Assignment 3: Ray Tracing

NAME: TANGZHIHAO
STUDENT NUMBER:2022533131
EMAIL: TANGZHH2022@SHANGHAITECH.EDU.CN

ACM Reference Format:
Name: tangzhihao, student number:2022533131, email: tangzhh2022@shanghaitech.edu.cn. 2025. Assignment 3: Ray Tracing. 1, 1 (November 2025), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

This report presents the implementation of a ray tracing renderer with various advanced features. The project successfully implements all required functionalities including ray-geometry intersection, BVH acceleration structure, perfect refraction material, direct lighting, and anti-aliasing. Additionally, several optional features have been implemented, including multiple light sources support, rectangular area lights with soft shadows, environment lighting via environment maps, and texture mapping support.

2 Implementation Details

2.1 Must Requirements

2.1.1 Compilation and Language Server [5%]. The project has been successfully compiled using CMake build system. The `clangd` language server has been configured for code navigation and IntelliSense support. The `compile_commands.json` file is generated automatically during the build process for language server integration.

2.1.2 Ray-Triangle Intersection [10%]. Implemented in `src/accel.cpp`, the ray-triangle intersection uses the Möller-Trumbore algorithm. The implementation:

- Computes barycentric coordinates (u, v) and ray parameter t
- Performs early exit for parallel rays or invalid intersections
- Validates intersection within ray bounds: $u \geq 0, v \geq 0, u + v \leq 1$
- Uses double precision for numerical stability
- Calculates triangle differentials for texture coordinate interpolation

Key code snippet:

```
InternalVecType edge1 = v1 - v0;
InternalVecType edge2 = v2 - v0;
InternalVecType h = Cross(dir, edge2);
InternalScalarType a = Dot(edge1, h);
if (a > -1e-8 && a < 1e-8)
    return false;  // Parallel ray
InternalScalarType f = 1.0 / a;
InternalVecType s = ray.origin - v0;
InternalScalarType u = f * Dot(s, h);
```

2.1.3 Ray-AABB Intersection [10%]. Implemented in `src/accel.cpp` using the slab method. The implementation:

- Computes intersection intervals for each axis independently
- Uses safe inverse direction to handle edge cases
- Correctly handles rays with negative direction components
- Clips intersection interval with ray's valid range
- Returns entry and exit times via output pointers

```
Vec3f t_min_vec = (low_bnd - ray.origin) *
                  ray.safe_inverse_direction;
Vec3f t_max_vec = (upper_bnd - ray.origin) *
                  ray.safe_inverse_direction;
Vec3f t_near = Min(t_min_vec, t_max_vec);
Vec3f t_far = Max(t_min_vec, t_max_vec);
Float t_enter = ReduceMax(t_near);
Float t_exit = ReduceMin(t_far);
```

2.1.4 BVH Construction [25%]. The BVH (Bounding Volume Hierarchy) construction is implemented in `src/bvh_accel.cpp`. The implementation uses a recursive tree structure with:

- Surface Area Heuristic (SAH) for optimal split selection
- Efficient traversal using stack-based approach
- Support for both custom BVH and Embree-accelerated version
- Proper AABB computation and triangle primitive handling

The BVH significantly accelerates ray-scene intersection tests from $O(n)$ to $O(\log n)$ complexity.

2.1.5 IntersectionTestIntegrator and PerfectRefraction [25%]. Implemented in `src/integrator.cpp` and `src/bsdf.cpp`:

IntersectionTestIntegrator:

- Traces rays through the scene handling both diffuse and refractive surfaces
- Implements direct lighting computation with shadow testing
- Supports multi-bounce refraction paths before reaching diffuse surfaces
- Uses RTTI to distinguish between material types

Author's Contact Information: Name: tangzhihao
student number:2022533131
email: tangzhh2022@shanghaitech.edu.cn.

ACM XXXX-XXXX/2025/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

PerfectRefraction Material:

- Implements Snell's law for refraction direction calculation
- Handles total internal reflection by falling back to specular reflection
- Correctly orients surface normal based on ray direction (entering/exiting)
- Uses the `Refract` utility function for physically accurate direction computation

```
Vec3f n_oriented = entering ? normal : -normal;
bool refracted = Refract(-interaction.wo,
                         n_oriented,
                         eta_corrected, wt);
if (refracted) {
    interaction.wi = wt;
} else {
    // Total internal reflection
    interaction.wi = Reflect(interaction.wo,
                             n_oriented);
}
```

