Final Project

IGR Project Report

Martin — Telecom Paris

February 19, 2026

Abstract

Implementation of a real-time rendering engine with rigid-body dynamics and a Monte Carlo path tracer.

Contents

1 Introduction

The objective of this project was to develop a sandbox environment capable of simulating complex optical phenomena while maintaining interactive physics. Unlike traditional rasterization-based pipelines, this engine utilizes a path-tracing approach to resolve global illumination, including soft shadows, reflections, and refractions.

2 Project Evolution and Implementation Timeline

The development of this engine followed an incremental approach, evolving from basic geometry processing to a full-scale hybrid simulator.

2.1 Phase 1: Geometry and Subdivision

The project began with the fundamental building block: the triangle. Early implementation focused on loading simple meshes and performing subdivision to increase geometric detail. We implemented the Loop subdivision scheme, which defines new vertex positions based on a weighted average of adjacent vertices.

$$v_{new} = (1 - n\beta)v_{old} + \beta \sum_{i=1}^{n} v_i \tag{1}$$

Here, v_{old} is the original vertex position, n represents the valence (number of connected edges) of the vertex, v_i are the positions of the neighboring vertices, and β is a weighting factor derived from n. This phase was crucial for understanding how to manage dynamic connectivity and vertex buffers before moving into simulation.

2.2 Phase 2: Physics Foundations

Initial physics experiments utilized a basic vertex-based solver. The original model simulated collisions by checking if any vertex of a cube intersected the ground plane ($y = 0$).

- **Early Model**: Simple force restitution without friction. It resulted in "bouncy" but unrealistic behavior where objects would rotate incorrectly or pass through each other.
- **Collision Challenges**: Implementing object-to-object collisions introduced significant jitter and "energy gain," where objects would shake until they exploded out of the scene.
- **Current Solution**: To solve this, we moved to the impulse-based solver described in Section 2. We also implemented `#pragma omp parallel for` to parallelize the collision detection phase. This allowed us to run the simulation at very high frequencies (small dt) with multiple sub-steps per frame, ensuring stability even during complex interactions.

2.3 Phase 3: From Raytracing to Path Tracing

The rendering pipeline underwent a major refactor after the first prototype.

- **Prototype**: A simple recursive raytracer that handled a single bounce (perfect reflection) and used a hardcoded sky gradient.
- **Architectural Refactor**: The code was restructured into a modular hierarchy:
  - `Renderer`: Handles the OpenGL context and Compute Shader dispatching.
  - `Scene`: Manages the collection of objects and light sources.

- `RigidSolver`: Encapsulates the physics logic, independent of the rendering.
- `Mesh/Object`: Handles geometry and material properties.

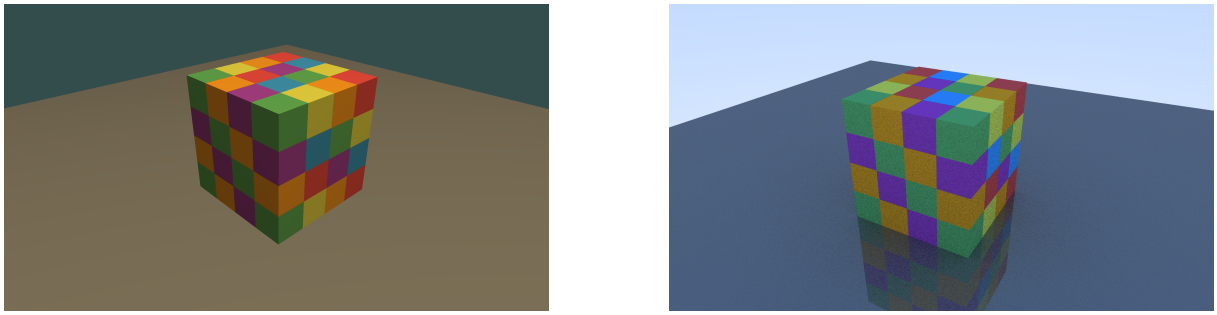

Figure 1: Left: The original rasterization output with Blinn-Phong shading. Right: The same geometry resolved via the Path Tracing pipeline with soft shadows and recursive reflections.

- **Monte Carlo Transition**: The final phase involved implementing the full Monte Carlo Path Tracer. This added support for diffuse inter-reflections, soft shadows, and the complex glare effects discussed in Section 5.

