TU WIEN Informatics

Sichtbarkeitsvorberechnung mit RTX Ray Tracing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Thomas Bernhard Koch, BSc
Matrikelnummer 01526232

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Michael Wimmer, Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn.

Wien, 3. Juni 2020

Thomas Bernhard Koch	Michael Wimmer

TU WIEN Informatics

Visibility Precomputation with RTX Ray Tracing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Thomas Bernhard Koch, BSc
Registration Number 01526232

to the Faculty of Informatics

at the TU Wien

Advisor: Michael Wimmer, Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn.

Vienna, 3rd June, 2020

Thomas Bernhard Koch	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Thomas Bernhard Koch, BSc

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

Wien, 3. Juni 2020

Thomas Bernhard Koch

Danksagung

Ich möchte mich bei allen bedanken, die mich während meines Studiums aktiv unterstützt und begleitet haben. Das umfasst ProfessorInnen, StudienkollegInnen und ganz besonders meine Familie, die das Studium überhaupt ermöglicht hat.

Auch möchte ich mich bei meinem Professor Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer für die gute Betreuung danken, wodurch es möglich wurde, an einem interessanten Thema zu arbeiten. Durch den Einsatz von neuesten Technologien konnte ich mir relevantes Wissen aneignen, das mich auch in meiner zukünftigen Laufbahn weiterbringen wird.

Ein weiterer Dank geht an die Kollegen von VIRES Simulationstechnologie GmbH, die mich während meiner Arbeit durch Ressourcen und Diskussionen unterstützt haben.

Acknowledgements

I would like to thank all the people who supported and accompanied me during my studies. This includes professors, colleagues, and, most notably, my family that made my studies possible in the first place.

I also wish to thank my advisor, Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer for his guidance, which enabled me to work on an exciting and relevant topic. The knowledge acquired by working on this thesis, especially through the use of cutting-edge technology, will also be beneficial for my future career.

I would also like to thank my colleagues at VIRES Simulationstechnologie GmbH for their support by providing resources and discussions.

Kurzfassung

Die Sichtbarkeitsberechnung ist ein typisches Problem im Feld der Computergraphik. Beispiele sind Occlusion Culling, bei dem verdeckte Teile der Szene verworfen werden, um die Performance zu steigern, oder auch die Berechnungen von globalen Beleuchtungseffekten, die auf der Sichtbarkeiten von Punktpaaren basieren. In dieser Arbeit wird ein aggressiver Sichtbarkeitsalgorithmus, Guided Visibility Sampling++, der die Sichtbarkeit von einer Fläche ausgehend berechnet, vorgestellt. Der Algorithmus basiert auf Guided Visibility Sampling und verbessert diesen, wodurch eine genauerere Lösung in kürzerer Zeit berechnet werden kann. Eine Kombination von verschiedenen Algorithmen und intelligenten Abtaststrategien werden verendet, um mittels Raycasting eine Menge an Dreiecken zu ermitteln, die von einer flachen oder volumetrischen Region aus sichtbar ist. Diese gefundene Menge an Dreiecken wird Potentially Visible Set (PVS) genannt. Der vorgestellte Algorithmus findet Dreiecke effizient durch initiales Zufallsabtasten der Szene und anschließend ausgeführte intelligente Erkundungsstrategien. Der Algorithmus terminiert durch ein Abbruchkriterium. Eine moderne Implementierung, basierend auf der Vulkan API und RTX Raytracing wird präsentiert. Unsere GPU-basierte GVS++ ist über vier Größenordnungen schneller als die ursprüngliche CPU-basierte Implementierung von GVS. Experimente auf verschiedenen Szenen zeigen, dass die vorgestellte Technik schneller und genauer ist als vergleichbare Techniken.

Abstract

Visibility computation is a common problem in the field of computer graphics. Examples include occlusion culling, where parts of the scene are culled away, or global illumination simulations, which are based on the mutual visibility of pairs of points to calculate lighting. In this thesis, an aggressive from-region visibility technique called Guided Visibility Sampling++ (GVS++) is presented. The proposed technique improves the Guided Visibility Sampling algorithm through improved sampling strategies, thus achieving low error rates on various scenes, and being over four orders of magnitude faster than the original CPU-based Guided Visibility Sampling implementation. We present intelligent sampling strategies that use ray casting to determine a set of triangles visible from a flat or volumetric rectangular region in space. This set is called a potentially visible set (PVS). Based on initial random sampling, subsequent exploration phases progressively grow an intermediate solution. A termination criterion is used to terminate the PVS search. A modern implementation using the Vulkan graphics API and RTX ray tracing is discussed. Furthermore, optimizations are shown that allow for an implementation that is over 20 times faster than a naive implementation.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Visibility is a fundamental problem, not just in computer graphics but also in robotics, computer vision, and architecture, to name a few. There are various approaches for applications that rely on visibility calculations. Examples in computer graphics include global illumination simulations, which are based on the mutual visibility of pairs of points in the scene to calculate lighting, and shadow generation, where the visibility of a point to a light source has to be calculated. In real-time rendering applications, visibility from a viewpoint is commonly determined using the z-buffer. Visibility may also be solved using ray casting, where rays are shot from a viewpoint through each pixel, finding the closest primitive along each ray.

Another common application for visibility algorithms is occlusion culling. Occlusion culling is an acceleration technique where parts of the scene–that are hidden by other parts in the scene–are culled. This way, hidden geometry does not have to go through the graphics pipeline. In general, occlusion-culling techniques can be categorized into methods that calculate visibility from a point or from a region [BW03]. From-region visibility is more expensive to compute since it is necessary to determine a point's visibility from any position of a given region. Therefore, from-region visibility is often precomputed. Some approaches utilize well-known real-time rendering techniques, such as the hierarchical z-buffer [GKM93]. At the same time, other methods use techniques such as ray casting to determine the visibility of objects and primitives, such as the aggressive from-region sampling-based visibility approach *Guided Visibility Sampling* (GVS) by Wonka et al. [WWZ+06].

Due to the recent developments in hardware-accelerated ray tracing, it is worth revisiting ray casting-based approaches such as GVS. In late 2018, Nvidia introduced the first GPU architecture, Turing [NVI18], that supports hardware-accelerated ray tracing. Two

years later, Nvidia released their second generation of GPUs [NVI20] that support real-time ray tracing, that also allows compute workloads, such as hardware-accelerated ray tracing, as well as graphics workloads, to be executed concurrently. Both architectures employ dedicated hardware units to accelerate ray-tracing tasks such as bounding volume hierarchy traversal and ray/triangle intersection testing. Recent graphics APIs, such as Vulkan and DirectX, offer ray-tracing APIs that allow developers to utilize hardware-accelerated ray tracing.

In this work, we present an aggressive from-region visibility algorithm called *Guided Visibility Sampling++* (GVS++) that uses ray casting and intelligent sampling schemes for visibility determination and works on general 3D scenes. Our algorithm builds upon the work of Wonka et al. [WWZ+06]. We provide a publicly available Vulkan implementation that uses Vulkan's ray tracing API. We analyze the efficiency of the algorithm on various test scenes and provide a comparison to similar approaches. Our contributions can be summarized as follows:

1.2 Aim of the Work

This work aims to improve the original GVS algorithm and offers an analysis of the performance that can be achieved when hardware-accelerated instead of software emulation-based ray tracing is used. Furthermore, an implementation of the improved algorithm using the Vulkan graphics API is provided. The efficiency of the algorithm on various test scenes is analyzed, and a comparison to the original GVS implementation, as well as a rasterization-based from-region visibility approach, is provided.

The contributions of this thesis can be summarized as follows:

- Guided Visibility Sampling++, an aggressive from-region sampling-based visibility approach based on *Guided Visibility Sampling* (GVS) [WWZ+06]. GVS++ is more accurate and offers more flexibility than GVS. Low error rates are achieved by intelligent sampling schemes that find new triangles and parallelize well.
- A publicly available Vulkan implementation of our algorithm that uses hardware-accelerated ray tracing. Our algorithm, using RTX ray tracing, is over four orders of magnitude faster that the original CPU-based GPU implementation.
- An in-depth analysis of GVS++ on multiple scenes and a comparison to a brute-force random sampling approach, a rasterization-based from-region visibility technique, and the GVS algorithm by Wonka et al. [WWZ+06].

1.3 Structure of the Thesis

This thesis is structured as follows. In Chapter 2, fundamental knowledge that this thesis is based on is revised. This includes the concept of visibility in the field of computer

graphics, and methods such as occlusion culling. Furthermore, the basics of ray tracing with a focus on acceleration structures and ray tracing with the Vulkan graphics API is provided. Later chapters require basic knowledge of the Vulkan API. Therefore, a brief introduction to important concepts of the graphics API is given. Chapter 3 reviews related work of visibility algorithms in chronological order based on the taxonomy presented in Chapter 2. Furthermore, an overview of recent advances of hardware-accelerated ray-tracing architectures is given.

Chapter 4 provides information about *Guided Visibility Sampling* (GVS), the from-region visibility technique that this thesis builds upon. In Chapter 5, modifications of the original GVS technique are explained and presented as the *Guided Visibility Sampling++* (GVS++) algorithm.

In Chapter 6, a modern implementation of GVS++ using the Vulkan graphics API and RTX ray tracing is presented. The source code is freely available on GitHub[1]. Important design decisions are discussed regarding the Vulkan API. In addition to the available source code, this chapter provides pseudo-code of the main algorithms.

Results are presented in Chapter 7, where GVS++ is compared to the original GVS algorithm, brute-force random sampling and a rasterization-based visibility algorithm on various scenes. The asymptotic behavior of GVS++ as well as the impact of different parameter choices are discussed. Chapter 8 shows how GVS++ could be used in a practical use case.

The thesis is concluded in Chapter 9, where the main contributions are summarized and possibilities for future work are provided.

[1] `https://github.com/einthomas/GVSPP`, last accessed 30. November 2020

CHAPTER 2

Background

In this chapter, background information about core concepts used in the remainder of this thesis is provided. Section 2.1 provides a general overview of the visibility problem, and discusses visibility culling and different visibility techniques. Section 2.2 gives a brief introduction to ray tracing with a focus on acceleration data structures. In Section 2.3, a quick Vulkan primer is given, providing essential background information and concepts of the graphics API, including Vulkan's ray tracing API.

2.1 Visibility

Visibility is a fundamental part of computer graphics. Given a scene and a viewpoint, the solution to the visibility problem is the set of primitives that are visible from a given viewpoint. There are different techniques to solve the visibility problem. In real-time computer graphics, commonly, a z-buffer is used to solve visibility, where each pixel contains the depth of the currently closest primitive [AMHH19]. Another technique to solve the visibility problem is ray casting: Through each pixel, a ray is shot and intersected with the scene. This way, along each ray, the closest primitive is found.

2.1.1 Visibility Culling

Visibility culling is an acceleration technique that aims to discard invisible parts of a scene. The goal is to avoid rendering geometry that does not contribute to the final image. Visibility culling usually happens before or in the early stages of the graphics pipeline to prevent executing expensive stages, such as fragment processing, for invisible geometry. There are three main visibility culling techniques (see Figure 2.1): View-frustum culling, back-face culling, and occlusion culling [AMHH19]. This thesis focuses on the latter.

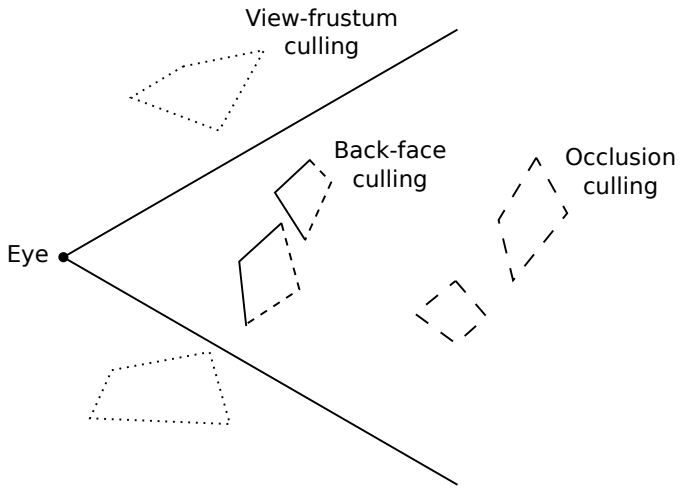


Figure 2.1: Three different visibility culling techniques (view-frustum, back-face and occlusion culling). Culled geometry is dashed. Adapted from Cohen-Or et al. [COCSD03].

View-frustum culling Before the rasterization stage in the graphics pipeline, triangles are tested against the view-frustum and are discarded if they are outside of the volume. *Clipping* handles triangles that intersect the view-frustum.

Back-face culling Back-face culling happens before the rasterization stage and eliminates triangles that face away from the viewer. A triangle is considered to be front- or back-facing depending on the order of the vertices of a triangle after it is projected onto the screen.

Occlusion culling Occlusion culling aims to eliminate geometry occluded by other geometry in the scene. Occlusion culling is typically more computationally expensive than view-frustum and back-face culling since it is necessary to calculate how different parts of a scene affect each other. Occlusion culling is especially important for heavily populated scenes where only a fraction of triangles are visible from a given viewpoint. This typically includes indoor as well as outdoor city scenes.

Occlusion culling algorithms determine a set of primitives that are visible, called the *potentially visible set* (PVS). Depending on the type of PVS that is calculated, occlusion culling algorithms can be classified as *aggressive*, *conservative*, *approximative* or *exact* [NBG02] (see Table 2.1). Aggressive algorithms can underestimate the visibility, resulting in a PVS missing geometry. This can lead to holes in the geometry in the final image. Conservative culling algorithms can overestimate the visibility, resulting in a PVS containing geometry that is invisible. When rendering a scene, no image errors are visible. However, rendering performance is suboptimal compared to a PVS computed by an aggressive algorithm due to the inclusion of triangles that are invisible. Approximative visibility algorithms can include triangles that are invisible but may also

leave holes. The extent of the image error produced by aggressive, conservative, and approximative techniques largely depends on the type of scene. Exact methods neither over- nor underestimate the visibility, resulting in a PVS without errors and optimal rendering performance.

	Runtime Rendering Performance	
Image Errors	**Optimal**	**Suboptimal**
No	Exact	Conservative
Potentially	Aggressive	Approximative

Table 2.1: The four classifications of culling algorithms. Recreated from Nirenstein et al. [NBG02].

Occlusion culling algorithms can further be distinguished by whether visibility is calculated from a point or from a region [BW03, COCSD03]. From-region visibility algorithms compute a PVS that is valid from any point of a given region. Such a region is also called *view cell* and can be flat (2D) or a volume (3D). From-region visibility methods are more computationally expensive than from-point visibility techniques since it is necessary to determine whether the geometry of the scene is visible from any point of the view cell. Therefore, from-region visibility solutions tend to compute the PVS in a pre-processing step, typically by subdividing the scene into multiple view cells and loading the stored PVS depending on the location of the viewer during runtime. Also, due to the pre-computation of the PVS, from-region visibility methods do not handle dynamic scenes as well as from-point visibility algorithms. Alternatively, fast from-region visibility algorithms can be executed on servers that stream the resulting PVS to clients. Depending on the application, the PVS may be valid for multiple frames.

2.2 Ray Tracing

Ray tracing is a technique to realistically simulate the propagation of light. The ray-tracing algorithm involves following paths that represent rays of light originating at the viewer's location through a scene. In the early days of computer graphics [Whi80], it has been shown that a number of natural phenomena, such as reflection and refraction of light as well as sound propagation, can be simulated realistically. The core of the ray-tracing algorithm is the interaction of rays with objects in a scene. Whitted et al. [Whi80] note that up to 95% of the rendering time is spent calculating the intersection of rays and triangles. That is, when a brute-force approach is used, where each ray is tested against each object in a scene. In order to accelerate the process of finding ray/triangle intersections, *acceleration data structures* have been developed. There are different classes [Gla89] of acceleration data structures that aim to accelerate the ray casting process in different ways. Acceleration data structures may aim to reduce the number of intersections that are computed, reduce the number of rays that are traced, or

replace rays as a whole with more general concepts. In the following section, acceleration structures of the first category are discussed.

2.2.1 Acceleration Data Structures

The major acceleration data structures to reduce the number of intersection tests are *bounding volume hierarchies* (BVHs) and *binary space partitioning* (BSP) trees [PJH16, DNL+17]. The commonality of both techniques is that a tree is recursively built that stores primitives in the leaves. BVHs group primitives, whereas BSPs split the space into subspaces. When casting a ray, the tree is traversed, resulting in intersection tests with primitives that are along the path of the ray. An example for a BSP tree is a k-d tree.

Bounding volume hierarchies The BVH is based on the idea that bounding boxes of individual objects or primitives can be enclosed into larger bounding boxes, thereby forming a hierarchy. Then, if a ray does not intersect a bounding box, it is guaranteed that the ray does not intersect any of its inner bounding boxes. A BVH is structured as a tree, where primitives are stored in the leaves, and bounding boxes that enclose all primitives of the respective subtree are stored in the nodes (see Figure 2.2). The bounding boxes of nodes on the same level in the tree may overlap. The advantage of BVHs over BSP trees is that they can be constructed more rapidly [DNL+17]. A typical

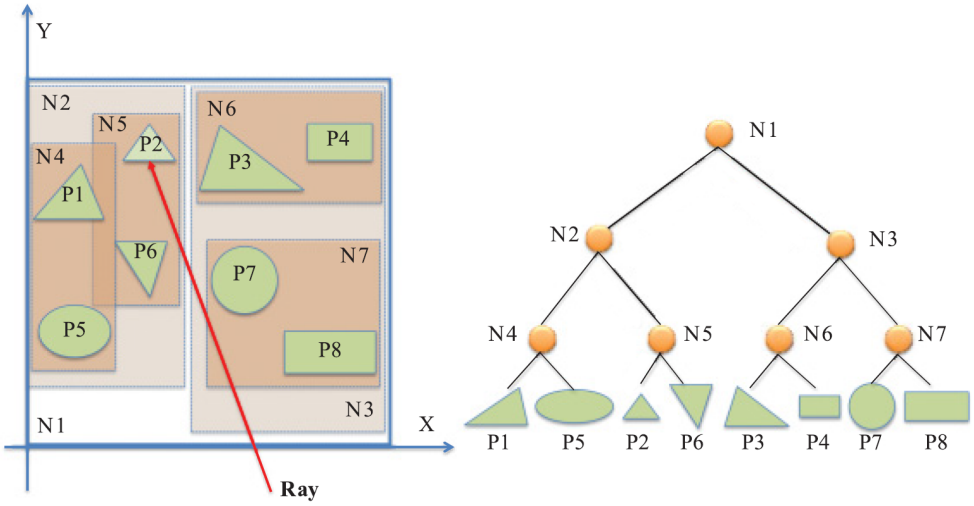


Figure 2.2: An example 2D scene (left) and its respective BVH (right). Objects (P) are grouped into hierarchies of bounding boxes (N). Figure adapted from Deng et al. [DNL+17].

traversal algorithm [Gla89] starts at the root bounding box, which encloses the whole scene. The tree is descended level by level, testing the ray against the bounding boxes stored in the nodes. If there is an intersection, the respective subtree is traversed. If a bounding box is missed, traversal of the respective subtree is skipped.

Binary space partitioning trees BSP trees recursively partition space using splitting planes. One type of BSP tree is the k-d tree, where axis-aligned splitting planes are used to subdivide space into smaller subspaces (see Figure 2.3). The subdivision process continues until the number of primitives within a subspace is below a certain threshold. This has the effect that primitives may only partially be contained within a subspace, intersecting a splitting plane. Similar to the BVH, a k-d tree is organized as a binary

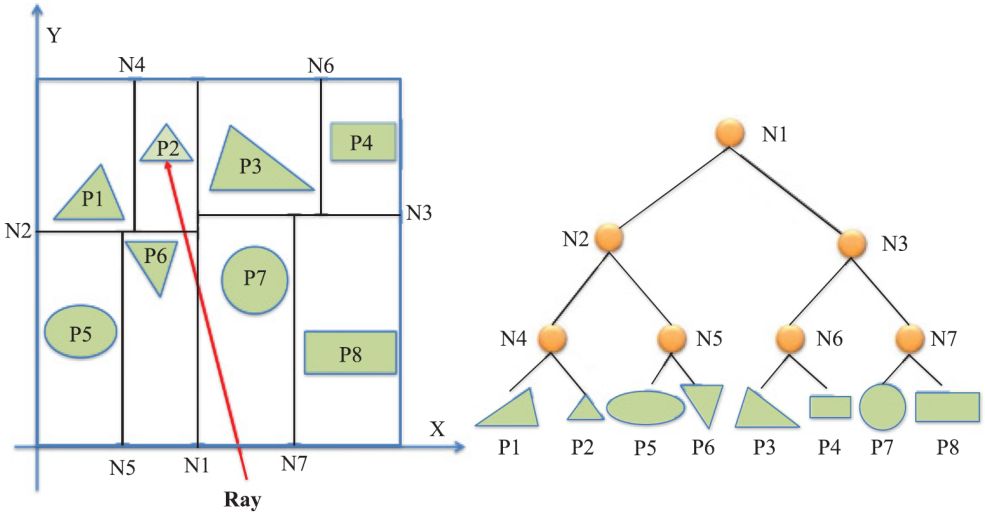


Figure 2.3: An example 2D scene (left) and its respective BSP k-d tree (right). Space is subdivided by axis-aligned splitting planes (N) as long as each subspace contains more than one object. Figure adapted from Deng et al. [DNL+17].

tree, where each node splits the space into two subspaces. Primitives are stored in the leaves, depending on the subspaces they are in. Primitives that intersect splitting planes are stored in multiple leaves. In contrast, a primitive in a BVH is stored in at most one leave. A similar type of BSP tree is the octree, where each node divides space into eight octants.

2.2.2 Sampling

For different ray casting tasks samples have to be generated. Examples include anti-aliasing, where multiple rays are traced through a single or when distributing ray origins on the surface of an object. Often pseudo-random sampling strategies that distribute samples uniformly are preferred. Several techniques [Shi91] have been developed to generate uniformly distributed samples: Jittered, Poison disk and N-rooks sampling, to name a few. Another sampling strategy uses the Halton sequence [Nie92] for generating uniformly distributed samples. One important property of this technique is that it is hierarchical [WLH97], meaning that for n generated samples, the first k samples are the same for all $n \geq k$. This allows generating samples incrementally. The generation of the Halton sequence is based on the following principles [Nie92, WLH97]: Any integer $n \geq 0$

with terms $a_j \in \{0, 1, \ldots, b-1\}$ in a base $b \geq 2$ can be expressed as:

$$n = \ldots a_2 a_1 a_0 = \sum_{j=0}^{\infty} a_j \, b^j \tag{2.1}$$

The radical-inverse function $\phi_b(n)$ reflects the terms a_j of the integer n around the decimal point and is defined by:

$$\phi_b(n) = 0.a_0 a_1 a_2 \cdots = \sum_{j=0}^{\infty} \frac{a_j}{b^{j+1}} \tag{2.2}$$

To generate s-dimensional Halton points, s different bases b are chosen. Typically, the bases are chosen to be coprime [FKP15]. The n-th s-dimensional Halton point x_n is then given by:

$$x_n = (\phi_{b_1}(n), \ldots, \phi_{b_s}(n)) \tag{2.3}$$

Multiple Halton points x_n form the Halton sequence.

2.3 The Vulkan API

Vulkan is a modern low-level graphics API that aims to keep the driver overhead to a minimum. The consequence is that, by default, the error checking is very limited. Compared to more high-level graphics APIs such as OpenGL, Vulkan requires the developer to do much of the resource management, including allocating and freeing device and host memory, and offers a wealth of parameters to be set, which also lead to a more verbose API. Due to the low-level approach, Vulkan provides more control over the graphics hardware, potentially allowing it to achieve better hardware utilization.

2.3.1 Instances, Devices, and Queues

Vulkan is initialized by creating a `VkInstance` handle, which stores application-specific state. In contrast, in OpenGL, the state is modified and stored globally. When creating a Vulkan instance, it can be specified which extensions should be used. Extensions are used to add additional functionality, such as ray tracing with `VK_NV_ray_tracing` or `VK_KHR_ray_tracing`, which may be provided by a Vulkan implementation. As initially mentioned, Vulkan does not provide extensive error checking by default. Vulkan uses *layers*, which allows intercepting Vulkan function calls to add additional functionality. Error checking capabilities are added via *validation layer*s, such as `VK_LAYER_KHRONOS_validation`. This allows to easily remove any debug error checking implemented as a layer from the production build. Layers are specified during the instance creation.

