Elena Ivanchenko

Massively Parallel Analytic Visibility for Streaming Virtual Reality

MASTER'S THESIS

to achieve the university degree of
Master of Science
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Ass.Prof. Priv.-Doz. Dipl.-Ing. Dr.techn. Markus Steinberger, BSc
Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg
Institute of Computer Graphics and Vision

Graz, January 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

In this work, we make the case for a renewed relevance of analytic visibility methods in the age of massively-parallel graphics processors and consumer virtual reality. While having been neglected for many years, due to their ability to solve visibility for an entire range of viewpoints at once, analytic methods show great promise to play a vital role in one potential solution to the latency problem in virtual reality. The key advancement needed to follow through on this promise is a massively-parallel potential visibility algorithm that can effectively utilize modern graphics processors. As a first step towards this goal, we design and implement a novel visibility method that has the potential to be extended to a full potential visibility method in the future. In an extensive evaluation, we demonstrate significant improvements over previous work in terms of performance, scalability, and robustness.

Kurzfassung

Aufgrund ihrer Fähigkeit das Sichtbarkeitsproblem für mehr als einen Blickpunkt gleichzeitig zu lösen, gewinnen analytische Verfahren zur Verdeckungsberechnung wieder zunehmend an Bedeutung, insbesondere im Bereich der Virtuellen Realität wo ein solches Verfahren eine Schlüsselrolle in einer potentiellen Lösung für das Latenzproblem einnimmt welchens momentan die größte Hürde auf dem Weg zu wahrlich realer Virtueller Realität darstellt. In dieser Arbeit widmen wir uns der Entwicklung eines massiv-parallelen Verfahrens zur Verdeckungsberechnung welches in der Lage ist, als Teil einer modernen Renderingpipeline effektiven Gebrauch der enormen Rechenleistung moderner Grafikprozessoren zu machen. Während wir uns in diesem ersten Schritt auf die Sichtbarkeitsberechnung von einem einzelnen Blickpunkt aus beschränken, ist unser Verfahren darauf ausgelegt, in Zukunft leicht auf die Sichtbarkeitsbereichnung für mehrere Blickpunkte erweitert zu werden. In einer extensiven Evaluierung zeigen wir, dass die Performance, Skalierbarkeit und Robustheit zu einer signifikanten Verbesserung des aktuellen Stands der Technik beiträgt.

Acknowledgements

I would like to thank Markus and Dieter for being extremely patient with me and allowing me to finish this work even though it took much longer than anticipated.

Apollo I. Ellis for being open to sharing their SVGPU source code and helping set it up.

My mum for investing time and money in my education and especially for making me learn English although I was convinced I'll have no use for it.

My dear Mikey for being with me through this wild ride.

Herta and Werner for providing food, shelter and no judgment during the last leg of the journey.

ReactiveReality and Philipp Grasmug personally, for seeing me as a capable person when I couldn't see it myself.

Myself for getting a therapist and my therapist for doing her job well, otherwise I would have given up on ever finishing this.

Contents

1. Introduction **13**

2. Method Analysis **17**
2.1. Pipeline 18
2.2. Visibility Algorithm Requirements 19

3. Related Work **23**
3.1. Traditional Visibility Methods 25
3.2. Visibility From a Region 26
3.3. Visibility From a Point 27
3.4. Summary 29

4. Geometry **31**
4.1. Basics 31
4.2. Plane 31
4.3. Polygon 33
4.4. Convex Polygons Overlap 34
4.5. Depth Relation 35
4.6. Clipping 36

5. Visibility Algorithm **37**

6. Implementation **49**
6.1. Megakernel 49
6.1.1. Scheduling Logic 50
6.2. Geometry Stage 53
6.3. Visibility Stage 53
6.3.1. Occluder Storage 54
6.3.2. Occlusion Test 56
6.3.3. Occluder State Update 57
6.3.4. Initialization 58

7. Evaluation **61**
7.1. Effect of Algorithm Parameters and Various Optimizations 64
7.1.1. Number of Thread Blocks 64
7.1.2. Grid Size 65

7.1.3. Sorting 66
7.1.4. Occluder Bounds Optimization 66
7.1.5. Occluder State 68
7.1.6. Discussion 74
7.2. Comparison with SVGPU 74
7.2.1. Quality 76
7.2.2. Performance 83
7.2.3. Discussion 85

8. Conclusion **89**

Appendices **93**

Appendices **95**

A. Overview Tables **95**

B. SVGPU parameters **99**

1. Introduction

With the advent of consumer *head-mounted display* (HMD) hardware, *virtual reality* (VR) is slowly but surely becoming a part of mainstream entertainment. However, driving an HMD to create an immersive VR experience presents a host of challenges in the field of real-time graphics, some new, some old, but yet unsolved. The two issues at the core of these challenges are latency and the sheer amount of data that must be produced in order to achieve the experience at least comparable to the level of PC or console.

Latency not only has a direct influence on how immersive the user's experience will be but, for some users, it can determine whether they will be able to have a VR experience at all due to onset of simulator sickness. Although sometimes not taken seriously by those who have not experienced it, simulator sickness can turn a fun activity into a nightmare. There is an ongoing debate on whether motion sickness and simulator sickness are the same thing, and two main theories explaining the latter. The cue conflict theory [RB75] explains simulator sickness as a result of conflicting sensory cues, such as e.g. signals of the body moving from the visual system and not moving from the vestibular system. The Treisman's theory [Tre77] argues that misaligning body signals are being interpreted by the brain as the body being intoxicated, and nausea and malaise serving as the early warnings. Despite the disagreement on the origin of motion sickness, there is a general agreement [KBS95] that visual cues play an important role in the unpleasant experience.

One major parameter to quantify to which degree visual cues are misaligned in the context of a VR experience is *motion-to-photon latency* : the time delay between the user's action (such as a head movement) and the moment the effect caused by the action can be perceived (photons, emitted by the screen, showing the correct view reach the player's eyes). Let us look at one example: the user that is standing still at a moment t_{still} starts to turn their head at t_{start} and stop at t_{end}, the latency is e. That means that at the moment t_{still} the person sees images corresponding to t_{still}. At the beginning of movement, t_{start} the person still sees the images from t_{still}. After some time, at $t_{start} + e$ the images still correspond to t_{still}, as if the person was not moving. The same happens at the end of movement: from t_{end} till $t_{end} + e$ the images correspond to turning of the head, although the real person is standing still. High motion-to-photon latency means that the visual cues are not synchronized with the vestibular cues during significantly long time intervals, which can be one of the causes of simulator sickness [Duh+04].

Although low latency is important for the PC gamers, it is even more important for the VR players. Gaming experience on a PC and in a VR headset are different in the way we interact with the virtual world, especially the ways in which we control the camera. Playing PC games, users typically use a mediator (a mouse, a keyboard or a game controller) to look around. The usage of the mediator is explicit, and some latency in the response is something that one can quickly get used to and that can be tolerated without any side effects for the real person. The headset can, too, be seen as a mediator, however, unlike the PC screen, it is supposed to disappear from the user's conscious mind. And, unlike the display that is stationary and the rest of the world provides an "anchor" to the real world, an HMD necessarily covers the user's entire view. The visual cues can only be delivered from the images presented to the user. When the motion-to-photon latency is high, the visual cues received by the eyes in VR do not correlate with the vestibular cues. And, as we have seen, disagreement in visual and vestibular cues is thought to be one of the causes of simulator sickness.

Motion-to-photon latency consists of many small delays that add up. In any interactive application latency can be broken down into two main parts: the time it takes from the moment the user performs some action that should register as an input until that input is actually registered by the system, and the time between registering the input until an image that reflects the consequence of the user's input is actually visible on the display Some people notice motion-to-photon latency of as little as 13 milliseconds [Bas+17]. Even if all other latency is 0, the device would have to run at the refresh rate of at least 77 Hz in order for the latency to not be noticeable for those people. Current high-end HMDs use displays are already capable of refresh rates of 90 Hz (Oculus, HTS Vive, Microsoft Mixed Reality), 120 Hz (Playstation) or even 144 Hz (Valve).

The time between the moment the output image is produced until it is displayed depends on the channel over which it is sent as well as the amount of data to be sent. In contrast to computer games, in virtual reality not one, but two images have to be generated to enable stereo vision, which doubles the amount of data. Those images are different, since each eye sees the world from a different perspective.Deriving or approximating one image from another will fail to create the binocular parallax effect, the main visual cue that creates depth perception.

Traditional PC monitors are stationary and have clearly defined borders that we accept as a natural frame defining the view frustum. It can be perceived as a window into the virtual world, and, as we are generally used to looking at things through windows, it is easy to get accustomed to the limited field of view of a monitor. One of the goals of virtual reality experiences is to eliminate the perception of the monitor. Thus, the displays in an HMD must cover the whole natural field of view (total of 180° horizontally, or about 120° per eye [SRJ11] while a typical PC monitor covers only about 90°. Since HMD displays are located much closer to the user's eyes, this means not only a larger image overall for the increased field of view, but a significantly higher image resolution to keep the angular resolution at the same level as PC monitor in order to keep individual

pixels indistinguishable. For example, between a monitor, that covers 90° to a monitor, that covers 120°, the screen resolution must increase by more than 50%.

When we put all of the above facts together, we find that VR demands at least 50% higher frame rates and two images instead of one for a 30% larger field of view which translates to an increase in resolution of at least 50%. Thus, simply producing an acceptable VR experience that is similar in visual quality to what we are used to seeing from classic interactive media such as video games requires about 4× the amount of image data to be generated for VR. While one may expect a future high-end desktop *graphics processing unit* (GPU) to be up to the task of rendering AAA content at the rates required for VR, placing a system with such a GPU directly inside a headset will, unfortunately remain infeasible for the foreseeable future due to weight and power constraints. Using a high-end desktop PC to run the application and perform all the rendering off the headset allows for the best possible visual quality but requires that inputs such as head pose and gaze point be transmitted from the HMD to the PC before an updated image can be rendered. The resulting image, once rendered, then has to be transmitted back to the HMD. This larger round trip results in added latency. Furthermore, transmitting data to an HMD at the required bandwidths will generally still require a wired connection which can severely impact the VR experience. The resulting tethering limits movement and cables present a potential fall hazard to a user in VR.

Alternatively, mobile-class processors found in devices such as smartphones can be used to run the application and render images directly on the headset. Some solutions such as the Samsung Gear VR or Google Cardboard even work by directly strapping a smartphone in front of the user's eyes. Other products such the Oculus Go or Lenovo Mirage Solo are standalone devices. While this approach avoids the round-trip latency from running the application on a separate machine as well as any need for a tether, the visual fidelity that can be achieved is generally very limited due to the restrictions of running on a low-power mobile GPU.

The approach taken by most current high-end consumer VR systems such as the Oculus Rift, HTC Vive, or Valve Index is that of rendering on a separate desktop PC. To speed up rendering and, thus, reduce at least part of the overall motion-to-photon latency, techniques such as foveated rendering and image warping can be employed. Foveated rendering decreases the computation time and the image size by lowering the image resolution of the parts of the image that cover the user's peripheral vision. Since the human eye has reduced resolution in the peripheral, the user usually does not notice the decreased image quality. Note that this technique requires an eye-tracking mechanism which is currently only present in some headsets.

Another set of techniques to somewhat reduce the perceived motion-to-photon latency is based around image warping and reprojection. Early forms of such approaches were described by John Carmack in 2013 [Car13]. The idea is simple: rendering of a frame takes some time and the resulting image will be outdated by the amount of time passed

during rendering. Simple geometrical reprojection of the frame to match the most current head tracking information creates an illusion of significantly smaller latency. This method works well for head rotations, but not so well for head translations, since it can't reproduce the parallax effect. It is also not suitable for the scene that contain moving or animated objects, since the delays in animation will quickly become obvious.

Asynchronous Timewarp (ATW) is the warping technique used to compensate for the dropped frames. Namely when a frame doesn't finish on time, it is dropped and the old frame is reprojected to match the current head pose instead. Because ATW is essentially the same method as the original warping, the problems stay the same: it doesn't works well for head translations and moving objects.

Asynchronous Spacewarp (ASW), also called *Motion Smoothing*, is meant to compensate for head translations. It extrapolates a new frame using differences between previous frames. Since it only considers color information, without depth, it often produces noticeable artifacts.

All warping methods have one thing in common: they do not address the underlying problem of high latency, only mask it. Future versions of warping techniques will likely incorporate depth information to reduce the amount of artifacts, but the warped frame would still be an approximation of how the real frame would have looked like.

From everything mentioned above, it is clear that high latency and the large amount of data are big obstacles on the path to the virtual reality world. Simply put, the problem is the following: a pair of images need to be rendered at high resolution and sent to the HMD with as small motion-to-photon latency as possible. So far we have only seen attempts to approximate the result by using old methods with heuristics. In this work we propose using a new method of producing and delivering frames that relies on analytical visibility. In particular, we investigate the possibility of computing analytical visibility efficiently on the GPU.

2. Method Analysis

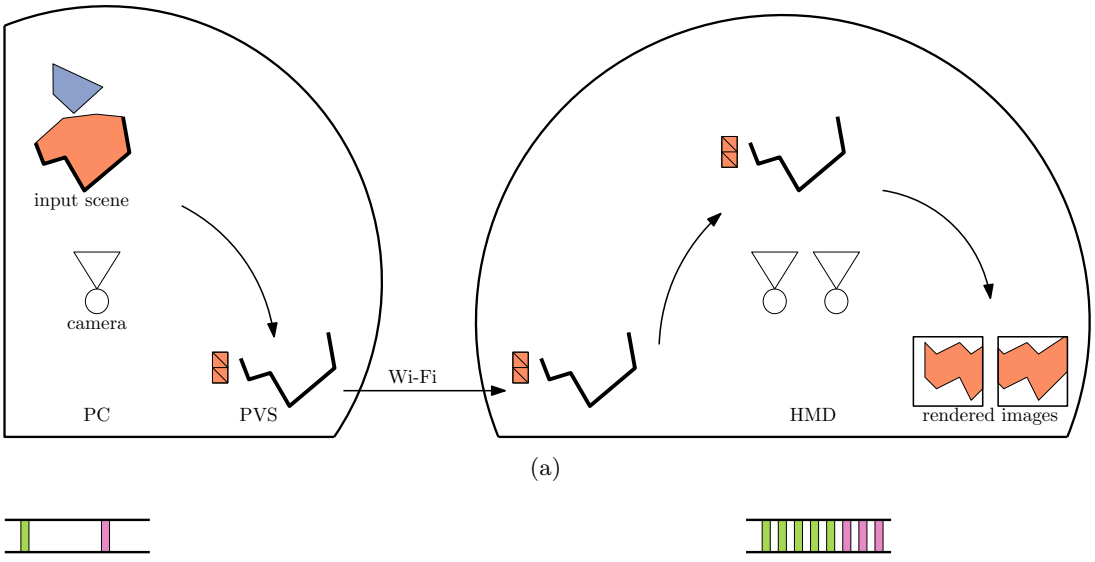


Figure 2.1.: Schematic illustration of proposed solution to the latency problem. (a) visibility is computed on a powerful PC, then visible geometry is sent to the headset over a wireless communication channel. The headset receives the PVS and renders the received geometry for two camera positions, generating an image for each eye. (b) frequency at which the PVS is computed (left) and displayed (right). A PVS is computed ones (one green bar on the left) and is used to produce images multiple times (multiple green bars on the right).

Ideally, in order to minimize latency, a VR headset would be able to render frames directly using the latest head-tracking information. To do that, the headset must be aware of the scene geometry. We propose to analytically compute a set of potentially visible triangles, the *potentially visible set* (PVS), and to perform the shading of those triangles on the server. The shaded PVS that consists of the original unprojected geometry is then streamed to the client. To render a frame, the client projects the latest received PVS using the current head position. Basic projection and rasterization can be done reasonably quickly even on a low-power mobile GPU and, since the headset can use the most-recent head tracking data, the latency will be as small as the duration of this final rendering step. A simplified illustration of the proposed pileline can be seen on Figure 2.1.

Since streamed PVS contains original geometry, projecting it creates a correct frame, not an approximation. As it contains *potentially* visible geometry, it can be reused for multiple frames beyond the one it was generated for, resulting in a correct image each time. Final rendering and visibility computation happen asynchronously and headset's frame rate is independent of the frequency at which the server produces PVS. The server can spend more time on visibility resolution and shading, which enables complex state-of-the-art shaders. View dependent shading, of course, can not be done on the server and must either be omitted, or performed on the headset. Geometry information, being just a list of triangles, is very compact and shading information need only be updated for the new visible geometry, thus PVS and texture updates will be relatively small, much smaller than two full-resolution images. This potentially allows the throughput requirements for the server-client communication channel to become so low, that even a wireless connection would be sufficient. Being analytical, the proposed method is also resolution independent: the same PVS can be streamed to different devices with different screen sizes, and be rendered correctly on each of them. Together with small update size it opens doors for cloud gaming: all computations can happen in the cloud and the user would not need to own an expensive workstation, but just a headset.

The only problem with the proposed solution is the absence of a suitable analytical visibility algorithm. Since the widespread adoption of z-buffer for visibility resolution almost no research was done in the direction of analytical visibility for 3D scenes. Thus the main goal of this work was to demonstrate that computing visibility for 3D scenes analytically is possible and to find such a method, that can in the future be trivially extended to also compute potential visibility.

2.1. Pipeline

When designing a visibility algorithm it is important to consider the architecture which will be using it. So first we must look at the rendering pipeline and find such that will allow us to split the rendering between the server (gaming PC) and the client (head mounted display) such that visibility computation is performed on the server and the projection is performed on the client. Fortunately such architecture has already been demonstrated in "Shading Atlas Streaming" [Mue+18]. In their work the pipeline consists of the visibility stage, shading stage, encoding stage, networking stage, decoding stage and display stage. The first three stages happen on the server, the last two on the client and the networking stage connects the two. The display stage receives pose updates directly, minimizing the perceived latency and running at high framerate, re-using the shading and geometry information for multiple frames. The visibility and shading stages run at a low framerate. Mueller et al. work focuses on the shading of geometry, computing visibility by rendering the scene into an id buffer and predicting future camera position with linear interpolation. They have successfully demonstrated that decoupling rendering

and display is highly beneficial in VR context. We will split rendering into visibility stage and rasterization stage as well, but will focus on the visibility algorithm.

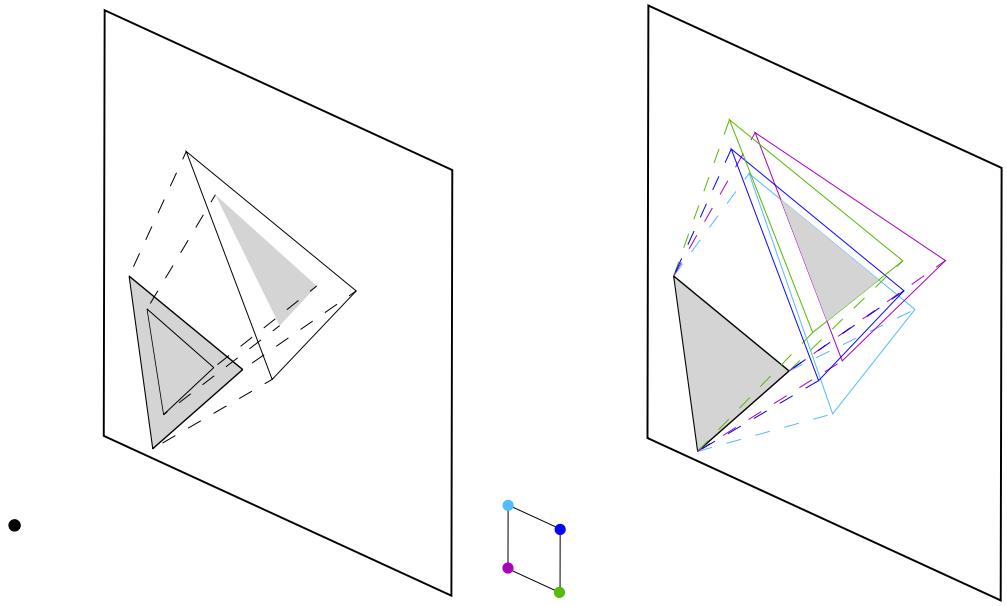
Modern game scenes are large and to be able to process the data efficiently we have decided to do it on a GPU. Especially since all scene data will sooner or later be sent to the GPU anyway. To be efficient, a GPU algorithm must be parallelizable. The primitives can not be processed in isolation, because visibility of one primitive depends on all other primitives, resulting in the $O(n^2)$ complexity. On the other hand, visibility of a triangle is a local characteristic, only nearby geometry affects it. We can exploit it by dividing the screen into areas for which the visibility is computed independently and merging the results. The simplest way to do so is to divide the screen into a regular rectangular grid. One such rectangular we call *a bin*. Having a stream of input geometry and a grid we need to somehow distribute the input into the bins. And thus we have two stages: *geometry stage* that sorts triangles into their bins and *visibility stage* that determines visibility of each triangle in each bin, and some mechanism to merge the results. This is commonly referred to as a *sort-middle* architecture [Mol+94]. Visibility and geometry stages, as well as multiple instances of each stage can run independently from each other. Since there are three clear steps in the process, it seems reasonable to implement three kernels that would perform given steps one by one. This would generate intermediate data between the stages and the size of this data will depend on the size of the input. Since the memory budget on the GPU is limited and is occupied not only by scene geometry, using this method it would be possible to process only relatively small scenes. Additionally, such multi-step approach is efficient when the GPU can be fully utilized. Unfortunately, since real world scenes do not typically fill the screen uniformly with geometry, it will likely result in an unbalanced load. We cannot avoid this disbalance, but if we can run different stages concurrently we could achieve better GPU utilization. We also would be able to process arbitrary scenes withing bounded memory, since one stage does not need to be fully finished and the intermediate results could be immediately used by the next stages. One way to implement such a software pipeline on a GPU is via a Megakernel approach which has been described, for example, by Steinberger et al. in “Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU” [Ste+14] and has been demonstrated for a graphics pipeline in particular by Kenzel et al. in “A High-Performance Software Graphics Pipeline Architecture for the GPU” [Ken+18].

2.2. Visibility Algorithm Requirements

The problem we are interested in has unique requirements. First of all, when we talk about visibility, we mean occlusion culling or detecting fully occluded polygons. We want the algorithm to run in real time, utilizing the processing power of the GPU, which in turn means it must be parallelizable. We want the output of the algorithm to be resolution independent. In order to be resolution independent it must be analytical. To

the best of our knowledge, no massively-parallel, analytical, real-time approach has yet been presented. Modern game scenes often contain non-manifold, self-intersecting meshes and visibility algorithm should be capable of handling such irregular, unconstrained geometry. It must support fully-dynamic scenes as well, which draws preprocessing of the input not possible. On top of that, we want a visibility algorithm that can be trivially turned into a potential visibility algorithm. Since the future PVS algorithm will necessarily overestimate visible geometry, we do not require the VS algorithm to compute exact visibility either. Recapitulating, the solution requires a massively-parallel, analytical, real-time visibility algorithm that can handle non-manifold, self-intersecting meshes and dynamic scenes. This algorithm should be trivially expendable into a PVS algorithm. Such algorithm does not yet exist. Since visibility problem is not new, in the chapter 3 Related Work we examine the visibility methods developed for other visibility problems, to potentially draw inspiration and borrow ideas suitable in our situation. In addition, for the algorithm to be incorporated into the overall VR system, we need it to have predictable resource requirements, e.g. perform within bound memory, and be an online algorithm to allow for stream processing in order to minimize latency.

One of the requirements we have for the visibility algorithm is possibility in the future to extend it into a potential visibility algorithm. Many visibility from a point algorithms can be extended in several ways, for example using occluder shrinking or visibility from a cell. Occluder shrinking means moving occluder vertices closer to each other by a given distance or percentage, which is a trivial operation in barycentric coordinates that keeps the original data unchanged. As illustrated in Figure 2.2a, shrinking an occluder results in a smaller occluder frustum, thus making more geometry visible in a similar way to slightly moving the viewpoint. Visibility from a cell, on the other hand, requires expanding the viewpoint. Figure 2.2b demonstrates expanding the viewpoint into a 2D cell for simplicity. Shadow frustum of the occluder becomes an intersection of 4 shadow frustums from 4 cell corners. This intersection can as well be generated using separating and supporting planes. Except for more complex occluder frustum computation, visibility algorithm can stay unchanged.



(a) (b)

Figure 2.2.: Strategies for changing a VS algorithm into a PVS algorithm. Original triangle and its projection onto a 2D plane are filled gray. Frustum lines are dashed. (a) occluder shrinking. The original triangle is "shrunk" so that the new triangle is smaller and lies in the same plane as the original. Occlusion (shadow) frustum of the new triangle is smaller and accounts for a small viewpoint movement. (b) occluder frustum of a triangle from a 2D view cell. Occluder frustum is an intersection of frustums created for each vertex of the cell. Each vertex and its frustum is shown in a different color. A 2D cell is used to demonstrate the idea, while in practice a 3D cell would be desired, the latter making the image hard to read. Projections of occluder frustums are shown instead of the whole frustums to keep the images easy to understand.

3. Related Work

Visibility problems are almost as old as computer graphics. At the beginning of the era of 3D computer graphics all geometry was drawn as a wireframe. Hidden line removal was the first type of visibility problem that needed a solution. Hidden line removal means removing invisible line segments or drawing them in a different style (e.g. as a dashed instead of a solid lines). Then, when rendering of solid surfaces became possible, hidden surface removal became an essential part of the rendering pipeline. It remains an important part of the pipeline to this day. Being such a core problem, it is not a surprise that a lot of research has been done on it over the years. So much, in fact, that there exist multiple works summarizing, classifying and giving an overview of the techniques.

Probably the first work to give a comprehensive overview and suggest a taxonomy of visibility algorithms was written by Sutherland et al. [SSS74] as early as in 1974. They have classified methods by the space in which the algorithms operate: object-space algorithms, that evaluate visibility in the original 3D coordinates of the object; image-space algorithms, that work in projected 2D space; list-priority algorithms that operate partially in image and partially in object space. Object-space algorithms resolve visibility before the rendering, on the object level, while image-space algorithms do it while rendering, on the pixel level. This fundamental difference, as the authors observed, is reflected in the performance of the algorithms. Performance of object space methods is a function of the scene complexity, while performance of image space method is limited by the image resolution.

