UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

IMMEDIATE RADIOSITY

DAVID GARCIA TEJEDA

Thesis supervisor
ANTONIO CHICA CALAF (Department of Computer Science)

Degree
Master's Degree in Innovation and Research in Informatics (Advanced Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

Real-time global illumination in interactive rendering has traditionally relied on per-frame ray traversal (through voxel cone tracing, hardware ray tracing, or radiance probe resampling) incurring a runtime cost that scales with scene complexity and competes directly with the broader rendering pipeline.

The thesis presents a voxel-based radiosity pipeline combining GPU-accelerated form factor precomputation via stochastic ray marching, a Jacobi iterative solver dispatched through GPU compute, signed distance field raymarching for indirect visibility, a specular contribution in shading time and a novel specular writeback path (SD^*) that transports specular contributions back into the diffuse radiance cache.

Contents

1. Introduction

Real-time global illumination remains one of the central unsolved problems in interactive rendering. Commercial engines approximate indirect light through a combination of voxel cone tracing, screen-space techniques, and radiance caches Crassin et al. (2011); Wright and Hillaire (2021), but all of these approaches share a common cost: some form of ray traversal must occur every frame, either through hardware ray tracing, voxel cone marching, or probe resampling. That per-frame traversal cost scales with scene complexity and imposes a hard budget that competes directly with the rest of the rendering pipeline.

This project investigates a different formulation. Radiosity Goral et al. (1984); Cohen et al. (1998a) expresses global illumination as a linear system over surface patches: the radiance of each patch is the weighted sum of radiance contributions from all visible patches, where the weights (the view factors) encode the geometry of visibility between patch pairs. Crucially, the view factors depend only on scene geometry, not on lighting. If the scene is static, they can be precomputed once and stored as a sparse matrix[1]. The per-frame cost then reduces to a single sparse matrix-vector product (*SpMV*), a well-understood linear algebra primitive that maps efficiently onto GPU compute pipelines.

The hypothesis driving this work is that precomputing visibility into a sparse radiosity matrix, and reducing the runtime GI problem to an iterative *SpMV* can produce a competitive real-time indirect illumination result at lower per-frame cost than ray-traversal-based alternatives.

2. Rendering Background

This project assumes familiarity with the standard real-time rendering pipeline. The following foundational concepts are used throughout without further explanation; references are provided for completeness.

Physically Based Rendering (PBR). The shading model used in the forward pass is based on the microfacet BRDF formulation introduced by Cook and Torrance Cook and Torrance (1982), extended with the GGX normal distribution function Walter et al. (2007). The specific parameterisation (roughness, metallic, and base colour) follows the Disney principled BRDF Burley (2012), which has become the de facto standard for real-time PBR pipelines. Albedo, roughness, and metallic values are supplied as texture maps sampled in the fragment shader.

Normal Mapping. Surface normals are perturbed at the fragment level using tangent-space normal maps Blinn (1978); Cohen et al. (1998b).

Shadow Mapping. Direct visibility between a light source and a surface point is evaluated using the shadow map technique introduced by Williams Williams (1978). The implementation uses percentage closer filtering (PCF) to reduce aliasing at shadow boundaries.

GPU Compute Pipeline. The GI pipeline described in this work is implemented entirely as Vulkan compute shaders. The reader is assumed to be familiar with compute shader dispatch, SSBO (Shader Storage Buffer Object) memory layout, and synchronisation primitives such as pipeline barriers and memory barriers. The Vulkan specification Khronos Group (2023) is the authoritative reference for these constructs.

3. State of the Art

In this section I will comment more specifically on implementations of current solutions for both real time GI and radiosity relight engines as well as screen-space based algorithms.

[1] A matrix is sparse if the majority of elements are 0

First a brief introduction to the rendering equation and general Global Illumination and a first insight of the computational complexity of its calculation.

3.1. The Rendering Equation and Global Illumination

The theoretical foundation of all global illumination algorithms is the rendering equation, formulated by Kajiya Kajiya (1986) (Figure 1):

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, (\omega_i \cdot \mathbf{n})\, d\omega_i \tag{1}$$

where $L_o(\mathbf{x}, \omega_o)$ is the outgoing radiance at surface point $\mathbf{x}$ in direction ω_o, L_e is emitted radiance, f_r is the bidirectional reflectance distribution function (BRDF), $L_i(\mathbf{x}, \omega_i)$ is the incident radiance arriving from direction ω_i, and the integral is taken over the hemisphere Ω centered on the surface normal $\mathbf{n}$.

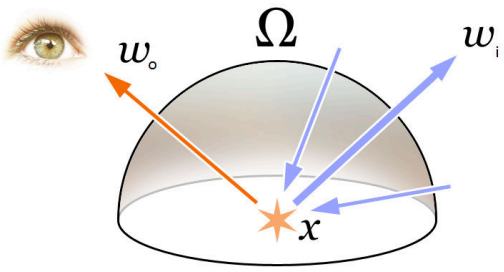


Figure 1: The rendering equation describes the amount of light leaving a point x along a particular viewing direction, given functions for incoming light and emitted light, and a BRDF. (Wikipedia).

3.2. Radiosity algorithms

The fundamental difficulty of global illumination stems from the recursive nature of this equation: $L_i(\mathbf{x}, \omega_i)$ is itself the outgoing radiance of whatever surface is visible from $\mathbf{x}$ along ω_i, making the full solution a system of coupled integral equations over all surfaces in the scene. Direct illumination truncates this recursion at one bounce, ignoring the indirect term entirely. Global illumination algorithms differ in how they approximate or solve the full recursive integral.

Radiosity algorithms were first introduced by (Goral et al., 1984) (Main core algorithmic contribution) and (Cohen et al., 1998a) (Introduces the hemicube representation, and progressive refinement radiosity), these three main contributions were going to set the backbone for future systems that were built in the industry for the same principles. These models were refined throughout the late 1980s and 1990s, approximates diffuse light transport by decomposing scene surfaces into finite patches and computing pairwise energy exchange through view factors. Each patch in radiosity is the result of dot product sum between its view factor weights and the current scene lighting vector, resulting in system of linear equations.

This method resulted as both powerful and interesting as well as producing compelling results it required $O(n^2)$ complexity in the number of patches and large amounts of memory to store the factor matrix which ended up making it quite impractical. This was the main reason why this algorithm's popularity was never very prominent. Lightmappers that worked using stochastic raytracing were much more popular due to being far more flexible than Radiosity algorithms and producing a good enough result.

One of the very first algorithms to perform global illumination for dynamic scenes was to bake the scene into light probes from different angles at different positions of the scene, and gather in this way the incoming light from different angles, then the surfaces approximated incoming indirect light by

sampling the neighboring light probes and interpolating the light given its distance. However this system required careful selection of which lightprobes to render to and had numerous memory and bandwidth constraints since it required rendering the scene for different points until hitting convergences from all the lightprobes.

3.3. Radiosity-based Real-Time Relighting Systems

One of the most significant industrial implementations of radiosity relighting is Geomerics *Enlighten* Geomerics Ltd. (2010), later acquired by ARM and integrated into Unity and Unreal Engine as an optional GI backend.

Enlighten precomputes a sparse radiance transfer matrix offline during a preprocessing step that merges scene surfaces into radiosity patches and computes view factors between them.

At runtime, when direct lighting changes, Enlighten executes the radiosity iteration again, propagating updated radiance through the network and the result into lightmap textures or light probe coefficients that are sampled by the forward shader.

This architecture is the closest prior work to the system described in this project, both systems decouple the expensive visibility computation from the per-frame energy propagation, reducing the runtime cost to a sparse matrix-vector product executed on the GPU.

However Enlighten operates at a coarse patch resolution dictated by the lightmap texel density of the scene geometry, and its transport matrix encodes only diffuse-to-diffuse (D→D) energy exchange.

Specular indirect contributions are approximated separately through reflection captures and screen-space techniques rather than being derived from the transport matrix itself.

3.4. Screen-Space based Real-Time Algorithms

Computing global illumination for a dynamic scene is a very hard task, given the advancement in computation power of the GPU, the introduction of compute shaders, and the availability for 3D textures, one of the main candidates for achieving this task was VCT (Voxel Cone Tracing).

Voxel Cone Tracing (VCT) was introduced by Crassin Crassin et al. (2011) as a method to approximate global illumination in real time. Since then it has been adapted into many rendering engines and has many commercial adaptations.

Smarter versions of this make use of computing a *Signed Distance Field* (SDF) for the voxelized volume allowing the cone tracer to use raymarching and speeding up the computation for subsequent frames where the camera orientation does not change much.

Some of them are NVIDIA's *VXGI* system NVIDIA Corporation (2014) integrated into Unreal Engine 4 as an optional GI backend. VXGI maintained a clipmap of voxel grids centered on the camera, allowing dynamic scenes to inject radiance changes incrementally rather than rebuilding the full voxel structure every frame. In Unreal Engine 5, this solution is changed by a hybrid software raymarcher[2] against SDF and a combination with hardware raytracing Lumen Wright and Hillaire (2021).

In the case of Unity's GI system *SDFGI* and the earlier *Voxel GI* baked probe system represent a family of related approaches. These approaches use a sparse volume representation of the scene sampled by probes placed in the scene, sampled and interpolated at shading pass. These probe-based methods trade continuous cone tracing for radiance lookups, reducing bandwidth at the cost of temporal stability and spatial resolution at the boundaries of probes.

The principal limitations of pure VCT are: voxel resolution imposes a lower bound on the spatial

[2]Ray marching is a class of rendering methods for 3D computer graphics where rays are traversed iteratively, effectively dividing each ray into smaller ray segments, sampling some function at each step.

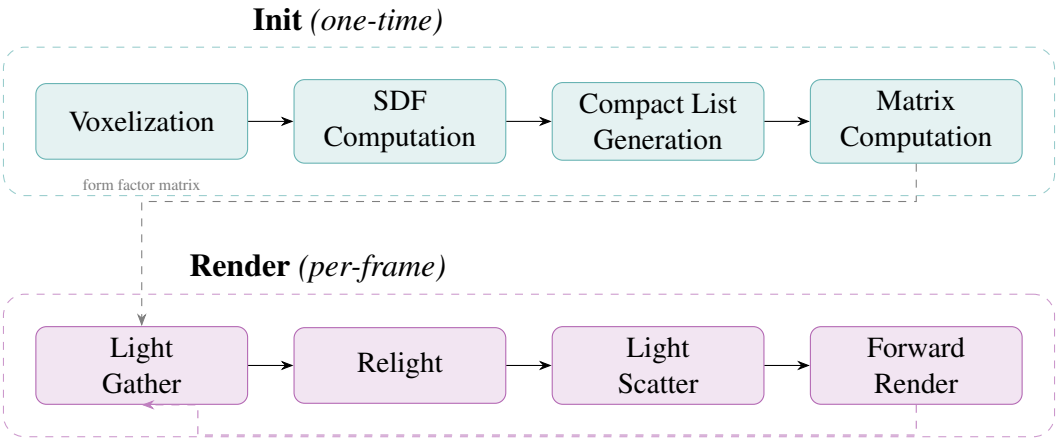


Figure 2: Overview of GI pipeline. The init passes (teal) run once at scene load; the render passes (violet) execute every frame.

frequency of the captured radiance field, leading to light leaking through thin geometry and loss of high-frequency shadow detail. Also the cone tracing step itself introduces a bandwidth-bound bottleneck when many fragments are shaded simultaneously.

Sparse voxel representations Crassin et al. (2011) and clipmap hierarchies NVIDIA Corporation (2014) partially mitigate the memory cost.

4. Immediate Radiosity

I introduce in this section the main algorithm presented in the project which I have decided to call Immediate Radiosity, the core idea is computing the Radiosity power iteration method on the GPU, so we can do real time per frame Radiosity iteration and propagate scene light in the scene in a physically accurate way using precomputed view factors. A complete layout is given in Figure 2.

This method is composed by different stages, which are grouped in two big sets. The *Init* set which contains stages that are required to be executed only once at *Set-up* time and every time there is a change in the geometry of the scene, its main goal is to generate the view factor matrix we are going to use for computing Radiosity.

Then next group of stages are executed every frame, and involve gathering the current light from the scene, computing one iteration of the Radiosity algorithm, and writing the result in a convenient layout so it can be read when rendering the full scene. Finally the result is fed to the forward renderer which composes the final result and presents it to the screen.

The relight stages are needed to be executed every time the light configuration of the scene changes, if the lighting configuration remains the same, the result of the Relight stage can be cached between frames.

The following sections are high level description of the different steps of the whole algorithm, an in-depth description that provides implementation technical details is given below.

4.1. Voxel maps computation

First we need to compute a voxelized representation of the scene in the form of 3D images, the basic idea is that we will project the scene geometry onto the 3D volume and ”raster” the geometry to it, storing the scene albedo[3] and normals. We will compute 2 voxel maps, one for storing albedo and normal of the

[3]Albedo refers to the diffuse component of a material in the PBR material system, commonly referred to the color component

scene and another a *Signed Distance Function* (*SDF* [4]) of the whole scene.

4.2. Matrix computation

We then proceed to compute the view factor matrix in the GPU. For doing so we first need to build a *SolidVoxelList* from the voxel maps, that is, we have to atomically construct a 1D buffer containing all solid voxel IDs. We do this so we can skip constructing view factors for voxels that are actually empty and for identifying every voxel with an ID so they can be assigned a row in the matrix.

This structure, which is further explained below, will allow us to establish a mapping between the entries of the vector of radiance during the Radiosity method execution and the equivalent 3D radiance volume that we will sample. We also need to construct the inverse map of the *SolidVoxelList* which maps IDs to voxel positions so we can scatter light back to a radiance volume to be sampled in shading time.

From this buffer we will be able to construct the view factor matrix for all the patches and store it in a *CSR*[5] form.

4.3. Relighting

Once we have computed the view factor matrix we can now relight the scene every time the lighting configuration changes.

In this process we first must gather the direct lighting in the scene, we have multiple ways for doing so.

The easiest is to run a compute shader that emits for every single solid voxel we have computed the incoming direct light from the light emitters, we can compute it the same way we do in a regular forward rendering shader for shadowmap based lights.

We can also gather illumination from emissive sources (emissive textures or shaders) so they can also be taken into account when computing light emission.

This is the most extensible part of the algorithm, this stage can be modified to gather any particular source of direct light, for this implementation, only spot/sun lights and emissive lights (coming from textures of the geometry in the scene) were programmed for time budget reasons, but in the future it should be easy to implement other sources of light.

Once we have gathered the light emission for each solid voxel we can proceed to compute an iteration of the Jacobi [6] method using traditional compute shader sparse vector matrix multiplication, particularly we do compute:

$$\mathbf{x}^{(k+1)} = \lambda \left(\mathbf{E} + \mathbf{A}\, \mathbf{x}^{(k)} \right)$$

where:

- $\mathbf{x}$ is the per-voxel radiosity vector
- $\mathbf{E}$ the emission vector
- $\mathbf{A} = \rho \cdot F$ the albedo-weighted form factor matrix

[4] A Signed Distance Function (SDF) is a scalar volume in which each texel stores the distance to the nearest solid surface

[5] CSR Sparse matrix format is a commonly used format to store matrices where the majority of elements are 0, typically encodes the data using a vector that contains all the non zero weights of the matrix, and then auxiliary vectors that hold indices to the start (offset) in the weights vector of each row for a given row index of the matrix and another vector that indicates for each value of data the index of the column it refers to

[6] Jacobi iteration solves $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$ by repeated application: $\mathbf{x}^{(k+1)} = \mathbf{A}\mathbf{x}^{(k)} + \mathbf{b}$, converging when $\rho(\mathbf{A}) < 1$ Saad (2003).

- λ a normalization scalar.

The rationale behind the parameter λ which is the only deviation from the original method is implementation specific. Its purpose is to dampen the light propagation of the light factors. Once we have finished, we scatter the radiance back into a 3D image so the fragment shader can read the computed radiance comfortably.

4.4. Method composition

We run the *One-Time* stages once per geometry update, and then we run *per-frame* stages once per frame.

Finally we sample in our regular forward pass the diffuse light volume by using the vertex coordinates as UV coordinates (transformed to the correct space).

4.5. Specular component

Since we calculate in stage **(Voxel maps computation)** all necessary maps to perform ray tracing, on a powerful enough GPU, we can raymarch the SDF radiance volume alongside relighting it with the Radiosity algorithm.

If we do such thing we can then approximate the final GI function per fragment of the scene as the composition of the specular GI function (approximated by the ray tracing algorithm on the forward pass) and the diffuse component (approximated by the **Relighting** stage).

It would be possible also to use a VCT instead, however, given that we already have a very good approximation of the diffuse component through the Radiosity algorithm I believe using an actual Raytracer for the project makes more sense.

Although an actual VCT is recommended because in quality and performance it performs better for semi reflective surfaces.

5. Algorithm

This section is included to give an in-depth description from an algorithmic point of view of the whole method. The purpose of this section is to give the reader enough hints and information to be able to implement the algorithm themselves, making references to the already established algorithms used in the process.

5.1. Voxelization

In this section I will describe the voxelization procedure that the algorithm follows to obtain the voxel maps needed to compute the factor matrix in the first place.

5.1.1 Voxel Images Mapping

The first thing we must establish, which is fundamental to the algorithm, is the voxel resolution for the 3D images used during voxelization. It is recommended to use a resolution that is a power of 2, since the algorithms employed work efficiently with images that satisfy this property but this is not a strict constraint.

Once the voxel resolution is chosen, we need to define the convention for transforming between world coordinates and 3D image coordinates. Two requirements must hold: the transformation must be *isotropic*

across all axes (a cubic region of the scene must remain cubic inside the voxel volume, with no stretching) and we want to make full use of the 3D image extents so that memory is used efficiently.

