3D Polyhedron Reconstruction and Analysis System

Your Name

October 19, 2024

Contents

1 Introduction

This system is designed to reconstruct 3D polyhedra from 2D projections, perform various geometric calculations, and apply transformations. The main components of the system include:

- Input processing and 3D reconstruction
- Geometric calculations (surface area, volume, center of mass)
- Moment of inertia calculation
- Orthographic projections (onto any plane)
- Transformations (rotation, translation, scaling, reflection)
- Validation of input and reconstruction

The system aims to provide a comprehensive tool for engineers and designers working with 3D polyhedra with "holes".

2 Detailed Algorithms

2.1 3D Reconstruction from 2D Projections

The system reconstructs 3D vertices from 2D projections using the following algorithm:

2.2 Surface Area Calculation

The surface area is calculated by summing the areas of all triangles formed by the edges of each face:

Algorithm 1 3D Reconstruction from 2D Projections

```
1: Input 2D coordinates for XY and XZ projections
2: Construct matrix A and vector B for each vertex
3: Solve the system AX = B using QR decomposition
4: Reconstruct 3D vertices from the solution X
```

Algorithm 2 Surface Area Calculation

```
1: totalArea = 0
2: for each face in polyhedron do
3:     if face is not internal then
4:         v0 = first vertex of the face
5:         for i = 1 to number of edges - 2 do
6:             v1 = (i+1)th vertex of the face
7:             v2 = (i+2)th vertex of the face
8:             edge1 = v1 - v0
9:             edge2 = v2 - v0
10:            triangleArea = magnitude(cross(edge1, edge2)) / 2
11:            totalArea += triangleArea
12:        end for
13:    end if
14: end for
15: return totalArea
```

2.3 Volume Calculation

The volume is calculated using the signed tetrahedron method:

2.4 Center of Mass Calculation

The center of mass is calculated using tetrahedral decomposition:

2.5 Moment of Inertia Calculation

The moment of inertia is calculated using the following algorithm:

2.6 Orthographic Projection

The orthographic projection is implemented as follows:

2.7 Transformations

Transformations are applied to each vertex of the polyhedron:

Specific transformations (rotation, translation, scaling, reflection) are implemented as matrix operations on individual vertices.

Algorithm 3 Volume Calculation

```
totalVolume = 0
for each face in polyhedron do
    if face is not internal then
        v0 = first vertex of the face
        for i = 1 to number of edges - 2 do
            v1 = (i+1)th vertex of the face
            v2 = (i+2)th vertex of the face
            tetrahedronVolume = dot(v0, cross(v1, v2)) / 6
            totalVolume += tetrahedronVolume
        end for
    end if
end for
return totalVolume
```

Algorithm 4 Center of Mass Calculation

```
totalVolume = 0
tempCenterOfMass = (0, 0, 0)
for each face in polyhedron do
    if face is not internal then
        v0 = first vertex of the face
        for i = 1 to number of edges - 2 do
            v1 = (i+1)th vertex of the face
            v2 = (i+2)th vertex of the face
            tetrahedronVolume = calculateTetrahedronVolume(v0, v1, v2)
            tetrahedronCentroid = calculateTetrahedronCentroid(v0, v1, v2)
            tempCenterOfMass += tetrahedronCentroid * tetrahedronVolume
            totalVolume += tetrahedronVolume
        end for
    end if
end for
if totalVolume ¿ 0 then
    centerOfMass = tempCenterOfMass / totalVolume
end if
return centerOfMass
```

Algorithm 5 Moment of Inertia Calculation

```
1: Calculate center of mass
2: Calculate volume and density
3: Initialize inertia tensor to zero
4: for each face in polyhedron do
5:     if face is not internal then
6:         v0 = first vertex of face - center of mass
7:         for i = 1 to number of edges - 2 do
8:             v1 = (i+1)th vertex of face - center of mass
9:             v2 = (i+2)th vertex of face - center of mass
10:            Calculate contribution to inertia tensor
11:            Add contribution to inertia tensor
12:        end for
13:    end if
14: end for
15: Scale inertia tensor by density
16: Calculate moment of inertia about specified axis
17: return moment of inertia
```

