Joerg H. Mueller

Shading Atlas Rendering for Virtual Reality

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg
Institute for Computer Graphics and Vision, Graz University of Technology

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger, BSc
Institute for Computer Graphics and Vision, Graz University of Technology

Referee

Prof. Dipl.-Inform. Dr.-Ing. Tobias Ritschel
Computer Science Department, University College London

Graz, Austria, Jan. 2021

Abstract

The demands for modern real-time rendering applications are tremendous, especially with virtual reality head-mounted displays. The computational effort required to render at an ever increasing resolution and frame rate are further amplified by a demand for more realistic and higher quality imagery. Since improvements in the processing power of modern graphics processing units cannot keep up with this demand, modern rendering paradigms are dropping the paradigm to compute every frame mostly from scratch. Instead, techniques for the temporal amortization of computations are gaining in popularity. However, many of these techniques suffer from the image-space storage of rendering data, which limits their performance gains. Object-space storage of shading data can overcome the limitations of image-space methods, but it lacks the required hardware support to become practical.

In this thesis, we demonstrate that the shading atlas, a texture-space storage of shading information, provides a superior caching space for shading information. The shading atlas stores shading information within a single texture. This is achieved by shading geometric primitives into rectangular blocks within the atlas. Atlas memory management runs fully parallel on the GPU with little fragmentation. Sampling from the cache utilizes the hardware acceleration for texture filtering. Thus, it requires no additional GPU extensions. Rendering with the shading atlas is comprised of multiple stages that can be decoupled and thus allow running them at different frequencies, depending on the application. The ultimate advantage of the shading atlas is the temporal coherence of its contents; once allocated, geometric primitives stay at the same location within the atlas, supporting the implementation of temporal amortization techniques.

We demonstrate that the shading atlas yields superior results in three application scenarios when compared to corresponding image-space methods. Firstly, the shading atlas supports spatial shading reuse, such as in spatial upsampling, stereo rendering and foveation, in a straightforward manner at a high quality. In the second application scenario, we explore temporal reuse of shading information in more detail. We present the temporally adaptive shading framework, which determines when shading can be reused or needs to be updated. We find that the texture-space caching within the atlas allows a much longer

reuse than with image-space methods, which suffer from accumulating reprojection errors. In streaming for virtual reality, we show that the shading atlas is better at hiding latency than state-of-the-art image warping methods. The design of the shading atlas is well suited for efficient MPEG compression, facilitating the first object-space streaming approach.

With this research, we demonstrate the utility of caching shading information in a texture-space representation. Many application scenarios can benefit from implementing the shading atlas, which allows focusing shading computation on the important parts of the scene. It is not limited to streaming for virtual reality but can benefit any real-time or even offline rendering application.

Kurzfassung

Die Anforderungen für moderne Echtzeit-Rendering-Anwendungen sind enorm, insbesondere für die Bilderzeugung für die virtuelle Realität in Head-Mounted-Displays. Der Rechenaufwand für das Rendern von realistischeren und qualitativ hochwertigeren Bildern mit immer höherer Auflösung und Bildrate steigt stetig. Da Verbesserungen der Rechenleistung moderner Grafikprozessoren dieser Nachfrage nicht gerecht werden können, lassen moderne Rendering-Methoden das Paradigma fallen, jeden Frame von Grund auf neu zu berechnen. Stattdessen werden Techniken zur zeitlichen Amortisation von Berechnungen immer beliebter, jedoch leiden viele dieser Techniken unter den Nachteilen der Speicherung von Daten im Bildraum. Die Speicherung von Shading im Objektraum kann diese Einschränkungen beseitigen, es fehlt jedoch meist Hardwareunterstützung, um praktikabel zu werden.

In dieser Dissertation zeigen wir, dass der Shading Atlas, ein Speicher für Shading im Texturraum, ein überlegener Zwischenspeicher für Shading-Informationen ist. Er benötigt nur eine einzelne Textur, da das Shading geometrischer Primitive in rechteckigen Blöcken innerhalb des Atlas gespeichert wird. Die Speicherverwaltung des Atlas läuft vollstandig parallel mit nur wenig Fragmentierung auf der GPU. Abfragen aus dem Zwischenspeicher verwenden die Hardwarebeschleunigung für Texturfilterung und benötigen somit keine zusätzlichen GPU-Erweiterungen. Das Rendern mit dem Shading Atlas erfolgt in mehreren Stufen, welche voneinander entkoppelt werden können und somit je nach Anwendungsfall mit unterschiedlichen Frequenzen ausgeführt werden können. Der größte Vorteil des Shading Atlas ist die zeitliche Kohärenz seines Inhalts; einmal reserviert, bleibt die Shading-Information an der gleichen Stelle im Atlas, was die Implementierung von zeitlichen Amortisierungstechniken erlaubt.

Wir zeigen, dass der Shading Atlas im Vergleich zu entsprechenden Bildraummethoden in drei Anwendungsfällen überlegene Ergebnisse liefert. Erstens unterstützt der Shading Atlas die räumliche Wiederverwendung von Shading, wie räumliches Abtastratenerhöhung, Stereo-Rendering und Foveation, auf unkomplizierte Weise und in hoher Qualität. Im zweiten Anwendungsfall untersuchen wir die zeitliche Wiederverwendung von Shading genauer. Wir präsentieren das Zeitlich-adaptive-Shading-System, welches bestimmt, wann

Shading wiederverwendet werden kann oder neu berechnet werden muss. Wir stellen fest, dass die Zwischenspeicherung der Shading-Ergebnisse im Texturraum des Shading Atlas eine viel längere Wiederverwendung ermöglicht als bei Bildraummethoden, welche über die Zeit immer mehr Reprojektionsfehler ansammeln. Beim Streaming für die virtuellen Realität zeigen wir, dass der Shading Atlas besser Latenz zu verbergen kann als moderne Image-Warping-Methoden. Das Design des Shading Atlas eignet sich hervorragend für effiziente MPEG-Komprimierung, was uns die Entwicklung des ersten objektraumbasierten Streaming-Ansatz ermöglicht.

Mit dieser Forschung demonstrieren wir die Nützlichkeit, Shading-Informationen in einer Texturraumdarstellung zwischenzuspeichern. Viele Anwendungsfälle können von der Implementierung des Shading Atlas profitieren, welche es erlaubt die vorhandenen Ressourcen auf die wichtigen Teile der Szene zu konzentrieren. Der Einsatz des Shading Atlas ist nicht auf Streaming der virtuellen Realität beschränkt, sondern kann jeder Echtzeit- oder auch Offline-Rendering-Anwendung zugute kommen.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Acknowledgments

I want to express my utmost gratitude for everyone who took part in my journey whether mentioned namely or not. First, I would like to thank my supervisor Dieter Schmalstieg, who kept me on my toes all the time while giving me the time to explore even paths that would be dead ends. He provides a constant flow of new and creative ideas, which hardly ever left me without a thing to try next. Complementing Dieter's qualities, I thank my co-supervisor Markus Steinberger, who excels with detailed technical knowledge and has an unstoppable drive for increasingly exceptional achievements. I want to thank Tobias Ritschel for taking his time to review this thesis. I am grateful for the help and support of my colleagues at the IGC: Thomas Neff, Philip Voglreiter, Martin Winter, Daniel Mlakar, Michael Kenzel, Bernhard Kerbl, Alexander Grabner, Karl Haubenwallner, Mathias Parger, Alexander Weinrauch, Georg Krispel, Lorenz Jäger and everyone else for the joint work, the cheerful lunches, the intense intellectual discussions, the games and the technical support. You all are not mere colleagues, but friends.

Next, I would like to thank all my friends who supported me through the easy and the hard times of PhD life. Whether it is the playing, the improvising, the swinging, getting airborne or some other kind of "sporty" activity. You know who you are and what I enjoy most about all these activities is to share them with friends.

Last, but not least I thank my family, foremost, Gabriela and Wolfgang Klösch for their unconditional support. I would especially like to thank my brother Thomas Müller who is the most caring brother anyone could wish for.

This work was supported by the Christian Doppler Laboratory for Semantic 3D Computer Vision, funded in part by Qualcomm Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Real-time rendering

Rendering is the process of image generation with a computer based on data that describes the contents of the image. The most commonly known applications of rendering are for movies and computer games, where the goal is to create an immersive visual experience. Other application areas of rendering can be more abstract, for example, moving from the virtual worlds of movies and computer games to the real world augmented by rendered information. Data visualization, which might be as simple as presenting some population in a bar chart, is also considered rendering. In fact, the whole field of computer graphics is concerned with the visualization of data, where the rendering of photo-realistic imagery can be considered a specialization.

The data that is visualized during rendering can assume different forms. For example, the data that is measured in magnetic resonance imaging (MRI) or computed tomography (CT) is typically stored as intensities within a regular, volumetric grid, where each volumetric element is called a voxel. Most commonly, however, a surface representation is used. The simplest surface representation is a triangle mesh, where each triangle is described by the three corner vertices. The triangles are assigned different surface materials to produce

a realistic look. Commonly, materials use different types of textures, i.e., images that are projected onto the surface, such as a color, transparency, normal, roughness and metallicity textures to model the real world behavior of the surfaces.

A practical classification of rendering is to differentiate between real-time rendering or online rendering in comparison to pre-rendering or offline rendering. The major difference between these two classes are the strict time constraints put on real-time rendering, which typically needs to produce multiple images per second. This is typically the case in any system that employs interactivity, such as computer games. Another classification of rendering techniques is whether they work in object or image order, i.e., whether they primarily iterate over the objects within the scene data or the pixels of the output image. Ray tracing based techniques are image order techniques that shoot a ray from the camera through each pixel of the output image. From the hit position, secondary rays are cast, for example, towards light sources to compute the direct illumination or randomly, such as in Monte Carlo based path tracing. These techniques can produce very realistic images, but are more expensive to compute. In contrast, rasterization projects the scene geometry to image space and draws each triangle pixel by pixel, while using a depth buffer to ensure that triangles in the back are not drawn over those in the front. This approach is fast and thus the typical choice for real-time rendering, especially since this is the type of rendering that hardware manufacturers have been primarily focused on for the past thirty years. Only recently hardware manufacturers focus on hardware acceleration for ray tracing, since processing power now allows at least partial real-time ray tracing. Unfortunately, the performance is still very limited, requiring advanced denoising and spatio-temporal filtering techniques to achieve acceptable image quality.

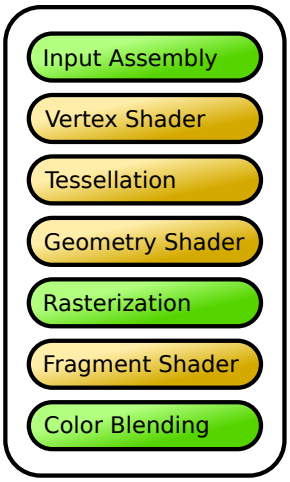


Figure 1.1: The stages of the hardware accelerated rendering pipeline are a mixture between configurable fixed function stages (in green) and programmable shader stages (in yellow). The tessellation and geometry shader stages that can alter the geometry to some extent are optional.

The hardware accelerated rendering pipeline is shown in Figure 1.1. With hardware improvements of the last two decades, real-time rendering transitioned from the fixed function pipeline to a more dynamic graphics pipeline with programmable shaders. First, the geometric primitives are loaded from the index and vertex buffers in the input assembly stage. Second, the vertex shader is run for every vertex of the geometry. Its main purpose is to transform the vertex positions from typically local mesh coordinates to screen space coordinates. After the vertex shader, optional tessellation and geometry shaders may alter the geometry to some limited extent. In the next stage, the rasterizer turns the transformed triangles into pixel by figuring out which of the screen's pixels are inside the triangle. The fragment shader then runs in parallel for every pixel and computes the shading. Finally, the color blending stage blends transparent pixels and writes the final color to the output framebuffer. By default, every frame is rendered from scratch, since there is no way to reuse previous shading results without modifications to this pipeline.

The simplicity of rasterization based rendering results in the necessity to approximate many visual effects with additional techniques. For example, shadows have to be rendered using either shadow mapping or shadow volumes, since no ray is cast to the light source that could detect possible occluders. Similar solutions are necessary for reflections such as in mirrors, or global illumination. Global illumination is sometimes partially approximated with ambient occlusion techniques. Ambient occlusion is the slight darkening that can be observed in corners or creases with little room between surfaces. Many of these techniques require preprocessing steps, such as the rendering of cube maps and shadow maps, or postprocessing steps, such as screen space ambient occlusion and screen space reflections. This reduces the load on the shading stage of the pipeline, but increases the complexity of the overall rendering pipeline. Real-time ray tracing can compute many of these effects more realistically and at a lower complexity, but requires more time. Thus, the more time can be spent on the shading computation, the higher the resulting quality.

1.2 Virtual reality

A head-mounted display (HMD) allows an even deeper immersion into virtual worlds. The term *virtual reality* (VR) is now frequently used synonymously with the use of such a device. Real-time rendering for VR puts some additional burden on the system. Since the screen of an HMD is closer to the user's eyes, the resolution needs to be higher, or the user will be able to see individual pixels or even the screen door effect, i.e., the black grid between pixels. The number of pixels is further doubled by having two screens, one for each eye, instead of just one requiring stereo rendering for two slightly different viewpoints.

Apart from a high resolution, another requirement of VR is to render at a high frame rate with low latency in order to avoid VR motion sickness. When a user wears an HMD and rotates their head, the image of the display needs to follow as quickly as possible. Otherwise, the sensory information reported by the vestibular system and the visual system conflict. The body reacts with a feeling of sickness and in the worst case leads to vomiting.

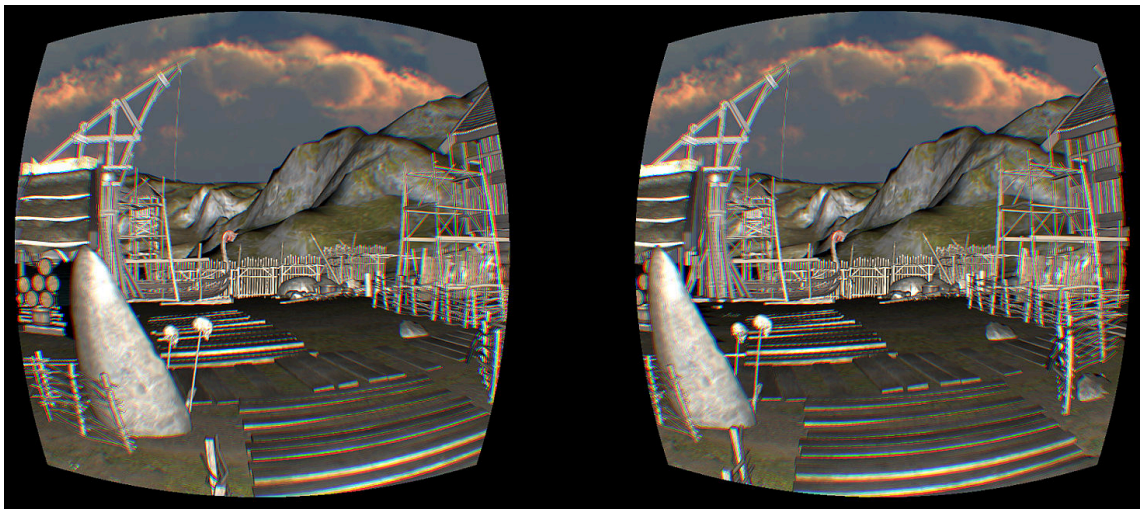


Figure 1.2: A head-mounted display requires rendering two views, one for each eye. The distortion and chromatic aberration is necessary to counteract the effects of the lenses inside the headset.

Thus, VR sickness has to be avoided at all costs. In contrast to a standard monitor which runs with a 60 Hz refresh rate, rendering for VR is typically done at 90 Hz to 120 Hz. Ideally, the image shown on an HMD would be updated immediately after the user moves. However, the latency depends on how long the system takes to detect a pose change, render a new view based on the new pose and displays it on the screen. This total time determines the motion-to-photon latency, which should be minimized in a VR system to avoid motion sickness.

Finally, another important aspect to be considered for VR rendering is mobility. Users are not sitting in front of a screen, but can ideally move around freely. Unfortunately, for high quality rendering, a powerful desktop computer with a state-of-the-art GPU is required, resulting in the user being limited in movement within the vicinity of the computer, likely, due to the cable of the HMD. Mobile all-in-one VR headsets, such as Oculus Quest, have much lower processing power, leading to a lower image quality and limited battery lifetime, which reduces possible application scenarios. A third alternative to tethered and all-in-one VR headsets is remote rendering. While a tethered network typically has a low latency, even powerful short range wireless networks add considerable latency. Completely free mobility would be granted with a WAN network, but those have even higher latency. This high latency is the reason why streaming so far is the least popular solution in VR systems.

Remote rendering systems can solve the issue of mobility and high quality rendering, but suffer from the added latency. The energy consumption can be reduced by offloading the performance hungry rendering to a rendering server, which can compute higher quality images. It also saves computations on the mobile client, increasing the battery lifetime. To

solve the latency issue, latency can be compensated in several ways. The pose estimation systems that use optical sensors for an accurate pose can be fused with a faster inertial measurement unit (IMU). Some of the latency can be compensated by predicting a future pose, but the further the prediction, the less accurate it will be. Another technique is late warping, which warps the rendered image with the most recent pose shortly before it is displayed. However, these techniques can only compensate short periods of time. A remote rendering system that involves network transfer has high latency that may not be compensated in this way.

Overall, there are three major requirements that conflict with each other: quality, latency and mobility. The frame rate and latency dictates an upper bound on the available computation time per frame and thus on the rendering quality. A powerful desktop computer can render at high quality, but either tethers the user to the computer or introduces high latency. While mobile, an embedded GPU in an all-in-one-device will not be able to deliver the high-end graphics required for VR in the foreseeable future, since it is limited by power budget, thermal restrictions and production cost. It appears that typical solutions focus on two of the three conflicting requirements, neglecting the third.

1.3 Improving rendering performance and quality

In modern graphics applications, shading typically constitutes a significant part of the workload. Due to the proliferation of complex shading models, such as physically-based rendering or real-time raytracing, this trend will certainly continue. Thus, efficient shading is vital for VR and for energy-efficient mobile graphics.

For better shading efficiency, the goal is to improve the throughput of a rendering system. The throughput depends on how many pixels can be rendered in a specific time. The target throughput depends on the frame rate and total resolution of the two displays. The most straightforward solution to increase the throughput is to increase processing powers, for example by parallelization, the major source of the higher efficiency of a GPU compared to a CPU. The target throughput determines the available time per pixel and thus limits the image quality. When the available processing time for pixels per frame is constant, one way to improve quality without reducing the throughput is to spend the available time on those pixels that have the biggest influence on image quality.

An important strategy to reduce shading load is to exploit spatial coherence. Stereo rendering, for example, can benefit from spatial reuse by reusing rendering results from one eye for the other. Meanwhile, foveated rendering focuses on those pixels where the user is actually looking at. Pixels in the periphery, where the resolution of the human visual system is reduced, can be rendered at a lower spatial rate. Variable rate shading has been introduced recently on the GPU to facilitate scenarios where multiple pixels share a single shading computation. Another example of spatial shading resue are upscaling methods that first render at a lower resolution and increase the resolution through spatial filtering.

A less exploited, but still huge potential for shading reuse is shading reuse over time. Many rendering systems, such as standard forward rendering, render every frame from scratch, discarding all previous results of previous frames. However, shading is spatially and temporally coherent and thus provides more opportunities for shading reuse. Existing techniques that exploit temporal coherence mostly do so only for brief time intervals using an assumption that shading only changes slowly over time.

Temporal coherence is typically exploited for temporal filtering, such as for temporal anti-aliasing (TAA). TAA is a technique to amortize spatial anti-aliasing over time. Samples in screen-space of previous frames do not exactly mach the samples of a current frame spatially. Thus, combining the current sample with the previous one results in spatial super-sampling, provided that shading does not change temporally. These techniques typically suffer from ghosting artifacts, if the shading changes too rapidly.

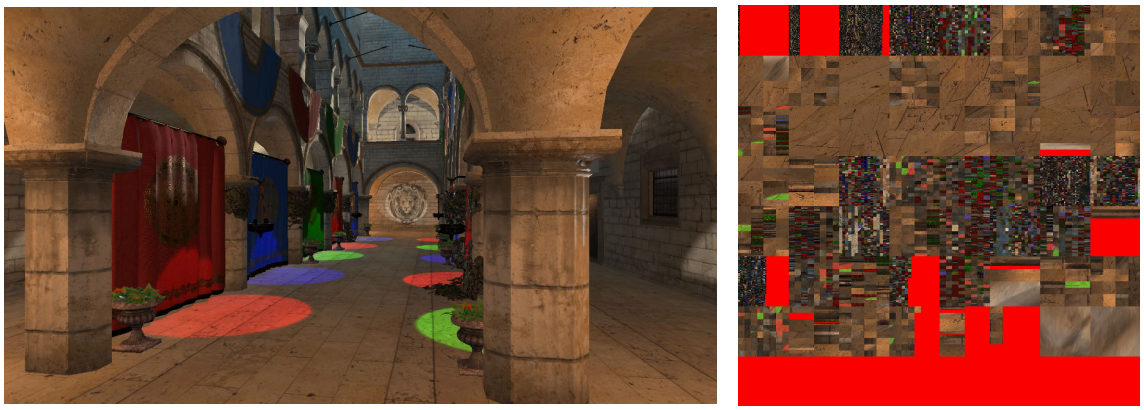
In contrast, temporal reuse is less frequently exploited, although it presents a significant opportunity. This is likely owed to the difficulties of predicting how shading points change over time. Even with the aid of advanced caching, most methods for exploiting temporal reuse rely on the assumption that shading varies slowly and steadily in the temporal domain. In general, this assumption does not hold beyond a trivial Lambertian model. Therefore, a uniform reduction of the temporal shading rate causes either missed opportunities for savings or leads to visual artifacts.

1.4 The Shading Atlas

This thesis discusses the design of the shading atlas for real-time rendering. While the shading atlas can be utilized for any type of rendering, including offline path tracing, it excels in its application for real-time rendering. Rendering for VR displays currently poses the most difficult application scenario. The goal of this thesis is to target the issues of real-time rendering for VR using the shading atlas.

Several algorithms and techniques need to store rendering results in some sort of shading cache. Naïve approaches may use screen space buffers and thus need to reproject the information from the previous frame to the current frame. Notably, sampling positions are typically not exactly the same. This property is used by TAA for spatial filtering, but can be detrimental when the information is not properly filtered, such as in reverse reprojection caching. Holes caused by disocclusions need to be filled with newly computed shading results that cannot rely on cached values.

An alternative is to store rendering information in a temporally invariant space. Object-space shading techniques, like the REYES rendering architecture, provide such a space. More recently, novel surface representations, like texture space shading gained popularity due to hardware support of GPU manufacturers. Such surface representations need to be non-overlapping and bounded, which is not always the case for texture coordinates of common models. Thus it may be necessary to generate a new surface parameterization.



(a) Sponza scene (b) Shading atlas

Figure 1.3: The sponza scene (a) is rendered with the shading atlas (b) that stores the shading of the visible geometry.

Surface representations can be considered a specialized form of an object-space representation, since it is typically addressed within an object's surface.

Another advantage of a screen-space independent storage of rendering results is the possibility to store results that are otherwise occluded or outside the viewing frustum. This allows the storage to cover samples that are visible at a later time. A remote rendering approach, for example, can utilize this to support warping without causing disocclusion artifacts. Rendering pipelines based on this approach further provide natural structure to decouple rendering between a server and a mobile client. The server can generate high quality shading information stored in object space. After streaming, the client may render multiple final views.

The shading atlas is a real-time cache for shading information as shown in Figure 1.3b. It stores shading samples in a single texture atlas independently of the samples' screen-space position. Samples are grouped together by their geometric primitives. In that sense it can be considered an object-space or texture-space shading cache. The shading atlas allows dynamic allocation and deallocation in parallel on the GPU. While allocated, sample positions do not change in the atlas. This behavior allows easy reuse of the stored shading samples.

1.5 Objectives

After presenting the design and functionality of the shading atlas, we will investigate two application scenarios for the shading atlas in detail. The first application scenario targets streaming for VR. We will use the shading atlas for a decoupled rendering approach that allows extensive latency hiding. In addition, we will look at the spatial shading reuse capabilities of the shading atlas, specifically for stereo rendering in VR. The second application

scenario looks more closely at temporal shading. Since this topic is less explored, we start with a more fundamental investigation, before developing a temporally adaptive shading framework. This framework works especially well when combined with the shading atlas, since it allows extended periods caching.

Based on the development of the shading atlas and the application scenarios, we define research objectives that are developed within this thesis.

O1 Find an efficient implementation – in terms of memory and performance – of object-space shading for real-time rendering on the GPU.

O2 Show how object-space shading provides an easy way for spatial reuse of shading information.

O3 Provide a framework that, together with object-space shading, allows extended temporal reuse of shading results resulting in performance gains that can be used for higher quality rendering.

O4 Show that object-space shading can be used efficiently for streaming and hides latency better than screen-space based approaches.

To reach objective **O1**, we design the shading atlas as a compact shading cache in a single texture. The memory management of the shading atlas is hierarchical and can run in parallel solely on the GPU in a negligible amount of time. Using this cache, we will explore several different spatial shading reuse applications, which differ considerably from screen-space solutions, for objective **O2**. The applications are spatial upsampling, stereo rendering and foveation. For objective **O3**, we will first investigate temporal shading reuse, trying to answer how outdated shading is perceived and how long it may stay valid when it is decoupled from visibility sampling. Based on these findings, we investigate how to predict shading reuse intervals and how this can be utilized to reduce shading costs. We will show that the utilization of the shading atlas allows longer reuse than repeated reprojection of screen-space samples. We will use the possibility to decouple rendering with the shading atlas to reach objective **O4**. The design of the shading atlas will also optimize for efficient MPEG compression that is used to stream the atlas from a powerful desktop computer to an Android based mobile HMD. The decoupling allows the client to render new views at a high frame rate effectively hiding the latency of the overall system, including the networking latency.

The organization of these thesis is structured mostly based on the objectives. Chapter 2 introduces and compares previous research that is related to this work. Chapter 3 details the design of the shading atlas that is based on the requirements for VR rendering and streaming. It then details the memory architecture and how the atlas is managed on the GPU in real-time. Next, Chapter 4 explores the possibilities of spatial shading reuse using the shading atlas. Chapter 5 investigates the temporal behavior of shading leading to the development of the temporally adaptive shading framework in Chapter 6 to investigate the

limits of temporal shading reuse. Chapter 7 uses the shading atlas in a decoupled remote rendering architecture for VR streaming. Finally, Chapter 8 draws overall conclusions of the research conducted and the results.

1.6 Publications

During the creation of this thesis, several papers have been published, which served as source of the content of this thesis. Additionally, other papers have been published that the author of this thesis contributed to that are also relevant for the field of high performance graphics and VR.

1.6.1 Individual Publications about this Thesis and Collaboration Statement

- **Joerg H Mueller**, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger and Dieter Schmalstieg:
  Shading Atlas Streaming
  ACM Transactions on Graphics, Proceedings of SIGGRAPH Asia, 2018

  The author of the thesis did most of the implementation work, experimenting and writing of the paper. Philip Voglreiter implemented parts of the geometry processing and edited the video. Mark Dokter implemented image based rendering reference methods for the evaluation. Thomas Neff implemented and evaluated the Android client. Mina Makar helped with his expertise on video streaming. Markus Steinberger and Dieter Schmalstieg helped writing the paper and contributed with their expertise on the overall system.

- **Joerg H Mueller**, Thomas Neff, Philip Voglreiter, Markus Steinberger and Dieter Schmalstieg:
  Temporally Adaptive Shading Reuse for Real-Time Rendering and Virtual Reality
  Accepted to ACM Transactions on Graphics

  The author of the thesis did most of the implementation work, experimenting and writing of the paper. Thomas Neff implemented the spatial filtering, parts of the evaluation and the Unreal Engine implementation. Philip Voglreiter implemented parts of the geometry processing. Markus Steinberger and Dieter Schmalstieg helped writing the paper and contributed with their expertise on the overall system.

1.6.2 Additional Publications Beyond the Scope of this Thesis

- Mathias Parger, **Joerg H Mueller**, Dieter Schmalstieg and Markus Steinberger:
  Human upper-body inverse kinematics for increased embodiment in consumer-grade virtual reality
  Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology, 2018

- Bernhard Kerbl, Michael Kenzel, **Joerg H Mueller**, Dieter Schmalstieg and Markus Steinberger:
  The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the GPU
  Proceedings of the 2018 International Conference on Supercomputing, 2018

Chapter 2

Related Work

Contents

Several fields of research are related to the topic presented in this thesis. Object-space shading and virtualized textures are the primary fields of interest related to the shading atlas. When the shading atlas is used for streaming, it turns into a remote rendering approach and consequently needs to be compared to existing image-based rendering (IBR), which most streaming approaches are based on, as well as the remote rendering literature. Finally, the shading atlas allows shading reuse spatially and temporally, relating it to previous work in the area of spatial and temporal sampling and shading reuse. This chapter is devoted to introducing related research and putting it into context with this thesis.

2.1 Object-space shading

Probably the first object-space shading architecture was the REYES system [30]. REYES split the scene geometry into micropolygons that were shaded independently of their position in the final image. This property distinguished REYES from the usual rendering architectures which would either shade rasterized polygons or ray traced surface points in

image space. Such object-space shading methods are especially useful to compute depth of field and motion blur. Later developments that focused on these application scenarios have reached almost real-time performance [5, 83]. Unfortunately, many sophisticated object-space shading ideas are implemented in software rendering [5, 19, 26, 27] and would require GPU extensions for competitive real-time use.

However, some variants of object-space shading can be implemented with shaders using multiple passes. A *visibility stage* determines visible geometry, a *shading stage* updates shading information of in the texture, and a *display stage* generates the image to be displayed from the texture. The explicit mapping into object-space makes it straightforward to take advantage of spatio-temporal coherence, for example, by adjusting shading rates. However, memory consumption and shader load are directly dependent on the effectiveness of the visibility pass.

Using this general architecture, object-space shading can be practical in real-time rendering when a proper storage is used. The compact G-buffer [60] stores shaded pixels organized in a linear buffer using a hash map. Hashing is convenient for stochastic rasterization, but not so suitable for filtering or video encoding, since it scrambles spatial coherence. Furthermore, the memory management relies on atomic counters and does not support deallocation. Therefore, the approach poorly supports temporal coherence, since shading data cannot be kept for multiple frames.

Based on the surface representation of triangle meshes, a surface-space storage using textures stands to reason and has been extensively used to cache lighting information. An early example of this is the surface cache in Quake [1], a technique that developed into the more commonly known light mapping. The surface cache stores lighting information at a lower resolution than the object textures, and final shading is computed as the combination of the two. Whereas light maps are indexed by position, other approaches may index by direction using cubemaps. Image-based lighting typically uses high dynamic range (HDR) images stored as cubemaps for lighting. Expensive importance sampling can be approximated using a split sum approach and pre-convolves the sums storing the results in the MIP layers of the cube map and a separate BRDF lookup texture [50]. A similar approach has been used by Scherzer et al. [86] for pre-convolved radiance caching. All of these techniques speed up the lighting computation, which, in combination with the material information of an object, is the basis for the computation of shading.

When shading samples are stored in textures, spatial coherence is guaranteed and hardware acceleration of texture sampling filters can be exploited. In scenes with limited overdraw, such as in games with a zoomed-out viewpoint, it can be sufficient to allocate a pre-charted texture per object on demand (for each object that is not culled) to store the shading information [8]. In scenes with high overdraw, visibility can be determined per triangle using a depth buffer. For example, texel shading [46] shades only the visible portion of the scene, but organizes the shaded pixels using pre-charted, mipmapped textures per object. Unfortunately, large portions of the allocated memory remain empty, making texel shading and similar approaches based on model texture coordinates unsuitable for video

streaming. Nevertheless, texture-space shading methods are already receiving support from GPU manufacturers [100].

In contrast to previous object-space methods, the shading atlas uses a texture-space representation and manages shading samples with a compact memory footprint in a single texture. The shading atlas can be implement efficiently on current hardware using multiple passes, requiring no GPU extensions. We fill our object-space texture with shaded information for individual triangles at appropriate sampling rates. Allocated space in the atlas remains at the allocated position, potentially for many frames, until the space is re- or deallocated in real-time, i.e., the atlas is temporally coherent. The per-triangle sampling rates of our approach are determined similarly to vertex color textures [121], but the triangle-to-texture mapping is determined using an approach inspired by virtualized textures, as described below.

2.2 Virtualized textures

Storing shading in texture space needs appropriate memory management. A memory-efficient representation of shading information usually requires a *texture atlas*, i.e., a one-to-one mapping from all object surfaces into a single texture space [20]. When the amount of surface space to be mapped is too large, it may be necessary to use a virtualized texture (VT). The texture is split into square tiles (pages), of which only a working set is loaded [22]. For terrain rendering, this working set is implicitly given by the viewing frustum [99]. For general polygonal objects, an *indirection texture* (pagetable) is required [55]. Dynamic updates to the working set additionally require a *tile fault* component to determine missing tiles and a *tile fetch* component that loads these tiles [57].

Today's GPU architectures expose hardware-accelerated paging for VT [21], albeit only with a single level of indirection and a fixed tile size. Therefore, VT has been mostly used with paging in software implementations without specialized hardware support. For example, the well-known id tech 5 engine [106] is based on rendering with very large virtual textures. One issue of VT software implementations is that they also have to deal with texture filtering, unable to fully use the hardware-accelerated texture filtering units.

Terrain rendering lends itself to a hierarchical VT implementation, since the terrain can easily be broken up into square chunks. Visibility and level of detail for each chunk can be implicitly determined [99], so dynamic packing of visible terrain chunks given at multiple resolutions into a single "stitch map" can be computed on the CPU [8, 24, 70].

In contrast to an implicit hierarchy in terrain rendering, the shading atlas supports dynamic packing of shading information associated with individual triangles. This is also in contrast to the id tech 5 engine which manages the textures of large chunks or objects, rather than of the more detailed level of individual triangles. Computing the packing of a large number of entities given at different levels of detail on the CPU would be too slow. The shading atlas efficiently manages memory on the GPU at full frame rate.

2.3 Image-based rendering

IBR synthesizes an image from a novel viewpoint by warping a previous image. Probably the most widely known warping is asynchronous time warping (ATW) popularized by the Oculus Rift [107] (Figure 2.1). ATW compensates for the perceived latency during rapid rotational head motion by re-adjusting the displayed viewport inside a slightly overscanned framebuffer. This adjustment happens just prior to display scan-out, based on the most recent tracked (or predicted) head motion. An important limitation of ATW is that it does not support full 3D camera motion. Rotations can be handled perfectly, as long as the extended field of view (FOV) of the framebuffer suffices. However, translation causes artifacts since the two dimensional framebuffer lacks depth information to correctly warp the view.

Full 3D motion can be handled by more advanced warping methods, which use a variety of intermediary representations. Shade et al. [90] use layered depth images, creating the same effect known as parallax scrolling in animation. Depending on their depth, objects are rendered to one of multiple images that can later be warped at their different depths following translational movement. Using only a single image, this method degrades to ATW.

Multiple techniques exploit the depth buffer that is generally available for rasterized images. Point splatting [25] splats slightly oversized pixels at the three-dimensional position of the original images. This works well for minor changes in viewing position, but causes noticeable holes if the camera position is to far from the original image position. The splat size allows a tradeoff between blurriness of the final image and susceptibility to such holes.

Other approaches reconstruct a grid mesh out of the depth buffer that is then used for warping [56, 69] (Figure 2.2). The quality can be influenced by the resolution of the grid, at the cost of performance. However, the loss in performance can be counteracted using a quadtree to simplify the geometry in flat areas [36]. Unfortunately, such mesh warping techniques suffer from “rubber sheet” artifacts appearing at depth discontinuities, i.e., at disocclusions.

All methods discussed so far are forward warping methods, warping the image to the space of the current view. Alternatively, backward warping tries to warp from the current view back to the original image and depth buffer to find the best fitting sample. Since this is not a straightforward computation, Bowles et al. [17] suggest an iterative approach that optimizes the sample position (Figure 2.3). Using the depth buffer, this approach still fails at disocclusions. Yang et al. [117] proposes an approach that, in addition to the last image, also utilizes a future image for bidirectional reprojection, since it is unlikely that samples are occluded in both frames. Unfortunately, this approach is not possible in low-latency or remote real-time rendering, since future frames are not available.

All techniques utilizing the depth buffer suffer from depth discontinuities, leading to approaches that use more complex proxy geometry. This proxy geometry can be the original scene geometry, but simplified meshes may suffice. Though the depth discontinuities

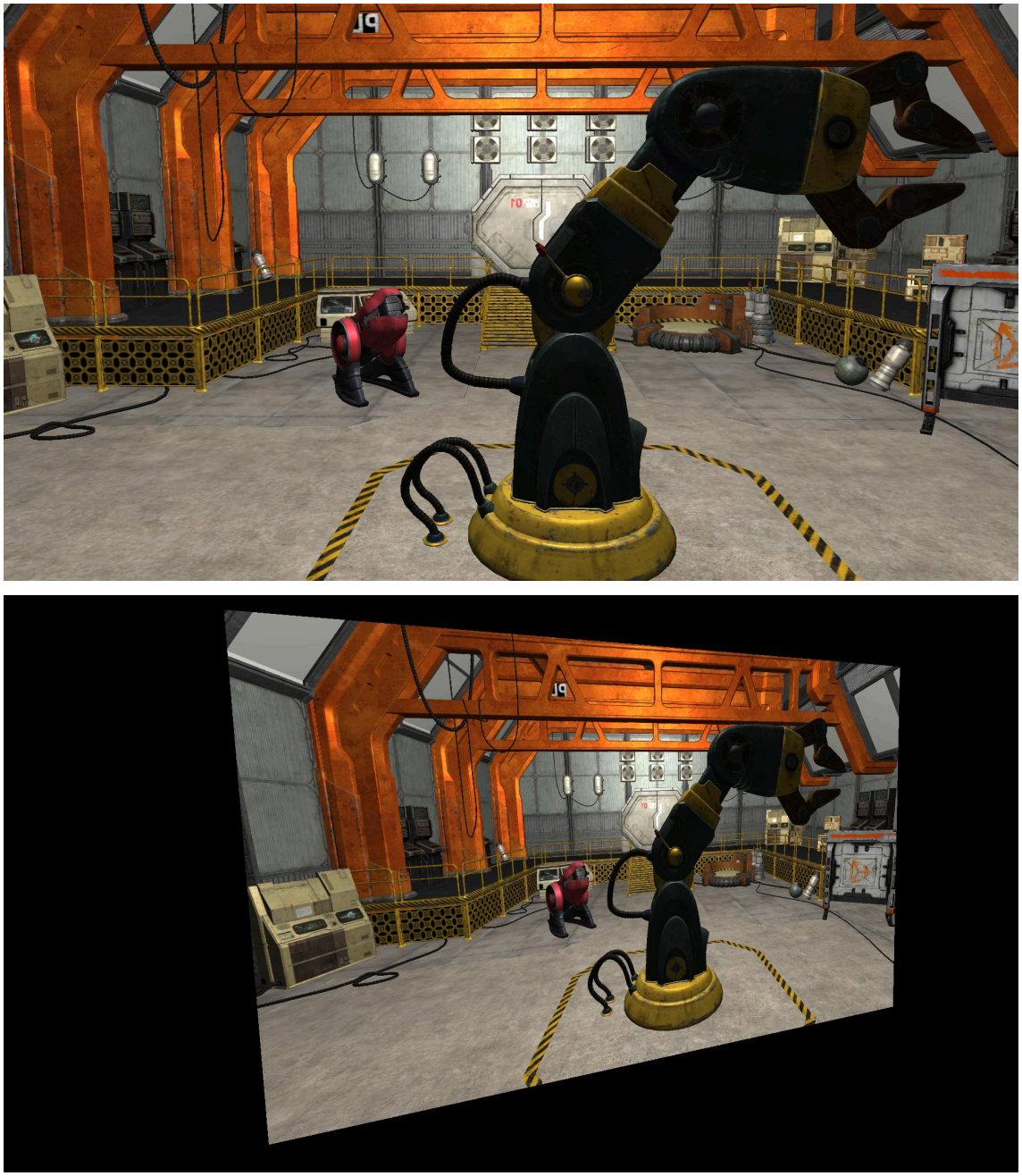


Figure 2.1: Asynchronous time warping [107] simple warps the image plane in accordance with the camera movement. A slight sideward translation (top) moves the whole image at the same rate, without considering the distance of the objects within the scene. Further translation of the camera (bottom) reveals the flat image plane that ATW is using.

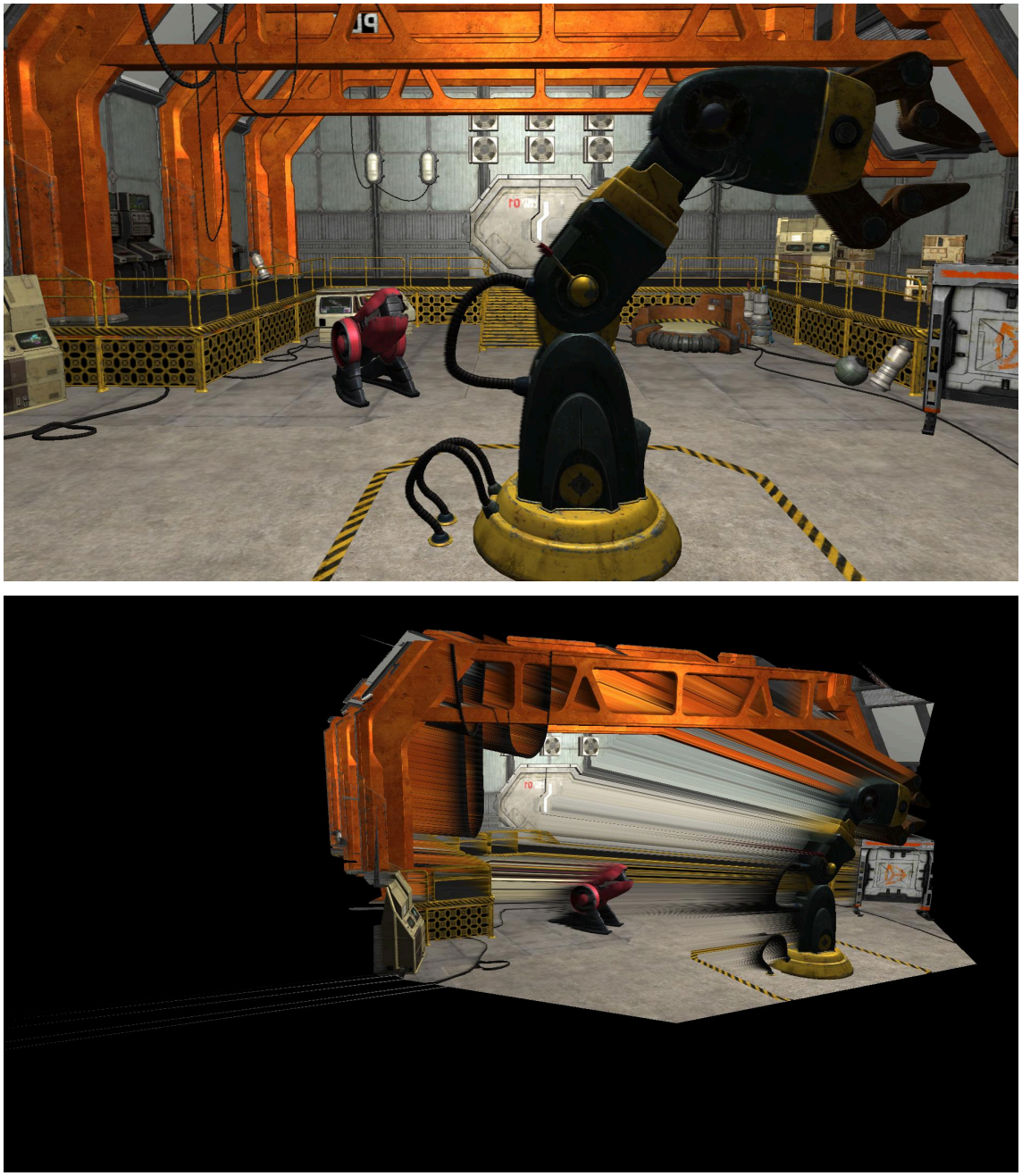


Figure 2.2: Mesh warping [56, 69] uses the depth buffer to generate a mesh onto which the image is projected. A slight sideward translation (top) already shows some artifacts at disocclusions, e.g., on the left side of the crane. Further translation of the camera (bottom) reveals the "rubber sheet" artifacts appearing at depth discontinuities that is caused by projecting the image onto a grid mesh based on the depth buffer.

Figure 2.3: Iterative image warping [17] uses backward warping to find the best fitting samples for a new camera position. A slight sideward translation (top) already shows some artifacts at disocclusions, e.g., on the left side of the crane. Further translation of the camera (bottom) reveals that the warping can partially capture the three-dimensional structure of the scene, but again disocclusions lead to appalling artifacts.

can be handled better in terms of geometry, the missing shading information in the area of disocclusions still causes artifacts. Reinert et al. [85] solve this issue with a spherical projection to increase the FOV, and a second view that is placed in front and above the primary view and renders at a lower resolution with depth peeling. Unfortunately, the fine tessellation required for the spherical projection and rendering the second view with depth peeling have a considerable impact on performance, and the placement of the second view does not cover all possible scenarios, such as a backward moving camera.

Unstructured lumigraphs [18] optionally supports geometry, degrading to a lumigraph/light field rendering approach without geometry. While the technique does not suffer from disocclusion artifacts as long as the camera remains within the viewing volume covered by the light field, it suffers from the same issues as most light field rendering approaches: redundant information and thus high memory consumption, more complex and slower rendering, as well as poor support for remote rendering. Lochmann et al. [62] developed a simpler approach that supports reflective and refractive shading effects in combination with image warping. However, like most IBR methods, it suffers from disocclusion artifacts, and its practicality for remote rendering, requiring appropriate compression of the used RGBZ ray trees, has yet to be shown.

In general, IBR methods make novel view synthesis simple, but suffer from fundamental issues. On the one hand, unless multiple images are used, the main problem of IBR is disocclusion at depth discontinuities. On the other hand, if multiple images are used, redundant information in the images makes the approach more complex, memory intensive and less feasible for spatial reuse, temporal reuse and streaming. In contrast, object-space shading method do not suffer from disocclusions, since they are, in general, not view dependent. The shading atlas is a compact and temporally coherent object-space representation suitable for aforementioned application scenarios.

2.4 Spatial shading reuse

The fact that shading shows spatio-temporal coherence is used in all domains of rendering, ranging from texture mapping to global illumination or denoising. Most related to our approach are techniques that reduce shader invocations by exploiting shading coherence. Spatial shading reuse for real-time rendering has received a lot of attention. It typically lowers the spatial sampling rate in areas with lower spatial frequency to achieve a reduction of costly shader invocations.

