Chapter 3 - Geometrical Transformations

1. Line transformations in 2D

Eg:- $\begin{cases} x' = 2x \\ y' = x + y \end{cases} \longrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

or more commonly in Computer Graphics :-

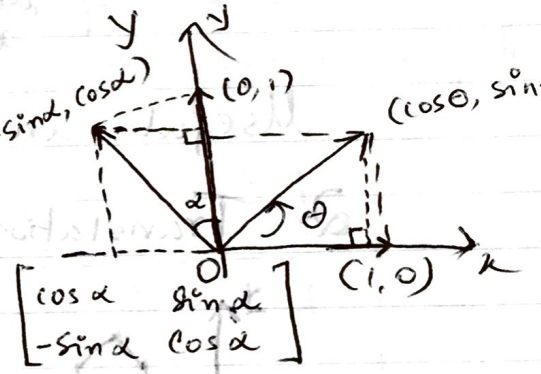
$$[x' \quad y'] = [x \quad y] \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

Rows of 2×2 matrix are the images of unit vectors (1,0) & (0,1) i.e

$$[1 \quad 0] \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} = [2 \quad 1], \quad [0 \quad 1] \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} = [0 \quad 1]$$

1.1 Rotation

Rotate about O



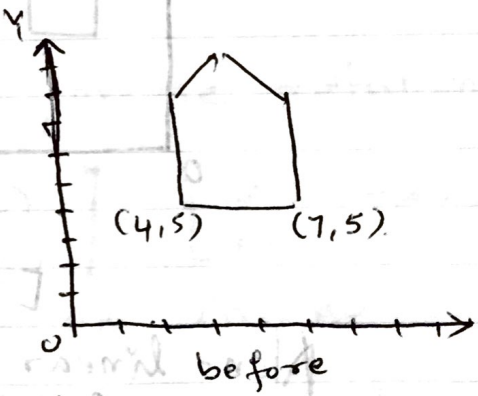
So, $(\cos\alpha, \sin\alpha)$ is image of $(1,0)$, $(-\sin\alpha, \cos\alpha)$ is image of $(0,1)$,

Rotation matrix : $[x' \quad y'] = [x \quad y] \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$

1.2 Scaling

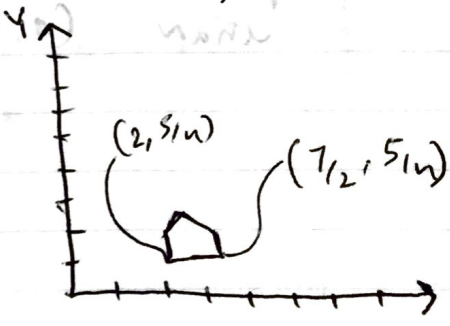
Scale with reference to O

$\begin{cases} x' = S_x x \\ y' = S_y y \end{cases}$ when $S_x = \frac{1}{2}$, $S_y = \frac{1}{4}$



Scaling matrix :

$$[x' \quad y'] = [x \quad y] \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$



Reflection as special cases :

$S_x = 1, S_y = -1$: reflection about x-axis

$S_x = -1, S_y = 1$: reflection about y-axis

- Given a matrix, how to rotate along a point at a particular degree.
- Apply these transformations.

$S_x = -1$, $S_y = -1$: reflection about origin O.

1.3 Shearing.

Along x-axis:

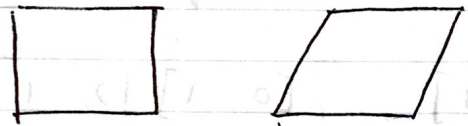
$$\begin{cases} x' = x + ay \\ y' = y \end{cases}$$

$$[x'\ y'] = [x\ y] \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix}$$

Along y-axis:

$$\begin{cases} x' = x \\ y' = bx + y \end{cases}$$

$$\begin{cases} [x'\ y'] = \\ [x\ y] \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \end{cases}$$

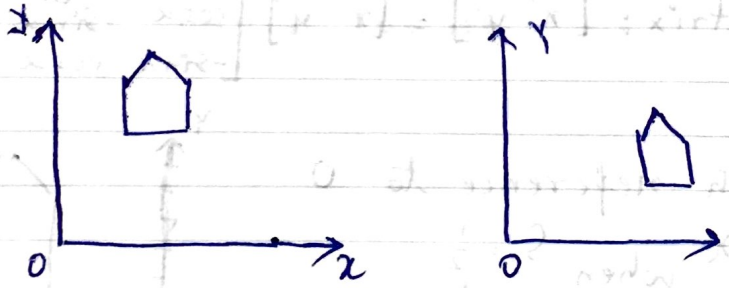


shearing along x-axis

shearing along y-axis.

Useful in making character italic :-

2. Translation. (Non-linear)



Shifting all points in a constant distance.

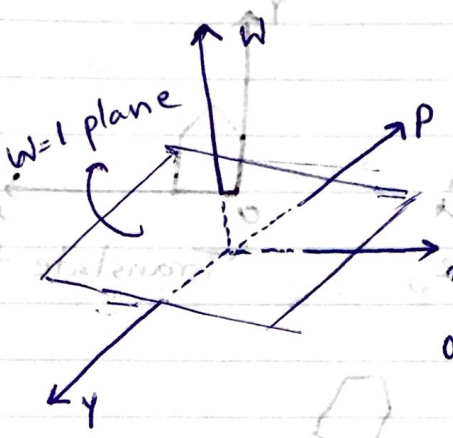
$$\begin{cases} x' = x + a \\ y' = y + a. \end{cases}$$

$$[x'\ y'] = [x\ y] + [a\ b]$$

Non-linear since image of origin is (a, b) rather than (0, 0)

3. Homogenous Co-ordinates.

To make all the transformations as matrix multiplications in order to combine them, we introduce one more dimension:



Each point has many different homogeneous co-ordinate representa

Eg:- (2,3,6) & (4,6,12) are the same point — or is the multiple of other

All the previous linear transformations can be written as 3×3 matrices for them to operate toget

Eg: rotation:

$$[x' \quad y'] = [x \quad y] \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4. Inverse transformations:-

If rotation matrix is R, inverse rotation through $-\alpha$ is $[x' \; y'] = [x \; y] R^{-1}$

where $R^{-1} = \begin{bmatrix} \cos(-\alpha) & \sin(-\alpha) \\ -\sin(-\alpha) & \cos(-\alpha) \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$

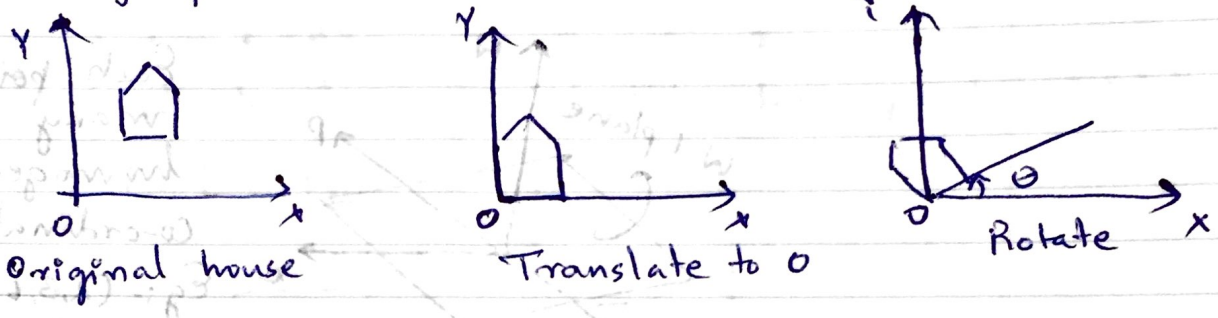
In general, if matrix A has an inverse A^{-1}, then: $AA^{-1} = A^{-1}A = I$.

