**GRAPHICS AND VISUAL COMPUTING**

**Mid-Term (50 marks)**

* **Digitize line from (10,8) to (18,6) using DDA line Algorithm. Plot points on Cartesian graph.**

**(numerical)**

* **Give Mid point circle generating algorithm. Why algorithm only computer pixel in one octant.**

The midpoint circle generating algorithm is an algorithm used to draw circles on a computer screen. It is based on the principle of symmetry and efficiency.

The algorithm only computes pixels in one octant of the circle (usually the first octant, where the x-coordinate is positive and the y-coordinate is positive or zero). The other seven octants of the circle can be determined by symmetry.

The reason for considering only one octant is to reduce computational complexity. Since circles have symmetry, we can determine the positions of pixels in the other octants by reflecting the computed pixels across the horizontal, vertical, and diagonal axes.

By focusing on one octant, we can iterate through the points on the circumference of the circle and compute their positions by starting from a specific point (e.g., the rightmost point on the x-axis). The algorithm uses the midpoint of the previously computed pixel positions to determine the next pixel position.

In simple terms, the algorithm starts from the rightmost point of the circle and moves along the circumference by considering the next closest pixel. It checks a midpoint between the previously computed pixel and the next pixel to determine whether it should move vertically or diagonally. This process continues until the circle is complete in the first octant.

By utilizing symmetry and computing pixels in only one octant, the algorithm minimizes the number of computations required to draw a circle, making it more efficient and faster to execute on a computer.

1. Initialize the center of the circle at coordinates (x0, y0) and the radius as 'r'.
2. Set the initial point on the circumference of the circle as (x, y) = (0, r).
3. Calculate the initial value of the decision parameter as P = 5/4 - r.
4. At each step, while x <= y, do the following:
5. Plot the eight symmetric points (using symmetry properties) corresponding to the current (x, y) position.
6. If P < 0, update the decision parameter as P = P + 2x + 1.
7. If P >= 0, update the decision parameter as P = P + 2x - 2y + 1 and decrement y by 1.
8. Increment x by 1.
9. Repeat step 4 until x > y.

* **Find the reflection of a triange A(3,5), B(5,8) and C(7,10) object about the line Y=X.**

**(numerical)**

https://www.javatpoint.com/computer-graphics-reflection

* **Show that equation ax+by+cz+d=0 represents a plane.**

**(numerical)**

* **Give the Boundary Fill algorithm for polygon filling. Explain how it is different from flood fill algorithm.**

**Boundary Fill Algorithm for Polygon Filling:**

* Select a seed point inside the polygon.
* Check if the current pixel is within the boundary of the polygon.
* If it is within the boundary, fill the current pixel with the desired color.
* Recursively apply steps 2 and 3 to the neighboring pixels until the entire polygon is filled.

**Difference from Flood Fill Algorithm:**

Target Area: Boundary Fill algorithm is used for filling closed polygons, whereas the Flood Fill algorithm is used for filling any bounded area, regardless of its shape.

Methodology: Boundary Fill starts from the inside of the polygon and proceeds towards the boundary, filling the area step by step. Flood Fill starts from a given point and spreads outwards, filling the area until it reaches a boundary.

Boundary Handling: Boundary Fill stops at the boundary of the polygon, whereas Flood Fill can continue filling until it encounters a specific boundary color or condition.

Complexity: Boundary Fill algorithm is usually more computationally expensive than Flood Fill because it needs to check the boundary conditions for each pixel, whereas Flood Fill algorithm typically checks only the color of neighboring pixels.

* **Write a short note on viewing transformation.**

Viewing transformation is an important concept in computer graphics that helps us see a three-dimensional (3D) scene on a two-dimensional (2D) screen, like your computer monitor. It involves transforming the coordinates of objects in a 3D world to the coordinates of pixels on your screen.

When you look at a 3D object in the real world, it appears differently depending on your viewpoint. Similarly, in computer graphics, the viewing transformation simulates this process by specifying where the "camera" is located and how it is oriented.

**To perform the viewing transformation, we need to define a few things**:

**Camera position:** This is the location of the virtual camera in the 3D world. It determines where you are looking at the scene from.

**Camera orientation**: This refers to the direction the camera is facing. It determines which part of the scene is visible and which parts are hidden from view.

