Technical Analysis: Ray Transport-Based Mesh Segmentation Algorithm

Ashton Jenson

Executive Summary

This report presents a detailed analysis of my ray transport-based mesh segmentation algorithm. The system leverages GPU ray tracing in conjunction with advanced graph theory to automatically partition 3D meshes into meaningful components. Through optimization techniques and careful implementation choices, the algorithm achieves efficient processing while producing high-quality segmentations that align with both geometric and functional characteristics of the input mesh.

The foundation of my system rests on simulating light transport within 3D meshes to identify natural segmentation boundaries. This approach differs fundamentally from traditional geometric methods by considering how light would theoretically propagate through an object, revealing functional and structural relationships between different mesh regions.

Algorithm Overview and Workflow

My mesh segmentation algorithm follows a multi-phase approach that combines physical simulation principles with advanced graph theory. Each phase builds upon the previous one to create a robust and efficient segmentation process.

Phase 1: System Initialization

The process begins with a comprehensive initialization phase where the system prepares the 3D mesh for processing. During this critical setup stage, the algorithm computes and caches essential geometric properties of the mesh, including face normals, centroids, and edge vectors. This pre-computation strategy significantly enhances performance during subsequent processing stages by eliminating redundant calculations.

Phase 2: Ray Generation and Transport Simulation

Following initialization, the algorithm implements a ray generation system. For each face within the mesh, the system generates a carefully designed set of rays to analyze potential ray transport paths. I originally had it pick randomly from within the bounding box, with a monte

carlo method. However, I found that even with a large amount of sampled points, there would be triangles that were missed in the clustering. So we have to pick an approach that inherently considers every triangle in the mesh.

The primary ray originates from the face's centroid and travels in the direction opposite to the face's normal vector. To ensure comprehensive coverage, the system generates additional rays within a 30-degree cone surrounding this primary direction. This multi-ray approach provides robust coverage of potential ray transport paths while maintaining computational efficiency.

The intersection detection employs an optimized version of the Möller-Trumbore algorithm to track ray intersections throughout the mesh. The system processes these intersections through multiple bounces, typically ten iterations, creating a detailed record of theoretical ray transport patterns. This phase leverages GPU acceleration through batch processing, allowing for efficient parallel computation of thousands of ray-triangle intersections simultaneously.

The ray transport system implements a configurable bounce analysis where the number of bounces can be adjusted based on specific requirements. While my testing established ten bounces as an effective default value, this parameter can be modified to accommodate different mesh characteristics and analysis requirements.

The number of bounces serves as a key configuration parameter that influences both the quality of segmentation and computational performance. Users can specify any number of bounces through the initialization parameters, allowing for fine-tuned control over the analysis depth.

Ray paths terminate under several conditions:

1. Reaching the maximum bounce count
2. Exiting the mesh through a back face (Moller Trumbore Failure)
3. When the ray's energy falls below a threshold due to accumulated numerical errors (Floating point Error)

Phase 3: Graph Structure Development

The algorithm transforms the accumulated intersection data into a graph structure that captures the essential patterns of ray transport within the mesh. Each mesh face becomes a vertex within this graph, while edges represent ray transitions between faces. The system assigns edge weights based on multiple factors, including transition frequency and characteristics, creating a rich representation of the mesh's internal structure and connectivity patterns.

Phase 4: Community Detection Analysis

Using the constructed graph, the system implements an enhanced version of the Louvain community detection algorithm to identify strongly connected groups of faces. This phase incorporates adaptive parameter adjustment based on mesh characteristics, ensuring

appropriate segmentation across varying mesh scales and complexities. The result is a preliminary segmentation that reflects the underlying ray transport patterns within the mesh.

Phase 5: Physical Connectivity Verification

The algorithm then ensures physical coherence through a robust connectivity enforcement system. Utilizing an optimized disjoint set data structure, the system verifies physical connectivity within each identified segment. This phase identifies and handles any disconnected components, ensuring that all final segments maintain physical coherence while preserving important boundary features.

Phase 6: Small Segment Processing

Following initial segmentation, the system identifies and processes segments that fall below a specified size threshold, typically set at 3% of the total mesh face count. The algorithm implements a merging process that considers multiple factors when combining small segments with their neighbors, including ray transport connectivity strength, physical adjacency patterns, and geometric compatibility measures.

Phase 7: Final Optimization

The final phase implements comprehensive refinement procedures to ensure optimal segmentation quality. This includes boundary smoothing operations, final connectivity verification, and coherence checks. The system performs these refinements while maintaining the fundamental principles established through the ray transport analysis.

Connectivity Management

Disjoint Set System

The disjoint set system implements both path compression and union by rank optimizations. This data structure efficiently maintains and updates connectivity information throughout the segmentation process. The implementation ensures nearly constant-time operations for both finding set representatives and merging sets.

The system maintains two key pieces of information for each element: a parent pointer and a rank value. The parent pointer allows for efficient set membership queries, while the rank value helps maintain balanced trees during merge operations.

