Memory-efficient Real-time Path Tracing for Computer Games

Master thesis by Jan Kelling
Date of submission: 19.02.2023

1. Review: Prof. Dr. Arjan Kuijper
2. Review: Daniel Ströter, M.Sc.
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

This thesis was only made possible by numerous wonderful people. First, I would like to thank Daniel Ströter for the detailed feedback and for never tiring of pointing out all my sloppy phrasing choices. I extend my gratitude to Prof. Dr. Arjan Kuijper for accepting my request to supervise this thesis. Furthermore, I am thankful to Ubisoft Mainz for supporting me in my endeavors, especially to Wolfgang Klose for making this thesis possible and helping me with any request. I thank Dr. Aaron Scherzinger and Martin Tillmann for their extended feedback, Tim Kaiser for being a fierce defender of the Oxford comma, and Frank Hoffmann and Ramin Safarpour for their many constructive discussions. Lastly, I would like to thank Clara for telling me that the path traced images look more beautiful than the rasterized ones. And for everything else.

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Jan Kelling, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 19.02.2023

Jan Kelling

Abstract

Rendering visually convincing and realistic images requires accurate lighting computation. Path tracing [Kaj86] has long been used in offline rendering to achieve this goal. Recent graphics processing unit (GPU) advancements and new sampling algorithms enable path tracing in real-time. This thesis investigates whether real-time path tracing on commodity hardware is feasible for the current generation of video games. We integrate and evaluate a real-time path tracer into an existing game engine and provide an in-depth investigation of performance and memory measurements. Our case study is done with *Anno*, a state of the art city building game series that renders large and highly dynamic user-built scenes. We present an efficient hybrid rendering pipeline able to compute path traced indirect lighting in real-time within the memory limits of consumer hardware. To solve observed performance and memory issues, we contribute two novel path tracing methods:

Light path guided culling (see section 3.3.2) tracks the number of light paths passing regions of the scene. The information is used to only selectively build and maintain the required parts of the scene representation in graphics memory. This technique allows for increased visual quality, improved performance, and significantly reduced memory consumption compared to existing simple heuristics for partial scene representation. This generic method can be combined with various kinds of hardware accelerated ray tracing methods.

Stochastic vegetation ray skipping (see section 3.4) is employed to allow fast ray tracing in scenes with high amounts of vegetation. We present a method employing hybrid rendering, relying on rasterization for primary rays and direct lighting where possible, and a heuristic utilizing ray skipping through entire vegetation instances to speed up indirect lighting computation. This method provides ray tracing speedups of factors greater than two for vegetation-heavy scenes and proves crucial to achieving real-time performance.

Contents

1 Introduction

Figure 1.1: Differently shaded Sponza [Int22] scene with added reflective curved surfaces.
Left: Approximating indirect diffuse lighting with a trivial position-independent term
Middle: Ray traced ambient occlusion (AO), correctly computing light coming from the sky.
Right: Fully path traced image showing accurate indirect lighting.

The field of computer graphics has seen massive improvements over the last decades, especially in the area of real-time rendering for games. Rendered graphics are getting more realistic and expressive, while new targets for frame rates and therefore perceived smoothness are achieved at the same time. In practice, systems have to find trade-offs between visual quality and performance. Memory consumption is also a limiting factor. In the context of games, this thesis defines "real-time rendering" to have a target of at least 30 frames per second (fps), since lower framerates have been shown to significantly reduce the immersion due to input latency [WB00].

To deliver high fidelity as quickly as possible, massively parallel processors in the form of graphics processing units (GPUs) were developed specifically for 3D rendering and games. To optimally parallelize computation, rendering techniques for games are traditionally based on rasterization [Gha+89] and therefore deviate significantly from more realistic offline (i.e. less time-critical) renderers. Lighting results are often crude approximations of the physically based principles of light transport. The more accurate techniques deployed in offline renderers usually rely on path tracing [Kaj86], in which entire light paths are traced for multiple bounces through the scene. To compute interactions between light paths and scene objects, ray tracing determines the intersection points. This technique has a higher computational cost factor and is harder to parallelize efficiently. Therefore, it has rarely been used in interactive rendering.

The ever growing complexity of GPUs and advancements in hardware accelerated ray tracing [Wym+18; HBW20] in recent years now allow the utilization of ray tracing techniques to greatly increase lighting quality

even in real-time renderers. But even when applying techniques such as ray traced shadows [BWB19], ambient occlusion (AO) [Bav+18], global illumination (GI) [Hal+21; Ouy+21], or reflections, many corner cases exhibiting incorrect lighting can still be observed. Path tracing solves these limitations by tracing rays through the scene for **all** lighting calculations in a physically-motivated manner. Therefore, it can capture complex visual phenomena such as light paths from arbitrary light sources with both diffuse and specular bounces. See figure 1.1 for a comparison between a simple indirect lighting approximation, ray traced AO and a fully path traced image.
Together with advancements in sampling strategies [Bit+20] and denoising [Sch+17; Zhd21], using path tracing in a real-time context such as a game engine becomes potentially possible [Sch19; Str23; Bur23a]. However, there are numerous challenges when adapting an existing real-time renderer to a path traced one, as the context offers many constraints.

Hypothesis: On modern consumer-grade GPUs, path tracing can be employed to shade the highly complex, dynamic scenes rendered in the current generation of video games, even with user-generated worlds and without the option of pre-processing scenes. There are various technical challenges, especially when the graphical assets and effects are not authored for a path traced pipeline, but they can be sufficiently solved. Techniques and optimizations balancing correctness, visual quality, tracing time, and memory consumption can be applied to completely replace legacy lighting approximations with path tracing.

In this thesis I use “we”, as is common in English scientific literature, to present my work. We will explore techniques to allow fast and memory-efficient path tracing of complex and highly dynamic scenes. There will be in-depth examinations of specific situations that are hard to solve with real-time path tracing, including a discussion of possible solutions. To investigate scalability in representative contemporary game scenes and deal with the challenges of user-generated game worlds that do not allow any pre-processing, we will integrate a fully path traced renderer into the existing game engine of the Anno series [Ubi23] and present methods to achieve real-time performance. We intend to use existing hardware-accelerated ray tracing implementations and aside from our minimum target of 30 fps on current hardware, our goal is to keep memory requirements within the limits of typical gamer commodity hardware.

Chapter 2 will outline the basic concepts of path tracing and related work which will provide the required background for the subsequent chapters of the thesis. After discussing various challenges to implementing real-time path tracing in section 3.1, we describe our approach to rendering game assets and realizing typical game effects in section 3.2. We will present our solution for acceleration structure management in this highly dynamic and generic context in section 3.3. The aforementioned trade-offs will be realized via level-of-detail mechanisms, heuristics for acceleration-structure building as well as the best possible distribution of our ray budget to the rendered pixels.

Furthermore, we will present a novel technique for fast and memory-efficient ray tracing of vegetation assets in section 3.4. While this technique only approximates aspects of light transport through vegetation, we will show that it delivers visually convincing results while improving runtime significantly, halving our path tracing timings in vegetation-heavy scenes.

Our evaluation in chapter 4 will consider a wide range of consumer GPUs from different vendors. We will show that real-time path tracing is possible on capable current generation hardware and outline approaches for older or less powerful GPUs.

2 Background and Related Work

While real-time path tracing is a recent development, path tracing has a long history in offline rendering and there is a rich body of literature on various ray tracing techniques. We intend to give a short overview of the history of ray tracing and path tracing, summarize the state of the art with respect to game rendering techniques, and establish some terminology used throughout the rest of this thesis.

2.1 Path Tracing

Using ray tracing to compute light transport instead of just evaluating visibility was first employed by Whitted et. al. in 1980 [Whi80]. This technique only considered perfect reflections and refractions. It developed into distributed ray tracing [CPC84] and, eventually, the rendering equation and full path tracing as we use it today [Kaj86]. To generate an image using the rendering equation, complex scene-dependent nested integrals are solved for each pixel using Monte Carlo sampling [MU49]. In each iteration, a single path is followed and at each surface intersection point, the next ray direction of the path is selected randomly. Accumulating the light along many of these random paths will finally converge to the solution of the rendering equation.

In practice, there are multiple ways to implement path tracing. One option is to trace paths starting from light sources until they hit the camera, comparable to how light behaves in reality. Due to the random bounces, the probability of paths hitting the camera is low. Therefore, path tracing techniques often reverse the physical light transportation process and start their paths at the camera, with rays bouncing around the scene, accumulating the product of throughput to the camera and gathering light along the way. This is known as backwards path tracing and what our method will build upon. See algorithm 1 for an algorithmic path tracing sketch using this technique. In this context, we speak of *primary rays/hits* for the rays emanating directly from the camera and their intersections with the scene, i.e. the tracing and the potential hit found during $bounce = 0$. While this technique of starting paths at the camera often requires fewer followed paths to converge to the correct solution, it is not efficient for capturing some specific effects such as caustics and specular-diffuse-specular light interactions. For those, the function of incoming radiance at a given surface can have high frequency (as high radiance is coming from just one specific direction) and is hard to predict, causing Monte Carlo sampling to converge arbitrarily slowly.

A useful notation and tool to discuss light paths is representing them as regular expressions [Hec90] with each letter representing a light interaction. We use L for light sources, E for eye (camera), D for diffuse bounces, S for specular reflections, and T for transmission. For instance, the expression `LD+E` will then describe all light paths that start at a light and involve only diffuse surface reflections before reaching the camera. This describes light paths captured by diffuse GI and radiosity approaches [Gor+84]. It is often useful to limit S and T to surfaces with low roughness and just use D otherwise for all surface interactions. With this notation, `L(S|T)+E` describes light path that have not been strongly scattered.

Algorithm 1 Simple single-sample path tracing algorithm starting at the camera.
$TraceRay$ tests for surface intersection, given ray origin and direction.
$ShadeHit$ evaluates lighting (for instance, emission) at a found surface hit.
$GenerateBounce$ will generate a random light bounce on the surface and update the throughput using the surface's bidirectional scattering distribution function (BSDF), see section 2.3 for more details.

```
accumLight ← 0                                   ▷ radiance (e.g. RGB tuple) in range [0, ∞]
throughput ← 1                                   ▷ radiance factor (e.g. RGB tuple) in range [0, ∞]
rayOrigin ← camera position
rayDir ← camera ray direction for this pixel
for bounce = 0 ...maximum number of bounces do
    hit ← TraceRay(rayOrigin, rayDir)
    if hit not valid then
        Compute incoming environment radiance γ, e.g. from the sky
        accumLight ← accumLight + throughput · γ
        break
    end if
    accumLight ← accumLight + throughput · ShadeHit(hit, rayDir)
    rayOrigin ← hit.position
    GenerateBounce(hit, rayDir, throughput)      ▷ Will update rayDir and throughput
end for
return accumLight                                ▷ accumLight is the final pixel radiance entering the camera
```

Path Tracing has successfully been integrated into computer games before, one of the first examples was a ray traced renderer for *Quake II* [Sch19]. *Metro Exodus* also had a ray tracing mode considering most light interactions [Arc+19]. Nvidia has released its *Omniverse* platform [Lla19] and detailed how to build a real-time path tracer [Str23]. Recently, the first state of the art games have released experimental path tracing modes, most notably *Cyberpunk 2077* [Bur23a].

On the one hand, we provide an overview of a method to integrate real-time path tracing into the current generation of games. In comparison to the previously released games, we employ it in a more general video game setting, involving user-generated content and a freely movable camera that can jump around in the scene. Moreover, we present new methods, especially for vegetation rendering and highly dynamic scenes with a high number of animations.

2.2 Optimizations

While the two previously mentioned path tracing approaches are called unidirectional path tracing, they can also be combined into bidirectional path tracing [Laf95], which attempts to leverage the strengths of both. The algorithm constructs light paths originating from the camera as well as light paths originating from light sources and then attempts to connect them, all while staying *unbiased*. We speak of an unbiased path tracing technique when the result indeed converges to the exact solution of the rendering equation. On the other hand, *bias* is the difference between the expected value of our Monte Carlo sampling and the exact solution, e.g. as a result of approximations and optimizations.

Metropolis Light Transport [VG97] is another unbiased path tracing implementation that attempts to reduce variance by sampling paths more efficiently, just mutating known paths that transport high magnitudes of light

instead of sampling randomly each time. Similar to bidirectional path tracing, photon mapping [Jen96; Jen01] and progressive photon mapping [HOJ08] attempt to reduce variance for scenes in which light sources are unlikely to be reached by constructing light paths originating at light sources, albeit at the cost of introducing bias. In comparison to full bidirectional and Metropolis path tracing, this technique is better suited for real-time constraints and parallelization on the GPU and therefore a suitable extension of our path tracing method.

Another important optimization for path tracing is next event estimation (NEE). Instead of bouncing rays randomly around the scene and just evaluating light emission at a given intersection, radiance coming from known light sources is explicitly evaluated at intersection points. This involves additional shadow rays but often leads to significantly faster convergence without introducing bias. In algorithm 1, instead of just considering surface emission in $ShadeHit$, we would additionally consider known light sources — e.g. the sun, the environment or randomly sampling local ones — and cast respective shadow rays. This technique is especially important for our real-time context and typical game lighting such as point lights or scenes mainly shaded by a directional light like the sun or moon. Therefore, our method builds upon NEE.

Path space regularization is another important approach. Originally, it was designed to capture light coming from infinitely small light sources such as traditional point lights that do not fit well into correct light transport computations [KD13]. Now, it is often used in a general manner to reduce variance by increasing the roughness at secondary bounces [Wei+21]. In algorithm 1, this could be implemented by clamping the surface roughness to a minimum after we had at least one highly-scattered (i.e. diffuse or high-roughness) bounce before executing $ShadeHit$ and generating the next bounce. We rely on this technique, even though it introduces a slight bias, since it removes "firefly" artifacts and can lead to significantly faster convergence for scenes involving glossy or light-transmissive materials.

Light paths can be arbitrarily long but we intend to terminate in a limited number of steps, ideally without introducing significant bias. The common technique to solve this problem is commonly referred to as "Russian Roulette", i.e. randomly determining paths and multiplying the contribution of the remaining paths by the inverse of the termination probability. Optimizing the used heuristics to compute the termination probability can significantly improve the ratio between convergence rate and runtime [Rat+22]. In section 3.2, we present a simpler and cheaper heuristic for path termination that is well-suited for real-time rendering and specifically tailored to typical game situations.

2.3 Importance Sampling and Path Guiding

One of the most important optimizations for path tracing is importance sampling. At each bounce in our path, we need to compute the non-trivial integral of the rendering equation. Monte Carlo integration with importance sampling means approximating the integral

$$\int_{\Omega} f(x)\,dx$$

with n samples x_i, drawn from a probability density function (PDF) p, by

$$\frac{1}{n} \cdot \sum_{i=0}^{n} f(x_i)/p(x_i)$$

As long as $p(x) > 0$ holds for all $x \in \Omega$ where $f(x) > 0$, this expression will converge to the desired result for $n \to \infty$. One can show that the convergence of the approximation is faster, the closer p is proportional to the integrated f. Intuitively, one can imagine importance sampling as variance reduction: if p were perfectly proportional to f, all summands would have the same value. The ratio between f and p would have to be exactly $\int_\Omega f(x)\,dx$ and even with $n = 1$, the approximation would return the exact solution. In practice, finding a PDF p that is proportional to f is usually not possible. Nonetheless, PDFs mimicking certain characteristics of f can often be found.

In algorithm 1, importance sampling is implemented in the $GenerateBounce$ function. A good importance sampling approach would return new ray bounce directions that result in higher throughput (and therefore higher accumulated light) proportionally more often. Ignoring the transmissive part of surfaces and just focusing on the bidirectional reflectance distribution function (BRDF) for simplicity, a simple $GenerateBounce$ implementation without importance sampling is shown in algorithm 2 and an implementation with basic importance sampling in algorithm 3.

Algorithm 2 Bounce Generation

function GENERATEBOUNCE(hit, inout rayDir, inout throughput)
    Generate $newDir$ randomly on hemisphere
    $throughput \leftarrow throughput \cdot \text{BRDF}(hit, rayDir, newDir)$
    $rayDir \leftarrow newDir$
end function

In algorithm 3, the BRDF is split into diffuse and specular parts since good approximating PDFs can be found for the individual parts. This can be understood as a form of multiple importance sampling (MIS) [VG95]. The algorithm prefers specular light bounces for metallic surfaces since the diffuse contribution approaches 0 for them. For the diffuse part of the BRDF, we can easily employ perfect importance sampling for the lambert BRDF by just generating random samples on the cosine-weighted hemisphere. Non-trivial physically based specular BRDFs can usually not completely be transformed into a PDF but a good approximation works well in practice. The geometric function is usually the biggest contributor, driving the shape of the specular reflection lobe. For the commonly used GGX geometric function [Wal+07], an efficient PDF transformation has been found [Hei18]. We can see how this function will have a lower variance in its $throughput$ modification, especially considering specular metallic surfaces with low roughness (i.e. mirror-like surfaces). See figure 2.1 for a comparison.

Algorithm 3 Importance-sampled Bounce Generation

function GENERATEBOUNCE(hit, inout rayDir, inout throughput)
    $specProb \leftarrow 0.5 + 0.5 \cdot hit.metalness$
    if $rand() < specProb$ **then**
        $newDir \leftarrow$ IMPORTANCESAMPLEGGX$(hit.normal, hit.roughness)$
        $throughput \leftarrow throughput \cdot$ SPECULARBRDFWITHOUTGGX$(hit, rayDir, newDir)/specProb$
    else
        Generate $newDir$ randomly on cosine-weighted hemisphere
        $throughput \leftarrow throughput \cdot hit.albedo \cdot (1 - hit.metalness)/(1 - specProb)$
    end if
    $rayDir \leftarrow newDir$
end function

MIS [VG95] is an important building block for importance sampling method since it allows the combination of many PDFs in a single estimator. Generation of good samples without an explicit PDF can be achieved using

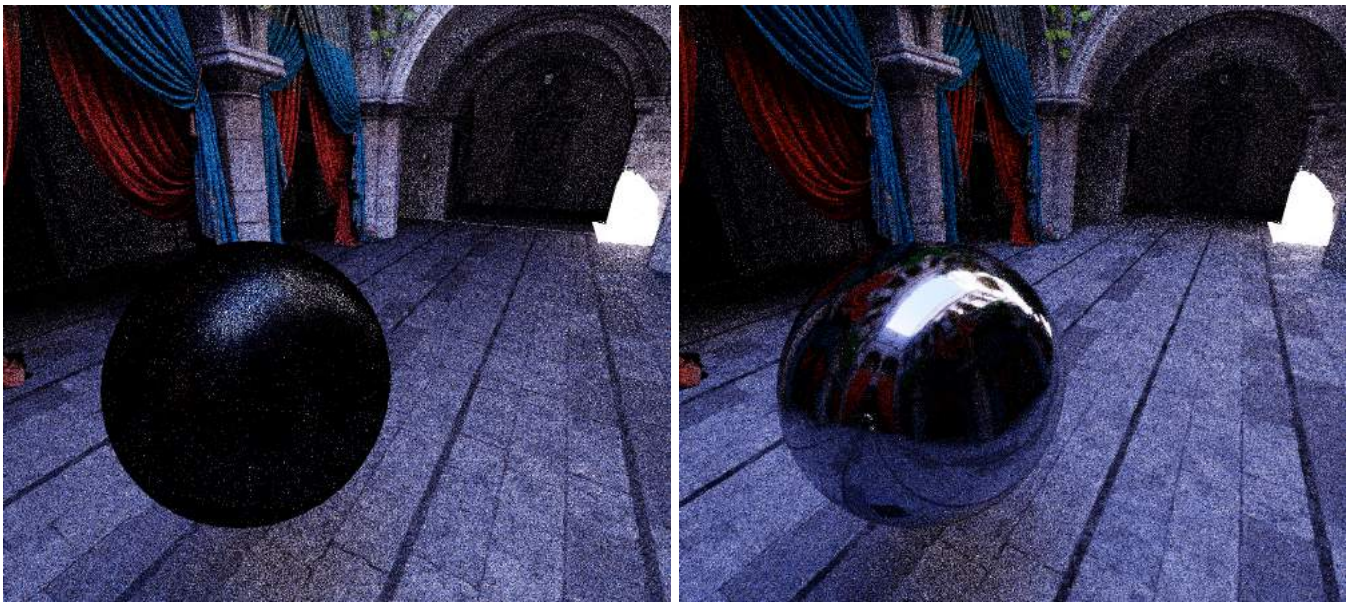

Figure 2.1: The importance of importance sampling. Sponza with a mirroring ball, 10 samples per pixel (spp).
Left: No importance BRDF sampling applied, as outlined by algorithm 2.
Right: With BRDF importance sampling, as outlined by algorithm 3.

reservoir sampling [TCE05]. In addition to importance sampling the BSDF, we can also attempt to employ importance sampling to generate rays preferably in directions where we expect more incoming radiance. This technique is known as path guiding and usually requires non-trivial data structures to estimate/remember incoming radiance [MGN17]. Deep neural networks have been proposed to generate samples [Mül+18]. Work has also been done to port path guiding to real-time path tracing [DHD20].

As previously mentioned, path tracers using NEE often stochastically choose local light sources for shading. This can be importance sampled as well, as done by reservoir spatiotemporal importance resampling (ReSTIR) [Bit+20]. Biased and unbiased variants of ReSTIR have been proposed. The biased variant drastically reduces the number of traced rays even further and allows for heuristics keeping the introduced bias acceptable for most cases. ReSTIR GI [Ouy+21] uses ReSTIR ideas for global illumination. Applying ReSTIR to volume rendering leads to significant variance reduction [LWY21]. ReGIR [BJW21] uses a global grid to extend ReSTIR ideas to non-primary hits. Generalized resampled importance sampling [Lin+22] generalizes ReSTIR by proposing a general path reuse technique.

We will employ the established BSDF importance sampling strategies. ReSTIR approaches are a suitable extension of our method for scenes suffering poor lighting conditions or involving many light-emissive surfaces. In section 4.4, we discuss an example of a scene advanced light transport or path reuse techniques.

2.4 Realtime Rendering and Rasterization

Computing the intersection between rays and complex scenes can be computationally expensive. Before the advent of hardware accelerated ray tracing [Wym+18; HBW20], ray tracing was not commonly used in real-time rendering settings such as games. The ubiquitous approach to display geometry, i.e. to determine which primitives are covering which pixels, was **rasterization** [Gha+89]. Surfaces are simplified into a set

of triangles, the most trivial planar primitive, with the idea that even complicated (rounded) surfaces can be approximated arbitrarily well with enough triangles. Rasterization allows efficient utilization of parallel hardware. Shading requires executing the same function for all pixels generated by rasterizing a primitive. While rasterization and ray tracing both solve the visibility problem, the fundamental difference is that rasterization requires coherent rays. Ray tracing, on the other hand, has no such restriction. In practice, this comes with the cost of worse utilization of highly parallel hardware due to the reduced coherency. Traditional render pipelines offer different possible approaches for solving the different parts of the rendering equation. Primary hits can be computed with rasterization just as well as with ray tracing, without a significant difference between the two. Visibility problems such as shadowing from punctual or directional lights can be realized using rasterization into shadow maps [SWP10; Lik+15]. However, arbitrary secondary rays for indirect lighting cannot be computed easily. Many computer graphics techniques relevant to games deal with the shortcomings of rasterization approaches compared to fully solving the rendering equation using path tracing. There is a large body of real-time techniques used in games for achieving high visual quality, especially to approximate GI:

- The most simple indirect lighting approach used by older games is to just choose a (possibly art-driven) constant indirect lighting term. This very crude approximation will result in loss of detail and sometimes hard readability of scenes in scenes that have just indirect lighting.
- To produce convincing reflections, environment maps [BN76] have long been used. They allow showing a static environment correctly reflected on glossy surfaces. This environment map is completely independent from (and might not match at all) the currently visible geometry or lighting setup.
- With the advent of physically based rendering (PBR), image-based lighting (IBL) became a more physically correct variant of environment maps [Kar13]. It provides good approximations of rough reflections and computes indirect diffuse lighting from the same source as specular reflections. This technique increases visible detail in materials and perceived depth readability in indirectly lit scenes.
- Screen-space techniques such as screen-space reflections (SSR) [SKS11; Wro14] or screen space ambient occlusion (SSAO) [Mit07] are used to recover some detail of indirect lighting with the information already available on the screen. Similarly, shadowing artifacts resulting from the limits of shadow maps can be improved using screen-space shadowing techniques [Cow+15].
- Assuming a (mostly) static scene and lighting, indirect diffuse lighting could be pre-computed offline into so-called lightmaps. In practice, this is often realized using radiosity [Gor+84]. This approach can result in a visual quality of indirect (diffuse) lighting close to fully path traced light transport simulations. However, it requires significant amounts of graphics memory, and the assumption of static scenes and lighting is unrealistic for many games. It leads to visual issues and lighting mismatches for the dynamic elements of a scene.
- IBL techniques can also be applied using probes placed in the scenes to capture indirect diffuse/specular lighting at specific points. The technique then applies the captures lighting as an approximation for the lighting of close surfaces. These probes can even be refreshed dynamically at runtime, allowing for dynamic scenes and lighting conditions. However, the spatial approximations of this approach will often lose details relevant for convincing indirect lighting.

Since the advent of hardware accelerated ray tracing, many new techniques have been proposed to solve some of the shortcomings and problems of rasterization-based techniques. Ray tracing based techniques to improve lighting, such as ray traced reflections [DS19; Hai19], shadows [BWB19], and AO [Bav+18] have been proposed. These techniques allow solving specific parts of the rendering equation more accurately

without solving everything. Due to their restricted nature, such techniques are more efficient than fully path tracing the scene. This comes at the cost of visual artifacts and missing details.

For instance, ray traced shadows [BWB19] from global and local light sources have the advantage of delivering accurate shadowing information. Such techniques are able to approximate shadow from emissive surfaces, i.e. non infinitely small light sources. In contrast, shadow maps are generally limited to small light sources. Furthermore, ray traced shadows often include more details than shadow maps and remove shadow map artifacts like missing contact shadows or aliasing patterns.

Ray tracing diffuse indirect lighting techniques [MGM19; Hal+21] in games often build upon spatial discretization of the scene. They heavily rely upon the assumption that the nature of diffuse indirect lighting is of very low frequency and therefore approximation at certain points in the scene and interpolating in between works well. Alternatives that do not rely on spatial discretization use denoising techniques discussed below to accumulate samples over multiple frames and thus deliver a higher degree of detail [Arc+19; Sch19].

One common optimization is to use rasterization or known screen-space information where possible and only fall back to ray tracing where needed [BS22; Bav+18].

Figure 2.2: Left: Rasterized scene from the game *Anno 1800*. Right: Path traced image of the same scene. Capturing the light bouncing off the glossy pipe onto the building requires following mixed diffuse-specular light paths.

