Chapter - 7 (10 Marks)

b

9. Differentiate between Image space and object space method. How is back face detection method used for visible surface detection? Explain in detail. How A buffer method eliminate the drawbacks of Z buffer method? [2+5+3]

9. How much memory is requires to implement z-buffer algorithms for a 512×512×24 bit-plane image? Explain how z-buffer algorithms determine the visibility of polygon surface along with necessary derivations algorithms limitation. [2+8]

b

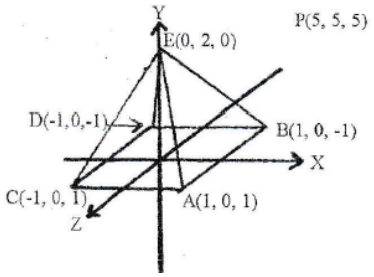
7. What is the limitation of Z- buffer method? How does A-buffer method overcome it? Explain. [4+4] [2+6]

7. Compare object space method and image space method. Explain depth buffer method in detail. Compare it with A-buffer method. [3+2+4] [2+5+3]

7. Describe Z-Buffer method of visible surface detection. Compare this method to other methods of visible surface detection. [6+2]

7. What is the limitation of Z-buffer method? How does A-buffer method overcome it, explain? [3+7]

7. Explain Back-face detection algorithm for visible surface detection. Find the visibility for the surface BED and ABCD where observer is at P(5, 5, 5). [3+5]



b

7. How hidden surfaces can be removed? Explain in detail about depth buffer methods. [8]

7. Explain backface detection algorithm. Determine whether two surfaces of a object with normals $2\vec{i}-3\vec{j}+4\vec{k}$ and $\vec{i}+\vec{j}-2\vec{k}$ respectively, viewed from a direction given by $\vec{i}-\vec{j}+\vec{k}$ are backface or frontface. [5+5]

b

8. What is the difference between object space method and image space method for visible surface determination? Describe scan line method to find visible lines with example. [4+8]

7. What are the consideration factors to choose the Visible Surface Detection Algorithm? What are the two classes of visible surface detection techniques, explain? What is limitation of Z-Buffer method? How does A-Buffer method overcome it, Explain? [12]

b

7. Differentiate image space and object space method for visible surface determination. Explain scanline method to determine visible surface of object. [8+4]

b

8. Compare object space method with image space method. Explain, How Back-face detection method is used to detect visible surface. Also explain z-Buffer method. [2+4+4]

7. Outline the Z buffer algorithm. List the advantages and disadvantages of the z-buffer algorithm. [6+2+2]

b

6. Compare Z-buffer and A-Buffer algorithm. Also write algorithm to find visible surfaces using scan-line method. [10]

7. Explain z-buffer algorithm along with necessary steps needed to calculate the depth. What is its drawback? [10]

b

7. Describe scan line method to find visible lines with example. [10]

7. **Explain z-buffer algorithm along with necessary steps needed to calculate the depth. What is its drawback?** [10]

7. **Describe z-buffer method for visible surface detection in detail. State its limitation and recommended method that addresses it.** [7+3]

8. **Differentiate between image space and object space methods of visible surface detection. Describe A-Buffer method of visible surface detection.** [4+6]

6. What are the object space and image space method of hidden surface removal? Describe back face detection method of hidden surface removal. [4+6]

6. What are object space and image space method of hidden surface removal? Describe one of the image space methods of hidden surface removal. [4+6]

Visible surface determination, also known as hidden surface elimination, is a fundamental concept in computer graphics that deals with determining which surfaces and parts of objects are visible from a particular viewpoint. When rendering a 3D scene, not all surfaces of objects are visible to the viewer. Some surfaces are obscured by other objects or parts of objects. The purpose of visible surface determination is to compute and render only the parts of objects that should be visible from a given viewpoint, thus optimizing rendering performance and producing a correct visual output.

Visible surface determination techniques are generally categorized into two groups based on the domain in which they operate: object space methods and image space methods.

1. Object Space (3D coordinates system)

Object space refers to the coordinate system in which objects are defined. It is also known as **model space**. In this space, operations are performed directly on the geometric descriptions of the objects. These descriptions include the vertices, edges, and faces that make up the object's shape.

