Master of Science, Computer Animation and Visual Effects

# Adaptive Wavefront Scheduling for Coherent CPU Path Tracing

Felipe Hidalgo Árguedas

Bournemouth University

August 2026

# Contents

# List of figures

# List of tables

# Abstract

Physically based path tracing can produce highly realistic images, but its performance is affected by the divergent workloads created during ray traversal and shading. Previous work has shown that sorting rays using static properties such as materials and textures can improve execution coherence and memory locality. This dissertation investigates whether using runtime information to adapt workload scheduling can provide further performance improvements. A CPU-based wavefront path tracer was developed using OpenUSD and Embree, together with a modular framework supporting unscheduled, material, texture, and Cost-Benefit scheduling.

This dissertation introduces a Cost-Benefit scheduling algorithm that uses information collected during rendering to prioritise shading work according to its expected contribution relative to its execution cost. The scheduling framework includes a None policy, which keeps the original shading order and is used as the baseline, as well as Material and Texture policies that group shading work using scene information. Cost-Benefit uses runtime information to change the order of the work. Each policy was run independently using the same two procedural benchmark scenes and rendering configurations, allowing their shading time, execution coherence, memory locality, and image differences to be compared. Cost-Benefit reduced shading time by up to 10% compared with the baseline. Material scheduling achieved a similar level of coherence, producing long runs of the same material and high cache-line homogeneity, but did not provide the same improvement in rendering time. These results suggest that accounting for the cost of shading work can be more useful than grouping work by material alone. Image comparisons also showed only small differences between the baseline and Cost-Benefit renders, indicating that the change in scheduling had little effect on the final image.

**Keywords:** Physically Based Path Tracing, Wavefront Rendering, Workload Scheduling, Execution Coherence, Memory Locality, Cost-Benefit Scheduling, and Rendering Performance.

# Acknowledgments

I would like to thank Jon Macey, Jian Chang, and Ian Stephenson for all their help, support, and guidance throughout this project. Thank you for sharing your knowledge and experience, as well as the references and advice that helped me throughout this journey.

My thanks also go to my loved ones, friends, and family for all the support I have received while pursuing a dream and this new adventure. Without all of you, I would not have been able to complete this dissertation.

# 1 Introduction

Physically based path tracing has become a widely used method in computer graphics for generating realistic images because it can simulate complex light transport and global illumination. Yet, obtaining this degree of visual accuracy is still very computationally costly. As the rays travel through a scene, they tend to spread out because of their interactions with various surfaces, materials, and lighting conditions. This spreading reduces execution coherence and leads to irregular memory access patterns, as well as inefficient CPU use during shading.

The wavefront path tracing naturally generates shading queues by decoupling the rendering process into independent stages, allowing rays to be reordered before shading. However, these queues are not orchestrated to maximize execution coherence, leading to a significant performance bottleneck in the shading stage due to the unpredictable nature of ray workloads. Previous work has shown that sorting these rays by properties like material and texture can improve coherence. Furthermore, the additional cost of scheduling introduces a trade-off that needs further investigation.

To investigate this problem, a CPU-based wavefront path tracer was developed in C++, using Embree for ray traversal and OpenUSD for scene loading. A modular scheduling framework was then developed to compare static and runtime-aware approaches to organizing shading work. The proposed Cost-Benefit scheduler uses runtime information to adapt the order in which shading work is processed, and its performance was evaluated using procedural benchmark scenes and a set of coherence and rendering metrics.

The rest of this dissertation is structured as follows. Chapter 2 reviews the development of coherence-aware rendering, from memory-coherent ray tracing to wavefront rendering and shading scheduling. Chapter 3 covers the main concepts behind this research, including physically based rendering, Monte Carlo path tracing, variance reduction, and execution coherence. Chapter 4 describes the renderer and adaptive scheduling system, while Chapter 5 covers the evaluation setup and performance metrics. Chapter 6 presents the results, and Chapter 7 discusses the main findings, limitations, and possible directions for future work.

# 2 Related Work

Production scenes are getting larger and more complex, making physically based rendering increasingly expensive. Large amounts of geometry, high-resolution textures, and complex materials can put considerable pressure on CPU caches and memory bandwidth. Because of this, making the ray tracing algorithm faster is only part of the problem. The way rays are sorted and processed can also have a significant effect on rendering performance (Pharr et al., 2023).

Initially, research into coherent ray tracing focused mainly on memory access during ray traversal. Pharr et al. (1997) showed that memory access could become an important part of the rendering cost when working with larger scenes. Their approach reordered rays according to their spatial locality, allowing rays that accessed similar parts of the scene to be processed together. This improved cache utilization and showed that the order in which rays are processed can have a direct effect on rendering performance. Moreover, spatial ordering can be represented using Morton order, which provides a way of arranging multidimensional data while preserving spatial locality (Morton, 1966).

Following this work, Wald et al. (2001) researched coherent ray traversal using ray packets. Rather than tracing every ray on its own, nearby rays were grouped together and processed using Single Instruction, Multiple Data (SIMD) operations. This was particularly useful for primary rays because they usually follow similar paths through the scene. However, as rays interact with different surfaces and materials, their paths start to diverge. Once this happens, the rays in a packet become less similar, reducing the benefit of processing them together.

Because of this limitation, later work began looking at coherence beyond the traversal stage. Eisenacher et al. (2013) introduced Sorted Deferred Shading, where rays are first collected and then sorted according to properties such as geometry, primitives, and shaders before shading takes place. This allowed rays with similar shading requirements to be processed together, improving memory access to materials and textures while reducing changes in shader state. The sorting decisions, though, were still based on information available from the scene and did not take the runtime cost of individual shading operations into account.

Around the same time, Laine et al. (2013) presented a wavefront approach to path tracing in which ray generation, intersection, and shading were separated into individual stages. Work was passed between these stages using queues, rather than processing the entire path in a single operation. This separation made it possible to reorganize rays between stages before they were processed further. Furthermore, it provided a suitable structure for

grouping similar workloads together, which made wavefront rendering particularly relevant to later scheduling and sorting techniques.

Building on these ideas, Bainbridge (2015) investigated the effect of sorting shading work according to material and texture properties. The results showed that organizing similar shading workloads together could improve cache behavior, memory access patterns, and overall rendering performance. The work also demonstrated that shading coherence could be improved within a physically based renderer without changing the underlying rendering approach. This provides the starting point for the scheduling work presented in this dissertation.

While these approaches showed that sorting can improve coherence, the information used to order the work is largely determined before execution. Eisenacher et al. (2013) demonstrated the benefits of grouping shading work according to properties available from the scene, but this does not account for differences in the cost of individual shading operations. The sorting process itself also introduces additional work, creating a trade-off between the cost of reordering the workload and the performance gained from doing so. Laine et al. (2013) showed how separating rendering into stages connected by queues allows work to be reordered between stages before it is processed further. This creates an opportunity to use information collected during rendering when deciding how the workload should be processed, rather than relying only on properties known before execution.

Overall, the research shows a gradual shift from improving coherence during ray traversal towards organizing workloads during shading. Earlier approaches used spatial information, while later methods considered properties such as materials and textures. However, these properties are generally known before the work is executed and do not describe how expensive a particular shading workload will be at runtime. This raises the question of whether runtime information can be used alongside these existing properties to make better scheduling decisions. This dissertation investigates this approach through a set of scheduling policies implemented within a CPU-based wavefront renderer.

# 3 Theoretical Background

The previous chapter introduced the development of coherence-aware rendering and the role of workload organization in ray tracing. This chapter covers the rendering concepts needed to understand the implementation and evaluation of the proposed scheduling framework.

## 3.1 Physically Based Rendering

Physically based rendering (PBR) uses the behavior of light to produce realistic images. At the center of PBR is the rendering equation introduced by Kajiya (1986). It describes the light leaving a surface as the sum of light emitted by the surface and light scattered toward it from the rest of the scene. The reflected light comes from different directions over the surface hemisphere and depends on the BRDF and the angle between the incoming light and the surface normal (Pharr et al., 2023).

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} L(x \leftarrow \Psi) f_r(x, \Psi \to \Theta) \cos(N_x, \Psi) \, d\omega_\Psi \tag{3.1}$$

In Equation 3.1, $L(x \to \Theta)$ represents the outgoing radiance in the viewing direction, while $L_e(x \to \Theta)$ represents the emitted radiance. The integral accounts for incoming radiance $L(x \leftarrow \Psi)$, with the BRDF $f_r(x, \Psi \to \Theta)$ which describes how the surface reflects that light. The cosine term, $cos(N_x, \Psi)$ accounts for the orientation of the surface relative to the incoming direction.

The rendering equation is difficult to solve directly since the light reaching a surface can come from many other surfaces in the scene. For this reason, the renderer uses Monte Carlo path tracing to estimate the result by sampling light paths through the scene (Pharr et al., 2023).

### 3.1.1 Monte Carlo Path Tracing

Monte Carlo path tracing is one way of solving the rendering equation by sampling light paths through the scene (Pharr et al., 2023). Kajiya (1986) showed how these random samples could be used to estimate the light reaching the camera. More samples usually give a better result, although the difference between successive samples becomes smaller as the sample count increases (Pharr et al., 2023).

The rendering equation can be approximated using the Monte Carlo estimator shown in Equation 3.2 (Pharr et al., 2023).

$$L_o(x, \omega_o) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \cos\theta_i}{p(\omega_i)} \tag{3.2}$$

In Equation 3.2, $L_o(x, \omega_o)$ represents the outgoing radiance, while each sample estimates the contribution from an incoming direction $\omega_i$. The BRDF $f_r$ describes how the surface reflects the incoming light, and the cosine term accounts for the angle between the light direction and the surface normal. The probability density $p(\omega_i)$ represents how likely each direction is to be sampled. Dividing the contribution by this probability keeps the estimator unbiased (Pharr et al., 2023).

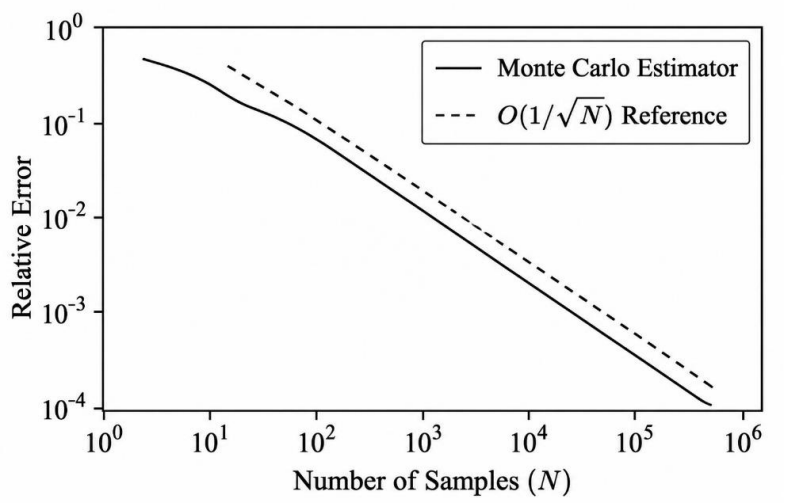



**Figure 3. 1** Monte Carlo convergence as the number of sample increases.

Figure 3.1 demonstrates how Monte Carlo convergence decreases as the number of samples gets higher. The error decreases approximately according to $O(1/\sqrt{N})$, meaning that additional samples provide progressively smaller improvements (Dutré et al., 2006). This is why variance reduction is important: increasing the sample count alone becomes increasingly expensive for relatively small improvements in image quality.

## 3.2 Variance Reduction

Although Monte Carlo path tracing can converge to the correct solution, a large number of samples may still be needed to get a clean image (Pharr et al., 2023). The renderer therefore uses Next Event Estimation, Multiple Importance Sampling, and Russian Roulette to reduce unnecessary sampling. These techniques are described in the following sections.

### 3.2.1 Next Event Estimation (NEE)

Next Event Estimation (NEE) samples the direct lighting at each surface interaction. A sample is taken from a light source, and a ray is traced towards it to check whether the light is visible. The contribution from the light is then added to the path, while the BRDF is used to continue tracing the ray (Dutré et al., 2006). This reduces the variance caused by relying only on randomly generated paths to find the light source (Pharr et al., 2023).

$$L_r = L_{direct} + L_{indirect} \quad (3.3)$$

In Equation 3.3, $L_r$ represents the total reflected radiance leaving a surface. It consists of $L_{direct}$, from direct illumination, and $L_{indirect}$, from light that has undergone one or more reflections. NEE evaluates the direct component explicitly, while the indirect component continues to be estimated using Monte Carlo path tracing (Dutré et al., 2006).

### 3.2.2 Multiple Importance Sampling (MIS)

Multiple Importance Sampling (MIS) significantly reduces variance by combining multiple strategies into the resulting estimator (Veach, 1998). In Physically Based Rendering (PBR) this usually means a combination of light sampling and BRDF sampling. Instead of a single strategy, MIS assigns a weight to every sample based on its probability density, allowing both strategies to contribute in their most effective capacities (Pharr et al., 2023).

$$w_i(x) = \frac{n_i p_i(x)}{\sum_{j=1}^{m} n_j p_j(x)} \quad (3.4)$$

In Equation 3.4, $w_i(x)$ is the weight assigned to sampling method $i$, $n_i$ is the number of samples generated by it, and $p_i(x)$ is its probability density. The balance heuristic uses these values to determine the contribution of each method (Veach, 1998).

When one sample is taken from each method, the balance heuristic can be simplified to Equation 3.5:

$$w_i = \frac{p_i}{p_i + p_j} \quad (3.5)$$

Here, the weight depends only on the probability densities of the two samples. In the renderer, this is used when combining a light sample from NEE with a sample generated from the BRDF (Pharr et al., 2023).