The difference between using these ray tracing effects to using full path tracing is in the details. Path tracing captures the combination of all these ray tracing effects: direct lighting, shadows, diffuse, and specular global illumination. The interplay between these effects makes it difficult — and in some situations impossible — to approximate them well with independent ray tracing techniques. For instance, even when using ray traced diffuse GI and specular reflections together, one can usually not expect to see light bounces as in figure 2.2 as this is caused by a coupling of specular with diffuse surface interactions. Light-transmissive surfaces are another problem that cannot fully be solved with independent ray tracing effects. Solving the rendering equation requires following *arbitrary* light paths involving bounces with all different types of reflections. There has also been work to integrate volumetric effects into GPU path tracing [Nov+18; HE21].

This thesis aims to evaluate whether path tracing is a viable option to replace all other lighting approximations and independent ray tracing effects. At the same time, our method will leverage traditional approaches

and optimizations such as hybrid rendering and fallbacks to shadow map rasterization and screen-space information.

2.5 Denoising

Due to the probabilistic Monte Carlo nature of path tracing, the resulting images need many samples to converge. Before convergence, the accumulated samples contain significant per-pixel variance. Denoising is the process of trying to remove that variance, potentially allowing us to obtain visually pleasing images much faster than waiting for the Monte Carlo algorithm to converge. This comes at the cost of correctness, denoising usually introduces bias into the otherwise correct Monte Carlo method.

Typical denoising algorithms use a combination of spatial bilateral blurs and temporal accumulation. The latter is similar to temporal anti-aliasing (TAA) [Kar14]. Both types of accumulation have different properties, different use cases, and drawbacks. For a static scene and camera, temporal sample reuse does not introduce any bias, as it can be understood as a proper accumulation of samples in this case. With careful reprojection (based on motion vectors mapping pixels from the last frame to the current one) and rejection strategies, the bias introduced for non-static scenes and camera movement can be reduced. However, one can often observe temporal lag and artifacts known as ghosting: Non-instant response to changes in the scene, such as lighting or occlusion chances. Spatial blurring effectively mixes Monte Carlo samples from different functions, this will automatically introduce bias. Samples from nearby surfaces with similar material properties will likely have been sampled from a *similar* function of incoming radiance, bilateral blurring can use these properties to keep the introduced error low. However, there are cases for which the sampled functions differ fundamentally without any significant change in surface parameters. This can lead to visual artifacts.

While denoising has been of interest for a long time, there have been some recent advancements tailored specifically for light transport. The methods make it an irreplaceable tool for path traced renderers, real-time as well as offline. Separating the lighting components of the rendering equation and employing different denoising strategies proved helpful [Sch+17]. Lighting is often split into direct, indirect diffuse, and indirect specular components, based on the type of bounce at the primary hit [Mar+17]. For the indirect lighting components, using large image-space filters where not enough temporal information is available showed promising results. This can be achieved with a-trous filters [Sch+17]. Later, the cheaper approach of bilateral upsampling from a downscaled texture was proposed [Zhd20; Zhd21]. Temporal lag is one of the most objectionable artifacts of modern denoising techniques. Solutions based on re-evaluating parts of the rendering equation to detect changes have been proposed [SPD18]. An important idea is to demodulate albedo and possibly other surface properties to avoid overblurring surface details [Zhu+21]. Denoising will then happen on inputs that can be understood as being closer to incoming irradiance instead of outgoing radiance. This means that high-frequency surface coloring details are not blurred doing denoising, as they are only applied afterward. The importance of this technique can be seen in figure 2.3.

Denoising low-roughness reflective or transmissive surfaces is still a problem since no or little data can be shared spatially and reprojection during movement is imperfect. A real-time suitable solution for this is primary surface replacement (PSR) [Pan18]: Instead of denoising the mirror-like surface directly, we run denoising on a virtual surface mirroring the first high-roughness or diffuse surface interaction encountered during path tracing. A generalization of this technique uses multiple denoising buffers, allowing one to denoise even complex perfect reflection and refraction bounces without significant issues [Zim+15]. This has profound memory and performance costs for the multiple denoising passes, but for a small number of additional buffers, this solution is a real-time and GPU-suitable extension of our method.

Figure 2.3: Anno scene only indirectly lit, showing importance of demodulation.
Left: Without demodulation, surface details are completely over-blurred.
Right: With demodulation enabled, denoising is applied (roughly) to incoming instead of outgoing light and surface details are only applied later.

New denoising approaches based on deep learning have been proposed [HY21; Bur23b]. They are especially important in conjunction with image upscaling algorithms since denoising is executed before upscaling and removes the jittering that is needed by modern temporal upscaling algorithms. Deep learning approaches could solve both problems in a single step.

For our method, we stick to denoising algorithms not based on deep learning since they are easier to deploy and do not depend on GPU-vendor-specific details. Namely, we rely on the open-source ReBLUR [Zhd21] denoiser and focus on improving the input data given to the denoiser since we believe this to have more potential for improvements.

2.6 Hardware Acceleration

Acceleration structures such as bounding volume hierarchies [KK86; Mei+21] can be used to significantly speed up the intersection test between ray and geometry. In recent years, consumer-grade GPUs additionally allow for hardware accelerated intersection tests.

Hardware accelerated ray tracing and therefore modern graphics APIs such as Direct3D 12 [Mic20] and Vulkan [HBW20] rely on a two-level acceleration structure scheme. Individual meshes are built into **bottom-level acceleration structures (BLASes)**, which in turn are then used to build a **top-level acceleration structure (TLAS)**. There are two ways to perform intersection tests with a TLAS: using ray queries or ray tracing pipelines. Ray queries are new shader intrinsics that can be called in any traditional shader stage to query intersection points between a given ray and the geometry present in a TLAS. Ray tracing pipelines, on the other hand, are a new type of pipeline that include new shader stages such as ray generation shaders, miss shaders, and then various hit shaders that are executed when rays intersect with meshes in the TLAS. The graphics APIs offer new entry points for invoking ray tracing pipelines. Ray tracing pipelines get invoked with a **shader table**, mapping geometries in the TLAS to hit shaders. The entry point of a ray tracing pipeline

is a *raygen* shader. It is similar to a compute shader but like all shader stages of a ray tracing pipeline, it can additionally call a `TraceRay` intrinsic that invokes new shader stages for intersected meshes. Ray queries do not use shader tables or automatically invoke shaders, they can be seen as a lower level alternative that can be used in any shader stage.

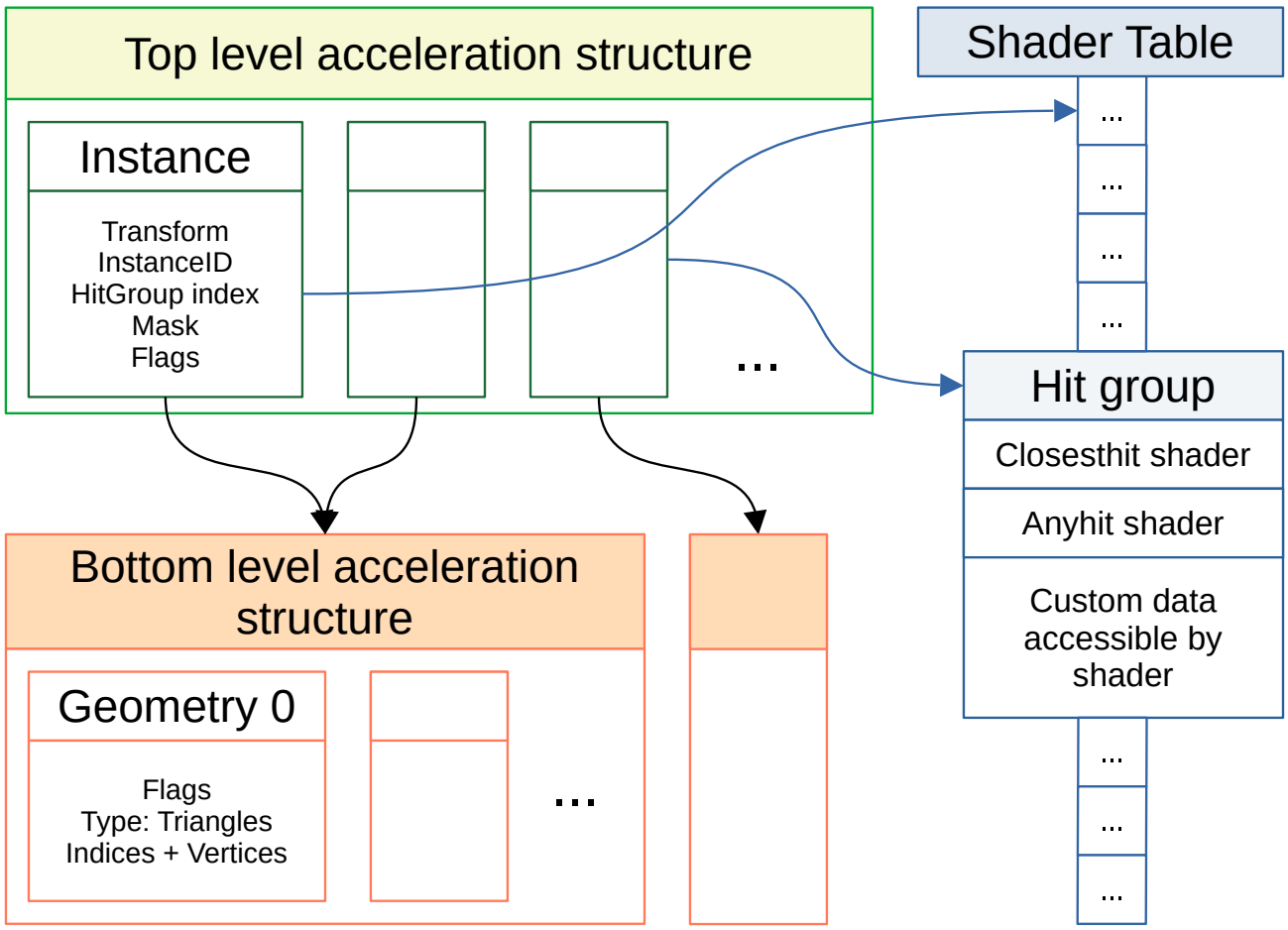

Figure 2.4: TLAS/BLAS handling and shader table of ray tracing graphics APIs. Figure inspired/Simplified from [Mic20]

One BLAS can contain multiple **Geometries**. These geometries can be of triangular or procedural type, with the latter one using custom shader code to determine the exact intersections. Since our method does not use them, they are ignored in the following. Each geometry has custom flags containing information about culling and whether the geometry is to be considered opaque. The numbered ID of the intersected geometry of a BLAS can be retrieved in shaders. As sketched out in figure 2.4, a single BLAS can be reused multiple times when building a TLAS. Besides the reference to a BLAS, the **instances** in the TLAS include an affine transformation matrix, an arbitrary ID that can be retrieved in the hit shader, a mask allowing ray tracing shaders to specify which instances should be hit as well as a hit group index. When using ray tracing pipelines instead of ray queries, this hit group index is used to calculate an index into the shader table. The shader table consists of many *hit groups*. Each *hit group* can contain an *anyhit* shader, a *closesthit* shader, and custom data accessible by the respective shaders. The *anyhit* shader is executed whenever a ray intersects with non-opaque geometry and is able to inform the hardware that this specific intersection should be ignored. As soon as the closest valid intersection along a ray was determined, the **closesthit** shader is run. When no closest intersection is found, the *miss* shader passed to the `TraceRay` call is executed. Data between *raygen* and other shaders can

be exchanged by a pipeline-specific *ray payload* structure.

The BLASes do not just reference existing vertex buffers but are self-contained, i.e. contain all required position data. Therefore, BLASes consume significant amounts of memory and re-using BLASes for multiple instances of the same mesh is desired. Building TLASes and BLASes can be done either on the central processing unit (CPU) or GPU. Since the latter is usually faster, our method exclusively builds acceleration structures on the GPU. While acceleration structures are considered mostly static after building, the graphics APIs offer *update* functionality. Updating an existing acceleration structure is usually faster than completely rebuilding it. However, it still has significant runtime cost even for the smallest changes and imposes additional constraints, such as not allowing a change in the number of primitives. Additionally, after updating an acceleration structure, it might be less efficient in terms of tracing performance compared to a newly built one. Using updates for big geometric changes is therefore discouraged.

For BLAS and TLAS building, the application must provide the hardware with a so-called *scratch buffer*. The required size of this buffer can be queried before recording the build command. BLASes are built directly from vertex and index buffers. The TLAS is built from a so-called *instance buffer*. It holds descriptions of the instances to be present in the TLAS. This description holds the data visible in figure 2.4: The BLAS of the instance, its transform, hit group, hit mask, and various flags controlling culling and opaqueness.

Implementing path tracing on the GPU is not trivial due to the high divergence of paths. One important technique to increase GPU utilization (occupancy) is path regeneration [NHD10]. With hardware accelerated hardware ray tracing, re-occupying already finished threads can be done by the hardware automatically as it is allowed to repack waves on each `TraceRay` intrinsic [GB22]. This is only possible in ray tracing pipelines, applications using ray queries have to manually repack traced rays. But even with ray tracing pipelines, applications can still improve this packing for better execution and data coherency [Mei+20]. Vendor-specific shader intrinsics allow manually passing information for better packing to the hardware [RH22].

A method for building acceleration structures lazily as rays pass through them has been proposed [LL20], using a new ray tracing shader type known as traversal shaders [LLV19]. This approach seems promising but ultimately relies on a shader type that is not yet widely available. Specific ray tracing methods such as volume path tracing benefit from entirely new acceleration structure schemes [Mor+23]. We stick to the acceleration structures already implemented by state of the art GPUs and do not build upon these alternative approaches since using them does not allow vendor-neutral hardware accelerated intersection testing. Instead, we focus on utilizing the acceleration structures usable by hardware efficiently.

Scene culling has a long tradition in various real-time graphics methods [Mül04; BPA16]. We will optimize acceleration structure management by utilizing a novel culling strategy based on light paths that is well-suited for all real-time ray tracing techniques.

3 Method

To achieve the goal of real-time path tracing in state of the art computer games, we describe a path tracer aimed at replacing typical video game renderers. There are many challenges to combining high visual quality with real-time performance. Additional problems arise when the renderer should work for already existing assets and scenes that were created without path tracing in mind. These challenges are described in section 3.1 In section 3.2, we first give an overview of our method. Then, we present our solutions to many technical problems, such as data access, motion vector generation, and some game-specific rendering techniques. Acceleration structure management is a major challenge for hardware accelerated ray tracing in general. We deal with this challenge in section 3.3, starting with our general approach and some crucial optimizations. Afterward, we present light path guided culling (LiPaC), our novel method based on hit feedback obtained during path tracing that reduces BLAS building runtime and memory consumption significantly. We observed vegetation to be a major bottleneck during intersection testing. In section 3.4, a novel method for vegetation ray tracing is presented that is essential to achieve real-time path tracing performance in our method.

3.1 Challenges

3.1.1 Accessing Meshes, Materials, and Textures

Game renderers relying on rasterization often leverage many different graphics pipelines for the different materials in the scene [Pet21]. They can render different meshes using separate index and vertex buffers, possibly even containing data in various formats, can change the shader-accessible bound textures and buffers between draw calls and thus bind respective per-instance, per-mesh, or per-material data. All of this is not easily possible with path tracing. A single pipeline handles all meshes, materials, and textures. It must be able to access all the relevant data.

Another issue is the level of detail (LOD) selection for meshes and textures. Accessing an appropriate LOD is still highly important to avoid aliasing artifacts and improve performance by reducing memory bandwidth. For textures, graphics pipelines usually use pixel shader derivatives to determine the optimal LOD. We do not have this information during ray tracing since we do not have the coherency of rasterization. Mesh LODs are often selected using heuristics such as distance to the camera. With ray tracing, this option is still viable but will cause problems in situations such as curved mirrors that can significantly magnify objects in the distance. We thus need robust strategies to decide which mesh LOD to use for ray tracing.

3.1.2 Vertex Transformation Effects

Another problem involves rendering effects modifying vertex positions, as done in the vertex or mesh shaders in renderers utilizing GPU rasterization. Once we build an acceleration structure on the GPU, it is static and cannot be changed anymore. For animations and skinning, we thus need a separate BLAS per instance of a mesh. In addition, even for static effects such as transforming objects to fit the terrain they are placed on, we will not be able to share BLASes as outlined in section 2.6. Unique BLASes increase memory consumption significantly and require more frequent BLAS rebuilds. Heuristics to optimize BLAS sharing as well as a highly efficient pipeline to transform vertices and build many BLASes per frame are important.

3.1.3 Generating Motion Vectors

Accurate motion vectors are required to accumulate samples across multiple frames, as done during denoising. Motion vectors need to be as accurate as possible to avoid temporal artifacts and improve accumulation. In games, motion vectors are often already used for temporal techniques like TAA. Games usually use 2D motion vectors, allowing reprojecting a pixel on the screen to its previous position. Denoising algorithms like ReBLUR [Zhd21] can leverage 3D motion information to avoid false-positive sample rejections caused by missing depth reprojection.

For path tracing, there are multiple ways to generate motion vectors, depending on whether rasterization is used for primary rays as an optimization. To use denoising techniques such as primary surface replacements [Pan18], generating motion vectors during ray tracing for arbitrary meshes must be possible. But computing the motion vectors in hit shaders is not trivial, especially for animated meshes. Doing so requires the state of the animation in the previous frame. For rasterization, the inputs to animations and transformations from the previous frame are often stored and then the effects are applied in the vertex shader for the current and last input, respectively. The difference in the result can then be used to generate motion vectors. But with ray tracing, we apply skinning and other per-vertex effects during vertex transformation, before BLAS building. Thus, we have no trivially accessible motion vectors in the hit shaders. We will discuss multiple possible ways to obtain accurate motion vectors and compare them.

3.1.4 Maintaining Acceleration Structures

One of the complexities of hardware accelerated ray tracing is the building and maintenance of acceleration structures. They have an inherently high memory consumption. As described above, when meshes are transformed uniquely per instance, unique BLASes are needed per instance. Memory consumption can therefore easily become a limiting factor. For scenes with many thousand uniquely transformed meshes and running animations, optimizing BLAS building and management is of high importance. Since not all meshes and running animations are visible at all times, or might just slightly influence a couple of pixels on the final screen, culling strategies to decide which animated BLASes to rebuild, which to update, and which to ignore, are needed. For static meshes, caching mechanisms that keep the consumed memory reasonable while not causing frame time spikes by building a high number of BLASes when the camera moves through the scene are needed. Culling for ray tracing is harder to implement compared to rasterization since rays can easily bounce outside of the frustum or encounter surfaces occluded from the camera. A valid representation of the scene is still required in these cases. Otherwise, artifacts, missing objects, or paused animations will quickly become visible, for instance in mirrors.

3.1.5 Low-light Situations and Cold Starts

We cannot rely on Monte Carlo integration to converge before we present our path traced approximation since that generally requires a high number of samples. Denoising can often produce visually pleasing results even for path tracing results generated by a small number of samples — possibly as low as one sample per frame per pixel. But there are some problematic situations to solve: When paths barely reach light sources, possibly only after many bounces, the variance of single a sample becomes prohibitive. An example of this situation is a closed, dark room that is barely indirectly lit by small windows. Most sampled paths do not find a light source. In such situations, denoising algorithms produce visually unpleasant images that still contain noise or are visibly overblurred.
Another problematic situation related to a low sample count is a cold start. Denoising algorithms accumulate the samples over many frames, whenever this is possible. When the visible geometry changes quickly, for instance after the camera jumps to a new position, no information can be re-used and even after denoising, the result might look unpleasant. More advanced light transport approaches [Laf95; VG97], path guiding [Rat+20], world-space irradiance caching [Hal+21; Gau22], path reuse [Ouy+21; Lin+22], and adaptive sampling strategies are possible approaches but no definitive solution has been established so far. We do not present a new solution to this problem but the existing approaches can be combined with our presented method.

3.1.6 Vegetation rendering

Traditional vegetation assets are a well-known bottleneck of GPU ray tracing [FO23]. Trees, bushes, and grass in games are often modeled just using a high number of planes with alpha testing to transform them into the form of leaves or grass blades. With rasterization, this leads to massive overdraw with many pixels just being clipped in the pixel shader, potentially already being a bottleneck. Translating this technique to a ray tracing pipeline means executing a high number of anyhit shaders, potentially many hundreds for just a single vegetation asset. This quickly becomes a bottleneck. Some renderers just completely ignore alpha-tested geometry for secondary bounces [Bou23]. This problem is particularly interesting and difficult because the vegetation is usually light translucent to some degree, opening the door for new approximation ideas.

3.2 Path Tracing in Computer Games

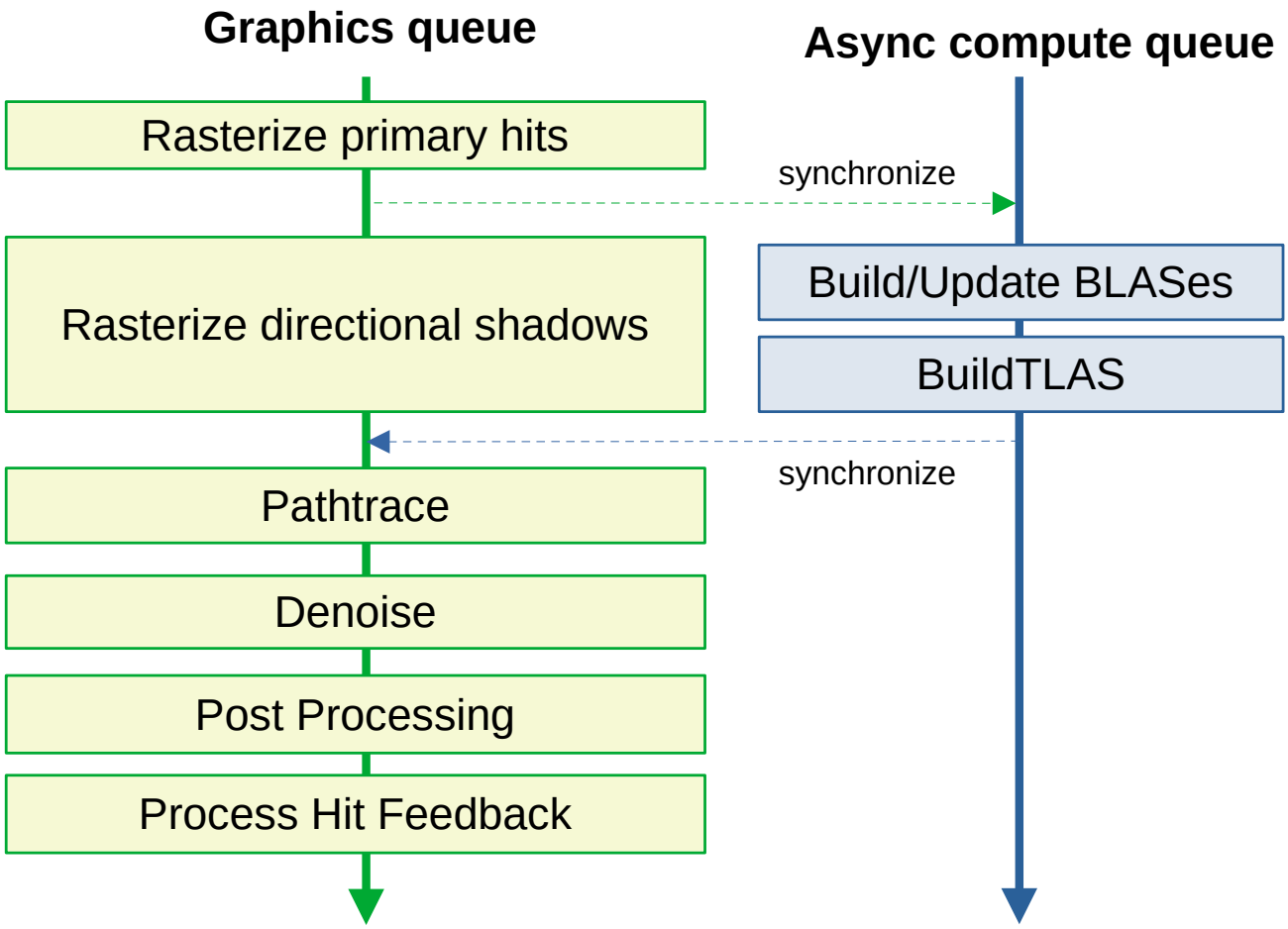

Figure 3.1: Overview of the path tracing pipeline on the GPU. Graphics and asynchronous compute queues execute in parallel to fully utilize the available hardware.

Figure 3.1 provides an overview of our path tracing method. The main motivation for utilizing rasterization for primary hits is vegetation. More details about vegetation handling can be found in section 3.4. Furthermore, we use shadow rasterization for vegetation and instances culled from path tracing by our LiPaC method. In section 3.3, we describe how we build BLASes and the TLAS and why it is executed on the asynchronous compute queue. In section 3.3.2, we describe LiPaC, our method utilizing hit feedback to cull the path traced scene. We will detail what happens in the *Process Hit Feedback* step and how this influences which BLASes are built in the next frame. We do not detail the denoising step in this chapter, as we rely on the publicly available Nvidia ray tracing denoiser (NRD) implementation of ReBLUR [Zhd21].

3.2.1 Hybrid Rendering Pipeline

Our rendering method makes use of hybrid rendering, i.e. mixing rasterization and ray tracing. The path tracing pass generates depth and material information buffers just as a *gbuffer* pass in a deferred renderer does [Pet+16]. This allows utilization of traditional video game effects relying on depth or normal information.

For instance, volumetric effects such as fog and clouds would add considerable overhead to our path tracing and we therefore apply them as a post-effect. Specific game assets that are not part of the physically lit scene, such as outlines or overlay effects, will also be rendered on top of our path traced image.

We rely on rasterization for the primary hits. Traced primary rays are completely convergent and therefore utilize the GPU hardware more efficiently than secondary rays but we found massive performance improvements for scenes with vegetation when replacing primary rays with rasterization, see figure 4.10 for details. In such scenes, even primary rays are much slower to trace than using rasterization due to the high number of *anyhit* shader invocations. In section 3.4, we describe our method of handling vegetation for other types of rays.