Characteristics of Object Space Methods:

- **Coordinate System**: Object space is the local coordinate system attached to the object itself. All transformations (like rotation, scaling, and translation) are applied to the object's coordinates.
- **Precision**: Since operations are performed directly on the objects' geometric data, object space methods tend to be more precise in terms of numerical accuracy.

- **Operations**: Common operations in object space include transformations (such as translation, scaling, and rotation), back-face culling, and hidden surface removal.
- **Visibility Determination**: In object space, visibility is determined by comparing objects or parts of objects directly against each other. Techniques like **back-face culling** (which removes faces not visible to the camera) and **depth sorting** (ordering objects by their depth from the viewer) are examples.
- **Complexity**: Object space algorithms can become complex and computationally expensive as the number of objects increases, because visibility and other calculations require pairwise comparisons between objects or polygons.

2. Image Space (2D coordinates system)

Image space refers to the coordinate system associated with the final image that is rendered on the screen. This space is defined by the pixel grid of the display device. Image space methods perform operations on a per-pixel basis, focusing on how each pixel in the final image will be colored based on the scene and objects.

Characteristics of Image Space Methods:

- **Coordinate System**: Image space uses the 2D coordinates of the display screen or image, where each coordinate corresponds to a pixel.
- **Focus on Pixels**: Operations in image space are focused on determining the color and depth of each pixel in the final image rather than manipulating object data directly.
- **Visibility Determination**: Visibility is determined on a per-pixel basis, often using methods like the **Z-buffer algorithm** (which keeps track of the depth of every pixel and ensures that only the closest pixel to the viewer is rendered).
- **Efficiency**: Image space methods are generally more efficient for rendering complex scenes because they focus only on the visible pixels rather than processing the entire geometric data of objects.
- **Memory Usage**: Image space methods typically require more memory because they store additional data for each pixel (like depth information in

the Z-buffer).

Summary

- **Object Space Methods**: Operate on geometric data directly in the local coordinate system of objects. They are precise but can be computationally expensive due to pairwise comparisons between objects.
- **Image Space Methods**: Operate on the image or screen space, determining visibility and pixel colors on a per-pixel basis. They are more efficient for complex scenes but may require more memory due to pixel data storage.

Both methods have their advantages and are often used together in various rendering pipelines to optimize performance and visual quality.

Feature	Object Space Methods	Image Space Methods
Definition	Operations are performed directly on the objects in the scene.	Operations are performed on the final image representation.
Examples	Ray tracing, backface detection.	Rasterization, z-buffering, scanline rendering.
Coordinate System	Object's local or world coordinates.	Screen or pixel coordinates.
Visibility Determination	Checks visibility by comparing objects and their geometry.	Checks visibility by comparing pixel depths (depth-buffer).
Performance	Generally slower; computationally expensive as complexity increases.	Generally faster; handles large numbers of objects efficiently.
Memory Usage	Lower memory usage; processes fewer elements at once.	Higher memory usage due to image buffers and depth buffers.
Accuracy	High accuracy in calculating intersections and object details.	Limited by screen resolution; can introduce aliasing artifacts.
Parallelization	Less parallelization; objects need to be processed sequentially.	Highly parallelizable; pixels can be processed independently.
Suitability	Suitable for scenes with complex interactions (reflections, refractions).	Suitable for real-time applications like games and interactive graphics.

Feature	Object Space Methods	Image Space Methods
Handling of Transparency	More accurate handling of transparency and overlapping surfaces.	Requires additional techniques (like alpha blending) for transparency.

Back-Face Detection

Back-face detection is a technique used in computer graphics to determine which faces of a polyhedron are not visible from a specific viewpoint. This helps optimize rendering by not drawing these hidden faces, saving computational resources.

Understanding the Polygon Surface and Its Plane Equation

A polygon in 3D space can be represented by a plane equation:

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

Here, A, B, and C are the plane parameters (components of the normal vector to the plane), and D is a constant.