Eventually hidden objects removal questions evolve into visibility questions. And what exactly one means when talking about visibility depends on the context. For example, the problem of hidden surface removal is fundamentally different from the problem of point-to-point visibility for radiosity. In fact, visibility problems are encountered in many different fields apart from computer graphics, such as robotics, computer vision or computational geometry. Every field presents different kinds of visibility problems and different types of visibility algorithms are used to solve them. Those algorithms are also used to solve problems at a first glance not related to visibility. Computing soft and hard shadows is one example. Question of which geometry is lit from a point light is dual to the question of which geometry is visible from a point and visibility from a point methods can assist or speed up the computation. In robotics, finding a path from a start position to the goal can also make use of visibility methods. The straight part of the robot's trajectory can be seen as an analog to the light ray. Both the robot and the light

can only reach areas that are "visible" from the start point. A broad interdisciplinary overview of visibility problems and the algorithms used to solve them was given by Durand in his PhD thesis [Dur99]. He also provides an updated classification of visibility algorithms by the space they operate in, this time expanding Sutherland's classification with viewpoint-space and line-space algorithms. Viewpoint-space he defines as a set of all possible viewpoints, for example for a parallel projection a viewpoint space is a set S^2 of directions, often called viewing sphere. Line-space is based on determining mutual visibility of two points: two points are visible to each other if no object intersects a line between them.

Kovalčík, on the other hand, presents a different classification method in his PhD thesis [Kov07]. It is based on certain properties of the methods, such as whether they require preprocessing, or work with static or dynamic scenes. He also considers whether a scene must be of a specific kind, whether the result is conservative or approximate, if visibility is computed from a point or from a region, in screen or in world space, if the occluders can be fused and whether they must be convex.

To our knowledge, the most recent taxonomy was presented by Bittner and Wonka [BW03]. It presents a complete taxonomy of state-of-the-art visibility algorithms. In contrast to previous classifications, it is based on the problem domain rather than on the space in which the problem is solved. They differentiate between: visibility along a line, visibility from a point, line segment, polygon and region, and global visibility. They also quantify complexity of the problem by giving its dimensionality. For example, visibility from a point is a 2D problem, while visibility from a polygon is a 4D problem, which shows that visibility from a polygon is much more complex.

Because visibility problems in general are very complex and solving them requires a lot of compute power, many algorithms were devised to solve specific scenarios. For example, if one can assume that the scene is static (which is usually the case in walk- or flythrough applications), then it can be preprocessed ahead of time. If a scene represents an indoor environment, one can take advantage of this knowledge divide it into *cells* and *portals* [ARB90; Jon71]. In real-time applications, the problem of hidden surface removal is mostly solved by the z-buffer [Str74; Cat74] today. The z-buffer has the big advantage of being able to handle fully-dynamic scenes without requiring any preprocessing. The film industry has yet different demands, where photorealism takes precedence over speed. Thus, methods based on ray-tracing are typically employed in production rendering, which implicitly solve the visibility problem.

Below we present a short overview of the existing methods relevant to our problem and reasons why they can not be used to solve that problem.

3.1. Traditional Visibility Methods

First we examine some of the most commonly used visibility methods.

Z-buffering is a technique described by Straßer [Str74] and Catmull [Cat74]. It has become a standard solution to resolve visibility from a point in rasterization pipelines. Due to its simplicity, it is implemented even in low-cost hardware nowadays. For each pixel in the framebuffer, the depth of the corresponding object surface at the respective location is stored in a buffer. During rasterization, the depth of each newly-rasterized pixel is compared to the depth of the pixel already in the buffer (if any). Only if the new pixel is closer in depth, the existing pixel is overwritten and the value in the depth buffer updated.Unfortunately, z-buffering is an image-space method based on rasterization and, thus, resolution-dependent. It is strictly a visibility form a point method and can not be trivially extended to compute visibility from a region.

List priority algorithms determine the appropriate order of input objects, such that, when drawn in that order, the result will be correct. Usually this means sorting elements back to front, with closer objects being rendered on top of further objects [NNS72]. This algorithm does not align with the streaming approach, as the scene needs to be taken into account in its entirety during processing. Many implementations also use specialized spatial data structures, such as binary space partitioning trees [FKN80]. Such data structures need to be built in a preprocessing step and continuously updating them for a dynamic scene, particularly on the GPU, is problematic.

Area subdivision algorithms recursively subdivide the image plane, until the decision which object is on top of which other object in each part of the image becomes trivial. Some algorithms subdivide the image in a predefined way (e.g. rectangles [War69]), while other follow the original geometry boundaries [WA77]. All kinds of area subdivision algorithms require efficient use of acceleration data structure, just like list priority methods, implementing which is beyond the scope of this project. Efficiently building, storing, and updating such data structure on a GPU is not a trivial task, and neither is updating it for dynamic scenes.

Ray casting was first developed by Appel [App68] for the purpose of digital printing. It has since then evolved into the large family of ray tracing algorithms that are only loosely related by the fact that they all are based on the idea of constructing rays (from either camera or light sources) in order to determine shading of pixels. It has become an industry standard when it comes to producing photorealistic images. Ray casting is an

embarrassingly parallel problem by nature, but the processing times that are required to achieve the realistic quality are high: hours are needed to render one still frame. The essence of those algorithms is discrete, and is bound by the number of shoot rays, so it will not be able to produce resolution independent results. Unfortunately, while being easily parallelizable, ray tracing algorithms are still far from being real time.

Scan line algorithms [Wyl+67] are extensions of scan conversion algorithms. They maintain a list of edges for each raster line, based on edge's y coordinate. Each list is also sorted by x coordinate. As visibility is solved per raster line, and within the raster line it is resolved per-pixel, this algorithm is also not suitable for producing resolution independent visibility information.

3.2. Visibility From a Region

The methods we have just discussed can neither solve our problem, nor be modified to do so. Probably that is because they are focused on the most common task in computer graphics, which is producing one image at a time for exactly one viewpoint. However, the visibility from a region problem has also been studied extensively. We can, potentially, take some lessons from existing work in this area.

Cells and portals were used by Aireyet al. [ARB90] to cull big portions of highly detailed indoor scenes. An inherent property of these kinds of scenes is that they have a clear structure of enclosed spaces with relatively small openings through which a limited portion of connected neighboring spaces is visible. Given the knowledge about the *cells* (rooms) and *portals* (doors, windows) of a scene, the information about which other cells are visible from any given cell can be precomputed. Using this information, most invisible parts of the scene can efficiently be culled at runtime. This method is often applied to games and architectural models, and the speedup, acquired through detecting cells and portals, is significant. The method was originally proposed by Jones [Jon71] in 1971. At this time the subdivision was done manually and no PVS were generated, but the graph of connected cells was traversed depth-first and the image rendered at the same time. Unfortunately, this method also does not alight with the streaming approach and can only be used with static scenes of a certain kind.

Extended projections suggested by Durand et al. [Dur+00] divides the scene into view cells and computes a PVS (potentially visible set) for each cell. It is a conservative method that works only with static scenes, and requires a preprocessing.

Voxelization Instead of dividing scene into view cells, Schaufler et al. [Sch+00] suggested to voxelize the scene. A voxel can be empty, opaque or boundary. Empty voxels contain no geometry, opaque are completely occluded, and the rest is boundary. Opaque voxels create occluded frustums, just like polygonal occluders usually do. A *blocker extension* operation combines multiple opaque cells to create bigger occluded frustum. Blocker extension happens at runtime, while voxelization octree is build at preprocessing. The method was mainly written for walkthrough applications, which use static scenes and allow for a costly preprocessing.

Plücker coordinates are often used in the visibility context as a way to represent geometric information in a form, that makes computation of intersection more natural and less expensive. For example, Bittner [Bit02] uses them to represent a set of directed lines that *stab* an occluder and a view cell. His algorithm uses a 5D BSP tree, where each node represents a subset of line space that stores 5D occluder polyhedra. Like the previous methods, it builds the tree at preprocessing. And as the author himself notes, "the exactness of the method requires higher computational demands".

Nirenstein, Blake, and Gain [NBG02] builds on the work of Bittner and extends his method to compute an exact visibility set from a view region. This method was also primarily written for walkthrough applications, but they also claim that it can be executed at runtime for visibility from a point, which I didn't really find/understand from the text.

3.3. Visibility From a Point

As we can see, the problem of visibility from a region has never been solved in a streaming way. So, as a first approximation, we have decided to narrow it down to a visibility from a point problem. As far as we know, a solution to that problem (a resolution-independent, parallelizable visibility from a point algorithm that supports funny-dynamic scenes and does not require preprocessing) has not been published yet either. The solution we design, nevertheless, has to be trivially expendable for visibility from a region. With this in mind let us look at more recent visibility from a point methods.

Cells and portals are also used for computing from-point visibility. Since the limitations of the algorithm stay the same, this method can not be used for our purposes.

Hierarchical methods, like hierarchical z-buffer [GKM93] or hierarchical occlusion maps [Zha+97] can be seen as an optimization of z-buffer algorithm. They all work

in image space and require preprocessing, thus can't be used to produce resolution independent and streaming result. Hierarchical occlusion maps in addition assume presence of *good potential occluders* - objects that are visible and cause significant occlusion. Multiple methods use the assumption, but, first of all, it usually implies detecting those good occluders in a preprocessing step, and second, does not fully support general scenes.

Shadow volume The idea of *good potential occluders* is based on the observation that most occluded geometry is usually occluded by just a few objects. Hudsonet al. [Hud+97] have made use of this idea by identifying some good polygonal occluders (at preprocessing) and computing their shadow volume. An object, that lies completely in a shadow volume, is then culled. Although Hudsonet al. method require a preprocessing step, and use of specialized data structures (the model must be represented in a spatial partition and a spatial hierarchy), it performs in the world space, which makes it resolution independent. One of the problems with that method is that many objects are not occluded by one occluder alone, but by a combination of them.

Visual events or visual event planes were used by Coorg and Teller [CT97], [CT99] to compute conservative visibility from a point. They have approached the classical problem from a different angle, and instead of computing "what objects are visible from a given viewpoint?", they answered the question "from which points is this object occluded?". They have introduces the notion of separating and supporting planes that subdivide space into three subsets of points, from which a given object is fully visible, partially visible or occluded by a given occluder. That has also allowed them to use connected polygons as occluders.This method requires a static scene, a preprocessing of that scene and use of kD-tree and look-up tables, which makes this method not directly applicable to our problem. But it has given us an insight and an inspiration to use planes and halfspaces in our own work. In the as-is state it can be beneficial in a game renderer, as the environment in games is usually static.

Modern GPU solutions One of the first attempts to doing visibility analytically and on a modern GPU was the one by Auzingeret al. in 2013 [AWJ13]. The main motivation for the method was to enable the implementation of analytic anti-aliasing. The method works in normalized device coordinates and is based on intersecting all projected scene edges with all projected scene triangles, followed by hidden line elimination. It is a multi-pass approach for computing visibility from a point that assumes non-intersecting scene geometry. There is no clear way to extend this algorithm to visibility from a region or to turn it into an online algorithm.

The most recent approach is SVGPU [EHH16]. SVGPU focuses on outputting a planar

triangle map that can be sent over the network and then rasterized on a client device. While the method by Auzinger et al. is based on hidden line removal, SVGPU uses a combination of silhouette extraction and clipping as well as a more sophisticated screen-space binning approach in combination with dynamic parallelism to achieve a better degree of utilization. One downside of this method is that complex scenes with many objects result in a very large number of triangles, which is not optimal for a network transmission. This becomes especially prominent when an object with a highly-detailed silhouette is located in front of a simple big object, such as a box, as the latter will be tessellated into many small parts. Another downside is that, as it produces a planar map for a given viewpoint, it cannot be used to produce visibility information for a range of different viewpoints.

3.4. Summary

None of the methods we have discussed presents an analytical, parallelizable visibility algorithm that complies to the streaming approach. All visibility from a region methods only work with static scenes and rely on heavy preprocessing. Therefore, those methods are inherently multi-pass approaches, and cannot be used in a streaming rendering pipeline. All of them rely on specialized spatial data structures to store scene geometry, which do not map well to the GPU. Methods, that solve the simpler visibility from a point problem, are subject to the same limitations: static scenes, preprocessing and spatial data structures. The most recent attempts at solving visibility analytically on a GPU are multi-pass and single viewpoint by their nature.

4. Geometry

Before talking about the visibility algorithm itself we need to discuss the context in which we use it and describe some general geometric operations.

4.1. Basics

Our visibility computation algorithm is intended to be used as a part of a graphics pipeline. Thus we assume the input to the visibility algorithm to be triangle vertices in clip space.

Assumption 1. *A* vertex *is a point in 3D space defined by a vector of its homogeneous coordinates* $\begin{bmatrix} x & y & z & w \end{bmatrix}^T$.

Definition 1. *An* edge *is a line segment connecting two vertices, defined by those vertices.*

Definition 2. *A* half-edge *is a directed line segment, defined by its start and end vertices.*

Any edge contains two half-edges.

Definition 3. *A* plane *is a flat, two-dimensional surface that extends infinitely far.*

Given a polygon P function vertices(P) returns an ordered set of its vertices. A polygon is *backfacing* if its vertices are oriented clockwise around z-axis and *frontfacing* otherwise. Function edges(P) returns a set of this polygon's edges.

4.2. Plane

A *plane* is defined by its normal vector and the distance along the normal from the origin. A plane divides space into two halfspaces: a *positive halfspace* and a *negative halfspace*.

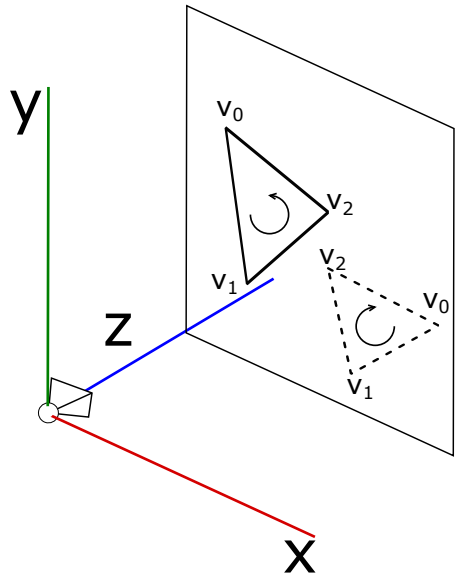


Figure 4.1.: Left-handed coordinate system with z axis pointing "away" from the camera, a clock-wise and a counter-clockwise oriented triangle. Counter-clockwise oriented triangle is front-facing (drawn in solid lines), clockwise triangle is back-facing (dashed lines).

Definition 4. *A* halfspace *is one of the two subspaces created by a plane.*

A point belongs to the positive halfspace ($\text{halfspace}_{>0}$) if the distance from that point to the plane is positive and to the negative if the distance is negative. A halfspace is *closed* if the points of the plane belong to it, and *open* otherwise. In other words, given a point $\mathbf{v}$ and a plane $\mathbf{p}$

$$\mathbf{v} \in \underset{\geq 0}{\text{halfspace}}(\mathbf{p}) \iff \text{dist}(\mathbf{v}, \mathbf{p}) \geq 0 \tag{4.1}$$

$$\mathbf{v} \in \underset{\leq 0}{\text{halfspace}}(\mathbf{p}) \iff \text{dist}(\mathbf{v}, \mathbf{p}) \leq 0 \tag{4.2}$$

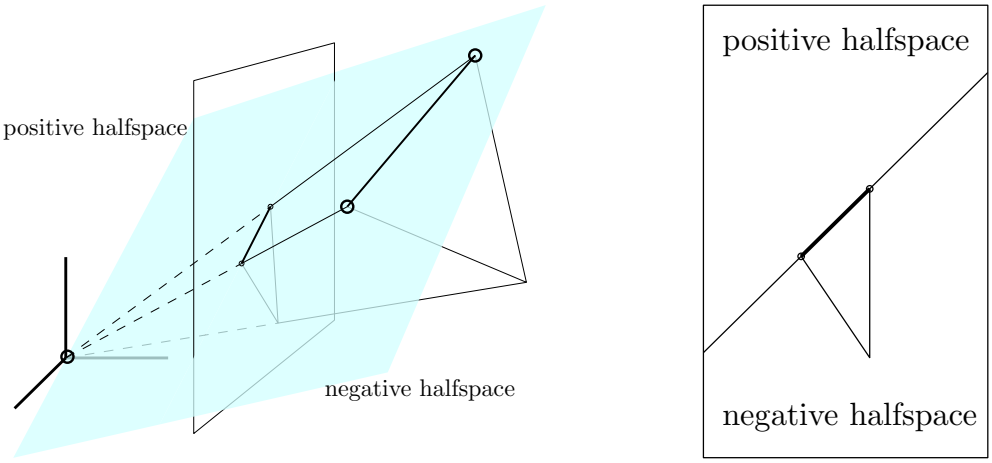
$$\mathbf{v} \in \underset{> 0}{\text{halfspace}}(\mathbf{p}) \iff \text{dist}(\mathbf{v}, \mathbf{p}) > 0 \tag{4.3}$$

$$\mathbf{v} \in \underset{< 0}{\text{halfspace}}(\mathbf{p}) \iff \text{dist}(\mathbf{v}, \mathbf{p}) < 0 \tag{4.4}$$

The distance between a point $\mathbf{v}$ and a plane $\mathbf{P}$ can be computed as following:

$$\text{dist}(\mathbf{v}, \mathbf{P}) = dot(\mathbf{P}.n, \mathbf{v}) + P.d \tag{4.5}$$

A plane can be defined by 3 points. Given the points $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ we can compute the plane's



(a) (b)

Figure 4.2.: A triangle in 3D space, its projection onto an image plane and an edge-plane defined by one of the triangle edges. (a) The edge-plane (semi-transparent blue) is dividing the 3D space into two halfspaces, the original triangle is completely in the closed negative halfspace. (b) The edge-plane intersects the image plane as a line that passes through the projected edge. This line divides the triangle plane into two halfplanes, with the projected triangle lying in the closed positive one.

normal and distance using the cross and dot products with the formulas 4.7 and 4.6.

$$\mathbf{P}.n = \frac{(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})}{\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|} \tag{4.6}$$

$$\mathbf{P}.d = -\langle \mathbf{P}.n, \mathbf{a} \rangle \tag{4.7}$$

A plane can also be defined for an edge, using the edge start and end points and the origin. Given an edge e, the distance and normal of its plane are computed using formulas 4.7 and 4.6 with $\mathbf{a} = start(e), \mathbf{b} = end(e), \mathbf{c} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$ An example of a plane through a triangle's edge is shown on the figure 4.2a, as well as the positive and negative halfspaces it creates.

4.3. Polygon

Definition 5. *A* polygon *is a plane figure with n sides (as defined in* VNR Concise Encyclopedia of Mathematics *[Gel+90]).*

A polygon can be defined by an ordered sequence of points lying in a plane, a set of edges or directed edges (half-edges) lying in a plane. If defined by points, every two consequent vertices (including the pair of the last and the first vertex) create a polygon edge. Defining a polygon by vertices or half-edges allows to determine its orientation: *clockwise (CW)*, or *counter-clockwise (CCW)* around the positive z-axis (fig. 4.1). Triangle is the simplest example of a polygon.

Definition 6. *A* convex polygon *contains all the line segments connecting any pair of its points [Gel+90].*

Given a convex polygon P and two points p, q that belong to it:

$$\begin{aligned} &\forall \mathbf{p}, \mathbf{q} \in P, t \in [0,1] \\ &tp + (1-t)q \in P \end{aligned} \tag{4.8}$$

One of the convex polygons' properties is that a convex polygon is entirely contained in a closed half-plane defined by each of its edges.

4.4. Convex Polygons Overlap

Definition 7. *Two polygons that lie in the same plane* overlap *iff the set of points that belong to both polygons is not empty.*

Definition 8. *If 2 convex shapes do not overlap, then there exists a* separator plane*, such plane that objects lie on the opposite sides of it [CT99].*

Although we are working with a 3D projective objects, we are interested in the properties and relations between their 2D projections. Input geometry is defined in clip space, which is 3D projective space. A point coordinate is thus 4-dimentional [x, y, z, w]. It can be projected into the cartesian 3D space: [x/w, y/w, z/w]. It can be rasterized into 2D image: [x/w, y/w].

We can either project the geometry and see if the property holds or find a *dual property* in the projective space. Such property, that if a dual property holds for geometry in projective space, the original property holds for the original geometry after projection.

A plane defined by an edge projects onto 2D image plane as a line and divides the 3D space into positive and negative halfspaces and image plane into positive and negative

halfplanes as shown on the figure 4.2b. A convex polygon on a plane divides it into the regions inside and outside itself and in 3D it divides the space into a frustum inside and outside. If we have two convex polygons in 3D and need to know whether their projections overlap, we can compute their projections and check in 2D, or we can check if their frustums in 3D space. If the frustums intersect, the projections intersect as well.

To determine the opposite — whether the projections of the polygons do not intersect — we can search for a separator line in 2D or a separator plane in 3D. Given two polygon projections, A and B, all points that belong to A lie in the negative halfspaces of all $edges(A)$ and the same can be said about B. When polygons overlap, there exist at least one point p that belongs to the negative halfspaces of all $edges(A)$ and all $edges(B)$. Conversely, when polygons do not overlap, no such point exists. Thus there exist at least one edge among $edges(A)$ and $edges(B)$ such that A and B lie in its separate halfspaces.

Thus, given convex polygons p_1 and p_2, their projections don't overlap iff

$$\begin{aligned} &\exists e_1 \in \text{edges}(p_1), \forall v_2 \in \text{vertices}(p_2) : v_2 \in \underset{\geq 0}{\text{halfspace}}(e_1) \\ \vee\ &\exists e_2 \in \text{edges}(p_2), \forall v_1 \in \text{vertices}(p_1) : v_1 \in \underset{\geq 0}{\text{halfspace}}(e_2) \end{aligned} \tag{4.9}$$

4.5. Depth Relation

Any polygon lies in a plane, which divides space into 2 halfspaces. If another polygon lies completely in one of those halfspaces the depth relation is clear and easy to determine: if the first polygon lies in the negative halfspace of the second, the first polygon is below the second, otherwise it is above the first polygon. Speaking in a more precise manner: Given two polygons p_1 and p_2, polygon p_1 is above p_2 if

$$\begin{aligned} &\forall v_1 \in \text{vertices}(p_1) : \text{dist}(v_1, \text{plane}(p_2)) \geq 0 \\ \vee\ &\forall v_2 \in \text{vertices}(p_2) : \text{dist}(v_2, \text{plane}(p_1)) \leq 0 \end{aligned} \tag{4.10}$$

Polygon p_1 is below p_2 if exactly opposite holds:

$$\begin{aligned} &\forall v_1 \in \text{vertices}(p_1) : \text{dist}(v_1, \text{plane}(p_2)) \leq 0 \\ \vee\ &\forall v_2 \in \text{vertices}(p_2) : \text{dist}(v_2, \text{plane}(p_1)) \geq 0 \end{aligned} \tag{4.11}$$

If none of above holds, or

$$\begin{aligned} &\exists v \in \text{vertices}(p_1) : \text{dist}(v, \text{plane}(p_2)) \geq 0 \\ \wedge\ &\exists v \in \text{vertices}(p_2) : \text{dist}(v, \text{plane}(p_1)) \leq 0 \end{aligned} \tag{4.12}$$

it means that polygons either intersect or their relation cannot be determined.

To compute an intersection point of an edge and a plane we use the fast triangle-triangle intersection method presented by Möller [Möl97]. Given a plane $\mathbf{p}$ and two points $\mathbf{v}_0$ and $\mathbf{v}_1$, the intersection point $\mathbf{x}$ of the edge, defined by those points and the plane is computed as following:

$$\begin{aligned} d_0 &= \text{dist}(\mathbf{v}_0, \mathbf{p}) \\ d_1 &= \text{dist}(\mathbf{v}_1, \mathbf{p}) \\ \mathbf{x} &= \mathbf{v}_0 + (\mathbf{v}_1 - \mathbf{v}_0) d_0/(d_0 - d_1) \end{aligned} \tag{4.13}$$

4.6. Clipping

From the definition of convexity (section 4.3) it follows that the intersection of any convex polygon with a plane consists of at most one line segment. For there to be an intersection at all, at least one point of the polygon must be on a different side of the plane than the rest of the polygon. Therefore, when clipping a polygon against a half-space, at least one point of the polygon is removed. Edge, created by the clipping, can create at most two new vertices, the beginning and the end of a new line segment. Thus, the result of clipping a convex polygon of N vertices with a plane will be a polygon with not more than $N + 1$ vertices. Intersection of two convex sets is a convex set. Convex polygons and half-spaces are convex sets as well. Thus the result of clipping a convex polygon with a half-space is a convex set.

5. Visibility Algorithm

Existing visibility computation algorithms do not satisfy the requirements we have identified earlier, namely, an online, analytical method that can solve visibility from a region while exposing sufficient parallelism to allow for an effective GPU implementation. As we have shown in chapter 2, a method that covers visibility from a point can be extended to solve visibility from a region by working with occluder shapes reduced to their umbra, or a (conservative) approximation thereof in the form of occluder shrinking. Thus, the focus of this work will be on developing an algorithm that can solve visibility from a point in a way that satisfies the criteria for use in a streaming VR rendering pipeline as previously defined. The extension of the algorithm we are about to present to incorporate occluder shriking or a similar method will be left to future work.