Algorithm 6 Orthographic Projection

```
1: Initialize SDL window and renderer
2: for each face in polyhedron do
3:     for each edge in face do
4:         Project edge vertices onto selected plane (XY, YZ, or XZ)
5:         Draw line between projected vertices
6:     end for
7: end for
8: Present rendered image
9: Wait for user to close window
```

Algorithm 7 Apply Transformation

```
1: for each face in polyhedron do
2:     for each edge in face do
3:         Apply transformation to edge.v1
4:         Apply transformation to edge.v2
5:     end for
6: end for
```

2.8 Input Validation

The input validation process includes the following checks:

Algorithm 8 Input Validation

```
1: Check edge length consistency
2: Check for collinearity of points in faces
3: Check for planarity of faces
4: Check if polyhedron is closed (each edge shared by exactly two faces)
5: if all checks pass then
6:     return true
7: else
8:     return false
9: end if
```

Check edge length consistency: Ensuring consistent edge lengths is crucial for maintaining the geometric integrity of the polyhedron. Inconsistent edge lengths can lead to distorted or impossible shapes, affecting the accuracy of subsequent calculations or visualizations. This check helps identify potential errors in the input data, such as measurement inaccuracies or data entry mistakes. Consistent edge lengths are particularly important for applications requiring precise geometric representations, such as computer-aided design, 3D printing, or structural analysis.

Check for collinearity of points in faces: Verifying that the points in each face are not collinear is essential for proper face definition. Collinear points would result in a degenerate face, which is essentially a line rather than a proper polygon. This can cause issues in various geometric algorithms, rendering processes, or physical simulations. Detecting and preventing collinear points ensures that each face is well-defined and contributes meaningfully to the overall structure of the polyhedron.

Check for planarity of faces: Ensuring that all points in a face lie on the same plane is critical for many geometric operations and representations. Nonplanar faces can introduce ambiguities in surface normal calculations, complicate intersection algorithms, and cause issues in rendering or 3D printing. Planar faces are often assumed in many geometric algorithms, and violating this assumption can lead to unexpected results or errors. Checking for planarity helps maintain the geometric validity and simplifies subsequent processing of the polyhedron.

Check if polyhedron is closed (each edge shared by exactly two faces): Verifying that the polyhedron is closed is fundamental to ensuring a valid 3D object. A closed polyhedron guarantees that there are no gaps or holes in the surface, which is crucial for applications involving volume calculations, fluid dynamics simulations, or 3D printing. Each edge being shared by exactly two faces confirms the topological correctness of the polyhedron. This check helps identify potential issues such as missing faces, overlapping geometry, or incorrectly connected components, all of which could lead to erroneous results in further analysis or manipulation of the polyhedron.

3 Data Structures

The system uses the following main data structures:

3.1 Vertex

```
struct Vertex {
    float x, y, z;
    bool operator<(const Vertex& other) const;
};
```

The Vertex structure represents a point in 3D space. The overloaded ¡ operator allows for sorting and use in map containers.

3.2 Edge

```
struct Edge {
    Vertex v1, v2;
    float length;
};
```

The Edge structure represents a line segment between two vertices, including its length.

3.3 Face

```
struct Face {
    vector<Edge> edges;
    bool is_internal;
};
```

The Face structure represents a polygonal face of the polyhedron, consisting of a vector of edges and a flag indicating whether it's an internal face.

3.4 Polyhedron

```
struct Polyhedron {
    vector<Vertex> vertices;
    vector<Face> faces;
};
```

The Polyhedron structure represents the entire 3D object, consisting of vertices and faces.