To achieve this, we compute the scene *Axis-Aligned Bounding Box* (AABB): the smallest box whose dimensions encapsulate the entire scene. For each world-space axis $a \in \{x, y, z\}$:

$$\text{Max}_a = \max_i V_a[i], \qquad \text{Min}_a = \min_i V_a[i] \tag{2}$$

The per-axis extents and the scene origin in world space are then:

$$(\Delta x,\, \Delta y,\, \Delta z) = \big(\text{Max}_x - \text{Min}_x,\ \text{Max}_y - \text{Min}_y,\ \text{Max}_z - \text{Min}_z\big) \tag{3}$$

$$\mathbf{o} = (\text{Min}_x,\, \text{Min}_y,\, \text{Min}_z) \tag{4}$$

Rather than allocating a cubic $N \times N \times N$ grid and mapping only the dominant axis to its full extent (which would leave voxels unused along the shorter axes), we allocate a *proportional* grid that assigns each axis exactly as many voxels as its share of the dominant extent warrants:

$$N_a = \left\lfloor N \cdot \frac{\Delta a}{\max(\Delta x,\, \Delta y,\, \Delta z)} \right\rfloor, \qquad a \in \{x,\, y,\, z\} \tag{5}$$

This guarantees that every allocated voxel maps to an occupied region of the scene, making full use of the available memory budget. Isotropy is still satisfied: all three axes share the same voxel edge length

$$\delta = \frac{\max(\Delta x,\, \Delta y,\, \Delta z)}{N} \tag{6}$$

because the extra voxels that would have been wasted in a cubic allocation are instead redistributed to match the scene proportions. The transformation from world coordinates to voxel coordinates is therefore:

$$(x',\, y',\, z') = \left\lfloor \frac{(x,\, y,\, z) - \mathbf{o}}{\delta} \right\rfloor \tag{7}$$

where $\lfloor \cdot \rfloor$ denotes the component-wise floor operation, mapping each continuous world-space position to its enclosing voxel index in $[0,\, N_x) \times [0,\, N_y) \times [0,\, N_z)$.

One consequence of this design is that the Jump Flood Algorithm (JFA) used to build the SDF volume operates in voxel space, where the grid is uniform but the underlying world-space distances are not. The SDF values stored in the volume represent distances in voxel units rather than world units, so they are consumed consistently in voxel space by the raymarcher. This makes the SDF a valid (though not necessarily tight) lower bound on the nearest occupied voxel along any ray, preserving correctness at the cost of some raymarching efficiency on scenes with a strongly non-cubic AABB.

5.1.2 Voxelization algorithm

The voxelization pass converts continuous scene geometry into a discrete 3D volume through a single GPU draw call that routes every scene triangle through a three-stage pipeline: a vertex shader that folds geometry to world space, a geometry shader that selects the optimal projection axis per triangle, and a fragment shader that writes into the 3D image via `imageStore`[7].

[7] *imageStore* is a special shader function that has the ability to write directly to a texture inside the shader at random locations per invocation.

World-space vertex transform. Each vertex position $\mathbf{p} \in \mathbb{R}^3$ is transformed from object space to world space by the per-instance model matrix $M \in \mathbb{R}^{4\times4}$:

$$\mathbf{p}_w = M\,\tilde{\mathbf{p}}, \qquad \tilde{\mathbf{p}} = (p_x, p_y, p_z, 1)^\top \tag{8}$$

The surface normal is transformed by using the inverse transpose of the matrix[8] and then renormalised:

$$\hat{\mathbf{n}}_w = \frac{(M^{-1})^\top\,\hat{\mathbf{n}}}{\left\|(M^{-1})^\top\,\hat{\mathbf{n}}\right\|} \tag{9}$$

Both $\mathbf{p}_w$ and $\hat{\mathbf{n}}_w$ are forwarded as interpolated outputs to the geometry stage; `gl_Position` is set using the voxel view-projection matrix P_v (an orthographic projection aligned to the volume extents) so that the rasterizer covers the correct screen-space footprint before the geometry shader overrides it.

Dominant-axis projection (geometry shader). A triangle $\mathcal{T} = \{\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\}$ has a geometric normal proportional to the cross product of its edges:

$$\mathbf{N} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0) \tag{10}$$

The *dominant axis* k^* maximises the projected area of $\mathcal{T}$ onto the corresponding coordinate plane:

$$k^* = \underset{k\in\{x,y,z\}}{\arg\max}\,|N_k| \tag{11}$$

Projecting along any other axis would foreshorten the triangle, potentially reducing its rasterized footprint below one pixel and causing voxels to be missed entirely.

Each world-space vertex is first mapped to the normalized voxel coordinate $\mathbf{u} \in [0,1]^3$ via the AABB mapping established in the previous section:

$$\mathbf{u}(\mathbf{p}_w) = \frac{\mathbf{p}_w - \mathbf{o}}{\mathbf{s}}, \qquad \mathbf{s} = \mathbf{p}_{\max} - \mathbf{p}_{\min} \tag{12}$$

where $\mathbf{o} = \mathbf{p}_{\min}$ is the scene AABB origin and $\mathbf{s}$ is the per-axis extent vector. The two coordinates of $\mathbf{u}$ orthogonal to k^* are then remapped from $[0,1]$ to the clip-space range $[-1,1]$ to form the `gl_Position` output that drives rasterization:

$$\begin{aligned} k^* = x \Rightarrow &\quad \texttt{pos} = (u_y,\, u_z) \\ k^* = y \Rightarrow &\quad \texttt{pos} = (u_x,\, u_z) \\ k^* = z \Rightarrow &\quad \texttt{pos} = (u_x,\, u_y) \\ \texttt{gl\_Position} = &\big(2\,\texttt{pos} - \mathbf{1},\ 0,\ 1\big) \end{aligned} \tag{13}$$

Setting the clip-space depth to 0 and $w = 1$ suppresses depth testing, ensuring every rasterized fragment reaches the fragment stage regardless of its depth relative to other geometry. The world-space position $\mathbf{p}_w$ is forwarded as a pass-through output so the fragment shader can recover the full 3D location without recomputing it from clip coordinates.

[8] Transforming normals directly by the model matrix produces incorrect results under non-uniform scaling. The correct normal matrix is the inverse transpose of the upper-left 3×3 submatrix of the model-to-camera transform (Lengyel, 2011, Ch. 1)

Voxel write (fragment shader). For each rasterized fragment the fragment shader receives the interpolated world-space position $\mathbf{p}_w$ and recovers the integer voxel index by applying Equation (12) followed by a component-wise floor:

$$\mathbf{i} = \lfloor \mathbf{u}(\mathbf{p}_w) \cdot \mathbf{r} \rfloor, \qquad \mathbf{r} = (N_x,\ N_y,\ N_z) \tag{14}$$

A bounds check discards any fragment whose normalized coordinate falls outside $[0, 1)^3$ (fragments that escaped the scene AABB due to floating-point imprecision or geometry extending outside the tracked volume). The albedo colour $\mathbf{c} \in [0, 1]^3$ is sampled from the material texture atlas, the surface normal $\hat{\mathbf{n}}_w$ is compressed into a single `uint8` value (encoding strategy described below) η, and the result is atomically written to the 3D albedo image:

$$\texttt{voxelAlbedo}[\mathbf{i}] \leftarrow \big(c_r,\ c_g,\ c_b,\ \eta\big) \tag{15}$$

Because the Vulkan `imageStore` operation is not guaranteed to be atomic for `rgba8` images and multiple triangles may project to the same voxel, this leads to undefined behaviour during runtime when multiple instances try to write to the same texel. In practice this is acceptable because all triangles sharing a voxel are spatially close and carry similar material properties; the resulting albedo error is perceptually negligible.

5.1.3 Normal Encoding

As explained before, surface normals are packed into the alpha channel of the voxel albedo volume alongside the colour albedo of each voxel. A zero value in the alpha channel serves as a *presence bit*, allowing a rapid solid-voxel test without a separate texture fetch. The remaining 255 non-zero values (levels 1–255) are used to encode the normal direction.

Encoding strategies. Several strategies of increasing fidelity are possible for distributing directions over the unit sphere S^2 within a fixed number of quantisation levels (Figure 3):

- **6-direction (axis-aligned):** The normal is snapped to the nearest face of the unit cube, yielding six possible directions $\mathbf{n} \in \{(\pm 1, 0, 0),\ (0, \pm 1, 0),\ (0, 0, \pm 1)\}$.
- **26-direction (face, edge, and corner):** The normal is snapped to the nearest vertex of the $3 \times 3 \times 3$ integer lattice with the origin excluded, giving 26 directions $\mathbf{n} \in \{a, b, c\}^3 \setminus \{\mathbf{0}\}$, $a, b, c \in \{-1, 0, 1\}$, each subsequently normalised.
- **Octahedral:** A bijective mapping between S^2 and the unit square distributes all 255 available levels uniformly over the sphere, giving substantially higher angular resolution at a modest decoding cost.

Octahedral encoding. The octahedral scheme projects a unit normal $\mathbf{n} = (n_x, n_y, n_z)$ onto the ℓ^1 unit sphere via

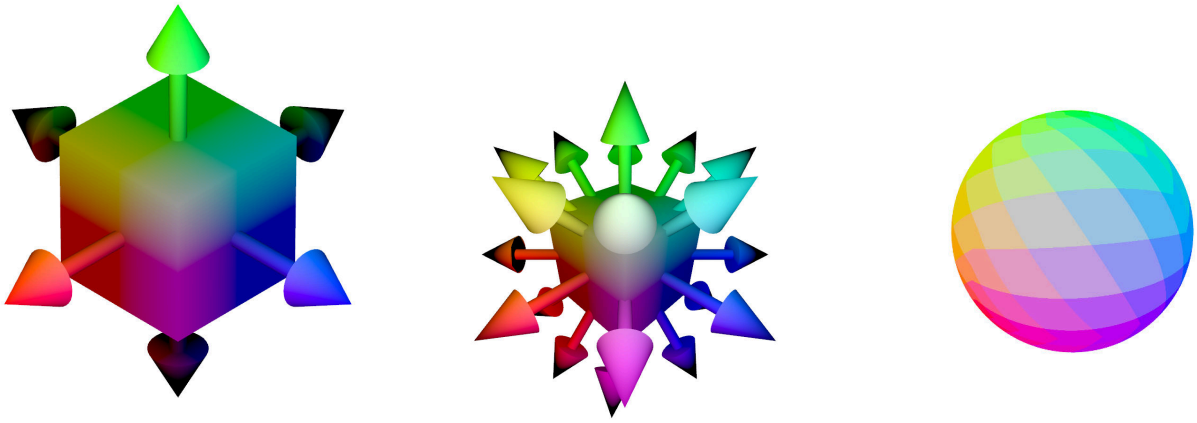
$$\mathbf{p} = \frac{(n_x,\ n_y)}{|n_x| + |n_y| + |n_z|} \ \in\ [-1, 1]^2. \tag{16}$$

When $n_z < 0$, the lower hemisphere is folded into the square by

$$\mathbf{p} \ \leftarrow\ \big(1 - |p_y|,\ 1 - |p_x|\big) \odot \operatorname{sign}(\mathbf{p}), \tag{17}$$

where $\odot$ denotes component-wise multiplication. The two components are then quantised independently to 4-bit unsigned integers

$$q_x = \operatorname{round}\big(\tfrac{p_x+1}{2} \cdot 15\big), \qquad q_y = \operatorname{round}\Big(\tfrac{p_y+1}{2} \cdot 15\Big), \tag{18}$$



(a) 1 Axis normal, the arrows represent all possible normal values that can be encoded.

(b) 3 Axis normals, the arrows represent all possible normal values that can be encoded.

(c) Octahedral normal encoding, the different color values around the surface of the screen represent the normal values in the encoding.

Figure 3: Illustration of the different normal encoding strategies proposed.

and packed into a single byte

$$e = (q_x \ll 4) \mid q_y, \qquad e \in [1,\, 255], \tag{19}$$

where $e = 0$ is reserved for the presence bit and any zero result is remapped to 1. The value is stored as $e/255 \in (0, 1]$ in the UNORM channel.

Decoding reverses the quantisation

$$p_x = \tfrac{q_x}{15} \cdot 2 - 1, \qquad p_y = \tfrac{q_y}{15} \cdot 2 - 1, \tag{20}$$

reconstructs the z component as $n_z = 1 - |p_x| - |p_y|$, applies the hemisphere unwrap when $n_z < 0$

$$(p_x,\, p_y) \;\leftarrow\; \big(1 - |p_y|,\; 1 - |p_x|\big) \odot \operatorname{sign}(p_x, p_y), \tag{21}$$

and returns $\mathbf{n} = \operatorname{normalize}(p_x,\, p_y,\, n_z)$.

Strategy selection. The 6- and 26-direction schemes require no arithmetic at decode time and trivially preserve the presence bit, but their coarse quantisation introduces visible banding in the indirect illumination. The octahedral scheme makes full use of all 255 levels and distributes quantisation error uniformly over S^2, at the cost of a small number of additional ALU instructions and a reserved zero level. An experimental comparison will be done to test if there is a visual difference between the 26-direction and octahedral encodings for the diffuse GI use case.

5.2. Signed Distance Field Computation

For computing the signed distance field, as explained before, we make use of the *Jump Flood Algorithm* (JFA) which is a standard $O(\log n)$ algorithm that produces a SDF texture (example in 2D Figure 4 of execution), we run this algorithm in the GPU directly[9].

JFA algorithm works by running several consecutive passes in a double buffer scratch texture memory, similar to how double pass gaussian blur is orchestrated. The number of passes is computed given the dimensions of the voxel volume as:

[9]For implementation reference it's advised to check Rong and Tan (2006)

$$\lfloor \log_2(\max(\text{dim.x}, \text{dim.y}, \text{dim.z})) \rfloor + 1$$

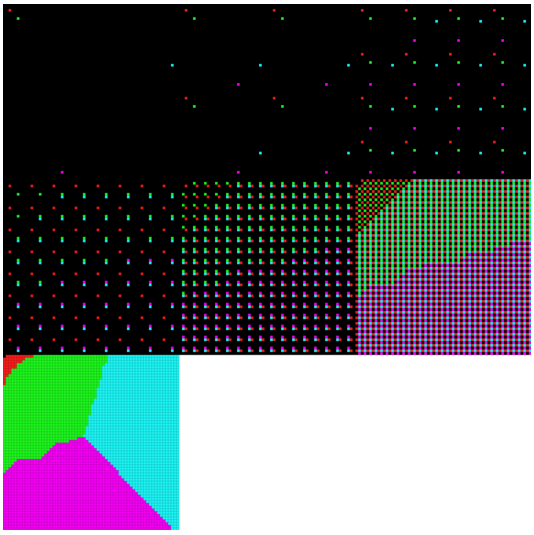


Figure 4: JFA execution on 2D grid. For visualization purposes the voronoi map is shown, JFA can be used to compute it in a equivalent manner, instead of displaying the distance to the seed from the current pixel we display the ID of the seed the pixel belongs to, which is guaranteed to be the closest. Mathematically speaking this is equivalent to the voronoi map, a simple way to check the correctness of the JFA algorithm each frame represents an iteration of the JFA algorithm, initial frame marks the seed with colors.

A final step is added to perform the final write into the SDF volume which is a single channel 8 bit texture, we call this the **SDF Map**.

The final result for the voxelization maps can be seen in (Figure 5).

5.3. Solid Voxel List

For computing the **SolidVoxelList** we are going to emit into a SSBO all necessary information to process voxels in the compute shader that builds the radiosity matrix.

As mentioned before, the rationale behind doing this is to avoid processing empty voxels as patches during the build matrix process and to decouple the following strategies from the voxelization data structures used (Figure 6):

In the compute shader, we make an invocation for each voxel in the maps, then check if the voxel is a solid voxel in the **AlbedoMap**, and then emit into the voxel list: *position*, *normal*, and *albedo*.

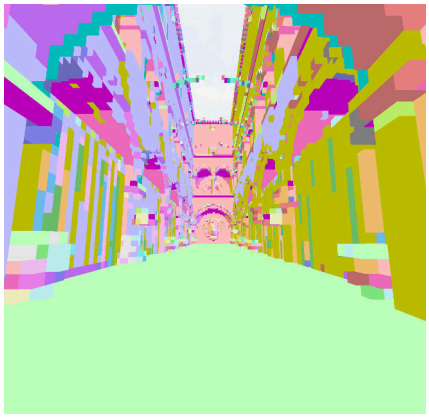
In the implementation the albedo is written into 4 component 32 bit channel. For most use cases this would be unnecessary since albedo is in texture maps is usually coded in RGBA8[10] representation packed into one single uint.

Given that the voxel grid resolution used in testing is typically under 512, the normal map is encoded in a single 8-bit channel, and the albedo is usually stored in RGBA8 format, there are more compact representation methods available that could significantly reduce memory bandwidth.

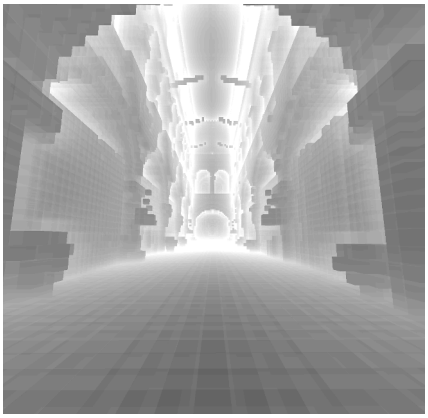
Additionally, a third voxel map is created called **Emissive Map** which will hold the emissive component gathered by the emissive textures of the scene. This texture serves as general purpose way to assign particular values of emissive to voxels, we can use it to write emission from a texture during voxelization pass for instance. It's an optional map since its only required for taking into account emission based lighting, like textures and shaders.

