Renderer Documentation

Neel Sankaran
Department of Electrical and Computer Engineering
University of California, Santa Barbara

January 2026

Section	Description
Scene Representation	How the scene is constructed from raw triangular mesh faces and how edge adjacency is assigned.
Ray–Triangle Intersection	How ray intersections are computed against triangles and how hit information is stored.
Path Sampling	How camera-to-light paths are sampled through successive surface intersections.
Path Mutations	How existing paths are locally and globally modified to explore nearby configurations.
Radiance Transport	How light contribution is evaluated and accumulated along a path.
Rendering Loop	How each pixel is rendered by sampling a path and applying repeated mutations.
Code Documentation	Reference for all classes, methods, and parameters used in the implementation.

Table 1: Content Summary per Section.

1 Scene Representation

The scene is represented as a collection of triangles containing local topological information. Each triangle stores:

- Vertex positions $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^3$
- Per-vertex normals $\mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2$
- Indices of adjacent triangles across each edge

All meshes are triangulated and flattened into a single global triangle list. Vertex positions are stored in a unified array with mesh-local indices remapped to global indices, allowing geometry to be accessed during intersection tests and path traversal.

1.1 Triangle adjacency

Each triangle stores adjacency information across its three *local* edges. We use a fixed local edge indexing convention:

$$e = 0 : \ (\mathbf{i}_0, \mathbf{i}_1), \qquad e = 1 : \ (\mathbf{i}_1, \mathbf{i}_2), \qquad e = 2 : \ (\mathbf{i}_2, \mathbf{i}_0),$$

where $(\mathbf{i}_0, \mathbf{i}_1, \mathbf{i}_2)$ are the triangle's vertex indices.
For each local edge e, two values are stored:

$\texttt{adj}[e]$ = index of the neighboring triangle, $\texttt{adjEdge}[e]$ = neighbor's local edge index.

If $\texttt{adj}[e] = t' \geq 0$, then triangle t' shares the same vertex pair as edge e. The field $\texttt{adjEdge}[e]$ records which of the neighbor's three local edges corresponds to this same shared edge. Because edge indices are local to each triangle, this information cannot be inferred from $\texttt{adj}[e]$ alone and must be stored explicitly. Edges without a neighbor are marked as boundary edges by $\texttt{adj}[e] = -1$.

1.2 Bounding box

An axis-aligned bounding box is computed over all vertices:

$$\mathbf{b}_{\min} = \min_i \mathbf{p}_i, \qquad \mathbf{b}_{\max} = \max_i \mathbf{p}_i,$$

with minima and maxima taken component-wise.

2 Ray–Triangle Intersection

All geometric queries are based on the Möller–Trumbore ray–triangle intersection algorithm, which solves directly for barycentric coordinates without explicitly computing the triangle plane.

2.1 Ray–triangle formulation

Given a ray

$$\mathbf{r}(t) = \mathbf{r}_0 + t\,\mathbf{r}_d, \qquad t \geq 0,$$

and a triangle with vertices $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$, an intersection occurs if there exist (t, u, v) such that

$$\mathbf{r}_0 + t\,\mathbf{r}_d = \mathbf{v}_0 + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0), \tag{1}$$

with

$$u \geq 0, \quad v \geq 0, \quad u + v \leq 1, \quad t > 0.$$

Define

$$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0, \qquad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0, \qquad \mathbf{tvec} = \mathbf{r}_0 - \mathbf{v}_0.$$

Equation (1) can be written as

$$\mathbf{tvec} = u\,\mathbf{e}_1 + v\,\mathbf{e}_2 - t\,\mathbf{r}_d. \tag{2}$$

2.2 Möller–Trumbore solution

The system (2) is solved using Cramer's rule, implemented via vector cross and dot products:

$$\mathbf{pvec} = \mathbf{r}_d \times \mathbf{e}_2, \tag{3}$$

$$\det = \mathbf{e}_1 \cdot \mathbf{pvec}. \tag{4}$$

If $|\det|$ is sufficiently small, the ray is parallel to the triangle and no intersection is reported.

