USN | | | | | | | | | | |

6th Semester CBCS Scheme Model Question Paper 1

Department of Computer Science and Engineering, C. Byregowda Institute of Technology

Computer Graphics and Visualization

Time: 3 hours Max. Marks: 80

Note: ***Answer FIVE full questions, choosing one full question from each module.***

Module- 1

1. a. Enlist the applications of computer graphics and explain. **(6 marks)**
b. Differentiate between raster scan displays and random scan displays. **(4 marks)**
c. Write a note on midpoint circle algorithm. **(6 marks)**

OR

2. a. Discuss about the video controller and raster scan display processor. **(4 marks)**
b. Explain the concept of Bresenham's line drawing algorithm. **(6 marks)**
c. What are OpenGL line and point functions? Illustrate about the line and point attribute functions. **(6 marks)**

Module- 2

3. a. What are the polygon classifications? How to identify a convex polygon? Illustrate how to split a concave polygon. **(4 marks)**
b. Discuss the steps involved in inside outside tests for a polygon filing. **(6 marks)**
c. List and explain polygon fill area functions and fill area primitives. **(6 marks)**

OR

4. a. Explain the concept of general scan line polygon fill algorithm. **(6 marks)**
b. What is a stitching effect? How does OpenGL deals with it. **(3 marks)**
c. Write a note on 2D rotation, scaling, and translation. Also Discuss the same using a pivot/fixed point. **(7 marks)**

Module- 3

5. a. Write the formulae used in mapping clipping window into a normalized viewport. Use the same and explain the concept of mapping the clipping window into a normalized square and then to screen viewport. **(4 marks)**
b. Consider an example and apply Cohen-Sutherland line clipping algorithm. Explain the steps. **(6 marks)**
c. Explain Sutherland-Hodgeman polygon clipping with an example. **(6 marks)**

OR

6. a. Write a note on 3D translation, rotation, and scaling. **(6 marks)**
b. Explain the 3D reflection and shearing. **(4 marks)**
c. Illustrate about RGB and CMY color models. Write a note on light sources. **(6 marks)**

Module- 4

7. a. Describe a 3D viewing pipeline with necessary diagrams. **(6 marks)**
b. Write a note on parallel and perspective projections. Also explain **(6 marks)**

orthogonal projections in detail.

c. What are vanishing points for perspective projections? **(4 marks)**

OR

8. a. Describe perspective projections with necessary diagrams **(4 marks)**

b. Write a note on oblique and symmetric perspective projection frustum **(8 marks)**

c. List and explain OpenGL 3D viewing functions **(4 marks)**

Module- 5

9. a. Illustrate how an interactive program is animated **(4 marks)**

b. What are quadratic surfaces? List and explain OpenGL Quadratic-Surface and Cubic-Surface Functions **(6 marks)**

c. With necessary codes, explain Bezier Spline Curves

OR

10. a. Represent simple graphics & display processor architectures. Explain the 2 ways of sending graphical entities to a display and list advantages and disadvantages **(6 marks)**

b. Discuss the following logic operations with suitable example. (i) copy mode (ii) Exclusive OR mode (iii) rubber-band effect (iv) drawing erasable lines **(4 marks)**

c. Write a note on design techniques for Bezier curves. **(6 amrks)**

*****

SOLUTIONS FOR MODEL QUESTION PAPER – 1

COMPUTER GRAPHICS & VISUALIZATION – 15CS62

Module -1

1. a. Enlist the applications of computer graphics and explain. (6 marks).

i. Graphs and charts
- Display of simple data graphs was the early application of computer graphics → plotted on a character printer.
- *Data plotting*→ one of the most common graphics applications
- *Graphs and charts*→ summarize financial, statistical, mathematical, scientific, engineering, and economic data for research reports, managerial summaries, consumer information bulletins, etc.
- *3D graphs and charts* are used to display additional parameter information, to provide more dramatic or attractive presentations of data relationships.
- *Time charts and task network layouts* are used in project management to schedule and monitor the progress of the projects.

ii. Computer-Aided Design (CAD)
- Also referred to as CADD (Computer-Aided Drafting and Design).
- Used in design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, home appliances, etc.
- Objects are first displayed in a wire-frame outline that shows the overall shape and internal features of the objects → allows designers to quickly see the effects of interactive adjustments to design shapes without waiting for the object surfaces to be fully generated
- *Animations* → Real-time computer animations using wire-frame shapes are useful for quickly testing the performance of a vehicle or system.
- When object designs are complete, or nearly complete, realistic lighting conditions and surface rendering are applied to show the appearance of the final product.
- *Computer-aided manufacturing (CAM)* → Used to automate the fabrication of a product in manufacturing process
- Architects use interactive computer graphics methods to lay put floor plan
- Architectural CAD packages also provide facilities for experimenting with 3D interior layouts and lightings and also pattern designs in products ***Virtual Reality Environments***

iii. Data visualizations
- Refers to producing graphical representations for
  - Scientific, engineering , and medical data sets and processes → scientific visualization
  - Commerce, industry, and nonscientific areas → business visualization.
- Thousands and millions of values or large sets of numbers generated by processes with some relationships among the values are better understood when converted and represented in a visual form
- A collection of data can contain scalar values, vectors, higher-order tensors, or any combination of these data types. These data sets can be distributed over a 2D region of space, a 3D-region or a high dimensional space.
- Color-coding → to visualize data set

iv. Education and training
- Models of physical processes, physiological functions, population trends, or equipment can help trainees to understand the operation of a system
- For some training applications, special hardware systems are designed.
- Example, simulators for practice session or training of ship captains, aircraft pilots, heavy equipment operators, and air traffic-control personnel

v. Computer art

- Both fine art and commercial art make use of computer graphics methods.
- The picture is painted electronically on a graphics tablet using a stylus that can simulate different brush strokes, brush widths, and colors.
- The stylus translates changing hand pressure into variable line widths, brush sizes, and color gradations.
- Computer-generated animations are also frequently used in producing television commercials.

vi. Entertainment

- Television productions, motion pictures, and music videos routinely use computer-graphic methods.
- Some graphic images are combined with live actors and scenes
- Some films are completely generated using computer-rendering and animation techniques
- CG methods can also be employed to simulate a human actor.
- Graphic object can be combined with the live action, or graphics and image processing techniques can be used to produce a transformation of one person or object into another object (morphing)

vii. Image processing

The modification or interpretation of existing pictures, such as photographs and TV scans, is called *image processing*.

- In CG, a computer is used to create a picture.
- Image processing techniques are used to improve picture quality, analyze images, or recognize visual patterns.
- Image processing is used in CG and CG methods are applied often in image processing.
- Process:
  - A photograph or other picture is digitized into an image file before image-processing methods are employed.
  - Digital methods can be used to rearrange picture parts, to enhance color separations, or to improve the quality of shading.

Image processing in medical applications:

- Picture enhancement in tomography
- A technique of X-ray photography that allows cross-sectional views of physiological systems to be displayed.
- Computer aided surgery
  - 2D cross sections of the body are obtained using imaging techniques.
  - Then the slices are viewed and manipulated using graphics methods to simulate actual surgical procedures and to try out different surgical cuts

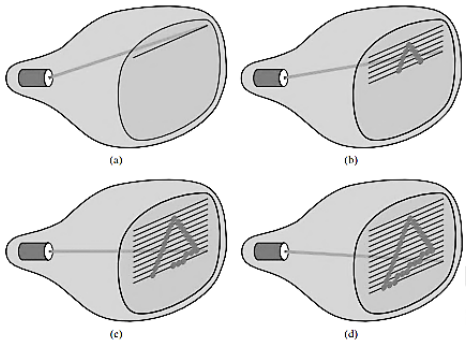
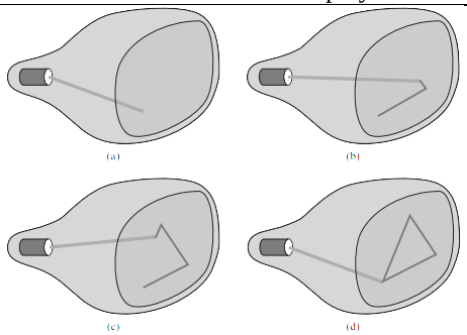
viii. Graphical User Interfaces

Major component- window manager

- Allows a user to display multiple, rectangular screen areas, called display windows.
- Each screen display can contain a different process, showing graphical or non-graphical information, and
- Various methods can be used to activate a display window.
  - Using interactive pointing device like mouse
  - Positioning the cursor within the window display area and pressing left mouse button
  - On other systems, click on the title bar at the top of the display window.
- Interfaces also display menus and icons for
  - Selection of display window
  - A processing option, or
  - A parameter value.

b. Differentiate between raster scan displays and random scan displays. (4 marks)

Raster and random scan display

Raster Scan Display	Random Scan Display
	
The picture is "painted" on the screen one scan line at a time	A CRT has the electron beam directed only to those parts of the screen where a picture is to be displayed
Picture definition is stored in a memory area called the **refresh buffer** or **frame buffer** for the total screen	Picture definition is stored as a set of line-drawing instructions → these have higher resolutions than raster systems
It is possible to view each frame as a smooth continuation of the patterns in the previous frame, so long as the refresh rate is not too low.	Refresh rate→ depends on the number of lines to be displayed on that system
The most common type of graphics monitor employing a CRT based on television technology.	Random-scan systems were designed for line-drawing applications. They cannot display realistic shaded scenes.

c. Write a note on midpoint circle algorithm. (6 marks)

For a given radius r and screen center position (x_c, y_c), calculate pixel positions around a circle path centered at the coordinate origin (0, 0).

Then each calculated position (x, y) is moved to its proper screen position by adding x_c to x and y_c to y.

Along the circle section from x=0 to x=y the slope of the curve varies from 0 to −1.0.

Take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible pixel positions in any column is vertically closer to the circle path

Define a circle function as

$$f_{circ}(x, y)=x^2+y^2-r^2 \quad (20)$$

Where,

$$f_{circ}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \quad (21)$$

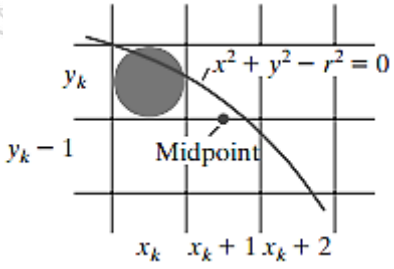


Figure 36: Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path

In Figure 36, (x_k, y_k) has been plotted. Next is to choose between two vertices (x_k+1, y_k) and (x_k+1, y_k-1). The decision parameter for circle function in Eq. 20 evaluated at the midpoint between the two pixels is

$$p_k = f_{\text{circ}}\left(x_k + 1, y_k - \frac{1}{2}\right)$$
$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

(22)

- If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary.
- Otherwise, the midposition is outside or on the circle boundary, and we select the pixel on scan line $y_k - 1$.

Successive decision parameters are obtained using incremental calculations. circle function at sampling position $x_{k+1}+1=x_k+2$:

$$p_{k+1} = f_{\text{circ}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$
$$= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

(23)

Where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of p_k.

Increments for obtaining p_{k+1} are either $2x_{k+1}+1$ (if p_k is negative) or $2x_{k+1}+1-2y_{k+1}$. Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as

$$2x_{k+1}=2x_k+2$$
$$2y_{k+1}=2y_k-2$$

At the start position (0,r), these two terms have the values 0 and 2r, respectively.

The initial decision parameter is obtained by evaluating the circle function at the start position$(x_0, y_0)=(0,r)$:

$$p_0 = f_{\text{circ}}\left(1, r - \frac{1}{2}\right)$$
$$= 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$
$$p_0 = \frac{5}{4} - r$$

(24)

If the radius r is specified as an integer, round p_0 to

$$p_0 = 1 - r \quad \text{(for r an integer)}$$

Algorithm

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), then set the coordinates for the first point on the circumference of a circle centered on the origin as
   $$(x_0, =y_0)=(0,r)$$
2. Calculate the initial value of the decision parameter as
   $$p_0 = 5/4 - r$$
3. At each x_k position, starting at k =0, perform the following test:
   If $p_k<0$, the next point along the circle centered on (0, 0) is (x_k+1, y_k) and
   $$p_{k+1}=p_k + 2x_{k+1}+1$$
   Otherwise, the next point along the circle is (x_k+1, y_k-1) and
   $$p_{k+1}=p_k+2x_{k+1}+1-2y_{k+1}$$
   where $2x_{k+1}=2x_k+2$ and $2y_{k+1}=2y_k-2$.
4. Determine symmetry points in the other seven octants.

5. Move each calculated pixel position (x, y) onto the circular path centered at (x_c,y_c) and plot the coordinate values as follows:

$$x=x+x_c,\ y=y+y_c$$

6. Repeat steps 3 through 5 until $x \geq y$

2. a. Discuss about the video controller and raster scan display processor. (4 marks).

Video Controller

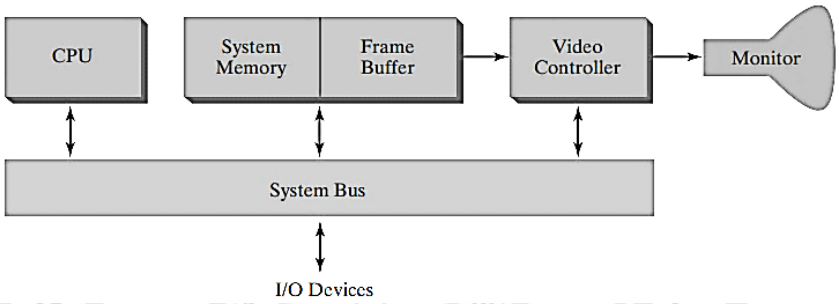


Figure 1: Architecture of a raster system with a fixed portion of the system memory reserved for the frame buffer

A fixed area of the system memory is reserved for the frame buffer, and the video controller is given direct access to the frame-buffer memory. Frame-buffer locations, and the corresponding screen positions, are referenced in Cartesian coordinates.

Commands within a graphics software package are used to set coordinate positions for displayed objects relative to the origin of the Cartesian reference frame.

The coordinate origin is referenced at the lower-left corner of a screen display area by software command (Figure 2).

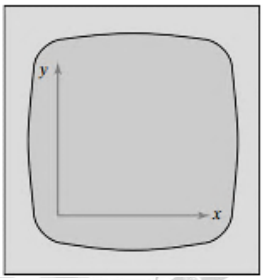


Figure 2: A Cartesian reference frame with origin at the lower-left corner of a video monitor.

The screen surface is then represented as the first quadrant of a two-dimensional system with

- Positive x values increasing from left to right (0 to x_{max})
- Positive y values increasing from the bottom of the screen to the top (0 to y_{max}).

Figure 3 represents the basic refresh operations of the video controller. Two registers are used to store the coordinate values for the screen pixels.

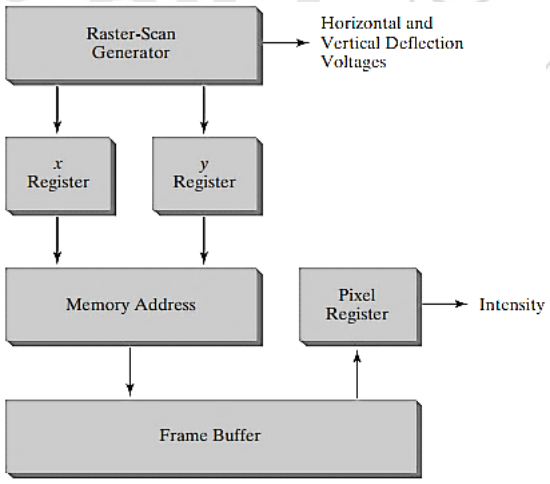


Figure 3: Basic video-controller refresh operations.

- Initially x=0 & y= value for the top scan line.
- Retrieve the contents of the frame buffer at this pixel position and set the intensity of the CRT beam
- Increment x and repeat the process for next pixel.
- After the last pixel on the top scan line has been processed, the x register is reset to 0 and the y register is set to the value for the next scan line down from the top of the screen.
- Repeat procedure for each successive scan line
- To speed up pixel processing, video controllers can retrieve multiple pixel values from the refresh buffer on each pass, store in separate register, and use to control the CRT beam intensity.
- Other operations:
- Retrieve pixel values from different memory areas on different refresh cycles.

Raster Scan Display Processor

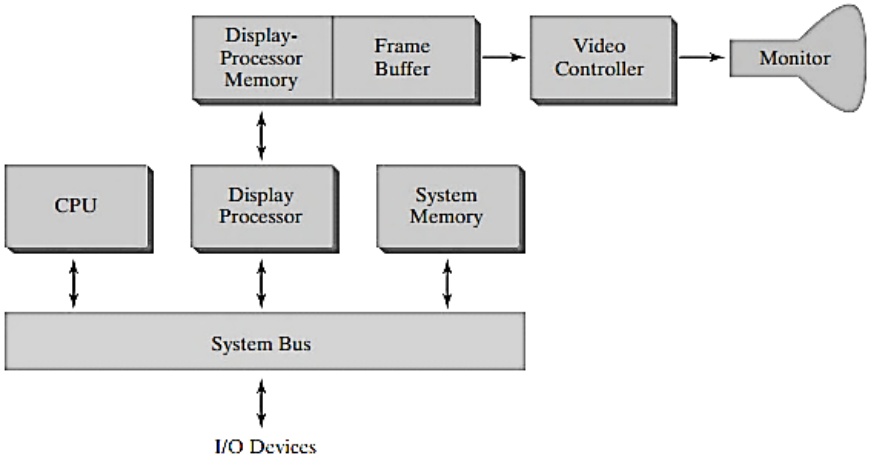


Figure 4: Architecture of a raster-graphics system with a display processor.

Display processor or graphics controller or a display co-processor

- It frees the CPU from the graphics chores.
- Takes a separate memory area.
- Major task→ Scan conversion
  - Digitizing a picture definition given in an application program into a set of pixel values for storage in the frame buffer.
  - Graphics commands specifying objects are scan converted into a set of discrete points, corresponding to screen pixel positions.
  - Example, Scan converting a straight line→ locate the pixel positions closest to the line path and store the color for each position in the frame buffer.
  - Example, representation of characters (Figure 5 & 6).

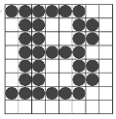


Figure 5: A character defined as a rectangular grid of pixel positions.

Figure 6: A character defined as an outline shape.

Character grid→ Superimposing the rectangular grid pattern into the frame buffer at a specified coordinate position

Outline shape→ Shapes are scan-converted into the frame buffer by locating the pixel positions closest to the outline

- Additional operations of display processor

- Generate various line styles (dashed, dotted, or solid)
- Displaying color areas.
- Applying transformations to the objects in a scene.
- Interface with interactive input devices

b. Explain the concept of Bresenham's line drawing algorithm. (6 marks)

It is an accurate and efficient raster line-generating algorithm, developed by Bresenham, that uses only incremental integer calculations.