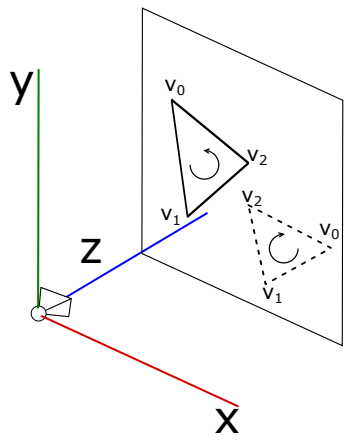


Figure 5.1.: Left-handed coordinate system with z axis pointing "away"

Without loss of generality we assume the input geometry to be defined in a left-handed coordinate system as depicted in Figure 5.1 with the viewer looking down the positive z axis (objects further from the observer have larger z value than closer objects). Clockwise oriented polygons are considered backfacing and, thus, invisible. Visibility algorithm input is expected to be defined in 3D projective coordinates, since visibility stage follows geometry stage (vertex shader). The output is a set of ids of the primitives determined visible. We assume no other prior knowledge of the input such as its size or order of polygons. We do not perform any preprocessing of the input.

Before the visibility can be computed, the input must be transformed into clip space.

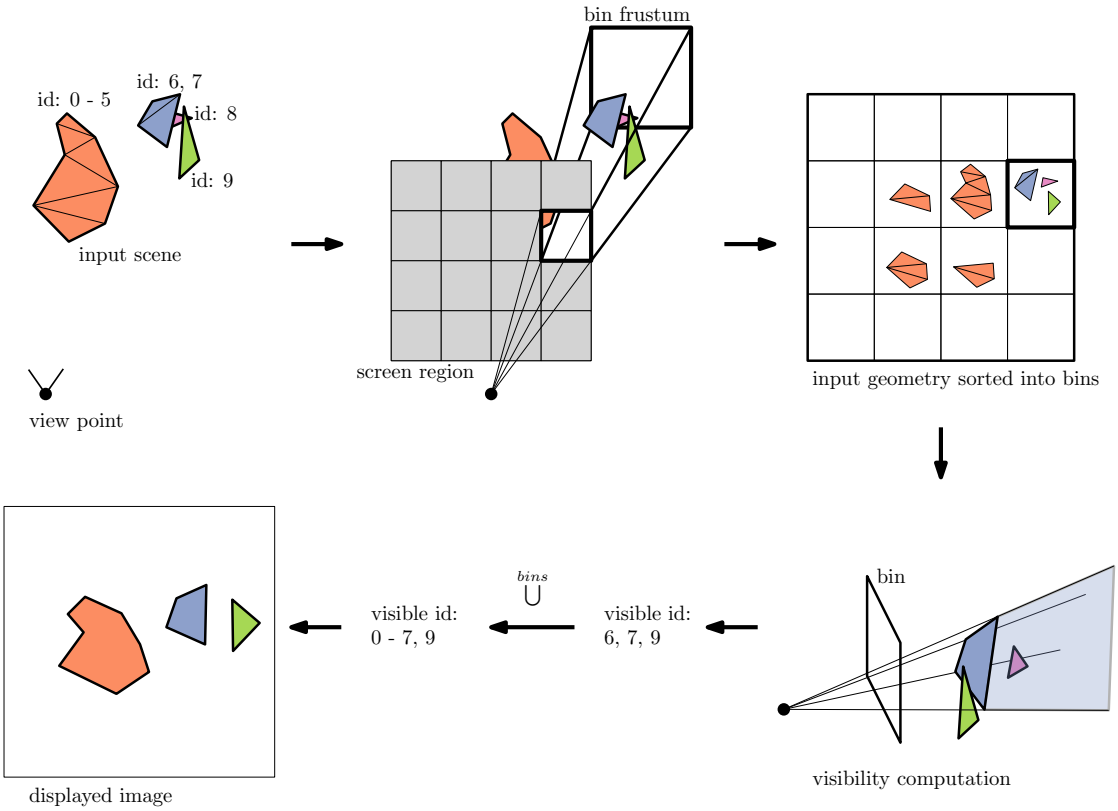


Figure 5.2.: Scene processing steps from input geometry to visible set to the final image. (1) Input geometry and the viewpoint. (2) Input geometry in clip space. Image plane is divided into bins and one bin's frustum is displayed. (3) Input geometry is distributed into bins. A triangle is added to every bin it intersects with. (4) Visibility is computed on a per-bin basis. (5) Visibility set of a bin is ids of the triangles visible inside this bin. (6) Visibility set of the whole scene is a union of bin visibility sets. (7) Final image, created by projecting the visible set onto the image plane.

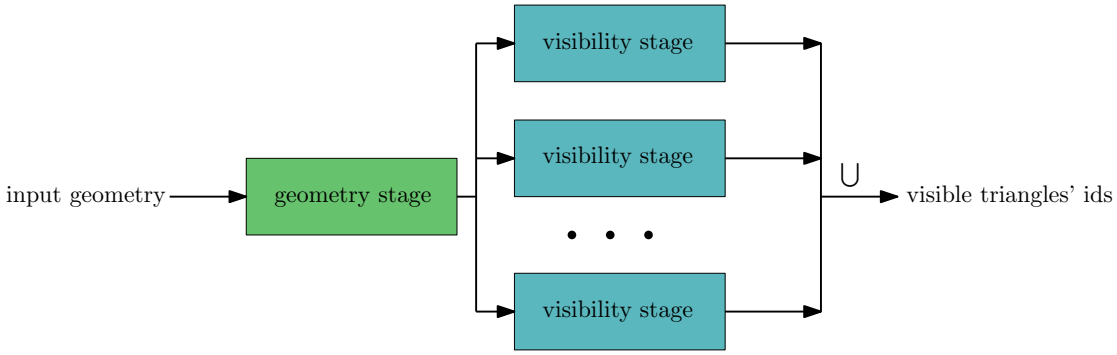


Figure 5.3.: Schematic diagram of the pipeline. Every block of megakernel at any point in time is running either a visibility or the geometry stage.

As discussed previously, in order to parallelize the algorithm, the screen is divided into rectangular bins and the input is distributed into them. Although we describe the algorithm in terms of triangles, it is applicable to all convex polygons and any convex tiles can be used for defining the bins. A triangle is added to a bin if any part of the triangle intersects with the bin frustum. After some geometry accumulates in the bin, the visibility test is performed between the accumulated geometry and the bin's internal state. Visibility test consists of detecting whether the new geometry is occluded by previously seen geometry, the occluders stored by that particular bin. After the visibility test is performed, a triangle is either deemed visible or occluded. An occluded triangle is discarded. A visible triangle can be added to the bin's occluders either as a new occluder or as a part of an existing occluder. It potentially merges some occluders as well. The process is illustrated on the Figure 5.2. Figure 5.3 illustrates the how visibility and geometry stages are connected. Geometry stage produces input to the visibility stages and the Megakernel blocks decide which stage to run based on the amount of input available for every stage.

Internal State As previously mentioned, every bin manages its internal set of occluders. Each occluder is defined by a set of its triangles and a set of its edges (Figure 5.4). Occluder edges are interchangeably referred to as a silhouette or silhouette edges. Two triangles that share an edge belong to the same occluder. Occluders are built of triangles considered visible and they grow during the runtime, but not all visible triangles are kept as occluders, since the memory budget is limited. We have experimented with occluders made of triangles that share a vertex as well, but the results were not satisfactory.

Simplest occluder consists of a single triangle along with its 3 edges. Occluder edges are, in fact, half-edges (directed edges). If an occluder consists of a triangle with vertices v_0, v_1 and v_2, occluder edges will be defined as edges from v_0 to v_1, from v_1 to v_2, from v_2 to v_0 (Figure 5.5).

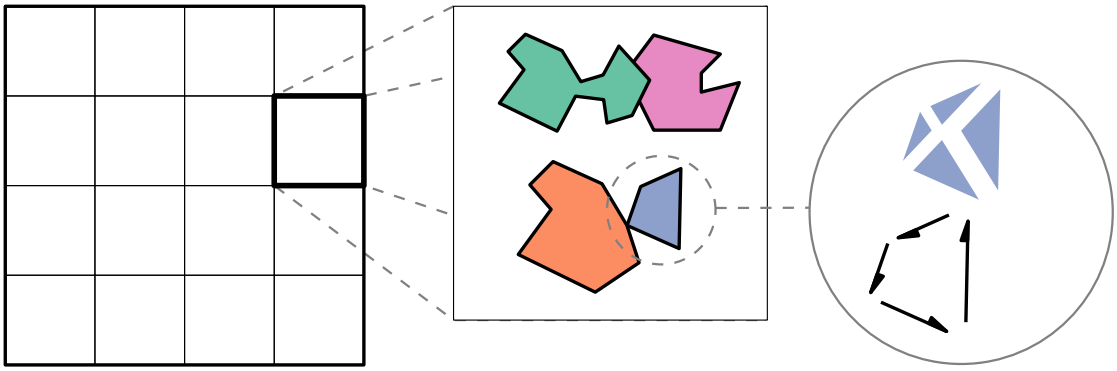


Figure 5.4.: Internal state of the system. Each bin contains a set of occluders, each occluder is defined by its triangles and silhouette edges.

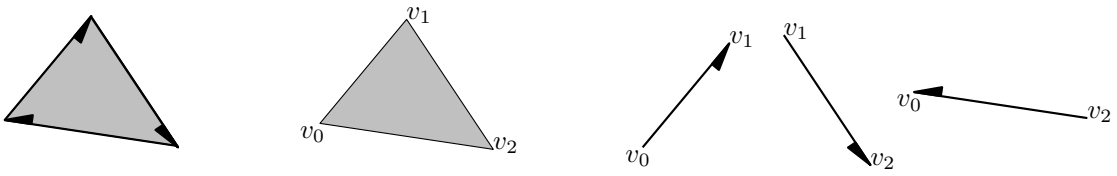


Figure 5.5.: Simplest occluder. Occluder triangles set contains a single triangle $v_0v_1v_2$. Occlude silhouette is edges v_0v_1, v_1v_2, v_2v_0.

Occluder Initialization Although the algorithm can begin with empty bins, it can be reasonable to initialize it. A simple initialization procedure takes a number of input triangles and combines those that touch (share an edge) into occluders. Shared edges are inverses of each other since front facing triangles are oriented in the same way. Not shared edges make the occluder silhouettes. An occluder that consists of three triangles is illustrated on the Figure 5.6). If a bin is not initialized, the first input triangle becomes the first occluder, since there is nothing to occlude it.

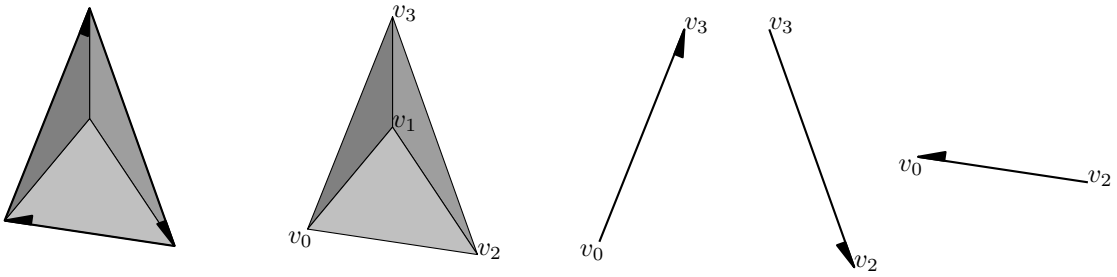


Figure 5.6.: Occluder initialized from three connected triangles. Occluder triangles contain all three triangles: $v_0v_1v_2$, $v_0v_3v_1$, $v_1v_3v_2$. Occlude silhouette contains only three outer edges: v_0v_3, v_3v_2, v_2v_0.

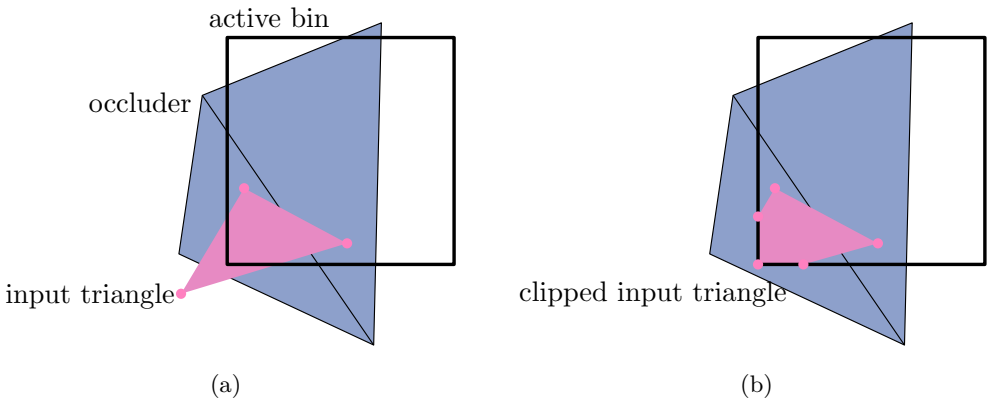
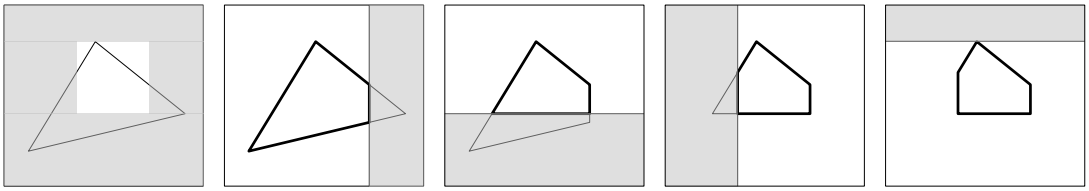


Figure 5.7.: (a) An input triangle that protrudes outside of the bin borders and occluder silhouette. Assuming this triangle lies behind the given occluder, it is occluded within the given bin. It might or might not be occluder outside of the bin. (b) After clipping the input triangle against the bin borders, its visibility within the bin can be evaluated correctly.

Clipping Against Bin Both the input triangle and the occluders can extend beyond the bin borders. Within one bin we are only interested in whether the part of the input triangle inside that bin is occluded and we only have information about geometry within the bin. We cannot know whether the part of the triangle outside of the bin is occluded or visible. Other parts of the triangle will be evaluated within other bins. As discussed earlier in this chapter, visibility of a triangle is a union of its visibility in all bins it falls into.

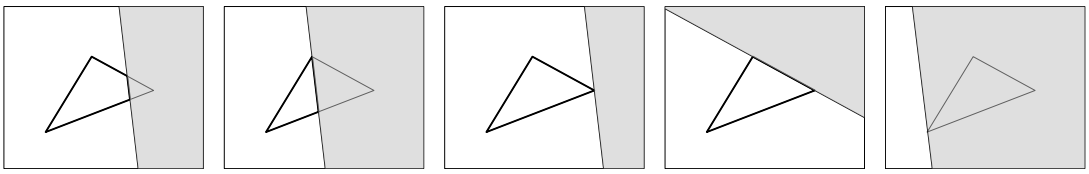
An example on the Figure 5.7 shows an input triangle that is occluded within bin boundaries, but not occluded by the same occluder outside. After clipping the triangle with the bin borders the visibility *within* the bin borders can be evaluated correctly. The occluders extending beyond the bin borders, on the other hand, do not affect the visibility inside the bin. Clipping them will not improve accuracy of the algorithm, but introduce additional workload. Thus the occluder triangles are not clipped.

A screen bin is defined in 2D screen coordinates by a rectangle. In the projective space, this rectangle corresponds to a frustum defined by four planes. Clipping a convex polygon against a frustum is equivalent to clipping it against each of the planes of the frustum one at a time (Figure 5.8). The resulting polygon is convex since clipping a convex polygon against a plane is essentially the same as taking a union of two convex sets. If a convex polygon intersects with a plane the intersection is a straight line (Figures 5.9a, 5.9b, 5.9e). This line creates exactly two new vertices, while removing at least one. If the plane touches the polygon the intersection is a point (Figure 5.9c) coinciding with a polygon's vertex or a line coinciding with the polygon's edge (Figure 5.9d). In both cases the polygon is unchanged. Thus clipping a convex polygon with N vertices against a plane results in a convex polygon with at most $N + 1$ vertices. Conversely, clipping a triangle



(a) (b) (c) (d) (e)

Figure 5.8.: Clipping a triangle against bin borders. (a) initial triangle and bin, area outside the bin is gray, inside is white; (b), (c), (d) (e) clipping against individual bin planes, input polygon in thin line, resulting polygon — in bold



(a) (b) (c) (d) (e)

Figure 5.9.: Possible combinations of a triangle and a clipping plane (a) general case, plane passes through two edges of the triangle (b) plane passes through the triangle at a vertex and an edge (c) plane touches the triangle at a vertex (d) plane touches the triangle at an edge (e) plane clips away the whole triangle

with four planes of the bin creates a convex polygon with at most seven vertices.

Occlusion Test When the input triangle is clipped and the bin contains some occluders, occlusion test can be performed. The input triangle is occluded if two conditions hold simultaneously for the same occluder:

- the polygon is completely enclosed in the occluder silhouette
- no part of the polygon is in front of the occluder triangles

In other words, there is a *silhouette test* and a *triangle test* component to the occlusion test. The flowchart of the occlusion test steps is displayed in Figure 5.10.

If the triangle (actually, the clipped triangle, but we will sometimes refer to it as just the triangle for brevity) fails the silhouette test, then some part of the triangle projection lies outside the occluder silhouette, which means the triangle is definitely not occluded. To declare a triangle occluded, we must perform all steps of the test, while, to declare a triangle not occluded by a given occluder, it is sufficient for just one test to fail. Especially when testing against multiple occluders at once (as will become relevant in our GPU

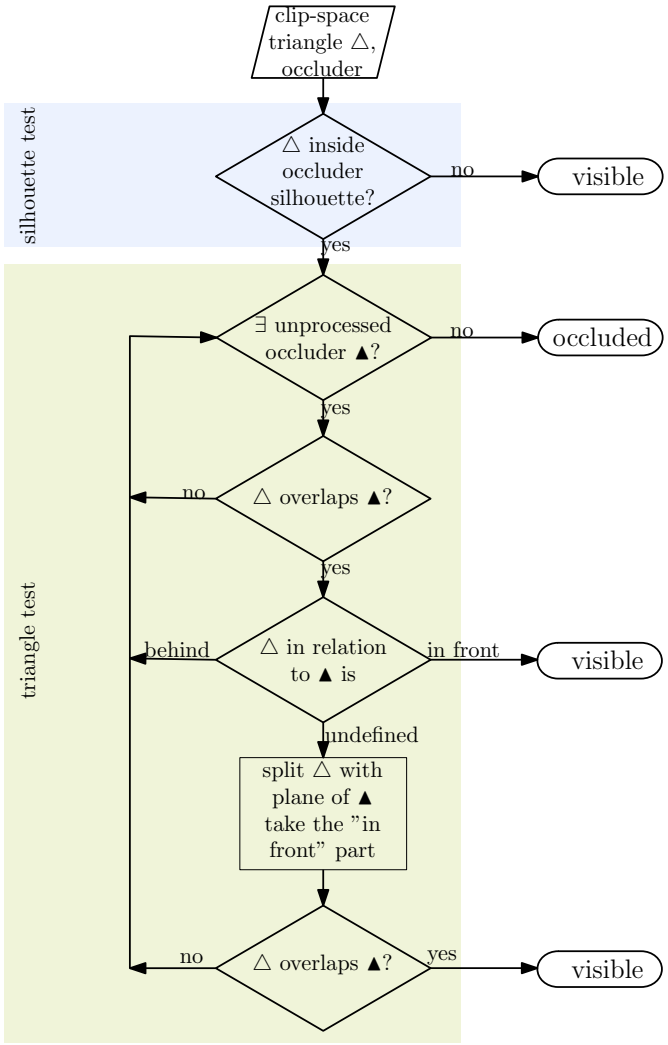
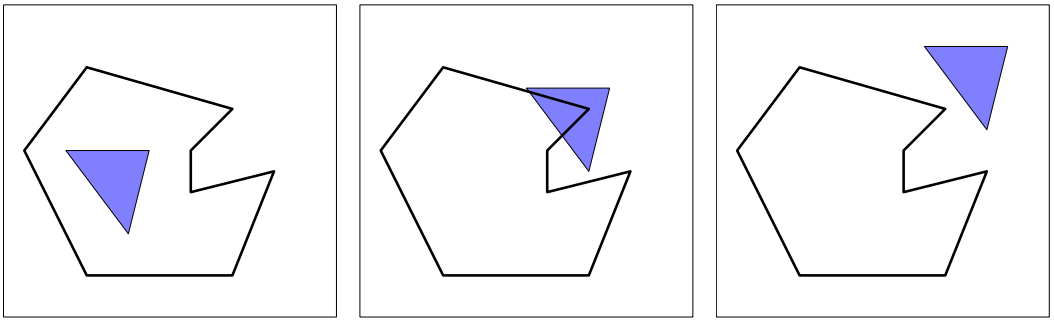


Figure 5.10.: Flowchart of the steps performed in order to determine whether a given occluder occludes a given clipped triangle. White triangle represents the new input triangle, black triangle represents an occluder triangle.



(a) (b) (c)

Figure 5.11.: possible configurations a silhouette and a triangle can be in. The triangle can be entirely contained within the silhouette (a), intersect the silhouette (b), or be entirely outside the silhouette (c).

implementation later on), this strategy of early out allows to discard potential occluders as quickly as possible and save on more expensive triangle tests.

Silhouette Test There are three possible configurations in which the projected triangle and the occluder silhouette can appear: the occludee inside the silhouette, the occludee outside of the silhouette or the occludee and the silhouette overlap. Those three possibilities are illustrated in the Figure 5.11. We need to be able to distinguish between the triangle being inside the silhouette and the other cases. We do not have a straightforward method for that, so we first test for the intersection case (Figure 5.11b), and then, if occludee and silhouette projections don't overlap, we check if occludee projects inside of the silhouette (Figure 5.12).

When the clipped triangle intersects with the silhouette, at least one of the silhouette edges intersects the clipped triangle. We can treat silhouette edges as bidirected edges for this test, since we are interested in them as line segments, objects independent from one another. With some freedom of interpretation we can see bidirected edges as convex objects that consist of 2 half edges (for example, we can think of them as of a triangle with one of its edges converging to zero length). This allows us to check whether a silhouette edge intersects the clipped triangle using the overlap test described in the section 4.4. Simply put, we check all edges of both polygons and if at least one of them happens to be a separator plane, the polygons do not overlap. On the other hand, edge's two half-edges are opposites of each other, and negative halfspace of one is positive halfspace of another. Thus it is sufficient to only test if the triangle is completely in one halfspace of the edge. If any of the edges overlap with the occludee, then silhouette and occludee intersect (Figure 5.11b). If none of the edges overlap, then the occludee is either completely inside (Figure 5.11a) or completely outside of the silhouette (Figure 5.11c). To distinguish between occludee projecting inside and outside silhouette we can use a classic ray casting algorithm: we shoot a ray from inside the occludee and count the

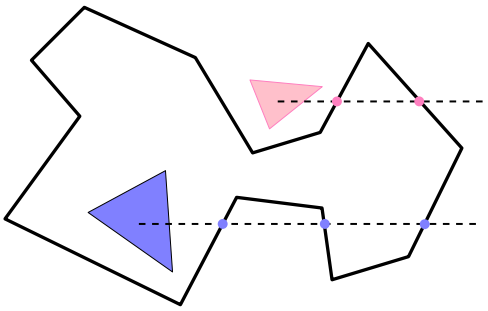


Figure 5.12.: Ray casting algorithm used to determine whether a polygon that does not intersect with the silhouette lies inside or outside the silhouette. A ray is shot from the polygon and the number of intersections with the silhouette is counted. If the intersections count is odd (blue triangle) then the polygon lies inside the silhouette and if even (pink triangle) it must be outside.

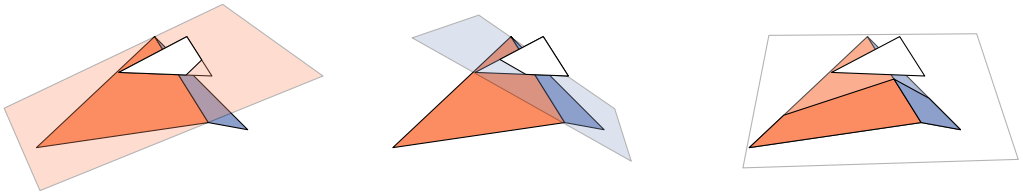
number of intersections with the silhouette edges, as described in Shimrat [Shi62]. This algorithm works in 3D. If the ray intersect the silhouette even number of times, the occludee is inside, and if odd, it is outside of the silhouette (Figure 5.12).

Triangle Test If the input triangle is inside some occluder's silhouette, we need to test that the input triangle is not in front of some occluder triangle. The occludee can only be in front of those triangles which is overlaps in projection. This is similar to the overlap test between the occludee and the silhouette edges (we are trying to see whether two convex shapes will overlap when projected). Except we are now interested in distinguishing between tho polygons not overlapping at all (Figure 5.11c) and all other cases (Figure 5.11a and Figure 5.11b) so the overlap test alone (described in the section 4.4) suffices. This allows us to further save computation with the triangles that are irrelevant for the occludee. For all the occluder triangles that overlap with the occludee we determine *depth relation* to the latter.

Definition 9. *The polygon A is* behind *the polygon B if all points of A projecting within B belong to the negative halfspace of A; and* in front *if they belong to the positive halfspace.*

Definition 10. *If some points of the polygon A projecting within B belong to the positive halfspace of B and some to the negative, the polygons* intersect*.*

It is necessary to note that if the polygon A is in front the polygon B, then the polygon B is behind the polygon A and vice versa. If the polygon A intersects the polygon B, then the polygon B intersects the polygon A and vice versa. Depth relation is clear if



(a) (b) (c)

Figure 5.13.: A rare situation in which depth relation cannot be determined withou splitting one of the triangles. White triangle is clearly in front of both blue and orange triangles, but it intersects the plane of orange (a) and blue (b) triangles. Both triangles intersect the plane of white triangle (c). Depth relation of white triangle cannot be determined.

the whole polygon A lies completely in one halfspace of the polygon B, or if polygon B lies completely in one halfspace of the polygon A. If neither is the case, which is rare but possible (an example of such case can be seen in Figure 5.13), we clip polygon A against the plane of the polygon B and look at the part of A that lies in the positive halfspace of B. If the projection of this clipped part overlaps with the projection of the polygon B, then some points of A that projects within B lies in the positive halfspace of B, and, thus, the polygons A and B either intersect or A is in front of B. If they do not overlap, then no point of A that belong to the positive halfspace of the polygon B projects inside the polygon B. Then we can conclude that the polygon A lies behind the polygon B. Finally, if an occluder is found, which satisfies all the tests — the occludee lies completely withing its silhouette and is not in front any of its triangles — the occludee can be pronounced occluded.