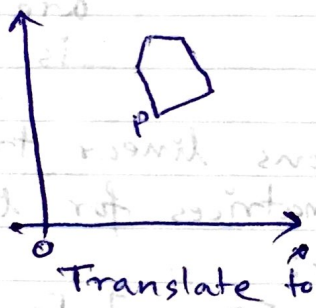
For 2D, the identity matrix is: $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

called Identity matrix.

5. Composition of 2D Transformations :-

Consider rotating an object about an arbitrary point :-





3 Steps :-

① Translate such that P is at origin O :

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_p & -y_p & 1 \end{bmatrix}$$

② Rotate then P, about O :

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

③ Translate back to O :

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_p & y_p & 1 \end{bmatrix}$$

Net transformation is:

$$R = T^{-1} R_\theta T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_p & -y_p & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_p & y_p & 1 \end{bmatrix}$$

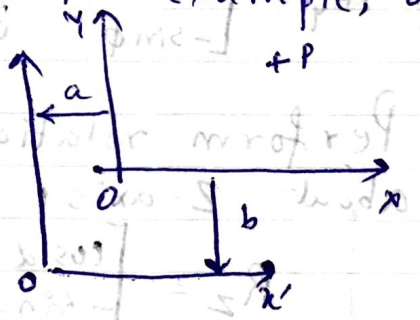
$$= \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ -x_p\cos\theta + y_p\sin\theta + x_p & -x_p\sin\theta - y_p\cos\theta + y_p & 1 \end{bmatrix}$$

So, $[x' \; y' \; 1] = [x \; y \; 1] \, R$ describes the desired rotation.

6. Transformations as a change in co-ordinate System :-

An alternative but equivalent way of thinking about a transformation is as a change of co-ordinate systems. For example, a translation

$$\begin{cases} x' = x + a \\ y' = y + b \end{cases}$$



reverse: +ve degree becomes -ve, + position becomes -ve

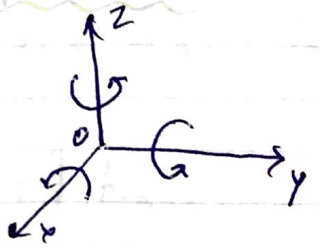
Can be thought of a shift of co-ordinate system by (a, b).

The same principle applies to other transformations.

7. 3D Transformations:

1.1. Rotation about a coordinate axis

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$



$$R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

7.2 Rotation about an arbitrary axis:

(A) To make z-axis merge with $\bar{V}$:

① rotate about z-axis for θ

② rotate about y-axis for ϕ

$$R_z^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y^{-1} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}$$

(B) Perform rotation about $\bar{V}$, now same as about z-axis:

$$R_z = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(C) Rotate $\bar{V}$-axis back to its original position (inverse transformation of (A))

$$R_y = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combined transformation :-

$$R = \underbrace{R_z^{-1} R_y^{-1}}_{(A)} \; \underbrace{R_{\bar{v}}}_{(B)} \; \underbrace{R_y R_z}_{(C)} \quad \text{in } [x' \; y' \; z'] = [x \; y \; z]$$

Now, if the rotation is about $\bar{v}$ from any point $A(a_1, a_2, a_3)$

Since translation is involved, use homogeneous co-ordinates to translate from A to O:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_1 & -a_2 & -a_3 & 1 \end{bmatrix}$$

Then, apply R in homogeneous co-ordinate, Call new matrix R^*

$$R^* = \begin{bmatrix} & & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, translate from O back to A:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix}$$

So, generalized rotation matrix R_{gen} is

$$R_{gen} = T^{-1} R^* T$$

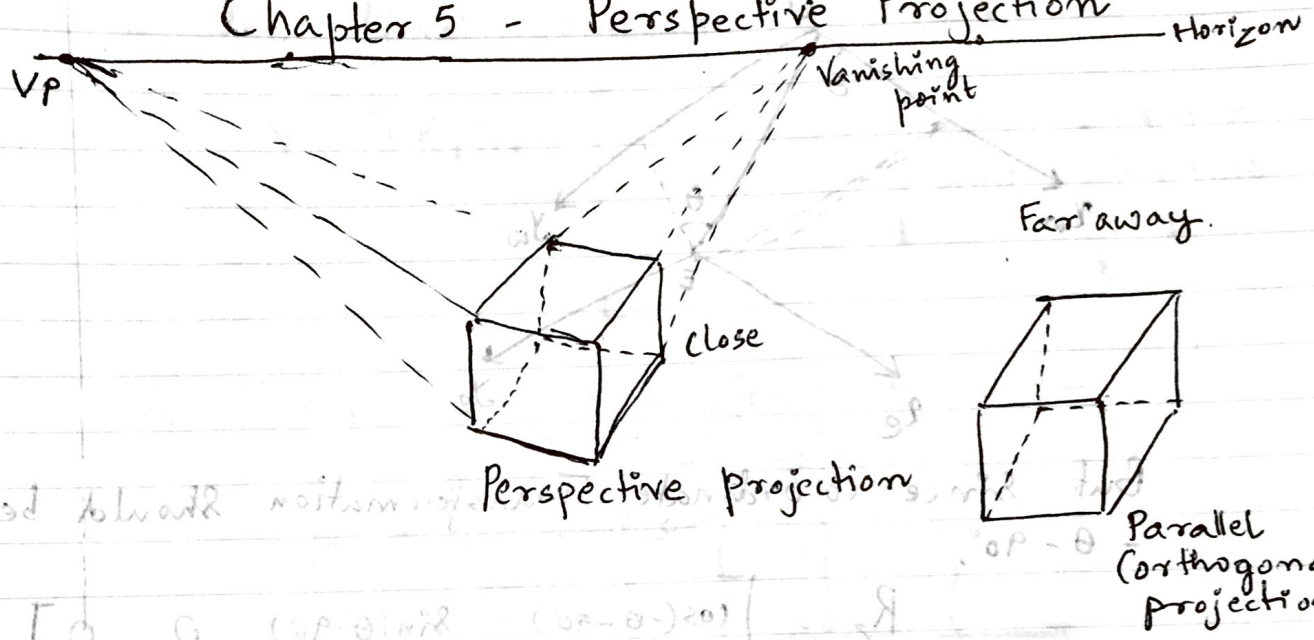
in $[x' \; y' \; z' \; 1] = [x \; y \; z \; 1] R_{gen}$.

For efficiency, R_{gen} is usually calculated beforehand, then many points could be rotated using R_{gen}.

* Concepts : Vanishing points, Horizon, viewpoint, orthographic (parallel) projection, perspective projection.

*

Chapter 5 - Perspective Projection



Distance to eye : View point.

World Co-ordinates (x_w, y_w, z_w) 3D

↓ Viewing Transformation.

Eye Co-ordinates (x_e, y_e, z_e) 3D

↓ Perspective Transformation.

