Exploration and Optimization of Radiance Cascades for Real-Time Applications

Master's thesis in High Performance Computer Systems

KYRIAKOS GAVRAS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden 2025

© KYRIAKOS GAVRAS, 2025.
Supervisor: Ulf Assarsson, Department of Computer Science and Engineering
Examiner: Erik Sintorn, Department of Computer Science and Engineering
Master's Thesis 2025
Computer Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The Sponza model by Dabrovic [8] illuminated by the sky and two emissive texts.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

KYRIAKOS GAVRAS
Computer Engineering
Chalmers University of Technology

Abstract

This thesis introduces the fundamentals of Radiance Cascades in screen space for 2D, with more focus on the implementation of a less explored approach called Screen-Space Probes with World-Space Intervals (SPWI) to solve single shot diffuse global illumination. A conventional 3D probe grid faces significant memory constraints; however, SPWI gives a solution to capture off-screen indirect light more memory efficiently. The implementation diverges from Alexander Sannikov's original approach in a few aspects. It employs ray-tracing instead of ray marching, which is just personal preference and it also incorporates Sannikov's post-publication novel depth aware up-scaling method "Bilinear 3D", which he released after the release of the original Radiance Cascades paper. This work presents comparative analysis between different upscaling techniques and documents experimental approaches that, while ultimately not incorporated in the final implementation, offer valuable insights for future research. The findings contribute to the foundational understanding of this approach and establish a framework for further improvements in Radiance Cascade with SPWI for real-time rendering.

Keywords: probes, SPWI, ray-tracing, 2D, 3D, global illumination, intervals

Acknowledgements

I want to express my gratitude to the Radiance Cascades community on the discord server for their invaluable resources of this novel global illumination technique, as well as Alexander Sannikov for providing feedback on my rendered results regarding their correctness as well as insights to further extend this technique.

Kyriakos Gavras, Gothenburg, April 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

JFA	Jump Flood Algorithm
SPP	Samples Per Pixel
SDF	Signed Distance Field
SPWI	Screen-space Probes World Intervals

Contents

List of Acronyms ix

List of Figures xiii

List of Tables xv

1 Introduction 1

2 Previous Work 3

3 Observations in 2D 5
3.1 Angular Observation . . . 5
3.2 Spatial Observation . . . 6
3.3 Penumbra Condition . . . 6
3.4 Radiance Cascade in 2D . . . 6
3.4.1 Efficient Ray Marching . . . 7
3.5 Cascades Creation Process . . . 9

4 Implementation in a Higher Dimension 13
4.1 Comparison of a few Approaches . . . 13
4.2 Placing the Probes in Depth . . . 14
4.3 Encoding Intervals using Octahedral Maps . . . 16
4.4 Cascade Hierarchy in SPWI . . . 17
4.5 Merging with the Upper Cascade . . . 18
4.5.1 Bilateral Filtering . . . 19
4.5.2 Bilinear 3D . . . 21
4.5.3 Cascade Dispatching Strategy . . . 22
4.5.4 Ray Tracing Kernel . . . 23
4.5.5 Tracing and Merging Results . . . 25
4.6 Final Composition . . . 26
4.6.1 Direction First Layout . . . 29

5 Experiments 31
5.1 Pre Averaging . . . 31
5.1.1 Thread Organization for Pre-Averaging . . . 32
5.1.2 SIMD Shuffle Operations (Metal Shaders) . . . 32
5.1.3 Pre-Averaging Algorithm . . . 33

5.1.4 Pre Averaging Result 34

6 Results 35

7 Discussion and Future Work 41

7.1 Min Max Depth Buffer 41

8 Conclusion 45

9 Ethical Considerations 47

Bibliography 49

List of Figures

3.1 A probe and two light source, one of which it couldn't detect due to the low angular resolution. 5
3.2 Interpolation of high spatial resolution probe grid. Image from Sannikov [1]. 6
3.3 Interpolation of low spatial resolution probe grid. Image from Sannikov [1]. 6
3.4 Drawn texture by the user that will be used as the geometry of the scene. 7
3.5 Jump Flood Algorithm end result. 8
3.6 Distance texture that is used to step through in ray marching. 9
3.7 Radiance Cascade layers from cascade level 5 (top-left) to level 0 (bottom-right). 12

4.1 Screen space probes of cascade 0. 15
4.2 Screen space probes of cascade 0 from a different angle. 15
4.3 Converting a sphere into an octahedral map. Image from Cigolle et al. [4]. 16
4.4 Ray distribution patterns encoded using octahedral maps for the first three cascade levels. The images show how angular resolution increases with each cascade level, allowing for more detailed sampling of distant light sources. The octahedral mapping ensures proper coverage of the hemisphere around each probe. 17
4.5 Probe-ray distribution in SPWI for the first 3 cascades. 18
4.6 Probe-ray distribution in SPWI for the first 3 cascades from a different angle. 18
4.7 Sets of probes between 2 cascades that are not on the same plane. . . 19
4.8 Bilinear interpolation and merging across 3 cascades. This image shows all the rays that contribute to a single ray in cascade 0 (in the center of the image). When interpolating between ray directions, depth awareness isn't needed, as depth differences are already handled by the probe weighting system. 20
4.9 Comparison between bilateral filtering and bilinear 3D. 22
4.10 Radiance store in octahedral maps from cascade 5 to cascade 0. . . . 25
4.11 The figures show how the octahedral maps are stored in the memory. Cascade 5 top left to cascade 0 bottom right. The dimensions follow the configuration seen on Table 4.2. 26

4.12 Visualization of cascades 0 and cascade 1 using the direction first layout on a 512×512 resolution drawable. 29

5.1 The orange probes represent the extra probes that will be traced, but will get averaged to the main probe of that neighborhood. These are real values for this thesis' cascade 0. 32
5.2 Left is regular probe grids. Right is twice as many probes on each direction on every cascade using pre-averaging. 34

6.1 The Sponza model rendered only with sky illumination. 35
6.2 Scene with four emissive cubes of different colors and two non emissive cubes. 36
6.3 A single emissive cube. 36
6.4 The Sponza model with 3 emissive hornbug models by McGuire [10] showcasing off-screen indirect light coming from a pink, blue and a yellow hornbug. 37
6.5 The Stanford bunny by Turk and Levoy [9] with a matte cube showcasing soft shadows. 37
6.6 The Stanford bunny with 2 emissive cuboids with daylight. 38
6.7 The Stanford bunny with 2 emissive cuboids with no daylight. 38

7.1 Min Max depth buffer mip hierarchy. The green highlights indicate regions with significant depth variation. Moving to higher mip levels (coarser resolutions), these highlights effectively mark depth discontinuities at increasingly larger scales. 43

List of Tables

4.1 Comparison of different Radiance Cascade implementations. 13
4.2 Cascade configuration for a 512×512 resolution scene. 15

6.1 Performance metrics for various scenes rendered with SPWI on a M1 MacBook Air with 8GB of unified memory. All measurements were taken at 1920×1080 resolution with 6 cascades. The memory shown in the table includes all the memory needed for the frame (acceleration structures, models, buffers and textures). 39

1 Introduction

Real-time global illumination remains one of the most challenging problems of modern computer graphics. While techniques based on path tracing have made significant strides, they often rely on denoisers and temporal accumulation, which can introduce artifacts in dynamic scenes. Radiance Cascades address this limitation by providing single-frame global illumination solutions that respond instantly to dynamic scenes without requiring temporal accumulation as discussed in [1].

This immediate response to changes in lighting, geometry, and viewpoint represents a significant advantage for applications where visual stability and responsiveness are paramount, such as video games and interactive visualizations. Unlike accumulation-based methods that may exhibit ghosting or blurring during fast movement, Radiance Cascades maintain consistent visual quality even in dynamic environments.

2
Previous Work

Real-time global illumination has been a fundamental challenge in computer graphics for decades, with various approaches attempting to balance visual quality, performance, and temporal stability. Solutions can be broadly categorized into three main approaches: screen-space techniques, probe-based methods, and path tracing approaches, each with distinct trade-offs that make them suitable for different applications.

Path tracing methods have achieved remarkable visual fidelity by simulating light transport with physical accuracy. Monte Carlo path tracing [11] provides the theoretical foundation for unbiased global illumination, while more recent advances like ReSTIR [12] and ReSTIR GI [13] have made real-time path tracing more practical by improving sampling efficiency. However, these methods typically require sophisticated denoising techniques [14] and temporal accumulation to achieve acceptable noise levels, which can introduce ghosting artifacts and temporal instability in dynamic scenes with moving objects or changing lighting conditions.

Screen-space global illumination techniques like SSAO [15] and SSGI [16] offer excellent performance but are fundamentally limited by the available screen-space information, making them unable to capture off-screen indirect lighting contributions. This limitation becomes particularly problematic in scenes with significant off-screen light sources or complex indirect lighting paths.

Probe-based methods, including Irradiance Volumes [17] and Light Propagation Volumes [18], have attempted to address these limitations by distributing light sampling throughout 3D space. More recent work on Dynamic Diffuse Global Illumination [19] has shown promise, but these approaches often suffer from high memory requirements and complex probe placement strategies.

While path tracing methods provide unbiased results, they require denoisers that SPWI avoids through its single-shot approach. Screen-space techniques offer excellent performance but cannot capture the off-screen indirect lighting that SPWI handles through world-space ray tracing. Traditional probe-based methods provide comprehensive 3D coverage but suffer from high memory requirements that SPWI addresses through its screen-space probe placement strategy. However, these existing approaches generally offer better temporal stability compared to SPWI's screen-space probe repositioning challenges.

3

Observations in 2D

The idea of Radiance Cascades is built on two important observations. The spatial and angular observations, which are key components to understand the *penumbra condition* that is mentioned in the paper.

3.1 Angular Observation

In Figure 3.1 a probe that casts 16 rays in the scene trying to locate light sources is presented. The angular resolution of a probe refers to the angle between the evenly spaced rays. A light source further away from the probe is not located due to the low angular resolution of the probe. The angle between rays for a probe depends on two factors.

i Δ The distance to the furthest object
ii **w** The size of the smallest object

In the paper this is formalized as $\Delta_\omega < w/\Delta$.
Which states that the angle between the evenly spaced rays Δ_ω should be smaller than w/Δ [2].

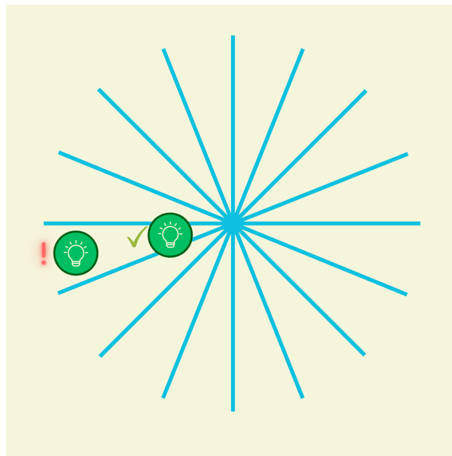


Figure 3.1: A probe and two light source, one of which it couldn't detect due to the low angular resolution.

3.2 Spatial Observation

In Figure 3.2 the closer a point is to the light source, the higher probe resolution it needs, to resolve the penumbra with interpolation and less spatial resolution but higher angular resolution in Figure 3.3. This illustrates the **inverse** relationship of the angular observation.

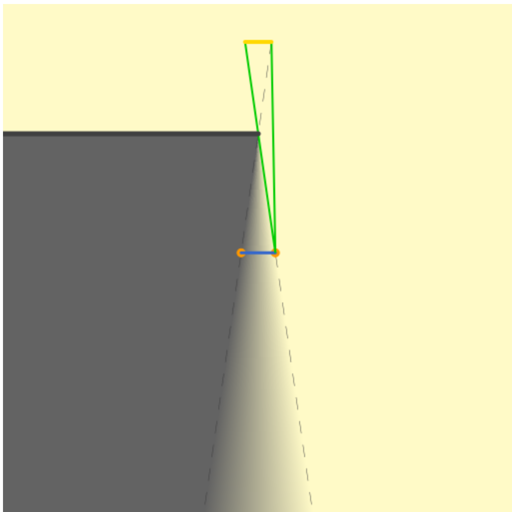


Figure 3.2: Interpolation of high spatial resolution probe grid. Image from Sannikov [1].