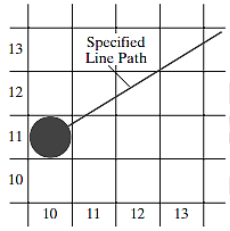


Figure 1: A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11

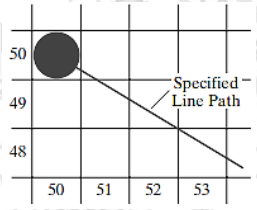


Figure 2: A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

In Figure 1 and 2, the vertical axes show scan-line positions, and the horizontal axes identify pixel columns. Sampling at unit x intervals, we need to decide which of two possible pixel positions is closer to the line path at each sample step.

Starting from the left end point,

- In Figure 28, determine at the next sample position whether to plot the pixel at position (11, 11) or the one at (11, 12)?
- In Figure 2 (-ve slope), select the next pixel position as (51, 50) or as (51, 49)?

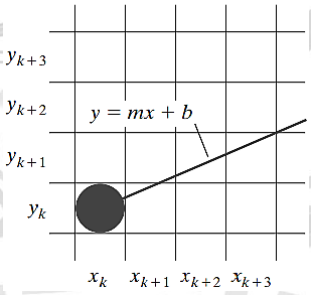


Figure 3: A section of the screen showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope 0<m<1

Consider the scan-conversion process for lines with positive slope less than 1.0.

- Pixel positions along a line path are then determined by sampling at unit x intervals.
- Starting from the left endpoint (x_0, y_0) of a given line, step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.

At k^{th} step in Figure 3, consider the pixel at (x_k,y_k) is displayed. Next pixel to be selected at column $x_{k+1}=x_k+1$ is either (x_{k+1},y_k) or (x_{k+1},y_{k+1}).

- At sampling position x_k+1, label vertical pixel separations from the mathematical line path as d_{lower} and d_{upper} (Figure 4).

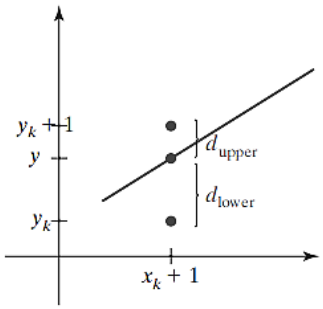


Figure 4: Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$

- The y coordinate on the mathematical line at pixel column position x_k+1 is calculated as

$$y = m(x_k + 1) + b. \quad (10)$$

Then

$$d_{lower} = y - y_k = m(x_k + 1) + b - y_k \quad (11)$$

and

$$d_{upper} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b \quad (12)$$

- To determine which of the two pixels is closest to the line path, set up efficient test that is based on the difference between the two pixel separations as follows:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1 \quad (13)$$

- The decision parameter, p_k, for the k^{th} step in the line algorithm can be obtained from Eq. 13 with $m = \Delta y/\Delta x$.

Where Δy & Δx are vertical and horizontal separations of the endpoint positions. The decision parameter is

$$p_k = \Delta x(d_{lower} - d_{upper}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \quad (14)$$

The sign of p_k is the same as the sign of $d_{lower} - d_{upper}$, because $\Delta x > 0$. Here, Parameter c is constant and has the value $2\Delta y + \Delta x(2b-1)$ and is eliminated in the recursive calculations for p_k.

If the pixel at y_k is "closer" to the line path than the pixel at $y_k + 1$ (that is, $d_{lower} < d_{upper}$), then decision parameter p_k is negative. In that case, plot the lower pixel; otherwise, plot the upper pixel.

At step $k+1$, the decision parameter is evaluated from Equation 14 as

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

Subtracting Equation 14 from the preceding equation

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

$x_{k+1} = x_k + 1$

Therefore,

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \quad (15)$$

Where, the term $y_{k+1} - y_k$ is either 0 or 1, depending on the sign of parameter p_k.

The first parameter, p0, is evaluated from Equation 14 at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y/\Delta x$ as follows:

$$p_0 = 2\Delta y - \Delta x \quad (16)$$

Algorithm

Bresenham's Line-Drawing Algorithm for |m|<1.0

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Set the color for frame-buffer position (x_0, y_0); i.e., plot the first point.
3. Calculate the constants Δx, Δy,$2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at k=0, perform the following test:

If $p_k < 0$, the next point to plot is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise, the next point to plot is (x_{k+1}, y_{k+1}) and

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 $\Delta x - 1$ more times.

c. What are OpenGL line and point functions? Illustrate about the line and point attribute functions. (6 marks)

Line function

Each line segment is defined by two endpoint coordinate positions. In OpenGlm enclose a list of *glVertex* functions between the *glBegin/glEnd* pair. There are **three symbolic constants** in OpenGL that we can use to specify how a list of endpoint positions should be connected to form a set of straight-line segments.

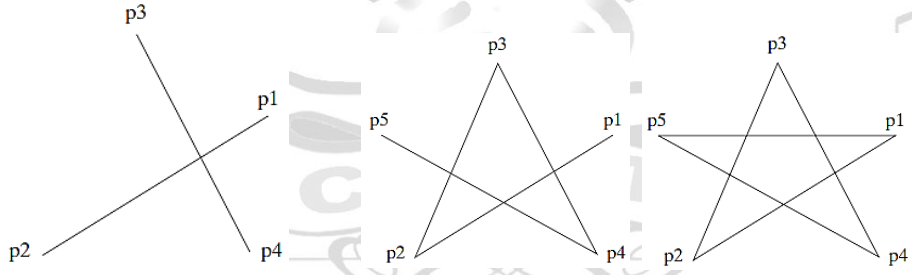
i. GL_LINES → It results in a set of unconnected lines unless some coordinate positions are repeated, because OpenGL considers lines to be connected only if they share a vertex

ii. GL_LINE_STRIP → A **polyline** is obtained. The display is a sequence of connected line segments between the first endpoint in the list and the last endpoint.

iii. GL_LINE_LOOP → Produces a **closed polyline.** Lines are drawn as with GL LINE STRIP, but an additional line is drawn to connect the last coordinate position and the first coordinate position

Example,

```
glBegin (GL_LINES); // glBegin (GL_LINE_STRIP) or glBegin(GL_LINE_LOOP)
 glVertex2iv (p1);
 glVertex2iv (p2);
 glVertex2iv (p3);
 glVertex2iv (p4);
 glVertex2iv (p5);
 glEnd ( );
```



GL_LINES GL_LINE_STRIP GL_LINE_LOOP

Point functions

Give a coordinate position in the world reference frame to specify the geometry of a point. This coordinate position is passed to the viewing routines.

- OpenGL primitives are displayed with a default size and color.
  - Color → white
  - Size→ size of a single screen pixel
- OpenGL function to state the coordinate values for a single position is:

glVertex* ( );

* → Suffix codes are required for this function

→ Used to identify the spatial dimension,

→ The numerical data type to be used for the coordinate values, and
→ A possible vector form for the coordinate specification.

- Calls to *glVertex* functions must be placed between a *glBegin* function and a *glEnd* function
- The *glVertex* function is used in OpenGL to specify coordinates for any point position
- Example, to plot points in Figure 24,

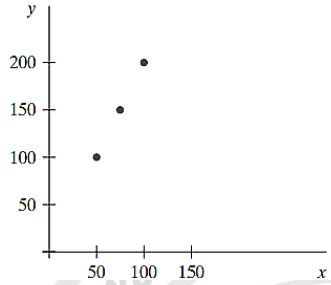


Figure 24: Display of three point positions enerated with glBegin (GL POINTS).

```
glBegin (GL_POINTS);
        glVertex2i (50, 100);
        glVertex2i (75, 150);
        glVertex2i (100, 200);
glEnd ( );
```

Point attributes functions

- Set the size for an OpenGL point with

  glPointSize (size);

  the point is then displayed as a square block of pixels.
  size→ positive floating-point value, which is rounded to an integer. Default size ia 1.0.
- The number of horizontal and vertical pixels in the display of the point is determined by parameter size.
  - Size =1 → 1 pixel
  - size = 2 → 2×2 pixel
- Attribute functions may be listed inside or outside of a glBegin/glEnd pair
- Example,

```
glColor3f (1.0, 0.0, 0.0);
glBegin (GL_POINTS);
        glVertex2i (50, 100);
        glPointSize (2.0);
        glColor3f (0.0, 1.0, 0.0);
        glVertex2i (75, 150);
        glPointSize (3.0);
        glColor3f (0.0, 0.0, 1.0);
        glVertex2i (100, 200);
glEnd ( )
```

The first is a standard-size red point, the second is a double-size green point, and the third is a triple-size blue point

Line attributes functions

OpenGL Line-Width Function

Line width is set in OpenGL with the function

glLineWidth (width);

width→ floating-point value assigned is rounded to the nearest nonnegative integer.
→ default→ 1.0
→ if 0.0 is given it is rounded to 1.0.

Antialiasing makes line smoother

OpenGL Line-Style Function

By default, a straight-line segment is displayed as a solid line. Styles can be

- dashed lines,
- dotted lines, or
- a line with a combination of dashes and
- dots, and
- vary the length of the dashes and the spacing between dashes or dots

OpenGL function is

glLineStipple (repeatFactor, pattern);

pattern→ reference a 16-bit integer that describes how the line should be displayed.

→ *1 bit = "on" pixel*

→ 0 bit = "off" pixel

The default pattern is 0xFFFF (each bit position has a value of 1), which produces a solid line.

repeatFactor→ how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied.

→ default value=1

Activate the line-style feature of OpenGL

glEnable (GL_LINE_STIPPLE);

Turn off the line-pattern feature with

glDisable (GL_LINE_STIPPLE);

This replaces the current line-style pattern with the default pattern (solid lines).

Module -2

3. a. What are the polygon classifications? How to identify a concave polygon? Illustrate how to split a concave polygon (4 marks).

Polygon Classifications

An ***interior angle*** of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges. There are two types of polygon surfaces are (Figure 3).

- If all interior angles < 180^0 then the polygon is ***convex***.
  - Its interior lies completely on one side of an infinite extension line of any one of its edges.
  - A line segment joining the two points is also in the interior
- A polygon that is not convex is called ***concave polygon***.

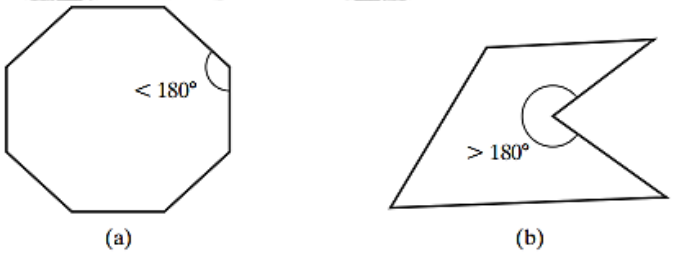


Figure 3: A convex polygon (a), and a concave polygon (b).

- ***Degenerate polygon***
  - Set of vertices that are
    - Collinear
      - Generate a line segment
    - Repeated coordinate positions
      - Generate polygon shapes with
        - Extraneous lines
        - Overlapping edges, or
        - Edges with length = 0.
- Problems with concave polygons:
  - Implementing fill algorithms and other algorithms is complicated

- Solution→ split the polygon into set of convex polygons.

Identifying Concave Polygons

Characteristics of concave polygon:

i. It has at least one interior angle $>180^0$.

ii. Extension of some edges will intersect other edges

iii. Some pair of interior points forms a line that intersects the polygon boundary.

To test a concavity

- Set up vector for each polygon edge,
- Compute cross product of adjacent edges.
  - If the sign of all vector products is +ve (or all –ve) → convex
  - If some are +ve and some are –ve → concave.
- Consider Polygon vertex positions relative to the extension of any line of any edge. A polygon is concave if some vertices are one side of extensions line or others are inside the polygon.

Splitting Concave Polygon

A concave polygon can be split into set of convex polygons.

i. Use edge vectors & edge cross products, or

ii. Use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other.

Consider that all the polygons are in *xy* plane. With the *vector method* for splitting a concave polygon, form the edge vectors.

Given,

Two consecutive vertex positions, V_k and V_{k+1}, the edge vector between these two vertices is given by,

$$E_k = V_{k+1} - V_k$$

Calculate the cross-products of successive edge vectors in order around the polygon perimeter, assuming that no series of three successive vertices are collinear as their cross product will be zero.

- If the z component of some cross-products is positive while other cross-products have a negative z component, the polygon is concave.
- Otherwise, the polygon is convex.

Apply the vector method by processing edge vectors in counterclockwise order. Consider figure 4.

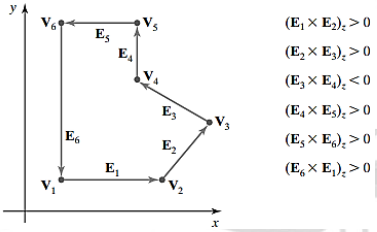


Figure 4: Identifying a concave polygon by calculating cross-products of successive pairs of edge vectors

- When cross-product has a negative z component, the polygon is concave
- Split it along the line of the first edge vector in the cross-product pair.

Rotational Method (Figure 6)

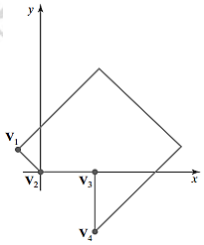


Figure 6: Splitting a concave polygon using the rotational method. After moving V_2 to the coordinate origin and rotating V_3 onto the x axis, we find that V_4 is below the x axis. So we split the polygon along the line of V_2V_3, which is the x axis.

i. Proceeding counterclockwise around the polygon edges, shift the position of the polygon so that each vertex V_k in turn is at the coordinate origin.
ii. Rotate the polygon about the origin in a clockwise direction so that the next vertex V_{k+1} is on the x axis.
iii. If the following vertex, V_{k+2}, is below the x axis, the polygon is concave.
iv. Split the polygon along the x axis to form two new polygons.
v. Repeat the concave test for each of the two new polygons until all vertices in the polygon list are tested.

Splitting a Convex Polygon into a Set of Triangles

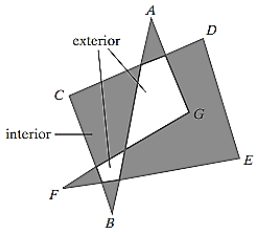
With vertex list for a convex polygon, transform it into a set of triangles.

i. Define any sequence of three consecutive vertices to be a new polygon (a triangle)
ii. Delete the middle triangle vertex from the original vertex list.
iii. Apply same steps to the modified vertex list, until original polygon is reduced to just three vertices → last triangle in the set.

b. Discuss the steps involved in inside outside tests for a polygon filing. (6 marks)

Inside-outside tests

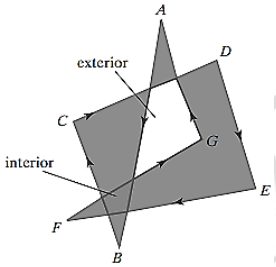
Odd-Even Rule (odd parity or even-odd rule)



Odd-Even Rule

i. Draw a line from any position P to a distant point outside the coordinate extents of the closed polyline.
ii. Count the number of line-segment crossings along this line.
    - If the number of segments crossed by this line is ***odd***, then P is considered to be an ***interior*** point.
    - If the number of segments crossed by this line is ***even***, then P is considered to be an ***exterior*** point.

Nonzero winding-number rule



Nonzero Winding-Number Rule

Logic→ Count the number of times that the boundary of an object "winds" around a particular point in the counter-clockwise direction→ ***winding number***

Procedure:

i. Initialize the winding number to 0
ii. Draw a line from any position P to a distant point beyond the coordinate extents of the object. The line must not pass through any endpoint coordinates
iii. Count the number of object line segments that cross the reference line in each direction, along the line from position P to the distant point.
    a. Add 1 to the winding number every time we intersect a segment that crosses the line in the direction from right to left
    b. Subtract 1 every time we intersect a segment that crosses from left to right

iv. The final value of the winding number determines the relative position of P
   a. If winding number is nonzero value→ P is interior
   b. If winding number is zero→ P is exterior

Determining directional boundary crossings

Approach 1: use vector cross products

- Set up vectors along the object edges (or boundary lines) and along the reference line.
- Compute the vector cross-product of the vector *u*, along the line from P to a distant point, with an object edge vector E for each edge that crosses the line.
- With object in xy plane, the direction of each vector cross-product will be either in the +z direction or in the −z direction.
  - If the z component of a cross-product u×E for a particular crossing is positive→ segment crosses from right to left → add 1 to the winding number.
  - If the z component of a cross-product u×E for a particular crossing is negative→ segment crosses from left to right → subtract 1 from the winding number.

Approach 2: use vector dot products instead of cross-products.

- Set up a vector that is perpendicular to vector u and that has a right-to-left direction as we look along the line from P in the direction of u.
- If the components of u are denoted as (u_x, u_y), then the vector that is perpendicular to u has components $(-u_y, u_x)$.
  - If the dot product of perpendicular vector and a boundary-line vector is positive→ segment crosses from right to left → add 1 to the winding number.

If the dot product of perpendicular vector and a boundary-line vector is negative→ segment crosses from left to right → subtract 1 from the winding number

c. List and explain polygon fill area functions and fill area primitives. (6 marks)

OpenGL procedures for specifying fill polygons are similar to those for describing a point or a polyline.

- A *glVertex* function is used to input the coordinates for a single polygon vertex
- Complete polygon is described with a list of vertices placed between a *glBegin/glEnd* pair.
- A polygon interior is displayed in a solid color, determined by the current color settings. Or optionally we can fill it with a pattern or display only outline
- There are six different symbolic constants used as the argument in the glBegin function to describe polygon fill areas → to display a single fill polygon, a set of unconnected fill polygons, or a set of connected fill polygons.

In OpenGL,

- Fill area→ convex polygon
- Vertex list→ at least 3 vertices
- No crossing edges
- All interior angles for the polygon must be less than 180°.

Each polygon has two faces: a ***back face*** and a ***front face***.

- Fill color and other attributes can be set for each face separately
- Back/front identification is needed in both 2D and 3D viewing routines.
- Specify polygon vertices in a counterclockwise order (View→ from "outside".

OpenGL provides a special ***rectangle function*** that directly accepts vertex specifications in the xy plane.

glRect* (x1, y1, x2, y2);

where, one corner of this rectangle is at coordinate position (x1,y1), and the opposite corner of the rectangle is at position (x2,y2).

Suffix codes→ data type and array elements expression

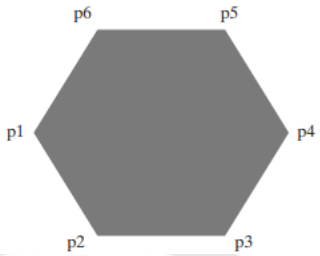
→ i (for integer), s (for short), f (for float), d (for double), and v (for vector).

Polygon fill area primitives

Specified with a symbolic constant in the glBegin function, along with a a list of glVertex commands.

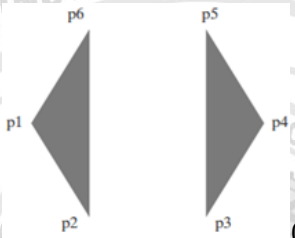
i. GL_POLYGON

Consider a list of 6 points, p1 to p6 specifying 2D polygon vertex positions in counterclockwise order.