Screen Co-ordinates (X, Y) 2D

1. Viewing Transformation

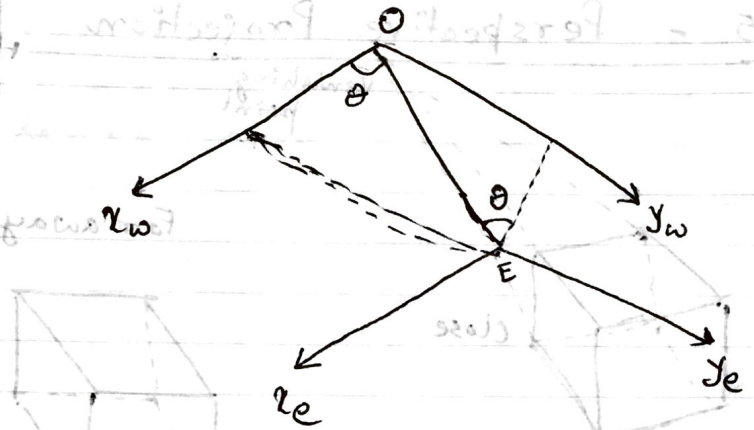
Transformations from world co-ordinates eye co-ordinates.

① Move the origin to E

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_E & -y_E & -z_E & 1 \end{bmatrix}$$

② Rotate about Z axis

Rotation angle we see is $\theta + 90$



But since co-ordinate Transformation should be $-\theta - 90°$,

$$R_z = \begin{bmatrix} \cos(-\theta-90°) & \sin(\theta-90°) & 0 & 0 \\ -\sin(-\theta-90°) & \cos(-\theta-90°) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, y-axis, Z-axis & OE are on the same plane.

(3) Rotate about x-axis.

Rotate to make Z axis, the same direction as OE, so angle ϕ. Again since co-ordinate system transformation, it should be $-\phi$.

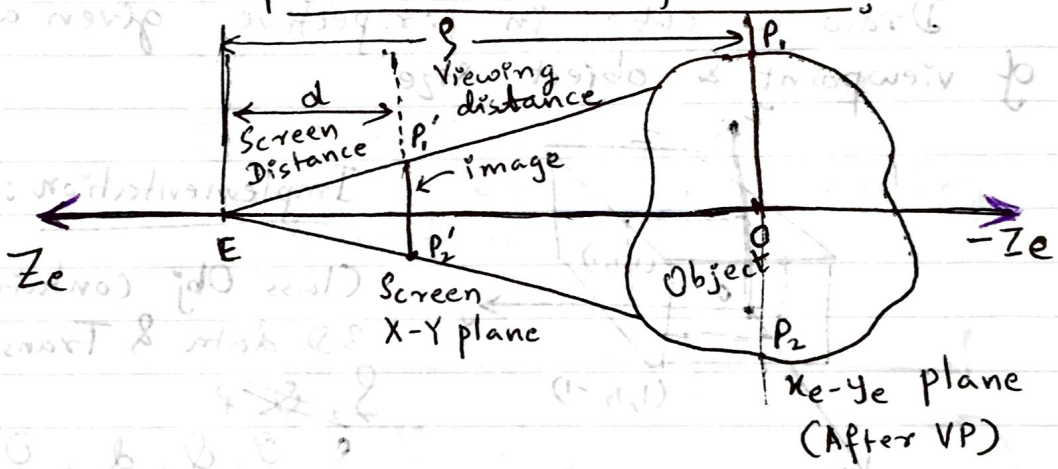
$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-\phi) & \sin(-\phi) & 0 \\ 0 & -\sin(-\phi) & \cos(-\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So, the final composite matrix :

$$V = TR_zR_x = \begin{bmatrix} -\sin\theta & -\cos\phi\sin\theta & \sin\phi\cos\theta & 0 \\ \cos\theta & -\cos\phi\sin\theta & \sin\phi\sin\theta & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & \rho & 1 \end{bmatrix}$$

2. Perspective Transformation :-



Changing ρ can change the perspective (After VP)

— Parallel projection if $\rho = \infty$

Due to similar triangles EQP₁' & EOP₁. (EQP₂' & EOP₂)

$$\frac{P_1'Q}{EQ} = \frac{P_1O}{EO}$$ which applies to X-xe & Y-ye relationships.

$$\frac{X}{d} = \frac{x_e}{-z_e}, \text{ so } X = -d\frac{x_e}{z_e}$$

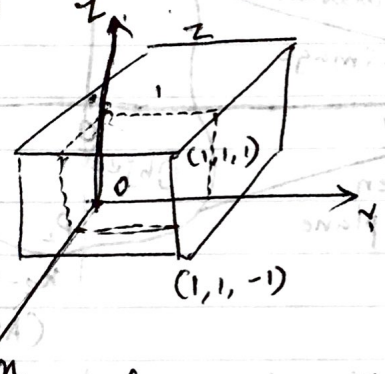
Similarly, $Y = -d\frac{y_e}{z_e}$

By the same principle,

$$\frac{d}{\S} = \frac{\text{image size}}{\text{Object size}}$$

3. An Example (CubePers.java)

Draw a cube in perspective, given a distance of viewpoint & object size.



Implementation:

Class Obj contains 3D data & Transformations

~~§ Θ P~~

§, θ, φ, d, Object size, elements of view Transformation

World co-ordinates for the cube (3D)

$$\text{Object size} = \sqrt{2^2+2^2+2^2} = \sqrt{12}$$

$$\S = 5 * \text{Object size}$$

Transformations:

① Viewing Transformations

$$[x_e \quad y_e \quad z_e \quad 1] = [x_w \quad y_w \quad z_w \quad 1]$$

$$= \begin{bmatrix} \sin\theta & -\cos\phi\cos\theta & \sin\phi\cos\theta & 0 \\ \cos\theta & -\cos\phi\sin\theta & -\cos\phi\sin\theta & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & -\S & 1 \end{bmatrix}$$

(2) Perspective transformation

$$X_s = -d \frac{x_e}{z_e}$$

$$Y_s = -d \frac{y_e}{z_p}$$

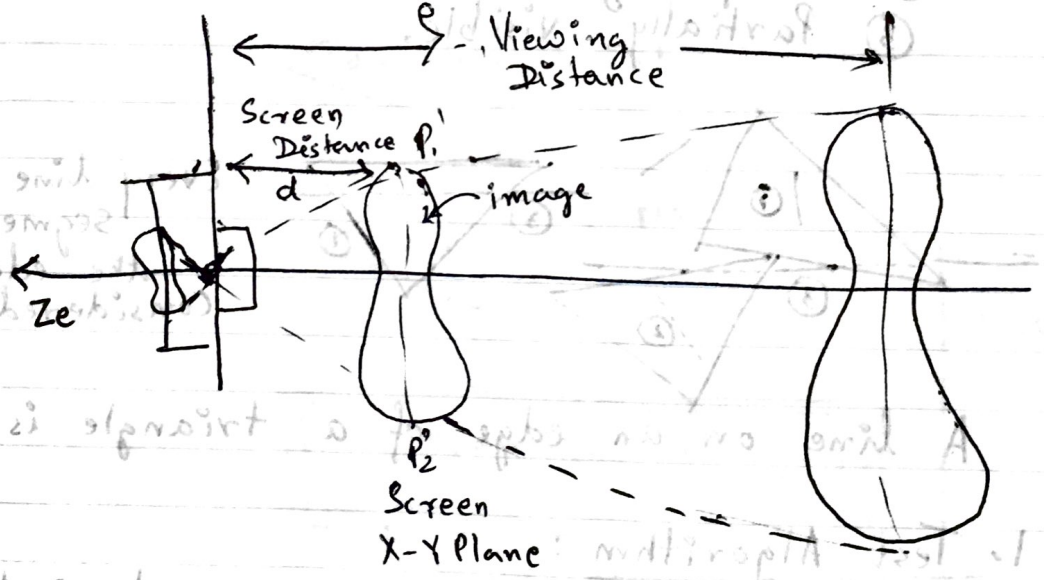
Draw Cube (paint)

(1) Find center of Screen co-ordinate system

(2) $d = \frac{\text{image size}}{\text{Object size}}$

(3) Transformations

(4) Draw cube edges according to Screen co-ordinates.