### 3.2.3 Russian Roulette

Russian Roulette cuts down the cost of Monte Carlo path tracing by stopping paths that are unlikely to contribute much to the final image (Pharr et al., 2023). A random test is used to continue or terminate a path with a given probability instead of exploring all paths up to a given depth. The throughput of the surviving paths is multiplied by a compensation factor. This keeps the estimator unbiased.

$$\beta' = \frac{\beta}{1 - q} \quad (3.6)$$

Equation 3.6 shows how the path throughput is changed when Russian Roulette is applied. Here, $\beta$ is the throughput before the test and $q$ is the probability of terminating the path. For a path that continues, the new throughput $\beta'$ is divided by $1 - q$ to account for the probability that the path could have been terminated (Pharr et al., 2023).

### 3.2.4 Adaptive Sampling

Adaptive sampling avoids spending the same number of samples on every pixel. Pixels with lower estimated variation can stop receiving samples, while pixels with higher variation continue to be sampled. The renderer estimates the variance incrementally using Welford's online method (Welford, 1962) and uses the resulting error to determine when a pixel has converged. This type of error-based stopping criterion can reduce unnecessary samples while maintaining the quality of the rendered image (Dammertz et al., 2010).

## 3.3 Wavefront Rendering

Traditional path tracers often use a mega kernel approach, where ray generation, traversal, shading, and secondary ray generation are handled within a single program. As rays follow different paths through the scene, they can require different operations and cause execution divergence (Laine et al., 2013). Wavefront rendering separates these operations into individual stages connected by work queues. Rays are placed into the queue for the next operation and processed together, allowing each stage to focus on one type of work (Laine et al., 2013).

The use of queues also creates points where the order of the work can be changed before it reaches the next stage. This was explored by Eisenacher et al. (2013) through deferred shading, and later by Bainbridge (2015), who extended this approach by sorting rays for traversal coherence and sorting hit points by object and primitive to improve shading and

texture coherence. In this work, the same general idea of reordering queued work is extended by using runtime information to decide how shading work should be ordered.

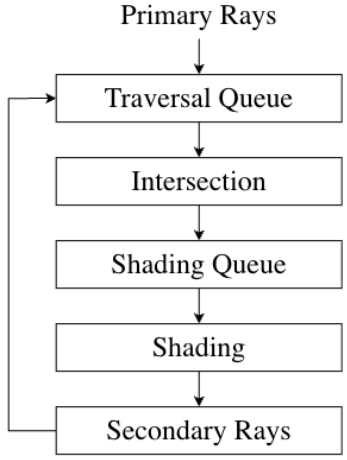



**Figure 3. 2** Wavefront rendering architecture.

Figure 3.2 shows the wavefront pipeline, where rays are moved between queues depending on what they need to do next. Secondary rays are added back to the traversal queue, so similar work can be processed together. The queues also provide a point where rays can be reordered before the next stage.

## 3.4 Execution Coherence and Adaptive Scheduling

Execution coherence describes how similar rays behave while they are being processed. Rays that use the same shaders, access similar data, or follow similar execution paths are more likely to make efficient use of the processor and memory system. When the rays diverge, the renderer must deal with different shaders and memory accesses, which can reduce this efficiency. Bainbridge (2015) showed that sorting rays can improve coherence and rendering performance in CPU-based path tracing.

Adaptive scheduling takes this idea further by changing the order of the work according to information about the current workload. In this dissertation, different scheduling policies are used to test whether runtime information can provide an advantage over sorting rays using fixed properties such as materials and textures.

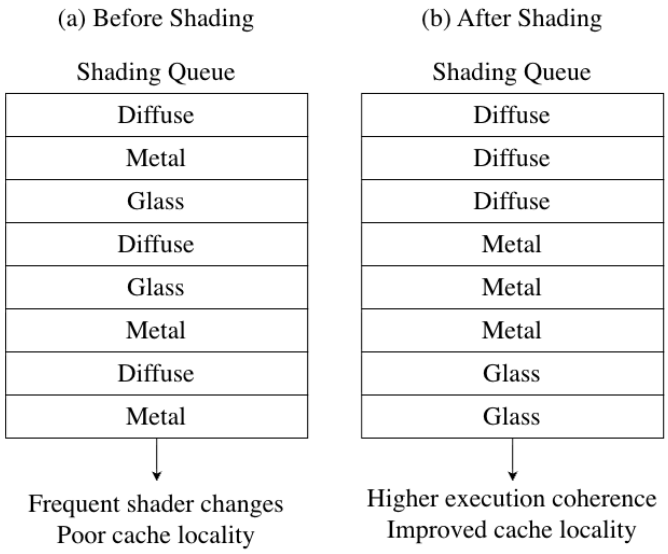



**Figure 3. 3** Comparison of an unsorted shading queue and a material-aware sorted shading queue.

In Figure 3.3 (a), the shading queue contains rays from different materials, so the renderer switches between shaders as it works through the queue. After sorting, shown in (b), rays with the same material are kept together, reducing these changes and giving a more consistent access pattern.

# 4 Design and Implementation

## 4.1 Renderer Architecture

The renderer developed for this work is a CPU-based wavefront path tracer written in C++17. It combines OpenUSD for scene representation and loading (Pixar Animation Studios, 2016), Embree for ray traversal, physically based shading, and the adaptive scheduling framework described later in this chapter. The overall architecture is shown in Figure 4.1.

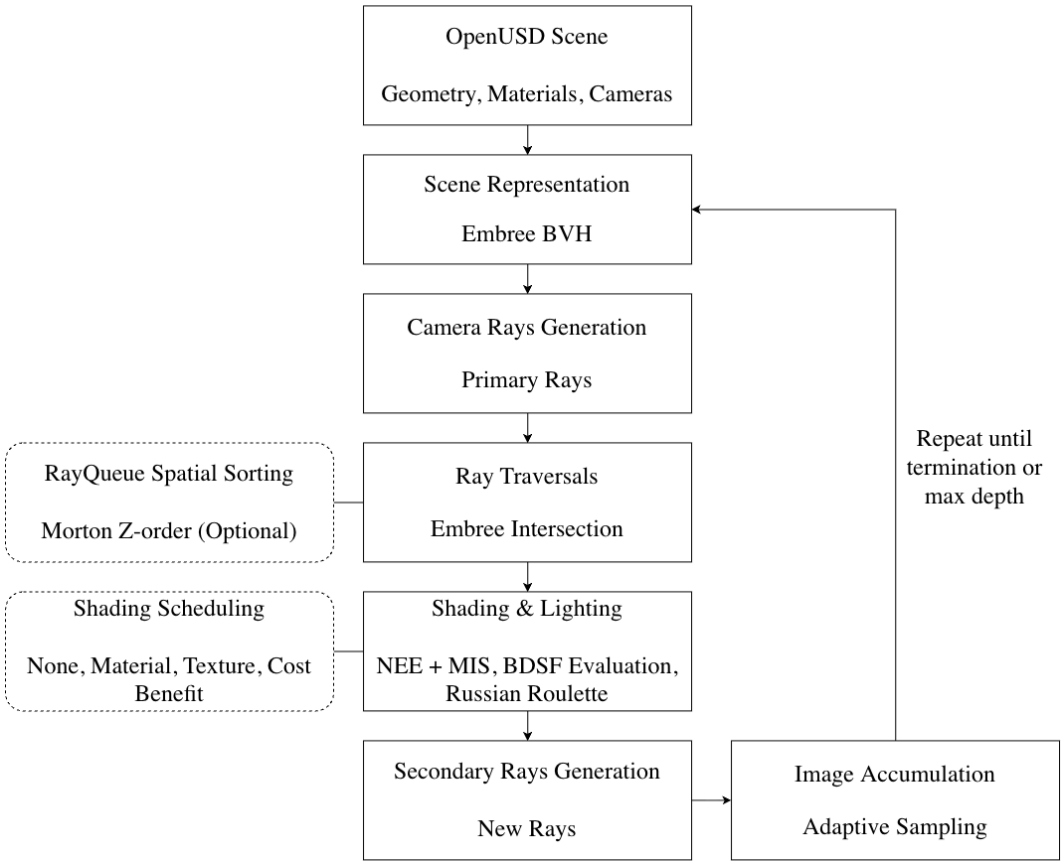



**Figure 4. 1** Architecture of the proposed wavefront renderer.

A scene is first loaded from OpenUSD and converted into internal geometry, material, texture, and camera representations used by the renderer. Embree then builds the acceleration structures needed for ray traversal. During rendering, rays move between the

traversal and shading stages, with new secondary rays being placed back into the pipeline until the required samples have been completed.
The following sections describe the implementation of the rendering pipeline, and the scheduling framework developed for this work.

## 4.2 Renderer Implementation

The renderer uses a queue-based execution model to process rays through the traversal and shading stages. Primary rays are generated from the camera and sent to Embree for intersection before being passed to the shading stage. Secondary rays are then added back to the traversal queue.

### 4.2.1 Wavefront architecture

The renderer is split into three main stages: ray generation, traversal, and shading. Figure 4.2 shows how rays move between these stages. The camera creates the primary rays and stores them in the RayQueue, where they are sent through the BVH. Rays that intersect the scene are then added to the ShadingQueue for material evaluation, direct lighting, and secondary ray generation. The new rays go back into the RayQueue, and the same process is repeated until the path ends or reaches the maximum depth.

Separating traversal and shading into different queues also provides the points where the scheduling policies can reorder the work. This allows the policies to be tested without changing the rest of the rendering pipeline.

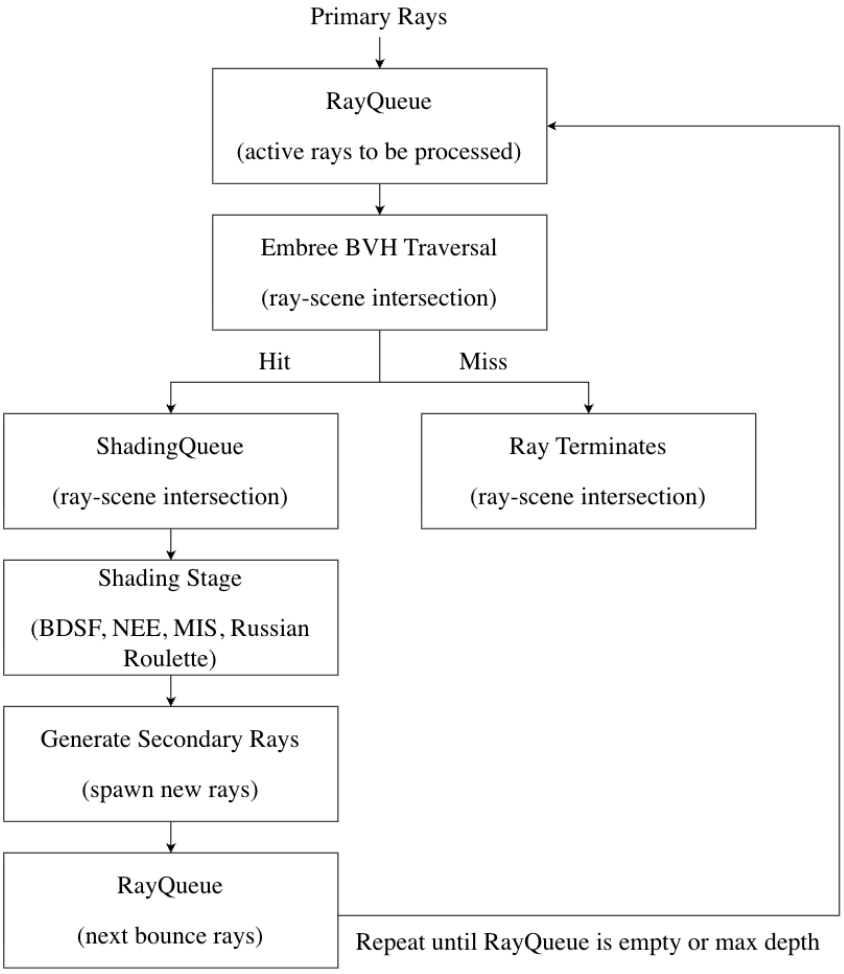



**Figure 4. 2** Execution flow of the wavefront rendering architecture.

### 4.2.2 RayQueue and ShadingQueue

The renderer uses two Structure of Arrays (SoA) work queues to move data between traversal and shading, as shown in Figure 4.3. The RayQueue contains the active rays waiting for BVH traversal. It stores their origins, directions, throughput, pixel coordinates, and depth. When a ray hits an object, the intersection information is added to the ShadingQueue, which stores the position, normal, material and texture IDs, and the path information needed for shading. Keeping these values in separate arrays means that the data used during traversal is stored contiguously in memory (Wald et al., 2014).

The ShadingQueue is also the queue used by the scheduling system. The scheduling policies first create an ordering for the shading records, while the actual data remains unchanged. The queue is reordered when the selected policy is applied before shading. The scheduling process is described in Section 4.3.2.

(a) Array of Structures

```
struct Pt {
   float x;
   float y;
   float z;
}

struct Pt myPts[N];
```

(b) Structure of Arrays

```
struct Pt {
   float x[N];
   float y[N];
   float z[N];
};

struct Pt myPts;
```

**Figure 4. 3** Structure of Arrays (SoA) organization of the renderer work queues.

### 4.2.3 Rendering Pipeline

The complete rendering pipeline is shown in Figure 4.4. During initialization, the Scene class imports geometry, materials, textures, cameras, and lighting from the OpenUSD stage and creates the required Embree acceleration structures. The WavefrontRenderer then generates the primary rays and moves work between the RayQueue and ShadingQueue during traversal and shading. The shading stage supports diffuse, metallic, dielectric, and emissive materials. Dielectric and plastic materials use Schlick's approximation for the reflectance term (Schlick, 1994). Direct lighting is calculated using Next Event Estimation (NEE) together with BSDF sampling through Multiple Importance Sampling (MIS), while indirect lighting uses cosine-weighted hemisphere sampling. Adaptive sampling, Russian Roulette, and firefly clamping are also part of the rendering pipeline used for the scheduling experiments.

