School of Computing
FACULTY OF ENGINEERING
AND PHYSICAL SCIENCES



Implementation of Light Propagation Volumes with modern rendering techniques

Kexuan Zhang

Submitted in accordance with the requirements for the degree of
MSc High-Performance Graphics and Games Engineering

2023/2024

The candidate confirms that the following have been submitted.

Submission	**Format**	**Recipient(s)**	**Date**
Project report	Report	SSO	12/08/24
C++ renderer	Source code (GitHub repository)	Supervisor, assessor	12/08/24

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

© 2023/2024 The University of Leeds and Kexuan Zhang

Summary

This dissertation presents an OpenGL implementation of Light Propagation Volumes (LPV) integrated into a deferred rendering pipeline. The project investigates how an approximate diffuse global illumination method can be combined with shadow mapping, screen-space effects and post-processing within a single interactive renderer.
The implementation builds on VGFW and a FrameGraph-based organisation of rendering passes. Reflective shadow maps provide the input for radiance injection, and low-order spherical harmonics represent directional lighting in a three-dimensional grid. The pipeline also incorporates cascaded shadow mapping, horizon-based ambient occlusion, screen-space reflections, fast approximate anti-aliasing, an ACES-inspired filmic tone-mapping curve and bloom. Depth-based position reconstruction and early termination of screen-space ray marching are used to limit storage and computation, although their individual performance benefits are not isolated experimentally.
The renderer is evaluated using the Crytek Sponza scene, visual comparisons and per-pass measurements obtained with Tracy Profiler. At a resolution of 1920×1080, the reported configuration achieves 175.4 frames per second with a frame time of 5.701254 ms on an NVIDIA GeForce RTX 4060 Laptop GPU. These measurements characterise the tested configuration; they do not establish performance across scenes or superiority over alternative global illumination methods.
The contribution is the implementation and integration of established techniques, together with an analysis of their behaviour and costs. The discussion identifies limitations of the LPV representation, screen-space visibility and the evaluation methodology, and outlines further work on occlusion, spatial resolution and systematic performance testing.

Acknowledgements

I would like to thank Dr. Rafael Kuffner dos Anjos for his invaluable guidance and assistance throughout the year and his comments during the progress meeting. I would also like to thank Dr. Markus Billeter for his excellent support and feedback throughout this project.

Contents

1 Introduction **1**
- 1.1 Project Aim . . . 1
- 1.2 Objectives . . . 2
- 1.3 Deliverables . . . 2
- 1.4 Ethical, legal, and social issues . . . 2

2 Background Research **3**
- 2.1 Related Work . . . 3
  - 2.1.1 Global Illumination Techniques . . . 3
  - 2.1.2 Shadow Techniques . . . 5
  - 2.1.3 Screen-space and Post-Processing Techniques . . . 6
  - 2.1.4 Conclusion . . . 9
- 2.2 Methods and Techniques . . . 9
  - 2.2.1 Graph-based Rendering Pipeline . . . 9
  - 2.2.2 Render Paths . . . 9
  - 2.2.3 Real-time Global Illumination Methods . . . 12
  - 2.2.4 Shadow Techniques . . . 14
  - 2.2.5 Screen-space Techniques . . . 16
  - 2.2.6 Post-Processing Techniques . . . 17
- 2.3 Choice of Methods . . . 18
  - 2.3.1 Selection Criteria . . . 18
  - 2.3.2 Chosen Methods . . . 18
  - 2.3.3 Integration Strategy . . . 19

3 Software Requirements and System Design **20**
- 3.1 Software Requirements . . . 20
  - 3.1.1 Development Environment . . . 20
  - 3.1.2 Graphics API . . . 20
  - 3.1.3 Core Libraries and Frameworks . . . 20
  - 3.1.4 Tools and Utilities . . . 21
- 3.2 System Design . . . 21
  - 3.2.1 System Architecture . . . 21
  - 3.2.2 Modules and Components . . . 22
  - 3.2.3 Workflow and Interaction . . . 23

4 Software Implementation **24**
- 4.1 VGFW: The Rendering Framework . . . 24
  - 4.1.1 Project Setup . . . 24
  - 4.1.2 Integration of Third-party Libraries . . . 25
  - 4.1.3 Shader Preprocessing . . . 26
  - 4.1.4 Deferred Rendering of Crytek Sponza . . . 26
- 4.2 Render Passes . . . 27
  - 4.2.1 Cascaded Shadow Mapping . . . 27

4.2.2 Reflective Shadow Map (RSM) Generation Pass . . . 28
4.2.3 LPV Radiance Injection Pass . . . 28
4.2.4 LPV Radiance Propagation Pass . . . 29
4.2.5 G-Buffer Generation Pass . . . 30
4.2.6 Gaussian Blur Pass . . . 31
4.2.7 Horizon-Based Ambient Occlusion (HBAO) Pass . . . 31
4.2.8 Deferred Lighting Pass . . . 32
4.2.9 Screen Space Reflections (SSR) Pass . . . 33
4.2.10 Blit Pass . . . 33
4.2.11 Bloom Pass . . . 34
4.2.12 ACES Tone-Mapping Pass . . . 34
4.2.13 Fast Approximate Anti-Aliasing (FXAA) Pass . . . 35
4.2.14 Final Composition Pass . . . 35
4.3 User Interface . . . 37
4.4 Documentation . . . 39

5 Software Testing and Evaluation 40
5.1 Debug and Profiling Tools . . . 40
5.1.1 RenderDoc . . . 40
5.1.2 Tracy Profiler . . . 40
5.2 Performance Testing . . . 41
5.2.1 Test Machine Environment . . . 41
5.2.2 Performance . . . 42
5.3 Results . . . 43
5.3.1 Visual Observations . . . 45
5.3.2 Performance Interpretation . . . 45
5.3.3 Evaluation Limitations . . . 46

6 Conclusions and Future Work 47
6.1 Conclusions . . . 47
6.2 Future Work . . . 47

References 48

Appendices 52

A External Material 53

B Project Repository 54

C Additional Code 55

D Additional Figures 56

Chapter 1

Introduction

Real-time global illumination is a challenging topic in the realm of real-time rendering. Global illumination accounts for light transported indirectly between surfaces, in addition to illumination received directly from light sources. Offline renderers commonly use ray tracing and Monte Carlo integration to estimate this transport, but the number of samples required for low-noise images can exceed an interactive frame budget. Hardware-accelerated ray tracing, including NVIDIA RTX [48], reduces some of these costs but introduces hardware requirements and does not eliminate the need for approximation. Rasterisation-based techniques therefore remain relevant when portability and predictable rendering cost are priorities. Unreal Engine's Lumen [19], for example, supports a software ray-tracing path alongside hardware ray tracing. Figure 1.1 illustrates a Lumen demonstration; it provides context rather than a comparison with the implementation evaluated in this dissertation.



Figure 1.1. Lumen technical demonstration introduced in Unreal Engine 5 [19].

1.1 Project Aim

The aim of this project is to implement Light Propagation Volumes (LPV) [35], a real-time approximation to diffuse indirect illumination that does not require dedicated ray-tracing hardware, and integrate it with complementary rendering techniques. These include Cascaded Shadow Maps (CSM) [17], Horizon-Based Ambient Occlusion (HBAO) [4], Screen-Space Reflections (SSR) [45, 6], Fast Approximate Anti-Aliasing (FXAA) [43], filmic tone mapping [2, 55, 46], and bloom [31]. The objective is to examine their

integration, visual behaviour and computational cost in a working renderer. The project does not propose a new global illumination algorithm or claim that post-processing resolves the underlying limitations of LPV.

1.2 Objectives

- Establish the theoretical basis of diffuse global illumination and the LPV representation.
- Review related methods and identify trade-offs relevant to the implementation.
- Define visual comparisons and profiling procedures for evaluating the renderer.
- Implement radiance injection and propagation using OpenGL within a reusable rendering framework.
- Integrate shadow mapping, screen-space effects and post-processing as separate rendering passes.
- Analyse the resulting images and measured pass timings, distinguishing observations from conclusions that require further experiments.

1.3 Deliverables

1. A GitHub repository that contains the source code and developer manual of the system.
2. The MSc project report.

1.4 Ethical, legal, and social issues

The project raises issues concerning visual comfort, software and asset attribution, and computational resource use. Unstable frame rates and flickering may affect visual comfort; configurable rendering parameters allow users to adjust the effects, although no user study is conducted here. Third-party software, models and images require appropriate attribution and compliance with their respective licences. The external resources used by the implementation are listed in Appendix A. Rendering also consumes computational resources and energy. Per-pass profiling helps identify expensive operations, but the performance measurements in this dissertation are not measurements of energy consumption or environmental impact.

Chapter 2

Background Research

2.1 Related Work

2.1.1 Global Illumination Techniques

Real-time global illumination methods approximate light transport under a restricted frame budget. Earlier approaches establish useful points of comparison. Radiosity [26] discretises diffuse interreflection as energy exchange between surface patches. Its conventional formulation assumes diffuse reflection and does not directly represent specular transport. Photon mapping [32] separates photon tracing from radiance estimation and can represent effects such as caustics. For both methods, the cost and quality depend on the discretisation or sampling strategy. They illustrate the broader trade-off between the accuracy of a transport approximation and the resources required to evaluate it [33].

Instant radiosity [38] approximates indirect illumination using Virtual Point Lights (VPLs) placed along sampled light paths. The resulting many-light problem can be evaluated using rasterisation, as illustrated in Figure 2.1. Updating the VPL set permits changes in lighting or geometry, but many VPLs may be required to reduce sampling artefacts. Visibility tests and the cost of evaluating numerous lights therefore remain important. This representation is relevant to LPV because reflective shadow-map samples provide a related set of secondary light sources.

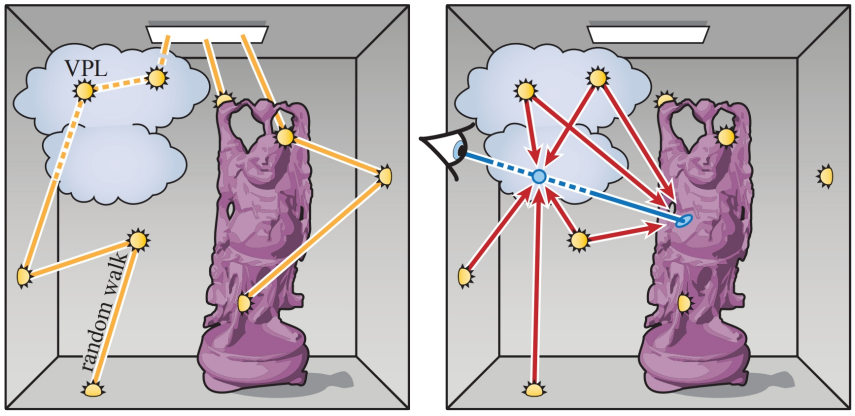


Figure 2.1. IR has two passes: first, it distributes a number of VPLs (left) and then uses them to illuminate the scene (right) [14].

Reflective Shadow Maps (RSMs) [15] extend light-view shadow maps with surface positions, normals and reflected flux. Each sample can be treated as a secondary emitter

for approximating indirect illumination. Rasterisation makes these samples inexpensive to generate, but the representation contains only surfaces captured from the light's view and does not by itself solve secondary visibility. Figure 2.2 shows the stored attributes and an example of their use.

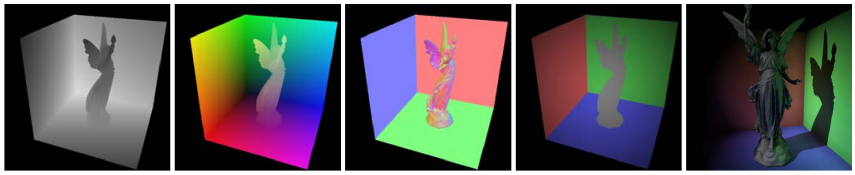


Figure 2.2. From left to right: depth, world space coordinates, normal, flux and the resulting image rendered with indirect illumination from the RSM [15].

Light Propagation Volumes (LPV) [35] store directional radiance in a regular three-dimensional grid and propagate contributions between neighbouring cells. Low-order spherical harmonics (SH) [54, 27] provide a compact directional representation. Samples from an RSM supply the initial radiance injection. The limited spatial and angular resolution makes the method appropriate for low-frequency diffuse indirect illumination, but can produce light leakage and coarse lighting. Sharp specular reflections require a different representation. Figure 2.3 illustrates the appearance of LPV in Sponza.

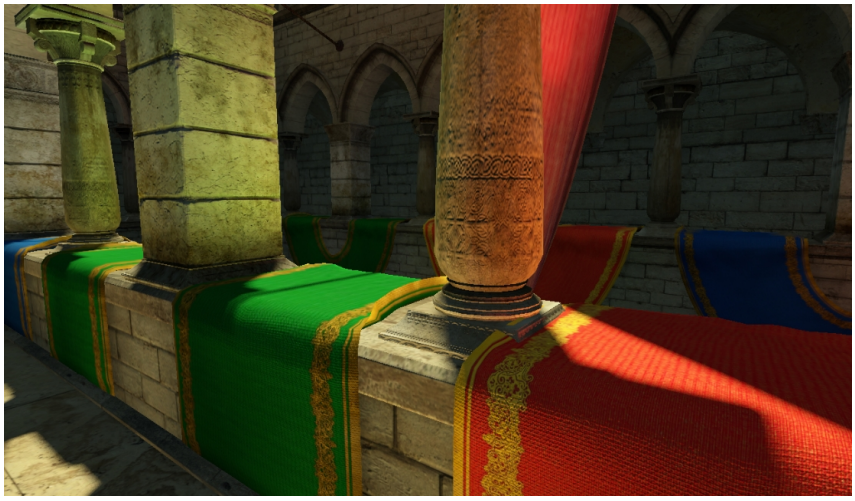


Figure 2.3. Crytek Sponza rendered with indirect illumination by LPV [36].

Voxel cone tracing [12] approximates indirect illumination by sampling a filtered voxel representation along cones. Increasing the sampling footprint with distance permits low-frequency diffuse transport and approximate glossy reflections to be evaluated using different cone apertures. The representation introduces costs for voxelisation, storage and filtering, and its quality depends on voxel resolution and directional filtering. Figure 2.4 shows an example. Voxel cone tracing is a rendering technique; NVIDIA's VXGI is a system based on related ideas, rather than an interchangeable name for every voxel cone-tracing method.