For any point (x, y, z) on the plane, this equation holds true. To determine if a point is "inside" (behind) a polygon surface, the condition used is:

$$Ax + By + Cz + D < 0$$

This inequality states that if a point (x, y, z) satisfies this condition, it lies inside the polygon from the perspective of the camera or viewpoint.

3. Normal Vector and Viewing Direction

Each polygon surface has a normal vector $\mathbf{N} = (A, B, C)$. To check if a polygon is facing away from the camera (and thus a back face), we consider the viewing vector $\mathbf{V}$.

Simplifying the Back-Face Detection Test

Consider the **normal vector** $\mathbf{N}$ to the polygon surface. If $\mathbf{V}$ is the vector in the viewing direction from the eye (or camera) position, the polygon is a back face if:

$$\vec{\mathbf{V}} \cdot \vec{\mathbf{N}} > 0$$

where:

- $\vec{\mathbf{V}} \cdot \vec{\mathbf{N}}$ represents the dot product of the viewing vector and the normal vector.

Explanation of the Dot Product Condition

The dot product $\vec{\mathbf{V}} \cdot \vec{\mathbf{N}} = V_x A + V_y B + V_z C$. For a polygon to be a back face, the result of the dot product should be positive.

4. Viewing Vector Simplification in Projection Coordinates

In many graphics systems, after transforming the object description to projection coordinates, the viewing direction aligns with the z_v axis. Thus, the viewing vector simplifies to:

$$\vec{\mathbf{V}} = (0, 0, V_z)$$

Substituting this into the dot product equation:

$$\vec{\mathbf{V}} \cdot \vec{\mathbf{N}} = 0 \cdot A + 0 \cdot B + V_z \cdot C = V_z C$$

This simplifies our back-face test to only consider $V_z C$.

5. Condition in a Right-Handed Viewing Coordinate System

- **Right-Handed System**: In a right-handed coordinate system with the viewing direction along the negative z_v axis, we typically have $V_z = -1$.

Therefore, the condition becomes:

$$(-1) \cdot C > 0 \implies C < 0$$

This means if $C < 0$, the polygon is a back face because the dot product $\mathbf{V} \cdot \mathbf{N}$ is positive, indicating the normal vector points away from the viewer.

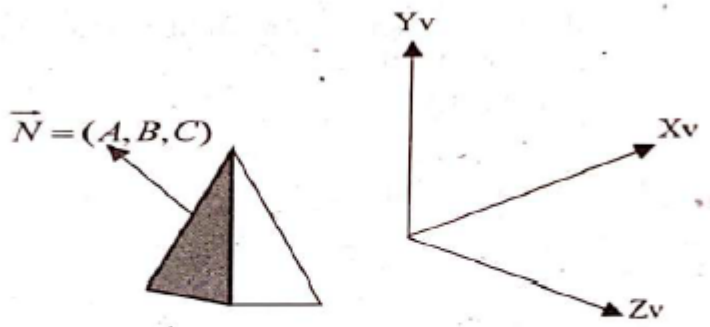


***Figure 7.1:** A polygon surface with plane parameter C < 0 in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.*

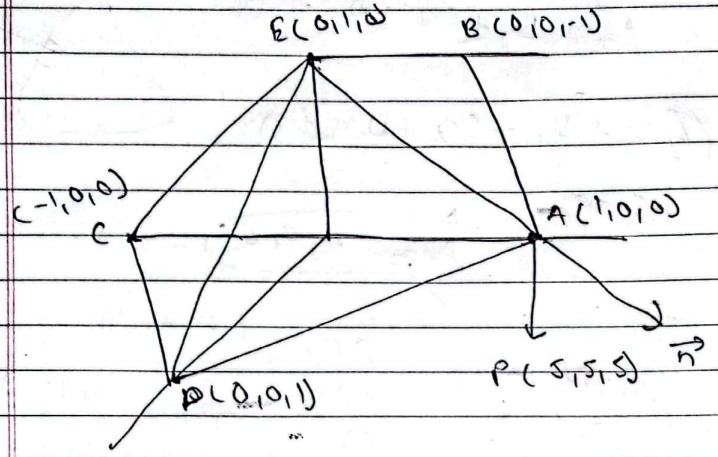
6. General Condition for Back-Face Detection

To generalize, if $C \leq 0$ in a right-handed viewing coordinate system (with the viewing direction along the negative z_v axis), the polygon is considered a back face.

