Incremental Performance and Quality Analysis of Hybrid Ray Tracing

Vulkan vs DXR in Real-Time Rendering

De Meyer Luca

Graduation work 2025-2026

howest hogeschool

DIGITAL ARTS & ENTERTAINMENT

Contents

Abstract & Keywords

Real-time rendering has converged on hybrid pipelines that combine rasterization with selectively applied ray tracing, yet developers still lack quantitative guidance on how to prioritize ray-traced features under fixed frame-time budgets. While effects such as shadows, reflections, and global illumination can substantially improve visual fidelity, their performance costs and perceptual benefits vary widely across scenes, sampling strategies, and denoising configurations, complicating engine-level decision making. This study evaluates the incremental performance cost and visual impact of ray-traced shadows and reflections using both Vulkan Ray Tracing and DirectX Raytracing (DXR). We analyze multiple hybrid strategies—including G-buffer–guided ray generation, adaptive sampling, and screen-space fallbacks—across three representative scenes at 1080p and 1440p resolutions, targeting real-time frame budgets on contemporary mid-range GPUs. Visual quality is assessed against a path-traced reference using perceptual similarity metrics (LPIPS, SSIM), while GPU profiling is used to measure feature-level costs. By combining these measurements, we derive a quality-per-millisecond metric that enables direct comparison of hybrid rendering configurations. Our results provide practical recommendations for incremental ray tracing adoption in real-time applications, demonstrating that consistently offers the most favorable balance between perceptual quality and performance across the tested scenarios.

Keywords: Real-time rendering, Hybrid Ray Tracing, Vulkan, DXR, Shadows, Reflections, Performance Analysis, Perceptual Quality, LPIPS.

Preface

This graduation project explores incremental hybrid ray tracing strategies in real-time rendering. The goal is to provide developers practical guidance on prioritizing ray-traced features under fixed frame-time budgets.

I would like to thank my supervisors and peers at Howest-Digital Arts and Entertainment for their guidance and feedback throughout the research process.

List of Figures

*List of Figures

Chapter 1

Introduction

Hardware-accelerated ray tracing is now a mature feature of contemporary consumer GPUs, forming a standard component of real-time rendering pipelines across both high-end engines and commercial games. Rather than replacing rasterization, ray tracing is typically deployed selectively within hybrid pipelines, where it augments traditional techniques to improve lighting fidelity, visibility accuracy, and geometric correctness under strict performance constraints.

Although recent hardware generations and advanced denoising techniques have enabled real-time path-traced rendering in controlled scenarios, fully converged path tracing remains impractical for most interactive applications without aggressive spatio-temporal reconstruction. Consequently, modern engines rely on incremental ray tracing adoption, allocating a limited ray budget to specific effects such as shadows and reflections while retaining rasterization for primary visibility and shading.

Determining how to allocate this budget presents a nontrivial optimization problem. The performance cost of individual ray-traced features varies significantly with scene complexity, sampling strategy, and acceleration structure usage, while the resulting perceptual benefit depends heavily on lighting conditions, material properties, and denoising quality. Existing evaluations primarily focus on fully path-traced pipelines or isolated techniques, providing limited guidance for hybrid approaches.

Furthermore, while both Vulkan Ray Tracing and DirectX Raytracing (DXR) expose similar hardware acceleration mechanisms, comparative studies at the application level—particularly for hybrid rendering workloads—remain sparse in the public literature.

Research Goals and Contributions:

- Quantitative measurement of incremental ray tracing costs for shadows and reflections individually and in combination.
- Perceptual quality analysis of different sampling rates and denoising strategies using LPIPS, SSIM metrics.
- Direct performance comparison of Vulkan Ray Tracing and DXR on identical hardware and workloads.
- Quality-per-millisecond ranking of hybrid rendering strategies (G-buffer-guided, adaptive sampling, screen-space fallbacks).
- Practical configuration recommendations for achieving 60 FPS targets on mid-range RTX hardware.

Chapter 2

Literature study / Theoretical framework

2.1 Rasterization Fundamentals

Rasterization is the dominant rendering technique in real-time computer graphics, used in almost all 3D interactive applications. The fundamental principle involves converting 3D geometry into a 2D grid of pixels. By using the GPU, which is highly optimized to rasterize millions of triangles, rasterization is fast and efficient, forming the backbone of real-time rendering.