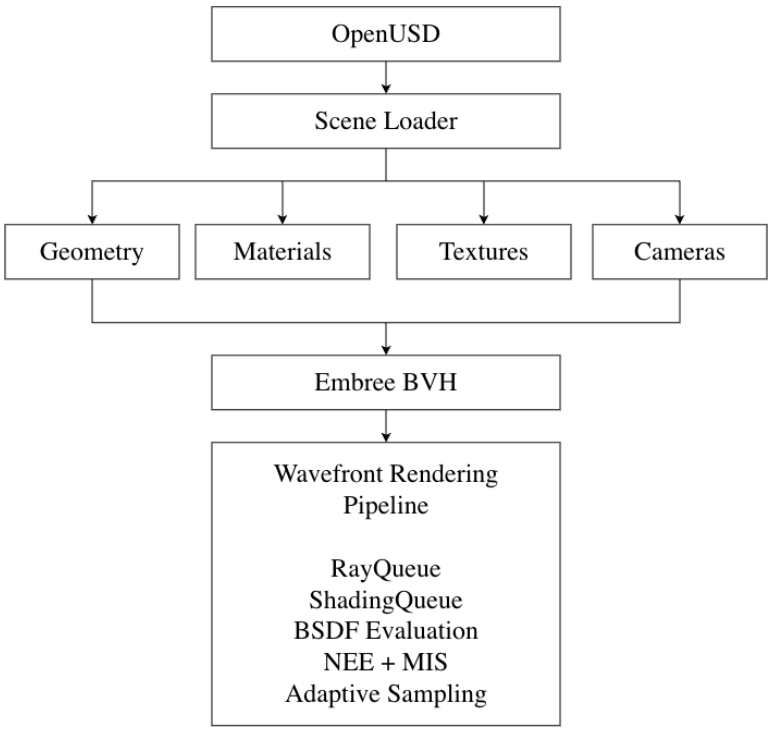



**Figure 4. 4** Implementation of the renderer pipeline from OpenUSD scene loading to wavefront rendering.

## 4.3 Adaptive Scheduling

The wavefront queues provide two points where the workload can be reordered before it is processed. Rays in the RayQueue can be reordered using optional Morton sorting before traversal, while records in the ShadingQueue can be reordered using the shading scheduling policies before BSDF evaluation. The same scheduling interface is used for each policy, allowing different ways of organizing the work without changing the rest of the renderer, as shown in Figure 4.5.

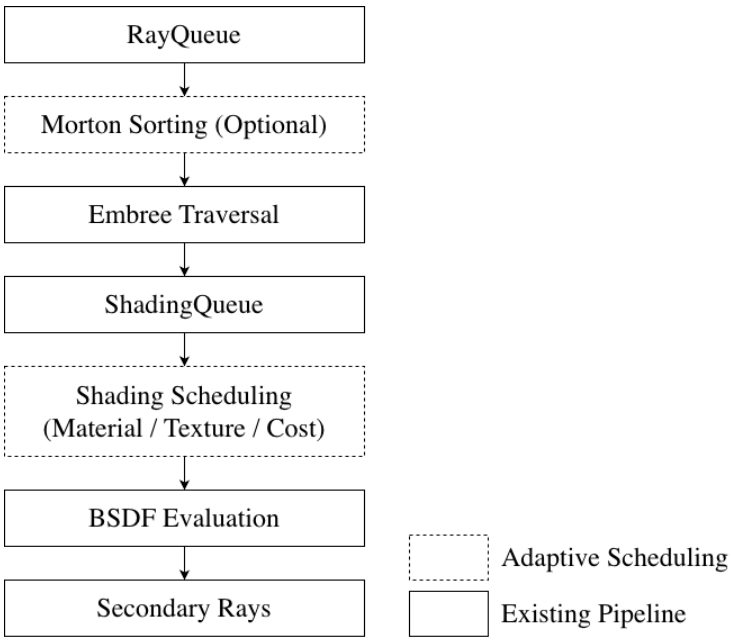



**Figure 4. 5** Overview of the wavefront rendering pipeline highlighting the adaptive scheduling points.

The renderer supports traversal, Material, Texture, and Cost-Benefit scheduling. Cost-Benefit scheduling also uses the runtime cost tracking and cost-aware path termination described in the following sections.

**Table 4. 1** Adaptive scheduling components implemented in the proposed renderer.

| **Component** | **Purpose** |
|---|---|
| Traversal Scheduling | Reorders rays using Morton Z-order to improve BVH traversal coherence. |
| Shading Queue Design | Separates logical and physical queue ordering to minimize unnecessary data movement. |
| Material Scheduling | Groups shading records with identical material identifiers. |
| Texture Scheduling | Groups shading records according to texture identifiers to improve texture cache locality. |
| Cost-Benefit Scheduling | Prioritizes shading work according to estimated execution cost. |

| | |
|---|---|
| Cost Tracker | Estimates per-material execution cost using an exponential moving average (EMA). |
| Cost-Aware Russian Roulette | Adapts path termination according to both path throughput and estimated shading cost. |

Table 4.1 gives an overview of the components used in the scheduling system, including the different scheduling policies and the additional components needed by Cost-Benefit scheduling.

### 4.3.1 Traversal Scheduling

Before each BVH traversal pass, the RayQueue is reordered using Morton Z-order. The quantized origin of each ray is used to generate a Morton key, and the resulting keys determine the order of the queue. This places rays with nearby origins next to each other before they are sent to the EmbreeAccelerator.

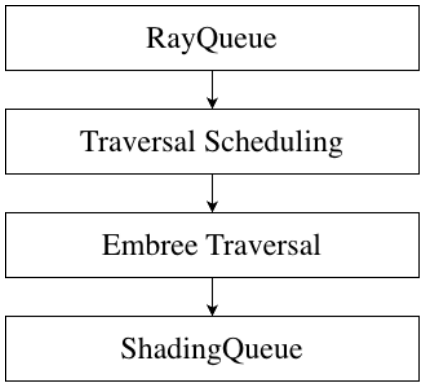



**Figure 4. 6** Integration of traversal scheduling within the wavefront rendering pipeline.

Figure 4.6 shows the position of traversal scheduling within the rendering pipeline, between the RayQueue and BVH traversal.

**Algorithm 4.1.** Traversal scheduling

```
for each active ray in RayQueue do
   Compute Morton key
end for

Sort RayQueue by Morton key

Perform Embree BVH traversal
```

Algorithm 4.1 summarizes the ordering process before each traversal pass. The sorted queue is then passed to the EmbreeAccelerator.

Traversal scheduling was kept enabled for all experiments in Chapter 5. This kept the traversal ordering the same for each scheduling policy, so the performance differences could be attributed to the shading scheduling strategies.

### 4.3.2 Shading Queue Design

After BVH traversal, rays that hit the scene are converted into shading records and added to the ShadingQueue. Scheduling is split into two steps. The records are first reordered through the sortedIndices array without changing the underlying SoA data. The selected ordering is then applied to the queue during materialize(), which reorders the arrays before shading.

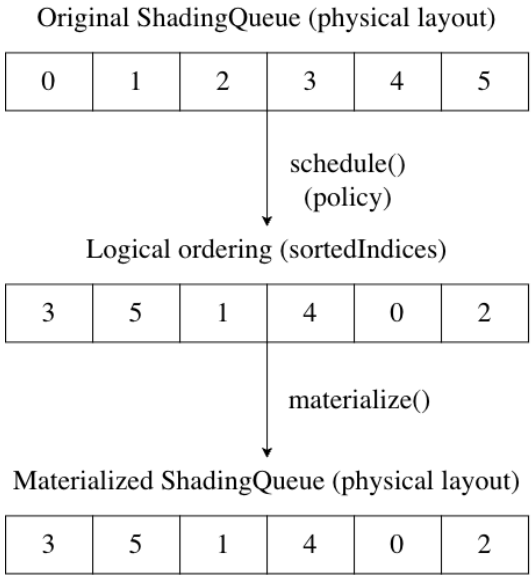



**Figure 4. 7** Logical scheduling and physical materialization within the ShadingQueue.

Figure 4.7 shows this process. Each scheduling policy changes sortedIndices according to its own heuristic, while the shading data stays in its original order. materialize() then uses these indices to reorder the SoA arrays, keeping the records together in memory.

**Algorithm 4.2.** Shading queue scheduling and materialization

```
Initialize sortedIndices = [0 ... N − 1]

Apply selected scheduling policy
   Reorder sortedIndices according to execution heuristic

Materialize ShadingQueue
   Reorder all SoA arrays using sortedIndices

Execute shading
```

Algorithm 4.2 shows the common process used by the scheduling policies. The policy only determines the order of the indices, while materialize() handles the actual data movement. This keeps the scheduling logic separate from the queue implementation.

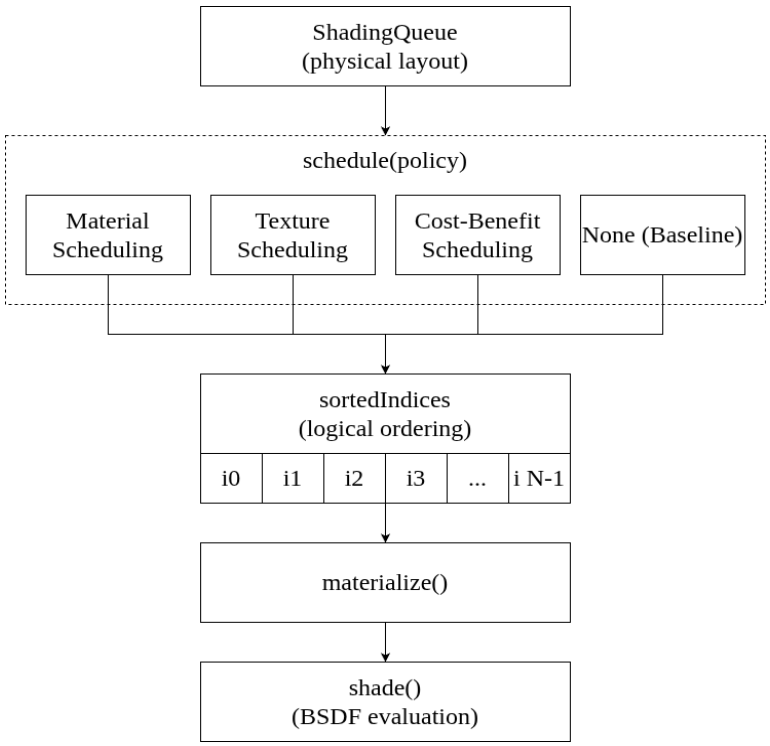



**Figure 4. 8** Scheduling framework implemented within the ShadingQueue.

Figure 4.8 shows how the different policies use the same scheduling interface. Each policy produces its own ordering, which is then applied by materialize() before BSDF evaluation.

### 4.3.3 Scheduling Policies

Building on the scheduling framework presented in the previous section, the renderer uses four scheduling policies to organize shading records in different ways. None preserves the original order and provides the baseline, while Material groups work by material type and use estimated material cost to order records within each group. Texture also groups records by material type but uses texture memory size to determine their order. The proposed Cost-Benefit policy combines material and texture cost estimates with ray throughput to prioritize shading work according to its estimated benefit relative to cost. Table 4.2 summarises the ordering criteria and objectives of each policy.

**Table 4. 2** Comparison of the scheduling policies implemented within the proposed framework.

| **Policy** | **Primary Ordering** | **Secondary Ordering** | **Objective** |
|---|---|---|---|
| None | Original order | - | Baseline |
| Material | Material type | Estimated material cost | Group similar shader work |
| Texture | Material type | Texture memory size | Improve texture locality |
| Cost-Benefit | Material type | Cost-Benefit score | Prioritise work by contribution and cost |

#### 4.3.3.1 None (Baseline)

The None policy leaves the shading records in the order produced by BVH traversal. sortedIndices therefore remains unchanged, and no reordering is performed before shading. This provides the baseline for comparing the other scheduling policies.

**Algorithm 4.3.** Execution of the baseline scheduling policy.

```
Initialize sortedIndices = [0 ... N - 1]

Execute shading
```

Algorithm 4.3 shows the baseline execution. The shading records are processed in the same order in which they were added to the ShadingQueue.

### 4.3.3.1 Material Scheduling

The Material policy organizes the shading records by material type before BSDF evaluation. Each material imported from OpenUSD is assigned an identifier, which is stored with the shading record. The scheduler uses this information to group records by material type, using the estimated material cost to order records within each group.

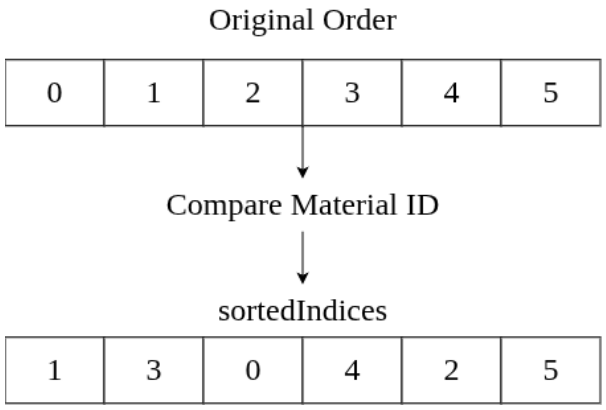



**Figure 4. 9** Logical ordering generated by the Material scheduling policy.

Figure 4.9 shows the ordering produced by the policy. As with the other scheduling strategies, only sortedIndices is changed at this stage. The ShadingQueue itself is reordered later during materialize().