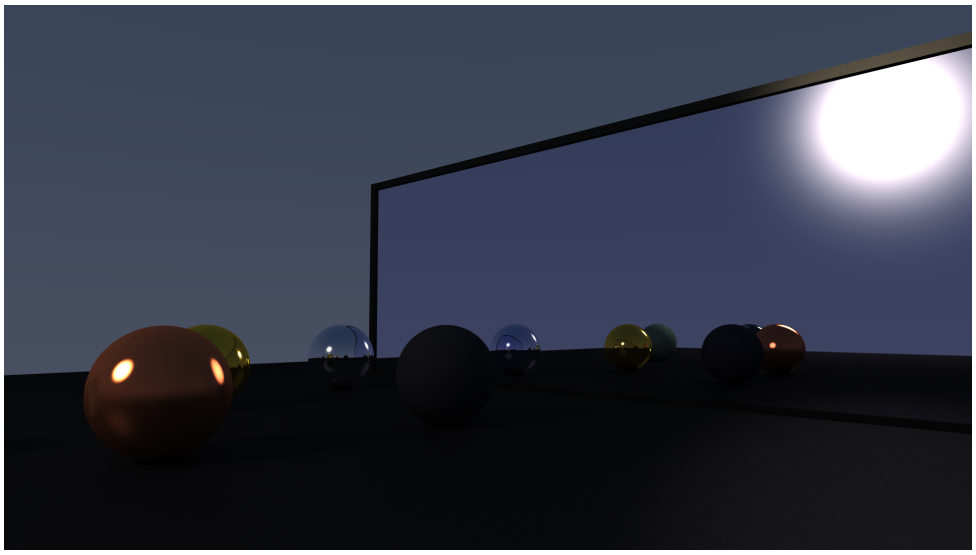

Figure 2: The engine's main interface showing various materials (Gold, Glass, Matte) reacting to a central light source.

3 Physics Engine Architecture

The physics module is based on a rigid-body solver using a discrete-time integration scheme.

3.1 Numerical Integration

We employ a semi-implicit Euler integration for both linear and angular components. The state of each object is defined by its position $\mathbf{x}$, velocity $\mathbf{v}$, orientation quaternion $\mathbf{q}$, and angular velocity ω.

$$\mathbf{v}_{t+dt} = \mathbf{v}_t + \frac{\mathbf{F}_{net}}{m} dt, \quad \mathbf{x}_{t+dt} = \mathbf{x}_t + \mathbf{v}_{t+dt} dt \tag{2}$$

In these equations, $\mathbf{F}_{net}$ is the sum of all forces acting on the body, m is the mass, and dt is the discrete time step. For rotations, we update the orientation using the angular velocity:

$$\mathbf{q}_{t+dt} = \text{normalize}\left(\mathbf{q}_t + \frac{1}{2}(\omega \otimes \mathbf{q}_t)dt\right) \tag{3}$$

where $\otimes$ denotes quaternion multiplication. The integration step is parallelized using OpenMP to handle large object counts efficiently:

```
#pragma omp parallel for
for (int i = 0; i < (int)objects.size(); ++i) {
    Object* obj = objects[i];
    if (obj->fixedObject) continue;
    obj->position += obj->velocity * deltaTime;

    float angVelLen = glm::length(obj->angularVelocity);
    if (angVelLen > 0.0001f) {
        glm::vec3 axis = obj->angularVelocity / angVelLen;
        float angle = angVelLen * deltaTime;
        glm::quat deltaRot = glm::angleAxis(angle, axis);
        obj->orientation = glm::normalize(deltaRot * obj->orientation);
    }
}
```

3.2 Sub-stepping and Temporal Stability

A critical feature of the physics engine is the use of a fixed time step with an accumulator. Instead of simulating with a variable Δt tied to the frame rate (which can lead to non-deterministic behavior and instabilities during frame drops), we use a constant discrete step $dt_{fixed} = 3ms$.

For each rendering frame, we calculate multiple iterations of the physics update:

$$N_{steps} = \left\lfloor \frac{\Delta t_{frame} + \text{remainder}}{dt_{fixed}} \right\rfloor \tag{4}$$

This sub-stepping approach is essential for several reasons:

- **High-Speed Collisions**: Small objects moving at high velocities might "tunnel" through thin geometry if the time step is too large. Frequent updates ensure the collision detection phase catches these intersections.
- **Energy Conservation**: Semi-implicit Euler integration is relatively stable but can still gain energy if the steps are large. Reducing the step size minimizes this error.
- **Convergence of the Impulse Solver**: In complex scenarios like physical stacking, multiple collisions affect the same body. Smaller steps allow the impulse propagation to stabilize more effectively across the structure.

