*Frame Buffer:

- 2D array (RGB) which stores all the pixels displayed on your screen.

———x———

Note:

- VGA, HDMI cables only send one pixel at a time from the frame buffer to the monitor etc.

———x———

*Raster Scanning: (VGA, HDMI standard)

- Going through the frame buffer left to right, row by row sending the pixel to the monitor etc is known as raster scanning.

———x———

Note:

1) HSync signal is sent through our cable when a row is done scanning
2) VSync is sent when entire image is done scanning. (A longer signal compared to HSync).

———x———

*Graphical Primitive Point:

1) Translation: → Vector addition

*2) Rotation; (Anticlockwise)(2D)

- $(x,y) \xrightarrow[\text{Angle}]{+\theta} (x',y')$

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

$$P' = R \cdot P$$

$R^{-1} = R^T$ ↑ orthonormal

- R is ~~orthonormal~~

$\because \bar{r}_i \cdot \bar{r}_i = \bar{c}_i \cdot \bar{c}_i = 1$
$\bar{r}_i \cdot \bar{r}_j = 0 \ (i \neq j)$

* • Each row of the rotation matrix gives us the line which after rotation will align with respective axes.

* • Each column of the rotation matrix gives us the line to which our respective axes will end up after rotation.

3) Rotation; (3D)

- $R_z(\theta) = R_{-z}(-\theta)$
- Anticlockwise = +ve.
- Looking down from the arrowhead. (into the arrowhead).

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

* ⇒ About z-Axis, $R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

*

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

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

DSB-

*4) Non Uniform Scaling:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & u & 0 \\ 0 & 0 & t \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$, Scale with s, u, t.

- If $s = u = t$, Angles blw points remain unchanged.

*5) Shearing:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & x_y & x_z \\ y_x & 1 & y_z \\ z_x & z_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

x_y → How much does y affect the x

x_z → How much z affects the x

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \lambda \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$ } Shear the x coord with λ effect of y coord

6) Reflection:

- Similar to Scaling but with -ve signs (-1).

———x———

*Homogeneous coordinates:

- Add a non zero scale factor w. (Generally $w = 1$)
- This is used so that addition of vectors could also be treated as multiplication.

$$\begin{bmatrix} x+a \\ y+b \\ 1 \end{bmatrix} = \begin{bmatrix} ? \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Homogeneous Coordinates

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \equiv M$$ ← Identity matrix.

- If $w \neq 1$, then at least we need to divide by k to get actual coordinates back.

∴ Translation is also a linear operation now.

⇒ But in order to do this, we increase the dimensions of the vectors & matrices by 1.

∴ Our multiplication becomes,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The actual 2D matrix used for the operation in hand (multiplication)

⇒ As $w \uparrow$, point goes closer to origin, and point goes to ∞ at $w = 0$. → This is the significance of w.

↳ Projective geometry.

———— x ————

Rotation & Translation:

$$T_{4\times4} = \begin{bmatrix} I_{3\times3} & t_3 \\ 0_3 & 1 \end{bmatrix}$$

$$R_{4\times4} = \begin{bmatrix} R_{3\times3} & 0_3 \\ 0_3 & 1 \end{bmatrix}$$

$$\Rightarrow TR = \begin{bmatrix} I & t \\ \phi & 1 \end{bmatrix} \begin{bmatrix} R & \phi \\ \phi & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ \phi & 1 \end{bmatrix}$$

$$RT = \begin{bmatrix} R & \phi \\ \phi & 1 \end{bmatrix} \begin{bmatrix} I & t \\ \phi & 1 \end{bmatrix} = \begin{bmatrix} R & Rt \\ \phi & 1 \end{bmatrix}$$

Translation & Rotation is not commutative.

$Rt = t$ when $R = I$, $t = 0$, t is an eigen vector of R.

Note: $T_1T_2 = T_2T_1$

$S_1S_2 = S_2S_1$

$R_1R_2 \neq R_2R_1$ (In 3D)

* exc: Consider $R_x(90) = R_1$

$R_y(90) = R_2$

$\Rightarrow$ Consider a point on the z-axis.