Simple image upsampling can be considered a form of spatial shading reuse. Various interpolation filters are commonly used with varying complexity depending on the required performance and quality. Nearest-neightbor interpolation is the fastest method, but has the lowest quality result, since it simply duplicates the nearest original pixel. Due to hardware support on the GPU, many applications rely on bilinear interpolation. Bilinear interpolation is typically used for texture mapping [44], but image upsampling often requires higher quality and thus higher order filters, such as bicubic, spline or Gaussian

filters [13, 47, 65]. The "ideal" interpolation filter would be a *sinc* filter, but, since it has infinite support, it is not practical. Instead, the *sinc* filter is truncated with a windowing function in order to make the filter practical. A simple truncation, which is equivalent to windowing with a rectangular window, causes strong artifacts such as ringing. While there exist many alternative windowing functions, the most widely used windowing function is that of the Lanczos filter, the central lobe of the *sinc* function, since it provides good quality filtering [14, 103]. While these filters are widely used in practice, recent research focuses on upsampling with neural networks trained with deep learning [7, 37, 119]. However, most of these approaches are either spatio-temporal or not real-time capable [78].

Foveated rendering [41, 49, 80, 98] is based on the human visual system. The highest resolution of the human eye is around the fovea, i.e., the central region of the visual field. Further away from the fovea, the spatial resolution decreases significantly. Foveated rendering builds on this fact, by rendering peripheral areas at a lower spatial resolution. Ideally, and especially in VR, the area of focus needs to be determined using an eye tracker.

More generally, variable rate shading can be applied to multiple application scenarios, such as diffuse shading, shadow mapping and motion blur [43]. Recently, hardware manufacturers added support for variable rate shading [77, 104], allowing applications to define which areas of an image are rendered at which spatial resolution. The resolution can be chosen out of a few fixed options, including several supersampling options for areas which need higher resolutions and filtering to avoid aliasing. Recently, Yang et al. [118] have shown that variable rate shading can result in significant performance gains in modern real-time applications with almost no difference in quality, provided that the system is tuned properly. However, purely spatially adaptive systems lose potential savings in the temporal domain.

Most spatial shading reuse is done in image space, though object-space shading especially lends itself to spatial shading reuse. The performance gain in stochastic rasterization achieved by object-space shading methods is primarily based on the spatial reuse of shading [5, 19, 26, 27, 60, 83]. The object-space representation is necessary, since image space can neither cover the required spatial samples, nor does it allow easy reprojection of the samples for this application case. Another application of spatial shading reuse, where object-space shading proves advantageous, is stereo rendering. IBR again suffers from disocclusion artifacts, and, without depth information, such as in the case of ATW, no stereoscopic depth effect can be achieved. The shading atlas allows stereo rendering with a negligible impact on performance in comparison to mono rendering.

2.5 Temporal Upsampling

Temporal upsampling methods reuse previous shading results without filtering or accumulation. ATW warps the image plane of the previous, fully rendered keyframe based on the latest head-tracking update. Advanced warping and reprojection techniques such as the render cache [108] or reverse reprojecetion caching [75, 94] may also use scene depth

or motion vectors for dense 3D warping [35, 117], while reusing the shading from the last keyframe and discarding samples based on their age and reprojection error. These techniques are based on the assumption that the temporal variation in shading is slow and, as spatial reprojection errors accumulate over time, a frequent refresh of the cache is required. However, we show that, while this assumption holds for a large number of shading samples, a fraction of samples typically violate this assumption and thus lead to perceivable artifacts, when not shaded more often.

It has been shown that visibility is temporally coherent [40], which is the main reason why temporal coherence is exploited in image-space methods. Nehab et al. [75] reported that more than 90 % of visibility samples stay valid between frames, but the temporal validity of *shading* samples was not thoroughly investigated in previous work. Indeed, the temporal coherence of shading independently of visibility has not been investigated at all. We close this gap by proposing a temporally adaptive shading method that exploits coherence of shading, in addition to visibility, over time.

The shading atlas can also be used for temporal upsampling. Due to the object-space representation, it does not suffer from accumulating reprojection errors. This makes it ideal for shading reuse in remote rendering for VR, where latency hiding is of utmost importance. For unchanging shading, the shading atlas allows shading reuse over long time periods. A fixed upsampling rate may, however, still cause artifacts for surfaces with dynamic shading changes at higher frequencies. This problem can be resolved with a proper way to decide which parts of the shading atlas have to be reshaded at what time.

2.6 Spatio-temporal filtering

Temporal coherence is commonly exploited by using information from previous frames for spatial filtering with temporal amortization. The popular family of TAA techniques [51, 115, 116] uses the slight variation in sampling positions, which is caused by the screen-space warping of a previous frame, for anti-aliasing. With the help of exponential-decay history buffers previous samples are filtered together with the current ones. In this way, TAA uses the temporal variation of sampling positions to achieve spatial anti-aliasing. TAA assumes that shading only changes slowly and thus can ignore the low-pass filtering of the temporal samples. Without any guidance in how quickly shading varies temporally, special care must be taken to avoid ghosting artifacts and low-pass filtering of high frequency shading changes, for example, using pixel-history linear models [48]. Thus, the exponential decay is typically rather rapid, and special measures are implemented to discard past samples, if the difference is above a certain threshold. Other spatio-temporal filtering techniques also build on the assumption that shading is temporally coherent, focusing on short-term temporal sampling.

Shading gradients can be used to estimate the variation of shading and are thus often used in spatio-temporal filtering. For example, Herzog et al. [45] uses shading gradients to guide spatio-temporal upsampling; adaptive frameless rendering [33] also uses gradients for

the reconstruction filter that produces final images from the spatio-temporal samples, and Schied et al. [87] estimate gradients for real-time adaptive temporal filtering in path tracing. The shading cache [102] for path tracing allows for spatio-temporal shading reuse by constructing a hierarchy of object-space primitives and storing shading samples in its mesh vertices. Based on age, current view and shading difference, a priority map is generated and used to determine sample locations for a path tracing renderer. While this enables spatially adaptive sampling for interactive global illumination, lighting or geometry changes may lead to temporal artifacts.

Ideally, a temporally adaptive shading system only decides to refresh shading samples that exhibit a strong temporal gradient. This does not necessarily have to be the gradient of the complete shading function, as different shader components can have different frequencies, allowing for different sampling rates [42, 95].

Gradients can be used in order to predict changes and necessary sampling resolution. Previously, Ramamoorthi et al. [84] have shown that a first order analysis achieves satisfying results for spatial sampling, without having to resort to a full frequency analysis [38]. Differential rendering [52, 63], which is increasingly used to combine machine learning with graphics, also makes extensive use of first-order gradients. However, differential rendering typically considers gradients of higher-level parameters, such as the camera position or material parameters [61] and not necessarily spatial or temporal gradients. While it is possible to tune temporally adaptive systems using spatial gradients of the shading function [38, 84], it is difficult to amortize the overhead of gradient computation for real-time rendering. In addition, just as in IBR, stability problems can arise at discontinuities caused by visibility gradients in image space, making it necessary to develop specialized solutions to circumvent this issue [59].

Most aforementioned approaches use gradients to guide spatio-temporal filtering or spatial sampling, but they have not been used to guide temporal sampling or shading reuse. Furthermore, spatio-temporal filtering in texture space allows reuse of samples that are not visible in image space [74]. Ideally, a temporally adaptive shading system only decides to refresh shading samples that exhibit a strong temporal gradient. Similarly to Ramamoorthi et al. [84], we use a first-order approximation to predict shading gradients and base the decision to temporally reuse shading on this gradient. Unlike other approaches, this allows shading reuse over extended periods of time.

Numerous image quality metrics try to model the human perception of images and videos, but lack consideration of temporal effects. While the peak signal-to-noise ratio (PSNR) provides an objective and easy to compute metric, it fails to capture human perception. The popular structural similarity index measure (SSIM) [110], which has been designed to more closely resemble perception, is still the prevalent image quality metric. Various improvements, such as IW-SSIM [111], have been made to SSIM to enhance the predictions of the metric. Other metrics, such as HDR-VDP-2 [67], even try to model the visual system to some extent. Swafford et al. [98] build on HDR-VDP-2 to adapt the metric for the evaluation of foveated rendering. Another approach is the combination of different

metrics, such as VMAF, which appears especially useful for video game content [10]. The novel FLIP [6] metric is derived from the manual method of comparing images by alternating between them and provides an error map showing where differences would be perceived between the two images in comparison. A major disadvantage of all these metrics is that they only compare images, disregarding any temporal artifacts, such as flickering. A few video quality metrics incorporate temporal aspects [34, 82, 89, 113, 114], but focus mostly on video compression related artifacts such as frame dropping and have not been evaluated for temporal artifacts appearing in rendered imagery. Therefore, conducting a user study remains the gold standard method for the evaluation of perceptual quality, especially when temporal effects have to be considered.

2.7 Remote rendering

Remote rendering systems can be classified based on different criteria. Shi and Hsu [92] distinguish the systems in their survey based on the amount of user interaction and the complexity of the 3D model data. The most difficult of these are cloud gaming systems with unrestricted user interaction and dynamic scenes. The limitations of some remote rendering systems prevent them from supporting certain of these application cases, restricting them, for example, to remote walk-throughs of static scenes.

A good technical classification of remote rendering systems is based on which type data is sent over the network. According to Shi and Hsu [92], this classification is directly correlated to bandwidth consumption and computational effort for rendering on the client. It leads to a spectrum ranging from model based techniques at one end to image based techniques at the other end. A pure model based technique transmits all necessary data to the client and lets it render everything. In contrast, a pure image based technique transmits rendered images to be directly viewed at the client.

Most previously discussed IBR techniques can be utilized for remote rendering. ATW [107], only needs to transmit the image and an average depth that the image plane is warped at. While such a minimal system is limited on the client side to simple image warping, the server can use rendering information for more efficient MPEG encoding [76]. IBR utilizing the depth buffer can transmit data as color+depth images and render via splatting or texture-mapping to a meshed depth buffer [23, 93]. Streaming the depth buffer may utilize proprietary color+depth compression [79] or adapt standard video codecs for depth streaming [81]. Deriving the proxy geometry from the rendered depth buffer makes it completely independent of the scene complexity, easing the load on the remote client.

Many geometric representations for remote rendering only support static scenes. Complex scenes can be converted to an IBR database in precomputation. The proxy geometry used to organize the IBR can take on a variety of forms, such as sparse [101] or dense [15] impostors, view-dependent texture maps [28], or geometry images [91]. For dynamic scenes, we can take advantage of the fact that, in most games, only the appearance changes dy-

namically, while most of the geometry is static. The static geometry can be used directly for perspective texture mapping, possibly with simplification [85].

Even if all data necessary to render a scene are transmitted, remote rendering can provide benefits by handling more expensive computations remotely. Mobile clients are usually not powerful enough to render indirect illumination effects, requiring remote servers to handle this payload [31, 58]. However, these effects typically have a low-frequency and thus the added latency of remotely rendering these can be neglected. Rendering the same model at server and client can reduce transmission to *residual images* remaining after considering disocclusion [66]. This strategy can be combined with IBR [9, 32, 120]. To better hide latency, IBR can utilize speculative rendering [56] which predicts and renders future frames.

One implicit assumption of most, if not all, IBR approaches is that real-time geometric rendering is inherently expensive and must be avoided as much as possible, especially on mobile clients. This may have been true 20 years ago, but today even a mobile GPU has a fairly high geometry and rasterization throughput. The real cost of today's rendering workloads lies in complex surface shaders. For scenes of reasonable complexity, the most interesting use lies in amortizing the shading results. The shading atlas allows to do exactly this with an object-space representation of the shading that is optimized for streaming.

2.8 Discussion

In this chapter, we put the work done in this thesis in relation to other works in several different fields of research, hinting at how the shading atlas can resolve remaining issues. Object-space shading is mainly focused on stochastic rasterization, but real-time rendering and applications in game development sparked support by GPU manufacturers for real-time texture-based methods. So far existing texture-based methods are memory hungry, lacking a compact representation that allows real-time memory management. The shading atlas resolves these issues opening up more application areas for object-space shading methods. For example, the shading atlas allows easy spatial shading reuse, by being able to choose shading resolution on a primitive basis, e.g., for foveated rendering. Spatial reuse also allows stereo rendering for VR headsets at a negligible performance impact. In comparison to IBR, it supports spatial and temporal reuse without warping issues. However, it is unclear how long shading can be temporally reused and how a decision to shade can be made. We suggest a first order gradient estimation for this prediction. Furthermore, image based rendering methods, which are predominantly used for remote rendering due to the low client complexity, inherently suffer from disocclusion artifacts. The shading atlas is the first object-space shading method suitable for remote rendering with a low client complexity. The client only has to texture map the atlas onto the geometry. All these benefits of the shading atlas will be evaluated and discussed within the following chapters.

Chapter 3

The Shading Atlas

Contents

This chapter discusses the basics of the shading atlas. The first part gives an overview of how rendering using the shading atlas works. The second part details the design principles behind the decisions made for the shading atlas. We will discuss the underlying memory structure and its correspondence to the geometry. Following that is a detailed description of the three phase process to allocate, deallocate and reallocate memory in the atlas in parallel on the GPU. The level selection process is another crucial part of the shading atlas that determines the allocation size within the atlas for a given geometric primitive. Finally, we will finish the chapter with an evaluation of the memory management of the shading atlas.

3.1 Rendering with the shading atlas

Before we detail the design of the shading atlas, we develop the framework of how rendering with the shading atlas works. Rendering with the shading atlas runs in different stages, shown in Figure 3.1, that can be decoupled and allow implementation variants based on the application's needs. The process can be roughly divided into a visibility stage, a memory management stage, that also contains the level selection, a shading stage and a display stage. Except for the memory management stage, these stages are similar to a

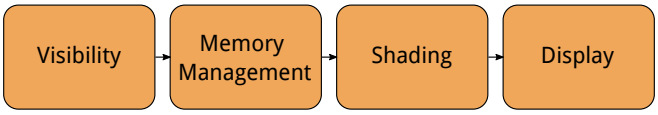


Figure 3.1: The shading atlas rendering pipeline consists of four major stages. Each of these stages can be run at different frequencies, since the stages can be decoupled. Additionally, some stages, like the visibility stage can be implemented in various ways, depending on the requirements of the application.

deferred shading pipeline, which uses a screen-space geometry buffer (G-buffer) to store the necessary geometric information for the deferred shading stage [4]. This section will give a brief overview of the four stages and some options of their implementation. The stages can run decoupled, but do not have to do so. For example, for streaming, the decoupling happens between the shading and the display stage.

Visibility The goal of the visibility stage is to determine the geometry, divided into *patches*, that is visible within all camera positions and orientations in the corresponding display stage runs. If this stage is coupled to the display stage, an exact visible set (EVS) suffices. The most simple way to compute an EVS is to render an index buffer that contains the indices of visible triangles which are then marked visible. However, if visibility is not coupled to display, a potentially visible set (PVS) will be determined that covers the possibly visible geometry of all subsequent views that are rendered until the next PVS is determined. Any PVS generation algorithm can be used for this stage, as long as it can output a list of visible patches for the following memory management stage. The possibility to use a PVS over an EVS is a major advantage of object-space shading in comparison to screen-space based methods such as deferred rendering, which can only use the EVS and thus may suffer from issues with disocclusions.

Memory management The memory management stage follows the visibility stage and manages the area of pixels corresponding to the visible patches, including their view-dependent size. Thus the stage begins with the level selection process from the previous section. If this stage runs at a higher rate than the visibility stage, the level selection may choose a different level for some patches, based on the movement of the camera. After the level selection, the three phases of memory management are executed. The result of this stage is the assignment of a block of memory within the atlas to each visible patch for shading.

Shading In the shading stage, visible patches are shaded into the atlas. The number of shader invocations should be proportional to screen size with some overhead factor. The first step of this stage is to write a vertex buffer containing the visible geometry including corresponding texture coordinates for the shading atlas. As an optimization if shading is

not decoupled from memory management, this geometry buffer can already be written in the memory management phase. Using the vertex buffer containing the visible geometry, the shading stage can be implemented as a standard geometry rendering pass with the following characteristics:

- No depth buffer is needed, since triangles do not overlap.
- Instead of screen space positions, the vertex shader needs to output the shading atlas texture coordinates as position.

We designed the shading stage to be able to use a standard geometry pass, because it has important advantages. First, there are no special limitations in terms of materials. Unlike many deferred rendering approaches, no uber-shader is required. Second, object-space rendering methods which perform shading in a compute shader [46] must interpolate all attributes in software instead of using hardware interpolation units, and they do not benefit from the free derivatives delivered by hardware quad shading.

Display The display stage generates the final view of the rendered scene. As input, it simply needs a vertex buffer containing positions and texture coordinates for the atlas and the shading atlas. It can then simply render the geometry, using the atlas for texture mapping. Alternatively, if the display stage is coupled to an index buffer based visibility stage, it can work similar to a deferred rendering system. In this case, the visibility stage writes a texture coordinate buffer with patch local coordinates. Using these together with the location within the atlas which can be looked up using the index buffer, the display stage can simply determine the atlas texture coordinates for each pixel of the final image.

Comparison with other approaches In Figure 3.2, we compare rendering with the shading atlas with different other rendering architectures in terms of major stages and the data exchanged between them. The figure also shows whether each stage is implemented as a graphics or compute pipeline, i.e., whether the scene geometry is rasterized or not.

The simplest approach is forward rendering, which does everything in a single stage. Visibility is determined with depth testing. Shading is performed in the final screen space that is used for display. Forward+ extends forward rendering by separating visibility determination by rendering a depth buffer first, thus avoiding overshading.

Deferred shading can be considered further extending from Forward+, writing all data that is necessary for shading into a G-buffer during visibility testing. Shading can then be implemented in a screen-space compute stage. However, deferred shading has some issues with high memory consumption due to the many attributes that need to be stored in the G-buffer, different shading functions and materials, as well as MSAA. Deferred lighting tries to solve these issues by generating a G-buffer with less attributes. Based on these, it computes only the part of the shading that depends on the lighting, storing diffuse and specular light color in accumulation buffers and finishing the shading in a *final geometry*

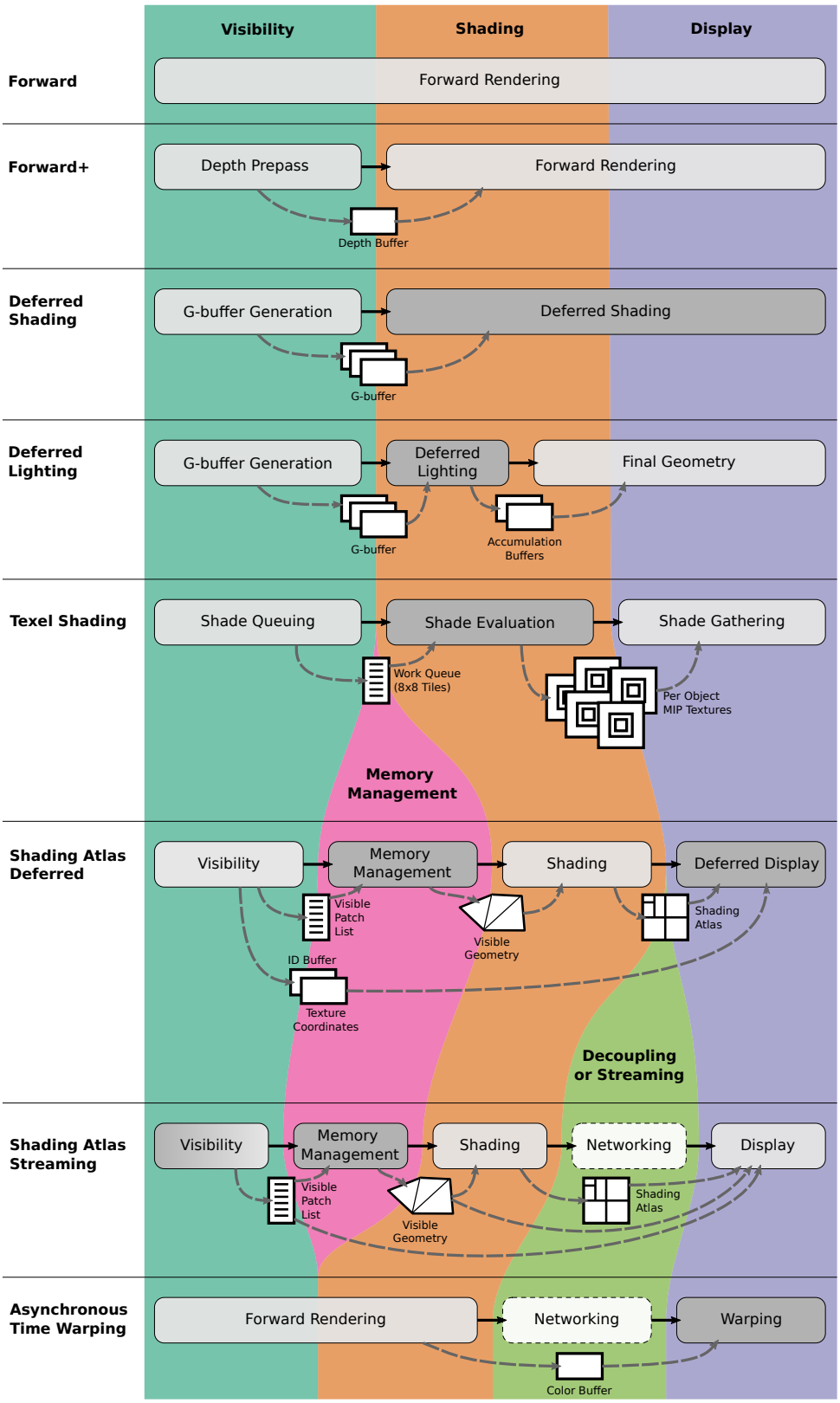


Figure 3.2: Comparison of different rendering architectures.

stage. It can thus be considered an intermediate solution, between forward+ and deferred shading.

In contrast to the previous mentioned architectures, which all work in screen-space, texel shading shades in texture space. The *shade queuing* stage determines visibile texels and mipmap-levels, and fills a work queue with 8×8 pixel tiles that have to be shaded. The following *shade evaluation* stage runs as a compute stage and shades the tiles storing the results in mipmapped, per-object texture buffers. The final *shade gathering* geometry stage simply samples from these texture buffers. The disadvantages of texel shading are similar to the ones of deferred shading, i.e., the memory consumption for the texture buffers and the different shading functions and materials, since shading runs in a compute stage.

Rendering with the shading atlas requires a *memory management* stage to handle the memory of the shading atlas as a fast compute stage and can then shade with a geometry stage that only shades the visible geometry. The deferred approach couples visibility determination of an EVS to the *deferred display* stage, which uses screen-space ID and texture coordinate buffers to assemble the final image in a compute stage. The more general pipeline, shown in the figure as *Shading Atlas Streaming*, allows various different implementations of the *visibility* stage and then renders the final view in a geometry stage. This setup has the advantage that it allows to fully decouple visibility and shading from display, e.g., for frame rate upsampling, i.e., running the display stage at a higher frame rate. In a streaming setup, this decoupling can be used between a server and a client, requiring the introduction of a *networking* stage that encodes, transfers and decodes visible geometry information, i.e., the patches and vertices, and the shading atlas.

Finally, image warping, such as ATW, is a screen-space method that adds on to other architectures, but mostly uses simple forward rendering for visibility and shading. A final warping stage allows decoupling and streaming of the color buffer and, for more complex warping methods, the depth buffer. However, the warping is only an affine transformation of the pixels in the screen-space image plane of the original image, and thus warping suffers from disocclusion artifacts.

3.2 Design principles

The design principles of the shading atlas are mostly based on the target application of using it for real-time rendering on a modern GPU without further extensions and in particular to be able to use it for streaming VR. Thus we distilled the following requirements for the design of the atlas:

- *Small footprint*: The total number of pixels must be small enough so a hardware-accelerated MPEG encoder can deal with the atlas in real time. The atlas should also be a single image, since mobile devices may not support more than a single decoding session.

- *Temporal coherence*: The location of shading information corresponding to a particular triangle must change as little as possible from frame to frame, so that efficient MPEG encoding is possible [29]. It allows lower bitrates at the same quality, since no movement needs to be encoded.

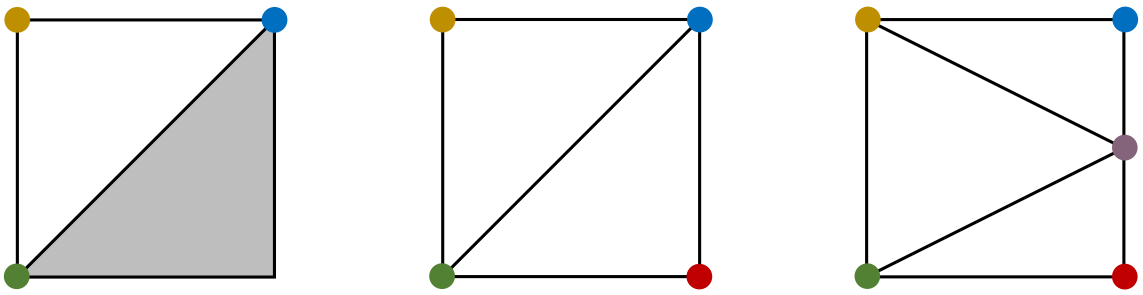
- *Compactness*: Memory fragmentation, i.e., interleaving of occupied and empty areas, makes it more difficult to find places for new memory allocation, especially bigger areas. Moreover, fragmentation reduces video encoding efficiency. Additionally, the entire space in the atlas should be utilized without leaving irregularly shaped empty spaces that cannot be used.

- *Dynamic memory management*: Since we do not know in advance what parts of the scene will become visible and how large a visible triangle will appear on the screen, we cannot precompute any atlas layout or uv-coordinates. Consequently, memory management needs to be dynamically changing between frames.

- *Parallel allocation*: Fine-grained memory allocations of variable-size memory chunks for a large number of triangle patches cannot be performed on the CPU at full frame rate. Allocation must happen on the GPU in a highly parallel manner, with as little need for thread synchronization as possible.

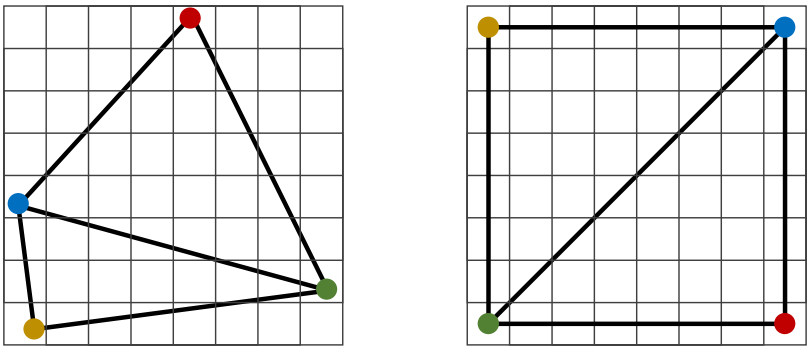
3.3 Structure

In order to fill the atlas, we base its structure on rectangular primitives in atlas space with a correspondence to a geometric primitive. We pack one, two or three adjacent triangles into a *patch*, as shown in Figure 3.3. All atlas operations, such as allocation, deallocation and shading, operate on whole patches as geometric primitives. The shading information corresponding to a patch fits into a *block*, as shown in Figure 3.4. Given the index of a triangle, we can look up the corresponding information in the atlas using a two-step



(a) One-triangle patch (b) Two-triangle patch (c) Three-triangle patch

Figure 3.3: Our system supports three different patch types with one, two or three triangles. The preferred patch type is the two-triangle patch.



Screen space Atlas space

Figure 3.4: A patch consisting of two neighboring triangles is mapped from screen space to a block in atlas space. The vertices are inset by half a pixel from the block edges in order to allow bilinear interpolation in the atlas without results being influenced by neighboring blocks.

indirection: For each triangle, we store a patch id; each patch contains a record on a block in the atlas if a block has been allocated for it.

Blocks have a rectangular, landscape format with each side length corresponding to a power of two. We call the block size a *level* because of its conceptual similarity to a mipmap level. The power-of-two dimensions enable subdivision without leftover space and allow for compact integer addressing using the binary logarithm of the dimension. They also support the video encoder in matching MPEG macroblocks. The disadvantage of this decision is that only a discrete set of block sizes are available and can cause occasional seams between neighboring patches, as they transition between different resolutions. This problem can be minimized by careful construction of the patches.

A half-pixel wide reserve at the border of the block is sufficient when sampling with bilinear filtering. No conservative rasterization [3] per triangle, as in other object-space methods [46], is required. Our block layout is motivated by similar goals as in the vertex color texture work [121]. A larger border could be used to support anisotropic sampling [72]. However, our atlas already contains pre-shaded, pre-filtered content. Avoiding anisotropic sampling during the final display stage benefits the client, which has only modest GPU performance.

The blocks are organized in a three-layer hierarchy (Figure 3.5):

1. The atlas is regularly subdivided into square *superblocks* of the same size.
2. Each superblock is subdivided into *columns* of equal width.
3. Each column is divided into *blocks* of the same width, but variable height.

The three-layer hierarchy allows a tight packing of blocks of variable sizes without having to resort to a global optimization scheme. The effort for insertions and deletions is linear

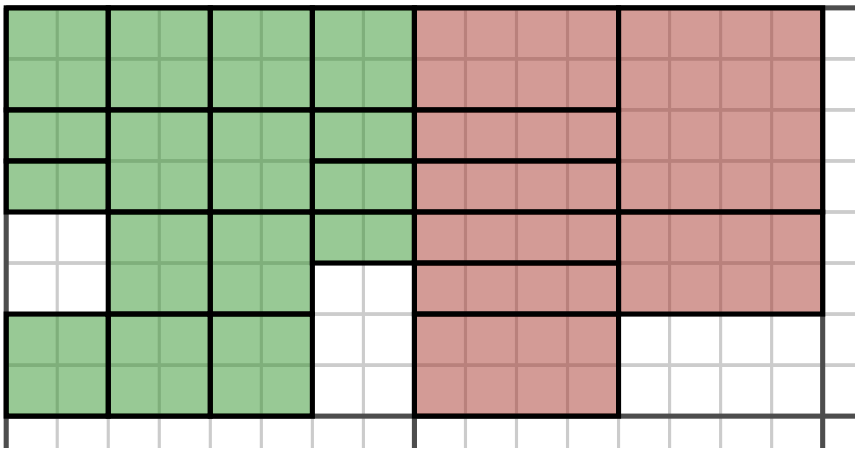


Figure 3.5: The atlas is divided into square superblocks (8x8 in the figure). The green superblock is subdivided into columns of width 2, and the red superblock is subdivided into columns of width 4. Columns are packed with blocks of variable height.

in the number of block changes, without being affected by block sizes. This is made possible by managing one block dimension (width) *across* superblocks, while the other block dimension (height) is managed *inside* each superblock.

We manage the atlas memory over multiple frames, with the goal of optimizing temporal coherence. Already allocated patches, which remain visible in subsequent frames, should remain unchanged, while new allocations should strive to fill gaps between previous blocks, so that fragmentation is minimized. This ensures a small memory footprint, but, more importantly, temporal coherence: Blocks remain in the same place over multiple frames. The shading stored inside a block may be updated in every frame. Since shading typically changes gradually, the MPEG encoder needs to transmit only local image differences. In other words, temporally coherent block positions reduce motion compensation, and samples change only based on temporal changes in shading, not due to the changing sampling positions of shading in screen space. This eases the workload of the encoder, which can spend the available bandwidth on the temporal changes of shading rather than the temporal changes of sample positions.

3.4 Parallel memory management

On a massively multi-threaded GPU, dynamic memory allocation is known to be a hard problem. However, the required fine-grained allocation is too computationally demanding to run it on the CPU on just 1-2 cores, as it would be too slow (see Section 3.6.2 for details). On the GPU, conventional solutions for memory management, such as global free-space lists, quickly become infeasible due to excessive locking requirements when many threads compete for access [96]. Instead, we address this problem by combining two strategies, a GPU-friendly allocation strategy and a global garbage collection step.

We use lock-free allocation, which is not affected by race conditions. Every thread performing an allocation step is guaranteed to finish within bounded time. Each block is serviced by an independent thread. To keep fragmentation minimal without requiring locking, we manage free blocks in *block stacks* and let the allocation run in three distinct phases. In each phase, either allocation *or* deallocation occurs, but not both. This approach lets us operate each stack by atomically increasing or decreasing a stack counter instead of using locks. Since the order of the free blocks is irrelevant, we can use a simple array-based stack implementation, instead of a more complex linked-list-based or ring-buffer-based FIFO queue [53].

For example, when multiple threads allocate blocks of the same size at the same time, an atomic decrement on the stack counter ensures that each thread gets a unique position on the stack, which is then used to read the block. Multiple blocks are removed from the stack at the same time without the threads blocking each other with a lock. Similarly, during the deallocation of blocks, i.e., when blocks are put on the stack, the counter is atomically incremented, and the blocks are written at the corresponding index. However, allocation and deallocation have to run sequentially with this simple method for parallel memory management.

Our memory management strategy uses three distinct phases, a request phase, a provisioning phase and an assignment phase. The phases have the following responsibilities:

- The *request phase* frees unused blocks and counts how many blocks will be allocated.
- The *provisioning phase* allocates superblocks depending on how many blocks are already available and have to be newly allocated from superblocks.
- The *assignment phase* uses the information prepared in the previous two phases to allocate the blocks.

The subdivision into phases, where either allocation or deallocation occurs, lets us operate each block stack by atomically increasing or decreasing stack counters, leading to a lock-free algorithm that is friendly to SIMD operation. We use one stack for each block size and an additional stack for unused superblocks. Free memory is either available on a block stack or inside an unused superblock. In the former case, the memory can only be used for a block of the corresponding size, while memory inside an unused superblock still has the potential to be allocated for any block size.

Superblocks are not freed separately, which possibly leads to memory fragmentation. As soon as blocks have been allocated from within a superblock, they are managed on the block stacks and cannot change their size. These blocks will only be reused if the same block size is requested again. An efficient parallel algorithm that finds free blocks that can be merged into larger ones is hard to implement. The problem is solved with a garbage collection step that frees all memory and newly allocates it.

Remaining fragmentation is taken care of by global garbage collection, which is aligned with the creation of a new MPEG I-frame. Unlike a P-frame, an I-frame does not rely on

temporal coherence, so the garbage collection is free to re-arrange all blocks arbitrarily in order to minimize fragmentation, without hurting MPEG bitrate. The temporal coherence is still ensured between I-frames. We call this process an *atlas reset*. The atlas reset is invoked adaptively: If the memory load exceeds a high water mark, an atlas reset is performed in sync with an I-frame. Clearing of unused blocks in the atlas happens for every I-frame to avoid compression of outdated shading information.

We can store a single color for a patch within the per-patch data structure. This addition is used either when the block size would be a single pixel or when the atlas runs out of memory. If the block size is determined to be a single pixel, it would be inefficient to store the color of that pixel within the atlas, since storing the position in the atlas takes up as much space as the color itself. Furthermore, when the atlas runs out of memory and no block can be allocated for a patch, it can be shaded as a block with a single pixel. This fallback solution ensures that the patch can still be visualized, though not at the desired quality. In addition to this last resort, we try to avoid running out of memory using atlas resets to counteract fragmentation in combination with adapting the global bias for the level selection, described in Section 3.5.

3.4.1 Request phase

The request phase runs in parallel for all patches as outlined in algorithm 1. Blocks of patches that have become invisible are inserted into a block stack corresponding to their level (lines 3 to 5). Patches that have become visible increase an atomic request counter corresponding to their level (line 7) storing the resulting *slot number* $P.s$. Patches that are reallocated, e.g., because their desired block size changed, execute both operations (lines 2 and 6). At the end of the phase, we have determined the total number of blocks required to be allocated for each level, and each patch stores its *slot*, i.e., the value it has drawn from the request counter.

3.4.2 Provisioning phase

The provisioning phase prepares the memory to be allocated by the following assignment phase and operates in parallel with one thread per column width. It determines how many requests from patches can be served from the block stacks. For the remaining patches, it draws free superblocks from a superblock stack. Its main purpose is the allocation and preparation of these superblocks for the following assignment phase. To do so it uses the request counts from the previous phase to provide a list of superblocks to allocate from for each column width. For each possible block height within the given column width, it also provides an offset within these superblocks, marking where allocation may begin, and a block count for the maximum number of allocations. The assignment phase then uses the slot number of each patch to determine the exact location of the block or tries to allocate from the block stack if the slot number is bigger than the available block count.

ALGORITHM 1: Request Phase

```
/* the request phase runs in parallel over all patches that are visible or
   allocated                                                              */
1 for all patches P that are visible or allocated do
    /* the patch should be deallocated as it became invisible or is
       reallocated                                                        */
2   if P.toDeallocate then
      /* the block b assigned to the patch P is pushed onto the block stack
         for level l                                                      */
3     p ← atomicAdd(blockStack[l].count, 1)
4     blockStack[l].entries[p] ← P.b
5     P.b ← INVALID
    /* the patch should be allocated as it became visible or is reallocated */
6   if P.toAllocate then
      /* a slot s is reserved as the number of blocks to be allocated for
         level l is counted                                               */
7     P.s ← atomicAdd(blocksRequested[l], 1)
```

The management of the superblocks over multiple frames increases the complexity of this phase slightly. In most cases, the last superblock drawn from the stack will not be fully filled by the requests. Therefore, we memorize the fill-rate of the last superblock for each level, and top it up in the next frames before requesting a new superblock for that level. Furthermore, within the columns, blocks are allocated in order with decreasing size. This ensures that there will be no alignment problems, such as a block split in half where one half is at the end of one column and the other half at the beginning of the next column. In order to be able to allocate a block of maximum height, i.e. a quadratic block, in the next frame, the remaining rectangular space of the last superblock is therefore deallocated onto the block stack by breaking it into as little blocks as possible.

An example of this procedure is shown in Figure 3.6. In this example, the provisioning phase runs for a column width of 4 pixels with 8×8 superblocks. The job description is given in Table 3.1, where each column represents a block height. The table lists, per height, the requested blocks, the blocks that are free on the block stack, the required blocks, i.e., requested blocks remaining after using up the block stack, and the offset (in height units) from the first superblock.

We start with the largest height 4 and first consider superblock S_1. (a) It already contains three blocks of height 4, which gives us a starting offset of 3. (b) Since we need to allocate space for two blocks b_1, b_2, but only b_1 fits into S_1, we allocate a second superblock S_2 with space for b_2. We proceed to a height of 2 and compute the offset from the starting point, S_1, as $(3+2) \cdot 2 = 10$ blocks of height 2. (c) We need space for seven blocks $b_3 \cdots b_9$, but S_2 has only enough space for six, so a third superblock S_3 is allocated, with space for b_9. (d) For height 1, we start with an offset of $(10+7) \cdot 2 = 34$ blocks and determine that we can fit all remaining blocks $b_{10} \cdots b_{12}$ into S_3. (e) Since S_3 is not completely filled, we

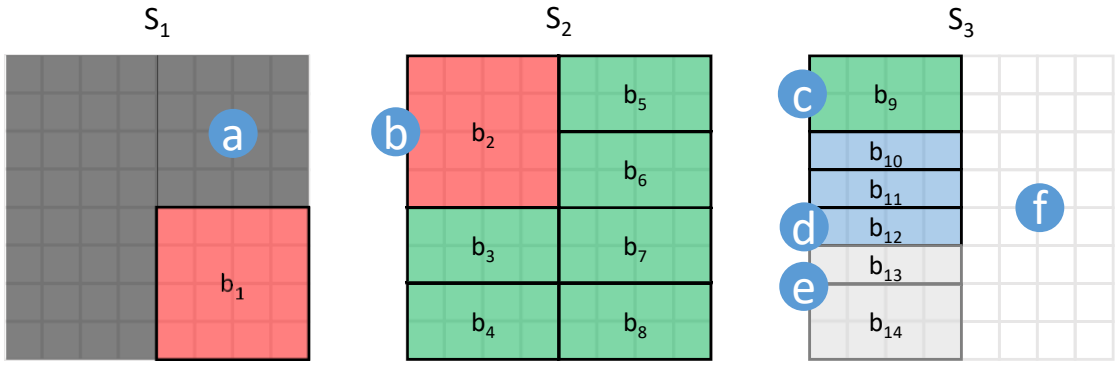


Figure 3.6: Example of the provisioning phase of the parallel memory allocation and the corresponding lines in algorithm 2. (a) The superblock carried over from the last frame S_1 is partially filled (line 2). The actual allocation of superblocks requested for each block size (b to d) is done in lines 3 to 22, separated into three distinct steps: counting how many superblocks are required (lines 3 and 4), allocating the superblocks (lines 5 to 13) and then assigning the superblocks to each level (lines 14 to 22). (b) The allocation of the 4×4 blocks in the first and second superblock S_2 is done in the first iteration of the loops in lines 3 to 4 and 16 to 22). (c) For the 4×2 blocks, another superblock S_3 is provisioned in the second iteration in the loops in lines 3 to 4 and 16 to 22. (d) The 4×1 blocks fit into the remaining space of S_3 (third iteration in the loops in lines 3 to 4 and 16 to 22). (e) Remaining rectangular blocks are put on block stacks (lines 23 to 27). (f) The remaining space in S_3 will be used in the provisioning phase of the next frame (lines 28 to 29).

Table 3.1: Job description for the provisioning example. The data corresponds to the example shown in Figure 3.6. It shows the state before (top rows), values computed during (center rows) and the state after (bottom rows) the provisioning phase.

Height of block	4px	2px	1px
Requested blocks	7	16	10
Free on block stack (before)	5	7	7
Required blocks	2	9	3
Offset (in height units)	3	10	34
Free on block stack (after)	5	8	8
Total available blocks	7	17	11

put the remaining rectangular blocks b_{13} on the block stack for level $(4, 1)$ and b_{14} on the block stack for level $(4, 2)$. This ensures that we can start with a block of height 4 again in the next frame. (f) We memorize that two blocks of height 4 are left in S_3 for the next frame.

The exact procedure of this phase is outlined in algorithm 2. First, the number of superblocks to be allocated has to be computed. We do this by add up the number of pixel

ALGORITHM 2: Provisioning Phase

```
/* the provisioning phase runs in parallel over all block widths                    */
for all block widths l_x do
    /* compute how many superblocks need to be newly allocated                      */
    r ← (carryOver[l_x].offset − BpSB([l_x l_x]^T)) · 2^{l_x}
    for all block heights l_y do
        r += max(0, blocksRequested[l] - blockStack[l].count) ·2^{l_y}
    /* try to allocate as many superblocks as required and fill in the sb
       array                                                                        */
    sb[l_x][0] ← carryOver[l_x].id
    superblockCount ← 1
    for i ← 1 to ⌈ r / BpSB([l_x 0]^T) ⌉ do
        p ← atomicAdd(superblockStack.count, -1) - 1
        if p < 0 then
            atomicAdd(superblockStack.count, 1)
            break
        sb[l_x][i] ← superblockStack.entries[p]
        superblockCount + = 1
    /* compute per level information of the slot range within the sb array and
       store results in the slotOffset and slotCount arrays                         */
    o ← carryOver[l_x].offset
    a ← superblockCount ·BpSB([l_x l_x]^T) − o
    for l_y ← l_x to 1 do
        r ← clamp(blocksRequested[l] - blockStack[l].count, 0, a)
        slotOffset[l] ← o
        slotCount[l] ← r
        blocksRequested[l] ← 0
        o ← 2 · (o + r)
        a ← 2 · (a − r)
    /* put remaining rectangular blocks on their respective stacks                  */
    for l_y ← 1 to l_x − 1 do
        o ← ⌈o/2⌉
        if o is odd then
            p ← atomicAdd(blockStack[l].count, 1)
            blockStack[l].entries[p] ← [o%BpSB(l), sb[l_x][superblockCount - 1]]
    /* store carryOver information for the next frame                               */
    carryOver[l_x].offset = o
    carryOver[l_x].id = sb[l_x][superblockCount - 1]]
```

lines r within the columns (lines 2 to 4) This can also be thought of as the number of blocks of the smallest block size - with one pixel height. Since we carry over the last superblock that has not been completely filled in the last frame, we first consider the remaining space within this superblock (line 2). The *carryOver* offset stores how many square blocks of this size have already been allocated in the block. Using the number of blocks that would fit in the superblock (*BpSB*), we can compute the remaining space. At this point, r is zero or negative, as this space has already been allocated and will be used up first. Before the first frame, the *carryOver* offset has to be initialized with the value from *BpSB* in order to initialize r correctly in the first frame. In the following loop, we add up the number of pixel lines necessary for each block size, counting only requested blocks that cannot be allocated from the corresponding block stack (lines 3 and 4).

Next, we allocate the required superblocks. The *sb* array is started with the carried-over superblock (lines 5 and 6). The following loop (lines 7 to 13) allocates the new superblocks from the superblock stack in the same way allocation works for block stacks in the assignment phase (lines 8 to 12). If we run out of memory, the loop is aborted early (line 11), resulting in less superblocks allocated (*superblockCount*) than requested.

The next step is to fill in the slotOffset and slotCount arrays for the assignment phase. We use an offset variable o that stores the offset and an available block count variable a that stores the number of still available blocks within the superblocks. The variables are initialized with values for the largest (square) block size (lines 14 and 15). In the following loop (lines 16 to 22), the offsets are assigned in the discussed order from largest to smallest block size. The number of blocks that can be allocated from the superblocks needs to be clamped between zero and the number of available blocks (line 17). This value and the offset are then stored in the *slotCount* and *slotOffset* arrays, and the number of requested blocks *blocksRequested* is reset for the next frame (lines 18 to 20). For the next iteration, the offset o and available blocks a are updated and doubled, as the next iteration's block size is halved (lines 21 and 22).

With the arrays *sa*, *slotCount* and *slotOffset* prepared for the assignment phase, the remaining task is to prepare the last superblock for the next frame. The alignment needs to be fixed to start with the allocation of square blocks in the next frame. In a loop, we deallocate any non-square blocks that remain at the end of the last superblock (lines 23 to 27). This is done by looping over the block sizes in the opposite direction - from smallest to largest - and halving the final offset o again (lines 23 and 24), until we arrive back at the square block size. If the offset o cannot be halved without a remainder (line 25), we deallocate a block of the current size (lines 26 and 27) in the same way it is done in the request phase. The modulo operator % is used to calculate the block's index within the superblock. Finally, we store the resulting final offset and the last superblock's id for the next frame (lines 28 and 29).

3.4.3 Assignment phase

The assignment phase runs in parallel for all patches, as outlined in algorithm 3. Based on its slot, each patch determines whether its request is served by a list of superblocks (lines 4 to 6) or the block stack (lines 8 to 13). If the slot is within the number of provisioned blocks, we directly compute the position within the provisioned superblock with the slot number and stored offset. Otherwise, the patch directly draws a block from the block stack. This strategy is not only lock-free, it also deals with provisioning on a superblock level, avoiding tedious bookkeeping during bulk allocations. In contrast, recycling happens on the level of individual blocks, but uses efficient atomically operated stacks to manage the recycled blocks.