The solution is given by

$$u = \frac{\mathbf{tvec} \cdot \mathbf{pvec}}{\det}, \tag{5}$$

$$\mathbf{qvec} = \mathbf{tvec} \times \mathbf{e}_1, \tag{6}$$

$$v = \frac{\mathbf{r}_d \cdot \mathbf{qvec}}{\det}, \tag{7}$$

$$t = \frac{\mathbf{e}_2 \cdot \mathbf{qvec}}{\det}. \tag{8}$$

The intersection is valid if the barycentric constraints hold and t exceeds a small offset used to avoid self-intersections.

2.3 Hit data and normals

For the closest valid intersection, the hit point and barycentric coordinates are

$$\mathbf{p} = \mathbf{r}_0 + t\,\mathbf{r}_d, \qquad (w, u, v) = (1 - u - v, u, v).$$

If per-vertex normals are available, the shading normal is interpolated and normalized. Otherwise, the geometric face normal

$$\hat{\mathbf{n}}_g = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|}$$

is used.

2.4 Visibility queries

Visibility tests are implemented by casting a bounded ray between two points. A point $\mathbf{x}$ is visible from $\mathbf{o}$ if no intersection occurs at distance

$$t < \|\mathbf{x} - \mathbf{o}\| - \varepsilon,$$

where ε prevents numerical self-occlusion.

3 Path Sampling

A path is stored as an ordered sequence of vertices

$$\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1},$$

with $\mathbf{x}_0 = \mathbf{C}$ (camera) and $\mathbf{x}_{k+1} = \mathbf{L}$ (light). Each internal vertex lies on a triangle and stores its position, triangle index, and barycentric coordinates.

Initial ray. Path sampling begins by tracing a single ray from the camera along a prescribed direction. If this ray does not intersect the scene, the path is rejected. This is the only rejection condition during sampling.

Bounce extension. After the first hit, the path is extended iteratively. At each bounce, directions are sampled from a hemisphere oriented about the surface normal. If an intersection is found within a fixed number of attempts, the next vertex is appended. If no valid intersection is found, the path terminates naturally.

Light visibility does not affect path construction and is evaluated only after the full geometric path is sampled.

Light visibility. Once sampling is complete, the light position is appended and a shadow ray is cast from each internal vertex toward the light. The resulting visibility flags are stored and updated locally during mutations.

4 Path Mutations

Path mutations modify a single internal vertex while keeping the path geometrically valid. A mutation is accepted only if visibility between adjacent vertices is preserved. Barycentric data and cached visibility are updated locally after acceptance.

Mutations are classified as *local* or *global* depending on their spatial extent.

4.1 Local Mutations

Local mutations produce small, surface-coherent changes and are used to refine existing paths.

4.1.1 Mesh-Walk Mutation

In a mesh-walk mutation, a vertex is displaced tangentially along the surface. A direction is sampled in the tangent plane of the current triangle. As the displacement crosses triangle edges, the mutation transitions to adjacent triangles using stored adjacency.

To maintain consistency across edges, the direction is rotated about the shared edge axis using Rodrigues' formula:

$$\mathbf{d}' = \mathbf{d}\cos\theta + (\mathbf{a} \times \mathbf{d})\sin\theta + \mathbf{a}(\mathbf{a} \cdot \mathbf{d})(1 - \cos\theta),$$

where $\mathbf{a}$ is the edge direction and θ is the dihedral angle between face normals. The direction is then projected onto the new triangle's tangent plane. The final position is stored using updated barycentric coordinates.

4.1.2 Projection Mutation

The projection mutation samples a nearby point in the tangent neighborhood of the vertex and projects it onto the mesh using a ray intersection. The ray is cast from a point offset along the surface normal toward the sampled target. The first intersection defines the new vertex location.

The mutation is accepted only if the new vertex remains visible to its immediate neighbors along the path.

4.2 Global Mutations

Global mutations allow larger changes in path geometry and improve exploration.