Dynamic Diffuse Global Illumination (DDGI) [44] updates an irradiance field using rays traced from spatial probes. The method stores direction-dependent irradiance and



Figure 2.4. Crytek Sponza rendered with indirect illumination using voxel cone tracing [12].

visibility information in probe textures, with temporal updates to reduce sampling noise [1]. Shading interpolates nearby probes using visibility-aware weights. This differs from propagating SH coefficients between LPV cells. DDGI targets diffuse indirect illumination; specular reflections require a complementary technique. Probe placement, visibility estimation and the number of rays affect both quality and cost. Figure 2.5 reproduces the comparison with a path-traced reference from the cited work.



Figure 2.5. DDGI compared with Path-traced reference [44].

2.1.2 Shadow Techniques

Shadow techniques are essential in computer graphics for enhancing visual realism. Here are some classic shadow techniques:
Shadow Mapping is a widely used real-time shadowing technique. It works by rendering the scene from the light source's perspective to generate a depth map (Shadow Map). During the main scene rendering, this depth map is used to compare the depth of each pixel to determine shadowed areas. While this method is straightforward and applicable to various light sources, it can suffer from aliasing and artefacts [62, 59].
Cascaded Shadow Mapping (CSM) is an enhancement to Shadow Mapping, particularly useful for shadows from directional lights in large scenes. CSM divides the view frustum

into multiple cascades, each with its own Shadow Map. This allows for high-resolution shadows close to the camera and lower-resolution shadows further away [17].
Shadow volumes use geometry-based techniques to create shadow volumes cast from a light source. During rendering, a stencil buffer is employed to determine whether pixels lie within a shadow region [13]. The result depends on geometric silhouettes rather than a sampled shadow map, although fill-rate and silhouette-processing costs can be substantial for complex geometry.
Ray-traced shadows evaluate visibility along rays directed towards a light source. Sampling an extended emitter can produce soft shadows, at a cost that depends on sampling, scene traversal and any denoising [5]. Hardware acceleration can reduce traversal costs, but it does not remove these quality and performance trade-offs.

2.1.3 Screen-space and Post-Processing Techniques

Screen-space techniques, such as Horizon-Based Ambient Occlusion (HBAO) [4] and Screen-Space Reflections (SSR) [45, 6], provide approximations of global illumination effects by operating directly on the rendered image. These methods are computationally efficient and well-suited for real-time applications but are limited by the information available in screen space, which can lead to inaccuracies and artefacts, particularly in scenes with significant occlusions or complex geometry. Figures 2.6 and 2.7 illustrate these effects.

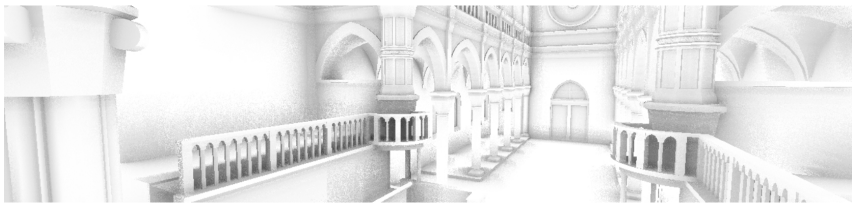


Figure 2.6. HBAO showcase [4].

Fast Approximate Anti-aliasing (FXAA) [43] is an anti-aliasing algorithm based on edge detection. It is employed as a post-processing step to mitigate the effects of jagged edges and enhance image quality by analysing the final frame. Given that it performs edge detection in image space, it is more efficient and well-suited for real-time rendering applications where performance is a primary consideration. Figure 2.8 illustrates the distinctions between various anti-aliasing algorithms.
Tone mapping maps high dynamic range (HDR) scene values to a range suitable for display. Reinhard et al. [55] provide a photographic operator, illustrated in Figure 2.9. ACES [2] is a broader colour-management framework that includes display rendering transforms. Real-time renderers also use compact filmic fits inspired by these transforms. The five-constant curve used in this project is Narkowicz's fit [46]; applying this curve alone is not a complete ACES workflow.

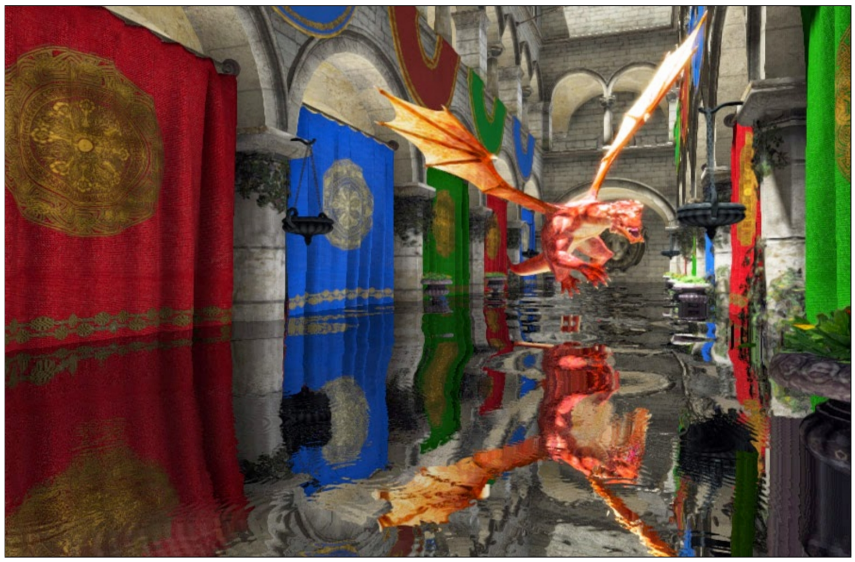


Figure 2.7. Reflections in ripply water approximated by a screen-space ray tracer [45].

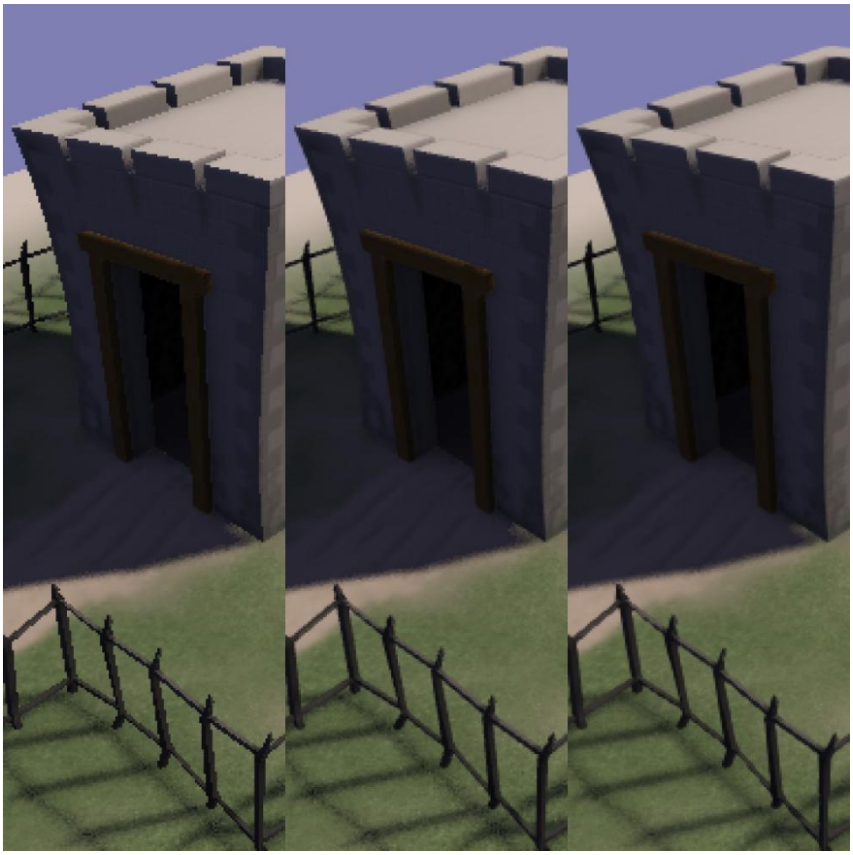


Figure 2.8. No AA (left) vs 4xMSAA vs FXAA (right) [43].

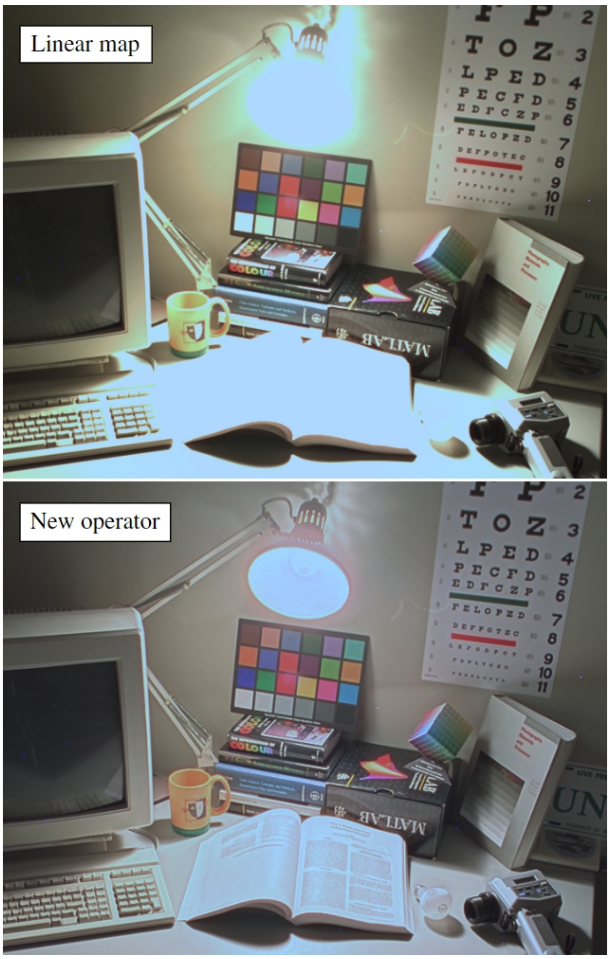


Figure 2.9. Reinhard tone-mapping (bottom) [55].

Bloom effects [31] are widely used to approximate the glow associated with bright image regions, giving bright areas of the image a glow that enhances the perception of brightness and realism. The implementation of Bloom can vary in complexity, but it generally involves blurring bright areas of the image and adding the result back to the original image, as demonstrated in previous work [42, 53, 20, 34]. The impact of applying the Bloom effect versus not applying it is illustrated in Figure 2.10.



Figure 2.10. Without (left) and with (right) bloom effect [42].

2.1.4 Conclusion

The reviewed methods address different components of image formation. LPV approximates low-frequency diffuse transport; shadow maps estimate direct-light visibility; HBAO and SSR supply screen-space approximations; and tone mapping, bloom and FXAA operate on the resulting image. Their integration can broaden the renderer's functionality, but the auxiliary effects do not remove LPV's transport and visibility limitations. This distinction guides the implementation and evaluation.

2.2 Methods and Techniques

2.2.1 Graph-based Rendering Pipeline

Graph-based rendering pipelines, often called frame graphs [52], represent passes and their resource dependencies explicitly. This representation supports reasoning about execution order, transient resource lifetimes and opportunities for reuse. A graph may also expose independent work, but actual parallel execution depends on the graphics API and scheduling implementation. Consequently, using a FrameGraph does not by itself establish a reduction in GPU time. In this project its principal role is to organise the dependencies between LPV and the complementary rendering passes.
FrameGraph provides a modular interface for declaring which resources each pass reads and writes. These declarations separate pass implementation from resource management and make the overall pipeline easier to inspect. O'Donnell discusses this architecture in the context of Frostbite [52]. Figure 2.11 shows the dependency structure used in a VGFW example; the project-specific graph is included in Appendix D.

2.2.2 Render Paths

Render paths refer to the sequence of rendering steps taken to produce the final image. Different render paths can be optimised for various scenarios, such as forward rendering [47] for simpler scenes and deferred rendering [16, 29, 41] for scenes with multiple light sources and complex shading. Advanced variants of these techniques, such as Forward+ [28], tiled deferred shading [50], tiled forward shading [7], and clustered deferred and forward shading [51], have been developed to address specific limitations of the traditional methods. These variants offer more efficient light management and better performance in complex scenes but come with increased implementation complexity. Understanding and choosing the appropriate render path is essential for achieving optimal performance and visual quality in real-time applications. To focus on the core aspects of LPV and other techniques, the implementation of the render path should be simplified.

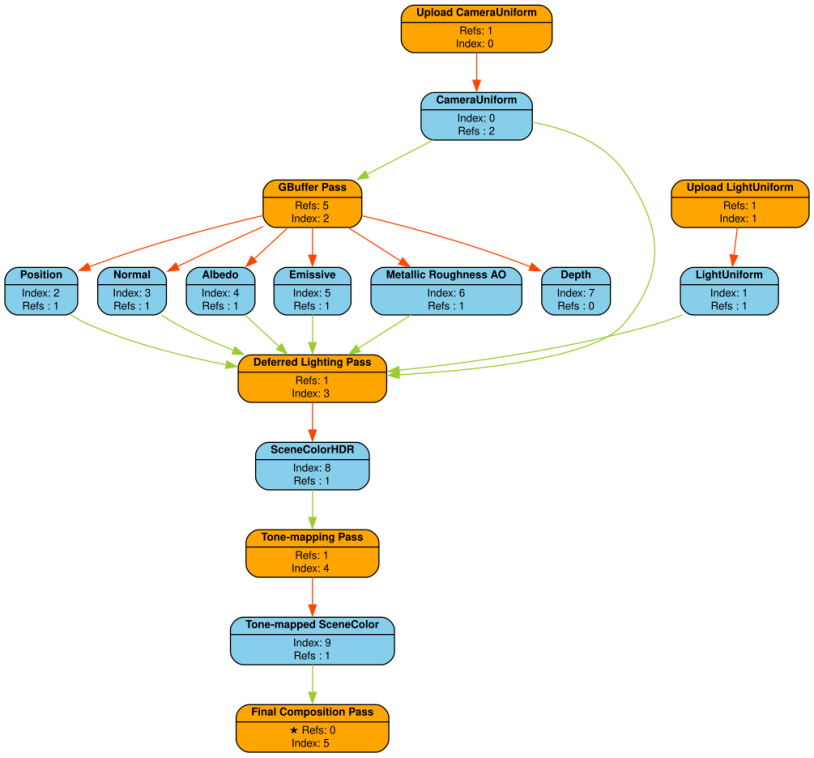


Figure 2.11. Example of a FrameGraph showing the dependencies and execution order of rendering tasks [64].