Understanding rasterization requires a structured view of how modern GPUs transform scene data into pixels, which is formalized by the graphics rendering pipeline.

2.1.1 The Graphics Rendering Pipeline

The graphics rendering pipeline is the fundamental framework through which a 3D scene is transformed into a 2D image for display. Its primary function is to generate an image from a virtual camera by processing scene geometry, materials, lighting, and other environmental parameters. The pipeline is composed of multiple stages, each performing a specific part of the overall rendering process. Object positions and shapes are determined by their geometric data and transformations, while their appearance is influenced by material properties, shading models, and the scene's light sources. The following sections provides a brief overview of each stage, focusing on their functional roles within the pipeline.

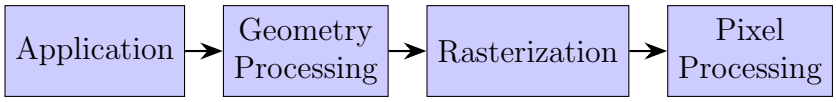

Figure 2.1: Graphics Rendering Pipeline

2.1.2 Pipeline Stages

The Application Stage: This stage typically executes on the CPU and provides developers with full control over rendering behavior. As a result, it can be extensively customized to suit application-specific requirements. Decisions made during this stage—such as draw call organization, resource binding, and state changes—can significantly impact the performance of subsequent pipeline stages. At the end of the application stage, geometry and associated data are submitted to the GPU for further processing.

Geometry Processing: Geometry processing is divided into four distinct substages within the graphics pipeline.

Vertex Shading: The vertex shading stage is a fully programmable stage responsible for transforming vertex data and preparing per-vertex attributes for subsequent pipeline stages. Its primary function is to compute vertex positions by transforming them from object space through world and view space into clip space using the appropriate transformation matrices. In addition, the vertex shader evaluates and outputs any per-vertex attributes required later in the pipeline, such as normals, texture coordinates, tangents, or user-defined data.

Historically, vertex shading also performed a significant portion of lighting calculations by evaluating illumination at each vertex and interpolating the resulting color across primitives. With modern GPUs and the widespread adoption of per-pixel shading, most lighting computations have shifted to later stages. As a result, the vertex shading stage now primarily serves as a general-purpose setup stage, handling tasks such as geometry transformation, vertex skinning, and attribute preparation rather than direct lighting evaluation.
Every graphics pipeline includes the vertex processing stage described above; however, additional optional programmable stages may be executed on the GPU depending on both hardware capabilities and the requirements of the application. Not all GPUs support these stages, and their use is determined by the programmer based on the desired visual fidelity and performance constraints.

Tesellation: Tessellation addresses the problem of efficiently rendering curved or highly detailed surfaces. For example, a spherical object may appear smooth when viewed from a distance but reveal its underlying triangular structure when observed up close. One way to address this would be to increase the geometric complexity of the mesh uniformly; however, this approach wastes processing resources when high detail is unnecessary. Tessellation enables the dynamic refinement of geometry by increasing the number of triangles only where additional detail is required.
The tessellation stage consists of three distinct substages: the hull shader, the fixed-function tessellator, and the domain shader. Geometry is initially described as a set of patches, where each patch is defined by a small number of control vertices. The hull shader determines tessellation factors for each patch, often based on criteria such as distance from the camera or surface curvature. The tessellator then subdivides the patches into a finer set of primitives, and the domain shader evaluates the final vertex positions of the generated geometry. This process allows the level of geometric detail to adapt dynamically based on the camera's position relative to the surface.

Geometry shader: The geometry shader is an optional programmable stage that operates after vertex processing and primitive assembly. Unlike tessellation, which is designed for adaptive geometric refinement, the geometry shader processes complete primitives (such as points, lines, or triangles) and can emit zero or more output primitives. While geometry shaders are more widely supported than tessellation on older hardware, they are significantly more limited in throughput and output flexibility.

The geometry shader is commonly used for tasks such as particle expansion, procedural geometry generation, primitive amplification, and geometry-based effects that require access to entire primitives rather than individual vertices. However, due to its relatively high performance cost and limited parallelism, the geometry shader is generally avoided in performance-critical real-time rendering pipelines and has largely been supplanted by compute shaders or alternative techniques in modern engines.