4.2.1 Retrace Mutation

In a retrace mutation, a new bounce direction is sampled from the hemisphere at the current surface point. A ray is cast along this direction, and the first intersection replaces the original vertex.

Since the new direction is independent of the previous path geometry, retracing can produce large changes. As with local mutations, the proposal is accepted only if visibility to neighboring vertices is preserved.

5 Radiance Transport Along a Path

Given a sampled path

$$\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-2}, \mathbf{x}_{N-1},$$

with $\mathbf{x}_0 = \mathbf{C}$ and $\mathbf{x}_{N-1} = \mathbf{L}$, radiance is accumulated by evaluating direct illumination at each internal surface vertex and propagating throughput along the path.

We assume a diffuse (Lambertian) surface model with constant albedo $\boldsymbol{\rho}$. The corresponding bidirectional reflectance distribution function (BRDF) is

$$f(\omega_i, \omega_o) = \frac{\boldsymbol{\rho}}{\pi}.$$

5.1 Throughput propagation

The path throughput $\boldsymbol{\beta}$ accounts for all scattering events prior to the current vertex. It is initialized as

$$\boldsymbol{\beta}_1 = \mathbf{1},$$

and updated at each bounce according to

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i \, f(\omega_i, \omega_o) \, \frac{\cos\theta_o}{p(\omega_o)},$$

where ω_o is the sampled outgoing direction, θ_o is its angle with the surface normal, and $p(\omega_o)$ is the sampling probability density. Since outgoing directions are sampled uniformly over the hemisphere,

$$p(\omega_o) = \frac{1}{2\pi}.$$

5.2 Direct lighting

At each internal vertex $\mathbf{x}_i$, direct lighting from the point light $\mathbf{L}$ is evaluated. Let

$$\mathbf{w}_i = \frac{\mathbf{L} - \mathbf{x}_i}{\|\mathbf{L} - \mathbf{x}_i\|}, \qquad r_i^2 = \|\mathbf{L} - \mathbf{x}_i\|^2,$$

and let $\mathbf{n}_i$ be the shading normal at $\mathbf{x}_i$. If $\mathbf{x}_i$ is visible to the light, the contribution is

$$\mathbf{L}_i = \boldsymbol{\beta}_i \, f(\omega_i, \mathbf{w}_i) \, \frac{\mathbf{I}_L}{r_i^2} \, \max(0, \mathbf{n}_i \cdot \mathbf{w}_i),$$

where $\mathbf{I}_L$ is the light intensity. Visibility is determined either from cached shadow-ray results or by an explicit visibility query.

The total radiance for the path is the sum of all such contributions:

$$\mathbf{L} = \sum_{i=1}^{N-2} \mathbf{L}_i.$$

6 Rendering Loop and Path Mutation Strategy

Rendering proceeds independently per pixel. For each pixel location (x, y), a primary ray direction is generated using the camera model and used to seed an initial path.

6.1 Per-pixel path evaluation

For each pixel:

1. A primary ray is generated through the pixel center.
2. A single path is sampled using the procedure described in Section 3.

3. If the initial ray fails to intersect the scene, the pixel contribution is zero.

4. Otherwise, the radiance of the sampled path is evaluated and accumulated.

This initial path serves as a guaranteed sample for the pixel, ensuring that each pixel receives at least one valid contribution whenever possible.

6.2 Mutation-based refinement

After the initial path is obtained, the renderer performs K mutation steps. Each step:

1. Selects a random internal vertex of the current path.

2. Proposes a new path using one of the mutation strategies (retrace, mesh-walk, or projection).

3. Accepts the proposal if it is geometrically valid.

When a mutation is accepted, the current path is replaced by the proposal, its radiance is evaluated, and the contribution is accumulated. Rejected mutations are ignored. No acceptance probabilities or Metropolis weighting are used *yet*.

6.3 Pixel estimate

Let $\mathbf{L}^{(j)}$ denote the radiance evaluated for the j-th accepted path (including the initial sample). The final pixel color is computed as the arithmetic mean:

$$\mathbf{C}_{x,y} = \frac{1}{M} \sum_{j=1}^{M} \mathbf{L}^{(j)},$$

where M is the number of accepted paths for that pixel.

