Flood fill Algorithm ...

BASIC CONCEPT

- Flood fill algorithm is useful in cases where there no single color boundary for the polygon, i.e. the boundary has multiple colors.
- In flood fill algorithm instead of filling color till you encounter a specific boundary color you just fill the pixels with default color.
- It is used in the "bucket" fill tool of paint programs to fill connected, similarly-colored areas with a different color

2. Flood-Fill Algorithm

1. What is Flood-Fill Algorithm?

- The **flood-fill algorithm** is used to fill a connected region with a specific color.
- It works like the **bucket tool in MS Paint**—when you click on an area, it fills that region with a new color.

How it Works

- Start from a given **seed point** (where the user clicks).
- Check the color of the current pixel:
- If it's the **same as the original color**, change it to the **new fill color**.
- If it's already the new color or a boundary, **stop**..
- Recursively **move to the neighboring pixels** (left, right, up, and down) and repeat the process.

Methods of Implementation:

- There are two methods in which the Flood-Fill Algorithm can be implemented:

1. 4-connected pixel
2. 8-connected pixel

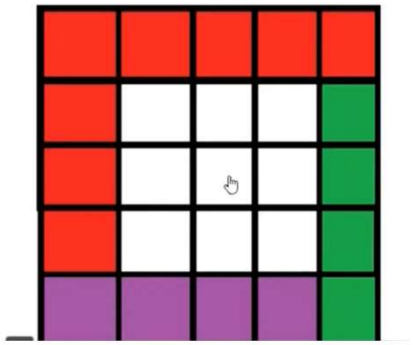
•Moves in four directions: left, right, up, down.

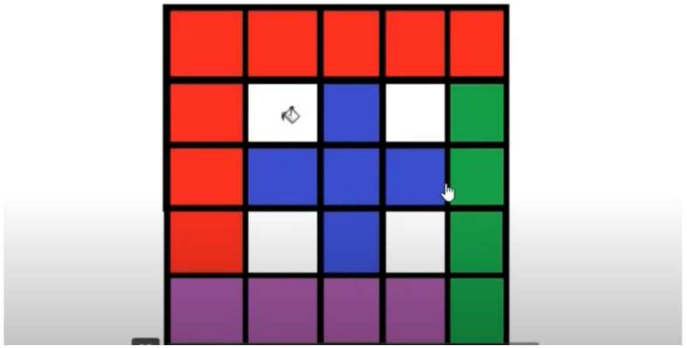
Moves in eight directions: left, right, up, down, and diagonals.

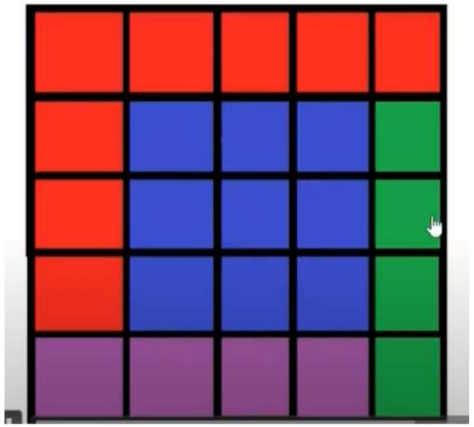
4-connected pixel

- In 4 connected pixel method you check 4 pixels adjacent to the current pixel, namely towards the North(top), South(bottom), west(left), East(right).
- So you fill the area with 4-connected pixel method by following these steps

8-connected pixel filling



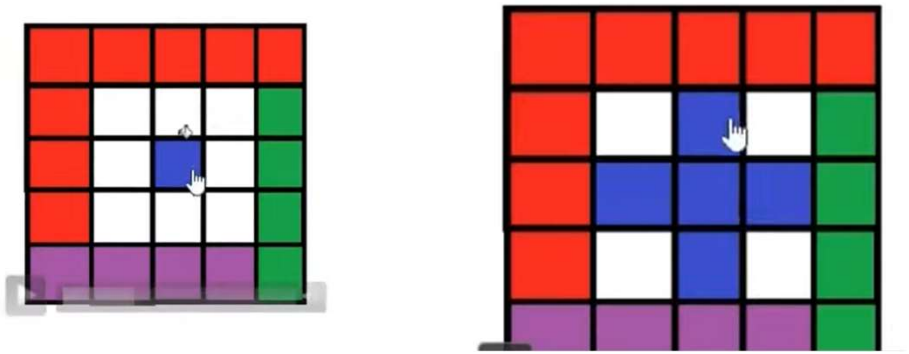




```
void floodFill(int x, int y, int fillColor, int interiorColor) {
  int color;
  getPixel(x,y,color)
  if (color==interiorColor) {
    setPixel(x,y,fillColor);
    floodFill(x+1,y,fillColor,interiorColor);
    floodFill(x-1,y,fillColor,interiorColor);
    floodFill(x,y+1,fillColor,interiorColor);
    floodFill(x,y-1,fillColor,interiorColor);
  }
```

Problem in 4-connected

- There is a problem with this technique that 4-connected pixels technique cannot fill all the pixels.