Forward Rendering

Forward rendering evaluates material shading and lighting while rasterising geometry [47], as illustrated in Figure 2.12. It has a relatively simple data flow and supports transparency naturally, but shading can be repeated for fragments later hidden by other geometry. Its lighting cost depends on how lights are assigned to fragments. Multiple passes are one possible strategy rather than an inherent requirement; tiled and clustered variants reduce the number of lights considered during shading.

Deferred Rendering

Deferred rendering separates the recording of surface attributes from lighting evaluation. A geometry pass writes depth, normals and material parameters to a G-buffer; later passes evaluate lighting using these attributes, as shown in Figures 2.13 and 2.14. This structure avoids repeating material evaluation for every light and provides reusable inputs for screen-space effects. The trade-off is additional storage and bandwidth for multiple render targets. Position can be reconstructed from depth instead of being stored explicitly. A conventional single-layer G-buffer represents one visible surface per pixel, so transparent layers require an additional strategy, often a separate forward pass. Whether deferred rendering is faster depends on the scene, light assignment and buffer layout.

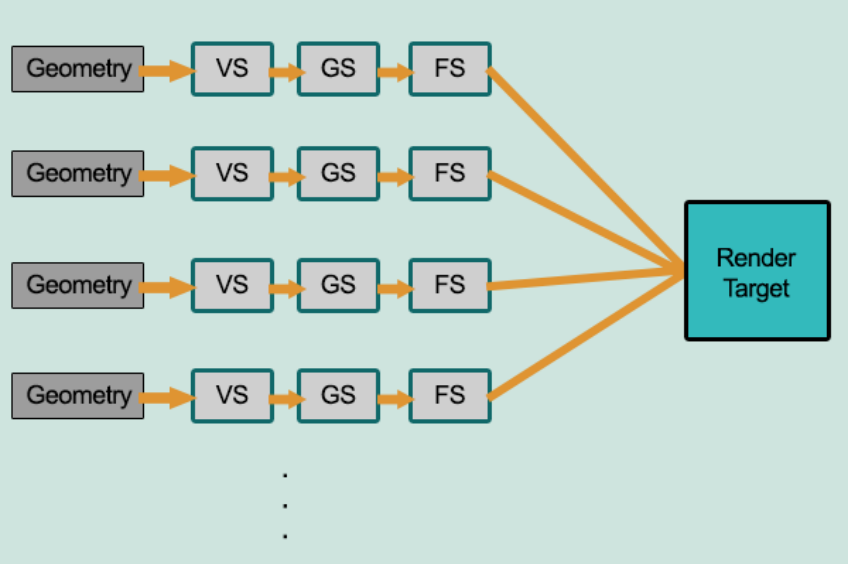


Figure 2.12. Forward Rendering Pipeline [18].

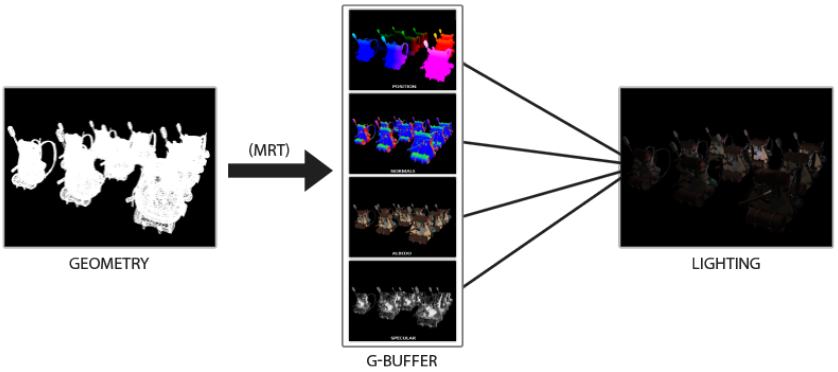


Figure 2.13. Deferred Rendering Pipeline [41].

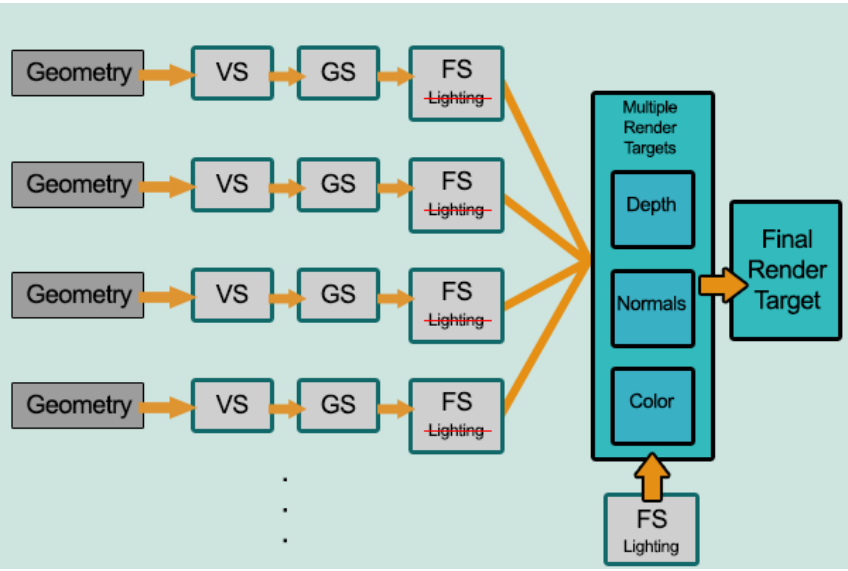


Figure 2.14. From Forward Rendering Pipeline to Deferred Rendering Pipeline [18].

2.2.3 Real-time Global Illumination Methods

Real-time Global Illumination (GI) techniques simulate the indirect light in a scene. Below are the methods with their core principles, corresponding mathematical formulations, and practical solutions.

Light Propagation Volumes (LPV)

Light Propagation Volumes (LPV) is a voxel-based method for approximating global illumination in real-time, particularly for dynamic scenes. The main stages are grid construction, radiance injection and radiance propagation.

Grid Discretisation The scene bounds define a regular grid with cell dimensions chosen to balance storage and spatial resolution. A world-space position is mapped to a cell using the grid origin and cell size. The LPV stores lighting coefficients in three-dimensional textures. This lighting grid should be distinguished from a full voxelisation of scene geometry: the implementation injects samples from the RSM and does not construct a complete occupancy representation.

Spherical Harmonics Representation In LPV, the light within each voxel is represented using Spherical Harmonics (SH), which approximates the radiance function $L(\omega)$ over the sphere:

$$L(\omega) \approx \sum_{l=0}^{L} \sum_{m=-l}^{l} c_{lm} Y_{lm}(\omega) \tag{2.1}$$

where $Y_{lm}(\omega)$ are the SH basis functions, and c_{lm} are the SH coefficients representing the lighting information. With a maximum degree of $L = 1$, the expansion contains two bands and four coefficients. One RGBA texture stores these four coefficients for a single colour channel; three such textures store RGB radiance. This low-order representation captures broad directional variation but cannot represent narrow specular lobes.

Radiance Injection and Propagation Radiance is injected using VPLs derived from the RSM. Each sample contributes a cosine-lobe projection oriented by its surface normal. Propagation then transfers directional contributions between neighbouring cells. For a coefficient vector $\mathbf{c}^{(k)}$ after iteration k, a schematic update is

$$\mathbf{c}^{(k+1)}(\mathbf{x}) = \sum_{\mathbf{d} \in \mathcal{N}} \mathcal{T}_{\mathbf{d}} \left[\mathbf{c}^{(k)}(\mathbf{x} - \mathbf{d}) \right] \tag{2.2}$$

where $\mathcal{N}$ contains the six axial neighbour offsets and $\mathcal{T}_{\mathbf{d}}$ denotes directional evaluation, solid-angle weighting and reprojection into SH. The implementation considers main and side-face directions, as described in Chapter 4. This transfer is not a scalar diffusion of

independent coefficients. Increasing the iteration count extends transport through the grid; an iteration is not equivalent to a physical surface bounce.

Voxel Cone Tracing

Voxel cone tracing uses voxelisation, hierarchical filtering and cone-shaped sampling footprints to approximate indirect illumination.

Voxelisation The scene is voxelized into a 3D grid where each voxel stores material properties $M(\mathbf{x})$ and light intensity $I(\mathbf{x})$. Light transport is simulated by tracing cones through the voxel grid.

Cone Tracing A cone represents an expanding sampling footprint rather than a single infinitesimal ray. Its angular filtering can be described conceptually by the mean radiance over the cone's solid angle Ω_c:

$$\overline{L}_c(\mathbf{x}) = \frac{1}{|\Omega_c|} \int_{\Omega_c} L(\mathbf{x}, \omega')\, d\omega' \tag{2.3}$$

where $|\Omega_c|$ is the cone's solid angle. Practical voxel cone tracing approximates a filtered transport query using hierarchical texture samples along the cone and opacity-aware accumulation; the equation expresses the angular average, not a complete implementation of that procedure.

Rendering The final image combines direct lighting with light calculated from cone tracing, capturing both diffuse and specular reflections.

Dynamic Diffuse Global Illumination (DDGI)

Dynamic Diffuse Global Illumination (DDGI) efficiently calculates diffuse global illumination using irradiance probes.

Irradiance Computation The irradiance $E(\mathbf{p})$ at a point $\mathbf{p}$ is computed by integrating the incoming radiance $L_i(\omega)$ over the hemisphere Ω:

$$E(\mathbf{p}) = \int_{\Omega} L_i(\omega) \cdot (\omega \cdot \mathbf{n})\, d\omega \tag{2.4}$$

where $\mathbf{n}$ is the normal at point $\mathbf{p}$.

Directional Probe Representation The DDGI formulation stores directional irradiance and distance moments in octahedrally mapped probe textures [44]. For probes

indexed by j, shading at position $\mathbf{p}$ with normal $\mathbf{n}$ can be expressed schematically as a normalised interpolation:

$$E(\mathbf{p}, \mathbf{n}) \approx \frac{\sum_j w_j(\mathbf{p}, \mathbf{n}) E_j(\mathbf{n})}{\sum_j w_j(\mathbf{p}, \mathbf{n})} \tag{2.5}$$

where E_j is a directional probe query and w_j includes spatial, normal and visibility terms. The expression assumes a non-zero total weight and summarises the interpolation rather than all implementation details. Unlike LPV, the method does not propagate four-coefficient SH vectors through neighbouring grid cells.

Temporal Accumulation Temporal accumulation blends a new probe estimate with its stored history:

$$E_{\text{new}}(\mathbf{p}) = (1 - \alpha) E_{\text{old}}(\mathbf{p}) + \alpha E_{\text{current}}(\mathbf{p}) \tag{2.6}$$

where $\alpha \in [0, 1]$ weights the current estimate. More history can reduce noise, but may slow the response to lighting changes. This probe-history update should not be confused with image-space motion-vector reprojection.

2.2.4 Shadow Techniques

Shadow Mapping

Depth Map Creation Shadow Mapping begins by rendering the scene from the perspective of the light source to create a depth map, known as the shadow map. This depth map records the distance from the light source to the nearest surface at each point in the light's view.

Shadow Determination During scene rendering, the surface point is transformed into light space and its depth is compared with the corresponding shadow-map sample. The compared depths must use the same projection convention. For a pixel $\mathbf{p}$ with depth d_{p} and the corresponding depth in the shadow map $d_{\text{shadow}}(\mathbf{p})$, the shadow test is defined as:

$$\text{Shadow}(\mathbf{p}) = \begin{cases} 0 & \text{if } d_{\text{p}} > d_{\text{shadow}}(\mathbf{p}) \text{ (in shadow)} \\ 1 & \text{if } d_{\text{p}} \leq d_{\text{shadow}}(\mathbf{p}) \text{ (not in shadow)} \end{cases} \tag{2.7}$$

Artifact Mitigation To address artefacts such as aliasing and shadow acne, techniques such as Percentage-Closer Filtering (PCF) are used to smooth shadow edges. Depth Biasing is applied to reduce self-shadowing by offsetting the depth values and higher-resolution shadow maps are employed to minimize aliasing effects.

Cascaded Shadow Mapping (CSM)

Frustum Division Cascaded Shadow Mapping enhances shadow quality in large scenes by dividing the view frustum into multiple regions, or cascades, each with its own shadow map. Allocating separate maps permits a greater sampling density near the camera without requiring a uniformly dense map over the entire view frustum.

Cascade Selection During rendering, the appropriate shadow map is selected based on the depth of each pixel relative to the cascades. This method provides detailed shadows where needed and efficiently handles larger scenes.

Transition Management Techniques such as soft transitions blend shadows between cascades to avoid abrupt quality changes, and adaptive resampling adjusts the shadow map resolution based on distance from the camera, balancing performance and visual fidelity.

Shadow Volumes

Volume Creation Shadow Volumes involve generating geometric volumes from the silhouette edges of objects relative to the light source. This volume, or shadow volume, extends away from the light and is used to determine shadowed regions.

Stencil Buffer Update The stencil buffer is updated to mark pixels as inside or outside the shadow volume. The pixel is considered in shadow if it is inside the shadow volume based on the stencil count.

Performance Optimization Silhouette clipping is used to limit the shadow volume to relevant areas, reducing computational overhead. Efficient stencil buffer usage minimizes performance impact while maintaining accuracy.

Raytraced Shadows

Ray Casting Raytraced Shadows compute shadows by casting rays from a point $\mathbf{p}$ towards the light source. The rays are checked for intersections with scene geometry to determine if the light path is obstructed.

Shadow Calculation For an area emitter with surface A_L, a normalised mean visibility can be written as:

$$\overline{V}(\mathbf{p}) = \frac{1}{|A_L|} \int_{A_L} V(\mathbf{p}, \mathbf{l})\, dA(\mathbf{l}) \tag{2.8}$$

where $V(\mathbf{p}, \mathbf{l})$ is one when the segment to emitter point $\mathbf{l}$ is unoccluded and zero otherwise. This expression describes mean visibility, not the full direct-lighting integral, which also

includes emission, geometry and reflectance. A point-light visibility query reduces to a single binary test.

Computational Management To handle the computational cost, approximate raytracing methods or hybrid techniques combining raytracing with shadow mapping are used. Adaptive sampling adjusts the number of rays based on scene complexity, balancing quality and performance.

2.2.5 Screen-space Techniques

Screen-space techniques approximate various effects by operating directly on the rendered image.

Horizon-Based Ambient Occlusion (HBAO)

Horizon-Based Ambient Occlusion (HBAO) is a screen-space method for approximating ambient occlusion.