Physical Connectivity Enforcement

My implementation ensures physical connectivity of segments through a refinement process. The system maintains a face adjacency graph that represents the physical connectivity of the mesh. This information is used to verify and enforce segment connectivity during the community detection and refinement phases.

Small Cluster Detection and Merging

My system implements an approach to handling small clusters. The process begins by identifying clusters below a specified size threshold, typically set to 3% of the total mesh face count. These small clusters are processed in order of increasing size to ensure stable merging behavior.

The merging process uses a weighted scoring system that considers multiple factors:

- The strength of ray transport connections between clusters
- Physical adjacency and boundary length
- Geometric compatibility based on normal vectors
- Relative cluster sizes to prevent creation of new small clusters

Optimization

GPU-Accelerated Ray Tracing Implementation

My implementation leverages PyTorch's GPU acceleration capabilities to achieve efficient ray-triangle intersection testing. The ray tracing system processes intersections in batches, with each batch containing thousands of rays. Through extensive testing, we determined that a batch size of 10,000 rays provides optimal performance on modern GPUs while maintaining reasonable memory usage. The batch processing approach allows us to fully utilize the GPU's parallel processing capabilities while avoiding memory constraints.

As stated before, the ray-triangle intersection testing employs the Möller-Trumbore algorithm with several crucial optimizations. We implemented vectorized operations that process multiple rays against multiple triangles simultaneously. The algorithm pre-computes and caches triangle edges and normals to minimize redundant calculations. Additionally, we implemented early termination checks that skip unnecessary computations when rays clearly miss triangles, significantly reducing processing time for complex meshes.

Memory Management and Data Structures

Memory efficiency proved crucial for handling larger meshes. My implementation uses custom sparse data structures to represent the ray transport graph, significantly reducing memory overhead compared to dense matrix representations. The system maintains a dynamic memory allocation strategy that releases intermediate computational results as soon as they are no longer needed, ensuring efficient memory utilization throughout the processing pipeline.

Clustering Implementation

The clustering phase employs an enhanced version of the Louvain community detection algorithm. My implementation includes several key optimizations. First, we use a multi-level approach that initially clusters at a coarse level and then refines the results. This hierarchical strategy significantly reduces the computational complexity while maintaining clustering quality.

The algorithm dynamically adjusts the resolution parameter based on mesh characteristics. Through empirical testing, we found that setting the initial resolution parameter to 1.0 and then adjusting it based on the mesh's face count and density produces optimal results. The resolution parameter is modified using a logarithmic scale relative to the mesh size, ensuring consistent clustering results across different mesh scales.

Disjoint Set Implementation

The Union-Find (Disjoint Set) data structure plays a crucial role in maintaining physical connectivity constraints. My implementation includes both path compression and union by rank optimizations, resulting in nearly constant-time operations. The data structure maintains an array of parent pointers and a rank array to ensure balanced trees.

The Union-Find implementation is particularly efficient due to its use of path compression during the find operation. When searching for a set's representative element, the algorithm updates all nodes along the path to point directly to the root, significantly reducing future query times. This optimization results in an amortized time complexity of $O(\alpha(n))$ per operation, where $\alpha(n)$ is the inverse Ackermann function.

Technical Limitations

Mesh Requirements

The current implementation requires manifold meshes without holes or self-intersections. This constraint arises from the need for consistent normal vectors and well-defined internal volumes for ray transport simulation. The STL format provides the most reliable results in my testing, as it ensures consistent triangle orientation and normal direction.

Scale Constraints

While the algorithm's theoretical complexity scales linearly with face count, practical limitations emerge with meshes exceeding one million faces. These constraints primarily stem from GPU memory limitations, though my batch processing approach helps mitigate these issues. The current implementation successfully handles most practical mesh sizes while maintaining reasonable processing times.

Future Improvements

Hardware Acceleration

The most promising avenue for performance improvement lies in leveraging modern GPU ray tracing hardware. Current GPU architectures, such as NVIDIA's RTX series, provide dedicated ray tracing cores that could significantly accelerate our intersection testing phase. Implementing the algorithm using compute shaders or hardware-accelerated ray tracing APIs like OptiX or DirectX Raytracing could potentially improve performance by an order of magnitude.

Algorithm Enhancements

Several algorithmic improvements could enhance the system's capabilities:

- Adaptive ray sampling based on local mesh complexity
- Multi-resolution analysis for handling large meshes
- Improved handling of non-manifold geometry
- Integration of machine learning techniques for parameter optimization

Scalability Improvements

To handle larger meshes more effectively, future versions could implement:

- Out-of-core processing for massive meshes
- Progressive refinement for interactive applications
- Distributed processing capabilities
- Hierarchical mesh representation

Conclusion

While our current implementation demonstrates the effectiveness of ray transport-based mesh segmentation, significant opportunities exist for improvement through modern graphics hardware and algorithmic refinements. The limitations we've identified primarily stem from implementation constraints rather than fundamental algorithmic issues, suggesting that future iterations could substantially expand the system's capabilities while maintaining its core advantages in identifying meaningful mesh segments.