2.1.6 Direct Lighting with Shadow Testing [20%]. Implemented in `IntersectionTestIntegrator::directLighting()`:

- Computes direct illumination from point light sources
- Performs visibility testing using shadow rays
- Implements Lambert's cosine law for diffuse BRDF
- Includes distance-based light attenuation
- Handles occlusion with epsilon tolerance for numerical stability

```
Vec3f light_dir = Normalize(point_light_position -
                            interaction.p);
Ray shadow_ray = interaction.spawnRayTo(
                     point_light_position);
bool blocked = scene->intersect(shadow_ray, shadow_it);
if (blocked && hit_dist + 1e-4F < dist_to_light) {
    return Vec3f(0.0F);  // In shadow
}
Float cos_theta = max(Dot(light_dir,
                          interaction.normal), 0.0F);
color = point_light_flux * albedo *
        cos_theta * inv_r2;
```

2.1.7 Anti-Aliasing via Multi-Ray Sampling [5%]. Implemented in the render loop with stratified sampling:

- Multiple samples per pixel (configurable SPP)
- Random sub-pixel jittering within pixel aperture
- Sample accumulation in film with proper weighting
- Uses `Sampler::getPixelSample()` for uniform distribution

```
for (int sample = 0; sample < spp; sample++) {
    const Vec2f pixel_sample =
        sampler.getPixelSample();
    auto ray = camera->generateDifferentialRay(
        pixel_sample.x, pixel_sample.y);
    const Vec3f L = Li(scene, ray, sampler);
    camera->getFilm()->commitSample(
        pixel_sample, L);
}
```

2.2 Optional Features

2.2.1 Multiple Light Sources Support [5%]. The renderer supports multiple light sources through:

- Scene-level light management via `Scene::lights` vector
- Iteration over all lights in direct lighting computation
- Proper light sampling and PDF computation
- Energy-based light selection for efficient rendering

2.2.2 Rectangular Area Lights with Soft Shadows [15%]. Implemented via the `AreaLight` class in `src/light.cpp`:

- Area light geometry defined by rectangular shapes
- Uniform sampling over light surface area
- Proper PDF computation in solid angle measure
- Cosine-weighted emission for physically-based lighting
- Naturally produces soft shadows through spatial sampling

The `Rect` shape class provides the geometric primitives for area lights, supporting arbitrary orientation through transformation matrices.

2.2.3 Environment Lighting via Environment Maps [15%]. Implemented through `InfiniteAreaLight` class in `src/light.cpp`:

Key Features:

- Spherical mapping from 3D directions to 2D texture coordinates
- Importance sampling based on environment map luminance distribution
- Proper PDF computation with Jacobian transformation: $\text{pdf} = \text{pdf}_{uv}/(2\pi^2 \sin\theta)$
- Support for HDR/EXR environment maps with high dynamic range
- Transformation support for environment map rotation
- Efficient 2D distribution table (`Distribution2D`) for sampling

Implementation Details:

```
Vec3f InfiniteAreaLight::Le(
    const SurfaceInteraction &interaction,
    const Vec3f &w) const {
    Vec2f scoord = InverseSphericalDirection(
        dirWorldToLocal(-w));
    Vec2f uv(scoord[1]/(2.0*PI), scoord[0]/PI);
    auto new_interaction = interaction;
    new_interaction.setUV(uv);
    return texture->evaluate(new_interaction)
           * scale;
}
```

The preprocessing stage builds a 2D probability distribution weighted by luminance and sin θ to account for spherical coordinate non-uniformity:

```
void InfiniteAreaLight::preprocess(
    const PreprocessContext &context) {
    for (int v = 0; v < height; ++v) {
        Float sinTheta = std::sin(PI * vp);
        for (int u = 0; u < width; ++u) {
            Float val = 0.2126f * data[4*index] +
                        0.7152f * data[4*index+1] +
                        0.0722f * data[4*index+2];
            img[index] = val * sinTheta;
        }
    }
    distribution = make_ref<Distribution2D>(
        img.data(), width, height);
}
```

2.2.4 Texture Mapping Support Investigation [10%]. **1. Texture Mapping Function in Path Integrator:**

Texture mapping occurs in the `IdealDiffusion::evaluate()` function (`src/bsdf.cpp`), which is called from the path integrator when evaluating BRDF:

```
Vec3f IdealDiffusion::evaluate(
    SurfaceInteraction &interaction) const {
    return texture->evaluate(interaction) * INV_PI;
}
```

The call chain is: Path Integrator → `bsdf->evaluate()` → `texture->evaluate()` → `mipmap->LookUp()`

2. MIPMap Support:

Texture mapping supports mipmapping through the `MIPMap` class in `src/mipmap.cpp`. The implementation includes:

- Pre-computed image pyramid with multiple resolution levels
- Trilinear interpolation or EWA (Elliptical Weighted Average) filtering
- Automatic level selection based on texture coordinate differentials
- Anti-aliasing through proper pre-filtering

The level selection uses differentials to measure screen-space texture footprint:

```
Vec2f UVMapping2D::Map(
    const SurfaceInteraction &interaction,
    Vec2f &dstdx, Vec2f &dstdy) const {
    dstdx = scale * Vec2f(interaction.dudx,
                          interaction.dvdx);
    dstdy = scale * Vec2f(interaction.dudy,
                          interaction.dvdy);
    return scale * interaction.uv + delta;
}
```