Ambient Occlusion Calculation The ambient occlusion $A(\mathbf{p})$ at a point $\mathbf{p}$ is calculated by sampling the depth buffer and computing the occlusion due to nearby geometry:

$$A(\mathbf{p}) = 1 - \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega)(\omega \cdot \mathbf{n})\, d\omega \tag{2.9}$$

where $V(\mathbf{p}, \omega)$ is one for an unoccluded direction and zero for an occluded direction. With this convention, $A = 0$ denotes no occlusion and $A = 1$ complete occlusion.

Horizon Search HBAO approximates this integral by searching for horizon angles in the depth buffer, which describe the maximum elevation of occluding geometry along each sampled screen-space direction.

Screen-Space Reflections (SSR)

Screen-space reflections (SSR) approximate reflections by tracing rays in the screen space.

Ray Tracing in Screen Space SSR traces a reflection ray $R(\mathbf{p}, \mathbf{v})$ from a surface point $\mathbf{p}$ along the reflected direction $R(\mathbf{p}, \mathbf{v})$, derived from the incident view ray and the surface normal and finds the intersection with the scene in screen space:

$$\mathbf{p}' = \mathbf{p} + t \cdot R(\mathbf{p}, \mathbf{v}) \tag{2.10}$$

where t parameterises distance along a normalised reflected ray in view space. Sample positions are projected to screen space and tested against the depth buffer. When a valid intersection is found, the corresponding scene colour approximates the reflected radiance.

Limitations and Artifacts Since SSR relies on a single screen-space depth layer, it cannot recover off-screen or occluded surfaces. Disocclusions, depth thickness assumptions and screen boundaries can therefore produce missing reflections or discontinuities.

2.2.6 Post-Processing Techniques

Post-processing techniques enhance the visual quality of the final rendered image through various image-based operations.

Fast Approximate Anti-Aliasing (FXAA)

Fast Approximate Anti-Aliasing (FXAA) is a post-processing technique that performs anti-aliasing based on edge detection and smoothing corrections.

Edge Detection FXAA uses neighbourhood luminance contrast to guide filtering. A simple contrast test can be expressed as:

$$\text{Edge}(\mathbf{p}) = |L(\mathbf{p}) - L(\mathbf{p} + \delta\mathbf{x})| > \epsilon \tag{2.11}$$

where $L(\mathbf{p})$ is the luminance at point $\mathbf{p}$, $\delta\mathbf{x}$ is a small offset, and ϵ is the threshold.

Edge Smoothing Once edges are detected, colour samples are combined along an estimated edge direction. A general weighted filter is:

$$C_{\text{smoothed}}(\mathbf{p}) = \sum_{i=1}^{n} w_i C(\mathbf{p} + \delta\mathbf{x}_i), \qquad \sum_{i=1}^{n} w_i = 1 \tag{2.12}$$

where C is image colour, w_i are filter weights and $\delta\mathbf{x}_i$ are offsets. This is a schematic filtering expression; FXAA uses luminance-dependent sample selection and blending rather than an unconditional box average.

ACES Tone Mapping

The implementation uses the compact ACES-inspired filmic fit described by Narkowicz [46] to map HDR values into a displayable range.

ACES Tone Mapping Curve The fitted curve compresses scene-linear RGB values component-wise:

$$C_{\text{tone}} = \frac{C \cdot (a \cdot C + b)}{C \cdot (c \cdot C + d) + e} \tag{2.13}$$

where C is an input colour component and the fitted constants are $a = 2.51$, $b = 0.03$, $c = 2.43$, $d = 0.59$ and $e = 0.14$. The shader clamps the result to $[0, 1]$. These are parameters of the approximation, not a specification of the complete ACES transform.

Color Management A full ACES workflow also specifies colour spaces and display transforms. The fitted curve alone does not perform those operations or establish colourimetric accuracy; display encoding must be considered separately.

Bloom

Bloom is a post-processing effect that simulates light scattering, creating a glow around bright areas.

Gaussian Blur The bloom effect is typically created by applying a Gaussian blur to bright areas of the image:

$$I_{\text{bloom}}(\mathbf{p}) = \sum_{i=1}^{n} w_i \cdot I(\mathbf{p} + \delta\mathbf{x}_i) \tag{2.14}$$

where $I(\mathbf{p})$ is the intensity at point $\mathbf{p}$, w_i are Gaussian weights, and $\delta\mathbf{x}_i$ are offsets around the pixel.

Additive Blending The blurred result is then added back to the original image to create the final bloom effect:

$$I_{\text{final}} = I_{\text{original}} + \beta I_{\text{bloom}} \tag{2.15}$$

where $\beta \geq 0$ controls the bloom contribution.

2.3 Choice of Methods

The choice of methods for this project is driven by the need to balance performance and visual fidelity while leveraging advanced rendering techniques.

2.3.1 Selection Criteria

The selection criteria for the methods used in this project include performance efficiency, ease of integration, and the ability to handle complex lighting scenarios. The chosen methods expose parameters that trade sampling or grid resolution against cost. Their suitability for the target frame budget is assessed through the implemented configuration rather than assumed from the method name.

2.3.2 Chosen Methods

Based on the selection criteria, the following methods have been chosen for implementation:

- **Deferred Rendering** to efficiently manage multiple light sources and to facilitate the realisation of individual post-processing effects.
- **Light Propagation Volumes** to provide real-time dynamic global illumination with decent performance.
- **Cascaded Shadow Mapping with Percentage-Closer Filtering** to provide filtered directional-light shadows. PCF smooths shadow-map edges rather than modelling an area-light penumbra.
- **Horizon-Based Ambient Occlusion and Screen-Space Reflections** to approximate local occlusion and reflections, with their costs evaluated separately.
- **Fast Approximate Anti-Aliasing** to reduce visible spatial aliasing in the final image.
- **ACES Tone Mapping** to compress HDR values using a filmic curve.
- **Bloom** to enhance the perception of brightness and realism.

2.3.3 Integration Strategy

Each selected method is implemented as a separate pass with declared inputs and outputs. The deferred G-buffer supplies depth, normals and material parameters for lighting and screen-space queries. LPV supplies the diffuse indirect term, while CSM estimates direct-light visibility. HBAO and SSR complement these terms using visible screen-space information. Bloom operates on HDR scene colour, followed by filmic tone mapping and FXAA. This modular arrangement permits intermediate results to be inspected and effects to be enabled independently. The quality and cost of the combined renderer are evaluated in Chapter 5; the architecture alone does not establish a performance improvement.

Chapter 3

Software Requirements and System Design

3.1 Software Requirements

This section outlines the necessary software components and tools required to develop and implement LPV for real-time global illumination.

3.1.1 Development Environment

- **Operating System**: The renderer requires a driver and OpenGL context compatible with the GLSL 4.60 shaders used by the implementation. The reported experiments use Windows 11. The portability of supporting libraries should not be taken as evidence that the complete renderer has been validated on every platform.
- **Integrated Development Environment (IDE)**: The C++20 source can be edited using tools such as Visual Studio, CLion or Visual Studio Code. The build configuration is maintained separately in xmake rather than depending on an IDE-specific project.
- **Version Control System**: Git, along with a platform such as GitHub or GitLab, is essential for managing code changes, collaboration, and version tracking.

3.1.2 Graphics API

- **OpenGL**: OpenGL supplies rasterisation, programmable shaders, framebuffer attachments and three-dimensional textures. These capabilities support RSM generation, layered radiance injection and propagation, deferred shading and screen-space passes without requiring a hardware ray-tracing API.

3.1.3 Core Libraries and Frameworks

- **GLM(OpenGL Mathematics)**: A header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications [56].
- **GLFW(Graphics Library Framework)**: A library for creating windows, contexts, and handling input, used to manage the rendering context and user input [24].
- **GLAD**: An open-source library that loads OpenGL functions at runtime. It simplifies managing different versions and extensions of OpenGL, exposing the function pointers provided by the active graphics driver [30].

- **shaderc**: A shader toolchain whose `glslc -E` mode is used here to preprocess GLSL include directives before OpenGL compilation [25].
- **FrameGraph**: A framework for managing rendering tasks in a graph-based structure, with explicit pass-resource dependencies [58].
- **imgui**: A popular, immediate-mode graphical user interface (GUI) library for C++. It is used to create simple and intuitive user interfaces for real-time applications, such as debugging tools and in-engine editors [11].
- **stb_image**: A single-header library that provides a simple API for loading images in various formats, such as PNG, JPEG, and BMP, making it easy to handle texture data in OpenGL applications [3].
- **tinyobjloader**: A small, simple, and fast C++ library for loading Wavefront .obj files, commonly used for loading 3D models in graphics applications [22].
- **tinygltf**: A C++ library for loading and saving glTF (GL Transmission Format) files, which are used to transmit 3D models and scenes efficiently. It supports both binary and JSON-based glTF formats [21].

3.1.4 Tools and Utilities

- **RenderDoc**: A graphics debugger that allows frame capture and inspection of OpenGL, Vulkan, and Direct3D applications.
- **xmake** [63]: A cross-platform build utility based on Lua.
- **Tracy Profiler** [60]: A frame profiler for performance profiling and optimisation.

3.2 System Design

This section details the architectural design and components of the LPV technique, describing how the different modules interact to achieve real-time global illumination.

3.2.1 System Architecture

The architecture separates scene data, rendering resources and pass execution. Scene data provide meshes, materials, transforms and lighting parameters. FrameGraph declarations express which resources each pass reads and writes, while the render context submits the corresponding OpenGL operations. This separation permits the LPV passes to share inputs with shadow mapping and deferred shading without embedding all effects in a single shader. Figure 3.1 summarises the main data flow.

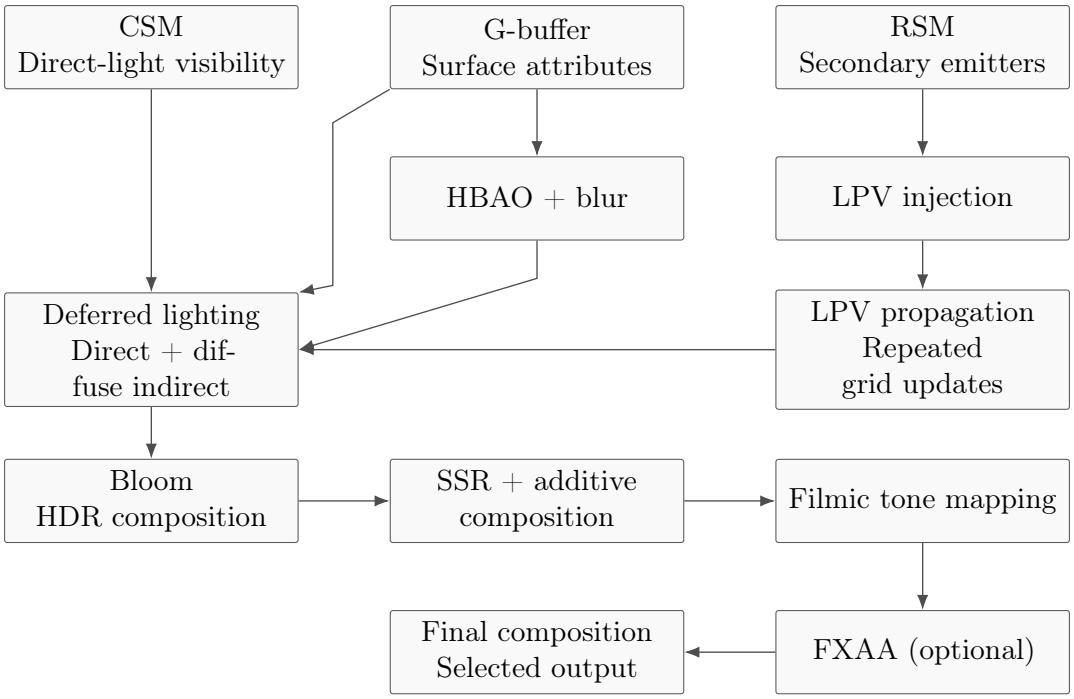


Figure 3.1. Overview of the implemented rendering pipeline. Arrows summarise principal data dependencies rather than parallel GPU execution. Auxiliary resource edges and debug outputs are omitted; the complete exported graph is retained in Appendix D. Bloom and SSR are optional, and disabling FXAA selects the tone-mapped output directly.

3.2.2 Modules and Components

Scene Management

The Scene Management module is responsible for:

- **Object Management**: Managing 3D objects, including their geometries, materials, and transformations.
- **Light Sources**: Providing light parameters such as direction and intensity. The evaluated RSM and shadow-map configuration uses a directional light; the design description does not establish support for every light type.

Rendering Engine

The Rendering Engine is the core module that handles the rendering process:

- **Frame Graph**: Using a FrameGraph to declare rendering tasks and resource dependencies (see Figure 2.11).
- **Deferred Rendering**: Implementing deferred rendering techniques to efficiently handle multiple light sources and complex shading.
- **LPV Implementation**: Integrating the LPV technique to propagate light through the voxel grid and calculate global illumination.

3.2.3 Workflow and Interaction

The workflow within the LPV technique can be summarised as follows:

- **Scene Setup**: Load and initialize scene objects, lights, and materials.
- **Grid Setup and Radiance Injection**: Define the lighting grid over the scene bounds and inject contributions from RSM samples into its cells.
- **Light Radiance Propagation**: Propagate light radiance through the voxel grid using the LPV technique, updating radiance values.
- **Rendering**: Render the scene using deferred rendering, applying LPV-based global illumination.
- **Post-Processing**: Enhance the final image with modern techniques.

The design provides a modular setting in which LPV and the auxiliary effects can be implemented and inspected. The resource dependencies establish the required data flow; measurements in Chapter 5 characterise the cost of the resulting renderer.

Chapter 4

Software Implementation

4.1 VGFW: The Rendering Framework

This project implements an open-source rendering framework written in C++, **VGFW** (V Graphics Framework) [64], designed for rapid prototyping and suitable for the development of rendering techniques. It is designed as a header-only library, making it easy to integrate into projects. This project uses VGFW to further develop and implement LPV and other rendering techniques, resulting in a lightweight renderer.