**Projection:** This step involves converting the 3D coordinates of objects in the scene to 2D coordinates on the screen. This is necessary because our screens can only display 2D images. There are various projection techniques, such as perspective projection and orthographic projection, each with its own characteristics.

By combining the camera position, orientation, and projection, the viewing transformation calculates how each object in the 3D world should appear on the 2D screen. This process is performed for every object in the scene, creating the illusion of a 3D scene on a flat screen.

In simple terms, the viewing transformation is like taking a picture with a camera, where you decide where to position the camera, which direction to point it, and how the resulting image should look on the screen.

* **For successive rotation establish R(θ1 + θ2) = R(θ1).R(θ2) = R(θ2).R(θ1).**

**(numerical)**

To establish the relationship R(θ₁ + θ₂) = R(θ₁) \* R(θ₂) = R(θ₂) \* R(θ₁) for successive rotations, we need to consider the properties of rotation matrices and how they interact when multiplied.

Let's assume that R(θ) represents a rotation matrix for a counterclockwise rotation by angle θ.

For a single rotation R(θ₁), the resulting transformation matrix can be represented as:

R(θ₁) = | cos(θ₁) -sin(θ₁) |

| sin(θ₁) cos(θ₁) |

Similarly, for another rotation R(θ₂), the resulting transformation matrix can be represented as:

R(θ₂) = | cos(θ₂) -sin(θ₂) |

| sin(θ₂) cos(θ₂) |

Now, let's calculate the product of R(θ₁) and R(θ₂):

R(θ₁) \* R(θ₂) = | cos(θ₁) -sin(θ₁) | \* | cos(θ₂) -sin(θ₂) |

| sin(θ₁) cos(θ₁) | | sin(θ₂) cos(θ₂) |

Multiplying these matrices, we get:

R(θ₁) \* R(θ₂) = | cos(θ₁) \* cos(θ₂) - sin(θ₁) \* sin(θ₂) -cos(θ₁) \* sin(θ₂) - sin(θ₁) \* cos(θ₂) |

| sin(θ₁) \* cos(θ₂) + cos(θ₁) \* sin(θ₂) -sin(θ₁) \* sin(θ₂) + cos(θ₁) \* cos(θ₂) |

By applying trigonometric identities (sum of angles), we can simplify this expression:

R(θ₁) \* R(θ₂) = | cos(θ₁ + θ₂) -sin(θ₁ + θ₂) |

| sin(θ₁ + θ₂) cos(θ₁ + θ₂) |

This result shows that the product of the rotation matrices R(θ₁) and R(θ₂) corresponds to a single rotation R(θ₁ + θ₂). Therefore, we can establish the relationship:

R(θ₁ + θ₂) = R(θ₁) \* R(θ₂) = R(θ₂) \* R(θ₁)

This means that the order in which rotations are applied does not matter; the final result will be the same.

* **What is clipping? Discuss the Liang-Barsky line clippling algorithm.**

Clipping in computer graphics refers to the process of determining which parts of an object or image should be displayed on the screen. It involves removing or discarding the portions of an object that lie outside the visible region, known as the viewport or clipping window.

The Liang-Barsky line clipping algorithm is a method used to efficiently clip lines against a rectangular clipping window. It helps determine which parts of a line segment lie inside the clipping window and discards the portions that are outside.

The algorithm works by checking the intersections of the line segment with the boundaries of the clipping window. It calculates and compares the parameter values (also known as "t-values") of the line segment where it intersects with each boundary. These parameter values represent the position of the intersection point along the line segment.

The Liang-Barsky algorithm divides the line segment into different segments based on the intersections with the clipping window boundaries. It assigns parameter values to each segment, indicating the range of the line segment that lies inside the clipping window.

The algorithm checks four conditions for each boundary of the clipping window: left, right, bottom, and top. By comparing the parameter values of the line segment with these conditions, it determines if any part of the line segment lies inside the clipping window. If it does, the algorithm calculates the new parameter values that define the clipped portion of the line segment.

The resulting clipped line segment is then drawn on the screen, showing only the visible portion within the clipping window.

In simple words, the Liang-Barsky line clipping algorithm is like using a pair of scissors to cut a line into smaller pieces based on where it intersects with the edges of a rectangular window. It helps us keep only the parts of the line that are visible within the window, allowing us to display objects accurately on the screen and remove any unnecessary portions that are outside the window.