[10]We refer to RGBA8 a texture that is encoded using 8 bits per channel with 4 channels, Red, Green, Blue and Alpha

(a) Albedo map, represents in 3D the resuling albedo from the voxelization stage



(b) Normal map, each value represents one different normal on 3 Axis normal codification



(c) SDF map, each value represents the accumulated inverse of SDF values across one single ray through the SDF volume

Figure 5: Voxel maps visualization for a same scene

5.4. Matrix computation

In this section I will explain the code for computing the view factor matrix. This, as explained before, it's done in a compute shader, when computing the view factor matrix (Figure 7) we need to first allocate memory for the block sparse representation of the matrix in the GPU in the form of a SSBO, for doing so we need to know the exact amount of solid voxels.

From an implementation point of view in Vulkan, not only we have to emit a barrier[11] between the stages, but submit the commands so they can be processed, and read-back the result, this introduces a small overhead of a $CPU \rightarrow GPU \rightarrow CPU$ roundtrip communication.

Alternatively, if one does not desire to pay such cost, the memory allocated for the matrix can be pre-allocated to have a maximum size regardless of actual voxel count, which is usually the right choice when dealing with recomputations of the matrix, since reallocating a buffer that large can cause memory fragmentation especially if it's sub-allocated from a larger pool.

For computing the per-voxel weights, rays are traced stochastically over the hemisphere; the stochastic selection naturally biases toward high-form-factor neighbours, as detailed below.

Additionally, in my implementation we can choose between using the **SDF Map** to speedup the raytracer (essentially becoming a raymarcher) or to use the traditional DDA[12]. I have executed multiple experiments to compare result in both terms of speed and quality, results are shown below.

5.5. Relaxation parameter

A small caveat of this procedure is that multiple rays can end up colliding to the same voxel if the distance between them is close enough, for this reason we can end up having repeated entries inside the factor matrix, this is problematic, since this can make up the full row of weights not sum up to 1, but exceeds it, which makes the algorithm violate energy conservation and not converging.

For this reason I have introduced the relaxation parameter, we can manually tune the total weight of the row to account for this problem. This solution is very simple and practical and I have shown it both works and produces realistic result.

[11] A barrier in the context of a compute shader is a synchronization point between all threads in a subgroup.

[12] DDA is a fast and correct algorithm to perform raytracing on a voxel volume, it's recommended to read Amanatides and Woo (1987) an example implementation can be found at rfb39ca4 (2013)

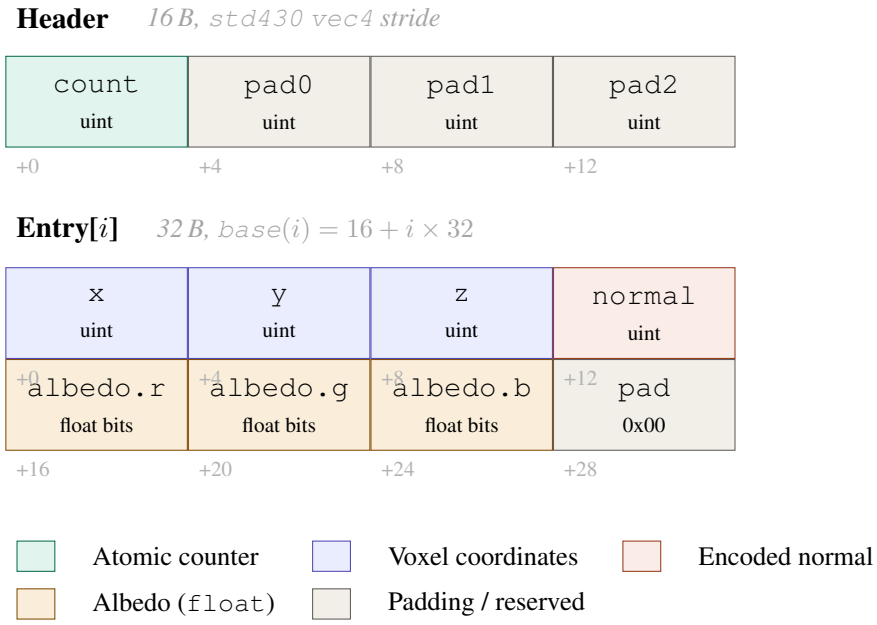


Figure 6: Linear memory layout of the `SolidVoxelList` SSBO. The header occupies the first 16 bytes, padded to a `vec4`. Each subsequent entry encodes the voxel's grid coordinates, surface normal, and albedo as a contiguous block of memory.

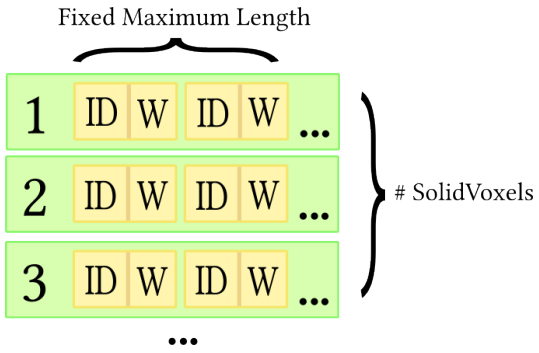


Figure 7: Matrix memory layout representation, each row is a voxel entry, each row contains in the form of ID-Weight pairs the radiosity view factors

The alternative would be much more computationally expensive, we would have to sort the row, remove the entries that are unique, and renormalize the weights, which would have made the procedure of generating the matrix much more expensive.

5.6. Form-Factor Matrix Construction

The transport matrix $\mathbf{A} = \rho \cdot F$ is built stochastically on the GPU. For each solid voxel i, the shader fires N cosine-weighted rays sampled from a jittered Hammersley[13] sequence over the hemisphere defined by its surface normal. Each ray that hits a visible solid voxel j contributes an entry to the sparse matrix with weight

$$w_{ij} = \frac{1}{N}, \tag{22}$$

which is the unbiased Monte-Carlo estimate of the geometric form factor F_{ij}.

[13]The Hammersley sequence is a low-discrepancy Monte Carlo point set constructed by pairing the Van der Corput radical-inverse function in base 2 with a uniform stratification over N samples, yielding superior hemisphere coverage compared to pseudo-random sampling Hammersley (1960); Wong et al. (1997). For hemisphere ray generation, each sample $(u_1, u_2) \in [0, 1)^2$ is mapped to a cosine-weighted direction via $\theta = \arccos\sqrt{1 - u_1}$, $\phi = 2\pi u_2$.

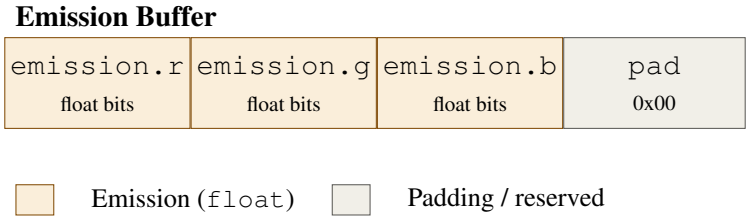


Figure 8: Emission Buffer memory layout

Here, $\rho \in [0, 1]$ is the diffuse albedo of the destination voxel (RGB channels of `voxelAlbedo`), and F_{ij} is the fraction of energy leaving patch i that reaches patch j, accounting for orientation and visibility. Visibility is evaluated by DDA traversal on the integer voxel grid, accepting a hit only when the ray is incident on the front face ($\hat{\mathbf{d}} \cdot \hat{\mathbf{n}}_j < 0$).

Low Probability - High Energy Paths : The scheme proposed to efficiently query which patches view factors are more relevant for each matrix row has a strong problem that is left unresolved, that is, *Low Probability - High Energy* paths. I refer to these paths as the light paths that hit a very small part of the scene but are extremely energetic, this produces diffuse bounces through the scene that represent an outlier in the emissive vector. Because we are not dealing with the full matrix and some view factors that would have been close to 0 are actually treated as 0 we will be missing an important amount of energy bounces in the scene, this is not problematic for the average case since as said before this is usually not a problem because radiance from a patch can be transmitted from multiple path bounces. However this is not the case when dealing with this type of bounces, since the energy received is very strong, even if we are dealing with a very small view factor weight the resulting radiance is also strong, making it relevant. Because some voxels might not have similar row entries to their neighbors it's possible to see noise artifacts being generated due to a discontinuity of the matrix. This will be tested during experimentation.

5.7. Light Gathering

In this stage we must run a compute kernel that runs for every solid voxel on the **SolidVoxelList** and gathers the light from the scene and emit it into an **EmissionBuffer** (Memory layout at Figure 8).

The methods for gathering the light depend on the type of light source we are dealing with:

- **Spotlights And Sunlights**: We can simply take the light caster information and perform shadowmapping on the voxels, the same way we would do on a shader, and compute the direct light and write it as emissive directly.
- **Omnilights**: For omnilights we would need to actually perform shadowmapping on the traditional approach of cubemap based shadowmapping. This is not implemented because of time constraints but the procedure would be the same as Spotlights.
- **Emissive Textures**: For emissive textures the straightforward approach is to read from the **Emissive Map** we have created before.

For other types of lights (IBL [14]), it's recommended to use the **EmissiveMap** and write directly to it, either from a fragment or a vertex shader during forward rendering.

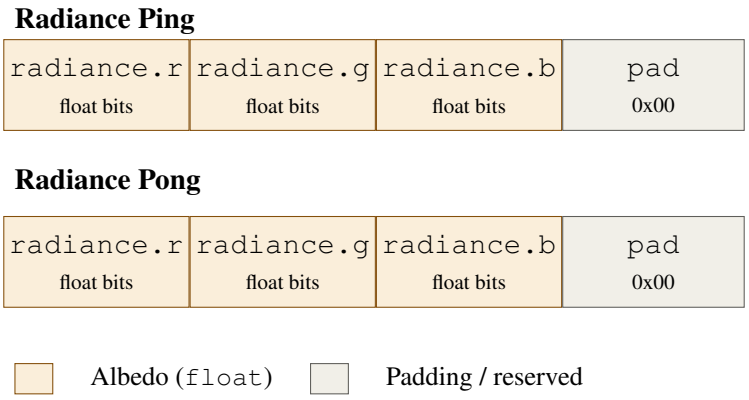


Figure 9: Radiance Buffers memory layout

5.8. Relight iteration

For computing the scene relight we run a compute shader in the GPU, taking the matrix, the **Solid Voxel List**, and the past result of the radiance.

Since we are computing the power iteration through the Jacobi method we must allocate a double buffer radiance buffer (we call it **Radiance Buffer** Figure 9) to perform the computation, on a CPU the Gauss-Seidel [15] formulation converges faster (even if we are at the same time reading and writing from the same buffer, and thus, the execution will depend on which entries are processed first, the value converged remains the same at a few iterations).

There is no efficient way however to perform Gauss-Seidel power iteration on the GPU since it requires writing and reading from the same buffer which is very problematic. For our purposes the Jacobi power iteration is enough.

The code implemented is a straightforward *SpMV* matrix vector computation on the GPU making use of the specified mathematical formulation and the chosen memory layout.

There are 2 notable remarks however.

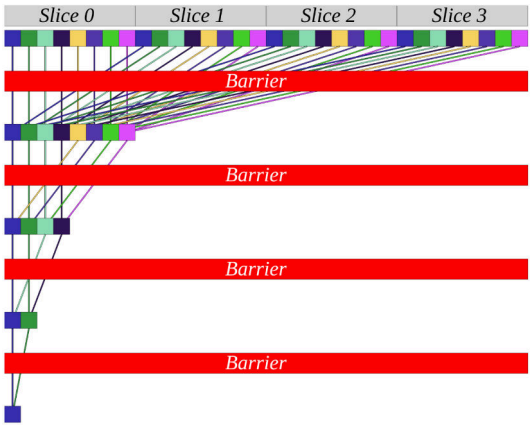


Figure 10: Each thread is represented by a color, every line of work blocks represent an item in the view factor matrix row for a particular patch, the kernel performs tree like based sum with intermediate barriers.

For speeding up the reading of the *SpMV* we have to make each thread read a different entry for all the

[14] IBL stands for Image Based Lighting, this type of lighting consists in sampling a blurred envmap given normal coordinates of a surface. Its primary use case is approximating ambient light in open areas

[15] Gauss-Seidel is identical to Jacobi but updates $\mathbf{x}$ in-place, so newer components are immediately used in the same iteration, typically halving the iterations to convergence at the cost of parallelism Saad (2003).

entries in a same row. That is, a single workgroup[16] is responsible for computing a single dot product of the SpMV iteration.

The size of the row must be divisible by the size of the workgroup, on the GPU that this was tested the optimal size for a workgroup that matches the threadcount on a wavefront[17] was 64.

In order to perform the *SpMV* product, we divide the total work in slices, each thread computes a value of each slice that matches its index. We store the accumulated values on shared memory and then perform tree based sum to reduce the result as it shown in the Figure 10.

Finally the first thread writes to the result buffer.

5.9. Light Scattering

We now have the result in a SSBO buffer that has the radiance of all solid voxels for the next sampling iteration, however for efficient sampling in the fragment shader we must write them back to a 3D texture, we call this the **RadianceTexture** and has RGBA32 format.

Doing this is quite straightforward we simply have to read from the **RadianceBuffer** (the correct one for this iteration) and **SolidVoxelList**, from this we issue a compute invocation per solid voxel on the list and write to the 3D texture indexed by the voxel position on the list and write the contents in the **RadianceBuffer**.

Important to note that we have to clean the **RadianceTexture** at the beginning of every iteration.

5.9.1 Shadow Hard Edges

A small thing to notice, we are going to write back the **RadianceBuffer** which contains embedded the direct light we gathered in previous steps, this is actually physically incorrect since in the forward pass we will be adding it again so we would have the direct light contribution two times stronger than actually is.

If we were to render with the current configuration (sampling the **RadianceTexture** in the forward pass) we would see that around the shadowmap borders we are visualizing a low resolution hard edge, this is the direct light values embedded in the **RadianceTexture**.

In order to fix this we simply subtract in this stage the content from **RadianceBuffer** by the values we gathered in the **EmissionBuffer**.

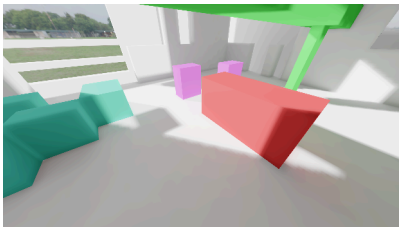
$$\mathbf{L}_{\text{vox}}(\mathbf{x}) \;=\; \mathbf{R}(\mathbf{x}) \;-\; \lambda\,\mathbf{E}(\mathbf{x}) \tag{23}$$

where:

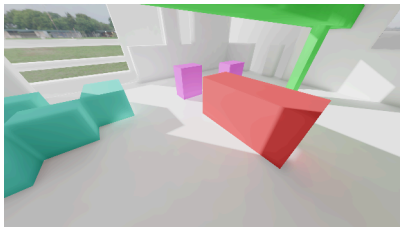
- $\mathbf{L}_{\text{vox}}(\mathbf{x})$ is the radiance written to the voxel volume at position $\mathbf{x}$.
- $\mathbf{R}(\mathbf{x})$ is the accumulated radiance at voxel $\mathbf{x}$ from the radiosity solver.
- $\mathbf{E}(\mathbf{x})$ is the emission stored at voxel $\mathbf{x}$, masked to the RGB channels.
- λ is the normalization factor previously introduced.

[16] A workgroup is a set of threads dispatched together to a single Compute Unit, sharing `LDS` (Local Data Share) and synchronizable via `barrier()`. On RDNA3 a workgroup contains one or more wavefronts of 64 threads each.

[17] A wavefront (called warp in NVIDIA GPUs) is the fundamental unit of GPU execution scheduling. Its a group of threads that executes simultaneously and share same instruction register



(a) Without edge correction



(b) With edge correction

Figure 11: Comparison of edge correction, focus on the boundaries of the shadows and their blocky sharpness.

However, notice we are losing something important that could be useful, if direct light values are directly embedded in the **RadianceTexture**, when we compute the specular contribution using the raytraced algorithm we are going to sample the direct light at the raycasted voxel through the reflection. This means that direct light can now be visible across the reflections and mirrors which is a very desirable result.

If we are going to take this route however we will have to face a compromise and accept that direct light contribution for all the scene will be scaled, this however is hard to notice if proper tonemapper is used.

We also need to write to the alpha channel *1.0f*, this will be important later to perform pre-multiplied alpha during sampling.

A visualization of the effect can be seen with (Figures 11).

5.10. Sampling The final Result

In this section I will describe briefly how are the results sampled in the forward shader for giving a complete look in the algorithm.

When sampling the result for diffuse GI we do the following: We have to sample the **RadianceTexture** indexed using the *fragmentPosition* in the voxel volume space coordinates, however this is not straightforward since it's possible that the fragment position lands on a neighbouring voxel that is actually non solid and thus have a black result.

For avoiding this we can make use of linear filtering[18] with pre-multiplied alpha [19], the technique is simple, since we are writing into the **RadianceTexture** a presence float in the alpha channel, when sampling in linear mode we are going to have in the alpha channel a blend between 0 and 1 for the values that are partially missing when doing linear sampling.

For correcting the sampled value, which is tinted by black, we can simply divide by the value in the alpha channel. This is a known technique to apply linear sampling in regions of textures that have borders in 2D, same idea can be applied here.

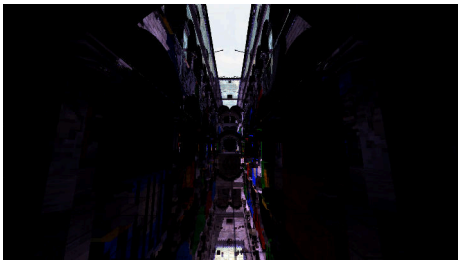
$$\mathbf{c}_{\text{GI}} = \frac{\mathbf{L}_{RGB}}{\alpha} \tag{24}$$

Additionally for further improving the quality and reducing the noise we can sample the 3D volume using a blur kernel, so we can further smooth the sampled radiance. This is partially technically incorrect since we can be sampling radiance values from voxels that don't share similar rows in the view factor matrix, however since Radiosity radiance has very low frequency by nature this types of artifacts are rare to appear or even notice.

