Real-time Global Illumination via Radiance Cascades A 2D/3D Implementation Study

LI YUETONG, DING CHANGRUI, and YANG HONGYI, ShanghaiTech University

1 INTRODUCTION

Radiance Cascades [1] is a novel real-time global illumination technique that enables efficient light propagation in dynamic scenes. The core idea is to place probes throughout the scene and store light information at multiple scales using cascaded grids. Light information propagates from finer to coarser cascades, providing fast approximate global illumination without expensive ray tracing.

This project implements Radiance Cascades in two distinct scenarios:

(1) **2D Real-time Implementation**: Users can interactively draw light sources using a brush tool. The system maintains Signed Distance Field (SDF) representations and real-time updates to SSDF textures, enabling instant global illumination feedback as users paint light sources onto the scene.
(2) **3D Offline/Real-time Hybrid Approach**: Rather than distributing probes throughout the 3D volume, we place probes only on object surfaces. A global radiance atlas is computed offline via raytracing, which is then merged across different cascade levels. This atlas is used for real-time rendering with dynamic camera control.

Both implementations demonstrate how Radiance Cascades can scale from simple 2D scenarios to complex 3D environments while maintaining interactive frame rates.

2 IMPLEMENTATION DETAILS

2.1 2D Real-time Implementation

2.1.1 System Architecture. The 2D implementation uses a multi-pass rendering pipeline with ping-pong framebuffer objects (FBOs) for efficient data flow. The main components are:

(1) **SDF Texture (Ping-Pong)**: Two RGBA32F textures maintain signed distance field information across frames
(2) **Cascade Textures**: Four levels of cascaded textures store radiance at different scales
(3) **Merge Textures**: Intermediate textures for hierarchical merging of cascade levels
(4) **Radiance Field Texture**: Final smoothed radiance output

2.1.2 Resource Initialization. The initialization process sets up all necessary GPU resources:

Listing 1. Resource initialization for 2D cascades

```
void initResources() {
    // Initialize ping-pong SDF buffers
    for (int i = 0; i < 2; i++) {
        glGenFramebuffers(1, &sdfFBO[i]);
        glGenTextures(1, &sdfTex[i]);
        glBindTexture(GL_TEXTURE_2D, sdfTex[i]);
        // Use RGBA32F for precision
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
                     WIDTH, HEIGHT, 0, GL_RGBA,
                     GL_FLOAT, NULL);
        setTexParams();
        glBindFramebuffer(GL_FRAMEBUFFER, sdfFBO[i
            ]);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
                               GL_COLOR_ATTACHMENT0
                                   ,
                               GL_TEXTURE_2D,
                                   sdfTex[i], 0);

        // Initialize with black color and large
            distance
        float clearColor[] = {0.0f, 0.0f, 0.0f,
            10000.0f};
        glClearBufferfv(GL_COLOR, 0, clearColor);
    }

    // Create cascade and merge textures for each
        level
    for (int i = 0; i < numCascades; i++) {
        // Cascade textures
        glGenFramebuffers(1, &cascadeFBOs[i]);
        glGenTextures(1, &cascadeTexs[i]);
        glBindTexture(GL_TEXTURE_2D, cascadeTexs[i
            ]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
                     WIDTH, HEIGHT, 0, GL_RGBA,
                         GL_FLOAT, NULL);
        setTexParams();
        glBindFramebuffer(GL_FRAMEBUFFER,
            cascadeFBOs[i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
            GL_COLOR_ATTACHMENT0,
                               GL_TEXTURE_2D,
                                   cascadeTexs[i],
                                    0);

        // Merge textures
        glGenFramebuffers(1, &mergeFBOs[i]);
        glGenTextures(1, &mergeTexs[i]);
        glBindTexture(GL_TEXTURE_2D, mergeTexs[i])
            ;
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
```

```
                    WIDTH, HEIGHT, 0, GL_RGBA,
                        GL_FLOAT, NULL);
        setTexParams();
        glBindFramebuffer(GL_FRAMEBUFFER,
            mergeFBOs[i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
            GL_COLOR_ATTACHMENT0,
                                GL_TEXTURE_2D,
                                    mergeTexs[i],
                                    0);
    }
}
```