ALGORITHM 3: Assignment Phase

```
   /* the assignment phase runs in parallel over all visible patches          */
1  for all visible patches P do
       /* if the patch is allocating a block                                  */
2      if P.toAllocate then
           /* check if the slot falls within range allocating from the superblock
              list                                                            */
3          if P.s < slotCount[l] then
               /* we compute the position in the list of allocated superblocks */
4              s ← P.s + slotOffset[l]
5              i ← ⌊ s / BpSB(l) ⌋
               /* get the corresponding superblock and compute the block id within
                  the superblock                                              */
6              P.b ← [sb[l_x][i], s − i · BpSB(l)]
7          else
               /* we try to allocate from the stack                           */
8              p ← atomicAdd(blockStack[l].count, -1) - 1
9              if (p < 0) then
                   /* if that failed, we are out of memory                    */
10                 atomicAdd(blockStack[l].count, 1)
11                 P.b ← INVALID
12             else
13                 P.b ← blockStack[l].entries[p]
```

To determine which of the two methods is used for a given patch, the *slotCount* for the patch's level $\mathbf{l}$ is compared to the patch's slot number $P.s$ in line 5. If the patch is allocated from a superblock, the patch uses its slot and the *slotOffset* to compute which superblock is provisioned to fulfill its request (lines 4 and 5). The *BpSB(**l**)* function gives the number of blocks a superblock can store for level $\mathbf{l}$ if all blocks were the same level. The slot can directly be used to compute the patch's offset in the superblock (line 6). By reading the superblock index corresponding to the slot from the *sb* array, the allocation is fulfilled (also line 6). The slot number $P.s$ of the patch was determined in the request

phase, but the remaining information about allocation from superblocks has to come from the provision phase. This information includes the *slotCount*, *slotOffset* and *sb* arrays, where the later is shared for all levels with the same block width.

If the slot index $P.s$ points to an entry outside the bounds of the array, it draws a block from the block stack (starting at line 8). The atomic decrement (line 8) returns the position p on the stack of the block that is allocated. If the position p is negative, allocation fails, since there are no blocks left on the stack (line 9). In this case, we revert the atomic decrement (line 10) and remember that allocation failed and the patch will later be shaded with a single pixel stored in the patch data structure (line 11). Otherwise, we simply get the block from the block stack (line 13).

3.5 Level selection

Level selection determines the block size to which a visible patch is mapped. We encode the rectangular block size as 2D vector of binary logarithmic values, the level $\mathbf{l}$. The goal of the level selection is to assign each patch an appropriate share of the texture atlas. Under memory pressure or if other objectives, such as foveated rendering, should be addressed, a *bias* can be applied to this level.

Shading information is stored in atlas space, but applied in screen space. Therefore, the aim of the level selection should be to preserve the shading rates of the triangle in screen space. Choosing the level based on screen-space edge lengths and area fulfills this requirement. We compute the edge length s of an edge as

$$s = \frac{r}{f}\cos^{-1}\left(\frac{\mathbf{v_1}\cdot\mathbf{v_2}}{v_1 v_2}\right), \tag{3.1}$$

where r is the resolution in pixels, f is the FOV, and $\mathbf{v_1}$ and $\mathbf{v_2}$ are the vectors from the camera to the vertices of the edge. This approximation has the advantage of being independent of the view direction, as it assumes a curved image plane.

Given the edge lengths, we can compute area (using Heron's formula) and aspect ratio of the block. The aspect ratio is computed from the maximum edge lengths of those edges that are axis-aligned in the patch layout. To choose one of the discrete block sizes, the binary logarithms of the area, A, and of the aspect ratio, a, are computed, rounded and clamped; a is rounded so that it has the same parity as A, and we arrive at integer values. With A and a, the binary logarithmic width w and height h are computed as

$$\mathbf{l} = \begin{bmatrix} w \\ h \end{bmatrix} = \begin{bmatrix} \frac{A+a}{2} \\ \frac{A-a}{2} \end{bmatrix}. \tag{3.2}$$

We arrive at the level $\mathbf{l}$ with a logarithmic width w greater or equal the logarithmic height h.

The chosen levels must respect the anticipated memory pressure, to avoid running out of memory if too many patches must be allocated. A bias b is derived from the ideal area I

(the sum of all triangle areas, clamped to the smallest and largest block area), and the target atlas space S. We correct the computation by removing the number of 1×1 blocks d that cannot be further scaled down.

$$b = \log_2 \left(\frac{I - d}{S - d} \right) \tag{3.3}$$

On this log-scale, a bias of 0 means that the ideal level is accepted, a bias of 1 means the ideal area is halved and so on. The bias is subtracted from the logarithmic area before rounding and clamping.

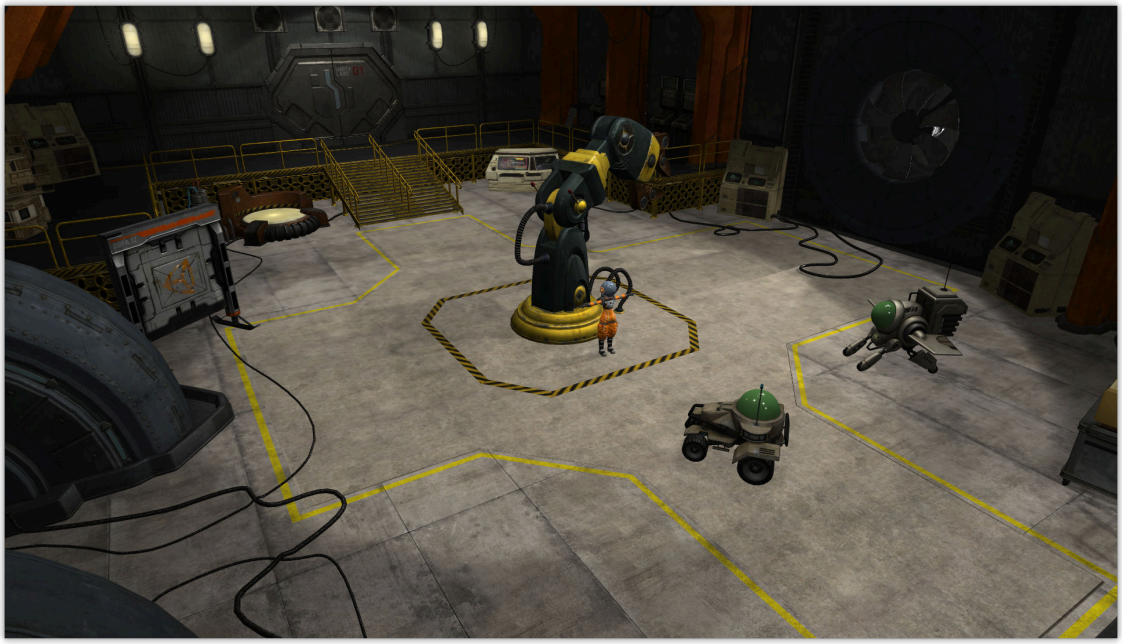
Since the bias is derived from the ideal area and not from the area that is actually allocated, it can happen that the computation under- or overestimates the available space in the atlas. This behavior can be counteracted by adapting the target value S. We calibrate S based on when the atlas runs out of memory. With this adaptive approach, we can reset the atlas every time the atlas runs out of memory or is about to do so, synchronized with an MPEG I-frame in the case of streaming. In such an event, S is also decreased by a fixed factor. If there have not been any atlas resets for a prolonged time, we gradually increase S again.

3.6 Evaluation and results

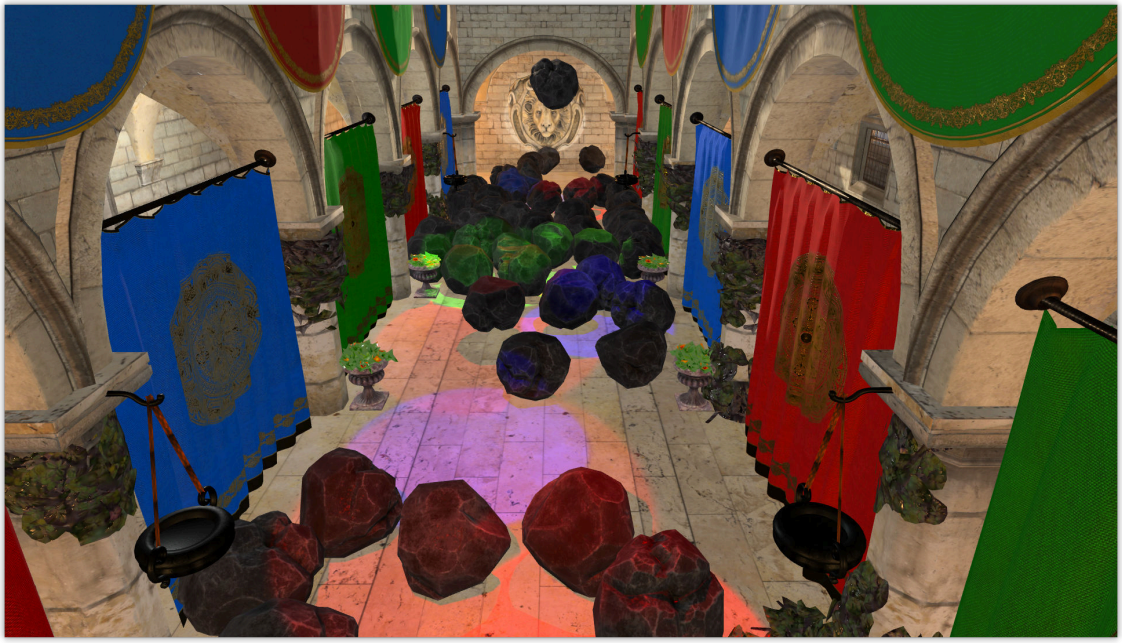
Our evaluation will focus on the design decisions made for memory management within the atlas. We evaluate how much memory needs to be allocated when visibility is determined at a coarser per-object granularity rather than the per-patch. Next, we compare the speed of memory management on the GPU to a CPU version, showing why parallel memory management is necessary. Finally, we compare how the aspects of the memory hierarchy and managements techniques influence memory usage and fragmentation within the atlas, by testing the shading atlas with some of the features disabled.

Our software prototype has been implemented using the Vulkan graphics API and runs under Windows and Linux. Tests within this thesis were conducted on a desktop PC (NVidia GeForce GTX 1080 Ti GPU, Intel i7-4820K CPU 3.7 GHz, 64 GB RAM). For each test scene, we recorded a camera path with a total length of 10 seconds at 120 frames per second (1200 frames). Unless otherwise noted, we used a rendering target with a resolution of 1920×1080 at a horizontal FOV of $90°$.

As test scenes, we use four scenes with physically-based materials, animations, multiple light sources and dynamic shadows as shown in Figures 3.7 and 3.8. *Robot Lab* (Unity), an industrial indoor scene with a robot arm, uses the animations coming with the scene, which include a light shining through a rotating fan and small robots driving along the floor. *Sponza* (Crytek) is the atrium of the Sponza Palace in Dubrovnik and was extended with falling boulders and moving, colored spotlights to create a more dynamic scene. *Space*, a scene made from freely available assets from the Unity asset store, contains a metallic platform in space with space ships, asteroids, moving spotlights and a planet in

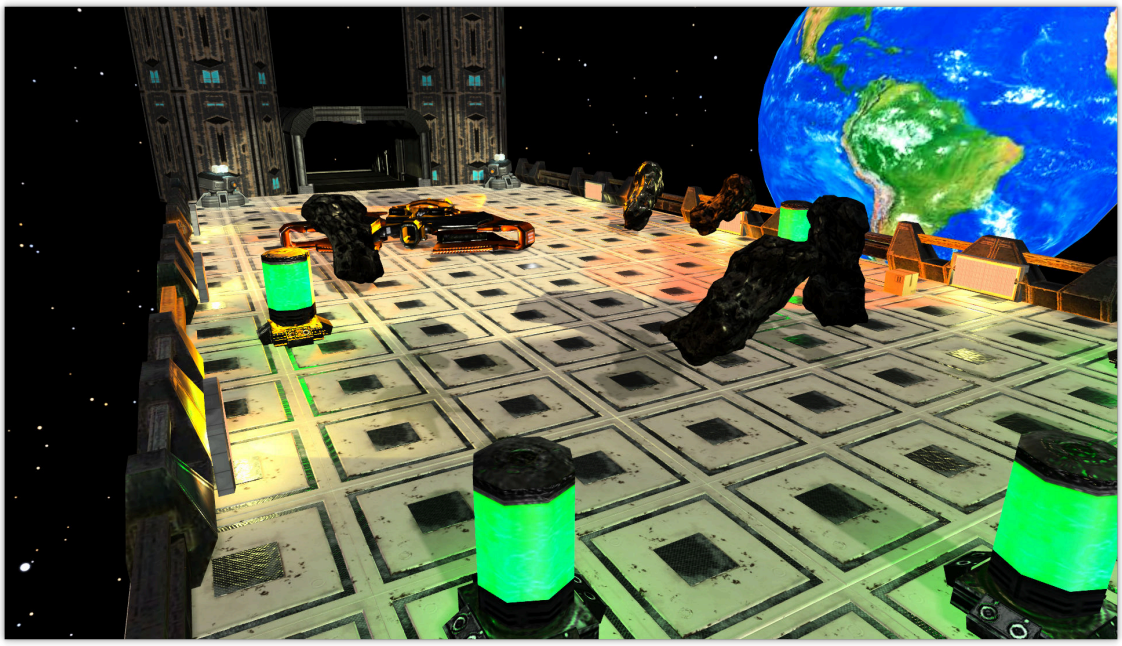


(a) *Robot Lab*



(b) *Sponza*

Figure 3.7: Example views of two test scenes used throughout this thesis. (a) *Robot Lab* contains animated robots as well as a rotating fan, casting moving shadows. (b) *Sponza* is a recreation of the Atrium Sponza Palace in Dubrovnik, modified to include physically animated boulders and animated spotlights.



(a) *Space*

(b) *Viking Village*

Figure 3.8: Example views of two test scenes used throughout this thesis. (a) *Space* models a large platform in space, including space ships, asteroids and animated spotlights. It features large color gradients and multiple moving glossy highlights. (b) *Viking Village* is an outdoor scene of a coastal viking village with animated water. All scenes use physically-based materials.

Table 3.2: Comparison of memory management speed CPU/GPU.

Scene	CPU	GPU	Speedup
Robot Lab	44.66	0.19	235×
Sponza	37.21	0.14	266×
Viking Village	377.20	0.24	1715×

the background. *Viking Village* (Unity) is a big outdoor scene of a viking style village surrounded by mountains and water, which is animated with a vertex shader.

3.6.1 Visibility algorithm vs memory requirements

We were interested in how the visibility computation strategy influences the memory requirements of the shading atlas. In conventional game engines, per-object visibility is often determined on the CPU during view-frustum culling and avoids a potentially expensive GPU visibility pass. However, without occlusion culling, the number of pixels that need to be represented in the atlas is dependent on the depth complexity, and thus not necessarily proportional to the screen resolution. Since streaming performance is sensitive to pixel count, we must strive for a small atlas size.

Therefore, we compare memory requirements (without bias) of per-patch visibility and per-object view-frustum culling. In all our test sequences, culling required a large multiple of allocated pixels compared to per-patch visibility: Robot Lab 16.42×, Sponza 11.25×, Viking Village 95.96×. Since even a factor of 2× would already imply using twice the memory and bandwidth, we conclude that coarse-grained visibility based on culling is not sufficient for shading in the shading atlas and streaming.

3.6.2 Memory management speed

Since fine-grained multi-layer memory allocation has been shown to be fast enough on the CPU for terrain VT [72], we investigated if our memory allocation could be run on the CPU as well, avoiding a more complex GPU-side memory management. For comparison, we implemented a single-threaded CPU memory allocation strategy, which uses a quadtree-like layout to allocate blocks and achieves fragmentation that is always better than or equal to our GPU allocation strategy. We compare the performance of CPU and GPU allocation. We report only the runtime of the memory allocation itself, assuming that all other stages of the pipeline are identical. In favor of the CPU variant, we do not consider the additional transfer times between GPU and CPU. Table 3.2 shows the measured times.

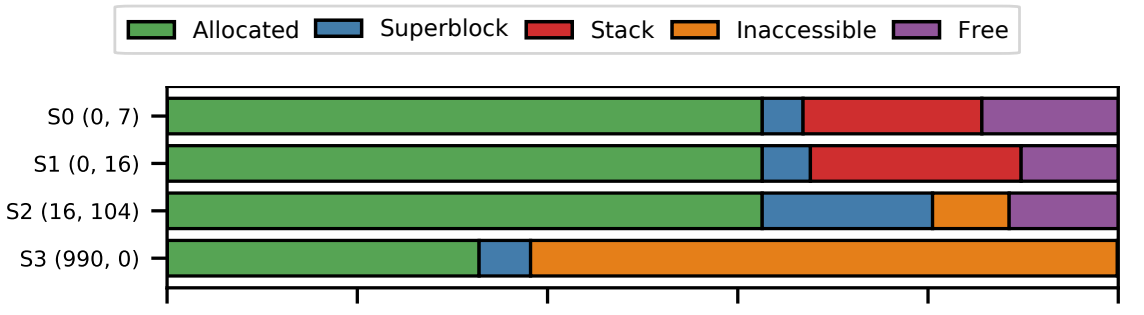
We see that, on average, the CPU variant takes at least 30 ms even for the smallest scene, while the time required by the GPU version is $<$0.25 ms. At a targeted server frame rate of 30 Hz (33 ms per frame), adding an additional 30 ms for memory management

would lower the frame rate to <15 Hz. We therefore rule out the possibility of memory allocation on the CPU.

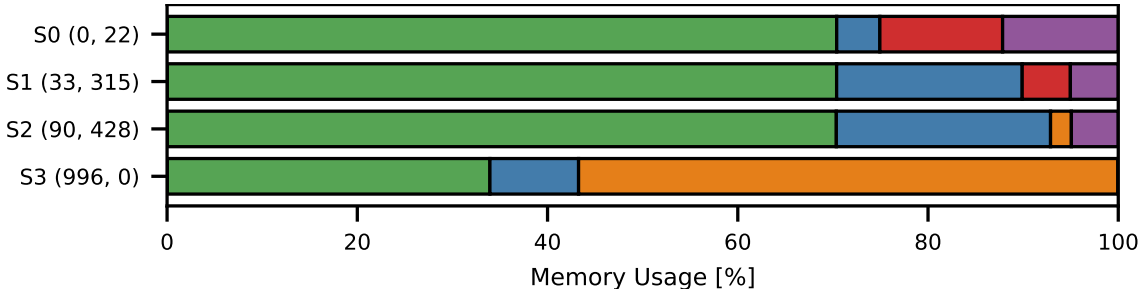
3.6.3 Fragmentation vs allocation strategy

When the atlas size is limited, fragmentation results in excessive memory pressure and reduced image quality. We demonstrate the difference between our recommended strategy, denoted by $S0$, to simpler strategies $S1$ (no columns), $S2$ (no columns, no block stack), and $S3$ (no columns, no block stack, no atlas reset). We compare how memory is used by each method, for an atlas size of 4 MPix. New memory can be either allocated in free superblocks, at the end of partially filled superblocks, or by claiming free areas in the middle of superblocks. We consider the latter two cases as fragmented memory. Free memory in the middle of superblocks is either available on the block stacks ($S0, S1$) or temporarily inaccessible until the whole superblock is freed ($S2, S3$).

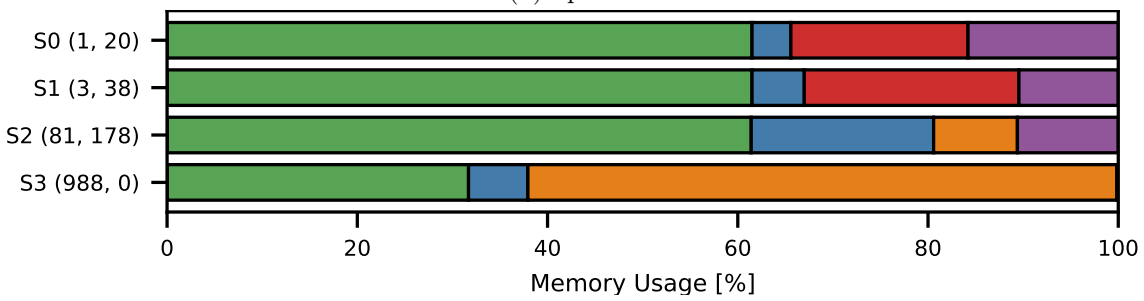
As can be seen in Figure 3.9, the recommended strategy $S0$ shows the least memory fragmentation and requires the least number of atlas resets. It only runs out of memory once across all tests, when we set an aggressive high water mark of 96% for Viking Village. The simpler strategies $S1$ and $S2$ require more atlas resets and run out of memory frequently. Strategy $S3$, which can neither suppress fragmentation nor recover on atlas resets, consistently runs out of memory within the first few frames of the test sequence. We conclude that our preferred memory management strategy $S0$ is mandatory for sustainable operation of the shading atlas.



(a) Robot Lab



(b) Sponza



(c) Viking Village

Figure 3.9: Comparison of the influence of the memory allocation strategy on atlas memory averaged over 1000 frames. Compared are $S0$ (reference strategy), $S1$ (no columns), $S2$ (no columns, no block stack), and $S3$ (no columns, no block stack, no atlas reset). Stated in parenthesis are the number of frames where out of memory occurs and the number of atlas resets. The 4 MPix atlas memory is either allocated (green), at the end of a superblock that is partially filled (blue), within blocks on the block stacks (red), temporarily inaccessible (orange) or within free superblocks (purple). Only $S0$ can reliably maintain a safety margin for all atlas sizes.

Chapter 4

Spatial Shading Reuse

Contents

In this chapter, we will show how the shading atlas can be used for spatial shading reuse. The goal of spatial shading reuse is to reduce the number of shading computations and thus improving performance by reusing already computed shading results spatially. We will discuss how shading can be reused in resolution upsampling, stereo rendering and foveation and how this can be implemented with the shading atlas. Subsequently, we will evaluate the implementation of the approaches and discuss the results. All evaluations will run on the four test scenes, shown in Figures 3.7 and 3.8, averaging results over dynamic camera paths, which are 900 frames long.

4.1 Resolution upsampling

Rendering at a lower resolution and subsequently upsampling the image to a higher resolution is the most basic application of spatial reuse. This can be done with practically any renderer. The quality, however, will depend on the filter that is used for the upsampling operation. The simplest filters are the nearest-neighbor, bilinear or bicubic interpolation filters. More advanced filters, like the Lanczos filter, are typically based on the *ideal*, but impractical sinc filter [14, 103].

Some popular upsampling methods are actually spatio-temporal upsampling methods and thus not purely spatial. Checkerboard rendering [39, 115], for example, renders the white and the black squares of the checkerboard pattern in alternating frames, combining

each image with the previous frame to produce a high resolution image. Modern implementations combine the filtering with TAA, i.e., they combine the current checkerboard image with the full image of the last frame. Deep learning based approaches, such as Nvidia's DLSS [78], make use of the tensor cores on the GPU to spatially upsample the image in real-time with a neural network. The freshly rendered low-resolution image is combined with the previous frame and possibly with motion vectors produced by the rendering system.

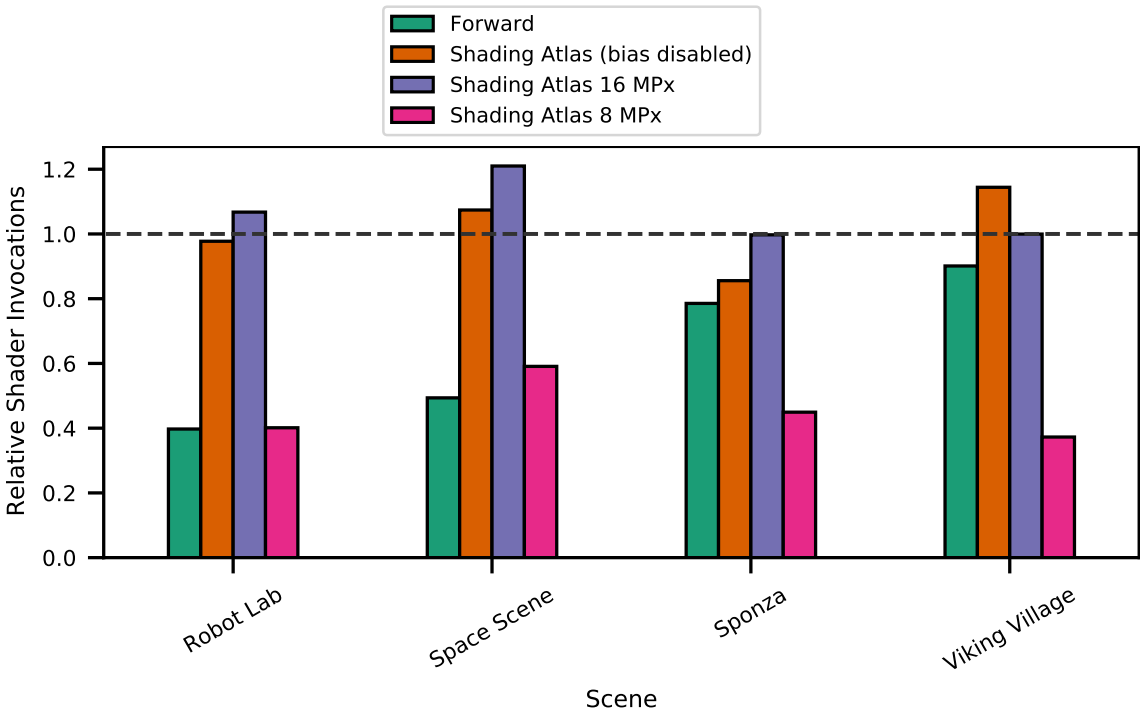
Using the shading atlas for resolution upsampling is straightforward. Since shading happens in object-space, its sampling rate can be completely decoupled from the resolution of the output image. The goal of the level selection process is to choose a resolution that best fits the final image, but this can easily be altered. Therefore, resolution upsampling can be achieved by applying a constant bias to the level selection. Alternatively, when the bias is adaptively changed by the fill rate of the atlas according to Equation 3.3, the spatial shading rate depends mostly on the size of the atlas, and, thus, the spatial sampling resolution directly depends on the atlas size.

To evaluate spatial upsampling, we compare rendering with the shading atlas to forward rendering with Lanczos resampling. The target resolution of this experiment is 4K (3840×2160 pixels). For forward rendering, we render at 1920×1080 pixels and upsample using a high quality Lanczos-3 filter. The shading atlas is utilized in three different variations. The first uses a 64 MPx atlas to ensure that it will not run out of memory, and the global bias is disabled. The other two are using an 8 MPx and a 16 MPx atlas, respectively, with simple bilinear interpolation for the sampling from the atlas for image generation. All approaches render with 4× MSAA.

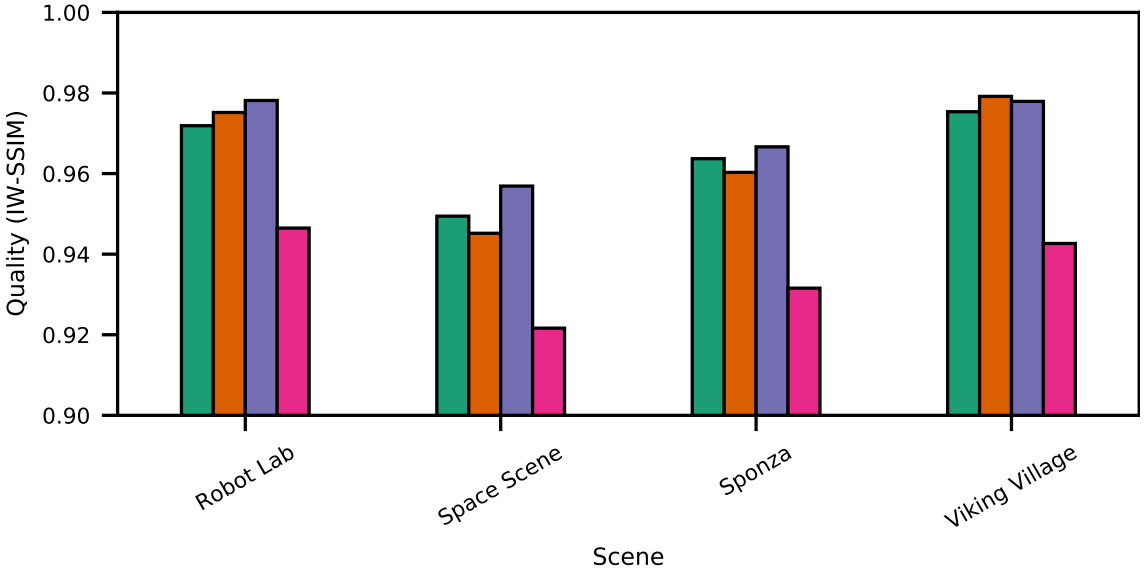
Figure 4.1a shows the number of shader invocations used by each method relative to the target resolution. Depending on the geometric detail and depth complexity of the scene, forward rendering shows considerable overdraw[1] and additional shader invocations caused by MSAA[2], since it renders at 25 % of the target resolution. The images rendered with the shading atlas with disabled bias select the levels for the 4K target resolution, which is kept within a reasonable margin. For the 8 MPx atlas, the bias automatically reduces the size of the blocks in order to prevent running out of memory, and thus number of shader invocations are well below the target, leading to spatial upsampling in the final display stage. The 16 MPx atlas, however, provides enough space for the requested block sizes, and the bias turns negative, increasing the block sizes accordingly in comparison to when the bias is disabled. The atlas size in combination with the bias allows to effectively control the spatial shading rate of the output images, while the final display stage runs at full resolution.

To evaluate the quality of the spatial upsampling, we evaluate our approaches with the IW-SSIM image quality metric [111]. The baseline for this experiment are forward

[1] Overdraw is 28.2 % for *Robot Lab*, 74.1 % for *Space*, 186.7 % for *Sponza* and 192.5 % for *Viking Village*.
[2] MSAA causes 17.4 % for *Robot Lab*, 12.5 % for *Space*, 14.1 % for *Sponza* and 36.5 % for *Viking Village* additional shader invocations.



(a) Shader invocations



(b) Image quality

Figure 4.1: (a) The shader invocations for upsampled forward rendering depend on the scene complexity causing considerable increases due to MSAA and overdrawing. The shading atlas aims to shade at the target resolution when the bias is disabled or tries to use the available space in the atlas as efficiently as possible for shading. (b) While forward rendering with upsampling suffers from undersampling, the shading atlas suffers from the same sampling issues that texture mapping has, which adds to undersampling due to the limited atlas size in the 8 MPx atlas configuration.

(a) Baseline (b) Forward (c) Shading Atlas 8 MPx

Figure 4.2: Forward rendering with Lanczos-3 upsampling shows more visible artifacts at geometric borders. The final display stage of the shading atlas runs at full resolution and therefore has smooth geometric edges. Though some areas such as on the floor appear more blurry with the shading atlas, the overall perception when spatially upsampling with the atlas leaves a sharper impression than an upsampled, forward rendered image.

rendered images at full 4K resolution with MSAA. As shown in Figure 4.1b, the spatial upsampling of forward rendering causes a drop in quality. When rendering without a bias or with a 16 MPx shading atlas, we also observe a drop in quality. Though the number of total samples equals that of the output resolution, the distribution of samples differs, since the triangles are shaded in rectangular, power-of-two-sized blocks in the atlas. This distortion of the triangles in comparison to their screen-space sampling pattern is the cause of the drop in the quality metric. Any texture mapping has the same issue, trying to strike a balance between oversampling in some areas and undersampling in others. The 8 MPx shading atlas also has this issue and renders at a lower resolution, causing a further slight drop in quality in comparison to the other configurations. Figure 4.2 shows a sample screenshot of the baseline 4K rendering, the upsampled forward rendered and the 8 MPx shading atlas. While the quality of the shading atlas image suffers from the low resolution and the distortion, noticeable, for example, on the ground in the striped area, it has higher quality edges due to the final image generation at the target resolution.

4.2 Stereo rendering

Stereo rendering is used to generate stereoscopic imagery to be displayed on devices that can show different images to each of the viewer's eyes. The viewing devices can be screens with special glasses in a cinema, a stereoscopic television set or an HMD and allow viewers

to get a better depth perception. The generation of the images is relatively straightforward with two (virtual) cameras that are positioned just as far apart as the eyes and with either the same viewing direction or a common focus point. Therefore, the contents of a stereo image pair share a substantial amount of shading information.

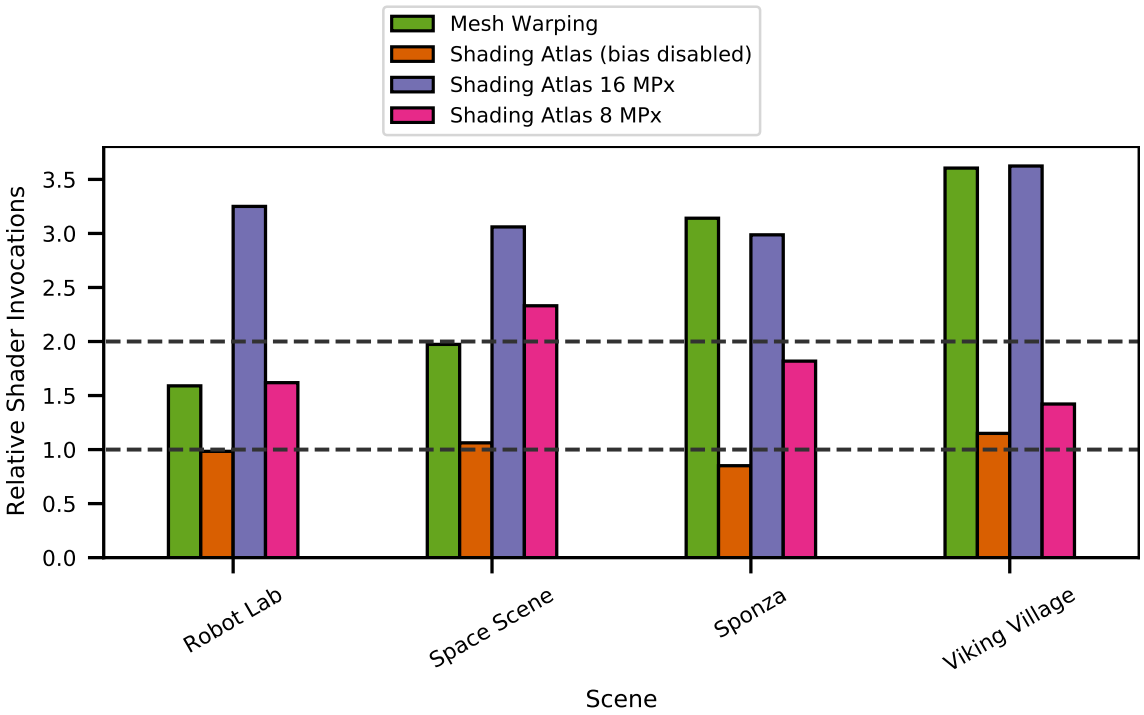
The big overlap of spatial samples between stereo pairs make them a prime application for spatial shading reuse. Screen-space warping methods using depth information can work. Without the depth information, the warping methods cannot generate images that allow a proper depth perception. However, screen-space methods suffer from artifacts in the areas of the image where samples are not shared, such as disocclusions.

The shading atlas can be used for stereo rendering without suffering from disocclusion artifacts. During the visibility stage, it is necessary to make sure that all patches within the view of the two cameras are marked as visible. This can easily be implemented by determining the EVS for both eyes and combining them to be shaded in the shading stage. The shading stage shades visible geometry only once for both views of the stereo pair. The final display stage can either be implemented with two geometry passes, one for each eye, or in a deferred rendering style fashion, using the two index buffers generated in the visibility stage.

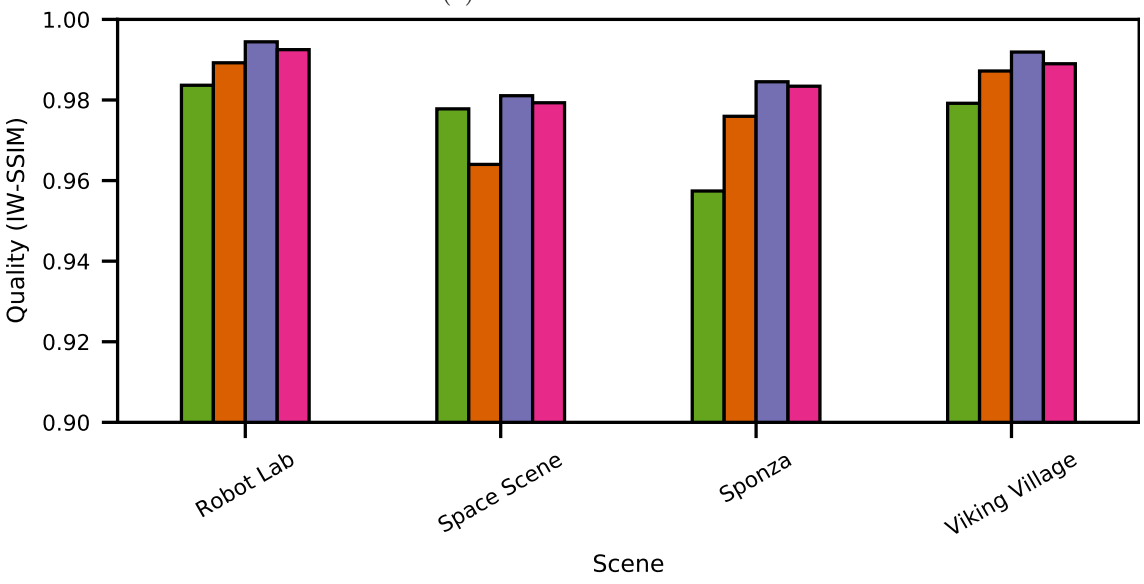
To evaluate spatial shading reuse for stereo rendering, we compare the same shading atlas configurations from the previous section with an image warping method. Mesh warping [56, 69] generates a mesh out of the depth buffer and texture maps the image onto the mesh for novel view synthesis. In stereo rendering, this means that one eye is forward rendered and the view for the second eye is generated by warping the forward rendered image. Each eye buffer for this test is rendered at 1920×1080 pixels. Forward rendering of both eye buffers provides the baseline for the evaluation of the quality of each of the approaches.

In Figure 4.3a, we see the shader invocations relative to the total number of pixels in one image of the stereo image pair. Mesh warping, using a forward rendering pass for the image generation, has exactly the same number of shader invocations as forward rendering in the previous section due to MSAA and overdraw, though only one eye buffer is rendered and later warped. Targeting the resolution of an eye buffer, the shading atlas with a disabled bias shades around 50 % of the total pixels of the stereo pair, since it can reuse the shading for both eyes. The other two shading atlas configurations have a higher number of shader invocations due to the bias turning negative, which increases the resolution of the blocks in the atlas. This increase is limited to a maximum of quadrupling the resolution of each block to ensure that no severe oversampling occurs.

Comparing quality, shown in Figure 4.3b, we see that the shading atlas approach outperforms mesh warping in all cases but one. The drop in quality of mesh warping is caused by the disocclusion artifacts, when one eye buffer is warped to the other. In other terms, mesh warping cannot correctly render areas of the image that are not visible for the forward rendered eye, but would be visible for the other eye. Rendering with the shading atlas does not suffer from this issue, since it shades all geometry that is visible from either eye. As



(a) Shader invocations



(b) Image quality

Figure 4.3: (a) The shader invocations for stereo rendering with mesh warping depend on the forward renderer due to MSAA and overdrawing. The shading atlas aims to shade at the target resolution, when the bias is disabled, and increases the resolution to utilize the available space in the atlas, when the bias is enabled. (b) Though the shading atlas suffers from the distortions introduced by the sampling pattern in the atlas, it shows a better quality than mesh warping, which cannot accurately reproduce disoccluded areas when warping the image of one eye to the other.

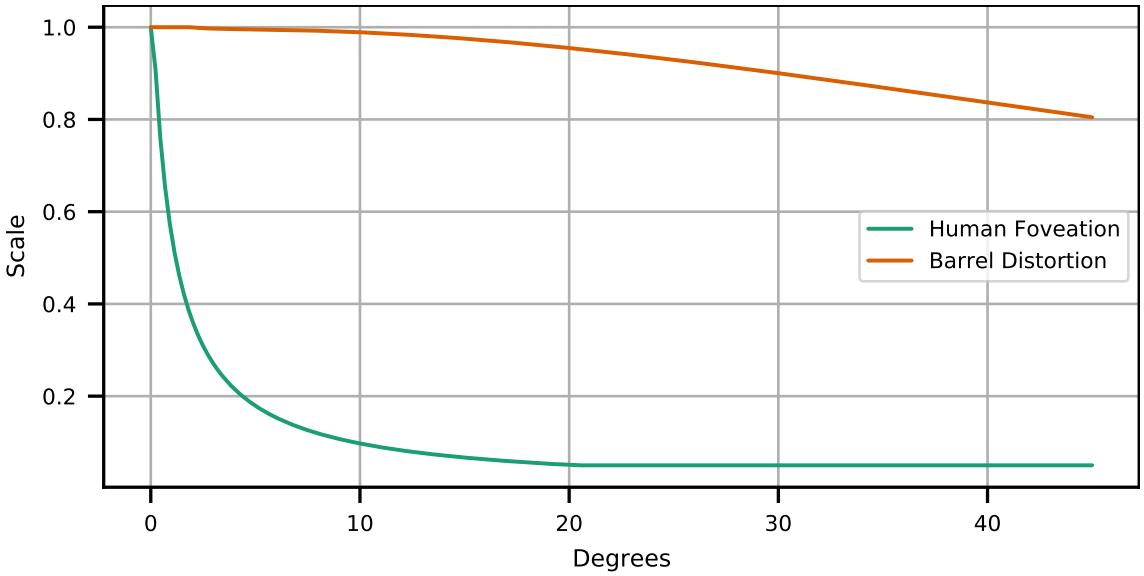


Figure 4.4: Scaling functions for two different effects depending on the radial distance from the center in degrees. Human foveation shows how the spatial resolution of the eye decreases with the distance from the fovea, the center of the human field of view. The scaling function for barrel distortion targets the barrel distortion, which necessary to be computed for the Oculus Rift in order to counteract the distortion of the lenses in the HMD.

in the previous section, quality is slightly reduced when comparing to a forward rendered image due to the different sampling patterns. Oversampling, as done in with the shading atlas configurations with enabled bias, counteracts this issue to some extent, at the cost of increased shader invocations. However, rendering with a disabled bias still provides good quality results, superior to mesh warping in terms of quality and shading workload.

4.3 Foveated rendering

In contrast to computer screens and other displays, the human visual system has a non-uniform resolution with a higher resolution in the fovea and a lower resolution in the periphery of the visual field. Consequently, foveated rendering estimates the area around where the viewer focuses with an eye tracker, and renders it at a higher resolution, while the periphery is rendered at a lower resolution. Since the displays have a uniform resolution, it is not straightforward to render the periphery at a lower spatial sampling rate, while gaining a performance benefit. In order to deal with this issue, Nvidia introduced variable rate shading, a special hardware extension that allows to set different spatial sampling rates for shading.

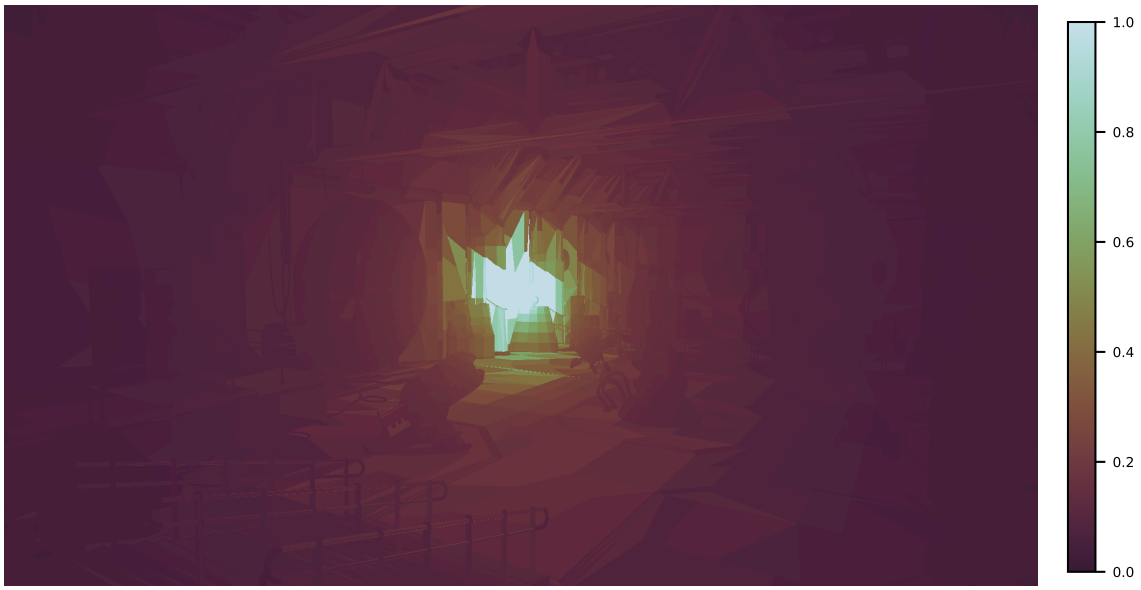


Figure 4.5: The scaling factor s for foveated rendering depends on the reciprocal of the eccentricity of the patches from the focal center and drops rapidly near the center. The conservative estimate of the eccentricity may cause slithery patches to be shaded at a higher resolution than necessary.

The physiological background for foveated rendering is based on the research into visual acuity [97]. The reciprocal of visual acuity can be approximated with a linear model

$$\omega = me + \omega_0, \tag{4.1}$$

where ω is the minimum angle of resolution (MAR); m, the MAR slope; e is the eccentricity, and ω_0 the smallest angle that is still resolvable at the center of the fovea. Guenter et al. [41] use this equation to define a scale

$$s = \frac{\omega}{\omega^*}, \tag{4.2}$$

based on the display's resolution ω^* in terms of degree per cycle, where one cycle corresponds to two pixels. This function is shown in Figure 4.4.

While Guenter et al. [41] use Equation 4.2 to determine the eccentricity at which they can define layers at which the resolution is reduced for rendering, we can directly use the scale during the level selection process. The level selection process determines the spatial resolution at which each patch will be shaded. Thus, we simply multiply the squared reciprocal of the scale s to the area of a patch to enable foveated rendering. No special GPU extensions are required for implementing foveated rendering with the shading atlas.