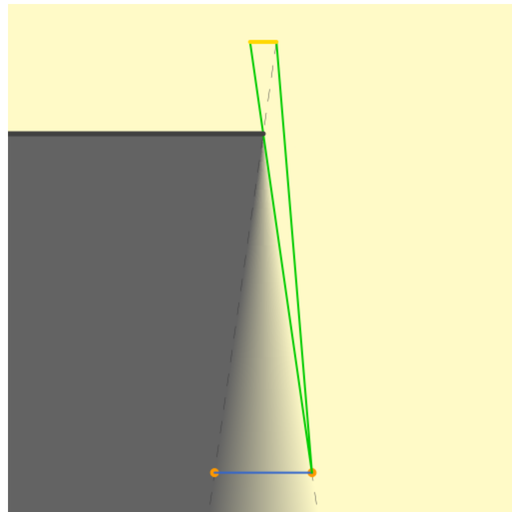


Figure 3.3: Interpolation of low spatial resolution probe grid. Image from Sannikov [1].

3.3 Penumbra Condition

The penumbra condition is the synthesis of both angular and spatial observations, establishing a fundamental relationship that guides the efficient sampling of light transport. It can be expressed mathematically as in Equation 3.1[1].

$$\begin{cases} \Delta_p <\sim D, \\ \Delta_\omega <\sim 1/D \end{cases} \tag{3.1}$$

Where Δ_p represents the spatial resolution (distance between probes), Δ_ω represents the angular resolution (angle between rays), and D is the distance to the occluder.

3.4 Radiance Cascade in 2D

In 2D, everything is happening on a flat plane as well as the casting of intervals. Before getting into the implementation of radiance cascades, this thesis is setting up an efficient way to march through the scene. The implementation is based on Jason's blog [3].

[1] $A <\sim B$ means that A is less than the output of some function, which scales linearly with B.

3.4.1 Efficient Ray Marching

To safely traverse the maximum possible distances while ray marching, without piercing through emissive surfaces that would create artifacts, a distance field is needed for the representation of the scene. This allows us to take optimally sized steps without penetrating surfaces. Starting with a drawing texture (Figure 3.4), a Signed Distance Field (SDF) was generated using the Jump Flood Algorithm (JFA).

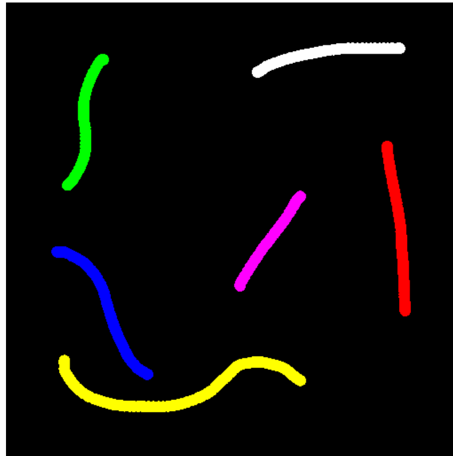


Figure 3.4: Drawn texture by the user that will be used as the geometry of the scene.

The Jump Flood Algorithm works by propagating the nearest "seed points" (edges of objects) throughout the entire texture. In each pass, pixels look at increasingly smaller neighborhoods to find the closest seed. The step size is halved in each iteration, allowing the information to flood through the entire texture in O(log n) time. Here is a simplified pseudocode representation of the JFA implementation.

Algorithm 1 Jump Flood Algorithm Implementation

1: **procedure** JFAPass(seedTexture, width, height)
2: renderA $\leftarrow$ CreateTexture(width, height)
3: renderB $\leftarrow$ CreateTexture(width, height)
4: input $\leftarrow$ seedTexture
5: output $\leftarrow$ renderA
6: passes $\leftarrow \lceil \log_2(max(width, height)) \rceil$
7: **for** stage = 0 to passes -1 **do**
8: stepSize $\leftarrow 2^{(passes-stage-1)}$
9: ApplyJFAShader(output, input, stepSize)
10: Swap(input, output)
11: **end for**
12: **end procedure**

Algorithm 2 JFA Shader Implementation

```
 1: procedure JFAShader(uv, texture, stepSize)
 2:     nearestSeed ← (0, 0, 0, 0)
 3:     nearestDist ← ∞
 4:     for x = −1 to 1 do
 5:         for y = −1 to 1 do
 6:             sampleUV ← uv + (x, y) × stepSize × (1/textureSize)
 7:             if sampleUV inside texture bounds then
 8:                 sampleSeed ← texture.Sample(sampleUV).xy
 9:                 if sampleSeed ≠ (0, 0) then
10:                     dist ← ||sampleSeed - uv||^2
11:                     if dist < nearestDist then
12:                         nearestDist ← dist
13:                         nearestSeed ← sampleSeed
14:                     end if
15:                 end if
16:             end if
17:         end for
18:     end for
19:     return nearestSeed
20: end procedure
```

This Jump Flood Algorithm produces the texture in Figure 3.5 where each pixel stores the coordinates of the nearest seed point (object edge). This JFA texture is then passed to a subsequent shader that converts these coordinates into actual distances, creating the final distance texture.

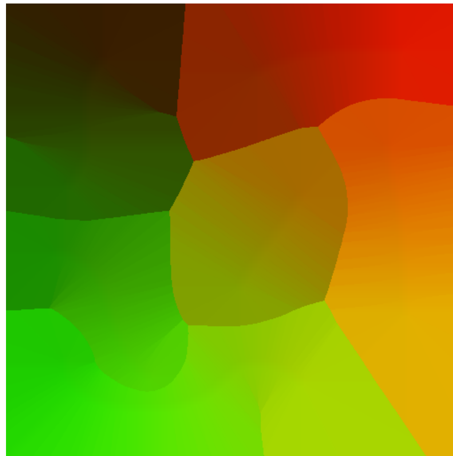


Figure 3.5: Jump Flood Algorithm end result.

Now the actual Euclidean distance can be calculated between the current pixel and the nearest seed point. Having a distance field as in Figure 3.6, it allows the ray marching algorithm to take optimally sized steps when casting rays, significantly improving performance by avoiding small, overly conservative steps in empty regions.

Figure 3.6: Distance texture that is used to step through in ray marching.

3.5 Cascades Creation Process

Calculate the maximum diagonal length can give the optimal number of cascades that will cover the whole screen:

```
1 float diagonal = sqrt(width * width + height * height);
2 float cascadeCount = ceil(log(diagonal) / log(baseRayCount)) + 1;
```

This formula ensures that there are just enough cascades to capture all potential on-screen light contributions. A branching factor of 4 was decided, of how the angular and spatial resolution will scale. the cascades are processed from highest to lowest, since the irradiance of furthest points from each pixel is carried down to the closest points while merging. Ping-pong textures, which is a technique to use only 2 render targets instead of one for each cascade, since only need 2 textures are needed for the merging process, saving a lot of memory.

Algorithm 3 Radiance Cascades Rendering Pass

```
1: procedure RCPass(resolution, drawingTexture, distanceTexture)
2:     diagonal ← sqrt(resolution.x^2 + resolution.y^2)
3:     cascadeCount ← ceil(log(diagonal) / log(baseRayCount)) + 1
4:     renderA ← CreateTexture(resolution)
5:     renderB ← CreateTexture(resolution)
6:     pingPongIndex ← 0
7:     lastMergeTexture ← null
8:     for cascade = cascadeCount-1 down to 0 do      ▷ Highest to lowest cascade
9:         rayCount ← pow(baseRayCount, cascade + 1)
10:        currentTarget ← (cascade == 0) ? finalOutput : [renderA, renderB][pingPongIndex]
11:        pingPongIndex ← 1 - pingPongIndex           ▷ Swap ping-pong texture
12:        RayMarch(currentTarget, resolution, drawingTexture, distanceTexture, lastMergeTexture, cascade, rayCount)
13:        if cascade > 0 then
14:            lastMergeTexture ← currentTarget
15:        end if
16:    end for
17:    return finalOutput
18: end procedure
```

The casting and merging process is used to hierarchically accumulate radiance information across multiple spatial scales. During the casting phase, rays are emitted from a grid of probes whose spacing increases exponentially with each cascade level while the number of rays increases. The spacing is determined using the square root of a base value raised to the power of the cascade index. Each probe emits a fixed number of rays distributed uniformly in angle over the full circle, and these rays march through the scene using the distance field created earlier to determine step sizes. At each step, the ray advances in the direction of its trajectory, and if a hit is detected on an opaque surface, the radiance is sampled from the color texture and used as the contribution for that ray. If no opaque surface is encountered within the allowed interval and if the cascade is not the highest, the algorithm merges the radiance from the upper cascade. This merging process involves mapping the current ray's position to a corresponding region in the upper cascade level, since they have different resolution, and sampling radiance from it using linear interpolation to ensure smooth transitions. This hierarchical approach allows radiance information to propagate from coarse to fine levels, ensuring that even regions not directly intersected by rays in finer cascades can still receive indirect lighting data from higher levels.

Algorithm 4 Simplified Ray Marching with Cascade Sampling

```
1: procedure RayMarch(probeUV, cascadeData)
2:     Calculate probe parameters (grid size, spacing, interval)
3:     radiance ← (0, 0, 0, 0)
4:     for each ray from the probe do
5:         startPoint ← probeUV + intervalStart × rayDirection
6:         radianceDelta ← (0, 0, 0, 0)
7:         if startPoint is within bounds then
8:             for step = 1 to maxSteps do
9:                 dist ← SampleDistanceField(currentPoint)
10:                currentPoint ← currentPoint + rayDirection × dist
11:                if out of bounds or traveled beyond interval then
12:                    break
13:                end if
14:                if hit surface (dist ≤ minStep) then
15:                    radianceDelta ← SampleColor(currentPoint)
16:                    break
17:                end if
18:            end for
19:            if no hit found and not in highest cascade then
20:                radianceDelta ← SampleFromHigherCascade()
21:            end if
22:        end if
23:        radiance ← radiance + radianceDelta
24:    end for
25:    return radiance / rayCount
26: end procedure
```

In Figure 3.7 the 6 cascades can be seen for a render target of 512×512 resolution. In this case, with a branching factor of 4, the highest cascade has a 16×16 probe grid with 4096 rays each probe, and subsequent cascades scale by having 4 times more probes and 4 times less rays compared to the previous pass. In the cascade images, every subsequent cascade carries down the radiance of the previous cascade by merging 4 rays from each of the 4 nearest probes.

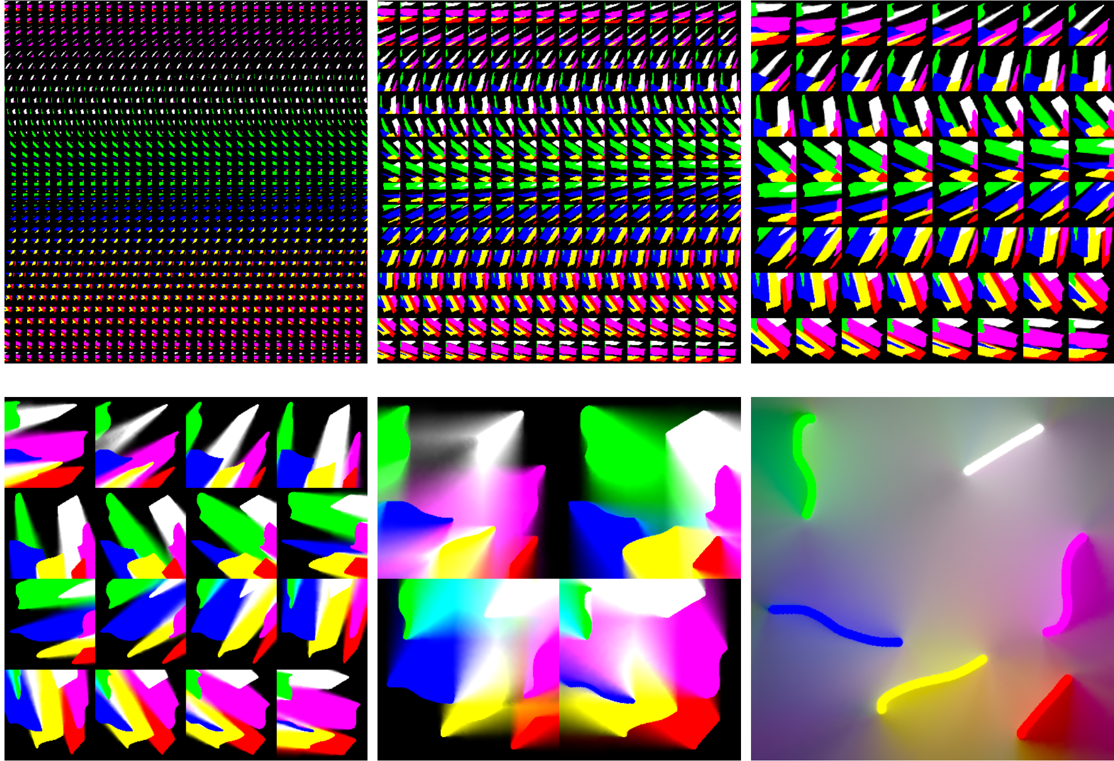


Figure 3.7: Radiance Cascade layers from cascade level 5 (top-left) to level 0 (bottom-right).