Adding visible triangle to occluders If the clipped triangle is indeed visible, the original triangle can potentially be added to the existing occluders. Whether to actually add it or not is a question of its own, and can be decided based, for example, on the occluder's size, or position, or available space, or any other criteria. Experimenting with different criteria is a task for the future research. Presently we add visible triangles as occluders as long as there is space in the bin.

A visible triangle can create a new occluder, be added to an existing occluder or connect two or more existing occluders. Larger occluders are better than small disconnected occluders, thus we grow existing occluders whenever possible. To test if a triangle can be added to one of the existing occluders we compare this triangle's edges to the occluder edges. It is only necessary to test the silhouette edges, since all other triangle's edges are already shared withing the occluder. A triangle can share any number of edges with one or multiple occluders. If no shared edges are found, a new occluder is initialized with the triangle and all its edges as silhouette edges (Figure 5.14b or Figure 5.14c). If one edge is shared, the triangle is simply added to that one occluder (Figure 5.14d). To do so, we remove the shared edge from the occluder's set of silhouette edges, add to them

two not shared edges of the triangle and add the triangle itself to the set of occluder triangles. If the triangle shares multiple edges with the same occluder, we do all the same but we remove all the shared edges from the set of silhouette edges (Figure 5.14e or Figure 5.14f). If the triangle shares edges with different occluders (Figure 5.14g or Figure 5.14h), we remove all shared edges from the corresponding occluder's silhouette edges, add the triangle to one of the occluder triangle sets and merge the occluders. In order to merge occluders we select an occluder with the most triangles, reassign all other occluder's edges and triangles to belong to the selected occluder. Details about efficient implementation of merging can be found in the section 6.3.2.

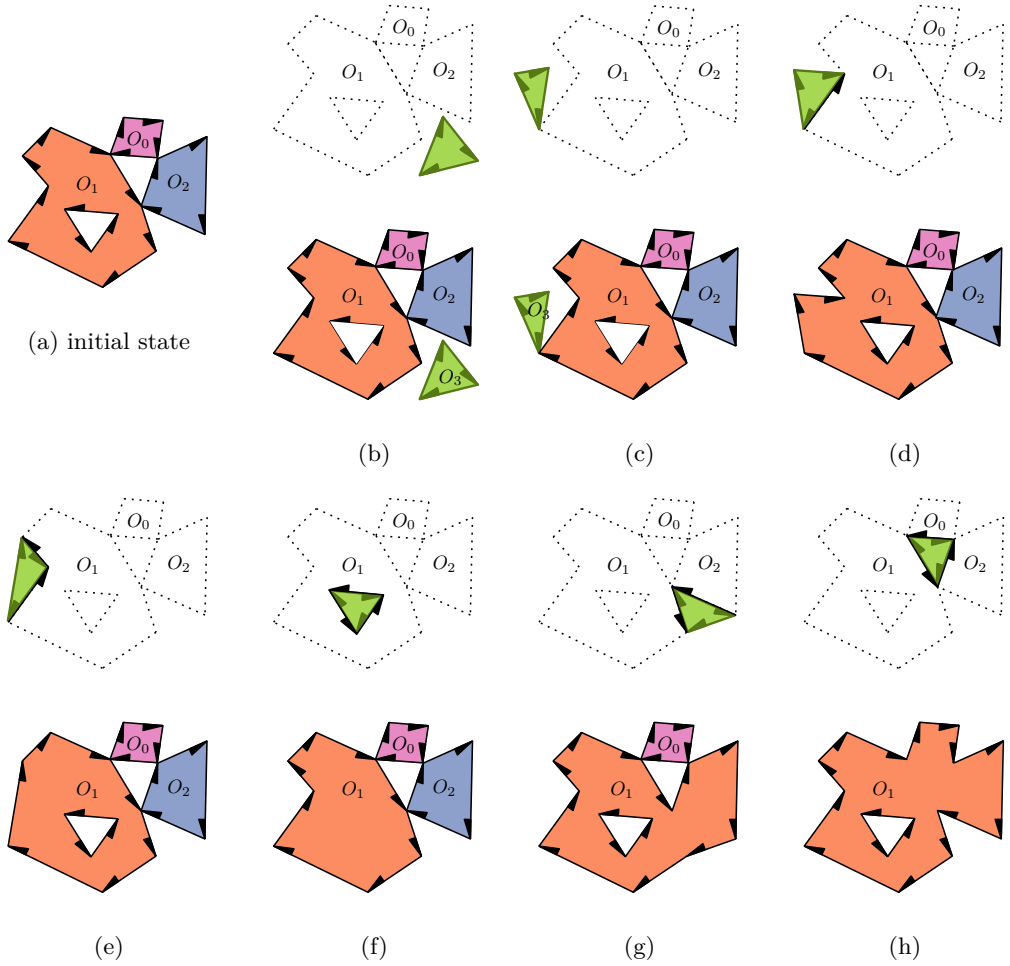


Figure 5.14.: The different possible configurations when adding a new occluder triangle (green) and the resulting changes to the occluder state. Occluders are defined by their occluder polygon and its directed silhouette edges. (a) Initial state with three existing occluders. (b, c) A triangle that does not share any point or only shares a single vertex with an existing occluder becomes the first triangle of a new occluder. (d, e, f) If a triangle shares one or more edges with an occluder, it is added to that occluder. (g, h) If a triangle shares an edge with more than one occluder it forms a connection between these occluders. In this case, all connected occluders are merged into a single occluder that also includes the newly added triangle.

6. Implementation

As previously discussed, our visibility pipeline consists of two stages: a geometry stage and a visibility stage. To allow the visibility computation to be parallelized, the screen is subdivided into rectangular bins. The geometry stage is responsible for projecting input scene triangles onto the screen and sorting them into bins. The visibility stage resolves visibility for the triangles in each bin.

In our testbed, visibility computation and display of the image is split in a manner similar to SAS. Each frame, we first run our pipeline to compute the ids of visible triangles and then use OpenGL [Khr17] to display the resulting scene. Our GPU implementation is based on NVIDIA CUDA C++ [NVI21]. We take advantage of the CUB library [NVI20] of parallel programming primitives to provide us with efficient implementations of algorithms such as block-level reductions or radix sort.

6.1. Megakernel

The visibility pipeline we seek to implement is similar in structure to the rendering pipeline presented in cuRE [Ken+18] except that, instead of a rasterization stage, we run an analytic visibility stage. Similarly to cuRE, we use a persistent Megakernel [Ste+14] to achieve a bounded-memory streaming implementation with both stages of the pipeline running concurrently to allow for dynamic load balancing.

Our Megakernel is one kernel function that contains both the geometry and visibility stages as well as scheduling logic. We launch as many blocks as possible to fill the GPU, but not overfill it. Thus, all blocks are running concurrently and we can control how different stages are scheduled inside each block. The number of blocks to launch is determined based on GPU-specific parameters such as the number of available multiprocessors as well as the desired number of threads per block and a given target occupancy of how many blocks should fit on each multiprocessor. Possible launch configurations are limited by resource requirements such as the number of registers or amount of shared memory required per block. We combine C++ templates with runtime compilation to automatically generate instances of this Megakernel that are specialized for the exact parameters of the GPU we are running on as well as certain parameters of the algorithm such as the dimensions of the grid of bins to use.

Each Megakernel block dynamically decides which stage to execute when based on the scheduling logic. If a block decides to run the geometry stage, it will fetch a batch of input scene triangles, project them, and sort the projected triangles into bins. If a block decides to tun the visibility stage, it will attempt to find a bin that has accumulated a number of projected triangles and then resolve visibility for the triangles currently in that bin.

Bin Queues The key component to enable concurrent execution of both pipeline stages is a set of bin queues that buffers the projected triangles for each bin between geometry and visibility stage. The queue implementation we use is based on the broker work distribution queue by Kerbl et al. [Ker+18]. Since the same triangle will likely fall into more than one bin, we use a separate *triangle buffer* to store per-triangle data and enqueue only indices pointing into this triangle buffer into the bin queues to reduce overall memory consumption and, most importantly, memory bandwidth. The triangle buffer is a global ring buffer which stores the complete triangle information, such as vertex positions, normals, or texture coordinates as well as a reference counter for each triangle. The reference counter is used to make sure that space in the triangle buffer is only reused once a triangle has cleared all bin queues into which is was enqueued. Figure 6.1 shows how the triangles are binned, stored into the triangle buffer, and only triangle ids are enqueued into to the visibility stage queues.

Shared Memory Many steps of our algorithm require large shared memory buffers. However, most of these buffers are only needed for a short and limited duration, for example, as temporary storage during a sorting operation. Since shared memory requirements directly influence the number of blocks that can fit on each multiprocessor, it is important to keep shared memory usage at a minimum. Thus, we make extensive use of unions to overlay any shared memory buffers that are never simultaneously in use.

6.1.1. Scheduling Logic

Multiple blocks cannot execute the visibility stage for the same bin simultaneously for reasons of speed and accuracy: firstly, it would introduce communication overhead between the blocks because it would mean that a bin's occluders would have to be modified in parallel and, secondly, it would increase the number of false positives since some occluder triangles would be processed in parallel with their occludees and, thus, not taken into account.

The geometry stage generates data that the visibility stage consumes, but the memory available to store this intermediate data is limited. Prioritizing geometry stage execution would lead to bin queues overflowing. Prioritizing the visibility stage would lead to

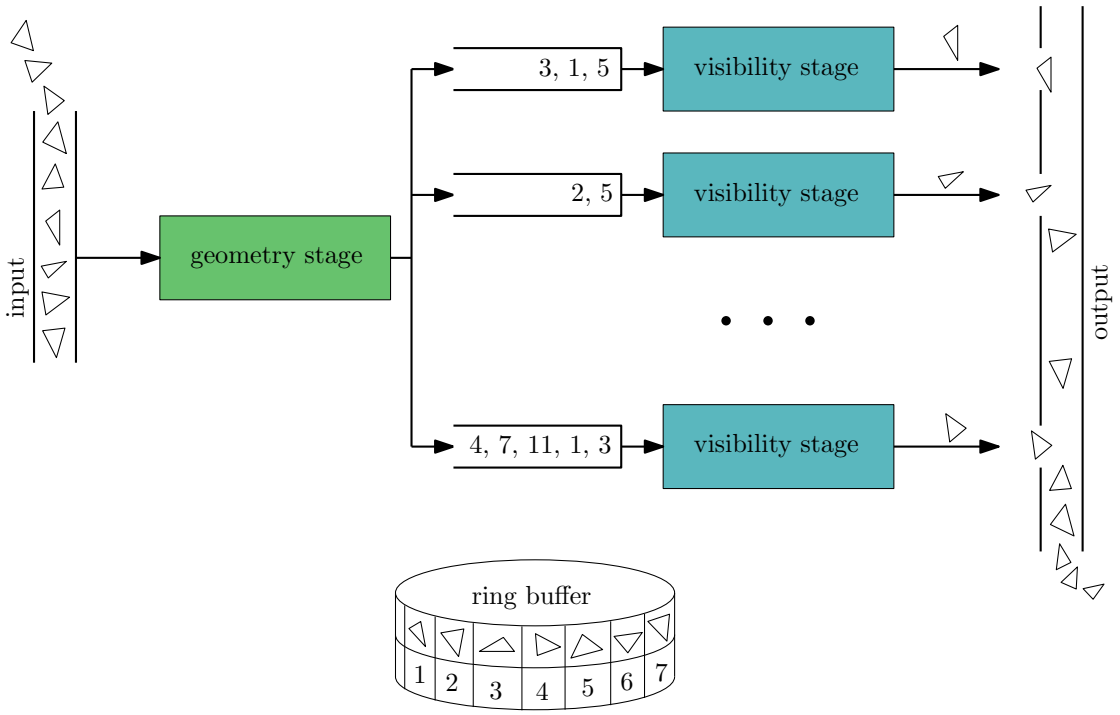


Figure 6.1.: General pipeline and flow of triangles through it. The input stream of scene triangles is processed by the geometry stage, which distributes projected triangles to the according bin queues. Some triangles fall into multiple bins. To avoid duplicating triangle data, the triangles are stored in a ring buffer and bin queues only store the ids of the triangles in this ring buffer. The output stream consists of triangles that were detected as visible.

inefficiencies since the bin queues will stay close to empty, causing blocks to switch between stages unnecessarily often. We need to find a balance between executing visibility and geometry stage in a way that keeps bin queues filled but not overflowing. Thus, if there is sufficient input for the visibility stage (number of triangles in the queue above a threshold set by the user), we choose to execute visibility. Otherwise, if possible, we run the geometry stage. If there is no more input for the geometry stage, we enter a "drain" mode in which some blocks might still be running the geometry stage but blocks only start new work running the visibility stage regardless of available input in order to finish the process. The megakernel finishes when no block is running the geometry stage anymore and all bin queues are empty.

When choosing which bin to execute the visibility stage on, we try to acquire the fullest available bin. The block fetches the current size of each bin queue and sorts the bin indices according to their fill level. This can be done in parallel. Each thread reads the fill levels of one or more bin queues and stores them along with their respective bin indices into local arrays. These local arrays are then sorted using a block-level parallel radix sort. As a result, thread 0 will have the id of the fullest bin, thread 1 the next fullest bin, and so on. There can be multiple blocks looking for work at the same time. To make sure that no more than one block works on the same bin concurrently, we keep a lock for each bin. Based on the sorted bin queue sizes, we try to acquire the lock of the next fullest bin in descending order using an atomic compare-and-swap operation. If we find that a bin was already acquired by another block, we keep trying the next bin. If we reach queues with a fill level below the threshold, we stop this process and try to run the geometry stage instead unless we are in "drain" mode in which case the threshold is ignored.

Running the geometry stage in too many blocks or running it for too long could result in bin queues overflowing. To avoid this problem, we keep a counter to keep track of the number of triangles currently *in flight*, i.e., being processed by the geometry stage and about to potentially be written into the queues. Before executing the visibility stage, we atomically increment this counter by the number of input triangles, which represents a conservative upper bound for the number of elements that may end up needing to be put into queues. After finishing the stage, the counter is decremented again. Based on the value of this counter, we check whether every bin has enough space to handle the worst-case scenario of receiving all the currently in-flight triangles.This check can be parallelized by using one thread per bin and using a synchronization barrier reduction to aggregate the resulting decision. If at leas one bin would not be able to fit all in-flight elements, the geometry stage cannot be executed.

6.2. Geometry Stage

The first stage of our pipeline is responsible for projecting input scene geometry in the form of triangles onto the image plane. The camera view projection matrix is supplied to the Megakernel as a constant. To represent the input stream, we upload geometry information such as vertex and index data into device memory and use a global atomic counter to track the part of the input that was already "streamed" into the pipeline. Before starting the processing, the visibility stage increases this counter by the number of threads in the block. The result of this operation is the index of the first unprocessed element which is communicated to all other threads in the block via shared memory. Each thread computes the index of the triangle it should process by adding its own thread index. This ensures a coherent memory access pattern. Each thread projects its triangle using the camera view projection matrix. If backface culling is turned on, backfacing triangles are simply discarded. Otherwise, the order of the triangle's vertices is reversed to ensure that all visible geometry follows a consistent ordering from here on.

Each thread enqueues its projected triangle into all bins it overlaps with. We first compute the bounding rectangle of the clipped triangle and iterate over all bins that bounding rectangle overlaps with. For each bin within the bounding rectangle, we additionally perform an overlap test between the bin and the triangle. Since both bin and triangle are convex shapes, we can use the overlap test described in the Geometry section 4.

6.3. Visibility Stage

For each bin, clip space triangles that overlap that bin are collected in bin queues. Visibility is then resolved per bin in the visibility stage. As described in section 5, our visibility algorithm requires that we keep a set of active occluders for each bin. Unlike the geometry stage, in which the output depends only on the input and, thus, each triangle can simply be processed by one thread, visibility stage output depends on the relation of each input triangle to the bin's active occluders. Each incoming triangle potentially modifies this set of occluders. Updating occluders concurrently while also testing new triangles against them would introduce significant synchronization overhead. Additionally, each input triangle could participate in occluding other input triangles. Thus, by processing input triangles in parallel we would risk missing occluded triangles. Therefore, instead of parallelizing the processing of input triangles, we have decided to process input triangles one by one and parallelize the check against occluder primitives instead. The way we store our set of active occluders is designed to allow for efficient memory access patterns under this approach.

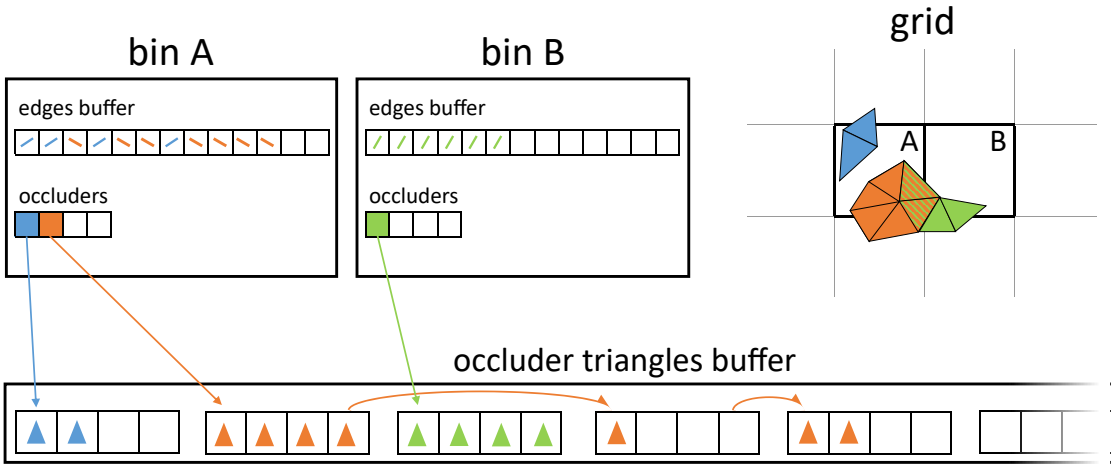
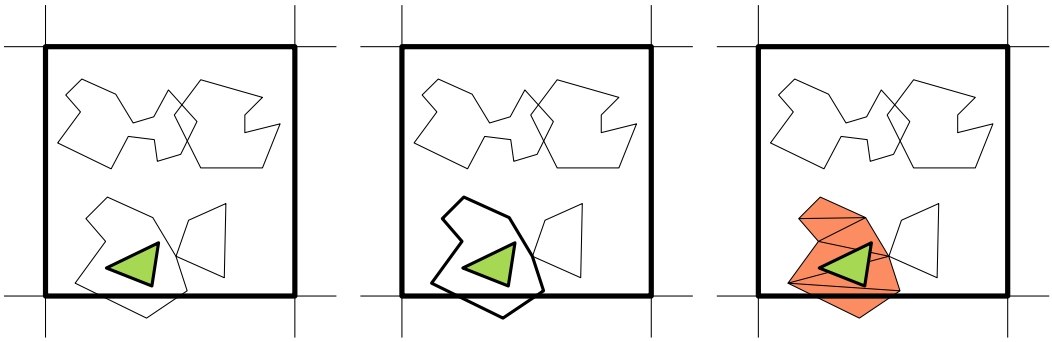


Figure 6.2.: Organization of visibility stage state storage. Each occluder is rendered in a separate color. Triangles are rendered in the color of the occluder they belong to. All visibility stage data is located in global memory. Each bin is associated with an array of occluder silhouette edges as well as an array of per-occluder information such as the size of the occluder or a pointer to the beginning of a list of triangle batches. Edges are not stored in any particular order, but are marked with occluder ids. Occluder triangles are stored in batches in a global batch buffer. One batch only contains triangles of the same occluder. Multiple batches can be linked together to store a longer list of triangles or as a result of two occluders merging into one.

6.3.1. Occluder Storage

Figure 6.2 gives an overview of how we store per-bin occluder state. Occluders are highly variable: their configuration, shape and number of elements varies during processing of a single frame. As occluders grow and merge together, edges are added and removed, their number and size changes unpredictably. Consisting of many elements, they are generally too large to be stored anywhere but in global memory. With every input triangle being tested against all existing occluders, fast read access is of high priority. Adding new triangles happens relatively rarely as we expect most triangles to be discarded as occluded rather than be visible and added to the occluders, which makes speed of adding a triangle to an occluder less of a concern.

We recall that occluder triangles and occluder edges are used in separate phases of the visibility test: First, the occlusion test needs to traverse all silhouette edges to determine which occluders the triangle overlaps with. Second, some occluders' triangles, if any at all, are needed to check whether the triangle is occluded by any overlapping occluders (see Figure 6.3). Thus, we store occluder silhouette edges and triangles separately and in different memory layouts, each geared towards the access pattern encountered in the respective phase of the algorithm.



(a) (b) (c)

Figure 6.3.: Occluder silhouette edges and occluder triangles are only needed in separate phases of the visibility algorithm: a) the input triangle (green) is tested against all silhouette edges b) the input triangle falls within the silhouette of one occluder c) only one occluder's triangles are required to determine the visibility of the input triangle.

Silhouette Edges Because the silhouette test requires the silhouettes of all occluders to be checked against the input triangle, we can simply test all occluder silhouette edges for overlap with the input triangle in any order and then combine the results on per-occluder basis. Thus, the first step of the silhouette test is free of thread divergence and the memory access pattern can be perfectly coalesced. We store all silhouette edges in one dedicated buffer per bin. Their order is simply the order in which they happened to be added, irrespective of which occluder they belong to. To allow for a perfectly coalesced memory access pattern, the silhouette edges buffer is organized into separate arrays for each component: six float arrays for the x, y, and w coordinates of the start and end points of the edge, and one integer array to store the id of the occluder the edge belongs to.

Occluder Triangles During the triangle test we need to check our input triangle against the triangles of those occluders it overlaps with — and only the triangles of those occluders. We initially attempted to store occluder triangles in a fix-sized array in each bin. However, this approach has proven to be too rigid: while some bins were empty, others had more occluders than would fit into the storage allocated per bin. To alleviate this issue, we store triangles in a linked list of batches instead. This approach allows us to keep a good memory access pattern while still allowing for the necessary flexibility to adapt to the size of each occluder. A batch of triangles has a fixed capacity. All batches are stored in a batch pool which also has a fixed size. Batch capacity as well as the size of the batch pool are compile-time constant parameters of our implementation which can be specified via a configuration file. Triangle batches are allocated from the global triangle batch pool only when necessary: when a new occluder is created or an old occluder has outgrown its capacity. As a result, empty bins do not occupy valuable memory and bins with a lot of geometry can allocate more space to store more occluders and, thus, perform visibility

tests more accurately. While the triangle data for occluders from different bins ends up interleaved in global memory, storing this data in batches still ensures a minimum degree of coherence as blocks work on occluder triangles in parallel. There is one disadvantage to this approach: slower allocation of new batches as a result of global memory management. Between the much more effective memory usage and some added allocation overhead, we have decided in favor of more effective memory usage. To allow for maximally-coalesced memory access, we store all components like the x, y, and w coordinates of each triangle vertex in separate arrays like we did with the silhouette edge data.

Visibility Buffer While our algorithm is designed for streaming output of visiblity data, current hardware and graphics APIs do not offer a way for us to stream output as the kernel is running. Thus, we simply write the output into a visibility buffer, a write-only bit array the size of the input. Each bit corresponds to a triangle; bit i represents the visibility of the i-th input triangle. A bit of 1 means the corresponding triangle is visible while 0 means it is not. At the start, this buffer is initialized to all zero. We initialize all triangles to "invisible" and mark the visible ones rather than the other way around because, to be invisible, a triangle must be occluded in all the bins it falls into. Thus, a definite decision about whether a triangle is invisible could only be made once all triangles in all bins have been processed. On the other hand, as soon as a triangle is determined to be visible in one bin, it is visible in general. Since individual bits are not addressable, we use an unsigned integer buffer as a compact way of storing the visibility bits of all triangles. Atomic bitwise logical OR operations can then be used to set individual bits.

6.3.2. Occlusion Test

To load an input triangle, the first thread of the block takes the index of the next triangle out of the queue and loads the triangle data from the triangle buffer into the block's shared memory. Then all threads load that triangle into their local memory and the block collectively performs the occlusion test. The first step in the occlusion test is the silhouette test. As described in chapter 5, the occlusion test consists of two steps: first, we run a silhouette test to determine which occluders we potentially have to consider and, second, we run a triangle test against each potential occluder.

Silhouette Test Since we store all occluder edges in one consecutive array, each thread in the block tests one edge at a time, independently of which occluder it belongs to. Each thread reads and processes one edge in the edge buffer according to its thread index (thread 0 reads edge 0, thread 1 reads edge 1 and so on). This ensures a coalesced memory access pattern. To keep track of the results, we keep a boolean for each occluder in shared memory, which is initialized to false (no overlap with any edges). In the next step — testing whether the triangle is inside or outside an occluder silhouette — the

silhouette edges would be needed again. Instead of performing that test separately, which would require looping over all occluder edges a second time, we fuse this step into the overlap test. Another shared memory array is used to keep count of the number of ray intersections with each occluder's silhouette. After initializing this array to zero, atomic add operations can be used to update the counters from each thread that detects an intersection. Once those two operations are performed on all edges, we can conclude which occluders' silhouettes contain the clipped triangle's silhouette. This step can be performed in parallel as well with each thread checking the overlap flag and ray intersection count of another occluder.

Triangle Test Potential occluders found as a result of the silhouette test must now be checked for their depth relation with the clipped triangle. Each triangle of the occluder needs to be tested, but the tests are, again, independent from one another. We process occluders one by one, each thread, as always, takes one triangle from the buffer according to its thread id.

