Assignment 3:
Basic Ray Tracing

NAME: CHEN JUNLIN
STUDENT NUMBER: 2022533009
EMAIL: CHENJL2022@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment I implemented the core components of a simple ray tracer focusing on robust intersection routines, an acceleration structure, direct lighting, material sampling (including refraction), and an optional GPU-accelerated LBVH builder. The renderer supports a correct ray–triangle intersection, an AABB slab test for bounding-box pruning, several BVH construction strategies (median split and a Karras-style LBVH), a direct illumination integrator with shadow rays and BSDF sampling, and an implementation path that uses CUDA/Thrust to accelerate the LBVH construction when enabled.

2 IMPLEMENTATION DETAILS

2.1 Ray-Triangle Intersection and Ray-AABB Intersection

Ray–triangle intersection is implemented using the Möller–Trumbore algorithm. The routine casts the ray in double precision for numerical robustness, computes the triangle edges and the determinant, and solves for the barycentric coordinates (u, v) and the ray parameter t. The code exits early if the determinant is near zero (parallel ray) or if barycentric coordinates fall outside the triangle ($u < 0$, $v < 0$, $u+v > 1$). When an intersection is confirmed, the ray's t_max is updated to the hit distance so subsequent tests respect the nearest-hit rule. Surface differentials are computed for shading when required.

```
// accel.cpp // TriangleIntersect
const InternalVecType orig = Cast<InternalScalar
    Type>(ray.origin);
const InternalVecType edge1 = v1 - v0;
const InternalVecType edge2 = v2 - v0;
const InternalVecType pvec = Cross(dir, edge2);
const InternalScalarType det = Dot(edge1, pvec);
const InternalScalarType eps = InternalScalarType
    (1e-12);
if (std::abs(det) < eps) return false;
const InternalScalarType invDet = InternalScalar
    Type(1) / det;
const InternalVecType tvec = orig - v0;
InternalScalarType u = invDet * Dot(tvec, pvec);
if (u < InternalScalarType(0) || u > Internal
    ScalarType(1)) return false;
const InternalVecType qvec = Cross(tvec, edge1);
InternalScalarType v = invDet * Dot(dir, qvec);
if (v < InternalScalarType(0) || (u + v) >
    InternalScalarType(1)) return false;
InternalScalarType t = invDet * Dot(edge2, qvec);
if (!ray.withinTimeRange(static_cast<Float>(t)))
    return false;
```

Axis-aligned bounding box intersection uses the standard slab method. A precomputed "safe" inverse ray direction is used to avoid division by zero; per-axis intersection intervals [t_min_axis, t_max_axis] are computed and the intersection interval is the intersection of these three ranges. The implementation returns the entering and exiting t values and also checks the ray's time range so that only intersections within the ray's domain are accepted. These two routines together produce a numerically robust base for coarse pruning and primitive testing.

```
[==========] Running 7 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from AABB
[ RUN      ] AABB.AxisAligned_EnterExit_PositiveAndNegativeDirs
[       OK ] AABB.AxisAligned_EnterExit_PositiveAndNegativeDirs (0 ms)
[----------] 1 test from AABB (1 ms total)

[----------] 6 tests from TriangleIntersect
[ RUN      ] TriangleIntersect.Basic
[       OK ] TriangleIntersect.Basic (0 ms)
[ RUN      ] TriangleIntersect.MissCases
[       OK ] TriangleIntersect.MissCases (0 ms)
[ RUN      ] TriangleIntersect.TimeWindow_RejectionAndClamping
[       OK ] TriangleIntersect.TimeWindow_RejectionAndClamping (0 ms)
[ RUN      ] TriangleIntersect.TriangleInXZPlane_Hit
[       OK ] TriangleIntersect.TriangleInXZPlane_Hit (0 ms)
[ RUN      ] TriangleIntersect.DegenerateTriangle_ReturnsFalse
[ RUN      ] TriangleIntersect.DegenerateTriangle_ReturnsFalse
[       OK ] TriangleIntersect.DegenerateTriangle_ReturnsFalse (0 ms)
[ RUN      ] TriangleIntersect.MultiTriangleMesh_HitCorrectTriangle
[       OK ] TriangleIntersect.MultiTriangleMesh_HitCorrectTriangle (0 ms)
[----------] 6 tests from TriangleIntersect (4 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 2 test suites ran. (7 ms total)
[  PASSED  ] 7 tests.
```

Fig. 1. All passed on intersection tests.

```
// accel.cpp // AABB::intersect
const Vec3f inv_dir = ray.safe_inverse_direction;
const Vec3f orig = ray.origin;
const Vec3f t0 = (low_bnd - orig) * inv_dir;
const Vec3f t1 = (upper_bnd - orig) * inv_dir;
const Vec3f t_min_vec = Min(t0, t1);
const Vec3f t_max_vec = Max(t0, t1);
const Float t_enter = static_cast<Float>(Reduce
    Max(t_min_vec));
const Float t_exit = static_cast<Float>(Reduce
    Min(t_max_vec));
```

student number: 2022533009
email: chenjl2022@shanghaitech.edu.cn

```
if (t_enter > t_exit) return false;
if (t_exit < ray.t_min || t_enter > ray.t_max)
    return false;
if (t_in) *t_in = t_enter;
if (t_out) *t_out = t_exit;
return true;
```