3.3 Collision Detection and Response

The solver implements an impulse-based model for collision response. When a collision is detected at a contact point with normal $\mathbf{n}$, the required impulse J to satisfy the law of restitution e is calculated as:

$$J = \frac{-(1+e)\mathbf{v}_{rel} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}(\frac{1}{m_A} + \frac{1}{m_B}) + (\mathbf{I}_A^{-1}(\mathbf{r}_A \times \mathbf{n}) \times \mathbf{r}_A + \mathbf{I}_B^{-1}(\mathbf{r}_B \times \mathbf{n}) \times \mathbf{r}_B) \cdot \mathbf{n}} \tag{5}$$

Here, m_A and m_B are the masses of the colliding bodies, $\mathbf{I}_A^{-1}$ and $\mathbf{I}_B^{-1}$ are their respective inverse inertia tensors in world space, $\mathbf{r}_A$ and $\mathbf{r}_B$ are the vectors from the center of mass to the contact point, and $\mathbf{v}_{rel}$ is the relative velocity at the point of impact.

4 Vertex Animation and GPU Synchronization

One of the unique features of the engine is the support for dynamic, CPU-animated meshes within a ray-tracing context. In Scene 5 (Sea Scene), we proceduralize water waves on the host side and synchronize the vertex data with the GPU every frame.

4.1 Wave Synthesis

We use a sum of Gerstner-like sinusoids to perturb the grid height. To give the water a "rippled" or "peaky" appearance, we utilize an absolute sine formulation:

```
float h = 0.0f;
h += 0.4f * (1.0f - std::abs(std::sin(0.15f * x + 1.2f * t)));
h += 0.25f * (1.0f - std::abs(std::sin(0.12f * z + 0.8f * t + 1.0f)));
h += 0.15f * (1.0f - std::abs(std::sin(0.08f * (x + z) + 1.5f * t)));
```

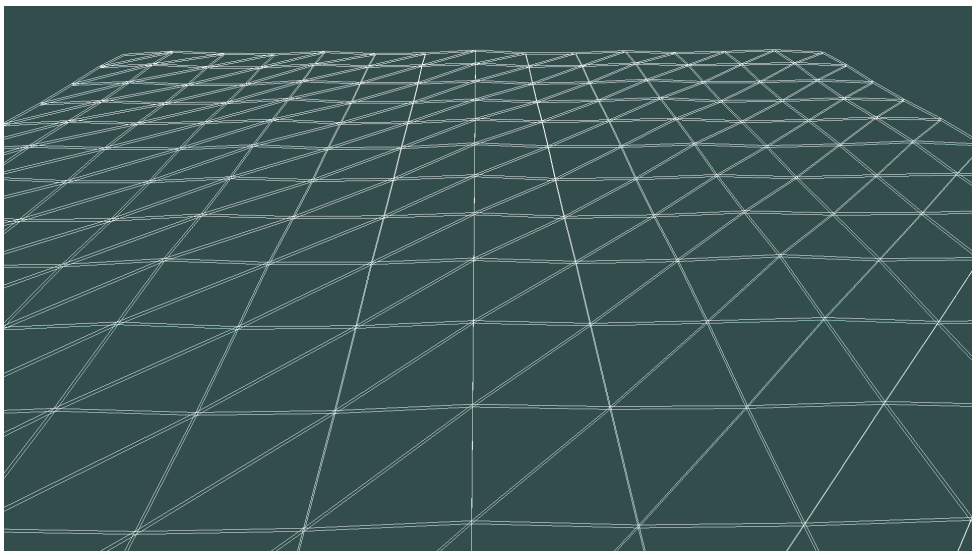

Figure 3: The dynamic sea mesh using a 5x5 grid resolution to maintain optimization while enabling real-time vertex displacement.

4.2 Data Synchronization (SSBOs)

Because the Compute Shader needs current triangle positions for intersection testing, we must update the Shader Storage Buffer Objects (SSBO). This is handled by mapping the vertex data and re-calculating normals on the CPU before a `glBufferSubData` call.