This is because:

- When $C < 0$, the polygon faces away from the viewer.
- When $C = 0$, the polygon is parallel to the viewer's line of sight (grazing the viewing plane).

① Find the visibility for the surface AED where the observer at P(5,5,5)



Solution,

Step 1 : Find the normal vector n for AED surface (always take anti clockwise direction convention)

i.e AE × AD NOT AD × AE

$AE = E - A = (0-1)i + (1-0)j + (0-0)k$

$= -i + j$

$AD = D - A = (0-1)i + (0-0)j + (1-0)k$

$= -i + k$

$N = AB \times AD$

$= (-i+j) \times (-i+k)$

$$= \begin{vmatrix} i & j & k \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{vmatrix}$$

$= i(1-0) - j(-1-0) + k(0+1)$

$= i + j + k$

Step 2: The observer is at P(5,5,5) so we can construct the view vector V from surface to view point A (1,0,0) as :-

$V = PA$

$= A - P$

$= (1-5)i + (0-5)j + (0-5)k$

$= -4i - 5j - 5k$

Step 3:

To find the visibility of the object we use dot product of view vector and normal vector as :-

$\vec{V} \cdot \vec{N} = (-4i - 5j - 5k)(i + j + k)$

$= -4 - 5 - 5$

$= -14 < 0$

Since $\vec{V} \cdot \vec{N} > 0$ is false this is visible (frontface)

Depth-Buffer Method (Z-buffer)

The Depth Buffer Method, also known as the Z-buffer method, is a technique used in computer graphics to determine the visible surfaces of a scene and handle the visibility of objects based on their depth values.

Overview of Depth Buffer Method

1. **Concept**: The depth buffer method is an image-space technique for detecting visible surfaces in a scene. It works by comparing the depth (z-values) of objects at each pixel on the projection plane (view plane). The depth is measured along the z-axis of the viewing system.
2. **Buffers**:
   - **Depth Buffer**: Stores the depth values (z-values) for each pixel position on the view plane.
   - **Refresh Buffer**: Stores the intensity (color) values for each pixel position.
3. **Initialization**:
   - Depth buffer is initialized to 0 (indicating maximum depth or farthest distance).
   - Refresh buffer is initialized to the background intensity.
4. **Processing**:
   - For each surface in the scene, the depth values are calculated and compared to the existing values in the depth buffer.
   - If the new depth value is closer (i.e., smaller) than the current value in the depth buffer, the depth buffer is updated with the new depth, and the refresh buffer is updated with the surface's intensity at that pixel.

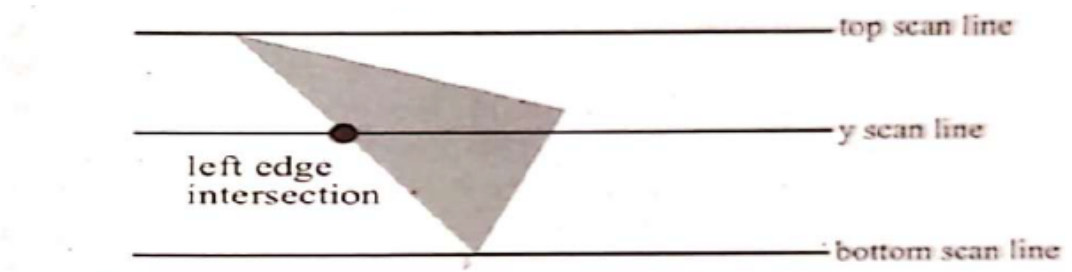
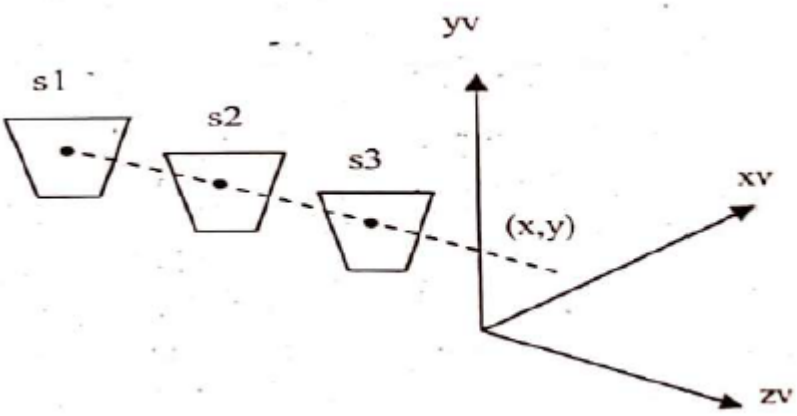