Using rasterization for primary hits has the disadvantage that mismatches between rasterized geometry and geometry present in the acceleration structures might create lighting artifacts such as light leaks and incorrect shadowing. On the other hand, we obtain the flexibility to leave details — vertex animation effects or whole geometry — out of our acceleration structures for performance reasons and still get correct primary hits. In particular, this also means we can use different mesh LODs for primary hits and ray tracing, allowing us to fall back on rougher geometry for ray tracing, improving memory consumption and performance. For small geometric details, the indirect lighting errors are barely noticeable, and mismatches on a per-vertex level can often be avoided by offsetting rays. Our LiPaC method presented in section 3.3.2 allows us to completely cull expensive instances from path tracing if they are expected to have little impact on the indirect lighting of the scene. Rasterizing primary hits is important in this case. At the same time, LiPaC ensures that meshes requiring high quality during path tracing (for instance, because they are seen through a mirror) are still included in path tracing in appropriate quality.

3.2.2 Path tracing pipeline

For path tracing, we use a ray tracing pipeline instead of ray queries. While performance differences vary per GPU vendor, ray tracing pipelines allow the GPU to schedule traced rays more efficiently [Jos23]. Additionally, we want to realize a multitude of effects and materials, resulting in hit shaders with high register pressure already. We thus can profit from the divergence optimization mechanisms possible in ray tracing pipelines. As recommended by various GPU vendors [Jos23; DiG22; Sjo20], we trace rays iteratively and not recursively, i.e. we never call `TraceRay` in hit shaders. Instead, the *closesthit* shaders write the material properties and various flags into the payload returned to the *raygen* shader.

Shooting rays in our path from the last encountered hit (c.f. algorithm 1) will lead to light artifacts for smoothly shaded surfaces. The shadowing of a seemingly rounded surface will reveal the shape of its primitives. This artifact is caused by interpolating surface normals to make it appear curved but not using an curved surface for shadow calculation. As a fixed offset does not solve this issue, we compute the actual offset by interpreting the surface as curved along their normals. For a detailed explanation of this method, we refer the reader to the work by Hanika [Han21].

A common optimization to real-time Monte Carlo path tracing is “Russian Roulette” termination. At each bounce, light paths are terminated randomly with a certain probability. The light found on future bounces of the non-terminated light paths is multiplied by the inverse of this probability. As long as this probability is greater than zero, the algorithm remains unbiased. But lower probabilities can lead to significantly faster tracing times, at the cost of introduced variance. Commonly, the probability of termination depends on the accumulated throughput. But with just this factor, highly occluded surfaces receive the same number of bounces on average as surfaces that are directly lit by a strong light source like the sun. In practice, indirect lighting often has very little impact on the final appearance of bright, directly lit surfaces while a high number

of bounces is needed to properly visualize surfaces far away from light. To account for this, our termination heuristic additionally considers the magnitude of the already accumulated light on the current light path. As soon as a well-lit surface is encountered, the probability of continuing this light path significantly drops. In equation (3.1), we use the variables from algorithm 1 to describe the probability p for a light path to be terminated after a given bounce. To determine useful termination probabilities in all kinds of scenes, we multiply the accumulated light by the currently used exposure for displaying the rendered image. In our renderer, this exposure is automatically determined, emulating the human eye's adaption to light [Wro16]. Equation (3.1) has two parameters: Decreasing $\alpha \in \mathbb{R}_{>0}$ leads to earlier path termination in all cases. The parameter $\beta \in \mathbb{R}_{>0}$ controls how significantly the accumulated light is considered. A higher value means earlier termination for light paths that already encountered light. We found values around $\alpha = 0.5, \beta = 10$ to work well. This means that as soon as we encounter light that is as strong as our current exposure (which correlates with the average image brightness), the chance to continue the ray will be multiplied by an additional 0.1.

$$p = \min(1, \frac{\alpha \cdot ||throughput||}{1 + \beta \cdot exposure \cdot ||accumLight||}) \tag{3.1}$$

For surfaces with strong direct lighting, light paths with more than one bounce are rarely considered. Note that this does not mean that indirect lighting on directly lit surfaces is weakened. The termination strategy remains unbiased as we multiply the extended light paths with the inverse of the probability. Instead, the variance of indirect lighting will increase. But when the expected value of light found further down the light path is lower than the light already accumulated, a high variance for the former is acceptable. We observed this to be a good trade-off between tracing performance and visual quality.

3.2.3 Data Access

As outlined in section 3.1.1, we need to be able to access all mesh, material, and texture data in our ray tracing pipeline. To access all our data in ray tracing shaders, we rely on a technique known as *bindless resources*, or *descriptor indexing* in Vulkan [Inc17], for multiple reasons. First and foremost, we support many different materials with an even higher number of distinct textures. All of this texture data needs to be accessible in our ray tracing pipeline. There exist approaches that solve this problem using texture atlases or texture arrays, but they all induce significant limitations and overhead [Kel19]. The drawback of bindless textures is that this technique is not supported on older hardware. But somewhat recent GPUs that support hardware accelerated ray tracing usually support this technique anyway. Secondly, we do not want to restrict ourselves to a single buffer resource for all meshes since that would induce buffer space management issues and overhead. We would have to handle fragmentation issues and find efficient allocation strategies when frequently streaming vertex data of different LODs in and out. This is possible to do efficiently but introduces additional complexity. Instead, we bind a variable-sized array of textures and another variable-sized array for mesh buffers that can be accessed by our shaders. The mesh buffer array contains the index and vertex buffers that we also use for rasterization, thus avoiding memory duplication. We allocate custom vertex and index buffers only for special meshes such as our terrain, as it makes use of custom tessellation not needed for rasterization, see section 3.2.7. They are equally accessed in the hit shaders via the variable-sized array of buffers. An overview of the GPU data structures can be found in figure 3.2.

The TLAS instance ID accessible in the hit shader is used to address a structured buffer holding all *model instances*. Meshes using multiple materials are realized by using multiple geometries inside a single BLAS. Having a separate BLAS for each distinct part of a mesh would result in overlapping BLASes, reducing the

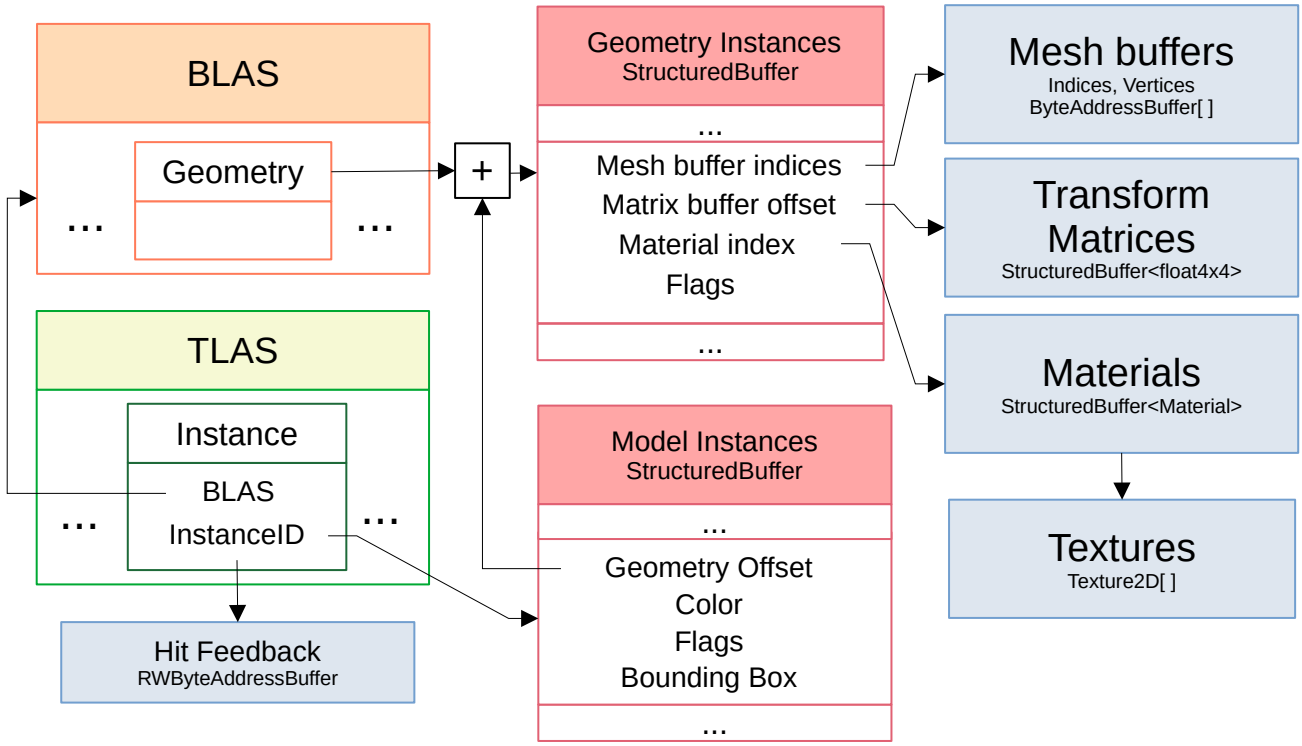

Figure 3.2: An overview of the most important GPU buffers for ray tracing.

efficiency of ray tracing. Furthermore, a higher number of BLASes also has a negative impact on TLAS building time and memory consumption. On the other hand, combining multiple geometries in a single BLAS requires storage of additional data per geometry. The instance data for the whole BLAS thus has a *geometry offset* that marks the start of the range of *geometry instance* data in another structured buffer. The hit shader addresses this buffer using the sum of this *geometry offset* and the index of the intersected BLAS geometry. Per geometry, we store indices into the mesh buffer arrays for indices and vertices, flags enabling/disabling effects, format information about the mesh data, and a material identifier.

This addressing scheme of geometry data means we have to sub-allocate consecutive ranges in the *geometry instance data* buffer. Since this operation is required anytime a geometry instance is added or removed, which also happens on LOD changes, this operation needs to be efficient. There are efficient ways to implement this such as a two-level segregate fit (TLSF) allocator [Mas+04].

3.2.4 Non-opaque materials

As mentioned in section 2.6, geometries in BLASes can be marked as opaque or non-opaque. For non-opaque geometry, *anyhit* shaders are executed. These can ignore hits, for instance after sampling an alpha value for the given hit from a texture. For opaque geometry, this additional shader invocation is skipped, and all hits are treated as valid. Thus, path tracing with opaque geometry is significantly faster, see chapter 4 for more details. Our method should be suitable for existing video game assets that require alpha testing. These assets usually do not specify which parts of the mesh require alpha testing and which can be considered opaque. To keep the number of non-opaque geometries as low as possible, we employ a separate offline processing step per asset that leverages GPU rasterization to quickly compute this information. The mesh is rasterized

into a framebuffer without render targets. The framebuffer has at least the size of the texture containing the alpha information. Each vertex outputs its UV texturing coordinates as the position for rasterization. For each generated pixel, the alpha texture is sampled in the pixel shader. If the alpha value falls below the alpha-testing threshold, the triangle is considered to be non-opaque. This information is written into a buffer that holds one bit per triangle. In practice, an atomic `OR` instruction is used to set the bit associated with the triangle to one. After this rasterization process, the buffer is copied to the CPU, where it is read to split the mesh into opaque and non-opaque triangle groups. Each of these groups will become its own geometry of the BLAS. This approach allows us to decide on a pixel-perfect basis whether a model needs alpha testing and provides a potential speedup without any visual impact.

3.2.5 Texture Mapping

Even with all the texture data accessible by our hit shaders, the problem of sampling and selecting proper texture map LODs remains. We rely on cone tracing [Ake+19] to properly compute texture mip levels in hit shaders as it allows us to compute good approximations of the required LODs without adding much performance overhead. To render large diverse scenes with many textures, we employ texture streaming in our engine, only ever loading as much texture detail into graphics memory as is currently needed for rendering the scene. We utilize feedback from the GPU to determine which mip levels of textures would be needed. We remember the most detailed needed mip level for each texture in a buffer that is updated during rendering. After rendering is finished on the GPU, this buffer can be copied and read on the CPU to determine which mip levels should be streamed in or out. This approach works well for ray tracing: Even for objects not in the frustum, we can ensure proper texture detail without wasting graphics memory. In Direct3D 12, sampler feedback [And19] provides an even more detailed way of getting feedback on sampled texture regions from the GPU to efficiently stream texture atlases.

3.2.6 Velocity vector generation

As described in section 3.1.3, we require motion vectors for denoising. There are multiple approaches to generating motion vectors with path tracing:

- When using rasterization for the primary hit, we can generate velocity vectors using the technique commonly applied in rasterization. The transformation matrices of the previous frame are kept around and all shader-effects that modify positions are applied twice. The difference between the two calculations then yields the velocity. As mentioned above, there are other advantages and disadvantages to generating primary hits via rasterization. Additionally, denoising techniques such as PSR or stable planes [Pan18; Zim+15] require motion vectors at secondary hits. Therefore, a strategy to generate motion vectors also during path tracing is required.
- For animated meshes, compute shaders are dispatched in each frame to update the vertices for BLAS building. In that step, we could generate motion vectors for each vertex, just as we do during rasterization, and interpolate them in the hit shaders. The downside of this is that we need persistent vertex buffers holding motion information for each animated mesh instance, potentially increasing memory consumption significantly.
- We could perform all motion vector computation completely in the hit shaders. This has the drawback of performing potentially complex operations such as skinning in the hit shaders for the current as well as the previous frame. In terms of rasterization terms, this is comparable to doing per-vertex operations in

the pixel shader, which is inefficient since the results are computed potentially many times. The runtime overhead of this approach and whether it is justified by the reduced memory usage compared to the previous approach depends on multiple factors: How complicated are the vertex transformations (for instance, how many bones are affecting each vertex)? How finely tessellated are the meshes and how large is the scene? When there are just a small number of hits per triangle, this approach does not result in a high number of redundant computations. Lastly, one has to consider if this becomes a bottleneck of the hit shaders — for instance, in terms of register pressure — which would then induce an even more significant performance impact.

These approaches can also be mixed, e.g. generating per-vertex motion vectors before tracing rays only for some objects and generating them dynamically in the hit shaders otherwise. Since we rely on rasterization for primary hits, we just generate motion vectors as usual for rasterization-based games during that pass. For most scenarios, where the use of PSR or stable planes remains fairly limited, motion vectors from hit shaders will only be needed in a few cases and the performance impact is acceptable. We therefore do not ever pre-compute motion vectors and purely generate them in the hit shaders, if needed.

3.2.7 Terrain and Tessellation

GPU tessellation is a common video game technique to increase mesh resolution without requiring larger mesh buffers by tessellating meshes as they are being rendered. This technique is especially used for procedural geometry such as terrain, where the actual geometry is often generated just inside vertex/tessellation shaders [YS11] by evaluating noise functions or sampling a heightmap. For ray tracing, we emulate tessellation shaders with compute shaders and generate a more detailed version of the mesh in memory before building the BLAS from it. This leads to high memory usage, the buffers and BLASes for terrain rendering appear significant in our memory evaluation in section 4.3.

3.2.8 Cut-out planes and volumes

Cutouts are one of the effects useful for rasterization but not trivial to implement for ray tracing. This technique allows to procedurally remove sections of terrain or water meshes, for instance, to create objects that seem like they go into the ground without actually deforming the intersected meshes themselves. The effect is achieved by defining cut-out volumes with simple meshes and then rasterizing them in multiple passes. This approach cannot be transferred to ray tracing but it is important to properly handle cut-out sections for all traced rays to avoid significant artifacts for the lighting of surfaces in cut-out regions. So even if ray tracing is not used for primary rays, it is still important to consider these effects for shadow and indirect lighting rays. The problem can be seen as a special case of boolean modeling, for which ray tracing approaches exist [AK21]. Due to the much more specific nature of cutouts, we do not have to implement its full complexity and can thus be significantly more efficient.

A counter in our tracing logic is increased upon hitting the front face of a cutout mesh and decreased upon hitting a cutout backface. The counter is needed since we want to support assets with overlapping cutout volumes and planes for greater flexibility and feature parity with a rasterization implementation. In any case, after hitting a cut-out object, we just continue tracing the ray from that position. When using rasterization for primary hits, we trace an initial ray with just the cut-out objects active in our tracing mask from the camera to our primary hit to initialize this counter. While our counter is greater than one, we do not include the special meshes, such as terrain or water, in our tracing mask.

3.3 Acceleration Structure Management

Modern games can have many thousands of meshes, materials, and animations in the scene. Therefore, efficient strategies for acceleration structure building, updating, caching, and culling are needed. First, section 3.3.1 outlines a pipeline for efficient BLAS building and various important optimizations. Section 3.3.2 describes LiPaC, our novel method to efficiently prioritize and cull the scene used for path tracing. Independently from strategies to manage acceleration structures in large scenes, many practical optimizations are crucial to achieving real-time performance and staying within the memory limits of commodity hardware [Dun19; Sjo20; DiG22; Jos23]:

3.3.1 General Optimizations

Optimizing the BLASes rebuilds by batching them together Batching builds is recommended by hardware vendors and shows considerable speedup since it increases GPU occupancy and avoids pipeline stalls. In practice, this means first running all vertex-transform compute shaders without a memory barrier in between, then having one barrier on all computed data, and then building all BLASes without a barrier or state change in between. We want to maximize the throughput and efficiency of BLAS building since there often are many small BLASes to be built in a single frame. That is why we optimized even our compute shader dispatches for vertex transformation. We pack vertex buffers and transformation operation descriptions into a single descriptor set (using Vulkan terminology) so that the only state change in between dispatch calls is a single push constant.

Building BLASes on an asynchronous compute GPU queue This allows the computation-heavy BLAS building to be executed in parallel to workloads that are limited by other GPU hardware and therefore do not use all computational resources. [HD17; BN21] We overlap BLAS building with the rasterization of the shadow map that is still needed for some post-effects as well as optimizations described below. Rendering the shadow map is known to be a task often limited by rasterization hardware as it usually does not involve computationally heavy shaders.

Recording BLAS building commands in parallel to other work Recording the commands for acceleration structure building can be slow, especially when hundreds or even thousands of BLASes are built. Executing the recording in parallel to other work on the CPU can help to reduce the effective cost, make use of parallel CPU hardware, and avoid frame spikes.

Sharing BLASes between instances where possible In large scenes, there are often many instances of the same mesh. We can leverage this by not building a BLAS per instance but per mesh, or to be more specific, per mesh LOD, where possible. This is only possible for meshes without vertex transformation effects depending on the instance such as skinned meshes with dynamically running animations or meshes adapting to their position on the terrain. For those, we can not share BLASes and describe other optimizations below.

BLAS compaction When building acceleration structures, initially it cannot be known how much space they will require due to the dynamic nature of the building process. Ray tracing graphics application programming interfaces (APIs) will just provide an upper bound for a given input and then offer a way to query the actual size after building is done. By compacting the BLAS, memory consumption is significantly reduced. We only compact static BLASes since all other acceleration structure builds are just used for a single frame, making compaction useless.

Mesh LOD handling Having proper mesh LODs is important for ray tracing to speed up intersection tests and reduce memory consumption. But each mesh LOD needs its own BLAS. Ideally, mesh LOD switches do not require a BLAS rebuild every time since many hundred meshes might change their LOD in a single frame for large, dynamic scenes. We use a hash map to store shared BLASes. Unused shared meshes are evicted as soon as a graphics memory threshold for the BLASes is exceeded or when they have not been used for many frames. This helps to achieve an upper bound estimation of the amount of required graphics memory. For instances that need unique BLASes or if shared BLASes are not present in the cache, we possibly delay LOD switches to avoid frame time spikes even during fast camera movement. In each frame, LOD change requests are only processed up to a certain number of BLAS builds or a number of processed primitives, whatever happens first.

Even with all the optimizations above, large and dynamic scenes are still a challenge. Rebuilding all BLASes of running animations in every frame takes too long, see section 4.2 for more details. Furthermore, in large scenes with many non-static instances that need their unique BLAS, memory consumption is unfeasible, see section 4.3.

3.3.2 Hit Feedback

Our method to path trace large and dynamic scenes in real-time while keeping CPU runtime, GPU runtime, and memory consumption low relies on hit feedback. The general idea of hit feedback is to track how many light paths encounter each model instance during path tracing and to use these counters for decisions during acceleration structure management. While it has been suggested to use hit feedback to determine which animated BLASes to update in a given frame [Mak23], we take the idea one step further, building our entire acceleration structure management process upon hit feedback. Information from hit feedback completely replaces all previous culling techniques known from rasterization. In our method, areas of the scene that are rarely encountered by light paths are not added to the TLAS anymore and instead replaced with bounding boxes that count light paths reaching this area of the scene to potentially activate instances again. In summary, we cull the path traced scene based on the hit feedback provided by the traced light paths. Therefore, we call the method light path guided culling, LiPaC for short.

The main goal of LiPaC is to reduce the number of instances in the TLAS as much as possible. This is important for two reasons. First, a lower number of instances improves performance: It reduces TLAS building times, lowers the number of animated BLASes we have to update each frame, and potentially reduces TLAS traversal time during path tracing. Second, fewer instances mean reduced memory consumption for multiple reasons. The TLAS itself and the TLAS instance buffer become smaller. In addition, the number of unique BLASes we need for animated or otherwise uniquely transformed meshes is reduced. It also allows the freeing of shared BLASes in the cache more quickly when they are not used anymore. At the same time, all model instances that are reached by rays should be present in the TLAS to avoid incorrectly lit scenes.

To gather hit feedback, the number of light paths intersecting an instance is counted in *closesthit* shaders during path tracing. Afterward, the hit counters are spatially accumulated into a coarse spatial grid managed

on the GPU. This acceleration structure was chosen for its simplicity but other choices such as quadtree, octree, k-d tree or bounding volume hierarchy (BVH) are possible as well [Sam84; Mei+21]. The elements in the grid are called hitboxes and accumulate the number of light paths that traverse the associated area. Each hitbox is marked as *active* or *inactive*. For *active* hitboxes, all model instances in their volume are added to the TLAS. *Inactive* hitboxes just add an axis-aligned bounding box (AABB) BLAS instance to the TLAS that accumulates the number of rays passing its volume. As the number of hits a single hitbox encountered passes a certain threshold ϑ_a, it switches to the *active* state. To consider whether a hitbox should be activated again, the accumulated hits from all its model instances are compared against another threshold, ϑ_d.

Figure 3.3: Visualized hit feedback in a big city scene. Left: path traced views of the scene. Right: the instances present in the TLAS for the associated perspective on the left. In the second row, a mirroring cube was placed into the scene. Object color is interpolated from white to red, the higher the hit count.

Figure 3.3 shows a visualization of a scene culled by LiPaC. The right image shows a top-down view of the TLAS state for rendering the respective images on the left. For the simple case in the first row, mainly instances inside the frustum get hit. Only a small number of instances outside, but closely around the frustum are encountered by a relatively high number of rays and therefore added to the TLAS as well. Similar results can be achieved by just activating instances close to the frustum. The second row shows LiPaC working in a much more general way. With a large mirroring block in the scene, instances visible in the mirror are activated while instances occluded by the mirror are not added. In both cases, only a small fraction of the whole scene is considered active.

To realize this method efficiently, we fill the instance buffer used to build the TLAS entirely on the GPU. The TLAS instance buffer building process that is run each frame is outlined in algorithm 4. When the application starts rendering a scene, all hitboxes are initialized to be in the *inactive* state. Their state changes are discussed later on. Furthermore, in each frame, the hitbox height bounds are reset. For each hitbox, $minY$ is set to $+\infty$ while $maxY$ is set to $-\infty$. The loop in the first line is parallelized on the GPU, with one thread per model instance. Each model instance is in one of three possible states:

active The instance has an associated BLAS that is added to the TLAS, see line 7. In practice, appending TLAS instances to the buffer is done via an atomic counter.

omitted The instance is in a region of the scene that is important for lighting the scene. However, the instance itself is not hit by many light paths and it is costly to add to the TLAS, for example, because it is animated and needs its unique BLAS. Therefore, instead of the exact geometry, a bounding box is added to the TLAS. It accumulates the number of encountered light paths to activate the instance when needed.

inactive The instance is in a region of the scene that is not encountered by many light paths, i.e. it is associated with an inactive hitbox. The instance is not added to the BLAS. Instead, the minimum and maximum height of all instances in a hitbox are evaluated in lines 16 and 17.

Hitboxes and instance bounding boxes in the TLAS are realized by having a single, static BLAS that contains the unit cube. The transform matrix of the instance description is used to transform it as needed, as outlined in lines 10 and 23.

Afterward, a GPU thread is dispatched per hitbox, outlined by the loop in line 20. For *inactive* hitboxes, it adds a transformed unit cube instance to the TLAS. The transform will consider the height bounds previously determined by all contained instances. This box instance is needed for inactive hitboxes so that hits reaching the associated area in world space will be registered and can be accumulated. After these steps, we build the TLAS using the newly assembled buffer of instances.

Algorithm 4 Build TLAS Instance Buffer

```
1: for every model instance M do                          ▷ Executed in parallel on GPU
2:     if M.state = active then
3:         X: TLAS instance description                   ▷ See Vulkan or D3D12 instance description specification
4:         X.blas ← M.blas
5:         X.transform ← M.transform
6:         X.hitGroup ← M.hitGroup
7:         Append X to TLAS instance buffer               ▷ Realized via increasing an atomic counter
8:     else if M.state = omitted then
9:         X: TLAS instance description
10:        X.blas ← global static unit cube BLAS
11:        X.hitGroup ← HITGROUP_HITBOX
12:        Set X.transform such that it maps unit cube to M.aabb
13:        Append X to TLAS instance buffer
14:    else if M.state = inactive then
15:        Get hitbox H at position of M
16:        AtomicMin(H.minY, M.aabb.min.y)
17:        AtomicMax(H.maxY, M.aabb.max.y)
18:    end if
19: end for

20: for every hitbox H do                                 ▷ Executed in parallel on GPU
21:    if H.state = active ∧ H.minY < H.maxY then
22:        X: TLAS instance description
23:        X.blas ← global static unit cube BLAS
24:        X.hitGroup ← HITGROUP_HITBOX
25:        Set X.transform such that it maps unit cube to H bounds
26:        Append X to TLAS instance buffer
27:    end if
28: end for
```

During path tracing, whenever a ray hits a model instance or hitbox, the associated hit feedback counter is increased atomically. Besides the hit counter, more information allowing for simple atomic updates can be stored such as whether a light path from the camera hit the instances that did not have a scattered bounce before, i.e. `E(T|S)*` paths. Instances encountered on such light paths, for example, when seen through a mirror or behind glass, require additional detail compared to instances just encountered by light paths after a diffuse or high roughness specular bounce. Such additional information can be used to develop more sophisticated heuristics to decide whether a model instance should be marked as *active*, *inactive*, or *omitted*. Hitboxes have a special *closesthit* shader, indicated by `HITGROUP_HITBOX` in algorithm 4. That shader will register hits and set a bit in the ray payload returned to the *raygen* shader causing this hit to be skipped and the ray continued to be traced along its current direction. In practice, we allow each ray to only hit one hitbox, masking out all other hitbox instances in tracing after the first intersection of a ray with a hitbox.