A polygon vertex list must contain at least three vertices. Otherwise, nothing is displayed

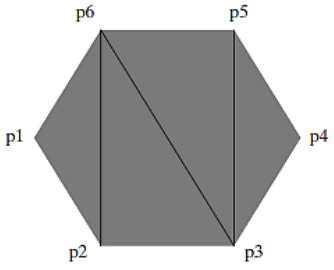
ii. GL_TRIANGLES



(P1,P2,P6,P3,P4,P5)

- The first three coordinate points define the vertices for one triangle, the next three points define the next triangle, and so forth.
- Vertex positions are defined in a counterclockwise order.
- A set of unconnected triangles is displayed with this primitive constant unless some vertex coordinates are repeated
- Nothing is displayed if we do not list at least three vertices.

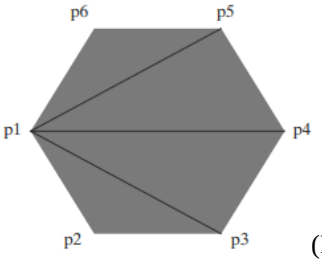
iii. GL_TRIANGLE_STRIP



(P1,P2,P6,P3,P5,P4)

- Assuming that no coordinate positions are repeated in a list of N vertices, it results into N-2 triangles. Where N≥3.
  For N=6→ 4 triangles are obtained.
- Each successive triangle shares an edge with the previously defined triangle.
- One triangle is defined for each vertex position listed after the first two vertices.
- Processing order→ n=1, n=2, …n=N-2.
- Arranging the order of the corresponding set of three vertices according to whether n is an odd number or an even number.
  - If n is odd, the polygon table listing for the triangle vertices is in the order n, n+1, n+2.
  - If n is even, the triangle vertices are listed in the order n+1, n, n+2

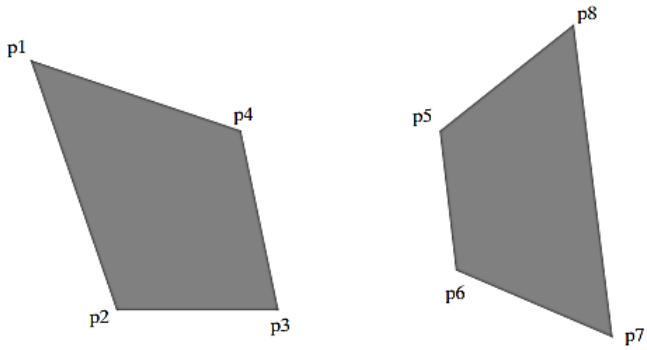
iv. GL_TRIANGLE_FAN



(P1,P2,P3,P4,P5,P6)

- For N vertices, N-2 triangles are generated, providing no vertex positions are repeated, and at least three vertices are listed.
- The first coordinate position listed (in this case, p1) is a vertex for each triangle in the fan.
- If triangles and the coordinate positions are listed as n=1, n=2, ..., n=N−2, the vertices for triangle n are listed in the polygon tables in the order 1,n+1,n+2.

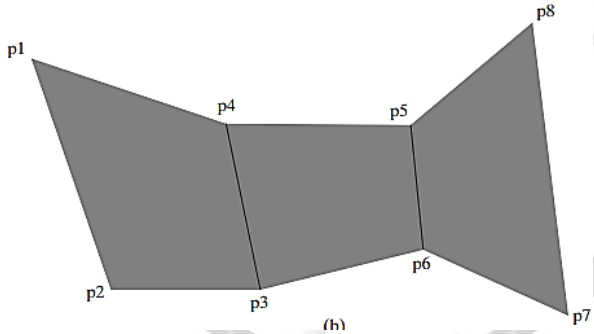
v. GL_QUANDS



(P1,P2,P3,P4,P5,P6,P7,P8)

- The first four coordinate points define the vertices for one quadrilateral, the next four points define the next quadrilateral, and so on.
- Specify the vertex positions in a counterclockwise order. List at least 4 vertices. if the number of vertices specified is not a multiple of 4, the extra vertex positions are ignored
- If no vertex coordinates are repeated, a set of unconnected four-sided fill areas is displayed

vi. GL_QUAD_STRIP



(P1,P2,P4,P3,P5,P6,P8,P7)

- For N vertices, (N/2) −1 quadrilaterals are obtained provided N≥4.
- If N is not a multiple of 4, any extra coordinate positions in the list are not used.
- Quads generated in sequence, n=1,n=2, ...,n=(N/2) −1
- Polygon tables will list the vertices for quadrilateral n in the vertex order number 2n−1, 2n,2n+2, 2n+1

4. a. Explain the concept of general scan line polygon fill algorithm. (6 marks).

- Determine the intersection positions of the boundaries of the fill region with the screen scan line.
- Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region.
- The scan-line fill algorithm identifies the same interior regions as the odd-even rule
- The simplest area to fill→ polygon
  Reason→Each scan-line intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations.
  Scan line equation→ y = constant.

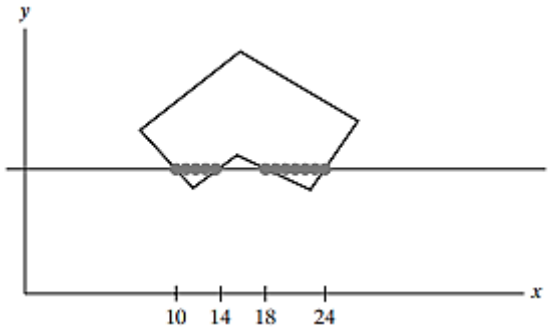


Figure 17: Interior pixels along a scan line passing through a polygon fill area.

Consider, Figure 17,

- It represents the basic scan-line procedure for a solid-color fill of a polygon.
  - For each scan line that crosses the polygon, the edge intersections are sorted from left to right
  - The pixel positions between, and including, each intersection pair are set to the specified fill color
  - In figure, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels.
    - Fill color is applied to the five pixels from x=10 to x=14 and to the seven pixels from x=18 to x=24.

Scan-line fill algorithm for a polygon

- Whenever a scan line passes through a vertex, it intersects two polygon edges at that point. → odd number of boundary intersections for a scan line (in some cases).
- Consider Figure 18.

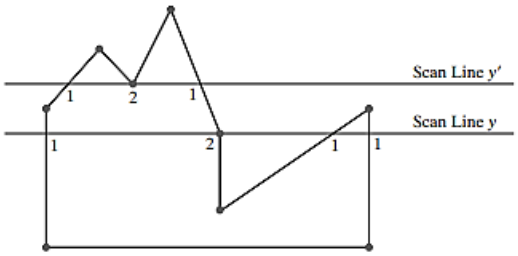


Figure 18: Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

- Scan line y' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.
- Scan line y intersects five polygon edges.
- To identify the interior pixels for scan line y, count the vertex intersection as only one point.
- The position of the intersecting edges relative to the scan line gives the topological difference between scan line y and scan line y'.
  - For scan line y, the two edges sharing an intersection vertex are on opposite sides of the scan line.
  - For scan line y', the two intersecting edges are both above the scan line.
  - A vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point.

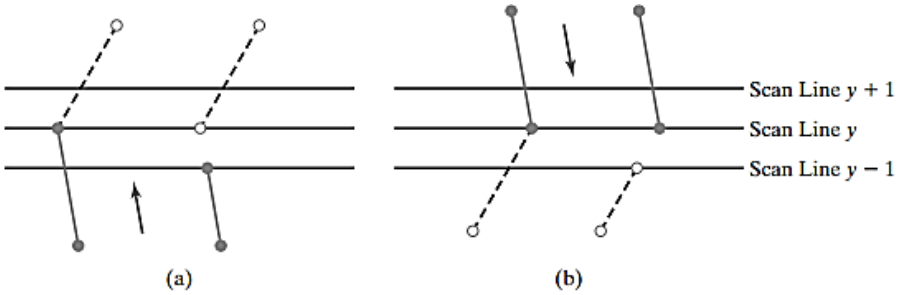


Figure a1: Adjusting end point y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

Certain properties of one part of a scene are related in some way to the properties in other parts of the scene→ ***coherence properties***.

Coherence methods involve incremental calculations applied along a single scan line or between successive scan lines.

Method:

Figure a2 shows two successive scan lines crossing the left edge of a triangle.

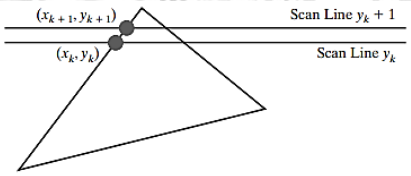


Figure a2: Two successive scan lines intersecting a polygon boundary.

The slope of this edge can be expressed in terms of the scan-line intersection coordinates:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \quad \ldots(\text{eq } 1)$$

Because the change in y coordinates between the two scan lines is simply

$$y_{k+1} - y_k = 1 \quad \ldots(\text{eq } 2)$$

The x-intersection value x_{k+1} on the upper scan line can be determined from the x-intersection value x_k on the preceding scan line as

$$x_{k+1} = x_k + \frac{1}{m} \quad \ldots(\text{eq } 3)$$

Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

Parallel implementation of the fill algorithm

→ assign each scan line that crosses the polygon to a separate processor.

→ Edge intersection calculations are then performed independently.

Along an edge with slope m, the intersection x_k value for scan line k above the initial scan line can be calculated as

$$x_k = x_0 + \frac{k}{m} \quad \ldots(\text{eq } 4)$$

In a sequential fill algorithm, the increment of x values by the amount 1/m along an edge can be accomplished with integer operations by recalling that the slope m is the ratio of two integers:

$$m = \frac{\Delta y}{\Delta x}$$

Where Δx and Δyare the differences between the edge end point x and y coordinate values. Thus, incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \quad \ldots(\text{eq } 5)$$

To perform a polygon fill efficiently, we can first store the polygon boundary in a *sorted edge table* that contains all the information necessary to process the scan lines efficiently.

- Use a bucket sort to store the edges, sorted on the smallest y value of each edge, in the correct scan-line positions
- Only non-horizontal edges are entered into the sorted edge table
- Shorten certain edges to resolve the vertex-intersection question
- Each entry in the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge.
- For each scan line, the edges are in sorted order from left to right. Polygon and the associated sorted edge table is shown in figure Figure a3.

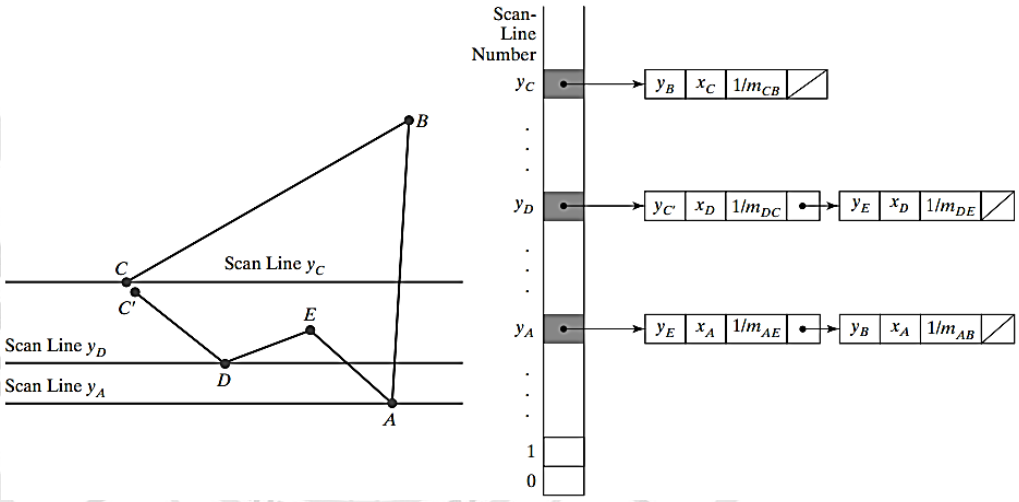


Figure a3: A polygon and its sorted edge table, with edge $\overline{DC}$ shortened by one unit in the y direction.

- Process the scan lines from the bottom of the polygon to its top, producing an active edge list for each scan line crossing the polygon boundaries.

The active edge list for a scan line contains all edges crossed by that scan line, with iterative coherence calculations used to obtain the edge intersections

b. What is a stitching effect? How does OpenGL deal with it? (3 marks)

For a three-dimensional polygon (one that does not have all vertices in the xy plane), this method for displaying the edges of a filled polygon may produce gaps along the edges. This effect, sometimes referred to as ***stitching*** ,is caused by differences between calculations in the scan-line fill algorithm and calculations in the edge line-drawing algorithm.

As the interior of a three-dimensional polygon is filled, the depth value (distance from the xy plane) is calculated for each (x, y) position.

Eliminate the gaps along displayed edges of a three-dimensional polygon

i. Shift the depth values calculated by the fill routine so that they do not overlap with the edge depth values for that polygon.

```
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (factor1, factor2);
```

The calculation for this depth offset is

$$depthOffset = factor1 \cdot maxSlope + factor2 \cdot const$$

As an example of assigning values to offset factors, we can modify the previous code segment as follows:

```
glColor3f (0.0, 1.0, 0.0);
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
/* Invoke polygon-generating routine. */
```

```
glDisable (GL_POLYGON_OFFSET_FILL);
glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */
```

ii. Use the OpenGL stencil buffer to limit the polygon interior filling so that it does not overlap the edges.

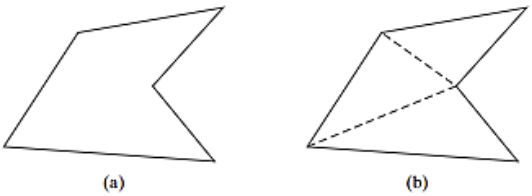


Figure 21: Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

To display a concave polygon using OpenGL routines

- Split it into a set of convex polygons.
- Divide a concave polygon into a set of triangles
- Display the triangles.

To show only polygon vertices,

- Plot the triangle vertices

To display the original polygon in a wire-frame form if GL-LINE is used show all the triangle edges that are interior to the original concave polygon.

Set that bit flag to "off" and the edge following that vertex will not be displayed. We set this flag for an edge with the following function:

```
glEdgeFlag (flag);
```

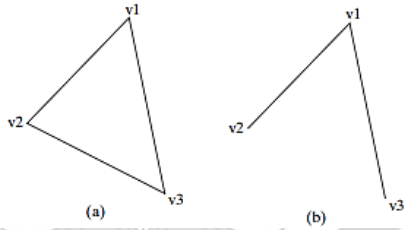


Figure 21: The triangle in (a) can be displayed as in (b) by setting the edge flag for vertex v2 to the value GL_FALSE, assuming that the vertices are specified in a counterclockwise order.

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
glBegin (GL_POLYGON);
    glVertex3fv (v1);
    glEdgeFlag (GL_FALSE);
    glVertex3fv (v2);
    glEdgeFlag (GL_TRUE);
    glVertex3fv (v3);
glEnd ( );
```

The statements for creating an array of edge flags are