* **Give architecture of the Frame Buffer. Define Aliasing and Anti-Aliasing. How long would it take to load a 1024 x 1024 frame buffer with 12 bit per pixel, if 103 bits can transfer per second?**

**(numerical)**

**Architecture of the Frame Buffer:**

The frame buffer is a dedicated portion of memory in a computer or graphics processing system that stores the pixel data for displaying images on a screen. It serves as a buffer or temporary storage for the pixels that make up a frame of the image.

**The architecture of a frame buffer typically consists of the following components:**

a) Pixel Array: This is a 2D grid of memory locations that corresponds to the dimensions of the screen or display. Each location in the array represents a single pixel on the screen.

b) Color Depth: The frame buffer has a defined color depth or bit depth, which determines the number of bits allocated to each pixel. This bit depth specifies the range of colors or shades that can be displayed for each pixel.

c) Addressing Mechanism: The frame buffer utilizes an addressing mechanism to access and modify individual pixels in the pixel array. The addressing mechanism maps the pixel coordinates to the appropriate memory location in the array.

d) Control Circuitry: The frame buffer is controlled by circuitry that manages the reading, writing, and manipulation of pixel data. This circuitry ensures synchronization with the screen refresh rate and handles operations such as pixel updates and color palette management.

**Aliasing and Anti-Aliasing:**

**a) Aliasing:** Aliasing is a visual phenomenon that occurs when a high-frequency pattern or signal is sampled or displayed at a lower resolution. It leads to distortion, jagged edges, or the appearance of unwanted patterns (moire patterns) in the image or object.

In computer graphics, aliasing often manifests as "jaggies" or staircase-like edges on diagonal lines or curves, particularly when the image or object is displayed on a screen with a lower resolution than the original data. It occurs due to the limited number of pixels available to represent the continuous shape or signal accurately.

**b) Anti-Aliasing:** Anti-aliasing is a technique used to reduce or eliminate the effects of aliasing in computer graphics. It aims to smooth out the jagged edges and improve the visual quality of images or objects.

Anti-aliasing works by adding extra pixels, blending colors, or applying filtering algorithms to reduce the abrupt transitions between different color or intensity levels. This helps create the illusion of smoother edges and more accurate representation of the original shape or signal.

**Loading Time for a 1024x1024 Frame Buffer with 12 bits per pixel:**

To calculate the loading time, we need to consider the total number of bits required to store the frame buffer.

Number of Pixels = 1024 x 1024 = 1,048,576 pixels

Bits per Pixel = 12 bits

Total Bits = Number of Pixels x Bits per Pixel

= 1,048,576 x 12

= 12,582,912 bits

Given that the transfer rate is 10^3 bits per second, we can divide the total number of bits by the transfer rate to find the loading time:

Loading Time = Total Bits / Transfer Rate

= 12,582,912 bits / 10^3 bits per second

= 12,582.912 seconds

Therefore, it would take approximately 12,582.912 seconds (or about 3 hours and 30 minutes) to load a 1024x1024 frame buffer with 12 bits per pixel, assuming a transfer rate of 10^3 bits per second.

* **Find Position of triangle with vertex A(0,0), B(2,0) and C(2,2) after rotation about origin through an angle +60o** .

**(numerical)**

To find the position of a triangle with vertices A(0,0), B(2,0), and C(2,2) after rotating it about the origin through a positive 60-degree angle, we can apply a rotation transformation to each vertex.

**The rotation transformation matrix for a counterclockwise rotation by angle θ is**:

| cos(θ) -sin(θ) |

| sin(θ) cos(θ) |

In this case, θ = 60 degrees. Let's calculate the new coordinates for each vertex of the triangle after the rotation.

Vertex A (0, 0):

x' = cos(60°) \* 0 - sin(60°) \* 0

= 0

y' = sin(60°) \* 0 + cos(60°) \* 0

= 0

Vertex B (2, 0):

x' = cos(60°) \* 2 - sin(60°) \* 0

= 1

y' = sin(60°) \* 2 + cos(60°) \* 0

= sqrt(3)

Vertex C (2, 2):

x' = cos(60°) \* 2 - sin(60°) \* 2

= 0

y' = sin(60°) \* 2 + cos(60°) \* 2

= 4

**The new coordinates after the rotation about the origin through a 60-degree angle are:**

Vertex A' (x', y'): (0, 0)

Vertex B' (x', y'): (1, sqrt(3))

Vertex C' (x', y'): (0, 4)

Please note that the coordinates have been simplified, but they are still in terms of radicals to represent the exact values after rotation.

