1. Briefly explain how a color lookup table works and why it is used.
   Pages 175-76 in text.
   - Provides a reasonable number of colors without requiring a large frame buffer.
   - Table values can be changed at any time allowing user to try different color combinations without having to change the attributes for graphic primitives.

2. Briefly explain how double buffering works and why it is used.
   Pages 734-35 in text.
   - With single buffer, primitives in a frame that are rendered earlier appear longer on the screen but those drawn towards the end disappear as soon as they are displayed. This leads to flickering of the rendered objects.

3. We are given the following bit patterns
   A = 11000111
   B = 10101010
   Give the resulting patterns when the raster operation xor is applied:
   a. C = (A xor B)
   b. D = (C xor A)

   a. C = 01101101
   b. D = 10101010

4. All "inside" regions are labeled in gray.
(a) a. (2.6,0.1) : 0010
b. (0.1,0.1) : 0000
c. (-0.1,-0.1) : 0101

(b) The line (1010, 0010) can be trivially rejected. See line (b) in above figure.

(c) 0111 is an invalid code for Cohen-Sutherland. All codes for Cohen-Sutherland regions have utmost two bits set to 1.

(d) Cohen Sutherland’s algorithm in 3D
   We need a 6 bit code taking values as follows,
The other steps in Cohen Sutherland’s algorithm remain the same.

5 (d) A 3D line may intersect with utmost 6 faces of the cube and hence may need to be clipped 6 times. The following figure shows one such line, the parts of the line in different regions are numbered.
6) (a) Given 2 end points \((x_1, y_1)\) and \((x_2, y_2)\) with integer coordinates, the number of pixels drawn on the screen \(n_p = \max(|x_2 - x_1|, |y_2 - y_1|) + 1\). We are hence rendering the same number of pixels for lines with different lengths. When we use a bitmask to specify a stipple pattern, the stipple pattern is only used to decide whether a pixel is rendered or not. Different orientations of the line with same values of \(n_p\) would have the same number of pixels turned on. This implies that filled and unfilled regions in inclined lines appear longer than those of a horizontal or vertical line. See the illustration for an example using a bitmask 11111000,

![Illustration](image)

(b) To avoid the above problem, we can use a floating point index into the bitmap. We increment the index by an amount that depends on the length of line drawn. We use the bit in the bitmask nearest to the index to decide the rendering of the current pixel. When the index moves beyond the bitmask it repositions to the start of the bitmap. The DDA line algorithm is modified as follows, Bresenham’s algorithm would be modified similarly.

```c
float index, index_incr, dx, dy; x, y, xincr, yincr;
initialize dx, dy, xincr, yincr as in the line drawing algorithm
index_incr = dist( (x_1, y_1), (x_2, y_2) ) / ( \max(|x_2 - x_1|, |y_2 - y_1|) + 1 );
x = x_1; y = y_1;
index = 0;
for (int i = 0; i < numPixels; i++){
    if ( bitmask [ round(index) ] )
        renderPixel( round(x) , round(y) );
    index = index + index_incr;
    if( index >= 8 )
        index = index % 8;
    x += xincr; y += yincr;
}
```
7) Getting either of the following bugs right is full credit.

a) The floodFill4 function does not do bounds checking to ensure that it always stays inside the window rectangle. We solve this by adding the following lines to floodFill4 after the declaration for currentColor,

```c
if( x < w.minx || y < w.miny || x >= w.maxx || y >= w.maxy)
    return;
```

b) The floodFill function should also do the following before assigning currentColor as otherwise it would go into an infinite loop when the function is called with same fill and interior colors.

```c
if( fillColor == interiorColor )
    return;
```

8) A quaternion \( q = (s,v) \) has the following property

\[
    s = \cos \left( \frac{\theta}{2} \right) \\
    v = \sin \left( \frac{\theta}{2} \right) \mathbf{u}
\]

(a) To determine angle of rotation \( \theta \) consider

\[
    \theta = 2 \cos^{-1}(s)
\]

Note: There are multiple solutions for \( \theta \).