5 GPU Path Tracing Pipeline

Rendering is offloaded to the GPU via OpenGL 4.5 Compute Shaders. All scene geometry, including mesh triangles and object properties, is uploaded to SSBOs.

5.1 Monte Carlo Integration

The renderer solves the rendering equation by sampling paths through the scene. For each pixel, multiple rays are jittered for anti-aliasing. The governing equation for the outgoing radiance L_o is:

$$L_o(\mathbf{p}, \omega_o) = \int_{\Omega} L_i(\mathbf{p}, \omega_i) f_r(\mathbf{p}, \omega_i, \omega_o) \cos\theta_i d\omega_i \tag{6}$$

where $\mathbf{p}$ is the hit point, ω_o the outgoing direction, L_i the incoming radiance from direction ω_i, f_r the Bidirectional Reflectance Distribution Function (BRDF), and Ω the hemisphere of possible incoming light directions. To handle the hemisphere sampling required for diffuse reflections, we implement a cosine-weighted distribution.

```
// Cosine-weighted hemisphere sampling
vec3 randomInHemisphere(vec3 normal) {
    float u1 = random();
    float u2 = random();
    float r = sqrt(u1);
    float theta = 2.0 * 3.14159 * u2;
    vec3 localDir = vec3(r * cos(theta), r * sin(theta), sqrt(max(0.0, 1.0 - u1)
    ));

    vec3 up = abs(normal.z) < 0.999 ? vec3(0,0,1) : vec3(1,0,0);
    vec3 tangent = normalize(cross(up, normal));
    vec3 bitangent = cross(normal, tangent);
    return normalize(tangent * localDir.x + bitangent * localDir.y + normal *
    localDir.z);
}
```

5.2 Ray-Triangle Intersection

We utilize the Möller-Trumbore algorithm for its efficiency and low memory footprint, as it avoids storing plane equations for every triangle.

```
bool intersectTriangle(Ray ray, vec3 v0, vec3 v1, vec3 v2, out float t) {
    vec3 edge1 = v1 - v0, edge2 = v2 - v0;
    vec3 h = cross(ray.direction, edge2);
    float a = dot(edge1, h);
    if (a > -0.00001 && a < 0.00001) return false;
    float f = 1.0 / a;
    vec3 s = ray.origin - v0;
    float u = f * dot(s, h);
    if (u < 0.0 || u > 1.0) return false;
    vec3 q = cross(s, edge1);
    float v = f * dot(ray.direction, q);
    if (v < 0.0 || u + v > 1.0) return false;
    t = f * dot(edge2, q);
    return t > 0.00001;
}
```

5.3 Temporal Accumulation

To reduce the variance (noise) without skyrocketing the per-frame sample count, we implement a temporal accumulation buffer. If the scene is static, each new frame is averaged with previous ones, effectively increasing the samples per pixel over time.

```
vec3 finalColor = currentFrameColor;
if (frameCounter > 1) {
    vec3 prevColor = imageLoad(accumulationBuffer, texelCoord).rgb;
```

```
    // Iterative average
    finalColor = mix(prevColor, currentFrameColor, 1.0 / float(frameCounter));
}
imageStore(accumulationBuffer, texelCoord, vec4(finalColor, 1.0));
```

5.4 Fresnel and Dielectric Materials

To simulate water and glass, we use the Schlick approximation for the Fresnel coefficient $R(\theta)$:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5, \quad R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{7}$$

where θ is the angle of incidence, and n_1, n_2 are the refractive indices of the two media. Additionally, we implement Beer-Lambert absorption for rays traveling inside transparent volumes:

$$I(\text{dist}) = I_0 \cdot e^{-\sigma \cdot \text{dist}} \tag{8}$$

where I_0 is the initial light intensity, σ is the absorption coefficient (related to the material color), and *dist* is the distance traveled within the medium.

