Minip

Noam Hadad **322766064**

Guedalia Sebbah **337966659**

Mini-Project 1 (Soft Shadow)

1. Overview

In this mini-project I implemented soft shadows via area sampling and anti-aliasing via supersampling (oversampling) in our existing ray-tracer. Instead of emitting a single ray per pixel or per shadow test, both camera rays and shadow-rays are replaced by a small “beam” of rays whose colors (or visibility factors) are averaged, producing smoother edges and more realistic penumbras.

2. Architecture & Key Classes

SamplingConfig & SuperSamplingBlackboard

Lives in renderer.sampling.

Holds three parameters:

- sampleCount (number of rays in the beam),

- SamplingPattern (GRID, JITTERED, or RANDOM),

- TargetShape (RECTANGLE or CIRCLE).

SuperSamplingBlackboard.getSampleOffsets(...) returns a list of 2D offsets covering the target area evenly, which the camera uses to jitter its primary ray across each pixel.

Camera

Extended to support a SamplingConfig.

In castRay(...), instead of firing one ray per pixel, it:

1. Calls blackboard.getSampleOffsets(...) to get N offsets,

2. For each offset, constructs a slightly shifted ray (pIJ + offset),

3. Traces each ray and sums the colors,

4. Averages by dividing by the number of samples.

SimpleRayTracer

Inherits from RayTracerBase.

Overrides traceRay(...) to do Phong shading plus recursive reflection/refraction.

Modifies the shadow-ray stage (transparency(...)):

- If light is a PointLight with numSamples > 1, it

1. Offsets the start point slightly to avoid acne,
2. For each of N samples, jitters a shadow ray toward a random point on the light's disk,
3. Checks for occluders up to that distance,
4. Counts how many rays are unblocked,
5. Returns visibility = # unblocked / N.

- Otherwise it falls back to a single hard shadow ray.

PointLight

Now supports a disk radius and a sample count.

getSamplePoint(p) returns a random point on the circular area light, used by the tracer to generate soft shadows.

StreetSceneTest

A JUnit test that builds a realistic nocturnal street scene (trees, buildings, moon, lamps) and:

- Turns the anti-aliasing feature ON by calling camera.setSamplingConfig(new SamplingConfig(81, RECTANGLE, GRID)),

- Turns soft shadows ON by configuring each PointLight with .setRadius(10).setNumSamples(81),

- Renders the image twice (with/without each feature) and writes out street2.png.

3. How I Did It

1. Anti-Aliasing (Supersampling)
- Chose a rectangular target area the size of one pixel.
- Generated an N×N grid of offsets via SamplingPattern.GRID in SuperSamplingBlackboard.
- In Camera.castRay(...), for each of the 81 offsets I cast a ray and averaged the results.
- Enabled it in the test with camera.setSamplingConfig(...).

2. Soft Shadows (Area Sampling)
- Extended PointLight to have a user-configurable radius and numSamples.
- In SimpleRayTracer.transparency(...), when numSamples > 1:
  - Generated a beam of shadow rays toward jittered sample points on the light's disc,
  - Measured how many rays reached the light unblocked,
  - Set the light's contribution to that fraction (soft penumbra).

3. Integration & Testing
- Kept all parameters (sample count, radius, focal distance, etc.) in clearly named setters so the unit test can toggle features on/off without changing core code.
- Ensured no hard-coded values—everything comes from SamplingConfig or PointLight.setRadius(...) etc.
- Wrote the StreetSceneTest so that each rendered image is produced twice (with and without supersampling and soft shadows) and timing is reported.

4. Results & Reflection

- Anti-aliasing eliminated jagged edges around objects and geometry.
- Soft shadows produced gradual penumbras under trees and around the bus shelter, replacing the previously sharp, unrealistic silhouette shadows.
- Both features increased render time by roughly the number of samples—but the visual quality improvement is substantial, and multithreading keeps overall times acceptable.

Mini-Project 2: (BVH)

1. Overview

Implement a BVH acceleration structure to speed up ray-scene intersection tests in the ray tracer. BVH groups geometry into a binary tree of axis-aligned bounding boxes, enabling efficient pruning of empty space.

2. Design & Implementation

• BVHNode Class:
 - Stores either a list of leaf primitives or two child BVHNode instances.
 - Each node caches an axis-aligned BoundingBox covering its contents.

• Construction:
 - Top-down recursive split by object-median along the longest axis.
 - Leaf threshold controls maximum primitives per leaf (e.g., 4).

• Traversal (calculateIntersectionsHelper):
 1. Test ray vs. node's AABB once.
 2. If leaf, intersect each primitive directly.
 3. Else, recurse into left and/or right child only if their boxes intersect the ray.
 4. Collect and merge intersection lists only when non-empty.

3. Results

Render time benchmarks on a complex street scene (1,000+ primitives, multiple lights):

Configuration	Render Time (s)
No BVH, Single-Threaded	490
BVH, Single-Threaded	1201
No BVH, Multi-Threaded	198
BVH, Multi-Threaded	123

Speed-up analysis:
• BVH + threads vs. No BVH + threads: 198 → 123 s (~1.6× faster).
• BVH single-thread vs. No BVH single-thread: 490 → 1201 s (~2.5× slower).
• BVH + threads vs. BVH single-thread: 1201 → 123 s (~9.8× faster).

4. Analysis

The BVH traversal overhead (AABB tests, recursion, and list merging) outweighs its culling benefits in a pure single-threaded context. However, when combined with multi-threading, the parallel work distribution amortizes BVH overhead, yielding significant net gains.

5. Improvements

- Integrate optimized helper to avoid duplicate AABB checks and minimize list allocations.
- Use Surface Area Heuristic (SAH) for better splitting and balanced trees.
- Flatten BVH into arrays for cache-friendly traversal loops.
- Implement iterative, stack-based traversal to reduce recursion overhead.
- Tune leaf size threshold and splitting heuristics for specific scenes.

6. Conclusion

BVH provides clear performance benefits in multi-threaded rendering but requires careful implementation to be effective in single-threaded mode. Further optimizations and heuristic improvements can ensure consistent speed-ups across all execution modes.