3D Polyhedron Reconstruction and Analysis System

Your Name

October 28, 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) (equivalent to slicing)
- Transformations (rotation, translation, scaling, reflection)
- Validation of input and reconstruction
- 3D Dynamic Projection (control through cursor)

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

Note that the scientific definition of the polyhedron is from ScienceDirect: "A polyhedron is a three-dimensional solid bounded by a finite number of polygons called faces." The bounded nature of the polyhedron implies it is closed. https://www.sciencedirect.com/topics/mathematics/polyhedron

2 Data Structures

The system uses the following main data structures:

2.1 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.

2.2 Edge

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

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

2.3 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.

2.4 Polyhedron

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

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

2.5 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.

2.6 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.

3 Consideration of Holes in Polyhedron Calculations

3.1 Introduction

In the implementation of polyhedron calculations, handling complex shapes with internal voids or holes is achieved through a recursive structure of polyhedrons, where each primary polyhedron can contain "sub-polyhedrons" representing its internal voids. This allows for accurate calculations of surface area, volume, center of mass, and moment of inertia by recursively including or excluding these sub-structures.

3.2 Polyhedron Structure with Sub-Polyhedrons

Each polyhedron can contain:

- An array of external faces that form the outer surface.
- A vector of sub-polyhedrons representing internal voids.

This structure enables the algorithm to process each sub-polyhedron recursively, effectively distinguishing between the primary body of the polyhedron and its internal voids.

3.3 Impact on Calculations

3.3.1 Surface Area Calculation

For surface area calculation, only external faces of the main polyhedron are included:

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

where $F_{external}$ is the set of all external faces, and A_f is the area of face f. This ensures that faces forming internal voids do not contribute to the surface area of the polyhedron.

3.3.2 Volume Calculation

Volume calculation considers the primary polyhedron and its sub-polyhedrons:

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

where V_f is the signed volume contribution of face f. By subtracting the volume of sub-polyhedrons, the algorithm correctly accounts for the volume of internal voids.

3.3.3 Center of Mass Calculation

The center of mass calculation adjusts for internal voids as follows:

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

where $\vec{CM}_f$ is the center of mass contribution from face f. The algorithm excludes contributions from sub-polyhedrons representing internal voids, allowing accurate computation of the overall center of mass.

3.3.4 Moment of Inertia Calculation

Moment of inertia calculation is updated by considering the mass distribution and geometry of each sub-polyhedron:

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

where $\rho(r)$ is the density at position r. Contributions from sub-polyhedrons are subtracted to account for voids. This is handled by recursively adding or subtracting the tensor contributions for each sub-polyhedron, ensuring that the mass distribution reflects the internal voids.

3.4 Conclusion

The sub-polyhedron structure provides a robust mechanism for handling complex polyhedra with internal voids. By recursively processing sub-polyhedrons, the algorithm ensures accurate calculations of surface area, volume, center of mass, and moment of inertia, making it suitable for three-dimensional objects with complex topologies.

4 Detailed Algorithms

4.1 3D Reconstruction from 2D Projections

The system reconstructs 3D vertices from 2D projections in the XY and XZ planes by solving a system of equations for each vertex. The algorithm leverages QR decomposition for efficient computation.

4.2 Surface Area Calculation

The surface area is computed by summing the areas of triangles formed by vertices on each face of the polyhedron. Each face is checked to ensure it is not an internal face, ensuring only external surface area is included.

Algorithm 1 3D Reconstruction from 2D Projections

```
Input: 2D coordinates for each vertex in XY and XZ projections
Output: 3D coordinates for each vertex
for each vertex with known 2D coordinates do
    Construct matrix $A$ based on projection relationships
    Construct vector $B$ based on 2D coordinates
    Solve the system $AX = B$ using QR decomposition, where $X$ contains the 3D coordinates
    Store the solution $X$ as the reconstructed 3D coordinates for the vertex
end for
return all reconstructed 3D vertices
```

Algorithm 2 Surface Area Calculation

```
Input: List of faces with vertices
Output: Total surface area of the polyhedron
$totalArea \leftarrow 0$
for each face $f$ in polyhedron do
    if face is external then
        $v_0 \leftarrow$ first vertex of face $f$
        for $i = 1$ to number of edges $- 2$ do
            $v_1 \leftarrow (i+1)$th vertex of face $f$
            $v_2 \leftarrow (i+2)$th vertex of face $f$
            $edge_1 \leftarrow v_1 - v_0$
            $edge_2 \leftarrow v_2 - v_0$
            $triangleArea \leftarrow \text{magnitude}(\text{cross}(edge_1, edge_2))/2$
            $totalArea \leftarrow totalArea + triangleArea$
        end for
    end if
end for
return $totalArea$
```

4.3 Volume Calculation

The volume is calculated using the signed tetrahedron method, which breaks down the polyhedron into tetrahedrons formed by the origin and each face of the polyhedron.

Algorithm 3 Volume Calculation

```
 1: **Input:** List of faces with vertices
 2: **Output:** Total volume of the polyhedron
 3: $totalVolume \leftarrow 0$
 4: for each face $f$ in polyhedron do
 5:     if face is external then
 6:         $v_0 \leftarrow$ first vertex of face $f$
 7:         for $i = 1$ to number of edges $- 2$ do
 8:             $v_1 \leftarrow (i+1)$th vertex of face $f$
 9:             $v_2 \leftarrow (i+2)$th vertex of face $f$
10:             $tetrahedronVolume \leftarrow \mathrm{dot}(v_0, \mathrm{cross}(v_1, v_2))/6$
11:             $totalVolume \leftarrow totalVolume + tetrahedronVolume$
12:         end for
13:     end if
14: end for
15: return $totalVolume$
```

4.4 Center of Mass Calculation

The center of mass is calculated by decomposing the polyhedron into tetrahedrons. Each tetrahedron's centroid and volume are used to calculate the weighted center of mass.

4.5 Moment of Inertia Calculation

The moment of inertia tensor is calculated by integrating over the polyhedron's volume. Contributions from each face are accumulated and scaled by density.

4.6 Orthographic Projection

This procedure projects the polyhedron's edges onto a specified 2D plane for visual representation.

4.7 Transformations

Transformations are applied to each vertex in the polyhedron, affecting its position and orientation in space.

Algorithm 4 Center of Mass Calculation

```
1: **Input:** List of faces with vertices
2: **Output:** Center of mass coordinates
3: $totalVolume \leftarrow 0$
4: $tempCenterOfMass \leftarrow (0, 0, 0)$
5: for each face $f$ in polyhedron do
6:     if face is external then
7:         $v_0 \leftarrow$ first vertex of face $f$
8:         for $i = 1$ to number of edges $- 2$ do
9:             $v_1 \leftarrow (i + 1)$th vertex of face $f$
10:            $v_2 \leftarrow (i + 2)$th vertex of face $f$
11:            $tetrahedronVolume \leftarrow$ calculateTetrahedronVolume($v_0, v_1, v_2$)
12:            $tetrahedronCentroid \leftarrow$ calculateTetrahedronCentroid($v_0, v_1, v_2$)
13:            $tempCenterOfMass \leftarrow tempCenterOfMass + (tetrahedronCentroid \times tetrahedronVolume)$
14:            $totalVolume \leftarrow totalVolume + tetrahedronVolume$
15:        end for
16:    end if
17: end for
18: if $totalVolume > 0$ then
19:     $centerOfMass \leftarrow tempCenterOfMass/totalVolume$
20: end if
21: return $centerOfMass$
```

Algorithm 5 Moment of Inertia Calculation

```
1: Input: Polyhedron vertices and faces
2: Output: Moment of inertia tensor
3: Calculate center of mass $CM$
4: Calculate total volume and density
5: Initialize inertia tensor $I \leftarrow 0$
6: for each face $f$ in polyhedron do
7:     if face is external then
8:         $v_0 \leftarrow$ first vertex of face $f - CM$
9:         for $i = 1$ to number of edges $- 2$ do
10:            $v_1 \leftarrow (i + 1)$th vertex of face $f - CM$
11:            $v_2 \leftarrow (i + 2)$th vertex of face $f - CM$
12:            Calculate contribution to inertia tensor from $v_0, v_1, v_2$
13:            $I \leftarrow I +$ contribution
14:        end for
15:    end if
16: end for
17: Scale inertia tensor $I$ by density
18: return $I$
```

Algorithm 6 Orthographic Projection

```
 1: Input: Polyhedron faces and edges
 2: Output: Rendered orthographic projection
 3: Initialize SDL window and renderer
 4: for each face in polyhedron do
 5:     for each edge in face do
 6:         Project vertices of edge onto chosen plane (XY, YZ, or XZ)
 7:         Draw line between projected vertices
 8:     end for
 9: end for
10: Present rendered image on screen
11: Wait for user to close window
```

Algorithm 7 Apply Transformation

```
1: Input: Transformation matrix, polyhedron faces and edges
2: Output: Transformed polyhedron
3: for each face in polyhedron do
4:     for each edge in face do
5:         Apply transformation matrix to $edge.v_1$
6:         Apply transformation matrix to $edge.v_2$
7:     end for
8: end for
9: return transformed polyhedron
```

4.8 Input Validation

Input validation ensures the polyhedron's geometric and topological correctness by checking consistency and closure properties.

Algorithm 8 Input Validation

```
Input: Polyhedron faces and edges
Output: Boolean indicating if input is valid
Check that all edge lengths are consistent
Check that points on each face are not collinear
Check that all points in each face are coplanar
Check if polyhedron is closed (each edge is shared by exactly two faces)
if all checks pass then
    return true
else
    return false
end if
```

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).

Prevent Program Crash: There is functionality to prevent crash, for eg: if user inputs a letter instead of a number, there are checks for that to prevent the program from crashing or giving outputs without stopping.

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.

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
Volume: 67.89
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:

- `input.h`: Header file declaring functions for input handling and parsing. Also defining the `Vertex`, `Edge`, `Face`, and `Polyhedron` and other structures.
- `validity.h`: Header file declaring functions for data validation and error checking.

- `geometry.h`: Header file defining the functions for computing geometric properties (e.g., surface area, volume).
- `projections.h`: Header file declaring functions for orthographic projections and custom plane projections.
- `constants.h`: Header file containing all necessary constant values used throughout the project.
- `transformations.h`: Header file declaring functions for geometric transformations (rotation, translation, scaling, reflection).
- `input.cpp`: Implementation of input handling functions, including file parsing and data validation.
- `validity.cpp`: Implementation of data validation and error-checking functions.
- `geometry.cpp`: Implementation of functions to calculate geometric properties and perform operations on vertices, edges, and faces.
- `projections.cpp`: Implementation of projection functions, including SDL2 rendering for visualization.
- `transformations.cpp`: Implementation of geometric transformation functions.
- `main.cpp`: The main entry point of the program, handling user interaction and coordinating the various operations.

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.