2.1.3 Multi-Pass Rendering Pipeline. The 2D implementation consists of five main passes:

Pass 1: Brush Accumulation When the user draws with the brush, the brush color is accumulated onto the SDF texture:

Listing 2. Brush rendering pass

```
int sdfWriteIdx = 1 - sdfReadIdx;
glBindFramebuffer(GL_FRAMEBUFFER, sdfFBO[
    sdfWriteIdx]);
brushShader.use();
brushShader.setVec2("resolution", glm::vec2(WIDTH,
     HEIGHT));
brushShader.setVec2("brushPos1", prevBrushPos);
brushShader.setVec2("brushPos2", brushPos);
brushShader.setFloat("brushRadius", brushRadius);
brushShader.setVec3("brushColour", brushColour);
brushShader.setBool("isDrawing", isDrawing);
brushShader.setBool("isErasing", eraseMode);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, sdfTex[sdfReadIdx]);
brushShader.setInt("sdfTexture", 0);
drawQuad();
sdfReadIdx = sdfWriteIdx;
```

Pass 2: Scene SDF Generation A fresh SDF is computed from the current brush state:

Listing 3. SDF generation pass

```
glBindFramebuffer(GL_FRAMEBUFFER, tmpSdfFBO);
sdfShader.use();
sdfShader.setVec2("resolution", glm::vec2(WIDTH,
    HEIGHT));
sdfShader.setVec2("brushPos", brushPos);
sdfShader.setFloat("brushRadius", brushRadius);
sdfShader.setVec3("brushColour", brushColour);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, sdfTex[sdfReadIdx]);
sdfShader.setInt("sdfTexture", 0);
drawQuad();
```

Pass 3: Cascade Computation For each cascade level, we compute radiance by tracing rays from probe positions:

Listing 4. Cascade computation pass

```
computeShader.use();
computeShader.setVec2("cascadeResolution",
                      glm::vec2(WIDTH, HEIGHT));
computeShader.setVec2("sceneResolution",
                      glm::vec2(WIDTH, HEIGHT));
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tmpSdfTex);
for (int i = 0; i < numCascades; i++) {
    glBindFramebuffer(GL_FRAMEBUFFER, cascadeFBOs[
        i]);
    computeShader.setInt("cascadeLevel", i);
    drawQuad();
}
```

The cascade computation shader traces rays from each probe position. Each pixel in the cascade texture represents a directional sample at a probe location:

Listing 5. Cascade computation fragment shader core logic

```
void main() {
  vec2 pixelIndex = (gl_FragCoord.xy - 0.5);
  CascadeInfo info = Cascade_GetInfo(cascadeLevel)
      ;
  ProbeIndex cascadeIndex = ProbeIndex_Create(
      pixelIndex, info);
  ProbeAABB aabb = ProbeAABB_Create(cascadeIndex,
      info);

  CascadePixelIndex coordsInCascade =
      CascadePixelIndex(ivec2(pixelIndex - aabb.
          min));
  float angleRadians = Angle_FromCascadeIndex(
      coordsInCascade, info);

  vec2 rayDirection = vec2(cos(angleRadians),
                           sin(angleRadians));
  vec2 rayOrigin = aabb.center * sceneResolution /
                   cascadeResolution;

  vec4 radiance = SampleRadiance_SDF(
      sdfTexture, sceneResolution, rayOrigin,
      rayDirection, info);

  FragColor = radiance;
}
```

Pass 4: Cascade Merging Cascades are merged in reverse order (coarse to fine) to propagate light information:

Listing 6. Cascade merging pass

```
mergeShader.use();
mergeShader.setVec2("cascadeResolution",
                    glm::vec2(WIDTH, HEIGHT));
mergeShader.setVec2("sceneResolution",
                    glm::vec2(WIDTH, HEIGHT));
mergeShader.setInt("numCascadeLevels", numCascades
    );
for (int i = numCascades - 1; i >= 0; i--) {
```

```
    glBindFramebuffer(GL_FRAMEBUFFER, mergeFBOs[i
        ]);
    mergeShader.setInt("currentCascadeLevel", i);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, cascadeTexs[i]);

    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D,
        (i == numCascades - 1) ? blackTex :
            mergeTexs[i + 1]);

    drawQuad();
}
```

Pass 5: Radiance Field Integration and Final Composition The final radiance field is computed via angular integration, which smooths the discrete probe directional samples into continuous illumination:

Listing 7. Radiance field and final pass

```
// Angular integration pass
glBindFramebuffer(GL_FRAMEBUFFER, radianceFieldFBO
    );
radianceShader.use();
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mergeTexs[0]);
drawQuad();

// Final composition to screen
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT);
finalShader.use();
finalShader.setVec2("brushPos", brushPos);
finalShader.setFloat("brushRadius", brushRadius);
finalShader.setVec3("brushColour", brushColour);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, radianceFieldTex);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, tmpSdfTex);
drawQuad();
```

2.1.4 Angular Integration. The radiance field shader performs bilinear interpolation over probe positions and uses the stored directional samples:

Listing 8. Radiance field fragment shader

```
// Helper function: get average radiance of a
    probe
vec4 GetProbeRadiance(ivec2 probeCoords,
                      CascadeInfo info) {
  vec2 probeBasePixels = vec2(probeCoords) *
                         float(info.dimensions);

  vec4 sum = vec4(0.0);

  for (int i = 0; i < info.dimensions; ++i) {
    for (int j = 0; j < info.dimensions; ++j) {
      vec2 samplePos = probeBasePixels + vec2(i, j
          ) + 0.5;
      sum += texture(mergedCascade0Texture,
                     samplePos / cascadeResolution
                         );
    }
  }
  return sum / float(info.dimensions * info.
      dimensions);
}

void main() {
  CascadeInfo ci0 = Cascade_GetInfo(0);

  float spacing = float(ci0.dimensions);
  vec2 currentPosInProbeGrid =
      gl_FragCoord.xy / spacing - 0.5;

  // Find adjacent probe indices
  ivec2 p00 = ivec2(floor(currentPosInProbeGrid));
  ivec2 p10 = p00 + ivec2(1, 0);
  ivec2 p01 = p00 + ivec2(0, 1);
  ivec2 p11 = p00 + ivec2(1, 1);

  // Get average radiance from probes
  vec4 r00 = GetProbeRadiance(p00, ci0);
  vec4 r10 = GetProbeRadiance(p10, ci0);
  vec4 r01 = GetProbeRadiance(p01, ci0);
  vec4 r11 = GetProbeRadiance(p11, ci0);

  // Bilinear interpolation
  vec2 f = fract(currentPosInProbeGrid);
  vec4 r0 = mix(r00, r10, f.x);
  vec4 r1 = mix(r01, r11, f.x);
  vec4 finalRadiance = mix(r0, r1, f.y);

  FragColor = finalRadiance;
}
```

2.2 3D Implementation

2.2.1 Offline Radiance Baking to Atlas. The 3D approach differs fundamentally from the 2D implementation. Instead of distributing probes throughout the volume, we use a *radiance atlas* that stores probe lighting information in a 2D texture (size: 1024×6144).

Listing 9. 3D resource initialization

```
void initResources() {
    // Initialize two FBOs and textures for ping-
        pong
    for (int i = 0; i < 2; i++) {
        glGenFramebuffers(1, &atlasFBO[i]);
        glGenTextures(1, &atlasTex[i]);
```

```
        glBindTexture(GL_TEXTURE_2D, atlasTex[i]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
                     ATLAS_WIDTH, ATLAS_HEIGHT, 0,
                     GL_RGBA, GL_FLOAT, NULL);

        glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MAG_FILTER,
                            GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_S,
                        GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_T,
                        GL_CLAMP_TO_EDGE);

        // Initialize with zeros and generate
            mipmaps
        std::vector<float> emptyData(
            ATLAS_WIDTH * ATLAS_HEIGHT * 4, 0.0f);
        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
                        ATLAS_WIDTH, ATLAS_HEIGHT,
                        GL_RGBA, GL_FLOAT,
                            emptyData.data());
        glGenerateMipmap(GL_TEXTURE_2D);

        glBindFramebuffer(GL_FRAMEBUFFER, atlasFBO
            [i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER,
                               GL_COLOR_ATTACHMENT0
                                   ,
                               GL_TEXTURE_2D,
                                   atlasTex[i], 0)
                                   ;
    }
}
```

2.2.2 Offline Baking Pass. The baking process computes global illumination and stores it in the atlas. This involves ray tracing to determine probe visibility and light contributions:

Listing 10. 3D baking pipeline

```
// Pass 1: Radiance Baking (offline computation)
glBindFramebuffer(GL_FRAMEBUFFER, atlasFBO[
    atlasWriteIdx]);
glViewport(0, 0, ATLAS_WIDTH, ATLAS_HEIGHT);
bakingShader.use();
bakingShader.setFloat("iTime", currentTime);

// Bind previous frame texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, atlasTex[atlasReadIdx
    ]);
drawQuad();

// Generate mipmaps for hierarchical sampling
glBindTexture(GL_TEXTURE_2D, atlasTex[
    atlasWriteIdx]);
glGenerateMipmap(GL_TEXTURE_2D);
```

2.2.3 Offline Ray Tracing and Cascade Merging. The 3D baking shader performs weighted sampling across multiple cascade levels. The process involves:

(1) Computing probe positions on object surfaces
(2) Tracing rays from each probe in all directions
(3) Accumulating contributions from previously computed cascades
(4) Storing results in the radiance atlas

Listing 11. 3D weighted sampling for cascade merging (excerpt)

```
vec4 WeightedSample(vec2 luvo, vec2 luvd, vec2
    luvp,
                    vec2 uvo, vec3 probePos,
                    vec3 gTan, vec3 gBit, vec3
                        gPos,
                    float lProbeSize) {

    vec3 lastProbePos = gPos + gTan*(luvp.x*
        lProbeSize/256.) +
                        gBit*(luvp.y*lProbeSize
                            /256.);
    vec3 relVec = probePos - lastProbePos;
    float theta = (lProbeSize*0.5 - 0.5)/
                  (lProbeSize*0.5)*PI*0.5;
    float phi = atan(-dot(relVec, gTan),
                     -dot(relVec, gBit));

    // Sample from atlas with proper phi
        discretization
    float phiI = floor((phi/PI*0.5 + 0.5)*
                       (4. + 8.*(lProbeSize*0.5 -
                           1.))) + 0.5;

    vec2 phiUV;
    float phiLen = lProbeSize - 1.;
    if (phiI < phiLen)
        phiUV = vec2(lProbeSize - 0.5, lProbeSize
            - phiI);
    else if (phiI < phiLen*2.)
        phiUV = vec2(lProbeSize - (phiI - phiLen),
            0.5);
    // ... more phi discretization logic

    float lProbeRayDist = SampleAtlas(
        luvo + floor(phiUV)*uvo + luvp, 0.).w;

    if (lProbeRayDist < -0.5 ||
        length(relVec) < lProbeRayDist*cos(PI*0.5
            - theta) + 0.01) {
```

```
        vec2 luv = luvo + luvd +
                   clamp(luvp, vec2(0.5), uvo -
                       0.5);
        return vec4(SampleAtlas(luv, 0.).xyz +
                    SampleAtlas(luv + vec2(uvo.x,
                        0.), 0.).xyz +
                    SampleAtlas(luv + vec2(0., uvo
                        .y), 0.).xyz +
                    SampleAtlas(luv + uvo, 0.).xyz
                        , 1.);
    }
    return vec4(0.);
}
```

2.2.4 Real-time Display Pass. After baking, the radiance atlas is rendered to the screen with a full 3D camera system:

Listing 12. 3D display pipeline

```
// Pass 2: Final Display (real-time rendering)
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

displayShader.use();
displayShader.setFloat("iTime", currentTime);
displayShader.setVec3("uCamPos", camera.Position);
displayShader.setVec3("uCamFront", camera.Front);

// Bind the atlas texture for display
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, atlasTex[
    atlasWriteIdx]);

drawQuad();

// Swap ping-pong indices
int temp = atlasReadIdx;
atlasReadIdx = atlasWriteIdx;
atlasWriteIdx = temp;
```

2.2.5 Cascade Merging in 3D. The 3D implementation performs cascade merging by sampling from multiple levels of the radiance atlas and blending contributions based on visibility occlusion:

Listing 13. 3D atlas sampling utility

```
// Sample from radiance atlas with level-of-detail
vec4 SampleAtlas(vec2 uv, float lod) {
    vec2 texUV = uv / ATLAS_RES;
    return textureLod(iChannel3, texUV, lod);
}
```

The cascade merging logic determines visibility through ray distance comparisons and selectively blends samples from different cascade levels to produce smooth global illumination.