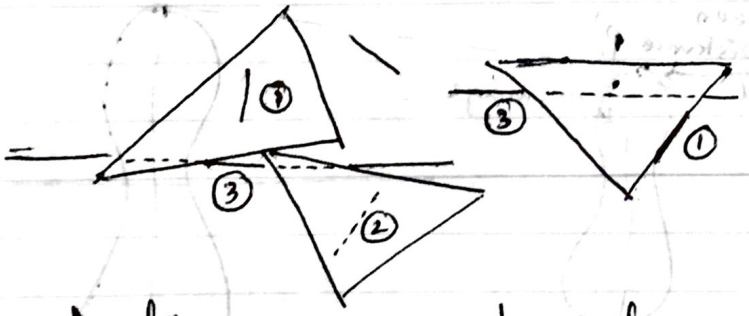
Chapter 6 : Hidden-Line Elimination

For line drawings, we need to remove invisible line segments.

A line drawing can be considered as a set of line segments and a set of faces. Faces can be triangulated into triangles. So the problem of hidden-line elimination becomes the test: "whether each line segment is blocked by one or more triangles".

3 Possibilities :

1. Line segment is completely visible
2. Completely invisible
3. Partially visible.



Every line segment on the edge is considered visible.

A line on an edge of a triangle is visible.

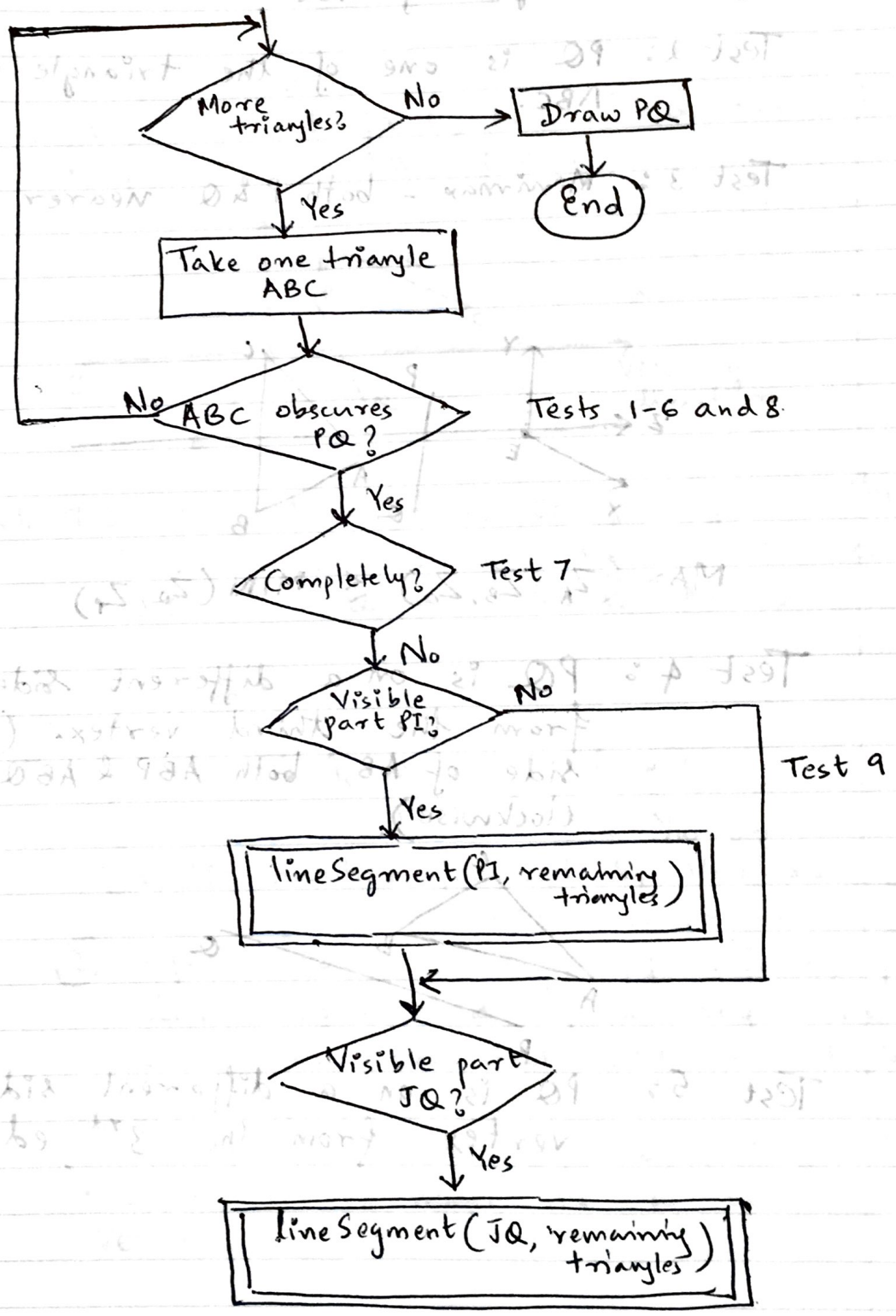
1. Test Algorithm :

Given a set of line segments and a set of triangles, then we work on each line segment PQ using method "LineSegment".

* 9 Test cases

* Input File format

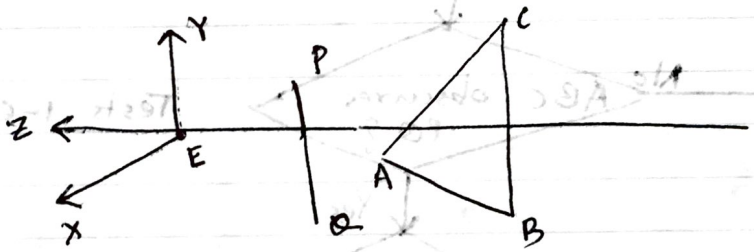
lineSegment (PQ, a set of triangles)



Test 1: Minimax – both P & Q on the left or right of ABC

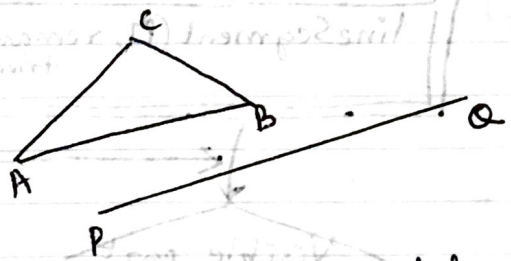
Test 2: PQ is one of the triangle edges of ABC.

Test 3: Minimax – both P & Q nearer than ABC.

