Computer Graphics: Implementation of Antiradiance for Interactive Global Illumination

Alessandro Massaad

August 2024

Contents

1 Introduction

Global illumination accurately models light interactions in scenes, crucial for realistic rendering. Traditional methods, like path tracing, are computationally expensive due to extensive sampling. Radiosity offers an alternative for diffuse scenes by solving the rendering equation through finite element methods but requires costly visibility checks.

The paper "Implicit Visibility and Antiradiance for Interactive Global Illumination" [1] introduces antiradiance, which eliminates the need for visibility checks by balancing excess light with negative light (antiradiance). This reformulation reduces computational costs while preserving the accuracy of global illumination.

This project implements the antiradiance method in C++ to explore its potential for efficient interactive global illumination.

2 Understanding the Paper

2.1 Overview of Antiradiance

The paper [1] reformulates the traditional rendering equation by introducing the concept of antiradiance. In the standard approach, the rendering equation involves the computation of radiance $L(x, \omega)$ using the geometry operator $\mathbf{G}$, which accounts for occlusions and shadows via explicit visibility checks. This

operator relies on a visibility function $\text{ray}(x, \omega)$ that determines the first front-facing point starting at x in direction $-\omega$. The reflection operator $\mathbf{K}$ then performs the shading integral:

$$(\mathbf{K}L_{\text{in}})(x, \omega) = \int_{\Omega_{n_x}} f(x; \omega_{\text{in}} \to \omega) L_{\text{in}}(x, \omega_{\text{in}}) \langle n_x \mid -\omega_{\text{in}} \rangle \, d\omega_{\text{in}}$$

where n_x is the normal at x, f is the BRDF, and $\langle \cdot \mid \cdot \rangle$ denotes the dot product. The incident radiance $L_{\text{in}}(x, \omega)$ is obtained through the geometry operator $\mathbf{G}$ as:

$$L_{\text{in}}(x, \omega) = (\mathbf{G}L)(x, \omega) = L(\text{ray}(x, \omega), \omega)$$

To simplify the computationally expensive visibility checks, the paper replaces $\mathbf{G}$ with an operator $\mathbf{U}$ that does not account for visibility:

$$L_{\text{in}}^{\text{unocc}}(x, \omega) = (\mathbf{U}L)(x, \omega) = \sum_{y \in \text{RAY}(x,\omega)} L(y, \omega)$$

This modification, however, introduces excess radiance, as $\mathbf{U}$ allows light to propagate through occluded surfaces. To correct this, the concept of antiradiance A is introduced, which serves to cancel out the extraneous light introduced by $\mathbf{U}$. The relation between $\mathbf{G}$ and $\mathbf{U}$ can be expressed as:

$$\mathbf{U}L = \mathbf{G}L + \mathbf{U}\mathbf{J}\mathbf{G}L$$

where $\mathbf{J}$ is a "go-through" operator that effectively acts as the identity, allowing light to pass through opaque objects:

$$(\mathbf{J}L_{\text{in}})(x, \omega) = L_{\text{in}}(x, \omega)$$

Thus, antiradiance A is defined as:

$$A = \mathbf{J}\mathbf{G}L$$

Replacing $\mathbf{G}L$ with its equivalent in terms of $\mathbf{U}$ and A, we get the recursive formulation:

$$A = \mathbf{J}\mathbf{U}(L - A)$$

This equation allows for the iterative computation of A, starting with $A = 0$, until convergence is reached.

2.2 Radiosity and Linear Systems

The implementation in `radiosity.cpp` follows this formulation by solving two coupled linear systems iteratively to compute both radiance L and antiradiance A. These are expressed as:

$$L = E + \mathbf{K}\mathbf{U}(L - A)$$

$$A = \mathbf{J}\mathbf{U}(L - A)$$

Here, E represents emissive values (analogous to L_e in traditional radiosity), $\mathbf{K}$ is the reflection operator defined as:

$$\mathbf{K}(x, x') = \frac{\rho(x)}{\pi} G(x, x')$$

where the form factor $G(x, x')$ is computed by considering the orientation and distance between triangles, but with visibility removed. The iterative solution alternates between computing L and A, following an asymmetric scheme where multiple steps of antiradiance propagation are applied after each radiance step until convergence, consistent with the approach described in the paper.