7 Code Documentation

7.1 `Renderer` (`Renderer.h`)

7.1.1 `Renderer::Camera`

- `center` : Camera (pinhole) origin $\mathbf{C}$ in world space.
- `forward` : Forward viewing direction (normalized internally).
- `focal_length` : Distance from pinhole to image plane (in world units).
- `pixel_size` : Size of a pixel on the image plane (world units per pixel).
- `width`, `height` : Output image resolution in pixels.
- `world_up` : Preferred "up" direction used to build the camera basis.

7.1.2 `Renderer::RenderParams`

- `maxBounces` : Maximum number of surface interactions stored in a sampled path.
- `retriesPerBounce` : Attempts per bounce to find a valid next intersection direction.
- `Kmutations` : Number of mutation proposals evaluated per pixel (in addition to the seed path).
- `visibilityEps` : Offset/epsilon used for visibility (shadow) rays to avoid self-intersection.
- `mutateRadiusFrac` : Local mutation radius as a fraction of scene diagonal length.
- `albedo` : Lambertian reflectance ρ used in the BRDF $f = \rho/\pi$.
- `lightIntensity` : Point light intensity vector used in $L_i \propto I/\|\mathbf{x} - \mathbf{L}\|^2$.

7.1.3 `Renderer` public methods

- `Renderer(const Scene&, const Camera&, const glm::vec3& lightPos, const RenderParams& params)`
  Constructs a renderer for a fixed scene, camera, and point light.
  - `scene`: Reference to the loaded triangle scene.
  - `cam`: Camera intrinsics/extrinsics (pinhole model).
  - `lightPos`: Point light position $\mathbf{L}$ in world space.
  - `params`: Rendering and sampling parameters.
- `glm::vec3 compute_radiance_for_path(const PathMutator::Path& path) const`
  Evaluates the path contribution under a Lambertian model using cached geometry and (optionally) cached light visibility.
  - `path`: Sequence of vertices plus per-vertex triangle/barycentric data.
- `std::vector<glm::vec3> render_scene(uint32_t seed=1337u, const int mutator_type=0) const`
  Renders an image by sampling one seed path per pixel and averaging over `Kmutations` accepted proposals.
  - `seed`: RNG seed controlling per-pixel mutation choices.
  - `mutator_type`: Selects the proposal mechanism (e.g. retrace, meshwalk, projection, resample).

- `static bool write_ppm(const std::string& path, const std::vector<glm::vec3>& img, int W, int H, float gamma=2.2f)`
  Writes an RGB image buffer to a binary PPM (P6) file with optional gamma correction.
  - `path`: Output filename.
  - `img`: Linear RGB buffer of length $W \cdot H$.
  - `W`, `H`: Image resolution matching `img`.
  - `gamma`: Gamma exponent used as $c \mapsto c^{1/\gamma}$ before quantization.

7.1.4 `Renderer` private helpers

- `glm::vec3 generate_primary_dir(int px, int py) const`
  Computes the normalized primary ray direction through pixel `(px,py)` using the camera basis and intrinsics.
- `glm::vec3 shading_normal_at(const PathMutator::Path& path, int i) const`
  Returns the (normalized) shading normal at vertex index `i` by barycentric interpolation of per-vertex normals, with geometric fallback.
- `static float clamp01(float x)`
  Clamps a scalar to $[0, 1]$.

7.2 `PathMutator` (`PathMutator.h`)

7.2.1 `PathMutator::Path`

- `vertices` : Path vertex positions $\{\mathbf{x}_i\}$, including camera and light endpoints.
- `faces` : Triangle hit at each vertex (dummy/default for non-surface endpoints).
- `bary_points` : Barycentric coordinates (w, u, v) of each surface vertex in its triangle.
- `light_visible` : For each internal surface vertex, cached boolean visibility to the light.
- `bounces` : Number of surface hits recorded (internal vertices count, excluding endpoints).