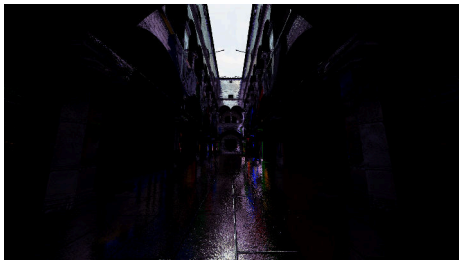
We instead gain the ability to smooth the noise made by low probability but high energy paths (In the experimentation section result can be seen at Figures 34 and 35).

[18] Linear filtering (often referred to as Texture filtering) is a method used to smooth textures by blending adjacent pixels using linear interpolations that blend the color in the neighbouring pixels of a sample coordinate

[19] A good reading explaining why this method works for getting rid of this artifact is Quilez (2020)

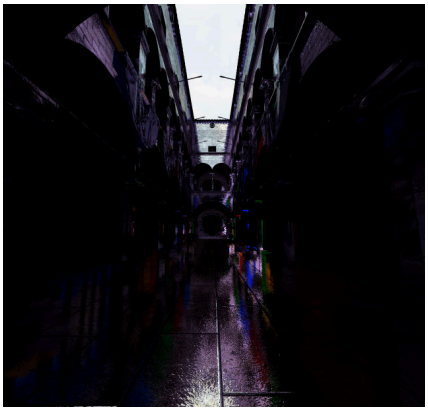


(a) Specular rays result only



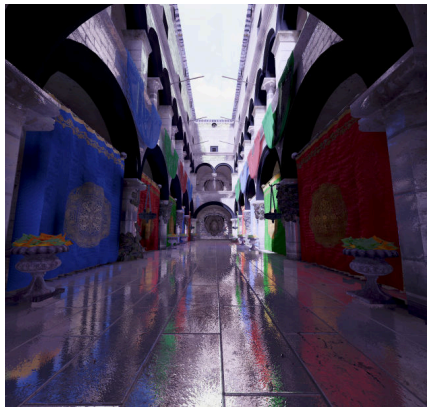
(b) Specular raytracing with PBR coefficients

Figure 12: Specular Light Tracing performed in the fragment shader with the **RadianceTexture**.



(a) Specular GI computed in the fragment shader from **RadianceTexture**

(b) Diffuse GI sampled in the fragment shader from **RadianceTexture**



(c) Combined result of direct lighting diffuse and specular GI with PBR coefficients

Figure 13: Final rendering combining specular GI with diffuse GI.

5.11. Specular Light Tracing

As mentioned earlier, and as a bonus for the whole GI algorithmic system, I introduce alongside the main core radiosity algorithm, a small DDA ray tracer inside the forward shader that is able to cast rays into the scene traversing the radiance voxel volume to approximate specular paths from the diffuse result.

The core idea is very simple and similar to standard ray tracing, for the given specular cone on a particular fragment in the forward shader we stochastically trace rays into the scene in the domain of the specular cone, that is the computed normal per fragment plus a random generated jitter influenced by the radius of the specular cone, which is computed using the PBR roughness estimation of the surface, then we average the interaction of each sampled specular value obtained from the hitting surface radiance of each ray multiplied taking into account PBR coefficients (Figure 12).

In order to generate stochastically the rays we take as seed the fragment world position and the ray index.

As a side note, it is to be noted that this method can cause a great amount of specular aliasing for high frequency variant normals along a surface.

I recommend to soften the normals for a proper implementation or directly using the geometric normal, however this is left out of the scope of the project since involves careful heuristics and antialiasing methods and rendering systems may use different techniques to solve the problem (The final result of the composite method can be seen in Figure 13).

5.12. Specular Writeback

As bonus, given the computed specular value, we can write back this value into the main diffuse algorithm by using the **Emission Map** using an *imageStore* operation from the fragment shader.

However this operation is very costly. We cannot simply issue an *imageStore* operation in shading time to the *Emissive Texture*, the reason for this is memory contention. In a fragment shader one single thread is destined to process a single fragment and if every single of them issues a texture write operation we will have data races, since multiple threads might write to the same voxel of the volume. For that reason we need to compute a mask that tells each thread if they will be the one writing to the closest texel they have in voxel space from their position in world space, ideally a maximum of one per texel.

However we have to remember the threads are executed in subgroups too, and all threads are forced to execute the two branches of an if instruction if at least two of them are diverging (Not to mention that fragment shaders must also execute in packs of 2x2 pixels to compute fast inter thread lookup functions like, `dFdx` so they must execute both branches regardless).

That's why this approach though mitigating some overhead caused by the amount of memory writes is not optimal, a better approach from a hardware point of view would be to write directly to another color attachment[20] and read it using a standard compute shader, which is capable of processing the writes much faster.

However due to time limit constraints the latter approach was not taken.

To proceed with the mask strategy the process is the following, we compute a mask per fragment that tells which fragment will do that particular job.

The computation of the mask takes into account the position of the fragment in world coordinates, and processes whether it's heuristically the closest to the center of a voxel in the *Emissive Texture*.

The mask is computed as follows. Let $\mathbf{p}_w$ be the fragment world position, $\mathbf{c}$ the camera position, and $\hat{n}$ the surface normal.

Camera distance weighting. An attenuated distance factor is computed as:

$$d = \|\mathbf{c} - \mathbf{p}_w\|^{\alpha}, \quad \alpha < 1 \tag{25}$$

with a sub-linear exponent α to soften the falloff with distance.

Voxel-space projection. The fragment is mapped into the voxel volume $\mathbf{v} = \texttt{world2volume}(\mathbf{p}_w)$, and its fractional part $\mathbf{g} = \text{fract}(\mathbf{v}) \in [0, 1)^3$ gives its position within the voxel cell.

Normal-weighted centering. Each axis of $\mathbf{g}$ is blended toward 0.5 (the voxel center) proportionally to the surface normal alignment with that axis:

$$g_i' = \text{mix}(g_i,\ 0.5,\ |\hat{n}_i|), \quad i \in \{x, y, z\} \tag{26}$$

A face perpendicular to axis i ($|\hat{n}_i| \approx 1$) collapses all its fragments to the same voxel-center coordinate along that axis.

[20] A color attachment is the texture that is bound as rendering target to a given pipeline, the output of the image, when using Vulkan and other APIs is possible to output to multiple targets at the same time even if they have different formats. In modern hardware, the memory that color attachments are backed with can be shared memory of the cores instead of VRAM which speedups subsequent reads from other passes that use those attachments, this can be achieved using subpasses which is a feature of Vulkan (Many drivers however, are able to detect this automatically even if this is not specified)

Proximity score. The distance from the blended grid point to the voxel center is:

$$s = \|\mathbf{g}' - \mathbf{0.5}\| \tag{27}$$

Normalized by camera distance and clamped to $[0, 1]$:

$$\hat{s} = \text{clamp}\left(1 - \frac{k \cdot s}{d},\ 0,\ 1\right) \tag{28}$$

where k is a tuning constant that maps the proximity-to-distance ratio into a workable range (other thresholding strategies are equally valid here).

Mask condition. A fragment is selected as the writeback representative when $\hat{s} \geq \tau$ for some threshold $\tau \in (0, 1)$, and only selected fragments issue the `imageStore` to the emission cache at $\lfloor \mathbf{v} \rfloor$.

The intuition is that $\hat{s}$ is high only for fragments simultaneously close to a voxel center and relatively near the camera (distant surfaces project many fragments per voxel, increasing contention). The threshold τ then heuristically selects at most one representative per voxel cell under normal viewing conditions. Result of the mask can be seen in (Figure 14).

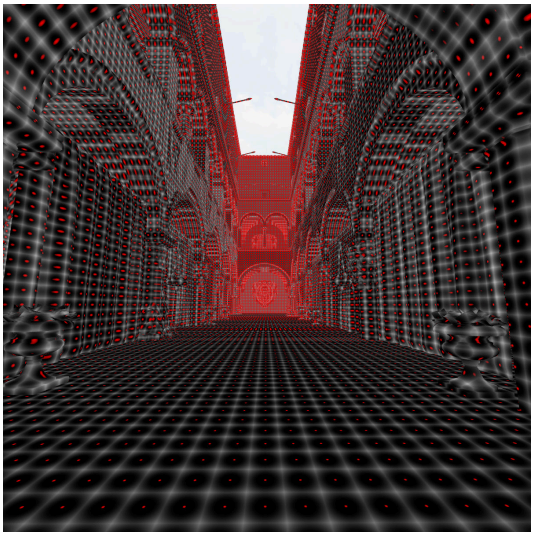


Figure 14: Visualization of debug sample points, the red points represent the fragments that will do the writeback to the **Emission Map** during the forward shader with the specular traced data and the white lines represent the voxel seams, this Illustration serves to prove that the sample points actually land in the middle of the voxels seams, the sample points size is exaggerated for display purposes, it's intended (since it's a mask) to be as minimal as possible so optimally few threads issue the write

However it is to be noted that although this works for faces that are aligned with the voxel volume axis it can generate out of grid positions for those that are not aligned.

5.13. Brute force approach

Additionally and as reference, the voxel maps obtained in the *Voxelization* stage can be used to compute the radiance volume using a brute force approach. This has been programmed into the implementation as a separate algorithm.

This algorithm works by running a compute shader that traces rays from each voxel in the volume stochastically, exactly the same way the rays are traced in the view factor computation stage, except that in this case the obtained result is directly used to compute the radiance in the volume doing implicitly the computation of *SpMV* iteration that we perform in the radiosity algorithm.

This serves as a rapid way to check the correctness and efficiency of the algorithm during testing.

5.14. Memory Consumption

The table below (Table 1) presents the total GPU memory consumption of the Global Illumination (GI) system for the Sponza scene at a voxel resolution of 256. It details the memory occupied by each component within the rendering system, including voxel volumes and matrix-based GI data, along with their respective formats and capacities.

Overall, the matrix GI subsystem dominates the memory usage, accounting for the vast majority of the total consumption.

Table 1: GPU memory consumption for the Sponza scene at voxel resolution N=256 (grid $256 \times 107 \times 157$, ≈4.3M voxels, 441,511 solid voxels, `maxHitsPerRow`= 384). Volume buffers are sized by voxel count; matrix buffers are sized by `maxSolidVoxels`= 2^{20} to allow GPU-side atomic allocation without stalls.

Subsystem	Buffer	Role	Format	Size (MB)	Capacity
Voxel Volumes	`voxelAlbedo`	Voxelized albedo + encoded normal	`R8G8B8A8_UNORM`	16.41	$256 \times 107 \times 157$ voxels
	`voxelRadiance`	Diffuse radiance cache (ping)	`R16G16B16A16_SFLOAT`	32.83	$256 \times 107 \times 157$ voxels
	`voxelRadiancePong`	Diffuse radiance cache (pong)	`R16G16B16A16_SFLOAT`	32.83	$256 \times 107 \times 157$ voxels
	`jfaScratch0`	JFA seed / SDF scratch volume	`R32_UINT`	16.41	$256 \times 107 \times 157$ voxels
	`jfaWork0`	JFA propagation working volume	`R16G16B16A16_SFLOAT`	32.83	$256 \times 107 \times 157$ voxels
	`reverseMap`	Voxel → solid-list index lookup	`R32_UINT`	16.41	$256 \times 107 \times 157$ voxels
			Volume subtotal	**147.72**	
Matrix GI	`solidList`	Solid voxel coords + albedo (8 × uint32)	`uint32[]`	32.00	2^{20} entries
	`matrixHits`	Sparse view-factor entries (col, weight)	`uint32[]`	3072.00	$2^{20} \times 384 \times 2$ entries
	`rowCounts`	Per-row hit counter (atomic)	`uint32[]`	4.00	2^{20} entries
	`radianceXBuf`	SpMV x vector (radiance/voxel)	`vec4[]`	16.00	2^{20} entries
	`emissionBuf`	Per-solid emission accumulator	`vec4[]`	16.00	2^{20} entries
			Matrix subtotal	**3140.00**	
			Grand total	**3287.72**	

Let $V = N_x N_y N_z$ be the total voxel count, s the number of solid (non-empty) voxels, $M_s = 2^{\lceil \log_2 s \rceil}$ the GPU solid-list capacity rounded to the next power of two, and R the per-row form-factor budget (`maxHitsPerRow`).

Volume buffers. Seven 3-D textures are allocated over the full grid: three at 8 B vox^{-1} (radiance ping/pong, JFA work volume) and other three at 4 B vox^{-1} (albedo, JFA seed, and reverseMap):

$$\mathcal{M}_{\text{vol}}(V) = 36\,V \quad \text{[bytes]} \tag{29}$$

Matrix GI buffers. All six buffers are allocated against M_s rather than s to allow lock-free atomic row assignment during parallel matrix construction. The solid list stores 8 `uint32` fields per entry (32 B) plus a 16 B header; each form-factor entry stores a column index and quantised weight (2 `uint32` = 8 B); row counters use 4 B; and three `vec4` arrays of 16 B each hold the Jacobi solution vector ping-pong pair and the per-solid emission cache:

$$\mathcal{M}_{\text{mat}}(M_s,\, R) = M_s\,(84 + 8R) + 16 \quad \text{[bytes]} \tag{30}$$

The dominant contribution is $8R\,M_s$ (`matrixHits`), which accounts for $\approx 97\%$ of $\mathcal{M}_{\text{mat}}$ at default parameters ($R = 384$).

Total.

$$\boxed{\mathcal{M}_{\text{total}}(V,\, M_s,\, R) = 36\,V + M_s\,(84 + 8R) + 16 \quad \text{[bytes]}} \tag{31}$$

Both terms scale as $\mathcal{O}(N^3)$; however $\mathcal{M}_{\text{mat}}$ carries the coefficient $8R$, making it the dominant cost in practice (97% of total at $N = 256$, $R = 384$). For the Sponza scene at $N = 256$ ($V = 4{,}300{,}544$, $s = 441{,}511$, $M_s = 2^{20}$, $R = 384$), equations (29)–(31) yield $\mathcal{M}_{\text{total}} = 3287.72$ MiB, matching the runtime measurement exactly.

6. Research Questions

These are the research questions that will be answered in the experimentation to test the algorithm quality:

ID	Research Question
RQ1	How does the result compare with a reference rendering engine, like **Cycles** and **EEVEE**?
RQ2	How does the result of the relight engine compare with the reference brute force approach?
RQ3	What is the mean computation time of the factor matrix on the GPU?
RQ4	How does voxel resolution affect on average both the factor matrix size and computation time?
RQ5	How does row size of factor matrix affect on average computation time and quality?
RQ6	How does the result change when using DDA or SDF for calculating the matrix coefficients?
RQ7	How does the result change when using 3 axis normals vs. flat normals, octahedral normals?
RQ8	What is the mean computation time for the relight iteration?
RQ9	What is the optimal relaxation factor to obtain a physically accurate result?
RQ10	How does blurring the sampled result affect the quality of the algorithm?
RQ11	How does the radiosity edge removal algorithm perform in both time and quality?
RQ12	How does specular writeback affect the result?

7. Experimentation

In this section I will describe the experiments to be made and dataset of scenes chosen to perform the tests.

7.1. Scenes used

For conducting the tests I have decided to use the following scenes, as well as the scene extent (Tables 2 3) :

- **Sponza Crytek**: This is a reference scene commonly used for showcasing real time Global Illumination algorithms (publicly available at Frank Meinl (2026)).
- **Box**: This is a custom scene that was made to test the program, it's similar to a Cornell Box (reference image at Figure 15, available through the repository of the project at Tejeda (2026c)), serves as the most basic case for testing the algorithm with a reasonable variety of geometry types (Thin geometry, walls, complex objects, and curves with variations in materials).
- **Abstract Park**: This is another custom scene that was made to test the program (reference image at Figure 16, available through the repository of the project at Tejeda (2026a)), the driving goal behind the design of this scene is to test the light bounces and reflection correctness against multiple flat surfaces, with and without reflections
- **Arena**: This is a desert-like scene/level (reference image at Figure 25b, available through sketchfab at Tejeda (2026b)). The reason to include the scene in the testing is because of its broad simple geometry, that spans for large zones, and it's cave system which makes the most appropriate case for testing the Radiosity algorithm bounces through the entrance of the caves.

7.2. System Specification

These are the specifications of the systems the experiments where conducted on (Table 4).

Table 2: Scene bounding box extents, proportional voxel grid ($N_x \times N_y \times N_z$), and isotropic voxel edge length $\delta = \max(\Delta x, \Delta y, \Delta z) \,/\, N$ at three values of N.

		$N = 128$		$N = 256$		$N = 512$	
Scene	$(\Delta x,\ \Delta y,\ \Delta z)$	Grid	δ	Grid	δ	Grid	δ
AbstractPark	$(45.52,\ 27.44,\ 45.52)$	$128 \times 77 \times 128$	0.3556	$256 \times 154 \times 256$	0.1778	$512 \times 308 \times 512$	0.0889
Arena	$(304.56,\ 79.18,\ 543.48)$	$71 \times 18 \times 128$	4.2459	$143 \times 37 \times 256$	2.1230	$286 \times 74 \times 512$	1.0615
Box	$(10.65,\ 10.65,\ 10.65)$	$128 \times 128 \times 128$	0.0832	$256 \times 256 \times 256$	0.0416	$512 \times 512 \times 512$	0.0208
Sponza	$(3720.85,\ 1555.88,\ 2288.23)$	$128 \times 53 \times 78$	29.069	$256 \times 107 \times 157$	14.535	$512 \times 214 \times 314$	7.267

Table 3: Scene geometry statistics.