After path tracing is done, the accumulated hit feedback is processed on the GPU. The relevant functions are outlined in algorithm 5. First, the number of encountered light paths is summed from the active model instances into their spatially associated hitboxes in line 4. The hit counts get reset to zero for each model instance and hitbox at the beginning of each frame. Then, in a second pass, a GPU thread is dispatched for each hitbox. In this pass, the state of the hitbox is reconsidered: If the hitbox is marked as *inactive* but the received number of hits exceeds a threshold ϑ_a it is marked as *active* in line 9. When the hitbox is marked as *active* but the number of hits received for the instances in its volume is below a threshold ϑ_d it is marked *inactive* again, shown in line 11. Making the activation threshold significantly higher (around factor 10) than the deactivation threshold proved a good setup to avoid frequent state cycles. The activation/deactivation logic shown in algorithm 4 is shortened for conciseness. In practice, we use more sophisticated heuristics considering the type of ray and light path. Additionally, we only deactivate hitboxes after the number of hits has been low for several frames.

In a third pass, a GPU thread is dispatched per model instance to consider if the state of this model instance needs to be changed, depending on the state of the associated hitbox and the number of encountered light paths. On state change, an encoded command value is appended to a buffer intended for CPU reading, as seen in lines 17, 25, 31, and 34. Appending concurrently is realized using an atomic counter which is increased for each added command, similar to how we add TLAS instances to the TLAS instance buffer in algorithm 4. The buffer holding the commands is afterward copied and processed on the CPU. For activated model instances, it is ensured that they have a valid BLAS associated. For model instances that are deactivated, associated resources such as unique BLASes are deallocated. As mentioned above, model instances with a unique BLAS are handled separately. When the associated hitbox is activated, they are put into the *omitted* state first, as seen in line 23. It is only activated when the bounding box added to the TLAS (see algorithm 4 line 13) encounters a number of light paths above a threshold σ_a, as shown in line 32. As soon as the *active* model instance is encountered by a small number of rays below σ_o again, it will be put into the *omitted* state again, as shown in line 35. The *remove* command written out in line 34 will make sure to free the unique BLAS. Once again, in practice, we choose the activation threshold to be around factor 10 higher than the deactivation threshold and consider additional heuristics for activation.

While this method of omitting costly model instances allows us to put a limit on the amount of consumed memory and BLASes that need to be updated or rebuilt each frame, it impacts lighting quality by omitting scene elements. Thus, the method can be seen as a realization of the trade-off between visual quality on the one hand, and memory/performance constraints on the other one. Instances encountered by a small number of light paths, such as animated models in the distance, are not crucial for the indirect lighting of the scene but have a significant cost, making this trade-off reasonable in the context of real-time path tracing. With the hybrid pipeline detailed in section 3.2, we can ensure that omitted elements indeed only impact indirect lighting. Using rasterization for primary hits, they will still appear in the final image. Omitted models are

Algorithm 5 Process Hit Feeback

for each model instance M **do** ▷ *Executed in parallel on GPU*
  if M.state = added $\vee$ M.state = omitted **then**
    Get hitbox H at position of M
    AtomicAdd(H.hitcount, M.hitcount)
  end if
end for

for each hitbox H **do** ▷ *Executed in parallel on GPU*
  if H.state = inactive $\wedge$ H.hits $> \vartheta_a$ **then**
    H.state $\leftarrow$ active
  else if H.state = active $\wedge$ H.hits $< \vartheta_d$ **then**
    H.state $\leftarrow$ inactive
  end if
end for

for each model instance M **do** ▷ *Executed in parallel on GPU*
  Get hitbox H at position of M
  if M.state = added $\wedge$ H.state = inactive **then**
    Append $remove(M)$ command to readback buffer ▷ *Realized via increasing an atomic counter*
    M.state $\leftarrow$ removed
  else if M.state = omitted $\wedge$ H.state = inactive **then**
    M.state $\leftarrow$ removed
  else if M.state = removed $\wedge$ H.state = active **then**
    if M needs a unique BLAS **then**
      M.state $\leftarrow$ omitted
    else
      Add $add(M)$ command to readback buffer
      M.state $\leftarrow$ added
    end if
  end if
  if H needs unique BLAS **then**
    if M.state = omitted $\wedge$ M.hitCount $> \sigma_a$ **then**
      Add $add(M)$ command to readback buffer
      M.state $\leftarrow$ added
    else if M.state = added $\wedge$ M.hitCount $< \sigma_o$ **then**
      Add $remove(M)$ command to readback buffer
      M.state $\leftarrow$ omitted
    end if
  end if
end for

also rasterized into a shadow map that is sampled during path tracing in addition to shadowing rays. This hybrid shadow technique is also used by the method described in section 3.4. While the rasterized shadow is of lower quality than a path traced shadow and only possible for simple light sources, it ensures a visual quality baseline. More sophisticated heuristics could be used to consider how much omitting a model instance will truly impact the final appearance of the scene, based on scene and light path information.

The method is explicitly designed to be GPU-driven. This has a major advantage: No iteration over all (visible) model instances in a scene is ever needed on the CPU side. This allows us to have many millions of model instances. On the GPU side, iterating over all instances in the scene, no matter if visible or not is no

performance concern in practice. See chapter 4 for more details on the performance of these dispatches.

One problem of the feedback approach is the latency between detecting that instances should be active and having them influence the lighting of the final image. The GPU-driven nature of our method amplifies this problem. The readback buffer containing which instances to add or remove can only be read on the CPU side later when execution on the GPU is known to have finished. For instance, when the camera suddenly jumps to a new position, no instance at this position might be added to the TLAS. In the first that is frame path traced at this new position, all hitboxes will receive high numbers of hits and thus activate all of their model instances. While this means that instances could already be active in the next frame, their associated BLASes might not have been built yet. In that case, they will only become visible after the readback buffer has been processed on the CPU and their BLASes have been created. This will potentially cause instances to appear with a delay of multiple frames after the camera position, viewing direction, or elements in the scene change.

An extension of our method solves this worst case: On the CPU we ensure each frame that all instances newly appearing in the frustum have an associated BLAS and are marked as active. This check also covers newly created instances in the frustum. With this, we do not have any latency at all for objects directly in the view frustum. On the other hand, significant camera changes might lead to spikes in the number of active instances and BLAS builds. Thus, an additional heuristic based on instance distance to camera and size selects only instances considered crucial for indirect lighting. Furthermore, the visible latency of several frames still persists for appearing instances that are not in the frustum and are only viewed through mirroring or refracting surfaces.

The tremendous impact on performance and memory consumption our BLAS management method achieves can be seen in chapter 4.

3.4 Efficient Ray Tracing of Vegetation Assets

Vegetation in games is often modeled using many planar quads and alpha testing to avoid excessive mesh complexity. In rasterization, alpha testing works by sampling a texture in the pixel shader and discarding the pixel if the alpha value is below a certain threshold. In ray tracing, this can be accomplished using *anyhit* shaders which sample the respective texture and disregard hits where the alpha value is below the threshold.

Vegetation assets often consist of many hundreds of these planes, requiring just as many separate anyhit shader invocations for a single traced ray in the worst case. The same issue also happens with rasterization, since the high amount of overdraw and pixel shaders being executed is a bottleneck. However, the constant performance factor for each shader invocation during ray traversal is even higher for ray tracing. Additionally, we have the same problem potentially for every single bounce during path tracing. Furthermore, for textures with a lot of small, high-frequency details, neighboring rays will often not end up hitting the same plane, resulting in high ray divergence. All these factors lead to alpha-tested geometry, especially vegetation, often being a limiting factor in ray tracing.

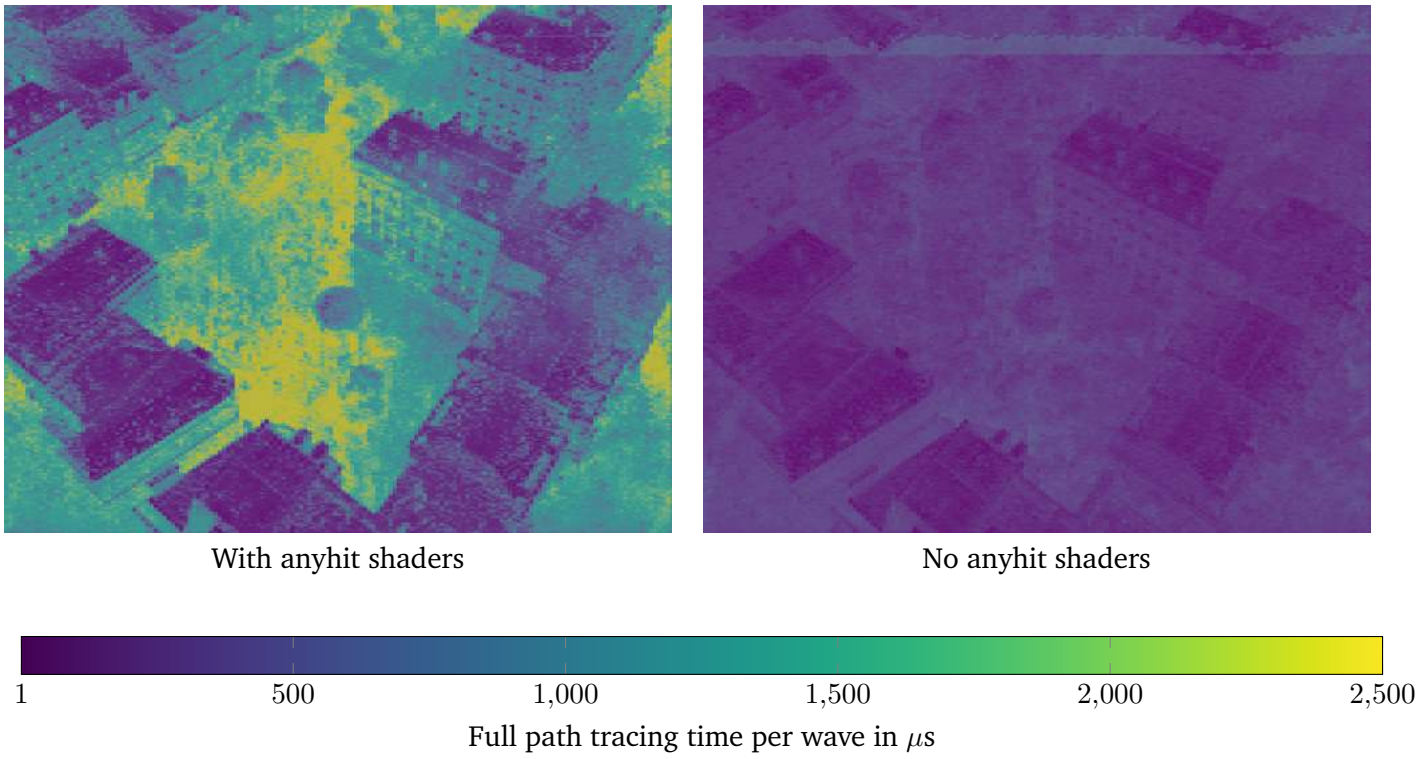

Figure 3.4: GPU path tracing timings for a vegetation-heavy scene with and without anyhit shaders.

As seen in figure 3.4, disabling alpha shading via anyhit shaders significantly speeds up path tracing. Unfortunately, even if we only disable it for secondary bounces, where approximations often have a less significant impact, vegetation will block too much light. The degraded visual quality is visualized in figure 3.5.

One approach aiming to improve performance without loss of visual quality is opacity micromaps [FO23]. This is achieved by encoding the alpha information into the BLAS using microtriangles. Besides this technology not being widely available in GPUs, it has the conceptual drawback of increasing memory consumption. Tessellating the mesh to more closely match the visible parts of the texture has the same drawback.

With anyhit shaders

No anyhit shaders

Figure 3.5: Demodulated indirect diffuse lighting comparison with and without anyhit shaders.

Instead, our approach aims to use heuristics within hit testing itself. We still want to leverage the fact that secondary bounces after diffuse or high-roughness surface interactions need less accurate lighting information for single rays and therefore less accurate alpha testing. The only important property is that integrals over hemispheres remain good approximations. Additionally, we observed that the visual quality of vegetation assets sometimes even suffered from correct light transport computation, as this can make the approximative geometry (of just planes) more visible. With rasterization, renderers often use additional techniques for lighting calculations to hide these problems.

Our approach applies a heuristic based on asset and ray to stochastically skip the ray through the entire vegetation asset, thus avoiding additional anyhit shader invocations altogether, as sketched out in figure 3.6. Vegetation instances do not have anyhit shaders activated. In the closesthit shader, we stochastically decide to either reflect the ray or let it skip through the **entire** object for hits that should have been ignored by alpha testing. An anyhit shader would only be able to ignore one specific intersection but cannot skip the ray through the entire instance. For rays that should pass through the object, we manually do an intersection of the ray with the bounding box and continue path tracing where it leaves. We do not use the AABB itself in our acceleration structure, even though intersection with them can be accelerated on GPUs as well. By using the actual geometry and just handling transparency differently, we stay closer to the actual shape of the vegetation asset.

We generate a per-hit random number and then compare that against a threshold generated by a heuristic. There are multiple ideas for heuristics to use. The most trivial heuristic is to just choose a fixed number as the threshold. That number could be configured per asset, possibly even per viewing direction. Another idea is to consider how far the ray needs to travel through the assets bounding box before reaching the other side. For some tree assets, we got good results with a heuristic considering how close to the center of the asset the remaining ray through the bounding box gets. But no single heuristic worked well for all of our assets and cases. Figure 3.7 visualizes intersections with a tree asset using our method. It shows how the shape and the blocked light of the tree are approximated by the heuristic while avoiding *anyhit* shaders.

A rough outline of our algorithm can be found in algorithm 6. The *raygen* shader attempts to trace the ray

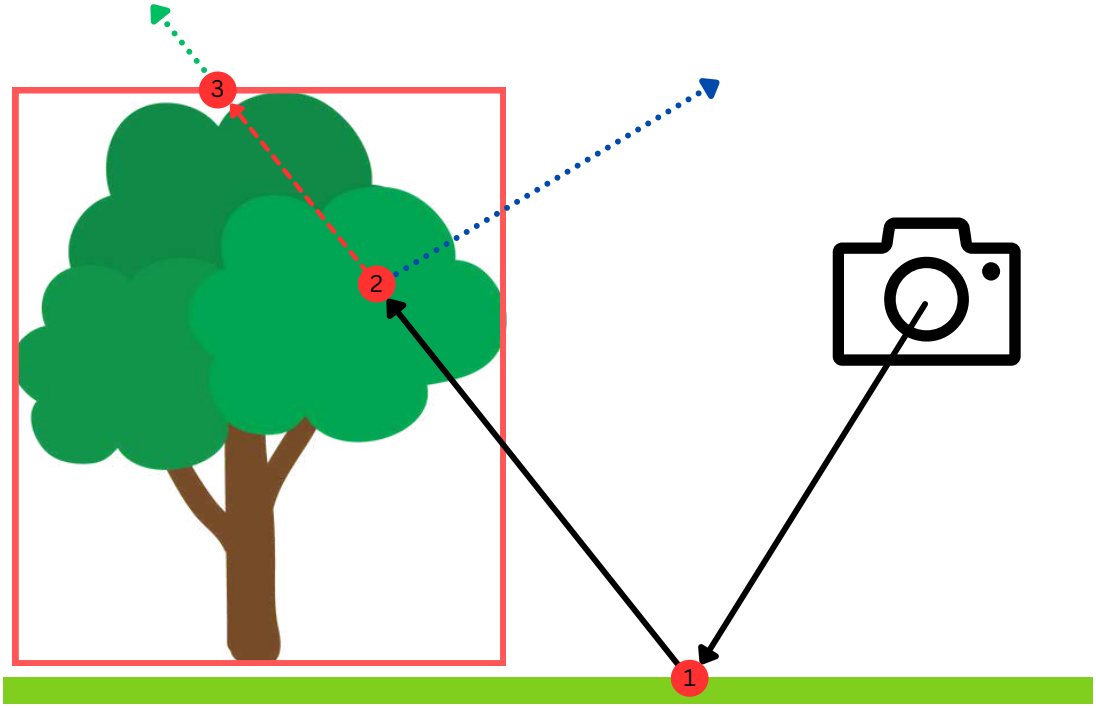

Figure 3.6: Our bounding-box approach to tracing scattered rays through vegetation. At the first red dot, the light path encounters a diffuse or high-roughness bounce. At the second red dot, it hits a transparent part of a vegetation instance. The associated bounding box is outlined in red. First, the intersection point of the current ray with the bounding box is computed, as indicated by the third red dot. A heuristic decides if the light path should skip through this instance completely, in which case the path is continued with the green ray. Otherwise, the hit is considered valid as if the geometry was opaque. The light path continues with a random bounce, as visualized by the blue ray.

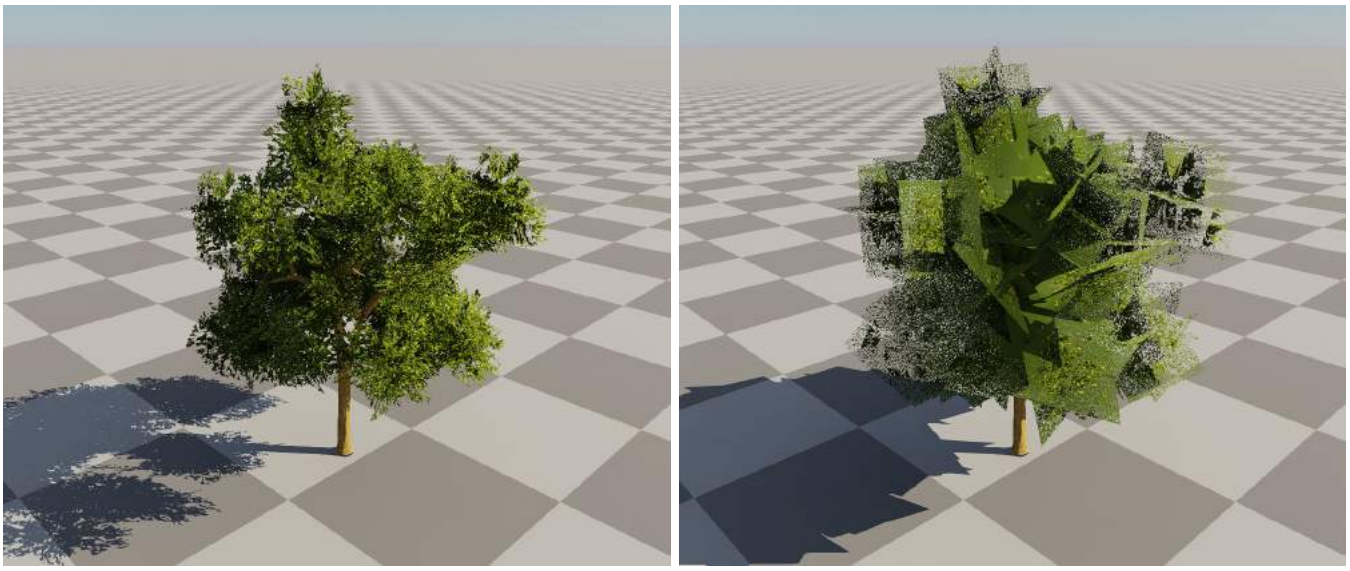

Figure 3.7: Left: typical tree asset rendering. Right: the same tree asset with our heuristic applied. Note that our heuristic is never used for primary or shadow rays, this is a visualization showing the representation of the asset used for highly scattered indirect lighting bounces.

Algorithm 6 Vegetation heuristic for stochastic ray skipping

```
1: function chsVegetation(ray payload p)                    ▷ Closesthit shader for vegetation instances
2:     Compute material of instance, sample alpha
3:     Increase hit count for instance                      ▷ Write hit feedback for LiPaC
4:     if payload.allowSkip ∧ alpha below threshold then
5:         Get axis-aligned bounding box aabb for instance
6:         Compute E, second intersection of ray with aabb
7:         Compute heuristic threshold σ depending on aabb, alpha, ray, E
8:         if random value < σ ∨ current primitive is invisible then
9:             p.skip ← true
10:            p.endpos ← E
11:            Set p.hitidst to intersection distance of traced ray   ▷ Available via shader intrinsic
12:            return
13:        end if
14:    else
15:        p.hit ← true
16:        Write material and intersection parameters to p
17:    end if
18: end function

19: Let p be the ray payload, for data exchange between shaders
20: for each ray to trace do                                ▷ Not for shadow rays
21:    Let origin, dir describe the ray and dist be the maximum ray distance
22:    Set p.allowSkip to true if light path encountered diffuse or high roughness bounce
23:    TraceRay(origin, dir, dist, p, mask: all instances)   ▷ Calls chsVegetation when closest hit is vegetation
24:    if p.skip then                                       ▷ Should ray be skipped through vegetation asset?
25:        end ← p.endpos
26:        origin ← origin + p.hitdist · dir
27:        rd ← ||end − origin||                            ▷ Maximum trace distance of retrace ray
28:        TraceRay(origin, dir, rd, p, mask: all instances without vegetation)   ▷ Does not hit vegetation
29:        if ¬p.hit then
30:            p.allowSkip ← false                          ▷ Allow just one vegetation skip per ray
31:            TraceRay(end, dir, dist, p, mask: all instances)   ▷ Might call chsVegetation but ray skipping disabled
32:        end if
33:    end if
34:    if p.hit then
35:        process hit
36:    end if
37: end for
```

defined by $origin, dir$, with a maximum traced distance of $dist$ and handles retraces needed for vegetation. The pseudocode intrinsic for TraceRay takes the origin, direction, and maximum distance of the traced ray, the payload for data sharing between shaders as well as a mask describing which types of instances to consider. When the closest hit of the ray with the scene is an instance marked as vegetation, chsVegetation will be invoked. If vegetation skipping is enabled and the hit is not valid, i.e. the sampled alpha value is below the alpha testing threshold, it computes the heuristic threshold σ in line 7. It has access to the ray direction and origin via GPU shader intrinsics. If the ray should be skipped through the vegetation, this is marked via the $skip$ bit in the ray payload. In addition, the second intersection from ray and AABB is passed back to the *raygen* shader via the payload as well. In the *raygen* shader, we check for the $skip$ bit in line 24. In line 31, after a vegetation instance was skipped and tracing is continued outside of its bounding box, vegetation

skipping is disabled for the rest of the ray. This is an explicit design decision to limit the performance impact of our method. After the first skip, geometry is just treated as opaque.

When we decide that our ray should pass through the object and continue tracing on the other side of the bounding box, we might miss intersections with other geometry inside of the bounding box. This can lead to visible light leaking artifacts. Ideally, we would trace the ray for the skipped distance again, just masking out the handled vegetation asset but current ray tracing hardware does not allow masking out specific instances efficiently — this is usually done via anyhit shaders and that is what we aim to avoid in the first place. We use one instance mask bit for vegetation, allowing us to mask out all vegetation and retrace the ray through the bounding box against solid geometry. This solves the artifacts we encountered. In algorithm 6, this can be seen in line 29. Only the skipped section of the ray is retraced without considering vegetation.

Even with secondary diffuse bounces solved, there are still other rays we trace. Shadow rays will cause vegetation to remain a bottleneck. Shadow rays executed for shading in primary hits after a diffuse bounce could be handled with a similar heuristic. But we cannot use the heuristic for bounces or shadow rays in a ES* paths, e.g. for shading the primary hit or bounces after a mirror, since we need correct shadows and visibility in that case. To solve this bottleneck, we fall back to rasterization for vegetation shadow, where needed and possible. Vegetation usually involves just comparably small objects, throwing small shadows. The advantages of path traced shadows are often not significant for vegetation. Detailed comparisons can be found in section 4.4.

Another difficulty are effects that cull individual primitives. For instance, using rasterization, the vertex shader could remove grass blades where they would intersect with procedural or user-placed geometry. For path tracing, these conditions are moved to the anyhit shader, ignoring hits where primitives should be invisible. This conflicts with our approach: If we execute the checks and ignore the hit where appropriate (instead of our heuristic) we will potentially have to execute many anyhit shaders again. In our evaluation cases, we found situations where large amounts of primitives are culled like this, thus significantly impacting performance. If we instead just always apply our heuristic, we might get a hit on a primitive that should have been culled. Our solution is to slightly modify our heuristic. We still choose the first hit, avoiding the cost of anyhit shaders. But then we check the culling conditions in the closest-hit shader and once again skip through the entire bounding box if the primitive is culled. This means we potentially cull too much but we found this an acceptable trade-off. In algorithm 6, this additional skipping condition can be found in line 8.

4 Evaluation

Integrating path tracing in a game can mean a new degree of visual quality. However, it can have a severe impact on CPU and GPU frame time as well as memory requirements. We evaluate these criteria in various game-related rendering scenarios and provide timings and memory consumption for various GPUs. First, we give the reasoning behind our evaluation scenarios and present the employed hardware setups. Section 4.2 then describes performance evaluations for various scenes and setups while section 4.3 details the memory consumption of our method. Section 4.4 examines the produced visual detail and compares it to alternative lighting strategies.

4.1 Evaluation Scenarios

As a first test scenario, we want to analyze known computer graphics benchmarks with high-quality textures and meshes, but limited scene size. Here, we focus on visual quality, testing the Sponza [Int22] scene including the curtains and ivy additional packages. The scene is completely static and contains a total of 6.4 million vertices and 10.8 million triangle primitives. We want to evaluate noise and convergence rates in difficult lighting situations that do not occur naturally for outside scenarios.

To answer our hypothesis, we want to evaluate our methods with a rendering engine used for real current generation games, preferably in a generic case easily transferable to other game scenarios. The *Anno* [Ubi23] game series is a good candidate for multiple reasons. The game series allows players to build their own cities and economies. It is known for the resulting complexity of rendered scenes, consisting of potentially large user-generated cities. The rendering engine therefore has to handle these completely dynamic scenes efficiently. Users can build or destroy many buildings with a single click. No scene information or upper bounds on the number of objects is known statically in contrast to other game genres with static levels. In *Anno 1800*, large cities bustle with life: Workers can be seen following their daily business, industry buildings can be seen working their machines, large amounts of residents and carts roam the streets, wildlife is running through the forest, flying through the air and swimming through the ocean. Large game scenes can contain more than 5000 animated instances in any given frame. On the other hand, the game is also very vegetation-heavy outside of its cities. Grass is spawned dynamically all over the terrain, and islands are full of trees, bushes, and entire forests. This puts our vegetation rendering methods to the test. *Anno* furthermore allows arbitrary camera movements and jumps. This forces us to handle cold starts well — situations in which little or no previous lighting information can be re-used. Furthermore, the arbitrary and quick movement and zooming of the camera will result in a high number of LOD switches, putting our acceleration structure management system under pressure.