GPUs are exposed as *physical device*s (`VkPhysicalDevice`), which can be queried and selected according to their properties and supported features. Multiple physical devices can be used in an asynchronous manner. Physical devices expose different *queues* where

work can be submitted to, as discussed in the next section. Each queue is of a *queue family* that defines which type of work a queue is suited for. Queues can be capable of a combination of graphics, compute, data transfer or sparse memory management operations. Work submitted to different queues may be executed in parallel. Therefore, it can be beneficial in terms of performance to use a transfer-only queue, if available, for host/device transfer operations. This is because some GPUs have special hardware that maximizes transfer rates for such queues[1]. Table 2.2 shows the queue families available on an Nvidia RTX 2080.

Queues	**Flags**

16	`GRAPHICS_BIT`
	`COMPUTE_BIT`
	`TRANSFER_BIT`
	`SPARSE_BINDING_BIT`

8	`COMPUTE_BIT`
	`TRANSFER_BIT`
	`SPARSE_BINDING_BIT`

2	`TRANSFER_BIT` `SPARSE_BINDING_BIT`

Table 2.2: The queue families of an Nvidia RTX 2080.

A *logical device* (`VkDevice`) is the interface to communicate with a physical device. When creating a logical device handle, device features such as geometry shaders, as well as the queues that will be used, have to be specified.

2.3.2 Command Buffers

*Command buffer*s (`VkCommandBuffer`) are used to record commands, such as drawing or transfer commands. Suitable functions are prefixed with `vkCmd`. Command buffers are executed by submitting them to a suitable queue via `vkQueueSubmit`. The concept of command buffers has a number of advantages: Applications that do not require to rebuild a command buffer repeatedly, e.g., every frame, may only record a command buffer once and submit it multiple times. This way, the cost of rebuilding is avoided. Also, command buffers promote the use of host-side multithreading to maximize CPU utilization, since Vulkan allows populating separate command buffers per thread. Multiple threads can also be used to build a single command buffer by combining multiple *secondary* to one *primary* command buffer. Command buffers allocate memory from *command pool*s (`VkCommandPool`). Since each command pool must only be used by a single thread

[1] `https://gpuopen.com/performance/`, last accessed 21. October 2020

simultaneously, there is no implicit synchronization mechanism built-in, which also benefits performance.

2.3.3 Memory Management

Explicit memory management is a large part of Vulkan, especially when compared to higher-level APIs such as OpenGL. After creating a resource, e.g., a buffer or a texture, a suitable memory type has to be queried. Once a memory type is found, a block of memory can be allocated, which is then bound to the initially created resource. Depending on the platform and hardware, different memory types and heaps are available. The memory type and heap has to be chosen such that it is compatible with the resource and suitable for the intended usage. Table 2.3 shows the different memory types and heaps available on an Nvidia RTX 2080.

Memory Type	**Memory Heap**
`DEVICE_LOCAL_BIT`	0
`HOST_VISIBLE_BIT` `HOST_COHERENT_BIT`	1

`HOST_VISIBLE_BIT`
`HOST_COHERENT_BIT`
`HOST_CACHED_BIT`
	1

`DEVICE_LOCAL_BIT`
`HOST_VISIBLE_BIT`
`HOST_COHERENT_BIT`
	2

Table 2.3: Memory types and heaps available on a Nvidia RTX 2080 on Windows 10.

Memory heap 0 represents memory that is residing on the GPU, therefore allowing for fast access from the GPU. Heap 1 corresponds to memory on the host side, which is accessible to the CPU, and memory heap 2 represents memory on the GPU that can also be mapped into memory on the host side but is limited to 256MB. Memory with the `HOST_COHERENT_BIT` set does not require explicit flushing for host writes to memory.

When a memory type is used, such as memory allocated from heap 0, that cannot be mapped to the host, an intermediate *staging* buffer is required to transfer data between device and host. A staging buffer is usually allocated from host memory. Data that should be transferred to device local memory is first copied into the staging buffer. Via the copy command `vkCmdCopyBuffer`, the content of the staging buffer is then copied to the target device local memory. For such an operation, a transfer-only queue should be preferred (Section 2.3.1).

2.3.4 Pipelines and Shaders

A pipeline (`VkPipeline`) in Vulkan is a monolithic object describing all shader and fixed-function stages. Many parameters cannot be changed dynamically, which either requires to rebuild the pipeline or create and store multiple pipelines beforehand. Vulkan offers three types of pipelines: A graphics pipeline, a compute pipeline, and a ray tracing pipeline. A graphics pipeline contains shader stages such as a vertex and a fragment shader as well as a description of the render pass, which describes the render targets. A compute pipeline contains a single shader stage–a compute shader. A ray tracing pipeline contains multiple shader stages such as programmable ray generation, hit and miss stages, and a fixed-function traversal stage.

Each pipeline has a pipeline layout (`VkPipelineLayout`) that contains a description of the resources shaders can access. The binding of a specific resource is described using a *descriptor*. Descriptors are grouped to form a *descriptor set* (`VkDescriptorSet`). Each descriptor set is described by a *descriptor set layout* (`VkDescriptorSetLayout`), which contains the binding location of the resource in the shader and the type of resource that is bound. When creating a pipeline, a descriptor set layout has to be specified. The descriptor set layout acts as a template for descriptor sets that can then be bound before binding the pipeline.

During the pipeline creation, *shader module*s have to be specified that contain shader code in the SPIR-V format. SPIR-V is an intermediate binary format. The advantage of using a binary format is that any shading language for which a compiler to SPIR-V exists can be used to write shaders for Vulkan. This includes GLSL and HLSL. GLSL has been extended to support Vulkan-specific features, such as the ability to specify a descriptor set in the layout specifier as well as the concept of *push constants*. The latter provides a fast way to transfer small amounts of uniform data to the GPU.

2.3.5 Synchronization

Due to the parallel nature of the GPU itself but also the parallel execution of CPU and GPU, it is important to ensure correct synchronization to avoid working with partial or unfinished results. Most functions in Vulkan must be *externally* synchronized, meaning that the application has to ensure correct synchronization between multiple threads or devices accessing the same resource simultaneously. Synchronization in Vulkan is an extensive topic, with Vulkan providing various different mechanisms.

Fences can be used to notify the host once a task on a device is finished. This is suitable for scenarios where generating work for the device depends on the result of the task the device is currently executing. This is also called *execution dependency*. The `vkQueueWaitIdle` command represents a coarse-grained special case of a fence. The command blocks the current thread until the specified queue finishes execution. Waiting for a queue to be idle until a thread can continue may be too coarse for most application scenarios since the goal should be to constantly submit work to the queue such that the device is always busy and best utilized.

Semaphores can have two states, signaled and unsignalled, and can be used to synchronize commands or access to resources between queues. A thread waits until the semaphore becomes signaled to continue execution.

Pipeline barriers provide a means to specify execution dependencies between individual pipeline stages. When creating a pipeline barrier, a source and a target stage have to be provided. The target stages wait for the source stage to be finished before being executed. Each shader stage "in between" is not blocked by the execution of the source stage.

2.3.6 Ray Tracing

Ray tracing has first been made available through the `VK_NVX_raytracing` extension developed by Nvidia. Later, the cross-vendor ray tracing extension `VK_KHR_ray_tracing` has been added and offers similar functionality. The Vulkan API is extended by new shaders as well as functionality to build and manage acceleration structures. In the scope of this thesis, Nvidia's extension is used since the latter was not yet released when this thesis was started. In the remainder of this document, "the ray-tracing extension" refers to `VK_NVX_raytracing`.

Acceleration structures Acceleration structures (AS) are organized into a two-level hierarchy (see Figure 2.4). The bottom-level AS contains the geometry, while the top-level AS contains references to bottom-level AS nodes together with transform and shading information. When building an AS, it can be specified if updates are allowed and whether faster build or faster ray tracing performance should be preferred. This two-level hierarchy has multiple advantages. If a scene contains multiple instances of some object, the object's geometry is only stored once in the bottom-level AS, and multiple references are stored in the top-level AS. This also allows faster rebuilds, because if transformation or shading information or an object's geometry changes, only the respective acceleration structure has to be rebuilt.

Ray-tracing pipeline Vulkan's ray tracing pipeline (see Figure 2.4) consists of five different shader stages: The ray generation, intersection, any hit, closest hit, and miss shader stage.

- *RayGen* shaders are programmable shaders and are the starting point of the ray tracing pipeline on the device side. Rays are generated, and the `traceNV` command is used to start the acceleration structure traversal.
- *Intersection* shaders compute ray/primitive intersections. Ray/triangle intersection shaders are built-in. Additional intersection shaders can be added for custom ray/primitive intersection logic.
- *Any hit* shaders are invoked whenever a ray/primitive intersection is registered.

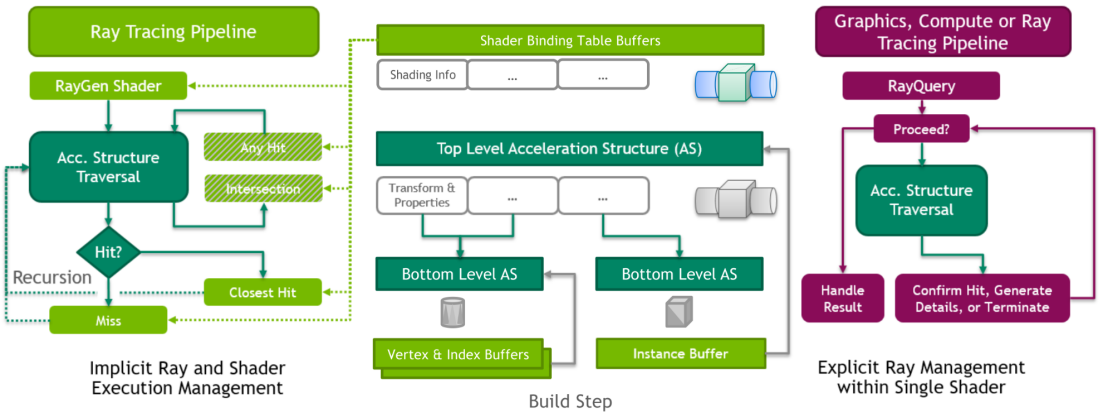


Figure 2.4: Vulkan's ray tracing API (`VK_KHR_ray_tracing`). The ray-tracing pipeline (left) consists of different shader stages that are invoked based on the traversal of the acceleration structure. Using the shader binding table (middle), geometry-specific shaders can be specified. Ray queries (right) offer an alternative to the ray-tracing pipeline and allow casting rays from any type of shader. Figure ©Khronos Group, Inc.

- *Closest hit* shaders are invoked for the closest ray/primitive intersection along a ray.
- *Miss* shaders are invoked if there is no ray/primitive intersection.

Shaders are specified via the *shader binding table* (SBT). Multiple shader stages of the same type can be specified such that different shaders are invoked depending on the intersected geometry, e.g., to be able to handle different materials.

An application-defined ray payload struct can be used to carry information along with the ray (see Figure 2.5). It is populated in the ray generation shader and can be read and modified in any and closest hit shaders. An example use case is to return the ID of the intersected primitive and the intersection position to a RayGen shader. Intersection shaders use a hit attribute vector to pass data to any hit and closest hit shaders (see Figure 2.5). The built-in ray/triangle intersection shader stores barycentric coordinates of the registered ray/triangle intersection in the hit attribute. Barycentric coordinates may then be used for shading calculations in hit shaders.

Ray queries Ray queries (see Figure 2.4) have been introduced with the cross-vendor ray tracing extension `VK_KHR_ray_tracing`. Ray queries are an alternative to the ray tracing pipeline and allow casting rays from any type of shader. The acceleration structure is traversed iteratively. A set of functions is available to access intersection data such as the hit position and the ID of the intersected primitive. This allows making decisions based on the geometry that is intersected. One use case is to use ray queries in existing fragment shaders to add effects such as ray traced shadows. Since no shader binding table is used, any overhead caused by the binding table is avoided.

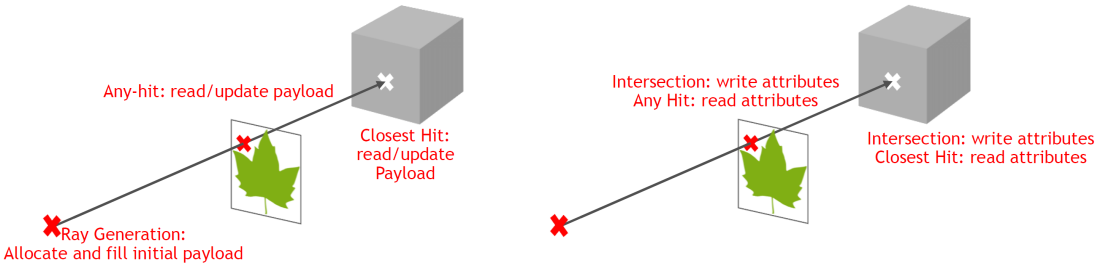


Figure 2.5: Inter-shader communication. Ray payloads (left) and hit attributes (right) are used to transfer data between shader stages. Figure adapted from Nvidia [SW18].

Watertightness One important property of ray/triangle intersection algorithms is that of *watertightness*. Woop et al. [WBW13] presented a novel ray/triangle intersection test that is watertight at edges and vertices. This means that it is guaranteed that a ray does not pass through shared edges or vertices. This especially concerns large scenes, where triangles are small and far away. In such a case, a non-watertight intersection test could pass through triangles due to numerical instabilities. In the algorithm by Woop et al., the ray/triangle intersection problem is simplified using affine transformations. The transformed problem is then solved in 2D. For problematic cases, double-precision floating-point arithmetic is used to avoid precision issues. Intersection testing is watertight in Vulkan with Nvidia's hardware-accelerated ray tracing implementation.

CHAPTER 3

Related Work

This chapter covers work related to the topics of this thesis. In Section 3.1, from-region visibility algorithms are reviewed. Conservative, aggressive, approximative, and exact techniques are discussed in chronological order within each section. Methods operating in image- as well as object-space are covered. In Section 3.3, recent research efforts of dedicated ray-tracing hardware architectures are reviewed, starting with one of the first works in this area up to current consumer-level hardware.

3.1 From-Region Occlusion Culling Techniques

3.1.1 Conservative Techniques

Durand et al. [DDTP00] presented a conservative visibility technique for general scenes. The authors introduced an *extended projection* operator to calculate the PVS. The idea is based on point-based visibility approaches. The visibility of an object can be determined by testing whether its projection onto the image plane lies within an occluder's projection. The authors extended this idea for volumetric view cells. Occludees and occluders are projected onto a plane. Let A be the union of an ocludee's projection onto the plane and B the intersection of an occluder's projection onto the plane. An occludee is regarded as hidden if it is behind an occluder, and A is contained in B.

The conservative visibility algorithm by Schaufler et al. [SDDS00] classifies regions in space as occluded by *fusing* occluders and also extending occluders into empty space. The algorithm operates on a discretized version of the scene. Therefore, an octree is used to subdivide the scene into voxels. A voxel is classified as empty, opaque, or a boundary that separates empty and opaque regions. Determining the PVS of a given view cell starts by selecting an opaque voxel, i.e., an occluder. The occluder is then extended along the coordinate axis as long as there are adjacent opaque voxels. Since an observer within a view cell cannot distinguish between empty and opaque for hidden voxels, an occluder

is also extended into hidden space. Figure 3.1 shows a scene and its extended occluders. The PVS is then calculated by testing whether the bounding box of the scene's objects intersects occluder voxels. During an interactive application, this approach also allows checking whether moving objects or objects that were not part of the original scene are visible.

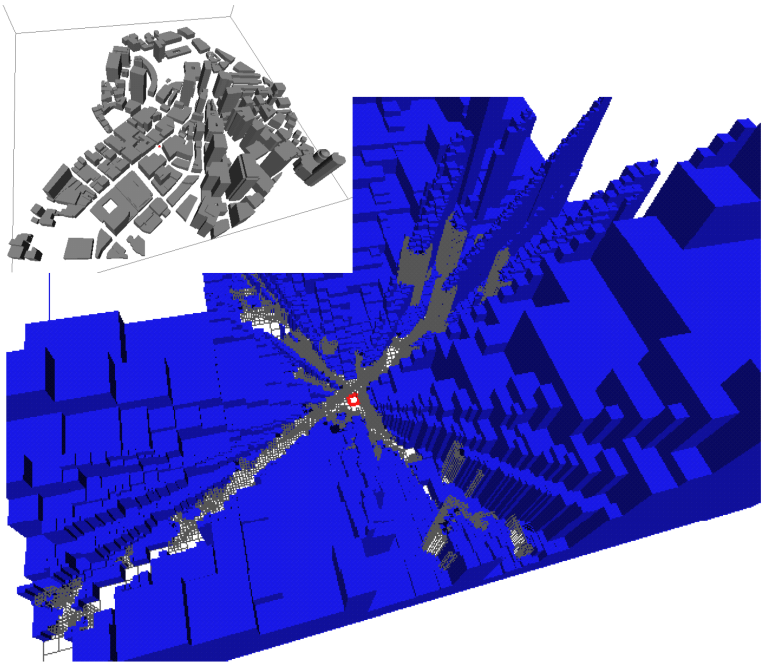


Figure 3.1: A city scene with a view cell placed between buildings. Blue boxes represent occluded voxels. Reprinted from Deng et al. [SDDS00].

Wonka et al. [WWS00] compute the visibility for a view cell using point samples. The approach is based on the idea that visibility can be computed by computing the joint umbra of all samples points on the view cell. To avoid missing triangles that are only visible from locations between sample points occluders are shrunk by ϵ. A triangle occluded by a shrunk occluder is also occluded by the original occluder within an ϵ-neighborhood of the sample point. A triangle is considered occluded if it lies in the intersection of the umbrae from all the sample points on the view cell.

Leyvand et al. [LSCO03] note that from-region visibility is inherently 4D. This is because a ray effectively leaves and enters the view cell and a target region through 2D surfaces. Therefore, the authors presented a factorization of the 4D visibility problem into 2D vertical and horizontal components. The horizontal component gives the horizontal direction of a ray. The component is defined by two concentric squares at the view cell's position, whereas one square is the top-down projection of the view cell onto the ground. The horizontal direction of a ray is given by two intersections of the concentric squares (see Figure 3.2). Each horizontal direction defines a vertical plane that gives the

vertical direction of the ray. For each object intersected by a vertical plane, its umbra is calculated and merged with the other umbrae associated with the current vertical plane. This is comparable to the occlusion fusion in the approach by Schaufler et al. [SDDS00]. The merged umbrae are then stored in an *occlusion map*.

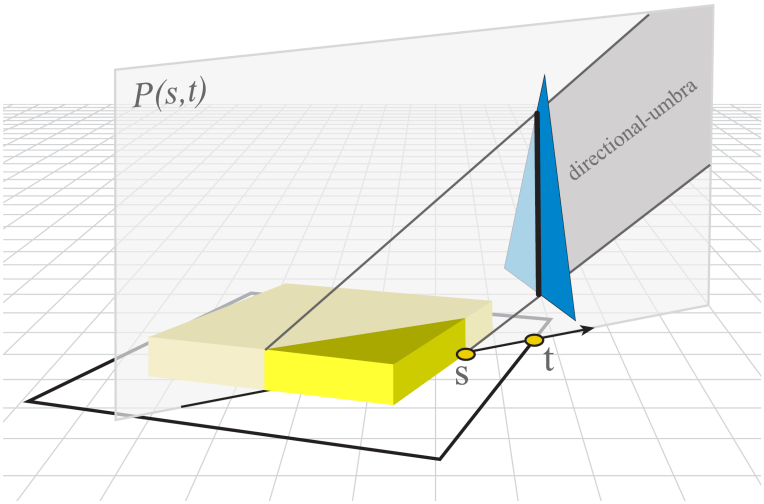


Figure 3.2: Ray space factorization. Intersections (s, t) with the concentric squares give a horizontal ray direction and define a vertical plane $P(s, t)$, which gives a horizontal ray direction. Reprinted from Leyvand et al. [LSCO03].

In a recent approach, Hladky et al. [HSS19] describe a conservative visibility algorithm that operates in image-space. The authors introduce the *camera offset space* to calculate under which camera offsets within a view cell a stored triangle is visible. To calculate the PVS, first, a fragment list of triangles that a ray through a fragment's center would intersect is stored. To also find triangles that cover a fragment from a different viewpoint of the view cell, each triangle is enlarged before creating the fragment list. Each triangle in the fragment list is then transformed into camera offset space, where each triangle's visibility is determined.

3.1.2 Aggressive Techniques

Nirenstein et al. [NB04] presented an aggressive visibility algorithm that operates in image-space. In this approach, the PVS of a given view cell is determined by rendering the scene from multiple viewpoints of the view cell. For each pixel, the ID of the rendered primitive is stored in a framebuffer. The buffer is then read to gather stored primitive IDs and to populate the PVS. Since aggressive techniques can underestimate the exact visible set, visibility errors by missed triangles can arise. Therefore, the authors developed error metrics and error minimization heuristics. To avoid under- or oversampling regions, the hemicubes from which the scene is rendered are placed on the view cell adaptively. The hemicube placement resembles a quad-tree structure (see Figure 3.3). An error metric is

used to decide whether further hemicubes should be placed. The simplest error metric calculates the difference between the PVS of four corner hemicubes. Further hemicubes are placed if the difference is above a specified threshold

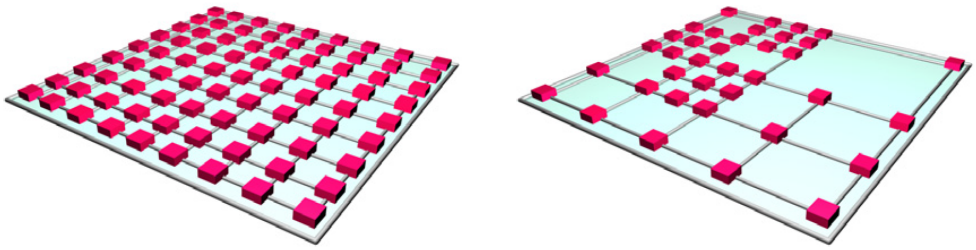


Figure 3.3: Left: Uniform hemicube placement. Right: Adaptive hemicube placement used by Nirenstein et al. Reprinted from Nirenstein et al. [NB04].

Wonka et al. [WWZ+06] developed Guided Visibility Sampling, an aggressive visibility algorithm that uses ray casting and intelligent sampling strategies to determine the PVS of a view cell efficiently. More detail is given in Chapter 4.

Bittner et al. [BMW+09] presented *Adaptive Global Visibility Sampling* (AGVS) [BMW+09], an aggressive solution that also uses ray casting. In this approach, the PVS of multiple view cells are calculated simultaneously. The idea is to let a ray contribute to each view cell it intersects. Whenever a ray intersects an object, the object is inserted into the PVS of each view cell intersected by the ray. The authors developed intelligent sampling strategies to adapt the visibility sampling to the scene. Initially, rays are chosen that are independent of other rays. A mutation-based strategy is then used to mutate previously used rays to sample the scene nearby. Guided Visibility Sampling [WWZ+06] follows a similar idea, where rays are mutated to penetrate unexplored regions. AGVS employs a *visibility filter* to alleviate errors by extending the PVS by objects that are likely to be visible. To this end, the error within a region is taken into account to focus on undersampled parts of the scene. The filter adds objects nearby other objects that are already visible and merges the PVS of adjacent view cells.

In a recent approach, Ho et al. [HCCL12] developed an image-space sampling algorithm that builds upon the work of Nirenstein et al. [NB04]. The authors present an importance sampling scheme in image-space that places hemicubes on the view cell such that new primitives are more likely to be found. Based on the image-space sampling results, the algorithm builds a *reliability function* on the view cell boundary. Samples are placed at points of minimum reliability. The reliability of a point is measured by calculating the size of *visibility portals* seen from this point. A visibility portal is a gap of depth as seen from a point. Its size is approximated by calculating the difference of depth values of adjacent pixels in the rendered image. Figure 3.4 shows a scene and a resulting reliability function.

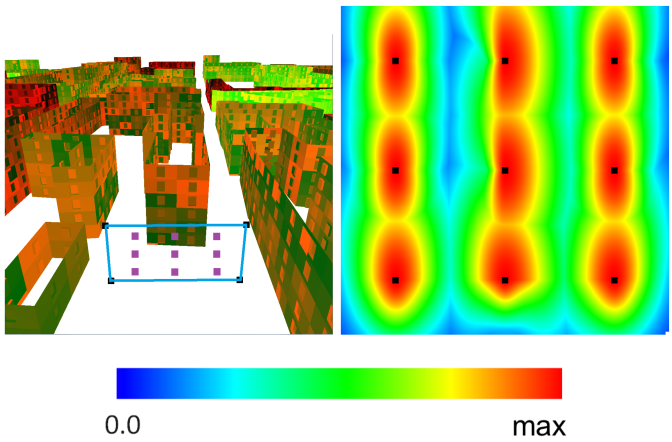


Figure 3.4: A scene with a view cell and the resulting reliability function on the view cell boundary resulting from nine samples (dots). Reprinted from Ho et al. [HCCL12].