2.3 Handling Visibility and Antiradiance

The recursive nature of antiradiance computation is crucial for accurately canceling out the excessive radiance introduced by the unoccluded operator **U**. In the implementation, the go-through matrix **J** is initialized as an identity matrix via the `initializeGoThroughMatrix()` function. The function `computeRadiosityAntiradiance()` then iteratively calculates the antiradiance A using the relation:

$$A^{(j+1)} = \mathbf{JU}(L^{(i)} - A^{(j)})$$

The radiance L is updated accordingly:

$$L^{(i+1)} = E + \mathbf{KU}(L^{(i)} - A^{(j)})$$

These iterations continue until the difference between successive updates falls below a convergence threshold, ensuring that the solution accurately models the balance between radiance and antiradiance.

3 Implementation Details

3.1 Overview of Code Implementation

Our implementation computes radiosity and antiradiance within a given mesh. Key functions in the code include:

- **Form Factor Calculation** (`calculateFormFactors`): This function computes the form factors between all pairs of triangles, accounting for visibility and geometric relationships between them.
- **Hemisphere Discretization** (`discretizeHemisphere`): This function discretizes the hemisphere above each triangle to properly account for the directional distribution of incoming radiance.
- **Go-Through Matrix Initialization** (`initializeGoThroughMatrix`): This function initializes the "go-through" matrix **J**, which is represented as an identity matrix to simulate light transmission without reflection.
- **Radiosity and Antiradiance Computation** (`computeRadiosityAntiradiance`): This function iteratively solves for the radiance L and antiradiance A using a method similar to Jacobi iterations.

3.2 Structure of the C++ Code

The code is organized around two primary classes:

- `Mesh` **Class**: This class manages the mesh data, including vertices and triangles, and is responsible for loading and saving mesh data from or to files.
- `Radiosity` **Class**: This class handles the computation of both radiosity and antiradiance. It contains member variables for storing radiance, antiradiance, form factors, reflection matrices, and the go-through matrix. The `Radiosity` class operates on a `Mesh` object, modifying its data to reflect the computed global illumination effects.

3.3 Construction of Radiosity and Antiradiance Matrices

The construction of the radiosity and antiradiance matrices involves the following components:

- **Form Factor Matrix**: Constructed within the `calculateFormFactors` function, this matrix stores the geometric relationship between pairs of triangles. For each pair, samples are taken to estimate the contribution of one triangle to the illumination of the other, and these contributions are stored in the `formFactors` matrix.
- **Reflection Matrix**: This matrix is similar to the form factor matrix but is specifically used in the radiosity computation to model the reflection of light between surfaces.
- **Go-Through Matrix**: This matrix, initialized as an identity matrix in the function `initializeGoThroughMatrix`, is used in the antiradiance computation. It allows for the simulation of light propagation through surfaces without being affected by occlusions.

3.4 Steps for Computing Radiance and Antiradiance

The computation of radiance and antiradiance proceeds through the following steps:

1. **Initialization of Emissivity**: The initial emissivity values for each triangle are set based on the specific properties of the scene.
2. **Iterative Computation**:
   - **Radiance Update**: For each triangle, the radiance L is updated by considering the reflection of light from other triangles in the scene.
   - **Antiradiance Update**: Concurrently, the antiradiance A is computed by accounting for the contributions from the "go-through" operator $\mathbf{J}$.
   - **Convergence Check**: After each iteration, the solution is checked against a convergence threshold. The iterations continue until the radiance and antiradiance values stabilize within this threshold.
3. **Normalization**: After the iterative process, the computed radiance values are normalized to ensure they remain within physical bounds, typically between 0 and 1.