4

Implementation in a Higher Dimension

The implementation of real-time global illumination with Radiance Cascades poses different challenges depending on the specific approach chosen. In this section, three main variants of Radiance Cascades are explored, each with distinct trade-offs in terms of performance, memory usage, and lighting quality.

4.1 Comparison of a few Approaches

Technique	**Description**	**Pros**	**Cons**
Screen-Space Ray Marching	Operates entirely in screen space with probes and sampling confined to visible surfaces	Extremely fast performance; Very memory efficient	Cannot capture off-screen indirect lighting;
3D Probe Grid	Distributes probes throughout the entire 3D scene space in a volumetric grid	Captures off-screen indirect lighting; View-independent results; Very stable lighting across camera movements	High memory requirements; Very slow compared to Screen-Space Ray Marching
SPWI	Places probes in screen space but samples rays in world space	Captures off-screen indirect lighting; Memory-efficient probe hierarchy	Very slow compared to Screen-Space Ray Marching;

Table 4.1: Comparison of different Radiance Cascade implementations.

The Screen-Space approach, notably implemented in Path of Exile 2, offers exceptional performance but is most suitable for specific viewpoints (such as top-down games) where off-screen lighting has limited visual impact. At the other end of the spectrum, 3D grid-based methods provide comprehensive light transport simulation at the cost of significant memory usage and longer frame times, which can become prohibitive in large environments.

The Screen-Space Probes with World-Space Intervals (SPWI) approach, which is the focus of this implementation, presents a compelling middle ground. By placing probes in screen space while tracing rays through world space, SPWI achieves much

of the lighting quality of full 3D approaches for 1 bounce of indirect light, while maintaining memory efficiency closer to screen-space methods. This makes it particularly suitable for applications that require capturing off-screen lighting without the memory overhead of a full 3D solution.

4.2 Placing the Probes in Depth

The first step in tracing rays for radiance is to find the starting position of each cascade's rays, which means placing the probes in the depth buffer. This implementation uses a linear depth buffer, which is necessary for the bilateral filtering to detect edges using a threshold. Therefore, reconstruction of 3D positions from the linear depth buffer is needed. Unlike standard depth buffers that store nonlinear values, the linear depth buffer contains actual distances from the camera.

As shown in Algorithm 5, the reconstruction process begins by creating a ray from the camera through the probe's screen position to the near plane. It then scales this ray by the linear depth value to find the exact world position at that depth. A small depth bias (0.98) is applied to prevent surface acne artifacts in subsequent ray tracing operations.

Algorithm 5 World Position Reconstruction from Linear Depth

1: **procedure** ReconstructWorldPosition($ndc, linearDepth$)
2: $clipPos \leftarrow (ndc.x, ndc.y, -1.0, 1.0)$
3: $viewPos \leftarrow projection_matrix_inverse \times clipPos$
4: $viewPos \leftarrow viewPos/viewPos.w$
5: $scale \leftarrow linearDepth/|viewPos.z|$
6: $depthBias \leftarrow 0.98$
7: $viewPosAtDepth \leftarrow viewPos.xyz \times (scale \times depthBias)$
8: $worldPos \leftarrow view_matrix_inverse \times (viewPosAtDepth, 1.0)$
9: **return** $worldPos.xyz$
10: **end procedure**

This accurate world-space positioning is crucial for SPWI, as it ensures probes are correctly placed on visible surfaces while enabling rays to be cast through world space. In Figure 4.1 the probes for cascade 0 are seen when placed on the depth buffer, and in Figure 4.2 the same probes from a different angle.

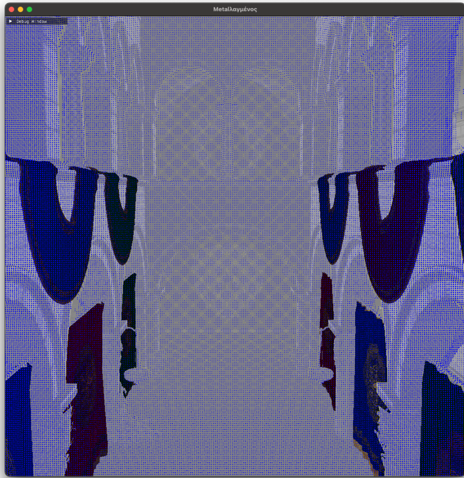


Figure 4.1: Screen space probes of cascade 0.

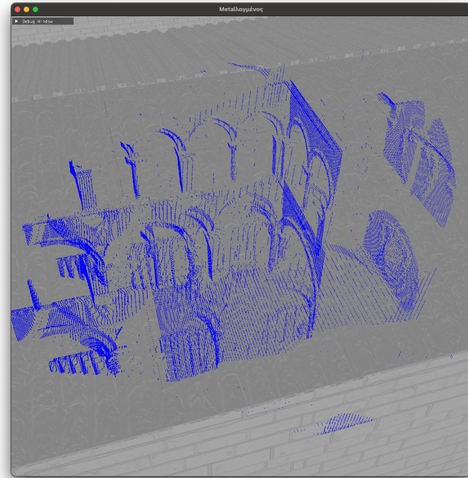


Figure 4.2: Screen space probes of cascade 0 from a different angle.

An optimal cascade configuration maintains consistent computational cost or less, across all cascade levels while providing appropriate sampling density at different distances. This implementation uses a branching factor of 4 for ray counts, combined with a corresponding reduction in probe density.

Specifically, each subsequent cascade level:

- Increases the number of rays per probe by a factor of 4
- Reduces the probe grid density by a factor of 2 in each dimension

This scaling approach ensures that the total computational cost remains approximately constant across cascade levels. With each cascade requiring roughly the same amount of resources, cascades can be added until the desired lighting coverage is achieved without disproportionately increasing rendering time.

For example, in a scene at 512×512 resolution, using a base ray count of 16 and initial probe spacing of 4 pixels, the cascade hierarchy is structured as follows:

Cascade Level	Probe Grid	Rays per Probe	Probe Spacing
0	128×128	16	4 pixels
1	64×64	64	8 pixels
2	32×32	256	16 pixels
3	16×16	1,024	32 pixels
4	8×8	4,096	64 pixels
5	4×4	16,384	128 pixels

Table 4.2: Cascade configuration for a 512×512 resolution scene.

Through experimentation, six cascade levels seemed to provide the optimal balance between performance and lighting quality for the test scenes. This configuration captures off-screen indirect lighting effectively.

4.3 Encoding Intervals using Octahedral Maps

An efficient way is required to encode directional data for the ray samples. Octahedral mapping provides an elegant solution to this problem by mapping 3D unit vectors onto a 2D surface, allowing for efficient storage and retrieval of directional information and making the merging step later on a lot easier. The octahedral mapping technique represents directions by projecting a unit sphere onto an octahedron as seen in Figure 4.3, which can then be unfolded into a 2D square. This mapping is both area-preserving and computationally efficient, making it highly suitable for graphics applications where direction vectors need to be stored and accessed frequently.

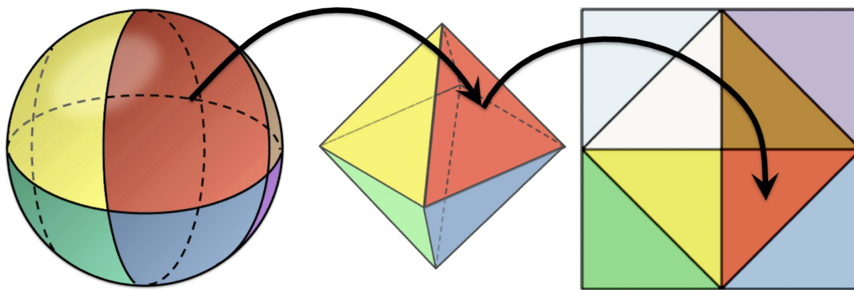
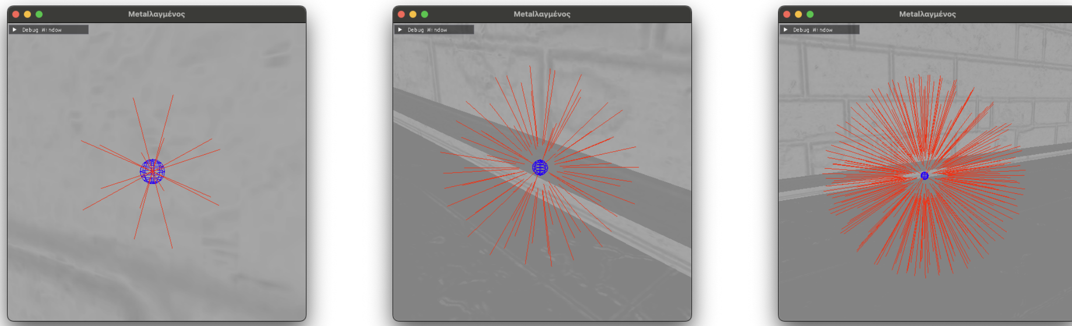


Figure 4.3: Converting a sphere into an octahedral map. Image from Cigolle et al. [4].

The visualization in Figure 4.4 clearly shows how the angular sampling density increases with each cascade level - from 16 rays in Cascade 0 to 64 rays in Cascade 1 and 256 rays in Cascade 2. The uniformity of ray distribution achieved through octahedral mapping is particularly valuable in the cascade merging step, where rays from higher cascades must be matched with their corresponding rays in lower cascades. This uniform distribution ensures that transitions between cascade levels remain smooth and coherent, making octahedral maps especially well-suited for the hierarchical structure of radiance cascades.



Cascade 0: 16 rays　　Cascade 1: 64 rays　　Cascade 2: 256 rays

Figure 4.4: Ray distribution patterns encoded using octahedral maps for the first three cascade levels. The images show how angular resolution increases with each cascade level, allowing for more detailed sampling of distant light sources. The octahedral mapping ensures proper coverage of the hemisphere around each probe.

4.4 Cascade Hierarchy in SPWI

Having established the theoretical foundation with the angular and spatial observations, implementing these principles using ray tracing intervals starts getting more intuitive. These intervals translate the penumbra condition into a practical sampling strategy. For each cascade level, a specific start and end distances are defined, that determine where rays should sample the scene:

Algorithm 6 Interval Calculation for Cascades

1: **procedure** CalculateInterval(cascadeLevel, intervalLength)
2: const float baseCascadeRange ← 0.016f
3: const float cascadeRangeMultiplier ← 4.0f
4: float cascadeStartRange ← (cascadeLevel == 0) ? 0.0f : (baseCascadeRange × pow(cascadeRangeMultiplier, float(cascadeLevel - 1)))
5: float cascadeEndRange ← baseCascadeRange × pow(cascadeRangeMultiplier, float(cascadeLevel))
6: **end procedure**

This interval structure directly implements the observations: moving to higher cascades, more distant regions are progressively sampled with increased angular resolution but decreased spatial resolution. The first cascade samples nearby geometry with high spatial precision, while subsequent cascades handle increasingly distant regions with enhanced angular detail. The factor of 4 used in the interval scaling matches the cascade branching factor, ensuring that the sampling distribution maintains consistent quality throughout the scene. When rays from cascade i find no intersections within their designated interval, they merge with information from cascade $i + 1$. Figure 4.5 illustrates how these intervals appear in 3D space from the top view and Figure 4.6 show the same illustration from a different angle.