4.1.1 Project Setup

xmake [63] is used to define and build the C++20 project. Its Lua configuration describes targets, language settings, include paths and package requirements. Dependencies are resolved through xmake's package system, while header-only targets expose their headers and dependency information to consumers. Keeping these settings in the build configuration makes the example less dependent on a particular IDE and provides a consistent way to prepare the rendering framework and its test applications. The following configuration shows how to set up a project, a header-only library and its dependencies using XMake:

```
-- set project name
set_project("vgfw")

-- set project version
set_version("1.0.0")

-- set language version: C++ 20
set_languages("cxx20")

-- ......

-- add requirements
add_requires("fg", "glad", "glfw", "glm", "spdlog", "stb", "tinyobjloader",
    "tinygltf")
add_requires("imgui v1.90.8-docking", {configs = {glfw = true, opengl3 = true,
    wchar32 = true}})

-- target defination, name: vgfw
target("vgfw")
```

```
    -- set target kind: header-only
    set_kind("headeronly")

    add_includedirs(".", { public = true })

    -- add header files
    add_headerfiles("vgfw.hpp")

    add_rules("utils.install.cmake_importfiles")
    add_rules("utils.install.pkgconfig_importfiles")

    -- add packages
    add_packages("fg", { public = true })
    add_packages("glad", { public = true })
    add_packages("glfw", { public = true })
    add_packages("glm", { public = true })
    add_packages("imgui", { public = true })
    add_packages("spdlog", { public = true })
    add_packages("stb", { public = true })
    add_packages("tinyobjloader", { public = true })
    add_packages("tinygltf", { public = true })

-- ......
```

4.1.2 Integration of Third-party Libraries

The implementation uses spdlog [23] for diagnostic logging. GLFW [24] creates the application window, handles input and provides the OpenGL context, while GLAD [30] loads the graphics function pointers exposed by that context. GLM (OpenGL Mathematics) [56] supplies vector and matrix operations for transformations, camera calculations and the lighting grid. These libraries provide infrastructure for the renderer; the rendering passes and their resource dependencies are described separately below. FrameGraph [58] organises pass dependencies and rendering-resource access. Image loading uses stb_image [3], while tinyobjloader [22] and tinygltf [21] provide mesh and material loading. These libraries reduce the amount of format-specific infrastructure required by the project; their header-only organisation is a packaging property rather than evidence of improved rendering performance. Dear ImGui [11] supplies the interface for inspecting intermediate outputs and changing rendering parameters.
For further information regarding all third-party libraries and other external resources used in this project, see Appendix A.

4.1.3 Shader Preprocessing

Shader preprocessing permits shared GLSL functions to be maintained in separate files. The implementation uses the `-E` option of shaderc's `glslc` tool [25] to expand `#include` directives and macros before passing shader source to OpenGL. This step produces preprocessed source rather than optimised GPU machine code. Although the build rule below supplies optimisation flags, `-E` selects preprocessing-only operation; those flags do not establish a shader execution-time improvement.
The following xmake rule preprocesses the shader compilation units while leaving shared include files as inputs:

```
rule("preprocess_shaders")
    set_extensions(".vert", ".frag", ".geom", ".glsl")

    on_build_file(function (target, sourcefile, opt) end)

    after_build_file(function (target, sourcefile, opt)
        if path.extension(sourcefile) == ".glsl" then
            print("Ignoring .glsl as a shader library file: " .. sourcefile)
            return
        end

        local shader_root = target:values("shader_root")
        local target_shaders_dir = path.join(target:targetdir(), "shaders")
        local output_path = path.join(target_shaders_dir,
    path.filename(sourcefile))
        os.mkdir(target_shaders_dir)
        os.execv("glslc", {"-I", shader_root, "-E", sourcefile, is_mode("debug")
    and "-O0" or "-Os", "-o", output_path})
        print("Preprocessing shader: " .. sourcefile .. " -> " .. output_path)
    end)
rule_end()
```

It should be noted that this rule ignores shaders with the .glsl suffix, because these files provide shared definitions rather than standalone shader compilation units.

4.1.4 Deferred Rendering of Crytek Sponza

A Crytek Sponza example was used to establish the geometry loading and FrameGraph-based deferred rendering path before LPV integration. Appendix C links to the example. Figure 4.1 visualises the normal component of the G-buffer, providing an intermediate check of geometry, material loading and normal transformations.

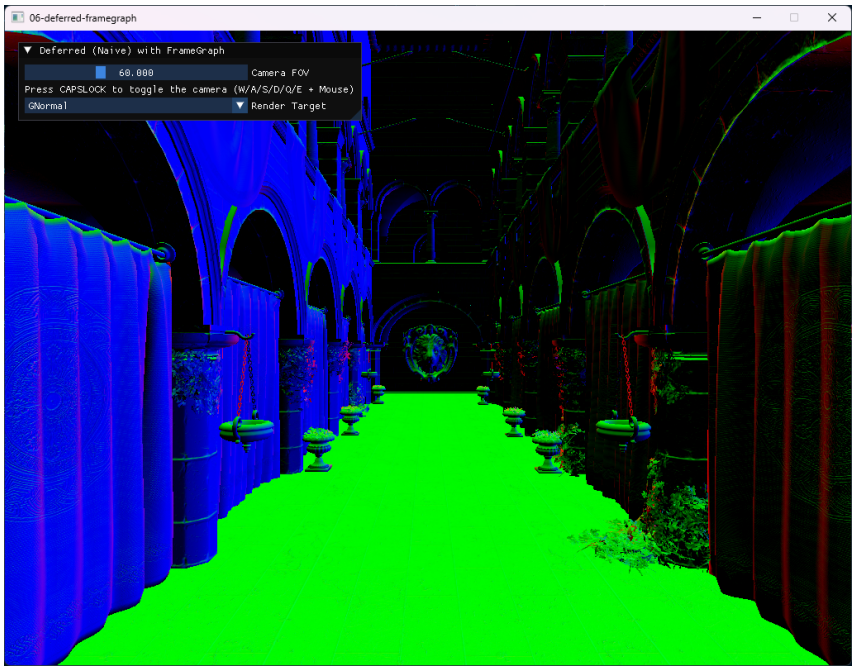


Figure 4.1. Crytek Sponza rendered with VGFW (with FrameGraph). Notice that the render target is GNormal [64].

4.2 Render Passes

The renderer combines multiple passes to produce direct lighting, approximate indirect illumination and post-processing effects. Their integration targets visually plausible images at interactive rates rather than a physically exact solution. For a more detailed workflow, see Figure D.1.

4.2.1 Cascaded Shadow Mapping

The CSM pass starts by setting up resources, including multiple shadow map textures and framebuffers, each linked to a cascade with its depth buffer. This setup ensures effective depth capture at varying distances in the scene.
The process begins by computing cascade splits to cover the entire view frustum. Using a split factor (lambda), the frustum is divided between the near and far clipping planes, ensuring detailed shadows at different depths.
Next, the frustum corners for each cascade are transformed into world space using the inverse of the view-projection matrix, aligning them with each cascade's specific depth range.
For each cascade, view and projection matrices are calculated. The orthographic projection matrix is derived from the frustum's centre and radius, while the view matrix aligns with the light source's direction. The combined view-projection matrix renders the scene from the light's perspective into the shadow maps.

The appropriate cascade index is chosen based on the fragment's depth, and the corresponding view-projection matrix is used to project the scene into the shadow map. To improve shadow quality, Percentage-Closer Filtering (PCF) is applied in the shader, averaging multiple samples around each shadow map sample to reduce aliasing and smooth out shadows.

4.2.2 Reflective Shadow Map (RSM) Generation Pass

The RSM generation pass begins with the initialisation of framebuffers and textures required to capture essential scene information. Specifically, framebuffers are created to store positions, normals, and flux data. These framebuffers are crucial for storing detailed information about the scene from the perspective of the light source, which is necessary for advanced lighting calculations in later stages.
Once the framebuffers and textures are prepared, the framebuffer is bound, and the scene is rendered from the light source's perspective, similar to the shadow mapping process. This rendering populates the textures with the following data for each pixel:

- **Position**: The 3D coordinates of the pixel in world space, representing the spatial location of the scene point.
- **Normal**: The surface normal vector at the pixel, which indicates the orientation of the surface at that point.
- **Flux**: An RGB reflected-light contribution used to seed the LPV. The surface orientation is stored in the normal attachment, not in the RGB flux vector. The expression below is the implementation's simplified source term rather than a complete radiometric integration over a surface patch.

```
RSMPosition = vFragPos;
RSMNormal = normal;
RSMFlux = baseColor * uLight.intensity + emissive;
```

This data is stored in the corresponding textures, which are then used as inputs for subsequent processing stages. The captured position, normal and flux attachments supply the sample locations, lobe orientations and RGB weights for radiance injection.

4.2.3 LPV Radiance Injection Pass

In the LPV radiance injection pass, the process begins with the creation of textures to store the LPV data. This includes reading RSM texture to get the radiance information across the LPV grid, as well as initialising separate RGBA textures for the Spherical Harmonic (SH) coefficients. The four coefficients are the degree-zero coefficient and the three degree-one coefficients. One RGBA16F texture stores this vector for each of the red,

green and blue channels, giving twelve scalar coefficients per cell. Higher SH degrees would require more coefficients and storage.
Once these textures are prepared, each VPL index identifies a sample in the RSM. The sample's stored world position is mapped into the LPV grid using the grid origin and cell size. This separates the two-dimensional indexing of the RSM from the three-dimensional cell coordinates used for radiance storage.
The injector reads the sample's normal and RGB flux from the RSM, constructs an oriented SH lobe, and accumulates the resulting coefficients in the destination cell. The RSM itself stores geometric attributes and flux; it does not contain precomputed SH coefficients.
The cosine-lobe projection represents the directional distribution associated with the sample's surface normal. The `SH_EvaluateCosineLobe` function generates the corresponding SH coefficients. The SH coefficients are computed as follows:

```
#define SH_cosLobe_C0 0.886226925 // sqrt(pi)/2
#define SH_cosLobe_C1 1.02332671  // sqrt(pi/3)

vec4 SH_EvaluateCosineLobe(vec3 direction) {
    direction = normalize(direction);
    return vec4(SH_cosLobe_C0, -SH_cosLobe_C1 * direction.y, SH_cosLobe_C1 *
    direction.z, -SH_cosLobe_C1 * direction.x);
}

const vec4 SH_Coefficients = SH_EvaluateCosineLobe(fs_in.normal) / PI;
```

These computed SH coefficients are then used to scale the radiance flux values. Specifically, the coefficients are applied to the flux components for the red, green, and blue channels as shown:

```
SH_R = SH_Coefficients * fs_in.flux.r;
SH_G = SH_Coefficients * fs_in.flux.g;
SH_B = SH_Coefficients * fs_in.flux.b;
```

Additive blending accumulates these contributions into the corresponding grid cells. The resulting SH textures are the initial field for propagation. The cell size and low-order angular representation limit how much spatial and directional detail the injected field can preserve.

4.2.4 LPV Radiance Propagation Pass

The LPV Radiance Propagation Pass is carried out using a geometry shader and a fragment shader that iteratively propagate radiance across the grid, simulating light diffusion between adjacent cells and enhancing scene realism.

During each iteration, the shader computes radiance contributions from neighbouring cells aligned with the six primary axes. This alignment is managed using orientation matrices that convert primary directions to neighbouring cell indices.
The shader retrieves radiance values for different colour channels from 3D textures for each neighbouring cell. It then computes contributions from the main direction using a cosine lobe approximation, evaluating how light is propagated from this direction.
Additionally, the shader considers contributions from the side faces of each neighbouring cell. It calculates evaluation and reprojection directions for these side faces based on the current orientation and predefined vectors, and scales the contributions according to a solid angle factor.
The total radiance contribution for each cell is obtained by summing contributions from neighbouring cells and their side faces. This accumulated radiance updates the LPV grid for the current cell.
Repeating this process extends the radiance field through the grid. The deferred pass samples the field produced by the selected iteration. The propagation does not perform geometric surface-intersection tests, and this implementation has no separate geometry-occlusion volume. As a result, increasing the iteration count can extend illumination but does not necessarily improve accuracy or prevent transport through thin walls.

4.2.5 G-Buffer Generation Pass

The G-Buffer Generation Pass is a crucial step in deferred rendering, responsible for capturing essential scene attributes. This begins with setting up a framebuffer and multiple textures to store different attributes, such as normals, diffuse colours, and specular information. Positions are often reconstructed from the depth map to save GPU bandwidth.

```
1     const float depth = getDepth(SceneDepth, vTexCoords);
2     if (depth >= 1.0) discard;
3
4     vec3 fragPosViewSpace = viewPositionFromDepth(depth, vTexCoords,
      uCamera.inverseProjection);
5     vec3 fragPos = (uCamera.inverseView * vec4(fragPosViewSpace, 1.0)).xyz;
```

After initialising the framebuffer and textures, the framebuffer is bound, and the scene is rendered. This process captures geometric data into the G-Buffer, with each fragment storing relevant information in the assigned textures. For instance, the normal texture captures surface normals, the depth texture records depth, and other textures store material properties.
The G-Buffer's data enables efficient deferred rendering by separating geometry processing from lighting calculations. This separation allows for flexible lighting

operations in screen space and supports various post-processing effects like ambient occlusion and screen-space reflections.
Once the G-Buffer is filled, the framebuffer is unbound, and the textures are ready for use in subsequent rendering passes to perform lighting computations and visual effects.

4.2.6 Gaussian Blur Pass

The Gaussian blur pass starts with setting up framebuffers and textures for the blurring process, which is executed in two stages: horizontal and vertical.
A 9x9 Gaussian kernel is used to determine the pixel influence based on its neighbours. Linear interpolation optimizes performance by reducing the number of texture samples required while maintaining visual quality [53].
In the horizontal blur stage, the kernel is applied along the x-axis, calculating a weighted average of neighbouring pixels. The result is a horizontally blurred texture. The texture is then passed to the vertical blur stage, where the kernel is applied along the y-axis. This two-pass approach ensures a smooth blur across both dimensions, enhancing the overall image.
The separable filtering structure can be reused by multiple post-processing passes, with separate inputs and filter parameters.

4.2.7 Horizon-Based Ambient Occlusion (HBAO) Pass

The HBAO pass reads G-buffer depth and reconstructs view-space positions. Its surface normals are derived from screen-space position derivatives rather than sampled from the G-buffer normal attachment.
Initially, the depth value is fetched and used to compute the fragment's position in view space. If this depth value is greater than or equal to 1.0, the fragment is discarded to avoid processing invalid or irrelevant data. Subsequently, a noise texture is sampled to introduce randomness into the sampling directions, which helps to reduce visual artefacts. Additionally, screen size parameters are calculated to adjust the sampling process according to the texture resolution.
Following this, the fragment normal is derived by differentiating the fragment position with respect to screen space coordinates. With these details, the shader calculates the step size and angle based on the HBAO parameters. For each directional sample, multiple points around the fragment position are sampled. The ambient occlusion is computed for each of these samples using a falloff function and an attenuation factor that depends on distance.
The computed ambient occlusion values from all sampled points are then accumulated and averaged across different directions and sampling steps. Finally, the accumulated occlusion is scaled by the intensity parameter and subtracted from one. The resulting visibility-like factor is used to modulate diffuse lighting. The result is a local screen-space

occlusion estimate. Its radius and intensity parameters control its appearance, but it does not recover off-screen occluders or replace diffuse light transport.