This iterative process ensures that the final computed values accurately represent global illumination in the scene, incorporating corrections for excess light using the antiradiance method.

3.5 Challenges

Several challenges were encountered along the way, notably the computational complexity of form factor calculations, which requires significant resources due to the quadratic scaling with the number of triangles, and degenerate triangles that cause form factor accuracy. The decision between floating-point and double precision was done in order to maintain numerical stability, as floating-point inaccuracies could lead to significant errors in lighting calculations. Memory management was another challenge, with large matrices risking memory leaks and inefficient memory use.

4 Results and Timings

4.1 Mesh Specifications

The implementation was tested on three different meshes with varying complexities. The specifications for each mesh are listed in Table 1.

Mesh	Vertices	Triangles
legoBrick.obj	412	223
room_radiosity_small.obj	4647	9264
bear.obj	7852	15596

Table 1: Specifications of the tested meshes.

4.2 Visual Output

The following figures present the before and after visual results obtained from the radiosity and antiradiance computations for each mesh.

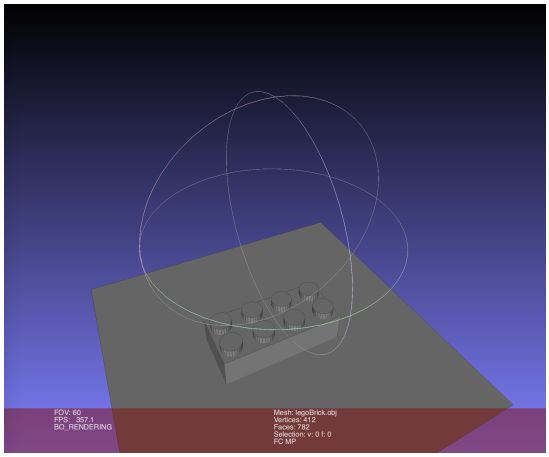

Figure 1: *
Before: legoBrick.obj

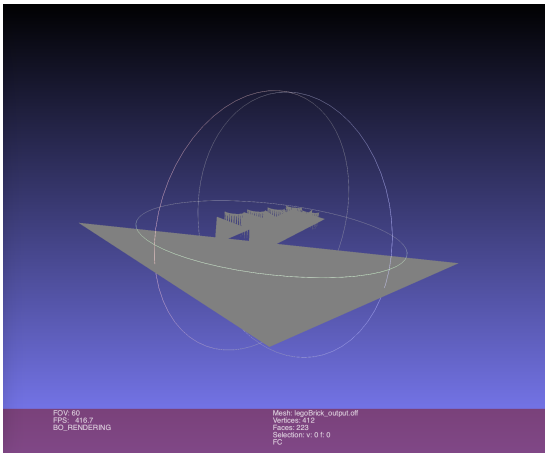

Figure 2: *
After: legoBrick.obj

Figure 3: Rendered output of legoBrick.obj before and after processing

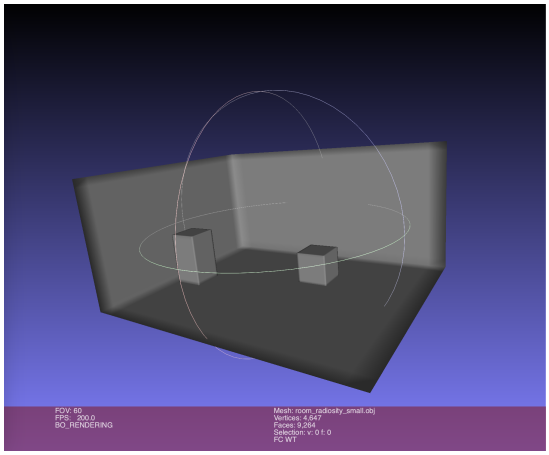

Figure 4: *
Before: room_radiosity_small.obj

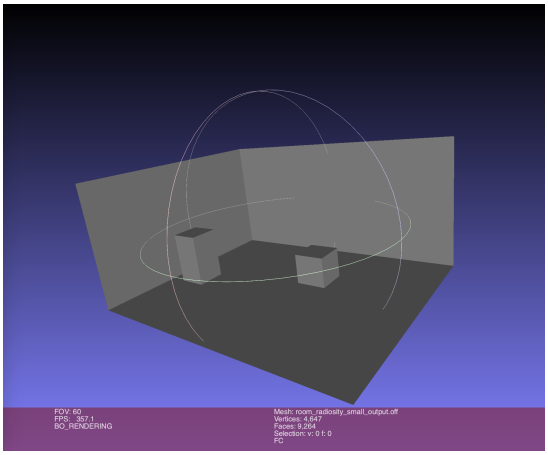

Figure 5: *
After: room_radiosity_small.obj

Figure 6: Rendered output of room_radiosity_small.obj before and after processing

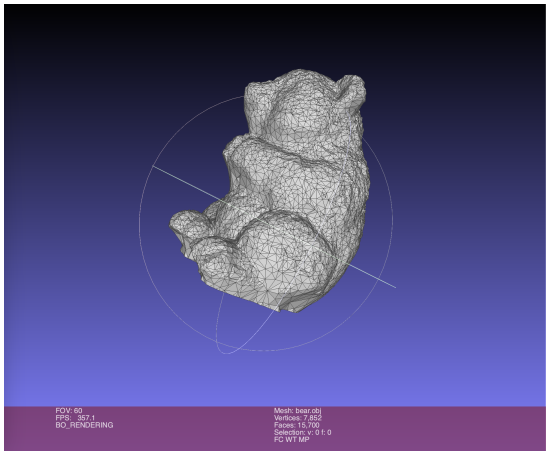

Figure 7: *
Before: bear.obj

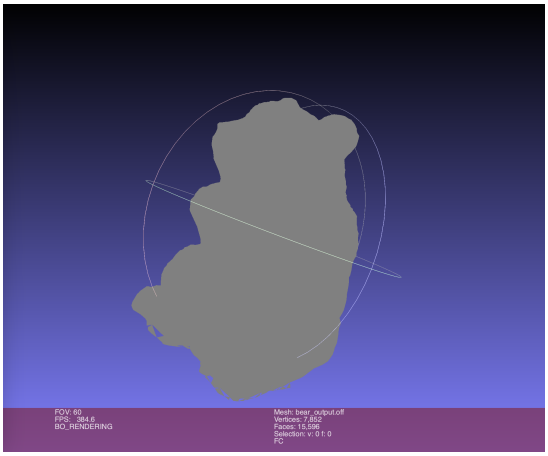

Figure 8: *
After: bear.obj

Figure 9: Rendered output of bear.obj before and after processing

4.3 Timing Analysis

The performance timings for each mesh were recorded for two different sample sizes: 15 and 20. The results are presented in Figure 10, which shows the processing times for all three meshes with separate curves for each sample size.