We measure memory consumption and frame timings in various settings and camera perspectives of the game. For runtime performance, we gather CPU and GPU timings in those settings and investigate scenes with many

	GPU	CPU	RAM	OS	C++ Compiler
Setup 1	Nvidia RTX 3090	AMD Ryzen 1600	16 GB	Linux 6.6.6	GCC 13.2.1
Setup 2	Intel Arc A770	AMD Ryzen 5900x	64 GB	Windows 10	MSVC 14.36
Setup 3	AMD RX 7900 XTX	AMD Ryzen 5900x	64 GB	Windows 10	MSVC 14.36
Setup 4	Nvidia RTX 2080	AMD Ryzen 5900x	64 GB	Windows 10	MSVC 14.36
Setup 5	Nvidia RTX 4090	AMD Ryzen 5900x	64 GB	Windows 10	MSVC 14.36

Table 4.1: The evaluation setups referenced throughout the performance evaluation sections

running animations and a moving camera, inducing many LOD changes every frame. Visual details will be evaluated especially regarding our approximations and path tracing optimizations.

The setups of the machines used for evaluation can be found in table 4.1. It includes GPUs from three major ray tracing capable hardware vendors. With the *Nvidia RTX 4090* and the *AMD RX 7900 XTX*, we evaluate the most capable of current generation consumer GPUs. With the *Nvidia RTX 2080*, we include a GPU that is more than five years old to investigate whether path tracing can also be considered on older hardware. The renderer is written in C++. We implement our path tracing method for the Direct3D 12 DXR [Wym+18] and Vulkan [HBW20] APIs. The used API is stated for each evaluation scenario.

4.2 Rendering Performance

Evaluation Method Discussing GPU runtime performance, analyzing bottlenecks, and optimizing the runtime of hardware accelerated ray tracing can be challenging since all tracing is capsuled into a single monolithic command (`vkCmdTraceRays` for Vulkan and `DispatchRays` for Direct3D). To get more fine-grained information to reason with, we use in-shader profiling via shader timestamps. This is possible via vendor-specific APIs [MS20] or using the `VK_KHR_shader_clock` extension in Vulkan [Inc19]. With this extension, we can evaluate the time needed to find ray intersections, execute specific hit shaders, or even just to execute specific code blocks. We average the timings over 10 frames to avoid their inherent noise and visualize them into heatmaps. These images appear more low-resolution due to timings being coherent between entire GPU waves but are generated at full rendering resolution. When we measure specific timings on the GPU, we avoid executing the workload on the asynchronous compute queue. Timings from this queue would depend strongly on the independent work executed concurrently on the graphics queue and therefore not be meaningful. Only when we measure full frame times will we execute acceleration structure building on the asynchronous compute queue. We measure CPU times using `std::chrono::steady_clock`. GPU timings for entire operations are obtained using timestamp queries available in Vulkan and Direct3D 12. We execute barriers before measuring the first timestamp and after measuring the second one to avoid other work overlapping our timed section. For timing measurements of static scenes, values are averaged over 100 frames.

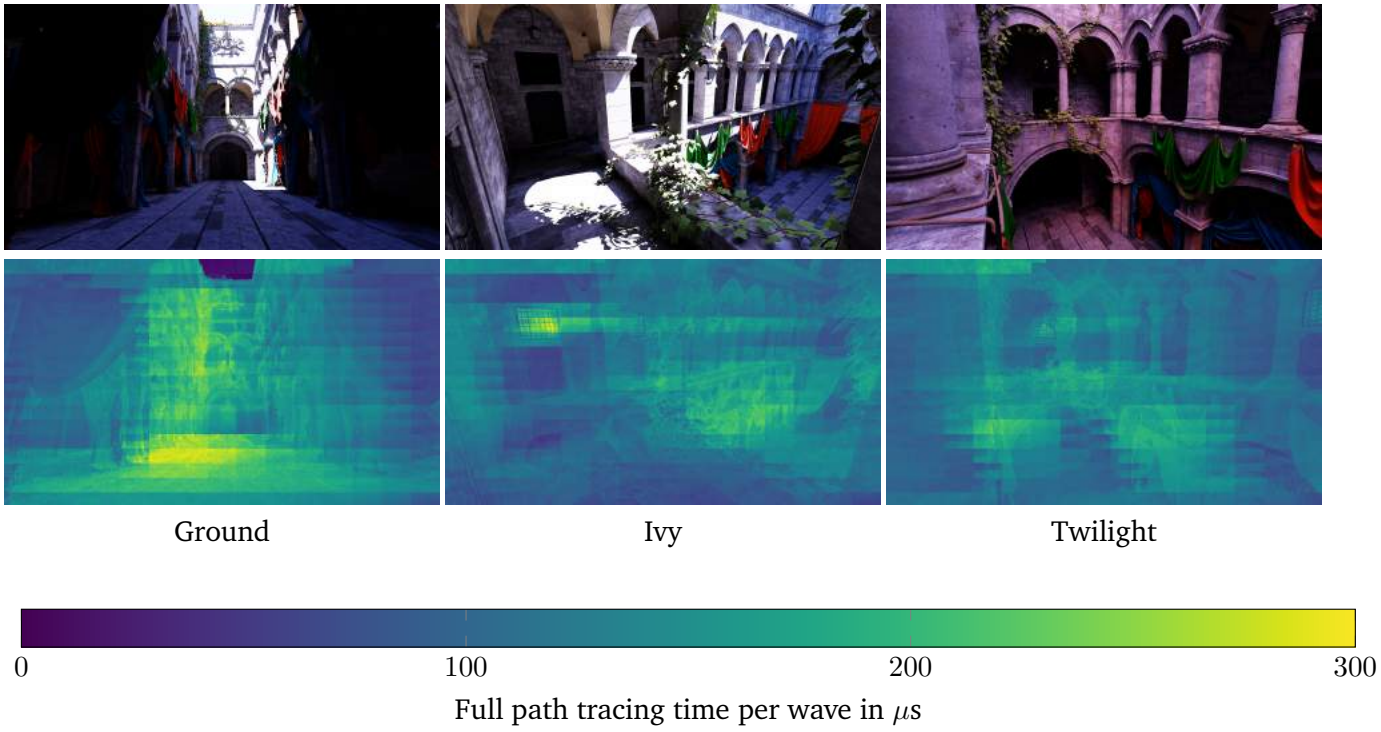

Figure 4.1: The evaluated views of the Sponza scene and their corresponding path tracing timing heatmaps for setup 1, Vulkan, 1080p.

Figure 4.1 shows the evaluated views of the Sponza scene and their GPU path tracing timing heatmaps. The corresponding path tracing and denoising timings for the entire frame are visualized in figure 4.2. The GPU frame time is determined mainly by these two passes since we do not employ any rasterization in this

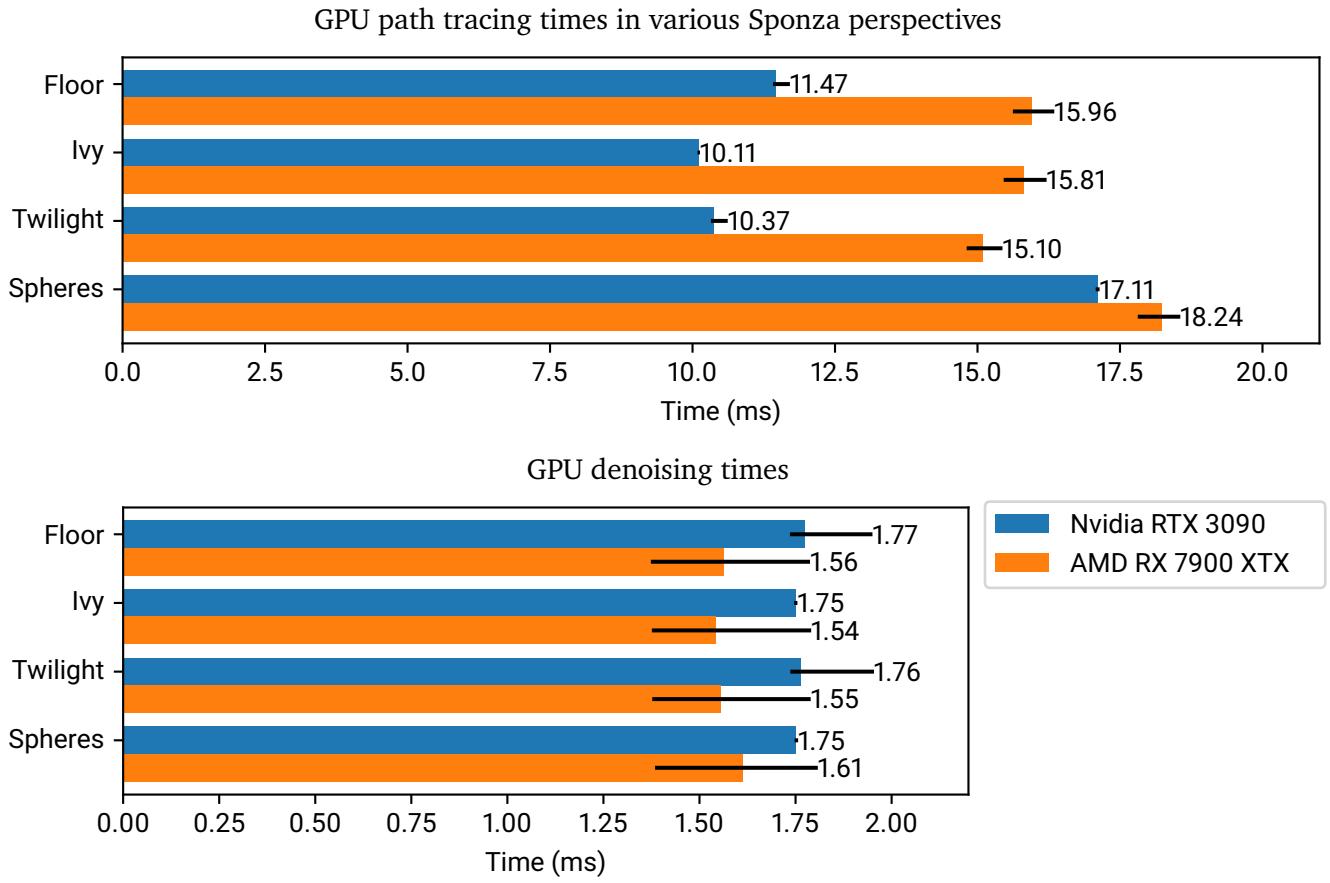

Figure 4.2: GPU path tracing and denoising timings for the Sponza views from figures 4.1 and 4.3. The bars show the timing mean while the black lines show the entire range of obtained timings. The figure compares setup 1 and 3, Vulkan, 1080p.

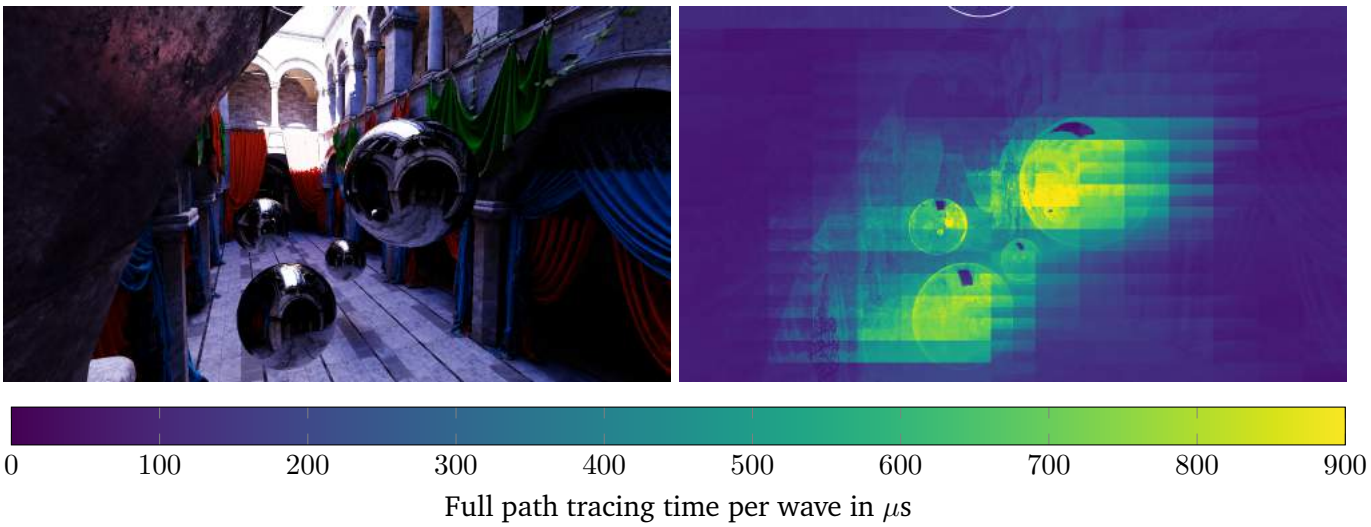

Figure 4.3: Path traced view of the Sponza scene with added mirroring spheres. The corresponding GPU timing heatmap on the right is obtained with setup 1, Vulkan, 1080p.

evaluation scenario. The primary motivation for using rasterization was to speed up vegetation path tracing and to obtain shadows for instances that are culled from path tracing. Both optimizations do not apply to the Sponza scene. It does not contain vegetation with massive amounts of overdraw and culling is not important due to the limited size of the scene. To evaluate path tracing of reflections and our implementation of PSR [Pan18], we add mirroring spheres to the Sponza scene in the *Spheres* evaluation scenario. A path traced image of the scene and the corresponding GPU timing heatmap is shown in figure 4.3. It shows that the additional specular bounces needed for PSR significantly increase path tracing time. However, this method allows for perfectly denoised reflections even on curved surfaces and the timings stay well within our real-time constraints. The timing heatmaps for the other viewpoints reveal divergence due to geometric discontinuities to be one limiting factor for ray tracing. The timings in figure 4.2 show that there are differences in path tracing times between different viewpoints of the same scene. At the same time, they exhibit an acceptable timing variance for a fixed point of view, even though bounce types and directions are selected randomly. This is important to avoid significant frame timing spikes. The denoising times stay consistent across perspectives. Building the BLASes for all meshes in the scene on Setup 1 took 39.4ms on average with a standard deviation of 1.9ms. Building the TLAS took 0.15ms on average with a standard deviation of 0.03ms. This already shows that even for detailed scenes with many primitives, acceleration structures can be built quickly on modern hardware.

To further investigate the GPU runtime performance of our method, we gathered data from many scenes of the game *Anno 1800* [Ubi23] on different GPUs. An overview of path tracing times on different GPUs in a typical Anno city scene with added vegetation can be seen in figure 4.4. A rendered image of the scene is shown in figure 4.11. The high-end GPUs execute path tracing fast enough for real-time framerates. In our tests, the AMD card could achieve real-time framerates even for a 1440p resolution while the Nvidia RTX 4090 achieved this even for a 2160p (4k). The older and less powerful GPUs need more than 15ms for path tracing alone at 1080p resolution. Together with our rasterization passes and other gameplay-relevant processing and rendering done on the GPU, the total frame time for these GPUs exceeds 30ms. However, reducing the resolution for indirect lighting and computing it for just one out of four pixels in a given frame allowed these GPUs to consistently achieve real-time framerates as well.

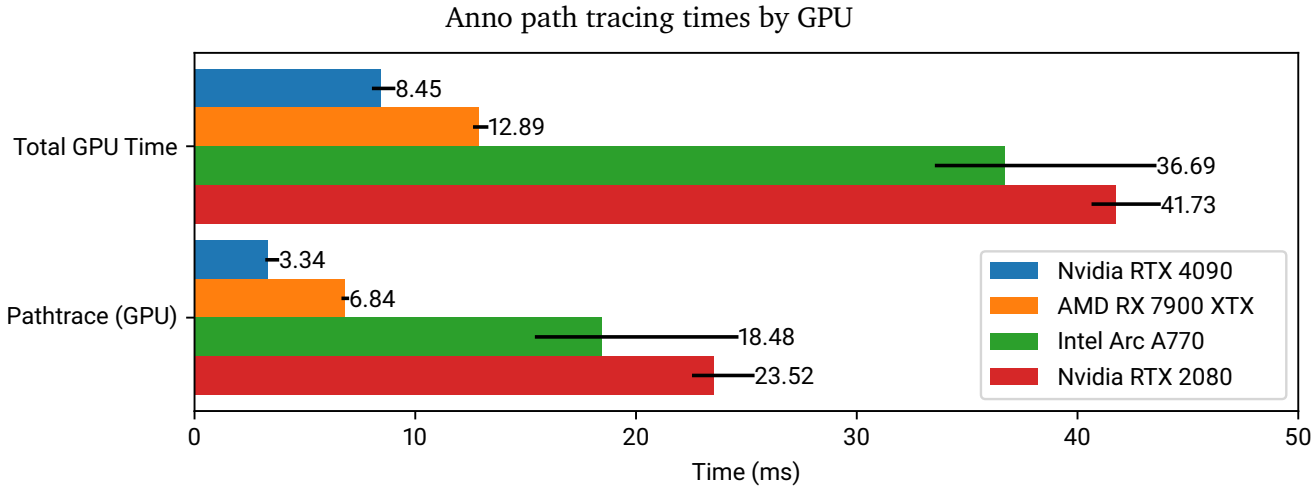

Figure 4.4: Comparison of path tracing and total frame times in a typical Anno scene by GPU. This compares setups 2, 3, 4, 5 at 1080p using Direct3D.

Figure 4.5 compares GPU rendering times in various demanding *Anno* scenes. The scenes are displayed in figure 4.6. In all scenes and viewpoints, path tracing dominates overall frame timings on the GPU. Timings

for other relevant work on the GPU can be found in figure 4.7. The pass rasterizing primary hits into deferred renderers' *gbuffers* is called *primary hits*. The *shadow* pass rasterizes shadows from the directional light source for vegetation and model instances not included in the TLAS by our culling mechanism. The findings can be explained per scene:

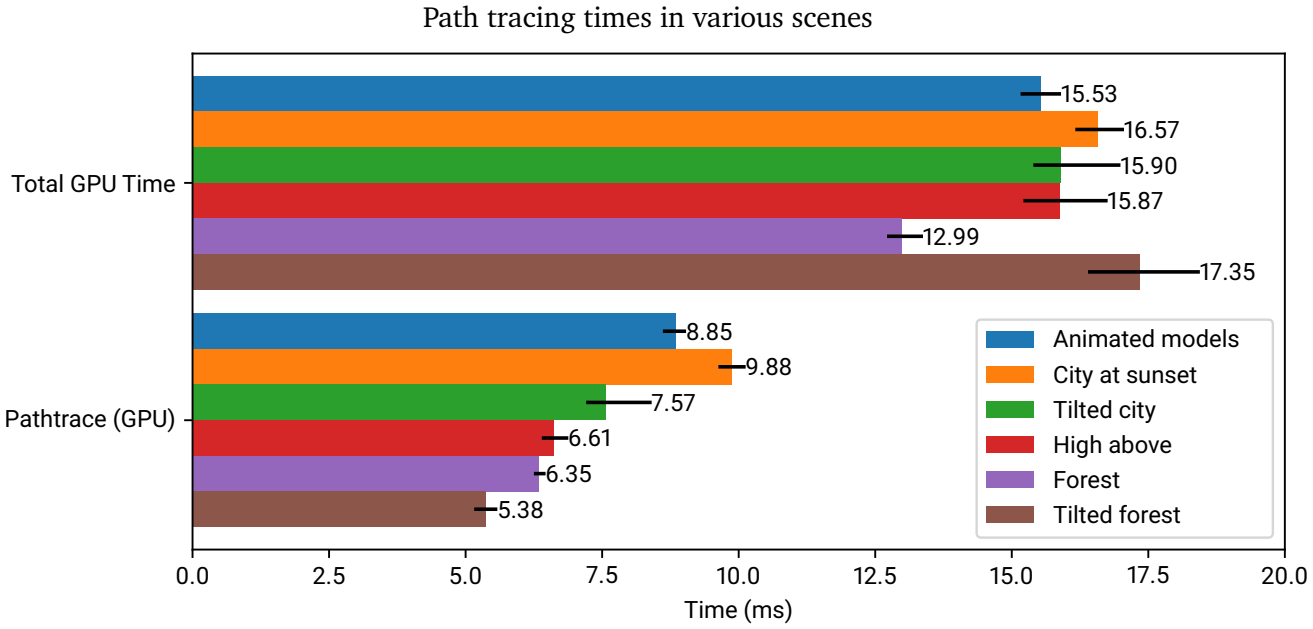

Figure 4.5: Total GPU timings of Anno in a variety of scenes and the portion of it required by the path tracing pipeline. Detailed timings for other GPU work can be found in figure 4.7. Setup 3, Direct3D, 1080p.

Figure 4.6: The scenes used for GPU timings in figures 4.5 and 4.7.

Animated models shows a city scene with a high number of animated units on the streets. The camera is placed so close to the animated crowd that many of them get included in the TLAS by LiPaC. This explains the measured time for the *Build BLASes* section in figure 4.7 being higher compared to all other

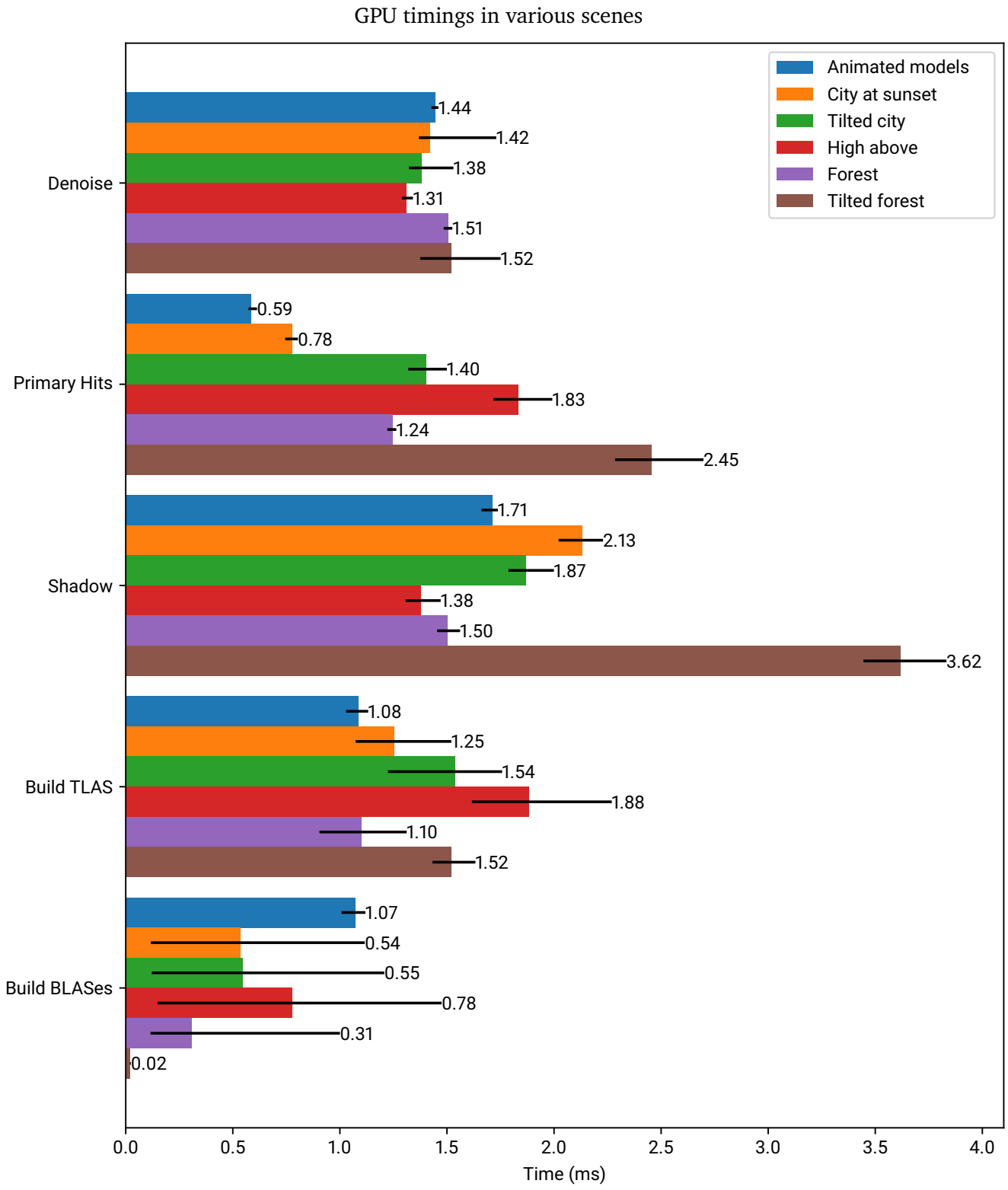

Figure 4.7: GPU timings for non path tracing sections from figure 4.5. The primary hits and shadow passes rely on rasterization.

scenes. The high path tracing times can be explained by our method relying on BLASes updates for animated units. This yields sub-optimal BLASes and slower intersection tests.

City at sunset shows a typical Anno city scene with the sun being close to the horizon. This influences the ray direction of traced shadow rays. In the worst case, shadow rays will have to traverse almost the entire scene. This explains the significantly higher path tracing times as well as the time needed to rasterize model shadows.

Tilted city view shows a typical Anno city with the tilted camera looking more towards the horizon. With this point of view, a larger portion of the scene, especially distant geometry, becomes visible. This leads to a higher TLAS instance count which entails slightly higher TLAS building times.

High above shows a full Anno island with a big city, mountains, farm fields, and forests. The viewpoint is far above the scene to include most of the island in view, much higher than allowed for the normal game perspective. Due to the high number of visible model instances, TLAS building times are increased. Otherwise, this viewpoint can be handled well. We would have expected no BLASes to be built in this perspective. Unfortunately, our LiPaC implementation still causes some *active-inactive* cycles for unique terrain-adjusted instances that require BLAS builds every couple of frames.

Forest represents a scene showing mainly trees, bushes, and grass viewed from above. Vegetation assets exhibit high amounts of overdraw, therefore rasterization times are higher for such a vegetation-heavy scene. We include this scene to evaluate our vegetation path tracing method.

Tilted forest view shows a forest scene with the camera tilted to look towards the horizon. In this perspective, many vegetation instances with high amounts of overdraw have to be rasterized for primary hits as well as directional shadows. This scene has by far the highest timings for the respective rasterization passes. At the same time, the path tracing times are the lowest due to our vegetation heuristic and completely ignoring vegetation in shadow rays. Furthermore, this point of view shows the sky which does not require any path traced indirect lighting computations.