3.5 Justification

These data structures were chosen for the following reasons:

- **Efficiency**: The use of vectors for storing vertices, edges, and faces allows for dynamic sizing and fast access to elements.
- **Flexibility**: The structure allows for polyhedra with varying numbers of vertices and faces.
- **Ease of implementation**: The hierarchical structure (Polyhedron $\rightarrow$ Face $\rightarrow$ Edge $\rightarrow$ Vertex) mirrors the natural composition of 3D objects, making it intuitive to work with.
- **Compatibility**: These structures are easily compatible with linear algebra libraries like Eigen, which is used for some calculations.

The system also uses standard C++ containers like `std::map` for operations such as edge counting in the validation process, providing efficient lookup and insertion operations.

3.6 Libraries and Frameworks

This project primarily utilizes the Simple DirectMedia Layer (SDL) library for graphics rendering and user interaction. While alternatives like Qt and OpenGL are popular choices for graphics applications, SDL offers distinct advantages for our purposes. SDL is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. Unlike Qt, which is a comprehensive application framework, SDL's lightweight nature allows for more fine-grained control and easier integration with existing codebases. Compared to raw OpenGL, SDL provides a higher-level abstraction that simplifies window creation, event handling, and basic drawing operations. Crucially, SDL excels in multimodal interaction support, offering robust handling of various input devices beyond just keyboard and mouse. This multimodal capability makes SDL particularly suitable for creating more inclusive software that can accommodate diverse user needs and preferences. For instance, SDL's extensive support for game controllers and haptic feedback devices opens up possibilities for alternative input methods, potentially benefiting users with motor impairments. Furthermore, SDL's audio capabilities can be leveraged to provide auditory feedback, enhancing accessibility for visually impaired users. The library's broad platform support also ensures that the application can reach a wide audience across different operating systems, contributing to its inclusive nature. While Qt and OpenGL have their strengths in certain domains, SDL's balance of simplicity, performance, and multimodal support makes it an ideal choice for this project, aligning well with our goals of creating an efficient, flexible, and inclusive 3D polyhedron analysis system.

4 Consideration of Holes in Polyhedron Calculations

4.1 Introduction

In the implementation of polyhedron calculations, a crucial feature is the ability to handle complex shapes, including those with internal voids or holes. This is achieved through the use of an "internal" and "external" face flagging system. This section provides a detailed explanation of how this flagging mechanism allows for accurate calculations of surface area, volume, center of mass, and moment of inertia for polyhedra with holes.

4.2 Face Flagging Mechanism

Each face in the polyhedron is assigned a flag:

- `is_internal = 0`: Indicates an external face
- `is_internal = 1`: Indicates an internal face

This binary classification allows the algorithm to distinguish between faces that form the outer surface of the polyhedron and those that form the boundaries of internal voids.

4.3 Impact on Calculations

4.3.1 Surface Area Calculation

In the surface area calculation, only external faces contribute to the total surface area:

$$A_{total} = \sum_{f \in F_{external}} A_f \tag{1}$$

where $F_{external}$ is the set of all external faces, and A_f is the area of face f. This is implemented in the code as:

```
if (face.edges.size() < 3 || face.is_internal) continue;
```

This condition ensures that internal faces are excluded from the surface area sum, correctly accounting for holes in the polyhedron.

4.3.2 Volume Calculation

The volume calculation uses a similar approach:

$$V_{total} = \sum_{f \in F_{external}} V_f - \sum_{f \in F_{internal}} V_f \tag{2}$$

where V_f is the signed volume contribution of face f. In the code, this is implemented as:

```
if (face.is_internal) continue;
```

By skipping internal faces, the algorithm effectively subtracts the volume of any internal voids from the total volume of the polyhedron.

4.3.3 Center of Mass Calculation

The center of mass calculation also considers the presence of holes:

$$\vec{CM} = \frac{1}{V_{total}} \left(\sum_{f \in F_{external}} \vec{CM}_f V_f - \sum_{f \in F_{internal}} \vec{CM}_f V_f \right) \tag{3}$$

where $\vec{CM}_f$ is the center of mass contribution from face f. The code implements this by excluding internal faces:

```
if (face.edges.size() < 3 || face.is_internal) continue;
```

This ensures that the center of mass is correctly adjusted for any internal voids.