Only those triangles the projections of which overlap with the clipped triangle need to be considered for the depth test. Thus, first we perform an overlap test on two convex shapes: the clipped input triangle and the occluder triangle. For triangles which overlap in their projections, a quick depth test is performed first: if polygon A is in the positive halfspace of a polygon B, polygon A is in front of polygon B and vice versa. If the depth relation is not clear from the quick test, the input triangle is split at the occluder triangle's plane and we test the part that is in front of the plane for overlapping silhouette with the occluder triangle. If the input triangle is not found poking in front of any occluder triangle, the input triangle can be determined occluded and does not have to be considered further. Otherwise, the input triangle is visible, we mark it as such in the visibility buffer and try to add this triangle to the bin's active occluders.

6.3.3. Occluder State Update

A visible triangle can either be added to an existing occluder, merge multiple occluders, be added as a new occluder of its own, or not be added to the set of occluders at all in case the memory allocated for storing occluder state is exhausted. Large and hole-less occluders are the most efficient, so it is preferable to merge occluders or at least to add triangles to existing occluders. A triangle is connected to an occluder if they share an edge. Thus, it is once again necessary to compare the input triangle to all silhouette edges in the bin. Note that occluders store the original, unclipped triangles, so we use the the original input triangle in this step. There are three possible results: The triangle may share no edges with any occluder, it may share one or more edges with the same occluder, or it may share edges with multiple occluders. In the first case, if there is a free slot for an occluder, enough space for three new edges in the edge buffer, and an

empty occluder triangle batch, a new occluder is initialized with this triangle and all its edges. In the second case, the triangle is added to the occluder it shares edges with: shared edges are removed from the edge buffer, edges of the input triangle that are not shared are added as new silhouette edges, and the triangle itself is added to the occluder's triangle batches. In the third case, we need to merge multiple occluders.

Merging of Occluders First of all, we compare ids of occluders we want to merge and pick the smallest one, which will become the id of the resulting occluder. The new triangle is simply added to this occluder. Then we repeat the following steps for every old occluder we need to merge into our new occluder: To merge occluder triangles, we simply make the last triangle batch of the new occluder point to the first batch of the old occluder and update occluder information such as fill level. This is done by one thread. To merge occluder edges, each thread takes an edge from the edge buffer, checks its occluder id and if it is the old occluder's id, it is replaced with the new occluder's id. After merging all occluders into the target, we remove the shared edges from the edge buffer based on the old occluder's ids. To fill in the holes they leave, we move edges from the end of the edge buffer into the vacant spots. Lastly, we add any remaining edges of the new triangle — if any — to the edge buffer.

6.3.4. Initialization

Per-bin occluder state is initially empty. The first triangles put into a bin will almost certainly end up simply moving through all visibility tests to be formed into occluders. To speed up this process, rather than building up occluder state triangle-by-triangle over many iterations of visibility processing, we start off the visibility stage for each bin in an initialization phase where we take an entire first batch of input triangles from the bin and form them into occluders right away.

Due to the way scene geometry is commonly constructed, it is likely for triangles that share locality in the input stream to be part of the same surface. Thus, it is likely that triangles within the first batch of input triangles form larger patches of occluders. Therefore, to form our initial set of occluders, we search for such connected patches in the first batch of input triangles.

Figure 6.4 illustrates this process. We begin by loading a batch of input triangles into shared memory, each thread of the block is responsible for loading one triangle. Each thread then compares the edges of its triangle to those of all other triangles to find shared edges and, thus, establish connectivity information. For each edge, the index of the thread responsible for the triangle the edge is shared with is kept in a local array. Additionally, we also store a bitmask to remember which edges are shared.

Once each thread knows which edges of its triangle are shared and who they are shared with, we can group connected triangles into occluders. First, we assign a unique id to each group by searching for the smallest thread id within each connected set of triangles. Instead of each thread traversing the entire group to independently find the smallest id, we make thread ids bubble up in parallel. In each step, each thread publishes in shared memory the smallest thread id it has found so far. Initially, each thread publishes the smallest neighbor id of its triangle or its own id depending on which is smaller. Each thread then looks at the values published by its neighbor triangle threads to determine a new minimum. The new minimum is published again, and the process repeats until no thread can find a minimum smaller than its current and, thus, the smallest thread id within each group of connected triangles has been propagated to all of the group's threads.

With all connected groups having been assigned a unique id, we can now simply sort all triangles and their edge masks according to their group id. After this sorting step, triangles that belong to the same occluder are located at consecutive threads. We then assign consecutive occluder ids (the group ids used in the previous sorting step were not necessarily consecutive) to each group of triangles. To do so, each thread checks whether it is the first or last thread in a group by comparing its group id to the group id of the previous and next triangle. We then have the last thread in each group publish a 1, all other threads a 0, and run an exclusive parallel prefix sum over this data to assign consecutive occluder ids to all groups and, at the same time, distribute the occluder id to each triangle. Note that the threads in the last group know the last occluder id and, thus, also the total number of occluders that needs to be created. We use the last thread in the last group to initialize the number of occluders for the bin.

With triangles grouped into occluders based on their connectivity and occluder ids assigned, we now have all the information needed to allocate and populate occluder state. The first thread of each triangle group publishes its thread id into a shared memory array. All other threads can then compute their intra-group offset by subtracting their thread id from the id of the first thread, thus, assigning a consecutive intra-occluder triangle index. Knowing the total number of triangles in its group, the last thread of each group can then proceed to allocate the necessary number of triangle batches and publish the id of the first triangle batch. Finally, each thread can write its triangle to the corresponding offset into the corresponding triangle batch allocated for its occluder.

To complete occluder state initialization, we still need to write out occluder silhouette edges. The edge mask of each triangle contains information about which edges are shared with another triangle. Edges not shared with any triangle are silhouette edges. To find the offset at which each thread should start writing the silhouette edges of its triangle, an exclusive parallel prefix sum over the number of silhouette edges for each triangle can be used. The thread responsible for the last triangle knows the total number of silhouette edges and, thus, is used to allocate the necessary memory in the silhouette edges buffer. Silhouette edges are then written to the edge buffer accordingly.

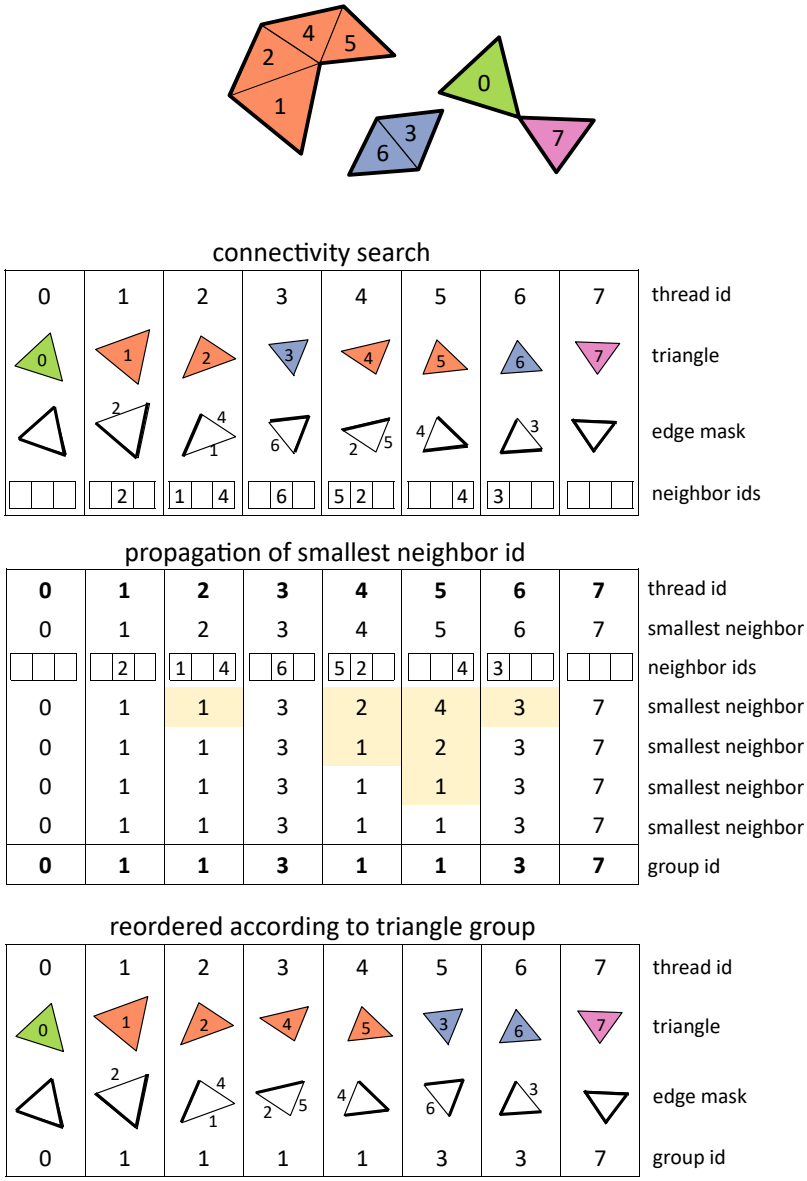


Figure 6.4.: Visibility stage initialization phase for a given example set of input triangles (heavier lines indicate silhouette edges). Connectivity search: after loading an input triangle into shared memory, each thread compares the edges of its triangle to those of all other triangles to produce an edge mask that keeps track of which edges are shared with another triangle as well as an array of neighbor ids that stores the ids of the threads corresponding to those triangles. Propagation of smallest neighbor id: to form triangles into occluders, we first assign a unique group id to each set of connected triangles. To do so, we perform a parallel search for the smallest thread id within each connected set of triangles. Each thread publishes the smallest id it has found within its connected set of triangles so far. In each step, each thread compares its current smallest id to the smallest ids found by any of its neighbors and updates its own value accordingly. This process is repeated until no thread finds a smaller value anymore (the highlighted cells indicate that a value was updated). Reordering according to triangle group id: by sorting using the group id assigned in the previous step as key, we reorder triangles such that there is a range of consecutive threads for each occluder that is to be created.

7. Evaluation

To evaluate our approach, we first investigate the overall behavior of the algorithm, in particular, the effect of its various parameters on performance. Once we have established which configurations present good sets of tradeoffs, we compare the performance of our method to that of previous work. At the time of writing, SVGPU [EHH16] is, to the best of our knowledge, the only recent analytic visibility algorithm designed for the GPU. Thus, we compare our approach — which we will refer to as RV for the remainder of this chapter — to SVGPU in terms of speed as well as quality of the resulting visibility set. As a baseline reference, we use OpenGL to render the scene at much higher (8×) resolution and take advantage of the hardware depth test to count which triangles end up being visible in the resulting image. While, in general, such a sampling-based approach can only produce an approximation of the exact visibility set for a given viewpoint, we expect the result to be reasonably accurate as long as resolution is high enough.

All our measurements were collected on an NVIDIA GeForce RTX 2060 GPU with 6 GiB of video memory on Windows 10 running on an Intel Core i7-8700K CPU at 3.7 GHz with 16 GiB of system memory. To measure pipeline runtime, we use CUDA events and OpenGL `TIME_ELAPSED queries`. To get a reasonably accurate estimate of overall memory usage, we record the amount of free video memory before algorithm initialization, after processing every frame, and then take the difference between the minimum and maximum memory usage observed.

To run our tests, we rely on a set of various test scenes designed to exhibit various aspects relevant to real-world applications. Figure 7.1 and Figure 7.2 give an overview of the different test scenes used. These test scenes include individual objects of various levels and kinds of complexity as well as larger scenes that contain multiple objects to generate inter-object occlusions. For each scene, there is an unsorted version in which the order of input triangles has been randomized as well as a sorted version in which the input triangles have been sorted according to their minimum clip-space z coordinate such that triangles closer to the camera, i.e., potential occluders, are ordered before triangles further away from the camera, i.e., potential occludees. Unless otherwise noted, the unsorted version of a given scene has been used.

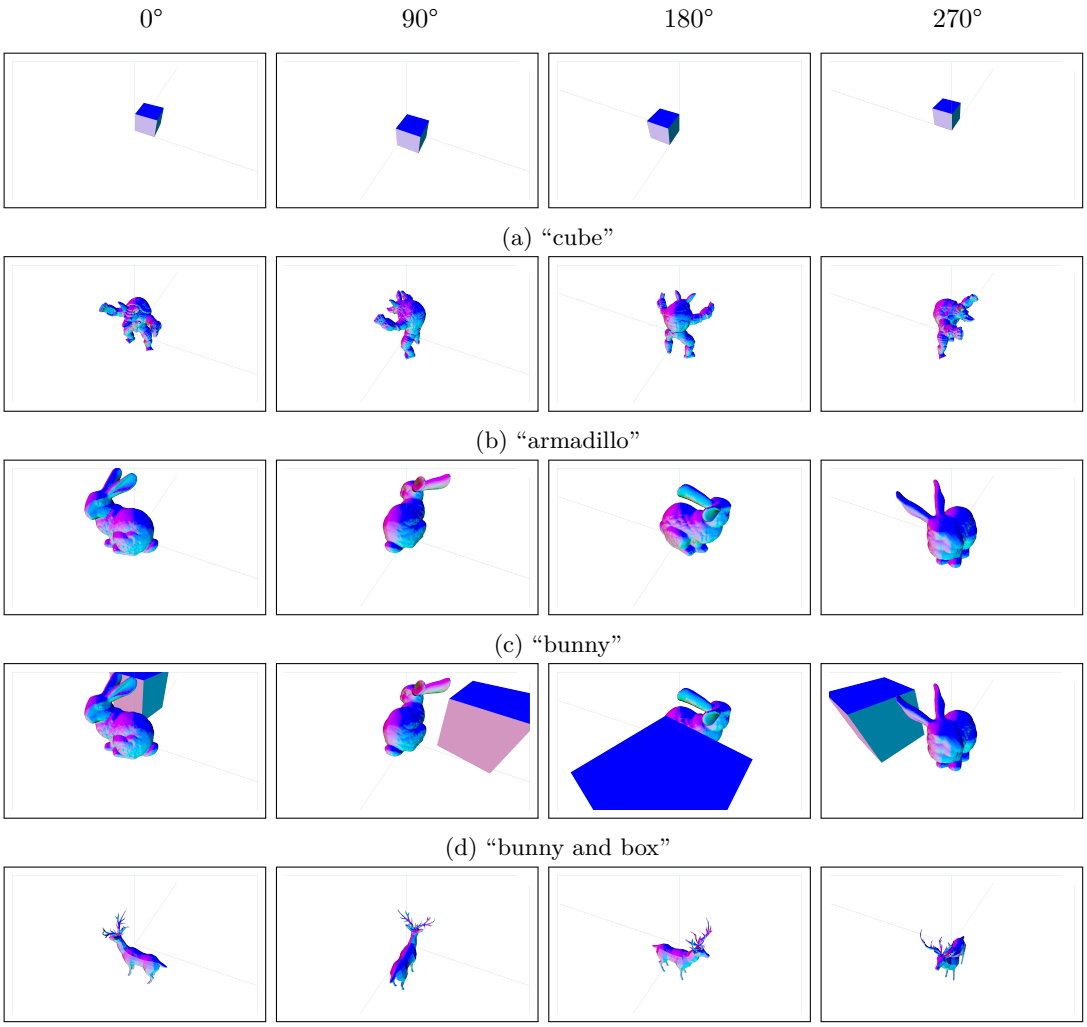
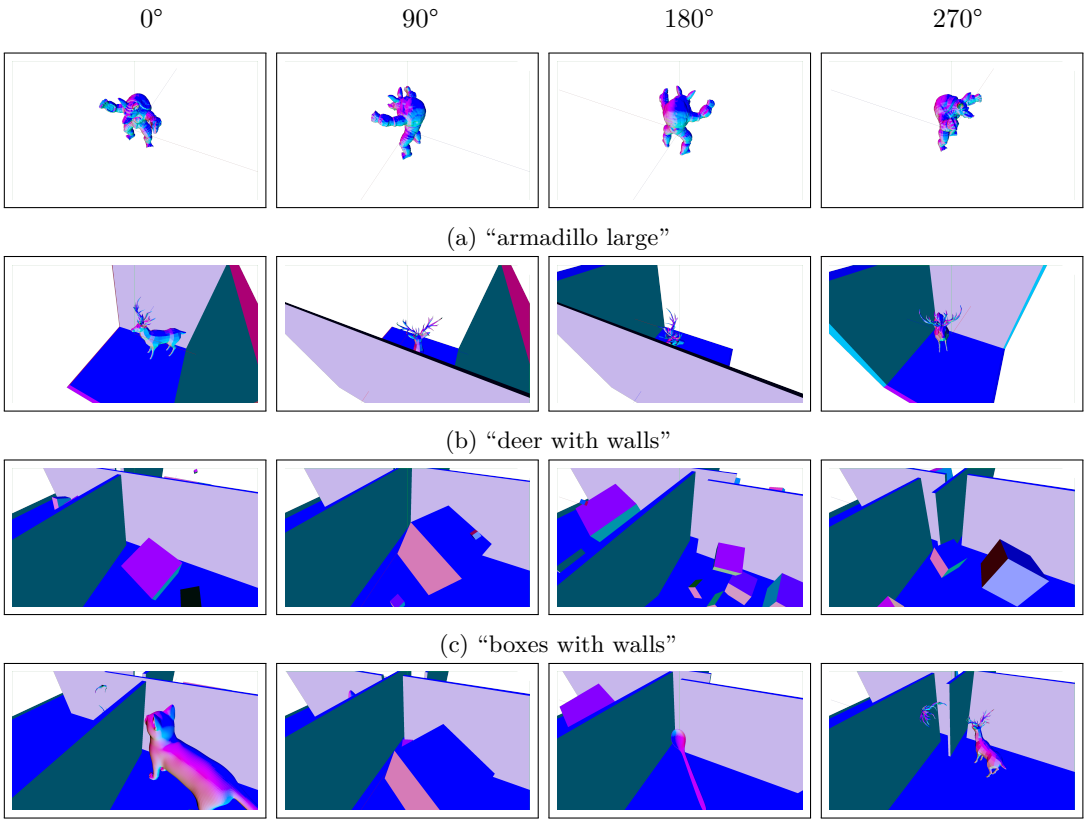


Figure 7.1.: Overview of the test scenes used in our evaluation from various camera angles. (a) A simple cube consisting of 12 triangles. (b) The Stanford armadillo: a single closed object consisting of a large number of mostly similarly-sized, small triangles. (c) The classic Stanford bunny scene; another closed object but with fewer triangles. (d) The Stanford bunny next to a large cube. The cube causes substantial occlusion and silhouette intersections while barely affecting the overall triangle count. (e) A low-polygonal deer model with detailed antlers that cause a lot of complex self-occlusion.



(a) “armadillo large”

(b) “deer with walls”

(c) “boxes with walls”

(d) “flat”

Figure 7.2.: Overview of the test scenes used in our evaluation from various camera angles. (a) A retopologized and subdivided version of the Stanford armadillo that consists of a single manifold surface with 64% more triangles than the original model. (b) Deer in a corner. From some angles the deer is occluded by walls while, from other angles, its antlers overlap with the walls. We expect this scene to be problematic for SVGPU due to high silhouette complexity. (c) A simplified model of a real world scene with many objects of different scale scattered around, including large and flat walls to occlude large portions of geometry. (d) A slightly more complex version of the “boxes with walls” scene with objects of varying complexity and triangle count located in four rooms, including a room with a deer and deer antlers on the wall.

7.1. Effect of Algorithm Parameters and Various Optimizations

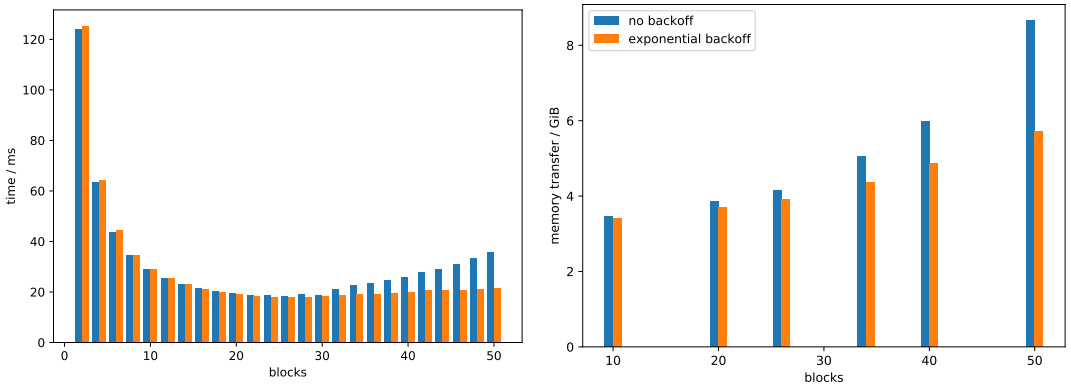
Performance of RV depends on the choice of certain parameters. Furthermore, there are a number of potential optimizations that can improve the performance of our algorithm over a basic implementation. To develop a better understanding of the influence these parameters and optimizations have on the performance characteristics of our method, we measure their effects by independently varying the value of each parameter and comparing performance with and without a certain optimization enabled. The "armadillo" scene was used in all of the following tests. To contrast the effect a certain configuration of the algorithm may have in one scene with the effect it has in another, we will occasionally also show results from the "flat" scene or an "armadillo" scene viewed from a different camera angle. Except where noted otherwise, we use a grid of 50×50 bins and 50 blocks. To take into account the effect of different viewpoints, the camera is rotated around the scene along the z axis by 3° each frame. Measurements are collected for 10 full 360° rotations during each test run. The first 20 frames are discarded to allow the system to warm up. In sorted scenes scene geometry is sorted from closest to the camera to the furthest before the test.

7.1.1. Number of Thread Blocks

Traditionally, a Megakernel occupies the entire GPU. However, during our experiments, we noticed larger runtimes and higher global memory loads than expected for some configurations of this kind. Upon further investigation, we observed the runtime and memory load decreasing when running the Megakernel at lower occupancy. As can be seen in Figure 7.3a, between 2 and 30 blocks, the more blocks are used, the faster the computation runs, just as one would expect. However, around 26 blocks, the runtime starts to increase despite more workers being available for performing the same amount of work.

Looking at the global memory transfer reveals the cause of this behavior. In Figure 7.3b, we can clearly see that global memory transfer is rising steadily as the number of blocks increases. Since the algorithm output is exactly the same in each case while runtime is increasing, this additional memory transfer cannot be as a result of performing any productive work. One potential cause could be not having enough work to keep all the blocks busy. In such a case, we would expect some blocks constantly trying and failing to acquire a queue, these failed attempts taking away memory bandwidth from blocks that are actually doing work.

To test this hypothesis, we added exponential backoff to the scheduling logic. Recent GPU architectures have introduced the ability for a warp to sleep which will result in the hardware not scheduling this warp for a given time period. If a block fails to acquire any



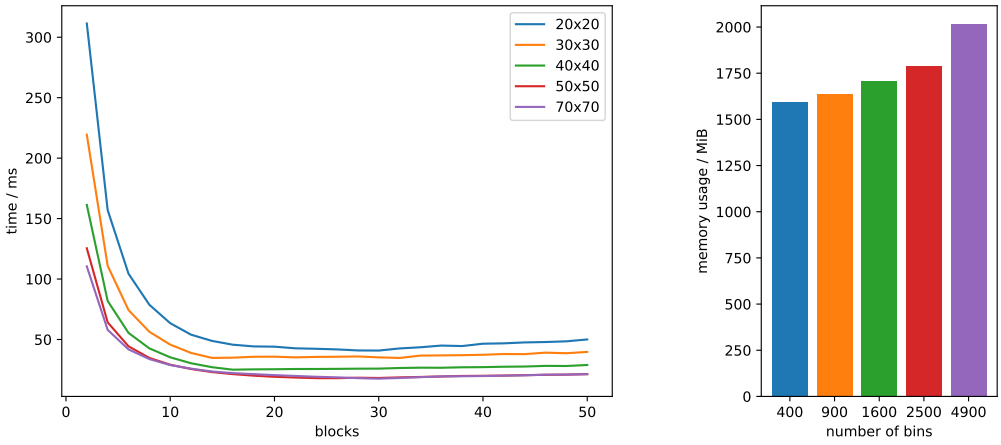
(a) (b)

Figure 7.3.: (a) Comparison of visibility computation time for different numbers of blocks with and without exponential backoff. The introduction of backoff, while not improving best-case performance, does mitigate scheduling overhead from blocks that fail to find work, making the system more scalable robust overall as it can then be set up to cover a wider range of workloads. (b) L2 memory transfer (sum of loads and stores) for different numbers of blocks with and without exponential backoff. Introducing backoff decreases amount of memory transfer. For small number of blocks the improvement is insignificant, but for full occupancy it is more than 30%.

work, all its warps sleep between 100 ns up to 10 000 ns. Sleep duration is doubled each consecutive time the block fails to find work. We would expect this approach to flatten the runtime curve such that it will stop improving at a certain block count but at least not start to increase again as we add more block, ideally keeping it close to the minimum value. This is exactly what we see in Figure 7.3a. Overall memory transfer was reduced by more than 30% for higher block counts. This is in accordance with our hypothesis that a lot of global memory transfer was being wasted on the failed attempts to acquire a queue. Adding exponential backoff made the system much more scalable and robust as it can now be set up to deal with a wider range of workloads without being throttled by scheduling overhead in cases where some views of a scene may not expose sufficient parallelism to utilize all blocks.

7.1.2. Grid Size

Using a finer grid means mapping smaller and smaller screen regions to more and more queues. This can improve performance overall by exposing more parallelism at the cost of increased memory consumption. In Figure 7.4, we see how larger grid sizes result in smaller runtimes. There is a point of diminishing returns, however. For our test scene, we find that there is almost no difference in runtime between using 50×50 bins and using 70×70 bins while the latter comes at significantly higher cost in terms of memory usage.



(a) (b)

Figure 7.4.: Effect of grid size on runtime as well as memory usage. (a) Finer grids are generally faster since they offer more parallelism and, thus, better overall utilization. Note that there is a point (in this example around 50×50 bins) after which using even more bins does not result in a relevant performance improvement anymore. (b) As one would expect, a finer grid also requires proportionally more memory.