4.2.8 Deferred Lighting Pass

The Deferred Lighting Pass is a crucial stage in the rendering pipeline that focuses on applying lighting calculations to the scene based on the information stored in the G-Buffer. The process begins by binding the necessary G-Buffer textures, which contain geometric and material data captured from the previous rendering stages. The lighting shader is then set up to perform the required computations.

The shader starts by retrieving depth information from the G-Buffer and discarding fragments with invalid or missing depth values. With valid depth data, the shader reconstructs the fragment's position in view space and extracts essential attributes, including surface normals, base colour, emissive properties, and material characteristics such as metallic and roughness.

If HBAO is enabled, the shader incorporates ambient occlusion data to enhance the shading and depth perception. Following this, the shader calculates shadowing effects by selecting the appropriate shadow cascade based on the fragment's view space position and using shadow maps to determine light visibility.

Direct lighting contributions are then computed using the Cook-Torrance BRDF model [10]. This model includes the Fresnel-Schlick approximation [57] for specular reflection and a Smith-type geometry term for microfacet masking and shadowing. This geometry term is distinct from distance attenuation. The BRDF supplies an approximate surface-reflection model within the renderer's broader lighting approximations. The result combines diffuse and specular lighting components, which are adjusted according to the shadow visibility factor.

For indirect lighting, the shader samples spherical harmonics coefficients from the LPV field produced by the preceding injection and propagation passes. It evaluates the light intensity based on the fragment's normal, blending this indirect lighting with the direct lighting contributions. Ambient occlusion is also applied to the diffuse component to refine the overall lighting effect.

The final output of this pass integrates both direct and indirect lighting contributions, resulting in a comprehensive lighting solution for the scene. If the rendering mode supports high dynamic range (HDR) effects, bright areas are output separately to facilitate HDR processing. The output combines the implemented lighting terms; it should not be interpreted as a complete solution of the rendering equation.

4.2.9 Screen Space Reflections (SSR) Pass

In the Screen Space Reflections (SSR) pass, framebuffers and textures are set up specifically for reflection computations. The pass calculates reflections in screen space using ray-marching techniques, which leverage depth and normal data.
The process begins by sampling depth and normal information from the G-Buffer at the fragment's coordinates. The fragment's view and reflection directions are then computed. Ray marching advances through projected texture-space positions and compares depth against the scene depth buffer. When the intersection test succeeds, the shader samples scene colour at the hit position.
Implementation choices include varying the ray step size based on surface roughness and applying jitter to reduce artefacts. The Fresnel-Schlick approximation [57] is used to calculate reflectance, while the visibility factor and screen edge factor are computed to refine the final reflection colour. These heuristics control the appearance of the reflection, but remain subject to missing screen-space geometry and depth-test artefacts. Their individual performance benefits are not isolated by the evaluation.

```
// Simple hash function for jittering
vec3 hash(vec3 a) {
    const vec3 scale = vec3(0.8);
    const float K = 19.19;

    a = fract(a * scale);
    a += dot(a, a.yxz + K);
    return fract((a.xxy + a.yxx) * a.zyx);
}
```

4.2.10 Blit Pass

The blit pass composites a source texture into a selected destination render target. In the SSR path it adds the reflection texture to the scene-colour attachment. This is an off-screen composition operation; presentation to the default framebuffer is handled by the final composition pass.
The pipeline is configured for additive blending to get the correct result. This involves setting the blend state to enable blending, with source and destination factors both set to One, allowing for additive colour blending. Additionally, the depth and stencil testing are disabled, while rasterization and scissor tests are configured for basic rendering.
A full-screen draw samples the source texture and adds it to the destination attachment. Source and destination resources are declared in FrameGraph so that the operation follows the passes producing their contents.

4.2.11 Bloom Pass

The bloom pass combines the blurred bright image and the original scene colour image. Bright areas of the image are extracted into a separate texture. A Gaussian Blur effect is then applied to these bright areas, which are subsequently blended back into the original image. This process enhances the appearance of bright regions and contributes to a visually appealing bloom effect.

```
#version 460 core

layout(location = 0) in vec2 v_TexCoords;

layout(location = 0) out vec4 FragColor;

layout(binding = 0) uniform sampler2D sceneColor;
layout(binding = 1) uniform sampler2D bloomBlur;

uniform float bloomFactor;

void main() {
    vec3 hdrColor = texture(sceneColor, v_TexCoords).rgb;
    vec3 bloomColor = texture(bloomBlur, v_TexCoords).rgb;
    hdrColor += bloomColor * bloomFactor;

    FragColor = vec4(hdrColor, 1);
}
```

4.2.12 ACES Tone-Mapping Pass

The tone-mapping pass applies Narkowicz's ACES-inspired filmic fit [46]. This conversion involves mapping HDR values into the LDR range through a specific tone-mapping function as mentioned in equation 2.13.

```
vec3 toneMapACES(vec3 color) {
    const float a = 2.51;
    const float b = 0.03;
    const float c = 2.43;
    const float d = 0.59;
    const float e = 0.14;
    return clamp((color * (a * color + b)) / (color * (c * color + d) + e), 0.0,
    1.0);
}
```

The shader clamps the fitted result to the display range. The function does not implement the complete ACES colour-space and display-transform pipeline, so its use alone does not establish conformance to ACES.

4.2.13 Fast Approximate Anti-Aliasing (FXAA) Pass

In the FXAA pass, the FXAA shader works by sampling the colours of the current pixel and its surrounding pixels to assess the amount of aliasing. It computes the luminance for each sampled colour to identify areas where aliasing is likely to occur. By comparing the luminance values of the current pixel and its neighbours, the shader determines the direction in which to blur.
The shader calculates a direction vector based on the differences in luminance values around the pixel, which helps in estimating where the jagged edges are most prominent. This vector is then normalised and clamped to ensure it remains effective for smoothing out edges.
Using this direction vector, the shader samples colours along this direction and blends them to reduce the appearance of jagged edges. The final output colour is determined by comparing the luminance of the blended colour with the minimum and maximum luminance values of the surrounding pixels. If the broader filtered estimate falls outside the neighbourhood luminance range, the shader selects a narrower two-sample estimate; otherwise it selects the broader estimate. This limits the spread of the filtered result across high-contrast boundaries.
This pass reduces visible spatial aliasing using final-image samples. It can soften fine detail and does not provide a measurement or guarantee of temporal stability.

4.2.14 Final Composition Pass

The final composition pass displays the render target selected by the user. The `compose` method selects the appropriate texture based on user settings and the chosen render target. It handles various targets such as scene colour textures (HDR and LDR), reflective shadow map data, G-buffer textures, HBAO results, and SSR results. The method checks whether post-processing effects like FXAA and HBAO are enabled and selects the corresponding texture accordingly.

```
1     switch (settings.renderTarget)
2     {
3         case RenderTarget::eFinal:
4             if (settings.enableFXAA)
5             {
6                 output = blackboard.get<SceneColorData>().aa;
7             }
8             else
9             {
10                output = blackboard.get<SceneColorData>().ldr;
11            }
12            break;
13
```

```
14          case RenderTarget::eRSMPosition:
15              output = blackboard.get<ReflectiveShadowMapData>().position;
16              break;
17
18          case RenderTarget::eRSMNormal:
19              output = blackboard.get<ReflectiveShadowMapData>().normal;
20              break;
21
22          case RenderTarget::eRSMFlux:
23              output = blackboard.get<ReflectiveShadowMapData>().flux;
24              break;
25
26          case RenderTarget::eGNormal:
27              output = blackboard.get<GBufferData>().normal;
28              break;
29
30          case RenderTarget::eGAlbedo:
31              output = blackboard.get<GBufferData>().albedo;
32              break;
33
34          case RenderTarget::eGEmissive:
35              output = blackboard.get<GBufferData>().emissive;
36              break;
37
38          case RenderTarget::eGMetallicRoughnessAO:
39              output = blackboard.get<GBufferData>().metallicRoughnessAO;
40              break;
41
42          case RenderTarget::eHBAO:
43              if (settings.enableHBAO)
44              {
45                  output = blackboard.get<HBAOData>().hbao;
46              }
47              break;
48
49          case RenderTarget::eSceneColorHDR:
50              output = blackboard.get<SceneColorData>().hdr;
51              break;
52
53          case RenderTarget::eSceneColorBright:
54              output = blackboard.get<SceneColorData>().bright;
55              break;
56
57          case RenderTarget::eSSR:
58              if (settings.enableSSR)
59              {
60                  output = blackboard.get<SSRData>().ssr;
61              }
```

```
            break;

        case RenderTarget::eSceneColorLDR:
            output = blackboard.get<SceneColorData>().ldr;
            break;
    }
```

Within the frame graph, the method sets up a callback pass to handle resource reading and side effects, binding the selected texture and graphics pipeline. It then renders a full-screen triangle to display the final image, integrating all visual effects and enhancements applied throughout the rendering pipeline.
This pass ensures that the final image is based on user settings and selected render targets.

4.3 User Interface

The user interface is essential to enable users to switch rendering features on and off to set rendering parameters, as well as to facilitate evaluation and comparison of results. This project uses `ImGui` to implement the user interface. For some configurable parameters, a `RenderSettings` data structure is defined as follows:

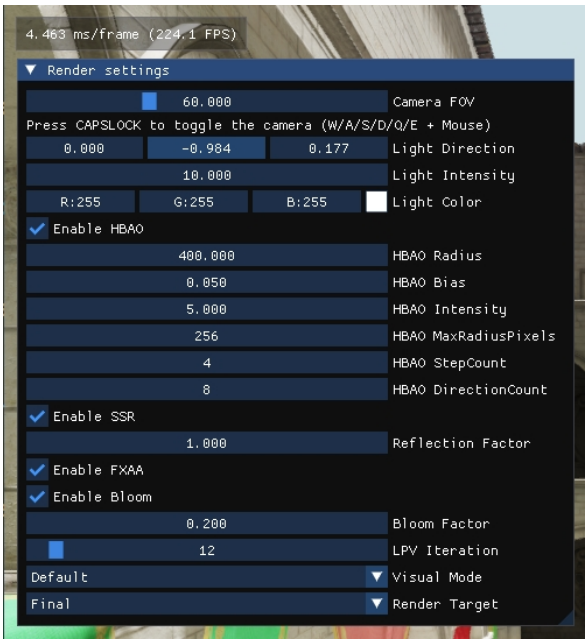


Figure 4.2. ImGui windows.

```
enum class RenderTarget
{
    eFinal = 0,
    eRSMPosition,
    eRSMNormal,
    eRSMFlux,
    eGNormal,
```

```
8       eGAlbedo,
9       eGEmissive,
10      eGMetallicRoughnessAO,
11      eHBAO,
12      eSceneColorHDR,
13      eSceneColorBright,
14      eSSR,
15      eSceneColorLDR,
16  };
17
18  enum class VisualMode
19  {
20      eDefault = 0, // Direct + LPV
21      eOnlyDirect,
22      eOnlyLPV
23  };
24
25  struct HBAOProperties
26  {
27      float radius {400.0f};
28      float bias {0.05f};
29      float intensity {5.0f};
30      int   maxRadiusPixels {256};
31      int   stepCount {4};
32      int   directionCount {8};
33  };
34
35  struct RenderSettings
36  {
37      // Render target
38      RenderTarget renderTarget = RenderTarget::eFinal;
39
40      // Visual mode
41      VisualMode visualMode = VisualMode::eDefault;
42
43      // Render features
44      bool enableHBAO  = true;
45      bool enableSSR   = true;
46      bool enableFXAA  = true;
47      bool enableBloom = true;
48
49      // LPV settings
50      int lpvIteration = 12;
51
52      // HBAO properties
53      HBAOProperties hbaoProperties {};
54
55      // SSR settings
```

```
56      float reflectionFactor = 1.0f;
57
58      // Bloom settings
59      float bloomFactor = 0.2f;
60  };
```

The final implementation is shown in Figure 4.2, with a floating semi-transparent window showing the elapsed time and FPS per frame, and a collapsible window plotting all the configurable rendering parameters.

4.4 Documentation

The `README.md` document, located within the repository, provides detailed instructions on how to prepare the development environment and how to compile and run this project.

Chapter 5

Software Testing and Evaluation

5.1 Debug and Profiling Tools

5.1.1 RenderDoc

RenderDoc [61] captures graphics commands and resources for frame-level inspection. Its pipeline views permit vertex data, shader inputs, uniform-buffer layouts and render-target contents to be checked against the intended implementation. This is useful when an incorrect image may result from resource binding, command order, data layout or shader logic. Figure 5.1 shows inspection of the HBAO pass. Such inspection supports debugging of individual frames; it does not by itself establish numerical accuracy or correctness across all scenes.

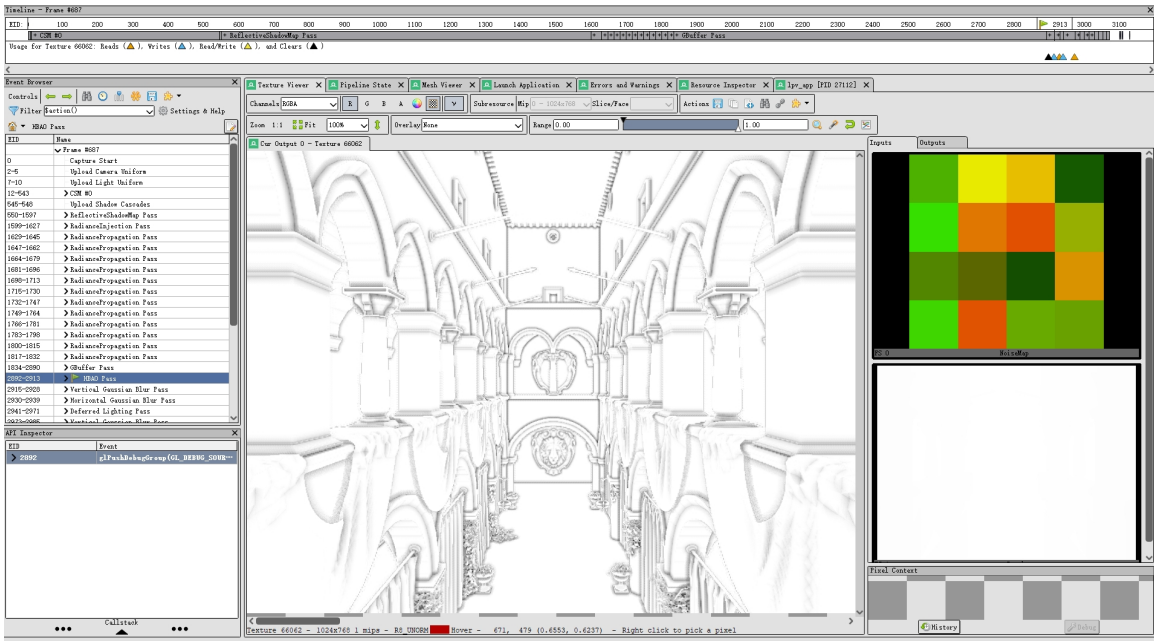


Figure 5.1. An example of using RenderDoc to view the result of the HBAO pass. The Event Browser, located in the top left corner, enables the user to select the events to be analysed. On the right, a variety of additional options are available, including the viewing of textures, geometric data, pipeline status, and more.