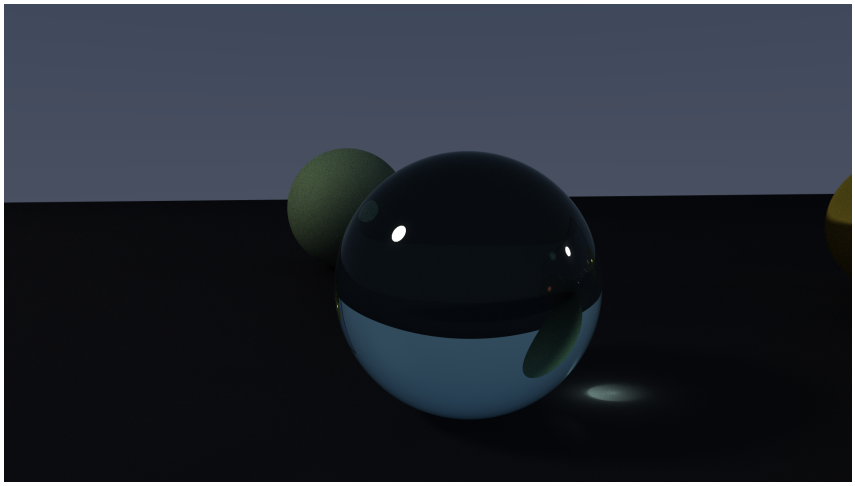

Figure 4: Implementation of dielectric materials using Schlick's approximation for reflections and Beer-Lambert absorption for volume thickness.

6 Optical Effects and Challenges

6.1 Volumetric Glare Approximation

One of the primary difficulties was simulating atmospheric scattering (glare) without the heavy cost of volumetric path marching. We implemented a power-law approximation based on the angular distance between the ray direction $\mathbf{d}$ and the vector to the light $\mathbf{l}$.

This effect simulates the "glow" seen around strong light sources in a humid or dusty environment. By separating the core intensity from the atmospheric halo, we can create a spherical illumination volume even with purely angular math.

```
// Simplified Glare Loop logic
for (int i = 0; i < objectCount; i++) {
    if (objects[i].emissive.w > 0.0) {
        float dotL = dot(ray.direction, dirToLight);
        if (dotL > 0.0) {
            float intensity = pow(dotL, 4096.0) * 2.0; // Sharp Light Core
```

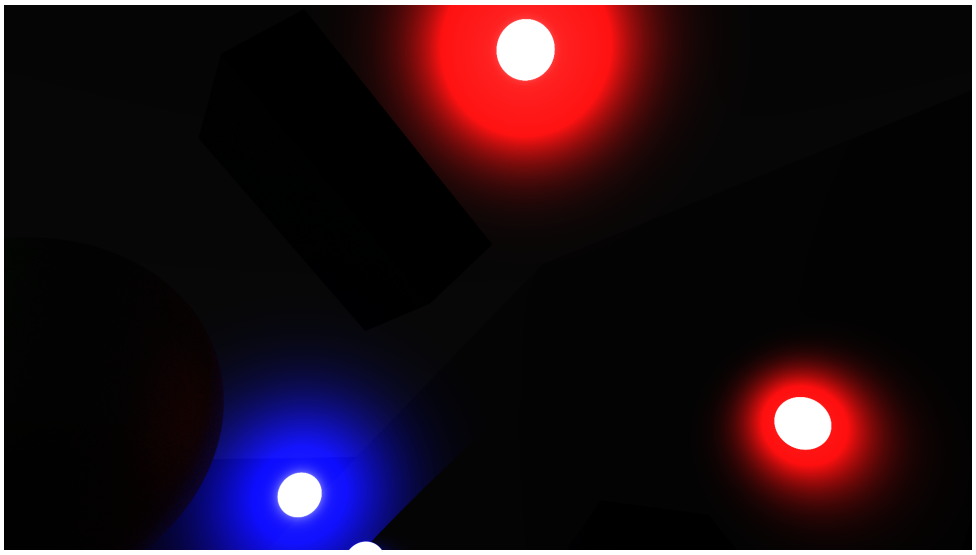

Figure 5: Visualizing the depth-aware glare approximation. The glow is masked by geometry to prevent it from appearing in front of occluding objects.

```
            float halo = pow(dotL, 128.0) * 0.07;       // Diffuse Atmospheric
    Halo
            sampleColor += throughput * lightColor * (intensity + halo);
        }
    }
}
```