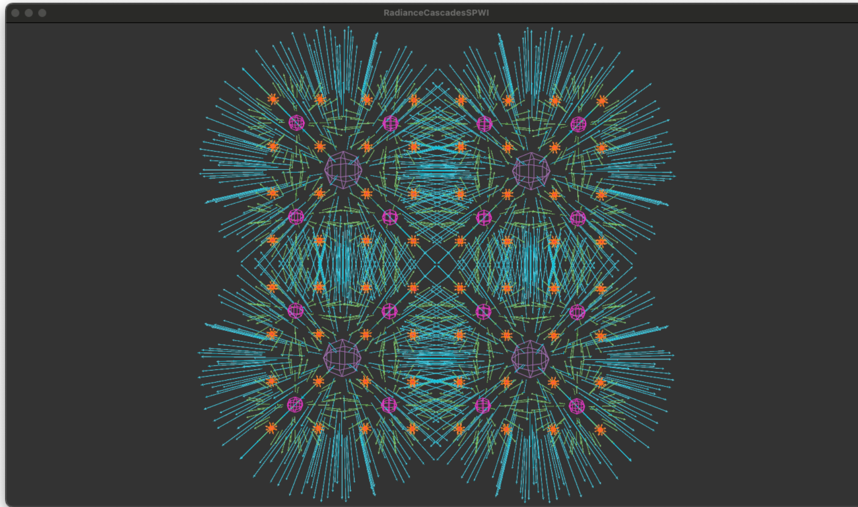


Figure 4.5: Probe-ray distribution in SPWI for the first 3 cascades.

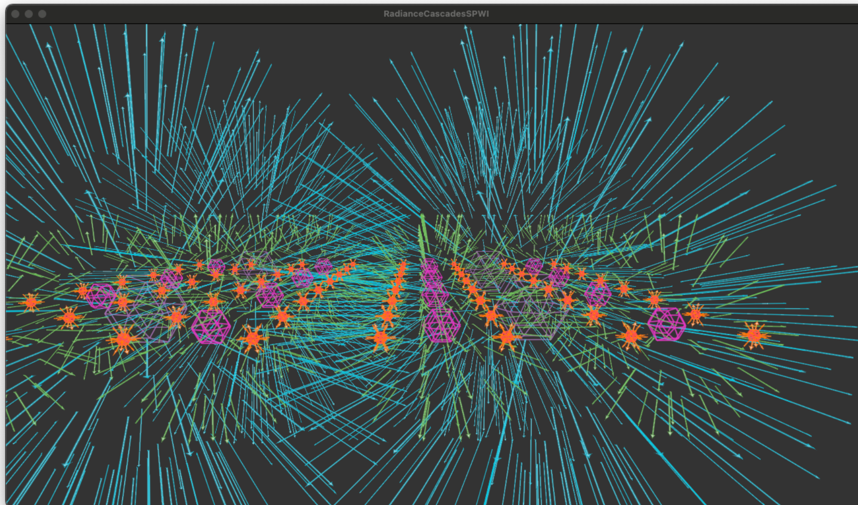


Figure 4.6: Probe-ray distribution in SPWI for the first 3 cascades from a different angle.

These are the first 3 cascades where they have the shorter intervals. One can now imagine how these screen space probes can capture off-screen radiance. In a case where the camera is positioned very close at a non-emissive surface, the probes will be placed on that surface, but the higher cascades' intervals will extend behind the camera.

4.5 Merging with the Upper Cascade

After tracing all the rays of the cascade and gathering the radiance, the next step is to merge the radiance with the upper cascade, except if it is the most upper cascade. Doing just bilinear interpolation to find the nearest 4 upper probes to merge with, will create artifacts. In Figure 4.7 the probes are not on the same plane as in the

previous visualization. If the merging happens naively (by doing bilinear interpolation), the pink probe will sample radiance from the 4 nearest blue probes (which are the upper cascade's) with similar weights. In reality, the outlier probes should have smaller weights, based on their distance to the pink probe.

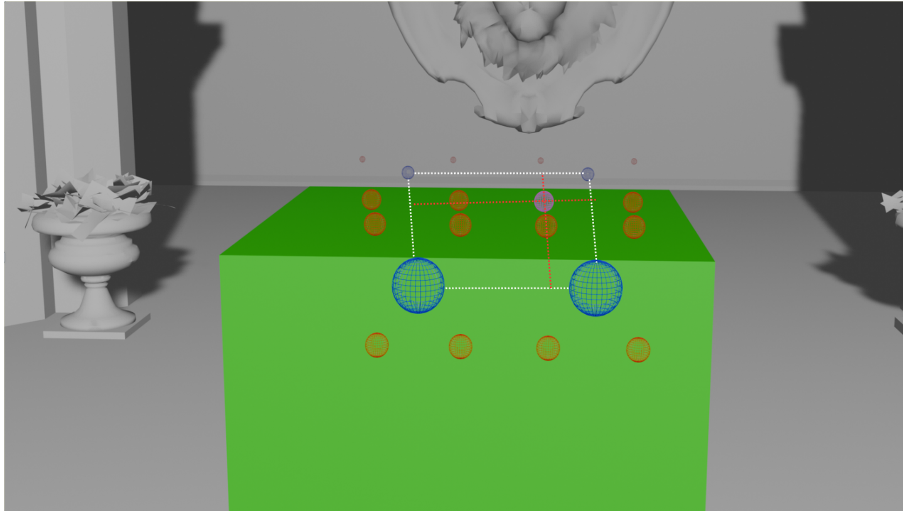


Figure 4.7: Sets of probes between 2 cascades that are not on the same plane.

4.5.1 Bilateral Filtering

To solve the issue of different probe depths, some depth aware upscaling method is needed when sampling upper probes. In this thesis, two different approaches were tested regarding this problem. First, the bilateral filtering technique, as seen by [5], adjusts the interpolation weights based on depth differences, giving less influence to probes that are at significantly different depths from the current sample point. This helps maintain proper spatial relationships and prevents bleeding of light across depth discontinuities. Algorithm 7 demonstrates the depth-aware cascade merging approach, which ensures smooth transitions between cascade levels in SPWI while respecting scene geometry.

The algorithm's key feature is its ability to detect depth discontinuities among the 4 nearest upper cascade probes. When significant depth variations are detected (indicating geometry boundaries), it applies a bilateral filtering technique that reduces the influence of probes at different depths than the current position. This prevents light bleeding across geometric boundaries while maintaining smooth illumination within similar depth regions.

The merging process also handles directional sampling through octahedral mapping, ensuring consistent ray direction mapping between cascade levels. For each probe, the algorithm samples from surrounding directions in the octahedral map and applies bilinear interpolation for smooth angular transitions as seen in Figure 4.8.

Algorithm 7 Depth-Aware Cascade Merging

```
1: procedure MergeWithUpperCascade(probeUV, rayDirection, currentDepth)
2:     upperLevel, upperGridSize, upperRaysPerDim ← CalculateUpperCascadeParams()
3:     upperProbeCoords ← FindNearestUpperProbes(probeUV)
4:     probeDepths ← SampleDepthForProbes(upperProbeCoords)
5:     dirCoords ← ToOctahedralCoordinates(rayDirection, upperRaysPerDim)
6:     bilinearWeights ← CalculateBilinearWeights(probeUV)
7:     // Bilateral depth filtering
8:     minDepth, maxDepth ← GetMinMaxDepth(probeDepths)
9:     depthRange ← maxDepth - minDepth
10:    avgDepth ← Average(probeDepths)
11:    isDepthEdge ← (depthRange / avgDepth) > 0.1
12:    weights ← bilinearWeights
13:    if isDepthEdge then
14:        depthDiff ← abs(probeDepths - float4(currentDepth))
15:        weights ← weights × (float4(1.0) / (depthDiff + float4(0.0001)))
16:    end if
17:    weights ← weights / (weights.x + weights.y + weights.z + weights.w)
18:    radiance ← (0, 0, 0, 0)
19:    for each probe with weight > 0 do
20:        probeRadiance ← BilinearSampleDirections(dirCoords, probeCoord)
21:        radiance ← radiance + probeRadiance × weights[probeIndex]
22:    end for
23:    return radiance
24: end procedure
```

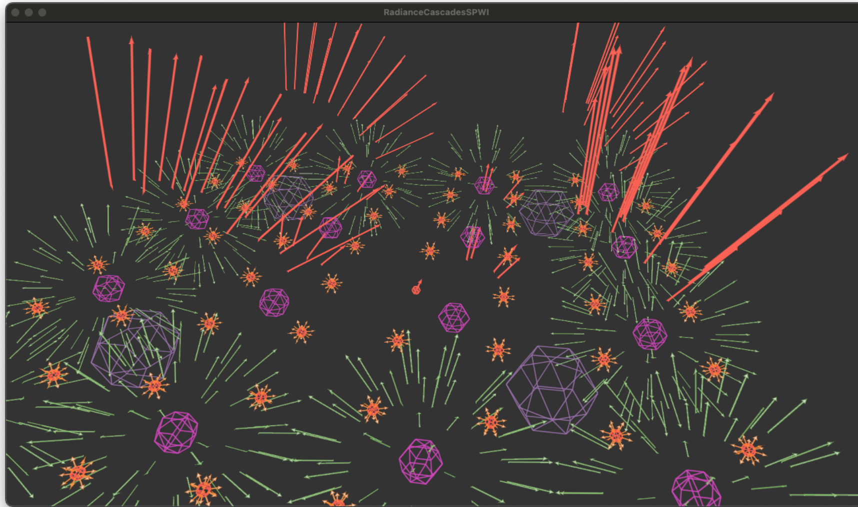


Figure 4.8: Bilinear interpolation and merging across 3 cascades. This image shows all the rays that contribute to a single ray in cascade 0 (in the center of the image). When interpolating between ray directions, depth awareness isn't needed, as depth differences are already handled by the probe weighting system.

4.5.2 Bilinear 3D

Alexander Sannikov came up with a better idea for depth aware upscaling which he showcased in shadertoy [6]. The 3D bilinear approach addresses a fundamental limitation of the bilateral filtering method. While bilateral filtering can adjust weights based on depth differences, it still operates fundamentally in 2D space and only considers depth as a factor to modify weights. In contrast, the 3D bilinear method directly performs interpolation in full 3D world space, resulting in more accurate spatial relationships. The approach of 3D bilinear is showcased in Algorithm 8, which is the same simplified function that was used for bilateral, but calculating the weights differently. It takes into account the actual 3D positions of probes rather than just their depths and it iteratively refines interpolation ratios by projecting the target point onto interpolation lines in 3D space.

Algorithm 8 3D Bilinear Interpolation for Cascade Merging

```
 1: procedure MergeWithUpperCascade3D(probeUV, rayDir, currentWorldPos)
 2:     upperParams ← CalculateUpperCascadeParams()
 3:     probeCoords, probeUVs ← FindNearestUpperProbes(probeUV)
 4:     probeWorldPositions ← ReconstructProbeWorldPositions(probeUVs)
 5:     initialRatio ← CalculateScreenSpaceRatio(probeUV)
 6:     ratio3d ← GetBilinear3DRatio(probeWorldPositions, currentWorldPos, initialRatio, 2)
 7:     weights ← CalculateBilinearWeights(ratio3d)
 8:     dirCoords ← ToOctahedralCoordinates(rayDir, upperParams.raysPerDim)
 9:     radiance ← GatherAndBlendSamples(probeCoords, dirCoords, weights)
10:     return radiance
11: end procedure
12: procedure GetBilinear3DRatio(srcPoints, dstPoint, initialRatio, iterations)
13:     ratio ← initialRatio
14:     for i = 0 to iterations-1 do
15:         // Interpolate along Y and find improved X ratio
16:         lineStart ← Mix(srcPoints[0], srcPoints[2], ratio.y)
17:         lineEnd ← Mix(srcPoints[1], srcPoints[3], ratio.y)
18:         ratio.x ← ProjectPointOntoLine(lineStart, lineEnd, dstPoint)
19:         // Interpolate along X and find improved Y ratio
20:         lineStart ← Mix(srcPoints[0], srcPoints[1], ratio.x)
21:         lineEnd ← Mix(srcPoints[2], srcPoints[3], ratio.x)
22:         ratio.y ← ProjectPointOntoLine(lineStart, lineEnd, dstPoint)
23:     end for
24:     return ratio
25: end procedure
```

In Figure 4.9 the differences between the 2 approaches can be seen. The striations that appear in the bilateral are easier seen from steep angles like the one in the figure. However, there is a better approach to solve the interpolation of probes for radiance cascades, which is discussed in the section Future Work.