3.1.3 Approximative Techniques

Early on, Airey et al. [ARBJ90] proposed a from-region visibility algorithm for indoor architectural scenes. A scene is automatically subdivided into view cells, e.g., such that each room is covered by one view cell. The idea is that the set of visible primitives is mostly the same for most viewpoints within a room but changes more rapidly near *portals*, e.g., doors or windows. A spatial subdivision technique such as a k-d tree or an octree is used to place view cells automatically and to find the view cell containing the current viewpoint during rendering. A portal is represented by a polygon. To find the primitives visible from a portal, only the primitives visible from a portal's polygon have to be computed. The authors find that this problem is equivalent to finding the primitives that receive light from an area light source. This formulation led to two different algorithms to solve the problem. One algorithm uses point sampling to find visible primitives, while another algorithm calculates shadow volumes.

Gotsman et al. [GSF99] presented an approximate visibility algorithm for general scenes. Similar to previously discussed techniques, ray casting is used for visibility computation. In this approach, visible objects instead of individual primitives are stored. Each ray is defined by a 3D position and a 2D direction giving a 5D space of possible rays that are traced. This space is subdivided using a k-d tree, where each leaf represents a view cell. The scene objects are then sampled using rays that originate at uniformly distributed random positions within the view cell. Rays are cast at each object at uniformly distributed random locations. An object is regarded as visible only if it is visible from a non-negligible part of the view cell.

3.1.4 Exact Techniques

Teller and Séquin [TS91] presented a visibility algorithm for indoor architectural scenes that builds upon the ideas of Airey et al. In this approach, visibility between cells is determined in a preprocessing step. First, an adjacency graph is built to store the neighboring cell reachable through portals. Based on the adjacency graph, *portal sequences* are found through which unobstructed sightlines can be constructed. Next, a *stab tree* is built for each view cell, storing the cells that are visible through portal sequences. During the runtime of an interactive application, cells visible from the current viewpoint are determined based on the stab tree of the current cell. Cells visible from a viewpoint are determined by testing whether an unobstructed sightline that contains the viewpoint can be constructed through the portal sequences stored in a given stab tree. Figure 3.5 shows an example.

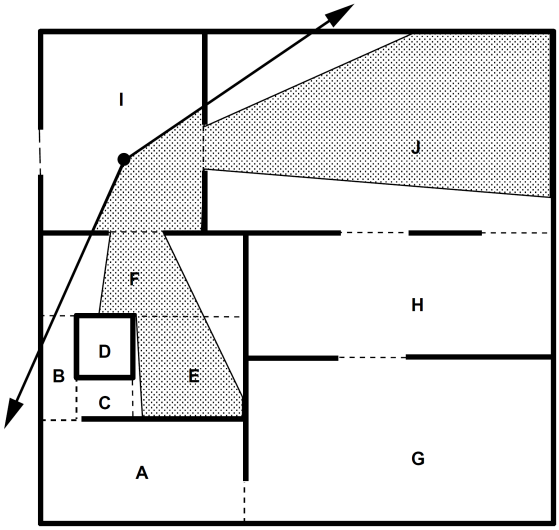


Figure 3.5: An indoor scene showing a viewpoint with its associated view cone. Cells intersected by the view cone are visible from the viewpoint. Reprinted from Teller and Séquin [TS91].

More recently, Bittner et al. [BWW05] presented an exact from-region visibility approach for 2.5D urban scenes. Similar to the approach by Teller and Séquin [TS91], the concept of portals and finding stabbing lines are used to calculate visibility. The authors use *line space*, a dual space in which each line in primal space corresponds to a point. The rays that intersect an occluder form a *blocker polygon* in line space. The idea is that the intersection of blocker polygons form a subdivison of line space. Such a subdivision represents all the rays in primal space that intersect the corresponding occluders in primal space. Visibility is then calculated by constructing 3D portals for each occluder and testing if there is a line originating at the view cell that intersects a portal.

An analytical solution was presented by Bittner [Bit02]. Line space and Plücker coor-

dinates are used to calculate exact from-region visibility for 3D scenes. The idea is to construct an occlusion tree that captures the visibility of a polygon and also represents the lines that are blocked by polygons in the scene. Plücker coordinates are used to describes the lines from the view cell blocked by a polygon in the scene.

3.2 View Cell Placement

The placement and size of view cells themselves can have a significant impact on visibility computation times and memory requirements for storing PVS data. Furthermore, the size of a view cell's PVS impact rendering performance. Therefore, intelligent view cell placement strategies were developed. Mattausch et al. [MBW06] present a view cell partitioning strategy that aims to construct view cells that minimize rendering costs. A three-step algorithm is presented. Rays are cast to estimate visibility in the scene. The resulting information is used to estimate the rendering cost. Based on the estimated rendering cost, the view space is hierarchically subdivided using a BSP tree. The number of view cells is then reduced by a bottom-up merging process. A similar approach is presented by Mattausch et al. [MBWW07]. In this approach, the view space and the object space are partitioned while minimizing rendering and memory cost. Approximate visibility information is acquired by sampling the scene which is then used to guide the subdivision process. View space and object space are subdivided simultaneously by splitting either a view cell or an object.

3.3 Hardware-Accelerated Ray Tracing

Deng et al. [DNL+17] generalized the ray-tracing process into a ray-tracing pipeline (see Figure 3.6). The pipeline consists of an acceleration structure construction stage as well as a traversal stage where rays are generated, the acceleration structure is traversed and intersection tests are executed. Recently, various ray-tracing hardware architectures have been proposed to accelerate parts of the ray-tracing pipeline, mainly the traversal of the acceleration structure and the ray/primitive intersection itself.

Schmittler et al. [SWS02, SWW+04] introduced one of the first dedicated hardware architectures for real-time ray tracing, called SaarCOR. SaarCOR comprises three units: One unit for ray generation and shading, one ray-tracing core, and one unit for memory management. The ray-tracing core traverses the acceleration structure and calculates the ray/triangle intersection. The SaarCOR chip traces bundles of rays, allowing to exploit coherence between rays. This is based on the observation that similar rays likely intersect similar nodes in the acceleration structure, which reduces the number of memory fetches. This approach corresponds to a *single instruction, multiple data* (SIMD) [Fly72] processing pattern.

StreamRay [RGD09] also follows a SIMD processing pattern of the rays. In this approach, the ray-tracing algorithm is reformulated as several stream filtering operations applied to a stream of rays. The architecture consists of a ray engine that generates a stream of rays

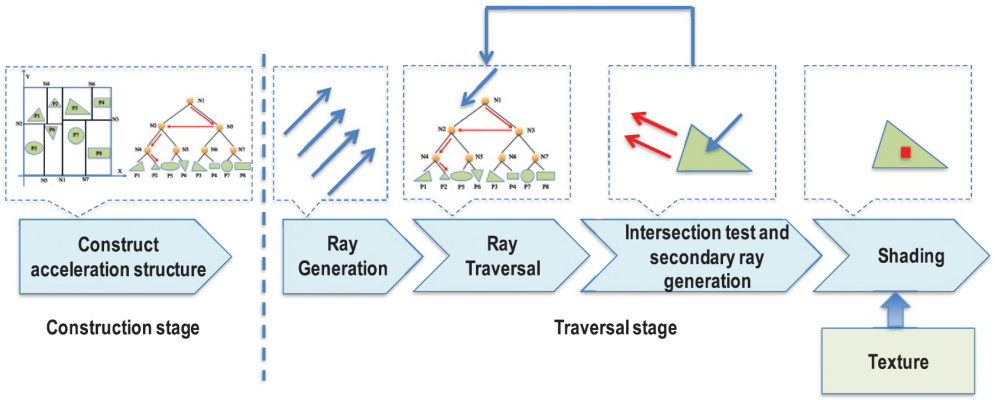


Figure 3.6: A general ray-tracing pipeline showing all main building-blocks of the ray-tracing process. Reprinted from Deng et al. [DNL+17].

and a filter engine that applies programmable filters to a ray stream. Each building-block of the ray-tracing pipeline, such as the traversal and the intersection process, is realized as stream filters. A stream filter partitions the incoming ray stream into coherent subsets of rays. This benefits the SIMD processing pattern by always processing sets of rays that require the same operations.

Woop et al. [WSS05] presented a successor architecture to SaarCOR that is comparable to modern GPUs. A hardware design based on a combination of SIMD and multithreading, known as *single instruction, multiple threads* (SIMT) [MRR12], is used. This architecture increases hardware utilization by better hiding memory latency. Furthermore, this architecture allows for programmable material, geometry, and illumination shaders.

Nah et al. [NPP+11] developed a *multi instruction, multi data* (MIMD) [Fly72] architecture that consists of a fixed traversal and intersection pipeline. Programmable shaders are used for ray generation and shading. In contrast to SIMD architectures, single rays are independently traced, which results in less performance degradation when incoherent rays are traced. The ray/intersection process is split into multiple phases to detect misses early. This reduces unnecessary operations, and memory fetches.

In late 2018, Nvidia introduced the Turing GPU architecture [NVI18], which is the first product to support hardware-accelerated ray tracing aimed at the mass market. Nvidia's second generation [NVI20] of GPUs, Ampere, capable of real-time ray tracing builds upon the Turing architecture, and further increases ray tracing performance. Special ray-tracing cores accelerate BVH traversal and ray/triangle intersection computations. Since both architectures are closed source, only a high-level overview of the architecture is published by Nvidia (see Figure 3.7).

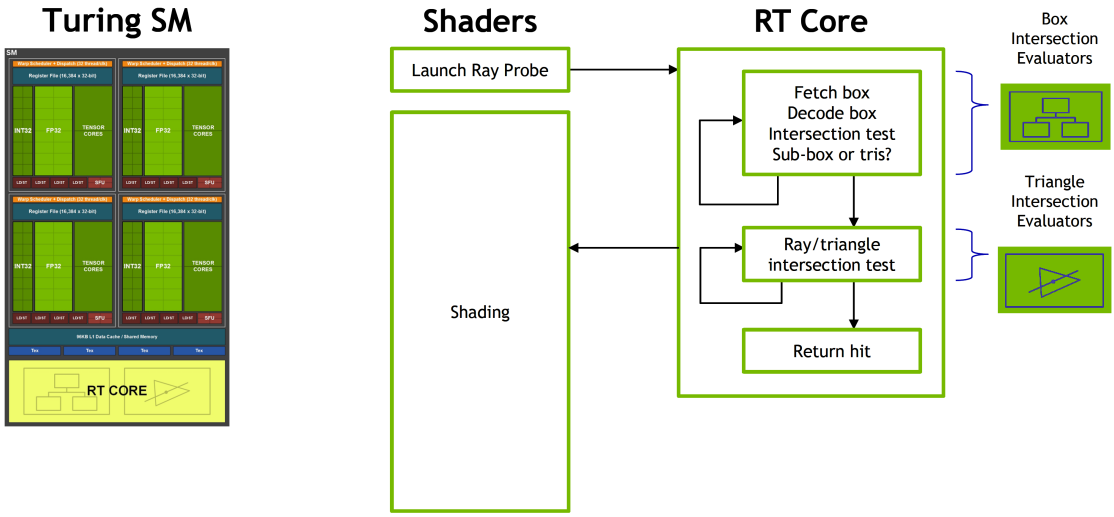


Figure 3.7: The Turing architecture contains multiple streaming multiprocessors, each containing a ray-tracing core. The ray-tracing cores are responsible for traversing the BVH and calculating intersections. This design offloads the SM such that it can be used for other workloads such as shading calculations. Figure adapted from Nvidia [NVI18].

CHAPTER 4

Guided Visibility Sampling

In this chapter, the from-region visibility technique *Guided Visibility Sampling* (GVS) is explained. The main contribution of this thesis, an improved GVS algorithm, is based on the ideas that are discussed in this chapter.

4.1 Introduction

Wonka et al. [WWZ+06] presented an aggressive visibility solution called *Guided Visibility Sampling* (GVS). The algorithm uses ray casting to precompute a set of triangles that are visible from a rectangular region in space. GVS does not rely on additional information about the scene and does not need additional memory, apart from the storage requirements of the PVS itself. The general idea of GVS is to rapidly find triangles, which are then used as seed points in a subsequent exploration phase that grows the intermediate PVS. This can alternatively be seen as a flood-fill strategy where, starting from a seed point, a region is expanded.

The main application scenarios of GVS are real-time rendering applications and games. GVS is well suited for such applications since it does not depend on scene-specific properties, and the resulting PVS is independent of the output resolution. The PVS is precalculated and is therefore intended to be used for static geometry. Further application scenarios include online and networked visibility, where the GVS algorithm is executed on a server, and the resulting PVS is streamed to a client.

4.2 Algorithm Overview

The Guided Visibility Sampling algorithm consists of two main parts: Initial random sampling and a subsequent exploration process. An initial seed point is placed via random sampling. If a new triangle has been found, the exploration process is started. The

subsequent exploration process efficiently samples the neighborhood of triangles that have previously been found to gradually grow the PVS. The process of random sampling and the subsequent exploration is repeated until the rate of convergence of the PVS falls below a certain threshold.

The idea of GVS is to intelligently place samples such that new triangles are likely to be found. In contrast, a pure regular sampling approach would sample the same triangles over and over again. Figure 4.1 illustrates a regular sampling strategy using ray space. A pure random sampling strategy shows the same problem: The higher the degree of convergence of the PVS, the higher the probability of a random ray to intersect a triangle that has already been intersected. GVS alleviates this problem by using an intelligent sampling scheme in the exploration phase to increase the likelihood of intersecting new triangles.

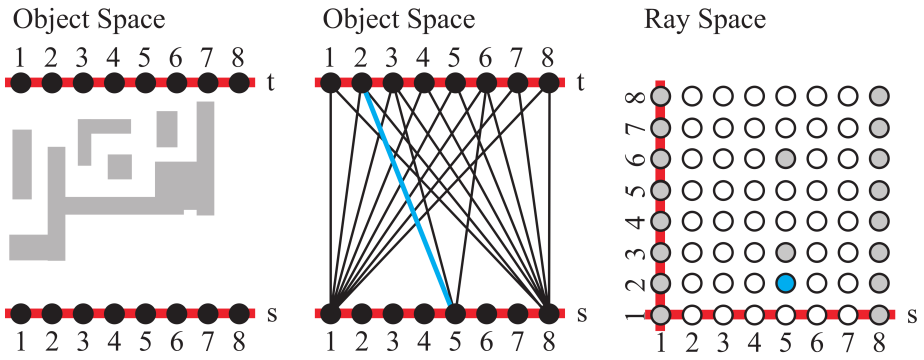


Figure 4.1: An example 2D scene (left) is sampled using regular sampling. A subset of the rays is shown in object space (middle) and in ray space (right). Rays are traced from the view cell (bottom red line parameterized with s) to a plane behind the scene (top red line parameterized with t). The ray space is defined by all the rays emanating from the view cell. Figure from Wonka et al. [WWZ+06].

4.2.1 Random Sampling

Random rays are generated with a pseudo-random sampling strategy using Halton points (see Section 2.2.2). Halton points are mapped to the view cell to get uniformly distributed ray origins:

$$u = h_{a,i},\ v = h_{b,i},\ \phi = 2\pi h_{c,i},\ \theta = \arcsin(h_{d,i}),$$

where $h_{a,i}$ is the i-th point of the Halton sequence h_a with base a. Intersected triangles are treated as "seed points" for the subsequent exploration phase and are stored in a queue (see Figure 4.2).

4.2.2 Exploration Phase

The exploration phase is a combination of two algorithms that implement intelligent sampling strategies. The first algorithm is *adaptive border sampling* (ABS), which is motivated by the goal of finding new triangles in the neighborhood of a triangle that has already been sampled. Additionally, a recursive subdivision strategy is employed to find triangles that may have been missed. The second algorithm of the exploration phase is the *reverse sampling* (RS) algorithm, which efficiently handles discontinuities.

Adaptive border sampling The adaptive border sampling algorithm (see Figure 4.2) iteratively processes the triangles in the queue, which initially only contains triangles found by random sampling. The current triangle t is enlarged to get a new polygon t'. t' is constructed to be as tight as possible to avoid missing any adjacent triangles. Furthermore, the distance between t and the border polygon t' should be constant in ray space. Therefore, instead of simply using an enlarged triangle, t' is constructed using nine vertices. The vertex positions of t' are sampled from the position on the view cell from where t has initially been found. Since the sample positions are all close to the vertices of t, triangles along the edges of t' can be missed. Furthermore, sampling different triangles through the position of two adjacent border polygon vertices indicates that there may be further triangles along the vertices' connecting edge. This is addressed by recursively subdividing the edge between two vertices x_a and x_b of t' if sampling x_a intersects a different triangle than sampling x_b. An edge is recursively subdivided up to a threshold. Any triangle that is found during this process is added to the queue. This leads to a behavior similar to a flood fill algorithm.

Reverse sampling The reverse sampling algorithm is used to sample discontinuities and regions of space that have not been discovered. The idea is that once a discontinuity is detected during the adaptive border sampling, the origin of the respective ray is mutated such that the ray penetrates the discontinuity.

Let t be a triangle that is currently processed by the adaptive border sampling algorithm. A discontinuity is present if a ray through a vertex of the border polygon t' of t intersects a triangle t_o that is closer than a predicted hit point:

$$|x_{\text{predicted}} - x_{\text{origin}}| - |x_{\text{hit}} - x_{\text{origin}}| > \Delta$$

The predicted hit point is calculated by intersecting the current ray with the plane of t. If there is a discontinuity, the ray origin is mutated on the view cell such that the ray passes by the occluding triangle t_0. Checking if a hit point is *farther* than the predicted hit point is not necessary since this discontinuity is detected during the adaptive border sampling of the farther triangle. Figure 4.3 gives a thorough overview of the algorithm.

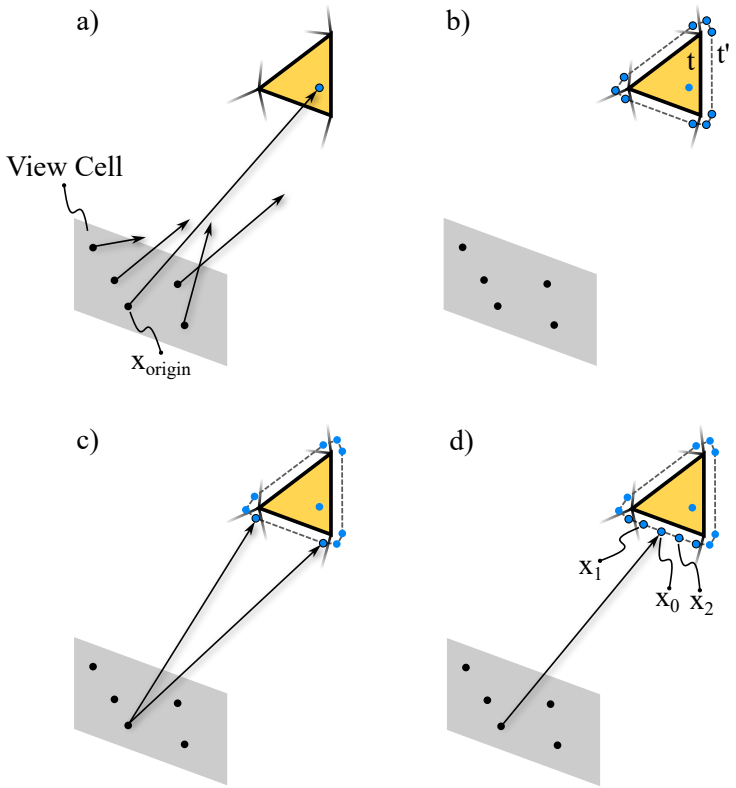


Figure 4.2: Random sampling is used to find initial triangles (seed points) (a). Adaptive border sampling enlarges the boundary of the current triangle t to get a 9-vertices polygon t' (b). Starting from x_{origin}, the point on the view cell from which t has been found during the random sampling, rays are traced to the vertices of t' (c). If the rays of two samples that form an edge of t' intersect different triangles, the edge is subdivided and a new sample (x_0) is placed. Further samples may be placed (x_1, x_2), by the recursive edge subdivision process, up to a given threshold (d). Figure adapted from Wonka et al. [WWZ+06].

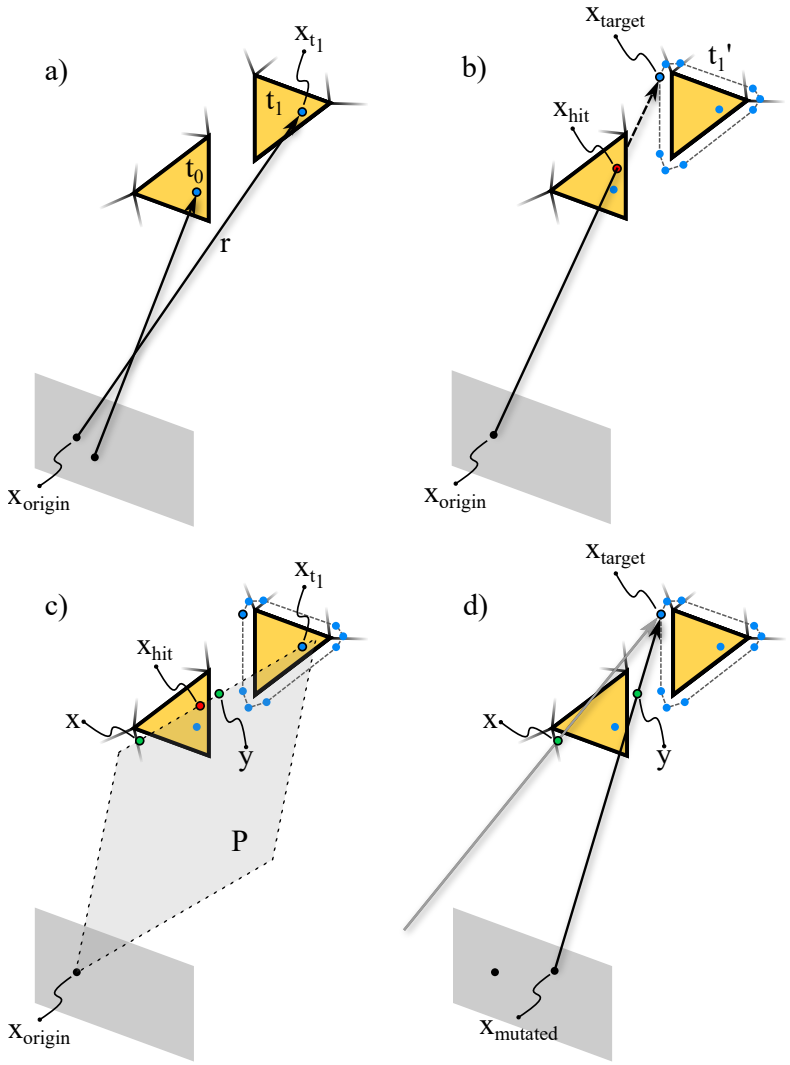


Figure 4.3: Triangle t_0 and t_1 are sampled and added to the queue (a). Adaptive border sampling is executed for t_1, sampling x_{target} of the border polygon t_1'. The ray intersects the closer triangle t_0 at position x_{hit}. A discontinuity is detected, since the intersection point is closer to the ray origin than the sample point x_{target} (b). A plane P is constructed though the sample point x_{t_1} of t_1, x_{hit} and x_{origin}. P is then intersected with t_0 to compute new sample points x and y that are just outside the occluding triangle t_0 (c). The point of intersection x_{mutated} of a line through x_{target} and x or y with the view cell is then the new mutated origin of the initial ray r (d). Figure adapted from Wonka et al. [WWZ+06].

CHAPTER 5

Guided Visibility Sampling++

In this chapter, an improved Guided Visibility Sampling algorithm is presented, which builds upon the ideas of the original algorithm. The main focus is on improving the accuracy of GVS by modifying the deterministic sampling strategies, i.e., the adaptive border sampling and reverse sampling algorithm.

5.1 Introduction

Guided Visibility Sampling++ (GVS++) builds upon the original GVS algorithm. Various changes and improvements were made to increase the overall efficiency of the algorithm. Changes are motivated by shortcomings of the original algorithm and the fact that current hardware is more capable and offers different features such as hardware-accelerated ray tracing than hardware during the time of the development of the original GVS algorithm. Our sampling schemes take the highly parallel nature of modern GPUs into account. The sample location computation is independent of the sampling result of other samples. This allows a modern GPU implementation to rapidly sample in parallel. The main advantage of GVS++ over GVS is that the resulting PVS is a more accurate estimation of the exact visible set, resulting in lower average and maximum pixel errors in the final image.

The general procedure of GVS++ is similar to that of GVS (Section 4.2): Initially, the scene is randomly sampled to find triangles that act as seed points for the subsequent exploration phase. Newly found triangles that are not part of the PVS are added to a queue. The exploration phase processes the triangles in the queue using intelligent sampling strategies. The whole process is repeated until a termination criterion is fulfilled (see Algorithm 5.1). Improvements and changes to the two building blocks of the exploration phase (*adaptive border sampling* (ABS) and *reverse sampling* (RS)) are discussed in the following.