5.1.2 Tracy Profiler

Tracy Profiler [60] records instrumented CPU scopes, GPU timing zones and application statistics. The recorded zones allow expensive passes to be identified and their reported

mean times to be compared within a capture. Memory statistics depend on the allocation tracking provided by the application and should be interpreted according to that scope. Figure 5.2 shows an example capture. The table below reports the renderer's performance measurements; the screenshot should be treated as an illustration of the profiling interface rather than proof that every displayed statistic belongs to the same measurement interval.

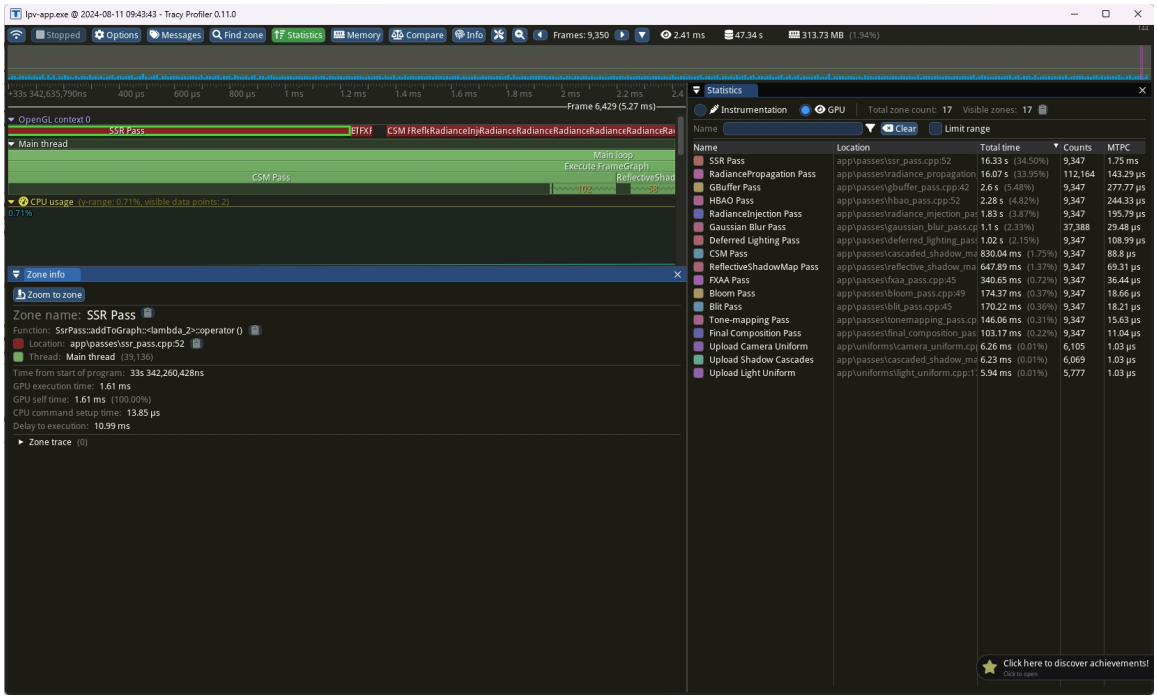


Figure 5.2. An example of using Tracy Profiler to evaluate the performance. As a result, the SSR pass is the most time-consuming.

5.2 Performance Testing

The evaluation examines the integrated renderer rather than proposing an optimisation to LPV or benchmarking it against competing global illumination algorithms. Two kinds of evidence are presented: pass-level performance measurements for a specified Sponza configuration, and image comparisons illustrating the effects of selected rendering features. The performance measurements address the cost of the complete configuration, while the images support qualitative observations about appearance. Neither alone establishes an improvement in physical accuracy or perceptual quality.

5.2.1 Test Machine Environment

The performance evaluation uses the hardware configuration in Table 5.1 and the rendering parameters in Table 5.2.

Table 5.1. Test system and output resolution.

Component	Configuration
Processor	Intel Core i9-13900H
Graphics processor	NVIDIA GeForce RTX 4060 Laptop GPU
Operating system	Windows 11 Home, 64-bit
Output resolution	1920×1080 pixels

Table 5.2. Rendering parameters used for the performance measurements.

Parameter	Value
CSM cascade count	1
RSM resolution	512×512 texels
LPV grid dimensions	$32 \times 20 \times 11$ cells
LPV propagation iterations	12
HBAO radius	400
HBAO bias	0.05
HBAO intensity	5
HBAO steps per direction	4
HBAO direction count	8
SSR reflection factor	1
Bloom factor	0.2

5.2.2 Performance

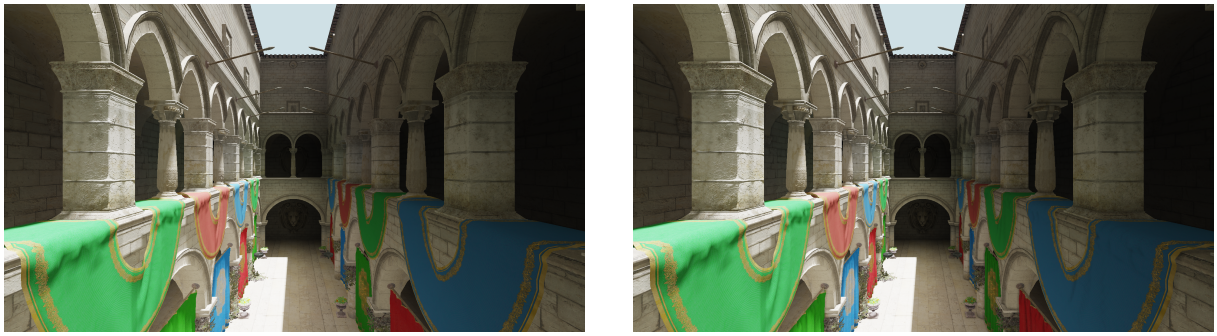
Table 5.3 reports the frame rate, frame time, memory statistic and mean pass times obtained using Tracy Profiler. The reciprocal of the reported 5.701254 ms frame time is approximately 175.4 frames per second, consistent with the listed frame rate. The table retains a single-iteration timing for LPV propagation; the tested configuration executes twelve iterations.

Table 5.3. Performance of the Sponza rendering configuration. Propagation is timed per iteration; the configuration executes twelve iterations.

Metric	Value	Unit
Frame statistics		
Frame rate	175.4	frames/s
Reported memory usage	100.08	MB
Frame time	5.701254	ms
Mean pass timings		
CSM	0.093310	ms
RSM	0.072040	ms
LPV radiance injection	0.187310	ms
LPV propagation (one iteration)	0.144450	ms
G-buffer generation	0.566730	ms
HBAO	0.635420	ms
Gaussian blur (horizontal + vertical)	0.072390	ms
Deferred lighting	0.324660	ms
Bloom	0.049570	ms
SSR	1.770320	ms
Blit	0.040620	ms
Tone mapping	0.036830	ms
FXAA	0.089040	ms
Final composition	0.024260	ms

5.3 Results

The following paired images illustrate the effects of selected rendering features at the shown camera views. They provide qualitative comparisons rather than quantitative error measurements against a reference solution. Full-frame comparisons are useful for overall appearance, although small differences, particularly from FXAA, require closer inspection.



(a) HBAO disabled (b) HBAO enabled

Figure 5.3. Sponza with HBAO disabled (left) and enabled (right).



(a) SSR disabled (b) SSR enabled

Figure 5.4. Sponza with SSR disabled (left) and enabled (right), using a reflection factor of 1.0.



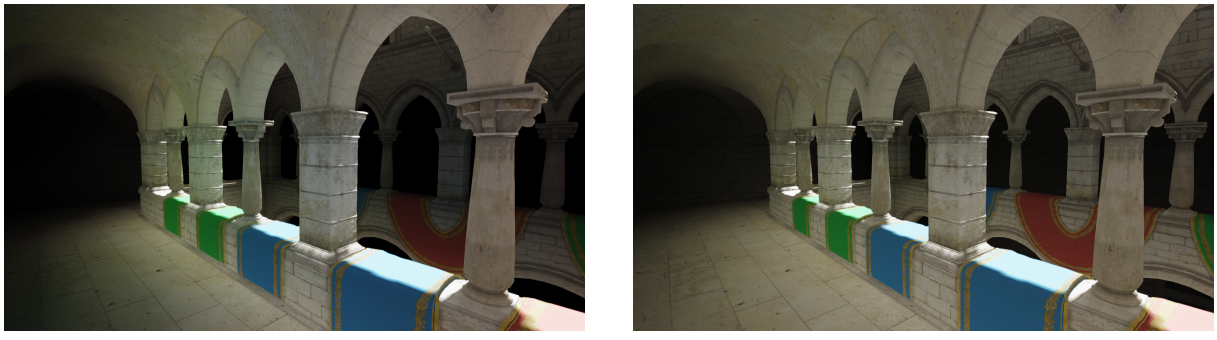
(a) FXAA disabled (b) FXAA enabled

Figure 5.5. Final image with FXAA disabled (left) and enabled (right).



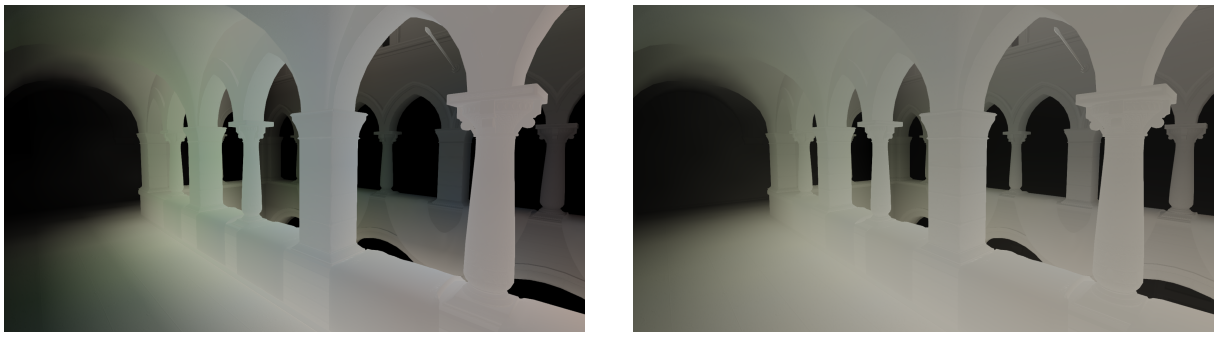
(a) Bloom disabled (b) Bloom enabled

Figure 5.6. Sponza with bloom disabled (left) and enabled (right), using a bloom factor of 1.0 for this comparison.



(a) $k = 6$ (b) $k = 12$

Figure 5.7. Combined lighting after $k = 6$ and $k = 12$ LPV propagation iterations.



(a) $k = 6$ (b) $k = 12$

Figure 5.8. Indirect lighting alone after $k = 6$ and $k = 12$ LPV propagation iterations.

5.3.1 Visual Observations

Figures 5.3 and 5.4 illustrate changes in local darkening and reflected scene colour. HBAO affects the visibility of contact regions and recesses through a screen-space approximation. SSR changes the appearance of reflective surfaces using the currently visible scene. The images demonstrate the contribution of these passes to the displayed result, but do not establish that either effect recovers the correct indirect illumination. Assessing transport accuracy would require a reference solution in addition to these appearance comparisons. Figure 5.5 compares the anti-aliasing output at full-frame scale. The relevant differences concern high-contrast edges and fine detail, which are less apparent at this scale. A stronger assessment would use matched enlarged crops and a spatial or temporal reference. Figure 5.6 shows the appearance of the glow effect. Its factor of 1.0 differs from the 0.2 value used in the reported performance configuration, so this image should not be treated as an exact visual record of that benchmark.
Figures 5.7 and 5.8 compare six and twelve propagation iterations. The indirect-only views help distinguish changes in the LPV field from the direct-lighting term. More iterations alter the spatial reach and distribution of the field, but do not necessarily reduce error. In a low-resolution grid without a geometry-occlusion volume, transport can also extend through surfaces that should block it. The iteration count controls a numerical propagation process rather than a count of physical surface bounces.

5.3.2 Performance Interpretation

The SSR pass has the largest reported time among the listed individual pass entries, at 1.770320 ms, approximately 31% of the reported frame time. LPV propagation must also be considered across all twelve iterations: multiplying the listed 0.144450 ms single-iteration value by twelve gives approximately 1.7334 ms. This arithmetic identifies propagation as a substantial component of the configuration, even though one iteration is relatively inexpensive. It is not a separate timing experiment or a measurement of how propagation scales with iteration count.

The table reports 100.08 MB of memory usage. The allocation domain is not specified, so it should not be interpreted as total GPU memory usage or total process memory. The profiler screenshot contains different displayed values and is not sufficient to reconcile the two measurements. Likewise, adding mean pass times does not establish the frame's critical path or account for every CPU, driver and presentation cost.

5.3.3 Evaluation Limitations

The measurements cover one scene, one resolution and one laptop GPU configuration. The sampling interval, number of measured frames, warm-up period, driver version, power settings and presentation constraints are not specified. The reported means therefore describe the available measurement rather than a statistically characterised benchmark. No confidence intervals, frame-time distribution or controlled cross-device comparison are available.