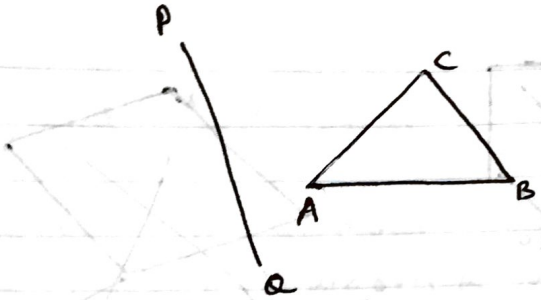


$$\text{MAX}(Z_A, Z_B, Z_C) \leq \text{MIN}(Z_Q, Z_P)$$

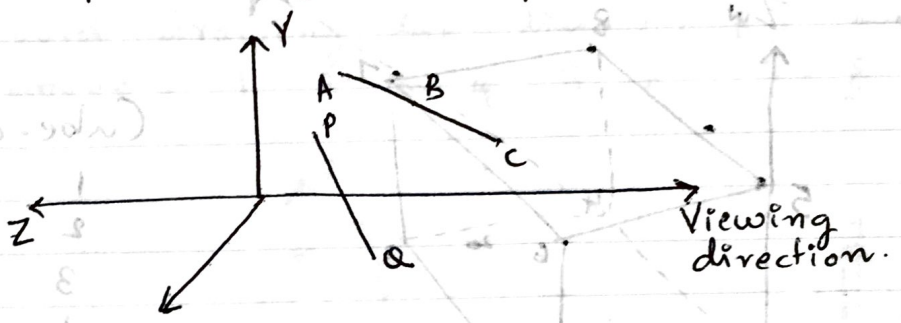
Test 4: PQ is on a different side of an edge from the third vertex. (if on one side of AB, both ABP & ABQ are clockwise).



Test 5: PQ is on a different side of a vertex from the 3rd edge.



Test 6 : Test 3 fails, but P & Q are still in front of ABC plane



Test 7 : PQ behind ABC, completely invisible. Case Inside Triangle method to test if both P & Q are inside ABC).

Input File format

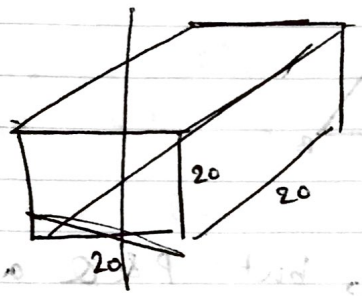
Two parts: ① A vertex per line :

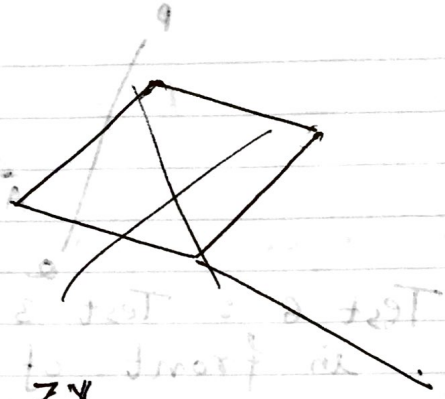
Vertex # x_w y_w y_z

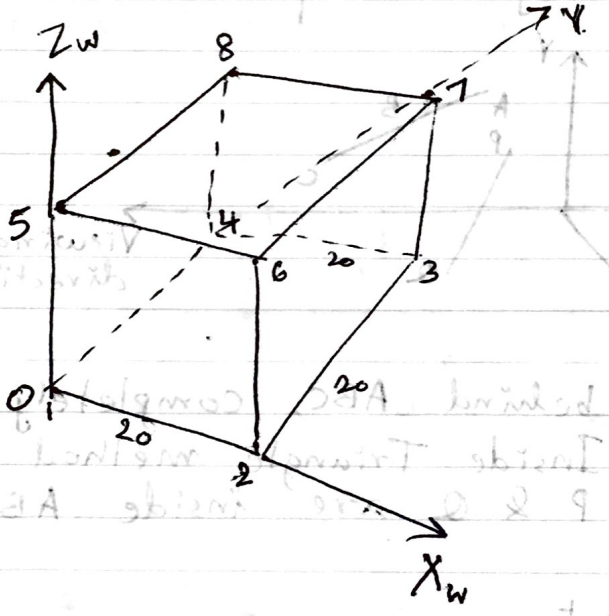
(Vertex number followed by vertex's world co-ordinates)

② After "Faces :", a polygon (face) per line represented by a sequence of vertex numbers and terminated by a full stop "."

(CCW when viewed from outside). The second vertex must be a convex vertex.







Cube.dat :

1	0	0	0
2	20	0	0
3	20	20	0
4	0	20	0
5	0	0	20
6	20	0	20
7	20	20	20
8	0	20	20

Faces :

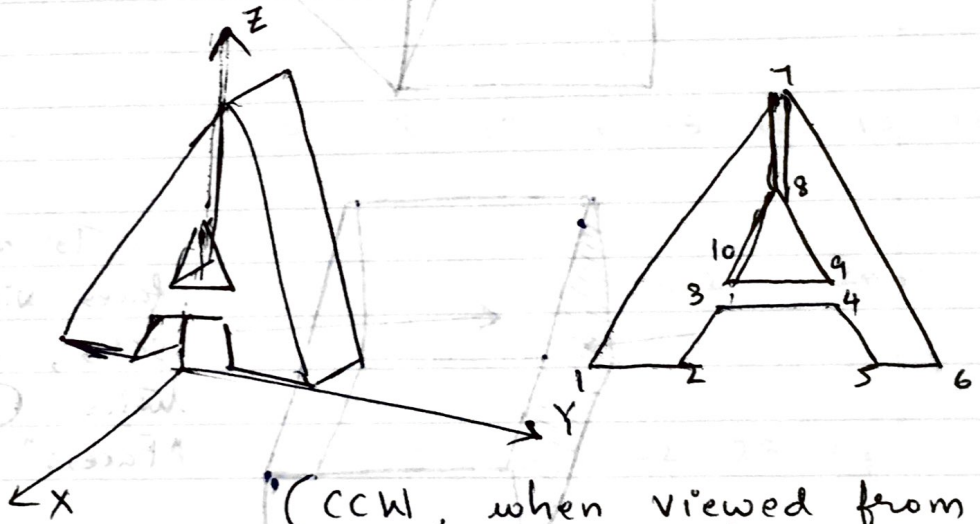
1	2	6	5.
3	4	8	7.
5	6	7	8.
4	3	2	1.
3	7	6	2.
4	1	5	8.

It has to be convex so that we can make sure in which direction it is facing.

2. Holes and Additional Undisplayed lines :-

Letter 'A' without the shown gap is not a proper polygon (a hole). With the gap, we can have the polygon : 1 2 3 4 5 6 7 8 9 10 8 7

To avoid drawing the line (7 8), we rewrite the above : 1 2 3 4 5 6 7 -8 9 10 8 -7



"(CCW, when viewed from outside). The 2nd vertex must be a convex vertex.

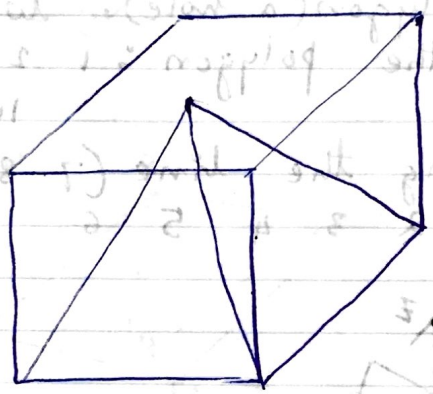
With the Rule : a line ended with a negative vertex is NOT drawn. So (7, -8) & (8 -7) are not drawn.
The rule is only used in "Faces:" part.