```
glEnableClientState (GL_EDGE_FLAG_ARRAY);
glEdgeFlagPointer (offset, edgeFlagArray);
```

c. Write a note on 2D rotation, scaling, and translation. Also Discuss the same using a pivot/fixed point. (7 marks)

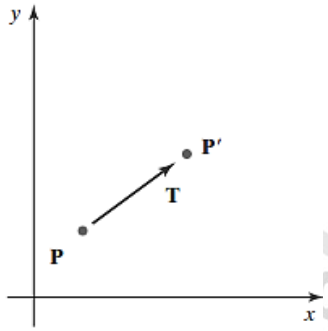
coordinates to generate a new coordinate position → Moves the original point position along a straight-line path to its new location.

It is applied to **an object** that is defined with multiple coordinate positions (e.g., quadrilateral) by relocating all the coordinate positions by the same displacement along parallel paths→ Complete object is displayed at the new location.

Translation distances→ added to translate a two-dimensional position.

Consider distances t_x and t_y are added to the original coordinates (x, y) to obtain the new coordinate position (x', y') as shown in Figure below.

$$x' = x + t_x \qquad y' = y + t_y \quad \text{...(1)}$$



The translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector.**

Equations 1 can be expressed as a single matrix equation by using the following column vectors to represent coordinate positions and the translation vector:

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{P'} = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad \text{...(2)}$$

This allows us to write the two-dimensional translation equations in the matrix form

$$P' = P + T \quad \text{...(3)}$$

Rotation transformation of an object→ Obtained by specifying a ***rotation axis*** and a ***rotation angle***.

→ This transforms all points to new positions by rotating the points through the specified angle about the rotation axis

2D Transformation:

- Rotating the object about a rotation axis that is perpendicular to the *xy* plane (parallel to the coordinate *z* axis)
- Reposition the object along a circular path in the *xy* plane.
- Parameters→ rotation angle θ and a position (x_r, y_r), called the **rotation point** (or **pivot point**), about which the object is to be rotated (Figure 24).
    - The pivot point is the intersection position of the rotation axis with the *xy* plane.

Positive value for the angle θ defines a counterclockwise rotation about the pivot point and a negative value rotates objects in the clockwise direction.

Two-Dimensional Scaling

- To alter the size of an object
- Procedure→ multiplying object positions (x, y) by scaling factors s_x and s_y to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y \quad \text{...(10)}$$

    - s_x→ scales an object in the x direction
    - s_y→ scales an object in the y direction
- The basic two-dimensional scaling equations 10 can also be written in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{... (11)}$$

Or

$$P' = S \cdot P \quad \text{...(12)}$$

Where S is the 2×2 scaling matrix in Equation 11. Scaling factor can be any positive value.

- Values < ***1*** reduce the size of objects;

- Values > ***1*** produce enlargements
- Values = ***1*** leaves the size of objects unchanged

When $s_x == s_y$ → ***uniform scaling*** is produced.

When $s_x != s_y$ → ***differential scaling*** is produced

General Two-Dimensional Pivot-Point Rotation

When a graphics package provides only a rotate function with respect to the coordinate origin, 2D rotation can be generated about any other pivot point (x_r, y_r) by performing the following sequence of translate-rotate-translate operations:

i. Translate the object so that the pivot-point position is moved to the coordinate origin.

ii. Rotate the object about the coordinate origin.

iii. Translate the object so that the pivot point is returned to its original position

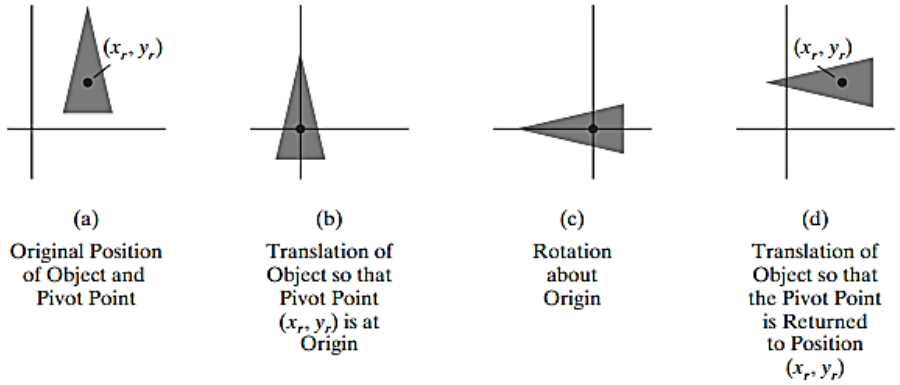


Figure 1: A transformation sequence for rotating an object about a specified pivot point using the rotation matrix R(θ)of transformation.

This transformation sequence is illustrated in Figure 1. The composite trans-formation matrix for this sequence is obtained with the concatenation

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(1)$$

which can be expressed in the form

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta) \quad \ldots(2)$$

Where $T(-x_r,-y_r) = T^{-1}(x_r, y_r)$.

vii. General Two-Dimensional Fixed-Point Scaling

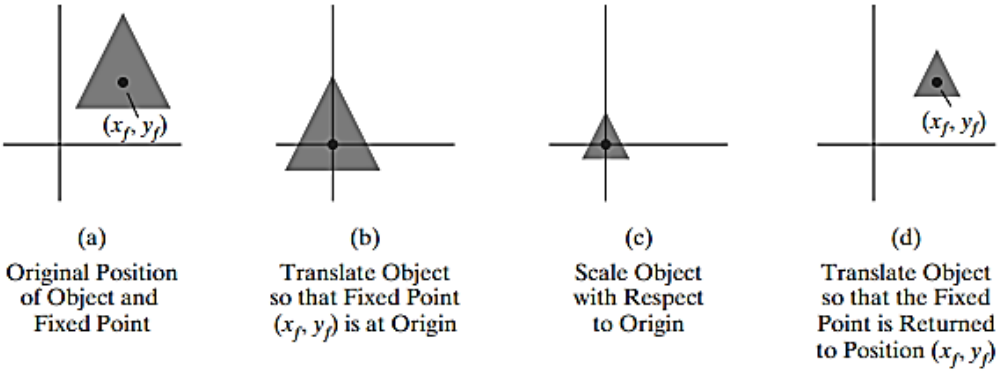


Figure 2: A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix $S(s_x,s_y)$ of transformation 21.

Figure 2 → Transformation sequence to produce a 2D scaling with respect to a selected fixed position (x_f, y_f) with the function that can scale relative to the coordinate origin only.

i. Translate the object so that the fixed point coincides with the coordinate origin.
ii. Scale the object with respect to the coordinate origin.
iii. Use the inverse of the translation in step (1) to return the object to its original position.

Concatenating the matrices for these three operations produces the required scaling matrix

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(3)$$

Or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y) \quad \ldots(4)$$

This transformation is generated automatically in systems that provide a scale function that accepts coordinates for the fixed point.

Module -3

5. a. Write the formulae used in mapping clipping window into a normalized viewport. Use the same and explain the concept of mapping the clipping window into a normalized square and then to screen viewport. (4 marks).

The formulae used in mapping clipping window into a normalized viewport

$$\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} = \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}}$$

$$\frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} = \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}} \quad \ldots(2)$$

Solving these expressions for the viewport position (x_v, y_v), we have

$$xv = s_x xw + t_x$$
$$yv = s_y yw + t_y \quad \ldots(3)$$

where the scaling factors are

$$s_x = \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}}$$

$$s_y = \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} \quad \ldots(4)$$

and the translation factors are

$$t_x = \frac{xw_{\max} xv_{\min} - xw_{\min} xv_{\max}}{xw_{\max} - xw_{\min}}$$

$$t_y = \frac{yw_{\max} yv_{\min} - yw_{\min} yv_{\max}}{yw_{\max} - yw_{\min}} \quad \ldots(5)$$

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1-s_x) \\ 0 & s_y & yw_{\min}(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(6)$$

Where s_x and s_y are the same as in Equations 4.

The 2D matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - xw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(7)$$

And the composite matrix representation for the transformation to the normalized viewport is

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(8)$$

Mapping the Clipping Window into a Normalized Square

Transform the clipping window into a normalized square, clip in normalized coordinates, and then transfer the scene description to a viewport specified in screen coordinates (Figure5).

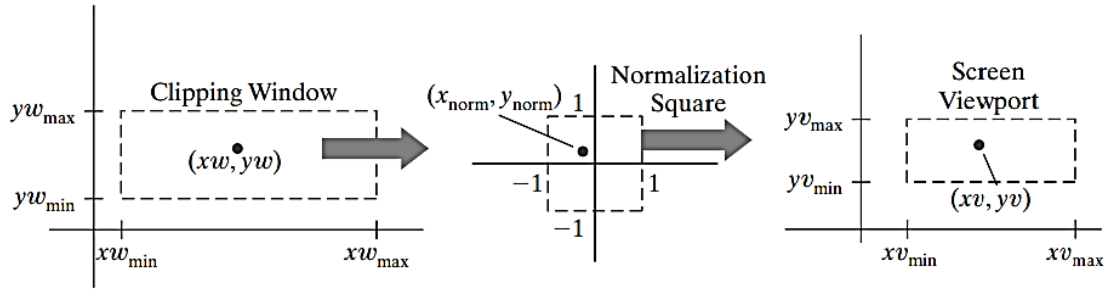


Figure 5: A point (x_w, y_w) in a clipping window is mapped to a normalized coordinate position (x_{norm},y_{norm}), then to a screen-coordinate position (x_v, y_v) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates occurs.

Normalized coordinates'range→ -1 to 1

Clipping algorithms→ objects outside the boundaries x=±1 and y=±1 are detected and removed from the scene description.

Final step→ the objects in the viewport are positioned within the display window.

Procedure,

i. Transfer the contents of the clipping window into the normalization square

ii. The matrix for the normalization transformation is obtained from Equation 8 by substituting −1 for xv_{min} and yv_{min} and substituting +1 for xv_{max} and yv_{max}.

iii. After the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport

iv. Transformation matrix from Equation 8 by substituting −1 for xw_{min} and yw_{min} and substituting +1 for xw_{max} and yw_{max}:

$$\mathbf{M}_{\text{normsquare, viewport}} = \begin{bmatrix} \frac{xv_{max} - xv_{min}}{2} & 0 & \frac{xv_{max} + xv_{min}}{2} \\ 0 & \frac{yv_{max} - yv_{min}}{2} & \frac{yv_{max} + yv_{min}}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad \ldots(9)$$

v. Position the viewport area in the display window. The lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window (Figure 8).

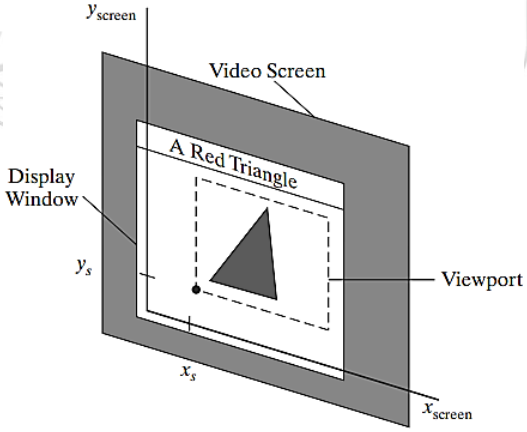


Figure 6: A viewport at coordinate position (x_s, y_s) within a display window

b. Consider an example and apply Cohen-Sutherland line clipping algorithm. Explain the steps. (6 marks)

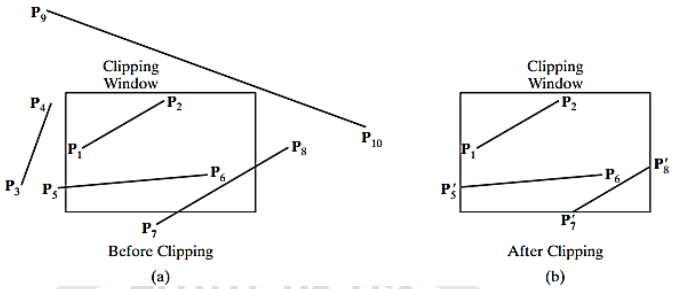


Figure 7: Clipping straight-line segments using a standard rectangular clipping window.

To formulate the equation for a straight-line segment use the following parametric representation where the coordinate positions (x_0, y_0) and (x_{end}, y_{end}) designate the two line endpoints:

$$x = x_0 + u(x_{end} - x_0)$$
$$y = y_0 + u(y_{end} - y_0) \qquad 0 \le u \le 1 \quad \ldots(11)$$

Equation 11 is used to determine where a line segment crosses each clipping-window edge by assigning the coordinate value for that edge to either x or y and solving for parameter u.

For example, the left window boundary is at position xw_{min}, so we substitute this value for x, solve for u, and calculate the corresponding y-intersection value.

- If the value of u is outside the range from 0 to 1, the line segment does not intersect that window border line.
- If the value of u is within the range from 0 to 1, part of the line is inside that border.
    - Process this inside portion of the line segment against the other clipping boundaries until either we have clipped the entire line or we find a section that is inside the window.

Cohen-Sutherland Line Clipping

- Fast line clipping
- Processing time is reduced in the Cohen-Sutherland method by performing more tests before proceeding to the intersection calculations
- Every line endpoint in a picture is assigned a ***four-digit binary value***, called a ***region code ("out"code)***, and each bit position is used to indicate whether the point is inside or outside one of the clipping-window boundaries.
- Window edges can be referenced in any order.

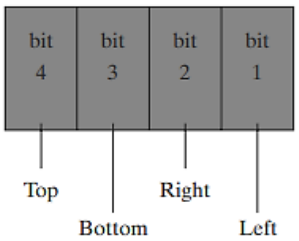


Figure 8: A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

- Figure 8 illustrates one possible ordering with the bit positions numbered 1 through 4 from right to left.
    - The rightmost position (bit 1) references the left clipping-window boundary,
    - The leftmost position (bit 4) references the top window boundary.
    - A value of 1 (or true) in any bit position indicates that the endpoint is outside that window border
    - A value of 0 (or false) in any bit position indicates that the endpoint is not outside (it is inside or on) the corresponding window edge.

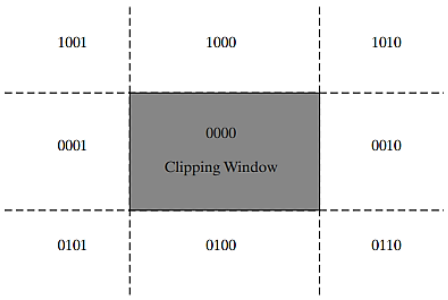


Figure 9: The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

- Each clipping-window edge divides two-dimensional space into an inside half space and an outside half space.
- Together, the four window borders create nine regions (Figure 9). Example, an endpoint that is below and to the left of the clipping window is assigned the region code 0101,

- Procedure:
  - Bit values in a region code are determined by comparing the coordinate values (x, y) of an endpoint to the clipping boundaries.
  - Bit 1 is set to 1 if $x<x_{wmin}$, and the other three bit values are determined similarly.
  - Determine the values for a region-code more efficiently using bit-processing operations and the following two steps
    - Calculate differences between endpoint coordinates and clipping boundaries.
    - Use the resultant sign bit of each difference calculation to set the corresponding value in the region code
      - Bit 1 is the sign bit of $x-xw_{min}$;
      - Bit 2 is the sign bit of $xw_{max}-x$;
      - Bit 3 is the sign bit of $y-yw_{min}$; and
      - Bit 4 is the sign bit of $yw_{max}-y$.
  - Determine which lines are completely inside the clip window and which are completely outside.
    - Any lines that are completely contained within the window edges have a region code of 0000 for both endpoints→ save these line segments
    - Any line that has a region-code value of 1 in the same bit position for each endpoint is completely outside the clipping rectangle→ eliminate that line segment
  - Inside-outside tests for line segments using logical operators.
    - When the ***or*** operation between two endpoint region codes for a line segment is false (0000), the line is inside the clipping window→ save the line
    - When the ***and*** operation between the two endpoint region codes for a line is true (not 0000), the line is completely outside the clipping window→ eliminate that line segment
  - Lines that cannot be identified as being completely inside or completely outside a clipping window by the region-code tests are next checked for ***intersection with the window border lines***.
    - Refer figure 10,

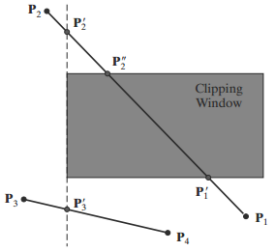


Figure 10: Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.

- Line segments can intersect clipping boundary lines without entering the interior of the window→ needs several intersection calculations to clip line
- As we process each clipping-window edge, a section of the line is clipped, and the remaining part of the line is checked against the other window borders.
- Continue eliminating sections until either the line is totally clipped or the remaining part of the line is inside the clipping window.

Example, (figure 12)

- Assumption: window edges are processed in the order: left, right, bottom, top.
- Determine whether a line crosses a selected clipping boundary
  - check corresponding bit values in the two endpoint region codes
    - If one of these bit values is 1 and the other is 0, the line segment crosses that boundary.

Process example,

- The region codes for the line from P_1 to P_2 are 0100 and 1001 → P_1 is inside the left clipping boundary and P_2 is outside that boundary.
- Calculate the intersection position P_2', and we clip off the line section from P_2 to P_2'.
- Check the bottom border.
  - EndpointP1is below the bottom clipping edge and P_2' is above it→ determine the intersection position at this boundary (P_1')
  - Eliminate the line section from P_1 to P_1' and proceed to the top window edge.
- Determine the intersection position to be P_2''.
  - Clip off the section above the top boundary and save the interior segment from P_1' to P_2''.

It is possible to calculate an intersection position at all four clipping boundaries, depending on how the line endpoints are processed and what ordering we use for the boundaries (figure 11).

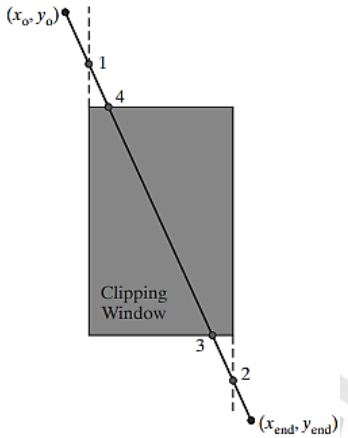


Figure 11: Four intersection positions (labeled from 1 to 4) for a line segment that is clipped against the window boundaries in the order left, right, bottom, top.

Procedure:

To determine a boundary intersection for a line segment, use the slope-intercept form of the line equation.

For a line with endpoint coordinates (x_0, y_0) and (x_{end}, y_{end}), the *y* coordinate of the intersection point with a vertical clipping border line can be obtained with the calculation

$$y = y_0 + m(x - x_0) \quad \ldots(12)$$

x→ set to either xw_{min} or xw_{max}

The slope of the line is calculated as $\boldsymbol{m=(y_{end}-y_0)/(x_{end}-x_0)}$.

For intersection with a horizontal border

$$x = x_0 + \frac{y - y_0}{m} \quad \ldots(13)$$

y→ set either to yw_{min} or to yw_{max}

c. Explain Sutherland-Hodgeman polygon clipping with an example. (6 marks)

Sutherland--Hodgman Polygon Clipping

"Send the polygon vertices through each clipping stage so that a single clipped vertex can be immediately passed to the next stage"

Advantage:

- Eliminates the need for an output set of vertices at each clipping stage.
- Allows the boundary-clipping routines to be implemented in parallel.
- Final output is a list of vertices that describe the edges of the clipped polygon fill area.

Sutherland-Hodgman algorithm produces only one list of output vertices→ it cannot correctly generate the two output polygons in figure 13(b).

Solution→ more processing steps can be added to the algorithm to allow it to produce multiple output vertex lists, so that general concave-polygon clipping could be accommodated.

Note: It can process concave polygons when the clipped fill area can be described with a single vertex list

General Strategy

- Send the pair of endpoints for each successive polygon line segment through the series of clippers (left, right, bottom, and top).
- After processing of one pair of vertices, the clipped coordinate values, if any, for that edge are sent to the next clipper
- The first clipper processes the next pair of endpoints.
- Hence, the individual boundary clippers can be operating in parallel.

Four possible cases

i. First edge endpoint is outside the clipping boundary and the second endpoint is inside, or

ii. Both endpoints could be inside this clipping boundary

iii. First endpoint is inside the clipping boundary and the second endpoint is outside

iv. Both endpoints could be outside the clipping boundary.

The process of passing of vertices from one clipping stage to the next is represented in figure 17.

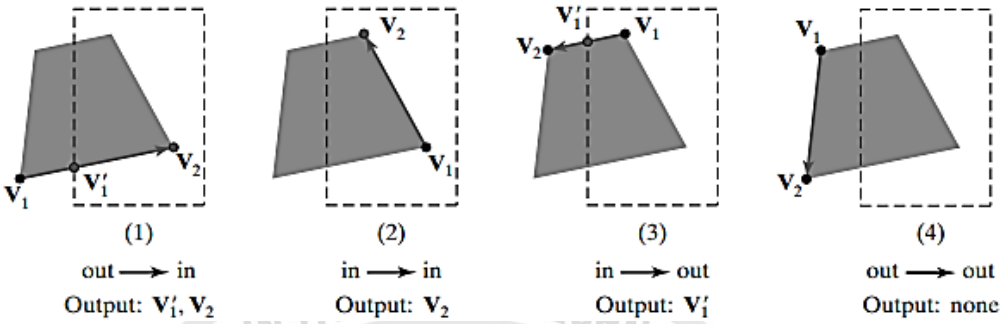


Figure 17: The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

At each clipper, the output is generated for the next clipper according to the results of following ***tests***:

i. If the first input vertex is outside this clipping-window border and the second vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper.

ii. If both input vertices are inside this clipping-window border, only the second vertex is sent to the next clipper.

iii. If the first vertex is inside this clipping-window border and the second vertex is outside, only the polygon edge-intersection position with the clipping-window border is sent to the next clipper

iv. If both input vertices are outside this clipping-window border, no vertices are sent to the next clipper.

v. The last clipper generates a vertex list that describes the final clipped fill area

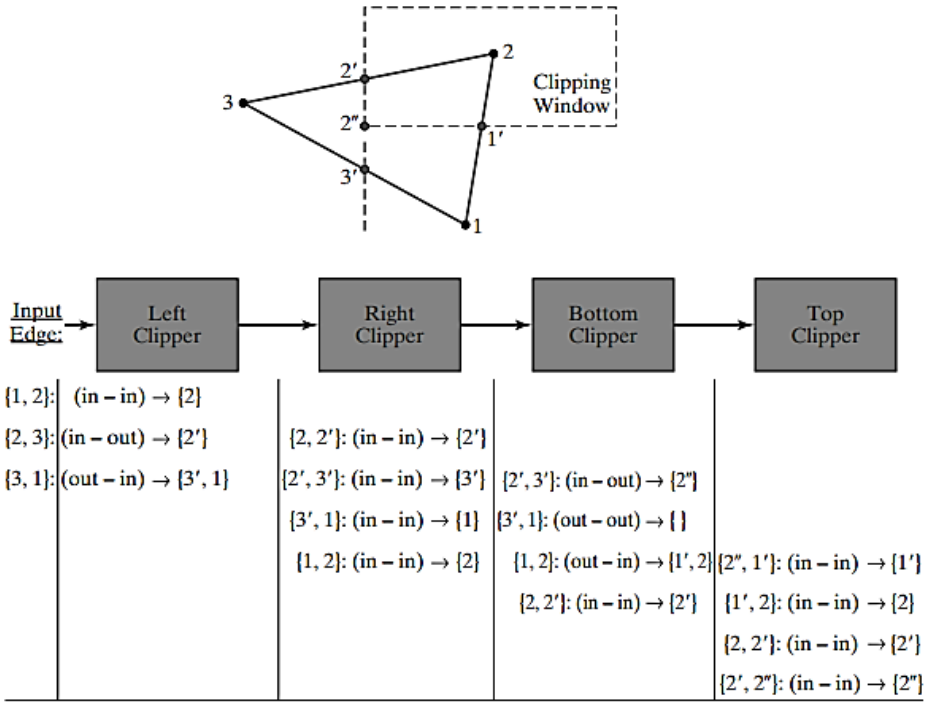


Figure 18: Processing a set of polygon vertices,{1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1', 2, 2', 2"}

Concave Polygon Clipping

With the Sutherland-Hodgman algorithm, extraneous lines may be displayed (figure 20).

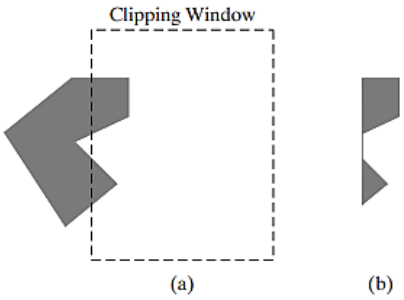


Figure 20: Clipping the concave polygon in (a) using the Sutherland-Hodgman algorithm produces the two connected areas in (b).

Since there is only one output vertex list, the last vertex in the list is always joined to the first vertex.
To display clipped concave polygons correctly:

i. Split a concave polygon into two or more convex polygons and process each convex polygon separately using the Sutherland-Hodgman algorithm, or
ii. Modify the Sutherland- Hodgman method so that the final vertex list is checked for multiple intersection points along any clipping-window boundary.
    a. If there are more than two vertex positions along any clipping boundary, separate the list of vertices into two or more lists that correctly identify the separate sections of the clipped fill area
iii. Or, use a more general polygon clipper that has been designed to process concave polygons correctly.

6. a. Write a note on 3D translation, rotation, and scaling. (6 marks).

3D Translation

A positionP=(x, y, z)in three-dimensional space is translated to a location P'=(x', y', z') by adding translation distances tx, ty, and tz to the Cartesian coordinates of P.