Scene	**Vertices**	**Triangles**	**Solid Voxels (For 256^3 resolution)**
AbstractPark	4,080	3,820	693,691
Arena	7,004	5,187	45,290
Box	5,265	5,022	450,793
Sponza	184,406	262,267	441,511

7.3. Tuning parameters

The algorithm has the following tuning parameters that can be changed (And of which we are going to study how does rendering quality and performance changes when changing them, Table 5):

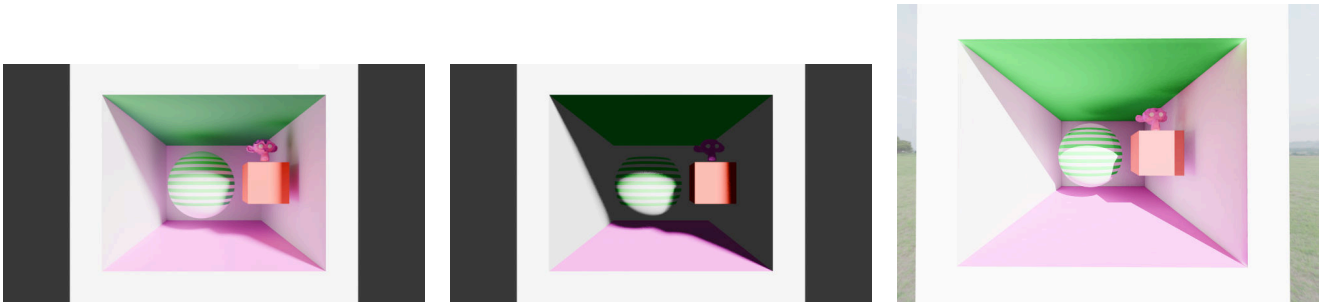
Table 4: Experimental Platform Specifications

Component	**Detail**	**Specification**
CPU	Model	AMD Ryzen 9 7900
	Cores	12C / 24T @ 4.34 GHz
GPU	Model	AMD Radeon RX 7700 XT
	VRAM	12 GB GDDR6
	Subgroup	64 lanes
	Timestamp	10 ns/tick
API	Vulkan	1.4.348
	Driver	RADV Mesa 26.1.1
OS	Platform	Arch Linux

Table 5: Experimental Parameters

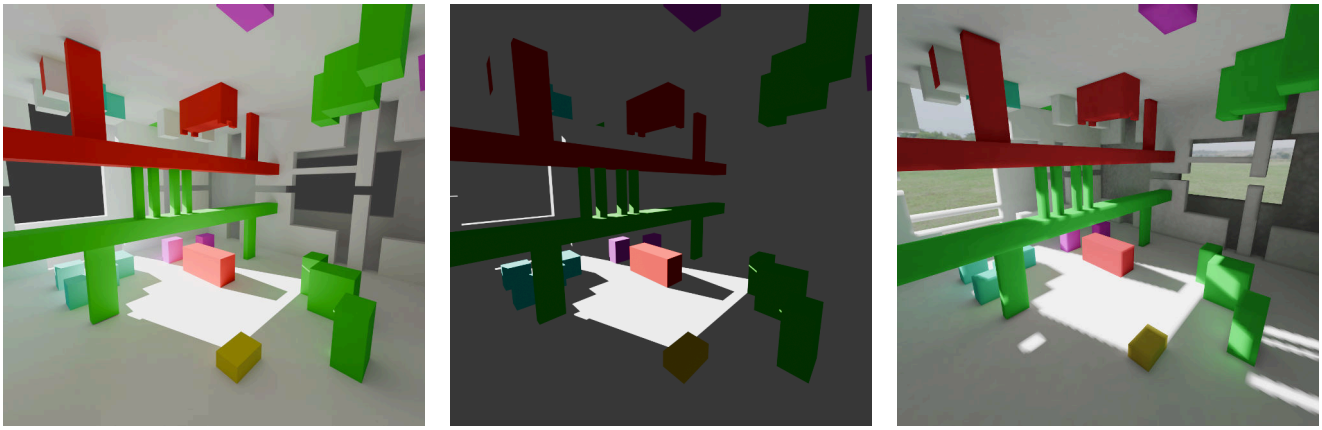
Parameter	**Default**
Voxel map resolution	256^3
Matrix row size	384
Normal Encoding	$3axis$
Build Matrix Mode (DDA/SDF)	DDA
Edge removal	Disabled
Sample blurred radiance	Enabled
Premultiplied alpha	Enabled
Specular Writeback	Disabled
Specular Raycount	2x fragment
Diffuse Raycount	384x voxel
Viewport Size	1200×1200

If not mentioned, the values for these parameters are by the default the ones specified in the table.



(a) Cycles reference **(b)** EEVEE reference **(c)** Implementation

Figure 15: Box scene comparison against Cycles and EEVEE as ground truth with Radiosity Implementation.



(a) Cycles reference **(b)** EEVEE reference **(c)** Implementation

Figure 16: AbstractPark comparison against Cycles and EEVEE as ground truth with Radiosity Implementation.

7.4. Comparing across implementations

In this test scene I compare Box and AbstractPark scenes, rendered with Blender *EEVEE*, *Cycles* against my algorithm. The goal of this experimentation is to compare the result of the given implementation with actual commercial solutions and check the correctness of the method.

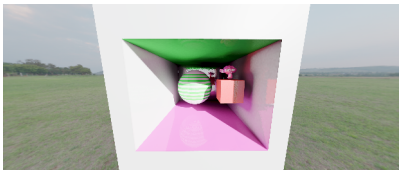
This is the comparison across implementations of the chosen scenes (Figures 15, 16).

Comparing this particular scene with **EEVEE** and **Cycles** we can see that the result given by the approximation produces a close match with the version produced by Cycles. Looking closely, the indirect shadows are appreciated to be in the same places but with a different grading, this can be more likely attributed to the amount of approximations that are done in the method and for the tonemapper and light shading constants in my implementation, which do not match exactly the ones Cycles uses.

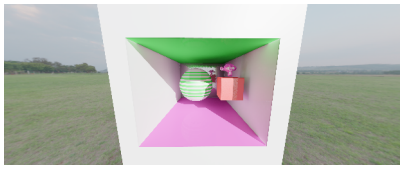
As expected there is great difference with the **EEVEE** since its Blender non raytraced based renderer meant to provide a quick render of the scene that does not take into account indirect illumination.

This indicates a very good result and that the method holds up to the standard of **Cycles** for this type of scenes.

The only meaningful difference was the shadow in the direct illumination component, which is due to the use of PCF ShadowMapping, however this was not part of the scope of the implementation and was an expected result in the comparison.

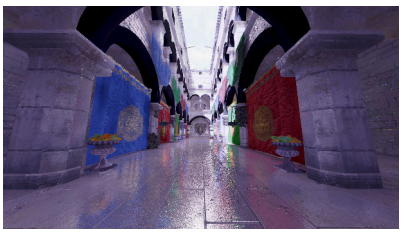


(a) Brute force approach

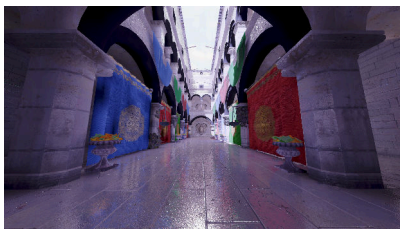


(b) Radiosity algorithm

Figure 17: Comparison of result between reference result computed by the brute force approach and the radiosity algorithm (Box).



(a) Brute force approach



(b) Radiosity algorithm

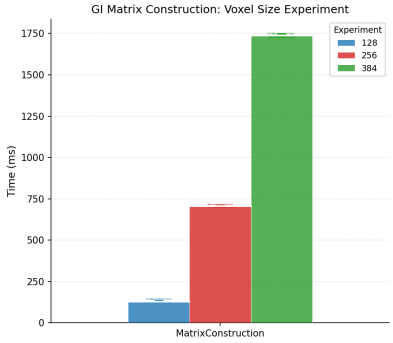
Figure 18: Comparison of result between reference result computed by the brute force approach and the radiosity algorithm (Sponza).

7.5. Comparing with builtin brute force approach

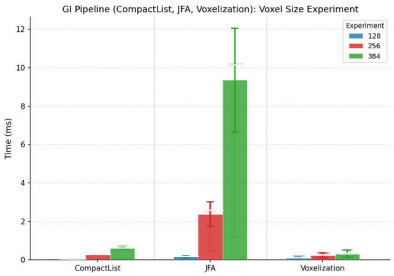
In this test I will compare the radiosity engine with the result obtained from the brute force approach of tracing rays in the fragment shader through the voxelized volume, a strategy similar to what a screen based approach of Global Illumination algorithm would do (Figures 17, 18).

There is meaningful difference between the brute force approach and the Radiosity algorithm, this is mostly due to the fact the brute force approach struggles with multibounce. This result shows that the Radiosity method implemented accurately performs the convergence of the Jacobi method.

One thing to note is that the Radiosity implementation struggles with the corners of the walls, there is some leaking of light from the front of the white wall to the ceiling and the floor, this shows also the limitation of the method presented and it's worth keeping it present.



(a) Computation time of stage *Matrix-Construction*



(b) Computation time of stages *CompactList JFA Voxelization*

Figure 19: Comparison of computation time of setup stages for variable voxel size resolutions (128, 256, 384).

7.6. Study on voxel resolution

In this experiment we are going to study how does the change in voxel resolution affect the result of the algorithm in both quality and performance for a default set of parameters.

The computation time for the different voxel resolution sizes can be seen in Figure 19 and its result in Figure 20.

The study on the different voxel grid resolution reveals the following. From a quality perspective, low scale resolutions look better than one could expect. However this is more or less subjective and depends on the degree of quality we want our representation to have.

But an important note is that low scale resolution show less artifacts providing a more conservative result, for example this can be seen in the curtains (The blue one at the left), the other resolutions struggle because the voxels inside the waving of curtains cant represent correctly the normal at high frequency scale and they are shown to be partially occluded in the Radiosity algorithm. This however does not happen with the lower resolution.

We are also loosing detail in the specular reflections which is expected, but not to a considerable degree. It's important to remark than in this implementation we were using 128, 256, 384 voxel resolutions, being able to retain a great amount of detail with a much smaller representation is very good news.

Considering also the execution cost in both computing the *JFA*,*CompactList*,*Voxelization* and *Matrix Construction* stages, we can see a considerable trend, it would seem that the cost is almost proportional to the number of voxels (Cubic to the resolution) which is expected.

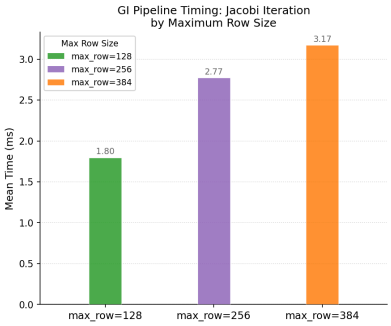
We can also see that the cost of constructing the matrix is much larger than the other two, this is also expected.

(a) Rendering for 128 voxel resolution

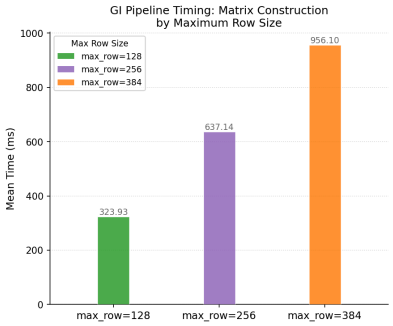
(b) Rendering for 256 voxel resolution

(c) Rendering for 384 voxel resolution

Figure 20: Comparison of the different voxel resolutions for a same configuration with full GI pipeline.



(a) Computation time of stage *Jacobi iteration*



(b) Computation time of stages *MatrixConstruction* stage

Figure 21: Comparison of computation time of *Jacobi iteration* and *MatrixConstruction* with maximum row sizes of (128, 256, 384).

7.7. Study on Maximum Row Size

For the study on maximum row size for a given entry in the radiosity matrix we have the computation time in both the construction of the matrix and its Jacobi iteration in Figure 21.

For its visual result we have the same scene configuration with 2 different light configurations (both results have the same compute time use).

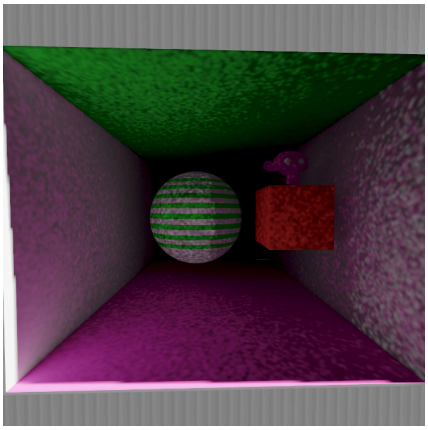
One is a scene that maximizes a configuration with *Low Probability - High Energy*, which as established before, it's the worst case for the radiosity algorithm and produces the most noise (Figure 22).

The other is an average case light situation for the algorithm, for reference (Figure 23).

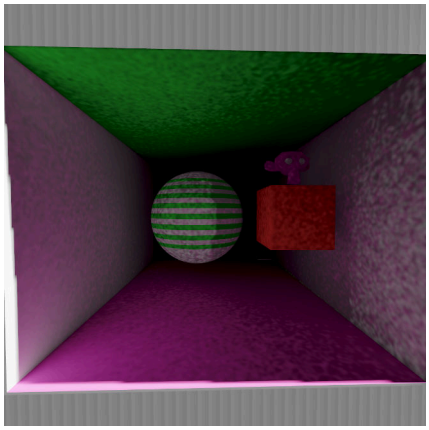
For the maximum row size study we had 2 cases, one with a light configuration that maximizes the amount of *Low Probability - High Energy* paths and one that didn't, which was an average lighting case. In this experiment we can see that increasing the maximum row size during computation directly affects the perceived noise in the Extreme case scenario.

However, for the average case this difference is almost unnoticeable, this also very interesting because it shows that for the hard case we would need a very large amount of entries in the matrix to completely remove the noise, which happens mostly when we are hitting this configuration.

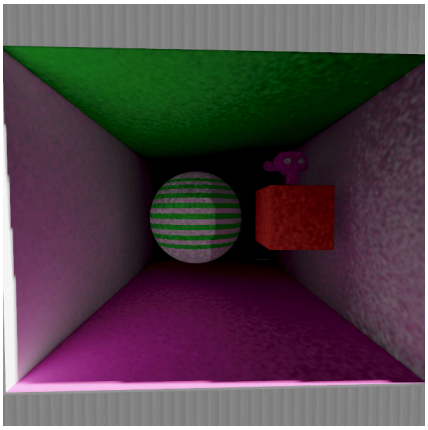
This inherent to any Radiosity algorithm that does not make use of a Hierarchical structure, or some form of sparsification in the matrix while having a prune heuristic to remove the less view factors.



(a) Rendering for 128 maximum row size

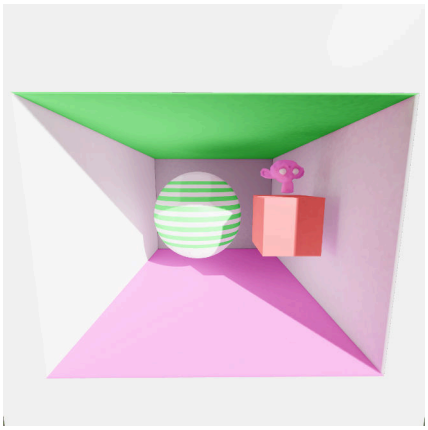


(b) Rendering for 256 maximum row size

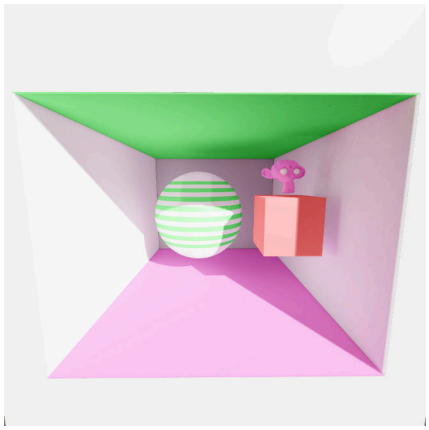


(c) Rendering for 384 maximum row size

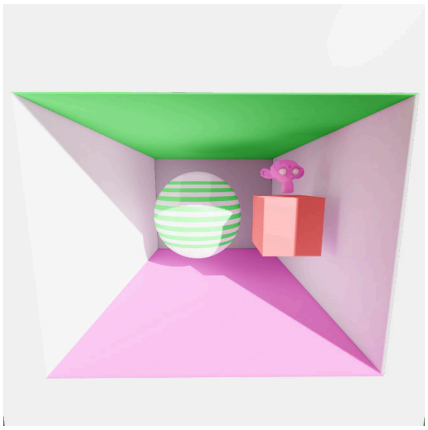
Figure 22: Comparison of the different maximum row size during matrix construction.



(a) Rendering for 128 maximum row size



(b) Rendering for 256 maximum row size



(c) Rendering for 384 maximum row size

Figure 23: Comparison of the different maximum row size during matrix construction (128, 256, 384).

7.8. Study on view factor matrix construction methods

In this experiment we are going to study how does the change in the strategy used to build the view factor matrix in the GPU changes the result in quality and performance of the algorithm. As said before, we have two possible strategies:

- **Raytracing using DDA**: This is the basic strategy that involves raytracing through the scene and emitting rays without SDF aware advancement.
- **Raymarching using SDF maps**: This is the strategy that makes use of the computed SDF field by the JFA algorithm, raymarching through the scene using the SDF values of each empty voxel.

The study will emphasize the quality comparison of the sampled diffuse component across the 4 scenes described above alongside the comparison of execution time of the relevant stages for the test, those being the *JFA* and the *buildMatrix* stages.

Here are the results on the study on view factor matrix construction methods, refer to (Figure 24 and for quality reference Figure 25).