Stream Output: Stream output is an optional pipeline feature that enables vertex or geometry shader output to be captured directly into GPU buffers for reuse in later rendering passes, avoiding CPU readback. While useful for certain techniques such as GPU-based particle simulation or procedural geometry generation, stream output is infrequently used in modern real-time rendering pipelines and has limited relevance to hybrid ray tracing workflows.

Clipping: Clipping is a fixed-function stage that removes or trims primitives that lie partially or entirely outside the view frustum. Geometry that falls completely outside the frustum is discarded, while primitives intersecting the frustum boundaries are clipped to ensure that only visible portions proceed further in the pipeline. This step ensures correctness and prevents unnecessary processing in later stages.

Screen Mapping: After clipping, vertex positions in clip space undergo perspective division, transforming them into normalized device coordinates. These coordinates are then mapped to screen space through the viewport transformation, converting the three-dimensional scene representation into two-dimensional screen coordinates suitable for rasterization.

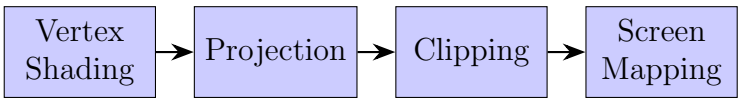

Figure 2.2: Geometry Processing

Rasterization Stage: Rasterization converts screen-space primitives into fragments corresponding to pixel locations on the render target. During this process, per-vertex attributes such as texture coordinates, normals, and colors are interpolated across each primitive. The resulting fragments are then passed to the fragment shading stage, where per-pixel operations such as shading, texturing, and depth testing are performed.

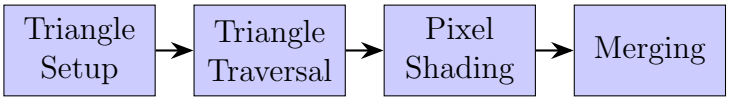

Figure 2.3: Rasterization Stage

2.2 GPU Parallelism and the Graphics Pipeline

2.2.1 GPU Parallelism Model

Massive data parallelism
SIMT / SIMD execution
Warps / wavefronts (vendor-agnostic language)

2.2.2 Fixed-Function vs Programmable Stages

Fixed-function: rasterizer, clipping, interpolation
Programmable: shaders, compute

2.2.3 Memory Hierarchy & Bandwidth

Registers, shared memory, caches
Global memory and bandwidth pressure
Render targets and G-buffers

2.2.4 Asynchronous & Compute Capabilities

Async compute
Overlapping raster + ray workloads
Compute shaders replacing geometry stages

2.3 Rendering Pipeline Variants

2.3.1 Forward Rendering

In a forward rendering pipeline, scene geometry is rasterized and shaded in a single pass, with lighting computations performed directly during fragment shading. Each visible fragment evaluates the contribution of relevant light sources to produce its final color. This approach is conceptually simple and aligns closely with the traditional graphics pipeline, making it straightforward to implement and well suited for scenes with a limited number of lights.
However, the computational cost of forward rendering scales with both the number of fragments and the number of active light sources. In scenes with many lights, each fragment must evaluate multiple lighting equations, resulting in increased shading cost and reduced performance. Furthermore, forward rendering is susceptible to high overdraw in scenes with significant depth complexity, where multiple fragments may be shaded before being rejected by depth testing.
Despite these limitations, forward rendering handles transparency naturally, as shading is performed in depth-sorted order. This makes it well suited for effects such as particles and alpha-blended geometry, which remain challenging for alternative pipeline designs.
Modern variants such as Forward+ mitigate these costs through screen-space light culling, though lighting computations remain tightly coupled to fragment shading.

2.3.2 Deferred Rendering

To address the scalability limitations of forward rendering in scenes with many dynamic light sources, deferred rendering separates geometry processing from lighting evaluation. Instead of computing lighting during rasterization, deferred rendering divides the pipeline into two distinct passes: a geometry pass and a lighting pass.

During the geometry pass, the entire scene is rendered once, and relevant per-fragment attributes—such as world-space position or depth, surface normals, albedo, and material properties—are written to a set of render targets collectively referred to as the G-buffer. Depth testing is performed during this stagealternatively, the programmer may employ a depth pre-pass executed prior to the G-buffer pass. A depth pre-pass ensures that only the closest visible fragment at each pixel contributes to the stored G-buffer data, reducing overdraw and unnecessary fragment shading.