Figure 7.4: Scan lines intersecting a Polygen surfaces



After processing all the pixels, the frame buffer will contain the colors of the visible surfaces, and the Z-buffer will contain the depth values of these surfaces.

Derivation of Depth Values

The process of calculating depth values involves equations derived from the plane equation of each surface.

1. **Plane Equation**: The depth (z) at any position (x, y) on a polygon surface is determined using the plane equation:

$$z = \frac{-Ax - By - D}{C}$$

where A, B, C, and D are constants defining the plane.

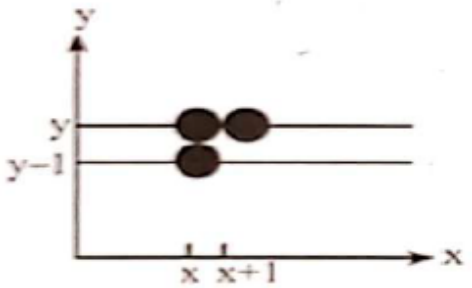


Figure 7.3 : From position (x, y) on a scan line, the next position across the line has coordinates (x+1, y), and position immediately below on the next line has coordinates (x, y-1)

2. **Calculating Depth Across a Scan Line**: To calculate the depth at the next position $(x+1, y)$ on the same scan line, you can use the plane equation:

$$z' = \frac{-A(x+1) - By - D}{C}$$

Simplify this using the depth z at position (x, y):

$$z' = \frac{-Ax - By - D}{C} + \frac{-A}{C}$$

Hence:

$$z' = z + \frac{-A}{C}$$

The ratio $\frac{A}{C}$ is constant for each surface, so depth values along a scan line can be computed incrementally.

3. **Scan Line Processing**:
   - Start by calculating the depth at the left edge of the polygon where it intersects the scan line.
   - Calculate subsequent depth values across the scan line using the incremental depth calculation.

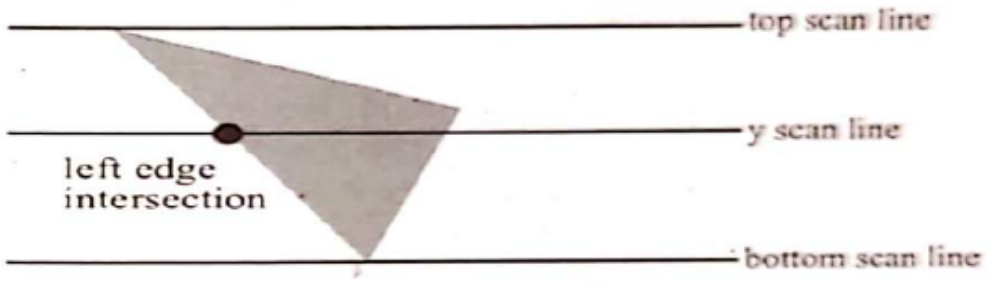


Figure 7.4: Scan lines intersecting a Polygen surfaces

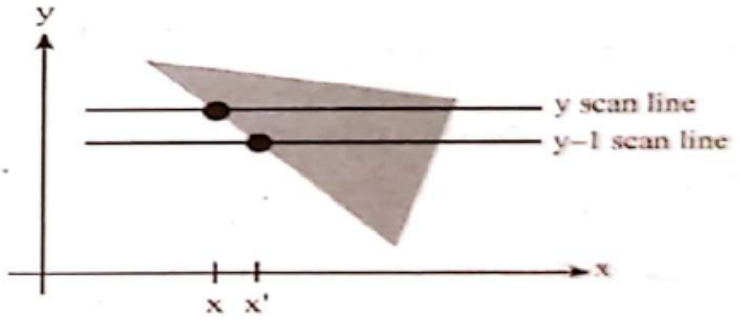


Figure 7.5: Intersection positions on successive scan lines along a left polygon edge.