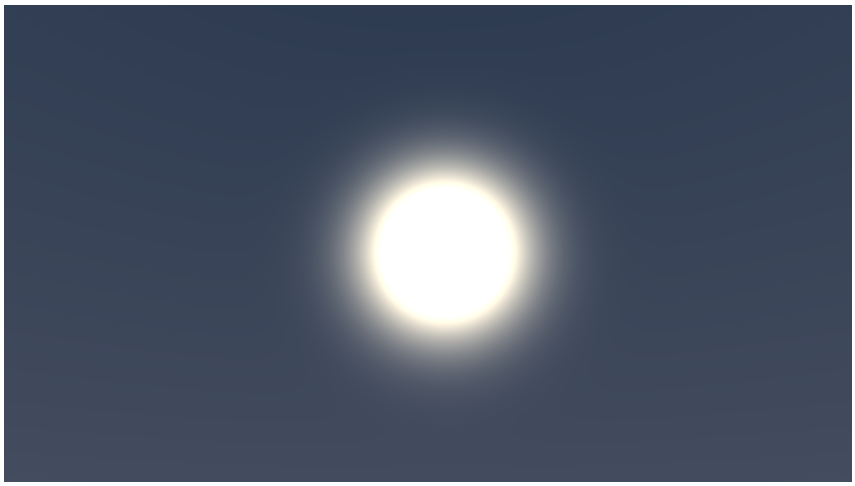

Figure 6: Testing atmospheric scattering using angular distance. The glare intensity follows a power-law distribution to simulate light bleeding.

6.2 Implementation Hardships

Physics Fine-tuning: Stability was the greatest challenge. Small errors in the inertia tensor or the collision epsilon caused objects to "jitter" or explode. Bypassing the solver for animated meshes (like the sea) was necessary to prevent unwanted collision resolution between fixed planes and moving vertices.

Noise and Convergence: As a Monte Carlo-based system, the engine suffers from high-frequency noise. While we implemented accumulation, it requires low movement to converge.

Reducing the triangle count for the sea grid (grid resolution of 5x5) was an essential optimization to maintain frame rates above 30 FPS.

7 Scene Gallery and Visual Analysis

To evaluate the performance and correctness of the engine, we designed five distinct scenes, each targeting a specific subset of the system's capabilities.

7.1 Scene 1: Rigid Body Stress Test

This scene is dedicated to validating the physics solver. It consists of a $4 \times 4 \times 4$ "cube of cubes" (64 individual rigid bodies) stacked in a grid.

- **Objective**: Test stable stacking and multi-object collision resolution.
- **Interaction**: The user can "shoot" high-velocity spheres into the structure to observe the impulse propagation and subsequent collapse.
- **Key Result**: Demonstrates the stability of the impulse-based solver and the efficiency of the OpenMP-parallelized collision loops.

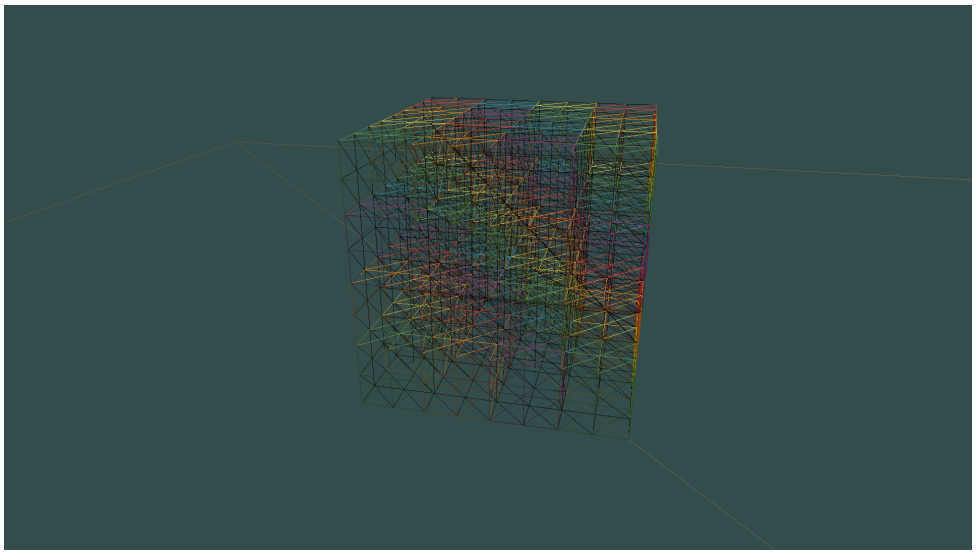

Figure 7: Physics stress test: 64 rigid bodies interacting simultaneously with minimal jitter.