3 RESULTS

3.1 2D Real-time Implementation Results

The 2D implementation demonstrates real-time interactive global illumination with user-driven light source placement. Key features:

- Interactive brush-based light painting with adjustable color and radius
- Real-time cascade computation and merging
- Smooth angular interpolation for natural light falloff
- Support for both additive (painting) and subtractive (erasing) operations

3.1.1 Visual Results.

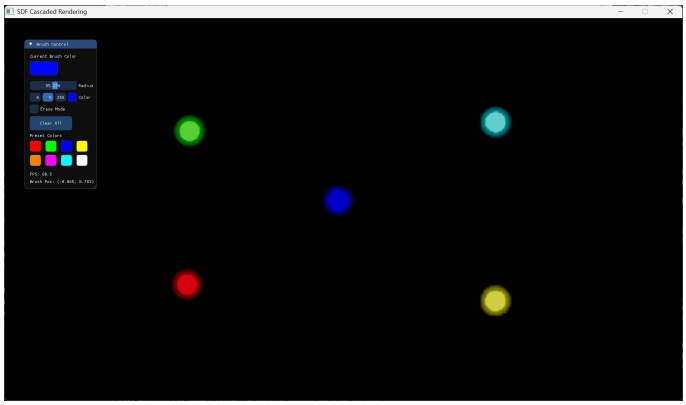

Fig. 1. Initial 2D scene with painted light sources. Multiple colored light sources.

Result 1: Initial Light Painting.

Fig. 2. Light propagation across cascade levels. Shows how light from the painted sources spreads and bounces off objects.

Result 2: Light Propagation Effects.

Result 3: Complex Multi-Light Scenarios.

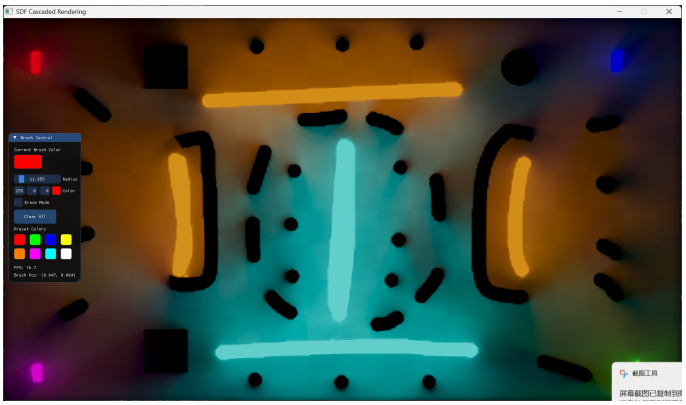

Fig. 3. Multiple interacting light sources with different colors, demonstrating color mixing in global illumination.

3.2 3D Implementation Results

The 3D implementation showcases Radiance Cascades applied to volumetric 3D scenes with offline baking and real-time display:

- Offline radiance atlas baking with global illumination computation
- Hierarchical cascade merging for multi-scale light distribution
- Real-time camera control and scene exploration
- High-resolution radiance atlas (1024 × 6144 pixels)

3.2.1 *Visual Results.*

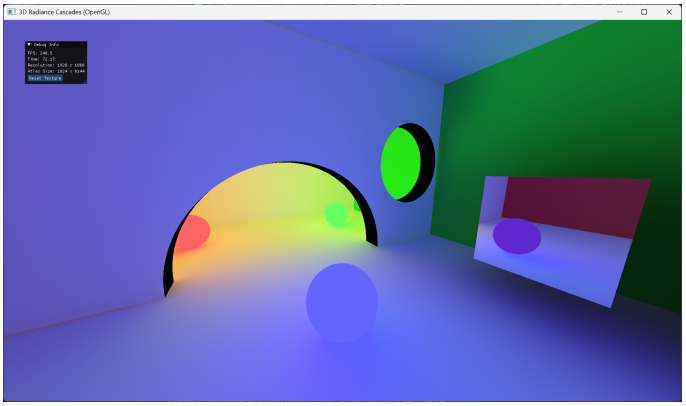

Fig. 4. 3D scene rendered with offline-baked global illumination. The radiance atlas captures light distribution across object surfaces.

Result 4: 3D Scene with Global Illumination.