Interestingly, for construction methods for the factor matrix we got the following results: The SDF version resulted faster in constructing matrix, but not in a very considerable degree, we are only getting at most 2 times faster matrix construction.

As for the visual results in different scenes we can observe result is mostly the same, except that we are getting a lower intensity in the indirect shadows in the SDF case, this is most notable in the *Box* scene. I have no clear explanation of why this effect particularly happens and it would require more careful debugging in the particular light traces.

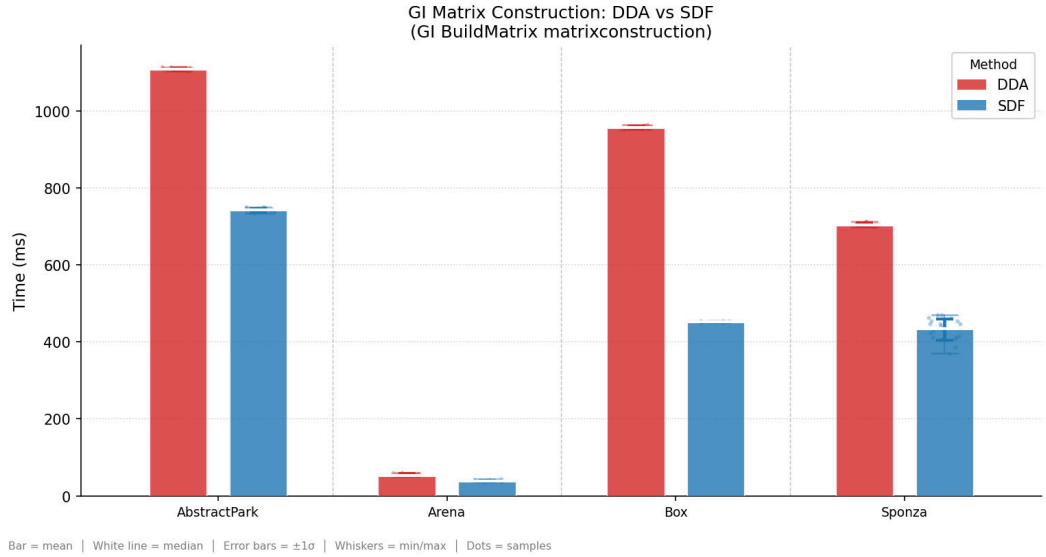
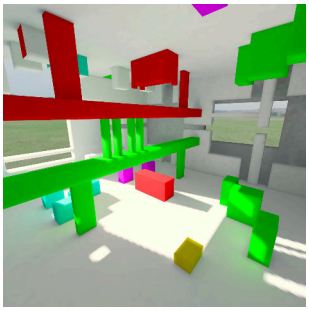
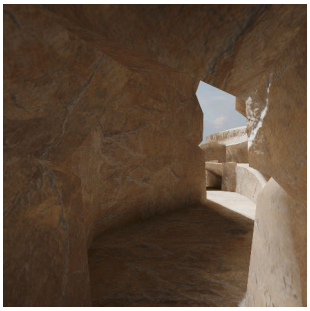


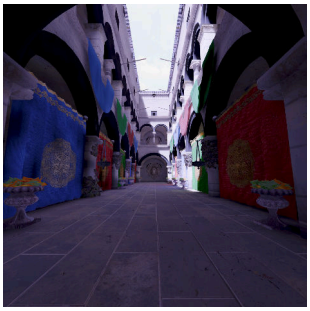
Figure 24: Barplot that show the necessary time taken to construct the radiosity matrix for each scene, red ones are the DDA version while the blue ones are the SDF version.



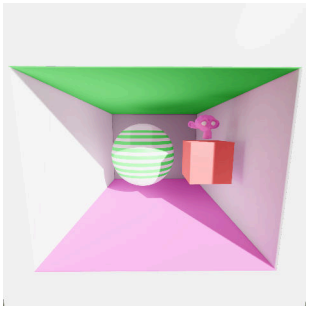
(a) AbstractPark (DDA)



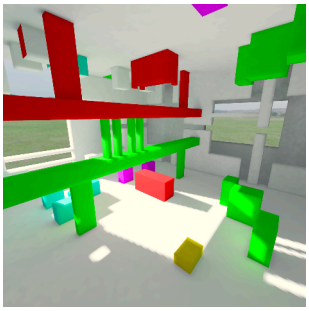
(b) Arena (DDA)



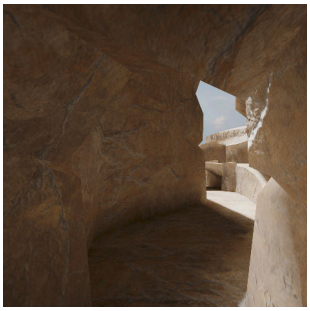
(c) Sponza (DDA)



(d) Box (DDA)

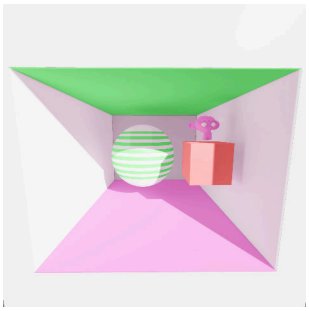


(e) AbstractPark (SDF)



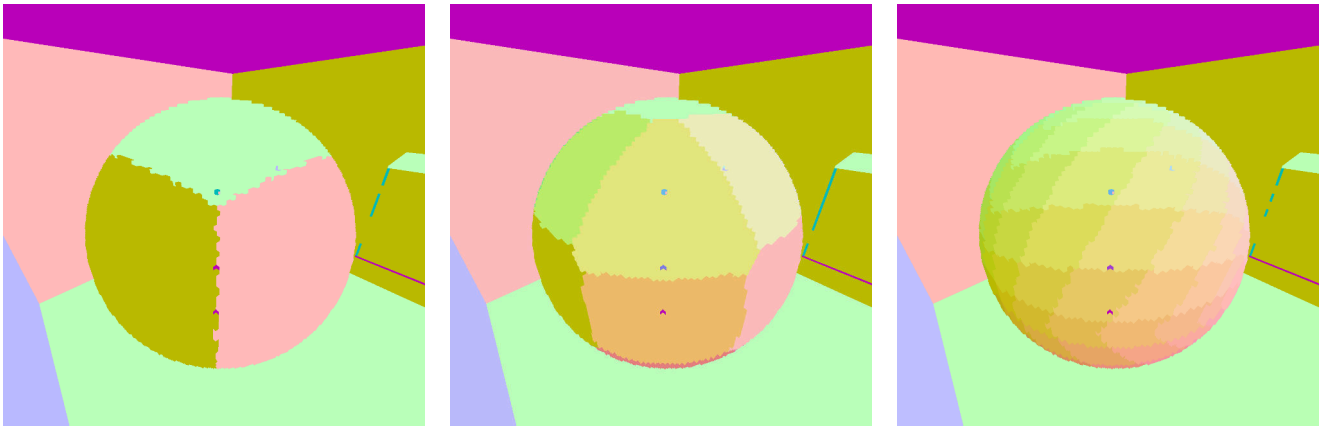
(f) Arena (SDF)

(g) Sponza (SDF)



(h) Box (SDF)

Figure 25: DDA vs. SDF Relight comparison (Diffuse Component) across test scenes.



(a) Normal with 1 axis. **(b)** Normal with 3 axis. **(c)** Normal with octahedral encoding.

Figure 26: Normal channel encoding map comparison for the three strategies (Box)

(a) Diffuse GI for normal with 1 axis. **(b)** Diffuse GI for normal with 3 axis. **(c)** Diffuse GI for normal encoded with octahedral.

Figure 27: Diffuse GI comparison for the three strategies (Box).

7.9. Comparison of Normal strategy

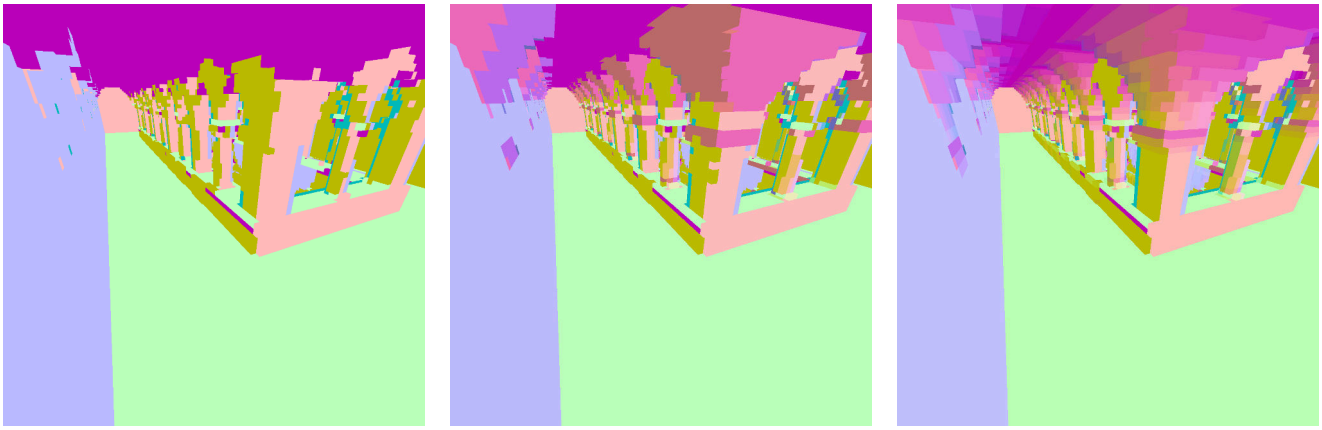
In this experiment we are going to test how does the result of the algorithm change depending on the normal encoding strategy chosen. For doing so we are going to capture the diffuse component result in different regions of the scene where surfaces have varying normal values, and capture alongside it the result of the voxel normal view. We are going to value only the result in quality since there is no overhead cost for using one method or the other.

Here are the results on the study of different normal encoding strategies for the code for two different scenes. The normal visualization can be seen in Figures 26 and 28, the final result of the computed radiosity can be seen in Figures 27 and 29.

For the normal encoding strategies used we can see a very interesting result particularly in the *Box* scene. We can observe that the voxelization algorithm did not generate the voxelization correctly for some voxels, this is an error I have not been able to correct for the voxelization algorithm.

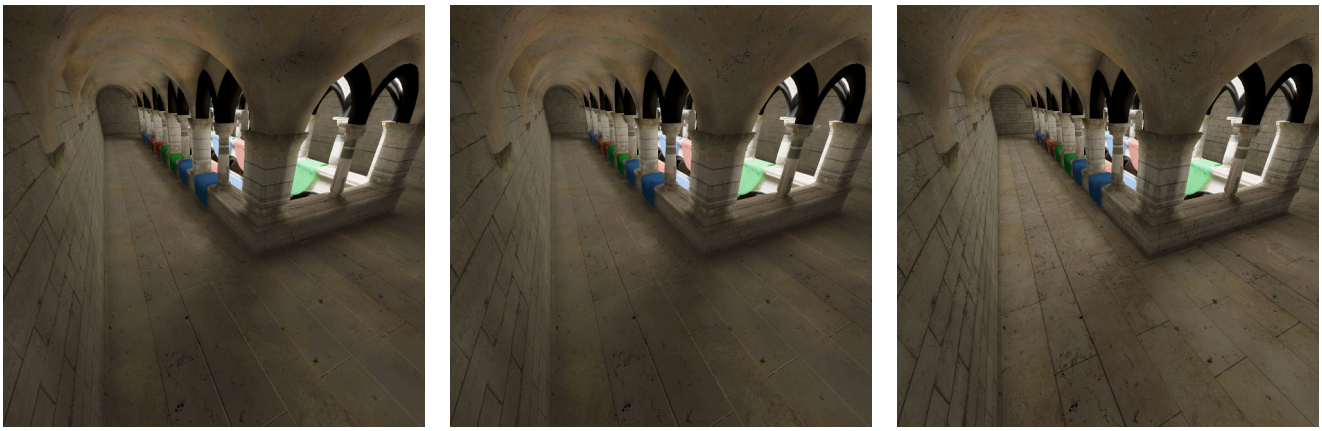
Interestingly, for the *Box* scene we can observe that the normals seems are clearly visible in the result, except when using the Octahedral encoding.

Although it's true the difference can be minimal, Octahedral encoding provides much better results specially in the *Box* scene since it can approximate much better a sphere normal values around its surface.



(a) Normal with 1 axis. **(b)** Normal with 3 axis. **(c)** Normal with octahedral encoding.

Figure 28: Normal channel encoding map comparison for the three strategies (Sponza).



(a) Diffuse GI for normal with 1 axis **(b)** Diffuse GI for normal with 3 axis **(c)** Diffuse GI for normal encoded with octahedral.

Figure 29: Diffuse GI comparison for the three strategies (Sponza).

7.10. Study on relaxation parameter

In this study, we will experiment with the relaxation parameter to determine if a specific value or range yields the most accurate results for a scene.

To do this, we will run the radiosity relight algorithm using different values of the relaxation parameter λ on the same scene configuration and compare the outcomes against a reference obtained from a previous experiment.

We will discuss not only which parameter produces the best results but also the specific effect of the parameter on the outcome. The evaluation will be based on visual comparison, as the results are expected to be subjective.

The result of the experiment on the relaxation paramater can be seen at Figure 30.

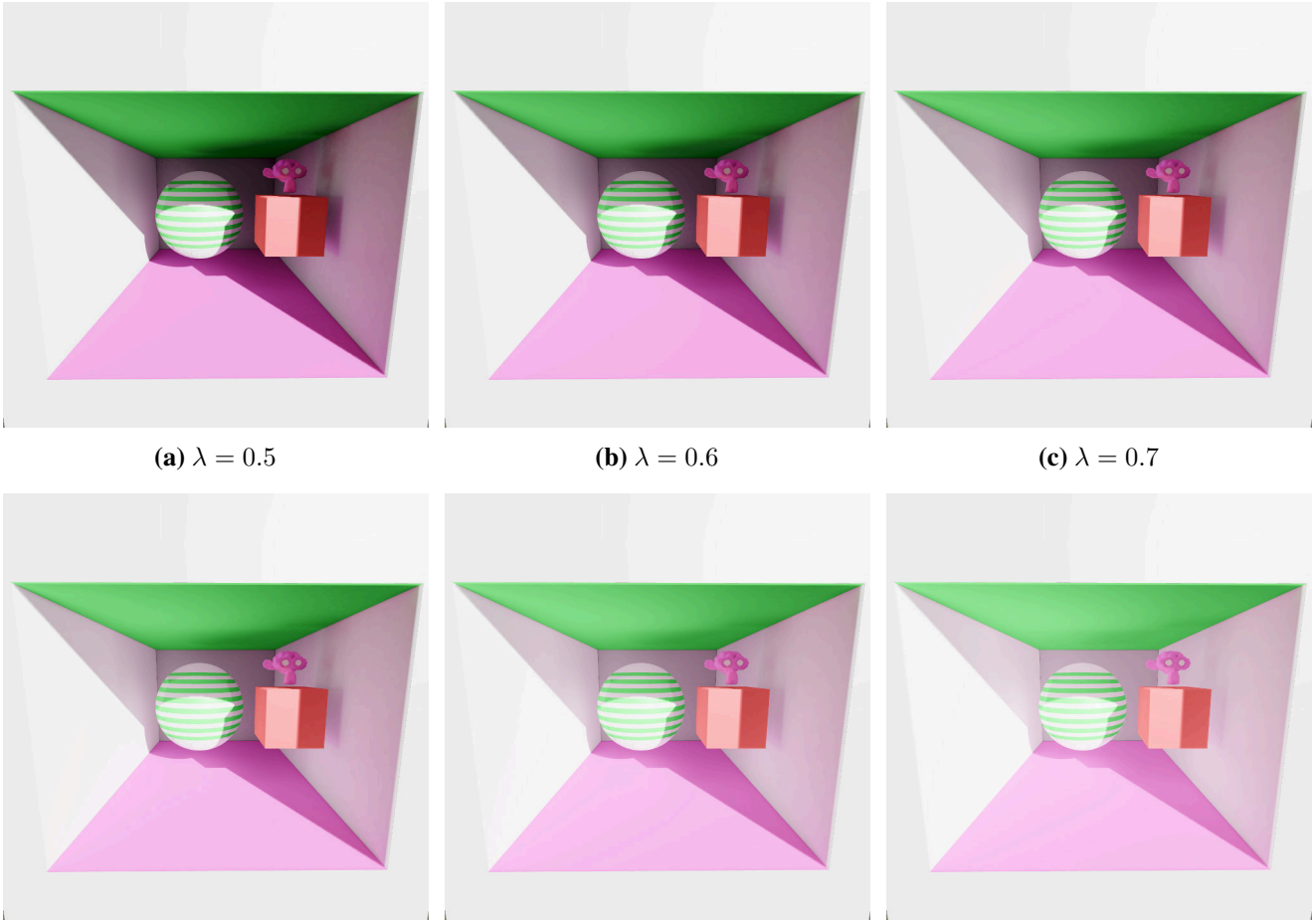
From the visual result we can see that the relaxation parameter, as expected, controls the propagation of light through the radiance volume. The one that looks the more realistic (Compared with previous result at Figure 15) seems to be in the range of *0.6* and *0.8*.

However, the underlying reasons cannot be fully explained due to the lack of normalization of the matrix rows (although deviations are expected to be minimal because of similar hitting rays), indicating that further study is needed.

While a more extensive investigation into the effects of this parameter and the search for an optimal

value was not conducted due to time constraints, the relaxation parameter can be dynamically adjusted at runtime to achieve a visually convincing result.

Furthermore, in video games, achieving a visual style that fits the game's aesthetic is often more important than physical accuracy. Therefore, having a parameter that directly controls the influence of indirect lighting is a valuable feature, even if it cannot reliably produce physically accurate results on its own and requires manual tuning.