7.2.2 `PathMutator` methods

- `PathMutator(const Scene& scene, glm::vec3 camera_center, glm::vec3 light_pos)`
  Constructs a mutator bound to a scene, camera center $\mathbf{C}$, and light position $\mathbf{L}$.

- `scene`: Reference to triangle scene for intersection/visibility queries.
- `camera_center`: Camera point used as the first path vertex.
- `light_pos`: Light point appended as the terminal path vertex.

- `bool sample_path(Path& path, int maxBounces, const glm::vec3& initialDir, int retriesPerBounce=8) const`
  Samples a geometric path starting from the camera along `initialDir`, then extends via random hemisphere directions.
  - `path`: Output path container (cleared and filled on success).
  - `maxBounces`: Maximum surface hits to attempt.
  - `initialDir`: Primary ray direction from the camera.
  - `retriesPerBounce`: Attempts per bounce to find a direction that produces an intersection.

- `bool mutate_vertex_meshwalk(Path& path, int index, float radius) const`
  Local mutation: moves vertex `index` by walking tangentially across triangle adjacency within a given radius.
  - `path`: Path to mutate (updated in-place on success).
  - `index`: Vertex index to perturb (typically an internal surface vertex).
  - `radius`: Maximum local displacement scale on the surface.

- `bool mutate_vertex_project(Path& path, int index, float radius) const`
  Local mutation: samples a nearby tangent offset and projects back to the mesh via ray intersection.
  - `path`: Path to mutate (updated in-place on success).
  - `index`: Vertex index to perturb.
  - `radius`: Neighborhood scale for the proposal.

- `bool mutate_vertex_retrace(Path& path, int index) const`
  Global mutation: re-samples vertex `index` by drawing a new hemisphere direction and intersecting the scene.
  - `path`: Path to mutate (updated in-place on success).
  - `index`: Vertex index to replace (internal surface vertex).

7.3 GeomUtil (GeomUtil.h)

7.3.1 GeomUtil::Triangle

- `v0,v1,v2` : Triangle vertex positions.
- `n0,n1,n2` : Per-vertex normals (may be zero if not provided).

- `i0,i1,i2` : Global vertex indices used to identify shared mesh edges.
- `adj[3]` : Neighbor triangle indices across each local edge.
- `adjEdge[3]` : For each local edge, which edge index on the neighbor is shared.
- `color` : Per-triangle display color (used for OBJ export / debug geometry).

7.3.2 `GeomUtil` methods

- `static glm::vec3 safe_normalize(const glm::vec3& v)`
  Returns $\frac{\mathbf{v}}{\|\mathbf{v}\|}$ if $\|\mathbf{v}\| > 0$, otherwise returns a safe fallback direction.
- `static void make_orthonormal_basis(const glm::vec3& w, glm::vec3& u, glm::vec3& v)`
  Builds an orthonormal basis $\{u, v, w\}$ with `w` as the target axis.
  - `w`: Input axis (assumed nonzero / normalized internally).
  - `u,v`: Output orthonormal tangent vectors.
- `static bool moller_trumbore(const glm::vec3& ro, const glm::vec3& rd, const glm::vec3& v0, const glm::vec3& v1, const glm::vec3& v2, float& t, float& u, float& v, float eps=1e-8f)`
  Ray–triangle intersection returning (t, u, v) (ray distance and barycentric coordinates).
  - `ro, rd`: Ray origin and direction.
  - `v0,v1,v2`: Triangle vertices.
  - `t`: Output ray parameter at intersection.
  - `u,v`: Output barycentric coordinates (with $w = 1 - u - v$).
  - `eps`: Determinant threshold for near-parallel cases.
- `static glm::vec3 sample_hemisphere_uniform(const glm::vec3& n, std::mt19937& rng, std::uniform_real_distribution<float>& u01)`
  Samples a unit direction uniformly on the hemisphere oriented by normal `n`.
  - `n`: Hemisphere normal direction.
  - `rng`: Random number generator.
  - `u01`: Uniform distribution on $[0, 1]$ used for sampling.
- `static glm::vec3 face_normal_geom(const Triangle& t)`
  Returns the geometric face normal from $(v1 - v0) \times (v2 - v0)$.