Figure 4.8 shows the importance of LiPaC to achieve these results. We compare our hit-feedback-based approach with two simple alternative approaches: Including all instances of the scene into the TLAS (full scene, no culling) or just including instances contained either in the frustum or in a radius around the camera into the TLAS. For this test, the radius was 800 meters. The timings were gathered in the *tilted city* viewpoint from figure 4.6, with many units roaming the streets. The scene has around 5000 running animations. The time needed to animate the BLASes of all instances in the scene is prohibitively high for the simple approaches. We do not rebuild all BLASes in each frame but only every 16th frame. In between, BLASes are updated to keep build timings as low as possible. Without this optimization, the *Animate BLASes* section would be around 5-10 times higher. But this approach explains the high variance in timings. Since animations are only run inside or near the camera frustum, including instances in a radius around the camera does not decrease the number of animated BLASes. It only helps with the number of instances in the TLAS, thus decreasing TLAS building times. By leveraging light path hit feedback, LiPaC reduces the number of active instances, and therefore TLAS building times significantly. In addition, only a small number of animated models, selected by the number of encountered light paths, are included in the TLAS and have their BLAS updated in each frame. This reduces the timings for BLAS updating by a factor close to 40. At the same time, gathering hit feedback during path tracing slightly increases tracing times since it requires atomic operations.

To ensure a smooth experience while playing video games, frame time spikes have to be avoided. In a game like Anno, these typically occur while the camera is moved through the scene. Our method lazily activates instances and builds BLASes that are needed for indirect lighting of the scene. Thus, during camera movement,

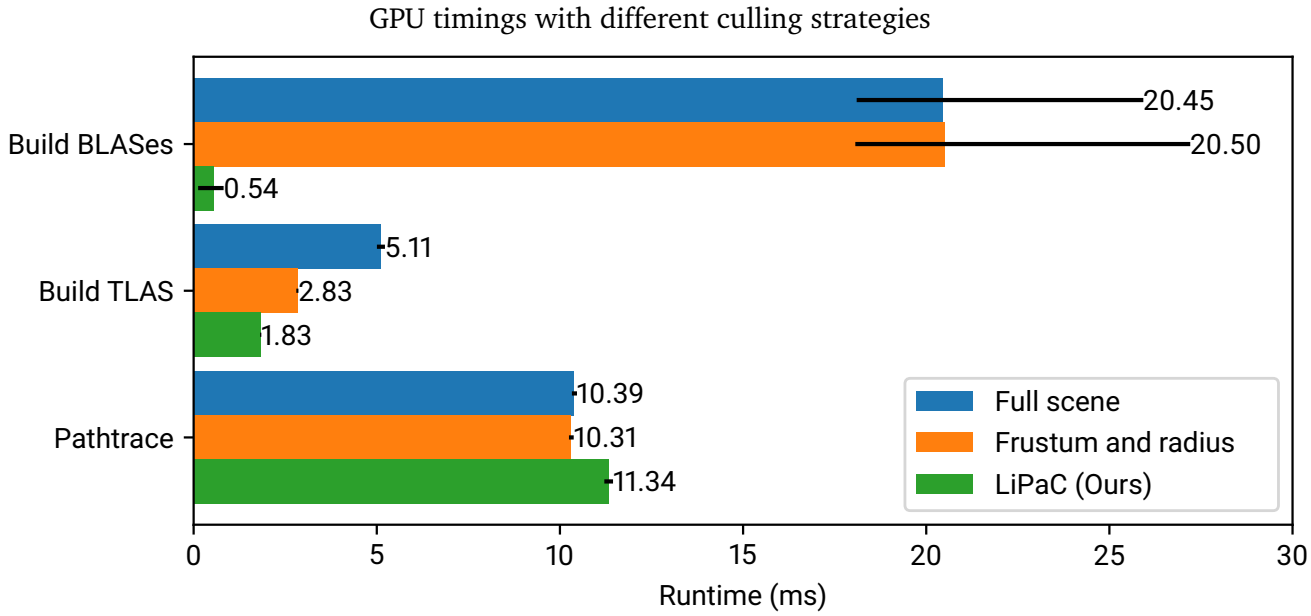

Figure 4.8: GPU runtime for an Anno city scene with many animated instances. Comparison between different TLAS instance strategies. Setup 3, Direct3D, 1080p.

many BLASes get built and GPU buffers are updated. Figure 4.9 shows the timings for the most significant CPU and GPU sections during camera movement. Frame timing spikes can be observed. In this regard, our method leaves room for improvement. However, it can be argued that the spikes remain reasonably small, as they do not extend timing spikes already observed for other parts of the game engine. Since GPU and CPU work is executed in parallel, the timings do not necessarily sum up as the frame time is given by the maximum of the CPU and GPU frame times. Additionally, some of the work on the CPU side can be parallelized as well. The top graph shows the time needed for processing the command written out by the *Process Hit Feedback* pass earlier. In this section, instances are activated or deactivated as requested by LiPaC. The *handle appearing instances (CPU)* pass implements the approach described at the end of section 3.3.2: To avoid a delay until instances appearing in the frustum are included in the TLAS, we activate all appearing instances in each frame on the CPU side. The figure shows how this mechanism is not needed while moving the camera closer to the city that was already visible, as shown by the first third of the graph. But it becomes useful when the camera moves through the city. The spike for the top two graphs in the end is the moment the camera tilts towards the horizon. Almost the entire city moves into the frustum at this point, leading to a high number of activations on the CPU side. After a couple of frames, another spike for the BLAS command processing task can be observed. This can be explained by a high number of processed commands for instances in the distance that have not been activated by the CPU activation mechanism before. These instances get activated after their associated hitboxes encounter a high number of light paths. The graph showing timings of the *Update Raytrace Buffers* CPU section does not show concerning spikes, even though all newly activated or deactivated instances as well as their entries in the geometry instance data buffer need to be updated. The bottom graph shows the time needed to build BLASes in each frame. The CPU section contains the command recording and batching of build commands while the GPU section executes the recorded commands. The increased timings around frame number 50 can be explained by the camera viewpoint close to the ground. In that case, many animated units roaming the street are activated in the TLAS and need BLAS updates or rebuilds in each frame. Other GPU tasks for LiPaC such as assembling the TLAS instance buffer or processing the hit feedback stayed well below 0.1ms for all setups and evaluated scenes.

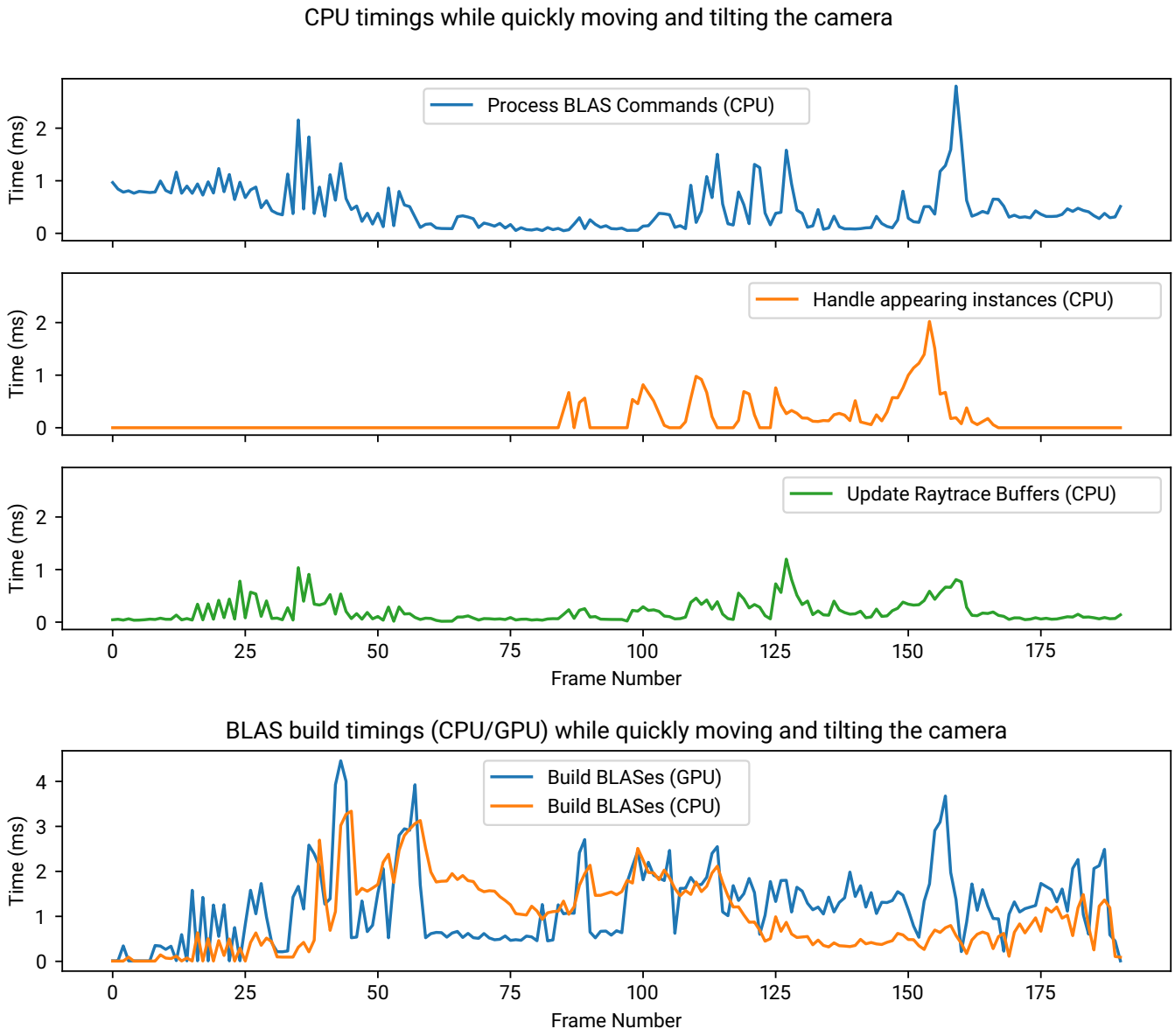

Figure 4.9: The most relevant acceleration structure management GPU and CPU timings while moving the camera through a city scene. In its initial resting post, the camera is high above the city, looking down. Then, it moves down, hovering close to a busy street. Afterward, it zooms slightly out and moves horizontally through the city. In the end, the camera is tilted to look toward the horizon, its final resting pose. All this happens in less than ten seconds, as the camera is moving quickly. Setup 2, Direct3D.

The worst case in terms of acceleration structure management is a camera jump leading to a cold start. In an evaluation scenario, we had the camera jump from an empty view to the *tilted city* view from figure 4.6. Our LiPaC latency mitigation mechanism then activates a large number of instances that newly appear in the frustum. We could observe problematic frame timing spikes, up to 15ms across our testing setups. Games that do not allow arbitrary camera jumps through the scene have to deal with this problem to a lesser extent since the hit feedback based approach will keep the scene close to the camera active. However, when rotating the camera in a wide, open world, similar issues might be encountered. While for games like Anno, such

frame spikes could be acceptable directly after a camera jump, accepting the delay and disabling the proposed approach for latency mitigation might be the preferred solution. In practice, we also limit the number of BLAS handling commands we process on the CPU in each frame to keep an upper limit on runtime. Commands that are not processed will just be written out in the following frames again. Future work could investigate a prioritization of these commands, ensuring that the most important instances get activated and added to the TLAS as quickly as possible even when just processing a low number of commands in each frame.

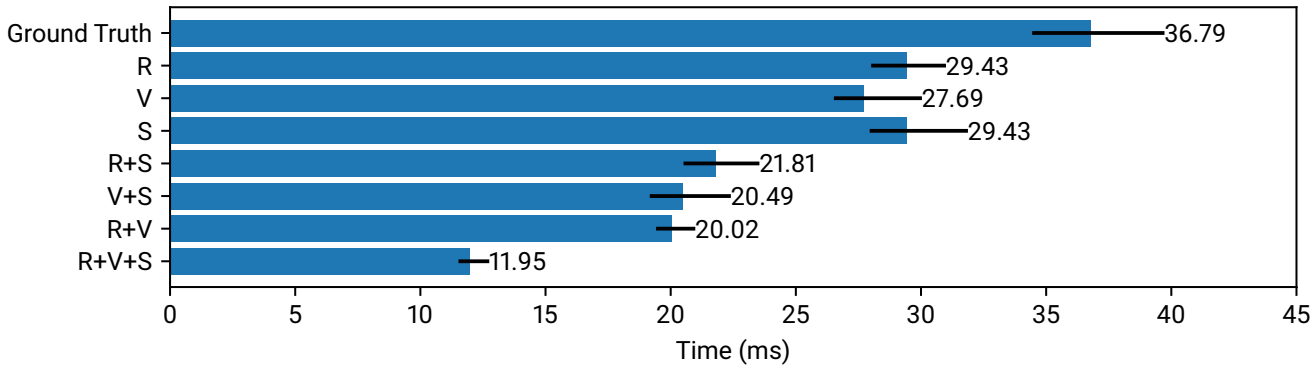

Figure 4.10: Total GPU path trace timings in an Anno gameplay perspective with high vegetation coverage. The different bars correspond to different permutations of optimizations that affect each other.
R: Using rasterization for primary hits.
V: Using the vegetation heuristic for indirect light detailed in section 3.4.
S: Sampling vegetation sun shadow from the shadow map instead of tracing shadow rays.
See figure 4.11 for timing heatmaps. Setup 3, Direct3D, 1080p.

Section 3.4 describes various optimizations to make path tracing of vegetation faster. Figure 4.10 shows respective timings with different optimizations enabled compared to the ground truth. The corresponding scene and timing heatmaps in figure 4.11 clearly show how vegetation is a bottleneck when no optimizations are applied. With all optimizations enabled, it is handled just as efficiently as any other opaque geometry, as we get rid of almost all *anyhit* shader invocations. Only for secondary rays on non scattered light paths (i.e. `E(S|T)+` light paths with the roughness of the respective specular or transmissive bounces being low) is vegetation geometry traced with all required *anyhit* invocations.

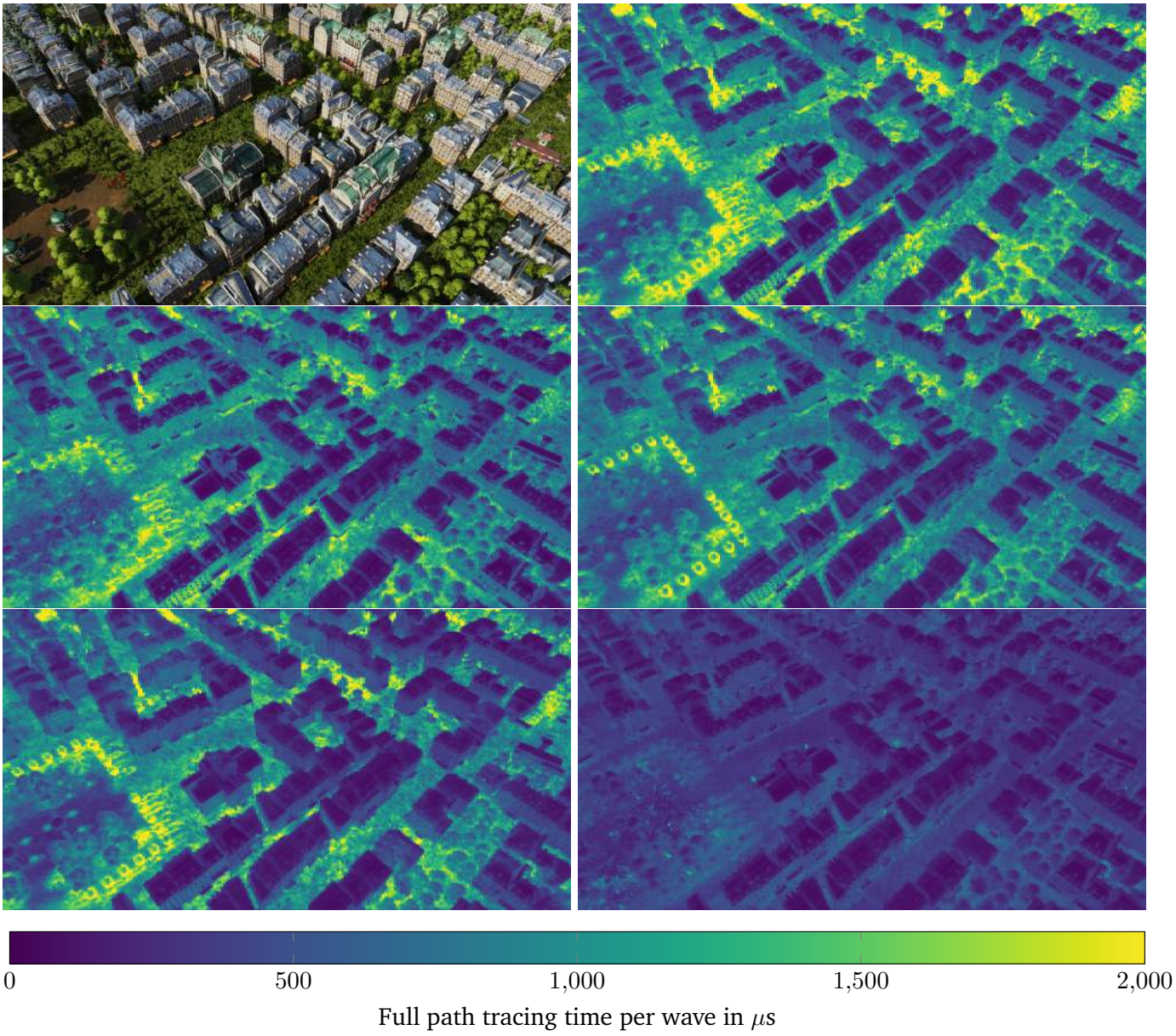

Figure 4.11: Path tracing timings of an Anno gameplay perspective with high vegetation coverage. See figure 4.10 for comparisons of timings for the entire path tracing pass and an explanation of the labels. Setup 3, Direct3D, 1080p.
Top left: ground truth path traced image of the scene.
Top right: timings of the ground truth path traced image.
Middle left: using rasterization for all primary hits (R).
Middle right: using rasterization to render vegetation sun shadow (S).
Bottom left: using our vegetation heuristic for indirect lighting (V).
Bottom right: using all available optimizations (R+S+V).

4.3 Memory Efficiency

Evaluation Method Memory consumption can be expected to be deterministic for our purposes. Therefore, we just measure it once per test case for the evaluation of static scenes. In modern graphics APIs, allocating memory for resources is handled by the application. We measure the sum of the sizes of the resources we use, any internal or external fragmentation is not considered since they could be improved by using different allocation strategies.

Memory consumption can be divided into three categories:

1. Memory that is needed by rasterization as well. This includes mesh buffers, textures, material information, and render targets not used specifically for our path tracing technique.
2. Memory consumed by data structures inherently needed for hardware ray tracing methods. This memory includes acceleration structures, the scratch buffer, temporary buffers for vertex transformation, and buffers connecting ray tracing instances with materials and mesh information.
3. Memory consumed specifically by path tracing, especially the targets used for denoising.

Graphics memory can be further differentiated by usage. The *working set memory* sums up the memory sizes of all used resources. But since large resources are sometimes only needed for specific operations, they can be *aliased*, i.e. placed in the same memory as other resources with are only needed temporarily. In an ideal aliasing scenario, where the aliased memory bottleneck is already in other passes of the frame, these resources do not increase total memory consumption. In the context of ray tracing, the scratch buffer and the buffer temporarily holding the transformed vertices for BLAS building can be aliased. Due to our various feedback and streaming approaches, memory consumption is highly resolution dependent beyond just render targets. We provide all numbers for a 1080p render resolution.

The Sponza scene (base mesh and curtains) with more than 5 million primitives and no mesh LODs consumes 380MB for BLASes on Setup 4 before compression. Memory consumption for TLAS and instance buffer stays well below 1MB due to the low number of instances.

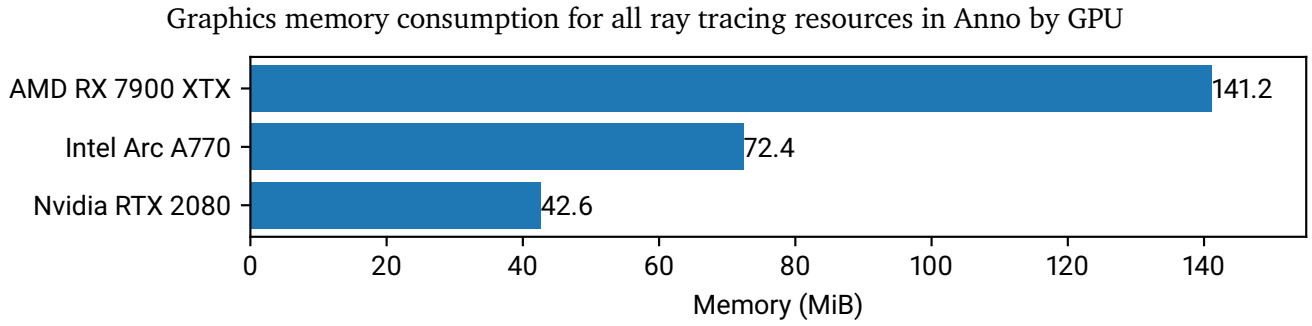

Figure 4.12: The bars contain the sum of all ray tracing specific memory resources. We compare Setup 2, 3, and 4 using Direct3D. See figures 4.13 and 4.14 for more details on the memory consumption of individual resources in this scenario.

For typical video game assets, the presence of proper mesh LODs and reduced mesh complexity results in reduced memory consumption, even though the rendered scenes are usually larger and include a high number of instances. In the Sponza scene, all meshes are extremely detailed and need to be included in the TLAS for almost all viewpoints. Thus, culling strategies do not help to reduce memory consumption in this scenario.

To evaluate memory consumption in a state of the art game engine, we once again look into various scenes from *Anno 1800*. Since acceleration structure memory consumption significantly varies between GPU vendors, figure 4.12 gives an overview of total ray tracing memory consumption per vendor in the *tilted city* scene shown in figure 4.6. We can see that memory consumption between the GPU vendors differs significantly. How much memory individual ray tracing resources consume on the different GPUs is displayed in figure 4.13. Finally, figure 4.14 shows the memory consumption of various additional ray tracing resources that show no significant difference between GPUs. Note that the scratch buffer and the buffer temporarily holding the transformed vertices for the BLAS building are initialized to be 8MB in size and extended if needed. This base size which is independent from individual GPU requirements is needed for the efficient BLAS build batching described in section 3.3 since all builds in a single batch need their own slice of these buffers.

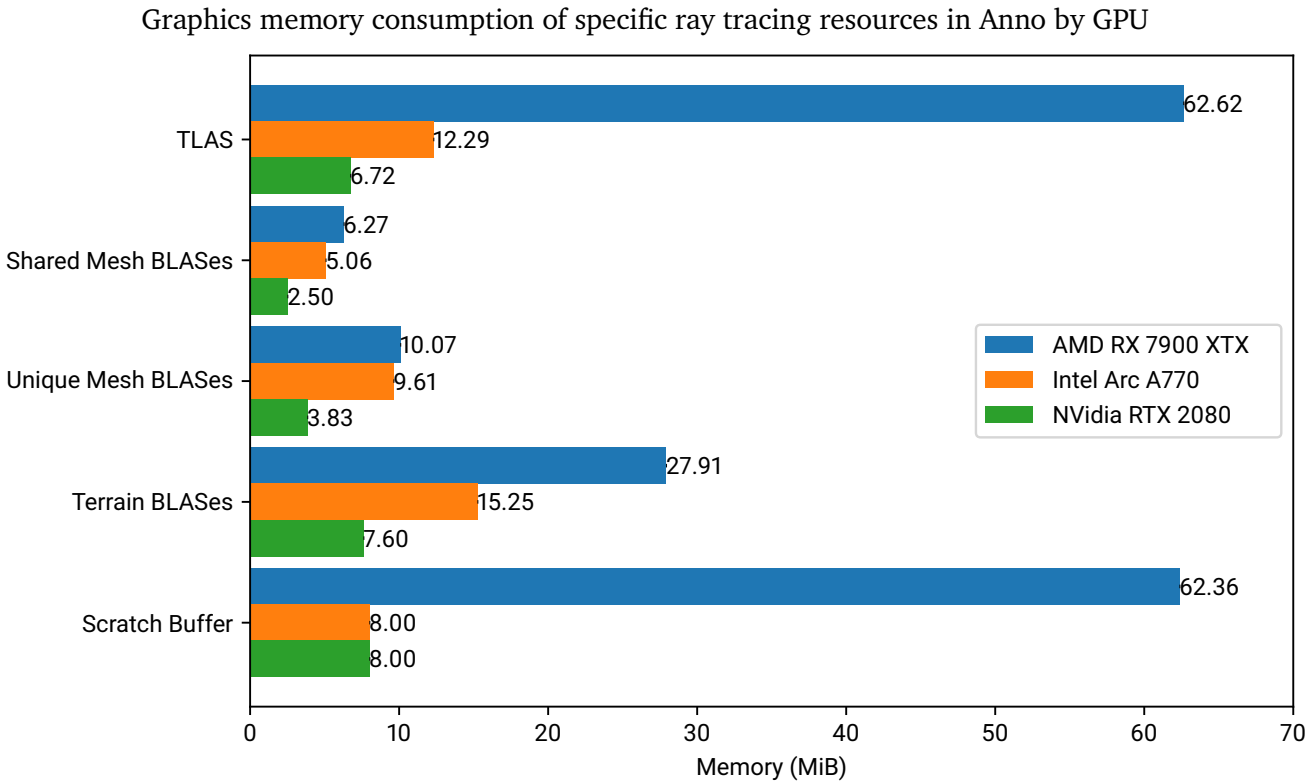

Figure 4.13: The bars describe how much memory the individual resources or resource categories consume. “Unique Mesh BLASes” describes BLASes that are not shared between instances because each instance needs unique per-vertex transformation, e.g. for skinning or terrain adaptation. Memory consumption was captured in the same scenario as figure 4.12.

To put these numbers into perspective, figure 4.15 provides an overview of the most important other categories of graphics memory consumption. The figure has no claim to cover all graphics memory consumption. Anno has many resources for specific gameplay-relevant features that are not relevant to our method. All except for the last bar represent memory required by a rasterizing renderer as well. The bar for render targets does not include any path tracing specific resources, as all of these resources are included in the last bar. Denoising render targets consume the biggest portion of path tracing memory. We use the combined diffuse-specular-denoising technique of the ReBLUR denoiser. At 1080p resolution, its working set includes 169MB but 97MB of that memory is aliased. More details on ReBLUR memory consumption can be found in its documentation.