(a) $\lambda = 0.5$ **(b)** $\lambda = 0.6$ **(c)** $\lambda = 0.7$

(d) $\lambda = 0.8$ **(e)** $\lambda = 0.9$ **(f)** $\lambda = 1.0$

Figure 30: Effect of the Jacobi relaxation parameter λ on diffuse global illumination convergence in the Cornell box scene. Lower values damp oscillations at the cost of slower convergence; $\lambda = 1.0$ corresponds to unweighted Jacobi iteration.

(a) Shadow boundary artifact, correction *OFF*

(b) Shadow boundary artifact, correction *ON*

Figure 31: Difference between shadow boundary correction procedure

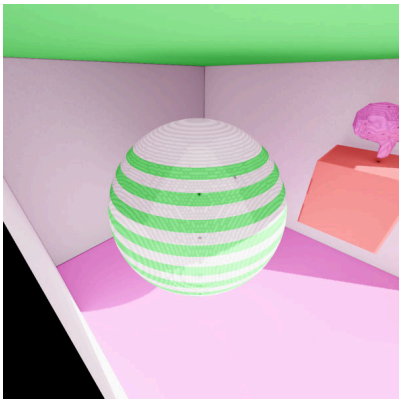
7.11. Study on the Edge Removal algorithm

We are going to study in performance and quality the result of the Edge Removal algorithm in the experimentation.

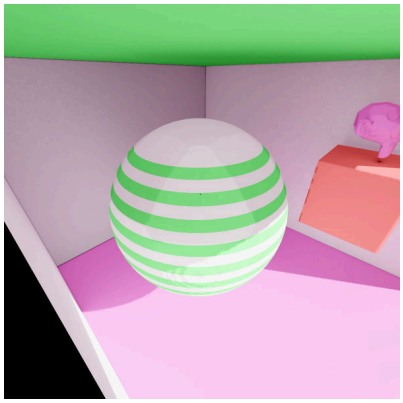
The edge removal algorithm is applied in the *EmissionScatter* stage so the primary focus of study will be the compute time spent in this stage. Also additional focus will be made to the specular component since this algorithm directly affects it.

The result of the edge boundary procedure application can be seen in Figure 31.

As observed, the proposed rule corrects the shadow boundary artifact; however, it causes us to lose the ability to see direct lighting through reflections. This results in an unfortunate tradeoff, forcing a compromise between the visual accuracy of specular and diffuse Global Illumination, which is undesirable.



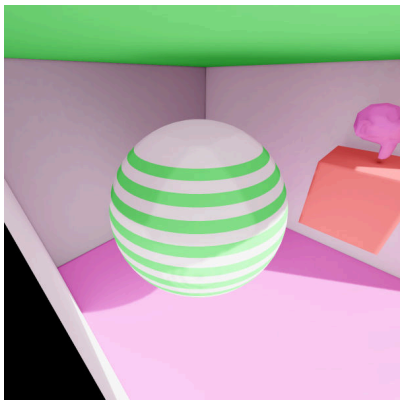
(a) Blur *OFF*, Premultiplied alpha *OFF*



(b) Blur *OFF*, Premultiplied alpha *ON*

Figure 32: Difference between linear with and without premultiplied alpha

(a) Blur *ON*, Premultiplied alpha *OFF*



(b) Blur *ON*, Premultiplied alpha *ON*

Figure 33: Difference between Blur with and without premultiplied alpha

7.12. Study on sampling modes

We are going to study how using different sampling modes affects performance and quality. That is, the 4 combinations previously stated:

- Linear sampling with no pre-multiplied alpha
- Linear sampling with pre-multiplied alpha
- Sampling with blur kernel with no pre-multiplied alpha
- Sampling with blur kernel with pre-multiplied alpha

We observe (Figures 32 33 34 35) that the best result is achieved using the blur sample mode with premultiplied alpha. The results for other options that do not use premultiplied alpha show noticeable shadow blur. Additionally, the artifacts previously caused by the voxelization algorithm are corrected when using the blurred method with premultiplied alpha.

(a) Radiance linear sampling

(b) Radiance with linear sampling and premultiplied alpha

Figure 34: Comparison of radiance sampling with linear sampling with and without premultiplied alpha

(a) Radiance Blur with linear sampling

(b) Radiance blur with linear sampling and premultiplied alpha

Figure 35: Comparison of radiance blur with linear sampling with and without premultiplied alpha

7.13. Study on Specular writeback

In this experiment we will study how Specular Writeback technique affects the rendering result in both performance an quality.

For doing so we are going to setup a scene in sponza such that we have minimum direct light coming in the scene, and rely only in the reflections through the scene of the envmap using IBL. This way we can visualize exactly what is the contribution of the Specular Writeback component to the scene since given the current configuration of the Radiosity algorithm presented, if no direct light is coming through the scene, the emissive component in the *SpMV* is left only to the specular writeback we are going to inject through the emissive volume.

The results found have been very poor, not only performance has dropped significantly (Figure 37. For the full performance report, refer to Table 10 at the end of the document) during the forward stage, but the result (Figure 36) has been found quite underwhelming.

During experimentation, it was observed that the *EmissiveTexture* experiences significant flickering and high-frequency artifacts. Although it adds realism by incorporating Image-Based Lighting (IBL) sources, the performance cost outweighs the benefits. As shown, the cost is multiple times higher than regular shading, which already involves heavy lighting computation and Global Illumination Specular Tracing. This comparison effectively disproves the original hypothesis that using the mask to select fragments for writing to the emissive texture is acceptable in terms of performance.

(a) Reference image of the full GI system without specular writeback

(b) Result in the radiance texture of the addition of the specular writeback algorithm

(c) Composition of the full GI system with specular writeback activated

Figure 36: Study on specular writeback

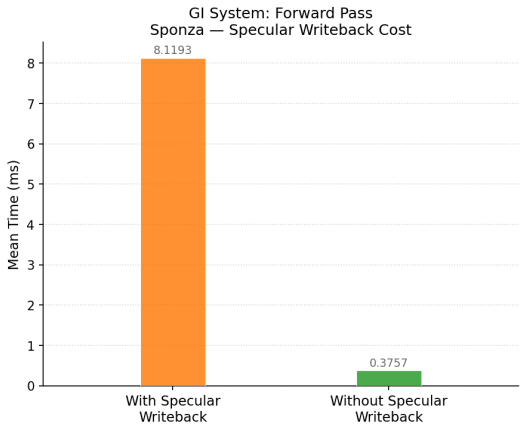


Figure 37: Comparison of renderpass cost of GI system with and without Specular Writeback, reference values gathered from Tables 10 and 7.

7.14. Visual comparison in different scenes

In this experiment we are going to run the radiosity algorithm with default parameters for each scene and evaluate them on the visual quality displayed, showing the version without GI, the version with Diffuse GI, and the version with Diffuse GI and Specular GI combined.

We are going to run the experiment on the 4 scenes proposed.

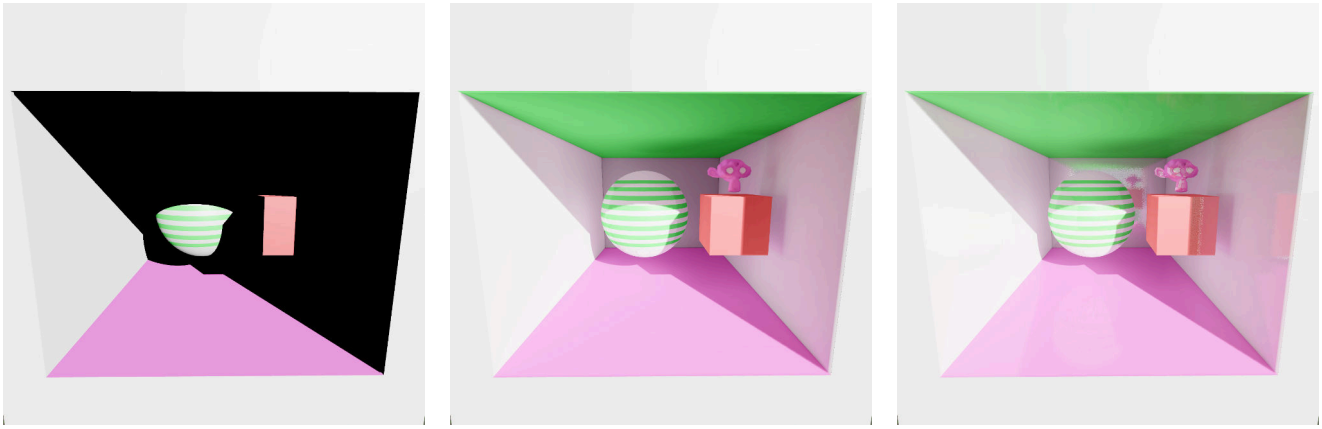
In this experiment we are going to run the radiosity algorithm with optimized parameters for each scene, showing the version without GI, the version with Diffuse GI, and the version with Diffuse GI and Specular GI combined (Figures 38, 39, 40, 41).

The images presented show a realistic approximation for the diffuse GI component and a credible result for the specular component of the algorithm, however the compression of the normals can be detected in the Box scene visualization, this points out that although the normal encoding proposed has been proven sufficient for most cases it can still show artifacts in the particular case of oblique surfaces with varying normals with direct contact with direct illumination.

Performance results in Tables 7, 9, 8, 6 at the end of the document.

(a) No GI. **(b)** With diffuse GI. **(c)** With full GI.

Figure 38: Sponza Scene comparison visualization.



(a) No GI. **(b)** With diffuse GI. **(c)** With full GI.

Figure 39: Box scene comparison visualization.

8. Conclusion

In this section I will give a final closure to the project and answer the Research Questions proposed for the experimental study of the algorithm.

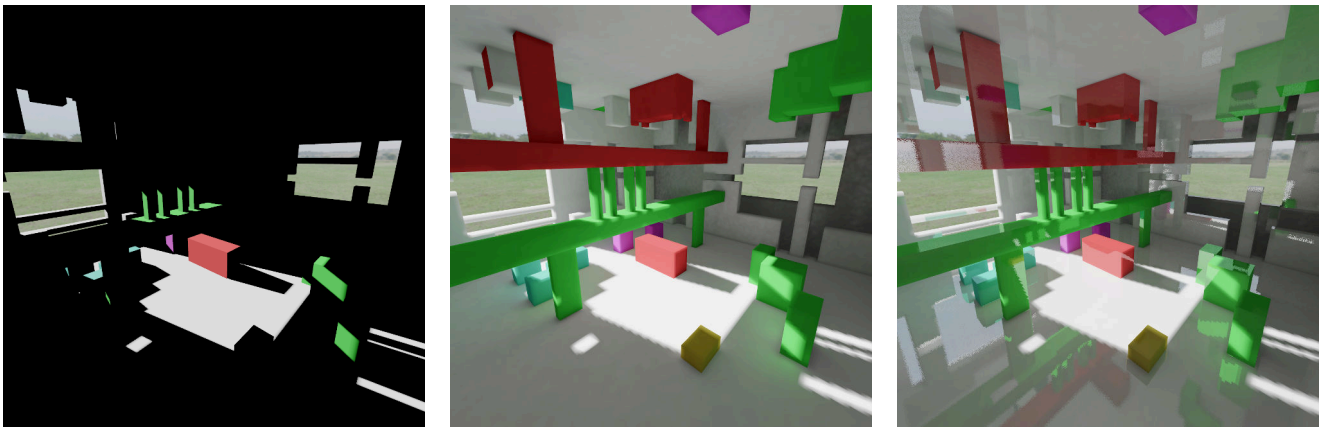
RQ1 The presented results in the experimentation have shown to give a reasonable approximation at least in the diffuse component to the one computed with *Cycles* for the tested scene.

RQ2 The result of the Immediate Radiosity algorithm compared with the brute force approach that has been coded alongside have given a quite different output, the brute force has struggled in propagating light through the scene, combined with the reference results of *Cycles* that match the result obtained with Immediate Radiosity it's fair to say that the difference is mainly due to an incorrectness in the brute force approach, not the radiosity algorithm, this can be explained because the brute force approach is a basic screen space algorithm that cant propagate light efficiently through multiple bounces in its accumulation buffer.

RQ3 The mean computation time of the factor matrix has proven to be quite large (in the scale of a handful of seconds) for the most scenes. This discards the availability of the algorithm to efficiently regenerate the matrix dynamically upon constant scene changes. For dynamic geometry another approach would need to be taken.

(a) No GI. **(b)** With diffuse GI. **(c)** With full GI.

Figure 40: Arena scene comparison visualization.



(a) No GI. **(b)** With diffuse GI. **(c)** With full GI.

Figure 41: Abstract park scene comparison visualization.

RQ4 The voxel resolution affects quality in a subtle way, can be distinguished the loss of quality in faces with high frequency normals, varying geometry or variation in direct lighting, however, due to the nature of the Radiosity equations this becomes subtler in the average cases. A lower resolution also provides with smoothed noise that helps reduce artifacts at the cost of quality, which is an interesting trade-off to be explored. Computation time scales rapidly in the increment of voxel resolution which was expected.

RQ5 The maximum row size of the matrix has shown to be directly connected with the amount of noise artifacts produced with *Low Probability - High Energy* paths which was expected. The computation time has been shown to increase almost linearly, this was expected too.

RQ6 The performance numbers show that raymarching the volume is faster than using DDA by a factor of 2 on average. Still not enough for dynamic scenes but a meaningful speedup that shows no clear quality loss for the most cases.

RQ7 The change of normal strategy has shown to actually not improve much quality, even with a very low resolution of normals the algorithm is still capable of producing credible results. However *Octahedral* encoding has been proven to provide better results for surfaces that have a great variation of normals, particularly curves. In the other strategies, although they can still provide a very similar result,

hard transitions of normals can be appreciated, this is however not the case for the *Octahedral* one. For this reason, the *Octahedral* encoding is recommended.

RQ8 The mean computation time for the relight iteration has been shown to be in the range of 2ms-4ms, which stands in the competitive range for similar algorithms.

RQ9 Optimal relaxation parameter has been shown to be optimal (comparing against *Cycles* reference) with a value in the range of 0.7 and 0.9, however as explained before, this parameter is scene dependent and a general rule cannot be derived. However the pattern that can be drawn from the visualization of the rendering affected by different values show the direct effect in light propagation. Beyond consideration of the optimal one for strictly physical realism reasons, it can be used for the exaggeration or attenuation of the indirect shadows that appear in the algorithm.

RQ10 Blurring the sampled result in rendering time shows an almost optimal correction for the sampled diffuse global illumination for curve surfaces, specially when premultiplied alpha is activated on top, which removes the black edge artifacts completely.

RQ11 The radiosity removal algorithm accurately removes the shadow boundary artifact presented, however at the cost of reflections of light across reflective surfaces in shading time.

RQ12 Specular writeback affects the result poorly, although adding light contributions from reflective surfaces and the environment through IBL shows strong flickering and emissive noise. Mostly due to implementation reasons.

8.1. Final Conclusion

This algorithm although producing a quite reasonable approach to the GI function. Does suffer from many problems which make it completely unusable on production for the most cases.

It has a very heavy toll on GPU memory which is unacceptable for a real time use, it discards this solution for a scene with dynamic geometry. The method has reasonable amount of noise that other alternatives to this method do not have.

The computation time is considerably high, for the setup stages this can go up to the mark of 10s for generating the initial setup structures (that is mainly, the view factor matrix). Also the computation time for the continuous iteration of the methods is for most scenes cases in the range of 2ms-4ms (Tables 8, 9, 6, 7), although this is in range of the current real-time Global Illumination algorithms (Epic Games (2026)), this is actually a poor result.

The reason why I expected a much better performance was because the method relied on the hypothesis (as said in the Introduction section) that baking the rays influences of the different patches in the scene to a Radiosity matrix and reducing global illumination computation to a simple *SpMV* product would heavily reduce computation toll.

This has been far from the case, what we are most likely experiencing in the computation time is a memory bandwidth issue, the GPU is trying to read Gigabytes of memory in very little span of time to perform one single iteration. While the current result is acceptable for real time rendering at high resolutions and framerates, the results given have to be taken with a grain of salt, the experiments were executed on a very powerful GPU, an AMD 7700XT, which is notoriously good at memory bandwidth and SSBO reads.

It's left as future experimentation to try this method on more conservative GPUs with enough VRAM to hold the matrix.

This is however, expected to largely improve by compressing the view matrix (Matrix hierarchization, randomized block SVD, or other methods), which would help reduce memory use and noise.

The only real case of use for the algorithm as it is now would be as an optional Global Illumination backend for renderers like *EEVEE* that would help close the gap with *Cycles*, since *Cycles* takes minutes in execution to light the scene while the algorithm presented took less than 3ms.

9. Sustainability and Ethical Implications

Scope of Ethical Implications This project is a rendering research tool with no direct societal ethical implications. It does not involve personal data collection, does not interact with human subjects, and produces no artifacts with potential for misuse. The primary ethical dimension worth examining is therefore environmental: the energy footprint of GPU-accelerated computation.

Energy Consumption and GPU Power Draw Graphics Processing Units, particularly high-end discrete cards, are among the most power-hungry components in modern computing hardware. The GPU used throughout this project has a Thermal Design Power (TDP) of 240W under full load. In a naively implemented real-time renderer, the GPU pipeline is kept saturated at all times in order to sustain the target framerate, resulting in sustained near-peak power draw for as long as the application runs. Over extended development and benchmarking sessions, this accumulates into a non-trivial energy cost.