4.3.4 Moment of Inertia Calculation

The moment of inertia calculation is perhaps the most complex, as it requires considering both the mass distribution and the geometry of the polyhedron:

$$I = \int_V \rho(r) r^2 dV \tag{4}$$

where $\rho(r)$ is the density at position r. The presence of holes affects this calculation by altering the mass distribution. In the code, this is handled by the face flagging system:

```
if (face.edges.size() < 3 || face.is_internal) continue;
```

This ensures that the contribution of internal voids is properly accounted for in the moment of inertia calculation.

4.4 Conclusion

The internal/external face flagging system provides a robust mechanism for handling polyhedra with complex topologies, including those with internal voids or holes. By selectively including or excluding faces based on their flags, the algorithm ensures accurate calculations of geometric properties, even for non-trivial shapes. This approach allows for a more versatile and realistic representation of three-dimensional objects in computational geometry applications.

5 Functionality and Limitations

5.1 System Capabilities

The 3D Polyhedron Reconstruction and Analysis System can:

- Reconstruct 3D polyhedra from 2D projections (XY and XZ planes).
- Calculate geometric properties:
  - Surface area.
  - Volume.
  - Center of mass.
  - Moment of inertia about a specified axis.
- Perform orthographic projections onto standard planes (XY, YZ, XZ) and custom planes.
- Apply transformations:
  - Rotation about an arbitrary axis.
  - Translation.
  - Scaling.
  - Reflection about a coordinate plane.
- Validate input and reconstructed polyhedra.
- Visualize orthographic projections using SDL.

5.2 Limitations

The system has the following limitations and constraints:

- **Input format**: The system requires 2D projections on XY and XZ planes as input. It cannot handle other types of input such as 3D scans or single 2D projections.

- **Convex polyhedra**: The current implementation assumes convex polyhedra. Concave polyhedra may produce incorrect results for some calculations (e.g., volume).
- **Precision**: Floating-point arithmetic is used, which may lead to small inaccuracies in calculations due to rounding errors.
- **Performance**: The system is not optimized for very large polyhedra with thousands of vertices or faces. Performance may degrade for complex shapes.
- **Visualization**: The orthographic projection visualization is basic and does not include features like hidden line removal or shading.
- **File I/O**: The system cannot export results, and it does not currently support importing polyhedra from standard 3D file formats (e.g., OBJ, STL).
- **User interface**: The system uses a command-line interface, which may be less intuitive for users accustomed to graphical interfaces.
- **Error handling**: While basic input validation is implemented, the system may not gracefully handle all possible error cases or invalid inputs.
- **Coordinate system**: The system assumes a right-handed coordinate system. It may not correctly handle or convert between different coordinate systems.
- **Units**: The system does not explicitly handle units. Users must ensure consistent units are used for input and interpret output accordingly.
- **Topology changes**: The system does not support operations that would change the topology of the polyhedron (e.g., boolean operations, subdivision).

These limitations should be considered when using the system in an engineering drawing context. For many basic polyhedra and standard operations, the system should provide accurate and useful results. However, for more complex scenarios or specialized requirements, additional development or integration with more advanced 3D modeling software may be necessary.

6 Error Handling: Strengths and Weaknesses

6.1 Strengths

- **Input Validation**: The code includes a comprehensive `validateInput()` function that checks for various geometric inconsistencies, such as edge length discrepancies, collinearity, and non-planar faces.

- **Graceful Termination**: In cases where invalid input is detected, the program exits with an appropriate error code (e.g., `return 1;`).
- **User Feedback**: The code provides informative error messages to the user, specifying which validation check failed (e.g., "Edge length inconsistency detected for edge X in face Y").
- **Boundary Checks**: Some functions, like `calculateTetrahedronVolume()`, implicitly handle edge cases (e.g., when the volume is zero).