7.1.3. Sorting

Due to the nature of our method as an online algorithm, the order in which input primitives are processed can influence the quality of the generated visibility set. Although our algorithm does not rely on sorted geometry, it is not completely unreasonable to expect scenes in real-world applications to be sorted to some degree, at least on the object level since many rendering engines already perform such sorting for various other reasons such as, e.g., to take advantage of an early depth test. In a scene with randomized triangle order, a significant portion of all triangles is processed before large occluders can be formed and, thus, many triangles avoid the computationally expensive occluder test altogether. As can be seen in Figure 7.5, sorting encourages front-most occluders to be built early in the process and, thus, results in a more precise visibility set. At the same time, it forces every successive triangle to be tested against occluders formed early in the process which results in an increase in total runtime.

7.1.4. Occluder Bounds Optimization

One potential optimization to speed up occluder testing (comparing of incoming triangles to the existing occluders) would be by keeping the smallest and largest z values of each occluder as a crude conservative approximation. If the new triangle's largest z is smaller than the occluder's smallest, then the triangle is definitely behind the entire occluder.

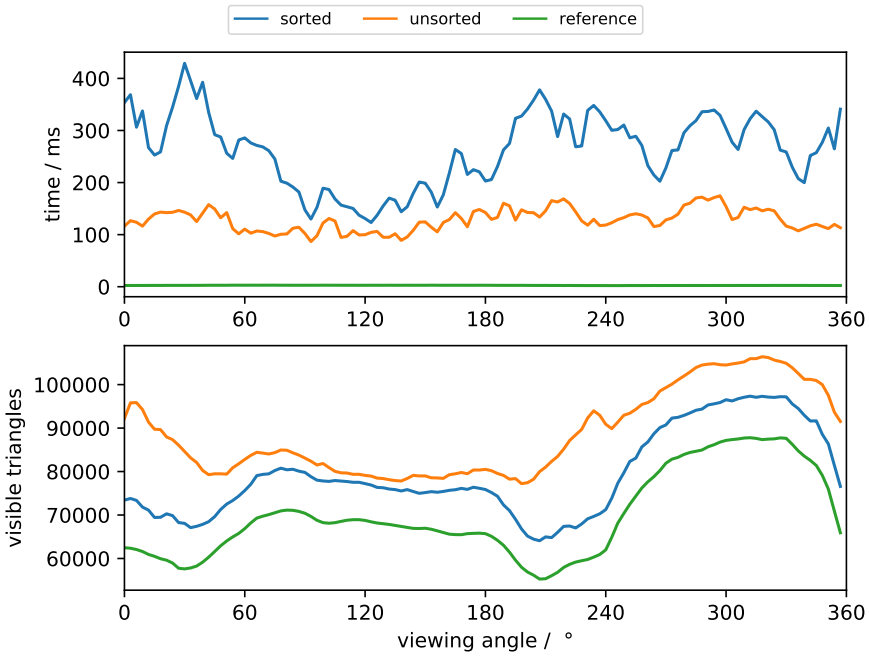
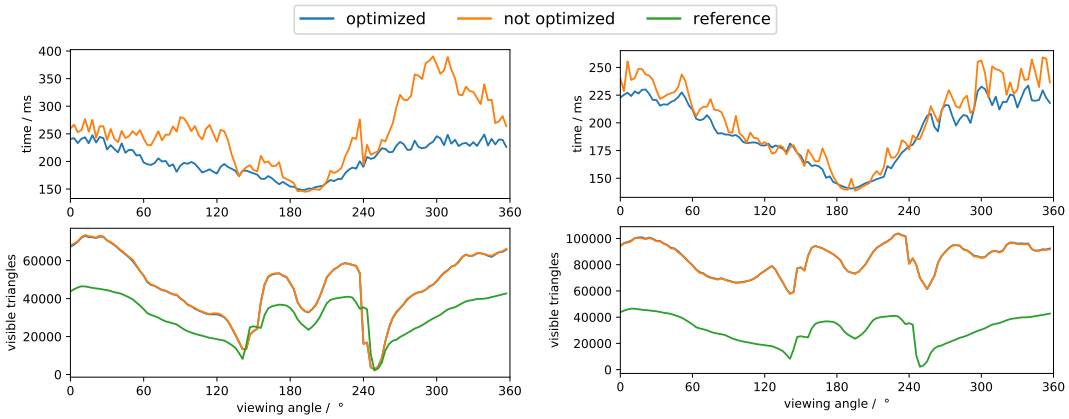


Figure 7.5.: Sorting the "armadillo" scene front-to-back decreases the number of output triangles, while increasing processing time. Note that, at certain camera angles, for example at 120°, both improvement in accuracy as well as increase in runtime are small, while, at other angles, both are significant. We conclude that the improved accuracy is achieved at the cost of more computation that had to be performed as a result of large occluder state being built up early on from the sorted input.



(a) sorted

(b) unsorted

Figure 7.6.: Keeping conservative depth bounds for each occluder to optimize triangle tests improves performance on the sorted scene significantly while having close to no effect on the unsorted scene. Just as expected, the quality of the visibility set itself remains the same.

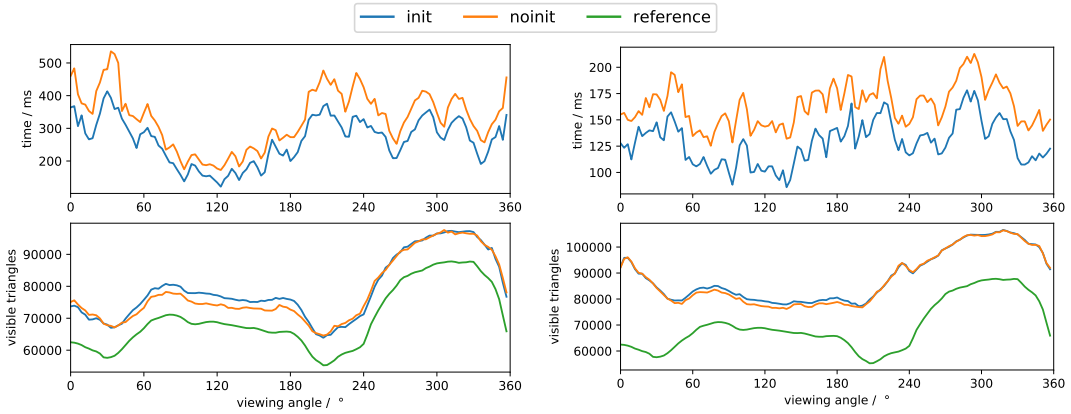
If the new triangle's smallest z is larger than the occluder's largest, then the triangle is definitely in front of the occluder and no further checks are required. This optimization has a significant effect only when presented with sorted input (Figure 7.6). We believe this is due to large occluders forming early in the processing in the sorted scene, while, in the unsorted scene, large occluders form slowly and, thus, many triangles do not overlap with any existing occluders and are just reported as visible without ever participating in a depth comparison to begin with.

7.1.5. Occluder State

Keeping track of per-bin occluder state presents a complex memory management problem that is central to our algorithm. It is subject to a number of parameters and other aspects that give rise to various tradeoffs between speed, memory consumption, and accuracy of the visibility set.

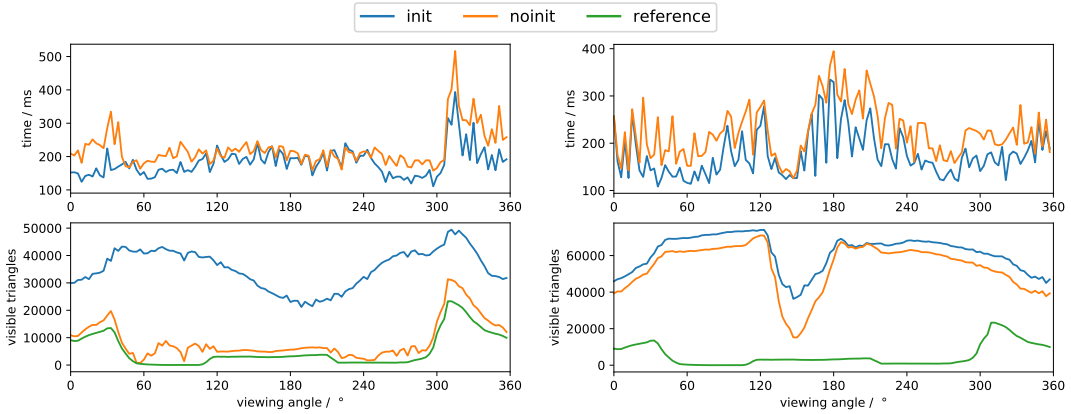
Initialization

In section 6.3.4, we described an optional initialization phase we can run the visibility stage in to quickly create a number of initial occluders before we start the main visibility phase. We can see in Figure 7.7 and Figure 7.8 that turning on this initialization phase has a significant effect on the speed of the algorithm. In case of the "armadillo" scene, initialization has almost no effect on the number of visible triangles, however, since most occluded triangles are discarded during backface culling. The "flat" scene without



(a) sorted (b) unsorted

Figure 7.7.: Effect of visibility stage initialization on the runtime and number of output triangles for the sorted and unsorted "armadillo" scene. Running the visibility stage initialization phase speeds up processing time at the cost of a slightly overestimated visibility set. With sorting, the effect is slightly more pronounced.



(a) sorted (b) unsorted

Figure 7.8.: Effect of visibility stage initialization on the "flat" scene without backface culling. Initialization has a small positive effect on the runtime for both the sorted and unsorted scene. The effect on the output triangle count is small in case of the unsorted scene but, surprisingly, there is a stark increase in overestimation for the sorted scene. This is explained by the fact that the scene in question consists of a few large occluders that occlude many very detailed objects. Without an initialization phase, using a sorted input means the few large occluders become the sole occluder in their respective bins right away, resulting in a visibility set very close to the ground truth. With the initialization phase, at least an entire batch of what would otherwise be occluded geometry is incorporated into the occluder state and, thus, marked as visible.

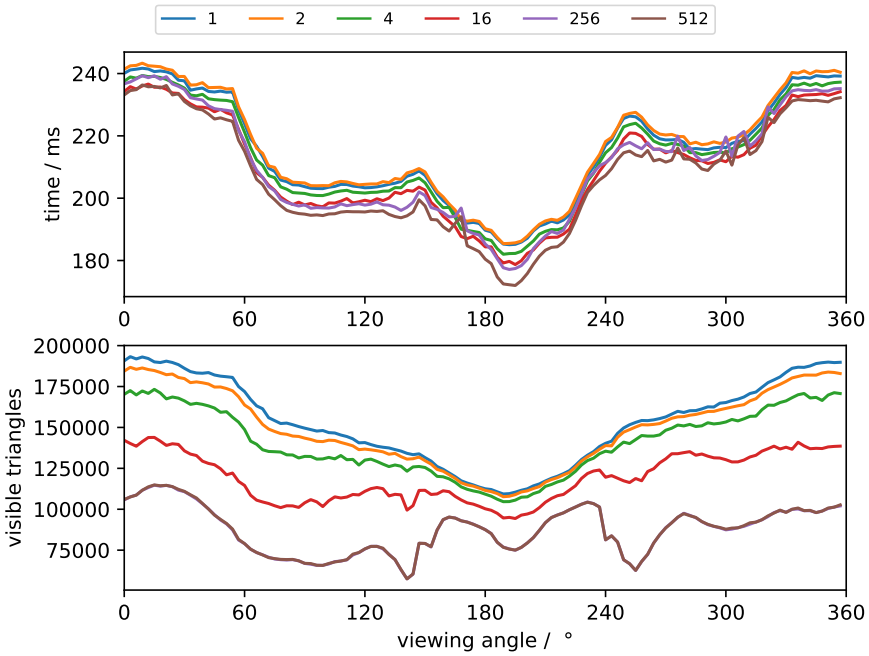


Figure 7.9.: The maximum number of occluders per bin has no relevant influence on the runtime, but a potentially large influence on the accuracy of the algorithm. When the maximum number of occluders is too small — one, two or four occluders — the algorithm fails to detect occlusion on many triangles and, thus, outputs a lot more visible primitives. At the same time, increasing the maximum number of occluders beyond a certain point (256 in this example) does not improve accuracy.

backface culling, on the other hand, offers many more triangles that can be occluded and demonstrates a clear effect of initialization, namely, a small decrease in runtime in return for a small additional level of overestimation in case of an unsorted scene. Unexpectedly, we see a very significant increase in visible triangles when turning on initialization for the sorted scene. Although counter-intuitive at first, there is a logical explanation: initialization takes the first 256 triangles and turns them into occludees, marking them visible in the process. For our 50×50 grid, overestimation can, in the worst case, reach 640 000 triangles. Careful examination of the "flat" scene reveals that, for almost every camera angle, a few large polygons occlude many complex objects. Without initialization, given a sorted input, these large polygons become the sole occluder for many bins while, with initialization, at least one batch of the geometry behind them is turned into occluders as well. Without initialization and given a sorted scene, the resulting visibility of the "flat" is very close to the ground truth, while an unsorted scene combined with initialization provides the fastest runtime.

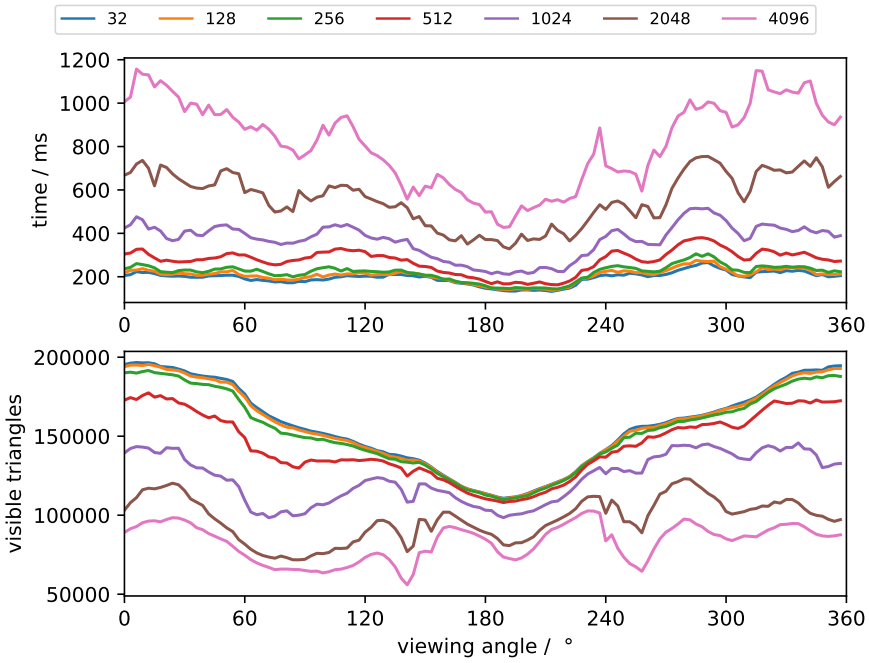


Figure 7.10.: Occluder silhouette edge buffer size has a direct effect on the runtime as well as accuracy of the algorithm. An edge buffer that is too small results in incomplete occluders which result in missed occluded geometry. At the same time, a smaller edge buffer means a lower bound on the maximum number of silhouette tests that may be performed, which results in smaller runtime.

Maximum Number of Occluders per Bin

The number of occluders we can store per bin directly affects the accuracy of the algorithm (see Figure 7.9). When the pool of occluders per bin is too small, what would have potentially grown into large occluders are not incorporated into the bin state, not considered in future visibility checks, and, as a result, occlusion of what would have been triangles behind them is not detected. A sufficiently large maximum number of occluders per bin results in an increased probability to keep around triangles that eventually grow into large occluders and, thus, a much improved visibility set. Note that, while a larger maximum number of occluders per bin means proportionally more memory usage, per-occluder state only consists of an occluder id as well as a pointer to the first triangle batch, making this cost comparatively small. Also note that increasing the maximum number of occluders per bin past a certain point does not result in any further improvement.

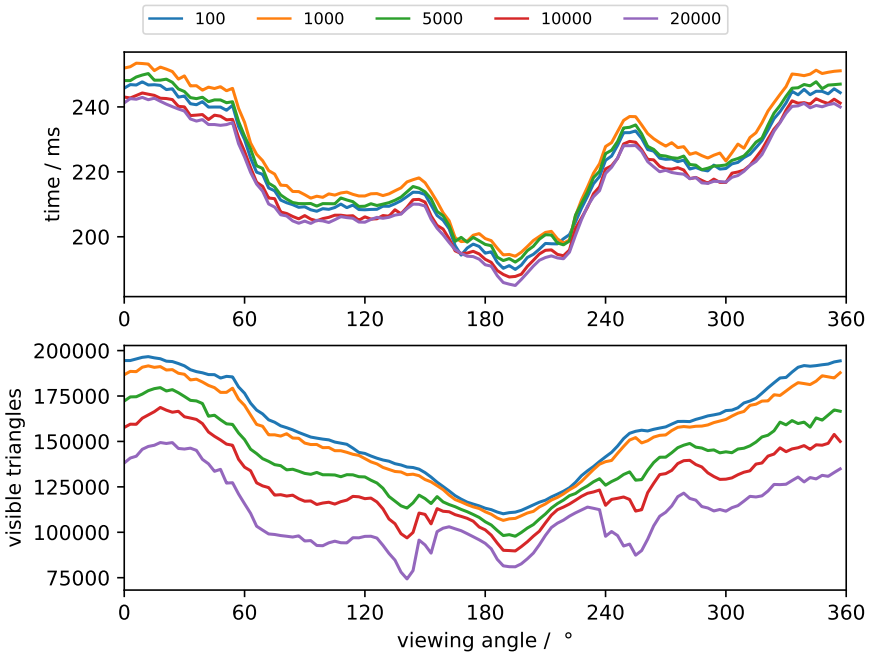
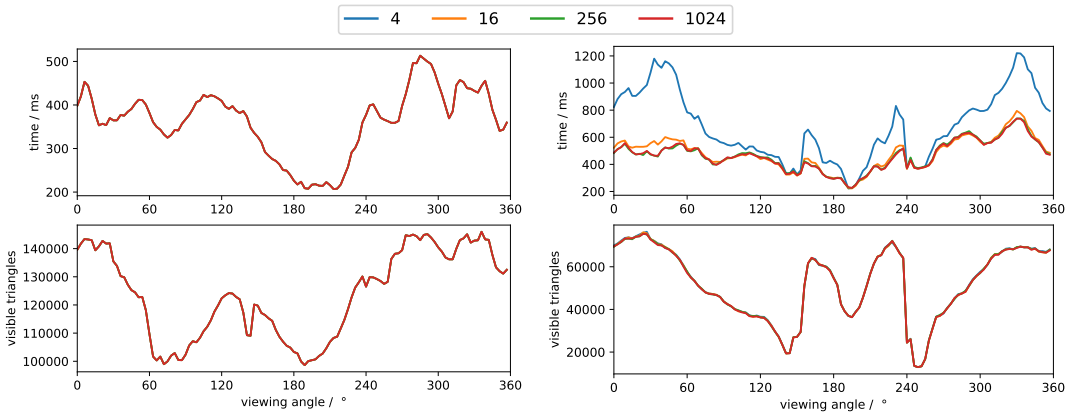


Figure 7.11.: Effect of triangle batch pool size on runtime as well as accuracy of the algorithm. The main thing of importance is to have a large enough pool of triangle batches to enable the visibility stage to create all the occluders that it needs to create.

Occluder Silhouette Edge Buffer Size

Contrary to the maximum number of occluders per bin, the silhouette edge buffer size affects both runtime as well as quality of the visibility set. Note that the occluders formed in our test scene using a fine 50×50 grid are too small to demonstrate this effect. Thus, we have used a coarser 10×10 grid for this test. As can be seen in Figure 7.10, a silhouette edge buffer that is too small results in fast processing but with poor results. This is no surprise: when the edge buffer is full, no more triangles can be added to occluders and, if occluders stop growing before they can start to merge, many occlusions will be missed. However, also for this parameter, we see diminishing returns past a certain point. While increasing the size of the edge buffer directly improves visibility set quality, it also increases runtime due to the larger number of silhouette edge tests that potentially need to be performed. Furthermore, there is a significant memory cost since silhouette edges are kept per bin. Thus, it will be advisable to choose the size of the silhouette edge buffer as large as necessary but as small as possible to balance speed and memory against the desired accuracy.



(a) unsorted

(b) sorted

Figure 7.12.: The size of occluder triangle batches has no notable effect on the accuracy of the algorithm and very little effect on the runtime. (a) For the unsorted scene, triangle batch size has no effect at all. (b) A very small effect can be seen for the sorted scene with visibility stage initialization turned on when batch sizes are small. We attribute the increase in runtime to triangle batch allocation overhead in the initialization phase where a presorted input would likely result in larger occluders being formed and, thus, move overhead from handling the many small batches.

Triangle Batch Pool Size

As we can see in Figure 7.11, allocating a large enough pool of occluder triangle batches is important for algorithm accuracy, less so for speed. As occluders grow and merge, silhouette edges are often removed while occluder triangles are only added to the storage. The occluder triangle storage is less flexible since it stores occluder triangles in linked batches in order to balance between the ability to process triangles within a batch in parallel while also limiting the amount of memory that goes unused on mostly empty occluders. When the pool is too small, available batches are quickly taken up by the first triangles, no further occluders can be created. The visibility stage then can only grow and use those few occluders that were able to allocate a batch. The more active occluders can be tracked, the higher the probability of one of these occluders growing into a large and effective occluder. Thus, a larger triangle batch pool can directly translate into improved accuracy at the cost of more memory usage while overall runtime is not affected by this choice.

Triangle Batch Size

While number of available occluder triangle batches has a strong influence on accuracy, the size of the individual batches has no effect on the accuracy and almost no effect

on the runtime of the algorithm. Tests were performed using a 10×10 grid since the effect was too small to notice on the default 50×50 grid. Only when the batch size is too small and the input is sorted for each camera angle can we notice that processing slows down (see Figure 7.12). Since it only happens for sorted scenes with occluder initialization turned on, we believe that the slow down is explained by the overhead of allocating and linking the batches during the initialization where the fact that geometry is sorted would contribute to larger and, thus, longer occluders being formed.

7.1.6. Discussion

Based on an extensive suite of tests, we were able to describe a number of parameters and optimizations for our algorithm that enable various tradeoffs between speed, accuracy, and memory usage. Most notably, adding exponential backoff made the algorithm overall more scalable and robust to variable workload conditions. Keeping track of depth bounds for each occluder to enable trivial accept or reject tests yielded a significant speedup in certain scenes. Using the visibility stage initialization phase results in further runtime improvement in return for some level of additional overestimation. The silhouette edge buffer and triangle batch pool sizes enable a direct tradeoff between accuracy and memory usage. And, finally, being able to supply our algorithm with a sorted input can vastly improve the quality of the produced visibility set in many cases.

7.2. Comparison with SVGPU

While very different in approach, SVGPU [EHH16] is the only other current method that attempts to solve analytic visibility from a point on the GPU. Contrary to our method, SVGPU is not designed to merely determine visibility of the original scene triangles from a given viewpoint but to instead produce a new set of 2D triangles that together form the projected 2D image of the 3D scene without any overdraw. While it also achieves parallelization through binning, SVGPU is based on silhouette clipping. SVGPU starts by projecting the input geometry into screen space and clipping it against the bins. It then detects silhouette edges with the help of a hash table. Silhouette edges are binned separately. Then every triangle in each bin is split on every silhouette in the same bin, producing polygons. After this step, the bin contains only polygons that are either fully occluded or fully visible. Produced polygons are then tessellated into triangles before triangle-to-triangle occlusion tests are performed on a smaller grid. The resulting triangles become the planar map of triangles that forms the output.

There are a number of consequences to basing a method around silhouette clipping to produce a 2D triangle set. First of all, the input must be a set of non-intersecting triangles which limits applicability since real-world scenes very often contain intersecting

geometry. Second, the complexity of silhouette intersections can vary drastically from frame to frame. Even slight changes in the viewpoint can result in large, unpredictable changes in silhouette complexity and, thus, the number of output triangles which can, in fact, grow larger than the number of triangles in the original scene.

SVGPU runs its stages as separate sequential kernel launches, relying on preallocated global memory buffers to pass intermediate data from one stage to the next. Stability of the algorithm is heavily dependent on the choice of the constants that control the allocation of space for these intermediate buffers. However, constants such as, e.g., the maximum number of silhouette intersections per bin are hard to predict and generally need to be adjusted manually for each scene. Furthermore, as a result of this static allocation strategy, the algorithm requires buffers to be dimensioned for the worst-case while — in scenes with uneven triangle distributions — much of that memory will go unused. As the authors of SVGPU have noted themselves, the memory requirements to accomodate scenes of moderate complexity can grow prohibitively large.

In order to compare our method to SVGPU, we will evaluate both the quality of the visibility set produced by each algorithm as well as runtime and memory usage. To collect this data, we have modified the SVGPU source code, which was graciously provided to us by the authors. We added code to measure algorithm runtime, number of output triangles, and overall memory usage in the same way we do for RV. Additionally, since SVGPU originally does not produce visibility set information, we added code to extract the ids of the input scene triangles which the output triangles produced by SVGPU correspond to. None of these changes should affect the algorithm's correctness or performance.

Despite our best efforts, we were unable to find parameters that would allow SVGPU to successfully process some of our more complex test scenes. For simpler scenes, we were able to determine sets of parameters that result in successful processing for most camera angles. While SVGPU outputs the set of what would have been optimal parameter values after processing a scene, the reported values are only valid for the given view of the given scene and are only reliable if the scene was initially processed using larger memory buffers than would have been required. In order to use optimal settings in our comparison, we would have to manually find overestimated sets of parameters that allow every camera angle of each scene to be processed correctly, and then use these to produce a custom executable for every camera angle of every scene. Doing so would be infeasible not just given the scope of this work but in practice in general.