While different scenes and viewpoints cause different memory consumption due to different instance counts

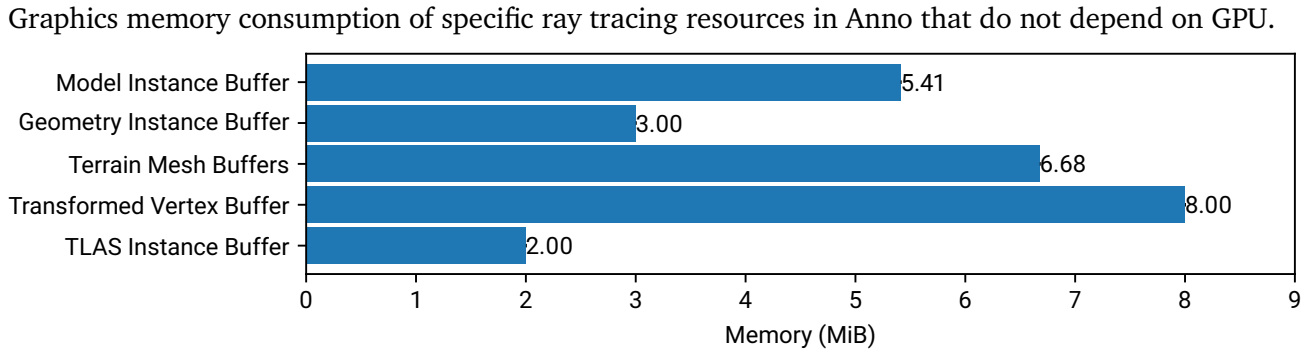

Figure 4.14: Memory consumption of the shown resources appears to be independent of the GPU vendor. Memory consumption was captured in the same scenario as figure 4.12.

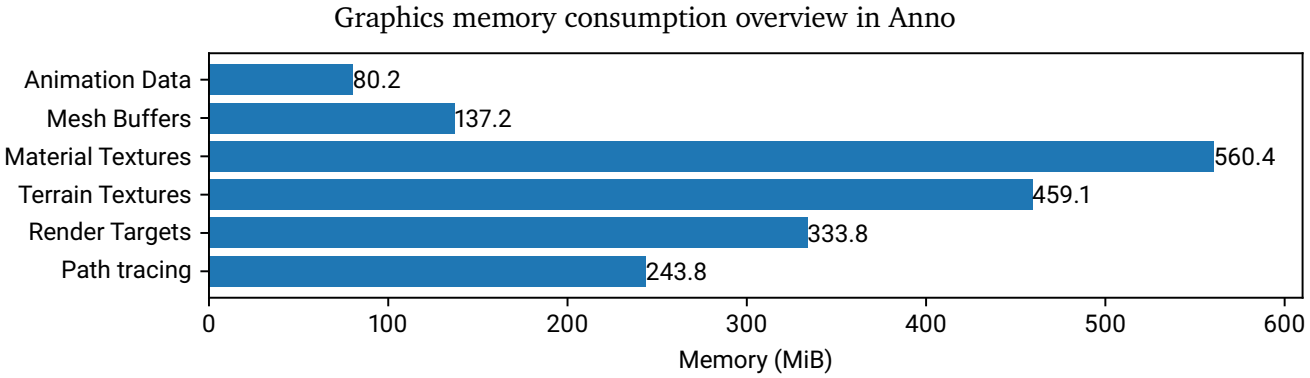

Figure 4.15: Graphics memory consumption in Anno by category, captured in the same scenario as figure 4.12. We observe these numbers to be almost independent from GPU vendors, up to a small percentage. Memory consumption of ray tracing resources is not included since it heavily depends on the used GPU.

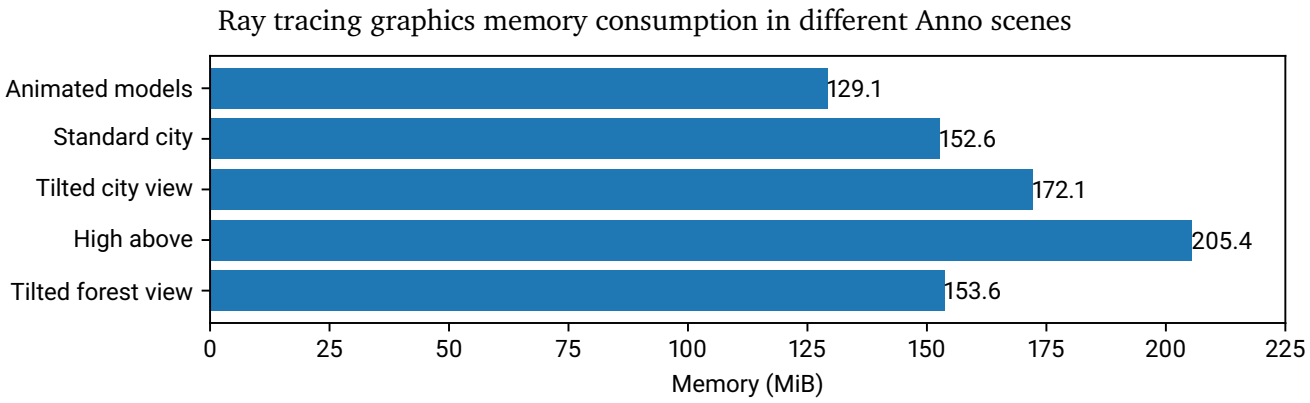

Figure 4.16: Graphics memory consumption for ray tracing resources for different Anno scenes and points of view. See section 4.2 for a detailed explanation of the scenes. Setup 3 using Direct3D at 1080p.

and different sets of BLASes present in memory, no scene exceeded reasonable memory bounds. Figure 4.16 shows memory consumption for the scenes from figure 4.6. The highest memory consumption can be observed for the camera perspective far above the ground. In this case, it is caused by a high number of active instances, leading to high TLAS, instance buffer, and scratch buffer size. Even though low quality mesh LODs can be

used in this perspective, a high number of shared BLASes have to be present in memory in this perspective. Such a perspective has not been included in Anno games so far since it causes a rasterizing renderer to quickly hit memory and performance limits as well.

In summary, with these numbers on graphics memory consumption, we can conclude that even large current generation game scenes fit into the memory of currently available commodity hardware. Recent GPUs capable of running real-time ray tracing techniques have at least 8GB of graphics memory available. Without our LiPaC method, graphics memory consumption is significantly higher, see figure 4.17.

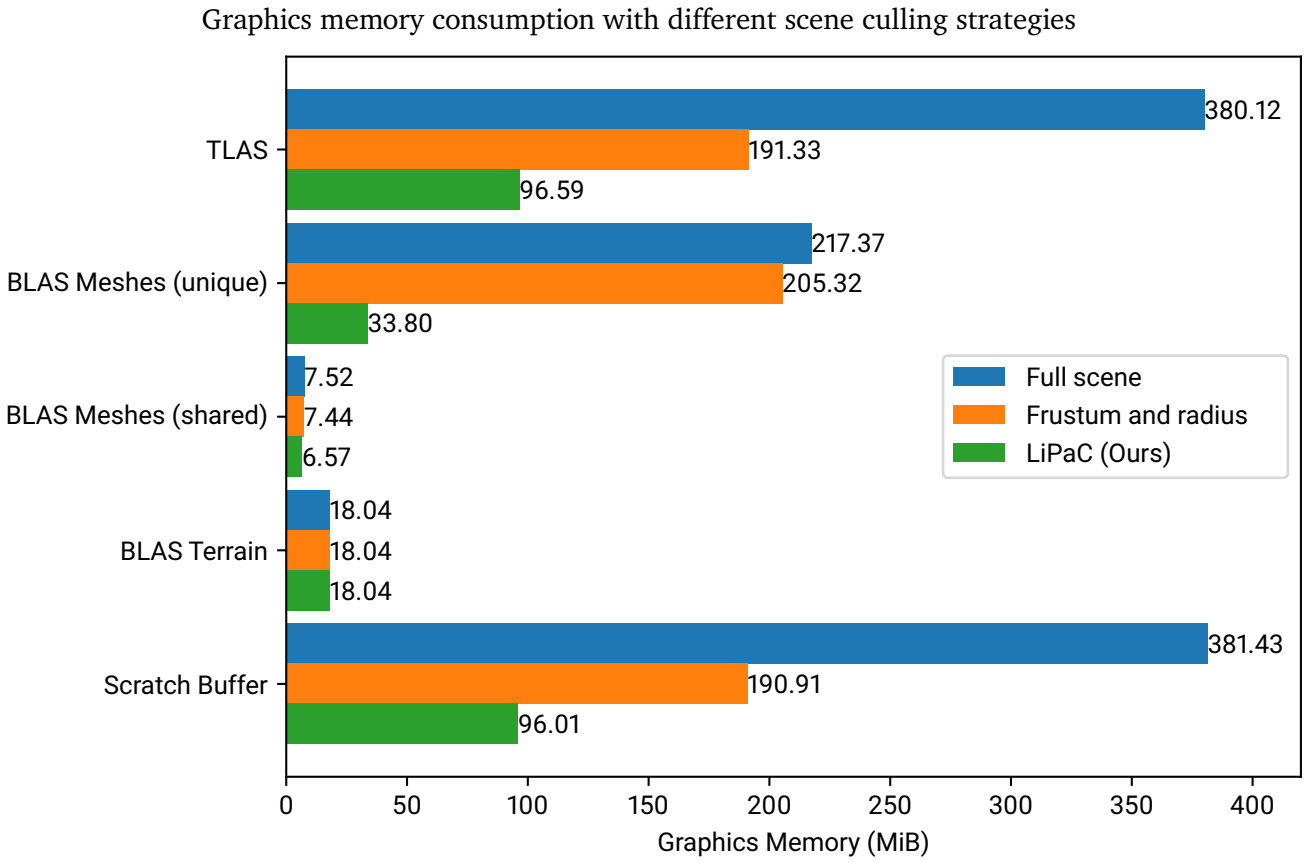

Figure 4.17: GPU memory consumption for the Anno *tilted city* scene with many animated instances. Setup 3 using Direct3D at 1080p. The "Frustum and radius" strategy includes all instances that are in the frustum or in a radius of 800 meters around the camera.

For CPU memory consumption, the biggest data structures for ray tracing are the model instance and geometry instance buffers and their update and free lists. Even in the biggest Anno game scenes, the combined memory consumption of these buffers on CPU side never exceeded 40MB. Other ray tracing data structures, such as the hash map of shared BLASes, terrain instance data buffers, and various maintenance lists for compaction, pending BLAS builds and LOD switches always stayed below 1MB. The CPU memory overhead of our method is thus negligible in the context of current computer hardware.

4.4 Visual Details

Path tracing has a significant impact on runtime performance and perceived smoothness of an interactive renderer, even on the most capable hardware. However, in some contexts and for some games, visual improvements are of greater importance.

The Sponza scene rendered by our path tracer can be seen in figure 1.1 and figure 4.1. Since large parts of the scene are barely illuminated by direct light, significant visual improvements can be achieved with path tracing. At the same time, exactly these lighting conditions show the limitations of our simple light transport method. Figure 4.18 illustrates a scenario with extremely poor lighting conditions. Exposure was increased by a factor of 1000 compared to what is needed in sun-lit parts of the scene. The image on the left shows the results accumulated over 10 frames. Most samples do not accumulate any light at all. While denoising manages to process that signal into somewhat coherent lighting of the scene in the middle picture, large-scale and blobby noise artifacts start to appear. The right image shows the same scene with quick camera movement and denoising. At the left edge of the image, significant lighting artifacts can be seen that would be unacceptable in games. Many games do not show scenes with such poor lighting conditions. However, the visible artifacts in such conditions could be significantly improved by extending our method with techniques such as radiance caches [Gau22; Hal+21; Sta21], spatiotemporal path reuse [Ouy+21; Lin+22], or more advanced light transport approaches [VG97].

No denoising, 10 spp | With ReBLUR denoising | Denoising and camera movement

Figure 4.18: Path tracing in an occluded corner of the Sponza scene.

Table 4.2 compares the rasterizing renderer in the Anno engine with our path tracing method. The first row shows a big city scene with additional vegetation in the streets. In shadowed regions, the lighting becomes much more realistic and the image gains a new sense of depth not present in the rasterized image. This effect becomes even more apparent in the forest scene depicted in the second row. Since the sun in that specific scene is low behind a mountain, most of the scene lies in shadow. Without any proper indirect lighting information on the left, it becomes hard to “read” the forest. Trees clump together and can barely be visually separated from bushes and grass. With path tracing, “understanding” the scene becomes easier. Path tracing does not just produce more visually appealing images. In some cases, it allows the human eye to get an improved understanding of scenes, geometry, and materials compared to images with incorrect lighting. The third and fourth rows of table 4.2 show selected assets at a closer distance. The sandy scene in the third row illustrates the impact of GI well. While the shadowed houses in the rasterized image have a blue tint, they look well integrated in the more warmly shaded scene in the path traced image. Finally, the path traced bank asset in the fourth row exhibits an increased sense of depth not only for the area behind the columns in the bottom

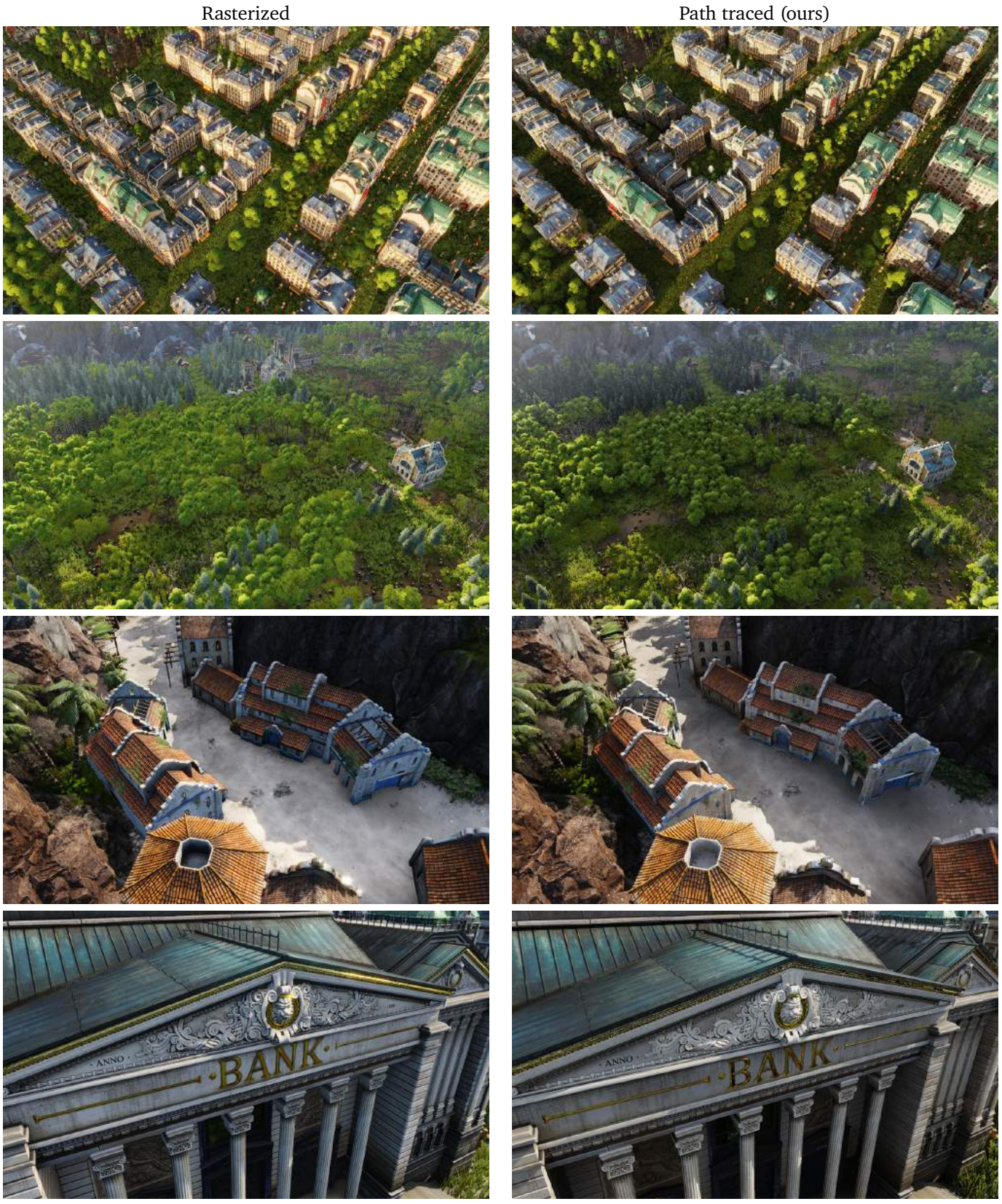

Table 4.2: Comparison of a rasterizing renderer with our path tracer in various Anno game scenes.

but also for the lion head and facade decoration at the top. The specular highlights added by path tracing on the golden surfaces make the material more recognizable and believable compared to the dull shading on the left. Such details do not only increase the overall visual fidelity of the game but may also provide additional value. For instance, they could increase player gratification after having achieved building this landmark. The potential of increased immersion and player satisfaction as well as gameplay advantages by easier readability of the scene make these results promising.

Rasterized (with SSAO) Ray traced AO Fully path traced

Figure 4.19: Comparison of an Anno city scene at sunset with different indirect lighting strategies.

The most significant impact on visual quality in Anno scenes was achieved by correct ambient occlusion. Figure 4.19 shows an Anno city scene at sunset with just SSAO on the left, ray traced AO in the middle, and finally full GI by path tracing on the right. While just adding ray traced AO adds significant visual details and depth, the image in the middle appears too dark. Especially the trees and streets on the right appear better integrated with the additional indirect light. Figure 4.20 shows a close-up of a city block, comparing ray traced AO and path tracing. While the difference is subtle, the shadowed regions on the right are shaded more warmly by the indirect sunlight with path tracing. One can also observe additional specular details on the path traced image, rendering the roof in the top-right of the image more readable. However, since computing ray traced AO is significantly cheaper than full GI and path tracing, it is a good trade-off to consider for games like Anno.

As described in section 3.4, we approximate indirect lighting from vegetation to significantly decrease tracing runtime by avoiding many *anyhit* shader invocations. Figure 4.21 shows a comparison of this approximation with ground truth. The final composited image in the top row shows almost no significant difference. However, the comparison of denoised indirect diffuse lighting in the second row presents us with much less detail in the grass-filled streets when using our heuristic. For trees, no significant difference can observed. To investigate how well our approximation works, table 4.3 evaluates the difference in indirect diffuse lighting for various vegetation assets. While the method works well for all trees and bushes we tested, it shows problems for grass assets. The heuristics we describe in section 3.4 approximate vegetation assets as bounding box volumes. This volume approach does not work well for grass, as its individual grass blades drive occlusion much more than their combined volume. A better heuristic could, for instance, bake the ambient occlusion a grass asset throws onto the ground in an offline step and then use this information to guide ray skipping.

Ray traced AO | Fully path traced

Figure 4.20: Comparison of an Anno closeup rendered with just ray traced AO and full path tracing.

Figure 4.21: Comparison between ground truth and our vegetation skipping heuristic for indirect lighting. While the first row shows the composited results, the second row illustrates the denoised indirect diffuse lighting. Differences can be observed for the grassy ground.

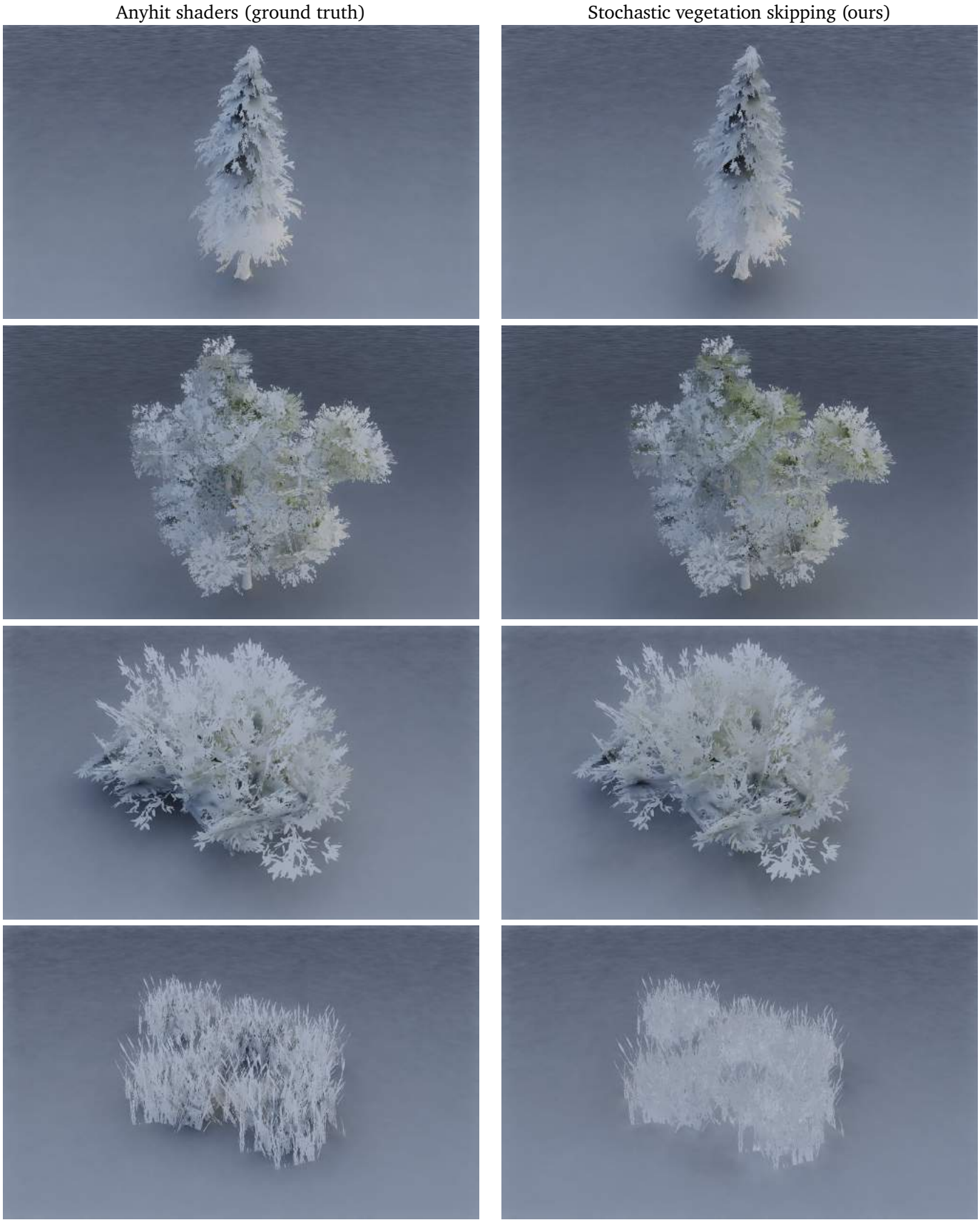

Table 4.3: Comparison of ground truth with our vegetation skipping method for diffuse radiance after denoising.

5 Conclusion

The methods presented in this thesis open the door for path tracing in real-time renderers. This facilitates photo-realistic graphics in future computer games. GPU runtime of path tracing at full resolution is still a concern and limited to only the most capable of current generation commodity hardware. However, future GPUs can be expected to change this and allow real-time path tracing on a wide range of hardware. For now, computing indirect lighting in lower resolutions on less capable hardware can present a good alternative. Another solution is to rely on established real-time ray tracing techniques that do not aim to solve the entire rendering equation. These techniques can be combined with our methods as well.

We have presented LiPaC, a scene culling method that is guided by light path feedback and therefore suitable for ray tracing techniques of all kinds. Our evaluations have shown significant advantages for acceleration structure build times and memory consumption, with only a small impact on tracing performance.

For ray tracing techniques dealing with diffuse or otherwise highly scattered indirect lighting, our stochastic vegetation ray skipping approach provides significant speedup at the cost of reduced indirect lighting quality. The performance improvements shown by our evaluation make this a good trade-off. Real-time path tracing of vegetation-heavy scenes would not have been possible even on the most capable current generation hardware without it.

5.1 Limitations

The indirect lighting differences for the vegetation ray skipping heuristic might be too significant for some contexts. In rendering contexts outside of video games, the results from section 4.4 might be unacceptable. We adjusted our heuristic parameters for trees since they have the largest impact on indirect lighting. Better results might be achieved with parameters depending on the specific asset but fine-tuning them manually is sub-optimal.

Our LiPaC method has a potential latency of multiple frames when objects first appear in the reflection of mirror-like surfaces. In practice, during normal camera movement, this was usually not noticeable since the bouncing of rays through the scene already activates instances in proximity to the currently visible surfaces. However, when the camera jumps to a new position and directly faces a mirror, this latency can be noticed.

For vegetation and instances culled by LiPaC, we rely on rasterization for shadows. This only works for traditional video game light sources and cannot provide good approximations of shadows for light coming from emissive surfaces. To render scenes that rely heavily on such light sources, our method has to be adjusted.

The presented path tracing method focuses on surface interaction and does not include volumetric light interactions such as fog, smoke, or light traveling through water. Volumetric effects are costly to implement into real-time path tracing and approximations are needed for games.

5.2 Future Work

More complex spatial acceleration structures, such as quadtrees [Sam84] or BVHs [Mei+21] for hitboxes in our light path feedback method could provide performance improvements for large scenes. Using other bounding volumes instead of just AABBs for *omitted* model instances might avoid unnecessary instance activation and activation-deactivation cycles we observed due to the coarse approximative nature of AABBs. Multiple other approaches could be built upon this method. Handling light sources in a similar manner, only ever considering light sources in active regions of the scene might be possible with some minor adjustments to the method. We also suspect that better heuristics for the activation of scene regions and individual instances can be found. More sophisticated scene analysis on the CPU to activate instances likely reachable by light paths before rendering the scene could allow us to mitigate the latency visible in appearing mirrors.

Our method exclusively builds upon backward path tracing. This causes some caustics to appear noisy to a degree that denoising cannot properly process into a visually appealing image, especially under motion. Integrating photon mapping [Jen96] approaches into our method would improve this issue. Bidirectional path tracing and photon mapping would interact with LiPaC, likely requiring adjustments to our method.

The evaluated heuristic for vegetation ray skipping could also be improved. Heuristics utilizing pre-computed scattering profiles or a low-resolution voxelized density approximation of the asset would likely deliver better results, especially for grass assets. This could be useful to visualize vegetation for which bounding box heuristics cannot capture the complex geometrical shape that should be preserved for indirect lighting. The performance and memory consumption differences between our vegetation ray skipping heuristic and opacity micromaps [FO23] should be evaluated.