8-connected pixel

- To solve the problems that can occur in 4-connected pixel method, a new method was introduced called 8-connected pixel.
- SO in 8-connected pixel method we instead of filling just 4 adjacent pixels we fill also the adjacent diagonal pixels positions, such as

  $(x + 1, y + 1)$.

. Flood Fill Algorithm

- The fill process stops when it reaches a **different color from the starting pixel**.
- Used for filling a region with the same color.
- The algorithm checks if the current pixel’s color is **different from the original seed pixel's color** before filling.

General Stopping Criteria:

1. **Pixel Already Filled**
    - The algorithm stops at a pixel if it is already filled with the new color.

- Boundary fill: expand and fill region until you reach boundary color

- •Flood fill: expand and fill region while you find interior color

Difference Between Flood-Fill and Boundary-Fill

Feature	Flood-Fill	Boundary-Fill
Stops at	Different color	A specific boundary color
Use case	Open and closed regions	Only closed regions
Connectivity	4-connected or 8-connected	4-connected or 8-connected

Step 1: First initialize the 4 values namely **x, y, Fill-color & Default-color**. Where x and y are co-ordinate positions of the initial interior pixel we start the flood fill algorithm & Fill-color is the color we want to fill and Default-color is the default color of the interior pixel

Step 2: If the color of node is not equal to default-color, return.

Step 3: Set the color of node to replacement-color.

Step 4: Set the color of node *to the south of node* to replacement-color.

Step 5: Set the color of node *to the north of node* to replacement-color.

Step 6: Set the color of node *to the east of node* to replacement-color.

Step 7: Set the color of node *to the west of node* to replacement-color.

Step 8: Exit.

Flood Fill Algorithm	Boundary Fill Algorithm
Flood fill colors an entire area in an enclosed figure through interconnected pixels using a single color	Here area gets colored with pixels of a chosen color as boundary this giving the technique its name
So, Flood Fill is one in which all connected pixels of a selected color get replaced by a fill color.	Boundary Fill is very similar with the difference being the program stopping when a given color boundary is found.
A flood fill may use an unpredictable amount of memory to finish because it isn't known how many sub-fills will be spawned	Boundary fill is usually more complicated but it is a linear algorithm and doesn't require recursion
Time Consuming	It is less time Consuming

2D Transformations.

Geometric Transformations.

- **Transformations play an important role in computer graphics to reposition the graphics on the screen and change their size or orientation.**
- **The geometrical change of an object from a current state to modified state.**

- **To manipulate the initially created object and to alter and display the modified object without having to redraw it.**

  Eg : designing, animation
- **Two types:-**
  - **Object Transformation**
    - **Changes orientation, size and shape of object**
    - **Alter the coordinates descriptions an object**
    - **Translation, rotation, scaling, shear, reflection, etc.**
    - **Coordinate system unchanged**
  - **Coordinate transformation**
    - **Produce a different coordinate system**

Matrix Representation

- Column-vector representation for a point.
- Pre-multiplication of the transformation – it appears before the point to be transformed in the equation.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Dy \end{bmatrix}$$

- Advantages of Matrix
  - More convenient organization of data
  - More efficient processing
  - Enable the combination of various concatenations

Why Matrix representations?

- Matrix representations are used in transformations because they provide a structured, efficient, and scalable way to perform linear transformations such as rotation, scaling, translation, and reflection in various dimensions.

1. Compact Representation

- A transformation can be expressed concisely using a matrix rather than writing out multiple equations.
- Example: A 2D rotation by angle θ is represented as:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

2. **Efficiency in Computation**

- Matrix multiplication enables efficient computation of transformations, especially in graphics, physics simulations, and engineering applications.
- Instead of transforming each point separately with a set of equations, we can multiply all points by a single matrix.

3. **Composition of Transformations**

- Multiple transformations (e.g., rotation, scaling, and translation) can be combined into a single matrix.
- This avoids redundancy and improves performance, especially in computer graphics.

2D Translation

A translation is a rigid body transformation that moves objects without deformation. That is every point on the object is translated by the same amount.