In order to determine the eccentricity e for each patch, we conservatively estimate the minimal angular distance of the patch from the user's focus point. This process first

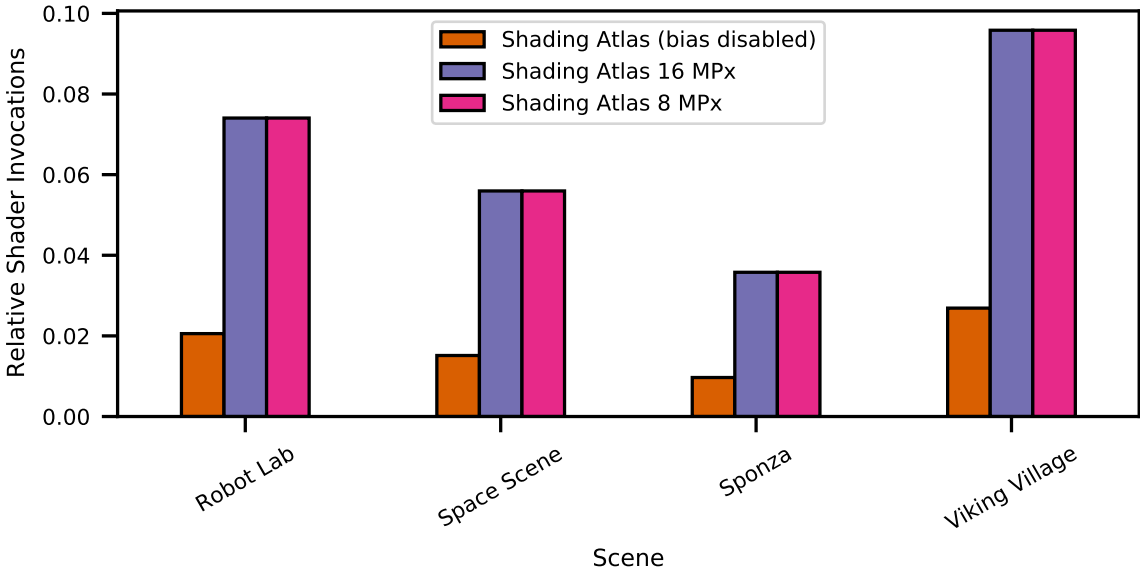


Figure 4.6: Foveated rendering manages to save up to 98 % of shader invocations in comparison to the display resolution. When the bias is enabled, the memory management of the shading atlas increases the resolution by up to a factor of four. This happens with the 8 MPx and 16 MPx shading atlas for all patches, since there is plenty of space left in the atlas due to foveated rendering.

determines a patch center by computing the average of all normalized vertex coordinates (thus lying on the unit sphere) in view space. We compute the maximum angle $\alpha_{\max}$ as the angle between the averaged center and the vertices to determine the eccentricity as

$$e = \max\left(\alpha_c - \alpha_{\max}, 0\right), \tag{4.3}$$

where α_c is the angle between the patch center and the focal center. Figure 4.5 shows an example of the final scaling s.

To evaluate our implementation, we measured the shader invocations of the same shading atlas configurations as in the previous sections with enabled foveated rendering. We choose $m = 0.022$ and $\omega_0 = 1/48°$, as determined by Guenter et al. [41], and render the test sequences at 1920×1080 pixels with the foveation center in the middle of the image. Figure 4.6 shows the shader invocations in relation to the full image resolution. With the bias disabled, only about 2.5 % as many pixels are shaded as are displayed on the screen. When the bias is enabled, the resolution determined by the level selection can at most be quadrupled, if the size of the atlas and the visible geometry allows it, which is the case for the 8 MPx and the 16 MPx shading atlas configurations. At a constant atlas size, together with the bias, this means that more pixels can be spent on the important parts of the scene. Alternatively, if the atlas is big enough or the bias is disabled, the reduced shading load leads to an increase in performance.

Unfortunately, there is no image error measurement that considers foveation yet. While we consider this problem out of scope for this thesis, there are a few ways that future work could implement this. One way would be to generate foveated reference images as a baseline for comparison with various image quality metrics. The disadvantage of this approach is that the image quality metrics would weigh differences to the reference in the central region as important as in the peripheral regions. Thus, a better solution would consider the varying resolution during image comparison. A possible implementation could be based on multiscale SSIM [112], considering scales depending on the eccentricity.

4.4 Discussion

We have shown the capabilities of rendering with the shading atlas reusing shading samples spatially in three application scenarios. For a final experiment, we combine the experiments to a single final experiment. Instead of human foveation, we will use a scaling function for the barrel distortion based on the Occulus Rift's barrel distortion, which is necessary to get a correct image when looking through the HMD's lenses. To map from a distorted point $\mathbf{p}_{\text{distorted}}$ to the undistorted point $\mathbf{p}_{\text{undistorted}}$, the Oculus Rift uses the equation

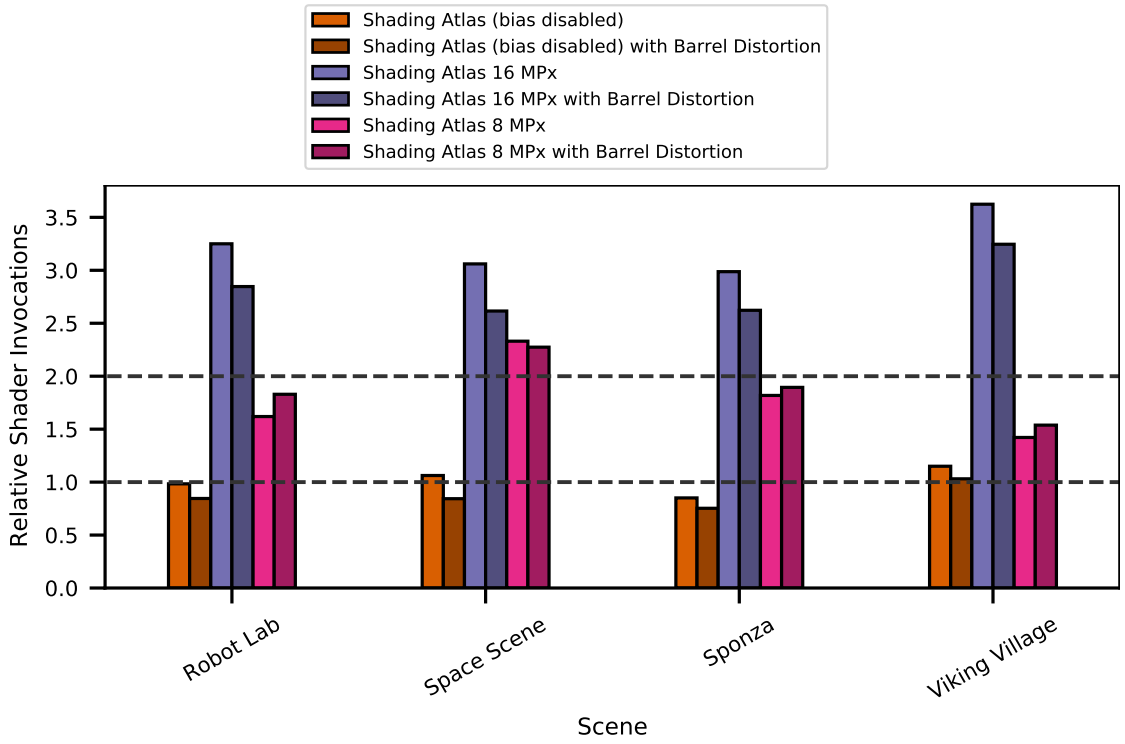
$$\mathbf{p}_{\text{undistorted}} = \mathbf{p}_{\text{distorted}}(1 + 0.22\,||\mathbf{p}_{\text{distorted}}|\,|^2 + 0.24\,||\mathbf{p}_{\text{distorted}}|\,|^4), \tag{4.4}$$

where the coordinates of the distorted point is within ± 1. In order to use transform this equation to use it for scaling in our level selection instead of the foveation function, we need to adapt the function and accordingly. In order to do so, we fit a fifth order polynomial to the radial version of Equation 4.4 in the range 0 to 1 for radial distance from the center, which is given as $||\mathbf{p}_{\text{distorted}}|\,|$. To arrive at the final scaling function

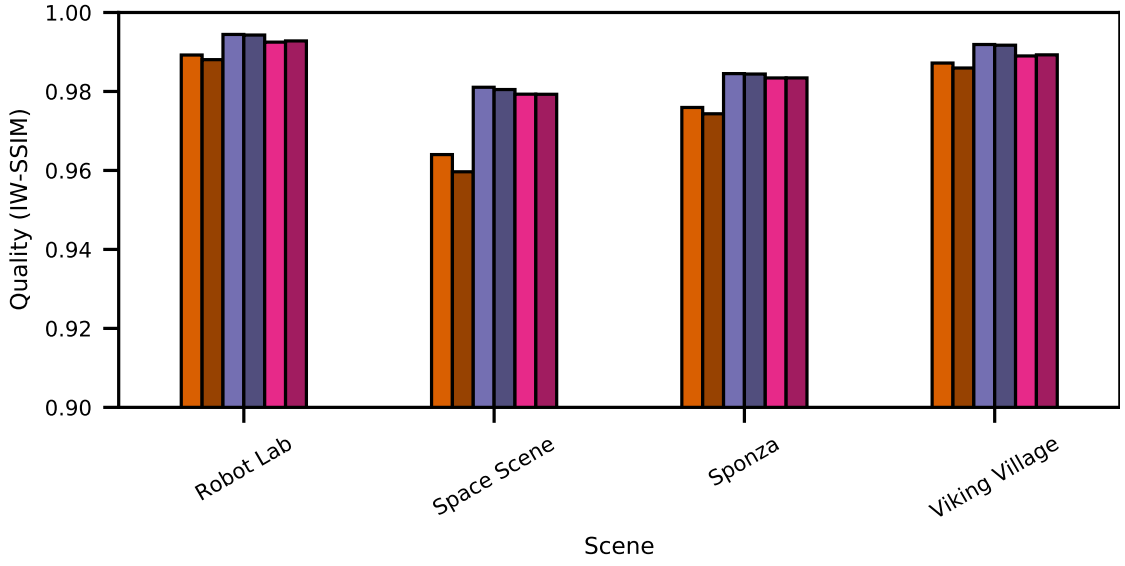
$$s\,(n) = 5.4 \cdot 10^{-4} n^{-1} + 0.98 + 0.13n - 0.56n^2 + 0.32n^3 - 0.06n^4, \tag{4.5}$$

we divide the polynomial by n, the normalized distance from the center, and clamp it to the range 0 to 1 to turn the transformation from distorted to undistorted distance to a scaling value. Since the level selection measures distances in angles, we compute n based on the angle from the center and normalized by the field of view. This scaling function is shown in Figure 4.4, together with the human foveation function.

Figure 4.7 shows the results of this final experiment, rendering stereo image pairs with different shading atlas configurations and with and without the described scaling based on the barrel distortion. The scaling for barrel distortion reduces target resolution in the level selection of the shading atlas depending on the distance of the patch from the image center. When the bias is disabled, shader invocations are lowered effectively. This is also true for the 16 MPx atlas, where low memory pressure causes the bias to approach its minimum and thus scale up all patches equally. However, for the 8 MPx atlas, the barrel distortion can even increase shader invocations due to different block sizes being allocated



(a) Shader invocations



(b) Image quality

Figure 4.7: (a) The barrel distortion scaling reduces target resolution in the level selection of the shading atlas slightly, depending on the distance of the patch from the image center. The combination of bias and differences in the memory fragmentation can nevertheless lead to an increase of shader invocations, as the 8 MPx shading atlas shows. (b) Image quality correlates with the shader invocations. Ideally, the barrel distortion scaling would not cause any difference in quality, but the discrete choice of block sizes in the atlas and corresponding sampling effects still influence image quality.

in the atlas and resulting lower fragmentation. In correlation with the shader invocations, the higher the shader invocations are, the higher the image quality is. Ideally, the barrel distortion scaling would not cause any difference in quality, but block sizes can only change by factors of two, causing a considerably different sampling pattern, which slightly changes the overall image quality.

(a) Mesh warping (b) Shading atlas

Figure 4.8: Slight camera movements can already cause disocclusions, such as the wall behind the robotic arm. Mesh warping [56, 69] reconstructs a mesh from the depth buffer with drawn-out triangles along depth discontinuities, causing artifacts. In contrast, the shading atlas does not suffer from disocclusion artifacts, as long as the disoccluded geometry resides in the PVS.

Overall, we observe that shading workload of the shading atlas is proportional to the screen space resolution of the geometry, but not necessarily equal to the sceen space resolution. With the bias enabled it is primarily proportional to the atlas size. Reusing shading spatially to create stereo image pairs is simple, requires no additional shading and, in contrast to image-based warping methods, does not suffer from disocclusion artifacts, shown in Figure 4.8. Finally, varying spatial resolutions and thus spatial shading reuse can be implemented by altering the level selection for each patch, as we have shown by scaling the block sizes based on foveation and barrel distortion.

Chapter 5

Temporal Shading Reuse

Contents

We will discuss the concept of temporal shading reuse in this chapter. Since many of our research questions relate to perceptual issues for which no acceptable technical metrics exist, we evaluated our approaches with user studies, for which we will first introduce a general framework. In the first user study, we will determine when users notice that shading information is outdated, depending on the color difference between current and outdated shading. Using this information, we evaluate how many samples can be reused for a given number of frames without exceeding a specific color difference. We will then discuss in detail how changes in shading can be predicted, which can be used to determine how often shading is necessary. All these findings are the foundation of temporally adaptive shading (TAS), which is discussed in the following chapter.

5.1 User study

To allow for recreation of our experiments, we first provide details about the user study from which the following results are derived. All results were obtained within a single study, randomly shuffling techniques and conditions to avoid learning effects. 34 participants were shown two video clips, of which one is generated with forward rendering as ground truth reference, and the other is generated with the tested rendering technique. The participants

were asked to rate the relative quality of the two video clips, following a pairwise comparison design [54, 68]. As the order of clips was randomized, they did not know which clip was the reference. From the rating, we compute an average relative quality score (Q), ranging from -2 to 2, where 2 means the reference is *significantly* better and 1 *slightly* better. 0 indicates that they have been rated equal. Additionally, we compute the probability p_{ref} of choosing the reference over the reuse approach. A p_{ref} of 50 % indicates that there is no difference between the approaches; p_{ref} of 75 % is referred to as 1 just-noticeable-difference (JND) unit [68]. Staying under 1 JND is considered high quality. For statistical analysis, we use repeated-measures ANOVA and Bonferroni adjustment for post hoc tests.

5.1.1 Task and conditions

Videos were presented on a 24" monitor at 1920×1080 resolution and 60 Hz refresh rate. Participants were seated approximately 60 cm from the monitor. Videos were encoded with H.264 using very high quality settings, ensuring that visual quality was not diminished by encoding artifacts and that playback was running at 60 Hz with no exceptions. The room lighting was dimmed, and monitor brightness was set to high.

The six test techniques were temporal forward rendering (TFR), shading atlas with uniform frame rate upsampling (SAU), reprojection caching with uniform upsampling (RRC), temporally adaptive reprojection cache (TARC), shading atlas (SA) and temporally adaptive shading atlas (TASA); these methods were compared to a Forward+ baseline. The exact functioning of these techniques will be explained in the following sections. For SA, we tested a single configuration, for TFR, SAU, RRC, TARC, and TASA, we varied a single parameter (3 to 4 parameter settings each). For SA and TASA, we tested another independent parameter (the atlas size) in two configurations. We used four camera paths for each of three scenes, i.e., 12 clips per condition. Thus, every participant had to rate 288 pairs of video clips.

As test scenes, we use three scenes with physically-based materials [88], animated models, animated light sources and dynamic shadows. *Robot Lab* uses the animations coming with the scene, which also include a light shining through a rotating fan which overall has a low number of dynamic changes. *Sponza* was extended with falling boulders and moving, colored spotlights leading to a moderate amount of changes within the scene. *Space* is the most dynamic of the scenes, reflecting games with many moving objects and changes on screen at the same time. The lighting in all scenes is based on standard point, spot and directional lights with shadows rendered using shadow mapping. For each scene, we created four 5-second long test animations, two with a stationary camera and two with a moving camera varying between 0.5 m/s to 6 m/s, which resembles a range of motion from slow walking up to fast running, while including fast head rotations. Rendering was restricted to 60 frames per second, the rate of the displays used in the user study. A higher frame rate, such as the 90 frames per second recommended for VR, would lead to even more potential for shading reuse, provided animation frequencies remain unchanged.

5.1.2 Procedure and participants

Each participant started by reading the printed set of instructions and filling a first demographic questionnaire that also inquired their experience with visual media in two categories: *games* ("How much experience do you have with computer games?") and *movies* ("How much experience do you have with movies containing computer generated imagery/content?"). Both questions were answered on a five-point Likert scale. Participants were not instructed as to what differences they should look for in the videos.

The main study was set up as a standalone application that automatically played pairs of video clips. Each five-second video was shown only once, with a neutral gray blank screen (1 s) preceding it. Afterwards, the rating scale was shown on screen with no time limit, and participants were prompted to make their choice. After choosing, the next pair of video clips was started. The entire study lasted for approximately 90 minutes. To keep participants focused, the test system enforced three breaks of at least two minutes.

The order in which individual video pairs were shown was systematically randomized. For each technique and parameter setting, the order in which the ground truth was shown (first or second clip) was counterbalanced to avoid bias.

We recruited 34 participants (aged 23 to 38, 25 males, 9 females) from local universities. The participants were from the fields of computer science, economics, medicine, and education. The average familiarity with both *games* and *movies* was 3.74. All participants had normal or corrected vision and reported no history of visual deficits, like color-blindness. All participants provided written consent before taking part in the study and were instructed that they could stop the experiment at any point, which no one did.

To assess whether familiarity has any influence on the differences seen by participants, we computed the Spearman's correlation between Q over all conditions and familiarity scores. In both cases we found a correlation: *games* ($r_s = .068$, $n = 34$, $p < .001$), *movies* ($r_s = .039$, $n = 34$, $p < .001$). Note that, although the results are significant, effect strength is very weak. However, it might hint at the fact that familiarity with visual content helps to identify artifacts. We did not find a correlation between Q and gender.

To analyze the quality of different techniques, we compute the quality score (Q) from the participant's answers, with 0 meaning the videos were identical, 1, that the reference was "slightly better", and 2, that the reference was "significantly better"; i.e., lower is better for the tested condition. To depict the quality progression over a varied parameter, we plot Q with 95 % confidence interval. To capture when two conditions are noticeably different, the JND is used. A JND of 1 is usually assumed if 75 % of the population rate one condition higher than the other [68]. To this end, we compute the probability (p_{ref}) as the integral over the t-distribution function of Q in the range $[0; +\infty]$, i.e., the probability of rating Forward+ as better than the tested condition when forcing a choice between the two. Obviously, for two identical conditions, p_{ref} would be 50 %. To compare individual techniques at specific parameter settings, we use repeated-measures ANOVA.

If more than two conditions were compared, we repeat ANOVA for post hoc tests with Bonferroni adjustments. As confidence interval, we use 95 % in all statistical tests.

5.2 Perception of shading differences

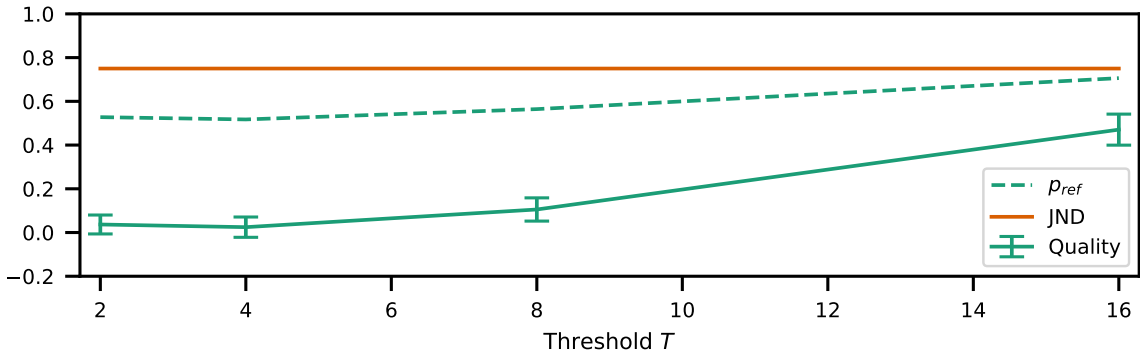
To determine the perceived quality reduction caused by reusing shading samples, we implemented an experimental method named *temporal forward rendering* (TFR) that decouples the temporal changes in shading from other major effects that influence the shading reuse ability: temporal changes in visibility and in spatial sampling. TFR uses a modified fragment shader to compute shading as if a fragment was shaded at a specific time in the past. We recreate all input parameters to the shader, including view, light and model matrices, textures and shadow maps for up to 120 frames (2 s) in the past and compare the shading results to the new shading. If the difference of a shading sample (r, g, b) is above a certain threshold T, i.e., $T < \max(|\Delta r|, |\Delta g|, |\Delta b|)$, we consider the shading to be changed. Shading, including gamma-correction and possibly tone-mapping, is computed and compared in floating point. However, we use clamped integer values between 0 and 255 for the color components r, g and b, as well as the threshold T in the following discussion, since these values represent the actual bit depth of the final output. This threshold is an approximation of Weber's law [12], which states that the just noticeable luminance difference is constant in relation to the base luminance.

To determine the limits of keeping shading over multiple frames in scenes with advanced shading and animation, we conducted a controlled user experiment with TFR as tested technique. As shown in Figure 5.1, reusing shading samples that are slightly different does not reduce perceived quality. For $T = 2$ and $T = 4$, Q cannot be separated from 0.0 with confidence, and p_{ref} is nearly 50 %. At a threshold of $T = 8$, the mean quality is above 0.0, indicating that some participants see a minor quality deterioration. The distribution is still nearly balanced, with $p_{ref} = 55\,\%$. At $T = 16$, the distribution is just shy of 1 JND ($p_{ref} = 75\,\%$). Q is close to 0.5, indicating that, on average, every second participant would rate the difference as "slight". For highest rendering quality, we conclude that *temporal artifacts of less than* 8 *after quantization, i.e., about* 3% *maximum absolute color difference, are virtually not noticeable by users.*

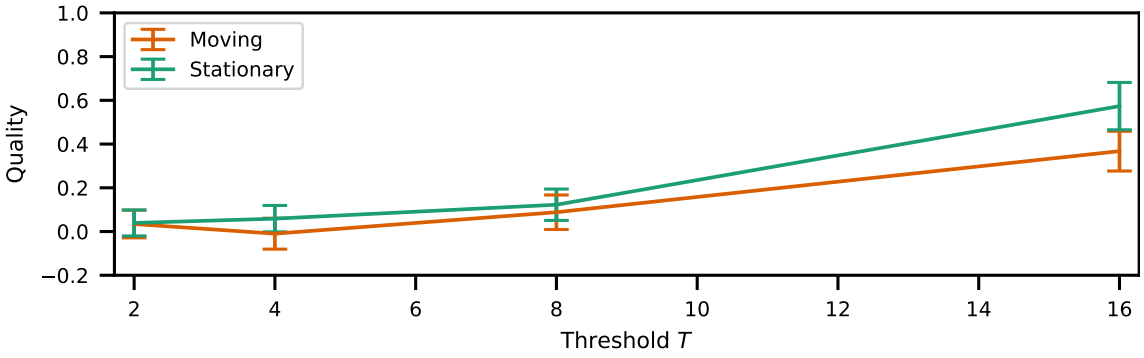
To gain deeper insight, we break down the measurements concerning *Camera* movement and *Scene*. For *Camera*, we found a difference in Q for $T = 16$[1]; the other thresholds are not significant. For *Scene*, we found that there is a difference between *Robot Lab* and *Space* (for $T = 8$) as well as *Robot Lab* and *Sponza* (for $T = 8$ and $T = 16$)[2]. Using a larger threshold reveals differences for when shading reuse can be detected. When the camera moves, the image undergoes perspective changes, and it becomes more difficult

[1] Q for $T = 16$: $F(1, 33) = 9.506$, $p < .005$

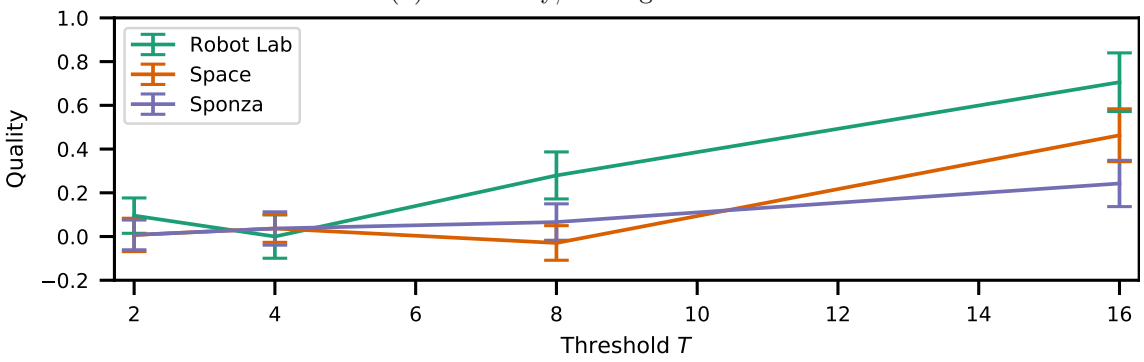
[2] Q for $T = 8$: $F(2, 66) = 7.348$, $p < .002$, post hoc *Robot Lab* and *Space* ($F(1, 33) = 10.064$, $p < .005$); *Robot Lab* and *Sponza* ($F(1, 33) = 8.1441$, $p < .01$). Q for $T = 16$: $F(2, 66) = 10.499$, $p < .001$, *Robot Lab* and *Sponza* ($F(1, 33) = 24.972$, $p < .001$)



(a) Overall result



(b) Stationary/moving cameras



(c) Scene type

Figure 5.1: Participants of our user study rated the quality of rendering with outdated shading using temporal forward rendering (TFR), with a color difference up to a threshold T, in comparison to standard forward rendering. The relative quality score (Q, with 95 % confidence interval) is very close to zero up to the color difference threshold $T = 8$. p_{ref}, the probability that participants choose forward rendering over TFR, is still close to the balanced 50 %. Only at $T = 16$, the probability of detecting a slight difference increases and p_{ref} approaches the just-noticeable-difference threshold (JND). The quality for stationary and moving cameras slightly differ for higher T. Slight differences can also be observed across scenes.

to see shading imperfections. In contrast, a stationary camera lets the user focus on animated objects, making it easier to notice local shading variations. This assumption was also reassured in our post-study interviews. ("The scenes with moving cameras were more challenging.").

Similar results can also be drawn from *Scene*: *Robot Lab* has the least amount of reflections and abrupt movements. Shadow boundaries move slowly and steadily; there is little overlap between moving objects, making it easy to spot errors. *Sponza* contains a large amount of movement, owed to moving spotlights and falling boulders. *Space* is shiny, with large brightness gradients drawing attention.

Combining the results from the in-depth analysis indicates that scenes like *Robot Lab* may have the highest reuse potential, but users may spot quality issues earlier. More dynamic scenes have lower reuse potential, but spotting shading imperfections is also more difficult. Thus, it may be possible to use more aggressive reuse settings in situations where more reshading is required, leading to an overall balanced reuse potential across scenes. One should bear in mind that the study participants were actively looking for differences in shading quality. Distracted observers, such as users engaged in a game, may tolerate even higher thresholds.

5.3 Temporal coherence for shading reuse

Many rendering acceleration techniques exploit temporal reuse, but do not provide proof of the method's quality beyond informal comparisons. This leaves the question unanswered to which extent or for how long shading results can be reused in practice.

5.3.1 Temporal coherence of visibility

To answer the question in detail, we start by recreating the experiment by Nehab et al. [75] on the degree of temporal coherence found in visibility, applied to our test scenes. We project every sample of a current frame back to the previous frame and determine whether the sample was visible before. In accordance with Nehab et al. [75], over 90 % of samples stay visible between frames, as can be seen in the box plot in Figure 5.2. The most significant visibility disruption is caused by large camera movement or fast moving objects. For instance, the falling boulders in *Sponza* can lead to visibility changes of up to 50 % of the samples.

5.3.2 Temporal coherence of shading

To analyze the potential ideal reuse of shading samples, we rely on the results of our first user study and analyze the number of required fragment shader invocation using TFR with a threshold of $T = 8$. Figure 5.3 shows that, in most cases, even for a complex scene such as *Space*, more than 74 % of the shading results stay valid (under $T = 8$) for more than 120 frames on average. Being the scene with the least dynamics, *Robot Lab* stays at above

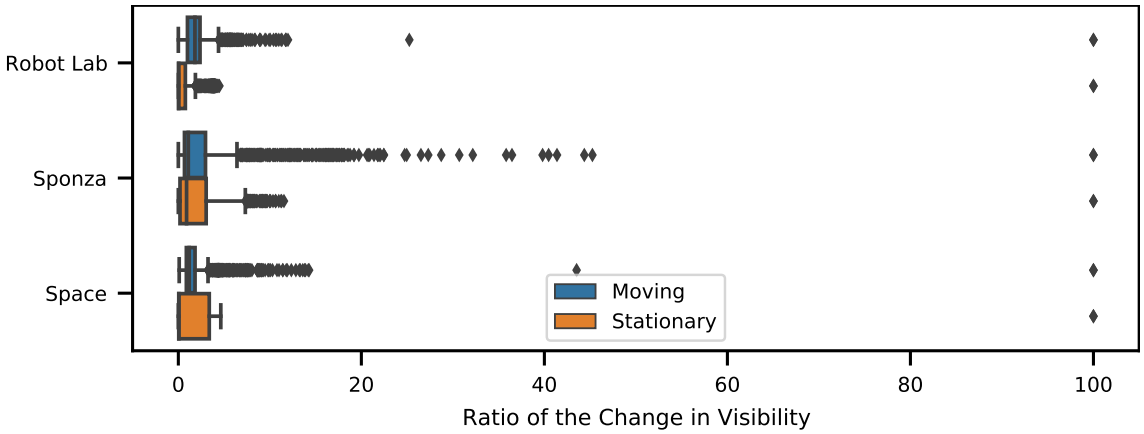


Figure 5.2: This box plot shows the ratio of change in visibility from one frame to the next in different scenes with either stationary cameras or moving cameras. The boxes show the borders of the 25th and 75th percentile, with the bar inside being the median and the range-bar showing minimum and maximum, excluding outliers. The change in visibility between frames typically stays under 10 % for all our test scenes. Outliers, determined using the interquartile range, are owed to fast camera rotations or objects being close to the camera.

90 % over the measured time frame. The stationary cameras in *Sponza* are focused on the falling boulders and thus actually produce more changes of samples than the more complex scene *Space*. A report containing detailed data on other thresholds can be found in the supplementary material of the paper.

Figure 5.4 shows the number of shading samples that stay valid depending on the threshold T for a few selected scene and camera combinations. Even at the lowest threshold $T = 1$, in most cases, more than 60 % of the shading results stay valid for more than 120 frames on average. In the best case, a *Robot Lab* sequence with little shading changes which are mostly below $T = 5$, reuse can last for almost the complete tested range of 120 frames. In the typical case, one can expect to be able to reuse between 80–90 % of shading over 20 to 60 frames, i.e., 0.3 s–1.0 s, depending on the allowed threshold T. In our worst case, *Space* with a lot of movement, longer reuse (> 1 s) is still possible for 50 % of shading samples.

5.3.3 Limits of applying temporal coherence

Both the temporal coherence of visibility and the temporal coherence of shading demonstrate a very high potential for reusing shading over many frames. However, practical implementations need to also consider the spatial sampling of shading, i.e., the drift of shading samples, their reprojection error and the required filtering.

To assess the potential of shading reuse in the light of these issues, we evaluate two practical rendering approaches for shading reuse. The first approach is *reverse reprojection*

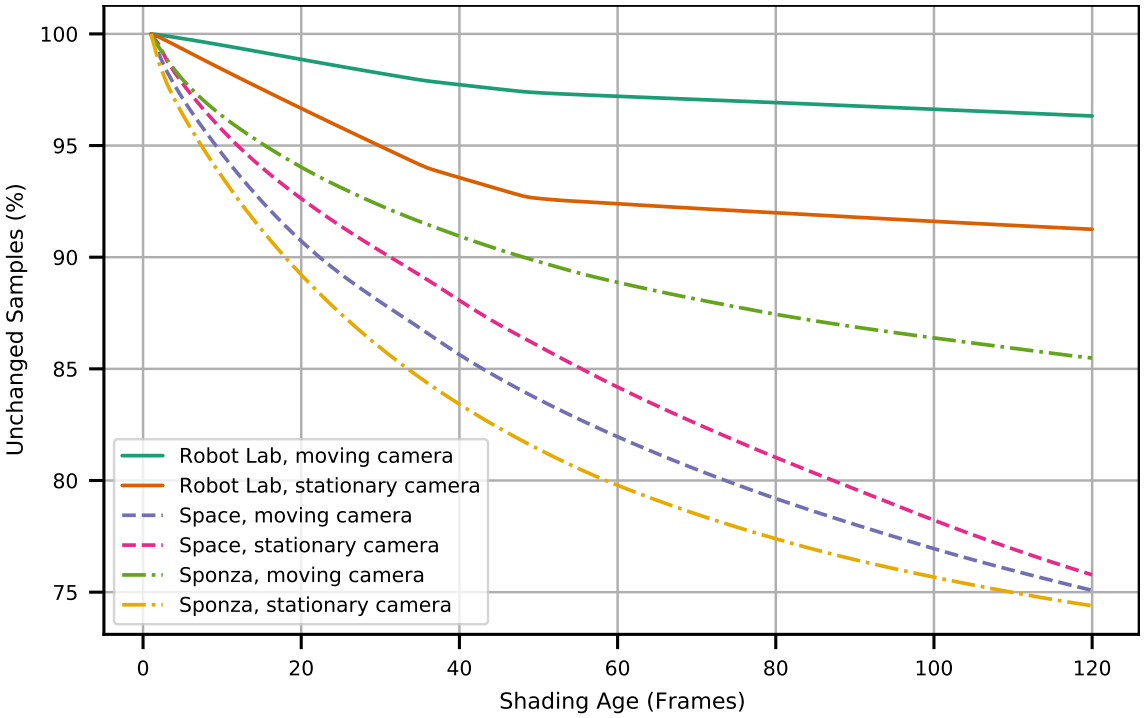
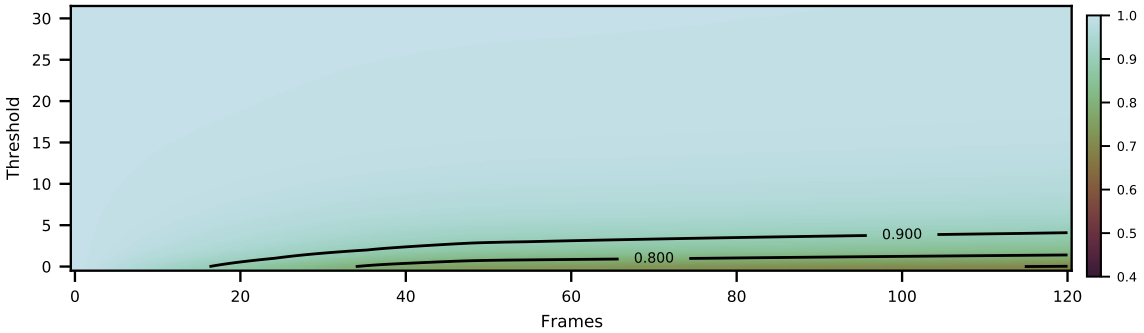


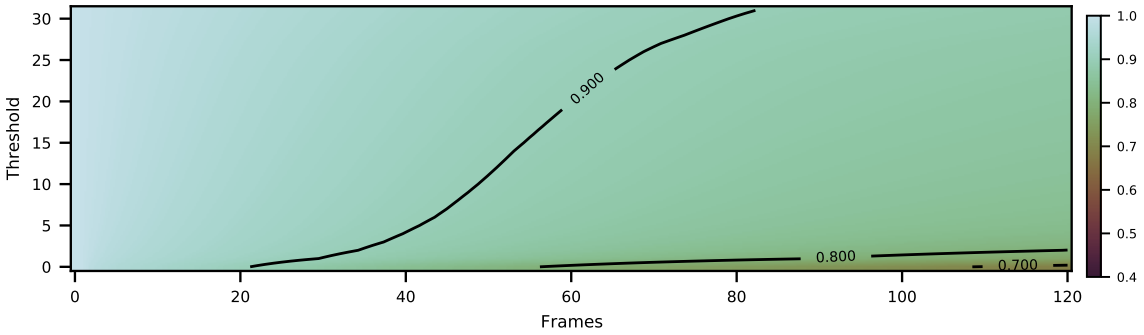
Figure 5.3: We use temporal forward rendering (TFR) to determine how shading behaves independently of changes in visibility and spatial sampling. More than 75 % of all samples change less than the color difference $T = 8$ for 120 frames in our test scenes. *Robot Lab* is the least dynamic scene and thus has the least changes. The stationary cameras in this scene's sequences are pointed towards the dynamic changes leading to a lower number of average unchanged samples than for the moving cameras. This is even more true for *Sponza*, where the stationary cameras see more changes than all paths of *Space*. *Space* nevertheless is the most dynamic of the three scenes.

caching, proposed by Nehab et al. [75], which was already used in the experiment for temporal coherence of visibility. Reverse reprojection caching reprojects samples from the previous frame to the current frame, potentially accumulating spatial sampling errors. Our implementation runs in two passes, a depth pre-pass and a forward rendering pass that either uses the cache or reshades. In order to avoid the accumulation of these errors, shading samples can be gathered in a temporally invariant space such as object space or texture space.

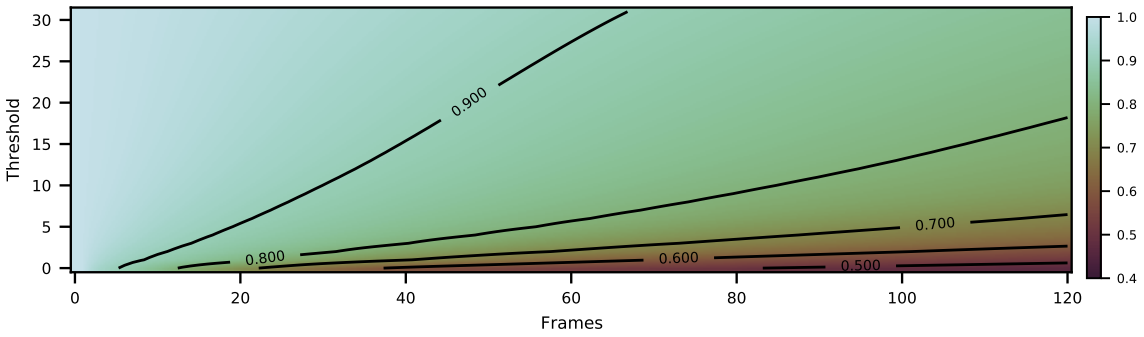
Thus, our second approach, *shading atlas* (SA), uses the rendering technique described in Chapter 3. Visibility and display stages are implemented as described in Section 3.1 using a deferred rendering pipeline which is similar to that of texel shading [46]. The location of the shading samples remains unchanged in the atlas, until the visibility of the triangles changes, or their resolution changes due to a level of detail change, in which case shading has to be recomputed, i.e., these samples are not reused.



(a) *Robot Lab*, moving camera

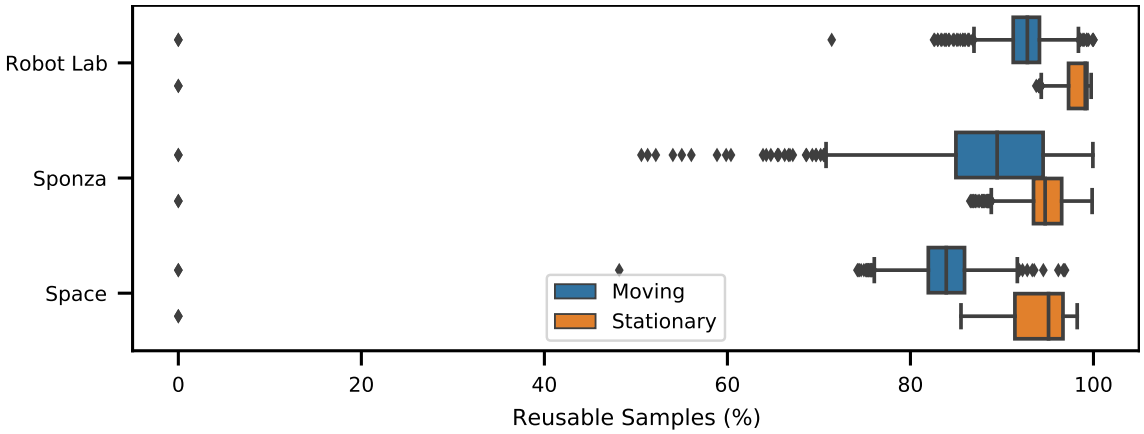


(b) *Sponza*, moving camera

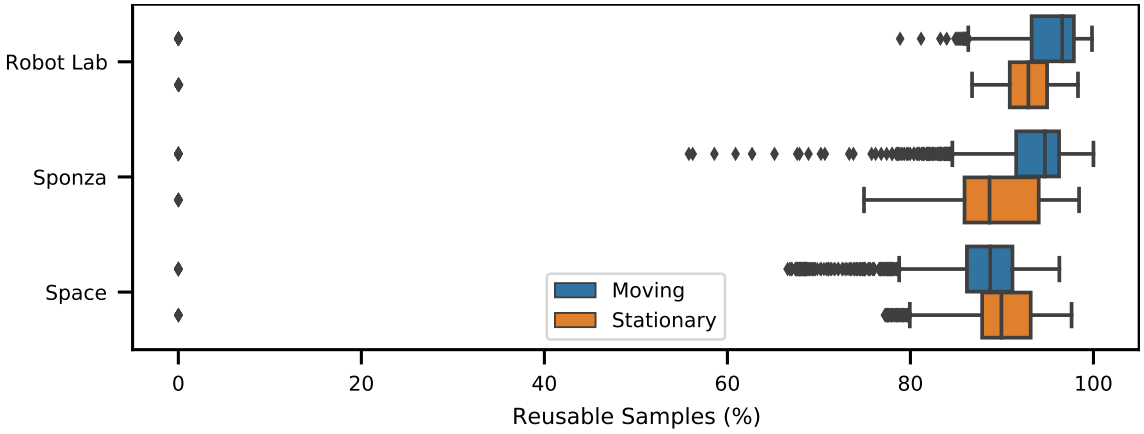


(c) *Space*, stationary camera

Figure 5.4: We use temporal forward rendering, which computes past shading independently of visibility and reprojection errors, to evaluate how many shading samples stay below a color difference threshold T during temporal upsampling, showing (a) the best, (b) the median and (c) the worst results among our test cases. Each row represents one specific threshold, and the percentage is monotonically decreasing with the age of the shading, i.e., if a sample can be reused for 10 frames, it is also valid in frames 1–9. Isolines highlight the change of the reuse potential over T. The shading of most samples stays valid for long periods of time, suggesting a huge potential for temporal reuse where samples only have to be shaded once during the time they are visible.



(a) Reverse reprojection caching



(b) Shading atlas

Figure 5.5: We determine the possible reuse of two rendering approaches based on visibility and a color difference threshold $T = 8$. The potential reuse depends on the scene complexity and camera movement. (a) Reverse reprojection caching accumulates spatial sampling errors over time, especially when the camera is moving. (b) In contrast, the shading atlas reuse is independent of spatial sampling, and thus the reuse correlates better with the dynamics of the shading as shown in Figure 5.3. We reuse a block of the shading atlas only if all the samples within the block can be reused. Outliers at 0 are caused by the first frame of the test sequence in which all samples need to be shaded.

With the two rendering approaches, we evaluate how many samples can be reused from frame to frame. For the reverse reprojection cache, we compute the possible reuse as fraction of the screen resolution, while, for the shading atlas, we consider the fraction of the total allocated space in the atlas in each frame. The results in Figure 5.5 show that, depending on the scene and camera movement, reuse of above 70 % could be achieved in most cases. Reverse reprojection caching has worse reuse potential with camera movement, as spatial reprojection errors accumulate. The shading atlas considers samples reusable only when an entire block is reusable, leading to a slightly worse overall reuse. Less reuse

for the shading atlas for the stationary cameras in *Robot Lab* and *Sponza* correlates with the temporal coherence of shading changes (see Figure 5.3), since the camera poses focus on dynamic parts of the scenes.

5.4 Constant frame rate upsampling

Previous strategies rely on uniform temporal upsampling, i.e., shading samples every K^{th} frame. In order for this assumption to work, the lifetime of a shading sample must be brief. This is essentially equivalent to assuming that the *shading gradient* is locally constant everywhere.

5.4.1 Rendering amortization from frame rate upsampling

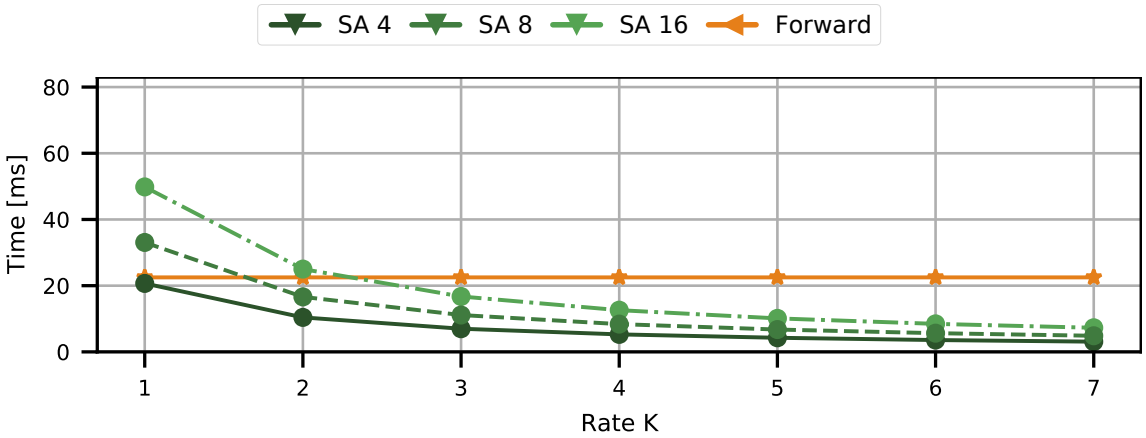
We first evaluate how much performance we can gain with constant frame rate upsampling. The upsampling factor K determines how many frames are rendered based on a single shading in the atlas. It enforces a constant factor between shading and display frame rate, e.g., $K = 4$ for upsampling from 30 to 120 Hz. In order to avoid disocclusion artifacts, the PVS for the shading atlas needs to contain all visible geometry of the displayed frames. For this experiment, we use a sampling based approach by combining the EVS of each displayed frame. In this experiment we use the correct future view matrices. In practical applications, we cannot always determine the PVS ahead of rendering, such as in a real-time rendering application where the user controls the camera. In Chapter 7, we discuss a prediction based PVS that is applicable for streaming.

In Figure 5.6, we compare render time measurements of forward rendering at 1920×1080 to our object-space rendering (measuring the combined time for visibility, shading and display stages) with atlas sizes of 4, 8, and 16 MPx. The test scenes were rendered with 1000 light sources and simple Phong shading to generate a uniform shading load. Depending on the scene, forward rendering can generate a frame in 20-33 ms. Object-space rendering becomes more efficient in all cases for $K \geq 4$, even for the 16 MPx atlas.

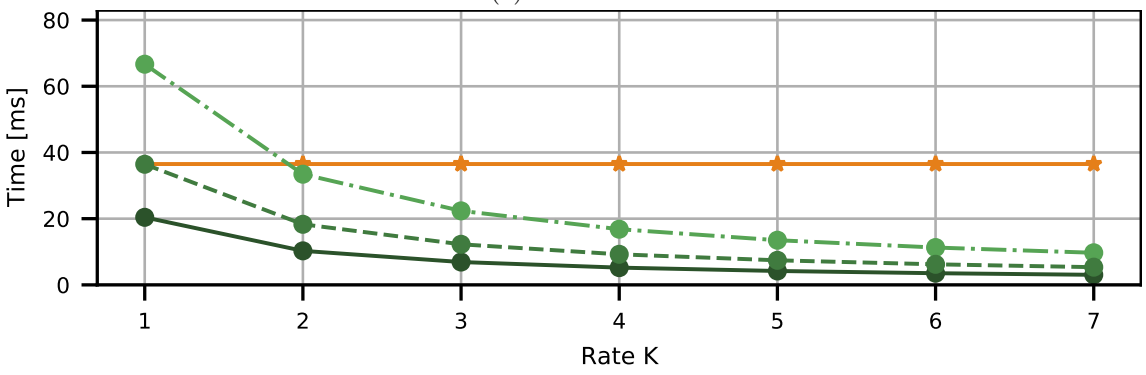
The disadvantage of constant frame factor upsampling are that, the longer shading is held, the more artifacts will be visible. Additionally, the gains for higher upsampling factors diminish as K gets bigger. Consequently, only a low upsampling factor K is practical and the full potential of shading reuse cannot be tapped into.