Acronyms

AABB axis-aligned bounding box. 31, 37, 39, 63, 64

AO ambient occlusion. 7, 8, 14, 59, 60, 64

API application programming interface. 30, 42, 43, 53, 64

BLAS bottom-level acceleration structure. 17–21, 23, 25–33, 35, 36, 45, 48, 49, 51, 53–56, 64

BRDF bidirectional reflectance distribution function. 12, 13, 64

BSDF bidirectional scattering distribution function. 10, 13, 64

BVH bounding volume hierarchy. 31, 63, 64

CPU central processing unit. 19, 27, 29, 30, 33–35, 41, 43, 49–51, 56, 63, 64

fps frames per second. 7, 8, 64

GI global illumination. 8, 9, 14, 15, 57, 59, 64

GPU graphics processing unit. 4, 7, 8, 11, 15–17, 19, 21–36, 39, 41–46, 49–51, 53–56, 62, 64

IBL image-based lighting. 14, 64

LiPaC light path guided culling. 4, 20, 23, 24, 29–31, 46, 48–50, 56, 62–64

LOD level of detail. 20, 24–27, 29, 30, 41, 42, 53, 55, 56, 64

MIS multiple importance sampling. 12, 64

NEE next event estimation. 11, 13, 64

NRD Nvidia ray tracing denoiser. 23, 64

PBR physically based rendering. 14, 64

PDF probability density function. 11, 12, 64

PSR primary surface replacement. 16, 27, 28, 45, 64

spp samples per pixel. 13, 64

SSAO screen space ambient occlusion. 14, 59, 64

TAA temporal anti-aliasing. 16, 21, 64

TLAS top-level acceleration structure. 17–19, 23, 25, 26, 30–33, 35, 45, 46, 48, 49, 51, 53, 55, 64

TLSF two-level segregate fit. 26, 64

Bibliography

[AK21] Dirk Gerrit van Antwerpen and Oliver Klehm. "Accelerating Boolean Visibility Operations Using RTX Visibility Masks". In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 643–657. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_40. URL: https://doi.org/10.1007/978-1-4842-7185-8_40.

[Ake+19] Tomas Akenine-Möller et al. "Texture Level of Detail Strategies for Real-Time Ray Tracing: High-Quality and Real-Time Rendering with DXR and Other APIs". In: Feb. 2019, pp. 321–345. ISBN: 978-1-4842-4426-5. DOI: 10.1007/978-1-4842-4427-2_20.

[And19] Claire Andrews. *Coming to DirectX 12— Sampler Feedback: some useful once-hidden data, unlocked*. Nov. 4, 2019. URL: https://devblogs.microsoft.com/directx/coming-to-directx-12-sampler-feedback-some-useful-once-hidden-data-unlocked/ (visited on 12/12/2023).

[Arc+19] Ben Archard et al. *Exploring the Ray Traced Future in 'Metro Exodus'*. Mar. 18, 2019. URL: https://www.gdcvault.com/play/1026186/RTX-in-Justice-(Presented-by (visited on 11/07/2023).

[Bav+18] Louis Bavoil et al. *Hybrid Ray-Traced Ambient Occlusion*. Poster at HPG18 ACM SIGGRAPH / Eurographics High Performance Graphics. Aug. 2018. URL: https://casual-effects.com/research/Bavoil2018AO/index.html.

[Bit+20] Benedikt Bitterli et al. "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39.4 (July 2020). DOI: 10/gg8xc7.

[BJW21] Jakub Boksansky, Paula Jukarainen, and Chris Wyman. "Rendering Many Lights with Grid-Based Reservoirs". In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 351–365. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_23. URL: https://doi.org/10.1007/978-1-4842-7185-8_23.

[BN21] Vladimir Bondarev and Sriharsha Niverty. *Advanced API Performance: Async Compute and Overlap*. Oct. 21, 2021. URL: https://developer.nvidia.com/blog/advanced-api-performance-async-compute-and-overlap/ (visited on 12/24/2023).

[BN76] James F. Blinn and Martin E. Newell. "Texture and Reflection in Computer Generated Images". In: *Commun. ACM* 19.10 (Oct. 1976), pp. 542–547. ISSN: 0001-0782. DOI: 10.1145/360349.360353. URL: https://doi.org/10.1145/360349.360353.

[Bou23] Darius Bouma. *Dynamic diffuse global illumination*. Dec. 30, 2023. URL: https://blog.traverseresearch.nl/dynamic-diffuse-global-illumination-b56dc0525a0a (visited on 12/31/2023).

[BPA16] A. Beacco, N. Pelechano, and C. Andújar. “A Survey of Real-Time Crowd Rendering”. In: *Computer Graphics Forum* (2016). ISSN: 1467-8659. DOI: 10.1111/cgf.12774.

[BS22] Stephanie Brenham and Ihor Szlachtycz. *Performant Reflective Beauty: Hybrid Ray Traced Reflections In Far Cry 6*. Mar. 21, 2022. URL: https://gpuopen.com/gdc-presentations/2022/GDC_Performant_Reflective_Beauty_Hybrid_Ray_Traced_Reflections_In_Far_Cry_6.pdf (visited on 11/28/2023).

[Bur23a] Andrew Burnes. *Cyberpunk 2077: Technology Preview Of New Ray Tracing Overdrive Mode Out Now*. Apr. 11, 2023. URL: https://www.nvidia.com/en-us/geforce/news/cyberpunk-2077-ray-tracing-overdrive-update-launches-april-11/ (visited on 11/24/2023).

[Bur23b] Andrew Burnes. *NVIDIA DLSS 3.5: Enhancing Ray Tracing With AI*. Aug. 22, 2023. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-3-5-ray-reconstruction/ (visited on 10/20/2023).

[BWB19] Jakub Boksansky, Michael Wimmer, and Jiri Bittner. “Ray Traced Shadows: Maintaining Real-Time Frame Rates: High-Quality and Real-Time Rendering with DXR and Other APIs”. In: Feb. 2019, pp. 159–182. ISBN: 978-1-4842-4426-5. DOI: 10.1007/978-1-4842-4427-2_13.

[Cow+15] Brent Cowan et al. “Screen space point sampled shadows”. In: *2015 IEEE Games Entertainment Media Conference (GEM)*. 2015, pp. 1–7. DOI: 10.1109/GEM.2015.7377248.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. “Distributed Ray Tracing”. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 137–145. ISSN: 0097-8930. DOI: 10.1145/964965.808590. URL: https://doi.org/10.1145/964965.808590.

[DHD20] Addis Dittebrandt, Johannes Hanika, and Carsten Dachsbacher. “Temporal Sample Reuse for Next Event Estimation and Path Guiding for Real-Time Path Tracing”. In: *Eurographics Symposium on Rendering - DL-only Track*. Ed. by Carsten Dachsbacher and Matt Pharr. The Eurographics Association, 2020. ISBN: 978-3-03868-117-5. DOI: 10.2312/sr.20201135.

[DiG22] David DiGioia. *Improving raytracing performance with the Radeon Raytracing Analyzer*. Sept. 15, 2022. URL: https://gpuopen.com/learn/improving-rt-perf-with-rra/ (visited on 12/24/2023).

[DS19] Johannes Deligiannis and Jan Schmid. *”It Just Works”: Ray-Traced Reflections in ’Battlefield V*. Mar. 18, 2019. URL: https://www.gdcvault.com/play/1026282/It-Just-Works-Ray-Traced (visited on 11/07/2023).

[Dun19] Alex Dunn. *Tips and Tricks: Ray Tracing Best Practices*. Mar. 20, 2019. URL: https://developer.nvidia.com/blog/rtx-best-practices/ (visited on 11/09/2023).

[FO23] Simon Fenney and Alper Ozkan. “Compressed Opacity Maps for Ray Tracing”. In: *High-Performance Graphics - Symposium Papers*. Ed. by Jacco Bikker and Christiaan Gribble. The Eurographics Association, 2023. ISBN: 978-3-03868-229-5. DOI: 10.2312/hpg.20231133.

[Gau22] Pascal Gautron. “Advances in Spatial Hashing: A Pragmatic Approach towards Robust, Real-Time Light Transport Simulation”. In: *ACM SIGGRAPH 2022 Talks*. SIGGRAPH ’22. Vancouver, BC, Canada: Association for Computing Machinery, 2022. ISBN: 9781450393713. DOI: 10.1145/3532836.3536239. URL: https://doi.org/10.1145/3532836.3536239.

[GB22] Holger Gruen and Joshua Barczak. *A Quick Guide to Intel’s Ray-Tracing Hardware*. Mar. 22, 2022. URL: https://www.intel.com/content/www/us/en/content-details/726653/a-quick-guide-to-intel-s-ray-tracing-hardware.html (visited on 11/09/2023).

[Gha+89] N. Gharachorloo et al. “A Characterization of Ten Rasterization Techniques”. In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 355–368. ISBN: 0897913124. DOI: 10.1145/74333.74370. URL: https://doi.org/10.1145/74333.74370.

[Gor+84] Cindy M. Goral et al. “Modeling the Interaction of Light between Diffuse Surfaces”. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 213–222. ISBN: 0897911385. DOI: 10.1145/800031.808601. URL: https://doi.org/10.1145/800031.808601.

[Hai19] Xueqing Yang Haiyong Qian. *RTX in Justice*. Mar. 18, 2019. URL: https://www.gdcvault.com/play/1026186/RTX-in-Justice-(Presented-by (visited on 11/08/2023).

[Hal+21] Henrik Halen et al. *Global Illumination based on Surfels*. 2021. URL: https://www.ea.com/seed/news/siggraph21-global-illumination-surfels (visited on 08/05/2023).

[Han21] Johannes Hanika. “Hacking the Shadow Terminator”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 65–76. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_4. URL: https://doi.org/10.1007/978-1-4842-7185-8_4.

[HBW20] Tobias Hector, Joshua Barczak, and Eric Werness. *Ray Tracing In Vulkan*. Dec. 15, 2020. URL: https://www.khronos.org/blog/ray-tracing-in-vulkan (visited on 09/10/2023).

[HD17] Stephan Hodes and Alex Dunn. *Deep Dive: Asynchronous Compute*. Feb. 27, 2017. URL: https://gpuopen.com/wp-content/uploads/2017/03/GDC2017-Asynchronous-Compute-Deep-Dive.pdf (visited on 12/24/2023).

[HE21] Nikolai Hofmann and Alex Evans. “Efficient Unbiased Volume Path Tracing on the GPU”. In: Aug. 2021, pp. 699–711. ISBN: 978-1-4842-7184-1. DOI: 10.1007/978-1-4842-7185-8_43.

[Hec90] Paul Heckbert. “Adaptive Radiosity Textures for Bidirectional Ray Tracing”. In: vol. 24. Sept. 1990, pp. 145–154. DOI: 10.1145/97880.97895.

[Hei18] Eric Heitz. “Sampling the GGX Distribution of Visible Normals”. In: *Journal of Computer Graphics Techniques (JCGT)* 7.4 (Nov. 2018), pp. 1–13. ISSN: 2331-7418. URL: http://jcgt.org/published/0007/04/01/.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. “Progressive Photon Mapping”. In: *ACM Trans. Graph.* 27.5 (Dec. 2008). ISSN: 0730-0301. DOI: 10.1145/1409060.1409083. URL: https://doi.org/10.1145/1409060.1409083.

[HY21] Yuchi Huo and Sung-eui Yoon. “A survey on deep learning-based Monte Carlo denoising”. In: *Computational Visual Media* 7 (Mar. 2021). DOI: 10.1007/s41095-021-0209-9.

[Inc17] The Khronos Group Inc. *VK EXT descriptor indexing*. Oct. 2, 2017. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_descriptor_indexing.html (visited on 12/12/2023).

[Inc19] The Khronos Group Inc. *VK KHR shader clock*. Apr. 25, 2019. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_shader_clock.html (visited on 08/20/2023).

[Int22] Intel. “GPU Research Samples Library”. In: (Apr. 20, 2022). URL: https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-research/samples.html (visited on 09/10/2023).

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. USA: A. K. Peters, Ltd., 2001. ISBN: 1568811470.

[Jen96] Henrik Wann Jensen. “Global Illumination using Photon Maps”. In: *Rendering Techniques ’96*. Ed. by Xavier Pueyo and Peter Schröder. Vienna: Springer Vienna, 1996, pp. 21–30. ISBN: 978-3-7091-7484-5.

[Jos23] Holger Gruen Joshua Barczak. *Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games*. Jan. 25, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html (visited on 02/02/2023).

[Kaj86] James T. Kajiya. “The Rendering Equation”. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150. ISBN: 0897911962. DOI: 10.1145/15922.15902. URL: https://doi.org/10.1145/15922.15902.

[Kar13] Brian Karis. *Real Shading in Unreal Engine 4*. 2013. URL: https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf (visited on 11/02/2023).

[Kar14] Brian Karis. *High-Quality Temporal Supersampling*. Nov. 8, 2014. URL: https://advances.realtimerendering.com/s2014/ (visited on 08/20/2023).

[KD13] Anton S. Kaplanyan and Carsten Dachsbacher. “Path Space Regularization for Holistic and Robust Light Transport”. In: *Computer Graphics Forum* 32.2pt1 (2013), pp. 63–72. DOI: https://doi.org/10.1111/cgf.12026. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12026. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12026.

[Kel19] Jan Kelling. “Integrating physically based rendering (PBR) techniques into a multi-object-draw rendering framework”. Bachelor’s Thesis. TU Darmstadt, Dec. 2019.

[KK86] Timothy L. Kay and James T. Kajiya. “Ray Tracing Complex Scenes”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916. URL: https://doi.org/10.1145/15886.15916.

[Laf95] Eric P. Lafortune. “Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering”. PhD thesis. Celestijnenlaan 200A, 3001 Heverlee, Belgium: Department of Computer Science, KU Leuven, Feb. 1995, p. 167.

[Lik+15] G. Liktor et al. “Stochastic Soft Shadow Mapping”. In: *Comput. Graph. Forum* 34.4 (July 2015), pp. 1–11. ISSN: 0167-7055.

[Lin+22] Daqi Lin et al. “Generalized Resampled Importance Sampling: Foundations of ReSTIR”. In: *ACM Trans. Graph.* 41.4 (July 2022). ISSN: 0730-0301. DOI: 10.1145/3528223.3530158. URL: https://doi.org/10.1145/3528223.3530158.

[LL20] Won-Jong Lee and Gabor Liktor. “Lazy Build of Acceleration Structures with Traversal Shaders”. In: *SIGGRAPH Asia 2020 Technical Communications*. SA ’20. Virtual Event, Republic of Korea: Association for Computing Machinery, 2020. ISBN: 9781450380805. DOI: 10.1145/3410700.3425430. URL: https://doi.org/10.1145/3410700.3425430.

[Lla19] Ignacio Llamas. *Omniverse RTX Real-Time Ray Tracer*. July 29, 2019. URL: https://on-demand.gputechconf.com/siggraph/2019/pdf/sig916-omniverse-rtx-real-time-ray-tracer.pdf (visited on 02/04/2024).

[LLV19] Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. "Flexible Ray Traversal with an Extended Programming Model". In: *SIGGRAPH Asia 2019 Technical Briefs*. SA '19. Brisbane, QLD, Australia: Association for Computing Machinery, 2019, pp. 17–20. ISBN: 9781450369459. DOI: 10.1145/3355088.3365149. URL: https://doi.org/10.1145/3355088.3365149.

[LWY21] Daqi Lin, Chris Wyman, and Cem Yuksel. "Fast Volume Rendering with Spatiotemporal Reservoir Resampling". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2021)* 40.6 (Dec. 2021), 279:1–279:18. ISSN: 0730-0301. DOI: 10.1145/3478513.3480499. URL: http://doi.acm.org/10.1145/3478513.3480499.

[Mak23] Evgeny Makarov. *Practical Tips for Optimizing Ray Tracing*. Jan. 5, 2023. URL: https://developer.nvidia.com/blog/practical-tips-for-optimizing-ray-tracing/ (visited on 01/07/2024).

[Mar+17] Michael Mara et al. "An Efficient Denoising Algorithm for Global Illumination". In: *Proceedings of High Performance Graphics*. Los Angeles, California, USA: ACM, July 2017. ISBN: 978-1-4503-5101-0. DOI: 10/gfzndq.

[Mas+04] M. Masmano et al. "TLSF: a new dynamic memory allocator for real-time systems". In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* 2004, pp. 79–88. DOI: 10.1109/EMRTS.2004.1311009.

[Mei+20] Daniel Meister et al. "On Ray Reordering Techniques for Faster GPU Ray Tracing". In: *Symposium on Interactive 3D Graphics and Games*. I3D '20. San Francisco, CA, USA: Association for Computing Machinery, 2020. ISBN: 9781450375894. DOI: 10.1145/3384382.3384534. URL: https://doi.org/10.1145/3384382.3384534.

[Mei+21] Daniel Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum* 40 (May 2021), pp. 683–712. DOI: 10.1111/cgf.142662.

[MGM19] Zander Majercik, Jean-Philippe Guertin, and Morgan McGuire. "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields". In: *Journal of Computer Graphics Techniques Vol. 8, No. 2*. 2019. URL: https://www.jcgt.org/published/0008/02/01/paper-lowres.pdf.

[MGN17] Thomas Müller, Markus Gross, and Jan Novák. "Practical Path Guiding for Efficient Light-Transport Simulation". In: *Computer Graphics Forum (Proceedings of EGSR)* 36.4 (June 2017), pp. 91–100. DOI: 10.1111/cgf.13227.

[Mic20] Microsoft. *DirectX Raytracing (DXR) Functional Spec*. Jan. 11, 2020. URL: https://github.com/microsoft/DirectX-Specs/blob/master/d3d/Raytracing.md (visited on 09/10/2023).

[Mit07] Martin Mittring. "Finding next Gen: CryEngine 2". In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, pp. 97–121. ISBN: 9781450318235. DOI: 10.1145/1281500.1281671. URL: https://doi.org/10.1145/1281500.1281671.

[Mor+23] Nate Morrical et al. "Quick Clusters: A GPU-Parallel Partitioning for Efficient Path Tracing of Unstructured Volumetric Grids". In: *IEEE Transactions on Visualization and Computer Graphics* 29.1 (2023), pp. 537–547. DOI: 10.1109/TVCG.2022.3209418.

[MS20] Jim MacArthur and Martin Stich. *Profiling DXR Shaders with Timer Instrumentation*. Apr. 29, 2020. URL: https://developer.nvidia.com/blog/profiling-dxr-shaders-with-timer-instrumentation/ (visited on 08/20/2023).

[MU49] Nicholas Metropolis and S. Ulam. "The Monte Carlo Method". In: *Journal of the American Statistical Association* 44.247 (1949), pp. 335–341. ISSN: 01621459. URL: http://www.jstor.org/stable/2280232 (visited on 02/06/2024).

[Mül+18] Thomas Müller et al. "Neural Importance Sampling". In: *CoRR* abs/1808.03856 (2018). arXiv: 1808.03856. URL: http://arxiv.org/abs/1808.03856.

[Mül04] Gordon Müller. "Object Hierarchies for Efficient Rendering". en. PhD thesis. May 2004. DOI: 10.24355/dbbs.084-200511080100-421. URL: https://doi.org/10.24355/dbbs.084-200511080100-421.

[NHD10] Jan Novák, Vlastimil Havran, and Carsten Daschbacher. "Path Regeneration for Interactive Path Tracing". In: Eurographics Association, 2010, pp. 61–64.

[Nov+18] Jan Novák et al. "Monte Carlo methods for volumetric light transport simulation". In: *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports)* 37.2 (May 2018). DOI: 10/gd2jqq.

[Ouy+21] Y. Ouyang et al. "ReSTIR GI: Path Resampling for Real-Time Path Tracing". In: *Computer Graphics Forum* 40.8 (2021), pp. 17–29. DOI: https://doi.org/10.1111/cgf.14378. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14378. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14378.

[Pan18] Alexey Panteleev. *Rendering Perfect Reflections and Refractions in Path-Traced Games*. Aug. 13, 2018. URL: https://developer.nvidia.com/blog/rendering-perfect-reflections-and-refractions-in-path-traced-games/ (visited on 08/20/2023).

[Pet+16] Alexandru-Lucian Petrescu et al. "Analyzing Deferred Rendering Techniques". In: *Control Engineering and Applied Informatics* 18 (Mar. 2016), pp. 30–41.

[Pet21] Matt Pettineo. *The Shader Permutation Problem*. Oct. 11, 2021. URL: https://therealmjp.github.io/posts/shader-permutations-part1/ (visited on 02/01/2024).

[Rat+20] Alexander Rath et al. "Variance-Aware Path Guiding". In: *ACM Trans. Graph.* 39.4 (Aug. 2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392441. URL: https://doi.org/10.1145/3386569.3392441.

[Rat+22] Alexander Rath et al. "EARS: Efficiency-Aware Russian Roulette and Splitting". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2022)* 41.4 (July 2022). DOI: 10.1145/3528223.3530168.

[RH22] Matthew Rusch and Evan Hart. *Improve Shader Performance and In-Game Frame Rates with Shader Execution Reordering*. Oct. 12, 2022. URL: https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/ (visited on 10/20/2023).

[Sam84] Hanan Samet. "The Quadtree and Related Hierarchical Data Structures". In: *ACM Comput. Surv.* 16.2 (June 1984), pp. 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: https://doi.org/10.1145/356924.356930.

[Sch+17] Christoph Schied et al. “Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination”. In: *Proceedings of High Performance Graphics*. HPG ’17. Los Angeles, California: Association for Computing Machinery, 2017. ISBN: 9781450351010. DOI: 10.1145/3105762.3105770. URL: https://doi.org/10.1145/3105762.3105770.

[Sch19] Christoph Schied. *Video Series: Path Tracing for Quake II in Two Months*. May 28, 2019. URL: https://developer.nvidia.com/blog/path-tracing-quake-ii/ (visited on 08/09/2023).

[Sjo20] Juha Sjoholm. *Best Practices: Using NVIDIA RTX Ray Tracing (Updated)*. Aug. 10, 2020. URL: https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/ (visited on 11/09/2023).

[SKS11] Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz. *Secrets of CryENGINE 3 Graphics Technology*. Aug. 18, 2011. URL: https://advances.realtimerendering.com/s2011/ (visited on 11/07/2023).

[SPD18] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. “Gradient Estimation for Real-Time Adaptive Temporal Filtering”. In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233301. URL: https://doi.org/10.1145/3233301.

[Sta21] Tomasz Stachowiak. *ReSTIR meets Surfels*. Nov. 23, 2021. URL: https://gist.github.com/h3r2tic/ba39300c2b2ca4d9ca5f6ff22350a037 (visited on 02/06/2024).

[Str23] Filip Strugar. *How to Build a Real-time Path Tracer*. Mar. 20, 2023. URL: https://www.nvidia.com/en-us/on-demand/session/gtcspring23-S51871/?ncid=em-even-124008-vt33 (visited on 11/22/2023).

[SWP10] D. Scherzer, M. Wimmer, and W. Purgathofer. “A Survey of Real-Time Hard Shadow Mapping Methods”. In: *Eurographics 2010 - State of the Art Reports*. Ed. by Helwig Hauser and Erik Reinhard. The Eurographics Association, 2010. DOI: 10.2312/egst.20101060.

[TCE05] Justin Talbot, David Cline, and Parris Egbert. “Importance Resampling for Global Illumination”. In: *Eurographics Symposium on Rendering (2005)*. Ed. by Kavita Bala and Philip Dutre. The Eurographics Association, 2005. ISBN: 3-905673-23-1. DOI: 10.2312/EGWR/EGSR05/139-146.

[Ubi23] Ubisoft. *Anno*. Ubisoft. 2023. URL: https://www.ubisoft.com/en-gb/game/anno/1800 (visited on 08/10/2023).

[VG95] Eric Veach and Leonidas J. Guibas. “Optimally Combining Sampling Techniques for Monte Carlo Rendering”. In: *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 419–428. ISBN: 0897917014. DOI: 10.1145/218380.218498. URL: https://doi.org/10.1145/218380.218498.

[VG97] Eric Veach and Leonidas J. Guibas. “Metropolis Light Transport”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 65–76. ISBN: 0897918967. DOI: 10.1145/258734.258775. URL: https://doi.org/10.1145/258734.258775.

[Wal+07] Bruce Walter et al. “Microfacet Models for Refraction through Rough Surfaces”. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR’07. Grenoble, France: Eurographics Association, 2007, pp. 195–206. ISBN: 9783905673524.

[WB00] Colin Ware and Ravin Balakrishnan. “Reaching for Objects in VR Displays: Lag and Frame Rate”. In: *ACM Transactions on Computer-Human Interaction* 1 (May 2000). DOI: 10.1145/198425.198426.

[Wei+21] Philippe Weier et al. “Optimised Path Space Regularisation”. In: *Computer Graphics Forum* 40.4 (2021), pp. 139–151. DOI: https://doi.org/10.1111/cgf.14347. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14347. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14347.

[Whi80] Turner Whitted. “An Improved Illumination Model for Shaded Display”. In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.

[Wro14] Bartlomiej Wronski. *Assassin’s Creed 4: Black Flag. Road to next-gen graphics*. Mar. 17, 2014. URL: https://bartwronski.files.wordpress.com/2014/03/ac4_gdc.pdf (visited on 11/07/2023).

[Wro16] Bart Wronski. *Localized tonemapping – is global exposure and global tonemapping operator enough for video games?* Aug. 16, 2016. URL: https://bartwronski.com/2016/08/29/localized-tonemapping/ (visited on 02/02/2024).

[Wym+18] Chris Wyman et al. “Introduction to DirectX Raytracing”. In: *ACM SIGGRAPH 2018 Courses*. SIGGRAPH ’18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018. ISBN: 9781450358095. DOI: 10.1145/3214834.3231814. URL: https://doi.org/10.1145/3214834.3231814.

[YS11] Egor Yusov and Maxim Shevtsov. “High-Performance Terrain Rendering Using Hardware Tessellation.” In: *Journal of WSCG* 19 (Jan. 2011), pp. 85–92.

[Zhd20] Dmitry Zhdan. *Fast Denoising With Self Stabilizing Recurrent Blurs*. Mar. 18, 2020. URL: https://www.gdcvault.com/play/1026701/Fast-Denoising-With-Self-Stabilizing (visited on 09/11/2023).

[Zhd21] Dmitry Zhdan. “ReBLUR: A Hierarchical Recurrent Denoiser”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 823–844. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_49. URL: https://doi.org/10.1007/978-1-4842-7185-8_49.

[Zhu+21] Tao Zhuang et al. “Real-time Denoising Using BRDF Pre-integration Factorization”. In: *Computer Graphics Forum* 40.7 (2021), pp. 173–180. DOI: https://doi.org/10.1111/cgf.14411. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14411. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14411.

[Zim+15] Henning Zimmer et al. “Path-space Motion Estimation and Decomposition for Robust Animation Filtering”. In: *Computer Graphics Forum (Proceedings of EGSR)* 34.4 (June 2015). DOI: 10/f7mb34.