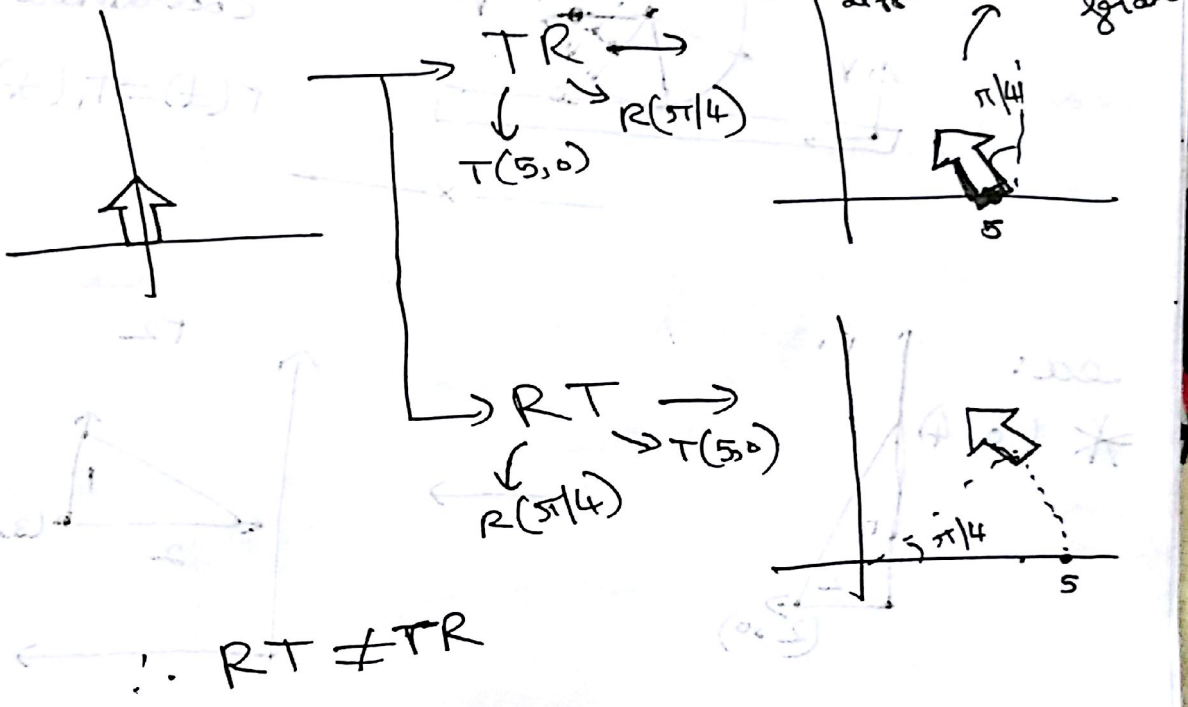
$\Rightarrow$ R_2R_1 puts point on -y axis

R_1R_2 puts point on +x axis.

———— x ————

exc:

*



$\therefore RT \neq TR$

———— x ————

Note: We rotate an object about its centroid if we want just rotation about the axis instead of about the origin.

$$\Rightarrow \Big(T(S) \cdot R() \cdot Scale() \cdot T(-S)\Big)(Point)$$

S: Centroid; −S: Centroid

———— x ————

Note: $P' = \underset{C_1}{R} \mid \underset{C_1}{T} \cdot \underset{C_2}{P}$ ($C_2 \rightarrow$ coordinates system of the object.)

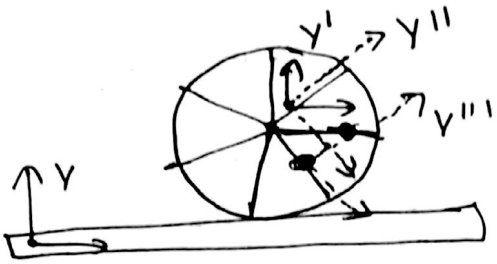
** ⇒ For points we do T, then R (Right to left) ⇒ to do the object.