5.2 Random Sampling

The scene is sampled using a pseudo-random sampling approach. Rays emanating from uniformly distributed pseudo-random locations on the view cell are intersected with the scene. A positions on the view cell is determined by mapping Halton points onto the view cell:

$$x_{\text{origin}} = v_p + h_{a,i} * v_{\text{sx}} * v_x + h_{b,i} * v_{\text{sy}} * v_y, \tag{5.1}$$

where v_p is the position of the view cell, $h_{a,i}$ is the i-th point of the Halton sequence h_a with base a, v_s is the size of the view cell and v_x and v_y form the two-dimensional coordinate frame of the view cell. Similar to the random sampling approach of GVS, Halton points are also used to calculate a pseudo-random direction [Shi20]:

$$\begin{aligned} \phi &= 2\pi h_{c,i} \\ \theta &= \arccos(1 - h_{d,i}) \\ x_{\text{dir}} &= (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \end{aligned} \tag{5.2}$$

Unnecessary sin and arccos calculations are avoided by the following transformation:

$$\begin{aligned} m &= 1 - h_{d,i} \\ \theta &= \arccos(m) \\ x &= \sin\theta\cos\phi = \sin(\arccos(m))\cos\phi = \sqrt{1-m^2}\cos\phi \\ y &= \sin\theta\sin\phi = \sin(\arccos(m))\sin\phi = \sqrt{1-m^2}\sin\phi \\ z &= \cos\theta = \sin(\arccos(m)) = m \end{aligned} \tag{5.3}$$

5.3 Exploration Phase

Similar to GVS, the exploration phase is a combination of the adaptive border sampling and reverse sampling algorithms. The ABS algorithm has been changed to address shortcomings of the original algorithm, where in some cases neighboring triangles can be missed. The original reverse sampling algorithm has been replaced by a new approach that is more robust and samples discontinuities more thoroughly.

5.3.1 Adaptive border sampling

Changing the adaptive border sampling was motivated by the fact that in specific cases, neighboring triangles were missed. In such scenes, as illustrated in Figure 5.2, the recursive subdivision is not executed due to the suboptimally formulated condition: An edge of the border polygon is only then recursively subdivided if sampling two vertices of an edge results in two different triangles (Section 4). This can lead to undersampled edges of the border polygon and may therefore miss neighboring triangles.

The problem of undersampled border polygon edges is alleviated by replacing the conditional recursive subdivision by a fixed number of samples that are placed along edges

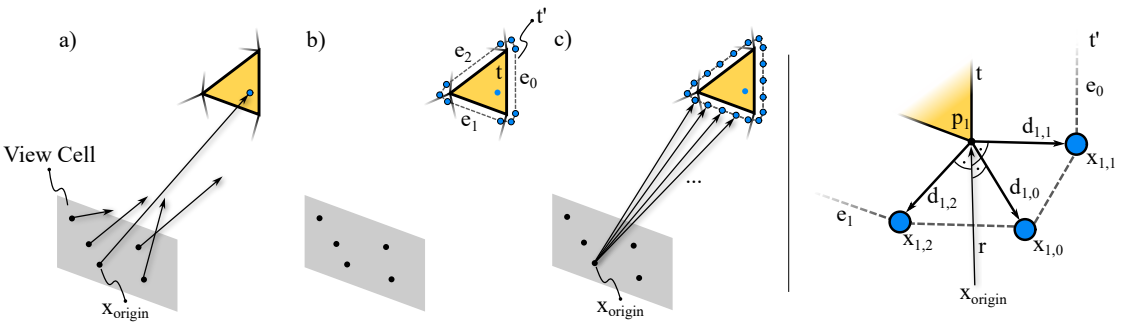


Figure 5.1: Left: Random sampling is used to find initial triangles (seed points) (a). Adaptive border sampling enlarges the boundary of the current triangle t to get a 9-vertices polygon t' (b). Additional samples are placed along the edges e_0, e_1 and e_2 of t'. Starting from x_{origin}, the point on the view cell from which t has been found during the random sampling, rays are traced to the vertices of t' (c). Right (adapted from [WWZ+06]): Close-up of the border polygon t', illustrating the placement of the border polygon vertices $x_{1,i}$.

of the border polygon. This way, the edges of the border polygon are always sampled, independently of the sampling result of other border polygon samples. This procedure is shown in Figure 5.1. As in the GVS [WWZ+06] approach, a triangle t is enlarged to get the border polygon t'. The nine vertices of the border polygon are placed as in the original approach (see Figure 5.1). Close (ϵ) to each corner p_i of a triangle t, three vertices $x_{i,1}$, $x_{i,2}$ and $x_{i,3}$ are placed. $x_{i,1}$ and $x_{i,2}$ are placed on vectors ($d_{i,1}$ and $d_{i,2}$) perpendicular to the ray. $x_{i,0}$ is placed on the angle bisector $d_{1,0}$.

$$
\begin{aligned}
r &= p_i - x_p \\
d_{i,i+1} &= \text{normalize}(r \times (p_{i+1} - p_i)) \\
d_{i,i-1} &= \text{normalize}(r \times (p_i - p_{i-1})) \\
d_{i,i} &= \begin{cases} \text{normalize}(d_{i,i-1} + d_{i,i+1}) & \text{if } d_{i,i-1} \cdot d_{i,i+1} > 0 \\ \text{normalize}(r \times d_{i,i-1} + d_{i,i+1} \times r) & \text{else} \end{cases} \\
x_{i,j} &= p_i + \epsilon \cdot d_{i,j}
\end{aligned}
\tag{5.4}
$$

The position of a sample s_i along an edge e with end points x_a and x_b is calculated by linearly interpolating between the end points:

$$
s_i = (1 - t) \cdot x_a + t \cdot x_b \tag{5.5}
$$

Instead of placing a fixed number of samples along an edge, the number of samples could be chosen adaptively depending on the length of edge e.

5.3.2 Reverse sampling

The original reverse sampling approach mutates the starting point of a ray on the view cell such that the ray passes by an occluding triangle. Up to two mutated starting points are computed and rejected if one is outside of the view cell. It cannot be guaranteed that the mutated ray penetrates the actual discontinuity, since it may intersect other triangles along its path. The discontinuity is not sampled at all if none of the two mutated starting points are within the boundaries of the view cell. This is commonly the case when very narrow view cells are used, which is illustrated in Figure 5.2. The fact that discontinuities may not be sampled due to mutated ray origins being outside of the view cell motivated working on an improved reverse sampling approach.

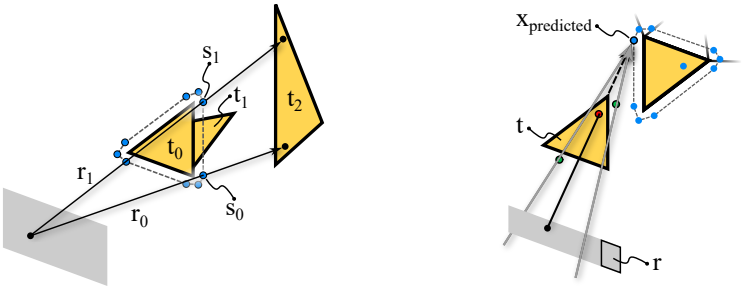


Figure 5.2: Left: An example scene illustrating the problem of a missed adjacent triangle due to an undersampled border polygon edge. Triangle t_0 is intersected by the rays r_0 and r_1 through the samples s_0 and s_1 of the border polygon of t_0. The edge between s_0 and s_1 would only be subdivided if rays r_0 and r_1 would intersect different triangles, which does not apply in this case. Therefore, the neighboring triangle t_1 is not found. Right: An example scene illustrating the problem of the original reverse sampling algorithm when using narrow view cells. The rays are mutated such that the occluding triangle t_0 is passed, however, neither ray intersects the view cell and are therefore discarded. Region r on the view cell would contain valid positions to sample $x_{\text{predicted}}$.

The reverse sampling approach in GVS++ combines the idea of mutating the starting point of the ray on the view cell as well as choosing mutated ray starting points such that the occluding triangle t_0 is not intersected. Instead of computing sample locations which may not be within the boundaries of the view cell, sample locations are directly distributed on the view cell. Sample locations are distributed along the edges and on the corners of the view cell. Additionally, points are uniformly distributed on the surface of the view cell based on the Halton sequence. To avoid intersecting the occluding triangle t_0 again, t_0 is projected onto the view cell. If a sample point lies within the projected triangle p_0, any ray from such a point to x_{target} would intersect t_0. Therefore, rays are only traced from points that are not within the boundaries of p_0 (see Figure 5.4).

The occluding triangle t_0 is projected onto the view cell, by constructing lines originating at x_{target} through the vertices of t_0. The lines are intersected with the plane of the view cell to get the projected triangle p_0 with coordinates (A, B, C). Baryzentric coordinates

are used to determine whether a sample point on the view cell is within p_0, since any point P of a triangle (A, B, C) can be expressed as $P = uA + vB + wC$, where (u, v, w) are barycentric coordinates and $u + v + w = 1$ [Eri04]. The barycentric coordinates (u, v, w) of each sample point are calculated. A sample point is within p_0 if

$$0 \leq v \leq 1,\ 0 \leq w \leq 1,\ v + w \leq 1. \tag{5.6}$$

Similar to the original reverse sampling approach, a discontinuity is detected, if the ray/triangle intersection point x_{hit} is at least Δ closer than x_{target}:

$$|x_{\text{target}} - x_{\text{origin}}| - |x_{\text{hit}} - x_{\text{origin}}| > \Delta \tag{5.7}$$

The improved reverse sampling approach also handles discontinuities, where sampling a vertex of a border polygon during the adaptive border sampling does not intersect any triangle. This helps finding triangles that are neither found by random sampling nor by adaptive border sampling (illustrated in Figure 5.3).

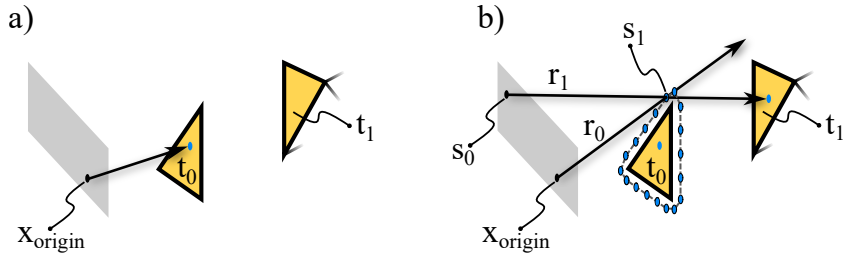


Figure 5.3: An example scene illustrating sampling discontinuities where no triangles are intersected. Triangle t_0 is sampled by a ray with origin x_{origin} (a). Sample s_1 of the border polygon of t_0 is sampled. Since no triangle is intersected, a discontinuity is detected and handled by reverse sampling. Reverse sampling constructed a ray with a muted starting points s_0 that samples s_1 and intersects t_1.

5.4 Termination Criterion

A termination criterion is used to decide when the visibility sampling should be terminated, and the PVS is considered to be converged. Different termination criteria may be used that take the PVS size, the total number of traced rays, or the rate at which the PVS grows into account. In our implementation, the termination criterion is checked after the exploration phase. We test whether the number of found triangles is below a threshold T, e.g., 10 or 50. If the number of found triangles is below T, the algorithm is terminated, and the PVS is considered converged. Otherwise, the algorithm starts anew by, again, randomly sampling the scene followed by an exploration phase. A different termination criterion where the number of newly found triangles per n rays is checked may be used instead. The termination criterion may also be checked more frequently, for instance, after each execution of adaptive border sampling and reverse sampling.

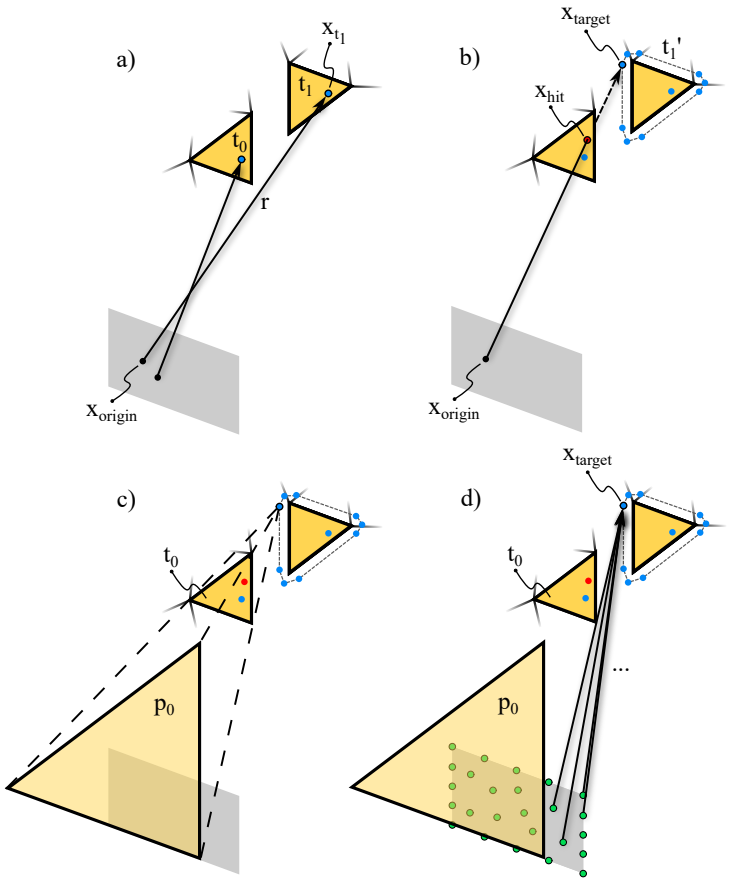


Figure 5.4: Triangle t_0 and t_1 are sampled and added to the queue (a). Adaptive border sampling is executed for t_1, sampling x_{target} of the border polygon t_1'. The ray intersects the closer triangle t_0 at position x_{hit}. A discontinuity is detected, since the intersection point is closer to the ray origin than the sample point x_{target} (b). The occluding triangle t_0 is projected onto the view cell from the viewpoint of x_{target} to get the projected triangle p_0 (c). Samples are distributed uniformly on the view cell and along its edges. For each sample on the view cell that is not within the bounds of p_0, a ray is traced to x_{target} (d).

5.5 3D View Cells

GVS as well as GVS++ can be used to compute the PVS of three-dimensional view cells by running the algorithm on each polygon of its boundary. In the simplest form, rectangular cuboids can be used that are formed by six two-dimensional view cells. As previously illustrated, GVS++ handles narrow view cells better than GVS due to the new reverse sampling algorithm. Therefore, our algorithm is better suited for scenarios in which long and narrow view cells are used. An example is given in Chapter 8, where three-dimensional view cells are placed along streets.

5.6 Data Structures and Optimizations

We execute the main workload of our algorithm, i.e. the sampling of the scene, on the GPU. This is possible since our intelligent sampling algorithms can easily be parallelized to be executed on the GPU.

We show that a naive implementation significantly impedes performance. In a naive implementation, whenever a triangle is intersected on the GPU, its ID and accompanying intersection data is stored in a buffer on the device. Once the shader finishes execution, the buffer is transferred to the host. Note that this buffer may contain duplicate elements. On the host-side, a *set* data structure is used to store the PVS. The intersected triangles are inserted into the PVS. Newly intersected triangles are also inserted into a queue for further processing in an exploration phase. This naive approach is over 20 times slower than our optimized implementation. This is due to the expensive set operations on the host-side during which the GPU is in an idle state, since further exploration phase shader dispatches depend on the result of the set operations. Therefore, to maximize performance, the goal should be to minimize idle times of the GPU.

In our optimized implementation, which is discussed in further detail in Chapter 6, we achieve minimal GPU idle times by by keeping the frequency and amount of data transferred between the CPU and the GPU to a minimum. Furthermore, we avoid expensive set operations on the CPU-side by ensuring uniqueness of stored elements on the GPU.

Instead of storing the PVS on the CPU, we store it in VRAM on the GPU to allow for fast read and write access by the shaders. Furthermore, the buffer storing the intersected triangle IDs and intersection data is allocated from memory on the host system that also allows access from the GPU. This way less data has to be transferred between the device and the host resulting in a 3× speedup compared to keeping this buffer in VRAM and frequently transferring it between Whenever a triangle is intersected, our implementation checks whether it is already stored in the PVS via an atomic compare-and-swap operation. If the triangle is sampled for the first time, it is inserted into the PVS and into the buffer that is later accessed by the host. This buffer's content is inserted into a queue on the CPU-side. The queue is realized as a dynamic size array to allow efficient insertion of the whole buffer at once. No set is required since the content of the buffer does not contain any duplicate elements. We further reduce CPU-GPU communication times by submitting data transfer workloads to a transfer-only queue to ensure maximum transfer speeds.

Algorithm 5.1: Guided Visibility Sampling++

```
1  function main():
2      while checkTerminationCriterion() do
3          for ray in generateRandomRays() do
4              processRay (ray)
5          end
6          for x in queue do
7              adaptiveBorderSampling (x)
8          end
9      end
10
11 function processRay(ray):
12     primitiveID, hitPoint = trace(ray)
13     if primitiveID not in PVS then
14         PVS += primitiveID
15         queue += { primitiveID, hitPoint, origin(ray), direction(ray) }
16     end
17     return { primitiveID, hitPoint }
18
19 function adaptiveBorderSampling(x):
20     t' = enlarge(primitiveID(x))
21     for x_target in t' do
22         result = processRay({ origin(x), x_target - origin(x) })
23         if checkDiscontinuity(result, x_target) then
24             reverseSampling (x, sample)
25         end
26     end
27
28 function reverseSampling(x, x_target):
29     p = projectPrimitiveOntoViewCell(primitiveID(x))
30     samples = generateSamplesAlongViewCellEdges() +
         generateUniformViewCellSamples()
31     for sample in samples do
32         if sample not withinBoundaries(p) then
33             processRay({ sample, x_target - sample })
34         end
35     end
```

CHAPTER 6

Implementation

An application has been developed that implements the GVS++ algorithm. It consists of host- and device-code. The main parts of GVS++ are implemented on the device using shaders. Data is stored in system memory on the host and the device depending on usage and size. This allows keeping data on the device that is not modified by the host, such as a buffer storing the PVS. This way, expensive data transfer between the host and the device is kept to a minimum. Wavefront OBJ files containing the scene are loaded by the application and a file containing various parameters is read. The file is user-specified and contains settings to alter the behavior of the GVS++ algorithm, such as sample counts. GVS++ samples the scene and creates a PVS per view cell. After the visibility sampling terminates for a view cell, the PVS is transferred from the device to the host, where it may directly be used for rendering or stored into a file or database for later use. In this chapter, this application, with a focus on the implementation of GVS++, is presented. The source code is freely available on GitHub[1]. Section 6.1 lists the software and libraries the implementation is based on. In Section 6.2 the implementation is discussed in detail, including pseudo-code of the main parts of the GVS++ algorithm.

6.1 Software and Libraries

The reference implementation of GVS++ is written in C++11 and GLSL and uses a few well-known libraries:

- Vulkan SDK[2] version 1.2.135.0 is used. Hardware-accelerated ray-tracing functionality is accessed through the `VK_NV_ray_tracing` extension.
- GLFW[3] version 3.3.2 is used to manage windows and handle input events.

[1] `https://github.com/einthomas/GVSPP`, last accessed 30. November 2020
[2] `https://www.khronos.org/vulkan/`, last accessed 20. October 2020
[3] `https://www.glfw.org/`, last accessed 20. October 2020

- GLM[4] version 0.9.9.7 is used for vector and matrix operations on the host side.
- tinyobjloader[5] version 1.0.7 is used to load scenes stored as Wavefront OBJ.
- glslc[6], included with the Vulkan SDK, is used to compile GLSL shaders down to SPIR-V.

6.2 GVS++ Implementation

A high-level overview of the whole system, including the GVS++ algorithm that has been implemented, is given in Figure 6.1. First, the Vulkan API is initialized, and settings and a model is loaded. After initializing the GVS++ algorithm, Halton points are generated that are then consumed by the random sampling process. The PVS is stored on the device. Whenever the random or ABS and RS shader intersects a new triangle, an identifying triangle ID is inserted into the PVS. If the triangle was previously not part of the PVS, its ID along with additional data such as the ray/triangle intersection position is stored in a buffer. After the shader has finished its execution, the buffer's content is inserted into a queue on the host-side. The adaptive border and reverse sampling shader processes the elements of the queue, which are transferred back to the device. This process repeats until the queue is empty. If the queue is empty, a termination criterion is checked. If execution is terminated, the PVS is transferred from the device to the host and is stored, e.g., into a file. Otherwise, the GVS++ algorithm starts anew by generating new Halton points. A high-level overview of the communication between the host and the device is given in Table 6.1.

6.2.1 Vulkan Specifics

The system starts with initializing the Vulkan API. The validation layer `VK_LAYER_KHRONOS_validation` that is included in the Vulkan SDK is used. The extension `VK_NV_ray_tracing` is loaded in order to access ray-tracing functionality.

Acceleration structures The top- and bottom-level acceleration structures are built, using the previously created vertex and index buffer of the loaded model. The creation of the acceleration structures deserves attention. The geometry of the bottom-level acceleration structure is described by a `VkGeometryNV` structure, where the `VK_GEOMETRY_OPAQUE_BIT_NV` flag is set. This signals the ray tracer that there is no transparent geometry, and therefore no any-hit shaders should be invoked. Furthermore, both acceleration structures are created with the `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV` flag set, which indicates that BVH traversal performance should be preferred over BVH update or rebuild speed.

[4] `https://glm.g-truc.net`, last accessed 20. October 2020
[5] `https://github.com/tinyobjloader/tinyobjloader`, last accessed 20. October 2020
[6] `https://github.com/google/shaderc`, last accessed 20. October 2020

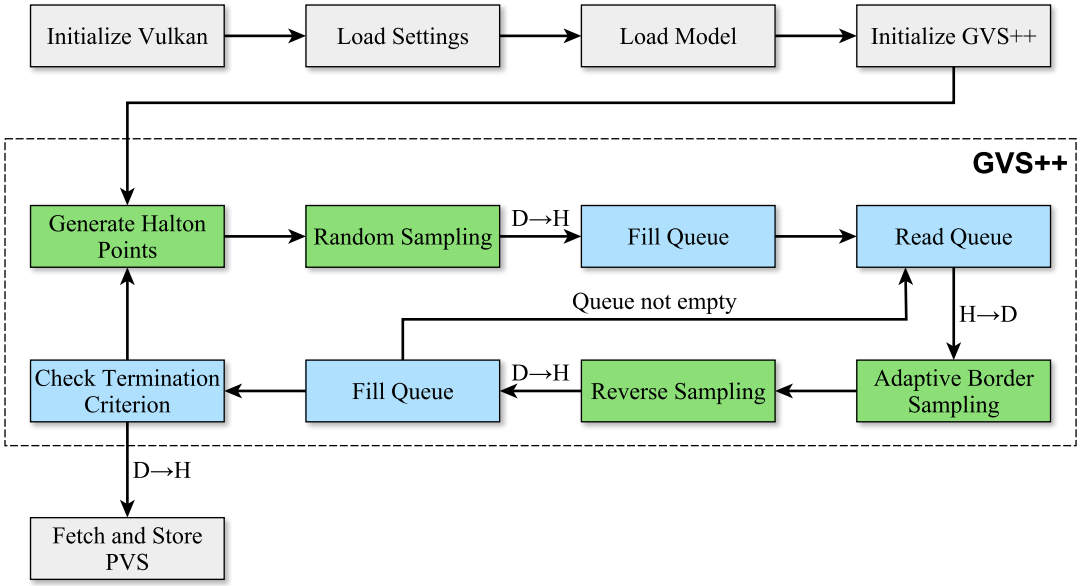


Figure 6.1: An overview of the implementation of GVS++. Green boxes represent components that are implemented on the device. Blue boxes are components that are implemented on the host side. The Halton point generation as well as the random sampling are implemented as separate shaders. The adaptive border and reverse sampling are realized as a single common shader. D→H and H→D shows that data is transferred from device to host and vice versa.

Pipelines and shaders Since the Halton point generation, random sampling, and ABS and RS components are implemented as separate shaders, multiple pipelines have to be created. A compute pipeline is created for generating Halton points, and two ray-tracing pipelines are created for random sampling and for the exploration phase, i.e., ABS and RS. The main logic of the random sampling, and adaptive border sampling and reverse sampling shaders are implemented in ray generation shaders, where ray tracing is started. A common closest hit shader is implemented that uses the ID of the intersected primitive to access an index and vertex buffer to get the vertex position of the intersected primitive. Barycentric weights of the intersection are accessible in the shader and are stored in `hitAttributeNV`. The barycentric weights and the vertex positions of the intersected primitive are used to calculate the world position of the ray/primitive intersection. The world positions and the primitive ID are then stored as the ray payload `rayPayloadInNV`. The ray generation shader can then access the ray payload. Also, a common miss shader is used that stores -1 as the ray payload, indicating that no primitive is intersected.