2.2 BVH Construction

The BVH supports multiple construction heuristics. The median-split approach uses std::nth_element to partition primitives by centroid along the chosen split axis (either the largest axis or depth-based), producing a balanced tree quickly without expensive cost evaluations.

```
// bvh_tree.h // BVHTree<_>::build
if (depth >= CUTOFF_DEPTH || span_right -
    span_left <= 1) {
const auto &node = nodes[span_left];
InternalNode result(span_left, span_right);
result.is_leaf = true;
result.aabb = prebuilt_aabb;
internal_nodes.push_back(result);
return internal_nodes.size() - 1;
}

auto mid_it = nodes.begin() + split;
std::nth_element(nodes.begin() + span_left,
    mid_it, nodes.begin() + span_right, [dim]
    (const NodeType &a, const NodeType &b) {
    return a.getAABB().getCenter()[dim] <
    b.getAABB().getCenter()[dim];});
```

```
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from BVH
[ RUN      ] BVH.BasicConstruction
[       OK ] BVH.BasicConstruction (0 ms)
[ RUN      ] BVH.SingleObject
[       OK ] BVH.SingleObject (0 ms)
[ RUN      ] BVH.EmptyTree
[       OK ] BVH.EmptyTree (0 ms)
[----------] 3 tests from BVH (1 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (3 ms total)
[  PASSED  ] 3 tests.
```

Fig. 2. All passed on BVH tests.

2.3 Implement a Direct Illumination Integrator

The direct illumination integrator implemented here supports flexible pixel sampling (stratified or uniform) and uses the camera to generate differential rays for each sample. For each visible surface interaction, the integrator evaluates direct lighting by sampling the light (implemented here for point lights) and casting a shadow ray toward the light. If the shadow ray is unoccluded, the integrator evaluates the BSDF at the hit point (for this assignment explicit Lambertian handling is demonstrated), applies the cosine term with respect to the surface normal, and attenuates by the inverse-square distance for point lights. The integrator uses BSDF sampling for more general materials to estimate contributions where appropriate. Shadow testing is implemented by casting a secondary ray and checking for any intersection before the light distance, providing correct hard shadows.

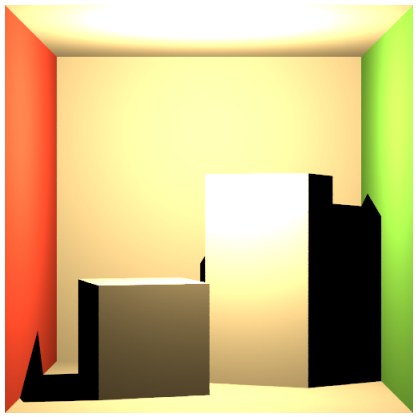

Fig. 3. Direct illumination without refraction.

```
// integrator.cpp
// IntersectionTestIntegrator::render
Vec2f pixel_sample;
if (use_stratified) {
int sx = sample % sqrt_spp;
int sy = sample / sqrt_spp;
Vec2f jitter = sampler.get2D();
pixel_sample = Cast<Float>(Vec2i(dx, dy)) +
     Vec2f((sx + jitter.x) / (Float)sqrt_spp,
     (sy + jitter.y) / (Float)sqrt_spp);
} else {pixel_sample = sampler.getPixelSample();}
DifferentialRay ray = camera->generateDifferential
    Ray(pixel_sample.x, pixel_sample.y);
assert(pixel_sample.x >= dx && pixel_sample.x <=
    dx + 1);
assert(pixel_sample.y >= dy && pixel_sample.y <=
    dy + 1);
const Vec3f &L = Li(scene, ray, sampler);
camera->getFilm()->commitSample(pixel_sample, L);

//IntersectionTestIntegrator::directLighting
Ray test_ray = interaction.spawnRayTo
    (point_light_position);
SurfaceInteraction shadow_it;
if (scene->intersect(test_ray, shadow_it)) {
```

```
return color;
}
const BSDF *bsdf = interaction.bsdf;
bool is_ideal_diffuse = dynamic_cast<const
    IdealDiffusion *>(bsdf) != nullptr;
if (bsdf != nullptr && is_ideal_diffuse) {
Float cos_theta = std::max(Dot(light_dir,
    interaction.normal), 0.0f);
interaction.wi = light_dir;
Vec3f albedo = bsdf->evaluate(interaction);
Vec3f attenuation = point_light_flux /
    (dist_to_light * dist_to_light);
color = albedo * cos_theta * attenuation;
}
```

2.4 Integrate with Refractive Materials

Refraction is handled inside the BSDF sampling routines. For perfect refraction, the code computes a refracted direction using Snell's law; when total internal reflection occurs, the code falls back to reflected direction. The integrator uses the BSDF sample to produce a new ray (spawned from the surface interaction) and continues path traversal by updating the current ray with the sampled direction and t-range. This straightforward approach supports dielectric transmission and ensures energy-conserving direction choices for refractive surfaces.