5.4.2 Perception of constant frame rate upsampling

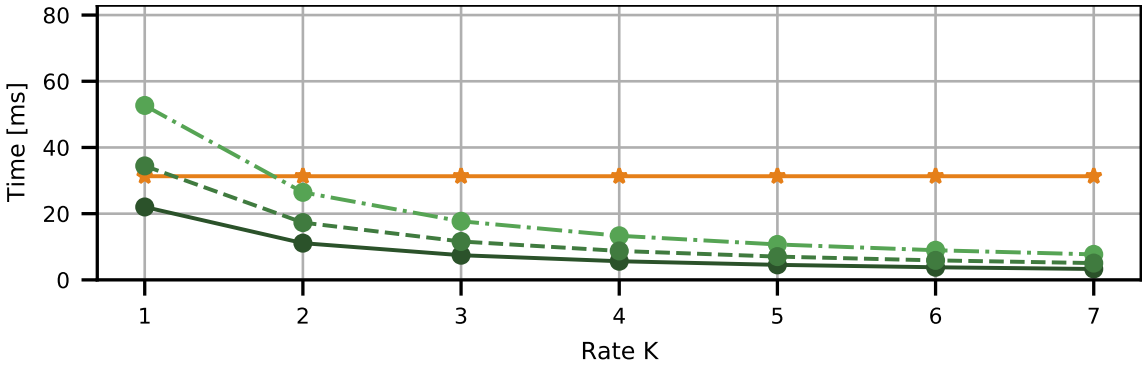
In order to evaluate the perception of outdated shading with constant frame rate upsampling, we implemented uniform temporal upsampling in reverse reprojection caching [75] (RRC), introduced in Section 5.3.3, in addition to the shading atlas with uniform temporal upsampling (SAU). RRC updates 16×16 pixel tiles with a constant refresh rate. A constant fraction of all tiles is updated in each frame, leading to a fixed livespan for each tile. For SAU, the livespan of cache entries is also constant, but every



(a) Robot Lab



(b) Sponza



(c) Viking Village

Figure 5.6: Comparison of render times for forward rendering and standalone object-space shading with our Shading Atlas. The x-axis displays the upsampling factor K, describing how many frames are generated from a single shading atlas.

cache entry has an individual remaining time to live depending on when it became visible. Both RRC and SAU distribute shading load across multiple frames without noticeable full-screen shading refreshes.

To evaluate how fixed-rate temporal upsampling influences rendering quality, we evaluated RRC and SAU with the same user study design as described in Section 5.1. As can be seen in Figure 5.7, uniform upsampling is not able to leverage the potential for shading reuse well, even when reusing shading only once (2× upsampling). RRC at 2× temporal upsampling leads to noticeable differences in 82 % of the cases and is, on average, reported to be "slightly worse" in quality. For higher upsampling rates (4 and 8), all participants always noticed differences and reported image quality to be close to "significantly worse". Some of this quality deterioration may be avoidable by using a better filter for temporal upsampling—we used linear interpolation. SAU at an upsampling factor of 2× remains below 1 JND. However, using SAU with upsampling factors larger than 2× is perceived as "significantly worse", with 100 % of participants noticing the difference. Overall, we conclude that a uniform upsampling frequency is not sufficient for longer shading reuse.

For a more thorough analysis, we separated the results by *Camera* and *Scene* variables. For RRC, *Camera* is significant for all tested upsampling rates[3]. At low upsampling rates, stationary cameras work better, apparently, because the problems introduced by the reprojection filter used to compensate for camera motion are avoided. At higher upsampling rates, a moving camera works better, likely, because movement hides some of the upsampling errors. For SAU, *Camera* was significant for 8× upsampling [4], with stationary cameras revealing more artifacts, which is consistent with our previous considerations. For *Scene*, we only found a significant effect for SAU with an upsampling of 4×[5], where *Space* is significantly better than *Robot Lab* and *Sponza*[6]. We again attribute this fact to the high frequency shading in *Space*.

5.5 Predicting shading changes

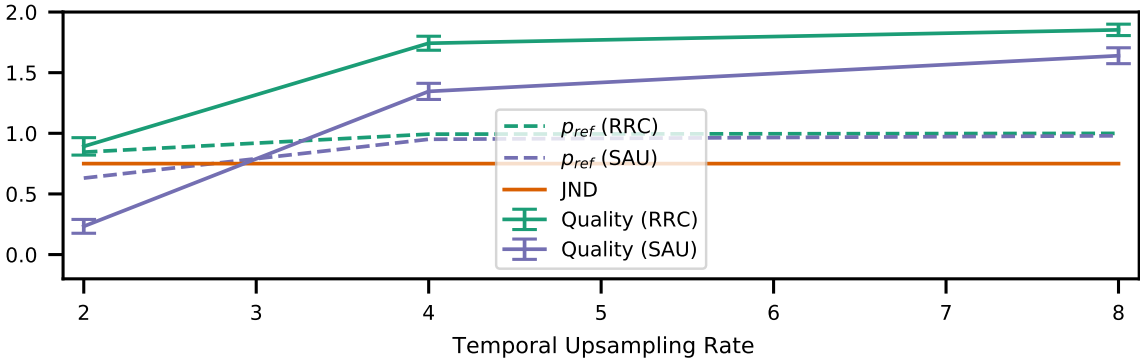
In the previous sections, we have established that there is a compelling potential for temporal shading reuse and that constant frame rate upsampling cannot fully utilize this potential. With constant frame rate upsampling, many samples that are still valid have to be discarded. Several existing methods enable us to map shading samples from one frame to the next either through image space reprojection or shading in a temporally unaffected space, such as object space or texture space. However, we require efficient prediction of the point in the future when shading samples will become invalid in order to know how

[3] Q for 2×: $F(1,33) = 26.117$, $p < .001$, 4×: $F(1,33) = 6.188$, $p < .02$ and 8×: $F(1,33) = 4.670$, $p < .05$

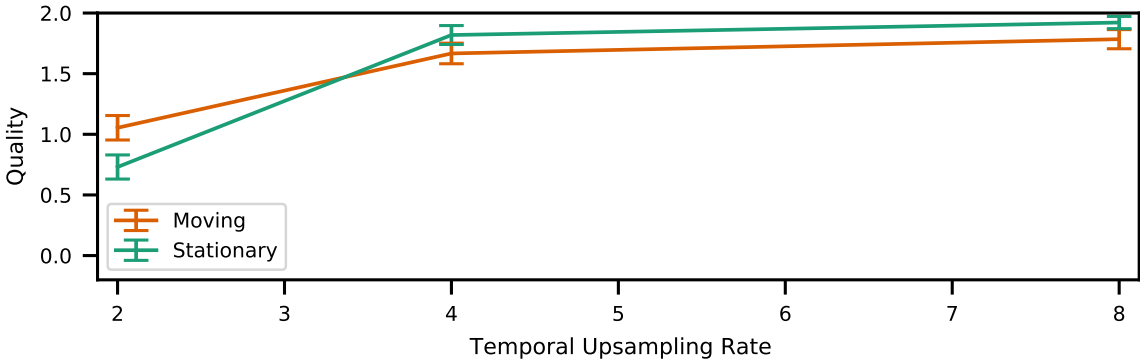
[4] Q for 8×, *Camera*: $F(1,33) = 13.393$, $p < .001$

[5] Q for 4×, *Scene*: $F(2,66) = 7.823$, $p < .001$

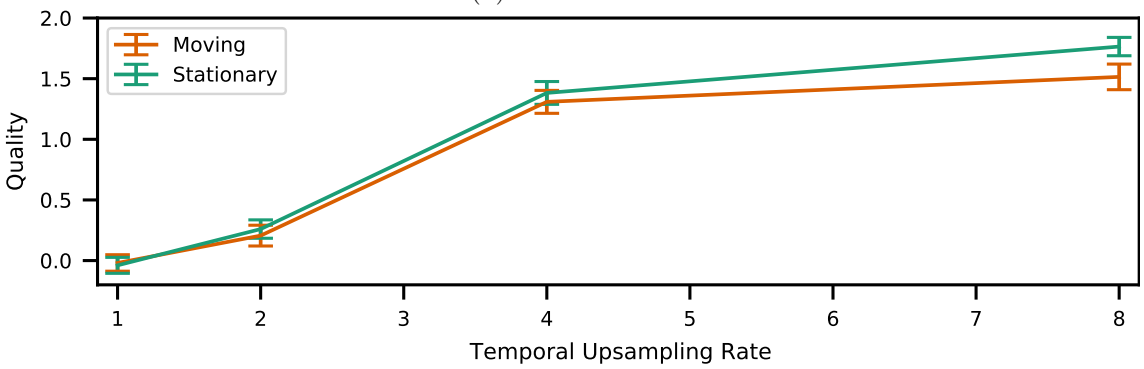
[6] Q for 4×, *Space* vs *Robot Lab*: $F(1,33) = 15.184$, $p < .001$ and *Space* vs *Sponza*: $F(1,33) = 11.880$, $p < .002$



(a) RRC vs SAU



(b) RRC *Camera*



(c) SAU *Camera*

Figure 5.7: User study results for RRC and SAU. While SAU is able to maintain a reasonable quality for a 2× upsampling (just below 1 JND), patterns are already visible for RRC with 2× upsampling. Holding shading for longer periods of time is clearly not reasonable. The visible reprojection errors for RRC 2× are confirmed by *Camera*, showing that a moving camera is worse than a stationary one. Higher upsampling reveals more errors for stationary cameras for both RRC and SAU, where shading errors are again easier to spot.

long shading can be reused. In this section, we discuss which mechanism is best suited to provide such a prediction.

5.5.1 Prediction with shading gradients

We consider whether mathematical analysis of shading data can be used as a predictor. Arguably, the optimal approach would be a frequency analysis of shading [38], but it is also the most expensive (and somewhat unwieldy) option. Ramamoorthi et al. [84] showed that a first-order gradient analysis is often sufficient in the spatial domain. Therefore, we propose to transfer some of these findings into the temporal domain. We consider a Taylor approximation of a shading function s as an obvious choice for prediction. A simple linear predictor can be formulated as a first-order Taylor expansion from time t_0 to t of the form

$$s\left(t\right) = s\left(t_0\right) + s'\left(t_0\right) \cdot \left(t - t_0\right) + e\left(t\right), \tag{5.1}$$

with a residual error $e\left(t\right)$. Based on a color threshold T, we can predict a reshading deadline

$$d = t - t_0 = \frac{T}{s'\left(t_0\right)}, \tag{5.2}$$

based on the per-channel maximum RGB color gradient $s'\left(t_0\right)$.

5.5.2 Analytic derivatives

One way to compute $s'(t)$ is to differentiate the shading function. However, this is usually not possible in practice, since parts of the function are not directly available in the shader or not differentiable. In most cases, these parts are time-varying scene parameters $\mathbf{q}$, including camera position, model position and potentially light transformation matrices. They can also encompass animated material parameters, vertex animations and other temporally varying values. We can rewrite the shading gradient $s'(t)$ as

$$s'(t) = \left\langle \nabla_{\mathbf{q}}\, s, \frac{\partial \mathbf{q}}{\partial t} \right\rangle \tag{5.3}$$

in order to separate it into differentiable and non-differentiable factors. The first factor (the gradient of s w.r.t. the parameters $\mathbf{q}$) is more expensive to compute, but predictable. It can be computed during shading and adds a manageable, constant overhead to shading operations. The second factor (the partial derivative w.r.t. t) needs to be supplied to the shader either differentiated outside the shader code if that is possible or estimated based on predictions or backward differences. For example, a camera displacement is unpredictable if it depends on user input and needs to be predicted, but the position of a physically animated rigid object can be differentiated and might not require additional computation since the velocity is usually available in the physics simulation.

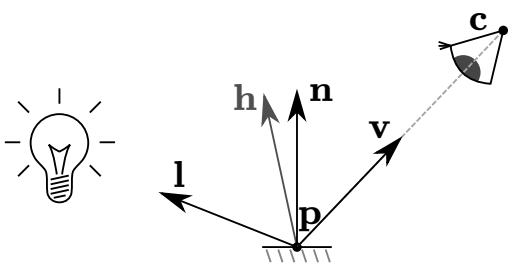


Figure 5.8: For a physically-based BRDF, the specular highlights are directly related to the surface normal $\mathbf{n}$ and half vector $\mathbf{h}$. The half vector $\mathbf{h}$ lies between the view vector $\mathbf{v}$ and the light vector $\mathbf{l}$. The view vector $\mathbf{v}$ is defined as the unit vector pointing from the surface point $\mathbf{p}$ to the camera position $\mathbf{c}$.

Arguably, the most important factor supplied to the shader is the camera position $\mathbf{c}$. As an example, we shall derive the derivative of the *Fresnel factor* with respect to camera motion. The Fresnel factor F defines the behavior of light reflection and refraction between different media. It depends on F_0, which is derived from material parameters such as the metallicity and color:

$$F = F_0 + (1 - F_0)\,(1 - \langle \mathbf{v}, \mathbf{h} \rangle)^5\,. \tag{5.4}$$

The half vector $\mathbf{h}$ is defined as the normalized vector between the view vector $\mathbf{v}$ and the light vector $\mathbf{l}$, as illustrated in Figure 5.8.

We begin the derivation by taking the gradient with respect to the view vector of the Fresnel factor by the view vector $\mathbf{v}$

$$\nabla_{\mathbf{v}} F = 5\,(F_0 - 1)\,(1 - \langle \mathbf{v}, \mathbf{h} \rangle)^4 \left(\mathbf{h} + \mathbf{J}_{\mathbf{hv}}^T \mathbf{v}\right) \tag{5.5}$$

where $\mathbf{J}_{\mathbf{hv}}$ is the Jacobian of $\mathbf{h}$ with respect to $\mathbf{v}$,

$$\mathbf{J}_{\mathbf{hv}} = \frac{\mathbf{I} - \mathbf{h}\mathbf{h}^T}{\|\mathbf{l} + \mathbf{v}\|}. \tag{5.6}$$

To get the gradient with respect to the camera position, we chain the Jacobian of $\mathbf{v}$ with respect to the camera position $\mathbf{c}$,

$$\mathbf{J}_{\mathbf{vc}} = \frac{\mathbf{I} - \mathbf{v}\mathbf{v}^T}{\|\mathbf{c} - \mathbf{p}\|}, \tag{5.7}$$

where $\mathbf{p}$ is the object point, to arrive at the final gradient

$$\nabla_{\mathbf{c}} F = \mathbf{J}_{\mathbf{vc}}^T \cdot \nabla_{\mathbf{v}} F, \tag{5.8}$$

of F w.r.t. the camera position.

The Fresnel factor is only one part of a modern physically based shading function. Differentiating a full physically-based rendering function becomes lengthy and tedious. Instead of manually differentiating the whole shading function, it is better to use an automatic differentiation approach. Such an approach can be easily implemented with operator overloading and custom data types [105]. While the commonly used shading languages support vector and matrix types, ideal for gradients and Jacobian matrices, they usually do not support operator overloading. Therefore, we implemented a simple automatic differentiation framework in Python, which outputs the corresponding GLSL code with support for all typical shader operations. Our solution handles not only scalar and vector-valued parameters and functions, but also texture lookup, using differences of two texture lookups per spatial dimension, and Poisson-sampled shadow maps.

Shader inputs such as camera, object or light transformations, are extended with their temporal derivatives. For all time-varying parameters where a future state can be calculated deterministically, such as prerecorded animations or physics simulations, we can obtain such derivatives directly by augmenting the animation code. For user-driven inputs, such as camera movement, we can usually compute a gradient based on an extrapolation of the input. Even though the derivatives can be computed alongside the shading, the overhead is non-negligible, resulting in a 1.85× (*Robot Lab*), 1.35× (*Sponza*) and 1.45× (*Space*) average increase of shading run-times, which would be increased even further by more complex shading models.

5.5.3 Finite differences

An alternative to costly analytic derivatives are finite differences. In the simplest case, we take the backward difference between two shading results. Such backward differences can either be computed between shading in consecutive frames or between frames that are further apart in time. The former has the advantage that it better approximates the limit of finite differences. However, it always requires shading twice in a row.

Computing finite differences over a longer interval is more economical and has a potentially beneficial low-pass filtering effect on spurious shading changes. Of course, when a shading sample is first considered, shading must always be done twice. In this way, a first deadline for reshading is extrapolated from the initial gradient. From then on, the long-range gradient is estimated whenever reshading occurs.

5.5.4 Comparison of gradient methods

We evaluate single-frame and long-range finite differences, and analytic gradients for TFR, using the previously found threshold $T = 8$ from the first user experiment. With TFR, we render multiple frames of increasing shading age, resulting in frames with the same sample position, but increasingly outdated shading. We use this data to retrospectively obtain the ideal deadline. Starting from the current frame containing the correct shading result, we determine the exact frame in the past where the shading difference exceeds the

threshold T. At this point in the past, both the analytical derivatives and finite differences are used to directly predict a future deadline. For the long-range differences, we repeat the process to find the next frame in the past that exceeds the threshold. We limit the search process to 119 frames into the past, effectively clamping the deadline in the range of 1 to 118 frames.

The results of our evaluation are presented in Figure 5.9 as bar charts separating the samples into ones that are shaded on time or too early, too late, or that do not change at all during 120 frames. Using TFR, this experiment again only considers shading changes and ignores changes in visibility or spatial sampling. As expected from our previous experiments, most of the samples are unchanged for the tested duration, though a few unchanged samples are incorrectly predicted to be in need of shading and thus are shaded too early. Early shading reduces the potential savings.

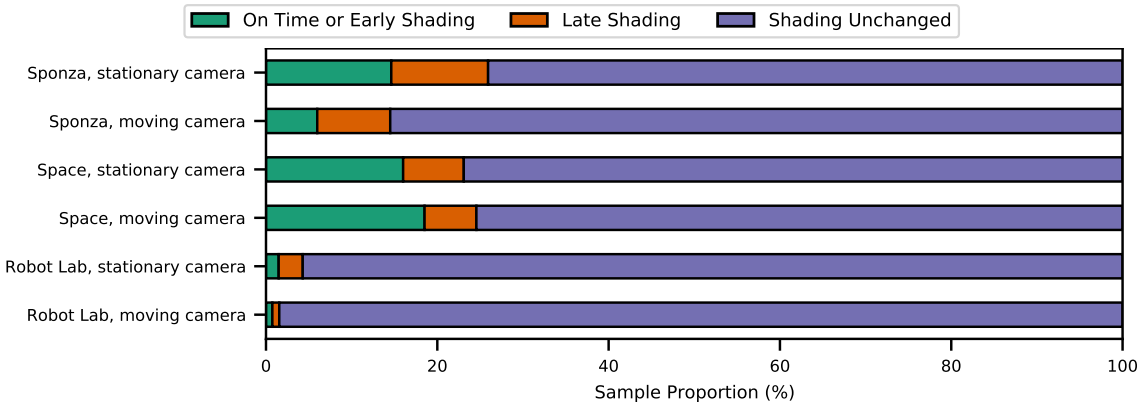
However, the critical factor for avoiding artifacts is correctly identifying those samples that actually *require* reshading. Recall that a uniform factor of 2× upsampling already significantly reduced perceptual quality. Our experiment places analytic derivatives and single-frame differences very closely, suggesting that finite differences are a good predictor for the analytic derivatives, with long-range differences coming in third with a small penalty. However, all these methods still miss some required shading, making further analysis necessary.

A different view on the data is presented in Figure 5.10 as a histogram that shows the percentage of samples sorted by the difference between ideal and predicted deadline. A positive difference means that the predicted deadline would have been too late to shade on time, a negative difference, too early. The area in the histograms shows only a subset of all samples. The remaining samples shown as the green part in the inset bar chart constitute samples where both prediction and ideal difference are > 118, i.e., where shading holds longer than two seconds. These samples compose 84 % to 85 % of all samples, again emphasizing the huge potential for extended shading reuse.

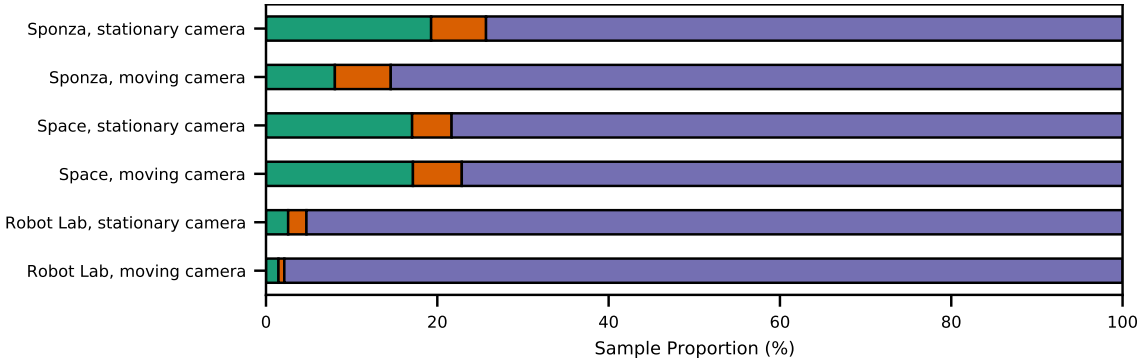
5.5.5 Spatial filtering of temporal gradients

Empirical inspection of the causes for shading gradient mispredictions revealed that the main cause are phenomena that move coherently though space, but abruptly change the shading of a single sample. Examples are the boundaries caused by moving lights or shadow borders. This suggests that we must capture the effects of *spatially coherent, time-varying changes*.

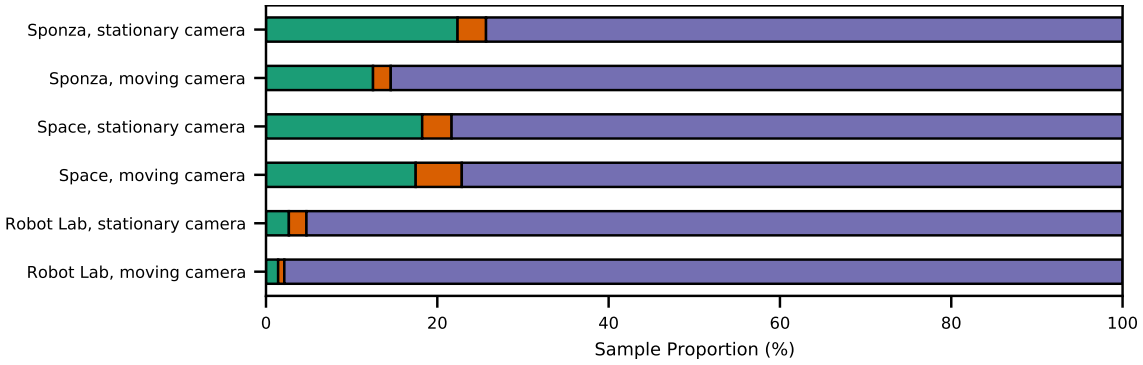
Therefore, we propose a simple maximum filter in image space. This approach is inspired by the render cache [108] and shading cache [102], which make a shading decision based on the estimated temporal gradient of neighboring samples. The effect of such a filter on shading gradients of a moving shadow border is shown in Figure 5.11. Clearly, the gradient filtering distributes the highly localized gradients of the shadow boundaries to the surroundings.



(a) Analytic derivatives

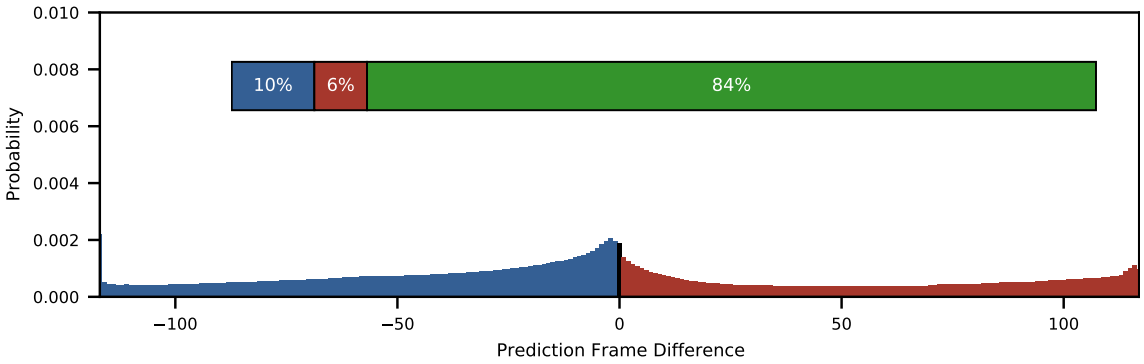


(b) Single-frame differences

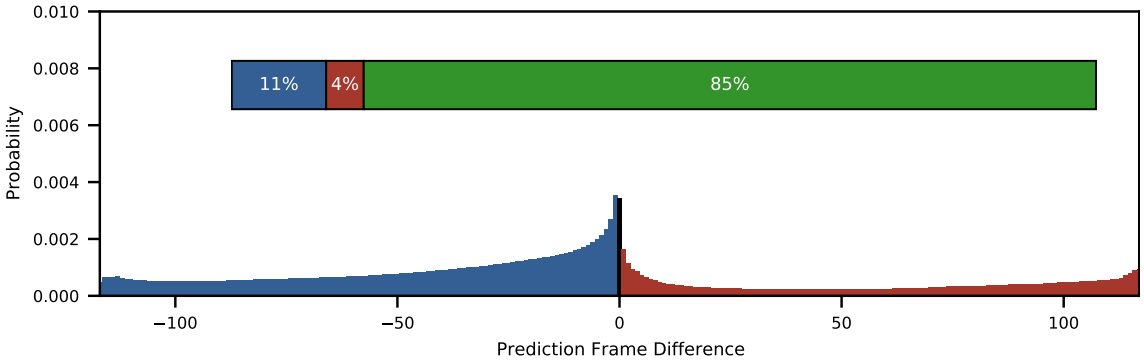


(c) Long-range differences

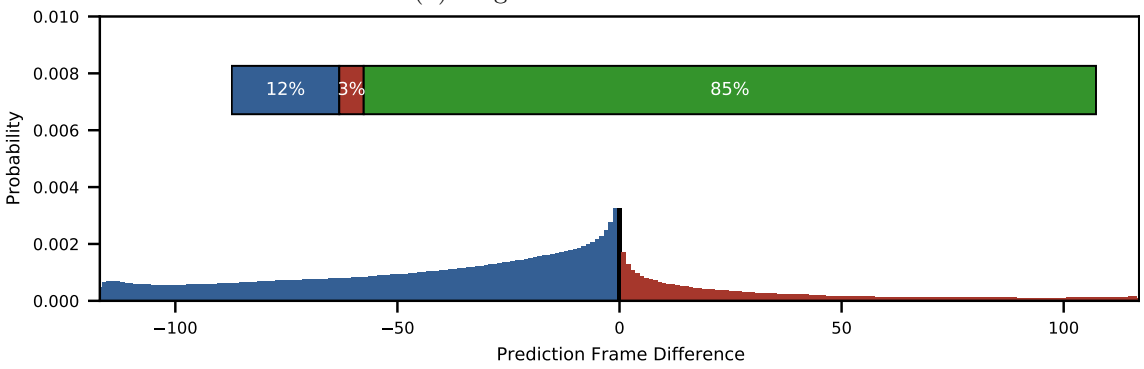
Figure 5.9: We use temporal forward rendering to evaluate different approaches to compute temporal gradients of shading and how well they predict future changes in shading. Late shading would cause artifacts in the final image output and thus should be avoided. Shading too early may harm performance, but does not lower quality. Ideally, a technique avoids late shadings completely, while keeping early shadings as low as possible. In accordance with our test for temporal coherence of shading, most samples stay unchanged for more than 120 frames. Note that the simple long-range differences (between two shading points) show the least amount of late shadings, while having only slightly increased early shading.



(a) Analytic derivatives

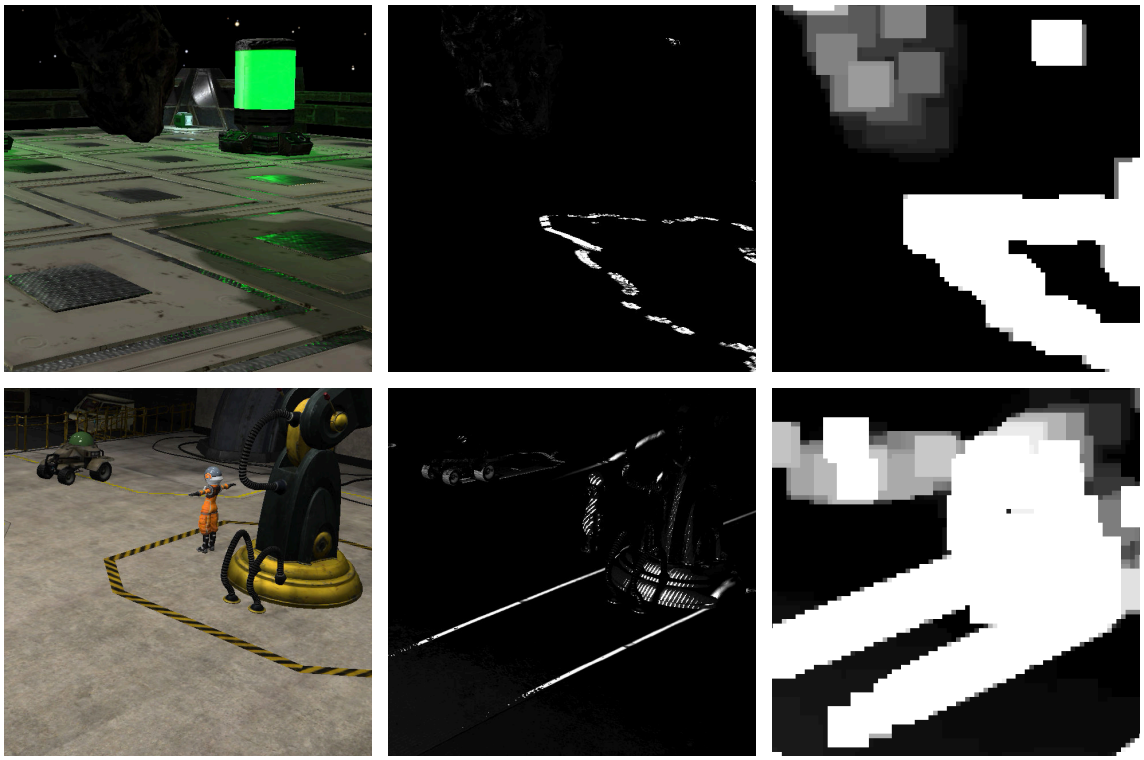


(b) Single-frame differences



(c) Long-range differences

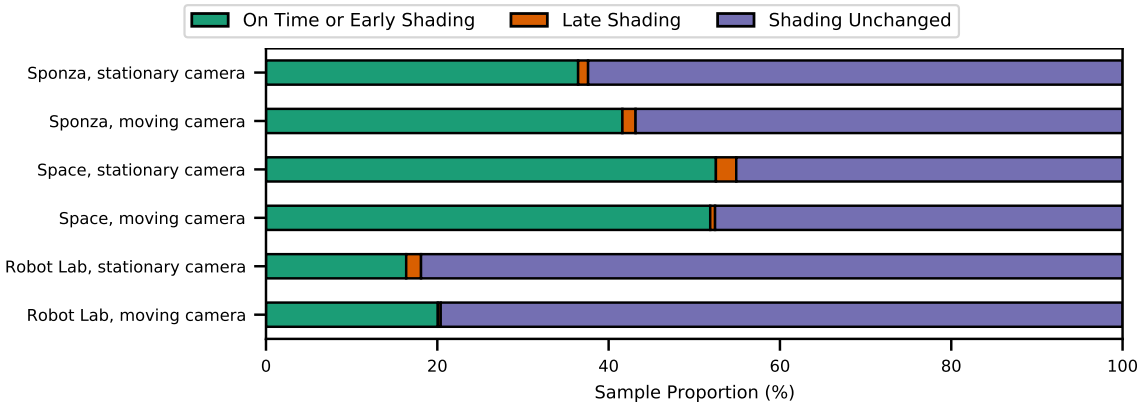
Figure 5.10: Using gradients to predict changes in shading leads to a peak around the target shading point (0 difference); blue bars indicate samples that would be shaded too early, and red bars, too late; averages are taken over all test scenes. Note that most shading samples (about 85 %, green) are not captured in the graph as they are correctly predicted to stay valid for longer than 118 frames. Artifacts arise for late samples depicted in red; interestingly, the simple long-range differences (between two shading points) show least of those, while having only slightly increased early shading.



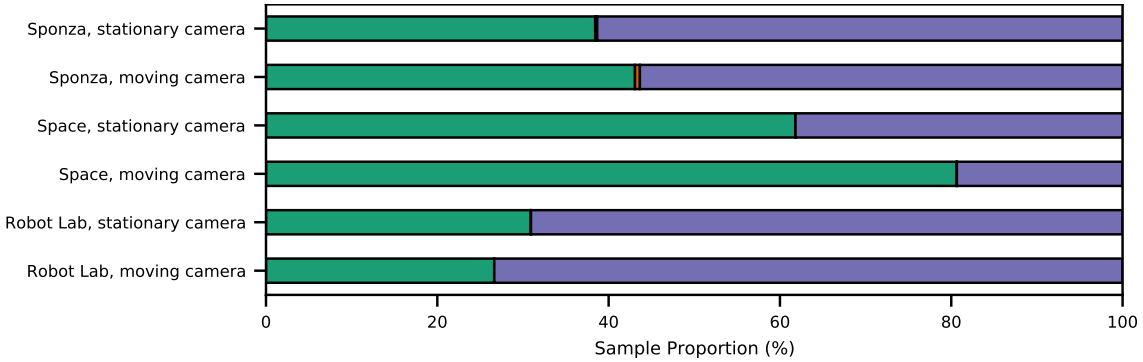
(a) Original (b) Gradients (c) Filtered gradients

Figure 5.11: (a) A partial view of *Space* (top) and *Robot Lab* (bottom) with moving asteroids and robots casting noticable or faint shadows on the floor. (b) While small shading gradients are typical for smoothly changing shading, such as on the asteroid or around the robot or the rotating arm, moving shadow boundaries show steep localized gradients. Even borders of slowly moving and faint shadows in *Robot Lab* are detected as the two white, diagonal lines on the floor. (c) Spatial maximum filtering within 72×72 blocks distributes those gradients to neighbors and correctly predicts where shading should take place. Obviously, such a spatial filter will also predict shading too early, e.g., in the opposite direction of the moving shadow boundary.

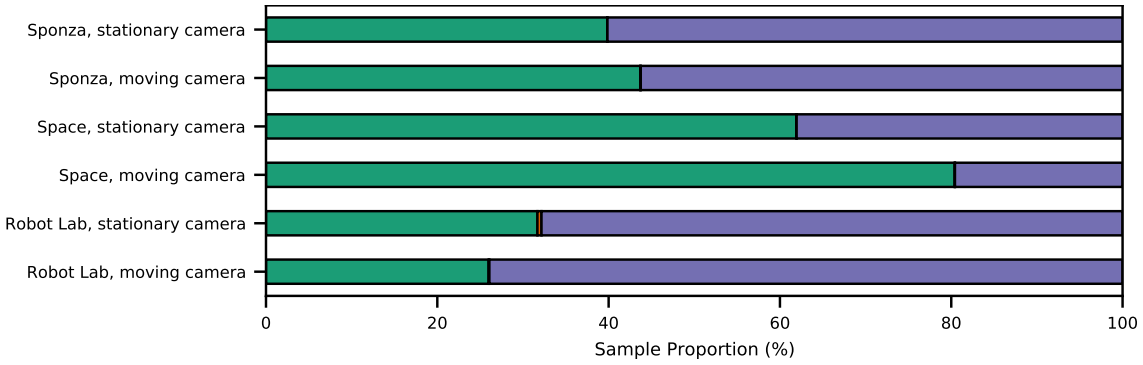
We evaluate this filter using TFR to isolate the effect of the spatial filtering, while avoiding other sources of misprediction, such as reprojection errors. We use a downsampling factor of 8×8, followed by a convolution with a rectangular kernel size 9×9 (overall touching 72×72 pixels). We empirically determined these parameters to be sufficient, since, at the cost of a lower shading reuse, we avoid late shading overall which we rate more important to avoid artifacts. When combining the previously described temporal gradient estimation methods with the spatial maximum filtering, we arrive at the results shown in Figure 5.12. When applying the filter, analytic derivatives, single-frame differences and long-range differences convincingly avoid shading too late at the cost of more early shadings within the 120 frames. Given the similar properties across these methods,



(a) Analytic derivatives



(b) Single-frame differences



(c) Long-range differences

Figure 5.12: Applying an image-space filter on top of the gradients strongly reduces late shading compared to Figure 5.9. While analytic derivatives still lead to late shading, finite differences—in both versions—effectively avoid late shading. Obviously, distributing shading predictions to neighbors increases early shading, leading to less reuse and thus less performance improvement, especially in the highly dynamic scene *Space*. Note that this visualization does not tell us how much reuse is possible, since we cannot distinguish by how many frames shading is early. Please see the supplementary material for the exact distribution of early and late shading.

long-range differences are most attractive, since they are the most efficient to compute. Figure 5.13 shows how many frames are shaded too early or too late when spatial filtering is applied.

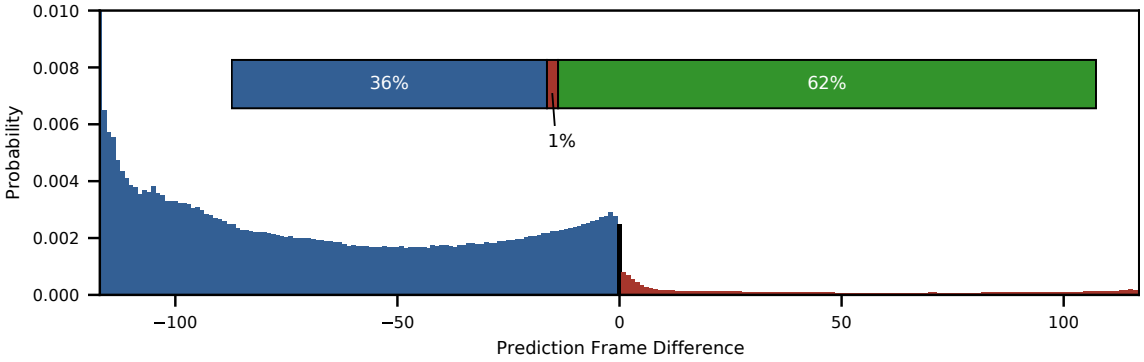
5.6 Discussion

In this chapter, we explored temporal shading reuse in various aspects. The main reason for performing a user study is the unavaiability of a temporal image error metric that captures the perception of temporal artifacts. Thus, we ran a user study to capture the actual perception of temporal changes in shading based on a just-noticeable-difference metric.

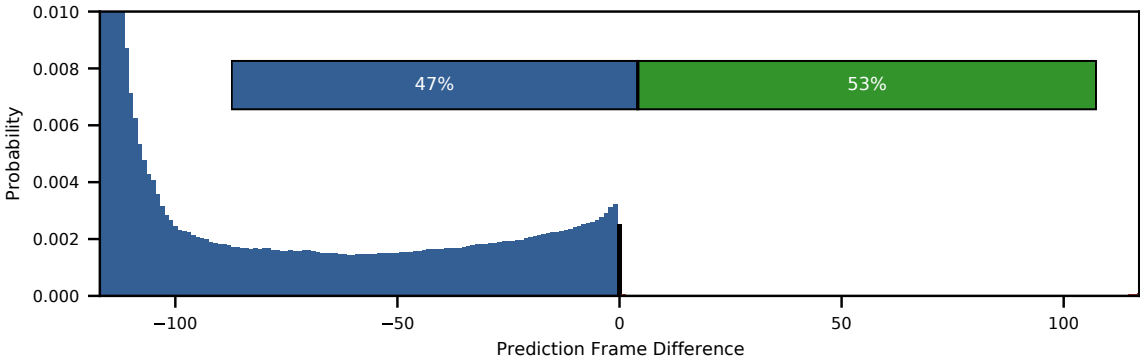
The first application of the user study was to figure out how outdated shading is perceived. To do so, we used TFR to create videos where shading was outdated to the extent of a specific color difference threshold T. We found that most study participants did not perceive a difference between the reference and the video with outdated shading at $T = 8$ and below. A closer look revealed that, at higher thresholds, differences are perceived earlier when the camera is static rather than moving and that errors are more easily spotted in simpler scenes with little animation. This behavior may be exploited for temporal shading reuse, since the higher the change rate in the scene, the higher we may choose the threshold, leading to a balanced workload.

Our next experiment concerned the temporal coherence of shading, where we found out how long shading could theoretically be kept and what would be the potential savings with various rendering techniques, if we knew exactly how long to keep it. First, we confirmed the results of Nehab et al. [75] that typically more than 90 % of samples stay visible from frame to frame. Next, we used TFR to determine how long shading changes stay below various thresholds depending on the scene and camera movement. As expected, shading stays temporally coherent longer when the camera is stationary and in simpler scenes. For the threshold $T = 8$, shading stays temporally coherent in all configuration for 75 % of all samples over 120 frames, highlighting the immense potential for savings. Testing reverse reprojection caching and the shading atlas, we found that, though the renderers put additional limitations on the shading reuse, typically more than 70 % of shading may be reused from frame to frame at this threshold, if we knew exactly how long shading can be kept.

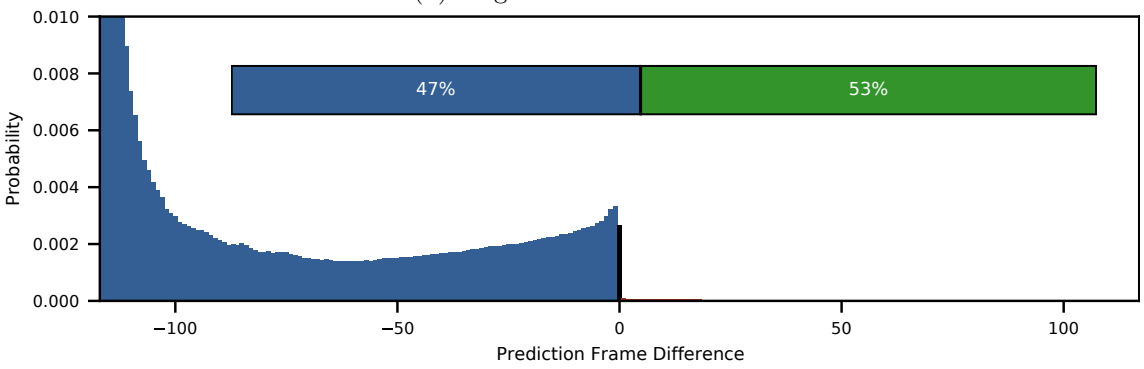
Before finding out how long shading can be kept, we investigated the practice of constant frame rate upsampling, where shading is simply assumed to change slowly enough for all shading samples. In constant frame rate upsampling, shading runs at a lower frame rate than display. Constant frame rate upsampling increases performance considerably. However, our user study results show that the higher the upsampling rate, the more artifacts become visible. At a target frame rate of 60 Hz, an upsampling factor of $K = 4$ is already clearly noticeable, showing that a more sophisticated approach is necessary for temporal shading reuse.



(a) Analytic derivatives



(b) Single-frame differences



(c) Long-range differences

Figure 5.13: Applying an image-space filter on top of the gradients strongly reduces late shading compared to Figure 5.10. While analytic derivatives still lead to 2 % late shading (red), finite differences—in both versions—effectively avoid late shading entirely. Obviously, distributing shading predictions to neighbors increases early shading (blue). Nevertheless, more than 50 % of shading samples are still correctly predicted to hold for longer than 120 frames (green).

We base the prediction of shading changes on a first order Taylor approximation of the shading changes. Comparing analytical gradients to single-frame and long-range finite differences, we find that long-range differences provide the best estimation for when shading has to be updated. However, there are still a few samples that are shaded too late, which needs to be avoided. We find that combining the temporal gradient with a screen-space filter manages to avoid late shadings almost completely, at the cost of slightly more early shadings. Using this model for prediction, we will design a framework for temporally adaptive shading reuse in the following chapter.

Chapter 6

Temporally Adaptive Shading

Contents

The findings of the previous chapter are the foundation of TAS, a renderer-independent framework for adaptively reusing shading information over time. A thorough evaluation of the framework brings us back to the shading atlas, since an object-space representation of shading information that is temporally coherent is ideal for TAS. Of course, TAS can also be applied in the streaming context of SAS to reduce required network bandwidth and either improve the image quality or decrease the rendering load on the server.

6.1 Temporally adaptive shading framework

Building on our previous findings, we design a TAS framework, which reliably avoids repeating redundant shading computations, while responding instantly to areas where rapid changes of shading occur. To make the framework largely independent of the rendering algorithm to which it is applied, we introduce the notion of *reuse units*. A reuse unit is a group of samples for which a uniform decision is made on whether the samples will be shaded anew or shading will be reused. The samples of these units are shaded together, and, consequently, must be stored together in the cache data structure. The renderer determines visibility independently for each unit. For example, a forward or deferred renderer uses a depth buffer to determine per-pixel visibility, while other renderer types may determine visibility per primitive. Thus, a reuse unit can be a single pixel as in the case of reverse reprojection caching [75] or a whole block within the shading atlas [73].

With these considerations, we concisely specify the three-step algorithm underlying the TAS framework:

1. Spatially-filtered shading gradients from the last frame are multiplied with the time elapsed since the last shading of each reuse unit and compared to the threshold (T) to decide whether reshading is necessary. Newly visible units are always shaded for two consecutive frames to determine a gradient from finite differences.

2. The shading is either reused, or the unit is reshaded. In the latter case, a new shading difference to the previous shading result is computed for each sample.

3. The shading gradient is estimated based on the shading difference, scaled by the time difference between them, and a spatial filter is applied to distribute the shading gradient information.

This framework is general enough to be applied to almost any rendering architecture capable of referring to previous shading results.

Algorithm 4 shows the shading decision process. To compute the shading gradients, we need to store and update the time period since the last two shading computations ($\mathrm{RU}.\Delta t_{last}$). The time period since the last shading ($\mathrm{RU}.\Delta t_{current}$) is required to compute the currently expected shading change and to update the former shading period in case of shading. Finally, the relevant value for the reuse unit needs to be looked up in the spatially filtered shading gradients ($\Delta s/\Delta t$) to make the shading decision.

Shading and reuse thereof is shown in Algorithm 5. Samples are shaded anew or reuse previous shading results, based on the decision made for the corresponding reuse unit. In either case, the framework requires access to the previous shading result (s_{last}) of all

ALGORITHM 4: Shading decision per reuse unit.

