Specification Document: Points In Mesh Project

Objective: Determine which points from a given set are inside a 3D mesh.

Implementation Overview:

1. Data Structures

- Point: Represents a 3D point with x, y, and z coordinates.
- Triangle: Represents a triangle in 3D space, composed of three vertices (Points).
- AABB (Axis-Aligned Bounding Box): Represents a 3D box defined by its minimum and maximum corners (Points).

2. Algorithms

- Ray-AABB Intersection: Determines if a ray intersects with an AABB. This is used as a quick preliminary check before the more detailed ray-triangle intersection (Slab method).
- Ray-Triangle Intersection: Determines if a ray intersects with a triangle (Moller-Trumbore implementation). This is the core algorithm to check if a point is inside the mesh.
- Point Inside Mesh: Uses the raycasting method to determine if a point is inside a mesh. It casts a ray in an arbitrary direction and counts how many times it intersects with the mesh. An odd number of intersections indicates the point is inside the mesh.

3. Utilities

- Loading Data: Functions to load points from a text file and triangles from an OBJ file (note: a library like tinyobjloader would be appropriate here if allowed).
- Saving Data: Functions to save points to a text file and to save points and triangles to an OBJ file for visualization.
- Computing Outside Points: Determines which points are outside the mesh by subtracting the inside points from the total set of points.

4. Main Execution

- Loads points and triangles from the provided files.

- Check if each point is inside the mesh in parallel. The number of threads is determined by the hardware concurrency of the machine.
- Collects and prints the results, including the number of points inside the mesh and the execution time.
- Saves the results for visualization and further analysis.

Algorithmic Choices and Considerations:

Due to the breadth of the field of computational geometry, determining the position of a point relative to a 3D mesh presents a variety of algorithmic options. The decision-making process was influenced by the need for efficiency, clarity, and the constraints of the project timeline.

Ray Casting vs. Winding Number:

- Ray Casting: By casting a ray and counting its intersections with the mesh, we can determine the position of a point. As a simple but elegant algorithm, it is a suitable choice for this project.
- Winding Number: While the Winding Number method offers a high degree of accuracy, especially for complex meshes, it is more computationally demanding. Given the project's constraints and emphasis on speed, Ray Casting was determined to be an appropriate choice.

Spatial Partitioning - Octrees:

- Octrees: Spatial partitioning, particularly using Octrees, can significantly optimize performance. By segmenting the 3D space, we can reduce the number of intersection checks. However, the complexity of Octrees might extend beyond the scope of this particular challenge.

AABB (Axis-Aligned Bounding Box):

- Incorporating AABBs provided a preliminary filtering mechanism. Before delving into detailed ray-triangle intersection checks, the AABBs efficiently assess potential intersections. This step contributes significantly to performance optimization.

Multi-threading:

- To further enhance the solution's efficiency, multi-threading was utilized to distribute the point-checking process across multiple CPU cores.

GPU Acceleration:

- As GPUs are designed for parallel processing they can drastically reduce execution times. A potential approach would involve frameworks like CUDA or OpenCL, which would parallelize the ray-mesh intersection checks. This presents a promising avenue for future iterations.

Performance Reflection: The current solution, bolstered by AABBs and multi-threading, is tailored for efficiency within the constraints of the challenge. However, the field of computational geometry is vast, and there are always opportunities for further refinement. Exploring techniques like Octrees or GPU-based acceleration could be the next steps in elevating the solution's performance.

The algorithm runs in approximately 40 seconds when utilizing 3 cores of a 2.6Ghz 6-core laptop-grade CPU (2019 Macbook Pro) and finds 2898 points within the mesh. On the same hardware with 1 thread manually selected, the algorithm runs in approximately 115 seconds with the same result.

Limitations and Improvements:

- Limitations: The raycasting method assumes that the mesh is a closed surface without any holes. If the mesh has holes or is not watertight, the results might be inaccurate.
- Improvements: The performance can be further improved by using spatial partitioning techniques like Octrees or BVH (Bounding Volume Hierarchy) to reduce the number of intersection checks.

User Interface (Not Implemented):

For the user interface, several avenues were considered to enhance visualization and user experience:

OpenGL Integration: OpenGL is a powerful, low-level graphics API, offering extensive control over rendering details. However, this introduces substantial complexity, necessitating significant development time, and changes the focus of this project.

Blender with Python Scripting: Blender's API provides a convenient platform for visualization through Python scripting, ensuring faster development. The trade-off is its confinement within Blender's environment, bringing along software-specific dependencies. Additionally, the use of Python code and the Blender environment may not satisfy the C++ emphasis of the project.

Three.js for Web Visualization: Three.js offers a browser-based solution, making it universally accessible without additional software installations. This accessibility, combined with a relatively fast development cycle, makes it an attractive choice. However, browser inconsistencies might pose occasional challenges and similar to Blender, the C++ emphasis of the project may be lost.

Testing Strategy (Partially Implemented):

Functionality Testing:

- Test the rayIntersectsTriangle function with known ray and triangle combinations to ensure it correctly identifies intersections.
- Test the isPointInsideMesh function with known points and mesh combinations to validate its accuracy.
- Test the rayIntersectsAABB function with known ray and bounding box combinations to ensure it correctly identifies intersections.

Boundary Testing:

- Use points that lie exactly on the edges of the mesh to test the algorithm's handling of boundary conditions.
- Use points that are very close to the mesh but not inside it to test the precision of the algorithm.

Integration Testing:

- Test the entire workflow from reading the input files, processing the points, and generating the output. Ensure that the output matches the expected results.

Performance Testing:

- Measure the time taken by the algorithm to process varying sizes of point sets and meshes. This will help in understanding the scalability of the solution.
- Test the multithreading aspect of the algorithm by running it on machines with different numbers of cores and comparing the performance.

Edge Cases:

- Use a mesh with complex geometries and holes to test the robustness of the algorithm.
- Use a very large number of points to test the efficiency and speed of the algorithm.
- Introduce noise or slight inaccuracies in the input data to see how the algorithm handles imperfect data.

Sample Test Cases:

- Simple Cube Mesh:
    - Mesh: A simple cube.
    - Points: A set of points both inside and outside the cube.
    - Expected Result: Points inside the cube are correctly identified.
- Complex Mesh with Holes:
    - Mesh: A complex shape with multiple holes.
    - Points: A set of points scattered around the mesh.
    - Expected Result: Points inside the mesh, including those inside the holes, are correctly identified.