**Algorithm 4.4.** Material scheduling policy

```
Initialize sortedIndices = [0 ... N − 1]

Compare shading records by material ID

Sort sortedIndices

Execute shading
```

Algorithm 4.4 shows the ordering step used by the Material policy before the common materialization stage.

### 4.3.3.2 Texture Scheduling

The Texture policy orders shading records using the material type and the texture associated with each record. Texture IDs are stored in the ShadingQueue and used to access the corresponding texture during scheduling. Records are first grouped by material type, and within each group, the textures are ordered by their memory size.

By grouping records that access the same texture, neighboring shading operations improve memory locality and increase the likelihood of reusing texture data during shading.

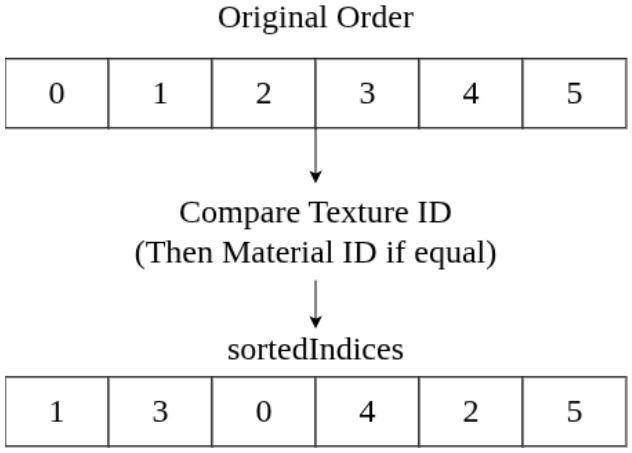



**Figure 4. 10** Logical ordering generated by the Texture scheduling policy.

Figure 4.10 shows the ordering produced by the Texture policy. As with the other policies, the ordering is stored in sortedIndices first, leaving the ShadingQueue unchanged until materialize().

**Algorithm 4.5.** Texture scheduling policy

```
Initialize sortedIndices = [0 ... N − 1]

Compare shading records by texture ID

If texture IDs are equal
   Compare by material ID

Sort sortedIndices

Execute shading
```

Algorithm 4.5 shows the ordering process used by the Texture policy before the common materialization stage.

### 4.3.3.3 Cost-Benefit Scheduling

The Cost-Benefit policy uses information collected during rendering to change the order of the shading work. For each material, the scheduler uses the accumulated throughput of its rays as the expected benefit and combines the estimated material and texture costs to calculate its priority. A higher priority means that the work is expected to provide more contribution for its estimated cost.

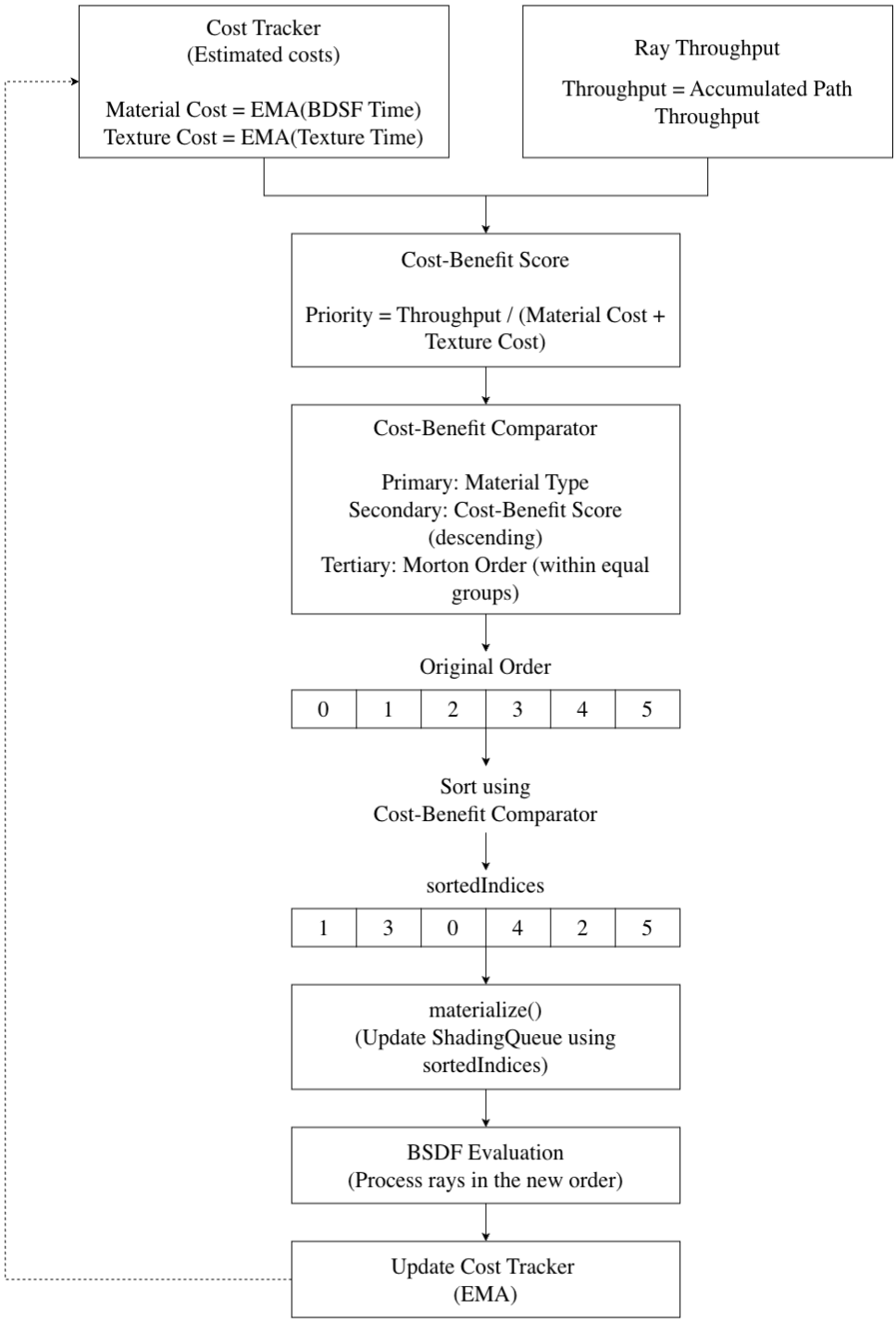



**Figure 4. 11** Execution flow of the Cost-Benefit scheduling policy.

Figure 4.11 shows the order used by the Cost-Benefit policy. Records are first arranged by material type, then by their Cost-Benefit priority. If two records have the same priority, their Morton codes are used to decide their order. This ordering is then applied to the ShadingQueue before shading.

The cost estimates are updated as rendering continues, so the scheduler has more runtime information available for later passes. At the beginning of rendering, the cost estimates are limited, meaning that the ordering has less information to work with. As measurements accumulate, the priority calculation can better reflect the observed cost of the shading work.

**Algorithm 4.6.** Cost-Benefit scheduling policy

```
Initialize sortedIndices = [0 ... N − 1]

Estimate Cost-Benefit score
   Throughput / (Material Cost + Texture Cost)

Sort sortedIndices using comparator
   Primary: Material Type
   Secondary: Cost-Benefit Score
   Tertiary: Morton Order

Materialize ShadingQueue

Execute shading

Update CostTracker
```

Algorithm 4.6 shows the steps used by the Cost-Benefit policy. Unlike the other policies, its ordering depends on measurements collected during the render rather than only on information already available from the scene.

### 4.3.4 Cost Tracker

The Cost Tracker was created to provide the Cost-Benefit scheduler with an estimate of how expensive the shading work is. The renderer keeps separate cost estimates for materials and textures, which are updated with timing measurements collected during rendering. An exponential moving average (EMA) is used to update these values instead of replacing the estimate with every new measurement. The updated values are then used by the Cost-Benefit scheduler when the queue is sorted again.

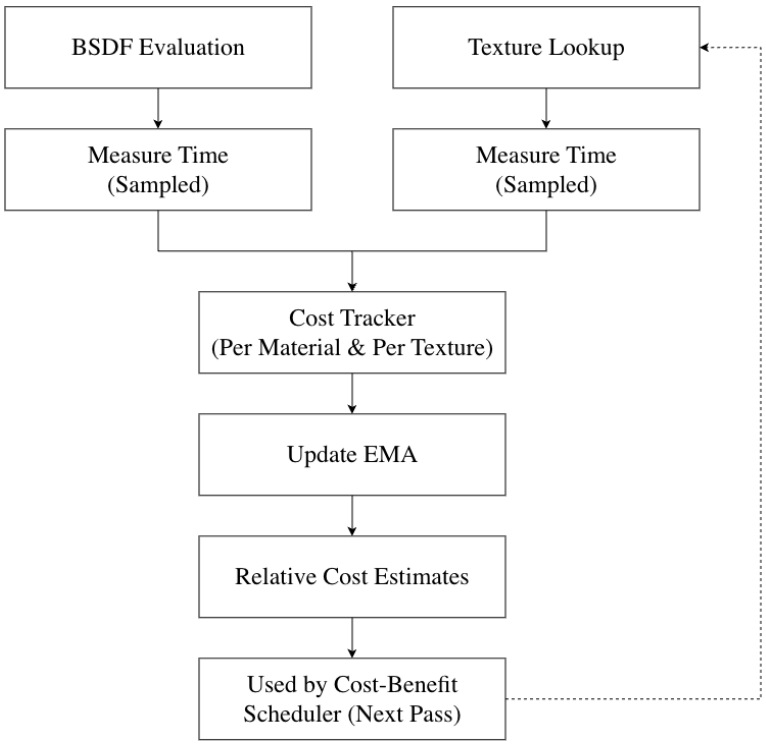



**Figure 4. 12** Runtime cost estimation performed by the CostTracker.

### 4.3.5 Cost-Aware Russian Roulette

The Cost-Aware Russian Roulette strategy adds the estimated shading cost to the path termination decision. The continuation probability uses the ray throughput and the estimated material cost, after which a random test determines whether the path continues. For paths that survive, the throughput is adjusted by the continuation probability so that the estimator remains unbiased.

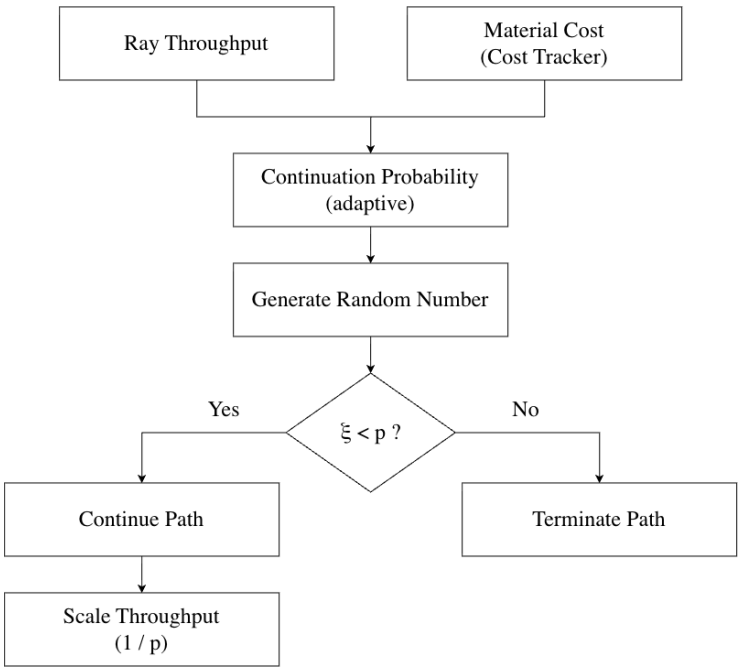



**Figure 4. 13** Cost-Aware Russian Roulette termination process.

This completes the adaptive scheduling framework used in the renderer. The scheduling policies change the order of the shading work, while the CostTracker and Cost-Aware Russian Roulette provide the runtime information and path termination used by the Cost-Benefit approach.

# 5 Evaluation Methodology

Chapter 4 described how the renderer and scheduling system were built. Here, the different scheduling policies are tested using the same scenes and rendering setup. The results are compared using rendering time, execution coherence, memory locality, scheduling overhead, and image differences.

## 5.1 Experimental Setup

Although the renderer supports both macOS and Linux, all experiments were run on Linux to keep the environment consistent. Two procedural OpenUSD scenes were used: a Mixed Scene containing Utah teapots and Stanford XYZRGB dragons with different materials and textures, and a Dragon Scene containing only Stanford XYZRGB dragons. The scenes were rendered at four sample counts and resolutions: 256 samples at 600 x 600, 1024 at 720 x 720, 2048 at 840 x 840, and 4096 at 1080 x 1080. Runtime data was saved to CSV, while difference heatmaps were generated to check for changes between the rendered images.

(a) Mixed Scene (b) Dragon Scene

**Figure 5. 1** Benchmark scenes used during the evaluation.

The two scenes provide different workloads for the scheduling policies. The Mixed Scene contains a more varied combination of geometry, materials, and textures, while the Dragon Scene has a higher triangle count and a more consistent distribution of geometry and

materials. Both scenes include a planar mirror to introduce reflections and additional light bounces. Their main characteristics are shown in Table 5.1.