* ⇒ We could also view this in terms of coordinate systems, when we go from left to right so R, then T. (Rotation/Scaling) about the axis of the object

** of the object

— x —

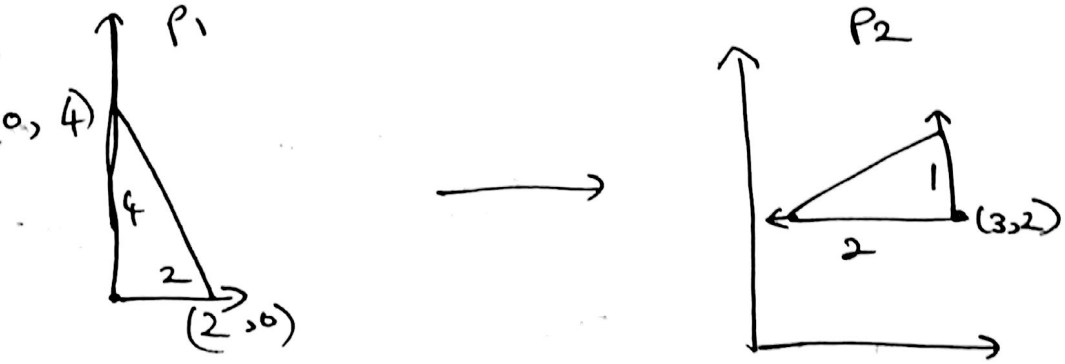
ex: Many times, we can solve questions by * using multiple coordinate systems.



- So we come to the head using 4 coordinate systems.

$P(t) = T_1(t) \cdot R(\alpha(t)) \cdot T_2(p(t)) \cdot P'''$

— x —



- Scale by 1/2, Rotate 90°, then translate (3,2)

$$P_2 = M_3 \cdot M_2 \cdot M_1 \cdot P_1$$

⇒ M_1 = Scale
M_2 = Rotate
M_3 = Translate

Visualize in both coordinate systems.

or, $S(1/2, 1/2) \cdot T(6,4) \cdot R(90)$ can also work! etc

— x —

Rotation about any axis α:

- We first align α with either X, Y, or Z axes. then rotate. Then do an inverse of the alignment

$$R_\alpha(\theta) = R_{\alpha x}^{-1} R_x(\theta) R_{\alpha x}$$

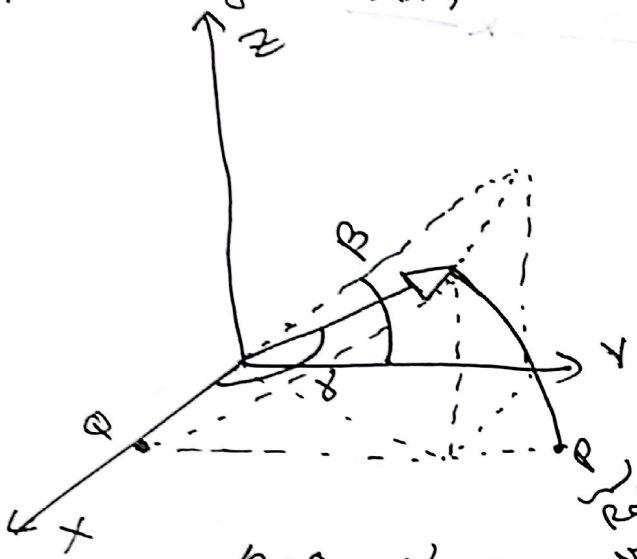
or

$$R_\alpha(\theta) = R_{\alpha y}^{-1} R_y(\theta) R_{\alpha y}$$

or

$$R_\alpha(\theta) = R_{\alpha z}^{-1} R_z(\theta) R_{\alpha z}$$

Now, to get $R_{\alpha x}$,



- $\text{Tan}\beta = \alpha_z/\alpha_y$
- $\text{Tan}\gamma = \frac{PQ}{\alpha_x} = \sqrt{\frac{\alpha_y^2+\alpha_z^2}{\alpha_x^2}}$

$\downarrow$

$\sqrt{\frac{\alpha^2-\alpha_x^2}{\alpha^2}}$ (length of vector $\overline{XP}$)

Rotate vector α to meet XY plane. $\Rightarrow$ length of vector remains same.