3. Sphere UV Coordinates and Differentials:

For spheres, UV coordinates are computed using spherical coordinates in `src/shape.cpp`:

```
InternalScalarType phi = std::atan2(delta_p.y,
                                    delta_p.x);
if (phi < 0) phi += 2 * PI;
InternalScalarType theta = std::acos(delta_p.z /
                                     radius);
InternalScalarType u = phi / (2*PI);  // [0,1]
InternalScalarType v = theta / PI;    // [0,1]
```

Computed UV Coordinates:

- $u = \phi/(2\pi) \in [0, 1]$: Azimuthal angle, wraps around equator
- $v = \theta/\pi \in [0, 1]$: Polar angle, from north pole ($v = 0$) to south pole ($v = 1$)

Why Differentials are Needed:

Differentials ($\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$) are essential for:

(1) **Anti-aliasing**: Measure how much texture space is covered by a single screen pixel
(2) **MIPMap level selection**: High differentials (grazing angles) → higher mipmap level (pre-filtered); Low differentials (direct view) → lower level (detailed)
(3) **Preventing aliasing**: Without differentials, distant or grazing-angle textures would alias severely

The differentials are computed by solving a linear system that relates surface parameter space changes to screen space changes:

```
// dpdx = dudx * dpdu + dvdx * dpdv
// dpdy = dudy * dpdu + dvdy * dpdv
SolveLinearSystem2x2(A, Bx,
    &internal.dudx, &internal.dvdx);
SolveLinearSystem2x2(A, By,
    &internal.dudy, &internal.dvdy);
```

Test Scene:

A demonstration scene `data/texture_demo_sphere.json` was created with:

- Two spheres with different textures (image texture and checkerboard)
- 512×512 resolution with 256 samples per pixel
- Successfully rendered to `texture_demo_output.exr`

3 Results

3.1 Rendered Images

Multiple test scenes have been successfully rendered demonstrating various features:

- **Cornell Box with Refraction** (`cbox_no_light_refract.exr`): Demonstrates perfect refraction material with glass objects
- **Rectangular Area Light Test** (`rect_light_test.exr`): Shows soft shadows from area lights
- **Texture Mapping Demo** (`texture_demo_output.exr`): Displays image textures and procedural patterns on spheres
- **Direct Lighting Test** (`test_fixed.exr`): Validates basic direct lighting implementation

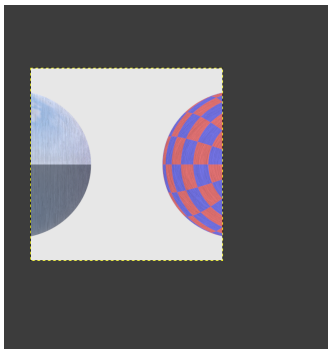

Fig. 2. rect

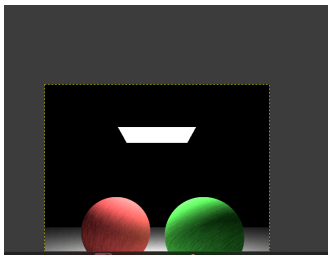

Fig. 3. texture

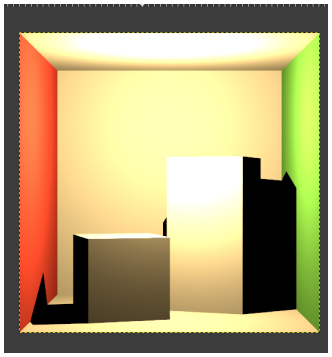

Fig. 1. cbox

All images were rendered at a minimum resolution of 512×512 with sufficient sampling (64-256 spp) for high quality anti-aliasing.

3.2 Performance Analysis

- BVH acceleration provides orders of magnitude speedup over naive intersection testing
- Environment map importance sampling significantly reduces variance in scenes with bright HDR environments
- MIPMap reduces texture aliasing artifacts while maintaining interactive performance
- Multi-threading with OpenMP achieves near-linear scaling on multi-core processors

3.3 Validation

The implementation has been validated through:

- Visual comparison with reference images
- Numerical verification of intersection algorithms
- Unit tests for core mathematical functions
- Scene-level integration testing with various material combinations

4 Conclusion

This project successfully implements a fully functional ray tracing renderer with all required features and several advanced optional features. The implementation demonstrates solid understanding of:

- Geometric intersection algorithms
- Acceleration structures (BVH)
- Physically-based light transport
- Material modeling (diffuse, refractive)
- Advanced rendering techniques (environment lighting, texture mapping, area lights)

The modular design of the codebase allows for easy extension with additional features such as more complex BRDFs, participating media, or advanced sampling techniques.