Launch: per reuse unit RU of the renderer
Input : Δt (time since last frame),
$\Delta s/\Delta t$ (spatially filtered shading gradient)
Output: do_shade (whether reuse units should be shaded)

```
1  RU.Δt_current ← RU.Δt_current + Δt
2  if RU just became visible then
3  |  do_shade ← true
4  |  RU.Δt_last ← 0
5  |  RU.Δt_current ← 0
6  else if lookup (Δs/Δt) · RU.Δt_current < T and
7   RU.Δt_last > 0 then
8  |  do_shade ← false
9  else
10 |  do_shade ← true
11 |  RU.Δt_last ← RU.Δt_current
12 |  RU.Δt_current ← 0
```

ALGORITHM 5: Shading or shading reuse per sample.

Launch: per sample S of the renderer
Input : do_shade (output from Algorithm $s(blue)4$)
Output: s (final color of the sample)

```
1 if do_shade then
2 |   s ← clamp (shade ())
3 |   difference ← (s − S.s_last) or 0 if S.s_last is invalid
4 |   S.s_last ← s
5 |   S.Δs ← max (abs (difference))
6 else
7 |   s ← S.s_last
```

ALGORITHM 6: Spatial filtering of shading gradients.

Launch: per spatial unit of the spatial filter
Input : spatially relevant shading samples
Output: $\Delta s/\Delta t$ (spatially filtered shading gradient)

```
1 Δs/Δt ← 0
2 for each S of the spatially relevant samples do
3 |   Δs/Δt ← max (S.Δs/ S.RU.Δt_last, Δs/Δt)
```

samples, since those are needed for the finite differences (Δs), even if the current shading results (s) are computed from scratch.

Finally, Algorithm 6 shows the spatial maximum filtering process. The algorithm here is kept general for various different spatial data structures. It first computes the shading gradient based on the information of the reuse units and their samples and then computes the maximum over all of the spatially relevant samples.

6.1.1 Temporally adaptive reprojection caching

In our first reference implementation, the *temporally adaptive reprojection cache* (TARC), extending the method of Nehab et al. [75], image-space pixels serve as reuse units. We replace the periodic refresh of the reverse reprojection caching shader with the first two steps of the framework. Furthermore, we store per-unit and per-sample variables in a double-buffered G-buffer. The input buffers are reprojected, and the potentially altered values are stored in the output buffers. During the depth pre-pass, we also store the estimated shading gradient in the G-buffer. We implement spatial maximum filtering by downsampling the gradient buffer using a maximum filter with overlapping square kernels (as before, we use a downsampling factor of 8×8, followed by a filter of size 9×9). In comparison to standard reverse reprojection caching, TARC needs additional memory for screen size buffers to store the shading difference and the time since the last shading.

6.1.2 Temporally adaptive shading atlas

Our second reference implementation, the *temporally adaptive shading atlas* (TASA), adds temporally adaptive shading to the shading atlas. Using a texture-space representation for storing shading samples avoids the accumulation of reprojection errors faced by TARC. It is also convenient to define reuse units by proximity of shading samples on object surfaces (or by general proximity in a 3D scene). The reuse units in TASA correspond to two triangles packed into a rectangle of $2^N \times 2^M$ texels, where each unit's size in the atlas is determined based on its image-space projection. However, other granularities, e.g., 8×8 texels [46], per-object texture charts [8] or micro-polygons in REYES [30], could be chosen.

By retaining the maximum of all shading gradients across an entire reuse unit, a conservative object-space filter is applied to the unit at almost no additional cost, since the samples of a reuse unit are processed together. The resulting object-space filtering is particularly relevant when some of the shading samples are currently occluded in image space.

However, limiting the filter to the boundaries of a reuse unit fails to capture spatial gradients that cross the boundaries of adjacent reuse units: For example, a shadow boundary may be creeping slowly across an entire surface consisting of multiple neighboring reuse units. We could extend the object-space filter to support a convolution-style kernel larger than a single reuse unit, but this would be a costly operation. Instead, we opt to concatenate the per-reuse-unit filter to an image-space filter that determines the maximum over direct image-space neighbors. This image-space filter is not only very inexpensive, but it also captures spatial coherence of perspectively close shading samples, which may not be apparent in object space.

The resulting pipeline works as follows:

1. Exact visibility is computed per frame in a geometry pre-pass and stored in a G-buffer as primitive ID with corresponding shading gradients.

2. Reading the primitive ID, the atlas is updated such that it has room for the visible reuse units. The shading gradients are maximum filtered using a 2×2 window in image-space for each reuse unit to propagate the maximum gradient in an image-space neighborhood.

3. Shading decisions are made on all reuse units. Reuse units for which samples are newly allocated and reallocated in the atlas are always shaded, i.e., they are considered newly visible.

4. The shading workload is executed including the computation of the shading differences and shading gradients are directly maximum filtered per reuse unit.

5. The G-buffer is revisited for the final deferred rendering pass.

The additional memory requirements include a copy of shading atlas to compute the shading differences, per-patch shading differences and times, and a screen space buffer for the spatial filter.

We describe the results obtained with TARC and TASA in the following section.

6.2 Evaluation and results

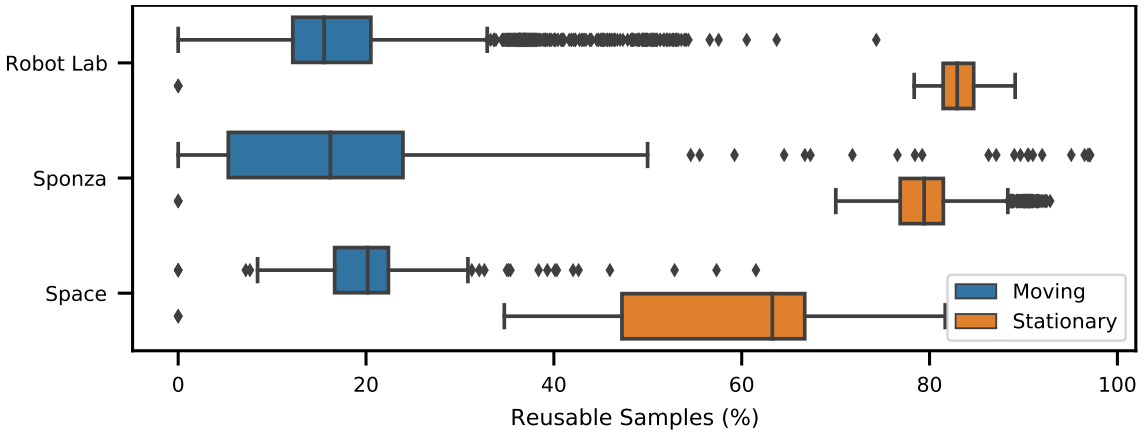
We present results for the reuse, quality and runtime of our TAS implementations based on technical experiments and two user studies. First, we evaluate the practically achievable reuse with the temporally adaptive shading framework and compare it to the theoretically possible reuse that we determined in the previous chapter. Using the same study setup as before, we determine the perceptual quality of the practical implementations, again at different thresholds. As the overall goal is reduction of rendering time at high shading loads, we present detailed timing results in comparison to Forward+ rendering. Finally, based on these results, we implement TASA in *Unreal Engine 4* and conduct another smaller, quantitative user study in VR with additional modern game scenes to confirm our findings.

Unless stated differently, we use the following experimental setup in all experiments. To avoid sampling artifacts from the atlas when displaying the final image, we test TASA with a 16 MPx atlas (TASA 16) and with an 8 MPx atlas (TASA) to evaluate actual use. We use a threshold of $T = 8$, aiming to stay below 1 JND. We again use the experimental setup as described in Section 5.1.1 with three test scenes and an image resolution of 1920×1080. Test sequences for technical evaluations are 900 frames long, as used previously.

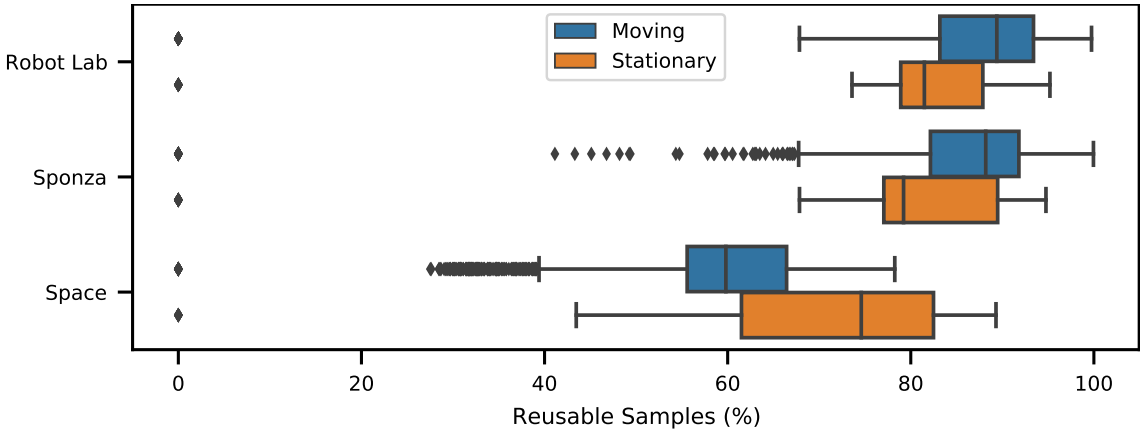
6.2.1 Reuse

To assess the influence of the TAS algorithm, we evaluate the practical shading reuse for TARC and TASA. In Section 5.3.3, we discussed the theoretically possible reuse with a perfect prediction of when to shade, resulting in a reuse of 80–90 % for both TARC and TASA. About 1–5 % of shading is due to changes in visibility.

As seen in Figure 6.1, TARC shows low reuse for dynamic camera movements. We found that the reprojection error for camera movements also effects shading gradient predictions, which are slightly too high and, in combination with the spatial filter, invalidate shading often. While a smaller filter size would increase the reuse potential, it leads to clearly visible artifacts due to missing shading in some scenes. Overall, reuse drops to about 20 % for camera movement. For stationary cameras, reuse stays at about 80 %, except for *Space*—which shows a lot of motion—where it drops to 50–60 %. A better reprojection filter for gradients and an adaptively sized image-space filter may increase reuse potential for TARC. However, more advanced filtering and filter size adjustments would also increase overheads.



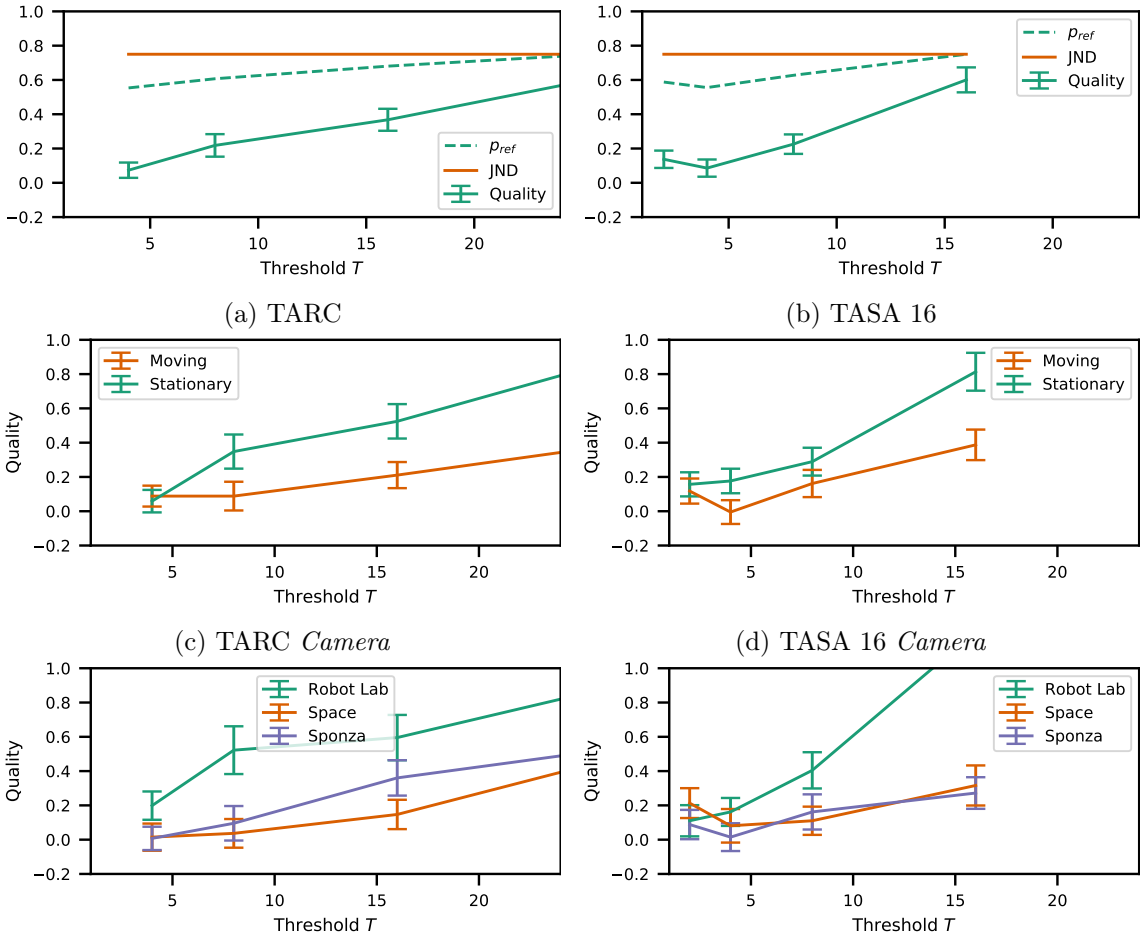
(a) Temporally Adaptive Reprojection Caching



(b) Temporally Adaptive Shading Atlas (8 MPx)

Figure 6.1: Actual reuse for the TAS implementations with a color difference threshold $T = 8$. TARC has a reasonable reuse for stationary cameras. However, when the camera is moving, the spatial errors introduced by the reprojection, paired with the spatially filtered differences, drops the reuse to impractically low numbers. TASA works significantly better, although it reuses the shading of whole triangle pairs and reshades even when a single sample within the triangle pairs needs reshading. TASA with 16 MPx is nearly identical to 8 MPx and thus omitted here.

TASA is able to retain a high amount of reuse (on average 57 % to 90 %) in comparison to its *ideal* version. The largest drop can again be observed in *Space*, with many high frequency changes being distributed to neighbors. For TASA, the reuse reduction is similar for both stationary and moving cameras, underlining that shading in texture space enables consistent addressing of shading samples. Also, view-dependent shading effects do not heavily influence shading reuse; only in *Space* with its many highly metallic materials, a moving camera significantly reduces shading reuse.



(a) TARC (b) TASA 16

(c) TARC *Camera* (d) TASA 16 *Camera*

(e) TARC *Scene* (f) TASA 16 *Scene*

Figure 6.2: User study results comparing TARC and TASA with a 16 MPx atlas. Both approaches show high quality at $T = 8$, staying well below 1 JND. At $T = 2$, Q seems to be a statistical outlier for TASA 16. Note that, in accordance with our pre-study result, we tested TARC and TASA over different intervals. However, their results are very similar, especially for $T = 8$, where both are clearly below 1 JND.

6.2.2 Quality

The user study results for both TAS implementations are shown in Figure 6.2. For identical target reshading thresholds of T, both approaches behave similarly. For $T = 4$, p_{ref} is close to 50 %, and Q is at about 0.1. For $T = 8$, p_{ref} is about 60 %, still significantly below 1 JND, and Q is 0.2, indicating a very high quality. A setting of $T = 16$ is about twice as bad in Q and very close to 1 JND, thus, we would suggest to use $T = 8$. For $T = 32$, TARC is already above 1 JND, and Q is close to "slightly worse". We did not find a reason for the slight drop in Q for TASA 16 from $T = 2$ to $T = 4$; as the confidence intervals overlap, this may just be a statistical outlier.

A detailed analysis again reveals that there is a significant difference for *Camera* for TARC with $T = 8$ and $T = 16$[1], as well as TASA 16 for $T = 4$ and $T = 16$[2], again pointing towards the fact that, for dynamic scene content, shading errors are more difficult to spot. Similarly, there is a significant difference for *Scene* using TARC with $T = 4$, $T = 8$, and $T = 16$[3] as well as TASA 16 with $T = 8$ and $T = 16$[4]. Post hoc testing revealed that only the difference between *Robot Lab* and the other two scenes was always significant. This fact again confirms the previous consideration that scenes with little movement, but clear moving shading discontinuities, like shadow boundaries, are sensitive with respect to correct shading.

Overall, the results qualitatively resemble the findings of TFR in the previous chapter (Figure 5.1). At thresholds of $T = 8$ and below, users hardly notice any temporal artifacts. Looking closer at the influence of scene complexity and camera movement, we again see that simpler setups make it easier to perceive artifacts. Quantitative differences between TARC and TASA 16 are negligible, since both are clearly below 1 JND at $T = 8$.

6.2.3 Runtime

To be of practical value, TAS not only needs to be similar in quality, i.e. rendering at a threshold that is indistinguishable from the ground truth, but must also reduce runtime when including all overheads. Since the approach targets rendering with complex shading, we run our measurements with high shading loads and do not consider any pre- or post-processing (which are orthogonal to TAS). The overheads of TAS include computing shading differences, spatial filtering, and dynamically deciding whether to shade or not, which may lead to thread divergence during shading and thus reduce the efficiency. To this end, we first measure the overhead of TARC and TASA in addition to the full shading. The overhead is between 14.5 % to 16.9 % for TARC and between 2.3 % to 5 % for TASA. This overhead has to be overcome with the shading reuse to perform as well as the rendering approaches without TAS.

The actual speedups are shown in Figure 6.3. Among the tested scenes, *Space* is especially difficult to speed up using temporal coherence, since most of the scene's surface points are either very dynamic or belong to the sky box. The latter can easily be reused, but has practically no shading load and thus is unsuitable for TAS. As expected from the limited reuse and the higher overhead, TARC does not improve over Forward+ for moving cameras, but it does have some considerable speedups between 1.38× and 2.4× when the camera is stationary.

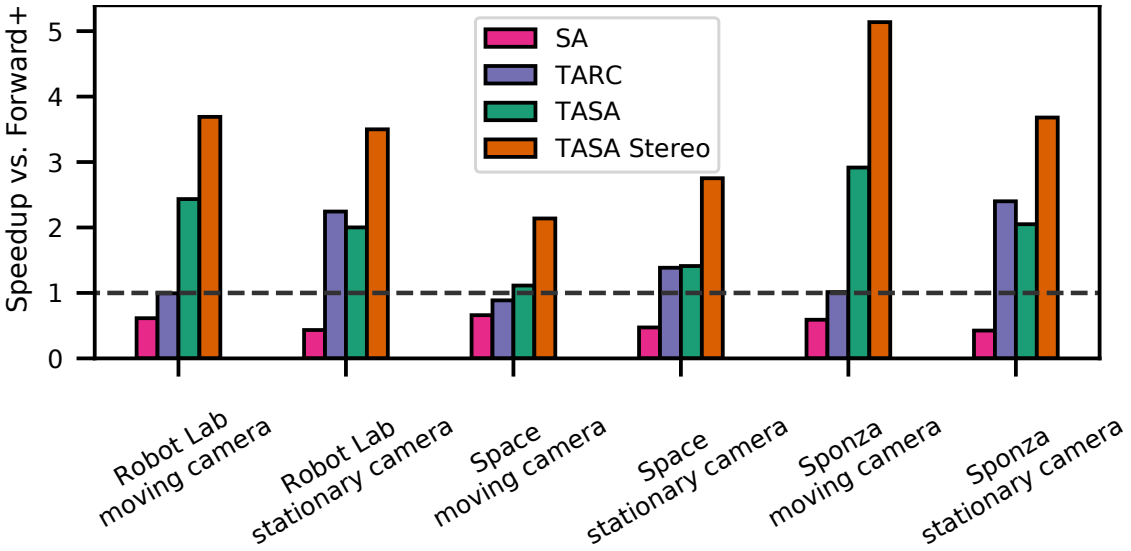
Our main focus is on the performance gains of TASA, which is able to re-use shading across the left and right eye buffers in VR stereo rendering. We see that TASA outperforms

[1] Q for $T = 8$: $F(1, 33) = 17.892$, $p < .001$ and $T = 16$: $F(1, 33) = 13.452$, $p < .001$

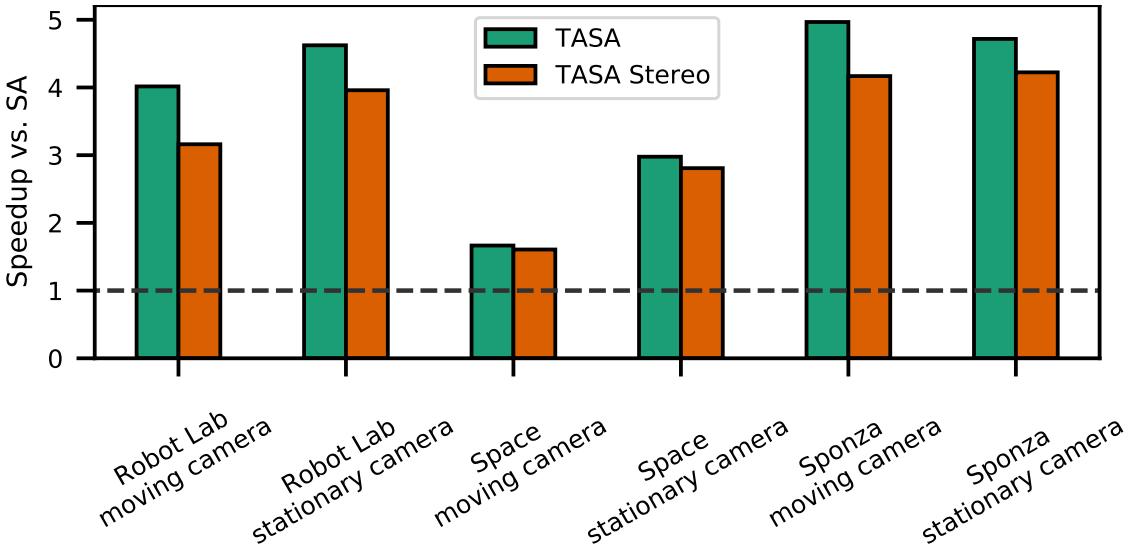
[2] Q for $T = 4$: $F(1, 33) = 7.875$, $p < .01$ and $T = 16$: $F(1, 33) = 38.588$, $p < .0001$

[3] Q for $T = 4$: $F(2, 66) = 6.732$, $p < .01$, $T = 8$: $F(2, 66) = 17.129$, $p < .001$, and $T = 16$: $F(2, 66) = 17.394$, $p < .001$

[4] Q for $T = 8$: $F(2, 66) = 6.366$, $p < .01$, and $T = 16$: $F(2, 66) = 75.338$, $p < .001$



(a) Speedup versus Forward+



(b) Speedup versus SA

Figure 6.3: (a) The speedup shows that the overhead of TAS is mostly compensated for TARC, but it fails to accomplish any runtime improvements for moving cameras in contrast to stationary cameras, likely, due to the errors caused by repeated reprojection of the shading samples. As expected, SA with an atlas 4× the final output resolution is slower than Forward+. However, TASA manages to compensate the overhead of both the TAS algorithm and the shading atlas, showing considerable runtime improvements for both stationary and moving cameras. When rendering stereo for VR, the benefits of the shading atlas lead to even bigger speedups of TASA in comparison to Forward+ rendering. (b) The spatial filter which needs to be applied twice in stereo rendering causes a slightly higher overhead, leading to a slightly decreased speedup in comparison to SA. Finally, results differ between scenes based on their potential for savings. The highly dynamic space scene performs worst, but still with a considerable speedup.

the other methods for all scenes, both in mono and stereo rendering. The speedup in stereo mode over Forward+ is in the range $2-5\times$ ($1.1-3\times$ in mono mode). This is noteworthy, as TASA must compensate the overhead of its SA foundation. SA alone is only around half the speed of Forward+ for monoscopic rendering, most likely due to its 8 MPx atlas size that is $4\times$ the resolution of the final output image.

Overall, it can be seen that adaptivity is key for temporal shading reuse. While uniform temporal reuse strategies reduce shader invocations, quality drops quickly. Using simple shading differences with spatial filtering for gradient estimates works well and is efficient. Especially placing shading samples in texture space appears to be an efficient strategy for reusing them over longer periods of time. However, using an atlas that matches the screen resolution introduces spatial sampling artifacts and reduces sharpness. Interestingly, TAS can easily compensate for these additional shading samples, leading to overall performance gains. However, for techniques that already shade in texture space [8, 46, 73] and for VR setups where shading can be reused for both eyes, TAS is highly effective.

6.2.4 Free-moving virtual reality experiment

To demonstrate the flexibility and applicability of our approach for modern high-quality game content, we integrated TASA into *Unreal Engine 4* and conducted a small user experiment in VR. For this purpose, we adapted the *Showdown VR Demo* scene[5], a slow-motion fly-through of a combat scenario involving several soldiers fighting a giant animated robot. *Showdown* is likely one of the most challenging test scenarios for TAS, as it contains a large amount of highly reflective surfaces, with view-dependent reflections throughout the whole scene, as shown in Figure 6.4b. The comparison to the threshold T is evaluated after tone mapping with the Academy Color Encoding System (ACES) Filmic Tonemapper used in Unreal Engine 4.

For our VR user experiment, we slightly modified the scene by subdividing large primitives that exceed the maximum block size in the shading atlas. Additionally, we modified the scene to include fully dynamic directional lighting with cascaded shadow mapping, which was only approximated in the original scene.

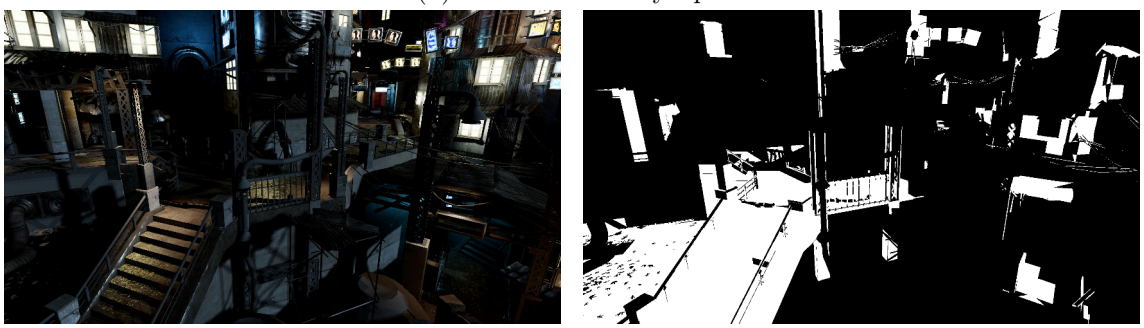
Eight participants (6 male, 2 female, age 24 to 33, with VR experience) tried the SA baseline, followed by TASA configurations using the thresholds $[4, 8, 16, 32, 64]$ and SAU with $4\times$ and $8\times$ upsampling in a randomized order. They were asked to describe their subjective experience during and between all runs. We specifically told all participants to look for visual artifacts of shading, lighting, shadows and reflections.

The atlas size for both SA and TASA was 8 MPx. We used an Intel Core i7-8700K with an NVIDIA RTX 2080Ti and displayed on an HTC Vive at a resolution of 1512×1680 per eye, using a fixed frame rate of 90 Hz. For TASA with $T = 64$ and $T = 32$, all participants detected artifacts, where $T = 64$ was "very bad", and $T = 32$ was "adequate with some annoying artifacts". For $T = 16$, four participants reported an identical experience

[5] `https://www.unrealengine.com/marketplace/en-US/product/showdown-demo`

(a) *Sponza* by Crytek

(b) *Showdown VR* by Epic



(c) *Soul City* by Epic

Figure 6.4: Practical temporally adaptive shading applied to three different scenes using our proposed approach: (a) The *Sponza* test scene contains physically animated boulders and moving spotlights. (b) Our technique implemented in Unreal Engine applied to the *Showdown VR* demo sequence, which contains a large amount of view-dependent effects, including specular highlights and reflections on shiny surfaces. (c) Even large amounts of reflective surfaces as well as animated, wet materials of the *Soul City* scene are handled by our approach without manual fine-tuning. The images on the right side show the steps underlying our approach: (a) We rely on shading gradients to estimate shading changes and the potential reuse of shading. (b) Using absolute shading differences that are maximum filtered, both per primitive and in an image space neighborhood, works well in practice. (c) The final shading decision, based on the previously computed shading differences, reduce shading by 57 % to 90 %, depending on dynamics, while staying visually indistinguishable from full shading in every frame.

(a) SA baseline (b) TASA ($T = 16$)

Figure 6.5: An example of an artifact of the TASA proof of concept running in VR: (a) Ground truth shading of a light fading to black after having a constant intensity for several seconds. (b) The lighting on the helmet is not updating uniformly due to TASA wrongly predicting the shading time, resulting in artifacts at geometric borders.

compared to the baseline, while the remaining four detected minor artifacts on shadows, reflective surfaces and the soldiers in the scene. These artifacts on the soldiers can be seen in Figure 6.5, and are related to the muzzle flashes of their guns, where a light source is turned on for a while, and then turned off again. Such lighting events are impossible to predict, since large shading gradients (when the lights appear) are interleaved with almost no shading gradient (when the lights stay at constant intensity for several seconds). For $T = 8$, six participants reported an identical experience compared to the baseline, while two participants were still able to identify minor artifacts on the soldiers. For $T = 4$, no participant was able to detect any visual artifacts, and all participants reported an identical experience compared to the baseline.

For SAU with 4× upsampling, four participants reported artifacts related to "jittery" and "flickery" motion, and they reported "low frame rate" for the reflections. When looking at SAU with 8× upsampling, all but one participant reported major artifacts of reflections and shadows, as well as major discomfort, particularly describing the experience as "very uncomfortable when moving around". One participant even reported a mild case of motion sickness. Overall, constant temporal upsampling is more likely to be perceived as jittery, which according to the participants is more discomforting and distracting than the artifacts of TASA, even for large thresholds.

TASA with $T = 8$ resulted in a mostly identical experience to the baseline, in accordance with results in section 6.2.2. This configuration shows a reduction of average shader invocations by 65 % (see Table 6.1), indicating that TASA is able to effectively reuse shading over long periods of time, while shading other regions almost every frame. Due to the different frame rates, results of SAU cannot be directly compared to the results

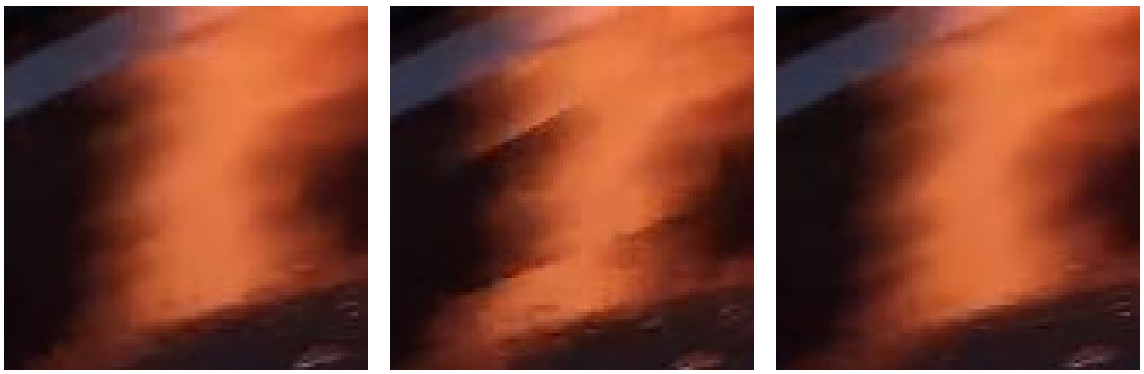
Table 6.1: Average shader invocations and relative shader invocation reductions of the *Showdown VR Demo* scene measured for all tested methods. Given roughly equal shader invocations (TASA $T = 16$ vs. SAU 4×, TASA $T = 32$ vs. SAU 8×), study participants prefer the adaptive shading of TASA over the constant upsampling of SAU, showing that TASA provides a more optimal performance-quality tradeoff compared to SAU.

Method	Shader Invocations	Reduction to SA
SA	4.49 M	
SAU 4×	1.14 M	74.55 %
SAU 8×	0.56 M	87.45 %
TASA $T = 4$	2.02 M	55.06 %
TASA $T = 8$	1.57 M	65.03 %
TASA $T = 16$	0.97 M	78.29 %
TASA $T = 32$	0.61 M	86.48 %
TASA $T = 64$	0.36 M	91.95 %

of Section 5.3, since shading with constant upsampling factors then also runs at different frame rates.

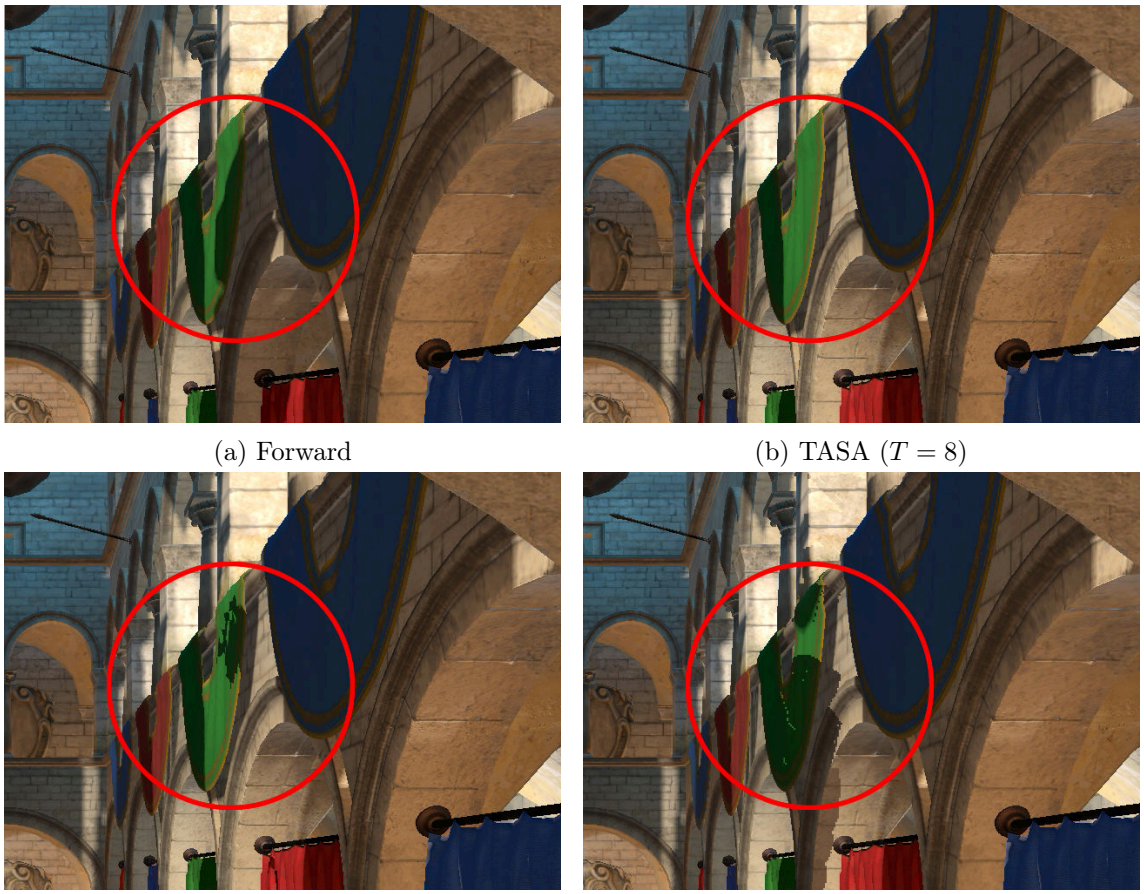
TASA does not depend on slow animations and movement to achieve this result; it handles fast head movements and animations equally well, while constant upsampling produces noticeable artifacts. An example of such an artifact can be seen in Figure 6.6. Avoiding those artifacts is especially important for fast-paced VR gaming, where fast head movements and animations are common.

Overall, our early prototype of TASA in Unreal Engine 4 shows promising results, with no additional fine tuning required to achieve results which are mostly indistinguishable



(a) SA baseline (b) SAU, 4× upsampling (c) TASA, $T = 0.01$

Figure 6.6: (a) A highly view-dependent reflection on the street of the *Showdown VR Demo* scene (bottom left of Figure 6.4b) rendered with the ground truth SA baseline. (b) A constant upsampling rate of 4× produces noticeable artifacts. (c) TASA correctly predicts the shading changes and avoids artifacts independent of camera or animation speed.



(a) Forward (b) TASA ($T = 8$)

(c) TASA ($T = 16$) (d) TARC ($T = 16$)

Figure 6.7: TAS cannot predict shading changes that exhibit neither temporal nor spatial coherence. The boulders in our *Sponza* scene generate distant shadows that first appear outside of the screen and thus are not picked up by our spatial filter. Depending on when shading samples are accidentally reshaded, some of those effects may be picked up: The shadow on the green cloth is completely missed by TASA ($T = 8$); TASA ($T = 16$) and TARC ($T = 16$) capture parts of it. Such failures could be caught with custom improvements, such as projecting shadow maps into the current view.

to the baseline, while reducing shader invocations by 65 %, even for the challenging slow-motion scene. Note that, to illustrate the capability of our approach to work out of the box, we did not perform any fine-tuning. More scene-specific tuning could increase shading reuse even further.

6.3 Limitations

The prediction when to shade is done with a combination of a linear temporal model and a spatial filter. A linear model for temporal changes may be simple and could be

improved, but we found it to work sufficiently well. The results of our user study show that the technique is temporally stable, and users hardly noticed any artifacts when a proper threshold is used. Our spatial filter for shading gradients catches effects such as moving highlights and moving shadow boundaries. However, the simple box filter with a rather big kernel size used in TARC leads to considerable amounts of unnecessary shading. A more advanced spatial filter could consider spatial gradients such as optical flow to resolve these issues at the cost of increased runtime and complexity. While the existing measures capture most changes, some less frequent ones can still cause artifacts—for example, discontinuous rendering (e.g., lights switching on or off) or changes that propagate from outside the image (Figure 6.7) or from an occluded area. Depending on the use case, specialized cases such as discontinuous changes, e.g. to light sources, can be caught on the scene object level. A more general solution to capture artifacts from discontinuous shading could speculatively update samples that are not due yet.

Furthermore, our evaluation is based on a single threshold applied to the per-channel maximum RGB color difference after tone mapping. This is can be seen as an effective solution based on Weber's law [12] and similar to the threshold used by Walter et al. [109]. Our constant threshold assumes a constant base luminance. However, the threshold might be too conservative in certain areas of the HDR spectrum and overlook additional gains. We tested a simple uniform threshold in HDR, but did not achieve satisfactory results; a more advanced method may be necessary.

A method for deriving the threshold for noticeable differences from the perception of the human visual system has the potential to lead to further temporal savings. This can possibly be done in a different color space or in the HDR space before tone mapping. For example, a higher threshold could be used for dark pixels that are close to bright ones, or a lower threshold needs to be used in dark areas where the visual system is more sensitive. The CIEDE2000 [64] color difference metric based on the CIELAB color space may prove advantageous over the RGB based metric and better reflect the perception of luminance [2] and chrominance.

While object or texture space storage of shading samples has clear advantages over screen space, these methods do come with their own limitations. The shading atlas shades pairs of triangles within rectangular blocks with power-of-two side lengths. When the level of detail changes, a block of a different size is allocated and shading cannot be reused. The change of the resolution is sometimes noticeable as popping of the textures. Furthermore, the sample distribution within the atlas necessitates some oversampling to achieve the same quality as forward rendering. While future methods for object or texture space storage for shading reuse may overcome these limitations, the shading atlas in its current form is already practical as a cache for real-time rendering, since it requires no additional GPU extensions and supports streaming.

Chapter 7

Shading Atlas Streaming

Contents

Shading atlas streaming (SAS) is the first object-space shading method that targets streaming for VR content. It is based on the shading atlas rendering pipeline discussed in the previous chapters, which is decoupled between a powerful server and a lightweight HMD client device. This chapter will first detail the system architecture of rendering with the shading atlas and streaming the required information between server and client. Following that is an extensive evaluation of the system, including specifics about the shading atlas that also apply outside the streaming context.

7.1 System architecture

To understand the design constraints of a remote rendering system using object-space shading, we begin with an overview of the server-client pipeline (Figure 7.1). The end-to-end latency for a round trip from client to server and back can be broken down as follows:

1. The client sends the current view matrix to the server. This duration is owed to network latency. It can be changed with faster networking technology.

2. The message waits at the server until a new frame starts rendering. This duration depends on how frequent the client sends pose updates, since the most recently received pose is used for the next frame.

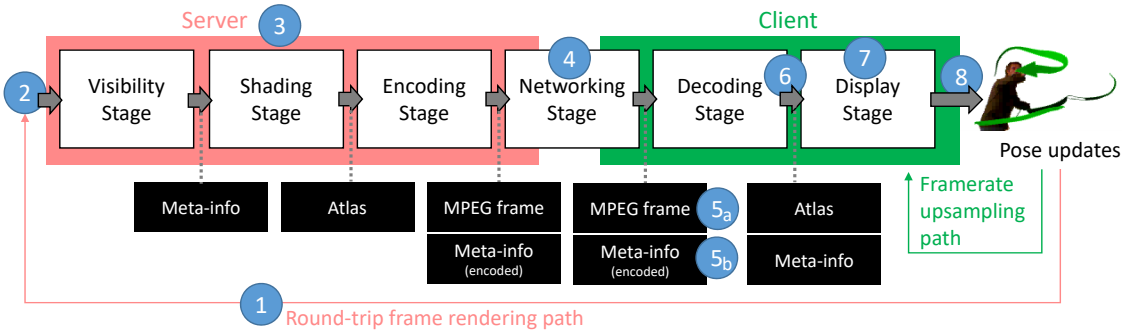


Figure 7.1: Our pipeline is split across a server and a client part, with the shading atlas as the central data structure connecting the two. Camera pose updates generated by the user are sent upstream, on a slow (networked) path to the server for rendering new shading into the atlas, and on a fast (direct) path to the client to render new images to the display.

3. The server renders the atlas frame. This duration corresponds to the processing times for visibility, shading and encoding stages. It depends on server CPU and GPU performance.

4. Atlas frame and corresponding meta-information are sent to the client. This duration is owed to network latency.

5. The client decodes (5a) the atlas frame and (5b) the meta-information. This duration corresponds to the maximum of client's processing time for the two decoding tasks, since they are carried out by different hardware units. The duration depends on client CPU and GPU performance.

6. The received data waits at the client until a new frame starts rendering. This duration depends on the client frame rate and varies between 0 and the client frame time.

7. The client renders the final frame. This duration corresponds to the client's processing times for the display stage. It depends on the client's CPU and GPU performance.

8. The final frame is displayed. The most recently received server frame is used to render additional frames at the client (*frame rate upsampling*). Thus, this duration increases with every upsampled frame, until a new atlas frame arrives.

The end-to-end latency of a streaming system not only depends on the network latency, but also on the server's frame rate for sending data and the client's frame rate for displaying new images. If the server has a lower frame rate than the client, this frame rate upsampling induces additional variable latency on top of the network latency. However, lowering the server frame rate has several advantages. First, the server can render higher quality results, as it has more time per frame. Second, a lower server frame rate can be encoded at a lower bitrate. Third, the client has to decode fewer frames, benefiting from faster processing and

power savings. We have found that these advantages outweigh the overhead of shading all polygons in a PVS, even when the PVS becomes large, since it needs to cover the latency and the frame rate upsampling in order to cover the client's views between updates.

The system design must balance server and client frame rates to deliver optimal quality for a given bitrate. In an optimal pipeline, every stage must be both fast and avoid any loss of image quality. In the remainder of this section, we discuss the design decisions of the various pipeline stages, before we report on a system evaluation in Section 7.2.

7.1.1 Visibility stage

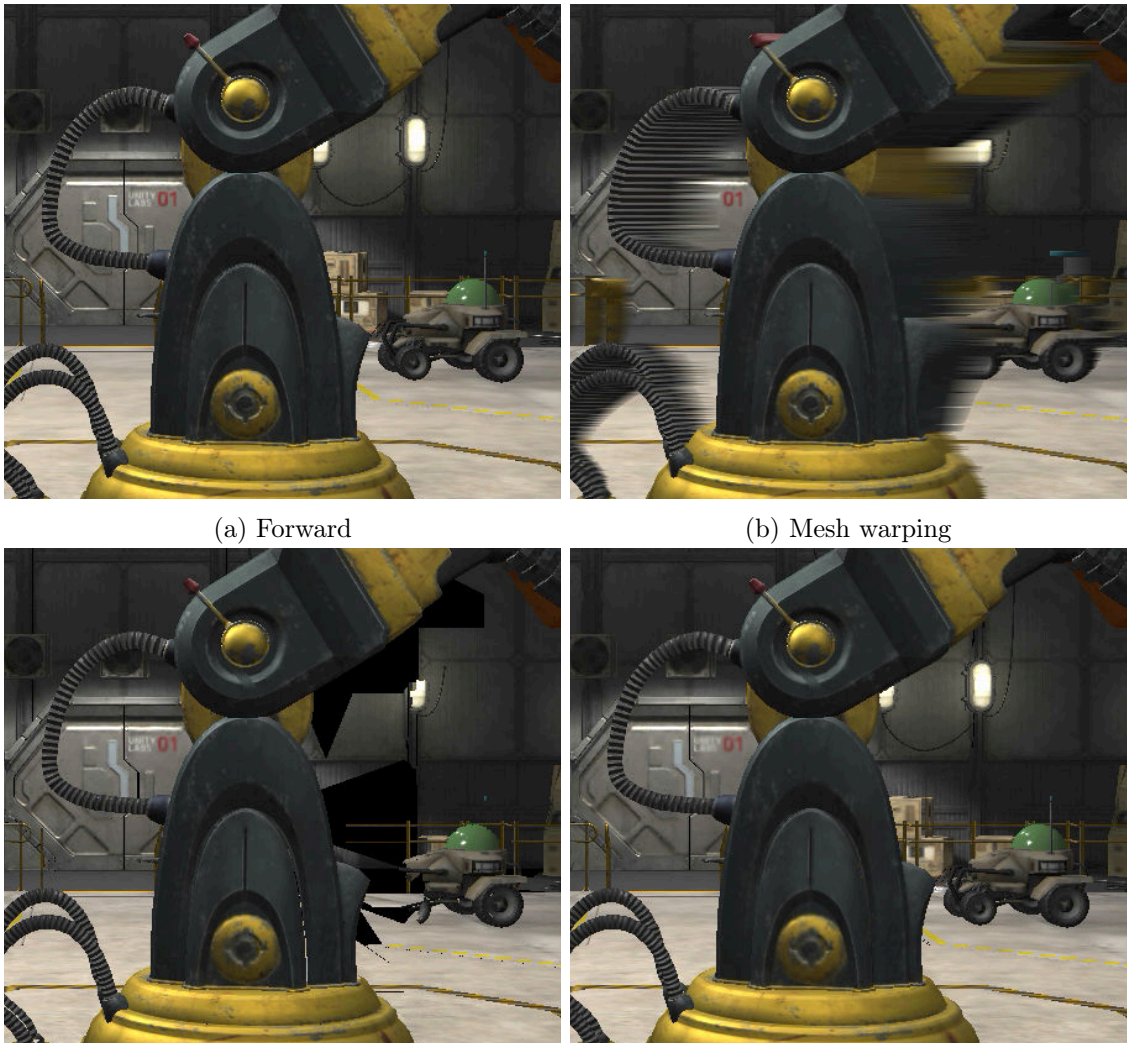
The target of the visibility stage is to determine which geometry is visible and needs to be shaded. It addresses the main problem IBR methods have with frame rate upsampling and latency – disocclusions. While disocclusions provoked by camera rotation can be avoided with an extended FOV, disocclusions provoked by camera translation would need additional views of the scene. Unlike most IBR methods, SAS only has disocclusion problems if the disoccluded geometry is not part of the PVS (Figure 7.2). We compute the PVS by predicting the future viewpoints that will be used at the client and include the associated visible geometry into the PVS.