Result 5: Camera Exploration.

Result 6: Alternate Camera View.

3.3 Performance Metrics

2D Implementation:

- Resolution: 1920 × 1080

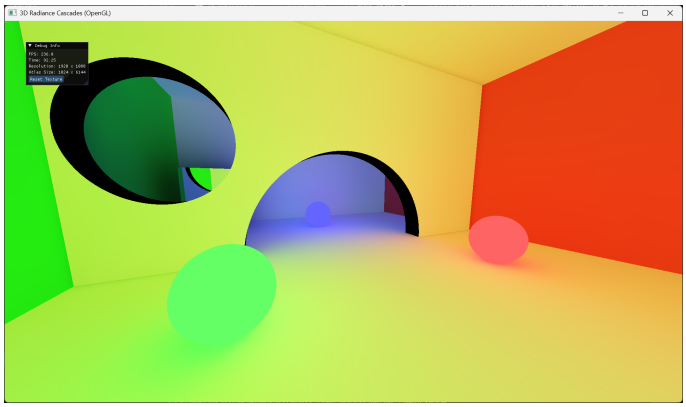

Fig. 5. First camera viewpoint of the 3D scene showing real-time rendering from the pre-computed radiance atlas.

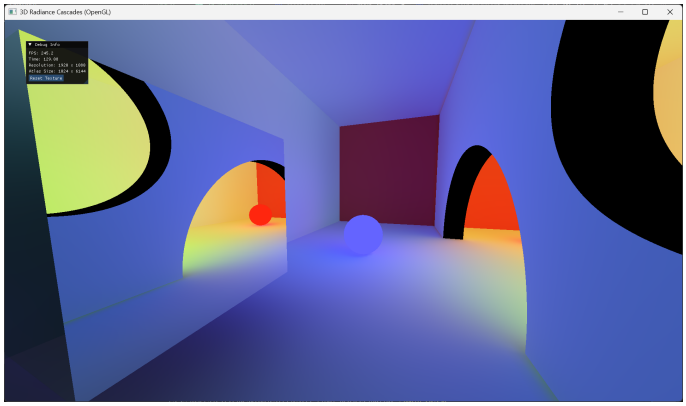

Fig. 6. Second camera viewpoint demonstrating the consistency and quality of the global illumination across different viewing angles.

- Cascade Levels: 4
- Average Frame Rate: 60+ FPS
- Memory Usage: ~ 100 MB

3D Implementation:

- Radiance Atlas Resolution: 1024 × 6144
- Real-time Rendering: 60+ FPS
- Baking Time: Varies with scene complexity (offline process)
- Memory Usage: ~ 150 MB

4 DISCUSSION

4.1 Advantages

(1) **Real-time Performance**: Both implementations achieve 60+ FPS, making them suitable for interactive applications.
(2) **Scalable Light Propagation**: The cascade structure allows efficient light propagation across different spatial scales, from local bounces to global illumination.
(3) **Flexible Scene Editing**: The 2D implementation enables real-time scene editing with immediate global illumination feedback.

(4) **Memory Efficiency**: Using atlas-based storage for 3D scenes reduces memory requirements compared to volumetric approaches.

4.2 Limitations and Future Work

(1) **Static 3D Scenes**: The current 3D implementation requires offline baking, limiting support for fully dynamic scenes. Future work could explore progressive refinement of the radiance atlas.
(2) **Visibility Handling**: The current approach uses distance-based visibility approximation. More sophisticated visibility testing could improve accuracy.
(3) **Temporal Coherence**: The 2D implementation could benefit from temporal reprojection to further increase frame rates and smoothness.
(4) **Probe Placement**: The 3D implementation places probes on object surfaces. Adaptive probe placement could optimize sampling density in complex regions.

5 CONCLUSION

This project successfully demonstrates the implementation of Radiance Cascades for real-time global illumination in both 2D and 3D scenarios. The 2D implementation showcases real-time interactivity with immediate visual feedback, while the 3D implementation demonstrates how the technique scales to complex volumetric scenes through offline baking and real-time playback.

Both implementations achieve the core goal of efficient global illumination computation, with the cascade hierarchy enabling effective light propagation at multiple scales. The work highlights the versatility of the Radiance Cascades approach and its potential for integration into modern real-time graphics pipelines.