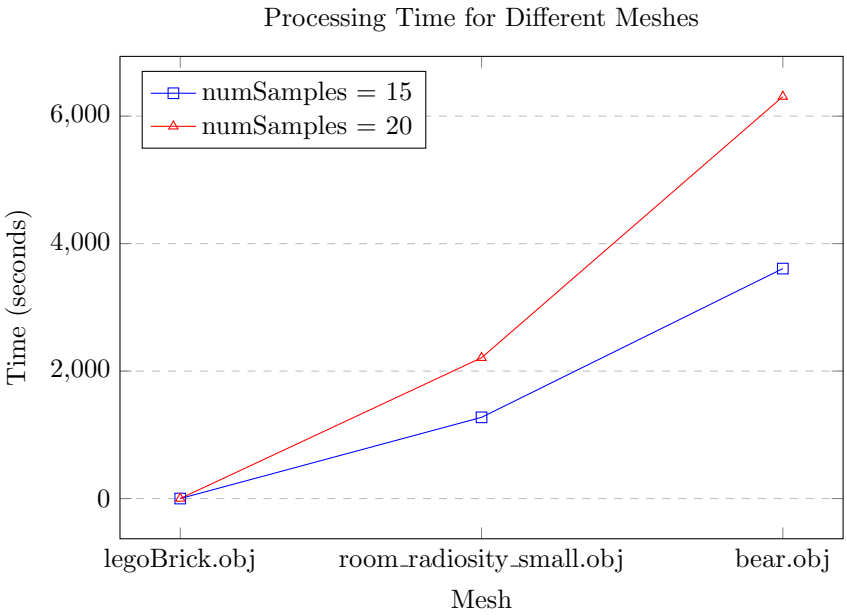

Figure 10: Processing time for each mesh with numSamples = 15 and numSamples = 20

For numSamples = 15, the legoBrick.obj mesh processed in 0 seconds, room_radiosity_small.obj in 1274.0 seconds, and bear.obj in 3608.0 seconds. For numSamples = 20, the times were 1.0 seconds, 2209.0 seconds, and 6307.0 seconds, respectively.

The results show that as the complexity of the mesh increases, the processing time also increases significantly. Additionally, increasing the number of samples further exacerbates the processing time, particularly for more complex meshes like room_radiosity_small.obj and bear.obj.

5 Potential Improvements

The function `validateFormFactors()` is designed to ensure that the computed form factors adhere to the principles of reciprocity and energy conservation. This function systematically checks that the sum of form factors for each triangle is close to 1, and that the form factors between any two triangles satisfy the reciprocity condition. The project encountered several challenges that indicate areas for potential improvement. As evidenced by the output images, the results suggest that the computed form factors contain significant errors. These inaccuracies likely stem from the complex nature of the form factor calculations and the challenges posed by degenerate triangles and choosing between floating-point precision vs. double precision. Further, the high error rates observed during validation and the visual artifacts in the rendered outputs suggest that the implementation requires more fine-grained debugging. Potential improvements would include refining the sampling strategy used in the form factor calculations.

6 Acknowledgements and Inspirations

This project is part of the course *Computer Graphics (CSE 306)* at École Polytechnique, where the lecture notes [2] by Nicolas Bonneel were essential in understanding the foundational concepts of radiosity and global illumination. The implementation in was also inspired by several GitHub repositories that provided valuable insights into the techniques used in radiosity and global illumination. References [3], [4], and [5] were particularly influential in implementing the class structures. Valuable discussions and insights into specific concepts, with the help of Ayman Khaleq, significantly contributed to the implementation.

References

[1] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Frédo Durand. Implicit visibility and antiradiance for interactive global illumination. *ACM Transactions on Graphics (TOG)*, 26(3):61–es, 2007.

[2] Nicolas Bonneel. *Computer Graphics – CSE 306*. École Polytechnique, 2024. Last update: March 27th, 2024. Accessed: 2024-08-24.

[3] Edward Kmett. mesh.h. https://github.com/ekmett/vr/blob/8d321730e9bbacddbe652e0fb501e82fb0b7d76b/mesh.h, 2015. Accessed: 2024-08-24.

[4] Bernhard Fritz. Radiosity.cpp. https://github.com/bernhardfritz/FortgeschritteneComputergrafik/blob/master/Radiosity/Radiosity.cpp, 2017. Accessed: 2024-08-24.

[5] githole. radiance.h. https://github.com/githole/edupt/blob/b925270bf2feb328f9e0881cefd361a59f99b773/radiance.h#L19, 2023. Accessed: 2024-08-24.