**Table 5. 1** Benchmark scenes used during the evaluation.

| **Scene** | **Purpose** | **Characteristics** |
|---|---|---|
| Mixed Scene | Evaluate heterogeneous workloads | 45 Utah teapots and 45 Stanford XYZRGB dragons with multiple materials and textures. A planar mirror was included to generate reflections and additional light bounces. |
| Dragon Scene | Evaluate coherent workloads | 90 Stanford XYZRGB dragons with a higher triangle count and a more consistent distribution of geometry and materials. A planar mirror was included to increase secondary ray generation. |

Every scheduling policy was tested using the same scenes, sample counts, resolutions, and rendering configuration. Three runs were performed for each scene, policy, and configuration, with the mean runtime reported. The complete configuration is given in Table 5.2.

**Table 5. 2** Experimental configuration.

| **Parameter** | **Configuration** |
|---|---|
| Evaluation Platform | Linux |
| Development Platforms | macOS and Linux |
| Benchmark Scenes | Mixed Scene, Dragon Scene |
| Scene Composition | 90 objects per scene.<br>Mixed Scene: 45 Utah teapots + 45 Stanford XYZRGB dragons.<br>Dragon Scene: 90 Stanford XYZRGB dragons. |
| Samples / Resolution | 256 @ 600 x 600, 1024 @ 720 x 720, 2048 @ 840 x 840, 4096 @ 1080 x 1080 |
| Scheduling Policies | None, Material, Texture, Cost-Benefit |

| | |
|---|---|
| Repetitions | 3 runs per configuration (mean reported) |
| Output | Runtime metrics (CSV), rendered images, and difference heatmaps |

All runs used the same hardware and software environment. The evaluation machine and software versions are listed in Table 5.3.

**Table 5. 3** Evaluation hardware and software environment.

| **Component** | **Specification** |
|---|---|
| Operating System | Red Hat Enterprise Linux 9.6 (Plow), 64-bit |
| Processor | Intel® Core™ i7-13700 (16 cores: 8 Performance + 8 Efficient, 24 threads, up to 5.2 GHz) |
| Memory | 64 GB RAM |
| Build System | CMake 3.20+, Ninja |
| C++ Standard | C++17 |
| Python | 3.12+ |
| Scene Format | OpenUSD 26.5 |
| Ray Tracing | Intel Embree 4 |
| Parallelism | Intel oneTBB |
| Image I/O | OpenImageIO 2.5 |
| Denoising | Intel Open Image Denoise 2 |
| GUI | PySide6 6.7+ |
| Plotting | Matplotlib, Pandas |
| Testing | GoogleTest |

## 5.2 Performance Metrics

Several runtime measurements were collected during each render to compare the scheduling policies. These cover shading performance, scheduling overhead, execution

coherence, memory locality, and the distribution of shading work. The metrics used in the evaluation are listed in Table 5.4.

**Table 5. 4** Performance metrics collected during the evaluation.

| Metric | Description | Purpose |
|---|---|---|
| Shade Time | Time spent performing BSDF evaluation. | Measures the impact of scheduling on shading performance. |
| Sort Overhead | Time required to generate the scheduled ordering. | Measures the cost of scheduling. |
| Run Length Coherence | Average number of consecutive shading records sharing the same material. | Measures execution coherence. |
| Cache Line Homogeneity | Percentage of neighboring records accessing the same cache line. | Estimates memory locality. |
| Shading Hits | Number of shading records processed for each material. | Shows workload distribution. |

## 5.3 Image Validation

Each rendered image was compared with the baseline golden render to check whether changing the scheduling policy affected the final result. Difference heatmaps were used to highlight pixel differences between the images. Since the policies only change the order in which shading records are processed, the renders are expected to remain visually consistent, with small differences possible due to floating-point calculations.

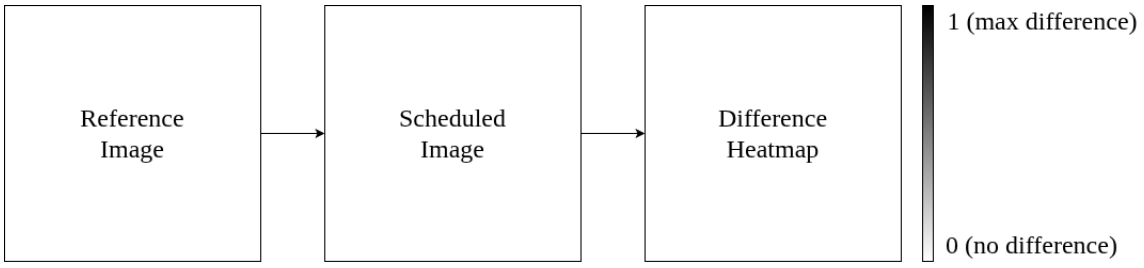



**Figure 5. 2** Image validation using reference render and difference heatmap.

# 6 Results

This chapter presents the results obtained from the experiments described in Chapter 5. It first shows the rendered benchmark scenes and image validation results, followed by the performance results for the different scheduling policies. The results are then compared using rendering time, execution coherence, memory locality, and scheduling overhead.

## 6.1 Rendering Platform

Before comparing the scheduling policies, the renderer was tested with several different scenes to verify the main rendering and scene-loading components. Figure 6.1 shows an example render produced by the wavefront renderer, including different material types, indirect lighting, shadows, reflections, and multiple light bounces.

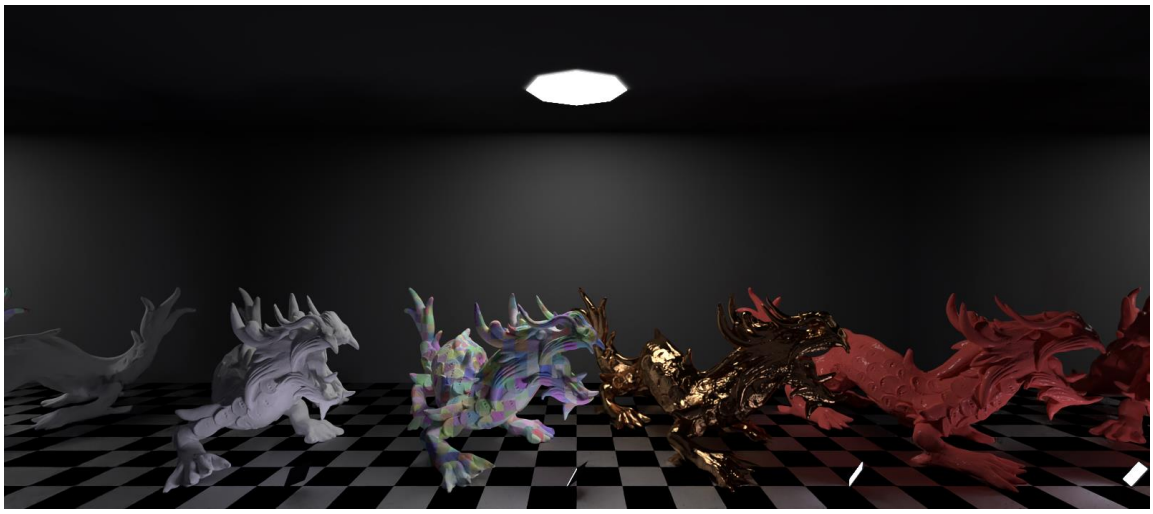


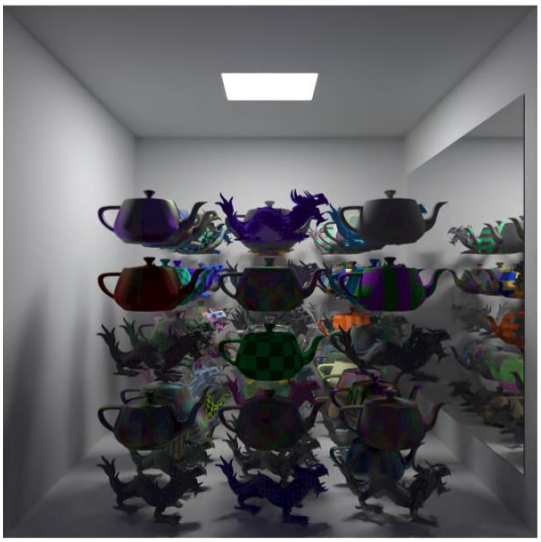
**Figure 6. 1** Example image produced by the proposed wavefront renderer.

The renderer was also tested using Pixar's Kitchen Set to check the OpenUSD scene loading pipeline. Unlike the procedural benchmark scenes, the Kitchen Set contains more detailed geometry, materials, and textures. The scene was loaded and rendered through the same pipeline, showing that the renderer could handle a larger asset.

**Figure 6. 2** Pixar Kitchen Set rendered using the proposed OpenUSD pipeline.

The two benchmark scenes used for the scheduling experiments are shown in Figure 6.3. The Mixed Scene contains a mixture of geometry, materials, and textures. The Dragon Scene contains more triangles and a more consistent distribution of materials. Reflective surfaces are included in both scenes to produce additional secondary rays during rendering.

(a) Mixed Scene



(b) Dragon Scene

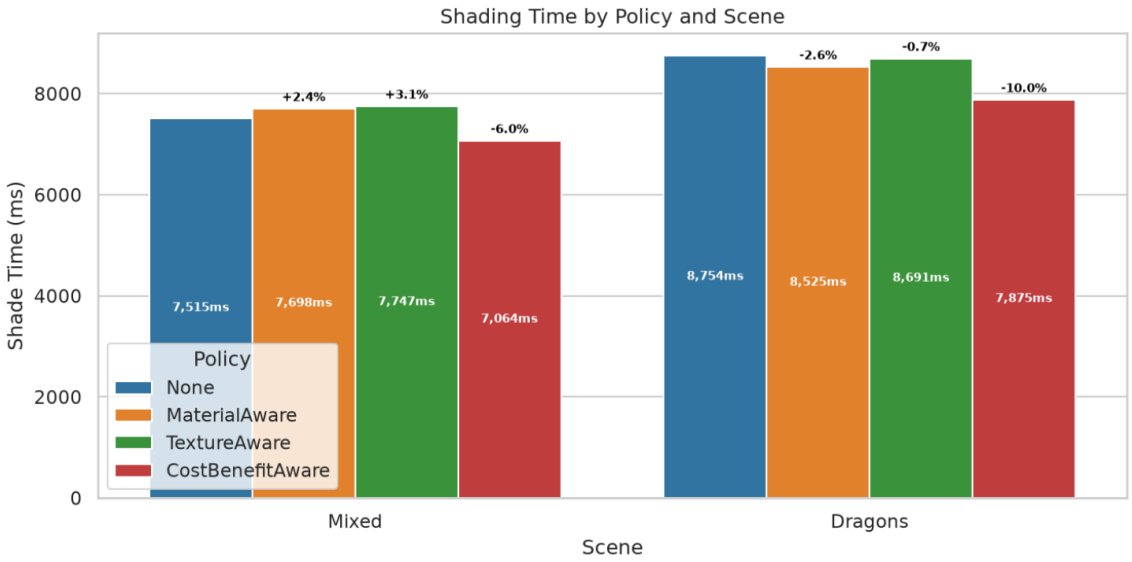
**Figure 6. 3** Benchmark scenes used throughout the experimental evaluation.

## 6.2 Scheduling Performance

This section compares the four scheduling policies using the benchmark scenes and the performance metrics described in Chapter 5. All policies were tested using the same rendering configurations. Unless stated otherwise, the reported values are the arithmetic mean of three independent runs.

### 6.2.1 Shade Time

(a) 256 samples (600x600)




(b) 4096 samples (1080x1080)

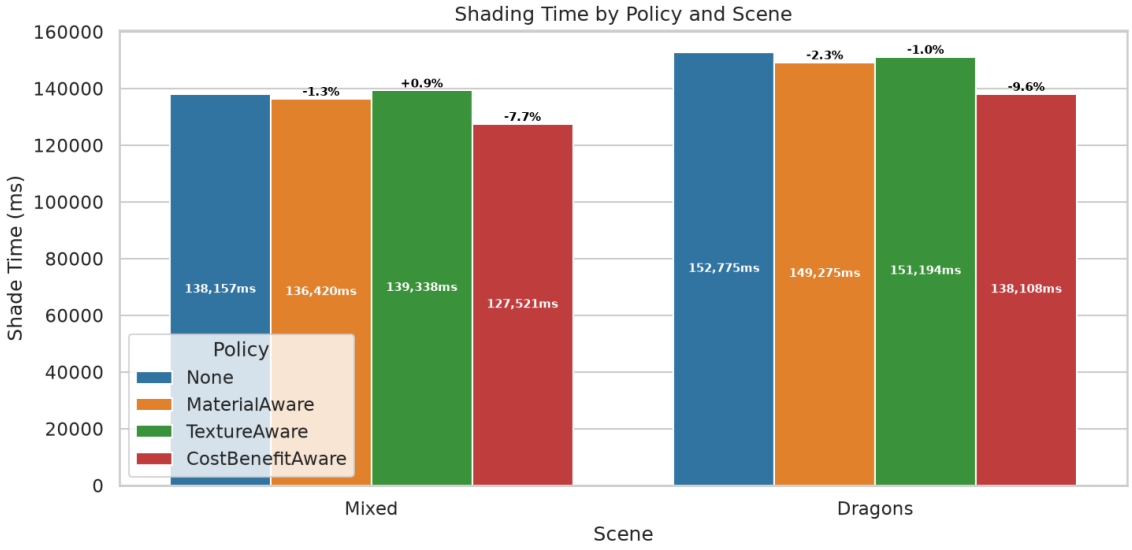



**Figure 6. 4** Comparison of shading time across the evaluated scheduling policies for the Mixed and Dragon benchmark scenes.