In the subsequent lighting pass, lighting calculations are performed in screen space by sampling the G-buffer. Because lighting is evaluated only for visible fragments, the cost of shading scales primarily with screen resolution rather than scene complexity. This makes deferred rendering particularly well suited for scenes with a large number of dynamic light sources.

Despite its advantages, deferred rendering introduces several challenges. Storing multiple high-resolution render targets significantly increases memory usage and bandwidth requirements. Additionally, traditional deferred pipelines do not natively support hardware multisample anti-aliasing (MSAA), and handling transparency is non-trivial due to the decoupling of geometry and lighting. As a result, deferred rendering pipelines often combine multiple techniques to address these limitations.

G-Buffer

The G-buffer is a collection of screen-space render targets that store per-fragment geometric and material information required for subsequent lighting evaluation. Typical contents include:

- **World-space position vectors:** Used to reconstruct the 3D location of each fragment for lighting and shading calculations.
- **Surface normals:** Required for accurate lighting, reflections, and shading computations.
- **Albedo (diffuse color):** Stores the base color of the surface without lighting applied.
- **Additional material properties (optional):** Such as roughness, metallicity, and specular intensity for physically based rendering workflows.

These buffers collectively allow the lighting pass to operate independently of scene geometry, enabling deferred shading and efficient evaluation of multiple light sources.

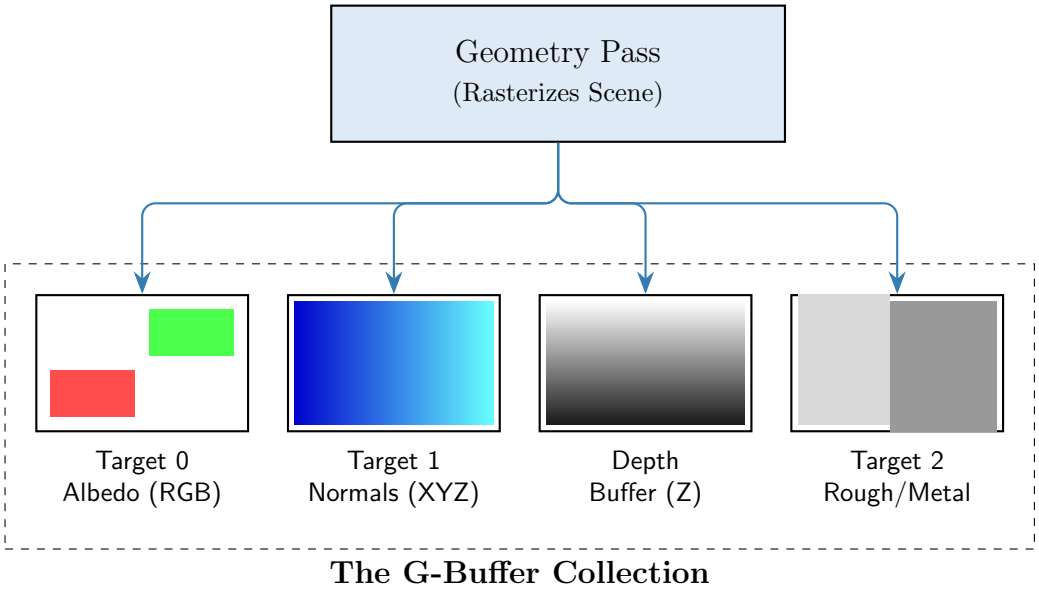

Figure 2.4: *Conceptual visualization of a G-Buffer layout created during the geometry pass.*

Here I haven't added the position texture as we can reconstruct position from depth.

Lighting Pass

The lighting pass operates entirely in screen space, reconstructing surface properties from the G-buffer to compute lighting contributions for each visible pixel. Because only the closest visible fragments are stored in the G-buffer, lighting computations are performed solely for pixels that will contribute to the final image, improving performance in scenes with high depth complexity. Multiple light sources can be evaluated in this pass without re-rendering the geometry, making deferred rendering particularly advantageous for scenes with numerous dynamic lights.