Queues Compute workloads, including Halton point generation and ray-tracing shader dispatches, are submitted to the queue with the compute bit set that is not capable of

CPU	GPU
Dispatch Halton Generation Shader	*Idle*
`vkWaitForFences`	Halton sequence generation
Dispatch Random Sampling Shader	*Idle*
`vkWaitForFences`	**Random sampling**
Add result to queue	*Idle*
Dispatch ABS & RS Shader	*Idle*
`vkWaitForFences`	**ABS & RS**
Add result to queue	*Idle*

Table 6.1: High-level overview of the communication between the CPU and the device.

graphics workloads (see Table 2.2). Data transfer tasks between the device and the host are submitted to a transfer-only queue.

Synchronization Only basic synchronization between the host and the device using fences is required since the tasks on the host-side, such as filling the queue or generating compute workload, directly depend on the results of the shaders on the device. Therefore, whenever compute or transfer workload is submitted, a `VkFence` handle is passed to `vkQueueSubmit`. `vkWaitForFences` is then used to wait for the execution of the submitted command buffer to be completed (see Table 6.1).

Buffers

Well thought-through memory management is crucial to achieving good performance when using Vulkan. When choosing the memory type of a buffer, its usage and size, as well as the available memory, should be taken into consideration.

In the implementation of GVS++, a storage buffer on the device (device local) is used to store the PVS. The size of the buffer is equal to the number of primitives of the current scene. The buffer stores primitive IDs p_i as integers and is initialized with -1. This allows efficiently determining if a primitive is part of the PVS by checking if the value of

the buffer at position p_i is equal to -1. The disadvantage is that the storage requirement of the buffer depends on the number of primitives of the loaded model.

Halton points are generated on the device and are stored in a storage buffer that resides in device memory (device local). Storing the Halton points directly on the device avoids unnecessary transfer operations between the host and the device. Furthermore, device access times to device local memory is lower than when accessing system memory.

The random sampling, and the ABS and RS shader share a single buffer residing in system memory (host visible) to store results. In this case, device local memory would not give much of a performance advantage since the device only executes write operations on the buffer. This also helps saving device memory, which is typically more limited than system memory. Since the buffer is shared between both shaders, sufficient memory has to be allocated during the initialization step. Memory is allocated such that a maximum of $\min(\max(n_{\text{rand}}, n_{\text{ABS}} * n_{\text{RS}}), N)$ elements, where N is the number of primitives of the scene, can be stored. The actual number of occupied indices is tracked via a counter. In contrast to OpenGL, atomic counter variable types are not supported by Vulkan. Therefore, the counter is realized as a storage buffer, and `atomicAdd` is used to increment the counter.

The queue of primitives that are to be processed by the ABS and RS shader is kept on the host. A `std::vector` of the C++ standard library is used. Queue elements are transferred to the device into a device local storage buffer for further processing by the ABS and RS shader.

Memory mapping In order to be able to read or write to host visible memory, a pointer to the memory has to be acquired by using `vkMapMemory`. To avoid repeated mapping and unmapping memory whenever data is transferred between the host and the device, memory is kept persistently mapped, meaning that memory is mapped once during the initialization and unmapped before the memory is freed.

6.2.2 Parameters

After initializing the Vulkan API, a settings file is loaded where various parameters controlling the behavior of GVS++ can be set:

- The number of random rays n_{rand} that are traced during the random sampling step, e.g., 500 000 random rays.
- Parameters that control the behavior of adaptive border sampling: The distance Δ_{ABS} that dictates the enlargement of the original triangle to get the border polygon, e.g., 0.001, and the number of samples n_{ABSEdge} placed along the three long edges of a border sampling polygon, e.g., 5. The total number of samples on the border polygon is then calculated as $n_{\text{ABS}} = n_{\text{ABSEdge}} * 3 + 9$, since the border polygon is formed by nine vertices and consists of three long edges.

- Parameters that control the behavior of reverse sampling: The number of samples n_{RSEdge} placed along each edge of a view cell, e.g., 20, and the number of samples n_{RSArea} uniformly distributed on the surface of the view cell, e.g., 10. The total number of samples on the view cell for reverse sampling is then $n_{\text{RS}} = n_{\text{RSEdge}} * 4 + n_{\text{RSArea}}$.
- Whether 3-dimensional view cells are used.
- The threshold T is used as a termination criterion. If less than a total of T new triangles are found during one iteration of the algorithm, i.e., during random sampling and subsequent processing of the queue, GVS++ is terminated.

After loading the settings file, tinyobjloader is used to load the scene.

6.2.3 Halton Points Generation

The first component of the GVS++ implementation is the generation of Halton points, which are used to generate random rays. The Halton point generation is implemented as a compute shader, which is faster than a host or a CUDA implementation. The random sampling shader traces n_{rand} random rays for which a random position on the view cell and a random direction is calculated. Therefore, $n_{rand} * 4$ Halton points are computed. The calculated Halton points are stored in a storage buffer, which is later used by the random sampling shader. Since new Halton points may be generated repeatedly, randomized Halton [Bha03] sequences are used. The idea is that a random number $\mu \in [0, 1]$ drawn from a uniform distribution is added to all Halton points h_i:

$$\overline{h_i} = \text{mod}(h_i + \mu, 1),$$

where $\overline{h_i}$ is the randomized Halton point.

Halton points are generated on the device and are stored in a storage buffer that resides in device memory (device local).

6.2.4 Random Sampling

The random sampling algorithm, implemented as a shader, uses the previously generated Halton sequences that are already stored on the device. This saves unnecessary H→D transfer oprations. Algorithm 6.1 gives an overview of the implementation. The body of the loop is implemented in the GLSL shader, which is invoked n_{rand} times. The function `traceNV` is used to trace rays. It takes several parameters, such as the acceleration structure that is traversed. The ray flags `gl_RayFlagsOpaqueNV`, indicating that no any-hit shaders should be invoked, and `gl_RayFlagsCullBackFacingTrianglesNV` are used. RTX ray tracing is watertight and therefore guarantees that a ray does not pass through shared edges or vertices (see Section 2.3.6), however, backfacing triangles can still be registered as visible when a silhouette edge is hit. If the model at hand is watertight, this is avoided by using the `gl_RayFlagsCullBackFacingTrianglesNV`

Algorithm 6.1: Random Sampling

for $i \leftarrow 0$, $i < n_{\text{rand}}$ **do**
  v_p, $v_s \leftarrow$ get view cell position and size
  $h_{a,i}$, $h_{b,i} \leftarrow$ get i-th Halton point from base a and base b sequence
  v_x, $v_y \leftarrow$ calculate view cell frame
  $x_{\text{origin}} = v_p + h_{a,i} * v_{\text{sx}} * v_x + h_{b,i} * v_{\text{sy}} * v_y$ ▷ See Equation 5.1

  $h_{c,i}, h_{d,i} \leftarrow$ get i-th Halton point from base c and base d sequence
  $r \leftarrow \sqrt{\max(1 - h_{d,i} * h_{d,i},\, 0)}$
  $x_{\text{dir}} \leftarrow (r * \cos(2\,\pi\, h_{c,i}),\, r * \sin(2\,\pi\, h_{c,i}),\, h_{d,i})$ ▷ See Equation 5.3

  primitive id, hit point $p \leftarrow$ trace ray $(x_{\text{origin}}, x_{\text{dir}})$
  if id **not in** PVS **then**
    insert id into PVS
    store tuple $(id, x_{\text{origin}}, p)$ in output buffer
  end
end

flag, which enables back-face culling. We found that with disabled back-face culling 1-2% more triangles are found, i.e., back-facing triangles.

Once a triangle is hit, a tuple consisting of the intersected primitive ID, the ray origin, and the intersection point, are stored in a storage buffer residing in system memory, as previously described (Section 6.2.1). The content of the storage buffer is appended to the queue on the host-side, without requiring further set operations to ensure uniqueness. This is possible since the shader guarantees that the buffer does not contain duplicate primitives or primitives that are already in the queue.

The actual method used for inserting primitive IDs into the PVS on the device deserves attention. In order to prevent inserting tuples with equal *id* into the output buffer, the PVS is checked if it contains *id*. If it does not contain *id*, it should be inserted into the PVS and into the output buffer. This requires an atomic operation due to the parallel execution of the shader. `atomicCompSwap` is used to compare and insert a primitive ID into the PVS. If the original value returned by the function is -1, *id* has successfully been inserted into the PVS, and the shader can proceed to insert into the output buffer. An alternative strategy is discussed in Section 6.2.7.

6.2.5 Adaptive Border Sampling and Reverse Sampling

The adaptive border sampling and reverse sampling are implemented as a single mutual shader. The ABS and RS algorithm processes the queue stored on the host where new found triangles are added. On the host side, elements are taken from the queue and are transferred to the device. Algorithm 6.2 is implemented such that the host dispatches

Algorithm 6.2: Adaptive Border Sampling

```
1  s ← get current sample
2  for i ← 0, i < 3 do                                  ▷ Calculate border polygon vertices
3  |  for j ← 0, j < 3 do
4  |  |  x_{i,j} ← calculate j-th border polygon vertex at the i-th vertex of the
   |  |   original triangle                              ▷ See Equation 5.4
5  |  end
6  end
7
8  for i ← 0, i < 3 do
9  |  x_i, x_{i+1} ← get border polygon vertices adjacent to edge e_i   ▷ See Figure 5.1
10 |  for k ← 1, k < n_ABSEdge do                        ▷ Sample border polygon along edge
11 |  |  a ← lerp(x_i, x_{i+1}, k/(n_ABSEdge + 1))
12 |  |  x_origin ← get ray origin from current sample s
13 |  |  x_dir ← normalize(x_origin − a)
14 |  |
15 |  |  primitive id, hit point p ← trace ray (x_origin, x_dir)
16 |  |  if id not in PVS then
17 |  |  |  insert id into PVS
18 |  |  |  store tuple (id, x_origin, p) in output buffer
19 |  |  end
20 |  |
21 |  |  if |a − x_origin| − |p − x_origin| > Δ_RS then   ▷ Check for discontinuity
22 |  |  |  reverse sampling                             ▷ See Algorithm 6.3
23 |  |  end
24 |  end
25 end
```

the shader n_{ABS} times, i.e., the shader is dispatched for each sample on the border polygon. This way, border polygon samples are processed in parallel, improving hardware utilization. In each shader launch, a single border polygon vertex or sample along an edge of the border polygon is computed, and a ray is traced. If there is a discontinuity, reverse sampling rays are traced in a loop within the same shader execution. Note that the reverse samplig algorithm is implemented as part of the ABS shader (Algorithm 6.3). After the shader has finished execution, the content of the storage buffer (Section 6.2.1) is appended to the queue on the host-side.

6.2.6 Comparison to the Original GVS Implementation

The original implementation of the GVS algorithm does not utilize the GPU for visibility sampling. A CPU ray tracer, the *multi-level ray tracing algorithm* (MLRTA) [RSH05], is used for ray/primitive intersection computation. One key idea of MLRTA is to exploit

Algorithm 6.3: Reverse Sampling

```
1  o_id ← get occluding triangle id
2  S ← {}
3  for i ← 0, i < 4 do                                   ▷ Generate sample points along view cell edges
4  |  c_i, c_{i+1} ← get i-th and adjacent i + 1-th view cell corner
5  |  for j ← 0, j < n_RSEdge do
6  |  |  S ∪ lerp(c_i, c_{i+1}, j/n_RSEdge)
7  |  end
8  end
9  for i ← 0, i < n_RSViewCell do                        ▷ Generate sample points on view cell area
10 |  S ∪ get Halton point and project onto view cell     ▷ See Algorithm 6.1
11 end
12
13 if o_id = −1 then                                     ▷ Handle discontinuity without an occluding triangle
14 |  for s ∈ S do
15 |  |  primitive id, hit point p ← trace ray (s, normalize(x_target))
16 |  |  if id not in PVS then
17 |  |  |  insert id into PVS
18 |  |  |  store tuple (id, x_origin, p) in output buffer
19 |  |  end
20 |  end
21 else                                                   ▷ Handle discontinuity with an occluding triangle
22 |  for i ← 0, i < 3 do                                 ▷ Project occluding triangle onto the view cell
23 |  |  v ← get i-th vertex position of occluding triangle x_origin ← x_target
24 |  |  x_dir ← x_target − v
25 |  |  hit point p_i ← intersect ray (x_origin, x_dir) with the plane of the view cell
26 |  end
27 |  for s ∈ S do
28 |  |  b ← calculate baryzentric coordinates of s in respect to the vertices p_i of
   |  |   the projected triangle
29 |  |  if b is within the projected triangle then
30 |  |  |  primitive id, hit point p ← trace ray (s, normalize(x_target))
31 |  |  |  if id not in PVS then
32 |  |  |  |  insert id into PVS
33 |  |  |  |  store tuple (id, x_origin, p) in output buffer
34 |  |  |  end
35 |  |  end
36 |  end
37 end
```

the spatial coherence between rays by tracing packets of rays at once. A k-d tree is used as an acceleration structure.

The general structure of the original implementation of GVS is similar to the modern Vulkan implementation of GVS++. The key difference is that the GVS++ implementation utilizes the GPU for the main parts of the algorithm. Limitations posed by the device, as well as the communication and synchronization between host and device, increases the complexity of the modern implementation compared to the host-side-only code of the original implementation of GVS. One main difference is that the GVS implementation uses dynamic arrays on the host-side, such as `vector` of C++'s standard template library, to store the PVS and the queue of primitives that are to be processed in the exploration phase. After a packet of rays has been traced, the queue and the PVS are updated. In contrast, the GVS++ implementation uses fixed-size buffers on the device- and on the host-side that are allocated at the start of the application.

6.2.7 GPU Hash Set Approach

An alternative way to store the PVS on the device side is to use a hash set data structure instead of an array. The main advantage is that a hash set potentially requires less memory than an array. In the latter case, the memory required is proportional to the number of triangles in a given scene, while a hash set can grow dynamically and ensures that each stored element is unique. Since the PVS is stored on the device, measures have to be taken to ensure consistency of the data structure under highly parallel access. This is commonly implemented by using locks to guarantee that sections of code are only executed by one thread at a time. In a highly parallel environment such as a GPU, having multiple threads blocked can lead to heavily degraded performance. Therefore, a lock-free GPU hash set implementation is used, based on the work by Farrell [Far].

The GPU hash set implementation employs a *linear probing* approach: To insert an element x, its hash $h(x)$ is calculated. If there is no element stored at $h(x)$, x is inserted at that position of the set. Otherwise, an empty position is searched. This process is also described in Algorithm 6.4. The lock-free approach is realized by using GLSL's `atomicCompSwap(mem, compare, data)` function. The function compares atomically whether `mem` is equal to `compare`. If that is the case, `mem` is replaced by `data`. In any case, the previously stored value is returned.

Initially, a storage buffer of fixed size is allocated which stores the elements of the hash set. Before a random sampling or ABS shader, which potentially store elements into the set, is dispatched, it is checked whether the set should grow. The set is grown by first transferring its content from the device to the host. The occupied memory by the initial storage buffer is then freed, and more memory is allocated. The original values of the set are then re-inserted into the larger buffer using Algorithm 6.4.

The main advantage of this approach compared to a simple array approach where a triangle ID i is stored at position i is that potentially less memory is required. However, this might most often not be the case. Let N be the number of elements in the

queue for which the ABS shader is executed. For each element, the shader is executed $n_{\text{ABSEdge}} * 3$ times, potentially tracing $n_{\text{RSEdge}} * 4 + n_{\text{RSArea}}$ RS rays. Therefore, up to $N * (n_{\text{ABSEdge}} * 3 + n_{\text{RSEdge}} * 4 + n_{\text{RSArea}})$ triangles may be intersected. With typical parameter values (Table 7.2), N must be kept small to avoid the set to consume as much memory as the simpler array approach. For example, for the parameters in Table 7.2 and the CANYON scene the initial random sampling finds 122 482 triangles. Since $n_{\text{ABSEdge}} = n_{\text{RSEdge}} = n_{\text{RSArea}} = 20$, up to 160 rays and therefore up to 160 triangles may be intersected for each of the 122 482 triangles in the queue. Dispatching the ABS shader for each element in the queue, the hash set would therefore have to be resized to be able to fit up to 19×10^6 triangles. However, the CANYON scene consists only of 2×10^6 triangles, causing the hash set to use as much memory as a simple array approach. Alternatively, the ABS shader could be dispatched only for a fraction of the elements in the queue, which would however negatively impact performance. Therefore, the simpler array approach is found to be more viable for storing the PVS.

Algorithm 6.4: Linear Probing GPU Hash Set Insert using MurmurHash3

```
   Input: An element x which should be inserted.
 1 i ← calculate MurmurHash3 hash
 2 while true do
 3 |   v ← atomically store x at position i, if the set at i is empty, and return the
   |     previously stored value
 4 |   if v = empty then
 5 |   |   return true                          ▷ x was successfully stored at i
 6 |   else if v = x then
 7 |   |   return false                         ▷ x is already in the set
 8 |   end
 9 |   i ← mod(i + 1, set capacity)
10 end
```

CHAPTER 7

Results and Evaluation

In this chapter, the GVS++ algorithm is evaluated on several different scenes. Results are presented in the form of graphs that show the algorithm's behavior over the number of traced rays and execution time. The focus is on the former, since this gives results that are largely independent of implementation specifics and the underlying hardware.

In order to quantify the quality of the resulting PVS, the *pixel error* [NB04] is measured. The pixel error is determined by calculating the number of incorrect pixels from 2000 different viewpoints on the view cell rendered in a resolution of 1000×1000. The viewpoints on the view cell are pseudo-randomly distributed, using randomized Halton sequences. The maximum pixel error over all viewpoints and the average pixel error of each view cell are calculated. A connected region of incorrect pixels may be more noticeable than individual incorrect pixels that cause the same pixel error. Therefore, the average and maximum number and size of connected pixel error regions are calculated as well. Table 7.1 contains statistics of the scenes that have been selected for the evaluation: CANYON, a model of the Grand Canyon, PPLANT, the UNC powerplant model[1], GERMANY, a city model ©VIRES Simulationstechnologie GmbH, and BISTRO [Lum17], which has been modified to include the hairball model by Laine et al. [LK10]. The PPLANT scene consists of the most triangles of the four scenes and includes heavily occluded regions. GERMANY is a typical city scene with long streets. This scene is used to cover the case of far away triangles, i.e., triangles at the other end of a street. BISTRO is a small but detailed excerpt from a city. Due to the hairball model's placement, this scene also contains highly occluded regions of small structures. In Figure 7.1, renderings of the scenes themselves are shown. Ten view cells are used for each model. The exact parameters of each view cell are listed in Table 4 to simplify reproducing the results.

The results were produced using a system constisting of an AMD Ryzen 9 3900X CPU, Nvidia GeForce RTX 3080 GPU, and 32GB of RAM. For the detailed parameter analysis

[1] `http://gamma.cs.unc.edu/POWERPLANT/`, last accessed 20. October 2020

Scene	Triangles
CANYON	2 242 504
PPLANT	12 748 210
GERMANY	3 667 284
BISTRO	5 657 097

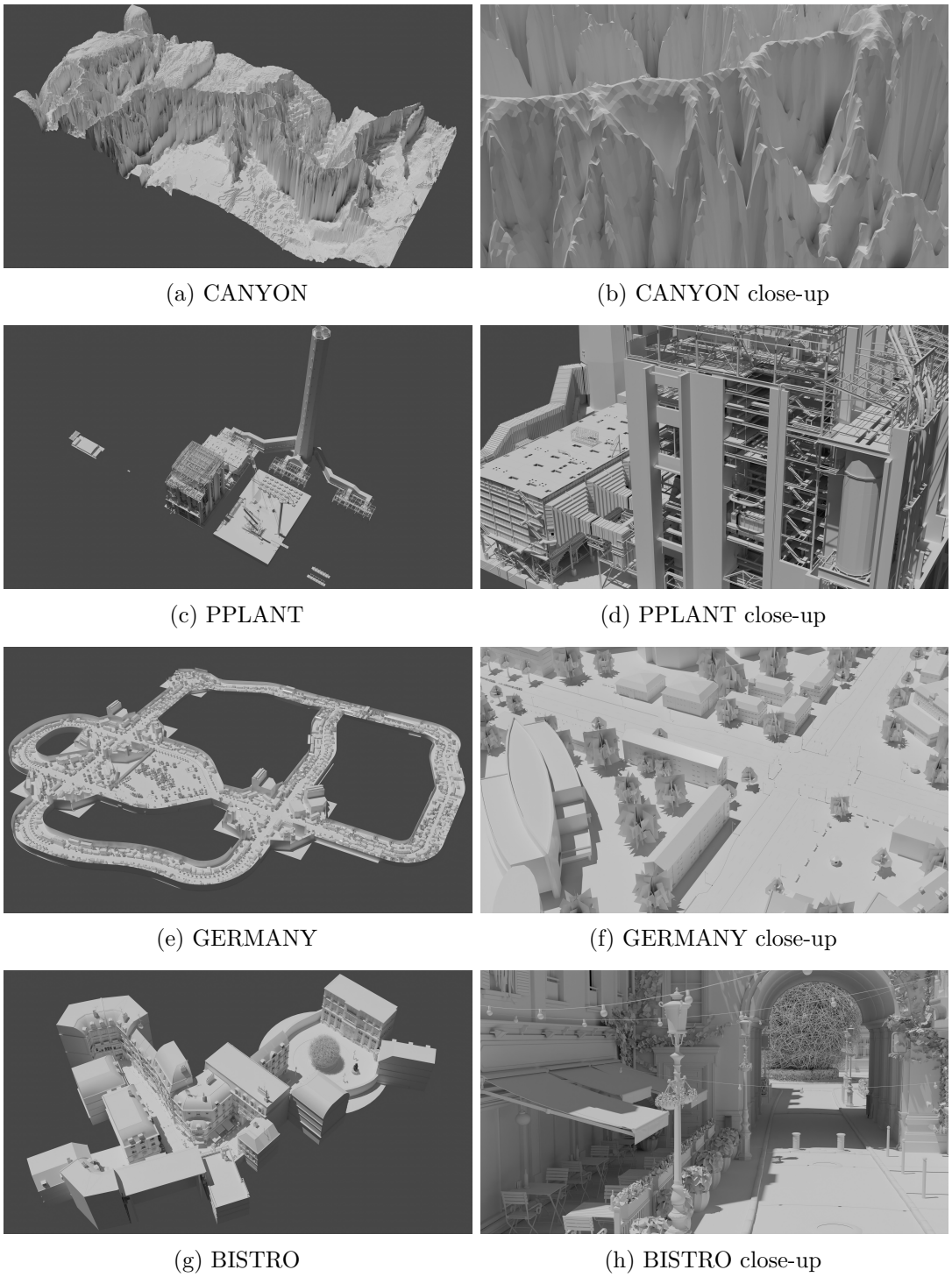
Table 7.1: Statistics of the scenes used for the evaluation.

in Section 7.3 and for the rendering performance impact measurements in Section 7.4, an RTX 2080 GPU was used.

7.1 Results Overview

In this section, GVS++ is compared to other from-region visibility sampling algorithms on four different scenes. GVS++ is compared to the original CPU-based implementation of GVS that uses MLRTA [RSH05] for ray tracing (*CPU-GVS)*, and a GPU-based implementation that uses hardware-accelerated ray tracing via the Vulkan API for the visibility sampling (*GPU-GVS*). Furthermore, we compare our approach to a brute-force random-sampling algorithm (RAND) and a rasterization-based visibility sampling algorithm RASTER. RAND is based on the implementation of the GVS++ algorithm but does not use any intelligent sampling mechanisms. RASTER is based on the visibility sampling algorithm by Nirenstein et al. [NB04]. In this approach, the scene is rendered from hemicubes that are intelligently distributed on the view cell. The IDs of the rendered primitives are gathered to populate the PVS. One notable difference between RASTER and GVS++ is that the runtime of the former approach inherently depends on the render resolution of the framebuffer that is used. On our four test scenes, however, we found that distributing hemicubes uniformly based on Halton points gives results that are on-par or better than when the adaptive placement is used. Therefore, RASTER places hemicubes uniformly on the view cell, according to Halton points. There are numerous techniques to render multiple views efficiently [UKS+20]. In our RASTER implementation, multi-view rendering is used, which is supported by Vulkan and allows rendering to multiple viewports simultaneously.