$$x' = x + t_x, \qquad y' = y + t_y, \qquad z' = z + t_z \quad \ldots(14)$$

2D translation is given in Figure 21.

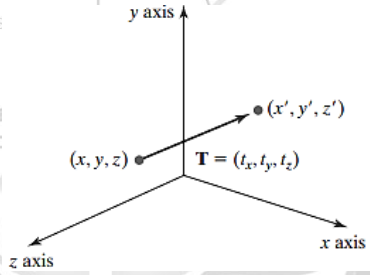


Figure 21: Moving a coordinate position with translation vector T=(t_x,t_y,t_z)

3D translation operations can be represented in matrix format. The coordinate positions, P and P', are represented in homogeneous coordinates with four-element column matrices, and the translation operator T is a 4×4 matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \ldots(15)$$

Or

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \quad \ldots(16)$$

An object is translated in three dimensions by transforming each of the defining coordinate positions for the object, then reconstructing the object at the new location. For an object represented as a set of polygon surface, translate each vertex for each surface (Figure 22) and redisplay the polygon facets at the translated positions

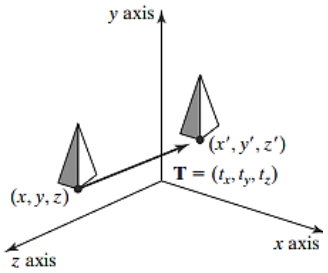


Figure 22: Shifting the position of a three-dimensional object using translation vector T.

3D Rotation

Rotate an object about any axis in space.

- Easiest rotation axes to handle are those that are parallel to the Cartesian-coordinate axes.
- Also can use combinations of coordinate-axis rotations (along with appropriate translations) to specify a rotation about any other line in space.

Positive rotation angles produce counterclockwise rotations about a coordinate axis (for negative direction along that coordinate axis) (figure 23).

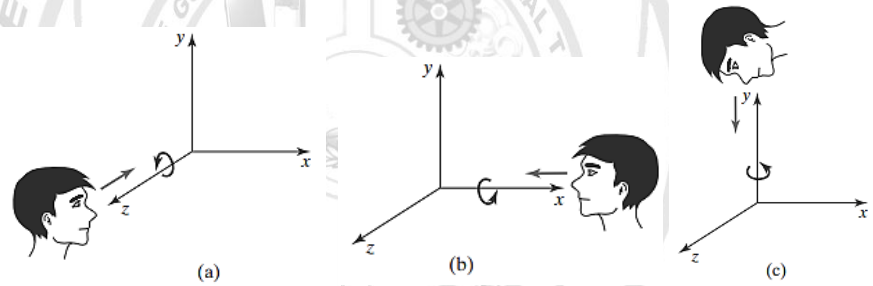


Figure 23: Positive rotations about a coordinate axis are counterclockwise, when looking along the positive half of the axis toward the origin.

Note: positive rotations in the xy plane are counter-clockwise about a pivot point (an axis that is parallel to the z axis).

Three-Dimensional Coordinate-Axis Rotations

The 2D z-axis rotation equations are easily extended to three dimensions, as follows:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$
$$z' = z \qquad \ldots(17)$$

Where, θ→ the rotation angle about the z axis

→ z-coordinate values are unchanged by this transformation

In homogeneous-coordinate form, the 3D z-axis rotation equations are:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \ldots(18)$$

Or

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P} \quad \ldots(19)$$

Figure 24 illustrates rotation of an object about the z axis.

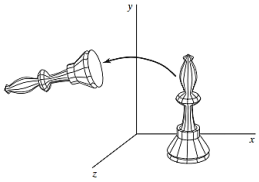


Figure 24: Rotation of an object about the z-axis.

Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters x, y, and z in Equations 17:

$$x \rightarrow y \rightarrow z \rightarrow x \quad \ldots(20)$$

To obtain the x-axis and y-axis rotation transformations, cyclically replace x with y, y with z, and z with x (figure 25).

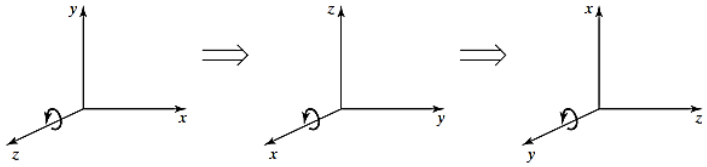


Figure 25: Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations

Substituting permutations 20 into Equations17, gives the equations for an x-axis rotation:

$$y' = y\cos\theta - z\sin\theta$$
$$z' = y\sin\theta + z\cos\theta$$
$$x' = x \quad \ldots(21)$$

Rotation of an object around the x axis is demonstrated in Figure 26.

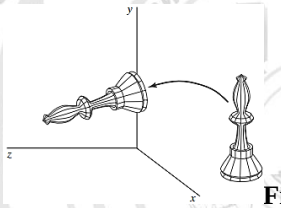


Figure 26: Rotation of an object about the x axis

A cyclic permutation of coordinates in Equations 21 gives the transformation equations for a y-axis rotation:

$$z' = z\cos\theta - x\sin\theta$$
$$x' = z\sin\theta + x\cos\theta$$
$$y' = y \quad \ldots(22)$$

An example of y-axis rotation is shown in Figure 27.

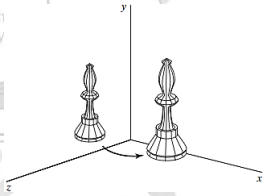


Figure 27: Rotation of an object about the y axis.

An inverse three-dimensional rotation → replace the angle θ with –θ.

- Negative values for rotation angles generate rotations in a clockwise direction
- The identity matrix is produced when we multiply any rotation matrix by its inverse.

Only the sine function is affected by the change in sign of the rotation angle→ inverse matrix can also be obtained by interchanging rows and columns → calculate the inverse of any rotation matrix R by forming its transpose ($R^{-1}=R^{T}$)

3D Scaling

The matrix expression for the three-dimensional scaling transformation of a position P=(x, y, z)relative to the coordinate origin is a simple extension of 2D scaling. Include the parameter for z-coordinate scaling in the transformation matrix:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \ldots(54)$$

The three-dimensional scaling transformation for a point position can be represented as

$$P' = S \cdot P \quad \ldots(55)$$

where scaling parameters s_x, s_y, and s_z are assigned any positive values. Explicit expressions for the scaling transformation relative to the origin are

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y, \qquad z' = z \cdot s_z \quad \ldots(56)$$

Scaling an object with transformation 54 changes the position of the object relative to the coordinate origin.

- A parameter value greater than 1 moves a point farther from the origin in the corresponding coordinate direction
- A parameter value less than 1 moves a point closer to the origin in that coordinate direction.

If the scaling parameters are not all equal, relative dimensions of a transformed object are changed

The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations:

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \ldots(57)$$

b. Explain the 3D reflection and shearing. (4 marks)

Three-Dimensional Reflections

A reflection in a three-dimensional space can be performed relative to a selected ***reflection axis*** or with respect to a ***reflection plane***. 3D reflection matrices are set up similarly to those for 2D.

- Reflections relative to a given axis are equivalent to 180° rotations about that axis.
- Reflections with respect to a plane are similar; when the reflection plane is a coordinate plane (xy, xz,or yz)→ transformation as a 180° rotation in four-dimensional space with a conversion between a left-handed frame and a right-handed frame.

An example of a reflection that converts coordinate specifications from a right-handed system to a left-handed system (or vice versa)is shown in Figure 39.

This transformation changes the sign of z coordinates, leaving the values for the x and y coordinates unchanged. The matrix representation for this reflection relative to the xy plane is

$$M_{z\text{reflect}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \ldots(58)$$

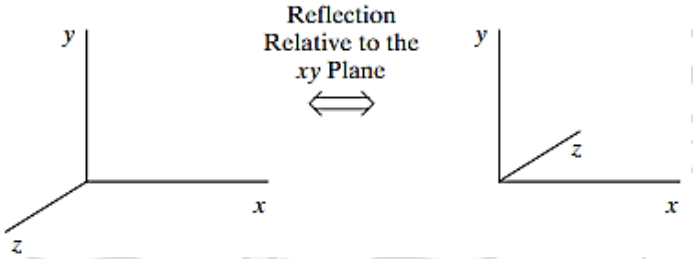


Figure 39: Conversion of coordinate specifications between a right-handed and a left-handed system can be carried out with the reflection transformation 58.

Transformation matrices for inverting x coordinates or y coordinates are defined similarly, as reflections relative to the yz plane or to the xz plane, respectively. Reflections about other planes can be obtained as a combination of rotations and coordinate-plane reflections.

Three-Dimensional Shears

- used to modify object shapes, just as in two-dimensional applications
- applied in three-dimensional viewing transformations for perspective projections.
- can also generate shears relative to the z axis.

A general z-axis shearing transformation relative to a selected reference position is produced with the following matrix:

$$M_{z\text{shear}} = \begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{ref} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{ref} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \ldots(59)$$

Shearing parameters sh_{zx} and sh_{zy} can be assigned any real values.

Result→ to alter the values for the x and y coordinates by an amount that is proportional to the distance from z_{ref}, while leaving the z coordinate unchanged.

Plane areas that are perpendicular to the z axis are thus shifted by an amount equal to $z-z_{ref}$.

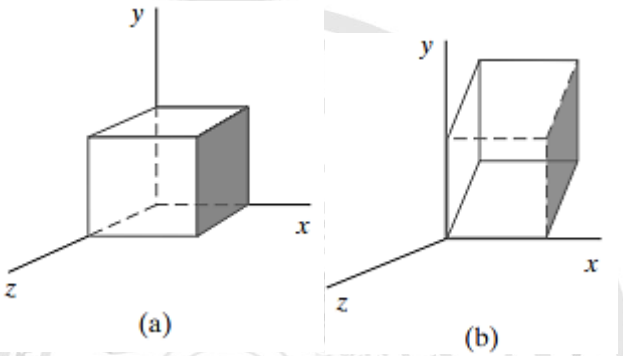


Figure 40: A unit cube (a) is sheared relative to the origin(b) by Matrix 59, with sh_{zx}=sh=zy =1

An example of the effect of this shearing matrix on a unit cube is shown in Figure20 for shearing values sh_{zx} =sh_{zy} =1 and a reference position z_{ref} =0.

NOTE:

- Three-dimensional transformation matrices for an x-axis shear and ay-axis shear are similar to the two-dimensional matrices.
- We need to add a row and a column for the z-coordinate shearing parameters.

c. Illustrate about RGB and CMY color models. Write a note on light sources. (6 marks)

The RGB Color Model

According to the ***tristimulus theory*** of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina.

- One of the pigments is most sensitive to light with a wavelength of about 630 nm (red),
- Another has its peak sensitivity at about 530 nm (green), and
- The third pigment is most receptive to light with a wavelength of about 450 nm (blue)

Based on the intensities of light, the color of the light is perceived.

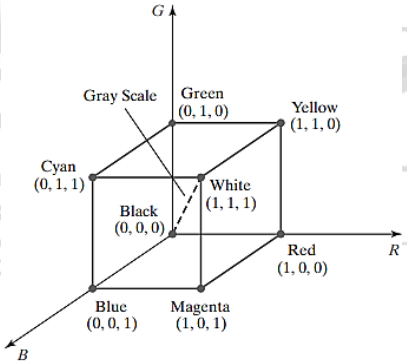


Figure 46: The RGB color model. Any color within the unit cube can be described as an additive combination of the three primary colors

Theory of vision is the basis for displaying color output on a video monitor using the three primaries red, green, and blue, which is referred to as the ***RGB color model***. This can be represented on a unit cube defined on R,G, and B axes, as in figure 46.

- The origin represents black and the diagonally opposite vertex, with coordinates (1, 1, 1), is white.

- Vertices of the cube on the axes represent the primary colors, and the remaining vertices are the complementary color points for each of the primary colors.

The RGB color scheme is an ***additive model***. Each color point within the unit cube can be represented as a weighted vector sum of the primary colors, using unit vectors R,G, and B:

$$C(\lambda) = (R, G, B) = R\mathbf{R} + G\mathbf{G} + B\mathbf{B} \ldots (61)$$

where parameter sR, G, and B are assigned values in the range from 0 to 1.0.

Example,

- The magenta vertex is obtained by adding maximum red and blue values to produce the triple (1, 0, 1), and
- White at (1, 1, 1) is the sum of the maximum values for red, green, and blue.
- Shades of gray are represented along
- Tthe main diagonal of the cube from the origin (black) to the white vertex.

Chromaticity coordinates for the National Television System Committee (NTSC) standard RGB phosphors are listed in Table1.

Table 1:

RGB (*x*, *y*) Chromaticity Coordinates

	NTSC Standard	CIE Model	Approx. Color Monitor Values
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)

Figure47 shows the approximate color gamut for the NTSC standard RGB primaries.

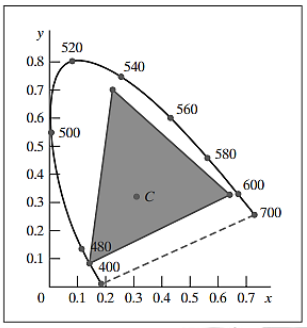


Figure 47: The RGB color gamut for NTSC chromaticity coordinates. Illuminant C is at position (0.310, 0.316), with a luminance value of Y=100.0.

The CMY Model

Hard-copy devices, such as printers and plotters, produce a color picture by coating a paper with color pigments. We see the color patterns on the paper by reflected light, which is a ***subtractive process***.

The CMY Parameters

A subtractive color model can be formed with the three primary colors cyan, magenta, and yellow.

For example, cyan is an a combination of green and blue when white light is reflected from cyan-colored ink, the reflected light contains only the green and blue components, and the red component is absorbed, or subtracted, by the ink.

A unit cube representation for the CMY model is illustrated in Figure 48.

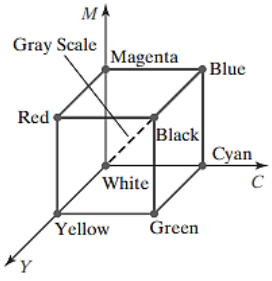


Figure 48: The CMY color model. Positions within the unit cube are described by subtracting the specified amounts of the primary colors from white.

- In the CMY model, the spatial position (1, 1, 1) represents black, because all components of the incident light are subtracted.
- The origin represents white light.
- Equal amounts of each of the primary colors produce shades of gray along the main diagonal of the cube.
- In this model
    - A combination of cyan and magenta ink produces blue light, because the red and green components of the incident light are absorbed.
    - A combination of cyan and yellow ink produces green light, and a combination of magenta and yellow ink yields red light

The CMY printing process

Uses a collection of four ink dots, which are arranged in a close pattern somewhat as an RGB monitor uses three phosphor dots. Therefore the CMY color model is referred to as the CMYK model, where K is the black color parameter.

- One ink dot is used for each of the primary colors (cyan, magenta, and yellow), and one ink dot is black.
- A black dot is included because reflected light from the cyan, magenta, and yellow inks typically produce only shades of gray.
- For black-and-white or gray scale printing, only the black ink is used

Module -4

7. a. Describe a 3D viewing pipeline with necessary diagrams. (6 marks).

Procedures for generating a computer-graphics view of a three-dimensional scene are similar to taking a photograph.

- Choose a viewing position corresponding to camera position.
- Viewing position is chosen according to whether we want to display a front, back, side, top, or bottom view of the scene.
- A position can also be picked in the middle of a group of objects or even inside a single object, such as a building or a molecule. Then decide on the camera orientation (Figure 5).

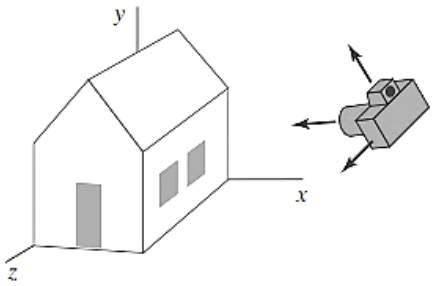


Figure 5: Photographing a scene involves selection of the camera position and orientation.

- Which way the camera is pointed from the viewing position or how the camera is rotated around the line of sight to set the "up" direction for the picture?
- When the shutter is snapped, the scene is cropped to the size of a selected clipping window, which corresponds to the aperture or lens type of a camera, and light from the visible surfaces is projected onto the camera film.

Computer-graphics program has more flexibility more options for generating views of a scene than in a real camera.

- Either parallel projection or a perspective projection, can be used.
- It is possible to
    - Selectively eliminate parts of a scene along the line of sight.
    - Move the projection plane away from the "camera" position,

- get a picture of objects in back of the synthetic camera.

Some of the viewing operations for a 3D scene are the same as or similar to those used in the 2D viewing pipeline.

- A 2D viewport is used to position a projected view of the 3D scene on the output device, and
- A 2D clipping window is used to select a view that is to be mapped to the viewport.
- Display window can be set up in screen coordinates as in 2D applications.
- Clipping windows, viewports, and display windows are specified as rectangles with their edges parallel to the coordinate axes.

In 3D viewing, the clipping window is positioned on a selected view plane, and scenes are clipped against an enclosing volume of space, which is defined by a set of ***clipping planes***.

- The clipping window is positioned on a selected view plane, and scenes are clipped against an enclosing volume of space, which is defined by a set of clipping planes

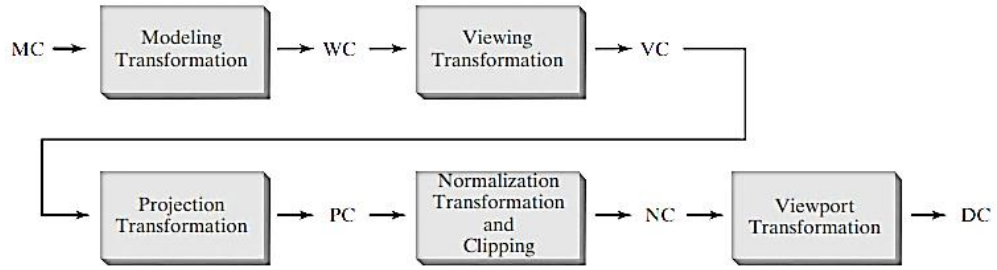


Figure 6: General three-dimensional transformation pipeline, from modeling coordinates (MC) to world coordinates (WC) to viewing coordinates (VC) to projection coordinates (PC) to normalized coordinates (NC) and, ultimately, to device coordinates (DC)

The general processing steps for creating and transforming a three-dimensional scene to device coordinates is shown in Figure 6.

- Once the scene has been modeled in world coordinates, a viewing-coordinate system is selected and the description of the scene is converted to viewing coordinates.
- The viewing coordinate system defines the viewing parameters, including the position and orientation of the projection plane (view plane), similar to the camera film plane.
- A 2D clipping window, corresponding to a selected camera lens, is defined on the projection plane, and a 3D *clipping region* is established → ***View Volume***.
- View volume's shape and size depends on the dimensions of the clipping window, the type of projection chosen, and the selected limiting positions along the viewing direction.
- Projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane.
- Objects are mapped to normalized coordinates, and all parts of the scene outside the view volume are clipped off.
    - The clipping operations can be applied after all device-independent coordinate transformations are completed.
- The viewport limits could be given in normalized coordinates or in device coordinates.

To develop viewing algorithms,

- Assume that the viewport is to be specified in device coordinates and that normalized coordinates are transferred to viewport coordinates, following the clipping operations.
- Identify visible surfaces
- Apply the surface-rendering procedures.
- Lastly map viewport coordinates to device coordinates within a selected display window.

b. Write a note on parallel and perspective projections. Also explain orthogonal projections in detail. (6 marks)

Object descriptions are projected to the view plane. Graphics packages support both parallel and perspective projections.

i. Parallel Projection

- Coordinate positions are transferred to the view plane along parallel lines

- Parallel projection for a straight-line segment defined with endpoint coordinatesP1andP2 is given in Figure 15.

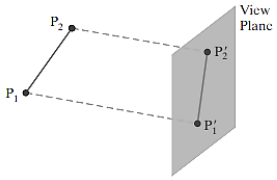


Figure 15: Parallel projection of a line segment onto a view plane.

- Preserves relative proportions of objects
- Used in computer-aided drafting and design to produce scale drawings of 3D objects.
- All parallel lines in a scene are displayed as parallel
- There are two general methods for obtaining a parallel-projection view of an object:
  - Project along lines that are perpendicular to the view plane
  - Project at an oblique angle to the view plane.

ii. Perspective projection

- Object positions are transformed to projection coordinates along lines that converge to a point behind the view plane.
- Perspective projection for a straight-line segment, defined with endpoint coordinates P_1 and P_2, is given in Figure 16.

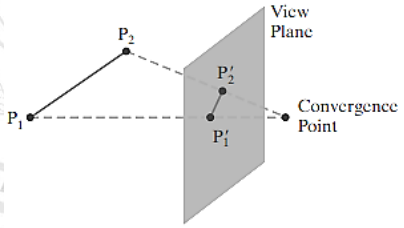


Figure 16: Perspective projection of a line segment onto a view plane

- It does not preserve relative proportions of objects.
- These views of a scene are more realistic because distant objects in the projected display are reduced in size.

Orthogonal projections

A transformation of object descriptions to a view plane along lines that are all parallel to the view-plane normal vector N is called an ***orthogonal projection (or, orthographic projection)***.

- Produces a parallel-projection transformation in which the projection lines are perpendicular to the view plane.
- Used to produce the front, side, and top views of an object (Figure 17).

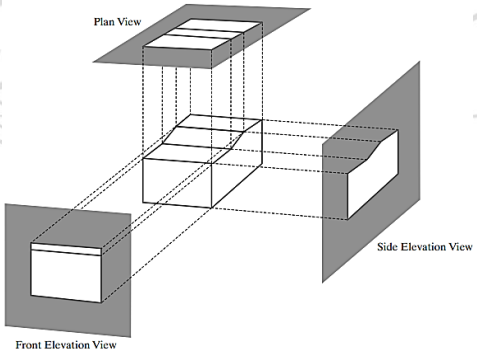


Figure 17: Orthogonal projections of an object, displaying plan and elevation views.

- Front, side, and rear orthogonal projections→ ***elevations***
- Top orthogonal view→ ***plain view***.

<u>Axonometric and Isometric Orthogonal Projections</u>

- Orthogonal projections that display more than one face of an object → ***axonometric orthogonal projections***.
- The most commonly used axonometric projection is the ***isometric projection***
    - Generated by aligning the projection plane (or the object) so that the plane intersects each coordinate axis in which the object is defined, called the ***principal axes***, at the same distance from the origin.