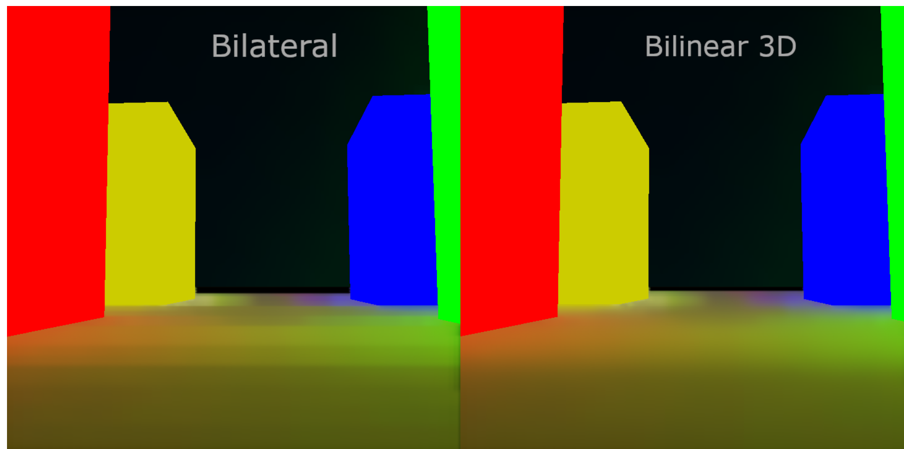


Figure 4.9: Comparison between bilateral filtering and bilinear 3D.

4.5.3 Cascade Dispatching Strategy

The dispatching strategy processes cascade levels in descending order, starting from the highest cascade (with the lowest spatial resolution but highest angular resolution) and working down to the lowest cascade. This order ensures that each cascade level can access the results from higher cascades during the merging process.

An easy optimization in the implementation is the use of ping-pong render targets, as seen in the 2D implementation of Jason [3]. Rather than allocating a separate texture for each cascade level, which would consume significant memory, alternation between two render targets is preferred. Since cascades are processed sequentially and each cascade only needs information from the immediately preceding cascade, this approach reduces memory usage without sacrificing correctness. In Algorithm 9 an outline of how to dispatch the threads can be seen. For each cascade, a ray per thread is dispatched, which is the same number for all of the cascades when using a branching factor of 4.

Algorithm 9 Cascade Level Dispatching

```
 1: procedure DispatchRaytracingKernel
 2:     renderTargets ← CreatePingPongRenderTargets()
 3:     lastMergedTexture ← nil
 4:     pingPongIndex ← 0
 5:     for level = maxCascadeLevel down to 0 do
 6:         if level == maxCascadeLevel - 1 then
 7:             currentRenderTarget ← renderTargets[pingPongIndex]
 8:             pingPongIndex ← 1 - pingPongIndex      ▷ Alternate between targets
 9:             lastMergedTexture ← null
10:         else if level > 0 then
11:             currentRenderTarget ← renderTargets[pingPongIndex]
12:             pingPongIndex ← 1 - pingPongIndex
13:         else
14:             currentRenderTarget ← finalGatherTexture      ▷ Final output target
15:         end if
16:         // Configure cascade-specific parameters
17:         cascadeData.level ← level
18:         // more parameters
19:         tileSize ← probeSpacing × (1 ≪ level)
20:         probeGridSize ← CalculateProbeGrid(framebufferSize, tileSize)
21:         raysPerDim ← (1 ≪ (level + 2))
22:         totalThreads ← probeGridSize.x × probeGridSize.y × raysPerDim^2
23:         SetupRayTracingResources(currentRenderTarget, lastMergedTexture)
24:         DispatchThreads(totalThreads)
25:         if level > 0 then
26:             lastMergedTexture ← currentRenderTarget
27:         end if
28:     end for
29: end procedure
```

4.5.4 Ray Tracing Kernel

The ray tracing kernel is the core component of the implementation, responsible for casting rays from each probe and gathering radiance information. Algorithm 10 shows the pseudocode for this kernel.

Algorithm 10 Ray Tracing Kernel

```
1: procedure RayTracingKernel(radianceTexture, upperRadianceTexture, cascadeData, depthTexture)
2:     tileSize ← probeSpacing × (1 ≪ cascadeLevel)
3:     probeGridSize ← CalculateProbeGrid(framebufferSize, tileSize)
4:     raysPerDim ← (1 ≪ (cascadeLevel + 2))
5:     probeIndex, rayIndex ← MapThreadIDToProbeAndRay(threadID)
6:     probeUV ← CalculateProbeUV(probeIndex, probeGridSize)
7:     probeDepth ← SampleDepthTexture(probeUV)
8:     worldPos ← ReconstructWorldPosition(probeUV, probeDepth)
9:     rayUV ← CalculateRayUV(rayIndex, raysPerDim)
10:    rayDir ← OctDecode(rayUV)
11:    // Calculate ray interval
12:    const float baseCascadeRange ← 0.016f
13:    const float cascadeRangeMultiplier ← 4.0f
14:    float cascadeStartRange ← (cascadeLevel == 0) ? 0.0f : (baseCascadeRange × pow(cascadeRangeMultiplier, float(cascadeLevel - 1)))
15:    float cascadeEndRange ← baseCascadeRange × pow(cascadeRangeMultiplier, float(cascadeLevel))
16:    ray ← CreateRay(worldPos, rayDir, intervalStart, intervalEnd)
17:    hitResult ← TraceRay(ray, scene)
18:    radiance ← (0, 0, 0, 0)
19:    occlusion ← hitResult.hit ? 0.0 : 1.0
20:    if hitResult.hit then
21:        radiance ← GetSurfaceRadiance(hitResult)
22:    else if cascadeLevel == maxCascadeLevel and enableSkyOrSun then
23:        radiance ← CalculateSkyAndSunRadiance(rayDir)
24:    end if
25:    if cascadeLevel < maxCascadeLevel then
26:        upperRadiance ← MergeWithUpperCascade(upperRadianceTexture, probeUV, rayDir, probeDepth, worldPos)
27:        if not hitResult.hit then
28:            radiance ← upperRadiance
29:        else
30:            radiance.rgb ← radiance.rgb + upperRadiance.rgb × occlusion
31:            radiance.alpha ← radiance.alpha × upperRadiance.alpha
32:        end if
33:    end if
34:    outputPosition ← CalculateOutputPosition(probeUV, rayUV, tileSize)
35:    WriteRadiance(radianceTexture, outputPosition, radiance)
36: end procedure
```

Several aspects of the ray tracing kernel in Algorithm 10 deserve special attention:

1. **Interval-based sampling**: Each cascade level is responsible for a specific distance interval. This ensures that cascade 0 samples nearby geometry, while higher cascades handle progressively more distant regions.

2. **Occlusion factor**: The occlusion factor plays a crucial role in the cascade merging strategy. When a ray hits geometry within its assigned interval, the occlusion factor is set to 0, indicating that light from beyond this point is fully blocked. When no geometry is hit, the occlusion factor is set to 1, allowing radiance from higher cascades to contribute fully.
3. **Sky sampling strategy**: Sky radiance is only calculated at the highest cascade level. This sky radiance is then propagated down through the cascade hierarchy via the merging process.
4. **Cascade merging**: The merging process is where the algorithm brings together information from different cascade levels. For rays that don't hit anything in their interval, the radiance from the upper cascade is used directly. For rays that do hit geometry, the upper cascade's contribution is weighted by the occlusion factor.

4.5.5 Tracing and Merging Results

After the complete cascade processing, all global illumination information converges into cascade 0, which contains the fully merged radiance from all higher cascade levels. Figure 4.10 illustrates the octahedral maps produced during each render pass of the cascade system. The rightmost image in the bottom row represents the final result in cascade 0, which encapsulates the complete global illumination solution, incorporating both nearby interactions from cascade 0's high spatial resolution and distant lighting contributions from higher cascades' enhanced angular resolution.

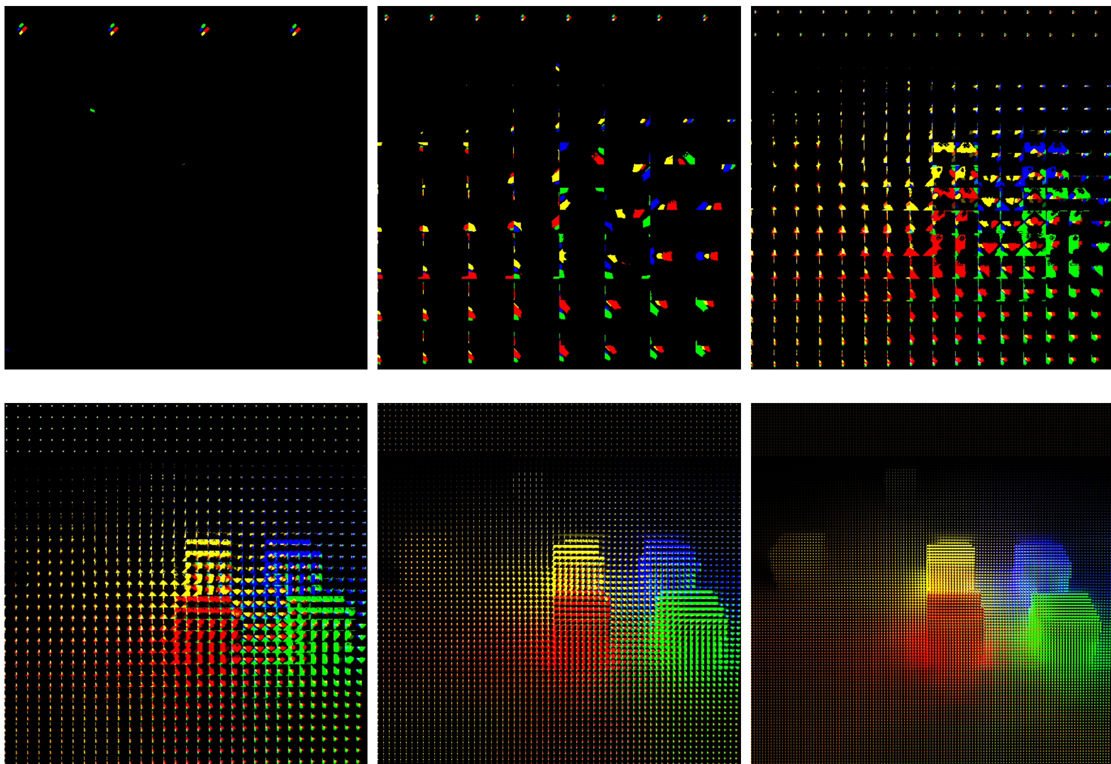


Figure 4.10: Radiance store in octahedral maps from cascade 5 to cascade 0.

4.6 Final Composition

The octahedral maps store radiance information indexed by direction, where the UV coordinates can be decoded to extract the corresponding 3D ray direction, and the stored value represents the incoming radiance along that direction. Figure 4.11 illustrates how these maps are represented in memory. For the final composition, only the octahedral maps from cascade 0 are needed, as they already contain the complete merged result from all higher cascades. While Figure 4.10 shows that the cascade 0 maps effectively capture the scene's illumination structure, this information cannot be directly applied to the geometry. Instead, as the cascade levels were merged using depth-aware upscaling and bilinear interpolation among rays, a similar procedure for the final scene composition is applied. Each pixel in the final image samples from the 16 directions of all 4 nearest probes of cascade 0 octahedral maps (bottom right of Figure 4.11), using the same depth-aware interpolation technique to correctly gather indirect illumination for its specific position and normal direction.