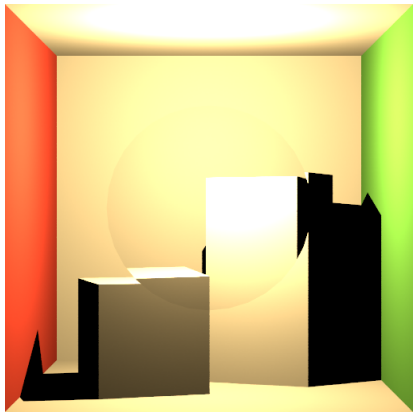

Fig. 4. Direct illumination with refraction.

```
// integrator.cpp
// IntersectionTestIntegrator::Li
Float pdf = 0.0F;
if (interaction.bsdf) {interaction.bsdf->
    sample(interaction, sampler, &pdf);}
Ray new_ray = interaction.spawnRay
    (interaction.wi);
ray = new_ray;
continue;
```

```
// bsdf.cpp // PerfectRefraction::sample
Vec3f wt = Vec3f(0.0f);
bool refracted = Refract(interaction.wo,
    normal, eta_corrected, wt);
if (refracted) {interaction.wi = wt;}
else {interaction.wi = Reflect(interaction.wo,
normal);}
```

2.5 GPU-parallel BVH Construction

An optional CUDA-based LBVH builder is provided and can be enabled with -DUSE_CUDA=ON. The GPU pipeline uploads centroids to the device, computes 30-bit Morton keys (10 bits per axis) in a small kernel, then performs a device sort using Thrust's high-performance sort_by_key (tracking an index permutation). After sorting, a Karras-style topology kernel runs in parallel: each thread computes common-prefix lengths (via device __clz) and uses exponential/binary search to locate split points, producing left/right child indices for every internal node. The host wrapper copies back the sorted permutation and child arrays and uses them to assemble the final LBVH topology.

```
// CMakeLists.txt
option(USE_CUDA "Enable␣CUDA␣Karras␣BVH
␣␣␣␣builder" OFF)
if (USE_CUDA)
    list(APPEND source "${PROJECT_SOURCE
␣␣␣␣␣DIR}/src/karras_bvh_gpu.cu")
endif()
```

On the host we first apply the GPU's permutation to the primitive list so leaf positions match the kernel's assumptions, then construct the internal nodes array and compute parent pointers. A post-order traversal computes each internal node's span_left/span_right from its children (leaves already have spans set), and a subsequent bottom-up pass merges child AABBs to populate internal AABBs. The implementation preserves the LBVH indexing convention (internal nodes [0..n-2], leaves [(n-1)..(2n-2)]) and is defensive: CUDA errors or kernel failures trigger a CPU fallback to guarantee correctness. This design ensures the GPU-produced topology and host-side primitive ordering remain consistent.

```
// bvh_tree.h
#if defined(USE_CUDA)
{
const int N = nodes.size();
if (N == 0) goto cpu_fallback;
std::vector<Vec3f> centroids(N);
AABB global;
for (int i = 0; i < N; i++) {
auto c = nodes[i].getAABB().getCenter();
centroids[i] = c;
global.unionWith(nodes[i].getAABB());
}
std::vector<int> left_idx, right_idx,
    parent_idx, sorted_index;
```

```
bool ok = karras_bvh_gpu_build(centroids,
    global.pMin, global.pMax, sorted_index,
    left_idx, right_idx, parent_idx);
if (!ok) goto cpu_fallback;
const int total_nodes = left_idx.size();
internal_nodes.resize(total_nodes);
for (int i = 0; i < total_nodes; i++) {
  auto &nd = internal_nodes[i];
  nd.left_index = left_idx[i];
  nd.right_index = right_idx[i];
  nd.parent = parent_idx[i];
  if (i >= N - 1) {
    nd.is_leaf = true;
    int leaf_id = i - (N - 1);
    nd.span_left = sorted_index[leaf_id];
    nd.span_right = nd.span_left + 1;
  } else {
    nd.is_leaf = false;
  }
}
for (int i = total_nodes - 1; i >= 0; i--) {
  auto &nd = internal_nodes[i];
  if (nd.is_leaf) {
    nd.aabb = nodes[nd.span_left].getAABB();
  } else {
    AABB box;
    if (nd.left_index >= 0)
      box.unionWith(internal_nodes[nd.left_
        index].aabb);
    if (nd.right_index >= 0)
      box.unionWith(internal_nodes[nd.right_
        index].aabb);
    nd.aabb = box;
  }
}
root_index = total_nodes - 1;
is_built = true;
return;
}
#endif
```

3 RESULTS

This assignment implements the core ray-tracing pipeline components with a focus on correctness and maintainability. The renderer produces correct intersection results and plausible lighting on the test scenes included with the assignment.