- Line segment - add translation vector to each of line end points and redrawing line between new end point positions.
- Polygons - add translation vector to coordinate position of each vertex and regenerate polygon using new set of vertex coordinates and current attribute settings
- Circle or Ellipse - add translation vector to center coordinate and redraw it in new location
- Curves (splines) - add translation vector to coordinate positions of object and reconstruct curve paths using new set of coordinates

2D Translation

A translation moves all points in an object along the same straight-line path from original position (x, y) to new position (x', y')

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

Translation equations are:

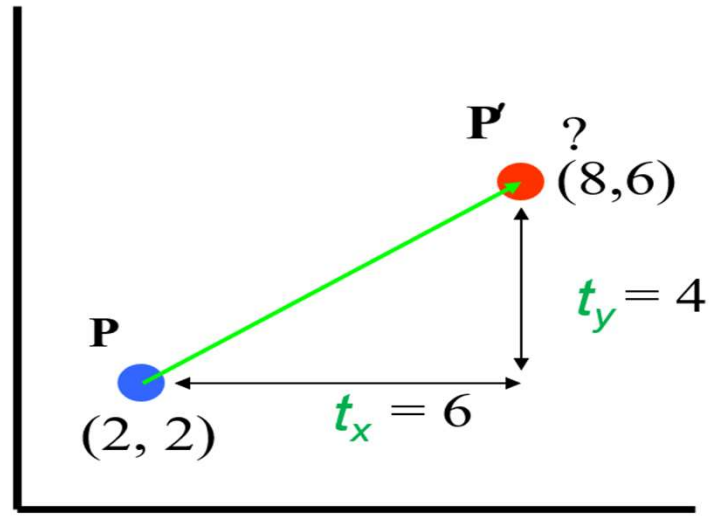
$$x' = x + t_x$$

$$y' = y + t_y$$

or in matrix form:

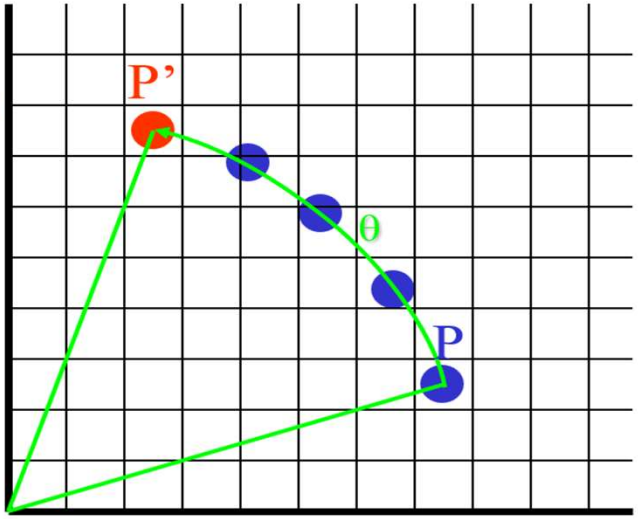
$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



2D Rotation

- Rotation - a rigid body transformation that repositions all points in an object along a circular path in the xy plane centered at the pivot point.
- A rotation angle **θ** and position (x_r, y_r) of rotation point or pivot point about which object is to be rotated is specified.
- θ can be clockwise (-ve) or counterclockwise (+ve).
- Rotation is about a rotation axis perpendicular to xy plane and passes through the pivot point.