7.2 Scene 2: Material Properties and Monte Carlo Noise

A showcase of the PBR material system, featuring various spheres (Gold, Silver, Glass, and Matte) set before a large mirror plane and illuminated by an artificial sun.

- **Objective**: Visualize the difference between specular and rough surface transport.
- **Visual Analysis**: This scene highlights the "noise" or "grain" inherent in Monte Carlo integration, particularly on rough materials where the probability density function (PDF) must sample a wider hemisphere.

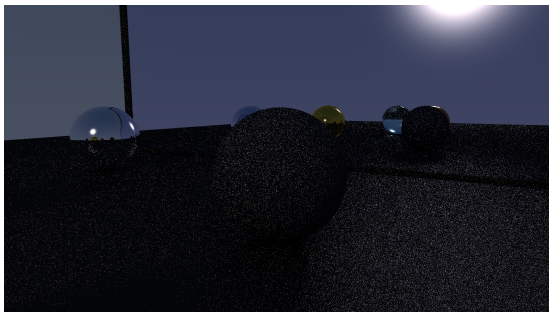
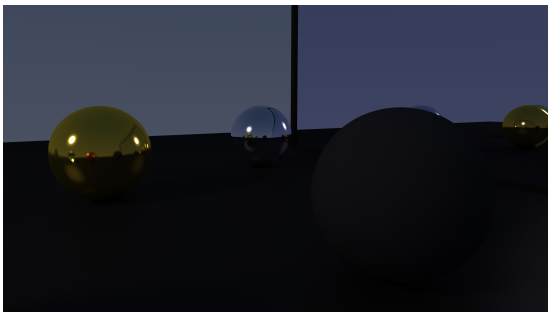

Figure 8: Left: Monte Carlo noise visible during early convergence on rough surfaces. Right: Converged materials after temporal accumulation.

7.3 Scene 3: Recursive Mirror Room

A single mesh object placed inside a "box" where all walls are perfect mirrors.

- **Objective**: Study the effect of recursion depth (ray bounces) on global illumination.
- **Comparison**: By toggling between 2 bounces and 12 bounces, one can see the "hall of mirrors" effect emerge as light paths are allowed to travel further through the scene before termination.

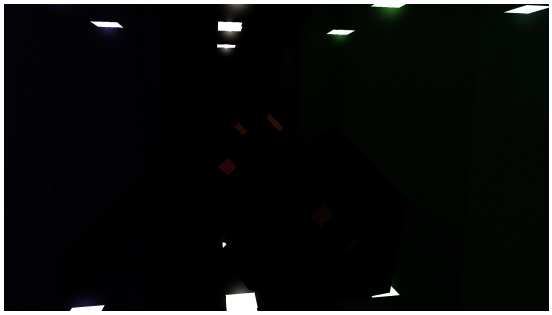
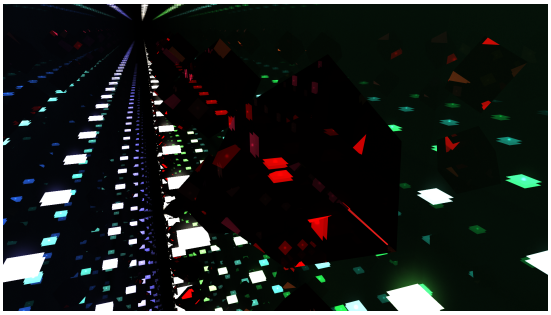

Figure 9: Visualizing ray depth. Left: 5 bounces results in black surfaces where light terminates early. Right: 50 bounces enables the "Hall of Mirrors" effect.

7.4 Scene 4: Low-Light Specular Response

Set in a void with a dark sky, this scene features one central sphere and three dim, colored light sources.

- **Objective**: Verify the photometric accuracy of light reflection on mirrors.
- **Observation**: The mirror-like sphere correctly picks up the three distinct light sources, proving that the bounce logic correctly handles light transport even in low-energy environments.

7.5 Scene 5: The Ocean Scene

The most complex scene, combining vertex animation, dual-layer transparency, and a high-intensity emissive sun near the horizon.

- **Objective**: Stress-test the hybrid nature of the engine (Animation + Ray Tracing).