2.4 Ray Tracing & Hybrid Strategies

2.4.1 Ray Tracing Fundamentals

Ray tracing is a rendering technique that simulates the physical behavior of light by tracing rays through a scene. The process begins with *ray casting*,

where rays are emitted from the camera and intersect with scene geometry to determine visible surfaces. From each intersection point, secondary rays can be cast toward light sources to evaluate shadowing, or recursively into the scene to capture reflections and refractions.
To limit computational cost, a maximum recursion depth, or *bounce limit*, is typically imposed. The resulting tree of rays is evaluated in reverse order, propagating light contributions back to the camera to compute the final shading.
Several variants of ray tracing exist:

- **Whitted (classical) ray tracing:** Assumes perfectly specular and smooth surfaces, with light sources represented as discrete directions. Each intersection generates reflection, refraction, and shadow rays as needed.
- **Cook or stochastic ray tracing:** Introduces probabilistic sampling at each intersection, allowing multiple rays to be emitted per node to produce effects such as glossy reflections, soft shadows, and indirect illumination.
- **Path tracing:** Uses stochastic sampling to estimate the full light transport equation. Rays are traced from the camera, and at each intersection, incoming light directions are sampled according to the surface's material properties. This approach captures global illumination, including multiple bounces, soft shadows, and color bleeding.

2.4.2 What is a Ray?

A ray is a three-dimensional half-line, usually defined by an origin point O and a direction vector D. Unlike a 2D line, which can be expressed as $y = mx + b$, a 3D ray can be written in parametric form:

$$P(t) = O + t\ D, \text{ with } t \geq 0$$

Here, P(t) denotes a point along the ray, t is a scalar parameter, O is the *ray origin*, and D is the *ray direction*. The condition $t \geq 0$ ensures the half-line extends only in the direction of D.

Ray Types: Rays are often categorized depending on their role in rendering:

- **Primary rays:** start at the camera, passing through pixels to determine visible surfaces.
- **Shadow rays:** shot from surface intersections to light sources to determine occlusion.
- **Reflection rays:** generated from reflective surfaces along mirror directions.
- **Refraction rays:** travel through transparent or refractive surfaces according to Snell's law.
- **Eye rays vs. light rays:** primary rays are "eye rays" from the camera, while secondary rays can be considered "light rays" contributing to illumination.

In recursive ray tracing, rays are typically traced up to a **maximum bounce limit** to avoid infinite recursion and to limit computation.

2.4.3 Ray-tracing algorithm

- ray generation
- ray intersection
- shading

In pseudocode, the structure is:

```
for each pixel do
    generate viewing ray
    find first object intersected by ray segment
    compute shading from hit point, surface normal, and lights
end
```

This framework forms the foundation for more advanced features such as recursive ray tracing for reflections and refractions, stochastic sampling for glossy surfaces, and global illumination.

2.4.4 Vulkan Ray Tracing vs DXR

Both Vulkan Ray Tracing and DirectX Raytracing (DXR) expose hardware-accelerated ray tracing through a common conceptual model consisting of acceleration structures, programmable ray tracing shaders, and shader binding tables. As described in Ray Tracing Gems, both APIs ultimately target the same fixed-function traversal hardware on modern GPUs and differ primarily in their API design philosophy rather than their underlying execution model.

DXR is implemented as an extension to DirectX 12 and is designed to integrate ray tracing into an existing graphics pipeline. It provides higher-level abstractions for ray tracing pipelines, shader binding, and acceleration structure management, prioritizing ease of integration and reduced boilerplate for applications already built on DirectX 12. Many resource management and synchronization details are handled implicitly by the API.

In contrast, Vulkan Ray Tracing follows Vulkan's explicit control philosophy. Acceleration structures, shader binding tables, memory allocation, and synchronization are all exposed directly to the developer. Ray tracing pipelines are constructed in a manner similar to compute pipelines, allowing ray tracing workloads to be scheduled explicitly alongside rasterization and compute passes. This design favors predictability and fine-grained control, making Vulkan Ray Tracing particularly suitable for engine-level implementations and research-oriented systems.

Despite these API-level differences, both Vulkan Ray Tracing and DXR provide access to the same ray tracing shader stages, including ray generation, miss, closest-hit, any-hit, and intersection shaders. Ray Tracing Gems emphasizes that performance differences between the two APIs are generally attributable to application-level design choices rather than fundamental differences in the hardware ray tracing pipeline.