**END-Term (100 marks)**

* **Discuss the shearing. Find the transformation equations for shearing about both the axes in 2 dimension.**

Shearing is a transformation technique in computer graphics used to distort or skew the shape of an object. It involves shifting one part of an object relative to another part along a specific axis. Shearing can be performed along both the x-axis (horizontal shearing) and the y-axis (vertical shearing).

**Transformation Equations for Shearing in 2D:**

Horizontal Shearing:

The x-coordinate of a point after horizontal shearing is given by: x' = x + shx \* y

The y-coordinate remains unchanged: y' = y

Vertical Shearing:

The y-coordinate of a point after vertical shearing is given by: y' = y + shy \* x

The x-coordinate remains unchanged: x' = x

In these equations, (x, y) represents the original coordinates of a point, and (x', y') represents the transformed coordinates after shearing. 'shx' represents the horizontal shear factor, which determines the amount of horizontal shearing, and 'shy' represents the vertical shear factor, which determines the amount of vertical shearing.

* **What is homogenous coordinate system? Discuss the case of successive scaling and successive rotation.**

Homogeneous coordinate system is a mathematical representation used in computer graphics and computer vision. It extends the traditional Cartesian coordinate system to include an additional coordinate, called the homogeneous coordinate. Homogeneous coordinates are useful for representing transformations, such as translation, rotation, scaling, and perspective projection, in a more convenient and unified way.

In homogeneous coordinates, a point in 2D or 3D space is represented as a vector (x, y, w) or (x, y, z, w), where (x, y, z) are the Cartesian coordinates and w is the homogeneous coordinate. The homogeneous coordinate w allows us to represent translations as well as projective transformations.

**Case of Successive Scaling:**

When successive scaling transformations are applied to an object, the order of the scaling operations can affect the final result. Let's consider two scaling transformations, S1 and S2, applied successively.

If we apply scaling S1 followed by scaling S2 to a point (x, y, w) in homogeneous coordinates, the resulting point would be:

(x', y', w') = S2 \* S1 \* (x, y, w)

In this case, the order of the transformations is from right to left since matrix operations are applied in reverse order. This means that scaling S1 is applied first, followed by scaling S2.

**Case of Successive Rotation**:

Similarly, when successive rotation transformations are applied to an object, the order of the rotation operations matters. Let's consider two rotation transformations, R1 and R2, applied successively.

If we apply rotation R1 followed by rotation R2 to a point (x, y, w) in homogeneous coordinates, the resulting point would be:

(x', y', w') = R2 \* R1 \* (x, y, w)

Again, the order of the transformations is from right to left, meaning that rotation R1 is applied first, followed by rotation R2.

In both cases, the order of the transformations affects the final result because matrix multiplication is not commutative. This order dependence is a characteristic of successive transformations and highlights the importance of considering the correct order of operations when applying multiple transformations to achieve the desired result in computer graphics.

* **Give Bresenham’s circle generation algorithm. How do we generate the entire circle?**

Bresenham's circle generation algorithm is a method used to draw or generate a circle on a raster display or grid. It is an efficient algorithm that determines the points along the circumference of a circle using only integer operations, without the need for floating-point calculations.

The algorithm starts at the top point of the circle and generates points in the first octant of the circle. The remaining points are obtained by symmetry.

**The steps to generate the entire circle using Bresenham's algorithm are as follows:**

* Initialize the center coordinates (xc, yc) and the radius (r) of the circle.
* Set the initial point (x, y) at (0, r).
* Initialize the decision parameter P as P = 3 - 2 \* r.
* Repeat the following steps until x ≤ y:
* Plot the eight symmetric points centered at (xc, yc) for the current (x, y) coordinates.
* If P < 0, update P as P = P + 4 \* x + 6.
* If P ≥ 0, update P as P = P + 4 \* (x - y) + 10 and decrement y by 1.
* Increment x by 1.
* Repeat the above steps until all eight octants of the circle are generated.

By applying the algorithm to generate points in the first octant and considering symmetry, the entire circle can be obtained. The algorithm ensures that the generated points lie as close to the true circle as possible while only using integer calculations, making it efficient for raster-based circle generation.

* **Discuss Cohen-Sutherland line clipping algorithm.**

The Cohen-Sutherland line clipping algorithm is a technique used to clip line segments against a rectangular clipping window. It helps determine which portions of a line lie inside the clipping window and discards the parts that are outside.