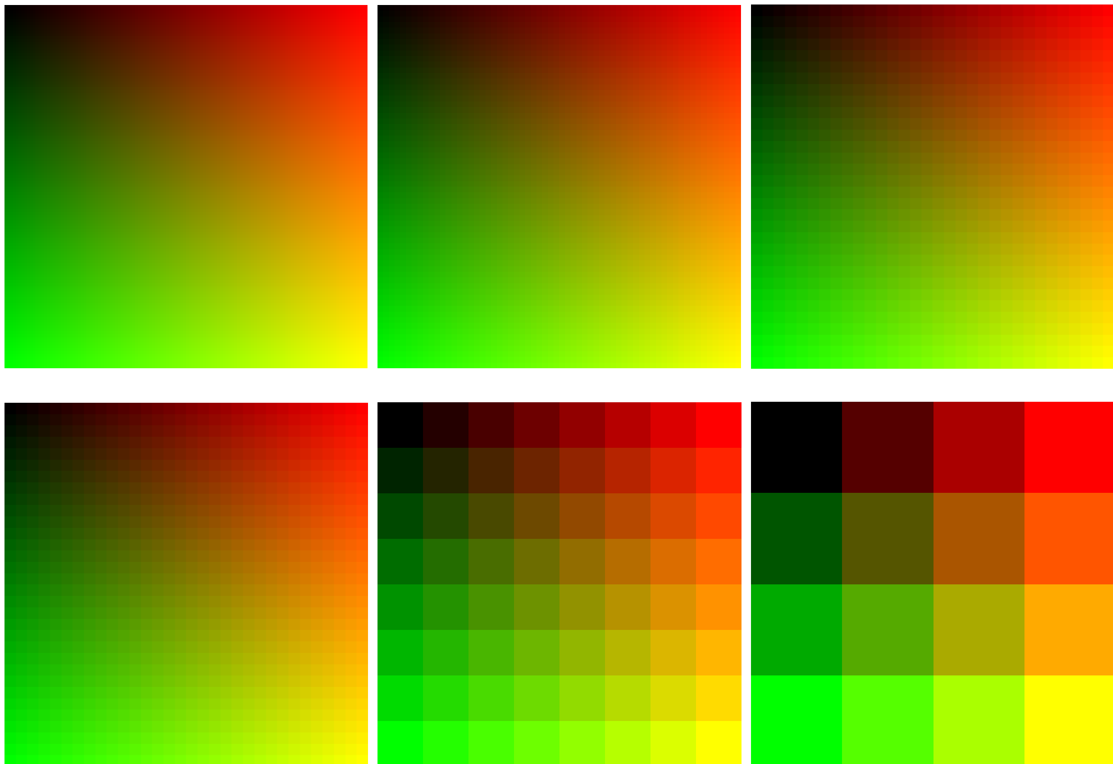


Figure 4.11: The figures show how the octahedral maps are stored in the memory. Cascade 5 top left to cascade 0 bottom right. The dimensions follow the configuration seen on Table 4.2.

The final gathering process is where the combination of the pre-computed global illumination happens, with the scene's surface properties to render the final image. The Algorithm 11 illustrates several aspects of this process:

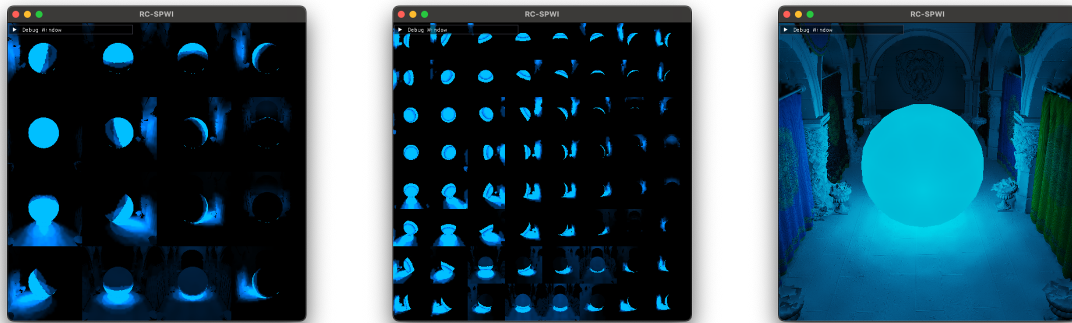
1. **Depth-aware probe interpolation**: Even at the final gathering stage, depth-aware interpolation must be applied between the current pixel and the 4 nearest probes from cascade 0. This is crucial because these probes may still be positioned at different depths than the current pixel, potentially causing light bleeding artifacts at geometric boundaries. Similar to cascade merging, the algorithm uses bilateral filtering to adjust interpolation weights based on depth differences. It was found that bilateral filtering alone was sufficient for depth-awareness at this stage, as the striation artifacts that motivated the use of bilinear 3D interpolation for cascade merging are not visible within the short-range intervals of cascade 0.
2. **Lambert's cosine law integration**: For each nearby probe, the algorithm samples the octahedral map across multiple directions, weighting each direction by the cosine of the angle between the surface normal and the light direction.
3. **Four-probe blending**: Just as with cascade merging, the final gathering step blends weighted contributions from the 4 nearest probes to produce smooth lighting transitions across the scene.

Algorithm 11 Final Image Composition with GI

```
1: procedure FinalGatherFragment(texCoords, radianceTexture, gBuffer)
2:     albedo, normal, depth ← SampleGBuffer(texCoords)
3:     if IsEmptySurface(albedo) then
4:         return RenderSky(texCoords)
5:     end if
6:     probeGridSize, probeCoords, probeUVs ← GetNearbyProbeData(texCoords)
7:     probeDepths ← SampleProbeDepths(probeUVs)
8:     // Apply bilateral filtering (similar to Algorithm 7)
9:     isDepthEdge ← (depthRange / avgDepth) > 0.05
10:    weights ← CalculateBilinearWeights(texCoords, probeGridSize)
11:    if isDepthEdge then
12:        weights ← ApplyDepthBasedWeighting(weights, probeDepths, depth)
13:    end if
14:    // Sample from octahedral maps with cosine weighting
15:    for each nearby probe i do
16:        radianceSum, totalWeight ← 0, 0
17:        for each direction in octahedral map do
18:            direction ← OctDecode(dirUV)
19:            cosTheta ← max(0, dot(normal, direction))
20:            radiance ← GetRadianceSample(probeUVs[i], dirUV, isEmissive, albedo)
21:            radianceSum ← radianceSum + radiance × cosTheta
22:            totalWeight ← totalWeight + cosTheta
23:        end for
24:        probeRadiance[i] ← radianceSum / max(totalWeight, 0.0001)
25:    end for
26:    finalRadiance ← weights[0] × probeRadiance[0] + weights[1] × probeRadiance[1] + weights[2] × probeRadiance[2] + weights[3] × probeRadiance[3]
27:    return albedo × finalRadiance.rgb
28: end procedure
```

4.6.1 Direction First Layout

When re-examining the ray storage approach for octahedral maps, the direction-first layout was found to provide significantly more cohesive results when debugging cascades. In the previous implementation, each octahedral stored all of its directions in proximity, which resulted in similar directions from different probes being scattered across distant locations in texture memory. The direction-first layout organizes the texture memory differently: each block represents a specific ray direction, with the pixels within that block containing the same direction from every probe. This organization explains the cohesive results shown in Figure 4.12. By storing similar rays adjacent to each other, hardware interpolation becomes possible during the merging step with higher cascades. Additionally, this approach makes debugging cascades substantially easier compared to the probe-first layout.



Cascade 0: 16 rays | Cascade 1: 64 rays | Final image

Figure 4.12: Visualization of cascades 0 and cascade 1 using the direction first layout on a 512×512 resolution drawable.

5 Experiments

Storing rays for radiance cascades can be approached in two ways: using storage buffers or textures. This thesis adopted the latter approach due to its simplicity, resulting in octahedral map textures matching the drawable's resolution. Increasing spatial resolution (e.g., reducing probe spacing from 4 to 2 pixels in cascade 0) could potentially improve visual quality, but it would require textures 4 times larger for each cascade, substantially increasing memory requirements.

5.1 Pre Averaging

A promising solution is the pre-averaging optimization proposed by Alex and Xor [7]. This technique essentially averages 4 probe rays into 1 during the ray tracing step. This approach offers multiple benefits:

- Maintains original texture sizes, significantly reducing memory consumption
- Improves performance by reducing the number of merging operations by 75%
- Captures radiance from a finer probe grid, delivering better visual quality

This optimization represents an effective compromise between visual fidelity and resource efficiency.

In Figure 5.1 is shown what needs to be implemented in order to achieve this. Probe #0 will be the main probe of the neighborhood that will be written to the final texture, and the orange probes are existing only during the ray tracing part. It needs to average all the similar directions of all the neighboring probes.

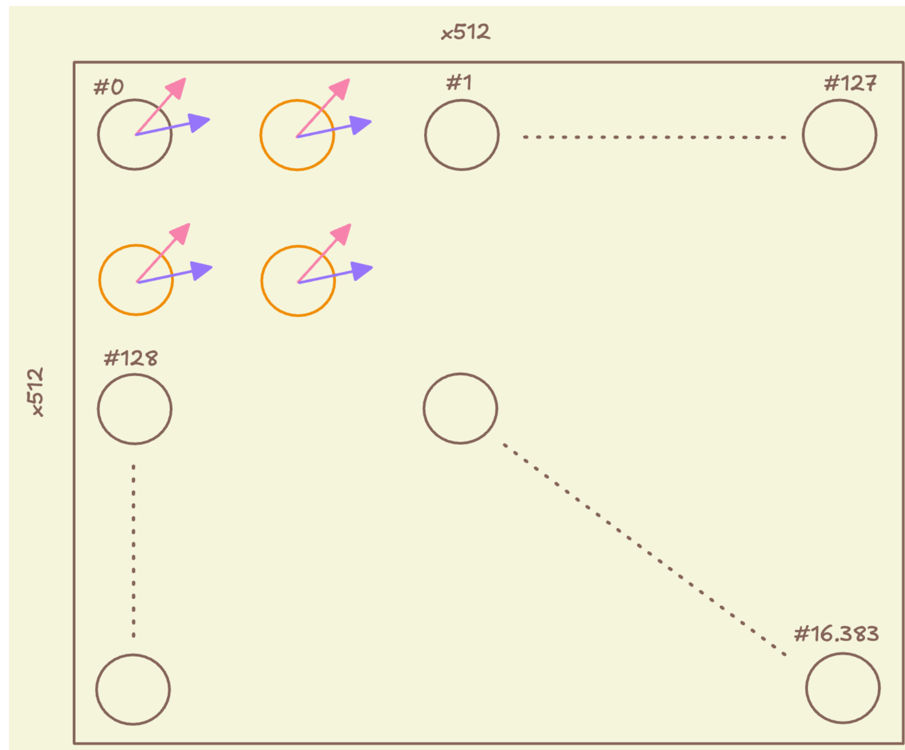


Figure 5.1: The orange probes represent the extra probes that will be traced, but will get averaged to the main probe of that neighborhood. These are real values for this thesis' cascade 0.

In the drawn representation of Figure 5.1, a way to optimally get the results of neighboring rays using SIMD subgroups can be easily seen. Since it needs to average 4 rays at the same time, and modern GPUs typically have SIMD groups (warps/wavefronts) of 32 or 64 threads executing in lockstep, it is possible to dispatch a logic that takes advantage of this hardware characteristic. This approach eliminates the need for shared GPU memory and synchronization primitives that would otherwise be required for the averaging operation.

5.1.1 Thread Organization for Pre-Averaging

The key insight for efficient implementation is to organize threads so that adjacent probes in a 2×2 neighborhood are processed by adjacent threads within the same SIMD group. This allows direct communication between threads using SIMD shuffle operations, which are faster than shared memory access.

5.1.2 SIMD Shuffle Operations (Metal Shaders)

SIMD shuffle instructions allow threads within the same SIMD group to directly read register values from other threads in the group. This enables each thread to access the ray results from its neighboring probes without any memory operations, making the averaging process extremely efficient.