(b) To determine the unit vector in the direction of axis of rotation \( \mathbf{u} \),

\[
    \mathbf{u} = \frac{\mathbf{v}}{||\mathbf{v}||} \quad \text{as we know} \quad \mathbf{v} \quad \text{is a vector along the axis of rotation, we only need to normalize it.}
\]
a) \( TS \neq ST \)

\[
TN = \begin{bmatrix}
    s & 0 & 0 & t_x \\
    0 & s & 0 & t_y \\
    0 & 0 & s & t_z \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

whereas \( ST = \begin{bmatrix}
    s & 0 & 0 & \alpha t_x \\
    0 & s & 0 & \alpha t_y \\
    0 & 0 & s & \alpha t_z \\
    0 & 0 & 0 & 1
\end{bmatrix} \)

b) \( SR = RS \)

Given \( R = \begin{bmatrix} R_g & 0 \\ 0 & 1 \end{bmatrix} \), we have \( SR = \begin{bmatrix} \alpha P_g & 0 \\ 0 & 1 \end{bmatrix} = RS \), where \( R_g \) = An orthonormal 3\( \times \)3 matrix.

c) \( S_1 S_2 = S_2 S_1 \)

\[
S_1 S_2 = \begin{bmatrix}
    s_1 s_2 & 0 & 0 & 0 \\
    0 & s_1 s_2 & 0 & 0 \\
    0 & 0 & s_1 s_2 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix} = S_2 S_1
\]

d) For general 3D rotation \( R_1 R_2 \neq R_2 R_1 \). Ex, 90° rotation about x axis followed by 90° rotation about y axis causes the new z axis to be aligned with the original x axis but when the sequence is reversed the new z axis coincides with -ve x axis.

2D rotations do commute as was shown in HW.

e) \( T_1 T_2 = T_2 T_1 \)

\[
T_1 T_2 = \begin{bmatrix}
    1 & 0 & 0 & t_{1x} + t_{2x} \\
    0 & 1 & 0 & t_{1y} + t_{2y} \\
    0 & 0 & 1 & t_{1z} + t_{2z} \\
    0 & 0 & 0 & 1
\end{bmatrix} = T_2 T_1
\]
10)

Given
\[
\begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
=
\begin{bmatrix}
x + ay \\
- y \\
1
\end{bmatrix}
\]

\[\Rightarrow m_{11}x + m_{12}y + m_{13} = x + ay \quad \cdots 1\]
\[m_{21}x + m_{22}y + m_{23} = -y \quad \cdots 2\]

Comparing coefficients of \(x, y\) and constant on both sides, we get

\[M = \begin{bmatrix}
1 & a & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix}\]

We can decompose \(M\) as follows,

\[M = \begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
1 & a & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}\]

= \text{Reflect}_x \times \text{Shear}(a,0)
Clipping using hierarchical bounding boxes. The scene objects (Ei, Ci, Di, Ei) need to be clipped against the clipping window.

These edges are traversed in the hierarchy as their corresponding bounding boxes intersect the clipping window.

These edges are ignored as the clipping window does not intersect the bounding box corresponding to these edges.
In a hierarchical bounding box representation, the bounding boxes are stored as nodes in a tree as shown in the above figure. A parent child relationship is defined between two nodes that share an edge in the tree. The parent bounding box is the bounding box of all its children nodes. When we need to clip the scene against a clipping window we begin at the root node and do the following: For each child of the current node, check if its bounding box intersects the clipping window. If true, recursively traverse the subtree for the child node.

12)

1. given $P_1, C_1, P_2, C_2, P_6$ compute $P_5, C_5$
2. given $P_2, C_2, P_3, C_3, P_6$ compute $P_4, C_4$
3. given $P_4, C_4, P_5, C_5$ and $P_6$, compute $C_6$

\[
P_3 (1,6) \\
C_2=(2,0,1)
\]

\[
P_4=2/3*P_3+1/3*P_2 = (2, 14/3) \\
C_4=2/3*C_3+1/3*C_2 = (4/3, 1/3, 4/3)
\]

\[
P_6=3/8*P_4+5/8*P_5 = (2, 3) \\
C_6=3/8*C_4+5/8*C_5 = (1/2+5/6, 1/8+25/24, 1/2+5/12) = 100*(8/6, 28/24, 11/12) = (400/3, 350/3, 275/3) = (133.3, 116.6, 91.6)
\]

\[
P_5=(2,2) \\
C_5=2/3*C_1+1/3*C_2 = (4/3, 5/3, 2/3)
\]