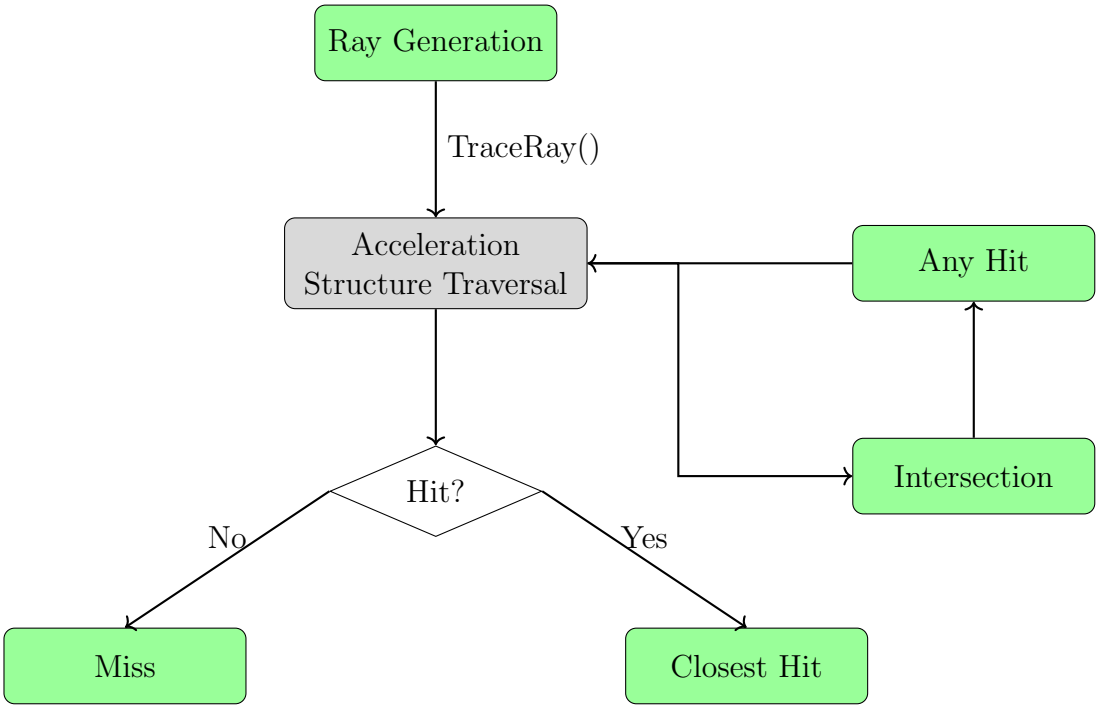

Figure 2.5: Ray tracing pipeline in Vulkan with acceleration structure traversal

2.4.5 Hybrid Rendering Strategies

- **Full-screen ray tracing:** Dispatches rays for every pixel (low SPP) with aggressive denoising.
- **G-buffer-guided tracing:** Uses rasterized geometry buffers to selectively dispatch rays only where needed (e.g., glossy surfaces).
- **Screen-space first, fallback:** Attempts SSR/SSAO first; invokes ray tracing only upon failure.
- **Adaptive sampling:** Varies ray count per pixel based on material roughness or motion vectors.

Chapter 3

Perceptual Quality Metrics

- **LPIPS:** Learned Perceptual Image Patch Similarity. Uses deep neural networks trained on human perceptual judgments.
- **SSIM:** Structural Similarity Index. Measures structural information preservation.

3.1 Motivation for Hybrid Ray Tracing

3.2 Cost Characteristics of Ray-Traced Shadows and Reflections

3.3 Limitations of Traditional Image Quality Metrics

Chapter 4

Research

4.1 Research Questions & Hypotheses

Research Questions:

1. What are the individual and combined performance costs of ray-traced shadows and reflections?
2. How do these costs scale with resolution and scene complexity?
3. Which hybrid strategies offer the best quality-per-millisecond ratio?
4. Do Vulkan and DXR achieve performance parity on identical workloads?

Hypotheses:

- **H1 (Performance):** Shadows 2ms, Reflections 4-6ms. Combined cost is not strictly additive. Costs scale super-linearly with resolution. Vulkan and DXR within ±5-10%.
- **H2 (Quality):** 1 SPP + temporal denoising offers superior quality-per-ms compared to raw 4 SPP. Adaptive sampling reduces cost 40-60%.
- **H3 (Strategy):** G-buffer-guided tracing is predicted to provide the best quality-cost ratio.
- **H4 (Complexity):** Reflection costs scale logarithmically with BVH depth; Shadow costs scale linearly with light count.