4. **Vertical Edge Handling**:
    - For vertical edges where $A = 0$, the depth changes linearly with y, and the depth update is simply:

$$z' = z + \frac{B}{C}$$

Advantages:

1. The depth-buffer method is easy to implement
2. Ensures accurate depth representation for each pixel..
3. Supports real-time rendering and dynamic scenes.
4. No need to sort surfaces; it directly handles visibility by comparing depth values.

Disadvantages:

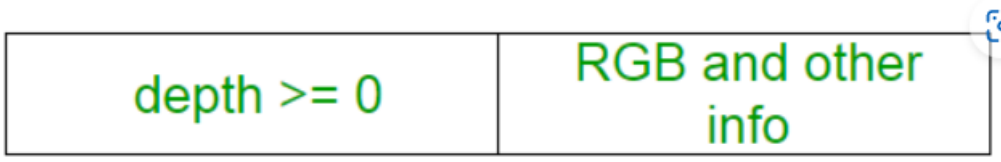
1. Requires additional memory for the depth buffer.
2. Can be less efficient for scenes with frequent depth changes.
3. May suffer from precision issues with limited bit-depth.
4. Only handles opaque surfaces, so it cannot manage transparency or multiple overlapping surfaces effectively.

A-buffer Method

The A-buffer method in computer graphics is an advanced technique for handling hidden face detection and anti-aliasing, especially useful in rendering scenes with transparent surfaces. Here's a breakdown of how it works:

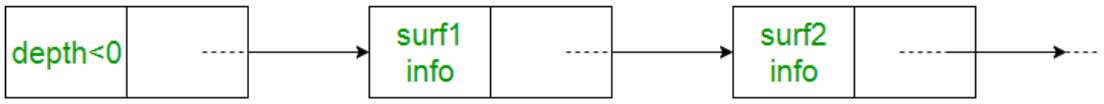
Overview of the A-buffer Method

1. **Extension of the Z-buffer Method:**
   - The A-buffer method builds on the depth-buffer (or Z-buffer) technique, which is primarily used for opaque objects. The Z-buffer method only stores depth values to determine which object is in front, which doesn't work well for transparent objects. The A-buffer method extends this by handling transparency and providing more accurate rendering of scenes with overlapping surfaces.
2. **Data Structure:**
   - **Accumulation Buffer:** The key data structure in the A-buffer method is the accumulation buffer. Each pixel in the A-buffer has two main fields:
     - **Depth Field:** Stores a depth value (a positive or negative real number) indicating the distance of the surface from the camera.
     - **Surface Data Field (or Intensity Field):** Stores either the RGB color components and coverage percentage of a single surface or a pointer to a linked list of surfaces contributing to that pixel.
3. **Handling Surfaces:**
   - **Opaque Surfaces:** If the depth value is non-negative, it indicates a single surface. The intensity field will store the color and coverage percentage of that surface.



(a) When a pixel overlap by only one surface

- **Transparent Surfaces:** If the depth value is negative, it indicates multiple overlapping surfaces. The intensity field then contains a pointer to a linked list of surfaces that contribute to the pixel's color.



(b) When a pixel overlaps by multiple surfaces

4. **Anti-Aliasing:**
    - The A-buffer method also provides anti-aliasing. Anti-aliasing is a technique used to smooth out jagged edges by averaging colors over sub-pixels. In the A-buffer method, each pixel is divided into sub-pixels, and the final color of a pixel is computed by summing up the colors of these sub-pixels.
5. **Memory Usage:**
    - The A-buffer method generally requires more memory compared to the Z-buffer method because it needs to store additional information for each pixel, including linked lists for multiple surfaces.

