Assignment 4: Design, Implementation and Testing of Engineering Drawing

Ziv Barretto

October 2024

1 Detailed Algorithms

1.1 Overview

The system implements several key algorithms to achieve the conversion between 2D projections and 3D models, as well as to render these models. The main algorithms are:

1. **Edge Intersection and Segmentation**
2. **Point-in-Polygon Test (Ray Casting Algorithm)**
3. **Hidden Line Removal**
4. **Wireframe Reconstruction from 2D Views**
5. **Surface Formation (Face Generation)**
6. **3D to 2D Projection**
7. **Rendering Techniques**

We will explore each algorithm in detail, providing pseudocode and explanations where necessary.

1.2 Edge Intersection and Segmentation

Purpose: When reconstructing a 3D model from 2D projections, edges in the 2D views may intersect. Identifying and handling these intersections is crucial to ensure the consistency and accuracy of the 3D model.

Algorithm Details:

- **Edge Intersection Detection:** For each pair of edges in a 2D view, check if they intersect.
- **Edge Segmentation:** If two edges intersect, split the edges at the intersection point, creating new edges and vertices.

Pseudocode:

```
for each edge e1 in view:
    for each edge e2 in view where e2 != e1:
        if e1 and e2 intersect:
            intersection_point = calculate_intersection(e1, e2)
            if intersection_point exists:
                // Split edges at the intersection point
                split e1 into e1a and e1b at intersection_point
                split e2 into e2a and e2b at intersection_point
                // Add new vertices and edges to the model
                add intersection_point to vertices
                replace e1 with e1a and e1b in edges
                replace e2 with e2a and e2b in edges
```

Implementation Highlights:

- Uses the **Parametric Form of Line Equations** to find the intersection point.
- Checks if the intersection point lies within both line segments.
- Updates the edge list to include the new segmented edges.

1.3 Point-in-Polygon Test (Ray Casting Algorithm)

Purpose: To determine whether a point lies inside a polygon (surface boundary) in a 2D view. This is crucial for identifying hidden edges.

Algorithm Details:

- Cast a horizontal ray from the point to infinity.
- Count how many times the ray intersects with the edges of the polygon.
- If the count is odd, the point is inside; if even, the point is outside.

Pseudocode:

```
function is_inside(point P, polygon V):
    count = 0
    for each edge (Vi, Vj) in polygon V:
        if ray from P intersects edge (Vi, Vj):
            if orientation(Vi, P, Vj) == 0:
                return on_segment(Vi, P, Vj)
            count = count + 1
    return (count % 2) == 1
```

Implementation Highlights:

- **Orientation Function:** Determines the orientation (clockwise, counterclockwise, colinear) of ordered triplets.
- **On-Segment Function:** Checks if a point lies on a line segment.
- Efficiently handles edge cases where the point lies exactly on an edge.

1.4 Hidden Line Removal

Purpose: To identify and mark edges that should be hidden in the 2D projections due to being obscured by surfaces.

Algorithm Details:

- For each edge, calculate its midpoint.
- For each surface, use the plane equation to determine if the midpoint is behind the surface.
- Use the point-in-polygon test to check if the midpoint lies within the projection of the surface.
- If both conditions are met, mark the edge as hidden.

Pseudocode:

```
for each edge E in view:
    midpoint = calculate_midpoint(E)
    for each surface S:
        if plane_equation(S, midpoint) * plane_equation(S, reference_point) < 0:
            if is_inside(midpoint, projection_of(S)):
                mark E as hidden
                break
```

Implementation Highlights:

- **Plane Equation Evaluation:** Uses the coefficients a, b, c, d of the surface plane equation $ax + by + cz + d = 0$.
- **Reference Point:** A point known to be outside the surface, used to determine the side of the plane.
- Efficiently iterates over edges and surfaces to minimize computations.

1.5 Wireframe Reconstruction from 2D Views

Purpose: To reconstruct the 3D coordinates of vertices by combining information from multiple 2D projections.

Algorithm Details:

- Align vertices from different views based on their vertex numbers.
- Combine the coordinates from different views to compute the full 3D coordinates.

Pseudocode:

```
for each vertex number vNo in top_view:
    x = top_view.vertices[vNo].x
    y = top_view.vertices[vNo].y
    z_front = front_view.vertices[vNo].z
    z_side = side_view.vertices[vNo].z

    // Ensure consistency in z-coordinate between front and side views
    if abs(z_front - z_side) < EPSILON:
        z = z_front
    else:
        report inconsistency

    create Vertex(x, y, z, vNo)
    add Vertex to 3D model
```

Implementation Highlights:

- Checks for consistency in coordinates across views.
- Reports inconsistencies to help identify errors in the input data.
- Uses a small epsilon value to account for floating-point inaccuracies.

1.6 Surface Formation (Face Generation)

Purpose: To create surfaces (faces) of the 3D model by identifying closed loops of edges.

Algorithm Details:

- For each vertex, find neighboring vertices.
- Use depth-first search or similar traversal to find cycles that form surfaces.
- Calculate the plane equation for each surface.

Pseudocode:

```
for each vertex V in model:
    neighbors = get_neighbors(V)
    for each pair of neighbors (N1, N2):
        if edges (V, N1) and (V, N2) are connected to form a face:
            surface = create_surface([V, N1, N2, ...])
            calculate_plane_coefficients(surface)
            add surface to model
```

Implementation Highlights:

- **Edge Connectivity:** Checks if edges form a loop without overlaps.
- **Plane Coefficients Calculation:** Uses three non-colinear points to define the plane of the surface.
- Handles complex models by iteratively building up surfaces.

1.7 3D to 2D Projection

Purpose: To generate 2D projections (views) of a 3D model by projecting the vertices onto standard planes.

Algorithm Details:

- For each vertex in the 3D model, project it onto the desired plane by ignoring one coordinate.
- For example, for the top view, ignore the z-coordinate.

Pseudocode:

```
function project_model(model, direction):
    projection = Model_2D(direction)
    for each vertex V in model.vertices:
        if direction == TOP_VIEW:
            projected_vertex = Vertex(V.x, V.y, 0, V.vNo)
        elif direction == FRONT_VIEW:
            projected_vertex = Vertex(V.x, 0, V.z, V.vNo)
        elif direction == SIDE_VIEW:
            projected_vertex = Vertex(0, V.y, V.z, V.vNo)
        projection.add_vertex(projected_vertex)

    // Edges are projected similarly
    for each edge E in model.edges:
        projected_edge = Edge(projection.vertices[E.a.vNo], projection.vertices[E.b.vNo], E.eno)
        projection.add_edge(projected_edge)

    return projection
```

Implementation Highlights:

- Maintains vertex numbers to ensure consistency between 3D and 2D representations.
- Edges are projected by projecting their vertices.

1.8 Rendering Techniques

Purpose: To visually display the 2D views and 3D models, including interactive manipulation of 3D models.

Algorithm Details:

- **2D Rendering:**
  - Set up an orthographic projection.
  - Scale the model to fit within the viewport.
  - Draw edges, using dashed lines for hidden edges.
- **3D Rendering:**
  - Set up a perspective projection.
  - Implement camera controls for rotation and zooming.
  - Draw surfaces and edges.

Implementation Highlights:

- Uses OpenGL for rendering.
- Handles user input for interactive controls.
- Efficiently renders large models by minimizing state changes and draw calls.

2 Data Structures

2.1 Overview

The system uses object-oriented design, encapsulating data and related operations within classes. The primary data structures are:

- **Vertex Class**
- **Edge Class**
- **Surface Class**
- **Model_2D and Model_3D Classes**

These classes use standard containers like `std::vector`, `std::map`, and `std::set` for efficient storage and retrieval.

2.2 Vertex Class

Definition:

```
class Vertex {
public:
    float x, y, z;
    int vNo;
    bool isTrue;

    // Constructors and methods
};
```

Justification:

- **Encapsulation:** Groups coordinate data and related methods.
- **Efficiency:** Simple data members allow for fast access and copying.
- **Consistency:** The `vNo` attribute ensures that vertices can be matched across different views.

2.3 Edge Class

Definition:

```
class Edge {
public:
    Vertex a, b;
    bool hidden;
    bool isTrue;
    int eno;

    // Constructors and methods
};
```

Justification:

- **Encapsulation:** Combines two vertices to define an edge.
- **Efficiency:** Storing vertices directly allows for quick access to coordinates.
- **Ease of Implementation:** The `hidden` flag is used directly during rendering.

2.4 Surface Class

Definition:

```
class Surface {
public:
    std::vector<Edge> boundary;
    float coeff[4];
    bool trueSurface;
    int sno;

    // Constructors and methods
};
```

Justification:

- **Flexibility:** The boundary can be any number of edges, allowing for complex surfaces.
- **Efficiency:** Coefficients are stored in an array for fast computation of plane equations.
- **Ease of Implementation:** Surfaces are built from edges, which are already defined.

2.5 Model_2D and Model_3D Classes

Definition:

```
class Model_2D {
public:
    int direction;
    std::vector<Vertex> v;
    std::vector<Edge> e;
    std::vector<Surface> s;

    // Constructors and methods
};

class Model_3D {
public:
    std::vector<Vertex> v;
    std::vector<Edge> e;
    std::vector<Surface> s;
    Model_2D topView, frontView, sideView;

    // Constructors and methods
};
```

Justification:

- **Organization:** Separates 2D and 3D data, while allowing for easy conversion between them.
- **Efficiency:** Using `std::vector` allows for dynamic resizing and efficient memory usage.
- **Ease of Implementation:** Standard containers provide built-in functionality for common operations.

2.6 Standard Containers

- `std::vector`**:** Used for dynamic arrays where elements are accessed by index.
  - **Justification:** Provides efficient random access and dynamic resizing.
- `std::map`**:** Used for mapping keys to values, such as vertex numbers to vertices.

  - **Justification:** Ensures fast lookup, which is essential when matching vertices across views.

- `std::set`: Used for storing unique elements, such as neighbor vertex numbers.
  - **Justification:** Automatically handles uniqueness, preventing duplicates.

3 Input/Output Formats

3.1 Overview

The system uses custom text-based file formats for both input and output. This allows for simplicity and ease of parsing, without the overhead of more complex formats like OBJ or DXF.

3.2 Input Formats

The input files can contain either 2D projection data or 3D model data.

3.2.1 2D Input Format

Structure:

```
2
[Direction]
[Number of Vertices]
[Vertex Data]
[Number of Edges]
[Edge Data]
[Direction]
[Number of Vertices]
[Vertex Data]
[Number of Edges]
[Edge Data]
[Direction]
[Number of Vertices]
[Vertex Data]
[Number of Edges]
[Edge Data]
```

Details:

- **First Line:** The number `2` indicates that the file contains 2D data.
- **Direction:** An integer indicating the view direction (`0`: Top, `1`: Front, `2`: Side).
- **Number of Vertices:** An integer specifying how many vertices are defined in the view.
- **Vertex Data:** Each vertex is defined on a new line as:

  ```
  [vNo] [coord1] [coord2]
  ```

  - `vNo`: Vertex number (integer).
  - `coord1`, `coord2`: Coordinates in the 2D plane (floats).
- **Number of Edges:** An integer specifying how many edges are defined in the view.
- **Edge Data:** Each edge is defined on a new line as:

  ```
  [eno] [vNoA] [vNoB] [hidden]
  ```

  - `eno`: Edge number (integer).

- `vNoA`, `vNoB`: Vertex numbers defining the edge.
- `hidden`: Flag indicating if the edge is hidden (`1`) or visible (`0`).

Example:

```
2
0
4
1 0.0 0.0
2 1.0 0.0
3 1.0 1.0
4 0.0 1.0
4
1 1 2 0
2 2 3 0
3 3 4 0
4 4 1 0
...
```

3.2.2 3D Input Format

Structure:

```
3
[Number of Vertices]
[Vertex Data]
[Number of Edges]
[Edge Data]
[Number of Surfaces]
[Surface Data]
```

Details:

- **First Line:** The number `3` indicates that the file contains 3D data.
- **Number of Vertices:** An integer specifying how many vertices are in the model.
- **Vertex Data:** Each vertex is defined on a new line as:

  ```
  [vNo] [x] [y] [z]
  ```

- **Number of Edges:** An integer specifying how many edges are in the model.
- **Edge Data:** Each edge is defined on a new line as:

  ```
  [eno] [vNoA] [vNoB]
  ```

- **Number of Surfaces:** An integer specifying how many surfaces are in the model.
- **Surface Data:** Each surface is defined as:

  ```
  [sno] [Number of Boundary Edges] [eno1] [eno2] ... [enoN]
  ```

Example:

```
3
4
1 0.0 0.0 0.0
2 1.0 0.0 0.0
3 1.0 1.0 0.0
4 0.0 1.0 0.0
4
1 1 2
2 2 3
3 3 4
4 4 1
1
1 4 1 2 3 4
```

3.3 Output Formats

The output files are in the same format as the input files but represent the transformed data (e.g., reconstructed 3D model from 2D input).

3.3.1 Output for 2D Input (Resulting 3D Model)

- Follows the **3D Input Format** as described above.
- Contains the reconstructed 3D vertices, edges, and surfaces.

What Users Can Expect:

- A complete 3D model representation that can be visualized or further processed.
- Vertices with calculated x, y, z coordinates.
- Edges and surfaces that define the shape of the 3D model.

3.3.2 Output for 3D Input (Resulting 2D Projections)

- Follows the **2D Input Format** but includes the generated 2D views.
- Contains the projected vertices and edges for top, front, and side views.

What Users Can Expect:

- Accurate 2D projections that represent the 3D model from different perspectives.
- Hidden lines appropriately marked based on visibility.

3.4 Examples

3.4.1 Example Input (2D Data)

```
2
0
4
1 0.0 0.0
2 2.0 0.0
3 2.0 2.0
4 0.0 2.0
4
1 1 2 0
2 2 3 0
3 3 4 0
4 4 1 0
1
```

```
4
1 0.0 0.0
2 2.0 0.0
3 2.0 2.0
4 0.0 2.0
...
```

3.4.2 Corresponding Output (Reconstructed 3D Model)

```
3
4
1 0.0 0.0 0.0
2 2.0 0.0 0.0
3 2.0 0.0 2.0
4 0.0 0.0 2.0
...
```

3.5 File Parsing and Validation

- The system includes error checking to ensure that the input files are correctly formatted.
- Inconsistencies or errors in the input data are reported to the user.

4 Known Limitations

4.1 Unsupported/Incomplete Functionality

- **Transformation Operations:** The system currently does not support operations like rotation, translation, or scaling of the models. Users cannot manipulate the models to change their orientation or position.
- **Object Segmentation:** The ability to segment (cut) the object into parts is not implemented. Users cannot divide models into smaller sections within the system.
- **Complex Surfaces:** While basic surface formation is supported, extremely complex surfaces or non-manifold geometries may not be correctly processed.

4.2 Model Reusability

- **Input-Output Compatibility:** One of the strengths of the system is that the output files it generates can be directly used as input files. This allows for iterative processing and testing without the need to convert between different file formats.
- **Bidirectional Usage:** Users can input 2D projections, obtain a 3D model, and then use that 3D model as input to generate 2D projections again, facilitating a two-way workflow.

5 Summary

- **Algorithms:** The system implements essential algorithms for geometric computations, including edge intersection detection, point-in-polygon tests, hidden line removal, and projection transformations.
- **Data Structures:** Uses classes and standard containers to efficiently store and manipulate geometric data.
- **Input/Output Formats:** Utilizes simple, custom text-based formats for ease of use and parsing, providing flexibility in representing both 2D and 3D data.
- **Limitations:** The system has constraints regarding input size and lacks certain functionalities like object transformation and segmentation, but offers the advantage of reusing output files as inputs.

By understanding these components and limitations, users can effectively interact with the system, provide the necessary inputs, and interpret the outputs for their applications.