Shading time was used as the main performance metric because the scheduling framework targets the shading stage. Cost-Benefit produced the lowest shading time in both benchmark scenes, reducing it by around 6–10% compared with the baseline at 256 samples and 8–10% at 4096 samples. Material and Texture showed little improvement overall, with Texture often slightly slower than the unscheduled renderer. The results suggest that static scene information alone is not enough to improve shading performance, while including

measured shading cost in the ordering was more effective. The improvement was consistent across both scenes and increased with the rendering workload, reaching its largest reduction at the highest sample count, as shown in Figure 6.4.

**Table 6. 1** Average shading time relative to the baseline at 4096 samples (1080x1080).

| **Policy** | **Mixed Scene** | **Dragon Scene** |
|---|---|---|
| None | 0.0% | 0.0% |
| Material | -1.3% | -2.3% |
| Texture | +0.9% | -1.0% |
| Cost-Benefit | -7.7% | -9.6% |

## 6.2.2 Pipeline Breakdown

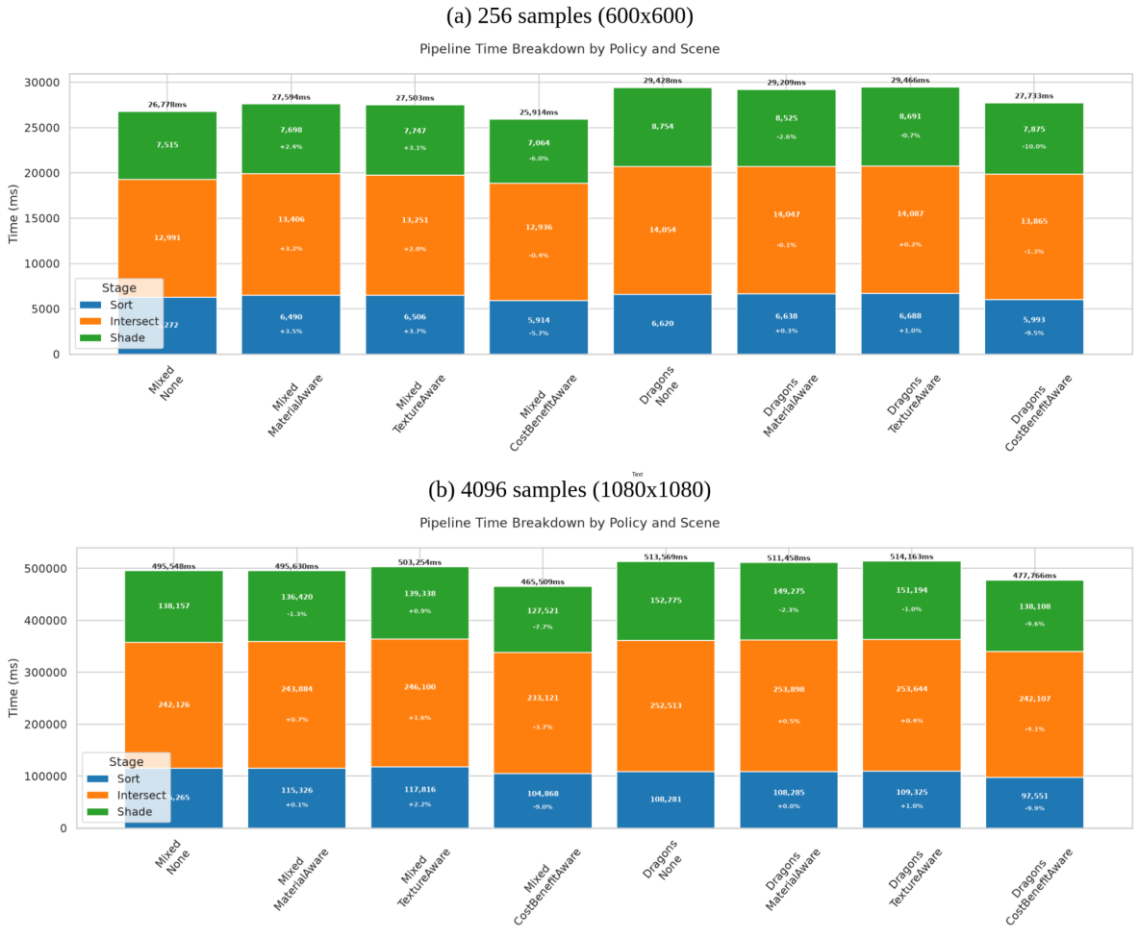



**Figure 6. 5** Pipeline time breakdown across the evaluated scheduling policies for the Mixed and Dragon benchmark scenes.

The pipeline breakdown shows where the performance changes occur between the different stages. Across both benchmark scenes and rendering configurations, Cost-Benefit reduced the time spent in each stage, with the largest reduction coming from shading. The cost estimates change the order in which work is processed, and the resulting secondary rays are also more coherent, giving a smaller improvement in traversal time. The scheduling stage adds some overhead for the cost analysis, but this remains small compared with the time saved elsewhere. As a result, Cost-Benefit achieved the lowest total pipeline time across the experiments, with improvements extending beyond the shading stage. The pipeline breakdown for the representative configurations is shown in Figure 6.5.

### 6.2.3 Execution Coherence

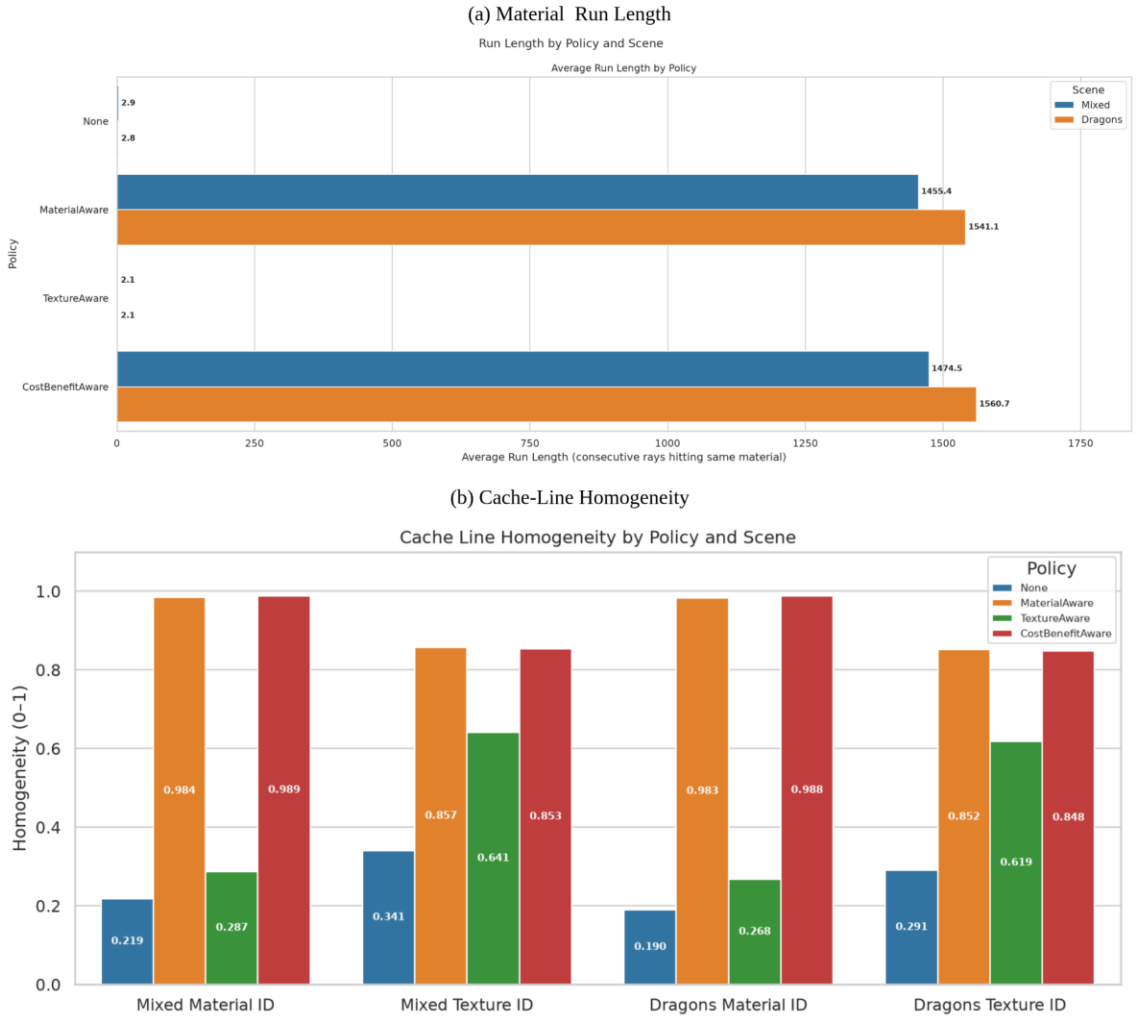



**Figure 6. 6** Comparison of execution coherence across the evaluated scheduling policies.

The rendering improvements are also reflected in the execution coherence of the scheduling policies. Without scheduling, the shading records remain largely mixed, with average material run lengths of only two to three records and very low cache-line homogeneity.

Material and Cost-Benefit both group similar workloads together, increasing the average run length to more than 1,400 consecutive records and material cache-line homogeneity to almost 99%. Although both reach a similar level of coherence, Cost-Benefit consistently gives better rendering performance because it also considers the runtime cost of the work. Rather than only grouping records by material, it can change their order according to the estimated execution cost. This suggests that coherence alone is not enough to obtain the best performance, and that the additional cost information contributes to the results. The coherence metrics for each scheduling policy are shown in Figure 6.6.

**Table 6. 2** Execution coherence metrics for the evaluated scheduling policies at 4096 samples (1080x1080).

| **Policy** | **Material Run Length (Mixed)** | **Material Homogeneity (Mixed)** | **Material Run Length (Dragons)** | **Material Homogeneity (Dragons)** |
|---|---|---|---|---|
| None | 2.9 | 0.219 | 2.8 | 0.190 |
| Material | 1455.4 | 0.984 | 1541.1 | 0.983 |
| Texture | 2.1 | 0.287 | 2.1 | 0.268 |
| Cost-Banefit | 1474.5 | 0.989 | 1560.7 | 0.988 |

### 6.2.4 Image Difference and Rendering Correctness

The rendered images were compared to check whether changing the scheduling policy affected the final result. Cost-Benefit was compared with the baseline using pixel-wise luminance differences, maximum differences, and mean differences for each RGB channel, as shown in Figure 6.7.

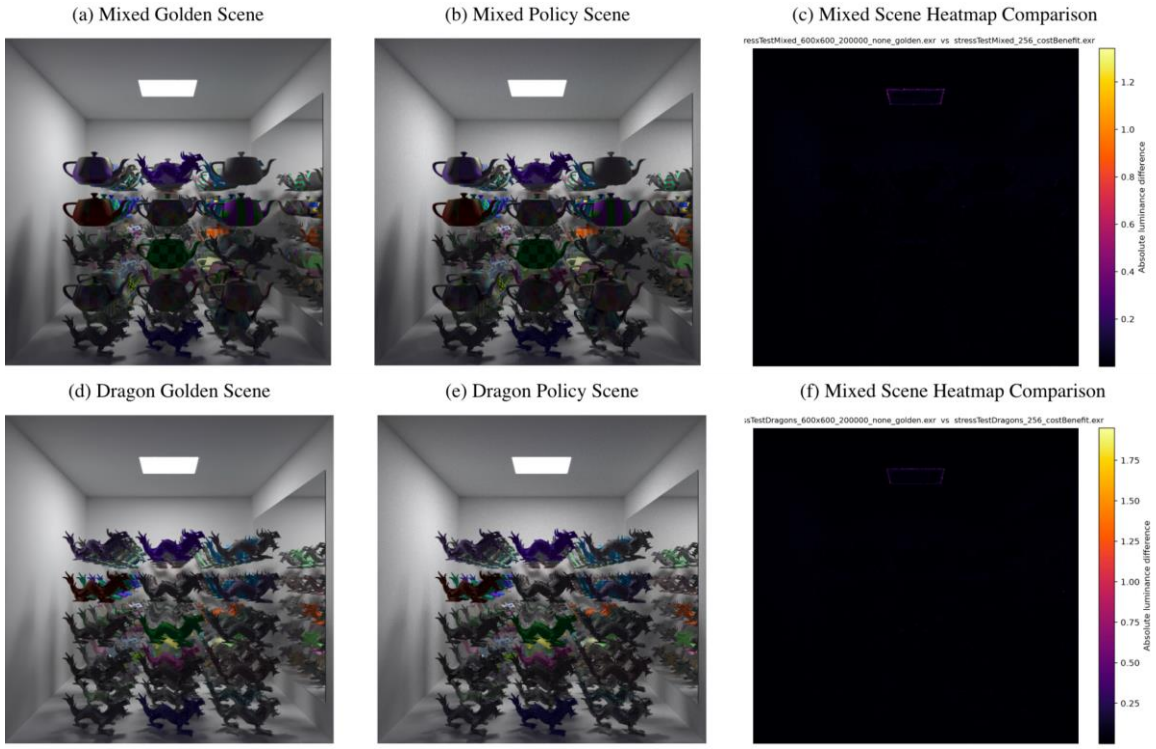



**Figure 6. 7** Image Difference Comparison for the Mixed and Dragon Benchmark Scenes.

**Table 6. 3** Image Difference measurement between the baseline and Cost-Benefit renders.

| **Benchmark Scene** | **Mean luminance difference** | **Maximum luminance difference** | **Mean R difference** | **Mean G difference** | **Mean B difference** |
|---|---|---|---|---|---|
| Mixed | 0.0088 | 1.3423 | 0.0089 | 0.0092 | 0.0096 |
| Dragon | 0.0100 | 1.9491 | 0.0103 | 0.0104 | 0.0109 |

Table 6.3 shows that the average differences between the baseline and Cost-Benefit renders are very small for both scenes, including the individual color channels. The maximum differences are higher because they come from individual pixels and do not describe the image as a whole. In the heatmaps, most of these differences are around the ceiling light, where the higher luminance makes small floating-point variations easier to see. Smaller differences can also be seen around parts of the scene geometry. Most of the image remains unchanged, so the scheduling changes only produce minor numerical differences without changing the final render. This supports the hypothesis that the workload can be reorganized to improve performance without affecting rendering correctness.

