object is completely contained within its red bounding box. Because of the shape of the object, the size of the bounding box is relatively large and the overlap test would be less effective. b) The bounding box is split into two, resulting in a tighter bounding box and reducing the overall size but also doubling the number of required overlap tests. c) Non-conservative bounding box that has a smaller volume but might introduce errors.

5.1.6 Patch-based visibility culling

Instead of applying the occlusion culling algorithm to every object, Nießner et al. [NL12] test visibility on patch level. In short, this means that the objects are divided into smaller meshes so that each feature of an object is processed individually.
For large scenes, we are of the opinion that the number of overlap tests required for all patches could quickly become overwhelming. However, our application could adapt this approach to adjust different parts of the model to different levels of geometric complexity while still only testing the visibility once for the entire model. For example, if only one half of the entire model is visible, the DCT would reflect this in its coefficients as can be seen in Figure 3.3a. As a consequence, we could render only the visible half at a high resolution and thus adjust the models to the visibility on a more fine-grained level.

List of Figures

List of Algorithms

Acronyms

AABR axis-aligned bounding rectangle. 7–9, 19–21, 23, 24, 26, 41

CPU central processing unit. 9, 14, 18, 19, 25, 26

DCT discrete cosine transformation. 2, 3, 5, 15–18, 21–24, 37–39, 41

GPU graphical processing unit. 9, 14, 17–19, 23, 25, 26, 28, 38

HVS hardly visible set. 13

LOD level of detail. xi, xiii, 9, 12–15, 19, 23, 35, 38

LOM level of occlusion map. 20, 23

LV level of visibility. 2, 17, 20, 21, 23, 24, 26, 29, 41

PVS potentially visible set. 6, 19, 23, 30

Bibliography

[AL99] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 307–316, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[AM82] Edward H Adelson and J Anthony Movshon. Phenomenal coherence of moving visual patterns. *Nature*, 300(5892):523–525, 1982.

[AMHH+18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[ANR74] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transfom. *IEEE Trans. Comput.*, 23(1):90–93, January 1974.

[AOS+17] Jyrki Alakuijala, Robert Obryk, Ostap Stoliarchuk, Zoltan Szabadka, Lode Vandevenne, and Jan Wassenberg. Guetzli: Perceptually guided JPEG encoder. *CoRR*, abs/1703.04421, 2017.

[ASVNB00] Carlos Andujar, Carlos Saona-Vazquez, Isabel Navazo, and Pere Brunet. Integrating Occlusion Culling and Levels of Detail through Hardly-Visible Sets. *Computer Graphics Forum*, 2000.

[BA08] Tamy Boubekeur and Marc Alexa. Phong tessellation. *ACM Trans. Graph.*, 27(5), dec 2008.

[BGH+05] Christoph Ballhause, Marco Gebbert, Hariolf Häfele, Christian Heinlein, Wolfgang Klas, Michael Plichta, Raimund Specht, Jochen Wandel, and Maia Zaharieva. VisJPEG, 2005.

[BWPP04] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 2004.

[DBD+07] George Drettakis, Nicolas Bonneel, Carsten Dachsbacher, Sylvain Lefebvre, Michael Schwarz, and Isabelle Viaud-Delmon. An Interactive Perceptual

Rendering Pipeline using Contrast and Spatial Masking. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques*. The Eurographics Association, 2007.

[Dud12] Bryan Dudash. My tessellation has cracks!, Mar 2012. visited on 2022-01-03.

[ESSS01] Jihad El-Sana, Neta Sokolovsky, and Cláudio T. Silva. Integrating occlusion culling with view-dependent rendering. In *Proceedings of the Conference on Visualization '01*, VIS '01, page 371–378, USA, 2001. IEEE Computer Society.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 231–238, New York, NY, USA, 1993. Association for Computing Machinery.

[Gre95] Ned Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, 1995. AAI9539493.

[GW07] Markus Giegl and Michael Wimmer. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum*, 26(1):46–49, March 2007.

[HD04] Tan Kim Heok and D. Daman. A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.*, pages 70–75, 2004.

[Jes05] Stefan Jeschke. *Accelerating the Rendering Process Using Impostors*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, March 2005.

[JWP05] Stefan Jeschke, Michael Wimmer, and Werner Purgathofer. Image-based representations for accelerated rendering of complex scenes. In Y. Chrysanthou and M. Magnor, editors, *EUROGRAPHICS 2005 State of the Art Reports*, pages 1–20. EUROGRAPHICS, The Eurographics Association and The Image Synthesis Group, August 2005.

[JWSP05] Stefan Jeschke, Michael Wimmer, H. Schumann, and Werner Purgathofer. Automatic impostor placement for guaranteed frame rates and low memory requirements. pages 103–110, 01 2005.

[KYFI16] Hayato Katase, Takuro Yamaguchi, Takanori Fujisawa, and Masaaki Ikehara. Image noise level estimation by searching for smooth patches with discrete cosine transform. In *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2016.

[LSSB20] Hsin-Kun Lin, Chi-Chia Sun, Ming-Hwa Sheu, and Mladen Berecovic. A new low-complexity approximate dct for image and video compression. *Journal of the Chinese Institute of Engineers*, 43(6):580–591, 2020.

[Mat10] Oliver Mattausch. *Visibility Computations for Real-Time Rendering in General 3D Environments*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, April 2010.

[MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)*, 27(2):221–230, April 2008.

[McG17] Morgan McGuire. Computer graphics archive, July 2017.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, page 95–ff., New York, NY, USA, 1995. Association for Computing Machinery.

[NL12] Matthias Nießner and Charles Loop. Patch-based occlusion culling for hardware tessellation. In *Computer Graphics International*, volume 2, 2012.

[Per20] Jakob Pernsteiner. Ensuring the effectiveness of chc++ in vulkan, October 2020.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[PT02] Ioannis Pantazopoulos and Spyros Tzafestas. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems*, 35:123–156, 10 2002.

[RPG99] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 73–82, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Sch97] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97*, pages 151–162, Vienna, 1997. Springer Vienna.

[SDB97] Francois Sillion, George Drettakis, and Benoit Bodelet. Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery. *Computer Graphics Forum*, 1997.

[SLCO+04] Erez Sayer, Alon Lerner, Daniel Cohen-Or, Yiorgos Chrysanthou, and Oliver Deussen. Aggressive visibility for rendering extremely complex foliage scenes. 2004.

[Sta] The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`.

[SWH15] Graham Sellers, Richard S. Wright, and Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 7th edition, 2015.

[VPBM01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved pn triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, page 159–166, New York, NY, USA, 2001. Association for Computing Machinery.

[Wal92] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[Wik21] Wikipedia, the free encyclopedia. Dct-8x8, 2021. [Online; accessed December 19, 2021].

[WM03] Andrew Wilson and Dinesh Manocha. Simplifying complex environments using incremental textured depth meshes. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, page 678–688, New York, NY, USA, 2003. Association for Computing Machinery.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 77–88, USA, 1997. ACM Press/Addison-Wesley Publishing Co.