The algorithm works by dividing the 2D space into nine regions based on the position of the line endpoints and the clipping window boundaries. Each region is assigned a unique binary code, known as a "bit code," which represents the relative position of a point with respect to the clipping window.

The bit codes are calculated as follows:

Bit 1: Represents the top boundary of the clipping window

Bit 2: Represents the bottom boundary of the clipping window

Bit 3: Represents the right boundary of the clipping window

Bit 4: Represents the left boundary of the clipping window

**The Cohen-Sutherland algorithm performs the following steps:**

Assign bit codes: Calculate the bit codes for the endpoints of the line segment and classify them based on their position with respect to the clipping window.

Check for trivial acceptance or rejection: If both endpoints have a bit code of 0000 (meaning they are inside the clipping window), the line segment is completely visible and can be drawn. If both endpoints have a bit code with a common 1-bit, the line segment lies completely outside the corresponding boundary and can be discarded.

Check for trivial rejection: If the bitwise AND operation between the bit codes of the two endpoints is not equal to 0000, the line segment is entirely outside the clipping window and can be rejected.

Perform line clipping: If the line segment is not trivially accepted or rejected, the algorithm proceeds to perform clipping. It uses the bit codes to determine which boundaries the line segment intersects and calculates new endpoints to restrict the line segment within the clipping window.

Update and repeat: After calculating the new endpoints, the algorithm repeats steps 1-4 until the line segment is either trivially accepted or rejected.

Draw the clipped line segment: Finally, the resulting clipped line segment is drawn on the screen, representing the visible portion within the clipping window.

In simple words, the Cohen-Sutherland line clipping algorithm is like using a set of rules and binary codes to check and determine if a line segment lies entirely inside, outside, or partially inside a rectangular window. By evaluating the positions of the endpoints and comparing them to the window boundaries, the algorithm determines which parts of the line segment are visible and should be drawn, while discarding the parts outside the window.

* **Show that equation ax + by + cz + d = 0 represents a plane.**
* **Write a short note on viewing transformation.**