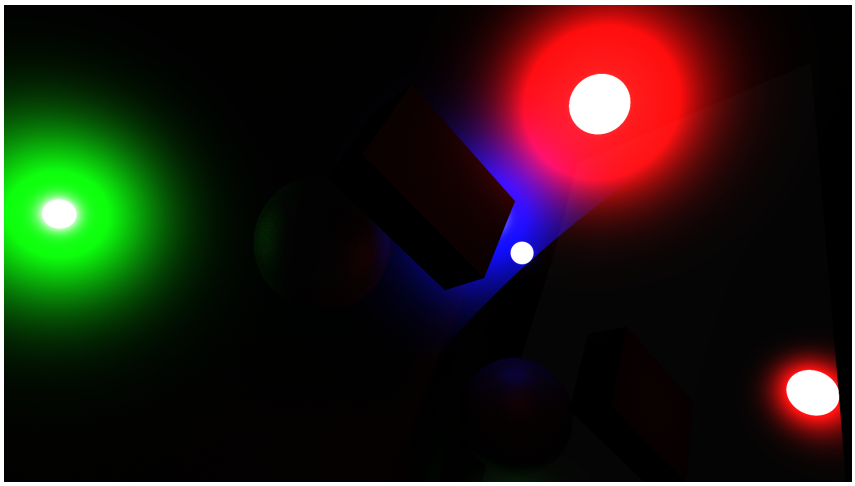

Figure 10: Testing specular reflection under colored light sources. The sphere correctly reflects the discrete light positions.

- **Optical Complexity**: This scene brings to light the difficulty of the glare approximation. Managing the glare intensity so that it reflects in the moving waves without "bleeding" through the objects is the primary challenge here.

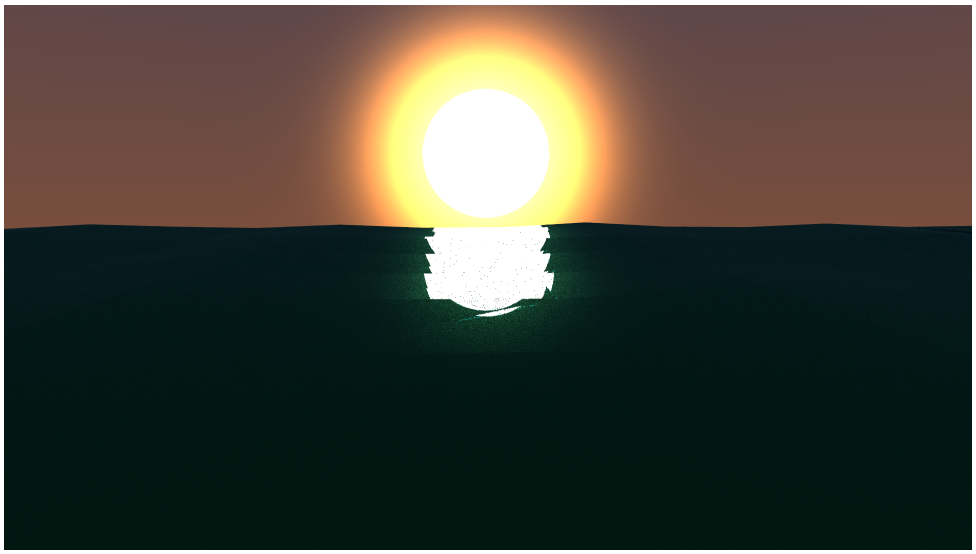

Figure 11: Final result of the Sea Scene, featuring animated waves, reflections of the glare, and dual-layer water depth.

8 Performance Analysis and Results

The performance of the path tracer is primarily limited by the number of samples per pixel (SPP) and the recursion depth (bounces).

The "Firefly" artifacting (isolated bright pixels) was a recurring issue when dealing with high-intensity emissive sources. This was partially mitigated using a clamping threshold on the sample contribution, though at the expense of energy conservation.

Configuration	Samples	Max Bounces	Frame Time (ms)
Draft Mode	1	2	2.2
Interactive	4	6	7.5
High Quality	16	12	29.8

Table 1: Rendering performance on a high-end Linux desktop (Scene 3 - Mirror Scene).

9 Conclusion

This project demonstrates the viability of hybrid engines. While physically-based effects like subsurface scattering and true volumetric glare are difficult to implement in real-time, mathematical approximations can yield visually compelling results for educational and research purposes.

References

[1] Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann.

[2] Baraff, D. (1997). *An Introduction to Rigid Body Simulation*. SIGGRAPH Course Notes.