Pixel error and runtime measurements of the four algorithms on our scenes can be seen in Table 7.2. Average pixel error measurements over time for the PPLANT and CANYON scene are shown in Figure 7.2. GVS++ produces the most accurate PVS and achieves the lowest average and maximum pixel errors on every scene among all of the tested algorithms. As seen in Table 7.2, the largest connected error region is, on average, 46% smaller than the maximum reported pixel error. This shows that in a given rendering, there are, on average, multiple smaller error regions instead of a single error region that



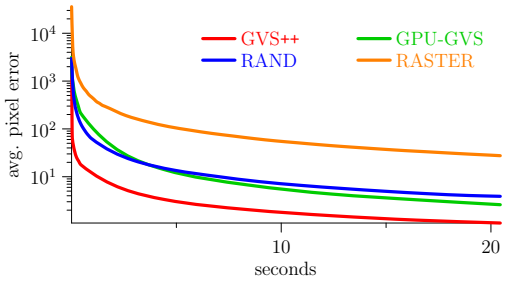
(a) CANYON (b) CANYON close-up

(c) PPLANT (d) PPLANT close-up

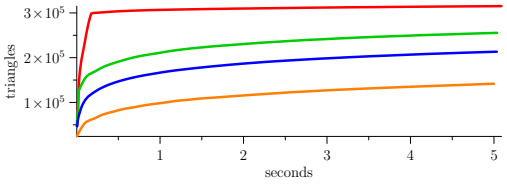
(e) GERMANY (f) GERMANY close-up

(g) BISTRO (h) BISTRO close-up

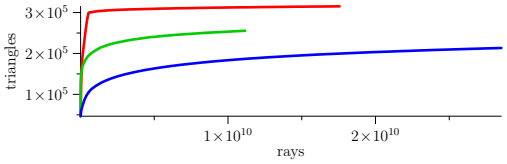
Figure 7.1: Scenes used for the evaluation.



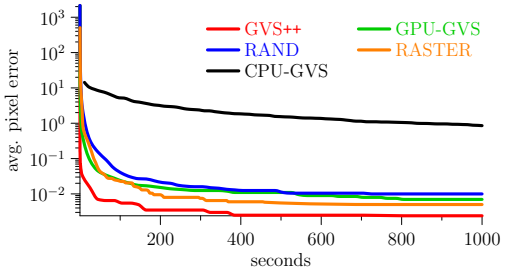
(a) Average pixel error over time on the PPLANT scene.



(b) Zoom-in of the PVS size over time for the first five seconds of each algorithm on the PPLANT scene.



(c) PVS size over the rays traced during the first five seconds of each algorithm on the PPLANT scene. GVS++ traces 57% more rays than GPU-GVS during the same time.



(d) Average pixel error over time on the CANYON scene.

Figure 7.2: Comparison of GVS++, GPU-GVS, CPU-GVS, RAND, and RASTER on the PPLANT (a-c) and CANYON (d) scene. A representative view cell is used for both scenes. GVS++, GPU-GVS, and CPU-GVS use the same parameters as in Table 7.2.

contains all of the incorrect pixels. We argue that a single large region is more noticeable than multiple smaller erroneous regions.

Comparing GVS++ to GVS shows that GVS++ achieves significantly lower average and maximum pixel errors. This result is not surprising since GVS++ uses more samples during the exploration phase and therefore samples neighborhoods and discontinuities more thoroughly than GVS. Figure 7.2 shows the difference between the two algorithms in more detail. It can be seen that GVS++ outperforms GPU-GVS. This is due to the new sampling schemes that allow for a better parallelization of the sampling procedure and therefore sample the scene more efficiently.

Our approach also outperforms the rasterization-based technique RASTER across all scenes. It should be noted that the times in Table 7.2 reported for RASTER only show the render times of the hemicubes to avoid distorting the result. The calculation time of RASTER inherently depends on the resolution of the framebuffer that is used and requires a dense placement of hemicubes on the view cell to avoid missing subpixel triangles. The latter is especially a problem in large scenes, where far away triangles are likely to cover less than a pixel.

Comparing GVS++ to the brute-force ray-tracing approach RAND shows the effectiveness of our intelligent sampling schemes, since RAND is based on the same implementation as GVS++ but does not use any intelligent sample placement. The time complexity of both algorithms, and the convergence of the PVS they produce, mostly depends on the complexity of the scene itself. Therefore, scenes with a high primitive count and highly occluded regions will lead to longer computation times than simpler scenes.

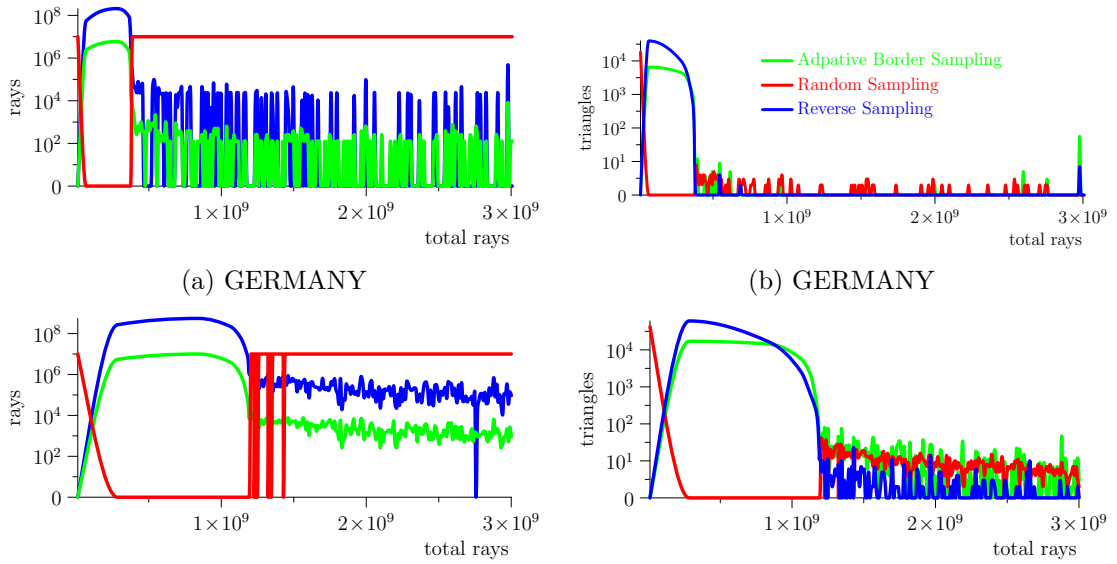
The performance uplift through hardware-accelerated ray tracing can be observed when comparing our GPU implementation of GVS, *GPU-GVS*, to the original CPU implementation, *CPU-GVS*, which uses MLRTA [RSH05] for ray tracing. GPU-GVS uses hardware-accelerated ray tracing via the Vulkan API. Figure 7.2 shows a significant speedup of the GVS algorithm when hardware-accelerated ray tracing is used (GPU-GVS). When comparing a single iteration of GVS, i.e., random sampling followed by an exploration phase, GPU-GVS is 63 times faster than CPU-GVS. Also, GVS++ is over four orders of magnitude faster than CPU-GVS to compute a PVS on the CANYON scene with comparable pixel errors. It is important to note that a direct runtime comparison is difficult since the CPU-GVS implementation is not watertight and therefore also finds triangles that are not visible. In contrast, our GPU-GVS implementation uses watertight RTX ray tracing. We found that, on the CANYON scene, the PVS of CPU-GVS is 4-6% larger after the first GVS iteration than the PVS of GPU-GVS when terminating the algorithm after 1000 seconds.

7.2 Asymptotic Behavior

In the following, the asymptotic behavior of the GVS++ algorithm is analyzed in terms of PVS size and pixel error. The same parameters for GVS++ were used for all view

	Algorithm	Avg. Err.	Max. Err.	Avg. Num. Err. Regions	Max. Err. Region Size	Calc. Time	PVS Size
CANYON	**GVS++**	**0.09**	**14**	**0.08**	**8**	**100ms**	**8.60%**
	GPU-GVS	29.15	507	11.08	64	100ms	8.39%
	RAND	234.79	5143	87.26	910	100ms	7.84%
	RASTER	2219.51	29052	249.24	1192	100ms	7.27%
PPLANT	**GVS++**	**2.85**	**77**	**2.37**	**55**	**1000ms**	**1.64%**
	GPU-GVS	21.98	296	13.58	100	1000ms	1.51%
	RAND	34.49	879	27.72	95	1000ms	0.93%
	RASTER	442.28	5110	243.43	3681	1000ms	0.54%
GERMANY	**GVS++**	**0.50**	**16**	**0.45**	**9**	**500ms**	**4.35%**
	GPU-GVS	2.87	150	2.51	23	500ms	4.03%
	RAND	19.56	667	15.11	146	500ms	2.84%
	RASTER	84.74	3856	44.08	714	500ms	2.24%
BISTRO	**GVS++**	**5.39**	**84**	**4.89**	**27**	**2000ms**	**6.82%**
	GPU-GVS	36.61	643	21.54	53	2000ms	5.94%
	RAND	84.91	1924	67.86	169	2000ms	5.23%
	RASTER	41.32	1803	34.33	652	2000ms	5.56%

Table 7.2: Statistics of different from-region visibility algorithms. The error measurements and the PVS size are averaged over ten view cells per scene. Errors are measured on 1000×1000 pixel renderings. For each scene, each algorithm had the same time budget. GVS++ and GVS used the same parameters for all view cells and scenes: $\Delta_{\text{ABS}} = 0.001$, and $n_{\text{rand}} = 10\,000\,000$. Furthermore, $n_{\text{ABSEdge}} = n_{\text{RSEdge}} = n_{\text{RSArea}} = 20$ was set for GVS++. Also, GVS recursively subdivided an edge of a border polygon up to three times. The column for the calculation time shows times for GVS, GVS++, and RAND without the time to generate Halton sequences. Calculation times for RASTER solely show render times of the hemicubes. GVS++ outperforms the other algorithms, giving a low average and maximum pixel error across all scenes.



(a) GERMANY (b) GERMANY

(c) PPLANT (d) PPLANT

Figure 7.3: Logarithmic scales of the random rays (red), ABS rays (blue), and RS rays (green) (a, c), and of the triangles found by random sampling (red), ABS (blue), and RS (green) over the number of total traced rays.

cells across all scenes: $n_{rand} = 10\,000\,000$, $n_{ABSEdge} = 40$, $n_{RSEdge} = 40$, $n_{RSArea} = 40$, $\Delta_{ABS} = 0.001$ and $T = 10$. This simplifies comparing the behavior of the algorithm across multiple scenes.

Examining the behavior of the algorithm in more detail (Figure 7.3), it can be seen that most triangles are found early on—that is during the initial random sampling and the subsequent exploration phase. 99.5% of the triangles of the resulting PVS are found during the initial random sampling and the subsequent exploration phase. After that, the PVS is already highly converged for most scenes. Examining Figure 7.3, it can be seen that after the first execution of the exploration phase, triangles found by the ABS and RS shader drops sharply, which also shows the high convergence of the PVS. The number of newly found triangles and the number of traced rays is then dominated by the random sampling algorithm. After the first exploration phase, most (60.2%) of the new triangles are found by random sampling, followed by ABS (35.6%) and RS (4.2%). This indicates that mostly smaller, disconnected regions of triangles or regions that are only visible from a fraction of the view cell are missed during the first random sampling and exploration phase and are found afterwards by repeated random sampling.

The average and maximum pixel errors are shown in Figure 7.5. Comparing the pixel error graphs to the graphs showing the PVS size over traced rays, it can be seen that the average and the maximum error correlate to the PVS size. Also, the average and the maximum pixel error behave very similarly for all scenes and view cells. When comparing

the graphed errors across the scenes, it can be seen that in the PPLANT, GERMANY, and BISTRO scene, the decline of the average pixel error starts to stagnate within the first third of the graph, before decreasing more rapidly again. The fact that the maximum pixel error of the same view cells shows a very similar behavior indicates that the error is likely to be caused by regions that are "hard to find", i.e., regions that are only visible from a small portion of the view cell. This claim is supported by the graphs for the PPLANT scene in Figure 7.3c and 7.3d. The graphs show that around 1.2×10^9 rays are traced during the initial random sampling and exploration. This is also the point where the pixel error starts to stagnate. The error declines faster again after the first exploration phase, when the scene is repeatedly random sampled. Comparing the average pixel error (Figure 7.5) to the average number of connected error regions (Figure 7.6), it can be seen that they correlate strongly. This indicates that the size of a connected error region is not significantly larger than a few pixels. The graphs also show that the maximum size of an error region is smaller than the maximum pixel error, especially in more converged stages. This shows that the pixel error is caused by multiple smaller regions instead of a single, more noticeable error region.

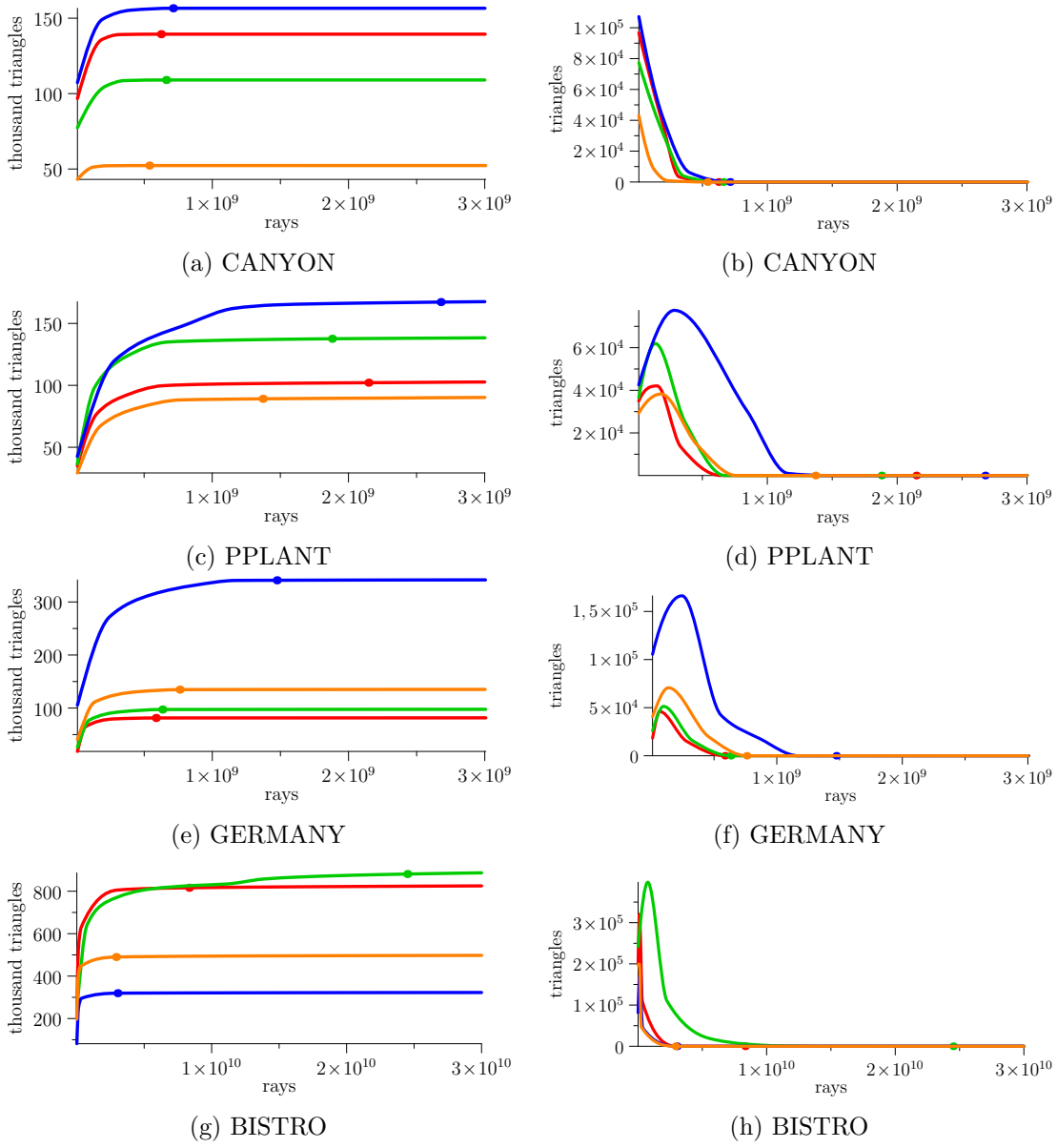
7.2.1 Comparison to Random Sampling

In the following, the asymptotic behavior of the GVS++ algorithm is compared to RAND. Figure 7.7 shows the PVS size and found triangles in comparison to GVS++. The overall behavior in terms of PVS growth is similar. Both the GVS++ algorithm as well as brute-force random sampling show logarithmic growth of the PVS. Right at the beginning, RAND finds more triangles using fewer rays compared to GVS++. This is due to the more uniform distribution of the samples. However, GVS++ converges quicker on all view cells and scenes. The quick convergence is due to the exploration phase. As seen in the right column of Figure 7.7, while brute-force random sampling finds new triangles quickly in the beginning, already intersected triangles are found repeatedly in subsequent iterations, causing the rate of finding new triangles to drop sharply. Due to the intelligent sampling strategies (ABS and RS) of GVS++, new triangles are more likely to be found. Thus a better convergence rate is achieved.

The average and maximum pixel error of both approaches are shown in Figure 7.8. It can be seen that the measured pixel errors decrease faster across all scenes and view cells when GVS++ is used, resulting in a pixel error that is several orders of magnitude lower for the same number of traced rays. The error for the green view cell of the BISTRO scene decreases quicker for the first third of the traced rays when the scene is randomly sampled. As previously discussed (Section 7.2), this is due to regions that are only visible from a fraction of the view cell and are only sampled after the first exploration phase.

7.3 Parameter Analysis

In the implementation of GVS++, various parameters are exposed to set the number of samples used for random sampling, adaptive border sampling, and reverse sampling.



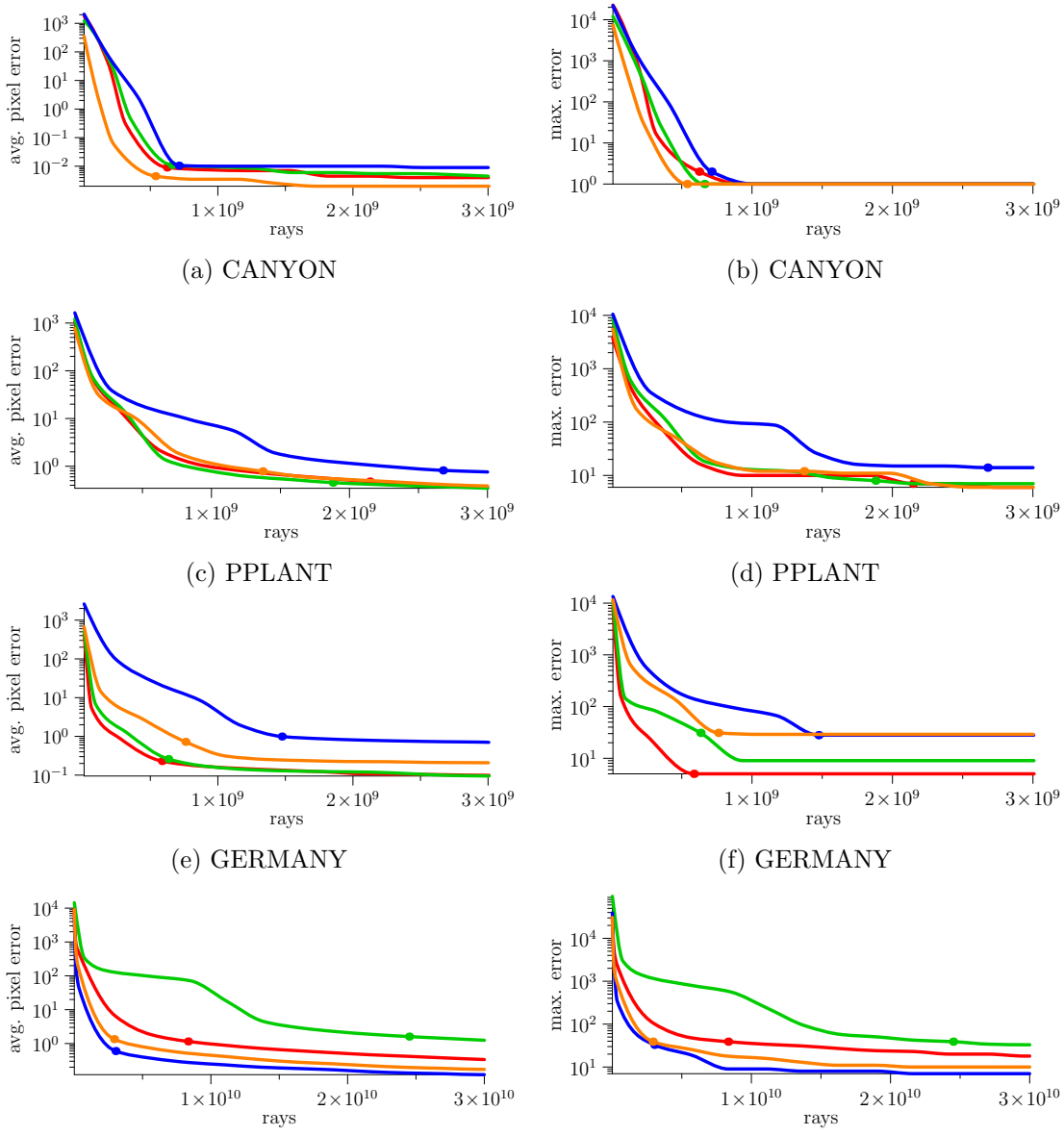
(a) CANYON (b) CANYON

(c) PPLANT (d) PPLANT

(e) GERMANY (f) GERMANY

(g) BISTRO (h) BISTRO

Figure 7.4: PVS size (left column) and found triangles (right column) over the number of traced rays for GVS++. Each line represents a view cell. A dot represents an example termination criterion of 10.



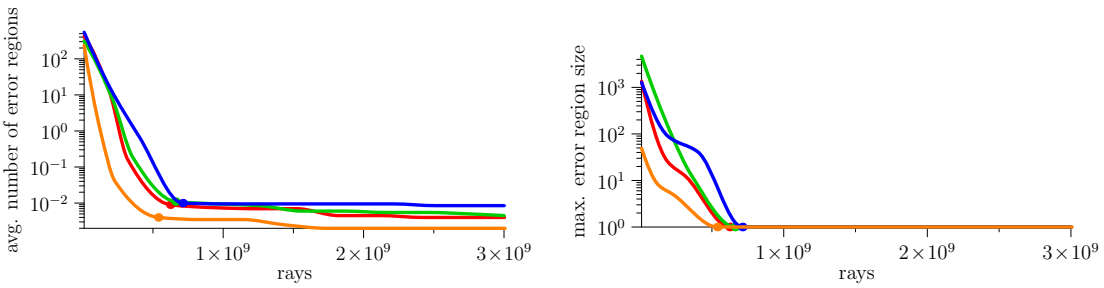
(a) CANYON (b) CANYON

(c) PPLANT (d) PPLANT

(e) GERMANY (f) GERMANY

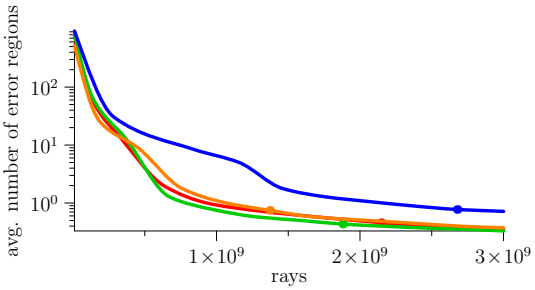
(g) BISTRO (h) BISTRO

Figure 7.5: Average (left column) and maximum pixel error (right column) over the number of traced rays for GVS++. Each line represents a view cell. A dot represents an example termination criterion of 10.