# 7 Conclusion

This dissertation investigated whether adaptive workload scheduling could improve execution coherence and rendering performance in a CPU wavefront path tracer. To investigate this, a complete OpenUSD-based renderer and a modular scheduling framework were developed, allowing different workload organization strategies to be implemented and compared within the same rendering pipeline. The results show that adaptive scheduling can improve rendering performance, but the scheduling decisions need to account for the runtime behavior of the workload rather than relying only on static scene information. The Material and Texture schedulers improved execution coherence, but neither consistently improved rendering performance. The Cost-Benefit scheduler combines execution coherence, runtime cost estimation, and adaptive path termination. In the results, it reduced shading time by up to 10%, while also giving the highest memory locality and execution coherence across the benchmark scenes and rendering configurations. The image comparisons showed that the baseline and Cost-Benefit renders were almost identical, with the differences mainly coming from a small number of pixels. This shows that changing the scheduling order did not change the final rendered image.

A reduction of up to 10% in shading time is significant for a rendering system, and its value becomes even greater when applied across a production workload. Animated films and visual effects projects can require significant amounts of CPU hours to render a single sequence. A similar reduction applied to larger production workloads could save hundreds or even thousands of compute hours, while also reducing render farm usage, energy consumption, and overall processing time. Faster renders also give artists across different departments more time to work, iterate, and evaluate their results rather than waiting for frames to finish. This extra time can benefit the wider production by allowing changes to be tested sooner and reducing delays between stages of the workflow. Even a small improvement can therefore be useful in production, especially when it can be achieved without changing the rendering algorithm or affecting image quality.

## 7.1 Limitations

Although the framework improved rendering performance, there are still some limitations to the current evaluation. The Cost-Benefit scheduler needs an initial warm-up period to collect enough runtime measurements, so the first scheduling decisions are made with limited cost information. The experiments were also carried out using procedurally generated benchmark scenes and a single CPU configuration. This kept the tests consistent, but does not show how the framework would behave with the range of assets and hardware

used in production. Traversal scheduling was enabled for all experiments as well, which kept the comparison between shading policies consistent but meant that its individual contribution could not be measured separately. Finally, the work was limited to CPU wavefront rendering. GPU rendering remains an interesting direction for future work, particularly because the higher level of execution divergence on GPUs could provide more opportunities for adaptive scheduling.

## 7.2 Further Development

The modular design of the framework also leaves several areas for further development. The renderer could be extended with additional physically based rendering features, including full MaterialX support, so that more complex production materials can be used directly through OpenUSD workflows. The scheduling framework could also be tested on GPU wavefront renderers, where the higher level of execution divergence may provide more room for performance improvements. The scheduling decisions could be further developed by selecting policies dynamically based on the scene, or by replacing the current exponential moving average with a learned cost model using features such as material type and texture complexity. Cost-awareness could also be extended beyond shading to the ray sorting and traversal stages. Finally, the framework should be tested with production-scale assets and other rendering methods, including bidirectional path tracing. This would show whether the scheduling approach also works with larger and different rendering workloads.

## 7.3 Broader Impact

The framework developed in this dissertation shows that workload scheduling does not have to be a fixed step in the rendering pipeline and can instead respond to the work being processed. Since the scheduling framework is separate from the rest of the renderer, similar approaches could be added to other wavefront renderers without changing the sampling, shading, or light transport methods themselves. As rendering workloads become more complex, adaptive scheduling could become another way of improving renderer performance alongside acceleration structures, sampling, and denoising. More importantly, reducing render time gives artists more time to iterate and refine their work. This research provides a starting point for applying adaptive scheduling to future rendering systems.

# References


Bainbridge, J. (2015). Sorted Shading for Uni-Directional Pathtracing. Bournemouth University.

Dammertz, H., Hanika, J., Keller, A., & Lensch, H. P. A. (2010). *A hierarchical automatic stopping condition for Monte Carlo global illumination*. WSCG 2010.

Dutré, P., Bala, K., & Bekaert, P. (2006). *Advanced global illumination* (2nd ed.). A K Peters.

Eisenacher, C., Davidović, T., Keller, A., Slusallek, P., & Dachsbacher, C. (2013). Sorted deferred shading for production path tracing. Computer Graphics Forum, 32(4), 85–95.

Kajiya, J. T. (1986). The rendering equation. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*, 20(4), 143–150. https://doi.org/10.1145/15922.15902

Laine, S., Karras, T., & Aila, T. (2013). Megakernels considered harmful: Wavefront path tracing on GPUs. Proceedings of High-Performance Graphics, 137–143. https://doi.org/10.1145/2492045.2492060

Morton, G. M. (1966). *A computer oriented geodetic database and a new technique in file sequencing*. IBM Ltd.

Pixar Animation Studios. (2016). *Universal Scene Description (OpenUSD)*. https://openusd.org/

Pharr, M., Jakob, W., & Humphreys, G. (2023). Physically based rendering: From theory to implementation (4th ed.). MIT Press.

Pharr, M., Kolb, C., Gershbein, R., & Hanrahan, P. (1997). Rendering complex scenes with memory-coherent ray tracing [Figures PDF]. Stanford University. https://graphics.stanford.edu/papers/coherentrt/coherentrt-figs.pdf

Schlick, C. (1994). An inexpensive BRDF model for physically based rendering. *Computer Graphics Forum*, 13(3), C233–C246.

Veach, E. (1998). *Robust Monte Carlo methods for light transport simulation* (Doctoral dissertation, Stanford University).

Wald, I., Slusallek, P., Benthin, C., & Wagner, M. (2001). Interactive rendering with coherent ray tracing. Computer Graphics Forum, 20(3), 153–165.

Wald, I., Woop, S., Benthin, C., Johnson, G. S., & Ernst, M. (2014). Embree: A kernel framework for efficient CPU ray tracing. ACM Transactions on Graphics, 33(4), Article 143. https://doi.org/10.1145/2601097.2601199

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3), 419–420.

