Detailed Design CAD System - A4

Ziv Barretto

October 28, 2024

Contents

1 3D to 2D

This document provides a detailed design of a software system that performs various operations on 3D objects, including projecting them onto 2D planes (orthographic projections), applying geometric transformations, slicing the objects with a plane, performing hidden line removal, and calculating surface area and volume. The system is implemented in C++ and includes visualization using OpenGL.

2 Detailed Algorithms

This section explains the algorithms implemented in the system, including mathematical formulations and pseudocode where necessary.

2.1 Orthographic Projections

Orthographic projections are used to project a 3D object onto 2D planes without perspective distortion. The main views are the Top View (projection onto the XY plane), Front View (projection onto the XZ plane), and Side View (projection onto the YZ plane).

2.1.1 Projection Matrices

Orthographic projection can be represented using projection matrices. For each view, we use a matrix that eliminates one coordinate.

- **Top View (XY plane)**: The projection matrix P_{XY} is:

$$P_{XY} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Front View (XZ plane)**: The projection matrix P_{XZ} is:

$$P_{XZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Side View (YZ plane)**: The projection matrix P_{YZ} is:

$$P_{YZ} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.1.2 Projection Algorithm

For each vertex $\mathbf{v} = [x, y, z, 1]^T$ in the 3D object, we compute the projected point $\mathbf{v}'$ by multiplying it with the projection matrix:

$$\mathbf{v}' = P \cdot \mathbf{v}$$

Pseudocode:

Listing 1: Projection onto a plane

```
for each vertex v in object.vertices:
    if plane == "XY":
        projected_vertex = Vertex(v.x, v.y, 0)
    else if plane == "XZ":
        projected_vertex = Vertex(v.x, 0, v.z)
    else if plane == "YZ":
        projected_vertex = Vertex(0, v.y, v.z)
    add projected_vertex to projection.vertices
```

2.2 Geometric Transformations

Geometric transformations are applied to the 3D object to manipulate its position, orientation, and size. The primary transformations are rotation, translation, and scaling.

2.2.1 Transformation Matrices

- **Rotation about the X-axis by angle θ:**

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Rotation about the Y-axis by angle θ:**

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Rotation about the Z-axis by angle θ:**

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Translation by vector $\mathbf{t} = [t_x, t_y, t_z]^T$:**

$$T(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Scaling by factor s:**

$$S(s) = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.2 Transformation Algorithm

To apply a transformation to a vertex $\mathbf{v}$, we multiply it by the transformation matrix M:

$$\mathbf{v}' = M \cdot \mathbf{v}$$

Pseudocode for Rotation:

Listing 2: Rotating around an axis

```
for each vertex v in object.vertices:
    pos = glm::vec4(v.x, v.y, v.z, 1.0)
    pos = rotation_matrix * pos
    v.x = pos.x
    v.y = pos.y
    v.z = pos.z
```

2.3 Slicing the Object

Slicing involves cutting the 3D object with a plane, resulting in two separate objects. The plane is defined by a point and a normal vector.

2.3.1 Plane Equation

A plane can be defined by the equation:

$$Ax + By + Cz + D = 0$$

where $[A, B, C]$ is the normal vector $\mathbf{n}$ of the plane, and $D = -\mathbf{n} \cdot \mathbf{p}$ for a point $\mathbf{p}$ on the plane.

2.3.2 Slicing Algorithm

The algorithm involves:

1. **Classifying vertices:** Determine whether each vertex lies above, below, or on the slicing plane by evaluating the plane equation.
2. **Processing edges:** For edges crossing the plane, compute the intersection point.
3. **Constructing new objects:** Create two new objects (above and below the plane) by assembling vertices and faces based on their classification.

Pseudocode:

Listing 3: Slicing the object

```
for each vertex v in object.vertices:
    d = dot(plane.normal, v - plane.point)
    if d > 0:
        classify v as above
    else if d < 0:
        classify v as below
    else:
        classify v as on the plane

for each edge e in object.edges:
    if e crosses the plane:
        compute intersection point
        add intersection point to both objects
    else:
        add edge to the appropriate object
```

2.4 Hidden Line Removal

Hidden line removal determines which edges of the object are visible from a given viewpoint (projection plane) and which are obscured by faces.

2.4.1 Algorithm Overview

The algorithm involves:

1. **Projecting faces onto the plane:** Compute the 2D projection of each face.
2. **Determining visibility:** For each edge, check if its midpoint lies within any projected face that is closer to the viewer.
3. **Classifying edges:** Edges are classified as visible or hidden based on the above check.

2.4.2 Point-in-Polygon Test

To check if a point lies within a polygon, we use the ray casting algorithm.

Ray Casting Algorithm:

1. Cast a horizontal ray to the right from the point.
2. Count the number of times the ray intersects the polygon's edges.
3. If the count is odd, the point is inside; if even, it's outside.

2.4.3 Pseudocode for Edge Classification

Listing 4: Classifying edges

```
for each edge e in object.edges:
    project edge onto plane
    compute midpoint m of projected edge
    isHidden = false
    for each face f:
        if m lies inside projected face f:
            if face is closer to viewer:
                isHidden = true
                break
    if isHidden:
        classify edge as hidden
    else:
        classify edge as visible
```

2.5 Surface Area and Volume Calculation

2.5.1 Surface Area

The surface area is calculated by summing the areas of all the faces.

Area of a Triangle:

For a triangle with vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$:

$$\text{Area} = \frac{1}{2}\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|$$

Algorithm:

Listing 5: Calculating surface area

```
totalArea = 0
for each face in object.faces:
    if face has at least 3 vertices:
        for i from 1 to face.vertices.size() - 2:
            v0 = face.vertices[0]
            v1 = face.vertices[i]
            v2 = face.vertices[i+1]
            area = 0.5 * length(cross(v1 - v0, v2 - v0))
            totalArea += area
```

2.5.2 Volume

The volume is calculated using the divergence theorem (also known as Gauss's theorem), summing the volumes of tetrahedra formed between each face and the origin.

Volume of a Tetrahedron:

$$\text{Volume} = \frac{1}{6}|\mathbf{v}_0 \cdot ((\mathbf{v}_1 \times \mathbf{v}_2))|$$

Algorithm:

Listing 6: Calculating volume

```
totalVolume = 0
for each face in object.faces:
    if face has at least 3 vertices:
        v0 = face.vertices[0]
        for i from 1 to face.vertices.size() - 2:
            v1 = face.vertices[i]
            v2 = face.vertices[i+1]
            volume = (1.0/6.0) * dot(v0, cross(v1, v2))
            totalVolume += volume
totalVolume = abs(totalVolume)
```

2.6 Edge Intersection Processing

To handle hidden lines and slicing correctly, we process edge intersections to subdivide edges at intersection points.

2.6.1 Algorithm

1. **Iterate over edge pairs:** For each pair of edges, project them onto the same plane.
2. **Check for intersection:** Determine if the projected edges intersect.
3. **Compute intersection point:** If they intersect, compute the intersection point in 3D space.
4. **Split edges:** Replace original edges with new edges subdivided at the intersection point.

Pseudocode:

Listing 7: Edge intersection processing

```
for each plane in ["XY", "XZ", "YZ"]:
    for each pair of edges (e1, e2):
        project e1 and e2 onto plane
        if projected edges intersect:
            compute 3D intersection point
            add new vertex at intersection point
            split e1 and e2 at intersection point
            update edges
```

3 Data Structures

3.1 Classes and Their Relationships

- **Vertex**: Represents a point in 3D space with coordinates x, y, z.
- **Edge**: Connects two vertices, storing indices of the start and end vertices.

- **Face**: Represents a polygonal face of the object, storing a list of vertex indices.
- **Object3D**: Contains lists of vertices, edges, and faces, representing the entire 3D object.
- **Projection2D**: Stores the projected vertices and edges, including classifications of visible and hidden edges.
- **Plane**: Defines a slicing plane using a point and a normal vector.
- **Slicer**: Contains methods to slice the object using a plane.
- **Renderer**: Handles visualization using OpenGL, including managing shaders, buffers, and user input.

3.2 Justification of Data Structures

- **Vectors (std::vector)**: Used extensively to store lists of vertices, edges, and faces due to their dynamic resizing capabilities and efficient element access.
- **Maps (std::map)**: Used in slicing to map original vertex indices to new indices, which helps in tracking and updating vertices efficiently.
- **Sets (std::set)**: Used to keep track of edges to remove or process, ensuring uniqueness and efficient lookup.
- **Classes and Structs**: Encapsulate data and related functions, promoting modularity and reusability.

4 Input/Output Formats

4.1 Input Format

The system accepts a custom text-based file format to define the 3D object.

- **Number of Vertices (V)**: An integer specifying the total number of vertices.
- **Vertex Coordinates**: V lines, each containing three floating-point numbers representing x, y, z coordinates.
- **Number of Edges (E)**: An integer specifying the total number of edges.
- **Edge Indices**: E lines, each containing two integers representing indices of the vertices that form the edge.
- **Number of Faces (F)**: An integer specifying the total number of faces.
- **Faces Definition**: F blocks, each starting with an integer N (number of vertices in the face), followed by N integers representing vertex indices.

Example:

```
8
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
12
```

```
0 1
1 2
2 3
3 0
4 5
5 6
6 7
7 4
0 4
1 5
2 6
3 7
6
4 0 1 2 3
4 4 5 6 7
4 0 1 5 4
4 1 2 6 5
4 2 3 7 6
4 3 0 4 7
```

4.2 Output Formats

- **Projection Images**: PNG images combining top, front, and side views, saved using the `stb_image_write` library.
- **Projection Text Files**: Text files listing visible and hidden edges for each projection.
- **Console Output**: Surface area and volume calculations are printed to the console.

5 Project Structure

5.1 Directory and File Overview

- **src/**: Contains all the source code files.
  - **main.cpp**: Entry point of the application.
  - **object3d.h/.cpp**: Definition and implementation of the `Object3D` class.
  - **vertex.h**: Definition of the `Vertex` class.
  - **edge.h**: Definition of the `Edge` class.
  - **face.h**: Definition of the `Face` class.
  - **orthographic_projections.h/.cpp**: Functions for projections and rendering.
  - **transformations.h/.cpp**: Geometric transformation functions.
  - **slicer.h/.cpp**: Slicing functionality.
  - **renderer.h/.cpp**: OpenGL rendering.
  - **file_io.h/.cpp**: Functions for reading the object from a file.
- **build/**: Build directory containing compiled binaries and output files.
- **output/**: Directory where output images and text files are saved.
- **include/**: Directory containing header files.
- **lib/**: Directory for external libraries.
- **CMakeLists.txt**: Build configuration file for CMake.

6 Functionality Limitations

- **Input Format**: The system only accepts the custom text-based input format described. It does not support standard formats like OBJ or STL.
- **Complex Objects**: The system may not handle very complex objects with a large number of vertices and faces efficiently.
- **Overlapping Faces**: The hidden line removal algorithm assumes non-overlapping faces. Objects with overlapping faces may not render correctly.
- **User Interaction**: The slicing plane can only be adjusted via keyboard inputs during rendering. There is no graphical user interface for more intuitive control.
- **Limited Error Handling**: While basic error handling is implemented, the system may not gracefully handle all invalid inputs.
- **Visualization****: The renderer uses wireframe mode and does not support textures or advanced shading techniques.

7 2D to 3D Reconstruction System

This document provides a detailed design of a software system that reconstructs a 3D wireframe model from three orthographic 2D projections: Top View, Front View, and Side View. The system processes input data, identifies probable 3D vertices and edges, generates faces, and removes pseudo-elements using decision rules and algorithms. The implementation is in C++, and this document elaborates on the algorithms, mathematical formulations, and data structures used.

7.1 Input Data Processing

The system begins by reading the input data from a file containing the 2D projections. Each projection includes vertices and edges.

7.1.1 Reading Input Files

The input file contains sections for each view:

- **Top View**
- **Front View**
- **Side View**

For each view, the format is:

1. The number of vertices.
2. A list of vertices, each with x and y coordinates.
3. The number of edges.
4. A list of edges, each with the indices of the start and end vertices and the line type (solid or dashed).

7.1.2 Data Structures for 2D Graphs

- **Graph2D**: Represents a 2D projection, containing vertices and edges.
- **Point2D**: Represents a point in 2D space.
- **Line2D**: Represents an edge between two points in 2D space.

7.1.3 Reading and Parsing

The function `readGraphsFromFile` reads the input file and populates the `Graph2D` objects for each view.

7.2 Removing Duplicate Vertices

Duplicate vertices may exist due to redundant entries. The system removes duplicates by comparing all vertices and keeping unique ones.

Algorithm:

1. Initialize an empty list for unique vertices.
2. For each vertex in the original list:
   (a) Check if it already exists in the unique list.
   (b) If not, add it to the unique list.
   (c) Update a mapping from old indices to new indices.
3. Update edge indices to match the new vertex indices.

7.3 Processing Intersections

Edges in 2D views may intersect at points not explicitly specified as vertices. The system identifies these intersections and adds them as vertices.

7.3.1 Line Segment Intersection

Given two line segments $\overline{A_1A_2}$ and $\overline{B_1B_2}$, the intersection point can be found using parametric equations.

Parametric Representation:

$$\begin{cases} x = x_{A1} + s(x_{A2} - x_{A1}) \\ y = y_{A1} + s(y_{A2} - y_{A1}) \end{cases}, \quad 0 \le s \le 1$$

$$\begin{cases} x = x_{B1} + t(x_{B2} - x_{B1}) \\ y = y_{B1} + t(y_{B2} - y_{B1}) \end{cases}, \quad 0 \le t \le 1$$

Solving for Parameters s and t: Set the equations equal to each other and solve for s and t:

$$x_{A1} + s(x_{A2} - x_{A1}) = x_{B1} + t(x_{B2} - x_{B1})$$
$$y_{A1} + s(y_{A2} - y_{A1}) = y_{B1} + t(y_{B2} - y_{B1})$$

This leads to two equations with two unknowns.

Checking Intersection Conditions: If $0 \le s \le 1$ and $0 \le t \le 1$, the segments intersect at:

$$\begin{cases} x = x_{A1} + s(x_{A2} - x_{A1}) \\ y = y_{A1} + s(y_{A2} - y_{A1}) \end{cases}$$

7.3.2 Algorithm for Processing Intersections

1. For each pair of edges:
   (a) Check if they intersect.
   (b) If they do, compute the intersection point.
   (c) Add the intersection point to the list of vertices (if not already present).
   (d) Split the original edges at the intersection point, creating new edges.

7.4 Handling Collinear Lines

Collinear edges connected at a common vertex are merged into a single edge to simplify the graph.

7.4.1 Collinearity Check

For three points A, B, C, they are collinear if:

$$(x_C - x_B)(y_B - y_A) = (y_C - y_B)(x_B - x_A)$$

7.4.2 Algorithm for Merging Collinear Edges

1. For each vertex connected to more than one edge:
   (a) For each pair of edges connected to the vertex:
      i. Check if they are collinear.
      ii. If they are, merge them into a single edge.
      iii. Remove the original edges from the list.
      iv. Add the new edge to the list.

7.5 Generating Probable 3D Vertices

The system generates probable 3D vertices by matching vertices across the three views.

7.5.1 Algorithm for Constructing Probable 3D Vertices

1. For each vertex in the Front View:
   (a) For each vertex in the Top View with the same x coordinate:
      i. For each vertex in the Side View with the same y coordinate (from Front View):
         A. If z coordinates from Top and Side views match, create a probable 3D vertex (x, y, z).

7.6 Generating Probable 3D Edges

Edges are generated by connecting probable vertices if their projections correspond to edges in the 2D views.

7.6.1 Algorithm for Constructing Probable 3D Edges

1. For each pair of probable vertices:
   (a) Project the edge between them onto each 2D plane.
   (b) Check if the projected edge exists in the corresponding 2D view.
   (c) If the edge appears in at least two views and coincides in the third (if not appearing), consider it a probable edge.

7.6.2 Edge Containment Check

Ensure that new edges do not contain existing edges to avoid redundancy.

7.7 Validating Vertices and Edges

7.7.1 Connectivity Check

Each vertex must be connected to at least three edges.

7.7.2 Algorithm for Validating Vertices

1. Build an adjacency list of vertices and their connected edges.
2. For each vertex:
   (a) If it is connected to fewer than three edges, mark it as invalid.
   (b) Remove the vertex and associated edges from the lists.
3. Repeat until no invalid vertices remain.

7.8 Generating Probable Faces

Faces are generated by traversing the edges and forming closed loops.

7.8.1 Minimum Internal Angle Algorithm

1. For each vertex v_i:
   (a) Find all edges connected to v_i.
   (b) For each pair of edges e_i and e_j:
      i. Compute the internal angle between them.
      ii. Select the edge with the minimum internal angle.
   (c) Proceed to the next vertex connected by the selected edge.
   (d) Continue until the starting vertex v_i is reached, forming a face.

7.8.2 Planarity Check

Ensure that the edges form a planar face by checking the cross product of edge vectors.

$$\mathbf{n} = (\mathbf{e}_i \times \mathbf{e}_j)$$

All normals $\mathbf{n}$ should be consistent for the face to be planar.

7.9 Removing Pseudo Elements

Pseudo elements are invalid edges and faces that do not correspond to the actual 3D object.

7.9.1 Decision Rules

1. **Rule 1:** An edge adjacent to more than two faces can have at most two true faces.
2. **Rule 4:** An edge adjacent to only one face is a false edge; the face is also false.
3. **Rule 5:** An edge adjacent to two coplanar faces is a false edge; the faces can be merged.
4. **Rule 6:** An edge adjacent to two non-coplanar faces is a true edge; both faces are true.

7.9.2 Decision Chaining Algorithm

1. Initialize all edges and faces as undecided.
2. While there are undecided elements:
   (a) Apply decision rules to edges and faces.
   (b) If changes are made, repeat the process.
   (c) If contradictions occur, backtrack and try alternative assumptions.
3. Remove all false edges and faces.

7.9.3 Moëbius Rule Verification

Ensure that:

- Each true edge is adjacent to exactly two true faces.
- No two true faces intersect except at shared edges.

8 Data Structures

8.1 Classes and Their Relationships

- **Vertex3D**: Represents a point in 3D space.
- **Edge3D**: Represents an edge between two vertices in 3D space.
- **Face3D**: Represents a face bounded by edges in 3D space.
- **Graph2D**: Represents a 2D projection with vertices and edges.
- **WireframeModel**: Contains methods to generate and validate the 3D model.

8.2 Justification of Data Structures

- **Vectors**: Used to store lists of vertices, edges, and faces due to their dynamic nature.
- **Maps and Sets**: Used for quick lookups and to ensure uniqueness.
- **Classes**: Encapsulate data and related methods, promoting modularity.

9 Input/Output Formats

9.1 Input Format

The system accepts a text file with the following structure:

- **View Name**: Either "Top_View", "Front_View", or "Side_View".
- **Number of Vertices**: An integer.
- **Vertices**: Lines containing x and y coordinates.
- **Number of Edges**: An integer.
- **Edges**: Lines containing start index, end index, and line type ("solid" or "dashed").

9.2 Output Formats

- **Console Output**: Information about the generated 3D model.
- **Output File**: Contains the list of vertices, edges, and faces of the 3D model.

10 Project Structure

10.1 Directory and File Overview

- **src/**: Contains all the source code files.
  - **main.cpp**: Entry point of the application.
  - **graph2d.h/.cpp**: Definition and implementation of the `Graph2D` class.
  - **vertex3d.h**: Definition of the `Vertex3D` class.
  - **edge3d.h**: Definition of the `Edge3D` class.
  - **face3d.h**: Definition of the `Face3D` class.
  - **wireframe_model.h/.cpp**: Methods for generating and validating the 3D model.
  - **graph2d_utils.h/.cpp**: Functions for reading and processing 2D graphs.
- **build/**: Build directory containing compiled binaries and output files.
- **output/**: Directory where output files are saved.
- **include/**: Directory containing header files.
- **lib/**: Directory for external libraries.
- **CMakeLists.txt**: Build configuration file for CMake.

11 Algorithms and Formulas

11.1 Algorithm for 2D Vertices and 2D Lines

11.1.1 Overview

The algorithm preprocesses input data to remove redundancies and prepare for 3D reconstruction. It handles duplicate vertices, intersection points, and collinear lines.

11.1.2 Step 1: Input Data Processing

Intersection of Line Segments Refer to Section 2.3 for detailed equations.

11.1.3 Step 2: Handling Collinear Lines

Refer to Section 2.4 for detailed algorithms.

11.2 Constructing Probable 3D Vertices

Refer to Section 2.5 for detailed algorithms.

11.3 Step 3: Generation and Validation of Probable 3D Edges

Refer to Sections 2.6 and 2.7 for detailed algorithms.

11.4 Getting Probable Faces

Refer to Section 2.8 for the Minimum Internal Angle Algorithm.

11.5 Removing Pseudo Elements

Refer to Section 2.9 for the Decision Rules and Decision Chaining Algorithm.

11.6 Algorithm for Collinearity Check

Given two connected edges $\overline{AB}$ and $\overline{BC}$:

$$\frac{(x_C - x_B)}{(x_B - x_A)} = \frac{(y_C - y_B)}{(y_B - y_A)}$$

If true, the edges are collinear.

11.7 Minimum Internal Angle Algorithm Details

Internal Angle Calculation Given vectors $\mathbf{e}_1$ and $\mathbf{e}_2$:

$$\theta = \cos^{-1}\left(\frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_1\|\|\mathbf{e}_2\|}\right)$$

Cross Product for Planarity

$$\mathbf{n} = \frac{\mathbf{e}_i \times \mathbf{e}_j}{\|\mathbf{e}_i \times \mathbf{e}_j\|}$$

Consistent normals indicate a planar face.

12 Functionality Limitations

- **Assumption of Planar Faces**: The system assumes that all faces are planar.
- **Input Format Restriction**: Only the specified input format is accepted.
- **Complex Objects**: May not handle highly complex objects efficiently.
- **Visualization****: No graphical user interface; outputs are text-based.
- **Error Handling****: Limited error handling for invalid inputs.