To nevertheless allow for a meaningful comparison, we will proceed as follows: For the purpose of comparing the quality of the result as well as runtime, we will use the largest parameters we could find for SVGPU that still fit within available GPU memory. For comparing memory usage, we will use the optimal values reported by SVGPU for one of the camera angles. The exact sets of parameters used can be found in Appendix B. In cases where SVGPU fails to process a given input correctly, the data will simply be left blank. All RV tests were run with the exact same settings, which were chosen to

balance accuracy and execution time: we use a 50×50 grid, backface culling is enabled since SVGPU does it as well, we do not pre-sort the scene since doing so would give RV an unfair advantage, we do not run the initialization phase nor use the occluder bounds optimization. For every scene, RV, SVGPU, and our OpenGL reference were run for 120 different camera angles separated by 3° rotations around the z-axis while looking at the scene origin. For each camera position, measurements were averaged over 20 frames, the first 10 frames were discarded as a warm up period.

7.2.1. Quality

To measure the quality of a visibility set, we are interested in the number of *true positives*, i.e., visible triangles correctly identified as visible, the number of *false positives*, i.e., triangles reported as visible that are actually occluded, and the number of *false negatives*, i.e., triangles that should be visible but are incorrectly reported as occluded. We would expect both RV and SVGPU to detect all true positives, no false negatives, RV to conservatively overestimate visibility and, thus, report some false positives, and SVGPU to report no false positives since it is an exact visibility algorithm.

Figure 7.13 shows the results for the "bunny" scene. As expected, RV produces a conservative visibility set with a few false positives. The up to two false negatives reported by RV can be attributed to our reference visibility set being only an approximation obtained via a sampling-based method where occasional artifacts would be expected in combination with a scene that consists of many very small triangles. SVGPU fails to produce a correct result for many camera angles which manifests in some false positive and false negatives as well as one frame of no data. We attribute these issues to the many short silhouette edges of the bunny ears cutting across too many of the small triangles of the rest of the bunny in most frames. However, for those frames where SVGPU succeeds, it does produce an exact result as we would expect. A very similar result can be found for the "bunny and box" scene in Figure 7.14 where there is a clear correlation between the incorrect results and camera angles where the box occludes parts of the bunny.

Although the "boxes with walls" scene — which only consists of 28 non-intersecting boxes and one square for the floor — is comparatively small and simple as far as input geometry is concerned, it has caused substantial problems for SVGPU as can be seen in in Figure 7.15. While not a situation of one complex silhouette cutting across many small triangles like in the "bunny" scene, we believe the cause of the problems in this scene to be the large number of individually simple but overlapping silhouettes that can all come together in the same bin. Although many bins were visibly missing geometry, as can be seen in Figure 7.16, the plot does not show any false negatives which we assume is due to most of the larger triangles in this scene covering more than one bin and, thus, having a high probability of being reported as visible in at least one of the bins it happens to appear in, particularly in a situation where this may happen as a result

of undefined behavior. RV, once again, demonstrates what would be expected behavior. We suspect that the comparatively larger degree of overestimation in the visibility set produced by RV can be explained by the random input triangle order resulting in the few large triangles — which would be the most effective occluders — taking too long to be incorporated into occluder state. We can see some conservative overestimation, fluctuating between 50 and 100 triangles, which is reasonable given the small scene size, large number of bins, and randomized order of input triangles.

In the "deer" and "armadillo" scenes, we can once more observe how the complex interactions of overlapping and intersecting silhouettes can force SVGPU to run out of buffer space and fail (Figure 7.17 and Figure 7.18). Despite its overall low polygon count, the deer presents a challenge due to its antlers, most prominently at camera angles where the antlers are directly in front of the deer face or body which correlate with spikes in false negatives. Similarly to the "bunny" scene, the "armadillo" is problematic due to its many small triangles. Thanks to its more flexible memory management, RV, on the other hand, was able to complete all views of these scenes producing the usual conservative visibility set with some amount of overestimation but no false negatives.

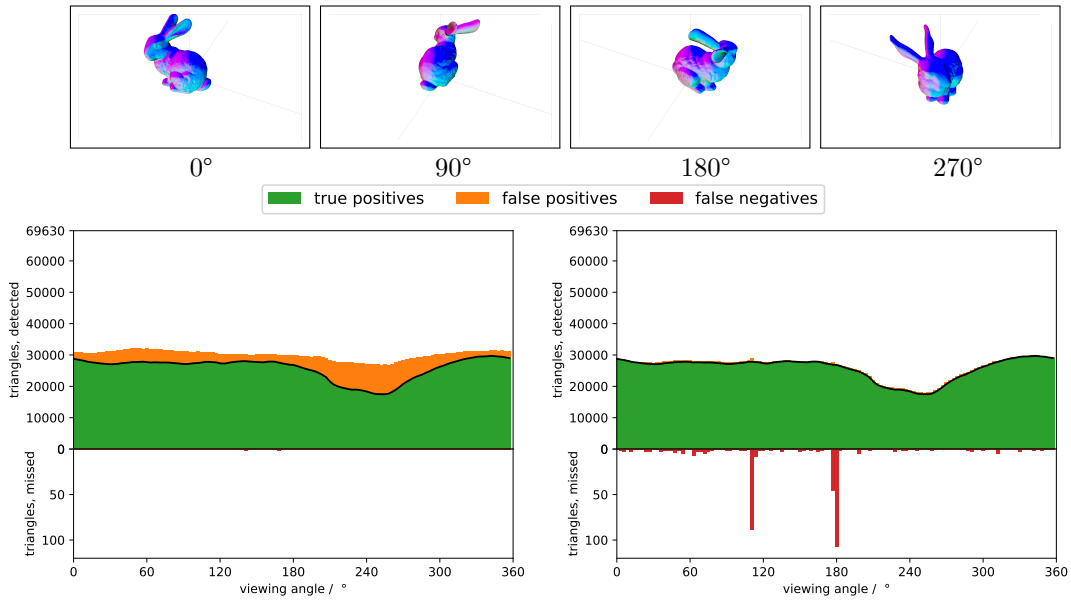


Figure 7.13.: Quality comparison of the visibility sets produced by (a) RV and (b) SVGPU for the "bunny" scene (69630 triangles). RV produces the expected conservative visibility set with some amount of overestimation. The up to two false negatives reported by RV can be attributed to artifacts in our reference visibility set due to being only an approximation. SVGPU fails to produce a result for one camera angle and overall struggles with the complex silhouette intersections caused by the bunny's ears crossing in front of the rest of its body. Note the two spikes in false negatives at around 90° and 180° which correlate with one of the ears being rotated towards the camera such that all the geometry is bunched up in one place. However, contrary to RV, SVGPU does produce an exact result for those camera angles where it does succeed.

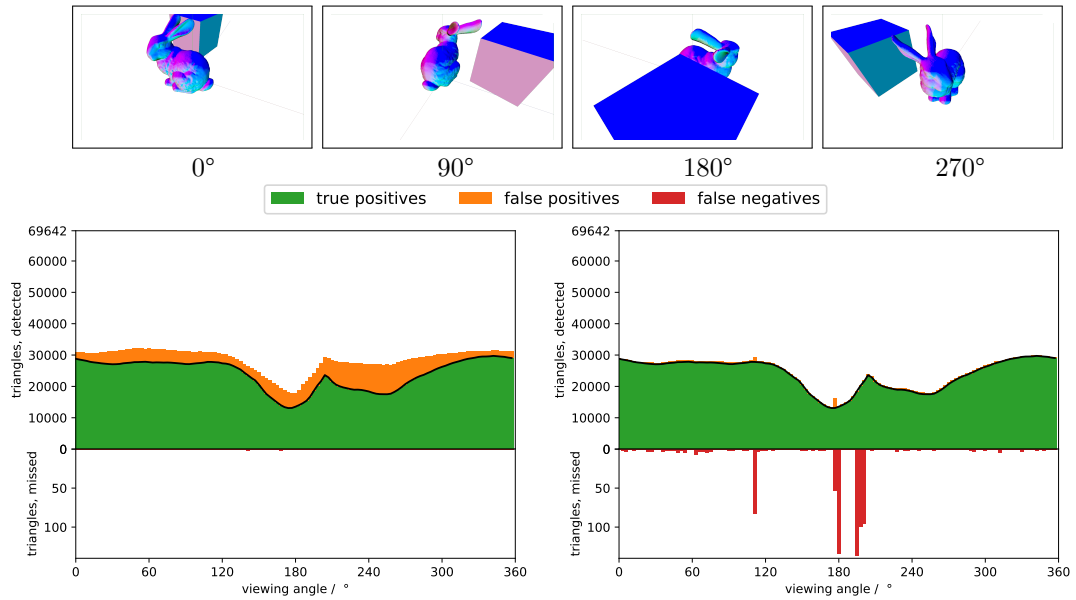
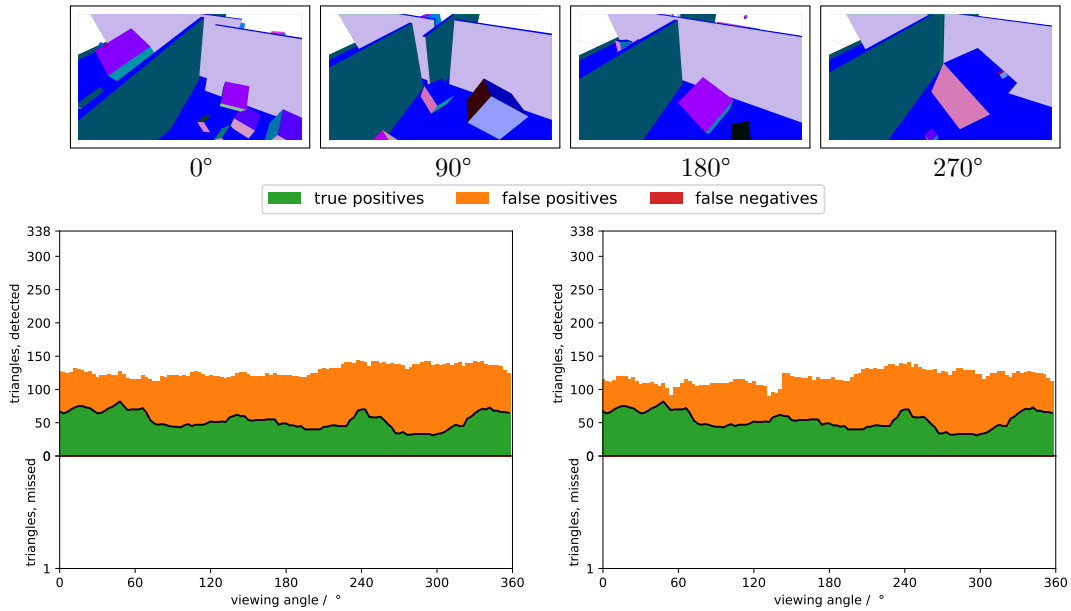
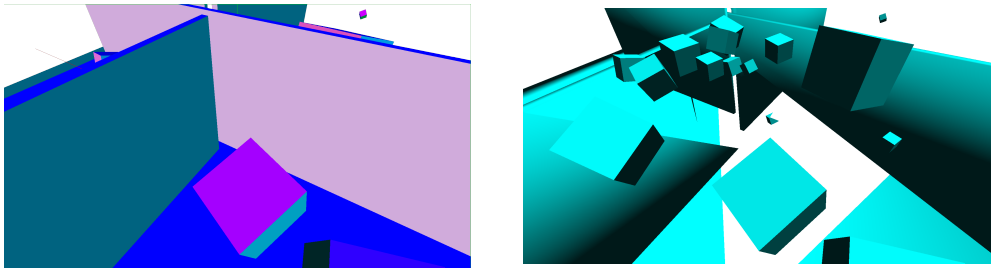


Figure 7.14.: Quality comparison of the visibility sets produced by (a) RV and (b) SVGPU for "bunny and box" scene (69642 triangles). RV produces the expected conservative visibility set with some amount of overestimation. The up to two false negatives reported by RV can be attributed to artifacts in our reference visibility set due to being only an approximation. SVGPU struggles with the complex silhouette intersections caused by the bunny's ears crossing in front of the rest of its body as well as the box occluding part of the bunny. Note the three spikes in false negatives at around 90° and 180° which correlate with one of the ears being rotated towards the camera such that all the geometry is bunched up in one place as well as the box crossing in front of the bunny. However, contrary to RV, SVGPU does produce an exact result for those camera angles where it does succeed.



(a) (b)

Figure 7.15.: Quality comparison of the visibility sets produced by (a) RV and (b) SVGPU for the "boxes with walls" scene (338 triangles). The scene consists of non-intersecting cubes and quads, mimicking a flat with 4 rooms and randomly placed objects of different size. We suspect the relatively large degree of overestimation in the visibility set produced by RV to be due to the few large triangles that would make for effective occluders taking too long to be incorporated into occluder state due to the random order of input triangles. SVGPU, displayed visibly incorrect results in this scene which is, however, not reflected in a large number of false negatives due to all scene triangles being comparatively large and, thus, having a high probability of being marked as visible as a result of a problem in one of the bins in which it happens to appear in. We believe the problems for SVGPU in this scene to be caused by the large number of individually simple but overlapping silhouettes that can all come together in the same bin.



(a) SVGPU (b) RV

Figure 7.16.: One frame from the "flat" scene rendered with RV and SVGPU at a camera angle of 186°. Although the test results do not report any false negatives, we can clearly see that parts of the floor and wall triangles are missing. We believe that this is due to scene triangles being comparatively large and, thus, having a high probability of being marked as visible by at least one of the bins they happen to appear in, especially when this might happen as a result of undefined behavior.

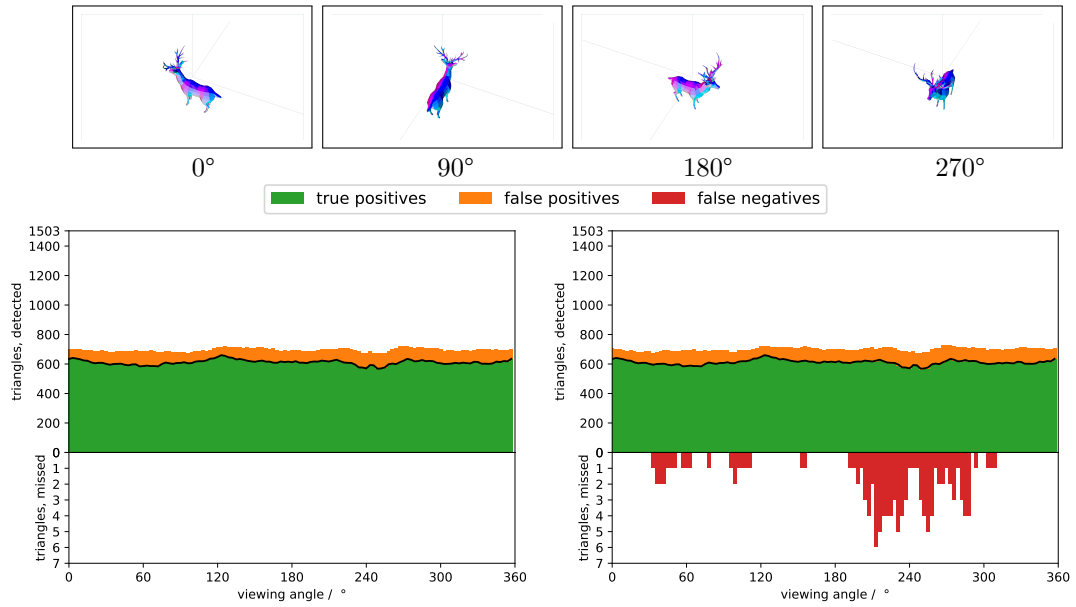
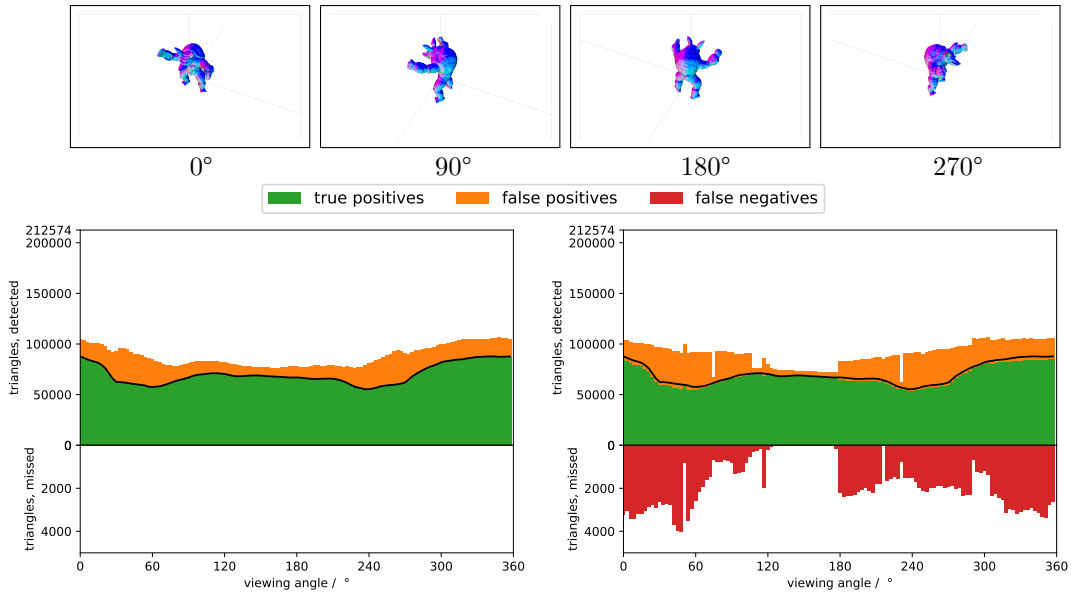


Figure 7.17.: Quality comparison of the visibility sets produced by (a) RV and (b) SVGPU for the "deer" scene (1466 triangles). Despite its overall low polygon count, the deer presents a challenge to SVGPU due to its antlers, particularly at camera angles where the antlers are directly in front of other geometry such as the deer face or body which correlate with spikes in false negatives. By design, the correctness of RV is not limited by scene complexity and, as a result, RV performs comparatively well.



(a) (b)

Figure 7.18.: Quality comparison of the visibility sets produced by (a) RV and (b) SVGPU for the “armadillo” scene (212574 triangles). Similarly to the “bunny” scene, this scene is very problematic for SVGPU due to its large number of small triangles. By design, the correctness of RV is not limited by scene complexity and, as a result, RV performs comparatively well.

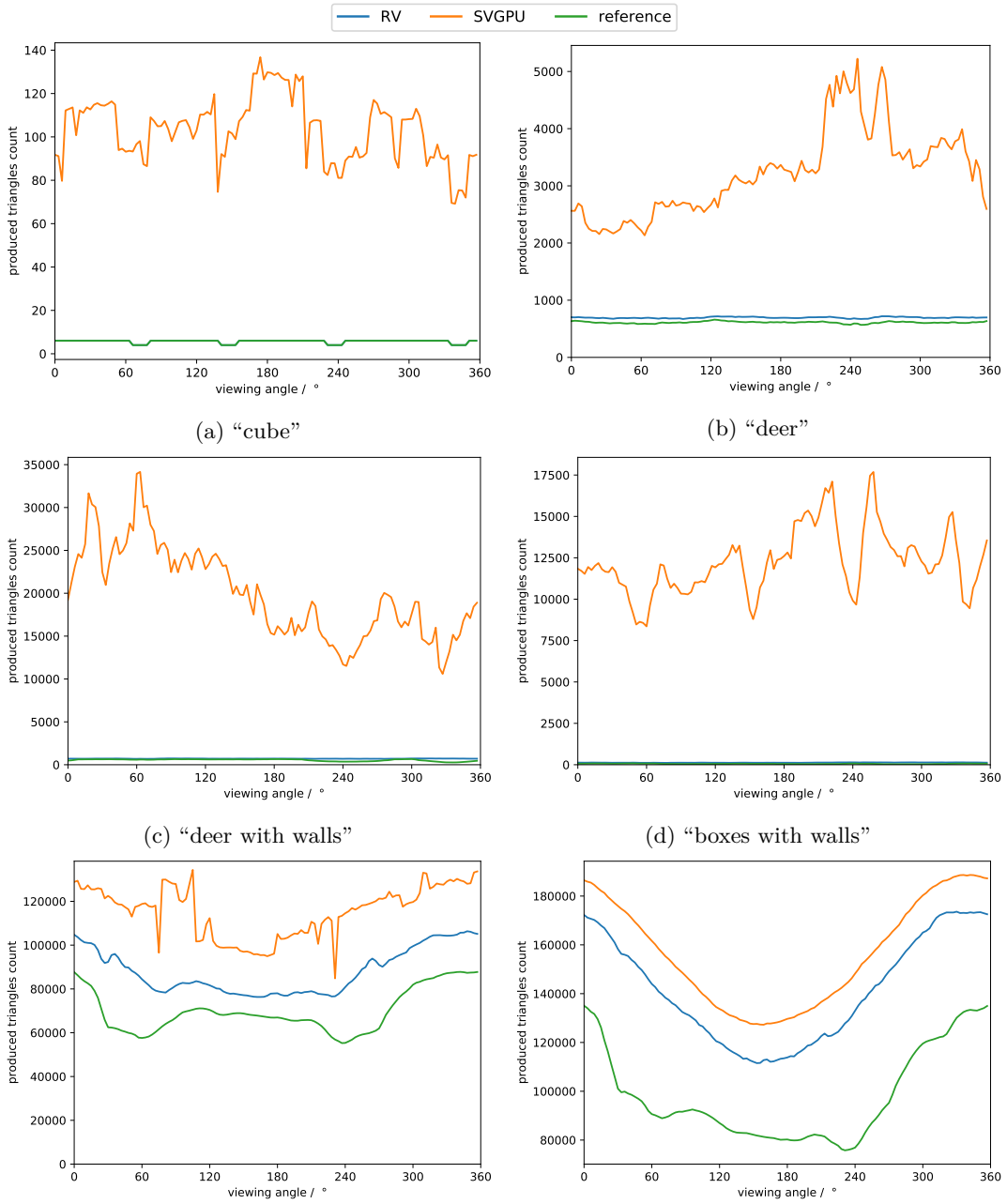
scene	triangle count	memory usage/MiB RV	SVGPU opt.	SVGPU max.
boxes with walls	338	1627	1563	3418
deer	1503	1626	1592	3423
deer and walls	1509	1628	1598	3427
bunny	69630	1634	1967	3825
bunny and box	69642	1634	1757	3825
flat	78045	1667	2808	3924
armadillo	212574	1649	2590	5153

Table 7.1.: Memory consumed by RV and SVGPU while running different test scenes. RV uses the same settings in all tests, thus, its memory footprint is almost identical across all scenes. The small differences come from the memory allocated for the input scene geometry as well as the small output buffer, both proportional to the scene size. For SVGPU, we report both the memory usage given optimal parameters (opt.) which work only for a specific camera angle as well as memory usage given maximum parameters (max.) which are necessary to cover for a range of different camera angles. While RV can operate in bounded memory, due to the chaotic nature of silhouette intersections, SVGPU requires infeasibly large amounts of memory to process certain viewpoints of a given scene.

7.2.2. Performance

While RV operates within bounded memory and the output of RV is limited in size to the size of the input scene itself, due to the chaotic nature of silhouette intersections, it is hard — if not impossible — to predict how much memory will be required by SVGPU in order to produce a correct result. Even if we were to use SVGPU to eventually produce a visibility set rather than a planar triangle map, the algorithm still has to run to completion for triangle-to-triangle occlusion tests to be performed, based on which visibility information could be derived. Thus, we turn to the overall number of triangles produced to serve as an indicator for the scalability and robustness of each method.

Figure 7.19 shows a comparison of the number of output triangles produced by RV, SVGPU, and our OpenGL reference for various scenes. Even for very simple scenes such as the “cube” scene, the number of triangles produced by SVGPU during visibility computation is not only orders of magnitude larger than the size of the input scene itself but also fluctuates as the camera turns, a clear illustration of the issue that unpredictability of silhouette intersections can pose to the stability and scalability of a visibility algorithm. One notable exception is the “armadillo large” scene. Since it consists of many small and similarly-sized triangles, the complexity of silhouette intersections becomes much more well-behaved and coherent across different camera angles. RV, on the other hand, while overestimating visibility sometimes significantly, never produces an unpredictable amount of geometry.



(a) “cube”

(b) “deer”

(c) “deer with walls”

(d) “boxes with walls”

(e) “armadillo”

(f) “armadillo large”

Figure 7.19.: Average number of triangles produced by RV, SVGPU and the OpenGL reference method for increasingly large scenes. We can clearly see the complexity and chaotic nature of silhouette intersections in the excessively large numbers of triangles produced by SVGPU and in how this number fluctuates as the camera angle varies. While there is noticeable degree of overestimation in some scenes, the number of triangles produced by RV is bounded by the size of the input scene.