6.2 Weaknesses

- **Lack of Exception Handling**: The code primarily uses return values and conditional statements for error handling instead of C++ exceptions, which could provide more robust error propagation.
- **Inconsistent Error Reporting**: Some functions (e.g., `calculateSurfaceArea()`) don't have explicit error handling for edge cases like empty polyhedra.
- **Limited Input Sanitization**: While there is geometric validation, there's minimal sanitization of user inputs for non-geometric data (e.g., checking for valid numeric input when reading coordinates).
- **Absence of Logging**: The program lacks a systematic logging mechanism, which could aid in debugging and tracking the flow of execution, especially for complex geometric operations.
- **Minimal Recovery Mechanisms**: In most error scenarios, the program opts to terminate rather than attempting to recover or prompt the user for corrective action.
- **Lack of Error Codes**: The program doesn't use defined error codes, which could make it difficult for potential calling programs to interpret the nature of failures programmatically.

7 Project Structure and IO

8 Input and Output Formats

8.1 Input Format

Our system accepts input in the following format:

- Raw 2D coordinates for vertices
- Face definitions using vertex indices
- Internal/external face flags

The input is provided through a custom file format with the following structure:

```
<num_vertices>
<2Dvertices - XY, XZ>
<num_faces>
<num_edges_face1> <is_internal_face1>
<v1> <v2> ... <vk>
...
<num_edges_faceN> <is_internal_faceN>
<v1> <v2> ... <vm>
```

Where:

- `<num_vertices>` is the total number of vertices
- `<xi> <yi> <zi>` are the 3D coordinates of each vertex
- `<num_faces>` is the total number of faces
- `<num_edges_facei>` is the number of edges in face i
- `<is_internal_facei>` is 1 if the face is internal, 0 if external
- `<v1> <v2> ...` are the vertex indices (0-based) defining each face

8.2 Output Format

The system provides output in the following formats:

1. Console output for numerical results (e.g., surface area, volume, center of mass, moment of inertia)
2. Graphical output using SDL2 for orthographic projections

8.2.1 Input Format for 3D Object Description

The input format for describing a 3D object consists of several components:

Number of Faces A single integer representing the total number of faces in the 3D object.

2D Projections of Vertices For each vertex, two sets of coordinates are provided:

- (x, y) coordinates for the XY plane projection
- (x, z) coordinates for the XZ plane projection

Each vertex is labeled (e.g., A, B, C, D) and given a numeric identifier.

Face and Edge Descriptions For each face, the following information is provided:

- Number of edges in the face
- Whether the face is internal (0 for no, 1 for yes)
- For each edge in the face:
  - The two vertices that form the edge, represented by their numeric identifiers

This format allows for a complete description of the 3D object's geometry, including its vertices, edges, and faces, as well as their spatial relationships.

Example outputs:

```
Surface Area: 123.45 square units
Volume: 67.89 cubic units
Center of Mass: (1.23, 4.56, 7.89)
Moment of Inertia: [
    [10.1, 0.2, 0.3],
    [0.2, 20.2, 0.4],
    [0.3, 0.4, 30.3]
]
```

For orthographic projections, the system will display a 2D rendering of the polyhedron using SDL2.

9 Project Structure

The project is organized into the following files:

- `polyhedron.h`: Header file containing the definitions for the `Vertex`, `Edge`, `Face`, and `Polyhedron` structures.
- `polyhedron.cpp`: Implementation file all functions together.
- `main.cpp`: The main entry point of the program, handling user interaction and coordinating the various operations.
- `input.h`: Header file declaring functions for input handling and parsing.
- `transformations.h`: Header file declaring functions for geometric transformations (rotation, translation, scaling, reflection).
- `projections.h`: Header file declaring functions for orthographic projections and custom plane projections.
- `input.cpp`: Implementation of input handling functions, including file parsing and data validation.

- `transformations.cpp`: Implementation of geometric transformation functions.
- `projections.cpp`: Implementation of projection functions, including SDL2 rendering for visualization.

This structure separates concerns and promotes modularity, making it easier to maintain and extend the codebase. The `.h` files contain function declarations and structure definitions, while the `.cpp` files contain the actual implementations. The `main.cpp` file ties everything together, providing the user interface and program flow.