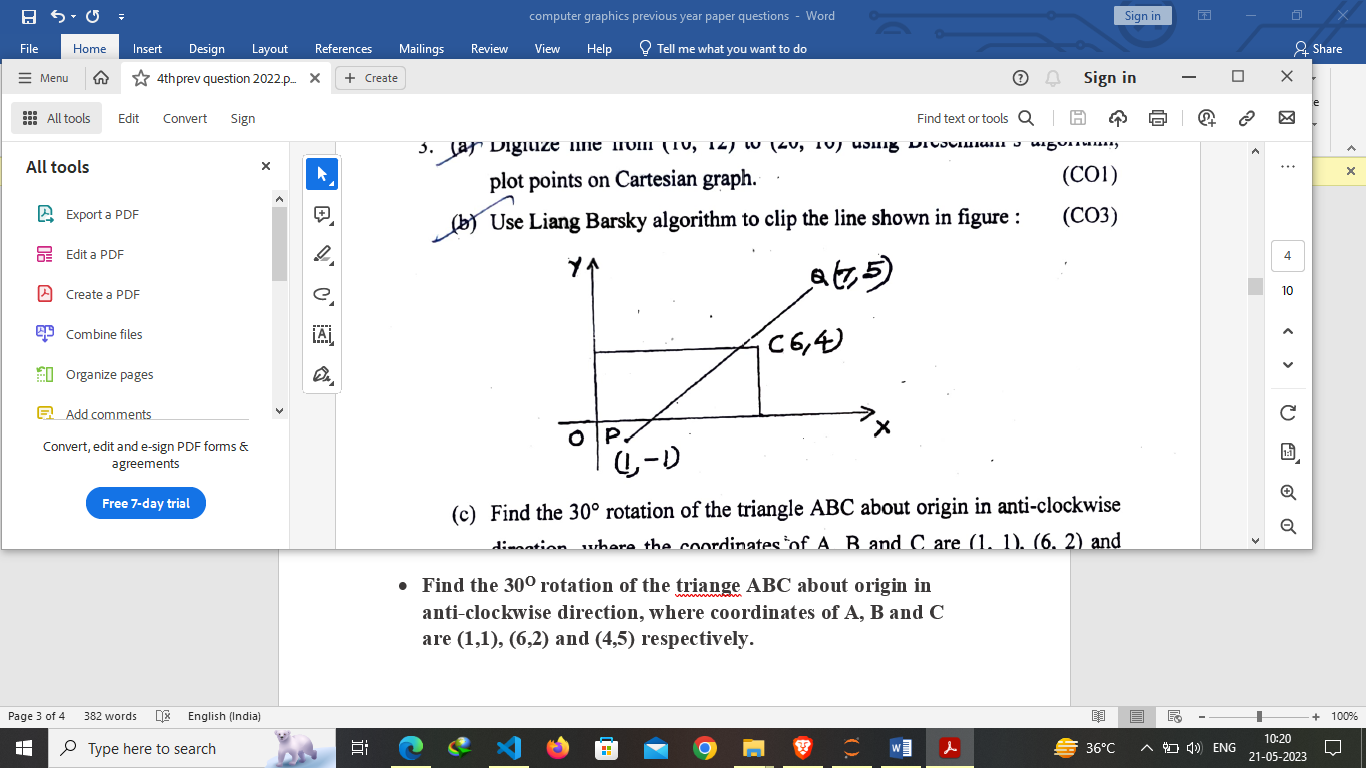
Done above

* **Digitize line from (10, 12), to (20, 16) using Bresenham’s algorithm, plot points on Cartesian graph.**

**(numerical)**

* **Use Liang Barsky algorithm to clip the line shown in figure:**

**(numerical)**



* **Find the 30O rotation of the triange ABC about origin in anti-clockwise direction, where coordinates of A, B and C are (1,1), (6,2) and (4,5) respectively.**

To find the 30-degree rotation of triangle ABC about the origin in the counterclockwise direction, we need to apply a rotation transformation to each vertex of the triangle. The rotation transformation matrix can be used to perform this operation.

**The rotation transformation matrix for a counterclockwise rotation by angle θ is:**

| cos(θ) -sin(θ) |

| sin(θ) cos(θ) |

In this case, θ = 30 degrees. Let's calculate the new coordinates for each vertex of the triangle.

Vertex A (1, 1):

x' = cos(30°) \* 1 - sin(30°) \* 1

= sqrt(3)/2 - 1/2

y' = sin(30°) \* 1 + cos(30°) \* 1

= 1/2 + sqrt(3)/2

Vertex B (6, 2):

x' = cos(30°) \* 6 - sin(30°) \* 2

= 3\*sqrt(3) - sqrt(3)

y' = sin(30°) \* 6 + cos(30°) \* 2

= 3/2 + 2\*sqrt(3)

Vertex C (4, 5):

x' = cos(30°) \* 4 - sin(30°) \* 5

= 2\*sqrt(3) - (5/2)

y' = sin(30°) \* 4 + cos(30°) \* 5

= 2 + (5\*sqrt(3))/2

**The new coordinates after the 30-degree counterclockwise rotation about the origin are:**

Vertex A' (x', y'): (sqrt(3)/2 - 1/2, 1/2 + sqrt(3)/2)

Vertex B' (x', y'): (3sqrt(3) - sqrt(3), 3/2 + 2sqrt(3))

Vertex C' (x', y'): (2sqrt(3) - 5/2, 2 + (5sqrt(3))/2)

Please note that the coordinates have been simplified, but they are still in terms of radicals to represent the exact values after rotation.

* **Give architecture of the frame buffer. Define Aliasing and Anti-aliasing. How long would it take to load a 1024 x 1024 frame buffer with 12 bit per pixel, if 103 bits can transfer per second?**

Done above

* **Find the reflection of the triangle ABC about line Y = X, where the coordinates of A, B and C are (3,5); (7,10) and (6,8) respectively.**

https://www.javatpoint.com/computer-graphics-reflection

* **Give Z-buffer algorithm for elimination of hidden surface. Why is removal of hidden surface required?**

The Z-buffer algorithm is a widely used method for hidden surface removal in computer graphics. It is a depth-based algorithm that determines which surfaces are visible or hidden in a 3D scene and ensures that only the visible surfaces are rendered.