5.1.3 Pre-Averaging Algorithm

The following algorithm demonstrates how SIMD operations can be used to implement the pre-averaging technique:

Algorithm 12 Pre-Averaging Ray Tracing with SIMD Groups

```
 1: procedure RayTracingWithPreAveraging(radianceTexture, cascadeData)
 2:     quadIndex, localProbeIndex, rayIndex ← MapThreadID(threadID)
 3:     probePos ← CalculateProbePosition(quadIndex, localProbeIndex)
 4:     probeUV ← ConvertToScreenSpace(probePos)
 5:     worldPos ← ReconstructWorldPosition(probeUV)
 6:     rayUV ← CalculateRayUV(rayIndex, raysPerDim)
 7:     rayDir ← OctDecode(rayUV)
 8:     hitResult ← TraceRay(worldPos, rayDir, cascadeInterval)
 9:     radiance, occlusionFactor ← ProcessHitResult(hitResult)
10:     // SIMD group averaging - each group handles a 2x2 probe neighborhood
11:     baseLane ← simdLaneID - localProbeIndex            ▷ Get first lane in group
12:     // Gather results from all 4 probes in the group
13:     r0 ← simdShuffle(radiance, baseLane)                ▷ Probe 0 result
14:     r1 ← simdShuffle(radiance, baseLane + 1)            ▷ Probe 1 result
15:     r2 ← simdShuffle(radiance, baseLane + 2)            ▷ Probe 2 result
16:     r3 ← simdShuffle(radiance, baseLane + 3)            ▷ Probe 3 result
17:     averagedRadiance ← (r0 + r1 + r2 + r3) * 0.25
18:     o0 ← simdShuffle(occlusionFactor, baseLane)
19:     o1 ← simdShuffle(occlusionFactor, baseLane + 1)
20:     o2 ← simdShuffle(occlusionFactor, baseLane + 2)
21:     o3 ← simdShuffle(occlusionFactor, baseLane + 3)
22:     averagedOcclusion ← (o0 + o1 + o2 + o3) * 0.25
23:     // Only the first probe in each group writes to the output
24:     if localProbeIndex == 0 then
25:         finalRadiance ← MergeWithUpperCascade(averagedRadiance, averagedOcclusion)
26:         outputPos ← CalculateOutputPosition(quadIndex, rayIndex)
27:         WriteToTexture(radianceTexture, outputPos, finalRadiance)
28:     end if
29: end procedure
```

5.1.4 Pre Averaging Result

In Figure 5.2 an early implementation of SPWI can be seen, before depth aware upscaling (hence the halos around the emissive surfaces). The current grid layout, now matches the one that Alexander proposed in his implementation, which is four times more that what it was used in the final solution of this thesis. The hornbugs in Figure 5.2 are having a more accurate representation of the radiance thanks to the finer probe grids. However, tracing four times more rays on the current system of the implementation was not viable, and the performance optimization gained from the SIMD subgroups implementation proved negligible as the real performance hit is the ray tracing. This suggested that other optimizations, such as the depth-aware techniques implemented later, will provide better quality-to-performance ratios for global illumination.

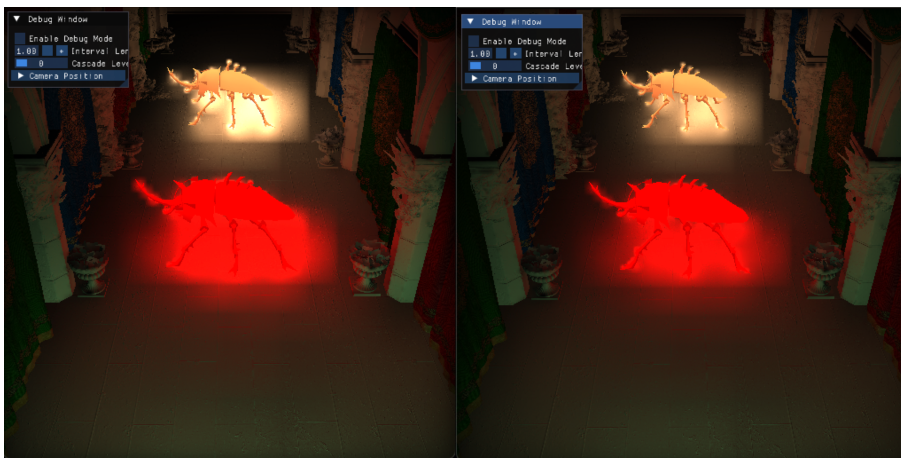


Figure 5.2: Left is regular probe grids. Right is twice as many probes on each direction on every cascade using pre-averaging.

6
Results

In this section, various scenes rendered with the SPWI implementation are presented to demonstrate its effectiveness across different lighting conditions and geometric complexity. Figures 6.1 through 6.7 showcase these results, highlighting the algorithm's ability to capture soft shadows and indirect illumination. Table 6.1 provides performance metrics for each scene, including frame rendering time and memory consumption. All scenes are rendered with 6 cascades, where each cascade has a ray per pixel, which would be equivalent to 6 samples per pixel (spp) in traditional ray tracing. A key advantage of this approach is that performance remains consistent regardless of the number of light sources in the scene, as the algorithm traces fixed rays per cascade level independent of lighting complexity. However, performance is primarily bounded by acceleration structure traversal during ray tracing, with more complex geometry typically resulting in longer trace times. The memory footprint scales primarily with screen resolution and cascade count, making the technique suitable for both small and large environments.

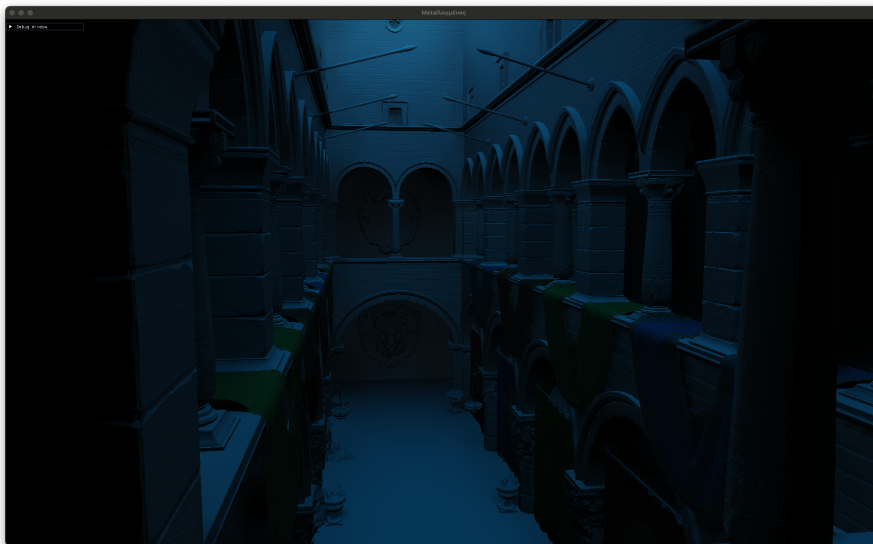


Figure 6.1: The Sponza model rendered only with sky illumination.

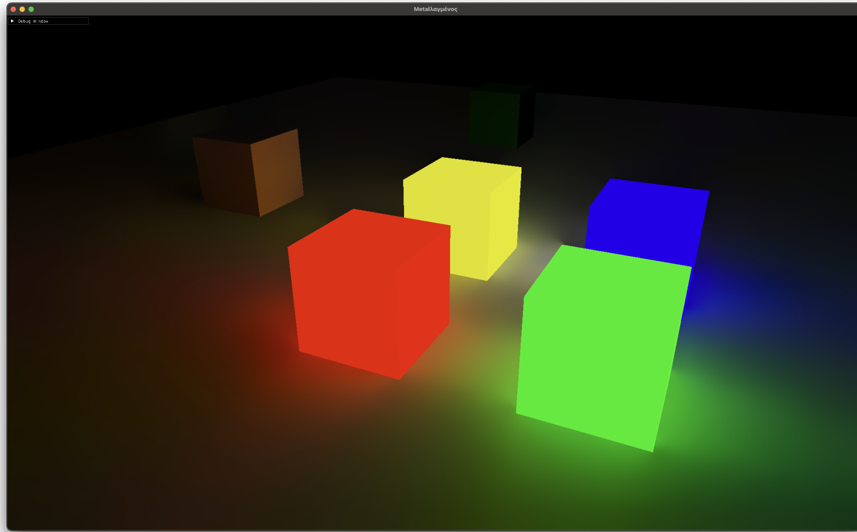


Figure 6.2: Scene with four emissive cubes of different colors and two non emissive cubes.

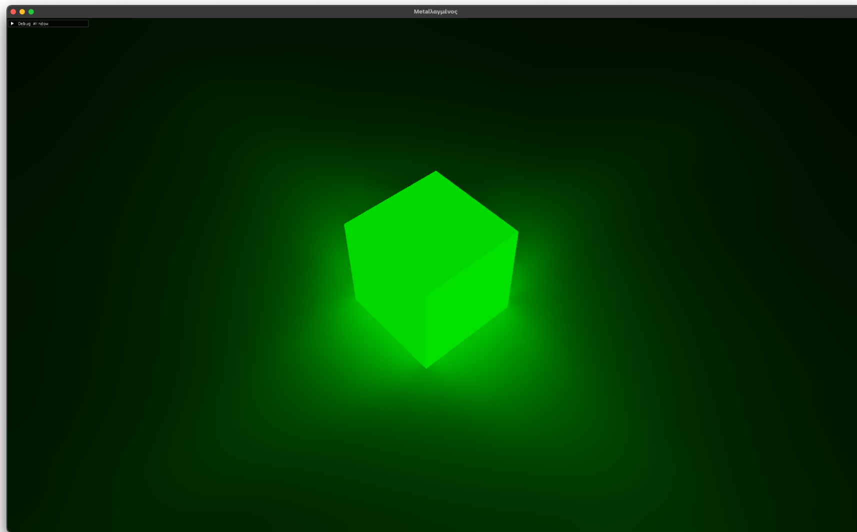


Figure 6.3: A single emissive cube.

Figure 6.4: The Sponza model with 3 emissive hornbug models by McGuire [10] showcasing off-screen indirect light coming from a pink, blue and a yellow hornbug.

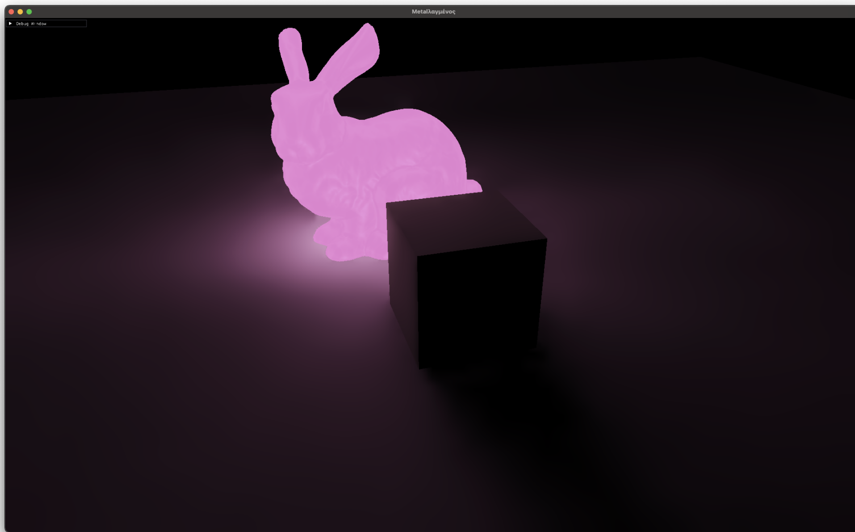


Figure 6.5: The Stanford bunny by Turk and Levoy [9] with a matte cube showcasing soft shadows.