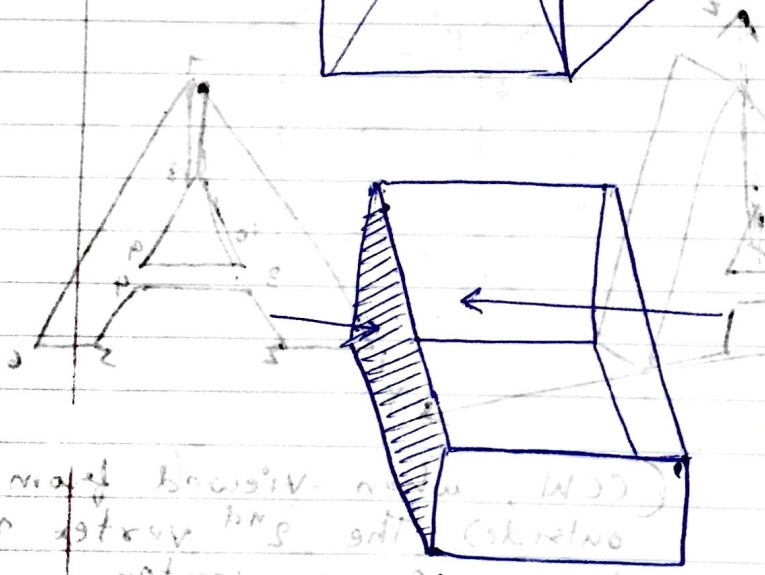
* How to generate input files.

3. Special Lines & Faces.

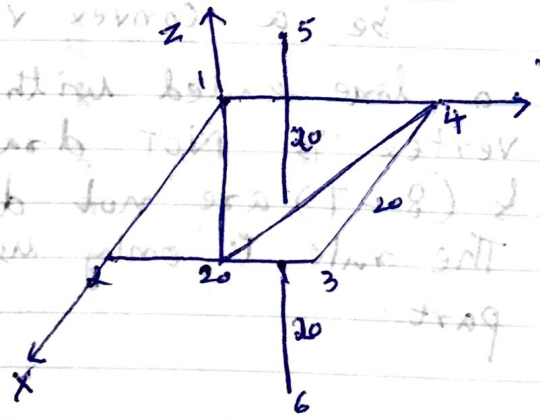
We sometimes want to treat some lines specially, not to obscure objects behind.

We simply put these lines in "Faces" part.





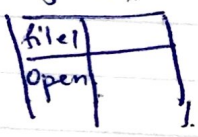
To make individual faces visible from both sides, we specify them twice (CCW & CW) in "Faces:" part



Try this using HLines.java.

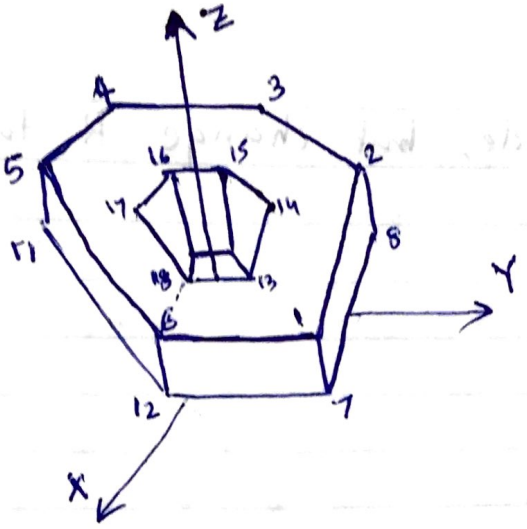
write in a file

1	0	0	0
2	20	0	0
3	20	20	0
4	0	20	0
5	10	10	20
6	10	10	-20



Faces:

4	3	2	1
1	2	3	4
6	5		



Cylinder

# of sides $n = 6$

Top face : $Z = 1$

Bottom face : $Z = 0$

Outer Radius : R

Inner Radius : r

In "faces:" part:

Top face:

1 2 3 4 5 6 -18 17 16 15 14 13 18 -6

Generalize:

1 2 3 ..... n -3n 3n-1 3n-2 ... 2n+1 3n -n

Bottom face:

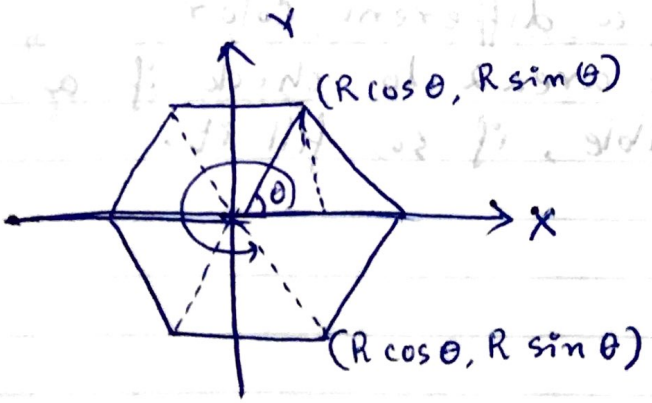
12 11 10 9 8 7 -19 20 21 22 23 24 19 -7

Generalize:

2n 2n-1 2n-2 ..... n+1 -(3n+1) 3n+2 3n+3 ... 4n 3n+1 -(n+1)

If n is too large, it becomes circle.

If r=0, it becomes Solid



$\theta = \frac{2\pi}{n}$

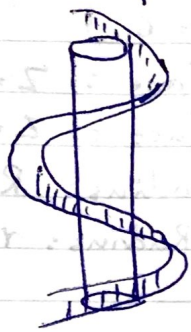
So, outer circle:

$x_i = R \cos i\theta$

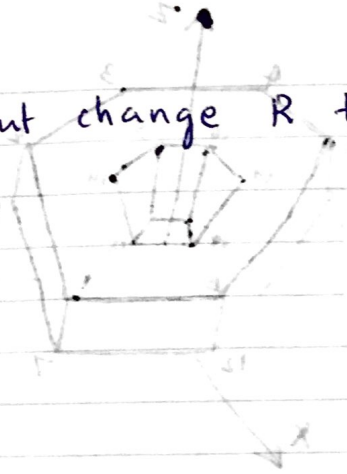
$y_i = R \sin i\theta$

$(i = 1, \ldots n)$

Same for inner circle, but change R to r.



Spiral Beams.



Chapter 7. Hidden Face Elimination.

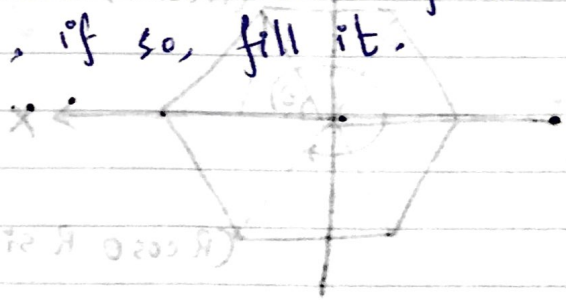
Given a viewpoint, invisible faces (back faces) should not be drawn. How to determine visible / invisible : orientation of face vertices (Example cube in chapter 5).

Backface.java

Similar to cubepes.java, but in paint :

Draw Cube :

1. Find center of world co-ordinate system.
2. d = p * imageSize / ObjectSize
3. Transformations.
4. For each vertex of 6 faces, find screen co-ordinates of the vertex & save them
5. Set a different color
6. Use area2 to check if a face is visible, if so, fill it.



The above technique does not work for removing partially invisible faces (portions).

1. Painter's Algorithm :

Main idea : Display (triangulated) Polygons in the order of their distances toward viewpoint (according to their Z-coordinates)

Main algorithm :

① Compute eye & screen coordinates for whole 3D object (Obj3D.eyeAndScreen)

② Compute a, b, c & h of $ax+by+cz=h$ for every polygon face. (Obj3D.planeCoeff).

③ Triangulate every polygon face (Polygon3D.triangulate)

④ Decide the color of every triangle.