The image comparisons also have a limited scope. There is no path-traced reference for this implementation, numerical image-error analysis, temporal sequence evaluation or user study. The performance table does not isolate the benefit of depth-based position reconstruction, shader preprocessing or early-exit heuristics. A more complete study would hold the camera and settings fixed, repeat the measurements, disable one feature at a time and report both visual differences and changes in frame time. These limitations do not invalidate the implementation demonstration, but restrict the conclusions to the tested configuration and the illustrated behaviours.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This project implemented LPV in an OpenGL deferred renderer and integrated HBAO, SSR, FXAA, filmic tone mapping and bloom through a FrameGraph-based pipeline. The Sponza comparisons illustrate the contribution of individual effects and the influence of the propagation iteration count. The reported configuration achieves 175.4 frames per second at 1920×1080 on an NVIDIA GeForce RTX 4060 Laptop GPU, demonstrating interactive operation for the tested scene and settings.

The main contribution is a working integration of established methods, with an account of resource organisation, implementation choices and pass costs. LPV's spatial and angular resolution and the screen-space visibility available to SSR remain important limitations. The evaluation supports conclusions about this configuration; temporal stability, perceptual quality and comparative transport accuracy require further experiments.

6.2 Future Work

A Vulkan implementation [39] would permit investigation of explicit resource management, synchronisation and command submission. A controlled comparison should hold the scene, shaders, resolution and effect settings constant, and measure CPU and GPU costs separately. Repeated measurements with a defined warm-up period and frame count would help distinguish improvements from measurement variation.

Light leakage, limited occlusion and coarse illumination should be examined in controlled scenes containing thin walls and narrow openings. Cascaded LPV [36] and related extensions [9, 49, 40, 37] provide possible directions. Fixed-camera comparisons against reference renders would help assess whether these changes improve accuracy at an acceptable cost.

Participating media are another direction for extending the renderer. Propagating light through fog or smoke [8] would require a volumetric representation and a scattering model alongside the surface-lighting pipeline.

Further evaluation should include multiple scenes, temporal sequences, controlled feature ablations and clearly scoped memory measurements. Reporting frame-time distributions and image differences would strengthen the evidence for future changes while retaining the current renderer as a documented implementation example.

References

[1] N. Alarcon. Integrating dynamic diffuse global illumination (DDGI), 2019. Accessed: 2026-10-17. URL: https://developer.nvidia.com/blog/an-engineers-guide-to-integrating-ddgi/.

[2] W. Arrighetti. The academy color encoding system (ACES): A professional color-management framework for production, post-production and archival of still and motion pictures. *Journal of Imaging*, 3(4):40, 2017. doi:10.3390/jimaging3040040.

[3] S. Barrett. stb_image, 2024. URL: https://github.com/nothings/stb.

[4] L. Bavoil, M. Sainz, and R. Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks*, pages 1–1. ACM, 2008. doi:10.1145/1401032.1401061.

[5] C. Benthin and I. Wald. Efficient ray traced soft shadows using multi-frusta tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 135–144, 2009. doi:10.1145/1572769.1572791.

[6] A. P. Beug. Screen space reflection techniques. Master's thesis, Faculty of Graduate Studies and Research, University of Regina, 2020.

[7] M. Billeter, O. Olsson, and U. Assarsson. Tiled forward shading. In *GPU Pro 360 Guide to Rendering*, pages 261–276. AK Peters/CRC Press, 2018. doi:10.1201/9781351261524-17.

[8] M. Billeter, E. Sintorn, and U. Assarsson. Real-time multiple scattering using light propagation volumes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 119–126. ACM, 2012. doi:10.1145/2159616.2159636.

[9] J. Børlum, B. B. Christensen, T. K. Kjeldsen, P. T. Mikkelsen, K. Ø. Noe, J. Rimestad, and J. Mosegaard. SSLPV: subsurface light propagation volumes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 7–14, 2011. doi:10.1145/2018323.2018325.

[10] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (ToG)*, 1(1):7–24, 1982. doi:10.1145/357290.357293.

[11] O. Cornut. Dear ImGui, 2024. URL: https://github.com/ocornut/imgui.

[12] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, 2011. doi:10.1111/j.1467-8659.2011.02063.x.

[13] F. C. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):242–248, 1977. doi:10.1145/965141.563901.

[14] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák. Scalable realistic rendering with many-light methods. *Computer Graphics Forum*, 33(1):88–104, 2014. doi:10.1111/cgf.12256.

[15] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 203–231, 2005. doi:10.1145/1053427.1053460.

[16] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *ACM SIGGRAPH Computer Graphics*, 22(4):21–30, 1988.

[17] R. Dimitrov. Cascaded shadow maps. Technical report, NVIDIA Corporation, 2007.

[18] Envato Tuts+. Forward rendering vs. deferred rendering, 2024. Accessed: 2024-08-12. URL: `https://gamedevelopment.tutsplus.com/forward-rendering-vs-deferred-rendering--gamedev-12342a`.

[19] Epic Games. Lumen in unreal engine 5, 2021. Accessed: 2024-08-12. URL: `https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine`.

[20] Epic Games. Bloom, n.d. Accessed: 2024-08-12. URL: `https://udn.epicgames.com/Three/Bloom.html`.

[21] S. Fujita. TinyGLTF, 2024. URL: `https://github.com/syoyo/tinygltf`.

[22] S. Fujita. tinyobjloader, 2024. URL: `https://github.com/tinyobjloader/tinyobjloader`.

[23] Gabime. spdlog, 2024. URL: `https://github.com/gabime/spdlog`.

[24] M. Geelnard and C. Berglund. GLFW, 2024. URL: `https://www.glfw.org/`.

[25] Google. shaderc, 2024. URL: `https://github.com/google/shaderc`.

[26] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH computer graphics*, 18(3):213–222, 1984. `doi:10.1145/964965.808601`.

[27] R. Green. Spherical harmonic lighting: The gritty details. Game Developers Conference, 2003.

[28] T. Harada, J. McKee, and J. C. Yang. Forward+: Bringing deferred lighting to the next level. In *Eurographics (Short Papers)*, pages 5–8. The Eurographics Association, 2012. `doi:10.2312/conf/EG2012/short/005-008`.

[29] S. Hargreaves and M. Harris. Deferred shading. Game Developers Conference, 2004.

[30] D. Herberth. GLAD, 2024. URL: `https://github.com/Dav1dde/glad`.

[31] G. James and J. O'Rorke. Real-time glow. In *GPU Gems*, pages 343–362. Pearson Education, 2004. URL: `https://developer.nvidia.com/gpugems/gpugems/part-iv-image-processing/chapter-21-real-time-glow`.

[32] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96: Proceedings of the Eurographics Workshop on Rendering*, pages 21–30. Springer, 1996. `doi:10.1007/978-3-7091-7484-5_3`.

[33] J. T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986. `doi:10.1145/15922.15902`.

[34] G. Kalogirou. How to do good bloom for hdr rendering, 2006. Accessed: 2024-08-12. URL: `http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/`.

[35] A. Kaplanyan. Light propagation volumes in CryEngine 3. In *Advances in Real-Time Rendering in 3D Graphics and Games, ACM SIGGRAPH Courses*. ACM, 2009. URL: https://advances.realtimerendering.com/s2009/Light_Propagation_Volumes.pdf.

[36] A. Kaplanyan and C. Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107, 2010. doi:10.1145/1730804.1730821.

[37] A. Kaplanyan, W. Engel, and C. Dachsbacher. Diffuse global illumination with temporally coherent light propagation volumes. In W. Engel, editor, *GPU Pro 360 Guide to Lighting*, chapter 8. AK Peters/CRC Press, 2019. URL: https://www.routledge.com/GPU-Pro-360-Guide-to-Lighting/Engel/p/book/9781351201230.

[38] A. Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, 1997. doi:10.1145/258734.258769.

[39] Khronos Group. *Vulkan API Specification*, 2016. URL: https://www.khronos.org/vulkan/.

[40] M. D. Koa and H. Johan. ESLPV: enhanced subsurface light propagation volumes. *The Visual Computer*, 30:821–831, 2014. doi:10.1007/s00371-014-0952-3.

[41] LearnOpenGL. Deferred shading, 2019. Accessed: 2024-08-12. URL: https://learnopengl.com/Advanced-Lighting/Deferred-Shading.

[42] LearnOpenGL. Bloom, 2024. Accessed: 2024-08-12. URL: https://learnopengl.com/Advanced-Lighting/Bloom.

[43] T. Lottes. FXAA: Fast approximate anti-aliasing. Technical report, NVIDIA Corporation, 2011. URL: https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

[44] Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai, and M. McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques*, 8(2):1–30, 2019. URL: https://jcgt.org/published/0008/02/01/.

[45] M. McGuire and M. Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, 2014. URL: https://jcgt.org/published/0003/04/04/.

[46] K. Narkowicz. ACES Filmic Tone Mapping Curve. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/, 2016.

[47] H. Nguyen, editor. *GPU Gems 3*. Addison-Wesley Professional, 2007. URL: https://developer.nvidia.com/gpugems/gpugems3.

[48] NVIDIA. Nvidia rtx technology, 2018. Accessed: 2024-08-12. URL: https://www.nvidia.com/en-us/geforce/rtx/?nvid=nv-int-solr-404279.

[49] J. D. Olovsson and M. Doggett. Octree light propagation volumes. In *SIGRAD, 2013*. Eurographics-European Association for Computer Graphics, 2013.

[50] O. Olsson and U. Assarsson. Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, 2011. doi:10.1080/2151237x.2011.621761.

[51] O. Olsson, M. Billeter, and U. Assarsson. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 87–96. The Eurographics Association, 2012. doi:10.2312/EGGH/HPG12/087-096.

[52] Y. O'Donnell. FrameGraph: Extensible rendering architecture in frostbite. In *Game Developers Conference*, 2017. URL: https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in.

[53] D. Rákos. Efficient gaussian blur with linear sampling, 2010. Accessed: 2026-10-17. URL: https://www.rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/.

[54] R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 497–500, 2001. doi:10.1145/383259.383317.

[55] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3):267–276, 2002. doi:10.1145/566654.566575.

[56] C. Riccio. OpenGL Mathematics (GLM), 2024. URL: https://github.com/g-truc/glm.

[57] C. Schlick. An inexpensive BRDF model for physically-based rendering. *Computer graphics forum*, 13(3):233–246, 1994. doi:10.1111/1467-8659.1330233.

[58] skaarj1989. FrameGraph, 2024. URL: https://github.com/skaarj1989/FrameGraph.

[59] M. Stamminger and G. Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, 2002.

[60] B. Taudul. Tracy Profiler, 2024. Accessed: 2024-08-12. URL: https://github.com/wolfpld/tracy.

[61] The RenderDoc Team. RenderDoc, 2024. URL: https://renderdoc.org/.

[62] L. Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, 1978. doi:10.1145/800248.807402.

[63] xmake-io. xmake, 2024. Accessed: 2024-08-12. URL: https://github.com/xmake-io/xmake.

[64] K. Zhang. VGFW (V Graphics Framework), 2024. URL: https://github.com/zzxzzk115/vgfw.

Appendices

Appendix A

External Material

In this project, the following external materials were used:

- **glm**: `https://github.com/g-truc/glm`
- **glfw**: `https://github.com/glfw/glfw`
- **glad**: `https://github.com/Dav1dde/glad`
- **shaderc**: `https://github.com/google/shaderc`
- **FrameGraph**: `https://github.com/skaarj1989/FrameGraph`
- **FrameGraph-Example**: `https://github.com/skaarj1989/FrameGraph-Example`
- **imgui**: `https://github.com/ocornut/imgui`
- **stb_image**: `https://github.com/nothings/stb`
- **tinyobjloader**: `https://github.com/syoyo/tinyobjloader`
- **tinygltf**: `https://github.com/syoyo/tinygltf`
- **Crytek Sponza (glTF 2.0)**: `https://github.com/KhronosGroup/glTF-Sample-Models/tree/main/2.0/Sponza`

The Sponza asset's accompanying credits identify Frank Meinl's Crytek model, Marko Dabrovic's original scene, Morgan McGuire's model corrections and Alexandre Pestana's PBR textures. These scene assets are credited separately from the rendering software. Practical implementation references include FrameGraph-Example (listed above) and Eric Polman's LPV implementation notes: `https://ericpolman.com/2016/06/28/light-propagation-volumes/`. The external libraries, reference implementations and scene assets are distinct from the integration and analysis presented in this dissertation.

Appendix B

Project Repository

To view the source code, see
`https://github.com/zzxzzk115/Light-Propagation-Volumes`.

Appendix C

Additional Code

For a more detailed example of the Crytek Sponza rendered with VGFW (with FrameGraph), see `https://github.com/zzxzzk115/vgfw/tree/master/examples/06-deferred-framegraph`

Appendix D

Additional Figures

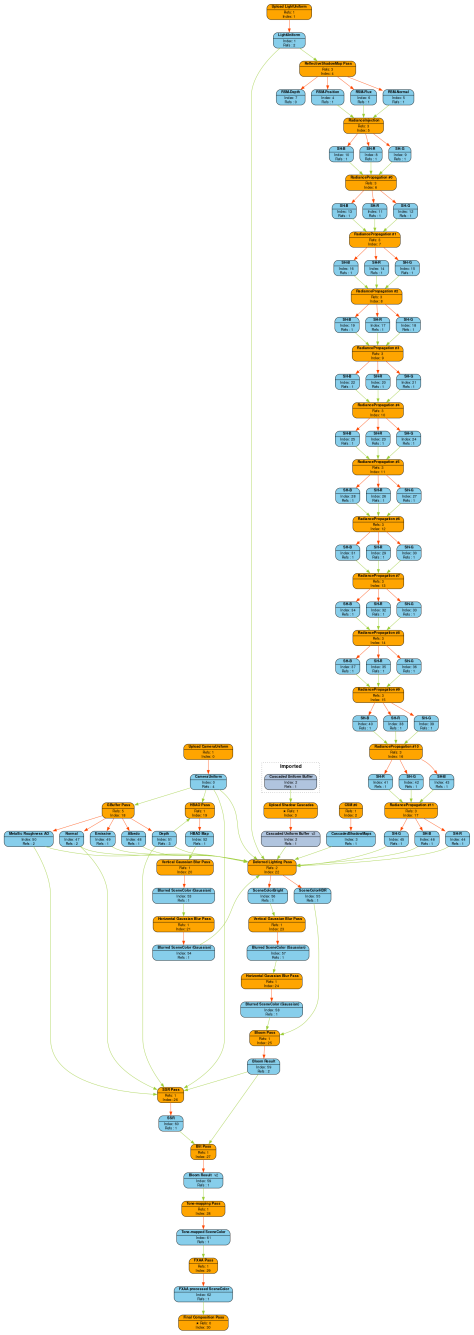


Figure D.1. Exported FrameGraph of the project.