- `static glm::vec3 project_to_plane(const glm::vec3& d, const glm::vec3& n)`
  Projects vector `d` onto the plane orthogonal to normal `n`.

- `static float cross2(const glm::vec2& a, const glm::vec2& b)`
  Returns the 2D scalar cross product $a_x b_y - a_y b_x$.

- `static glm::vec3 rodrigues(const glm::vec3& x, const glm::vec3& k_unit, float theta)`
  Rotates vector `x` about axis `k_unit` by angle `theta` (Rodrigues' formula).

- `static float signed_dihedral(const glm::vec3& n0, const glm::vec3& n1, const glm::vec3& e_unit)`
  Computes the signed dihedral angle between normals `n0` and `n1` around edge direction `e_unit`.

- `static void edge_endpoints(const Triangle& t, int edgeIdx, glm::vec3& a, glm::vec3& b)`
  Returns the endpoints `a,b` of triangle edge `edgeIdx` using the convention:

  `edgeIdx` $= 0 : (v_0, v_1)$, `edgeIdx` $= 1 : (v_1, v_2)$, `edgeIdx` $= 2 : (v_2, v_0)$.

7.4 Scene (scene.h)

7.4.1 `Scene::Hit`

- `t` : Ray parameter at the closest hit.
- `p` : Hit position $\mathbf{p} = \mathbf{o} + t\mathbf{d}$.
- `n` : Hit normal (shading normal if available, else geometric normal).
- `bary` : Barycentric coordinates (w, u, v) at the hit.
- `triIndex` : Index of the hit triangle in the scene triangle list.

7.4.2 `Scene` methods

- `bool load(const std::string& filename)`
  Loads meshes from disk, triangulates, builds a flat triangle list, and precomputes triangle adjacency.
  - `filename`: Input mesh path (any Assimp-supported format).

- `const std::vector<GeomUtil::Triangle>& triangles() const`
  Returns the scene triangle list.

- `const glm::vec3& bounds_min() const`, `const glm::vec3& bounds_max() const`
  Returns the axis-aligned bounding box corners computed during loading.

- `bool visible(const glm::vec3& origin, const glm::vec3& target, float eps=1e-4f) const`
  Returns `true` if the segment from `origin` to `target` is unoccluded.
  - `origin`: Segment start.
  - `target`: Segment end.
  - `eps`: Tolerance used to ignore hits very near `target` / self-occlusion.
- `bool visible_along_ray(const glm::vec3& origin, const glm::vec3& dir, float t_target, float eps=1e-4f) const`
  Returns `true` if no intersection occurs for $t < t_{\text{target}} - \varepsilon$ along the ray.
  - `origin`: Ray origin.
  - `dir`: Ray direction (assumed normalized by caller).
  - `t_target`: Maximum distance to test along the ray.
  - `eps`: Safety margin against numerical self-intersections.
- `bool intersect(const glm::vec3& origin, const glm::vec3& dir, Hit& outHit, GeomUtil::Triangle& tri) const`
  Intersects a ray against all triangles and returns the closest hit (if any).
  - `origin`: Ray origin.
  - `dir`: Ray direction.
  - `outHit`: Output hit record (filled on success).
  - `tri`: Output triangle data for the hit triangle.
- `void draw_ray(const glm::vec3& origin, const glm::vec3& dir, float length, const glm::vec3& color, float radius=0.0f, int sides=0)`
  Adds debug geometry representing a ray segment as a thin cylinder.
  - `origin`: Ray start position.
  - `dir`: Ray direction.
  - `length`: Segment length.
  - `color`: Debug color stored per triangle.
  - `radius`: Cylinder radius (if 0, a scene-scale default is chosen).
  - `sides`: Cylinder tessellation (if 0 or < 3, a minimum is enforced).
- `bool export_obj(const std::string& obj_path) const`
  Exports the scene with debug ray traces included.
  - `obj_path`: Output OBJ filename (MTL is written alongside it).