⑤ Sort all the triangles according to their z-coordinates.

$$\text{Average} = \frac{Z_1 + Z_2 + Z_3}{3}$$ (Sum of 3 vertices z co-ordinates for each triangle)

⑥ Display all triangles in their pre-determined colors (in order of sums)

It's fast and works for most cases but not for some special cases (such as interlocking triangles / faces)

2. Z buffer (or Depth-buffer)

* Z-coordinates denote the distance from the viewpoint.
* Z-buffer is a large 2D array of canvas dimension size, storing Z co-ordinates.
* Z-buffer algorithm uses 2 buffers of the same size:
  1. Frame buffer storing color values, initialized as background color.
  2. z buffer storing Zvalue for each pixel, initialized to
* Main idea:
  For each face, we compute z-values of all pixels if such a pixel is nearer than the corresponding pixel in Z-buffer, we display the pixel in the color for the face.

Main algorithm:

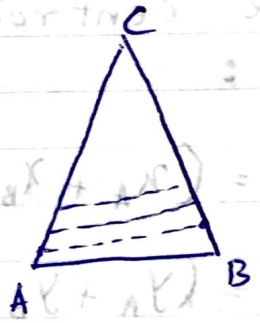
```
int pZ;        // triangles Z at P(x, y)
for (y=0; y < ymax; y++)
    for (x=0; x < xmax; x++) {
        putPixel(x, y, background-color)
        Zbuff(x, y) = 0;
    }
for each face, Triangulate;
for (each triangle)
    for (each pixel in triangle's
         projection on screen) {
        pZ = triangle's Z-value at
             P(x, y);
        if (pZ >= Zbuff(x, y)) // ne
        {   putPixel(x, y, triangle's color at
                     P(x, y));
```

// initialization.

```
        zbuff(x, y) = pz;
    }
}
```

Z-coordinates are all negative, the larger, the closer to VP. But in the textbook, Z's inverse $1/Z$ is used. So $Z_i < z\,buff(x, y)$.

Instead of drawing individual pixels for each line in a triangle, can we optimize it by drawing line segments.



To find Z_i $(= 1/Z)$ for each triangle ABC:

Consider plane $ax + by + cZ_i = k$

We wish to know how much Z_i increases/decreases if the points move one pixel up or one pixel to the right, then

$$Z_i = (k - ax - by)/c$$

$$\frac{\partial Z_i}{\partial x} = -\frac{a}{c}, \quad \frac{\partial Z_i}{\partial y} = -\frac{b}{c}$$

To find a, b, c we make $\vec{u} = AB$, $\vec{v} = AC$,

$$u_x = x_B - x_A \qquad v_x = x_C - x_A$$
$$u_y = y_B - y_A \qquad v_y = y_C - y_A$$
$$u_z = Z_{Bi} - Z_{Ai} \qquad v_z = Z_{Ci} - Z_{Ai}$$

$$\bar{u} \times \bar{v} = \begin{vmatrix} \bar{i} & \bar{j} & \bar{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = (u_y v_z - u_z v_y)\,i + (u_z v_x - u_x v_z)\,j + (u_x v_y - u_y v_x)\,k$$

then:

$$a = u_y v_z - u_z v_y$$
$$b = u_z v_x - u_x v_z$$
$$c = u_x v_y - u_y v_x$$

Based on the centroid $D(x_D, y_D)$ to compute Z_i:

$$x_D = (x_A + x_B + x_C)/3$$

$$y_D = (y_A + y_B + y_C)/3$$

$$Z_{Di} = (Z_{Ai} + Z_{Bi} + Z_{Ci})/3$$

$$Z_i = Z_{Di} + (y - y_D)\frac{\partial z}{\partial y} + (x_L - x_D)\frac{\partial z}{\partial x}$$

Boundary Problem:

Since P is on both triangles T_1 & T_2, the pixel will be colored twice as both T_1's & T_2's colors. To make P counted once for T_1, a factor of 1.01 is used in Z_i:

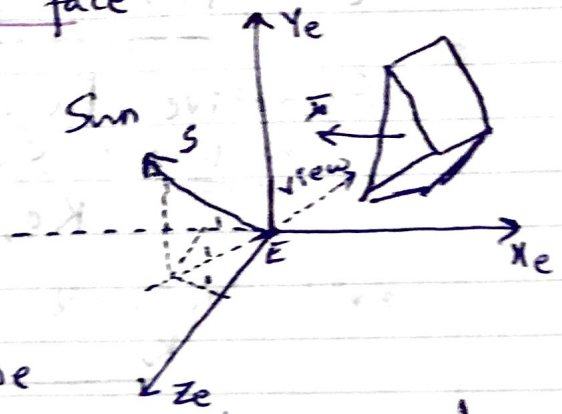
$$Z_i = 1.01 * Z_{Di} + (y - y_D) * dz\,dy + (x_L - x_D) * dz\,dx$$

01/15/2016.

Obtain a Color Code for a face

Predetermine direction of Sun

$$S = \left(\frac{1}{\sqrt{3}}\right)(-1, 1, 1)$$



Its inner product with $\bar{n} = (a, b, c)$ can determine what color to use for face : $ax + by + cz = h$.

Assume a range of inner products :

inprodRange = inProdMax − inProdMin.

```
int ColorCode (double a, double b, double c)
{
  double inprod = a * SumX + b * SumY + c * Sum
  return (int) Math.round ((( inprod - inProdMin)/
                                inProdRange) *
}
```

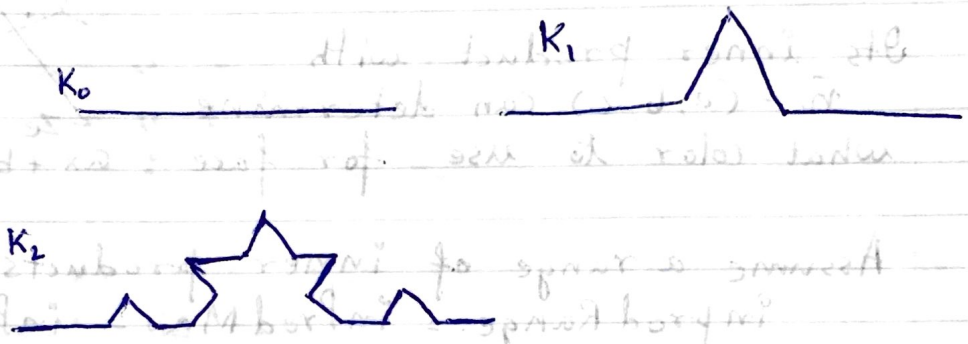
So, we can get color code by calling :

```
int cCode = obj3D.colorCode (a, b, c);
g.setColor (new Color (cCode, cCode, 0));
```

Chapter 8 : Fractals and Self-Similarity

Iterative Function System (IFS)

Koch : Successive generation of Koch curve :
$K_0, K_1, \ldots\ldots K_n$



(recursive)

To produce K_{n+1} from K_n :

① Divide each segment of K_n into 3 equal parts :

② Replace the middle part with a bump in the shape of an equilateral triangle.

Features :-

① Each segment is increased in length by a factor of 4/3. So, K_{n+1} is 4/3 as long as K_n.

② If n is very large, the curve still appears to have the same shape and roughness.

③ Infinite length, but occupies a finite region in the plane.

To draw a Koch curve, consider K_{n+1} as 4 versions of K_n tied end to end with certain angles between.