Summary

The A-buffer method improves on the Z-buffer technique by handling transparent surfaces and providing anti-aliasing. It uses an accumulation buffer where each pixel can store depth information and a linked list of surfaces for correct color composition. This method can be more memory-intensive but offers better visual results, particularly in scenes with complex overlapping transparent objects.

Pros of A-buffer Method:

1. Handles transparency and multiple overlapping surfaces accurately.
2. Provides anti-aliasing by averaging colors over sub-pixels.
3. Capable of correctly compositing different surface colors.

4. Extends the Z-buffer method for more complex rendering scenarios.

Cons of A-buffer Method:

1. Requires significantly more memory than the Z-buffer method.
2. More computationally intensive due to linked list management.
3. Can be slower due to increased processing complexity.
4. Implementation can be more complex compared to simpler methods.

Feature	Z-Buffer Method	A-Buffer Method
Primary Function	Depth buffering to handle visibility of objects.	Antialiasing and handling transparency.
Buffer Type	Z-buffer stores depth information for each pixel.	A-buffer stores color, coverage, and alpha information.
Handling Visibility	Uses depth values to determine which pixel is in front.	Handles transparency and partial coverage.
Complexity	Relatively simple and efficient.	More complex due to managing multiple color and alpha values.
Transparency Handling	Limited; can only handle simple transparency.	Handles complex transparency and overlapping objects.
Performance	Generally fast; suitable for real-time applications.	Can be slower due to additional calculations for transparency and antialiasing.
Memory Usage	Requires additional memory for depth buffer.	Requires more memory due to storing multiple color and alpha values.
Use Case	Suitable for real-time rendering, e.g., games and simulations.	Suitable for high-quality rendering with complex transparency effects, e.g., movies and detailed visualizations.

Scan Line Method

The Scan-Line Algorithm is a method used in computer graphics to handle Hidden Surface Removal (HSR) in 3D rendering. It's designed to determine

which surfaces of polygons are visible and should be displayed on the screen. Here's a step-by-step explanation of how it works:

Overview

1. **Concept**: The Scan-Line Algorithm processes one horizontal line (scan line) at a time across the image. It uses this method to determine the visible surfaces and update the frame buffer accordingly.
2. **Data Structures**:
    - **Edge List Table**: Maintains all edges of the polygons with their endpoint coordinates.
    - **Active Edge Table (AET)**: Contains edges intersected by the current scan line, sorted by their x-coordinates.
    - **Polygon Table**: Stores information about each polygon, including its ID, plane equation, color information, and a flag indicating whether it's visible or not.

Steps in the Algorithm

1. **Initialization**:
    - **Edge Table**: Populate with all edges of the polygons.
    - **Active Edge Table**: Initialize with edges intersected by the current scan line.
    - **Polygon Table**: Set up with details of each polygon.
2. **Processing Scan Lines**:
    - **For Each Scan Line**:
        - **Update AET**: Add edges that intersect the current scan line and remove those that have passed it. Sort the active edges by their x-coordinates.
        - **Color Calculation**:
            - **Single Polygon**: If only one polygon's edge intersects the scan line, use its color intensity for that line.
            - **Multiple Polygons**: If multiple polygons overlap, calculate the depth (z-coordinate) for each polygon to determine which one

is closer to the viewer. The polygon with the minimum depth is considered visible.

- **Update Frame Buffer**: Write the color intensity of the visible polygon into the frame buffer.

3. **Handling Overlapping Polygons**:
   - When polygons overlap, the algorithm determines which surface is in front by comparing their depths. The surface with the smaller (closer) depth value is considered visible.
4. **Concept of Coherence**:
   - **Coherence** refers to leveraging regularities in the scene. For example, if a scan line that intersects some polygons is processed, the next scan line is likely to intersect similar polygons. This concept helps to optimize the algorithm by reusing previously computed data.