2D Rotation

- First, assume that pivot is at the origin (0,0).
- Review Trigonometry

=> $\cos\phi = x/r$, $\sin\phi = y/r$

$x = r\cos\phi$, $y = r\sin\phi$

=> $\cos(\phi+\theta) = x'/r$

$x' = r.\cos(\phi+\theta)$

$x' = \underline{r\cos\phi}\cos\theta - \underline{r\sin\phi}\sin\theta$

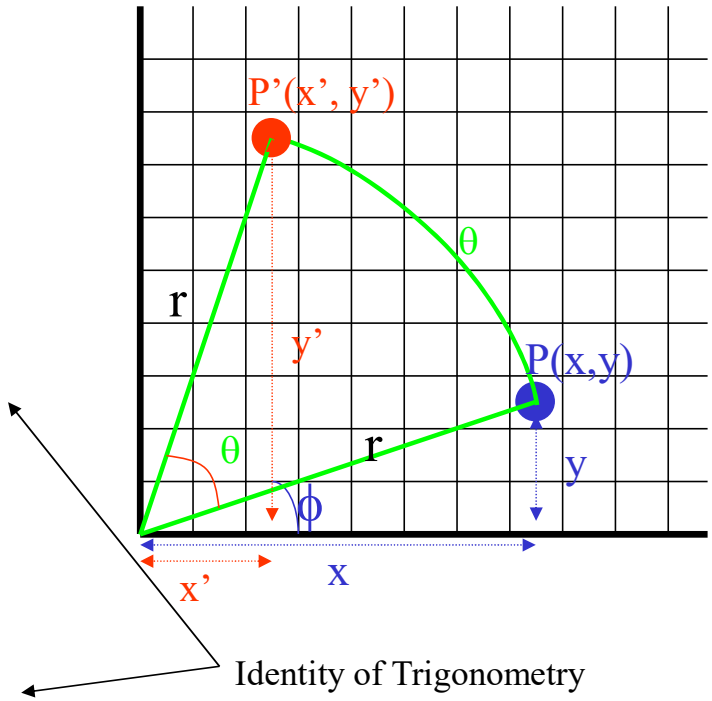
$x' = x\cos\theta - y\sin\theta$

=> $\sin(\phi+\theta) = y'/r$

$y' = r.\sin(\phi+\theta)$

$y' = \underline{r\cos\phi}\sin\theta + \underline{r\sin\phi}\cos\theta$

$y' = x\sin\theta + y\cos\theta$



Identity of Trigonometry

Cos (A+B) Formula

$$\cos(A + B) = \cos A \cos B - \sin A \sin B$$

Sin (A + B) Formula

$$\sin(A + B) = \sin A \cos B + \cos A \sin B$$

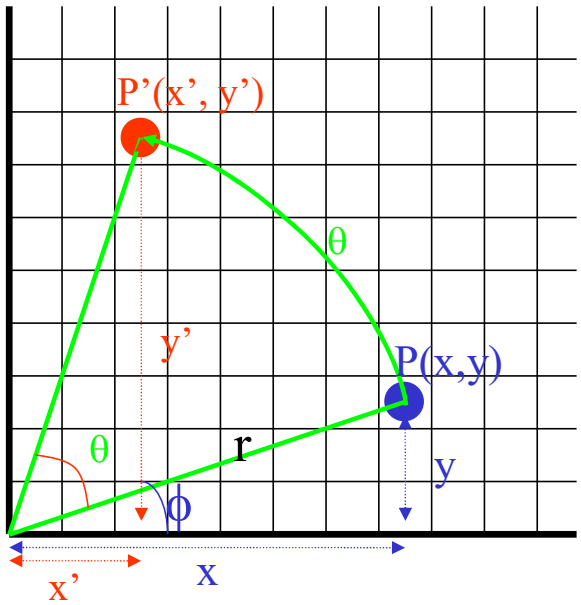
2D Rotation

- Rotation equations:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

- or in matrix form:

$$P' = R \bullet P$$



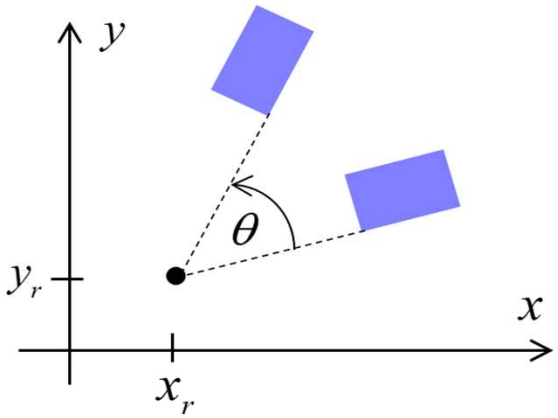
- Rotation matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

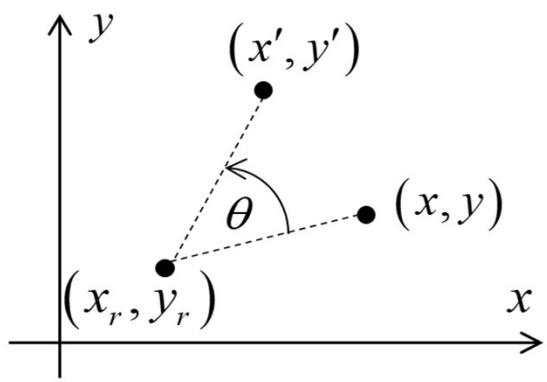
- For a rotation matrix transpose is got by simply changing sign of sine terms

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

2D Rotation



Rotation in angle θ about a pivot (rotation) point (x_r, y_r).



$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

$$\mathbf{P}' = \mathbf{P}_r + \mathbf{R}\cdot(\mathbf{P} - \mathbf{P}_r)$$

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

2D Rotation

Example - Find the transformed point, P', caused by rotating P= (5, 1) about the origin through an angle of 90°.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cdot \cos\theta - y \cdot \sin\theta \\ x \cdot \sin\theta + y \cdot \cos\theta \end{bmatrix}$$

$$= \begin{bmatrix} 5 \cdot \cos 90 - 1 \cdot \sin 90 \\ 5 \cdot \sin 90 + 1 \cdot \cos 90 \end{bmatrix}$$

$$= \begin{bmatrix} 5 \cdot 0 - 1 \cdot 1 \\ 5 \cdot 1 + 1 \cdot 0 \end{bmatrix}$$

$$= \begin{bmatrix} -1 \\ 5 \end{bmatrix}$$

2D Scaling

- Scaling alters the size of an object and involves two scale factors, s_x and s_y for the x and y coordinates respectively.
- Scales are about the origin (0,0).
- Scaling equations:

$$x' = s_x \bullet x$$

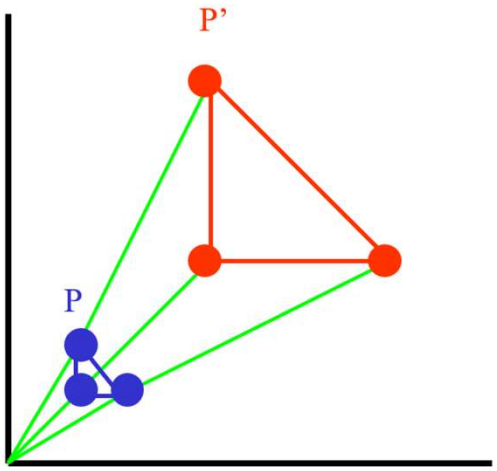
$$y' = s_y \bullet y$$

or in matrix form:

$$\mathbf{P'} = \mathbf{S} \bullet \mathbf{P}$$

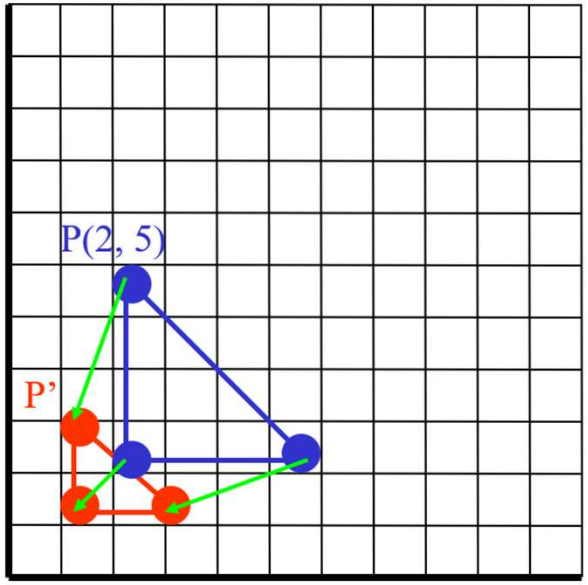
Scale matrix as:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$



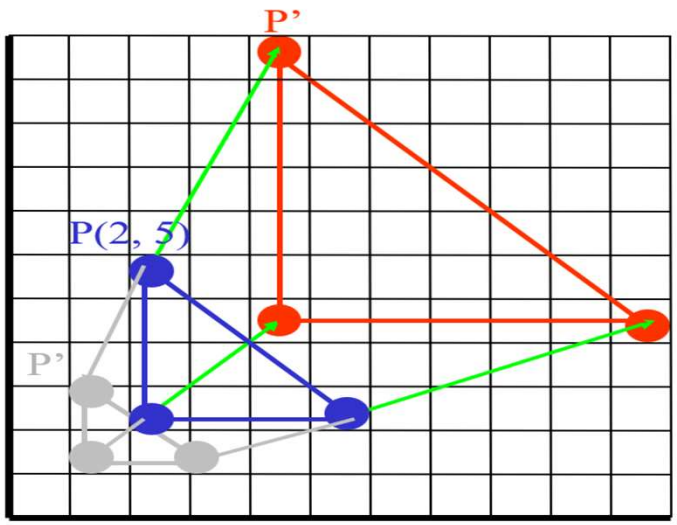
Scaling

- If the scale factors are in between 0 and 1 ➔ the points will be moved closer to the origin ➔ the object will be smaller.
- Example :
  - P(2, 5), Sx = 0.5, Sy = 0.5
  - Find P` ?



Scaling

- If the scale factors are in between 0 and 1 ➔ the points will be moved closer to the origin ➔ the object will be smaller.
- Example :
  - P(2, 5), Sx = 0.5, Sy = 0.5
  - Find P' ?



- If the scale factors are larger than 1 ➔ the points will be moved away from the origin ➔ the object will be larger.

- Example :
  - P(2, 5), Sx = 2, Sy = 2
  - Find P' ?