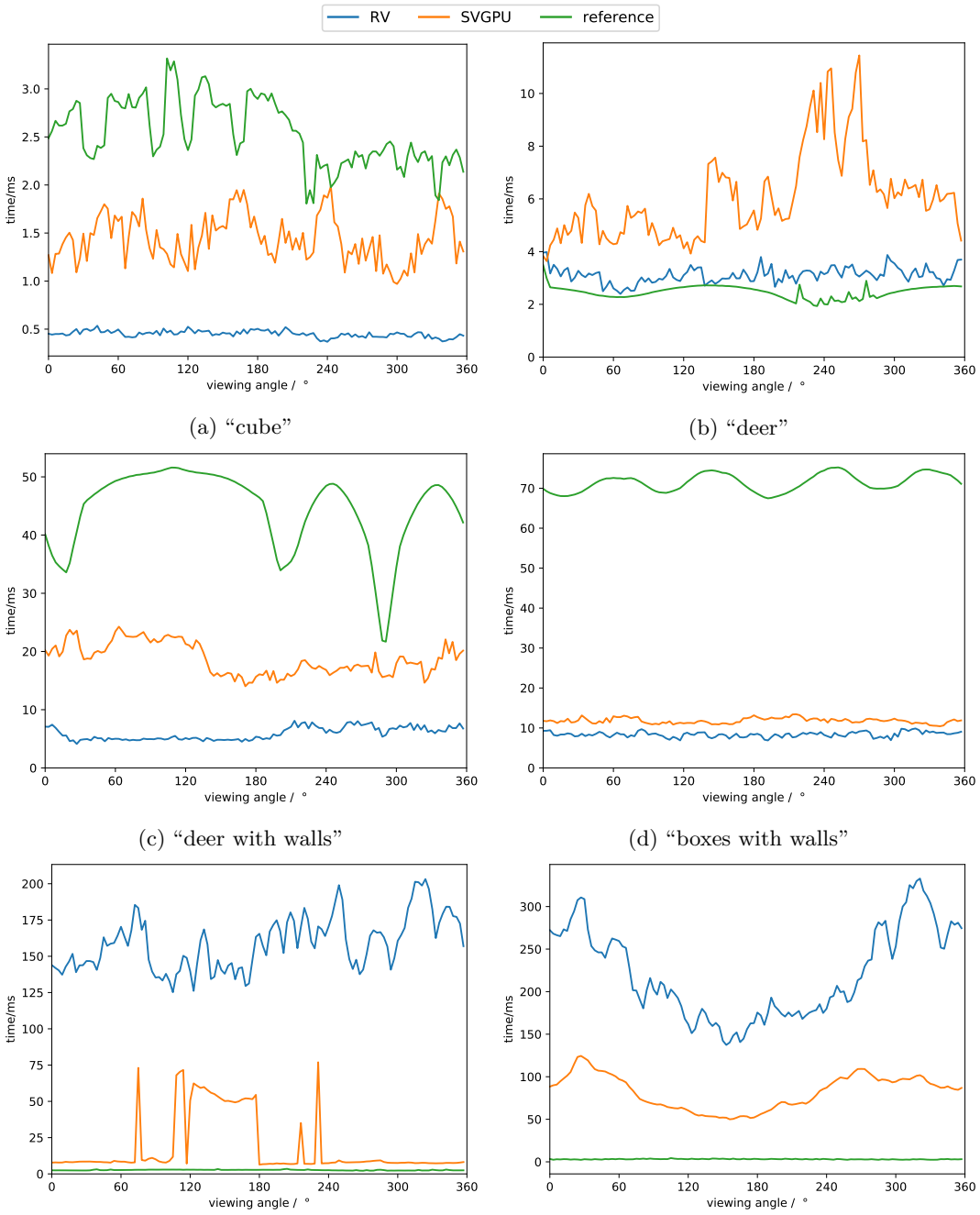
A similar picture emerges when looking at the memory usage of RV and SVGPU in Table 7.1. We compare the memory footprint of SVGPU for hand-picked maximum parameters that are needed to successfully process a range of views as well as optimal settings for a particular viewpoint. Thanks to its streaming architecture, RV can operate in bounded memory while we once again see the complexity of silhouette intersection reflected in the large discrepancy between memory needed to successfully process one particular viewpoint and memory needed to successfully process a range of multiple viewpoints. Note that the maximum parameters for the armadillo scene require close to 80% of the entire 6 GiB of video memory available on the graphics card used in our testing and still fail to produce correct results for many camera angles.

Finally, we compare the runtime of each method on our familiar set of test scenes. As can be seen in Figure 7.20, both analytic methods are faster than the OpenGL reference for simple scenes. We attribute this is to the rasterization-based reference having to render the scene at a very high resolution while the two analytic approaches have to process a comparatively small number of input triangles. For very large scenes, the OpenGL reference is generally much faster than RV or SVGPU. However, we have to keep in mind that RV can be extended to cover a range of viewpoints while the OpenGL implementation can only produce an approximate result for a single viewpoint. For simple to moderately-complex scenes, RV outperforms SVGPU but the gap in performance narrows as scenes get more complex. In very large scenes, SVGPU seemingly outperforms RV. However, it is at this point that we need to recall the results from Figure 7.18 and note that SVGPU was not actually able to produce a correct result in this test case. In fact, we can directly correlate the spikes in runtime for the “armadillo” scene to those camera angles for which the results produced by SVGPU were closer to the correct result. Thus, while the gap in performance between RV and SVGPU narrows as scenes get more complex, we are also getting closer and closer to, and eventually move beyond, the limits of scalability for SVGPU.

7.2.3. Discussion

Comparing runtime and memory usage, RV was able to demonstrate the clear advantages in terms of robustness and scalability of the bounded-memory approach afforded by its streaming pipeline design. While the failure mode for SVGPU when running out of resources is to simply start producing incorrect results, the failure mode for RV is merely a loss in performance and quality of the conservative visibility set, but never a loss in correctness.

Although SVGPU would theoretically allow for generation of a triangle set that can be rendered with no overdraw, the shier amount and unpredictable nature of geometry generated this way makes such an approach seem infeasible given the bandwidth, power, and real-time constraints of transmission to and rendering on an HMD.



(a) “cube”

(b) “deer”

(c) “deer with walls”

(d) “boxes with walls”

(e) “armadillo”

(f) “armadillo large”

Figure 7.20.: Average runtime of RV, SVGPU and the OpenGL reference method when processing increasingly large scenes. While analytic methods outperform the OpenGL reference on simple scenes, the situation is reversed for very complex scenes. Note, however, that analytic methods can potentially be extended to cover a range of multiple viewpoints which is not possible with the rasterization-based reference. While RV is faster than SVGPU for scenes of moderate complexity, it seems that SVGPU would outperform RV in very large scenes such as “armadillo”. However, if we recall the results from chapter 7.2.1, we note that SVGPU failed to produce correct results in these large scenes which explains the suddenly faster runtime.

Overall, we would like to think that the performance, reliability, and capability to process real-world scenes displayed above establish RV as a first step towards truly practical, high-performance analytic potential visibility set computation on the GPU.

8. Conclusion

While being one of the focal points of early computer graphics research, the field of analytical visibility has been neglected in recent years. But, as we have shown, it can become relevant once again in the context of virtual reality. Contrary to the ubiquitous rasterization-based methods, which determine visibility only for one view point, analytical methods can solve visibility for a range of points or an area at once. This ability makes analytical methods a core component of a potential future VR-rendering pipeline that combats latency by rendering frames directly on the headset. A VR-headset processing power is limited by its form factor and the workload on the headset can be kept low by streaming an optimized scene description that covers a number of necessary view points, computed on a powerful server PC.

In this work we have focused on developing an algorithm that can function as a part of described pipeline and efficiently utilize resources of a modern GPU. To do so the algorithm has to satisfy certain criteria, such as streaming input and output, work within bounded memory and expose sufficient parallelism. In an extensive literature review we were unable to find an existing algorithm that would satisfy all our requirements. Thus, as a first step towards potential visibility, we have developed an analytical visibility from a point algorithm, suited for GPU and designed in a way that allows it to be extended to compute visibility from a region in the future.

In order to efficiently utilize computational resources of a modern GPU, our algorithm needs to expose a sufficient level of parallelism. We take advantage of the fact that visibility is a local characteristic in screen space and use binning to parallelize visibility computation. By employing a sort-middle approach we are able to have sufficient parallelism in all stages of the algorithm. By using Megakernel architecture we can run all stages of the algorithm concurrently, which enables flexible load balancing and input streaming. Scheduling logic and queueing system in the Megakernel makes it possible to run in bounded memory.

To understand the developed alrogithm we have performed a detailed investigation of the effect different parameters and optional steps have on it and discussed various tradeoffs they enable. We have found that memory allocation plays a important role in our algorithm's accuracy. Our flexible occluder storage model allows to allocate more memory where it is needed, which improves accuracy, although it comes at a cost of some memory management overhead. Ordering of the input geometry affects the quality of the

result as well, which is expected given an online algorithm. Presorting input geometry produces significantly less overestimated visibility set. Working on even more flexible memory allocation and integrating sorting into the pipeline are potential avenues for future improvements. With the right parameters (pre-sorting of the scene plus allocating enough memory) RV accuracy gets sufficiently close to the reference solution.

While accuracy is important, so it runtime. By changing algorithm parameters we can define the balance between speed and accuracy of our algorithm. As a result of the bounded memory and streaming requirements, our algorithm parameters are scene independent. The algorithm always produces the result, parameters only affect how efficiently it is computed. In comparison to previous work on analytical visibility our algorithm was not only faster, but has also shown more robustness, stability and scalability. It is the first GPU algorithm that's capable of processing real world scenes, both in terms of scale, as well as input geometry requirements, since we do not require manifoldness, watertightness or no self-intersection of the input scene.

While our solution is an order of magnitude slower than hardware-accelerated rasterization-based reference, the prospect of generating potential visibility from a region withing such a time frame is very promising, since the visibility does not have to be updated at the same rate as frames in the VR headset. In fact, taking more time to generate high quality visibility sets may be preferable, since the goal is to reduce the amount of data that need to be transmitted to the headset.

We believe that our work lays out a solid base for future research and the obvious next step would be to work on extending the algorithm to compute visibility from a region. An interesting direction to explore would be exploiting temporal coherence, for example by reusing and updating occluder state over multiple frames instead of rebuilding it from scratch. To reduce memory bandwidth, rather than working with exact scene geometry it might be beneficial to investigate the possibility of working with simplified occluder shapes. Another way to improve effectiveness of occluders could be smarter logic for building occluders, for example prioritizing large polygons or polygons that are larger in screen space. As mentioned before, incorporating sorting into the pipeline could help reduce visibility overestimation. As a side effect, integrating sorting would allow the algorithm to provide sorted output, which in turn enables rendering of transparent objects. As mentioned before as well, more flexible memory management would be another key area of potential improvement. One potential way of making memory more flexible could be to use adaptive grid with dynamic queues, so that the work is distributed more evenly over the queues and empty bins don't occupy space in the memory. Processing every queue by multiple blocks is another interesting research direction, although less straightforward than other suggestions, since it will require updating all visibility data structures concurrently.

All in all we feel that we have only tipped our toes into the ocean of analytical visibility, an extremely interesting and undeservedly forgotten topic. The advances of the modern

GPU and new challenges the field of computer graphics is faced with call for its return. We believe that there is a lot of potential in developing those topics together further and we can't wait to see what the future holds.

Appendices

A. Overview Tables

scene	triangle count	memory/MiB	overestimation/%		false "-"		produced tris/%	
			min	max	min	max	min	max
"boxes with walls"	338	1626	12.72 0	32.25 0	0.0	0.0	33.14	42.43
deer	1503	1626	3.69 0	7.22 0	0.0	0.0	44.64	47.86
deer and walls	1509	1626	3.69 0	30.19 0	0.0	0.0	45.38	48.70
bunny	69630	1634	2.43 0	13.72 0	0.0	2.0	38.54	45.99
bunny and box	69642	1634	2.43 0	13.69 0	0.0	2.0	25.69	45.99
flat	78045	1667	5.00 0	51.32 0	0.0	3.0	11.64	55.54
armadillo	212574	1649	3.95 0	15.96 0	0.0	3.0	35.91	50.03

Table A.1.: Overview of test results for RV, for rotating scenes in increasing scene size (triangle count) order. The table shows memory consumed when processing every scene, as well as statistical information. Fourth and fifth columns contain smallest and largest overestimation of visibility (smallest and largest number of false positives over all rendered frames and rotation angles) in percentage of scene size. Sixth and seventh columns contain smallest and largest false negatives in absolute values, over all frames and rotation angles as well. The last two columns contain smallest and largest number of triangles produced by the algorithm, in percentage of the original scene size.

scene	triangle count	memory/MiB	overestimation/%		false "-"		produced tris/%	
			min	max	min	max	min	max
"boxes with walls"	338	3418	6.51 0	29.29 0	0.0	0.0	2471.92	5232.13
deer	1503	3423	3.93 0	7.72 0	0.0	6.0	141.98	347.48
deer and walls	1509	3427	3.91 0	30.53 0	0.0	5.0	701.81	2263.42
bunny	69630	3825	0.21 0	1.96 0	0.0	107.6	41.54	70.65
bunny and box	69642	3825	0.18 0	4.43 0	0.0	136.0	31.60	71.42
flat	78045	3924	13.25 0	57.43 0	0.0	11.15	52.42	106.24
armadillo	212574	5153	2.23 0	19.36 0	8.0	4015.05	39.90	63.22

Table A.2.: Overview of test results for SVGPU for rotating scenes in increasing scene size (triangle count) order. The table shows memory consumed when processing every scene, as well as statistical information. Fourth and fifth columns contain smallest and largest overestimation of visibility (smallest and largest number of false positives over all rendered frames and rotation angles) in percentage of scene size. Sixth and seventh columns contain smallest and largest false negatives in absolute values, over all frames and rotation angles as well. The last two columns contain smallest and largest number of triangles produced by the algorithm, in percentage of the original scene size.

B. SVGPU parameters

parameter	value
BIN_OCCUPANCY	6000
REBIN_OCCUPANCY	3000
CANDIDATES	3000
CLIP_POLYS	3000
CLIP_VERTICES	6000
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	3000
MAX_COLLISIONS	100
MAGNIFACTION	800

Table B.1.: SVGPU parameter values used for visibility set quality comparison.

parameter	value
BIN_OCCUPANCY	3559
REBIN_OCCUPANCY	3559
CANDIDATES	1237
CLIP_POLYS	504
CLIP_VERTICES	3291
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	181
MAX_COLLISIONS	8
MAGNIFACTION	2.201267

Table B.2.: SVGPU parameter values used for memory comparison for the “bunny” scene.

parameter	value
BIN_OCCUPANCY	1137
REBIN_OCCUPANCY	1087
CANDIDATES	605
CLIP_POLYS	287
CLIP_VERTICES	1443
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	312
MAX_COLLISIONS	2
MAGNIFACTION	2.050851

Table B.3.: SVGPU parameter values used for memory comparison for the “deer” scene.

parameter	value
BIN_OCCUPANCY	1144
REBIN_OCCUPANCY	1094
CANDIDATES	605
CLIP_POLYS	295
CLIP_VERTICES	1526
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	312
MAX_COLLISIONS	2
MAGNIFACTION	2.053326

Table B.4.: SVGPU parameter values used for memory comparison for the “deer with walls” scene.

parameter	value
BIN_OCCUPANCY	1283
REBIN_OCCUPANCY	65
CANDIDATES	22629
CLIP_POLYS	1318
CLIP_VERTICES	6588
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	43487
MAX_COLLISIONS	7
MAGNIFACTION	3.162370

Table B.5.: SVGPU parameter values used for memory comparison for the “flat” scene.

parameter	value
BIN_OCCUPANCY	310
REBIN_OCCUPANCY	292
CANDIDATES	142
CLIP_POLYS	210
CLIP_VERTICES	1040
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	26
MAX_COLLISIONS	2
MAGNIFACTION	6.744566

Table B.6.: SVGPU parameter values used for memory comparison for the “small flat” scene.

parameter	value
BIN_OCCUPANCY	7998
REBIN_OCCUPANCY	3831
CANDIDATES	3169
CLIP_POLYS	1311
CLIP_VERTICES	7426
FINAL_POLYS	0
FINAL_VERTICES	0
SILHOUETTE_BIN_SIZE	300
MAX_COLLISIONS	10
MAGNIFACTION	4

Table B.7.: SVGPU parameter values used for memory comparison for the “armadillo” scene.

Bibliography

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks Jr. "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments." In: *SIGGRAPH Comput. Graph.* 24.2 (Feb. 1990), pp. 41–50. ISSN: 0097-8930. DOI: 10.1145/91394.91416. URL: http://doi.acm.org/10.1145/91394.91416.

[App68] Arthur Appel. "Some Techniques for Shading Machine Renderings of Solids." In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 37–45. DOI: 10.1145/1468075.1468082. URL: http://doi.acm.org/10.1145/1468075.1468082.

[AWJ13] Thomas Auzinger, Michael Wimmer, and Stefan Jeschke. "Analytic Visibility on the GPU." In: *Computer Graphics Forum (Proceeding of EUROGRAPHICS 2013)* 32.2 (May 2013), pp. 409–418. ISSN: 1467-8659. URL: https://www.cg.tuwien.ac.at/research/publications/2013/Auzinger_2013_AnaVis/.

[Bas+17] E. Bastug, M. Bennis, M. Medard, and M. Debbah. "Toward Interconnected Virtual Reality: Opportunities, Challenges, and Enablers." In: *IEEE Communications Magazine* 55.6 (June 2017), pp. 110–117. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1601089.

[Bit02] Jiří Bittner. "Hierarchical Techniques for Visibility Computations." PhD thesis. Nov. 2002.

[BW03] Jiří Bittner and Peter Wonka. "Visibility in Computer Graphics." In: *Environment and Planning B: Planning and Design* 30.5 (2003), pp. 729–755. DOI: 10.1068/b2957. eprint: http://dx.doi.org/10.1068/b2957. URL: http://dx.doi.org/10.1068/b2957.

[Car13] John Carmack. *Latency Mitigation Strategies*. 2013. URL: https://web.archive.org/web/20140719053303/http://www.altdev.co/2013/02/22/latency-mitigation-strategies/ (visited on 01/13/2020).

[Cat74] Edwin Earl Catmull. "A Subdivision Algorithm for Computer Display of Curved Surfaces." AAI7504786. PhD thesis. 1974.

[CT97] Satyan Coorg and Seth Teller. "Real-time Occlusion Culling for Models with Large Occluders." In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. I3D '97. Providence, Rhode Island, USA: ACM, 1997, 83–ff. ISBN: 0-89791-884-3. DOI: 10.1145/253284.253312. URL: http://doi.acm.org/10.1145/253284.253312.

[CT99] Satyan Coorg and Seth Teller. "Temporally Coherent Conservative Visibility." In: *Comput. Geom. Theory Appl.* 12.1-2 (Feb. 1999), pp. 105–124. ISSN: 0925-7721. DOI: 10.1016/S0925-7721(98)00037-6. URL: http://dx.doi.org/10.1016/S0925-7721(98)00037-6.

[Duh+04] Henry Been-Lirn Duh, Donald E. Parker, James O. Philips, and Thomas A. Furness. ""Conflicting" Motion Cues to the Visual and Vestibular Self- Motion Systems Around 0.06 Hz Evoke Simulator Sickness." In: *Human Factors* 46.1 (2004). PMID: 15151161, pp. 142–153. DOI: 10.1518/hfes.46.1.142.30384. eprint: https://doi.org/10.1518/hfes.46.1.142.30384. URL: https://doi.org/10.1518/hfes.46.1.142.30384.

[Dur99] Fr' edo Durand. "3D Visibility: Analytical Study and Applications." PhD thesis. Grenoble, France: Universit' e Joseph Fourier, 1999.

[Dur+00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. "Conservative Visibility Preprocessing Using Extended Projections." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 239–248. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344891. URL: http://dx.doi.org/10.1145/344779.344891.

[EHH16] Apollo I. Ellis, Warren Hunt, and John C. Hart. "SVGPU: Real Time 3D Rendering to Vector Graphics Formats." In: *Proceedings of High Performance Graphics*. HPG '16. Dublin, Ireland: Eurographics Association, 2016, pp. 13–21. ISBN: 978-3-03868-008-6. URL: http://dl.acm.org/citation.cfm?id=2977336.2977339.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. "On Visible Surface Generation by a Priori Tree Structures." In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 124–133. ISSN: 0097-8930. DOI: 10.1145/965105.807481. URL: http://doi.acm.org/10.1145/965105.807481.

[Gel+90] W. Gellert, S. Gottwald, M. Hellwich, H. Kästner, and H. Künstner. *VNR Concise Encyclopedia of Mathematics*. Springer, 1990. ISBN: 0442205902.

[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. Anaheim, CA: Association for Computing Machinery, 1993, pp. 231–238. ISBN: 0897916018. DOI: 10.1145/166117.166147. URL: https://doi.org/10.1145/166117.166147.

[Hud+97] T. Hudson, Dinesh Manocha, J. Cohen, Ming Lin, K. Hoff, and H. Zhang. "Accelerated occlusion culling using shadow frusta." English (US). In: *Proceedings of the Annual Symposium on Computational Geometry*. Ed. by Anon. ACM, 1997, pp. 1–10.

[Jon71] C. B. Jones. "A new approach to the 'hidden line' problem." In: *The Computer Journal* 14.3 (1971), pp. 232–237. DOI: 10.1093/comjnl/14.3.232. eprint: /oup/backfile/content_public/journal/comjnl/14/3/10.1093/comjnl/14.3.232/2/140232.pdf. URL: http://dx.doi.org/10.1093/comjnl/14.3.232.

[Ken+18] Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. "A High-Performance Software Graphics Pipeline Architecture for the GPU." In: *ACM Trans. Graph.* 37.4 (Nov. 2018). DOI: 10.1145/3197517.3201374.

[Ker+18] Bernhard Kerbl, Michael Kenzel, Joerg H Mueller, Dieter Schmalstieg, and Markus Steinberger. "The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU." In: *Proceedings of the International Conference on Supercomputing*. ICS '18. Beijing, China, 2018. DOI: 10.1145/3205289.3205291. URL: http://doi.org/10.1145/3205289.3205291.

[Khr17] Khronos. *The OpenGL Graphics System: A Specification*. Ed. by Mark Segal and Kurt Akeley. Version 4.5. The Khronos Group Inc. July 2017.

[KBS95] E.M. Kolasinski, U.S. Army Research Institute for the Behavioral, and Social Sciences. *Simulator Sickness in Virtual Environments*. Tech. rep. 1027. U.S. Army Research Institute for the Behavioral and Social Sciences, 1995. URL: https://books.google.at/books?id=9Na5rQEACAAJ.

[Kov07] Vít Kovalčík. "Occlusion Culling in Dynamic Scenes [online]." Disertační práce. Masarykova univerzita, Fakulta informatiky, Brno, 2007. URL: http://is.muni.cz/th/4269/fi_d/.

[Möl97] Tomas Möller. "A Fast Triangle-triangle Intersection Test." In: *J. Graph. Tools* 2.2 (Nov. 1997), pp. 25–30. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487472. URL: http://dx.doi.org/10.1080/10867651.1997.10487472.

[Mol+94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. "A Sorting Classification of Parallel Rendering." In: *IEEE Comput. Graph. Appl.* 14.4 (July 1994).

[Mue+18] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. "Shading Atlas Streaming." In: *ACM Trans. Graph.* 37.6 (Dec. 2018), 199:1–199:16. ISSN: 0730-0301. DOI: 10.1145/3272127.3275087. URL: http://doi.acm.org/10.1145/3272127.3275087.

[NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. "A Solution to the Hidden Surface Problem." In: *Proceedings of the ACM Annual Conference - Volume 1*. ACM '72. Boston, Massachusetts, USA: ACM, 1972, pp. 443–450. DOI: 10.1145/800193.569954. URL: http://doi.acm.org/10.1145/800193.569954.

[NBG02] S. Nirenstein, E. Blake, and J. Gain. "Exact From-region Visibility Culling." In: *Proceedings of the 13th Eurographics Workshop on Rendering*. EGRW '02. Pisa, Italy: Eurographics Association, 2002, pp. 191–202. ISBN: 1-58113-534-3. URL: http://dl.acm.org/citation.cfm?id=581896.581921.

[NVI20] NVIDIA. *CUB. API Reference*. Version 11.2. NVIDIA Corporation. Dec. 2020.

[NVI21] NVIDIA. *CUDA C++ Programming Guide*. Version 11.2. NVIDIA Corporation. Jan. 2021.

[RB75] J. T. Reason and J. J Brand. *Motion sickness*. Academic Press, 1975. ISBN: 0125840500.

[Sch+00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. "Conservative Volumetric Visibility with Occluder Fusion." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 229–238. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344886. URL: http://dx.doi.org/10.1145/344779.344886.

[Shi62] M. Shimrat. "Algorithm 112: Position of Point Relative to Polygon." In: *Commun. ACM* 5.8 (Aug. 1962), pp. 434–. ISSN: 0001-0782. DOI: 10.1145/368637.368653. URL: http://doi.acm.org/10.1145/368637.368653.

[Ste+14] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. "Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU." In: *ACM Trans. Graph.* 33.6 (Nov. 2014), 228:1–228:11. ISSN: 0730-0301. DOI: 10.1145/2661229.2661250. URL: http://doi.acm.org/10.1145/2661229.2661250.

[SRJ11] Hans Strasburger, Ingo Rentschler, and Martin Jüttner. "Peripheral vision and pattern recognition: A review." In: *Journal of vision* 11 (May 2011), p. 13. DOI: 10.1167/11.5.13.

[Str74] Wolfgang Straßer. "Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten." PhD thesis. 1974.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. "A Characterization of Ten Hidden-Surface Algorithms." In: *ACM Comput. Surv.* 6.1 (Mar. 1974), pp. 1–55. ISSN: 0360-0300. DOI: 10.1145/356625.356626. URL: http://doi.acm.org/10.1145/356625.356626.

[Tre77] M Treisman. "Motion sickness: an evolutionary hypothesis." In: *Science* 197.4302 (1977), pp. 493–495. ISSN: 0036-8075. DOI: 10.1126/science.301659. eprint: http://science.sciencemag.org/content/197/4302/493.full.pdf. URL: http://science.sciencemag.org/content/197/4302/493.

[War69] John Edward Warnock. "A Hidden Surface Algorithm for Computer Generated Halftone Pictures." AAI6919002. PhD thesis. 1969.

[WA77] Kevin Weiler and Peter Atherton. "Hidden Surface Removal Using Polygon Area Sorting." In: *SIGGRAPH Comput. Graph.* 11.2 (July 1977), pp. 214–222. ISSN: 0097-8930. DOI: 10.1145/965141.563896. URL: http://doi.acm.org/10.1145/965141.563896.

[Wyl+67] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl. "Half-tone Perspective Drawings by Computer." In: *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. AFIPS '67 (Fall). Anaheim, California: ACM, 1967, pp. 49–58. DOI: 10.1145/1465611.1465619. URL: http://doi.acm.org/10.1145/1465611.1465619.

[Zha+97] 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. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 77–88. ISBN: 0897918967. DOI: 10.1145/258734.258781. URL: https://doi.org/10.1145/258734.258781.