The Radiosity Caching Advantage A key architectural property of the diffuse global illumination component computed by the radiosity algorithm is that it is view-independent: the irradiance distribution across surfaces depends solely on the geometry and the configuration of light sources in the scene, not on the position or orientation of the camera. This stands in contrast to specular effects, which must be recomputed per frame as the camera moves. This property has a direct sustainability implication. Because the diffuse radiance solution is invariant under camera motion, it does not need to be recomputed every frame. The sparse radiosity pipeline implemented in Detramotor exploits this by decoupling the GI solve from the per-frame render loop: the radiosity algorithm is only dispatched when a change in scene lighting is detected, and its result is cached in the voxel emission volume for subsequent frames to read from at negligible cost.

10. Implementation Note

In this section I will comment on the actual implementation and shader pipeline for both the forward rendered fragment and GI compute shaders, as well as some implementation notes of this in C++ that I have found to be relevant for the method described.

The whole rendering engine is coded in C++ and Vulkan, the choice of writing a custom renderer instead of using a commercial is to be able to use some Vulkan features directly that allow us to improve performance and memory usage in both GPU and CPU, and allows us to easily use async computation for the radiosity matrix computation.

10.1. AMD RDNA3

Part of the code of the implementation was done taking into account the specifications of the GPU in which this was primarily tested, that is AMD XT 7700, which is part of the RDNA3 family. This type of

GPUs have a Wavefronts of 64 threads and are particularly good in memory bounded code which might have or might not have taken a heavier toll on the overall algorithm.

I have not been able to test the code on other GPUs that are powerful enough to run the Radiosity procedure presented.

10.2. Async compute

Vulkan exposes multiple hardware Queues and different Queues families to which submit our commands, this allows us to provide the GPU with commands that are outside the regular rendering loop, potentially allowing the GPU to schedule them better and allowing for independent commands to execute concurrently if enough cores are free. This is very useful in scenarios where the rendering pipeline has stalls between stages or some producing some effects that can be allowed to have some frames of latency.

As we have seen there are multiple points in the project where this idea can be applied. I distinguished two different cases.

10.2.1 Async matrix computation

The recomputation of the matrix can be done on a separate hardware queue, there is no need to cause a hard stall on the rendering pipeline for the voxelization stages, however doing this requires carefully double buffering the matrix (since we cannot overwrite the matrix while the rendering is potentially relighting the scene), or a lighter approach where we reconstruct the matrix at the same time we are proceeding with forward rendering. This is an interesting idea however the renderer is too simple for there be any benefit right now as it is, I am already hitting high levels of occupancy in the GPUs cores. However it's interesting idea to note for a future online matrix recomputation kernel where we can recalculate regions of the view factor matrix in async queue and perform a cheap GPU *memcpy* operation in the beginning of the following frame.

10.2.2 Async scene relight

This is a more interesting approach, right now the renderer as it's proposed requires a hard stall between the GI relight stage and the regular rendering, however if we were allowed to add an additional frame of latency between the relight engine and the forward shading we could improve performance. It's important to remember that the relight stage is primarily memory bounded while the forward shading it's not, this allows for potential parallelism between two stages. However this is very hard to profile since it requires hardware counters to visualize GPU occupancy and frame memory stalls, most appropriate way to test the performance of the approach is to directly measure the difference in mean FPS.

11. Future work

The algorithm described though powerful has the downside of requiring heavy optimizations and problem solving for lots of use cases. Given the approximate nature and the availability of computation responsiveness of the lighting it would be interesting to use the algorithm in a game engine. However for doing that we would need to resolve some of the main downsides of the algorithm proposed first, that are not computationally expensive but quite difficult to code correctly.

11.1. Dynamic mesh diffuse lighting

For most videogames that use lightmappers or similar baked indirect lighting, dynamic objects are able to receive dynamic illumination although they are not able to cast it, this is usually done by using GI probes that sample for every point in space the diffuse light of the scene, right now this is not available in the presented method and thus dynamic objects cannot receive light.

Given the current setup this has very easy fix, we are using a 3D texture to store the radiance component, the following possible solutions exists.

- Create another 3D volume with much less resolution that has for each voxel the radiance of the closest radiance voxel in the main 3D texture, each texel of the new volume would act as GI probes.
- Create the full chain of mip map levels for the 3D texture volume, these option would consume more memory than the other but at the cost of being way cheaper to compute.
- Manually/Smartly placing GI probes in the scene to capture radiance, and then sampling the closest GI probes for each fragment for computing the final GI for dynamic objects.

It's quite acceptable for videogames standard that dynamic objects do not contribute to the indirect lighting of the scene as occluders, shadowmap information can be used to fake the effect at a much cheaper rate, specially if we can use it to discard indirect light produced by the light caster.

11.2. Matrix dynamic recomputation

Currently the system does not have an efficient way to reconstruct the matrix dynamically, that is, rebuild only one section of the matrix for updated geometry. This is the main limitation of the system. An efficient solution that can recompute small parts of the matrix (add or remove sections) and calculate optimized compressed versions would increase a lot the number of use cases this algorithm could have, essentially we could have an almost real time lightmapper.

11.3. Variable sized patches

Most lightmappers and radiosity engines don't assume the same resolution for the patches of the scene, some surfaces might need more GI detail than others, specially in videogames. This is a very important realization since for this reason alone since this limits heavily the resolution of details for broads scenes while it increases in small sized ones, however since the main goal of the project was to demonstrate the algorithmic possibilities and overall potential and not to provide a full-fledged implementation that can work fine on a variety of screen types this was dismissed. But it's easily noted too that this has a solution in the proposed algorithm.

The scene can in theory be built with a combination of voxel volumes, or if we are using Vulkan to its full extent, a single voxel volume with sparse memory allocation and variable in-world resolution.

Once we have constructed the matrix and established a mapping between the radiance volume (or volumes if we went for this technique), we don't actually care anymore about voxel sizes, we can then reuse the very same simple *SpMV* compute kernel in the same way.

11.4. Matrix hierarchization

A typical optimization for radiosity is the use of a hierarchized factor matrix, this enables faster light transport between sections that are closely related and removes the need for per pair of patches direct propagation during iteration.

11.5. Improved shadow mapping

Right now the implementation uses the traditional PCF shadow mapping, a better alternative to demonstrate the capabilities of the method and possibly help to provide a more physically accurate radiosity would be to use variable filter width VSM. This type of shadow mapping have been heavily explored for providing soft shadows with variable-width kernels and to show some nice properties which would have made it a better choice than PCF for sampling the shadows in the Light gather stage and during forward pass sampling direct light. In fact, the most noticeable difference between some comparisons of **Cycles** and **Immediate Radiosity** have been due to the hard edges of the PCF shadows in the implementation.

References

Amanatides, J. and Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, pages 3–10. Eurographics Association.

Blinn, J. F. (1978). Simulation of wrinkled surfaces. *Computer Graphics (Proceedings of SIGGRAPH 1978)*, 12(3):286–292.

Burley, B. (2012). Physically based shading at Disney. SIGGRAPH 2012 Course: Practical Physically Based Shading in Film and Game Production.

Cohen, J., Olano, M., and Manocha, D. (1998a). Appearance-preserving simplification. In *Proceedings of SIGGRAPH 1998*, pages 115–122.

Cohen, J., Olano, M., and Manocha, D. (1998b). Appearance-preserving simplification. In *Proceedings of SIGGRAPH 1998*, pages 115–122.

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24.

Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. (2011). Interactive indirect illumination using voxel-based cone tracing. In *Computer Graphics Forum (Proc. Pacific Graphics)*, volume 30, pages 1921–1930.

Epic Games (2026). Lumen performance guide.

Frank Meinl, K. (2026). Sponza model.

Geomerics Ltd. (2010). Enlighten: Real-time global illumination for games. Technical report, Geomerics Ltd.

Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, pages 213–222. ACM.

Hammersley, J. M. (1960). Monte Carlo Methods for Solving Multivariable Problems. *Annals of the New York Academy of Sciences*.

Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150. ACM.

Khronos Group (2023). Vulkan 1.3 specification.

Lengyel, E. (2011). *Mathematics for 3D Game Programming and Computer Graphics*.

NVIDIA Corporation (2014). VXGI: Voxel global illumination. NVIDIA Developer Blog.

Quilez, I. (2020). Premultiplied alpha. `https://iquilezles.org/articles/premultipliedalpha/`.

rfb39ca4 (2013). Branchless voxel raycasting.

Rong, G. and Tan, T.-S. (2006). Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 109–116, New York, NY, USA. ACM.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition.

Tejeda, D. G. (2026a). Abstractpark model.

Tejeda, D. G. (2026b). Arena model.

Tejeda, D. G. (2026c). Box model.

Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. (2007). Microfacet models for refraction through rough surfaces. pages 195–206.

Williams, L. (1978). Casting curved shadows on curved surfaces. *Computer Graphics*, 12(3):270–274.

Wong, T.-T., Luk, W.-S., and Heng, P.-A. (1997). Sampling with Hammersley and Halton Points. *Journal of Graphics Tools*.

Wright, D. and Hillaire, S. (2021). Lumen: Real-time global illumination in unreal engine 5. In *SIGGRAPH Advances in Real-Time Rendering in Games*.

12. Annex

Rendering metrics The following tables (6, 7, 8, 9) present the empirically obtained rendering time consumption for each scene. For each stage of the algorithm, the tables provide the average, minimum, maximum, standard deviation, and percentiles based on a set number of samples collected during rendering. These tables offer a detailed overview of the performance of the proposed method. The parameters used to obtain these results correspond to those specified in the default parameter table in the experimentation section (Table 5).

The stages prefixed with "GI" correspond to those involving the radiosity method algorithm, while those starting with "RE" refer to the main rendering stages of the application. Below is a brief description of the "RE" stages. Although these stages are not part of the algorithm described in this thesis, they are essential for composing the final image. Due to the variety of approaches available in graphics programming to accomplish this, they are left unexplained here, as they do not significantly differ from a traditional naive implementation of forward shading.

- *RE Depth*: A depth prepass performed over the entire scene. Since forward shading is computationally expensive, minimizing fragment shader invocations is crucial for good performance. Therefore, a depth prepass strategy was employed to reduce overdraw during shading.
- *RE Forward*: The standard shading stage where forward shading is applied, and the Global Illumination component is composed.
- *RE Shadow*: The shadow mapping pass.

The same analysis of the writeback experiment is displayed in Table 10.

Table 6: Empirical Metrics on Arena

Subsystem Name	Mean (ms)	Min (ms)	Max (ms)	Std Dev	p50 (ms)	p95 (ms)	p99 (ms)	Samples
Real-Time Per-Frame Execution Subsystems								
GI Iteration EmissionGather	0.0771	0.0303	0.1746	0.0071	0.0774	0.0826	0.0856	1390
GI Iteration Jacobi	0.2183	0.0198	0.2338	0.0240	0.2212	0.2259	0.2278	1388
GI Iteration Scatter	0.0080	0.0066	0.0292	0.0013	0.0079	0.0082	0.0117	1388
GI clearEmission	0.0622	0.0189	0.1624	0.0069	0.0624	0.0677	0.0703	1390
RE Depth	0.0280	0.0134	0.0309	0.0019	0.0283	0.0288	0.0290	1387
RE Forward	0.3820	0.1859	0.3917	0.0244	0.3853	0.3878	0.3893	1386
RE Shadow	0.0549	0.0304	0.0635	0.0028	0.0552	0.0558	0.0560	1390
One-Time Setup / Structural Initialization Subsystems								
GI BuildMatrix Clear	7.8514	7.7918	7.8985	0.0232	7.8510	7.8812	7.8812	20
GI BuildMatrix CompactList	0.0772	0.0756	0.0802	0.0013	0.0766	0.0798	0.0798	20
GI BuildMatrix matrixconstruction	52.9653	50.9121	69.5263	3.8690	51.9483	54.1098	54.1098	20
GI JFA	0.3396	0.0607	1.0845	0.2978	0.0795	0.5727	0.5727	20
GI Voxelization	0.0246	0.0109	0.0309	0.0044	0.0255	0.0287	0.0287	20

Table 7: Empirical Metrics on Sponza

Subsystem Name	Mean (ms)	Min (ms)	Max (ms)	Std Dev	p50 (ms)	p95 (ms)	p99 (ms)	Samples
Real-Time Per-Frame Execution Subsystems								
GI Iteration EmissionGather	0.8501	0.1433	1.3529	0.1882	0.8742	1.1174	1.2317	860
GI Iteration Jacobi	2.1341	0.2296	2.5527	0.2226	2.1308	2.4744	2.5193	848
GI Iteration Scatter	0.0880	0.0855	0.0908	0.0006	0.0879	0.0892	0.0898	843
GI clearEmission	0.7616	0.0484	1.2659	0.1908	0.7867	1.0309	1.1452	861
RE Depth	0.1733	0.1633	0.1806	0.0027	0.1732	0.1776	0.1790	862
RE Forward	0.3757	0.3671	0.3904	0.0042	0.3752	0.3833	0.3859	861
RE Shadow	0.2036	0.1966	0.2356	0.0034	0.2033	0.2091	0.2122	868
One-Time Setup / Structural Initialization Subsystems								
GI BuildMatrix Clear	7.8091	7.7670	7.8731	0.0289	7.8124	7.8495	7.8495	21
GI BuildMatrix CompactList	0.2834	0.2768	0.2900	0.0030	0.2839	0.2879	0.2879	21
GI BuildMatrix matrixconstruction	704.6190	697.3982	717.7565	4.2556	704.6292	711.3079	711.3079	21
GI JFA	1.2863	0.1158	2.5813	1.2080	0.2922	2.5638	2.5638	21
GI Voxelization	0.2335	0.1069	0.3891	0.1253	0.1609	0.3813	0.3813	21

Table 8: Empirical Metrics on AbstractPark

Subsystem Name	Mean (ms)	Min (ms)	Max (ms)	Std Dev	p50 (ms)	p95 (ms)	p99 (ms)	Samples
Real-Time Per-Frame Execution Subsystems								
GI Iteration EmissionGather	0.8334	0.2323	1.5616	0.1730	0.7713	1.1582	1.2775	1017
GI Iteration Jacobi	3.6449	0.3638	3.8950	0.1907	3.7133	3.8068	3.8394	982
GI Iteration Scatter	0.1667	0.1560	0.1876	0.0109	0.1599	0.1835	0.1850	979
GI clearEmission	0.7114	0.1042	1.4404	0.1748	0.6495	1.0371	1.1570	1017
RE Depth	0.0298	0.0247	0.0538	0.0047	0.0286	0.0324	0.0511	998
RE Forward	0.2202	0.2062	0.2698	0.0113	0.2142	0.2369	0.2454	944
RE Shadow	0.0786	0.0737	0.1029	0.0054	0.0775	0.0991	0.1012	1068
One-Time Setup / Structural Initialization Subsystems								
GI BuildMatrix Clear	7.8220	7.7731	7.8800	0.0272	7.8260	7.8646	7.8646	21
GI BuildMatrix CompactList	0.4849	0.4796	0.4913	0.0032	0.4846	0.4900	0.4900	21
GI BuildMatrix matrixconstruction	1108.9796	1102.3842	1115.1282	3.9661	1110.3291	1113.8878	1113.8878	21
GI JFA	3.6337	0.4096	7.1741	3.3190	1.0678	7.1250	7.1250	21
GI Voxelization	0.0469	0.0324	0.0791	0.0104	0.0419	0.0616	0.0616	21

Table 9: Empirical Metrics on Box

Subsystem Name	Mean (ms)	Min (ms)	Max (ms)	Std Dev	p50 (ms)	p95 (ms)	p99 (ms)	Samples
Real-Time Per-Frame Execution Subsystems								
GI Iteration EmissionGather	0.3161	0.2592	0.3436	0.0088	0.3172	0.3270	0.3306	877
GI Iteration Jacobi	3.1754	0.2125	3.4682	0.2740	3.2036	3.2678	3.3009	865
GI Iteration Scatter	0.0948	0.0918	0.0984	0.0007	0.0947	0.0959	0.0969	858
GI clearEmission	0.2379	0.1689	0.2648	0.0112	0.2397	0.2488	0.2520	880
RE Depth	0.0126	0.0032	0.0141	0.0006	0.0127	0.0132	0.0133	877
RE Forward	0.2253	0.0601	0.2474	0.0083	0.2256	0.2323	0.2340	877
RE Shadow	0.0668	0.0530	0.0788	0.0026	0.0669	0.0698	0.0715	884
One-Time Setup / Structural Initialization Subsystems								
GI BuildMatrix Clear	7.8280	7.7548	7.8709	0.0293	7.8271	7.8676	7.8676	20
GI BuildMatrix CompactList	0.5969	0.5901	0.6054	0.0040	0.5965	0.6027	0.6027	20
GI BuildMatrix matrixconstruction	957.9435	949.7906	963.7004	3.9593	958.1627	963.1392	963.1392	20
GI JFA	6.5213	0.6460	12.9034	5.8732	0.6638	12.5404	12.5404	20
GI Voxelization	0.0346	0.0224	0.0471	0.0063	0.0306	0.0422	0.0422	20

Table 10: Empirical GPU Performance on Specular Writeback Experiment

Subsystem Name	Mean (ms)	Min (ms)	Max (ms)	Std Dev	p50 (ms)	p95 (ms)	p99 (ms)	Samples
GI Iteration EmissionGather	0.8747	0.4912	1.4879	0.2139	0.8955	1.1901	1.3369	95
GI Iteration Jacobi	2.1019	1.6643	4.3489	0.3375	2.0680	2.3224	3.4627	78
GI Iteration Scatter	0.0884	0.0868	0.0918	0.0008	0.0882	0.0898	0.0906	78
GI clearEmission	0.7685	0.3854	1.3630	0.2086	0.7902	1.0802	1.2010	102
RE Depth	0.1594	0.1057	0.4670	0.0466	0.2557	0.3221	0.3500	1855
RE Forward	8.1193	0.4743	30.2343	1.5666	3.3898	3.7763	11.2402	1768
RE Shadow	0.1798	0.1263	0.4172	0.0576	0.1732	0.3174	0.3989	7231