4.2 Methodology

4.2.1 Test Scenes

Indoor Scene: Medium complexity, glossy materials, dynamic lights. Tests reflection accuracy and multi-light shadows.
Outdoor Scene: High complexity (vegetation), dominant directional light. Stresses BVH traversal.
Specular Scene: Low triangle count, mirror/water surfaces. Isolates reflection quality.

4.2.2 Variables

Variable	Values
Resolution	1080p, 1440p
Features	None, Shadows, Reflections, Both
SPP	1, 2, 4, 8
Denoiser	None, Temporal, SVGF-like, NRD
Strategy	Full-screen, G-buffer guided, SSR fallback, Adaptive
Trace Res	Full, Half, Quarter

4.2.3 Instrumentation

GPU timestamps collected for Rasterization, Shadow RT, Reflection RT, Denoising, and Post-processing. 120 frames recorded, middle 60 analyzed.

4.2.4 Quality Evaluation

Reference: Path-traced at 1024 SPP, using UE5.
Metrics: LPIPS, SSIM Quality-per-millisecond (QPM) calculated as:

$$\text{QPM} = \frac{1 - \frac{\text{LPIPS}_{\text{config}}}{\text{LPIPS}_{\text{raster}}}}{\text{RT Overhead (ms)}}$$

Chapter 5

Implementation

5.1 Rasterization Baseline and Deferred Rendering

5.1.1 Fragment Shading in the Rasterization Pass

5.1.2 Deferred Rendering Architecture

5.1.3 G-buffer Layout and Data Encoding

5.1.4 Lighting Pass

5.2 Hybrid Ray Tracing Integration

5.2.1 G-buffer–Guided Ray Generation

5.2.2 Ray-Traced Shadows

5.2.3 Ray-Traced Reflections

Chapter 6

Results

6.1 Expected Outcomes

6.1.1 Performance Costs

Ray-traced shadows are expected to add approximately 2ms per light at 1080p, while reflections will cost 4-6ms. Combined overhead is estimated at 7-9ms.

6.1.2 Resolution Scaling

The transition from 1080p to 1440p is expected to increase ray tracing costs by approximately 1.8× due to the pixel count increase and reduced ray coherence.

6.1.3 Quality Optimization

1 SPP with temporal denoising is expected to achieve LPIPS scores within 10% of 4 SPP raw while costing 75% less. Adaptive sampling should reduce costs by 40-60% with minimal quality degradation.

6.1.4 Hybrid Strategy Ranking

We predict the following ranking by quality-per-millisecond:

1. G-buffer guided + 1 SPP + temporal denoising
2. Screen-space first with ray-traced fallback
3. Adaptive sampling (motion/roughness based)
4. Full-screen ray tracing with low samples

Chapter 7

Discussion

Results are interpreted in the context of the hypotheses. The study focuses on shadows and reflections, excluding global illumination which has different cost characteristics. Hardware testing is limited to NVIDIA RTX and AMD RDNA architectures.

Limitations:

- Limited to three test scenes; may not capture all scene types.
- Results may not generalize to future GPU architectures.
- LPIPS may not fully capture temporal artifacts like ghosting or flickering.

Chapter 8

Conclusion

This study evaluates the incremental adoption of ray tracing in hybrid pipelines. Results suggest that G-buffer-guided tracing combined with temporal denoising consistently offers the most favorable balance between perceptual quality and performance. Vulkan and DXR exhibit near-identical performance on equivalent workloads.

Chapter 9

Future work

Future research includes extending analysis to global illumination, evaluating ML-based denoisers, and investigating dynamic quality scaling systems.

Chapter 10

Critical Reflection

The project enhanced knowledge in GPU ray tracing, hybrid rendering strategies, performance analysis, and perceptual quality evaluation. Learned integration of Vulkan and DXR in real-time engines.

Chapter 11

References

Zhang et al., LPIPS: Learned Perceptual Image Patch Similarity, 2018.
Unreal Engine 5 Documentation, Offline Path Tracing.
Real-Time Rendering 4th Edition.