Turtle graphics:
Using turtle graphic method (drawing always relative to the current position of the turtle), recursive pseudocode:

To draw K_n:
if (n == 0) Draw a straight line;
else {
Draw K_{n-1};
Turn left 60°;
Draw K_{n-1};
Turn right 120°;
Draw K_{n-1};
Turn left 60°;
Draw K_{n-1};
}

Implementation in C:

```
void drawKoch (double dir, double len, int
{
  double dirRad = π/180 (0.017453 3) * dir;  // Radian

  if (n == 0)
    lineRel (len * cos (dirRad), len * sin(dir
  else
    n--;
    len = len/3;
    drawKoch (dir, len, n);
    dir += 60;
    drawKoch (dir, len, n);
    dir -= 120;
    drawKoch (dir, len, n);
    dir += 60;
    drawKoch (dir, len, n);
}
```

Grammar Based Models :-

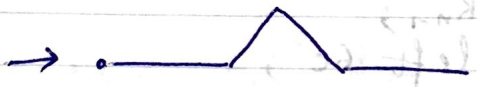
Proposed by A. Lindenmayer in 1968, So L-grammars, L-systems. We give turtle some simple instructions.

'F' - forward, drawTo(D): go forward visibly a distance D in the current dir.

'+' - turn(A): turn right through angle A degree ↻

'−' - turn(-A): turn left through angle A degree ↺

For example, the string "F-F++F-F" draws K,



To draw more generations, we define string production rule. For example, for Koch curves:

'F' ⟶ "F-F++F-F"

The initial string is called "atom" 'F' for Koch

Assuming that we may add more complex rules, use a template of 5 elements.

(Atom, F-string, X-string, Y-string, Angle)

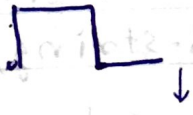
Combined with F-string, X- & Y-strings can produce more complex curves.

Example String Productions :-

1. Koch curve : (F, F-F++F-F, mil, mil, 60).
2. Hilbert curve : (X, F, -YF+XFX+FY-, +XF-YFY-FX+, 90)
   - atom
   - Fstring
   - X-string
   - Y-string.
3. Dragon curve : (FX, F, X+YF+, -FX-Y, 90)
   'X' & 'Y' characters are ignored by the turtle. For example, with dragon :

Dragon : 1st generation :
String is "FX+YF+"

Second generation :- FX+YF+ + -FX-YF+
(cancel each other)
X-string, Y-string



Recursive String Production Routine ("order" = "generation")

```
void produceString(char *str, int order)
{
  // fixed length, so picture will increase in size.
  for ( ; str ; str++).
      switch(*str) {                    // current direction.
          case '+' : CD = CD - angle;
                     break;   // turn right
          case '-' : CD = CD + angle;
                     break;   // turn left
          case 'F' : if (order > 0)
                       produceString(F=str, order-1)
```

```
        else
            drawTo(length);
        break;
case "X": if (order > 0)
            produceString(X_str, order-1);
          break;
case 'Y': if (order > 0)
            produceString(Y_str, order-1);
}
```

Move without Drawing

To keep some curve components apart from each other, we can let turtle move forward without drawing:

* f — move forward the distance D without drawing a line.

We introduce one more parameter:

(Atom, F-string, F-string, X-string, Y string, Angl

Islands: (F+F+F+F, F+F-F F+ F +FF + FF+FF-F+FF-F -FF -FFF, fffffff, nil, nil, 90)

(Atom ← F+F+F+F; fstring ← F+F-F F+ F +FF + FF+FF-F+FF-F -FF -FFF; fstring ↑; X ↑ nil; Y ↑ nil; Le ⇑ 90)

Adding Branches:

To allow the turtle to "pick up where it left off", we introduce 2 more characters:

'[': saveTurtle — store the current

State of the turtle

']' : restore Turtle - set turtle's state to the last stored value.

State of turtle = { Current position (CP), Current direction (CD) }

Implementation using a stack:

* push current state when encountering '['.
* Pop from stack when encountering ']'

Fractal tree examples:

(F, FF-[-F+F+F]+[+F-F-F], nil, nil,
(X, FF, F-[[X]+X]+F[+FX]-X, nil, 22.5)

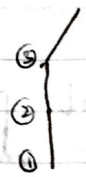
Adding Randomness and Tapering:

Eg: Adding a small random offset to each angle turned when '+' or '-' is encountered. Lines can be given thickness.

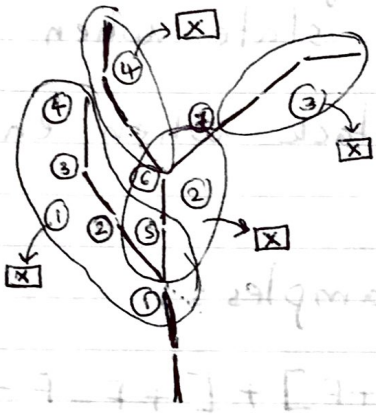
01/22/2016

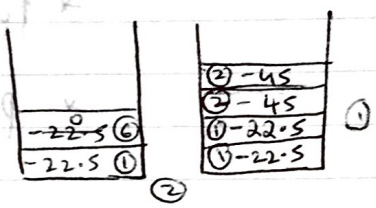
(X, FF, F-[[X]+X]+F[+FX]-X, nil, 22.5)

Generation 1:



Generation 2: FF-[[X]+X]+FF[+FFX]-X





The Mandelbrot Set

Julia & Mandelbrot Sets – Iterative Theory or Dynamic Systems Theory.

The theory investigates what happens when one iterates a function endlessly (forever).

Mandelbrot Set uses a very simple function:

$$Z_{n+1} = Z_n^2 + c \quad (c - \text{complex constant})$$

Given a starting value 0, System generates a Sequence of values (also called orbit).

a set of complex numbers.

$$\begin{aligned} Z_0 &= 0 \\ Z_1 &= C \\ Z_2 &= Z_1^2 + C = C^2 + C \\ Z_3 &= Z_2^2 + C = (C^2+C)^2 + C \\ &\text{- - -} \\ &\text{- - - - -} \\ &\text{- - - - -} \\ &\text{- - - - -} \end{aligned}$$

If the sequence (orbit) $Z_1, Z_2, \ldots$ remains within a distance of 2 of the origin forever then the point C is said to be in the Mandelbrot set. If the sequence diverges from the origin, then the point is not in the set. So,

$$M = \left\{ c \in C \mid \lim_{n \to \infty} Z_n \neq \infty \right\}$$

How to draw a 2D Image from the Mandelbrot Set? Complex Numbers are 2D!

$Z = x + iy$ (x - real part, y - imaginary part)

↓

is displayed at position (x, y)

$|Z| = \sqrt{x^2 + y^2}$ is the distance from the origin 0.

To Compare the distance with 2, we save the Square root computation by comparing

$$|Z|^2 > 4 \quad \text{or} \quad x^2 + y^2 > 4.$$

To compute the next Z value,

$$Z_{n+1} = Z_n^2 + C,$$

$$\begin{aligned} Z^2 &= (x + iy)^2 \\ &= (x^2 - y^2) + (2xy)i \end{aligned}$$

Then:

real part: $x_{n+1} = (x_n^2 - y_n^2) + C_r$ ($\rightarrow z^2$)

imaginary part: $y_{n+1} = 2x_n y_n + C_i$

Julia Sets :-

There is a different Julia set for each value of C. To produce a Julia image, we use $Z_{n+1} = Z_n^2 + C$, with a fixed starting value of Z_0.

Mandelbrot set can be used to select C for a Julia set and thus forms an index for Julia sets:

A Julia set is either connected (within M set) or disconnected (dust).

Chapter 8 Review

* Idea and features of Koch curves
* Grammar based models:
  - Grammar expressed in 5-element template
  - Meaning of each string.
  - Processing string grammars
  - Brand handling
* Mandelbrot Set
  - definition, condition for a point to be in M Set, how to draw, relationship with Julia set.