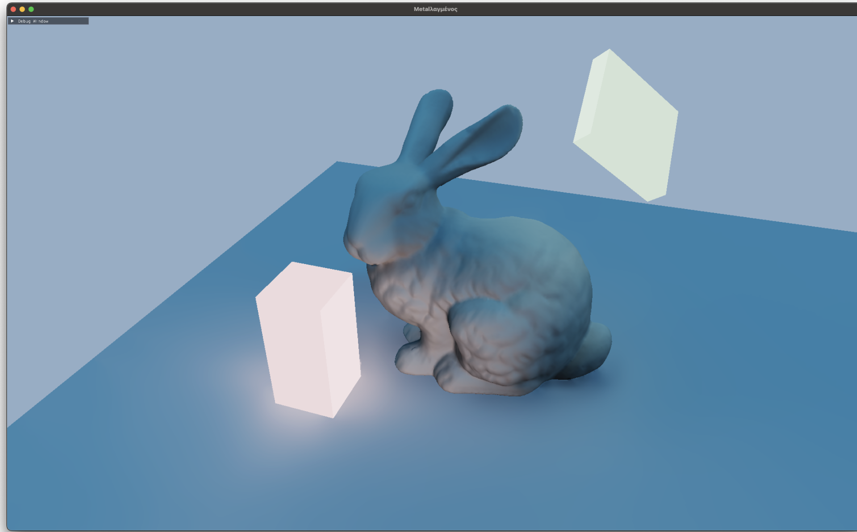


Figure 6.6: The Stanford bunny with 2 emissive cuboids with daylight.

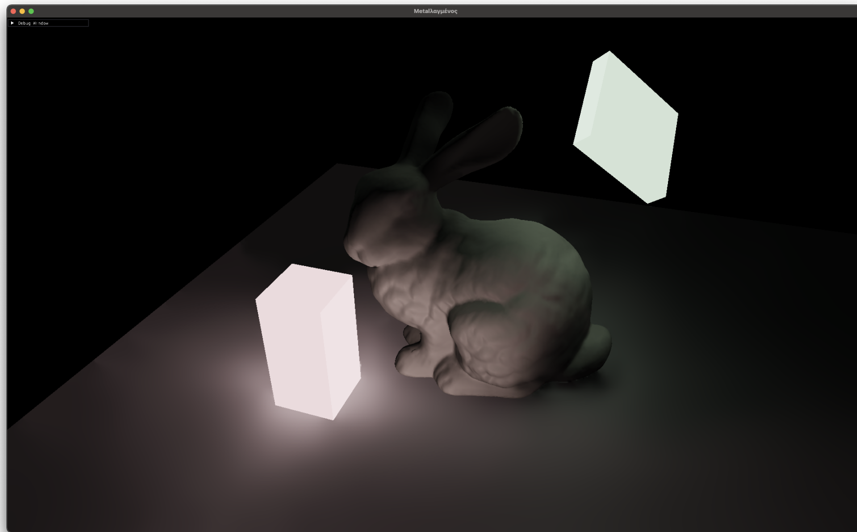


Figure 6.7: The Stanford bunny with 2 emissive cuboids with no daylight.

For comparison, in the original paper [1] Alexander mentions that his implementation of SPWI with world space ray marching on a scene represented as an analytical SDF, each frame takes about 30-50ms to calculate on a RTX3060.

Table 6.1: Performance metrics for various scenes rendered with SPWI on a M1 MacBook Air with 8GB of unified memory. All measurements were taken at 1920×1080 resolution with 6 cascades. The memory shown in the table includes all the memory needed for the frame (acceleration structures, models, buffers and textures).

Scene	**Rendering Time (ms)**	**Memory (MiB)**
Cube Scene (Fig. 6.3)	32.77	147.00
Cubes Scene (Fig. 6.2)	34.42	146.88
Bunny Cube Scene (Fig. 6.5)	37.07	184.66
Bunny Cuboids Scene (Fig. 6.6)	39.87	184.90
Sponza Sky Illuminated (Fig. 6.1)	143.74	405.00
Sponza with Emissive Hornbugs (Fig. 6.4)	155.90	408.50

7
Discussion and Future Work

As mentioned in the chapter 4, there is a better approach to merge with the upper cascade known as Min Max Probes. The name comes from the use of a min max depth buffer, which is a specialized depth texture that stores both the minimum and maximum depth values for regions of the scene. This technique can significantly improve the quality of cascade merging by providing more accurate depth information for bilateral filtering and bilinear 3D interpolation.

7.1 Min Max Depth Buffer

A min max depth buffer stores two depth values per texel - the minimum and maximum depths within a region. This provides more comprehensive depth information than a standard depth buffer, allowing for better handling of depth discontinuities and complex geometric boundaries.

The min max buffer is generated through a two-step process:

1. Initialization: Copy the original depth values to both the minimum and maximum channels.
2. Hierarchical reduction: Generate increasingly coarser versions by finding the minimum and maximum values in 2×2 regions.

The resulting texture pyramid contains depth bounds for regions of various sizes, which can be used to more accurately determine geometric continuity between probes.

Algorithm 13 Min Max Depth Buffer Generation

```
 1: procedure InitializeMinMaxDepth(depthTexture, minMaxTexture)
 2:     if IsValidCoordinate(gid, minMaxTexture) then
 3:         depth ← depthTexture.Read(gid)
 4:         // Write the same depth value to both min and max channels
 5:         minMaxTexture.Write(depth, gid, 0)          ▷ Red channel for minimum
 6:         minMaxTexture.Write(depth, gid, 1)        ▷ Green channel for maximum
 7:     end if
 8: end procedure
 9: procedure GenerateMinMaxMipLevel(srcTexture, dstTexture)
10:     dstSize ← GetTextureSize(dstTexture)
11:     if IsValidCoordinate(gid, dstSize) then
12:         srcCoord ← gid × 2
13:         depth00 ← srcTexture.Read(srcCoord)
14:         depth10 ← srcTexture.Read(srcCoord + (1, 0))
15:         depth01 ← srcTexture.Read(srcCoord + (0, 1))
16:         depth11 ← srcTexture.Read(srcCoord + (1, 1))
17:         // Find new min and max across the 2 × 2 block
18:         newMin ← Min(depth00.r, depth10.r, depth01.r, depth11.r)
19:         newMax ← Max(depth00.g, depth10.g, depth01.g, depth11.g)
20:         dstTexture.Write((newMin, newMax, 0, 0), gid)
21:     end if
22: end procedure
```

In this algorithm, the min-max texture is first initialized by writing the same depth value to both the red and green channels (for minimum and maximum, respectively). This initialization is necessary because subsequent mip levels will compare these channel values separately. The second procedure generates each successive mip level by examining 2×2 blocks from the previous level, finding the minimum of all minimum values and the maximum of all maximum values. This hierarchical approach efficiently captures depth boundaries at multiple scales.

Figure 7.1: Min Max depth buffer mip hierarchy. The green highlights indicate regions with significant depth variation. Moving to higher mip levels (coarser resolutions), these highlights effectively mark depth discontinuities at increasingly larger scales.

The Figure 7.1 shows the complete mip chain of the min-max depth buffer. In this visualization, regions where minimum and maximum depth values are similar appear in shades of yellow due to the similar red and green channel values, while areas with significant depth differences are highlighted in green.

In the context of radiance cascades, the min-max information can be leveraged by placing two probes instead of one, positioning them on the minimum and maximum depth planes respectively. For each cascade level, the mip level that precisely matches the resolution of the cascade's grid is selected. Then, rays are traced from both sets of probes, merge radiance for both planes separately, and perform trilinear interpolation between the current probe and the min/max probes of the upper cascade.

This approach would impact performance due to the doubled ray tracing workload. One potential solution would be to split the number of rays in half to maintain similar performance characteristics. Since the number of rays per probe is divisible by two, odd-numbered rays could be assigned to the minimum depth plane while even-numbered rays are assigned to the maximum depth plane, in order to keep uniformity on both probes.

8
Conclusion

This thesis has explored the implementation and optimization of SPWI using radiance cascades. The investigation has shown that radiance cascades provide an effective approach to single-shot diffuse global illumination, efficiently capturing lighting effects across multiple distance scales.

The implementation successfully captures soft shadows and indirect illumination. Performance scales primarily with scene geometry complexity rather than light count, making the technique suitable for scenes with numerous light sources. Moreover, the length of the intervals need to be adjusted accordingly with the scale of a world.

However, significant challenges remain. The screen-space placement of probes introduces temporal instability during camera movement, creating flickering artifacts. Although the depth-aware interpolation techniques mitigate some of these issues, the proposed min-max depth buffer approach shows greater promise for improving both visual quality and temporal stability.

Performance remains a concern, with complex scenes requiring substantial computational resources for the multiple ray tracing passes. This limits the technique's applicability for real-time applications on mainstream hardware.

Future work should focus on improving temporal stability through better probe placement strategies. Despite its current limitations, SPWI with radiance cascades represents a valuable middle ground between screen-space techniques and full path tracing, with potential for wider adoption as hardware capabilities continue to advance.

9
Ethical Considerations

From an environmental perspective, the computational efficiency of rendering algorithms directly impacts energy consumption across the graphics industry. While the current implementation requires substantial computational resources for complex scenes, it represents a more efficient alternative to full path tracing solutions for offline rendering. As the technique matures and hardware capabilities advance, improved efficiency could contribute to reduced energy consumption film production, and architectural visualization workflows where such algorithms are deployed at scale.

Finally, the research methodology employed in this work emphasizes honest presentation of limitations, including temporal instability issues and performance constraints. This approach supports the broader research community's ability to build upon and improve existing techniques rather than pursuing unrealistic expectations based on incomplete reporting.

Bibliography

[1] Sannikov, A. (2023). Radiance Cascades. Unpublished manuscript. Retrieved from https://github.com/Raikiri/RadianceCascadesPaper

[2] Coppen, M. (2024). The Fundamental Observation of Radiance Cascades. Retrieved from https://m4xc.dev/articles/fundamental-rc/

[3] MGCHEE, J. (2024). Building a Real-Time Global Illumination. Retrieved from https://jason.today/gi

[4] Cigolle, Z. H., Donow, S., Evangelakos, D., Mara, M., McGuire, M., & Meyer, Q. (2014). A Survey of Efficient Representations for Independent Unit Vectors. Journal of Computer Graphics Techniques (JCGT), 3(2), 1-30. Retrieved from http://jcgt.org/published/0003/02/01/

[5] Gjoel, M. (2021). Depth Aware Upscaling. Retrieved from https://gist.github.com/pixelmager/a4364ea18305ed5ca707d89ddc5f8743

[6] Sannikov, A. (2024). Novel Depth Aware Upscaling. Retrieved from https://www.shadertoy.com/view/4XXSWS

[7] Alex and Xor (2024). GM Shaders Guest: Radiance Cascades 2. Retrieved from https://mini.gmshaders.com/p/radiance-cascades2

[8] Marko Dabrovic (2002). Sponza Atrium. Retrieved from https://casual-effects.com/data/

[9] Greg Turk and Marc Levoy (1994). The Stanford 3D Scanning Repository. Retrieved from http://graphics.stanford.edu/data/3Dscanrep/

[10] Morgan McGuire (2017). Computer Graphics Archive. Retrieved from https://casual-effects.com/data

[11] Kajiya, J. T. (1986). The rendering equation. ACM SIGGRAPH Computer Graphics, 20(4), 143-150.

[12] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., & Ramamoorthi, R. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. ACM Transactions on Graphics, 39(4), 148:1-148:17.

[13] Ouyang, Y., Liu, S., Kettunen, M., Pharr, M., & Pantaleoni, J. (2021). ReSTIR GI: Path resampling for real-time path tracing. Computer Graphics Forum, 40(8), 17-29.

[14] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., & Aila, T. (2017). Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. ACM Transactions on Graphics, 36(4), 98:1-98:12.

[15] Mittring, M. (2007). Finding next gen: CryEngine 2. In ACM SIGGRAPH 2007 Courses (pp. 97-121).

[16] Ritschel, T., Grosch, T., & Seidel, H. P. (2009). Approximating dynamic global illumination in image space. In Proceedings of the 2009 symposium on Interactive 3D graphics and games (pp. 75-82).

[17] Greger, G., Shirley, P., Hubbard, P. M., & Greenberg, D. P. (1998). The irradiance volume. IEEE Computer Graphics and Applications, 18(2), 32-43.

[18] Kaplanyan, A. S., & Dachsbacher, C. (2010). Cascaded light propagation volumes for real-time indirect illumination. In Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (pp. 99-107).

[19] Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., & McGuire, M. (2019). Dynamic diffuse global illumination with ray-traced irradiance fields. Journal of Computer Graphics Techniques, 8(2), 1-30.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY