**UNIT-II: Raster Graphics Algorithms and Transformations

1. Raster Graphics Algorithms

Raster graphics algorithms are responsible for converting geometric shapes into a pixel-based representation on a raster grid (like a screen).

1.1 Line Drawing Algorithms

Line drawing algorithms are used to approximate straight lines between two points on a raster grid. These algorithms are essential for drawing lines efficiently.

1.1.1 Digital Differential Analyzer (DDA) Algorithm

- **Description**:
  The DDA algorithm is an incremental scan conversion algorithm. It calculates intermediate points of the line between two endpoints by incrementing the values of and at each step.
- **Working Principle**:
  - Given two points and , the algorithm computes the differences and .
  - The step count is determined by the maximum of the absolute values of and .
  - The values of and are updated incrementally to plot intermediate points along the line.
- **Formula**:

```
Δx = x1 - x0
Δy = y1 - y0
steps = max(|Δx|, |Δy|)
x_inc = Δx / steps
y_inc = Δy / steps
```

- **Advantages**:
  - Simple to implement.
  - Works well for small lines and basic raster systems.
- **Disadvantages**:
  - Uses floating-point arithmetic, which is computationally expensive.

1.1.2 Bresenham's Line Algorithm

- **Description**:
  Bresenham's algorithm improves upon DDA by using integer-only calculations, making it faster and more efficient for raster displays.
- **Working Principle**:
  - The decision variable determines whether the next point is on the line or offset.
  - The algorithm steps through the grid, adjusting based on the comparison between the decision variable and 0.
- **Formula**:

  ```
  p = 2 * Δy - Δx
  p_next = p + 2 * Δy (if p < 0)
           p + 2 * Δy - 2 * Δx (otherwise)
  ```

- **Advantages**:
  - Efficient integer-based computations.
  - Works well for lines of any slope, including steep lines.
- **Disadvantages**:
  - More complex than DDA but computationally faster.

1.2 Circle and Ellipse Drawing Algorithms

Circle and ellipse drawing algorithms are used to generate round shapes on a raster display. These algorithms take advantage of symmetry to reduce the number of calculations required.

1.2.1 Midpoint Circle Algorithm

- **Description**:
  The Midpoint Circle Algorithm uses the midpoint principle to determine the closest pixel for each step along the circle. It benefits from the symmetry of a circle, allowing all eight octants to be computed at once.
- **Working Principle**:

  - The algorithm starts at the top of the circle and uses integer calculations to plot points in each of the eight octants.

- **Formula**:

```
p = 1 - r
p_next = p + 2x + 3 (if p < 0)
         p + 2x - 2y + 5 (otherwise)
```

- **Advantages**:
  - Efficient use of integer arithmetic.
  - Uses symmetry to reduce the number of operations.

- **Disadvantages**:
  - Limited to circle drawing only.

1.2.2 Midpoint Ellipse Algorithm

- **Description**:
  The Midpoint Ellipse Algorithm is an extension of the Midpoint Circle Algorithm to draw ellipses. The ellipse is treated as two separate curves, with different decision parameters for each region.

- **Working Principle**:
  - The ellipse is divided into two regions: one where the slope is less than 1 and the other where the slope is greater than 1. Different formulas are used for each region to efficiently calculate the next point.

- **Formula (Region 1)**:

```
p1 = b² - a²b + 0.25a²
```

- **Formula (Region 2)**:

```
p2 = b²(x + 0.5)² + a²(y - 1)² - a²b²
```

- **Advantages**:

- Efficient drawing of ellipses.
- Integer-based calculations for better performance.

- **Disadvantages**:
  - Requires two separate regions for calculation.

1.3 Filling Algorithms

Filling algorithms are used to fill enclosed areas like polygons and shapes with a specific color or pattern.

1.3.1 Scan-Conversion Polygon Filling

- **Description**:
  This method scans each horizontal line (scan line) and computes where it intersects with the edges of the polygon. The intersections are then filled between pairs of points.
- **Working Principle**:
  - For each scan line, calculate the intersection points with the polygon edges.
  - Sort these intersection points and fill pixels between them.
- **Advantages**:
  - Efficient for filling convex and concave polygons.
- **Disadvantages**:
  - Requires sorting of intersection points, which can be computationally expensive.

1.3.2 Inside-Outside Test

- **Description**:
  The Inside-Outside Test determines whether a point is inside or outside a polygon. It is commonly used in flood fill algorithms to identify whether a pixel should be filled.
- **Working Principle**:
  - A ray is cast from the point in question, and the number of intersections with polygon edges is counted.

- If the number of intersections is odd, the point is inside; if even, the point is outside.

1.3.3 Boundary Fill Algorithm

- **Description**:
  Boundary fill works by replacing the color of the seed point and recursively filling adjacent pixels until the boundary color is encountered.
- **Working Principle**:
  - Start from a point inside the polygon, replace its color, and recursively fill the adjacent pixels that are not part of the boundary.
- **Advantages**:
  - Simple implementation.
- **Disadvantages**:
  - Not suitable for large areas as it may lead to stack overflow due to recursion.

1.3.4 Flood Fill Algorithm

- **Description**:
  Flood fill is a non-recursive algorithm used to fill all pixels connected to a given seed point, until the boundary is encountered. It is similar to boundary fill but doesn't consider boundary colors.
- **Working Principle**:
  - Starts from a seed point, checks for connected pixels, and fills them with a target color.
- **Advantages**:
  - More efficient than recursive boundary fill.
- **Disadvantages**:
  - Can still be computationally expensive for complex shapes.

---

2. Transformations

Transformations in computer graphics are mathematical operations that alter the position, size, or orientation of objects. They are essential for rendering objects in different perspectives and for animating scenes.

2.1 2D Transformations

2D transformations include translation, rotation, scaling, reflection, and shearing. These transformations manipulate objects in a two-dimensional plane.

Translation

- **Description**:
  Translation shifts an object from one location to another by adding a constant value to the coordinates of each point.
- **Formula**:

  ```
  x' = x + t_x
  y' = y + t_y
  ```

- **Where**:
  - and are the translation distances along the - and -axes.

Rotation

- **Description**:
  Rotation turns an object around a point (usually the origin) by a specified angle.
- **Formula**:

  ```
  x' = x * cos(θ) - y * sin(θ)
  y' = x * sin(θ) + y * cos(θ)
  ```

- **Where**:
  - is the angle of rotation.

Reflection

- **Description**:
  Reflection produces a mirror image of an object across a specified axis (e.g., -axis, -axis, or any arbitrary axis).
- **Formula**:
  - Reflection across -axis:

    ```
    x' = x
    y' = -y
    ```

  - Reflection across -axis:

    ```
    x' = -x
    y' = y
    ```

Scaling

- **Description**:
  Scaling changes the size of an object by scaling its coordinates along the - and -axes.
- **Formula**:

  ```
  x' = x * s_x
  y' = y * s_y
  ```

- **Where**:
  - and are the scaling factors along the - and -axes.

Shearing

- **Description**:
  Shearing distorts an object in such a way that it becomes slanted, either horizontally or vertically.
- **Formula**:

```
x' = x + y * sh_x
y' = y + x * sh_y
```

- **Where**:
  - and are the shearing factors along the - and -axes.

2.2 Homogeneous Coordinate Representation

- **Description**:
  Homogeneous coordinates add a third coordinate () to the 2D point, allowing transformations like translation to be represented as matrix multiplications.

- **Formula**: For translation:

```
[x']   [1  0  t_x] [x]
[y'] = [0  1  t_y] [y]
[1 ]   [0  0   1 ] [1]
```

---

3. 3D Transformations

3D transformations manipulate objects in a three-dimensional space. They include translation, rotation, scaling, and reflection, extending the concepts of 2D transformations into the third dimension.

3.1 3D Translation

- **Description**:
  Similar to 2D translation, 3D translation moves an object in 3D space by adjusting its coordinates along the -, -, and -axes.

- **Formula**:

```
x' = x + t_x
y' = y + t_y
z' = z + t_z
```

- **Where**:

  - are the translation distances along the -, -, and -axes, respectively.

- **Advantages**:
  - Simple to apply.
  - Essential for moving objects within a 3D scene.

3.2 3D Rotation

- **Description**:
  3D rotation rotates an object around one of the axes (x-axis, y-axis, or z-axis). In 3D graphics, rotation is more complex than in 2D, as the object can be rotated around multiple axes.

- **Formula**:
  - **Rotation about the -axis**:

    ```
    x' = x
    y' = y * cos(θ) - z * sin(θ)
    z' = y * sin(θ) + z * cos(θ)
    ```

  - **Rotation about the -axis**:

    ```
    x' = x * cos(θ) + z * sin(θ)
    y' = y
    z' = -x * sin(θ) + z * cos(θ)
    ```

  - **Rotation about the -axis**:

    ```
    x' = x * cos(θ) - y * sin(θ)
    y' = x * sin(θ) + y * cos(θ)
    z' = z
    ```

- **Where**:
  - is the angle of rotation, typically in degrees or radians.

- **Advantages**:
  - Allows manipulation of objects in 3D space by rotating them around any axis.
  - Essential for 3D rendering and animation.

3.3 3D Scaling

- **Description**:
  3D scaling alters the size of an object by changing its coordinates along the -, -, and -axes. It is used to increase or decrease the size of 3D models.
- **Formula**:

  ```
  x' = x * s_x
  y' = y * s_y
  z' = z * s_z
  ```

- **Where**:
  - are the scaling factors along the -, -, and -axes, respectively.
- **Advantages**:
  - Scaling is fundamental for creating objects of varying sizes.
  - It can be non-uniform (different scaling factors for each axis) or uniform (same scaling factor for all axes).

3.4 3D Reflection

- **Description**:
  Reflection in 3D graphics involves flipping an object across one of the coordinate planes or a custom plane. It produces a mirror image of the object.
- **Formula**:
  - **Reflection about the -axis**:

    ```
    x' = x
    y' = -y
    z' = -z
    ```

  - **Reflection about the -axis**:

    ```
    x' = -x
    y' = y
    z' = -z
    ```

- **Reflection about the -axis**:

  ```
  x' = -x
  y' = -y
  z' = z
  ```

- **Advantages**:
  - Used in creating mirror images, symmetry in 3D models, and simulating reflections in graphics.

3.5 3D Shearing

- **Description**:
  Shearing in 3D operates similarly to 2D shearing but introduces distortion along multiple axes. It skews the object by changing its shape while maintaining the relative position of its vertices.
- **Formula**:

  ```
  x' = x + y * sh_x + z * sh_xz
  y' = y + x * sh_yx + z * sh_yz
  z' = z + x * sh_zx + y * sh_zy
  ```

- **Where**:
  - are the shearing factors along the respective axes.
  - represent shear factors involving multiple axes.
- **Advantages**:
  - Useful for creating distorted perspectives or effects like slanted objects or skewed shapes.

---

4. Homogeneous Coordinate Representation in 3D

Homogeneous coordinates extend the concept of homogeneous transformations to 3D space. They simplify the matrix operations needed to perform translations and other transformations, allowing all transformations (including translation) to be represented as matrix multiplications.

4.1 Matrix Representation

- **3D Translation** in homogeneous coordinates:

```
[x']    [1   0   0   t_x] [x]
[y'] = [0   1   0   t_y] [y]
[z']    [0   0   1   t_z] [z]
[1 ]    [0   0   0    1 ] [1]
```

- **3D Rotation** in homogeneous coordinates: For rotation about the -axis:

```
[x']    [1    0       0      0] [x]
[y'] = [0   cos(θ) -sin(θ)   0] [y]
[z']    [0   sin(θ)  cos(θ)   0] [z]
[1 ]    [0    0       0      1] [1]
```

  For rotation about the -axis:

```
[x']    [cos(θ)  0   sin(θ)   0] [x]
[y'] = [0       1    0       0] [y]
[z']    [-sin(θ) 0   cos(θ)   0] [z]
[1 ]    [0       0    0       1] [1]
```

  For rotation about the -axis:

```
[x']    [cos(θ) -sin(θ)  0   0] [x]
[y'] = [sin(θ)  cos(θ)  0   0] [y]
[z']    [0       0       1   0] [z]
[1 ]    [0       0       0   1] [1]
```

---

5. Composite Transformations

- **Description**:
  Composite transformations combine multiple transformations (e.g., translation, rotation, scaling) into one transformation. This is done by multiplying the corresponding transformation matrices.

- **Formula**: For a combination of scaling (), rotation (), and translation ():

```
M = T * R * S
```

- **Advantages**:
    - Saves computational resources by applying a single transformation matrix instead of multiple operations.
    - Simplifies the manipulation of complex objects.