Example Illustration

Let's understand more by the example as shown in the below in Fig.4 figure:

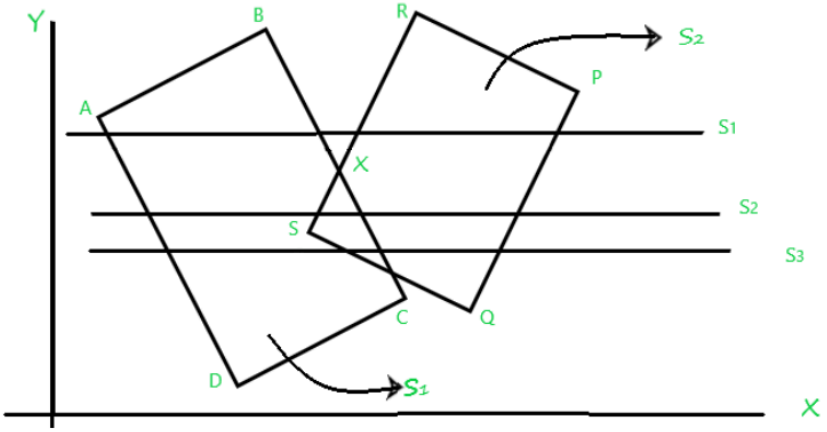


Fig.4

Imagine you have two overlapping polygons and you're processing three scan lines (S1, S2, S3):

1. **Scan Line S1**:
   - Determine which edges intersect S1.
   - Update the frame buffer with the color intensity of the polygons visible at S1.

2. **Scan Line S2**:
    - Again, determine intersecting edges.
    - If polygons overlap, compare their depths to decide which polygon's color to use for the frame buffer.
3. **Scan Line S3**:
    - Process similarly to S2, leveraging coherence to avoid redundant calculations.

The algorithm efficiently manages the visibility of polygons by focusing on one scan line at a time and using depth information to resolve visibility issues.

Pros of the Scan-Line Algorithm

- **Efficient for Polygon Filling**: Directly processes one line at a time, which is straightforward for polygon filling.
- **Handles Multiple Polygons**: Capable of managing scenes with multiple overlapping polygons.
- **Leverages Coherence**: Reuses information from previous scan lines to improve performance.
- **Simple Implementation**: Conceptually easier to understand and implement compared to more complex algorithms.

Cons of the Scan-Line Algorithm

- **Limited to Convex Polygons**: May struggle with complex concave polygons or non-manifold geometries.
- **Depth Calculation Complexity**: Requires depth calculations for overlapping polygons, which can be computationally expensive.
- **Memory Usage**: Needs to maintain and update multiple tables (Edge Table, Active Edge Table, Polygon Table), which can increase memory usage.
- **Not Suitable for Dynamic Scenes**: Inefficient for scenes with frequent changes or animations, as it may require extensive recalculation.

When choosing a visible surface detection algorithm, you should consider these factors

1. **Accuracy**: How precisely the algorithm renders visible surfaces and handles occlusions.
2. **Efficiency**: The algorithm's computational complexity and its impact on performance.
3. **Memory Usage**: The amount of memory required to process and store data.
4. **Scalability**: How well the algorithm handles increasing complexity or larger scenes.
5. **Real-Time Capability**: The algorithm's suitability for applications needing immediate feedback.
6. **Implementation Complexity**: The ease or difficulty of integrating and modifying the algorithm.

To implement the Z-buffer algorithm for a 512 x 512 image with 24-bit color depth, you'll need memory for both the color buffer and the depth buffer:

1. **Color Buffer**:
   - Image dimensions: 512 x 512
   - Color depth: 24 bits (3 bytes per pixel)

   Memory required: $512 \times 512 \times 3 \text{ bytes} = 786,432 \text{ bytes}$ or approximately 768 KB.
2. **Depth Buffer**:
   - Image dimensions: 512 x 512
   - Depth buffer depth: Typically 24 bits (3 bytes per pixel)

   Memory required: $512 \times 512 \times 3 \text{ bytes} = 786,432 \text{ bytes}$ or approximately 768 KB.

Total Memory Required: $786,432 \text{ bytes (color buffer)} + 786,432 \text{ bytes (depth buffer)} = 1,572,864 \text{ bytes}$ or approximately 1.5 MB.