The unit for determining visibility is a *patch*, i.e., a group of two or three adjacent triangles (Figure 3.3), which are suitable for dense atlas packing [20]. In Section 3.6.1, we demonstrated that this fine-grained visibility determination is necessary to keep the atlas size small. Assignment of triangles to patches is determined during preprocessing with a heuristic optimization; solo triangles are used only when unavoidable.

The visibility stage renders a patch index buffer, with depth buffer enabled. A subsequent compute pass marks the patches in the index buffer as visible, effectively computing an EVS. To extend the EVS into a PVS covering the geometry that may become visible as the camera moves, we combine two measures (Figure 7.3):

- We enlarge the FOV at the borders by increasing the resolution of the index pass and adapting the projection matrix to keep the original FOV and resolution in the center of the index buffer. This procedure is typically applied in IBR methods to ensure visibility for small rotations of the view.

- Based on past camera motion, we extrapolate motion for a short period into the future and sample an EVS for multiple viewpoint locations, starting from the current viewpoint location. The PVS is determined as the union of all patches contained in at least one EVS sample. We heuristically sample the EVS several times, using either prediction based on the visual-inertial tracking system of the client device or simple linear extrapolation (for deterministic testing). We also consider the motion of animated objects corresponding to the predicted time for which an EVS is created. Note that this only applies to deterministic object motion or object motion that can be reliably predicted. We do not consider speculative rendering [56].



(a) Forward (b) Mesh warping

(c) Shading atlas (EVS) (d) Shading atlas (PVS)

Figure 7.2: (a) In this example, the camera is panning quickly to the right with a streaming latency of 83 ms. (b) Image-based rendering methods typically suffer from artifacts caused by disocclusions, when the viewpoint is translated. (c) When the shading atlas works only with an exact visible set, some geometry is missing in disoccluded areas. (d) SAS can avoid artifacts by predicting future viewpoints and rendering all geometry in the corresponding potentially visible set.

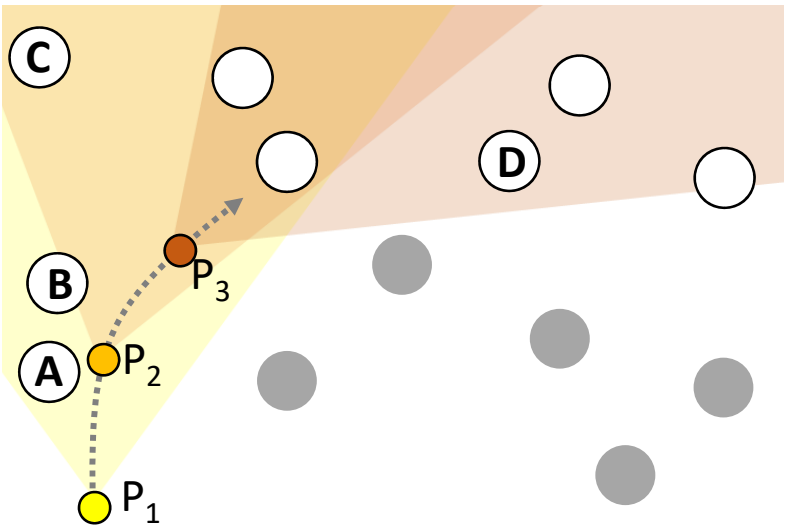


Figure 7.3: From the current viewpoint P_1, two future viewpoints, P_2 and P_3 are predicted, with the corresponding FOV shown in color. An EVS is computed for each of the three viewpoints, and the PVS is determined as the union of all objects visible in any EVS (visible objects are marked as white circles). In the example, object A is only visible from P_1, but not from P_2 or P_3. Conversely, object D is only visible from P_3. Object C is jointly occluded by objects A and B from P_1, but becomes visible from P_2.

The success of a sampling approach to PVS depends on the quality of the motion prediction. For short look-ahead periods, we found our simple approach to deliver perceptually correct results. If the PVS computation misses a triangle that would otherwise be visible, the client cannot render this triangle, and a hole may appear. We report in Section 7.2.1 on the correctness of the PVS computation and demonstrate that missed triangles are rare.

7.1.2 Shading stage

Since visibility, level selection, memory management and shading are all coupled in this streaming implementation using the shading atlas, we combine these phases into one stage in this chapter. Following the PVS determination, patch sizes are determined based on the projected area of a patch on the screen as detailed in Section 3.5. Atlas memory management—allocating and deallocating space for patches—is directly derived from the PVS. Details on the atlas memory management are given in Section 3.4. The atlas memory management writes a vertex buffer containing the visible geometry including corresponding texture coordinates for the shading atlas. This vertex buffer is used for shading, as described in Section 3.1.

7.1.3 Encoding stage

In the encoding stage, the updates to atlas and meta-information that must be sent to the client are prepared for network transmission. We collect changes to the PVS data structures directly on the GPU during the previous stages, so that a single readback to the CPU in the encoding stage suffices.

The network load is dominated by the atlas, while the meta-information consumes comparatively little bandwidth. However, the atlas content is temporally coherent, as both the visible triangles and their shading changes only slowly with a moving viewpoint. Therefore, the atlas can be efficiently encoded as an MPEG stream.

Hardware-accelerated MPEG encoding on the GPU is very efficient and offers sufficient throughput for our use case. Therefore, we designed our atlas in an MPEG friendly way, as a single large texture filled with temporally coherent shading loads. MPEG encoders maintain internal state, which makes it expensive to run multiple encoders in parallel. Therefore, storing the shading information as a collection of variable-sized per-object textures [8] would be difficult to fit to MPEG encoding.

We optimize our video coding configuration for low complexity and low latency. We encode the texture atlas using the H.264 video coding standard. The compressed frames are small enough so that the readback time to the CPU is negligible. Moreover, readback (and subsequent network transmission) can run in parallel with rendering of the next frame.

The client has no need to know of the memory management associated with the atlas. The client only requires visible triangles with corresponding vertices and texture coordinates pointing into the atlas. Thus, we encode the vertices, triangles, and texture coordinates for the shading atlas as meta-information. Vertices and triangles are only transmitted once, when they first become visible. Texture coordinates are transmitted periodically, whenever patches are re-located inside the atlas. Rigid body animations are supported by tracking per-object transformation matrices. Animations computed in the vertex shader must be transmitted in every frame, provided the vertex is visible. Note that we implemented meta-information transmission to demonstrate an operational end-to-end system. However, meta-information transmission is not yet optimized and not explicitly considered in the system evaluation.

7.1.4 Networking stage

The network transmission must consider two channels with different characteristics. The atlas is transmitted as an MPEG stream over a potentially lossy channel optimized for throughput. For this channel, we rely on RTP over UDP. Since the UDP connection is prone to packet losses, the video encoder inserts I-frames at regular intervals (typically every five frames) to stop error propagation in the decoded atlas due to a packet loss.

The meta-information (geometry, patches, and animation updates) use a comparatively small bandwidth, but must be transmitted over a reliable channel to ensure that the scene structure at the client is kept intact. We considered using a custom sub-channel of the

MPEG stream [29] or APP messages of RTCP for this purpose, but ultimately decided for a simple TCP connection, because it trivially ensures reliable transmission.

7.1.5 Decoding stage

The client decodes received messages immediately upon arrival and updates its internal structures. MPEG decoding on the mobile device is handled by a dedicated hardware unit. The meta-information is decoded on the GPU.

Since the client prototype receives network updates over two separate connections, it must ensure that both streams are synchronized after each frame is received. Decoding and display stages run in multiple threads at different update rates and communicate via triple buffering to avoid any latencies from locked buffers.

7.1.6 Display stage

The display stage at the client relies on a very simple forward rendering pass. The client issues a single draw call on a vertex buffer that has been generated by processing the incoming server data. This vertex buffer already contains the correct texture coordinates for the atlas, requiring no indirect lookup. Rendering is decoupled from the remaining pipeline; if an update has been received from the server and is ready to be used, the rendering thread simply switches buffers and continues rendering at full speed.

Since we have a direct uv-mapping from vertices to shading atlas, no projective or indirect texture mapping is required, as necessary in other IBR methods. The variable shading resolution baked into the atlas even relieves the client from having to perform mipmapping. The remaining requirements of rendering to a VR headset are taken care of in two sub-stages:

1. frame rate upsampling uses the latest pose from the head tracker for rendering, while the shading information in the atlas is generated based on an earlier pose. Moreover, a stereo image pair is generated by rendering twice, with a statically calibrated offset for left and right eye. The results of the first sub-stage are stored in a framebuffer object.

2. We compensate for the barrel distortion induced by the magnifying lenses in the VR headset by applying radial corrections for every pixel on the screen. The distortion is considered separately in each RGB color channel to compensate for the color abberations of the lens system.

7.2 Evaluation

Our client software prototype has been implemented using the Vulkan graphics API and runs under Android. Tests were conducted on a desktop PC and a headset prototype based

Figure 7.4: Untethered smartphone-based headset prototype based on the Qualcomm Snapdragon^{TM} 835 Mobile Platform, with a resolution of 2560×1440, displaying a stereo view of the Robot Lab scene.

on the Qualcomm Snapdragon^{TM} 835 Mobile Platform (Figure 7.4). The PC was used for frame-by-frame tests (sections 7.2.2-7.2.4) and as a server for end-to-end timings with the headset as the client (Section 7.2.5). For each test scene, we recorded a camera path with a total length of 10 seconds (1200 frames).

In our tests, we used a rendering target for the client with a resolution of 1920×1080 at a horizontal FOV of 90°. To compute the PVS for the shading atlas and for the evaluated IBR methods, we enlarge the server-side FOV by 20%. Moreover, the PVS uses linear extrapolation of the camera path based on the two most recent view matrices. Unless otherwise noted, prediction is done in steps of 33.3 ms. The number of prediction steps varies based on the overall maximum latency, which depends on an assumed static network latency and the server frame rate. For example, at a server frame rate of 30 fps (33.3 ms per frame) with a static latency of 100 ms, the maximum latency is 133.3 ms. In this case, prediction intervals of 33.3 ms, 66.6 ms, 100 ms and 133.3 ms are used.

7.2.1 PVS prediction

We demonstrate that our sampling strategy to determine a PVS is effective. Since we want to avoid disocclusion coming from PVS underestimation, the main interest lies on the false negatives, i.e., the patches that are not classified as part of the PVS, yet become visible. We also record the false positives, i.e., the patches that are reported as part of the PVS, although they never become visible.

We compare the following modes: *Linear prediction* uses a linear extrapolation of motion in five steps, with the EVS computed at 120 Hz (0 ms, 8.3 ms, 16.7 ms, 25 ms, 33 ms) and added to the PVS. *Reference* prediction works like linear prediction, but using

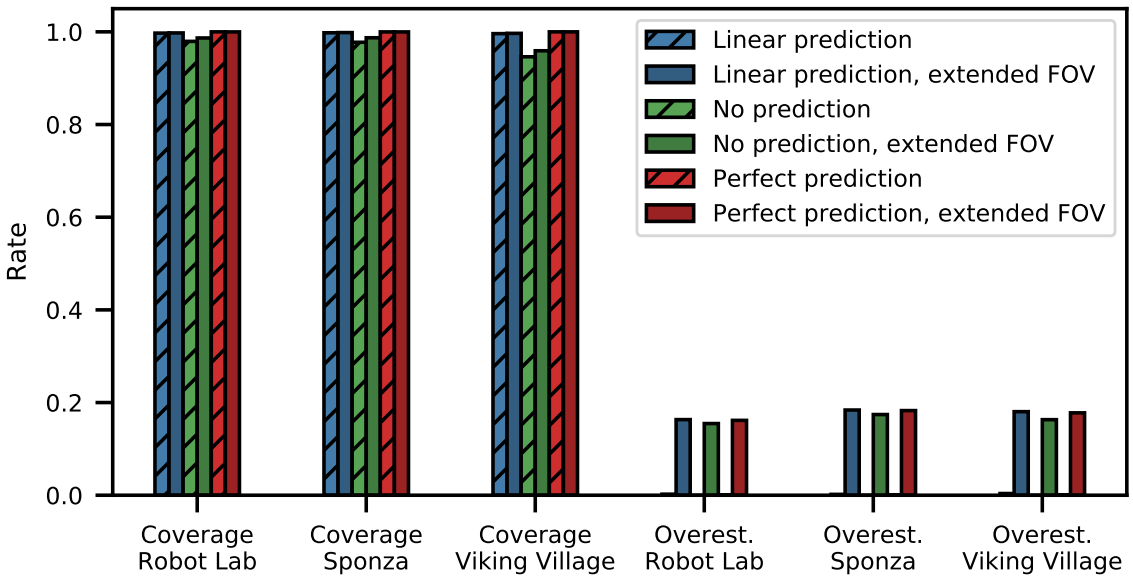


Figure 7.5: For PVS computation, we report coverage rate (how many visible patches were not overlooked) and overestimation rate (how many patches were needlessly classified as visible).

the actual (recorded) camera path instead of a predicted one. *No prediction* uses the EVS sampled at 0 ms as the PVS, establishing a baseline.

Let v denote the total number of patches visible over the five EVS sample positions, computed by reference prediction and weighted by the projected area of the patches' triangles in screen space, as a measure of visual importance. Similarly, let f_n denote the number of area-weighted false negative patches, and, f_p denote the number of area-weighted false positive patches. We measure the coverage rate $r_c = 1 - f_n/v$ and the overestimation rate $r_o = f_p/v$, both for the regular FOV and the extended FOV.

Figure 7.5 shows averages for our test scenes. We see that even simple linear prediction achieves an average coverage rate >99 %. In contrast, no prediction has an average coverage rate of 96 %. We expect that the result of no prediction would be even lower for a faster moving camera. We conclude that our PVS algorithm is suitable for frame rate upsampling in the considered interval.

The main reason why linear prediction misses a small percentage of the triangles is geometric aliasing from rasterizing tiny, sub-pixel sized triangles in the test scenes. Such tiny triangles can occasionally produce isolated flickering. They can be avoided with better level of detail management (see Section 7.3.4). A similar problem with numerical limitations of rasterization precision causes reference prediction to miss the 100 % mark for the extended FOV.

As expected, the overestimation of the regular FOV is negligible, but the overestimation of the extended FOV compared is around 10 %. It could be reduced by using a more modest setting for the extended FOV, provided a better prediction model is used.

7.2.2 Image quality vs network latency

Conventional streaming combines forward rendering with frame rate upsampling via IBR. The key measure for a good streaming solution is the relationship of image quality to pixel rate (i.e., pixels per second that must be transmitted).

We compare SAS against three IBR methods. ATW is the most simple IBR method which just applies a homography transformation to the source image. Mesh Warping (MW) [69] uses a dense vertex grid with one quad per pixel and transforms the grid based on the depth buffer. Iterative Image Warping (IIW) [117] reverses the warping flow and searches for an input pixel starting from the output pixel.

The experiment shown in Figure 7.6 varies the (simulated) network latency, while keeping the client and server frame rate fixed at 120 Hz. Without latency, IBR methods have the advantage of using exactly the same input view as the ground truth, while SAS depends on the atlas size.

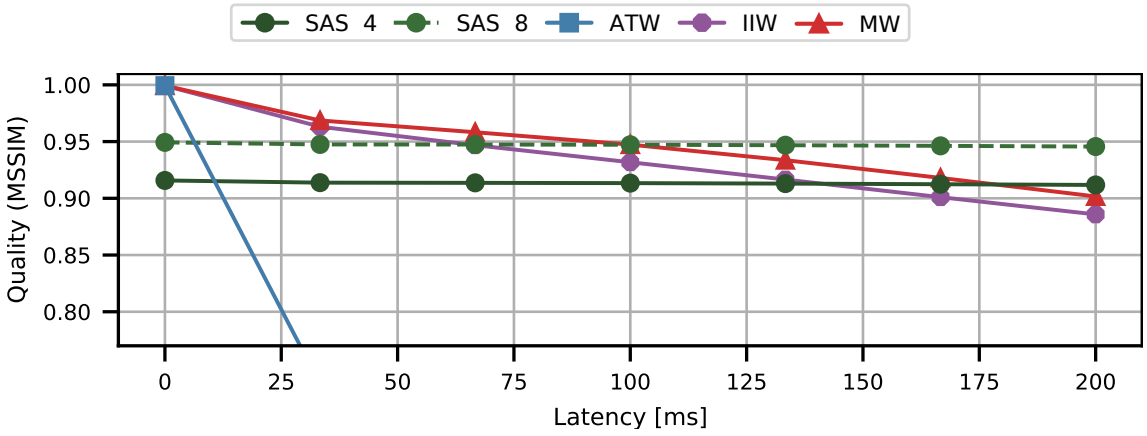
To measure image quality, we use the structured similarity image measure (SSIM) [110]. We compute the SSIM using a Gaussian filter with a window size of 11 pixels and a standard deviation of $\sigma = 2$, and constants $C_1 = 0.0001$ and $C_2 = 0.0009$. The ground truth for the comparison uses the scene rendered with a forward rendering pass. We compute the mean SSIM (MSSIM) for the whole sequence of 1200 frames.

As can be expected, quality decreases at different rates as network latency increases: While the quality of IBR methods starts to decrease immediately, SAS manages to keep the quality at a high level due to its geometric model. For large scenes, the crossover point is already at 20 ms. This is noteworthy, since end-to-end latency is typically much larger (see also Section 7.2.5).

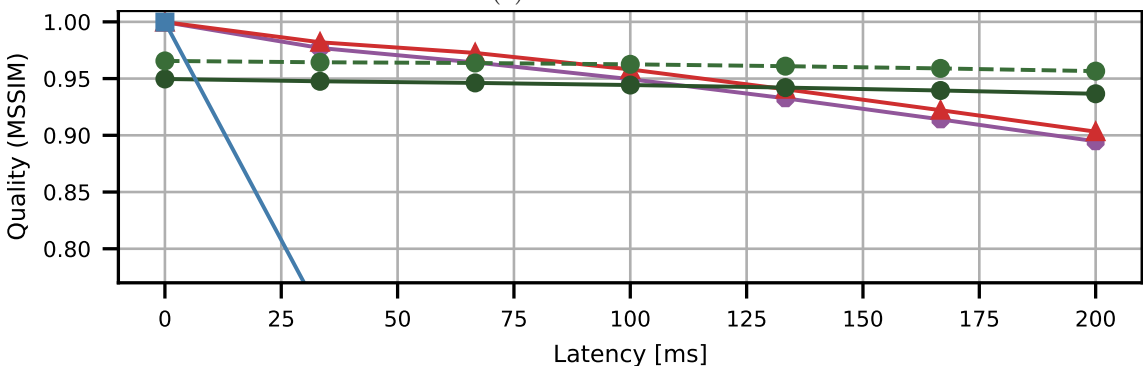
7.2.3 Image quality vs server-side upsampling rate

In this experiment, we vary frame rate upsampling rather than network latency. The latter is set to zero. The client frame rate is fixed at 120 Hz, while the server frame rate varies (7.5, 15, 30, 60, and 120 Hz). Increasing the frame rate upsampling can be seen as introducing variable latency (duration (8) in Figure 7.1) from zero to a maximum just before a new server frame arrives. The average latency resulting from a server frame rate of K Hz is $\frac{1}{2K}$ s. The results in Figure 7.7 are consistent with this average latency estimate. We conclude that average latency resulting from frame rate upsampling can be added to network latency in our system model.

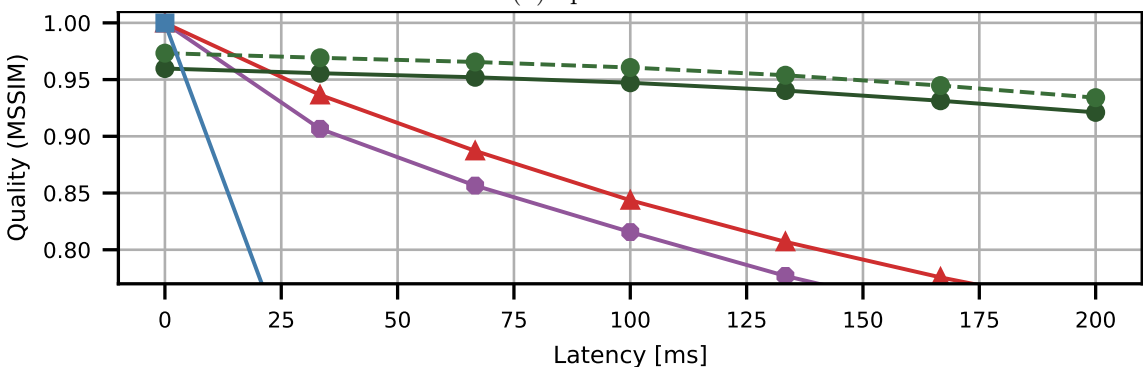
ATW cannot compete with the other methods quality-wise. SAS beats MW and IIW at upsampling rates of 4× (30Hz) or higher for Viking Village, but not for the smaller scenes, Robot Lab and Sponza. However, in practice, the effects of latency from frame rate



(a) Robot Lab

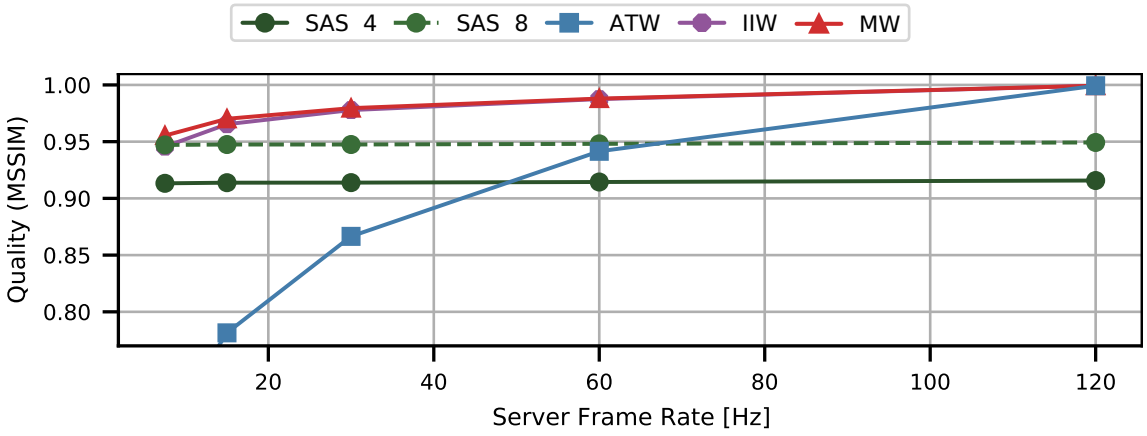


(b) Sponza

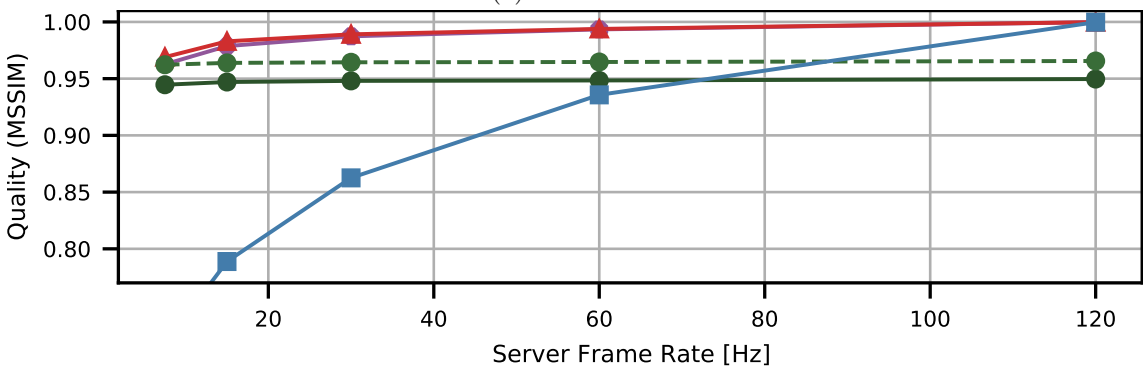


(c) Viking Village

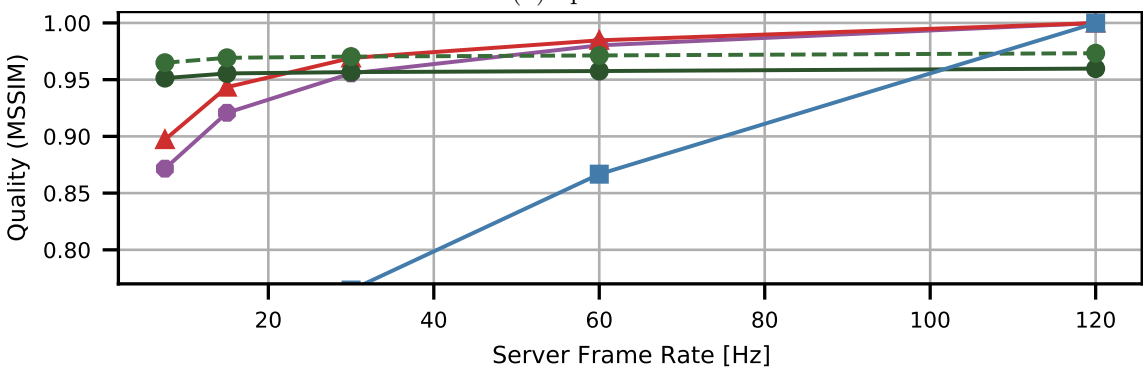
Figure 7.6: Image quality (reported as MSSIM) for varying network latency, compared for SAS (using multiple atlas sizes). As latency increases, SAS maintains image quality better than asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).



(a) Robot Lab



(b) Sponza



(c) Viking Village

Figure 7.7: Image quality (reported as MSSIM) for varying upsampling rate, compared for SAS (using multiple atlas sizes), asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).

upsampling must be added to network latency, which is at least 60ms (see Section 7.2.5), giving SAS an advantage for all scene sizes.

7.2.4 Rate distortion

All experiments presented so far did not consider the impact of lossy MPEG compression on image quality. In an end-to-end system, the main concern is network utilization. Consequently, we report a rate distortion curve, relating image quality (measured as SSIM) to bitrate (Mbps after compression).

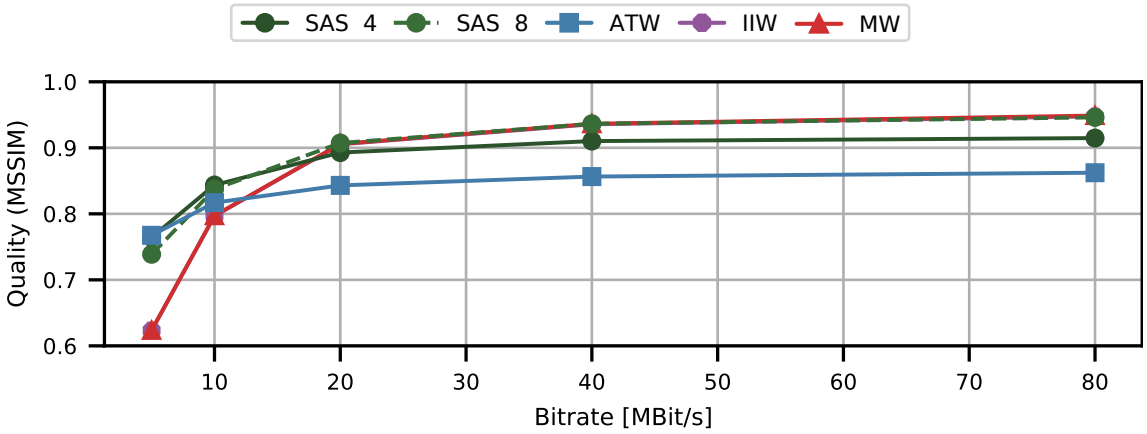
We compare SAS with atlas sizes of 4 and 8 MPix to ATW, MW and IIW. The experiment uses a server frame rate of 30 fps, while the client target frame rate is 120 fps. Network latency was assumed to be zero. First, the server generates uncompressed texture atlases. Then, we apply H.264 video coding, with CAVLC entropy coding, a single reference frame, and no B-frames. We vary the bit-rate in 2× steps (5, 10, 20, 40, and 80 Mbps). Atlas reset and I-frame interval is set to five frames. The decoded atlases are passed to the client, and are used to render the output frames at the target frame rate.

Figure 7.8 presents the results for our test sequences. Even at a bitrate of 10 Mbps, ATW is already beaten by the other methods. For the smaller scenes, MW and IIS benefit more from higher bitrates, outperforming SAS at 20-40 Mbps. However, this measurement does not account for the need of MW and IIS to encode stereo pairs, so the actual crossover point would be at half this bitrate. Moreover, all methods except ATW have SSIM above 90%, which is already a very high image quality with little room for improvements. For the large scene, Viking Village, SAS always delivers the best image quality, by a large margin. In general, we observe diminishing returns for bitrates above 40 Mbps in the tested configuration, approaching the quality of rendering from an uncompressed atlas.

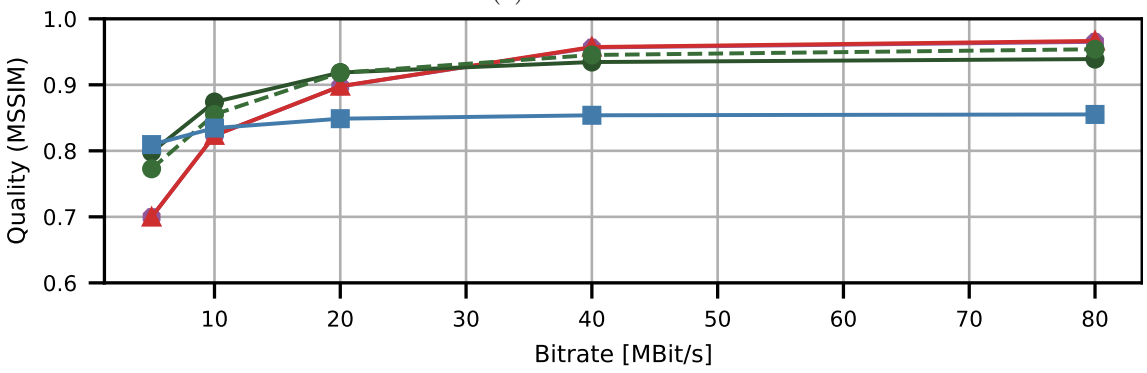
7.2.5 End to end performance

Finally, we report on system performance on a mobile client device based on the headset running Android. The headset has a single display (shared by both eyes) with 2560×1440 resolution at 60 Hz, driven by an Adreno 540 GPU. Typically, a mobile GPU has at least an order of magnitude less compute power than a high-end desktop GPU, like the GeForce used in our tests. Moreover, the larger scenes, such as Viking Village and Robot Lab, do not even fit in the RAM of the mobile device. Therefore, we cannot report on any native VR rendering results on the headset for comparison to SAS.

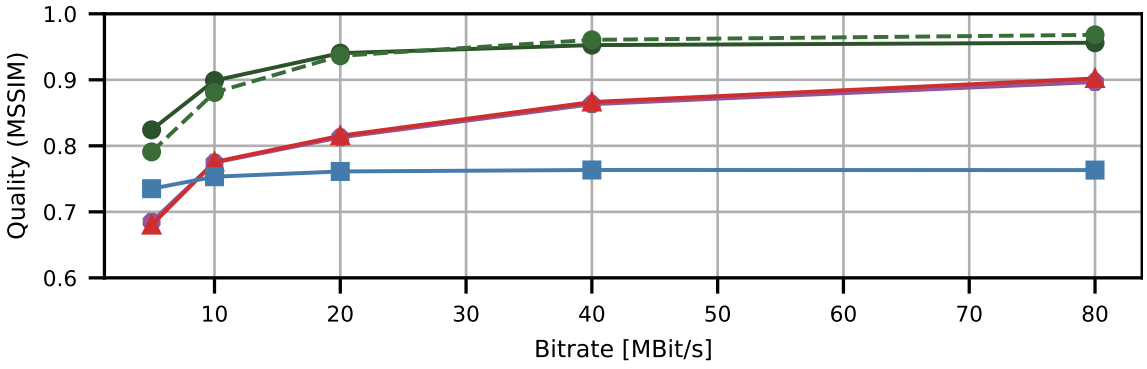
We connected the headset to the desktop computer by 802.11ac WiFi at a throughput of about 526 Mbps. The client code is not yet optimized, so overall results have to be interpreted with caution. Figure 7.9 reports performance figures for the various system stages (Figure 7.1). The server frame rate was locked to 30 Hz, and the client frame rate was unlocked. We tested the system with an atlas size of 4 MPix and high shading load (400 light sources).



(a) Robot Lab



(b) Sponza



(c) Viking Village

Figure 7.8: Rate distortion curves after MPEG compression. We report image quality (MSSIM) as a function of network bitrate for SAS (using multiple atlas sizes), asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).

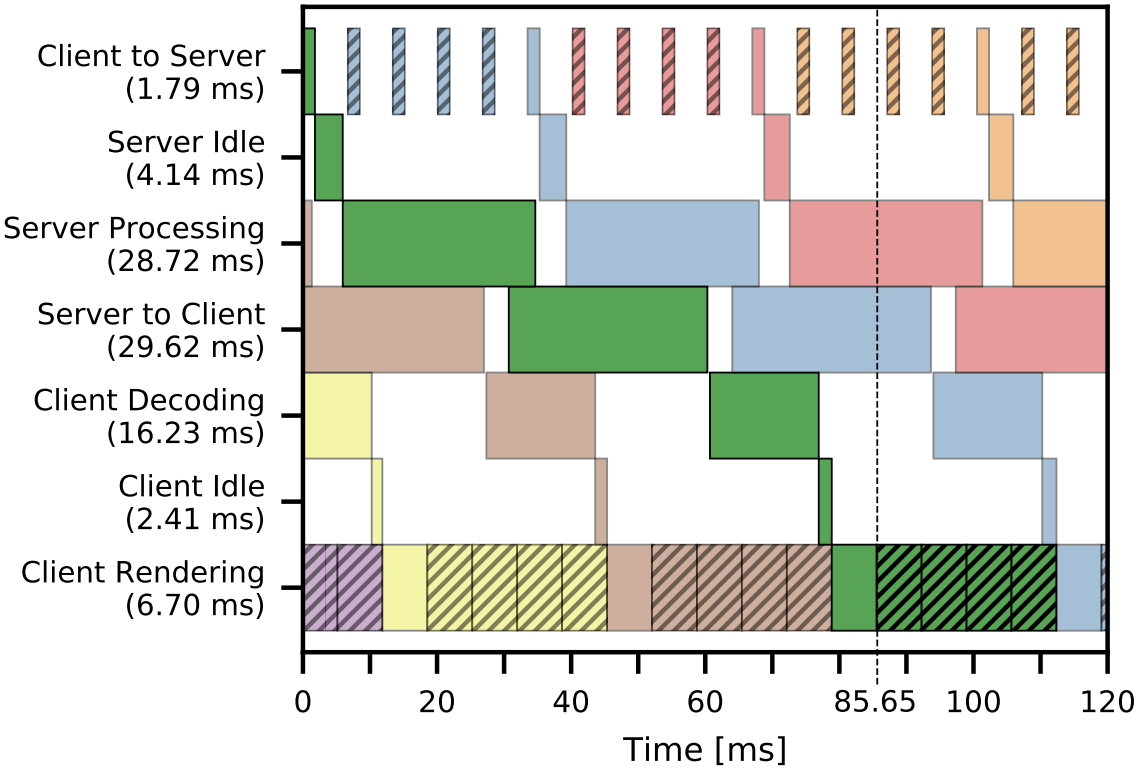


Figure 7.9: End to end latency for the Robot Lab scene, measured with the headset as the client device. We report median times for all stages in addition to the median end to end latency (dashed line). The 'Server to Client' and 'Client Decoding' stages show the timing of the longest sub-path.

The sum of phases 1-7 (until the first upsampled image appears) leads to a median total round-trip-latency of around 86 ms. Pose transmission time from client to server is about 2 ms; server idle time around 4 ms; server processing takes around 29 ms. MPEG transmission time is a significant factor at $\sim$ 30 ms, but it can be amortized by creating stereo pairs from a single atlas, and by frame rate upsampling. This early-generation headset is limited to 60 Hz, but future generations will be able to display 120 Hz, which works in favor of frame rate upsampling, as more frames can be amortized within the same prediction period.

Meta-information transmission takes a non-negligible amount of time of around 19 ms. We believe this is because of our inefficient implementation. First, we apply only rudimentary compression to this data (Section 7.3.5). Second, TCP should be replaced with a more throughput-oriented protocol. Occasional packet losses can be rectified during atlas reset.

The client decoding stage is taking around 16 ms total, whereby the atlas and the meta-information are decoded in parallel. The MPEG frame is sent to a dedicated hardware

decoder unit, at a rate of 1.7-2.2 Mpix/s (slightly higher for more complex shading). The meta-information is decoded on the GPU. As expected, MPEG decoding is the limiting factor, requiring roughly twice the time of meta-information decoding. While MPEG decoding will become faster with future generations of hardware, the meta-information decoding is currently most affected by scattered memory accesses. This problem can be addressed by a more batch-oriented encoding of meta-information.

The final part of the pipeline consists of waiting for the GPU (2 ms) and rendering the final image (7 ms). The last phase (illustrated using hatched rectangles in Figure 7.9) renders multiple frames using the same atlas, until new data from the server arrives.

While the throughput of our early prototype obviously needs improvements, we can already make two important observations: First, the fraction of the round trip latency that does not depend on the rendering method (network transmission, encoding, decoding, idle times) already exceeds 50 ms. We have demonstrated in Section 7.2.2 that SAS is a competitive solution at this latency. Second, the display stage at the client is very lightweight. At the observed 6.7 ± 2.6 ms for drawing an average of 32K triangles per frame, we can reliably achieve a frame rate of 120 Hz. Outliers of frame times rarely happen (< 0.6 % of all frames). We conclude that SAS is a valid approach for future generation VR headsets.

7.3 Limitations and extensions

In this section, we discuss limitations of our system, and we report some preliminary results on extensions that address them.

7.3.1 View-dependent rendering

The streaming implementation does not consider fast changing view-dependent effects, like crisp highlights. If shading information is re-used by a constant frame rate upsampling, fast camera motion can reveal incorrectly extrapolated shading. One obvious method to suppress such problems is by introducing temporally adaptive shading to streaming. Since the shading information belonging to a particular object is explicitly represented in the atlas, each object can be updated at any time. More important objects, or objects with highly view-dependent materials, could be updated more often than others. If an atlas is partially updated in this way, the bitrate required by an MPEG P-frame produced from such a partially updated atlas will be proportional to the amount of changes, and not the overall atlas size. Consequently, bitrate can be allocated based on object priorities without changing the streaming format.

7.3.2 Transparent geometry

Transparent geometry in the scene, for which IBR methods need specialized solutions [62], can be handled in SAS by adding a traditional (order-dependent) pass for transparent

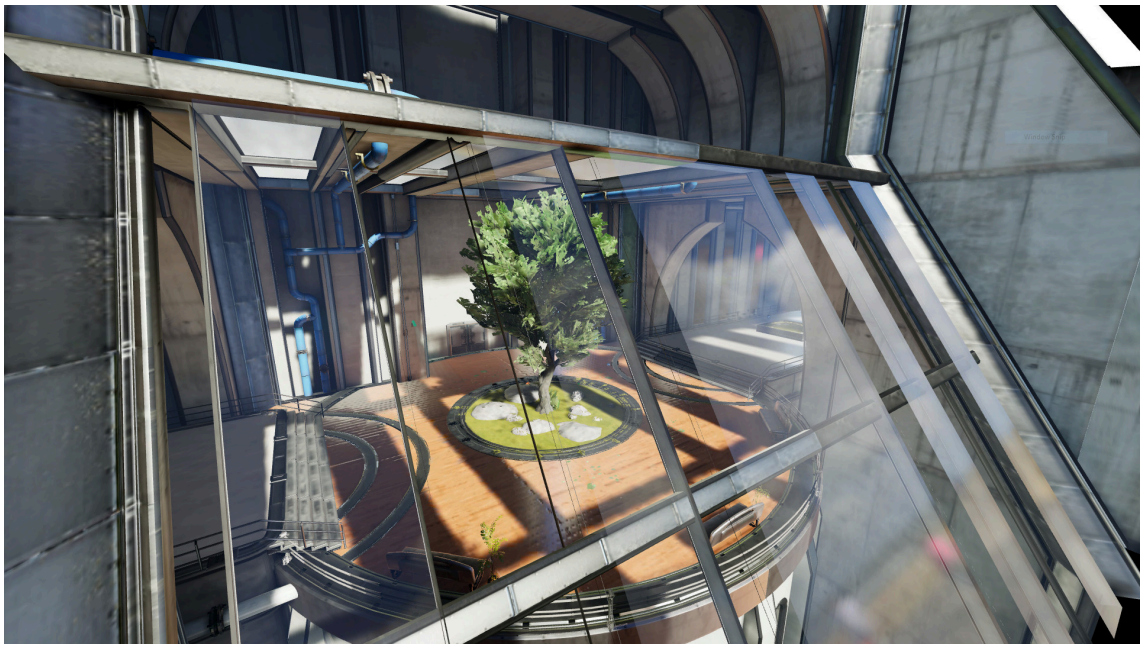


Figure 7.10: The shading atlas streaming client, in contrast to most warping methods, can render Epic's "Blueprint" test scene with transparent windows. Alpha channel information is streamed in additional blocks which are shared by up to three transparent patches to efficiently use the three color channels. Image courtesy of Wolfgang Tatzgern.

geometry to the client. The transparent geometry must be marked by the server, and rendered by the client after completing the rendering of the opaque geometry, in back to front order, using alpha blending. This requires a second draw call to be executed by the client, but otherwise, the client's rendering pipeline is not significantly complicated. This makes the approach compatible with mobile GPU capabilities, since no costly fragment shaders need to be run.

Only the meta-information must be augmented to distinguish transparent geometry; the shading information of transparent and opaque geometry can be stored together in the atlas in a uniform way. Alpha values that are uniform per triangle can be stored in the triangle meta-information; per-pixel alpha values would have to be stored by extending the atlas from RGB to RGBA channels. If this is undesirable (for example, because hardware MPEG encoders do not support RGBA formats), a 1-bit alpha value can be expressed in a regular RGB atlas by chroma keying. Alternatively, the alpha channel can be stored in a separate block in the atlas. In order to not waste two color channels, up to three patches can share one block for their alpha channels. The author's research group has implemented this technique in the Unreal Engine, as shown in Figure 7.10.

7.3.3 Postprocessing effects

Many game engines use postprocessing effects after the main rendering pass, which operate on color, depth and other rendering results stored in a G-buffer. Technically, a postprocessing stage can be added to the shading stage on the server or to the display stage of the client. However, adding general-purpose postprocessing to the client would imply shipping a G-buffer to the client (which would be too costly in terms of network bandwidth) and running proprietary shaders on the client (which would be too costly on a mobile GPU and violate the "thin client" idea).

Therefore, we believe it is more beneficial to add postprocessing to the shading stage of the server. One caveat is that view-dependent postprocessing effects are subject to the same restrictions as view-dependent forward rendering effects. They could be handled with variable shading rates, as discussed in Section 7.3.1. For shipping postprocessing effects to the client, they can be classified according to if/how they interpret the scene geometry:

Effects that modify surfaces (e.g., depth of field approximation or ambient occlusion) can be baked into the shading atlas. For example, the depth of field effect could be roughly approximated by varying the level selection based on the depth of the patch and the focal depth, or blocks could be blurred with a depth-aware filter.

Effects that are purely screen-space aligned can be rendered into a transparent billboard, aligned with the near clipping plane. The billboard is handled in the same way as the transparent geometry described in Section 7.3.2. This would, for example, work for lens flare, etc. For best results, the 2D area affected by the screen-space effects should be subdivided into a quad or triangle mesh, such that moderate block sizes can be assigned to the individual pieces of the screen-space effect.

Effects located in free-space (e.g., particle effects) can be rendered onto strategically placed billboards as well. The billboard is best placed at the average depth of the corresponding 3D object. If the client renders the billboard with depth test enabled, occlusion between the effect pixels and the scene can be approximately resolved, even when the camera is moved during frame rate upsampling. The author's research group has implemented this technique in the Unreal Engine, as shown in Figure 7.11.

As a proof of concept, screen-space ambient occlusion (SSAO) rendering [71] was added to SAS. Our SSAO implementation uses multiple depth layers [11] via depth peeling on the server to acquire correct results also for occluded triangles. The result is baked into the shading atlas for every polygon in the PVS, including those that only become visible in the future. Figure 7.12 shows the result rendered at the client from the shading atlas with SSAO enabled and disabled. For better demonstration of the effect, only simple Phong shading with color textures was used.

In a similar way, HDR can be added to the server rendering. A post-perspective HDR G-buffer can be replaced by an object-space HDR atlas. Tone mapping on the server converts the HDR atlas to a standard RGB atlas, which is shipped to the client. Like

Figure 7.11: The shading atlas streaming server renders particle effects of Epis's "Epic Zen Garden" scene as transparent billboard which are streamed to the client. The client simply renders the transparent billboards like any other transparent geometry. Image courtesy of Wolfgang Tatzgern.

with all other shading effects, the client does not have to be concerned with how the tone-mapped result was created.

7.3.4 Geometric scene complexity

A fundamental assumption of SAS is that the client is able to handle the geometric complexity of the visible part of the scene. The size of the visible scene will generally be much smaller than the size of the total scene, but the visible part can still exceed the capabilities of a mobile GPU. We also observe that our multi-sampled PVS puts a high geometry processing load on the server.

However, if either client or server suffer from too large scenes, the geometric complexity can be reduced with standard methods: In the simplest case, a conventional geometric level of detail (LOD) system uses a fixed set of simplifications and chooses to display one of them based on the distance from the viewer. Removed details after simplification can optionally be baked into normal maps.

Geometric LOD is also beneficial to suppress subpixel-sized triangles, which produce geometric aliasing and can be troublesome for PVS estimation. On the one hand, the LOD system chooses a level where triangles in screen-space are large enough to avoid geometric aliasing and reduce the geometric load on the system, especially on the client. On the other hand, the screen-space triangle size should not exceed the size of a superblock, so that the

Figure 7.12: To demonstrate how postprocessing effects can be integrated into SAS, we added screen-space ambient occlusion (SSAO) to the shading stage. The screenshots show Phong shaded images without (left) and with (right) SSAO.

triangle can be shaded at a high enough resolution. Therefore, we favor a combination of simplification and tessellation of overly large triangles to effectively place a band-limit on triangle sizes, such that they fit into the available block sizes.