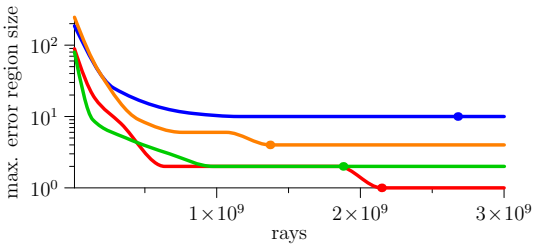


(a) CANYON

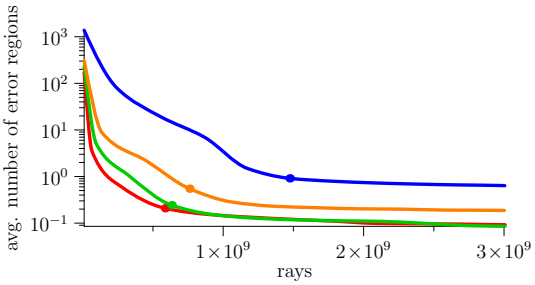
(b) CANYON



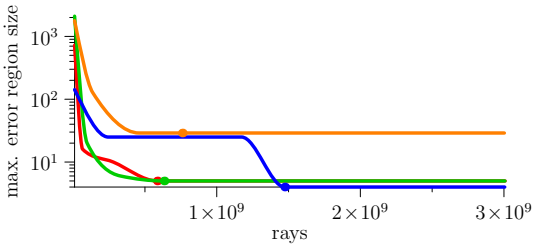
(c) PPLANT



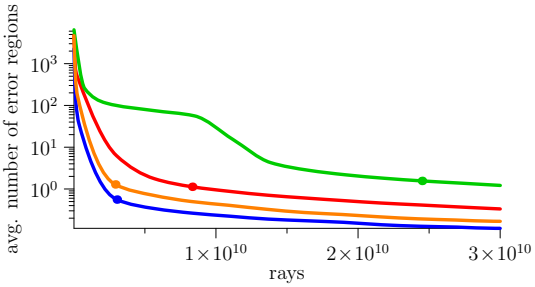
(d) PPLANT



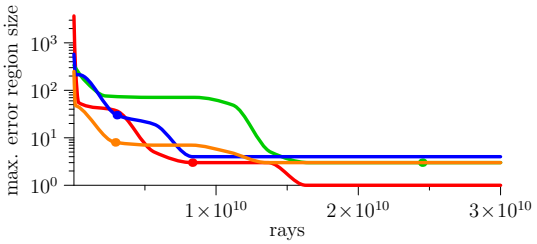
(e) GERMANY



(f) GERMANY



(g) BISTRO



(h) BISTRO

Figure 7.6: Average number of error regions (left column) and maximum error region size (right column) over the number of traced rays for GVS++. Each line represents a view cell. A dot represents an example termination criterion of 10.

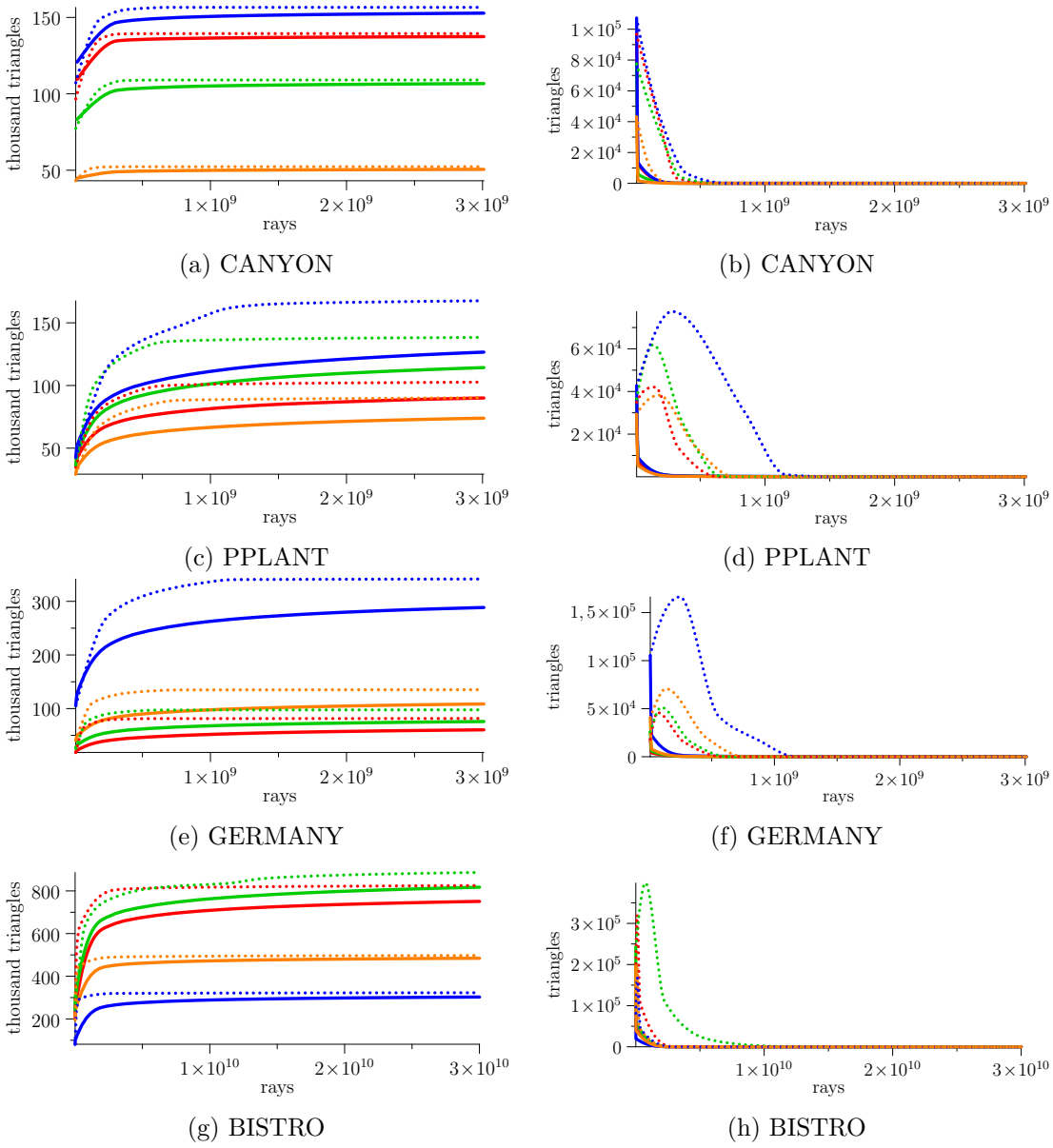
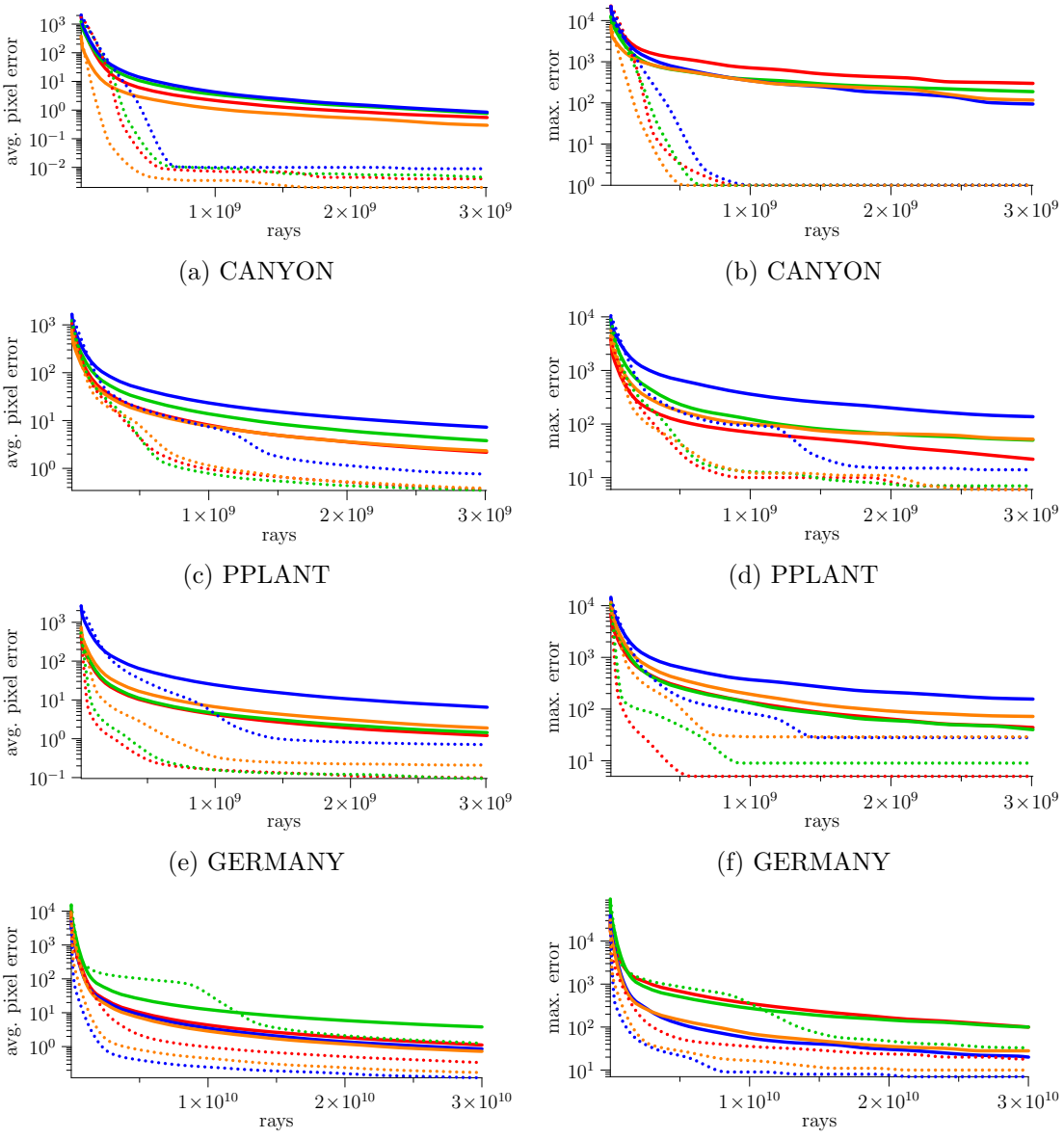


Figure 7.7: PVS size (left column) and found triangles (right column) over the number of traced rays for brute-force random sampling. Each line represents a view cell. Dotted lines represent the result of GVS++ for comparison.



(a) CANYON (b) CANYON

(c) PPLANT (d) PPLANT

(e) GERMANY (f) GERMANY

(g) BISTRO (h) BISTRO

Figure 7.8: Average (left column) and maximum pixel error (right column) over the number of traced rays for brute-force random sampling. Each line represents a view cell. Dotted lines represent the result of GVS++ for comparison.

In this section, the impact of different parameter choices on runtime and pixel error is analyzed. To gauge the impact on the behavior of GVS++, the algorithm is executed for the PPLANT and BISTRO scene with different random, ABS, and RS sample counts. The scene choice is motivated by the fact that for both scenes, the longest runtimes were reported in Table 7.2. Table 7.3 and 7.4 show the results. In the last two sections of the tables, sample counts are shifted from n_{RSEdge} to n_{RSArea}, while keeping the total RS sample count to 100, 120, 180, and 240, respectively. For each run, a termination criterion of 10 was used.

Both scenes show that with an increasing number of random samples, the measured pixel errors decrease while the calculation time increases. The average pixel error decreases almost linearly with increasing sample count. The largest sample count gave the lowest errors and the highest runtime. The high runtimes are justified by the fact that more random rays are more likely to find new triangles even in a fairly converged state. Also, the termination criterion is only checked *after* tracing n_{Rand} random rays.

In the second section of Table 7.3 and 7.4, the number of samples for exploring the neighborhood of a triangle (ABS) is varied. Increasing the number of ABS samples reduces the pixel errors and the algorithm's runtime until a point of diminishing returns is reached. For the PPLANT scene, the configurations with 40 and 80 ABS samples give a good combination of low pixel errors and runtimes.

The sample counts for the reverse sampling algorithm are varied in the last two sections of the tables. In section three, the number of samples along the view cell's edges is varied, while in the last section, the number of samples distributed on the view cell is varied. The corresponding lines of the two sections use the same total number of RS samples. It can be observed that increasing the number of total RS samples minimally lowers the average and maximum pixel error, while the runtime of GVS++ is not impacted significantly. Also, using more RS samples along the edges of the view cell instead of on the area, and vice versa, gave roughly the same results.

In summary, increasing the number of samples for the adaptive border sampling significantly lowers the average and maximum pixel errors while also lowering computation times. However, there is a point of diminishing returns when excessive sample numbers are used. Furthermore, increasing the number of RS samples on the area or on the edges of a view cell mostly helped lowering the maximum pixel error. This result shows that, for a given scene, these parameters can be used to fine-tune the behavior of the GVS++ algorithm.

7.4 Render Performance Impact

With the capabilities of current graphics hardware, the question arises, how much performance can be gained through occlusion culling. Table 7.5 shows frame times, i.e., time measurements of `vkCmdDraw`, with and without occlusion culling. It can be observed that occlusion culling reduces frame times by at least a factor of eight across all

Parameters				Avg. Err.	Max. Err.	Calc. Time [ms]	PVS Size
n_{Rand}	n_{ABSEdge}	n_{RSEdge}	n_{RSArea}				
500 000	20	20	20	11.67	93	228.1	0.90%
5 000 000	20	20	20	1.82	32	724.2	0.93%
50 000 000	20	20	20	0.28	22	5 187.2	0.98%
10 000 000	**5**	20	20	1.34	36	1 716.4	0.92%
10 000 000	**15**	20	20	1.08	31	1 352.0	0.94%
10 000 000	**40**	20	20	0.97	25	959.4	0.96%
10 000 000	**80**	20	20	0.80	19	961.9	0.97%
10 000 000	20	**5**	20	1.13	21	1 313.8	0.93%
10 000 000	20	**10**	20	1.10	20	1 145.8	0.94%
10 000 000	20	**25**	20	1.06	15	1 157.2	0.95%
10 000 000	20	**80**	20	0.94	16	1 283.3	0.95%
10 000 000	20	5	**20**	1.11	20	1 310.7	0.93%
10 000 000	20	5	**40**	1.01	21	1 361.1	0.94%
10 000 000	20	5	**100**	1.10	22	1 113.2	0.94%
10 000 000	20	5	**160**	1.01	19	1 228.5	0.95%

Table 7.3: GVS++ run with different parameter variations on the PPLANT scene. The highlighted rows show configurations that offer a good combination of pixel error and runtime.

Parameters				Avg. Err.	Max. Err.	Calc. Time [ms]	PVS Size
n_{Rand}	n_{ABSEdge}	n_{RSEdge}	n_{RSArea}				
500 000	20	20	20	22.05	284	1 555.3	10.44%
5 000 000	20	20	20	3.40	82	5 760.5	10.76%
50 000 000	20	20	20	0.43	21	48 731.9	11.16%
10 000 000	**5**	20	20	2.11	60	13 417.7	10.72%
10 000 000	**15**	20	20	1.95	52	10 728.6	10.84%
10 000 000	**40**	20	20	1.72	58	10 504.2	10.96%
10 000 000	**80**	20	20	1.52	40	12 232.7	11.07%
10 000 000	20	**5**	20	1.87	60	11 445.5	10.75%
10 000 000	20	**10**	20	1.88	55	11 129.4	10.81%
10 000 000	20	**25**	20	1.80	53	10 722.1	10.90%
10 000 000	20	**40**	20	1.82	54	10 331.7	10.94%
10 000 000	20	5	**20**	1.89	67	11 516.5	10.77%
10 000 000	20	5	**40**	1.88	44	10 830.8	10.78%
10 000 000	20	5	**100**	1.75	54	11 452.8	10.83%
10 000 000	20	5	**160**	1.55	52	12 889.8	10.89%

Table 7.4: GVS++ run with different parameter variations on the BISTRO scene. The highlighted row shows a configuration that offers a good combination of pixel error and runtime.

scenes. Frame times were reduced significantly in the PPLANT scene. In summary, it can be said that occlusion culling significantly reduces frame times for all tested scenes, especially for heavily occluded scenes consisting of millions of triangles.

	Avg. Frame Times [ms]			
	Without Occ. Culling	**With Occ. Culling**	**Speedup**	**Avg. PVS Size**
CANYON	1.72	0.21	x8.09	5.09%
PPLANT	9.63	0.18	x53.50	0.94%
GERMANY	2.81	0.29	x9.77	4.40%
BISTRO	4.36	0.44	x9.91	10.90%

Table 7.5: Frame times (`vkCmdDraw`) with and without occlusion culling.

CHAPTER 8

Use Case

In this chapter, a practical use case scenario of GVS++ is described, showing how it could be used in an application, such as a driving-simulation application. The idea is that especially applications that use highly populated models, such as city scenes, where a significant amount of primitives are occluded from any given position, could benefit the most from occlusion culling. In the following, it is assumed that such an application is given and that the viewer's possible locations, e.g., a car, are along streets. In a dense city, naturally, from the relatively low viewpoint of a car, many primitives are occluded. In this chapter, the JAPAN model, a city scene ©VIRES Simulationstechnologie GmbH, consisting of 4,483,926 triangles, is used.

8.1 Application Overview

The application into which GVS++ is implemented is responsible for initializing the Vulkan graphics API and loading and providing all the necessary inputs for GVS++. The algorithm consumes various parameters that may be read from a simple settings file or a database. Parameters that modify the behavior of the algorithm are listed in Section 6.2.2. Also, a scene and one or more view cells have to be provided for which GVS++ computes a PVS. The specific format in which the scene's model is stored is not relevant. The loaded model is used to build the acceleration structure that is used for ray tracing. In Vulkan, an acceleration structure is built from an index and vertex set of the loaded model. The view cells are defined by a position, size, and orientation, such as a normal vector.

8.2 GVS++ Usage

To compute the visibility along the streets of a scene, using GVS++ involves the following tasks:

1. Place 3D view cells along streets.
2. Run GVS++ to calculate the PVS for each view cell.
3. Store the PVS of each view cell for later use.
4. When rendering, load a stored PVS depending on the position of the viewpoint.

In the following sections, these tasks are discussed in more detail.

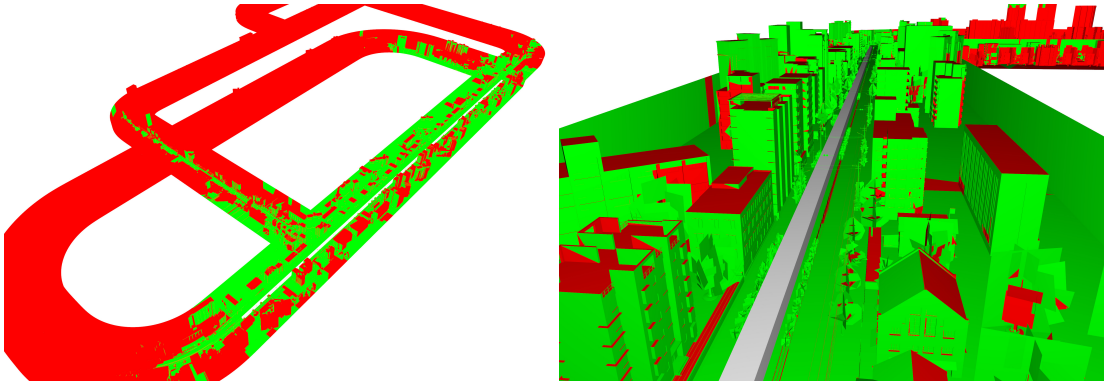
8.2.1 View cell placement

In this section, different strategies for view-cell placement strategies are discussed, each having advantages and disadvantages. Both strategies follow simple heuristics. More intelligent strategies [MBW06, MBWW07] that subdivide space driven by visibility information can be used instead (see Chapter 3).

The first strategy is motivated by the idea of placing as few view cells as possible. Therefore, each view cell covers a large part of the street. Straight parts of a street are covered by a single view cell. Figure 8.1 shows the result of this strategy on the JAPAN model. This strategy's main disadvantage is the relatively large size of the resulting PVS since a large view cell is also associated with a larger PVS. In this case, the PVS contains 13% of all triangles. This is unfavorable since the viewer might only visit a fraction of the view cell. A viewpoint might move along the street and take the first intersection on the JAPAN scene, as seen in Figure 8.1. Primitives visible from the other part of the view cell would be unnecessarily loaded. This problem can be alleviated by using multiple smaller view cells. Furthermore, a large part of the reverse sampling algorithm's mutated rays might not sample the detected discontinuity. This is due to the large distance of some mutated rays' ray origin to the point x on the view cell from which the discontinuity was detected. Such mutated rays are not "wasted", since other, not previously found primitives might be intersected. This problem could be alleviated by only using mutated rays whose origin is not farther than a given threshold to the point x. However, this is similar to using multiple smaller view cells in the first place.

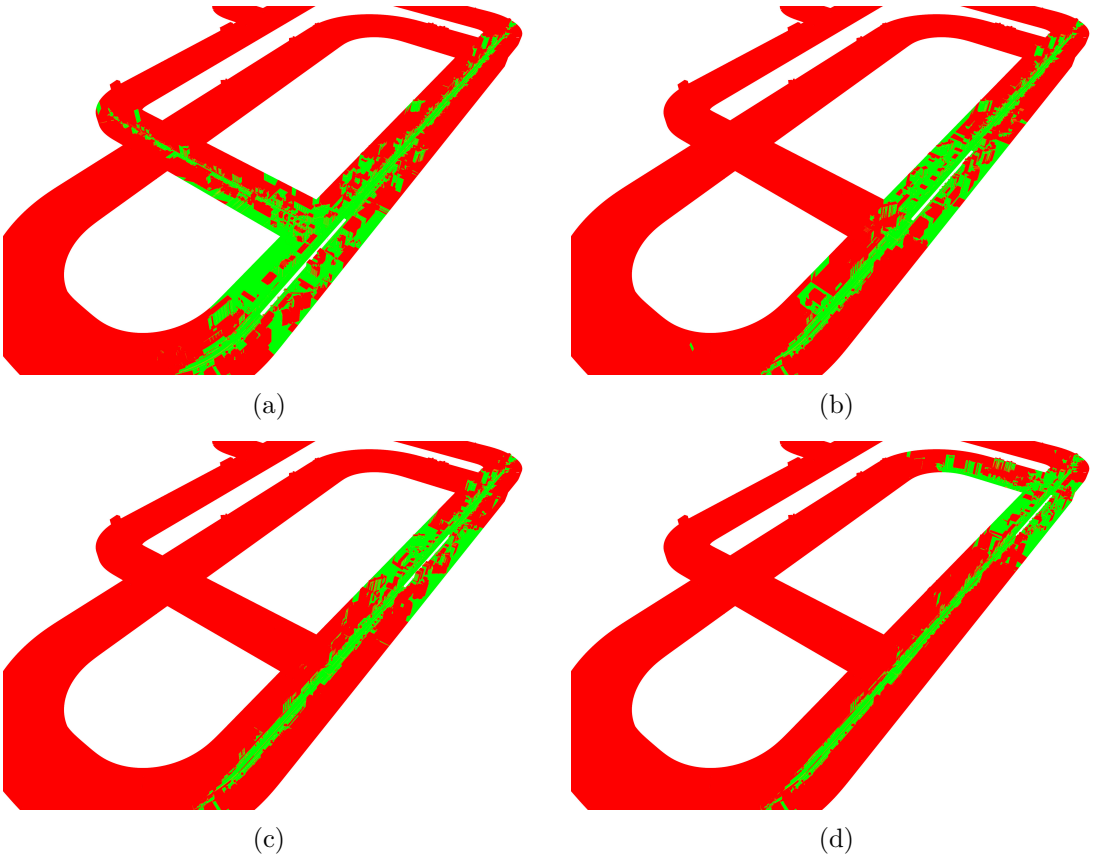
In the second strategy, multiple smaller view cells are used. The idea is that many smaller view cells better represent the path that a viewer might take. In our use case, instead of using a single view cell for the whole straight part of a street, multiple view cells are used. Such a view cell placement is shown in Figure 8.2. Unnecessarily small view cells along the street are also unfavorable. In such a case, the application would have to load and switch view cells frequently. Also, due to the overlap in terms of primitives of the PVS, more storage space is required. In this case, the PVS associated with the single large view cell contains 582 910 primitives, while the total number of primitives stored for the four smaller view cells is 1 158 202, which is almost double the number of primitives of the large view cell.

In conclusion, it can be said that the best strategy is a balance between the two strategies. As seen in later sections, both strategies require a similar number of total rays to get



(a) Overview of JAPAN. (b) Close-up of the view cell.

Figure 8.1: A single view cell is used to cover the whole straight part of a street. The resulting PVS contains 13% of all triangles.



(a) (b)

(c) (d)

Figure 8.2: Multiple view cells are used to cover the straight part of a street. The resulting PVS contain 5.8% (a), 6.1% (b), 6.4% (c) and 7.4% (d) of all triangles.

similar average and maximum pixel errors. However, in this scene, the multiple view cell strategy resulted in significantly higher rendering performance due to the smaller PVS. The goal should be to balance the size and the number of view cells such that the view cell covers large parts of the area a viewer visits while keeping the number of view cell switches, storage requirements, and samples to a minimum.

8.2.2 Running GVS++

Once the model and the view cells are loaded, the visibility sampling process can be started, calculating the PVS of each view cell sequentially. GVS++ can either be run on a client or a server. The PVS is calculated off-screen since the algorithm does not write to any framebuffers that are meant to be presented to a display. Vulkan is well suited for applications that exclusively render off-screen and may also be used with devices that are only capable of processing compute workloads. Since each view cell is processed sequentially, the algorithm may be executed on multiple devices simultaneously, where each device would process a different scene or view cell.