    - An isometric projection for cube is given in Figure 18.

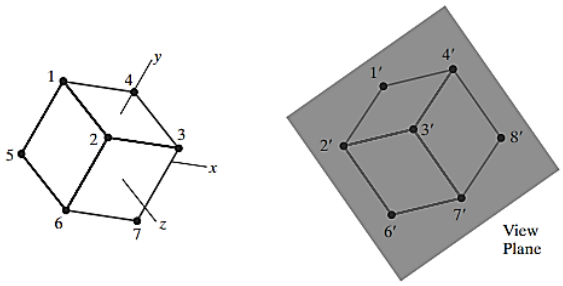


Figure 18: An isometric projection of a cube

- This is obtained by aligning the view-plane normal vector along a cube diagonal.
- There are eight positions, one in each octant, for obtaining an isometric view.
- All three principal axes are foreshortened equally in an isometric projection to maintain relative proportions

- In a general axonometric projection, scaling factors may be different for the three principal directions

Orthogonal Projection Coordinates

With the projection direction parallel to the zview axis, the transformation equations for an orthogonal projection are trivial. For any position(x, y, z) in viewing coordinates, as in Figure 19, the projection coordinates are

$$x_p = x,\ y_p = y$$

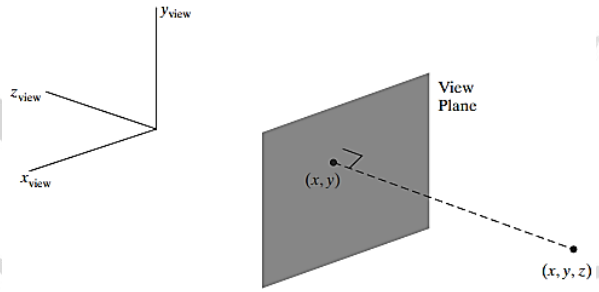


Figure 19: An orthogonal projection of a spatial position onto a view plane.

The z-coordinate value for any projection transformation is preserved for use in the visibility determination procedures. And each three-dimensional coordinate point in a scene is converted to a position in normalized space.

Clipping Window and Orthogonal-Projection View Volume

In the camera analogy, the type of lens is one factor that determines how much of the scene is transferred to the film plane. A wide-angle lens takes in more of the scene than a regular lens. For computer-graphics applications, we use the rectangular clipping window for this purpose. As in two-dimensional viewing, graphics packages typically require that clipping rectangles be placed in specific positions.

In OpenGL, we set up a clipping window for three-dimensional viewing just as we did for two-dimensional viewing, by choosing two-dimensional coordinate positions for its lower-left and upper-right corners.

For three-dimensional view-ing, the clipping window is positioned on the view plane with its edges parallel to the x_{view} and y_{view} axes, as shown in figure 20.

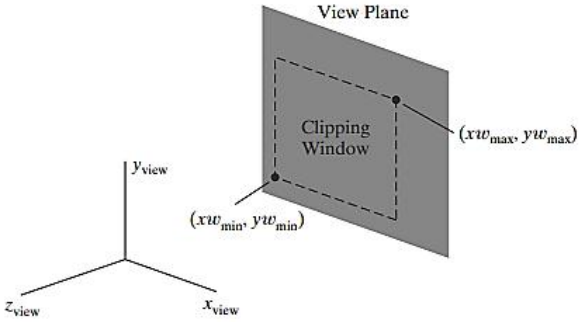


Figure 20: A clipping window on the view plane, with minimum and maximum coordinates given in the viewing reference system.

The edges of the clipping window specify the x and y limits for the part of the scene that we want to display. These limits are used to form the top, bot-tom, and two sides of a clipping region called the orthogonal-projection view volume.

Because projection lines are perpendicular to the view plane, these four boundaries are planes that are also perpendicular to the view plane and that pass through the edges of the clipping window to form an infinite clipping region, as in Figure 21

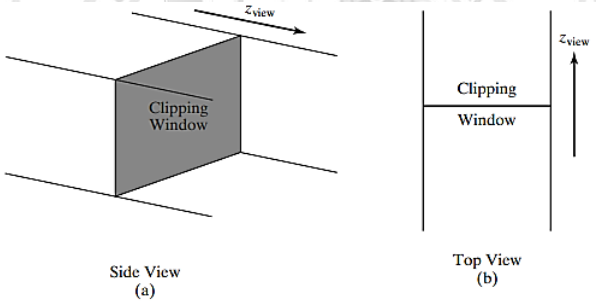


Figure 21: Infinite orthogonal-projection view volume.

We can limit the extent of the orthogonal view volume in the z_{view} direction by selecting positions for one or two additional boundary planes that are parallel to the view plane. These two planes are called the near-far clipping planes, or the front-back clipping planes. The near and far planes allow us to exclude objects that are in front of or behind the part of the scene that we want to display.

Some graphics libraries provide these two planes as options, and other libraries require them. When the near and far planes are specified, we obtain a finite orthogonal view volume that is a *rectangular parallelepiped*, as shown in figure 22 along with possible placement for the view plane.

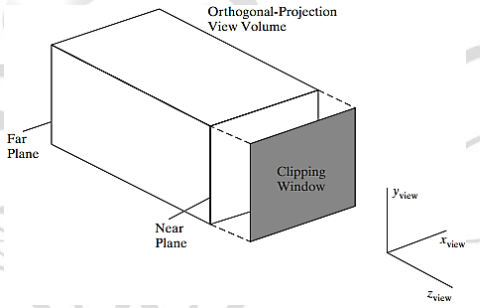


Figure 22: A finite orthogonal view volume with the view plane "in front" of the near plane.

c. What are vanishing points for perspective projections? (4 marks)

Vanishing Points for Perspective Projections

- When a scene is projected onto a view plane using a perspective mapping, lines that are parallel to the view plane are projected as parallel lines.
- Any parallel lines in the scene that are not parallel to the view plane are projected into converging lines
- The point at which a set of projected parallel lines appears to converge is called a ***vanishing point***. → Each set of projected parallel lines has a separate vanishing point.

- For a set of lines that are parallel to one of the principal axes of an object, the vanishing point is referred to as a ***principal vanishing point***.
- Number of principal vanishing points (one, two, or three) is controlled with the orientation of the projection plane, and perspective projections are accordingly classified as one-point, two-point, or three-point projections.
    - The number of principal vanishing points in a projection is equal to the number of principal axes that intersect the view plane.
- Figure b3 illustrates the appearance of one-point and two-point perspective projections for a cube.

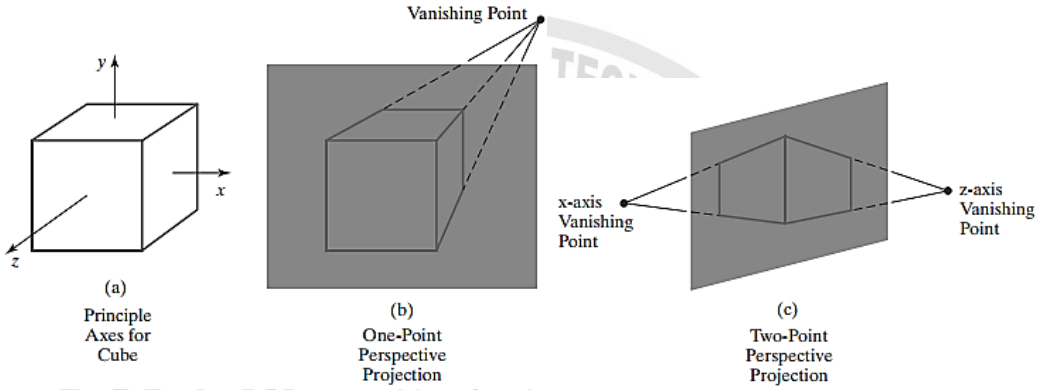


Figure b3: Principal vanishing points for perspective-projection views of a cube. When the cube in (a) is projected to a view plane that intersects only the z axis, a single vanishing point in the z direction (b) is generated. When the cube is projected to a view plane that intersects both the z and x axes, two vanishing points (c) are produced.

- In the projected view (b), the view plane is aligned parallel to the xy object plane so that only the object z axis is intersected→ produces a one-point perspective projection with a z-axis vanishing point
- For the view shown in (c), the projection plane intersects both the x and z axes but not the y axis→ produces two-point perspective projection contains both x-axis and z-axis vanishing points

8. a. Describe perspective projections with necessary diagrams. (4 marks).

We can approximate this geometric-optics effect by projecting objects to the view plane along converging paths to a position called the projection reference point(or center of projection)

Objects are then displayed with foreshortening effects, and projections of distant objects are smaller than the projections of objects of the same size that are closer to the view plane (Figure 24)

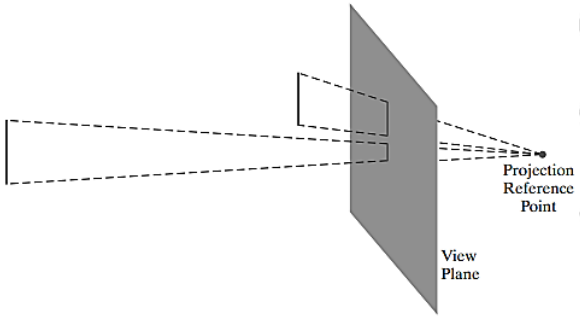


Figure 24: A perspective projection of two equal-length line segments at different distances from the view plane

Perspective-Projection Transformation Coordinates

We can sometimes select the projection reference point as another viewing pa-rameter in a graphics package, but some systems place this convergence point at a fixed position, such as at the view point. Figure 25 shows the projection path of a spatial position (x, y, z) to a general projection reference point at ($x_{pr}p$, y_{prp}, z_{prp}).

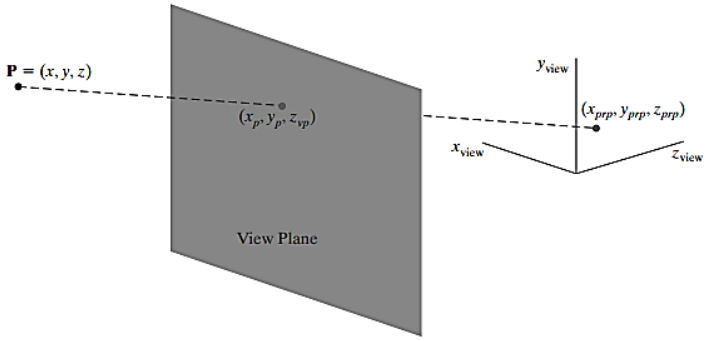


Figure 25: A perspective projection of a point P with coordinates (x, y, z) to a selected projection reference point. The intersection position on the view plane is (x_p,y_p,z_{vp})

The projection line intersects the view plane at the coordinate position (x_p,y_p,z_{vp}), where zvp is some selected position for the view plane on the zview axis. We can write equations describing coordinate positions along this perspective-projection line in parametric form as

$$x' = x - (x - x_{prp})u$$
$$y' = y - (y - y_{prp})u \qquad 0 \le u \le 1$$
$$z' = z - (z - z_{prp})u$$

On the viewplane,z' =zvp and we can solve the z'equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Substituting this value of u into the equations for x' and y', we obtain the general perspective-transformation equations

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

Perspective-Projection Equations: Special Cases

To simplify the perspective calculations, the projection reference point could be limited to positions along the z_{view} axis, then

1. $x_{prp} = y_{prp} = 0$:

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) \quad \text{…(eq 1)}$$

Sometimes the projection reference point is fixed at the coordinate origin, and

2. $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$:

$$x_p = x\left(\frac{z_{vp}}{z}\right), \qquad y_p = y\left(\frac{z_{vp}}{z}\right) \quad \text{…(eq 2)}$$

If the view plane is the uv plane and there are no restrictions on the placement of the projection reference point, then we have

3. $z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) - x_{prp}\left(\frac{z}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) - y_{prp}\left(\frac{z}{z_{prp} - z}\right) \quad \text{…(eq 3)}$$

With the uv plane as the view plane and the projection reference point on the z_{view} axis, the perspective equations are

4. $x_{prp} = y_{prp} = z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) \quad \text{…(eq 4)}$$

The view plane is usually placed between the projection reference point and the scene, but, in general, the view plane could be placed anywhere except at the projection point.

- If the projection reference point is between the view plane and the scene, objects are inverted on the view plane (Figure b1).

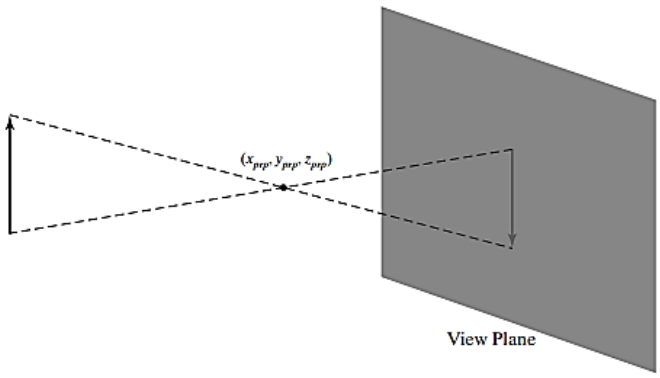


Figure b1: A perspective-projection view of an object is upside down when the projection reference point is between the object and the view plane.

- With the scene between the view plane and the projection point, objects are simply enlarged as they are projected away from the viewing position onto the view plane

Perspective effects also depend on the distance between the projection reference point and the view plane, as illustrated in Figure b2.

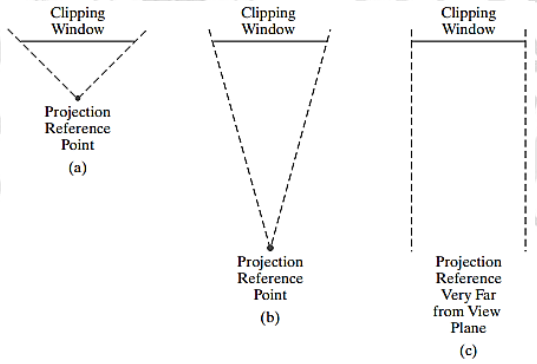


Figure b2: Changing perspective effects by moving the projection reference point away from the view plane.

- If the projection reference point is close to the view plane, perspective effects are emphasized;
  - Closer objects will appear much larger than more distant objects of the same size
  - As the projection reference point moves farther from the view plane, the difference in the size of near and far objects decreases.
  - When the projection reference point is very far from the view plane, a perspective projection approaches a parallel projection.

b. Write a note on oblique and symmetric perspective projection frustum. (8 marks)

Symmetric Perspective-Projection Frustum

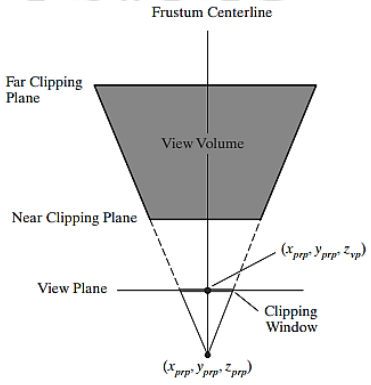


Figure b6: A symmetric perspective-projection frustum view volume, with the view plane between the projection reference point and the near clipping plane. This frustum is symmetric about its centerline when viewed from above, below, or either side

The line from the *projection reference point through the center of the clipping window* and *on through the view volume* is the *centerline for a perspective*-projection frustum. If this ***centerline is perpendicular to the view plane***, we have a ***symmetric frustum*** (with respect to its centerline) as in Figure b6.

Frustum centerline intersects the view plane at the coordinate location (x_{prp}, y_{prp}, z_{vp}) → express the corner positions for the clipping window in terms of the window dimensions:

$$xw_{\min} = x_{prp} - \frac{\text{width}}{2}, \qquad xw_{\max} = x_{prp} + \frac{\text{width}}{2}$$

$$yw_{\min} = y_{prp} - \frac{\text{height}}{2}, \qquad yw_{\max} = y_{prp} + \frac{\text{height}}{2}$$

A symmetric perspective-projection view of a scene can be specified using the width and height of the clipping window instead of the window coordinates.

Oblique Perspective-Projection Frustum

If the centerline of a perspective-projection view volume is not perpendicular to the view plane→ ***oblique frustum***.

An appearance of an oblique perspective-projection view volume is illustrates in figure b11.

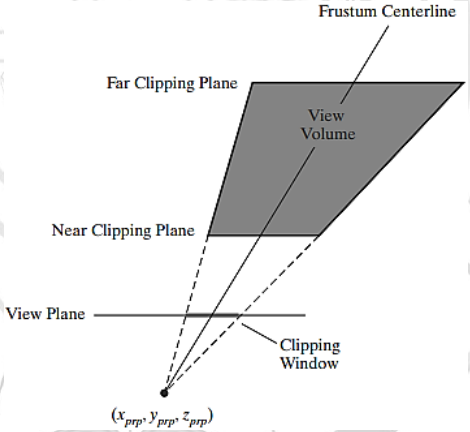


Figure b11: An oblique frustum, as viewed from at least one side or a top view, with the view plane positioned between the projection reference point and the near clipping plane

An oblique perspective-projection view volume can be converted to a symmetric frustum by applying a z-axis shearing-transformation matrix.

This trans-formation shifts all positions on any plane that is perpendicular to the z axis by an amount that is proportional to the distance of the plane from a specified z-axis reference position (the reference position is z_{prp}).

Shift→ by an amount that will move the center of the clipping window to position (x_{prp}, y_{prp}) on the view plane.

The computations for the shearing transformation and for the perspective and normalization transformations are greatly reduced if projection reference point is considered to be viewing-coordinate origin. Or set up the viewing-coordinate reference frame so that its origin is at the projection point of a scene.

Taking the projection reference point as (x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0), we obtain the elements of the required shearing matrix as

$$\mathbf{M}_{z\,\text{shear}} = \begin{bmatrix} 1 & 0 & sh_{zx} & 0 \\ 0 & 1 & sh_{zy} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \ldots(\text{eq } 13)$$

To move the center of the clipping window to coordinates (0, 0) on the view plane, we need to choose values for the shearing parameters such that

$$\begin{bmatrix} 0 \\ 0 \\ z_{near} \\ 1 \end{bmatrix} = \mathbf{M}_{z\,shear} \cdot \begin{bmatrix} \frac{xw_{min} + xw_{max}}{2} \\ \frac{yw_{min} + yw_{max}}{2} \\ z_{near} \\ 1 \end{bmatrix} \quad \ldots(\text{eq } 14)$$

Therefore, the parameters for this shearing transformation are

$$sh_{zx} = -\frac{xw_{min} + xw_{max}}{2\, z_{near}}$$

$$sh_{zy} = -\frac{yw_{min} + yw_{max}}{2\, z_{near}} \quad \ldots(\text{eq } 15)$$

Similarly, with the projection reference point at the viewing-coordinate origin and with the near clipping plane as the view plane, the perspective-projection matrix 9 is simplified to

$$\mathbf{M}_{pers} = \begin{bmatrix} -z_{near} & 0 & 0 & 0 \\ 0 & -z_{near} & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \ldots(\text{eq } 16)$$

Concatenating the simplified perspective-projection matrix 16 with the shear matrix 13, we obtain the following oblique perspective-projection matrix for converting coordinate positions in a scene to homogeneous orthogonal-projection coordinates.

The projection reference point for this transformation is the viewing-coordinate origin, and the near clipping plane is the view plane

$$\mathbf{M}_{obliquepers} = \mathbf{M}_{pers} \cdot \mathbf{M}_{z\,shear}$$

$$= \begin{bmatrix} -z_{near} & 0 & \frac{xw_{min} + xw_{max}}{2} & 0 \\ 0 & -z_{near} & \frac{yw_{min} + yw_{max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \ldots(\text{eq } 17)$$

If we choose the clipping-window coordinates so that $xw_{max}=-xw_{min}$ and $yw_{max}= -yw_{min}$, the frustum view volume is symmetric and matrix 17 reduces to matrix 16

c. List and explain OpenGL 3D viewing functions. (4 marks)

OpenGL Viewing-Transformation Function

set the modelview mode with the statement:

glMatrixMode (GL_MODELVIEW);

Viewing parameters are specified with the following GLU function, which is in the OpenGL Utility library because it invokes the translation and rotation routines in the basic OpenGL library.

gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

Values for all parameters in this function are to be assigned double-precision, floating-point values. This function designates the origin of the viewing reference frame as the world-coordinate position P0 =(x0, y0, z0), the reference posi-tion as Pref=(xref, yref, zref), and the view-up vector asV=(Vx, Vy, Vz). The positive z view axis for the viewing frame is in the direction N=P0−Pref.

If we do not invoke the gluLookAt function, the default OpenGL viewing parameters are

P0=(0, 0, 0)

Pref =(0, 0,−1)

V=(0, 1, 0)

For these default values, the viewing reference frame is the same as the world frame, with the viewing direction along the negative z_{world} axis.

OpenGL Orthogonal-Projection Function

Projection matrices are stored in the OpenGL projection mode. So, to set up a projection-transformation matrix, we must first invoke that mode with the statement

glMatrixMode (GL_PROJECTION);

Then, when we issue any transformation command, the resulting matrix will be concatenated with the current projection matrix.

Orthogonal-projection parameters are chosen with the function

glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

All parameter values in this function are to be assigned double-precision, floating-point numbers. Parameters dnear and dfar denote distances in the negative z_{view} direction from the viewing-coordinate origin.

Default parameter values for the OpenGL orthogonal-projection function are ±1, which produce a view volume that is a symmetric normalized cube in the right-handed viewing-coordinate system. This default is equivalent to issuing the statement

glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

OpenGL Symmetric Perspective-Projection Function

A symmetric, perspective-projection, frustum view volume is set up with the GLU function

gluPerspective (theta, aspect, dnear, dfar);

with each of the four parameters assigned a double-precision, floating-point number. The first two parameters define the size and position of the clipping window on the near plane, and the second two parameters specify the distances from the view point (coordinate origin) to the near and far clipping planes. Parameter theta represents the field-of-view angle, which is the angle between the top and bottom clipping planes.

OpenGL General Perspective-Projection Function

We can use the following function to specify a perspective projection that has either a symmetric frustum view volume or an oblique frustum view volume.

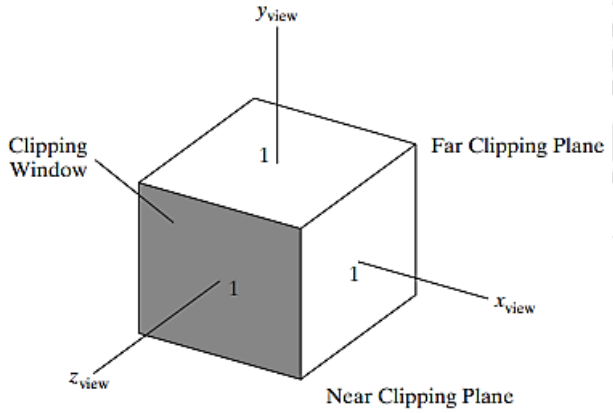


Figure 26: Default orthogonal-projection view volume. Coordinate extents for this symmetric cube are from−1to+1in each direction. The near clipping plane is at znear=1, and the far clipping plane is at zfar =−1.

glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

All parameters in this function are assigned double-precision, floating-point numbers. As in the other viewing-projection functions, the near plane is the view plane and the projection reference point is at the viewing position (coordinate origin).

The first four parameters set the coordinates for the clipping window on the near plane, and the last two parameters specify the distances from the coordinate origin to the near and far clipping planes along the negative zview axis. Locations for the near and far planes are calculated asznear =−dnear and zfar =−dfar.

OpenGL Viewports and Display Windows

A rectangular viewport is defined with the following OpenGL function

glViewport (xvmin, yvmin, vpWidth, vpHeight);

The first two parameters in this function specify the integer screen position of the lower-left corner of the viewport relative to the lower-left corner of the display window. And the last two parameters give the integer width and height of the viewport.

Module -5

9. a. Illustrate how an interactive program is animated. (4 marks).

Consider a two-dimensional point where

x =cosΘ,

y =sinΘ

This point lies on a unit circle regardless of the value of Θ. The three points(- sinΘ, cosΘ), (- cosΘ, - sinΘ), and (sinΘ, - cosΘ) also lie on the unit circle, and the four points are equidistant apart along the circumference of the circle, as shown below.

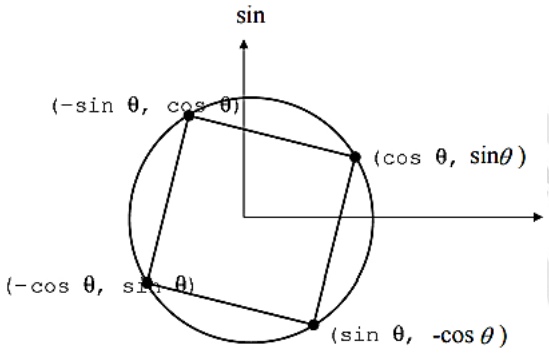


Figure 16: Square Constructed from Four Points on a Circle

Thus, if the points are connected to form a polygon, a square centered at the origin whose sides are of length √2, is obtained.

Assuming that the value of Θ is a global variable, the cube can be displayed with the following display function:

```
void disp1ay()
{
    glC1ear();
    glBegin(GL_POLYGON);
                /* convert degrees to radians */
        thetar = theta/((2*3.14159)/360.0);

        glVertex2f(cos(thetar), sin(thetar));
glVertex2f(-sin(thetar), cos(thetar));
        glVertex2f(-cos(thetar), -sin(thetar));
        glVertex2f(sin(thetar), -cos(thetar));
    glEnd();
}
```

- This will work for any value of Θ.
- Assume that the Θ is changed as the program is running, thus rotating the square about the origin. Then the rotated cube can be displayed by simply re-executing the display function through a call to glutPostRedisplay.
- Assume that Θ is to be increased by a fixed amount whenever nothing else is happening. Then the idle callback can be used for this operation. Thus, in the main program, idle callback function is specified as follows.

```
glutIdleFunc(idle);
```

and this function is defined as follows:

```
void id1e()
{
    theta+=2;       /* or some other amount */
    if(theta >= 360.0) theta-=360.0;
    glutPostRedisp1ay();
```

```
}
```

- To turn on and turn off the rotation feature: This can be done by using the mouse function to change the idle function. The mouse callback is

```
glutMouseFunc(mouse)
```

Then, this function can be written as

```
void mouse(int button, int state, int x, int y)
{
    if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
        glutId1eFunc(id1e);
    if(button==GLUT_MIDDLE_BUTTON&&state=GLUT_DOWN)
        glutId1eFunc(NULL);
}
```

Thus, the left mouse button starts the cube rotating, and the middle mouse button stops the rotation.

If the program is run, the display probably will not look likea rotating cube. Rather, parts of cubes changing over time are seen.

To have a smooth animation

1. Request a double buffered display in mainfunction by
   glutlnitDisp1ayMode(GLUT_RGB | GLUT_DOUBLE);
2. Add the line
   glutSwapBuffers();

after the cube is drawn in display function

b. What are quadratic surfaces? List and explain OpenGL Quadratic-Surface and Cubic-Surface Functions. (6 marks)

A frequently used class of objects are the quadric surfaces, which are described with second-degree equations (quadratics). They include spheres, ellipsoids, tori, paraboloids, and hyperboloids.

i. Sphere

In Cartesian coordinates, a spherical surface with radius centered on the coordinate origin is defined as the set of points(x, y, z) that satisfy the equation

$$x^2+y^2+z^2=r^2 \quad \text{.....(1)}$$

We can also describe the spherical surface in parametric form, using latitude and longitude angles (Figure below):

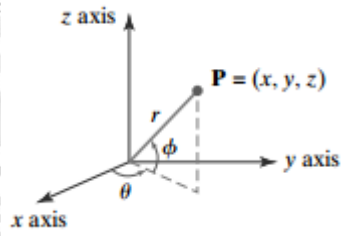


Figure 20: Parametric coordinate position (r,θ, φ) on the surface of a sphere with radius r.

$$x = r\cos\phi\cos\theta, \quad -\pi/2 \le \phi \le \pi/2$$
$$y = r\cos\phi\sin\theta, \quad -\pi \le \theta \le \pi$$
$$z = r\sin\phi \qquad \text{....(2)}$$

The parametric representation in Equations 2 provides a symmetric range for the angular parameters θ and φ.

Alternative→ write the parametric equations using standard spherical coordinates, where angle φ is specified as the colatitude (Figure below). Then, φ is defined over the range $0 \le \varphi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$. We could also set up the representation using parameters u and v defined over the range from 0 to 1 by substituting $\varphi = \pi u$ and $\theta = 2\pi v$.

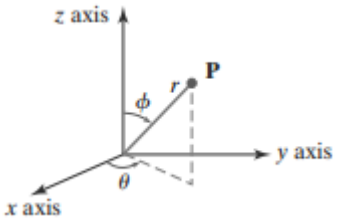


Figure 21: Spherical coordinate parameters (r, θ, φ), using colatitude for angle φ

ii. Ellipsoid

An ellipsoidal surface can be described as an extension of a spherical surface where the radii in three mutually perpendicular directions can have different values (Figure below).

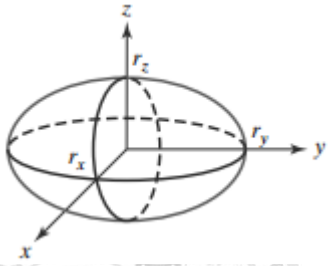


Figure 22: An ellipsoid with radii r_x, r_y, and r_z, centered on the coordinate origin

The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad \ldots(3)$$

And a parametric representation for the ellipsoid in terms of the latitude angle φ and the longitude angle θ in Figure 20 is

$$\begin{aligned} x &= r_x \cos\phi \cos\theta, \qquad -\pi/2 \le \phi \le \pi/2 \\ y &= r_y \cos\phi \sin\theta, \qquad -\pi \le \theta \le \pi \\ z &= r_z \sin\phi \end{aligned} \quad \ldots(4)$$

iii. Torus

A doughnut-shaped object is called a torus or anchor ring. It is the surface generated by rotating a circle or an ellipse about a coplanar axis line that is external to the conic.

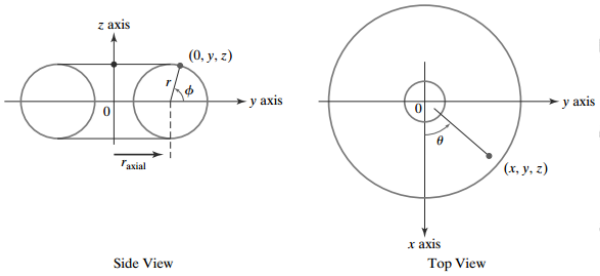


Figure 23: A torus, centered on the coordinate origin, with a circular cross-section and with the torus axis along the z axis.

- The defining parameters→ the distance of the conic center from the rotation axis and
  → the dimensions of the conic.

A torus generated by the rotation of a circle with radius r in the yz plane about the z axis is shown in Figure 23

GLUT Quadric-Surface Functions

GLUT sphere can be generated with either of these two functions:

i. glutWireSphere (r, nLongitudes, nLatitudes); or

ii. glutWireSphere (r, nLongitudes, nLatitudes);

r→ sphere radius→ double-precision floating-point number

nLongitudes and nLatitudes→ to select the integer number of longitude and latitude lines that will be used to approximate the spherical surface as a quadrilateral mesh.

A GLUT cone is obtained with

glutWireCone (rBase, height, nLongitudes, nLatitudes);

or

glutSolidCone (rBase, height, nLongitudes, nLatitudes);

- double-precision, floating-point values for the radius of the cone base and for the cone height using parameters rbase and height
- nLongitudes and nLatitudes→ integer values that specify the number of orthogonal surface lines for the quadrilateral mesh approximation

The cone→ center of the base at the world-coordinate origin and with the cone axis along the world z axis.

Wire-frame or surface-shaded displays of a torus with a circular cross-section are produced with

glutWireTorus (rCrossSection, rAxial, nConcentrics,nRadialSlices);

or

glutSolidTorus (rCrossSection, rAxial, nConcentrics,nRadialSlices);

Resulting torus→ surface generated by rotating a circle with radius rCrossSection about the coplanar z axis, where the distance of the circle center from the z axis is rAxial.

nConcentricsandnRadialSlices→ size of the quadrilaterals in the approximating surface mesh for the torus is set with integer values for parameter.

GLUT Cubic-Surface Teapot Function

- Three-dimensional objects that could be used to test rendering techniques
- Surfaces of a Volkswagen automobile and a teapot→ University of Utah.
- Utah teapot→ Martin Newell in 1975→ 306 vertices, defining 32 bicubic Bezier surface patches

→ mesh of over 1,000 bicubic surface patches, using either of the following two GLUT functions:

glutWireTeapot (size);

or

glutSolidTeapot (size);

Size→ double-precision floating-point value for the maximum radius of the teapot bowl

→ generated using OpenGL Bezier curve functions

→ centered on the world-coordinate origin coordinate origin with its vertical axis along the y axis.

GLU Quadric-Surface Functions

- To generate a quadric surface using GLU functions
  - Assign a name to the quadric,
  - Activate the GLU quadric renderer, and
  - Designate values for the surface parameters
  - Set other parameter values to control the appearance of a GLU quadric surface.
- Example, sequence of calls for displaying a wire-frame sphere centered on the world-coordinate origin:

```
GLUquadricObj *sphere1; /*sphere1-name of the quadric object defined. gluNewQuadricfunction- activates quadric renderer*/
sphere1 = gluNewQuadric ( );
gluQuadricDrawStyle (sphere1, GLU_LINE);/*GLU_LINE - selected for sphere1 with the gluQuadricDrawStyle command.*/
gluSphere (sphere1, r, nLongitudes, nLatitudes);
```

The sphere is displayed in a wire-frame form with a straight-line segment between each pair of surface vertices.

r→ double-precision value for the sphere radius

Sphere surface is divided into a set of polygon facets by the equally spaced longitude and latitude lines.

nLongitudes and nLatitudes→ integer number of longitude lines and latitude lines

Three other display modes are available for GLU quadric surfaces:

i. GLU_POINT→ a point is displayed at each surface vertex formed by the intersection of a longitude line and a latitude line

ii. GLU_SILHOUETTE→ wire-frame display without the shared edges between two coplanar polygon facets

iii. GLU_FILL→ display the polygon patches as shaded fill areas.

To produce a view of a cone, cylinder, or tapered cylinder, we replace the gluSphere function with

gluCylinder (quadricName, rBase, rTop, height, nLongitudes, nLatitudes);

base→xy plane (z=0)

axis→ z-axis

rBase→ double-precision radius value

rTop→ assign a radius to the top of the quadric surface

if rTop=0.0→ cone

if rTop=rBase→ cylinder

height→ double-precision height value

nLongitudes and nLatitudes→ equally spaced vertical and horizontal lines as determined by the integer values

A flat, circular ring or solid disk is displayed in thexyplane (z=0) and centered on the world-coordinate origin with

gluDisk (ringName, rInner, rOuter, nRadii, nRings);

rInner and rOuter→ double-precision values for an inner radius and an outer radius

If rInner=0→ the disk is solid.

Otherwise→ it is displayed with a concentric hole in the center of the disk

nRadii and nRings→ to divide disk surface into facets

→ specifies→ the number of radial slices to be used in the tessellation and

→ the number of concentric circular rings

Orientation→ the z axis, with the front of the ring facing in the +ve z direction and the back of the ring facing in the−z direction

To specify a section of a circular ring, use GLU function

gluPartialDisk (ringName, rInner, rOuter, nRadii, nRings,startAngle, sweepAngle);

startAngle→ angular position in degrees in the xy plane measured clockwise from the positive y axis

sweepAngle→ angular distance in degrees from the startAngle position

Here, section of a flat, circular disk is displayed from angular position startAngle to startAngle+sweepAngle.

Consider,

startAngle=0.0

sweepAngle=90.0

This displays he section of the disk lying in the first quadrant of the xy plane.

To reclaim and the surface eliminate, use

gluDeleteQuadric (quadricName);

To define the front and back directions for any quadric surface, use

gluQuadricOrientation (quadricName, normalVectorDirection);

normalVectorDirection→ either GLU_OUTSIDE or GLU_INSIDE

→ represents direction for the surface normal vectors

→ outside→ front-face direction

→ inside→ back-face direction

→ default→ GLU_OUTSIDE.

To generate surface-normal vectors, use

gluQuadricNormals (quadricName, generationMode);

generationMode→ how to generate the surface-normal vectors

→ default→ GLU_NONE→ no surface normals are to be generated and no lighting conditions typically are applied to the quadric surface.

To designate a function that is to be invoked if an error occurs during the generation of a quadric surface:

gluQuadricCallback (quadricName, GLU_ERROR, function);

c. With necessary codes, explain Bezier Spline Curves. (6 marks)

Bézier splines

- useful and convenient for curve and surface design
- easy to implement.

A Bézier curve section can be fitted to any number of control points.

- The number of control points are limited to 4 in some graphic packages.
- degree of the Bézier polynomial is determined by the number of control points to be approximated and their relative position.

Bézier Curve Equations

Consider the general case ofn+1 control-point positions, denoted as $p_k = (x_k, y_k, z_k)$, with k varying from *0* to *n*.

These points are blended to produce the following position vector *P(u)* (path of an approximating Bézier polynomial function between p_0 and p_n)

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \, \mathrm{BEZ}_{k,n}(u), \qquad 0 \le u \le 1 \qquad \text{...(9)}$$

The Bézier blending functions $BEZ_{k,n}(u)$ are the Bernstein polynomials.

$$\mathrm{BEZ}_{k,n}(u) = C(n,k)u^k(1-u)^{n-k} \qquad \text{...(10)}$$

where parametersC(n,k)are the binomial coefficients

$$C(n,k) = \frac{n!}{k!(n-k)!} \qquad \text{...(11)}$$

Equation 9 represents a set of three parametric equations for the individual curve coordinates:

$$x(u) = \sum_{k=0}^{n} x_k \, \mathrm{BEZ}_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k \, \mathrm{BEZ}_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k \, \mathrm{BEZ}_{k,n}(u) \qquad \text{...(12)}$$

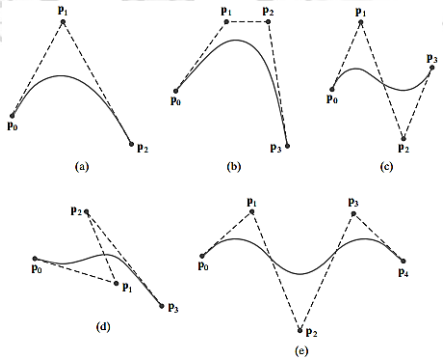


Figure 25: Examples of two-dimensional Bézier curves generated with three, four, and five control points. Dashed lines connect the control-point positions.

Bézier curve is a polynomial of a degree that is one less than the designated number of control points: Three points generate a parabola, four points a cubic curve, etc.

Figure 20: Bézier curves for various selections of control points in the xy plane (z=0).

Degenrate Bézier curve

→ With control-point placements.

For example,

- Bézier curve generated with three collinear control points is a straight-line segment;
- A set of control points that are all at the same coordinate position produce a Bézier "curve" that is a single point.

Recursive calculations can be used to obtain successive binomial-coefficient values as

$$C(n,k) = \frac{n-k+1}{k} C(n,k-1) \quad \ldots(13)$$

For n≥k. Also, the Bézier blending functions satisfy the recursive relationship

$$\mathrm{BEZ}_{k,n}(u) = (1-u)\mathrm{BEZ}_{k,n-1}(u) + u\,\mathrm{BEZ}_{k-1,n-1}(u), \qquad n > k \geq 1 \quad \ldots(14)$$

With $\mathrm{BEZ}_{k,k} = u^k$ and $\mathrm{BEZ}_{0,k} = (1-u)^k$.

Example Bézier Curve-Generating Program

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
/* Set initial size of the display window. */
GLsizei winWidth = 600, winHeight = 600;
/* Set size of world-coordinate clipping window. */
GLfloat xwcMin = -50.0, xwcMax = 50.0;
GLfloat ywcMin = -50.0, ywcMax = 50.0;
```

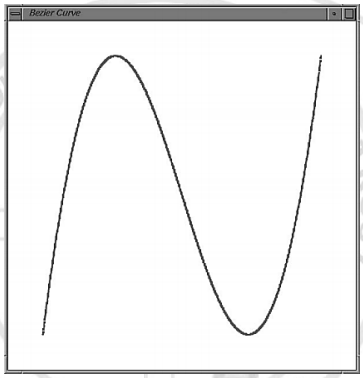
Spline Representations

```
class wcPt3D {
        public:
        GLfloat x, y, z;
};
void init (void)
{
        /* Set color of display window to white. */
        glClearColor (1.0, 1.0, 1.0, 0.0);
}
void plotPoint (wcPt3D bezCurvePt)
{
        glBegin (GL_POINTS);
        glVertex2f (bezCurvePt.x, bezCurvePt.y);
        glEnd ( );
}
/* Compute binomial coefficients C for given value of n. */
void binomialCoeffs (GLint n, GLint * C)
{
        GLint k, j;
        for (k = 0; k <= n; k++) {
                /* Compute n!/(k!(n - k)!). */
                C [k] = 1;
                for (j = n; j >= k + 1; j--)
                C [k] *= j;
                for (j = n - k; j >= 2; j--)
                C [k] /= j;
        }
```

```
}
void computeBezPt (GLfloat u, wcPt3D * bezPt, GLint nCtrlPts,
wcPt3D * ctrlPts, GLint * C)
{
        GLint k, n = nCtrlPts - 1;
        GLfloat bezBlendFcn;
        bezPt->x = bezPt->y = bezPt->z = 0.0;
        /* Compute blending functions and blend control points. */
        for (k = 0; k < nCtrlPts; k++) {
                bezBlendFcn = C [k] * pow (u, k) * pow (1 - u, n - k);
                bezPt->x += ctrlPts [k].x * bezBlendFcn;
                bezPt->y += ctrlPts [k].y * bezBlendFcn;
                bezPt->z += ctrlPts [k].z * bezBlendFcn;
        }
}
void bezier (wcPt3D * ctrlPts, GLint nCtrlPts, GLint nBezCurvePts)
{
        wcPt3D bezCurvePt;
        GLfloat u;
        GLint *C, k;
        /* Allocate space for binomial coefficients */
        C = new GLint [nCtrlPts];
        binomialCoeffs (nCtrlPts - 1, C);
        for (k = 0; k <= nBezCurvePts; k++) {
                u = GLfloat (k) / GLfloat (nBezCurvePts);
                computeBezPt (u, &bezCurvePt, nCtrlPts, ctrlPts, C);
                plotPoint (bezCurvePt);
        }
        delete [ ] C;
}
void displayFcn (void)
{
        /* Set example number of control points and number of
        * curve positions to be plotted along the Bezier curve.
        */
        GLint nCtrlPts = 4, nBezCurvePts = 1000;
        wcPt3D ctrlPts [4] = { {-40.0, -40.0, 0.0}, {-10.0, 200.0, 0.0},
        {10.0, -200.0, 0.0}, {40.0, 40.0, 0.0} };
        glClear (GL_COLOR_BUFFER_BIT); // Clear display window.
        glPointSize (4);
        glColor3f (1.0, 0.0, 0.0); // Set point color to red.
        bezier (ctrlPts, nCtrlPts, nBezCurvePts);
        glFlush ( );
}
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
        /* Maintain an aspect ratio of 1.0. */
        glViewport (0, 0, newHeight, newHeight);
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ( );
        gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);
        glClear (GL_COLOR_BUFFER_BIT);
```

```
}
void main (int argc, char** argv)
{
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition (50, 50);
        glutInitWindowSize (winWidth, winHeight);
        glutCreateWindow ("Bezier Curve");
        init ( );
        glutDisplayFunc (displayFcn);
        glutReshapeFunc (winReshapeFcn);
        glutMainLoop ( );
}
```

OUTPUT:



10. a. Represent simple graphics & display processor architectures. Explain the 2 ways of sending graphical entities to a display and list advantages and disadvantages. (6 marks).

Simple Graphics Architecture:

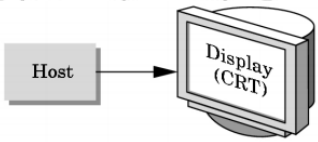


Figure 11: Simple Graphics Architecture

- Based on a general-purpose computer (or host) connected, through digital-to-analog converters, to a CRT.
- The computer would send out the necessary information to redraw the display at a rate sufficient to avoid noticeable flicker.
- Computers were slow and expensive, so the cost of keeping even a simple display refreshed was prohibitive for all but a few applications. As a solution to this, display processor architecture was built.

Display Processor Architecture: It uses a special-purpose computer, called a display processor.

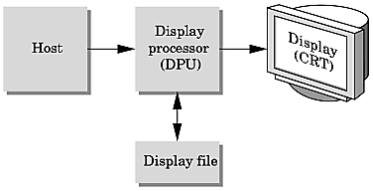


Figure 12: Display-processor Architecture

- The display processor had a limited instruction set, most of which was oriented toward drawing primitives on the CRT.
- The user program was processed in the host computer, resulting in a compiled list of instructions that was then sent to the display processor, where the instructions were stored in a display memory as a *display file or display list*.

For a simple non-interactive application, once the display list was sent to the display processor, the host was free for other tasks, and the display processor would execute its display list repeatedly at a rate sufficient to avoid flicker.

In addition to resolving the bottleneck due to burdening the host, the display processor introduced the advantages of special-purpose rendering hardware.

Two ways of sending graphical entities to a display:

1. *Immediate mode (fundamental mode):* As soon as the program executes a statement that defines a primitive, that primitive is sent to the server for display, and no memory of it is retained in the system. To redisplay the primitive after a clearing of the screen, or in a new position after an interaction, the program must redefine the primitive and then must resend the primitive to the display. For complex objects in highly interactive applications, this process can cause a considerable quantity of data to pass from the client to the server.
2. *Retained mode graphics:* It uses the display lists. Object is defined once, and then its description is put in a display list. The display list is stored in the server and redisplayed by a simple function call issued from the client to the server.

Advantage:

- ✓ It reduces the network traffic.
- ✓ It allows the client to take advantage of any special-purpose graphics hardware that might be available in the graphics server. Thus, in many situations, the optimum configuration consists of a good numerical-processing computer that executes the client program and a special-purpose graphics computer for the server.

Disadvantage:

- ✓ Use of display lists require memory on the server

There is overhead of creating a display list. Although this overhead often is offset by the efficiency of the execution of the display list, it might not be if the data are changing

b. Discuss the following logic operations with suitable example. (i) copy mode (ii) Exclusive OR mode (iii) rubber-band effect (iv) drawing erasable lines. (4 marks)

Copy mode

- By default these pixels replace the corresponding pixels that are already in the frame buffer.
- For example,
  - Initial buffer→ cleared to *black*
  - To draw a blue rectangle of 10×10 pixels, 100 blue pixels are copied into the color buffer replacing 100 black pixels.
- Pixels are copied irrespective of what the color of the buffer is.
- Each bit in source pixel replaces the corresponding bit in the destination pixel.

Exclusive OR mode

Corresponding bits in each pixels are combined using exclusive OR logical operation.

Let *s* and *d* be corresponding bits in source and destination pixel respectively. Then the new destination *d'* is given by,

$$d' = d \oplus s$$

where, denotes the xor operation.

In xor operation, if it is applied twice, it returns the original state. Therefore,

$$d = (d \oplus s) \oplus s$$

That is, if anything is drawn using xor, it can be erased by drawing it second time.

- OpenGL supports all 16 logic modes.
- Copy mode→ GL_COPY → default mode
- To change modes, enable the logic operations using

```
glEnable(GL_COLOR_LOGIC_OP);
```

- To change to xor mode use

```
glLogicOp(GL_XOR);
```

Rubber-band effect

- Rubber band line can be obtained by using same operation in conjunction with a motion callback.

- Save the first location returned from mouse press through mouse callback.
- As mouse moves with the button presses, line segments are drawn from first point to a second point determined through the motion callback, redrawing previous line segment each time to erase it

When mouse button is released, the final line segment is drawn in copy mode from the mouse callback

Drawing erasable lines

Erasable lines can be drawn using xor mode. Consider an OpenGL window of 500×500 pixels and a unit square clipping window with the origin at lower left corner.

Use the mouse to get the first end point and store it in object coordinates as follows.

```
xm=x/500;
ym=(500-y)/500;
```

Obtain second point and draw a line segment using xor mode as follows:

```
xmm=x/500;
ymm=(500-y)/500;
glLogicOp(GL_XOR);
glBegin(GL_LINES);
glVertex2f(xm,ym);
glVertex2f(xmm,ymm);
glLogicOp(GL_COPY);
glEnd();
glFlush();
```

The mode is switched back to copy mode to draw other objects in normal mode.

If another point is entered with the mouse, same line is drawn using xor mode and second line is drawn using first endpoint and mouse input.

Use right mouse button to enter these data and left mouse button to accept the line segment and draw it in its final form as follows:

```
glLogicOp(GL_COPY);
glBegin(GL_LINES);
glVertex2f(xm,ym);
glVertex2f(xmm,ymm);
glEnd();
glFlush();
glLogicOp(GL_XOR);
```

c. Write a note on design techniques for Bezier curves. (6 marks)

Design Techniques Using Bézier Curves

- A closed Bézier curve is generated when we set the last control-point position to the coordinate position of the first control point (Figure 26).

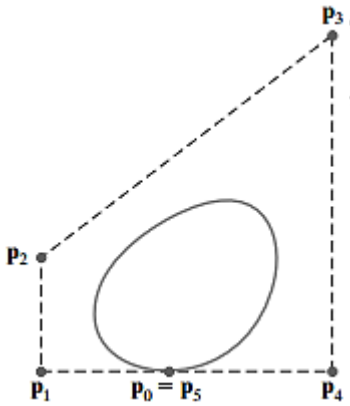


Figure 26: A closed Bézier curve generated by specifying the first and last control points at the same location

- Also, specifying multiple control points at a single coordinate position gives more weight to that position (Figure 27).

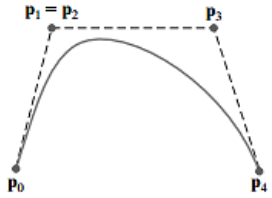


Figure 27: A Bézier curve can be made to pass closer to a given coordinate position by assigning multiple control points to that position.

When complicated curves (with more control points) are to be generated, they can be formed by piecing together several Bézier sections of lower degree.

- Because Bézier curves connect the first and last control points, it is easy to match curve sections (zero-order continuity).
- The tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point.

To obtain first-order continuity between curve sections,

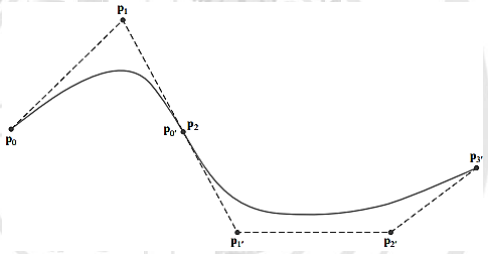


Figure 28: Piecewise approximation curve formed with two Bézier sections.

- Pick control points p_0 and p_1 for the next curve section to be along the same straight line as control points p_{n-1} and p_n of the preceding section (Figure 28).
- If the first curve section has n control points and the next curve section has n Control points, then we match curve tangents by placing control pointp_1' at the position

$$\mathbf{p}_{1'} = \mathbf{p}_n + \frac{n}{n'}(\mathbf{p}_n - \mathbf{p}_{n-1}) \quad \ldots(19)$$

C^2 continuity can be obtained by using the expressions in Equations 17 to match parametric second derivatives for two adjacent Bézier sections.

This establishes a coordinate position for control point p_2, in addition to the fixed positions for p_0 andp_1 needed for C^0 & C^1.