7.3.5 Dynamic geometry

Our current system supports dynamic geometry (rigid and non-rigid animations), as can be seen in Figure 7.13, a "water scene" where the water is animated within the vertex shader. Non-rigid animation or tessellation shaders do not require special handling; changed vertices are transmitted in the same way as newly visible vertices. However, dynamic geometry requires more network bandwidth. Since skeletal animation is the most common non-rigid animation, we can save bandwidth by transferring the skeletal structure to the client and streaming the animation parameters instead of vertex updates. The client then animates the skeleton and extrapolates the animation from past movements, if future animation data is missing, as shown in Figure 7.14.

Static scenes typically create a bitrate peak only in the first frame, when many vertices and triangles need to be transmitted at once. This peak can be hidden by delaying the display start of the client until the second frame. In contrast, scenes with dynamic geometry may have a continuous, non-negligible bandwidth requirement for vertex updates. To keep the client simple, we prefer not having to run any shaders on it that are proprietary to the game, such as vertex animation shaders.

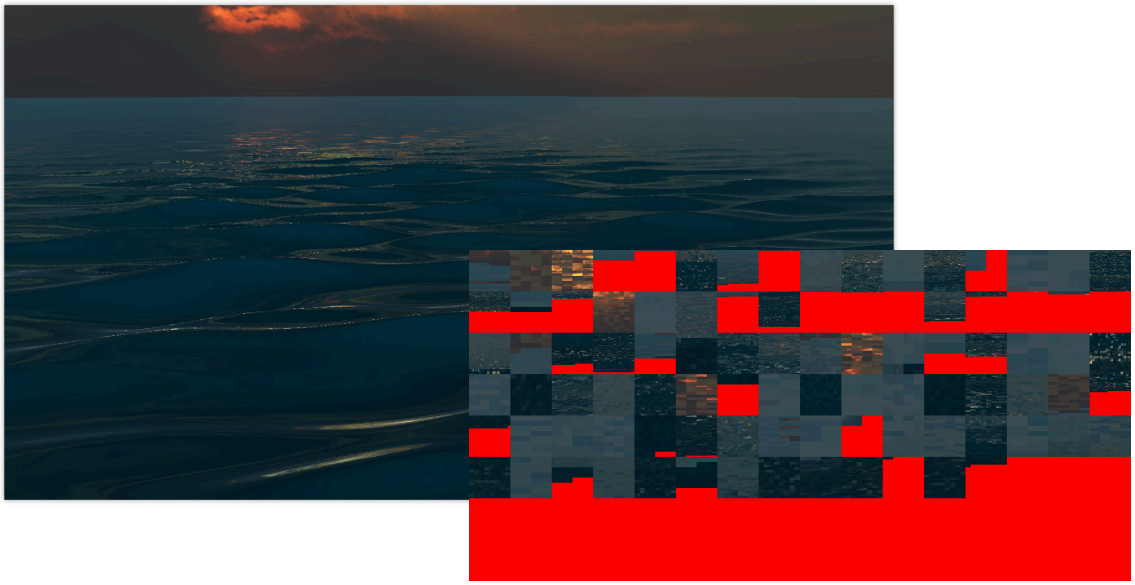


Figure 7.13: The "water scene" contains an ocean surface with animated waves. The waves are animated in the vertex shader depending on the current time. Our shading atlas streaming system supports this dynamic geometry by transmitting the updated vertex positions to the client.

Figure 7.14: Skeletal animation can be streamed to the client, which then applies the animation itself. Missing animation information is extrapolated from past movements. The image sequence shows the movement of the character, while the shadow remains stationary between shading atlas updates. Image courtesy of Florian Komposch.

Instead, we have added geometry compression as an experimental extension of our system. First, vertices are uniformly quantized; above a certain number (> 30) of visible vertices, we compress the vertices using an octree encoding [16] to about 15 bits per 3D point. We also apply simple lossless compression (difference encoding per attribute followed by exponential Golomb coding) to the triangle and texture coordinate messages, as these

Table 7.1: Detailed compression rates achieved on average for selected message types. The overall compression rate is about 3×.

Message Type	Robot Lab	Viking Village
Vertex coordinates	6.68×	5.95×
Vertex id	1.78×	1.70×
Triangles	2.22×	2.20×
Texture coordinates	3.19×	3.09×

messages tend to follow the object scene structure and thus exhibit a lot of coherence. These measures typically reduce the overall bandwidth requirements for dynamic geometry by >3× over a naive uncompressed representation (Table 7.1). Moreover, compression rates tend to get better with larger amounts of dynamic geometry, since more entropy can be detected.

7.3.6 Game engine integration

In general, integrating SAS into a game engine is similar in effort and difficulty to integrating a deferred rendering pipeline or another type of advanced graphics pipeline. Because of the high demands of VR systems, contemporary game engines, such as Unreal Engine [1], give developers a choice of forward rendering with less post-processing to reduce the latency compared to their standard pipeline. Integrating the SAS pipeline into a game engine with reduced post-processing is considerably less challenging.

[1] https://docs.unrealengine.com/en-us/Engine/Performance/ForwardRenderer

Chapter 8

Conclusion

Contents

In this chapter we will summarize the research around the shading atlas presented in this thesis. We will refer back to the hypotheses in Chapter 1 and discuss the results of our investigations with respect to each of them. Finally, we will give an outlook for possible directions for future research with the shading atlas and its benefits.

8.1 Summary

The shading atlas (Chapter 3) is a texture atlas for storing shading information. Compared to other object space shading methods, the shading atlas requires neither custom GPU extensions nor generating unique texture coordinates. Moreover, it uses only a moderate amount of memory, with a compact memory layout and fast, parallel memory management running directly on the GPU. The level selection process ensures that the available space in the atlas will be optimally used without running out of memory, balancing between the rendering quality with the available memory. Rendering with the shading atlas runs in four main stages that can be decoupled an run at different frequencies, depending on the application scenario, proving ample flexibility. For example, this decoupling is appealing for applications, where it has the potential to replace existing "split rendering" pipelines that perform frame rate upsampling in order to achieve sustained frame rates. While the design principles behind the shading atlas originated from its application for streaming, it is universally applicable, solving issues of other texture-space shading methods such as high memory usage and constraints on texture coordinates.

Shading storage in the shading atlas is temporally coherent, allowing easy spatial and temporal reuse of shading information. If more than one or two views must be generated, such as on autostereoscopic TV screens, or for future VR headsets with extreme resolutions, such as 8K displays, the doubled resolution has a huge performance impact when rendering stereo images. In contrast, the shading atlas can be used with a minimal increase in computational cost when doubling the resolution from mono to stereo rendering. Furthermore, the shading atlas effectively decouples display from image generation for short periods of time, a property that we believe will become increasingly important.

To demonstrate the capability of spatial shading reuse with the shading atlas with three application scenarios. Spatial upsampling with the shading atlas results in smooth edges, since the final display stage can run in full resolution without any impact on the shading workload. The shading workload depends mostly on the size of the atlas, if the bias is used for efficient memory management. The bias automatically decreases the resolution and thus the workload due to memory pressure, or it can increase the resolution and thus the quality of the output. In the second application, stereo rendering, the shading atlas does not suffer from disocclusion artifacts, since it can determine the visible geometry for both views before shading and then efficiently reuses the shading for both eyes. Finally, foveated rendering with the shading atlas requires no additional GPU extensions to benefit from a lower spatial resolution in the periphery. The resolution of geometry can simply be reduced depending on the eccentricity during the level selection.

Another objective in this work is to investigate how shading reuse is perceived and could benefit rendering, considering visibility, spatial sampling and temporal behavior of shading, separated and combined. To this end, we evaluate the perception of outdated shading using a rendering approach that separates shading from visibility and spatial sampling effects, finding that a shading difference of 3 % ($T = 8$ after 8-bit quantization) is not noticed by study participants. We find that, even in highly dynamic scenes, many shading samples stay valid for extended periods of time, when considered independently of visibility and spatial sampling. At the 3 % threshold, the theoretical reuse potential of 90 % of samples reaches 40 frames, i.e., 0.6 s. However, reusing all shading samples for a constant number of frames is already noticeable for a reuse of 2 to 4 frames. Considering only visibility, we confirm the results by Nehab et al. [75] and show that typically more than 90 % of surface samples stay visible between two consecutive frames. We find that there is a potential of typically more than 80 % shading reuse from frame to frame even in highly dynamic scenes at 60 Hz. The higher frame rates of VR increase this potential further.

With the extensive knowledge on how long shading can be reused, we answer the question how to utilize this potential in practice. To find the right time when shading is required, we analyze numerical and analytical first-order approximations of the temporal shading gradient. The analysis shows that we can efficiently approximate expensive gradient computations by estimating the temporal gradient using backward differences every time we compute shading. Furthermore, we show that we can efficiently remove late predictions—and thus visible artifacts—by spatially filtering the estimated temporal gra-

dient in image space. Accumulating spatial resampling errors limits the temporal reuse. Thus, we favor the texture-space caching of shading samples in the shading atlas. While fixed upsampling techniques lead to noticable artifacts even at low upsampling rates, we show that extrapolating shading differences works very well when combined with a simple image-space filter for capturing spatio-temporal effects.

We generalize our findings by introducing the TAS framework, which is a flexible, efficient method to integrate temporally adaptive shading reuse into arbitrary rendering systems, such as the reverse reprojection caching [75] and shading atlas rendering. Our sample implementations, TARC and TASA, show that we can significantly improve performance due to efficient shading reuse. In a user study, we show that TAS can be used to increase performance at an almost imperceptible qualitative difference. Correctly tuned to a color difference threshold of $T = 8$, study participants notice no temporal instabilities, such as flickering or ghosting. TAS reduces shader invocations up to 5× and increases overall render performance by up to 1.4× compared to Forward+ rendering, at a setting where users cannot distinguish quality differences (below 1 JND). TAS is straightforward to integrate into any rendering system. Moreover, it can be combined with spatially adaptive shading techniques, further increasing possible savings on shader invocations.

To our knowledge, we have presented the first work that demonstrates an object-space shading approach for remote rendering with low perceived latency (Chapter 7). The shading atlas rendering pipeline for streaming is decoupled between shading and final display, introducing a networking layer that encodes the geometry of the PVS together with the shading in the shading atlas and transmits it from a powerful server to a mobile client. The client processing is limited to decoding and a straightforward rendering pass of the PVS with simple texture mapping from the shading atlas. This setup utilizes the available graphics processing on the client without overburdening it, keeping its power consumption low. Since the simple rendering procedure on the client is exceptionally fast, it can hide the latency introduced by the networking layer at a high frame rate as it is required for VR. While high throughput and low latency are essential properties of streaming VR applications, other graphics streaming applications, such as networked games, architectural preview or scientific visualization, can benefit as well.

8.2 Discussion

In Chapter 1, we established a three objectives around the application of the shading atlas, specifically in streaming and shading reuse. We will now look back at these objectives and discuss what the results of our experiments tell us with respect to each of them.

O1 Find an efficient implementation – in terms of memory and performance – of object-space shading for real-time rendering on the GPU.

- The shading atlas provides a practical caching data structure for shading information of visible triangles. We show that, in the standalone case with constant frame rate upsampling, the caching mechanism beats forward rendering and deferred rendering at 2-3× upsampling without reducing image quality.
- Shading information can be tightly packed into the shading atlas at a resolution that corresponds with the screen-space resolution. Our experiments with a 4 MPx atlas show that memory fragmentation is low with the hierarchical allocation scheme in the atlas. On average about 60 % of the samples in the atlas are in use, while the remaining space is either available within partially allocated superblocks, as empty blocks allocated and stored on a block stack, or free.
- Memory management of the shading atlas can run in real-time on the GPU and handles high memory pressure gracefully. The experiments show that the per-frame overhead of managing the atlas memory on the GPU stays below an insignificant 0.25 ms. Atlas resets and the bias for level selection ensure that the memory does not run out even when memory pressure is high.

O2 Show how object-space shading provides an easy way for spatial reuse of shading information.

- Spatial upsampling with the shading atlas can easily be implemented due to the decoupling of the shading resolution from the output image resolution. The shading resolution typically depends on the size of the atlas instead. Thus, varying the size of the atlas to be smaller than the output resolution inherently leads to spatial shading reuse when the output image is computed. The global bias which is used in level selection takes care that the atlas does not run out of memory and the resolution of patches is chosen smaller than it would be. Without the bias, the level selection targets the output resolution, but this could also easily be adopted to target a different resolution instead.
- Stereo rendering is a prime use case for spatial shading reuse. The difficulty of image-space warping methods is that some spatial samples that would be required for stereo rendering are not available, since they are occluded or out of view in the view that is warped. The object-space representation of the shading atlas allows to shade all visible geometry within the atlas that is required to generate a stereo image pair.
- Foveated rendering is a form of variable spatial upsampling, since the display resolution is uniform. Using the shading atlas, foveated rendering can easily be implemented by adapting the number of samples used to shade each primitive depending on its distance from the focal point. Image-space solutions instead need a more complex solution or special GPU extensions, like variable rate shading, to achieve the same effect and performance gain.

O3 Provide a framework that, together with object-space shading, allows extended temporal reuse of shading results resulting in performance gains that can be used for higher quality rendering.

- Shading information can temporally be reused up to a certain threshold, before the user notices any artifacts. Our user study shows that after tone-mapping a color difference of 3 % between the current shading and the reused old shading is typically not noticed by users.
- Shading can be reused for extended periods of time, longer than what is currently state-of-the-art shading reuse, for most of the shading samples, while some need frequent or even constant recomputation. After 120 frames, even in highly dynamic scenes, 75 % of samples can be reused, independently of visibility. We also confirm results of Nehab et al. [75] that temporal coherence of visibility is around 90 % from frame to frame. We show that a practical implementation could reuse more than 80 % of shading results in most cases, if a perfect prediction could be made whether shading reuse is possible or reshading is necessary.
- A linear model for predicting temporal changes in combination with a spatial filter is enough to provide an estimate for when shading needs to be recomputed. We show that simple linear gradients can already predict most of the shading changes. An additional spatial filter increases the prediction accuracy, leaving only rare, discontinuous cases that cannot be captured with a linear prediction. The prediction is conservative, since unnecessarily shading samples that would not need updating is preferable, slightly decreasing the possible performance gain, is preferable over noticeable artifacts that decrease rendering quality.
- The shading atlas provides an ideal cache for shading information to reuse shading for extended periods of time, not suffering from the reprojection artifacts of screen-space methods. We find that the TAS framework is applicable to screen-space reprojection methods. However, the performance gain is strictly limited, mostly due to the additional shading deterioration caused by repeated resampling that accumulates spatial sampling errors. Therefore, a texture-space representation that stays coherent over time is an ideal caching mechanism for extended shading reuse.

O4 Show that object-space shading can be used efficiently for streaming and hides latency better than screen-space based approaches.

- The shading atlas can handle disocclusions better than screen-space warping approaches, since it can store shading information of occluded objects. A screen-space warping approach typically only has access to a single screen-space image and consequently does not stand a chance in handling occlusions correctly. Some methods utilize multiple screen-space images that – given necessary views are available – can

deal with disocclusions. However, this comes at the cost of redundant computation, storage and possibly transmission of data. Thus, any object-space method will likely provide a better solution to the problem of disocclusions of screen-space warping.

- The shading atlas is ideal for transmission using hardware-accelerated MPEG encoding. It was specifically designed for streaming, using temporally coherent, rectangular blocks that are ideal for MPEG encoding. Our reference implementation streams from a server PC which encodes the atlas on the GPU to a Qualcomm Snapdragon™ device that also decodes in a special hardware unit.

8.3 Outlook

To finish this thesis, we will discuss possible future directions of research with the shading atlas. Some of these directions have already been mentioned within the limitations sections of the previous chapters. We will discuss some more novel directions as well as provide more detail on some of the previously mentioned ones.

The sample distribution within the atlas depends on multiple factors. Depending on the number of triangles and which actual screen space size each of the triangles has, they might be either over- or undersampled in comparison to the other triangles, since the level selection process depends mostly on the total area of the triangles. Furthermore, depending on the size and variation in depth of the triangles' vertices, perspective distortion may further lead to a discrepancy between screen-space and atlas-space sampling, though in preliminary experiments show that this effect is negligible. The distortion to form rectangles within the atlas also contributes to a different sample distribution, causing the highest difference in sampling. Considering the borders between patches that would be shared in a screen-space representation and the additional borders necessary for bilinear filtering cause the shading atlas to require a higher sample number. Finally, power-of-two block sizes are limiting and can cause popping when the level of a patch changes, as noticed by some participants of our TASA studies. We have already implemented a hysteresis on the level selection to avoid frequent popping. Filtering between two blocks of different level when the level of a block changes may alleviate popping further, but does not solve the other issues. Consequently, one opportunity for further research is the investigation of the sample distribution and other packing techniques of multiple triangles into the atlas without unnecessarily wasting space and a better sample distribution.

Some future research directions revolve around the processing of geometry for the shading atlas. As a result of the sampling based visibility determination, the algorithm can suffer from aliasing artifacts. In particular, it can miss tiny geometry which is not captured by the sampling resolution. Triangles sizes would need to stay within reasonable bounds. This band-limiting of triangle sizes can be achieved by appropriate tessellation of triangles that are too big and level-of-detail or mesh simplification for triangles that are too small. However, since the visibility determination is an exchangeable part of the

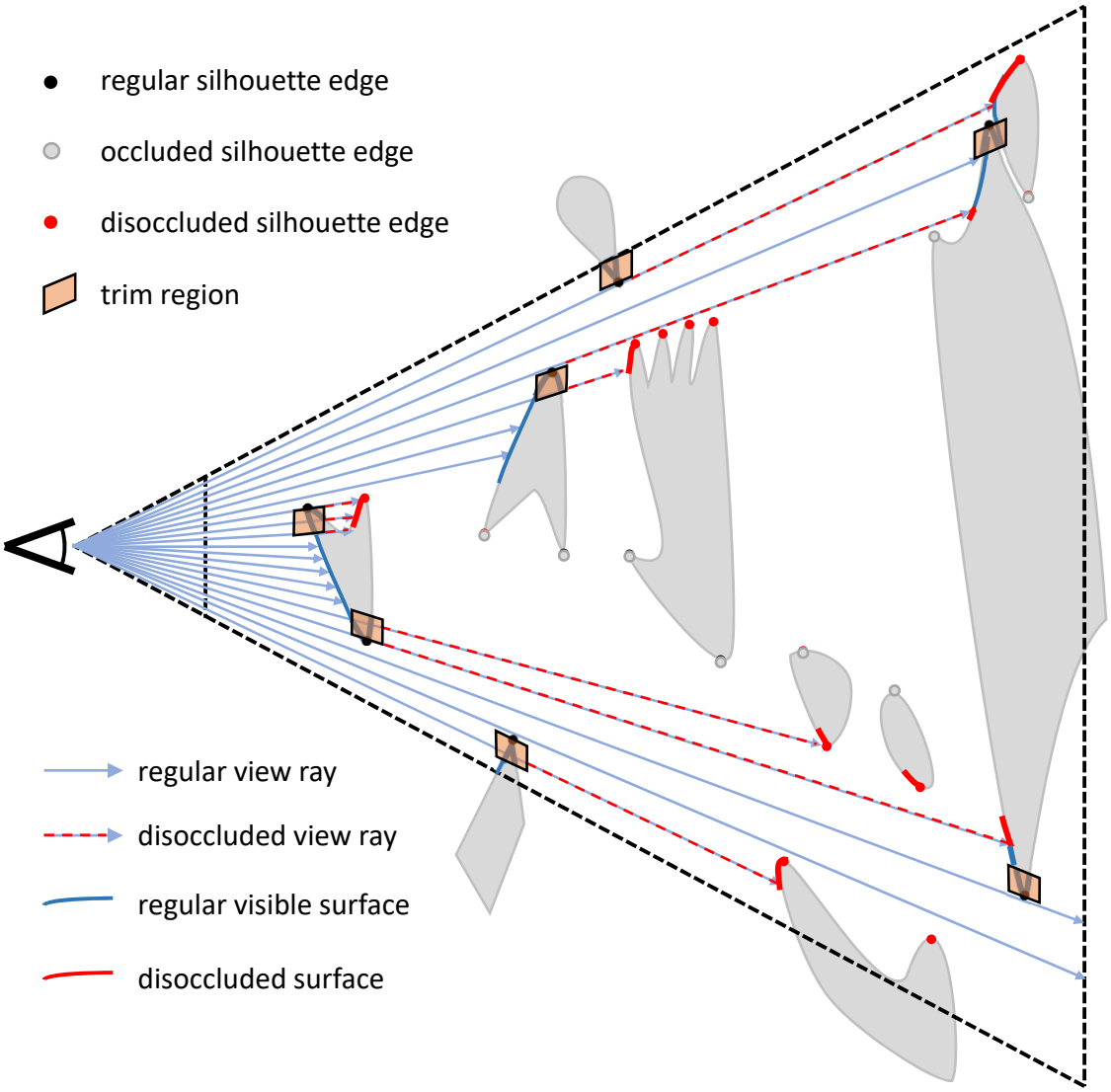


Figure 8.1: Through erosion of object silhouettes, which are marked as trim regions, it is possible to extend a visible set with surfaces that are most likely to become visible following slight camera and object movements. Image courtesy of Philip Voglreiter.

pipeline, it can easily be replaced by other algorithms. Ongoing research investigates a trim-region based approach, where the visible set is extended by areas that are hidden by visual edges of objects, i.e., the areas where disocclusions usually appear, as shown in Figure 8.1. As another example, the latest generation of graphics hardware supports meshlets, a small part of a bigger mesh, consisting of a few dozen triangles. A visibility algorithm based on these meshlets can probably work in real-time without the requirement for extensive sampling. Furthermore, meshlets could provide a good granularity for a new packing scheme within the atlas.

The shading atlas may also support rendering techniques that require higher dimensional sampling. For example, the typical application scenario for object-space rendering methods is stochastic rasterization. Stochastic rasterization can utilize the simplified sampling of the shading atlas to render realistic motion blur and depth of field effects. Specifically, the level selection can be based on the highest needed resolution of each patch and, if smaller resolutions are required, they can be downsampled for the shaded higher resolution. Moreover, the shading atlas can be used in light field rendering as an effective caching mechanism to store and sample shading results.

With its ability to efficiently cache shading results, the shading atlas, especially in the combination with TAS, can support more realistic shading computations. Specifically, real-time ray-tracing, which has been recently popularized by hardware support from GPU manufacturers, can utilize the shading atlas. Current path tracing implementations have to reuse samples from previous frames in order to run in real-time and typically require warping of the previous samples. The shading atlas provides an alternative space to store the samples, without having to reproject them. Furthermore, it can also store off-screen samples, which are also required in global illumination computation and typically cause artifacts in screen-space approximations such as screen-space reflections. As this example shows, many of the currently used pre- and post-processing techniques approximate global shading effects, such as shadows and reflections. If, by virtue of efficient shading reuse within the shading atlas, more time can be spent on the samples that actually require shading, this benefits the trend for moving global effect computation from post-processing to ray-tracing.

TAS reduces the shading load in areas where no shading change is required, freeing resources for more advanced shading of the remaining scene. Since, in the worst case (discontinuous view change), the whole scene has to be shaded, this can lead to a higher variability in frame rate: Only some frames can be accelerated; others remain at the baseline speed. In our measurements, the absolute frame time variability was unchanged in comparison to the baseline, while the mean frame time was reduced. The frame time variability depends mostly on the complexity of the current view (e.g., close-up view vs. wide overview). Lower frame rate variability could be obtained by using TAS as an oracle for a scheduling technique, which uses the predicted shading differences as priority, instead of making a decision based on a fixed threshold. This works especially well with our observation that fast moving scenes make it harder to detect errors in comparison to a still image.

As a technique focused on temporal reuse of shading, TAS is orthogonal to spatially adaptive techniques. Thus, it can be easily combined with spatial reuse of sampling, such as variable rate shading, foveation and checkerboard rendering. Further research combining these techniques may also reveal further insight into how to optimize the spatial filter that TAS uses. Our hope is that the future will bring more physically correct shading and fewer approximations that require pre-processing steps, like rendering shadow maps, or post-processing, like screen-space effects in deferred rendering.

Overall, we have shown in this thesis the huge potential of texture-space storing of shading in the shading atlas for spatial shading reuse, extended temporal reuse of shading information and streaming VR. Texture-space shading and its possibilities are novel and thus not yet widespread in current rendering applications. However, GPU manufacturers are already noticing their potential and support such techniques with corresponding extensions. The possibility to focusing shading computations on areas where they are actually needed is of utmost importance for realistic rendering. Latency and the hiding thereof will always be a problem of streaming rendering approaches and, as we have shown, screen-space solutions simply cannot handle disocclusions caused by latency, thus requiring a way to stream shading information of all possibly visible surfaces. We are confident that texture-space shading and temporally adaptive shading reuse will become more important as higher quality shading and real-time global illumination gain traction.

Bibliography

[1] Abrash, M. (2000). Ramblings in realtime. *Dr. Dobb's Sourcebook*. (page 12)

[2] Adelson, E. H. (2000). Lightness perception and lightness illusions. In Gazzaniga, M. S., editor, *The new cognitive neurosciences*, chapter 24, pages 339–351. MIT ACM ACM Press. (page 99)

[3] Akenine-Möller, T. and Aila, T. (2005). Conservative and tiled rasterization using a modified triangle set-up. *Journal of Graphics Tools*, 10(3):1–8. (page 31)

[4] Akenine-Möller, T., Haines, E., and Hoffman, N. (2018). *Real-time Rendering*. Taylor & Francis Ltd. (page 26)

[5] Andersson, M., Hasselgren, J., Toth, R., and Akenine-Möiler, T. (2014). Adaptive texture space shading for stochastic rendering. *Computer Graphics Forum*, 33(2):341–350. (page 12, 19)

[6] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. (2020). Flip: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2). (page 22)

[7] Anwar, S. and Barnes, N. (2020). Densely residual laplacian super-resolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1. (page 19)

[8] Baker, D. (2016). Object Space Lighting. Talk at Game Developers Conference. (page 12, 13, 88, 94, 106)

[9] Bao, P. and Gourlay, D. (2004). Remote walkthrough over mobile networks using 3-d image warping and streaming. *IEE Proceedings - Vision, Image and Signal Processing*, 151(4):329–336. (page 23)

[10] Barman, N., Schmidt, S., Zadtootaghaj, S., Martini, M. G., and Möller, S. (2018). An evaluation of video quality assessment metrics for passive gaming video streaming. In *Proceedings of the 23rd Packet Video Workshop on ZZZ - PV '18*. ACM Press. (page 22)

[11] Bavoil, L. and Sainz, M. (2009). Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, page 45. ACM. (page 118)

[12] Blackwell, H. R. (1972). Luminance difference thresholds. In *Visual Psychophysics*, pages 78–101. Springer Berlin Heidelberg. (page 62, 99)

[13] Blinn, J. (1989). Return of the jaggy. *IEEE Computer Graphics and Applications*, 9(2):82–89. (page 19)

[14] Blinn, J. (1998). Jim blinn's corner: Dixty pixels. pages 179–190. (page 19, 47)

[15] Boos, K., Chu, D., and Cuervo, E. (2016). Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proc. MobiSys*, pages 291–304. (page 22)

[16] Botsch, M., Wiratanaya, A., and Kobbelt, L. (2002). Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 53–64. (page 121)

[17] Bowles, H., Mitchell, K., Sumner, R. W., Moore, J., and Gross, M. (2012). Iterative image warping. *Computer Graphics Forum*, 31(2pt1):237–246. (page 14, 17)

[18] Buehler, C., Bosse, M., McMillan, L., Gortler, S., and Cohen, M. (2001). Unstructured lumigraph rendering. In *Proceedings SIGGRAPH*, pages 425–432. (page 18)

[19] Burns, C. A., Fatahalian, K., and Mark, W. R. (2010). A Lazy Object-space Shading Architecture with Decoupled Sampling. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 19–28, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. (page 12, 19)

[20] Carr, N. A. and Hart, J. C. (2002). Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131. (page 13, 103)

[21] Cebenoyan, C. (2014). Real virtual texturing – taking advantage of directx11.2 tiled resources. Game Developer Conference. (page 13)

[22] Chadjas, M., Eisenacher, C., Stamminger, M., and Lefebvre, S. (2010). Virtual Texture Mapping 101. (page 13)

[23] Chang, C.-F. and Ger, S.-H. (2002). *Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering*, pages 1105–1111. Springer Berlin Heidelberg, Berlin, Heidelberg. (page 22)

[24] Chen, K. (2015). Adaptive Virtual Texture Rendering in Far Cry 4. Talk at Game Developers Conference. (page 13)

[25] Chen, S. E. and Williams, L. (1993). View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*. ACM Press. (page 14)

[26] Clarberg, P., Toth, R., Hasselgren, J., Nilsson, J., and Akenine-Möller, T. (2014). AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Transactions on Graphics*, 33(4):1–12. (page 12, 19)

[27] Clarberg, P., Toth, R., and Munkberg, J. (2013). A sort-based deferred shading architecture for decoupled sampling. *ACM Transactions on Graphics*, 32(4):1. (page 12, 19)

[28] Cohen-Or, D., Mann, Y., and Fleishman, S. (1999). Deep compression for streaming texture intensive animations. In *Proceedings SIGGRAPH*, pages 261–267. (page 22)

[29] Collet, A., Chuang, M., Sweeney, P., Gillett, D., Evseev, D., Calabrese, D., Hoppe, H., Kirk, A., and Sullivan, S. (2015). High-quality streamable free-viewpoint video. *ACM Trans. Graph.*, 34(4):69:1–69:13. (page 30, 107)

[30] Cook, R. L., Carpenter, L., and Catmull, E. (1987). The Reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics*, 21(4):95–102. (page 11, 88)

[31] Crassin, C., Luebke, D., Mara, M., McGuire, M., Oster, B., Shirley, P., Sloan, P.-P., and Wyman, C. (2015). CloudLight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques (JCGT)*, 4(4):1–27. (page 23)

[32] Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., and Musuvathi, M. (2015). Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 121–135, New York, NY, USA. ACM. (page 23)

[33] Dayal, A., Woolley, C., Watson, B., and Luebke, D. (2005). Adaptive frameless rendering. In *ACM SIGGRAPH 2005 Courses*. Association for Computing Machinery. (page 20)

[34] den Branden Lambrecht, C. J. V. and Verscheure, O. (1996). Perceptual quality measure using a spatiotemporal model of the human visual system. In Bhaskaran, V., Sijstermans, F., and Panchanathan, S., editors, *Digital Video Compression: Algorithms and Technologies 1996*. SPIE. (page 22)

[35] Didyk, P., Eisemann, E., Ritschel, T., Myszkowski, K., and Seidel, H.-P. (2010a). Perceptually-motivated real-time temporal upsampling of 3d content for high-refresh-rate displays. *Computer Graphics Forum*, 29(2):713–722. (page 20)

[36] Didyk, P., Ritschel, T., Eisemann, E., Myszkowski, K., and Seidel, H.-P. (2010b). Adaptive Image-space Stereo View Synthesis. In Koch, R., Kolb, A., and Rezk-Salama, C., editors, *Vision, Modeling, and Visualization (2010)*. The Eurographics Association. (page 14)

[37] Dong, C., Loy, C. C., He, K., and Tang, X. (2016). Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307. (page 19)

[38] Durand, F., Holzschuch, N., Soler, C., Chan, E., and Sillion, F. X. (2005). A frequency analysis of light transport. *ACM Transactions on Graphics*, 24(3):1115. (page 21, 73)

[39] El Mansouri, J. (2016). Rendering Rainbox Six Siege. Talk at Game Developers Conference. (page 47)

[40] Gribel, C. J., Barringer, R., and Akenine-Möller, T. (2011). High-quality spatio-temporal rendering using semi-analytical visibility. *ACM Transactions on Graphics*, 30(4):1. (page 20)

[41] Guenter, B., Finch, M., Drucker, S., Tan, D., and Snyder, J. (2012). Foveated 3d graphics. *ACM Trans. Graph.*, 31(6). (page 19, 54, 55)

[42] He, Y., Foley, T., and Fatahalian, K. (2016). A system for rapid exploration of shader optimization choices. *ACM Transactions on Graphics*, 35(4):1–12. (page 21)

[43] He, Y., Gu, Y., and Fatahalian, K. (2014). Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Transactions on Graphics*, 33(4):1–12. (page 19)

[44] Heckbert, P. S. (1986). Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67. (page 18)

[45] Herzog, R., Eisemann, E., Myszkowski, K., and Seidel, H.-P. (2010). Spatio-temporal upsampling on the gpu. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM Press. (page 20)

[46] Hillesland, K. E. and Yang, J. C. (2016). Texel Shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*, EG '16, pages 73–76, Goslar Germany, Germany. Eurographics Association. (page 12, 27, 31, 66, 88, 94)

[47] Hou, H. and Andrews, H. (1978). Cubic splines for image interpolation and digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(6):508–517. (page 19)

[48] Iglesias-Guitian, J. A., Moon, B., Koniaris, C., Smolikowski, E., and Mitchell, K. (2016). Pixel history linear models for real-time temporal filtering. *Computer Graphics Forum*, 35(7):363–372. (page 20)

[49] Kaplanyan, A. S., Sochenov, A., Leimkühler, T., Okunev, M., Goodall, T., and Rufo, G. (2019). DeepFovea. *ACM Transactions on Graphics*, 38(6):1–13. (page 19)

[50] Karis, B. (2013). Real Shading in Unreal Engine 4. SIGGRAPH Course on Advances in Real-Time Rendering in Games. (page 12)

[51] Karis, B. (2014). High Quality Temporal Supersampling. SIGGRAPH Course on Advances in Real-Time Rendering in Games. (page 20)

[52] Kato, H., Ushiku, Y., and Harada, T. (2018). Neural 3d mesh renderer. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE. (page 21)

[53] Kerbl, B., Kenzel, M., Mueller, J. H., Schmalstieg, D., and Steinberger, M. (2018). The broker queue. In *Proceedings of the 2018 International Conference on Supercomputing*. ACM. (page 33)

[54] Kiran Adhikarla, V., Vinkler, M., Sumin, D., Mantiuk, R. K., Myszkowski, K., Seidel, H.-P., and Didyk, P. (2017). Towards a quality metric for dense light fields. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 58–67. (page 60)

[55] Kraus, M. and Ertl, T. (2002). Adaptive texture maps. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 7–15. (page 13)

[56] Lee, K., Chu, D., Cuervo, E., Kopf, J., Degtyarev, Y., Grizan, S., Wolman, A., and Flinn, J. (2015). Outatime. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '15*. ACM Press. (page 14, 16, 23, 51, 58, 103)

[57] Lefebvre, S., Darbon, J., and Neyret, F. (2004). Unified texture management for arbitrary meshes. INRIA Research report 5210. (page 13)

[58] Levoy, M. (1995). Polygon-assisted jpeg and mpeg compression of synthetic images. In *Proceedings SIGGRAPH*, pages 21–28. (page 23)

[59] Li, T.-M., Aittala, M., Durand, F., and Lehtinen, J. (2018). Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics*, 37(6):1–11. (page 21)

[60] Liktor, G. and Dachsbacher, C. (2012). Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '12*. ACM Press. (page 12, 19)

[61] Liu, G., Ceylan, D., Yumer, E., Yang, J., and Lien, J.-M. (2017). Material editing using a physically based rendering network. In *IEEE International Conference on Computer Vision (ICCV)*. IEEE. (page 21)

[62] Lochmann, G., Reinert, B., Ritschel, T., Müller, S., and Seidel, H.-P. (2014). Real-time Reflective and Refractive Novel-view Synthesis. In Bender, J., Kuijper, A., von Landesberger, T., Theisel, H., and Urban, P., editors, *VMV 2014: Vision, Modeling & Visualization*, pages 9–16. The Eurographics Association. (page 18, 116)

[63] Loper, M. M. and Black, M. J. (2014). OpenDR: An approximate differentiable renderer. In *Computer Vision – ECCV*, pages 154–169. Springer International Publishing. (page 21)

[64] Luo, M. R., Cui, G., and Rigg, B. (2001). The development of the CIE 2000 colour-difference formula: CIEDE2000. *Color Research & Application*, 26(5):340–350. (page 99)

[65] Maeland, E. (1988). On the comparison of interpolation methods. *IEEE Transactions on Medical Imaging*, 7(3):213–217. (page 19)

[66] Mann, Y. and Cohen-Or, D. (1997). Selective pixel transmission for navigating in remote virtual environments. *Computer Graphics Forum*, 16:C201–C206. (page 23)

[67] Mantiuk, R., Kim, K. J., Rempel, A. G., and Heidrich, W. (2011). HDR-VDP-2. *ACM Transactions on Graphics*, 30(4):1–14. (page 21)

[68] Mantiuk, R. K., Tomaszewska, A., and Mantiuk, R. (2012). Comparison of four subjective methods for image quality assessment. In *Computer graphics forum*, volume 31, pages 2478–2491. Wiley Online Library. (page 60, 61)

[69] Mark, W. R., McMillan, L., and Bishop, G. (1997). Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics - SI3D '97*. ACM Press. (page 14, 16, 51, 58, 110)

[70] McAnlis, C. (2009). Halo wars: The terrain of next-gen. Talk at Game Developers Conference. (page 13)

[71] Mittring, M. (2007). Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM. (page 118)

[72] Mittring, M. (2008). Advanced Virtual Texture Topics. In *Advances in Real-Time Rendering in 3D Graphics and Games Course SIGGRAPH 2008*, pages 23–51. (page 31, 44)

[73] Mueller, J. H., Voglreiter, P., Dokter, M., Neff, T., Makar, M., Steinberger, M., and Schmalstieg, D. (2018). Shading Atlas Streaming. *ACM Transactions on Graphics*, 37(6). (page 85, 94)

[74] Munkberg, J., Hasselgren, J., Clarberg, P., Andersson, M., and Akenine-Möller, T. (2016). Texture space caching and reconstruction for ray tracing. *ACM Transactions on Graphics*, 35(6):1–13. (page 21)

[75] Nehab, D., Sander, P. V., Lawrence, J., Tatarchuk, N., and Isidoro, J. R. (2007). Accelerating Real-time Shading with Reverse Reprojection Caching. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 25–35, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. (page 19, 20, 64, 66, 69, 81, 85, 87, 124, 125, 127)

[76] Noimark, Y. and Cohen-Or, D. (2003). Streaming scenes to mpeg-4 video-enabled devices. *IEEE Computer Graphics and Applications*, 23:58–64. (page 22)

[77] NVIDIA (2018). Nvidia turing gpu architecture whitepaper. Visited on January 05, 2020. (page 19)

[78] NVIDIA (2020). Nvidia ampere gpu architecture whitepaper. Visited on October 16, 2020. (page 19, 48)

[79] Pajak, D., Herzog, R., Eisemann, E., Myszkowski, K., and Seidel, H.-P. (2011). Scalable remote rendering with depth and motion-flow augmented streaming. *Computer Graphics Forum*, 30(2):415–424. (page 22)

[80] Patney, A., Salvi, M., Kim, J., Kaplanyan, A., Wyman, C., Benty, N., Luebke, D., and Lefohn, A. (2016). Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics*, 35(6):1–12. (page 19)

[81] Pece, F., Kautz, J., and Weyrich, T. (2011). Adapting Standard Video Codecs for Depth Streaming. In Coquillart, S., Steed, A., and Welch, G., editors, *Joint Virtual Reality Conference of EGVE - EuroVR*. The Eurographics Association. (page 22)

[82] Pinson, M. and Wolf, S. (2004). A new standardized method for objectively measuring video quality. *IEEE Transactions on Broadcasting*, 50(3):312–322. (page 22)

[83] Ragan-Kelley, J., Lehtinen, J., Chen, J., Doggett, M., and Durand, F. (2011). Decoupled sampling for graphics pipelines. *ACM Transactions on Graphics*, 30(3):1–17. (page 12, 19)

[84] Ramamoorthi, R., Mahajan, D., and Belhumeur, P. (2007). A first-order analysis of lighting, shading, and shadows. *ACM Transactions on Graphics*, 26(1):2–es. (page 21, 73)

[85] Reinert, B., Kopf, J., Ritschel, T., Cuervo, E., Chu, D., and Seidel, H.-P. (2016). Proxy-guided image-based rendering for mobile devices. *Computer Graphics Forum*, 35(7):353–362. (page 18, 23)

[86] Scherzer, D., Nguyen, C. H., Ritschel, T., and Seidel, H.-P. (2012). Pre-convolved radiance caching. *Computer Graphics Forum*, 31(4):1391–1397. (page 12)

[87] Schied, C., Peters, C., and Dachsbacher, C. (2018). Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–16. (page 21)

[88] Schlick, C. (1994). An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246. (page 60)

[89] Seshadrinathan, K. and Bovik, A. (2010). Motion tuned spatio-temporal quality assessment of natural videos. *IEEE Transactions on Image Processing*, 19(2):335–350. (page 22)

[90] Shade, J., Gortler, S., He, L.-w., and Szeliski, R. (1998). Layered depth images. In *Proceedings SIGGRAPH*, pages 231–242. (page 14)

[91] Sheng, B., Meng, W.-L., Sun, H.-Q., and Wu, E.-H. (2011). MCGIM-based model streaming for realtime progressive rendering. *Journal of Computer Science and Technology*, 26(1):166–175. (page 22)

[92] Shi, S. and Hsu, C.-H. (2015). A survey of interactive remote rendering systems. *ACM Comput. Surv.*, 47(4). (page 22)

[93] Shi, S., Nahrstedt, K., and Campbell, R. (2012). A real-time remote rendering system for interactive mobile graphics. *ACM Trans. Multimedia Comput. Commun. Appl.*, 8(3s):46:1–46:20. (page 22)

[94] Sitthi-amorn, P., Lawrence, J., Yang, L., Sander, P. V., and Nehab, D. (2008a). An Improved Shading Cache for Modern GPUs. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 95–101. Eurographics Association. (page 19)

[95] Sitthi-amorn, P., Lawrence, J., Yang, L., Sander, P. V., Nehab, D., and Xi, J. (2008b). Automated reprojection-based pixel shader optimization. *ACM Transactions on Graphics*, 27(5):1. (page 21)

[96] Steinberger, M., Kenzel, M., Kainz, B., and Schmalstieg, D. (2012). Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. (page 32)

[97] Strasburger, H., Rentschler, I., and Juttner, M. (2011). Peripheral vision and pattern recognition: A review. *Journal of Vision*, 11(5):13–13. (page 54)

[98] Swafford, N. T., Iglesias-Guitian, J. A., Koniaris, C., Moon, B., Cosker, D., and Mitchell, K. (2016). User, metric, and computational evaluation of foveated rendering methods. In *Proceedings of the ACM Symposium on Applied Perception - SAP '16*. ACM Press. (page 19, 21)

[99] Tanner, C. C., Migdal, C. J., and Jones, M. T. (1998). The clipmap: A virtual mipmap. In *Proceedings SIGGRAPH*, pages 151–158. (page 13)

[100] Tatarinov, A. and Sathe, R. (2018). Texture Space Shading. SIGGRAPH Exhibitor Talks. (page 13)

[101] Teler, E. and Lischinski, D. (2001). Streaming of complex 3d scenes for remote walkthroughs. *Computer Graphics Forum*, 20(3):17–25. (page 22)

[102] Tole, P., Pellacini, F., Walter, B., and Greenberg, D. P. (2002). Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics*, 21(3). (page 21, 76)

[103] Turkowski, K. (1990). *Graphics gems*, chapter Filters for Common Resampling Tasks. Academic Press, Boston. (page 19, 47)

[104] Vaidyanathan, K., Salvi, M., Toth, R., Foley, T., Akenine-Möller, T., Nilsson, J., Munkberg, J., Hasselgren, J., Sugihara, M., Clarberg, P., Janczak, T., and Lefohn, A. (2014). Coarse Pixel Shading. In *Proceedings of High Performance Graphics*, HPG '14, pages 9–18. Eurographics Association. (page 19)

[105] van Merrienboer, B., Breuleux, O., Bergeron, A., and Lamblin, P. (2018). Automatic differentiation in ml: Where we are and where we should be going. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31, pages 8757–8767. Curran Associates, Inc. (page 75)

[106] van Waveren, J. M. P. (2009). id tech 5 challenges - from texture virtualization to massive parallelization. In *SIGGRAPH 2009 Course: Beyond Programmable Shading*. (page 13)

[107] Van Waveren, J. M. P. (2016). The asynchronous time warp for virtual reality on consumer hardware. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST*, volume 02-04-November-2016, pages 37–46. Association for Computing Machinery. (page 14, 15, 22)

[108] Walter, B., Drettakis, G., and Parker, S. (1999). Interactive rendering using the render cache. In *Eurographics*, pages 19–30. Springer Vienna. (page 19, 76)

[109] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., and Greenberg, D. P. (2005). Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics*, 24(3):1098–1107. (page 99)

[110] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13:600–612. (page 21, 110)

[111] Wang, Z. and Li, Q. (2011). Information content weighting for perceptual image quality assessment. *IEEE Transactions on Image Processing*, 20(5):1185–1198. (page 21, 48)

[112] Wang, Z., Simoncelli, E., and Bovik, A. (2003). Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers*. IEEE. (page 56)

[113] Watson, A. B. (2001). Digital video quality metric based on human vision. *Journal of Electronic Imaging*, 10(1):20. (page 22)

[114] Yang, K.-C., Guest, C., El-Maleh, K., and Das, P. (2007). Perceptual temporal quality metric for compressed video. *IEEE Transactions on Multimedia*, 9(7):1528–1535. (page 22)

[115] Yang, L., Liu, S., and Salvi, M. (2020). A survey of temporal antialiasing techniques. *Computer Graphics Forum*, 39(2):607–621. (page 20, 47)

[116] Yang, L., Nehab, D., Sander, P. V., Sitthi-amorn, P., Lawrence, J., and Hoppe, H. (2009). Amortized supersampling. *ACM Transactions on Graphics*, 28(5):1. (page 20)

[117] Yang, L., Tse, Y.-C., Sander, P. V., Lawrence, J., Nehab, D., Hoppe, H., and Wilkins, C. L. (2011). Image-based bidirectional scene reprojection. In *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11*, volume 30, page 1. Association for Computing Machinery (ACM). (page 14, 20, 110)

[118] Yang, L., Zhdan, D., Kilgariff, E., Lum, E. B., Zhang, Y., Johnson, M., and Rydgård, H. (2019a). Visually lossless content and motion adaptive shading in games. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 2(1):1–19. (page 19)

[119] Yang, W., Zhang, X., Tian, Y., Wang, W., Xue, J.-H., and Liao, Q. (2019b). Deep learning for single image super-resolution: A brief review. *IEEE Transactions on Multimedia*, 21(12):3106–3121. (page 19)

[120] Yoon, I. and Neumann, U. (2000). Web-based remote rendering with ibrac (image-based rendering acceleration and compression). *Computer Graphics Forum*, 19(3):321–330. (page 23)

[121] Yuksel, C. (2017). Mesh color textures. In *Proceedings of High Performance Graphics*, HPG '17, pages 17:1–17:11. (page 13, 31)