8.2.3 Storing and Loading PVS

After a PVS is calculated, it is stored for later use, e.g., in a real-time rendering application. It may be stored in a file or database. The specific way a PVS should be stored mainly depends on the specific application. One way to store a PVS is to simply store the primitive IDs. When loading a PVS, the primitive IDs can then be used to index vertex and index buffers. Instead of storing primitive IDs, vertex data such as vertex positions, colors, and normals could directly be stored. Furthermore, the view cells have to be stored such that each PVS can be associated with its view cell. This is necessary since the PVS that should be used is determined by the view cell the viewpoint is located in. The fact that the PVS of adjacent view cells usually overlap in terms of visible primitives can be used to store the PVS in a compressed way by only storing the difference between two adjacent PVSs.

There are different strategies to load the PVS that are associated with the view cells of a scene. One strategy is to load all PVSs at the start of the application and keep them in memory. This allows fast switching between different PVS. Another strategy, which may be preferable for devices with very limited memory, is to only load the current PVS into memory. Depending on the size of the stored PVS, the PVS associated with view cells adjacent to the current view cell may be loaded as well. This way, loading times can be hidden. A combination of both strategies may also be used, where only the PVS of the view cells within a certain radius of the viewer is loaded.

Depending on the application, it may be necessary to load and store object-based PVSs instead of PVSs that store individual triangles. This saves memory but also introduces overdraw since whole objects are rendered.

8.3 Render Performance Impact

The impact on frame times of both view cell placement strategies is listed in Table 8.1. It can be seen that both strategies result in significant performance uplift. However, a larger performance increase can be observed when smaller view cells are used due to the smaller PVS size.

View Cell Strategy	Avg. Frame Times		Speedup	Avg. PVS Size
	Without PVS [ms]	With PVS [ms]		
Large view cell	3.43	0.46	x7.46	13.00%
Small view cells	3.43	0.24	x14.29	6.45%

Table 8.1: Frame times (`vkCmdDraw`) with and without occlusion culling for the two view cell placement strategies.

8.4 Asymptotic Behavior

In this section, the behavior of the GVS++ algorithm for the two view cell placement strategies (see Section 8.2.1) is analyzed and compared. The following parameters for the GVS++ algorithm were used for each view cell in Figure 8.2: $n_{rand} = 12\,000\,000$, $n_{ABSEdge} = 40$, $n_{RSEdge} = 30$, and $n_{RSArea} = 30$. For comparison reasons, the same parameters used for the large view cell in Figure 8.1. For the large view cell, the reverse sampling parameters scaled, such that the same number of reverse sampling points on the view cell per meter were used as for the smaller view cells: $n_{RSEdge} = 120$, and $n_{RSArea} = 120$. Also, the number of random samples scaled accordingly ($n_{rand} = 48\,000\,000$). Note that similar results may be achieved with fewer samples.

For both strategies, the PVS size and the newly found triangles are plotted over the number of traced rays in Figure 8.4. It can be seen that the asymptotic behavior across all view cells is similar. Most of the triangles are found in the early stages, where the exploration of triangle neighborhoods rapidly finds new triangles. Comparing the PVS size and the pixel error measurements over the number of traced rays (Figure 8.5), it can be seen that both, the average and the maximum pixel error, correlate to the size of the PVS. Terminating the PVS calculation for both strategies using the smaller termination thresholds (10 and 3) gives a similar average and maximum pixel error for both view cell placement strategies. Also, both strategies require a similar number of total traced rays (26×10^9). Larger termination thresholds may be used such that the algorithm terminates earlier. However, this can lead to large maximum pixel errors and may only be useful for generating a quick preview. For a production-ready PVS, the threshold may be set as low as possible to minimize the pixel error.

The largest remaining error region of the large view cell with a termination threshold of 10 can be seen in Figure 8.3. The error is caused by a heavily occluded region, which is

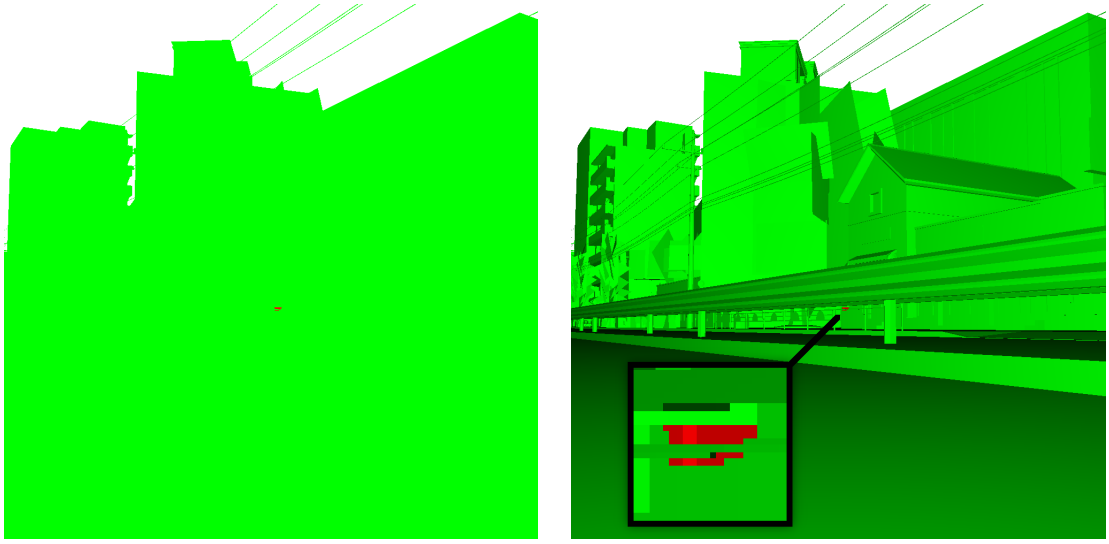
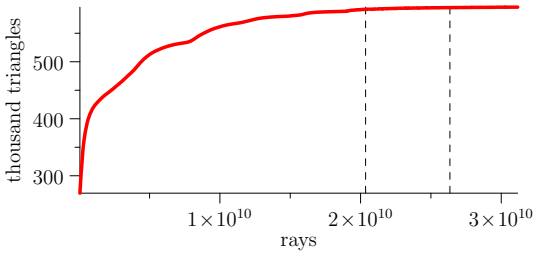
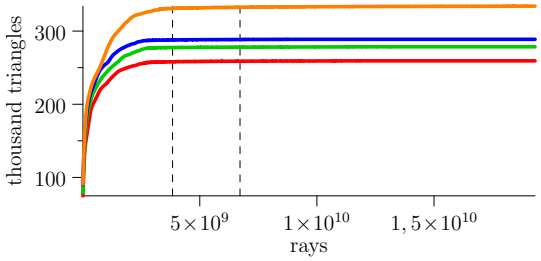


Figure 8.3: Largest error region of the first view cell placement strategy when a termination threshold of 10 is used.

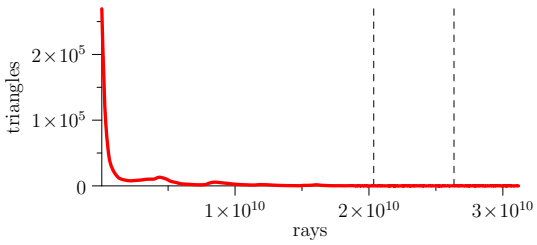
only visible from a very small area on the view cell. Such an error may be negligible in production-use, especially when the PVS is used to accelerate rendering. The error can further be reduced by using a smaller termination threshold.



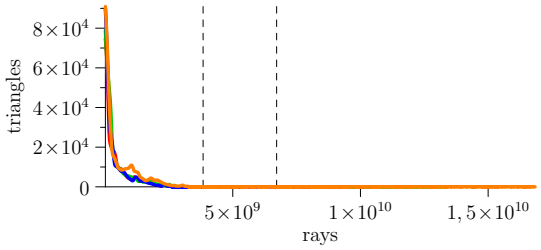
(a) PVS size of the single large view cell.



(b) PVS size of the of the four small view cells.

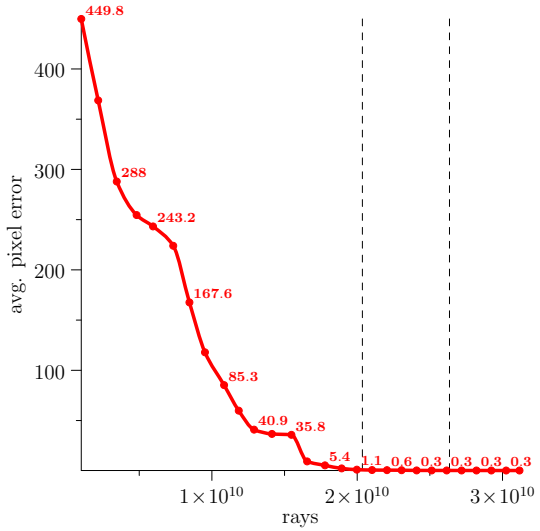


(c) New triangles found over million rays traced for the single large view cell.

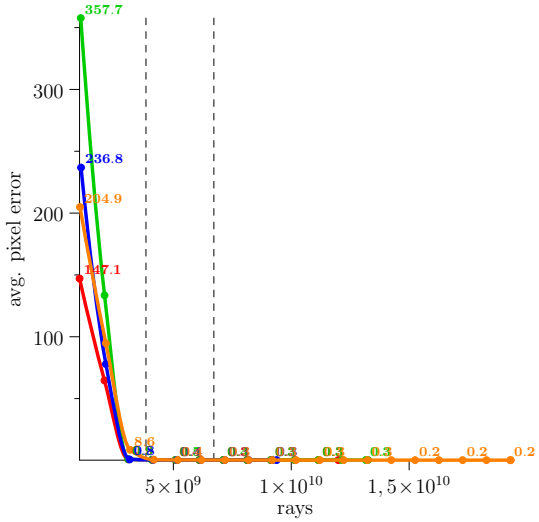


(d) New triangles found over million rays traced for the four small view cells.

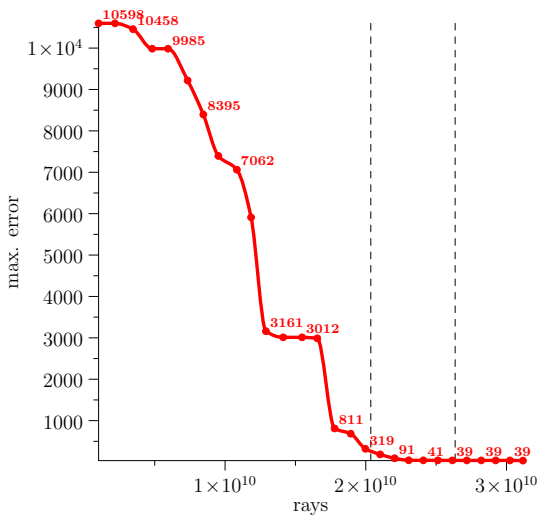
Figure 8.4: Asymptotic behavior of the PVS of the two view cell placement strategies. The vertical lines show possible thresholds (50 and 10 (a, c), 10 and 3 (b, d)) for the termination criterion.



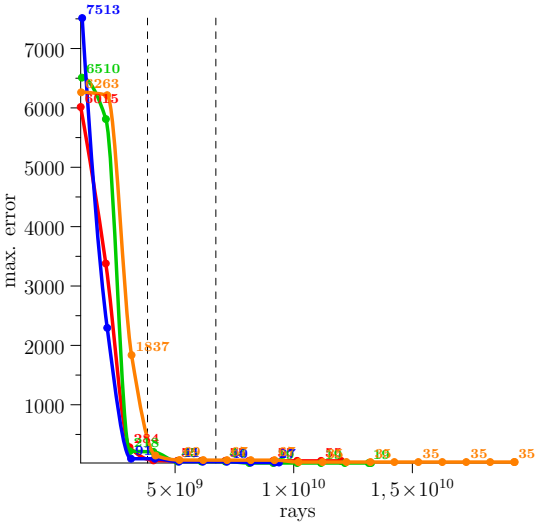
(a) Average pixel error of the single large view cell.



(b) Average pixel errors of the four small view cells.



(c) Maximum pixel error of the single large view cell.



(d) Maximum pixel errors of the four small view cells.

Figure 8.5: Pixel error measurements of the two view cell placement strategies.

CHAPTER 9

Conclusion and Future Work

In this final chapter, a summary of the proposed approach and the main contributions are given. Furthermore, limitations and opportunities for future work are discussed.

9.1 Summary

In this thesis, we presented an aggressive from-region visibility algorithm to compute the potentially visible set from a view cell in a general 3D scene. GVS++ calculates a more accurate solution in a shorter timespan compared to brute-force random sampling, GVS, and a comparable rasterization-based visibility sampling technique. The focus was on improving the intelligent sampling strategies to miss less triangles in edge cases. Our algorithms take the highly parallel nature and new features of modern GPUs, such as such as hardware-accelerated ray tracing, into account and allow for an efficient parallelization of the sampling procedure. The adaptive border sampling algorithm was improved to miss fewer triangles by sampling the neighborhood more thoroughly, and the reverse sampling algorithm was replaced by a different strategy that increases the likelihood of sampling discontinuities and unexplored regions.

The algorithm was implemented using the Vulkan graphics API. Hardware-accelerated ray tracing functionality is accessed through the ray tracing API of Vulkan. To this end, the main parts of the algorithm were implemented as GPU shaders. The presented implementation stores the PVS on the device in a buffer. An alternative implementation that uses a GPU hash set to store the PVS was discussed as well.

The presented reverse sampling algorithm allows GVS++ to efficiently handle narrow view cells. A practical use case where narrow three-dimensional view cells are placed along streets was presented. In a practical use case, the question of the best view cell placement arises. Therefore, two different strategies, where either a single large view cell is used or multiple smaller view cells that cover the same region, were discussed and

compared. Both strategies showed advantages and disadvantages, with the best strategy being a combination of both. Other practical problems concerning storing and loading a PVS were discussed as well.

Various scenes were used to evaluate the proposed algorithm. Results show that GVS++ is significantly faster than a comparable GPU-based implementation of GVS (GPU-GVS) and the original CPU-based implementation of GVS (CPU-GVS). On average, GVS++ is one order of magnitude faster than GPU-GVS. Furthermore, GVS++ is over four orders of magnitude faster than CPU-GVS. This result shows a significant performance uplift through hardware-accelerated ray tracing. Combined with the rapid development of recent products for hardware-accelerated ray tracing, sampling-based techniques are becoming more feasible, making it worth revisiting such approaches.

9.2 Limitations and Future Work

One of the main limitations of the proposed algorithm stems from the use of random sampling to find new seed points. The initial random sampling does find a good set of seed points. Once the first execution of the exploration phase does not find any more triangles, the PVS typically is already highly converged, and the algorithm proceeds with randomly sampling the scene. In such a highly converged stage, further triangles are mostly found through random sampling. Said triangles often are highly occluded and are only visible from a fraction of the view cell. Therefore, finding such regions may require a significant number of rays to be traced. This problem could be alleviated by employing an importance sampling approach similar to Ho et al. [HCCL12]. In the case of GVS++, initially, random sampling could still be used. In later, highly converged stages, a reliability function on the view cell boundary based on previously traced samples could be calculated. This function would then encode locations on the view cell from which highly occluded regions can be seen. The scene would then be sampled from these locations to improve the likelihood of finding missed triangles in later stages.

Another area of improvement may be the adaptive border sampling algorithm. Currently, the edges of a triangle's border polygon are sampled using a fixed application-defined number of samples. This could be suboptimal, especially in scenes that consist mostly of very large triangles. To avoid the need for fine-tuning GVS++ for such scenes, the adaptive border sampling algorithm could be changed to sample border polygon edges every x meters. This would alleviate the problem of possible large distances between samples in some cases.

List of Algorithms

Bibliography

[AMHH19] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. Crc Press, 2019.

[ARBJ90] John M Airey, John H Rohlf, and Frederick P Brooks Jr. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH computer graphics*, 24(2):41–50, 1990.

[Bha03] Chandra R Bhat. Simulation estimation of mixed discrete choice models using randomized and scrambled halton sequences. *Transportation Research Part B: Methodological*, 37(9):837–855, 2003.

[Bit02] Jirı Bittner. Hierarchical techniques for visibility computations. *Prague: Department of Computer Science and Engineering, Czech Technical University*, 2, 2002.

[BMW+09] Jiří Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. Adaptive global visibility sampling. *ACM Transactions on Graphics (TOG)*, 28(3):1–10, 2009.

[BW03] Jiří Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–755, 2003.

[BWW05] Jirí Bittner, Peter Wonka, and Michael Wimmer. Fast exact from-region visibility in urban scenes. In *Rendering Techniques*, pages 223–230, 2005.

[COCSD03] Daniel Cohen-Or, Yiorgos L Chrysanthou, Claudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[DDTP00] 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*, pages 239–248, 2000.

[DNL+17] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. Toward real-time ray tracing: A survey on hardware acceleration and

microarchitecture techniques. *ACM Computing Surveys (CSUR)*, 50(4):1–41, 2017.

[Eri04] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[Far] David Farrell. A Simple GPU Hash Table. `https://nosferalatu.com/SimpleGPUHashTable.html`. Last accessed 20. October 2020.

[FKP15] Henri Faure, Peter Kritzer, and Friedrich Pillichshammer. From van der corput to modern constructions of sequences for quasi-monte carlo rules. *Indagationes Mathematicae*, 26(5):760–822, 2015.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[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*, pages 231–238, 1993.

[Gla89] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[GSF99] Craig Gotsman, Oded Sudarsky, and Jeffrey A Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computers & Graphics*, 23(5):645–654, 1999.

[HCCL12] Tan-Chi Ho, Ying-I Chiu, Jung-Hong Chuang, and Wen-Chieh Lin. Aggressive region-based visibility computation using importance sampling. In *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry*, pages 119–126, 2012.

[HSS19] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. The camera offset space: real-time potentially visible set computations for streaming rendering. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019.

[LK10] Samuli Laine and Tero Karras. Two methods for fast ray-cast ambient occlusion. In *Computer Graphics Forum*, volume 29, pages 1325–1333. Wiley Online Library, 2010.

[LSCO03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. In *ACM SIGGRAPH 2003 Papers*, pages 595–604. 2003.

[Lum17] Amazon Lumberyard. Amazon lumberyard bistro, open research content archive (orca), July 2017. `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`.

[MBW06] Oliver Mattausch, Jirí Bittner, and Michael Wimmer. Adaptive visibility-driven view cell construction. In *Rendering Techniques*, pages 195–205, 2006.

[MBWW07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. Optimized subdivisions for preprocessed visibility. In *Proceedings of Graphics Interface 2007*, pages 335–342, 2007.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[NB04] Shaun Nirenstein and Edwin Blake. Hardware accelerated visibility preprocessing using adaptive sampling. 2004.

[NBG02] Shaun Nirenstein, Edwin Blake, and James Gain. Exact from-region visibility culling. Eurographics, 2002.

[Nie92] Harald Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, 1992.

[NPP+11] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. T&i engine: traversal and intersection engine for hardware accelerated ray tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, pages 1–10, 2011.

[NVI18] NVIDIA Corporation. NVIDIA Turing GPU Architecture. Technical report, September 2018. `https://www.nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/`.

[NVI20] NVIDIA Corporation. NVIDIA Ampere GPU Architecture. Technical report, September 2020. `https://www.nvidia.com/en-us/geforce/news/rtx-30-series-ampere-architecture-whitepaper-download/`.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[RGD09] Karthik Ramani, Christiaan P Gribble, and Al Davis. Streamray: a stream filtering architecture for coherent ray tracing. *ACM SIGARCH Computer Architecture News*, 37(1):325–336, 2009.

[RSH05] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (TOG)*, 24(3):1176–1185, 2005.

[SDDS00] 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*, pages 229–238, 2000.

[Shi91] Peter S Shirley. *Physically based lighting calculations for computer graphics*. PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[Shi20] Peter Shirley. Ray tracing: The rest of your life, December 2020. `https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html`.

[SW18] Nuno Subtil and Eric Werness. RTX on Vulkan. SIGGRAPH, 2018.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor: a hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, 2002.

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 95–106, 2004.

[TS91] Seth J Teller and Carlo H Séquin. Visibility preprocessing for interactive walkthroughs. *ACM SIGGRAPH Computer Graphics*, 25(4):61–70, 1991.

[UKS+20] Johannes Unterguggenberger, Bernhard Kerbl, Markus Steinberger, Dieter Schmalstieg, and Michael Wimmer. Fast multi-view rendering for real-time applications. 2020.

[WBW13] Sven Woop, Carsten Benthin, and Ingo Wald. Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82, 2013.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[WLH97] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with hammersley and halton points. *Journal of graphics tools*, 2(2):9–24, 1997.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)*, 24(3):434–444, 2005.

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Eurographics Workshop on Rendering Techniques*, pages 71–82. Springer, 2000.

[WWZ+06] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. Guided visibility sampling. *ACM Transactions on Graphics (TOG)*, 25(3):494–502, 2006.

Scene Data

CANYON View Cells		
Position (x y z)	**Rotation [deg]**	**Size (x y z)**
4998.78 2120.44 1600	0 90 0	100 100 0
3872.56 2027.55 -703.876	179.401 -101.599 0	200 40 0
8736.88 2757.98 -349.664	26.4636 -160.693 0	200 200 0
6598.41 2195.44 2997.27	0 180 0	100 80 0
6548.41 2692.12 3593.96	180 0 0	100 80 0
5752.93 2716.2 3035.93	53.7863 87.7553 0	100 80 0
5248.77 2045.44 2747.72	0 180 0	200 50 0
5048.79 2345.44 3745.9	0 180 0	100 30 0
2699.39 2045.44 2747.72	0 180 0	200 100 0
4010.12 2991.36 2720.37	162.2 52.7259 0	100 80 0

Table 1: CANYON view cells used in Chapter 7.

BISTRO View Cells		
Position (x y z)	**Rotation [deg]**	**Size (x y z)**
67.0595 8.58343 56.0716	7.06132 220.1035 0	3 2 0
67.0595 8.58343 56.0716	7.06132 41.1035 0	3 2 0
-11.1363 25.1356 0.38519	13.6135 45.6555 0	5 5 0
-11.1363 16.1356 0.38519	13.6135 45.6555 0	5 5 0
-14.8352 18.7713 -6.90301	2.04186 39.7178 0	6 6 0
2.19484 18.1663 -27.4158	2.1989 126.2234 0	4 7 0
4.30133 37.9113 -75.8382	58.5697 64.917 0	10 10 0
-4.66192 13.9506 8.5468	24.6563 64.174 0	3 1 0
76.8341 18.0458 61.6924	32.5462 221.6245 0	5 1 0
47.9899 6.66673 31.0112	177.8 70.415 0	1 1 0

Table 2: BISTRO view cells used in Chapter 7.

GERMANY View Cells		
Position (x y z)	**Rotation [deg]**	**Size (x y z)**
-151.024 0.2 106.585144	0 90 0	5 4 0
-151.024 0.2 106.585144	0 98 0	5 4 0
-65.7659 16.6493 -306.949	10.8123 267.8159 0	10 9 0
-59.1915 7.95323 103.275	-0.785388 89.8 0	30 4 0
-344.3 28.5307 103.343	17.1728 14.0529 0	40 50 0
-151.024 0.2 106.585144	10 20 0	5 4 0
-344.3 48.5307 103.343	17.1728 14.0529 0	40 50 0
-344.3 28.5307 103.343	17.1728 14.0529 0	40 50 0
-349.865 14.7674 101.32	14.3893 11.2484 0	10 6 0
-59.1915 27.95323 103.275	-0.785388 89.8 0	30 4 0

Table 3: GERMANY view cells used in Chapter 7.

PPLANT View Cells		
Position (x y z)	**Rotation [deg]**	**Size (x y z)**
-175.305 64.4371 122.255	10.0322 64.8687 0	70 20 0
-191.334 62.8204 25.8898	21.0101 83.0 0	20 40 0
-68.208 30.3286 -25.0804	6.74809 34.0 0	40 40 0
83.2512 28.115 12.9271	8.00049 258.9445 0	30 20 0
-26.2082 52.6414 78.7562	17.9431 47.0044 0	35 25 0
94.906 75.8755 187.143	21.773 240.0173 0	40 20 0
-42.6588867 39.8894287 95.0234497	0 180 0	20 20 0
-72.6588867 139.8894287 95.0234497	0 -45 0	50 50 0
-72.6588867 39.8894287 95,0234497	0 180 0	50 50 0
-42.6588867 70.00894287 95.0234497	-90 0 0	20 20 0

Table 4: PPLANT view cells used in Chapter 7.

Scene	View Cell			Used For
	Position (x y z)	**Rotation [deg]**	**Size (x y z)**	
JAPAN	312.102 0.151144 -930.0	0 0 0	2.5 2 600	View cell placement strategy one
	312.102 0.151144 -930.0	0 0 0	2.5 2 150	View cell placement strategy two
	312.102 0.151144 -780.0	0 0 0	2.5 2 150	
	312.102 0.151144 -630.0	0 0 0	2.5 2 150	
	312.102 0.151144 -480.0	0 0 0	2.5 2 150	

Table 5: View cells used in Chapter 8.