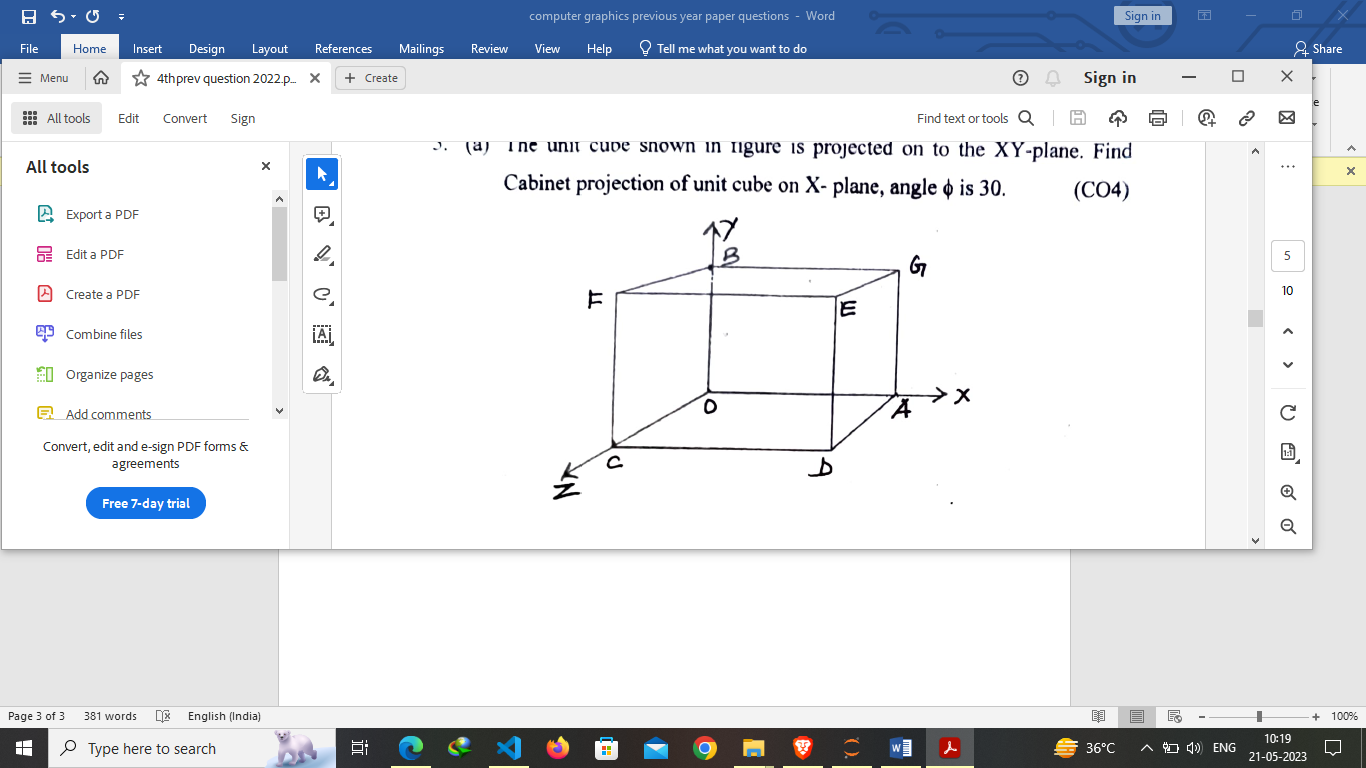
**The Z-buffer algorithm works as follows:**

* Initialize a buffer called the Z-buffer or depth buffer with a large value (e.g., infinity) for all pixels in the frame buffer.
* For each polygon in the scene:
  + For each pixel covered by the polygon:
  + Calculate the depth (Z-coordinate) of the pixel.
  + Compare the depth of the pixel with the value stored in the corresponding location of the Z-buffer.
  + If the pixel's depth is closer (smaller) than the value in the Z-buffer, update the Z-buffer with the pixel's depth and update the frame buffer with the color of the polygon.
* Repeat step 2 for each polygon in the scene, ensuring that polygons are rendered in the correct order based on their distance from the viewer.

By using the Z-buffer algorithm, hidden surfaces that are occluded by other objects or closer to the viewer can be properly identified and removed from the final rendered image. This process enhances the realism and visual quality of 3D scenes, as only the visible surfaces contribute to the rendered image.

Removal of hidden surfaces is necessary because in a 3D scene, multiple objects or surfaces can overlap or intersect with each other. If all surfaces were rendered without any consideration for their visibility, the final image would not accurately represent the scene and could lead to incorrect depth perception or visual artifacts. By eliminating hidden surfaces, the algorithm helps create a more realistic and visually coherent representation of the scene by only showing the surfaces that are visible to the viewer.