# Appendix

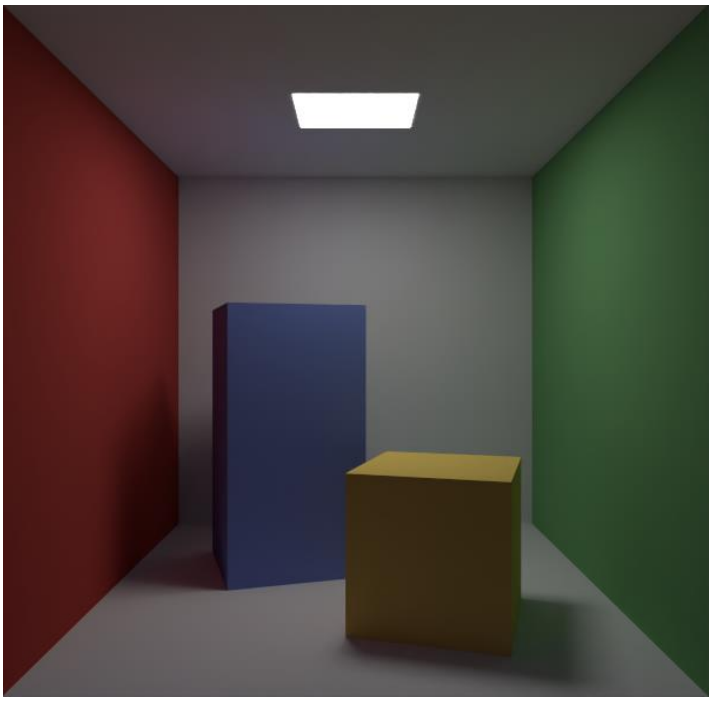


**Figure A. 1** Cornell Box scene rendered using the proposed wavefront renderer.

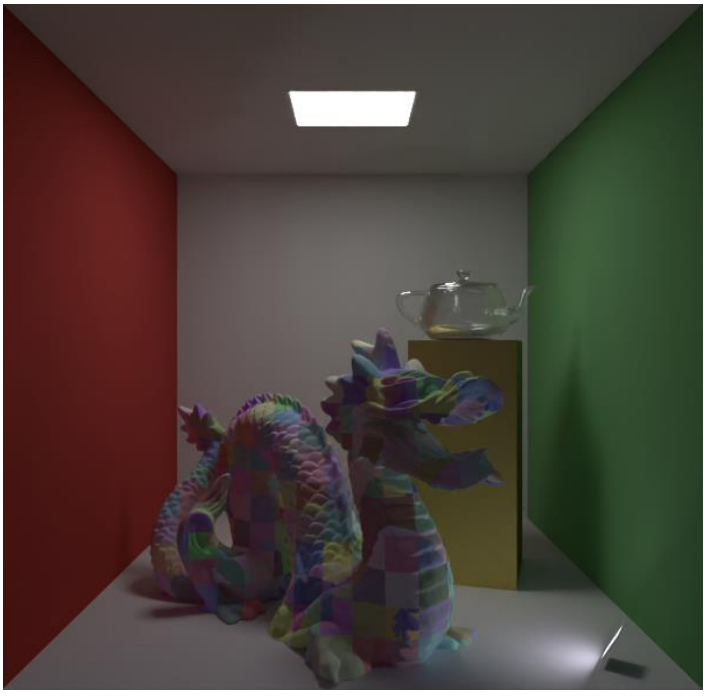


**Figure A. 2** Cornell Dragon scene rendered using the proposed wavefront renderer.

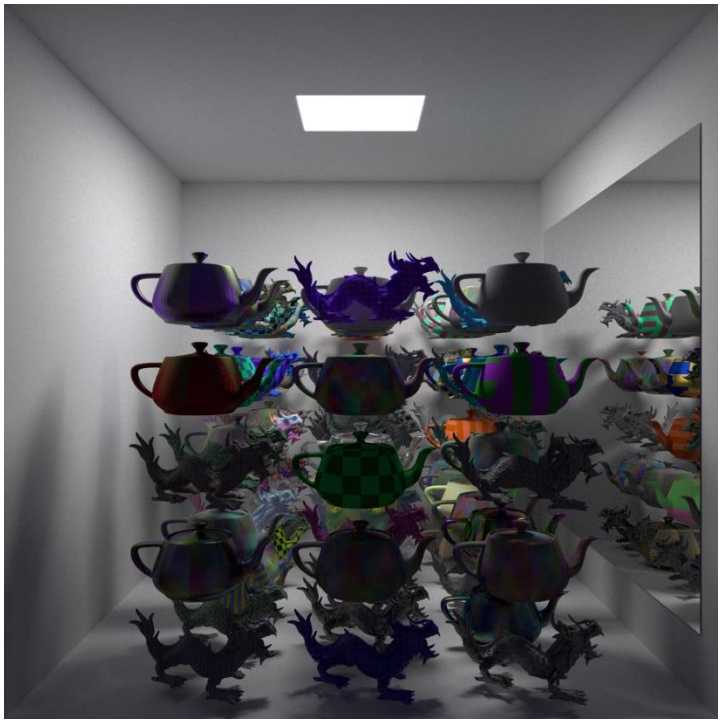


**Figure A. 3** Mixed scene rendered using the proposed wavefront renderer.

**Figure A. 4** Dragon scene rendered using the proposed wavefront renderer.

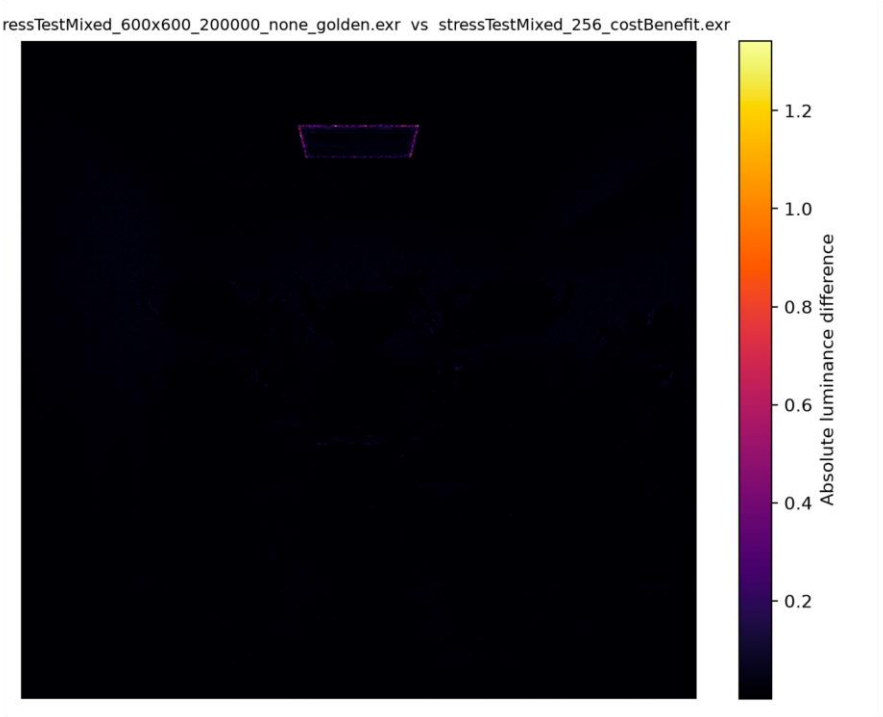



**Figure A. 5** Heatmap comparison between Mixed scene golden render and Cost-Benefit policy.

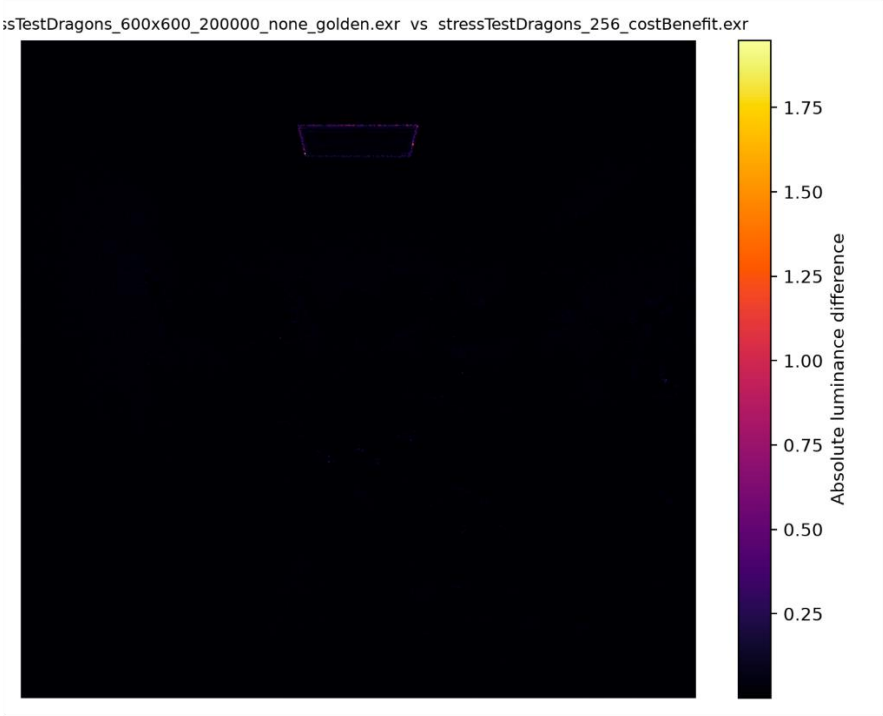



**Figure A. 6** Heatmap comparison between Dragon scene golden render and Cost-Benefit policy.

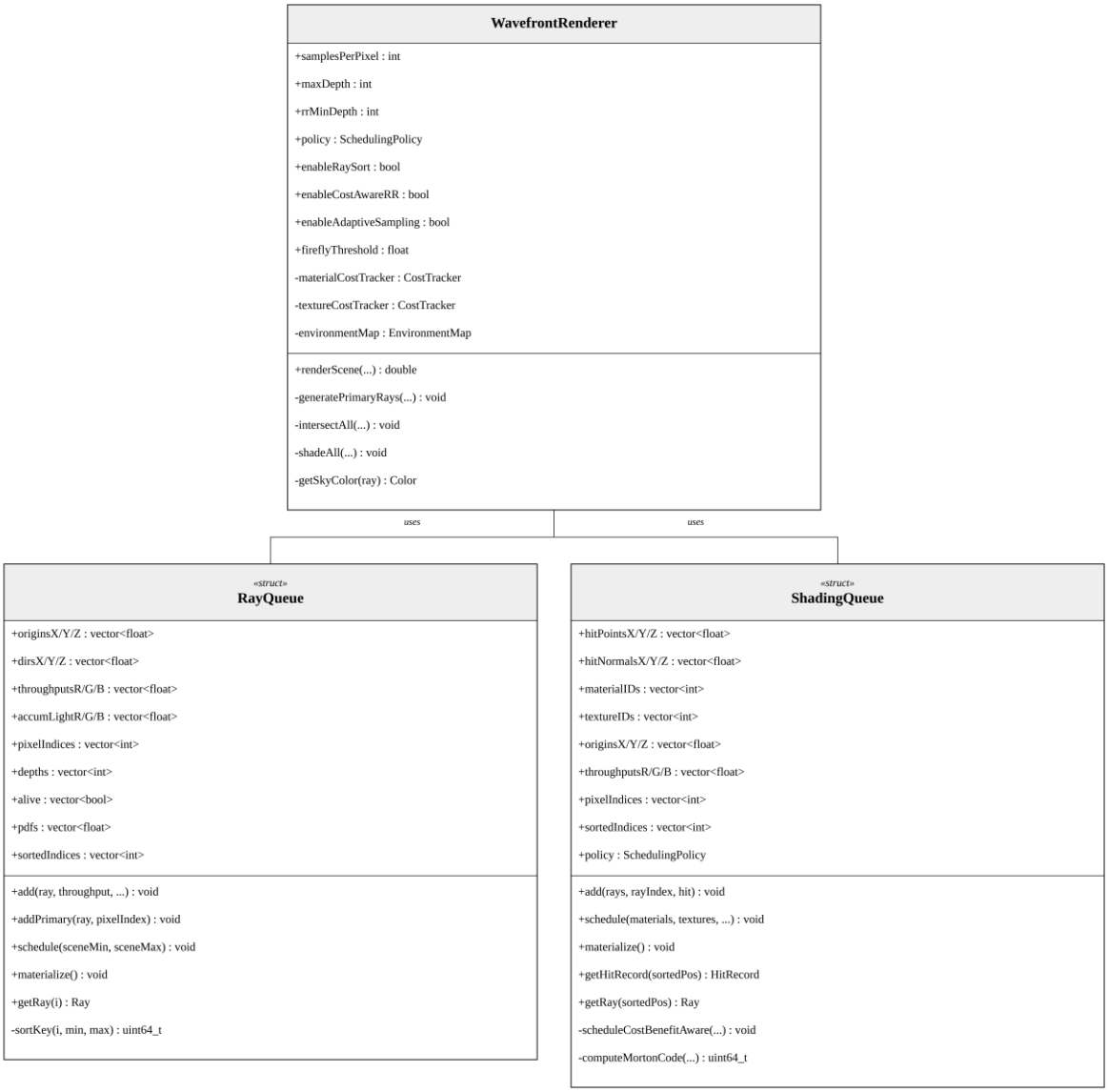



**Figure A. 7** Class structure of the WavefrontRenderer, RayQueue, and ShadingQueue.

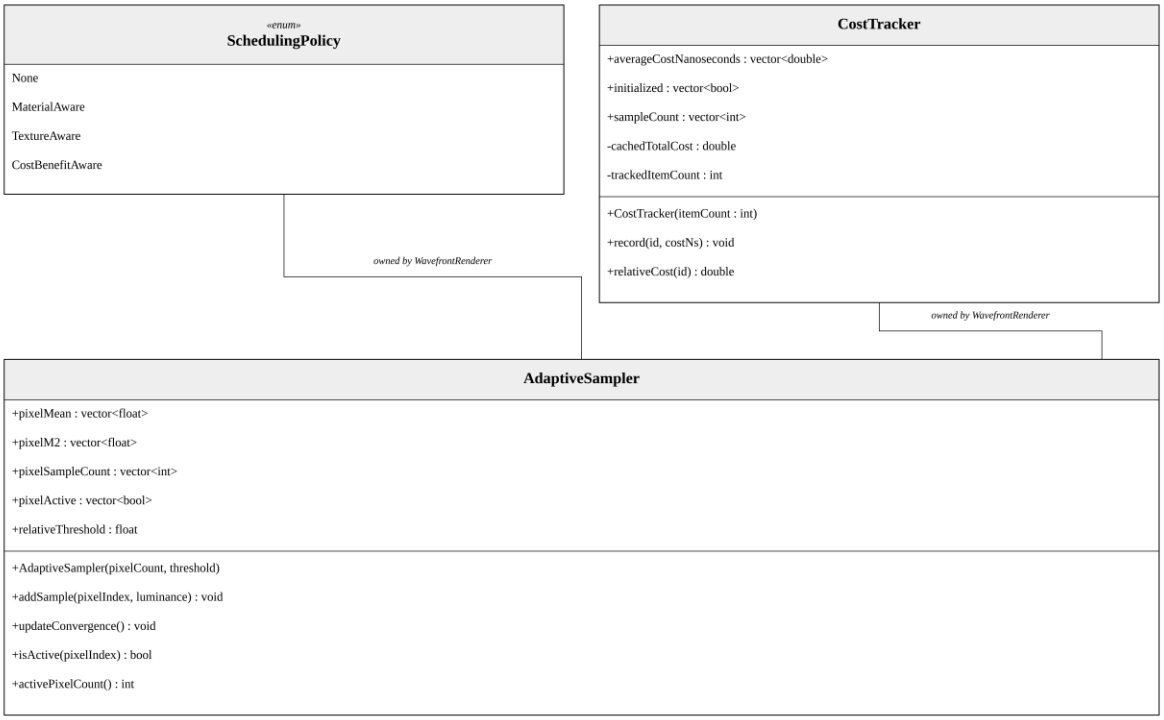



**Figure A. 8** Scheduling policies, runtime cost tracking, and adaptive sampling components.

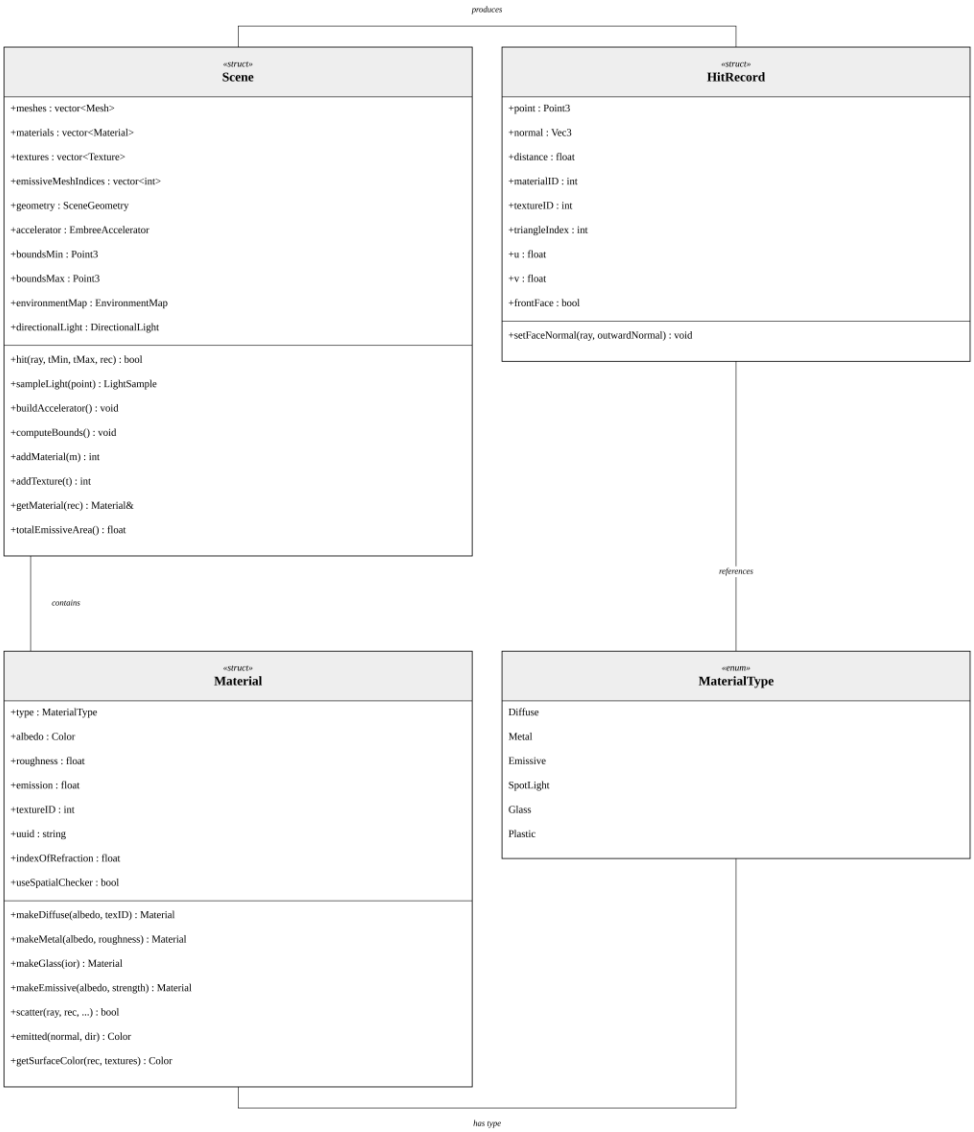



**Figure A. 9** Scene, material, and hit record class structure.

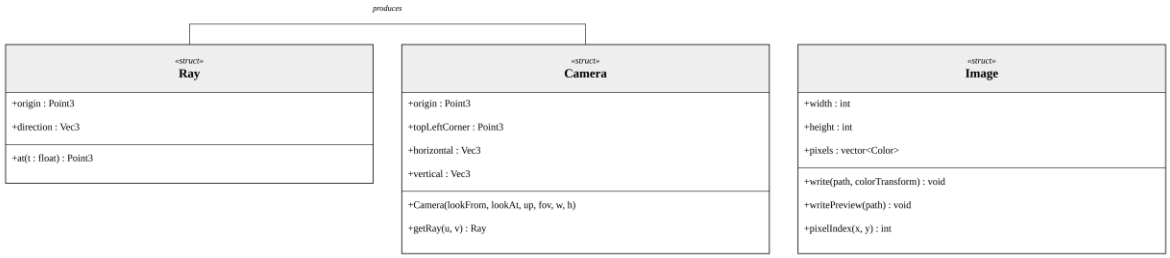



**Figure A. 10** Ray, camera, and image class structure.