* **The unit cube shown in figure is projected on to the XY-plane. Find Cabinet projection of unit cube on X-plane, angle θ is 30.**



* **Determine six point on the Bezier curve with equidistant parametric value, having control points (x0, y0) = (50, 180), (x1, y1) = (250, 100), (x2, y2) = (600, 300) and (x3, y3) = (500, 50) distributed over a screen of resolution 640 x 350.**

To determine six points on the Bezier curve with equidistant parametric values, we can calculate the parametric values at regular intervals and evaluate the corresponding points on the curve using the given control points. Given the control points:

P0 = (x0, y0) = (50, 180)

P1 = (x1, y1) = (250, 100)

P2 = (x2, y2) = (600, 300)

P3 = (x3, y3) = (500, 50)

And the screen resolution:

Screen Resolution = 640 x 350

To find the equidistant parametric values, we can divide the range [0, 1] into equal intervals. Let's use six intervals, resulting in seven points (including the endpoints).

Calculate the parametric interval:

Interval = 1 / 6 = 0.1667

**Evaluate the points on the Bezier curve for each parametric value:**

For t = 0:

P0 = (x0, y0) = (50, 180)

For t = 0.1667:

P1 = (1 - t) \* P0 + t \* P1

= (1 - 0.1667) \* (50, 180) + 0.1667 \* (250, 100)

= (41.6665, 170.8335)

For t = 0.3333:

P2 = (1 - t) \* P1 + t \* P2

= (1 - 0.3333) \* (250, 100) + 0.3333 \* (600, 300)

= (283.3332, 206.6667)

For t = 0.5000:

P3 = (1 - t) \* P2 + t \* P3

= (1 - 0.5000) \* (600, 300) + 0.5000 \* (500, 50)

= (550, 175)

For t = 0.6667:

P4 = (1 - t) \* P3 + t \* P4

= (1 - 0.6667) \* (500, 50) + 0.6667 \* (500, 50)

= (500, 50)

For t = 0.8333:

P5 = (1 - t) \* P4 + t \* P5

= (1 - 0.8333) \* (500, 50) + 0.8333 \* (500, 50)

= (500, 50)

For t = 1:

P6 = (x3, y3) = (500, 50)

**Note:** The calculations involve rounding the decimal values to the nearest whole numbers to match the screen resolution.

Therefore, the six points on the Bezier curve with equidistant parametric values are:

P0 = (50, 180)

P1 = (41, 171)

P2 = (283, 207)

P3 = (550, 175)

P4 = (500, 50)

P5 = (500, 50)

P6 = (500, 50)

These points can be plotted on the screen with the given resolution of 640 x 350.

* **What are shading methods? Discuss the Gouraud Shading method.**

Shading methods are techniques used in computer graphics to determine the colors or shades of objects or surfaces in a rendered scene. These methods aim to simulate the interaction of light with surfaces, creating the illusion of depth, texture, and realism.

**Gouraud Shading** is a popular shading method named after Henri Gouraud, who introduced it in 1971. It is an interpolation-based shading technique that computes colors for vertices of polygons and then interpolates those colors across the polygon's surface.

**The Gouraud Shading method works as follows:**

* For each vertex of a polygon, calculate the vertex color based on lighting calculations, material properties, and the position of the light sources. This is typically done using a lighting model, such as the Phong lighting model.
* Interpolate the vertex colors across the polygon's surface using the scanline interpolation technique or another method.
* Assign the interpolated colors to the corresponding pixels of the polygon during the rendering process.

By calculating colors at the vertices and interpolating them across the polygon's surface, Gouraud Shading provides a smooth transition of colors, resulting in a visually pleasing and realistic appearance. This method is particularly useful for low-polygon models and can be computationally efficient compared to other shading methods.

However, Gouraud Shading has some limitations. Since colors are computed at the vertices and interpolated, it may not capture fine details or sharp changes in the surface's appearance accurately. This can result in a lack of precise specular highlights or subtle variations in shading. Additionally, Gouraud Shading may produce artifacts, such as color banding or Mach bands, due to the interpolation process.

Despite these limitations, Gouraud Shading remains widely used in real-time rendering applications, such as video games and interactive graphics, where efficiency is crucial, and a high level of realism is not always required.