$\angle POQ = \gamma$ (Due to rotation of α)

$\Rightarrow$ First rotate $-\beta$ about x axis, then $-\gamma$ about z axis, to get $R_{\alpha x}$

$$\Rightarrow \boxed{R_{\alpha x} = R_z(-\gamma) R_x(-\beta)}$$

- We get γ & β from above.

$$R_{\alpha x}^{-1} = R_x(\beta) R_z(\gamma)$$

$\because (AB)^{-1} = B^{-1} \cdot A^{-1}$

Note: Since all we need is $\bar{\alpha}$ aligns with x axis, we can make up a rotation matrix,

$$R_{\alpha x} = \begin{bmatrix} \bar{\alpha}^* & 0 \\ [\bar{v} \times \bar{\alpha}]^* & 0 \\ [\bar{\alpha} \times \bar{v} \times \bar{\alpha}]^* & 0 \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

$\Rightarrow$ we can use any vector $\bar{v}$.

- Since 1st row is the vector which aligns with x axis from properties

$\therefore$ Unit vector.

⇒ We only care about the 1st row. But rotation matrices need to be orthonormal. So we can only use $\bar{v} \times \bar{\alpha}$ & $\bar{\alpha} \times \bar{v} \times \bar{\alpha}$ as our other 2 rows.
(Make sure your use unit vector $\hat{\alpha}$ & and $(\hat{\alpha} \times \hat{v} \times \hat{\alpha})$)

—— x ——

e: If a point is translated as $P' = MP$
A plane is translated as $\bar{n}' = M^{-T} \cdot \bar{n}$
↑ Normal of plane; M^{-T}: Inverse Transpose

—— x ——

—— x ——

: View Mapping transforms a 3D Image into a 2D map by applying a projection.

—— x ——

*Scan Conversion or Rasterization:

- Older days we used vector graphics, → draw screens using electron guns. (Nowadays vector graphics is just a type of representation) (Software)
- Rasterization is creating a discretized image in the framebuffer array. (2D).

⇒ ie. converting real valued coordinates to integers which correspond to pixels.

- Scan converting a point is just rounding off x & y coordinates to integers.

———x———

*Note: Scan conversion works on displaying a 2D projection of our scene. This 2D projection is generated by other methods (camera projections).

———x———

*Scan Conversion of a Line: (Efficiency/Speed)

- We are given 2 points (pixels) which are the 2 endpoints of the line.
- We need to select pixels to turn on so as to display the line.

*1) Incremental Algorithm: (Slides)

- Assume our line has slope b/w 0 & 1. This can be expanded later.
- Consider each point along x-axis from x_1 to x_2, and at each step $y' = y' + m$ where m is the slope ($\because x \to x+1, y \to y+m$) we use round(y) to select the y coord of pixel.

(This is slow since 'm' can be a float)

* 2) Integer Incremental Algorithm:

$(\Delta y = y_2 - y_1)$
$(\Delta x = x_2 - x_1)$
$(0 \leq m \leq 1)$

```
while(x < x2)
    x ← x+1
    sl ← += Δy
    if (sl ≥ Δx)
        y ← y+1
        sl -= Δx
    draw(x, y)
```

- No floating point numbers here, so its faster than our previous algorithm.
- Cannot be expanded to draw curves easily.

* 3) Mid point Line Algorithm : (Slides)

- Suppose our line equation is $ax+by+c=0$ (Assume $a>0$)
- $0 < \text{slope} = \Delta y/\Delta x = -a/b < 1$
- Now for each point on x, starting from x_1, y_1. If at any point $d = ax_{p}+by_{p}+c$ is true then the ~~mid~~point (x,y) lies below the line, so we select the north east pixel. Elese we select just the east pixel.
- At each x, we check if $x, y+\frac{1}{2}$ lies above or below the line, if its below, then $x \to x+1$, $y \to y+1$ (NE) else $x \to x+1$ (E).

$\Rightarrow$ We need not calculate d at every step, since if we move East(E), then $x \to x+1$ so $d \to d+a$, and (NE), then $d \to d+a+b$.

⇒ So starting with x, y where $x = x_1$ & $y = y_1$ we can draw our line till $x = x_2$ & $y = y_2$ without floating numbers.

(If we use $d = 2ax + 2by + 2c$, then even though we use $y = y_0 + \frac{1}{2}$, still we will only deal with integers).

———x———

1) Different Coordinate Systems:

- Object Reference → ORC } object reference coordinates
- World Reference → WC
- Camera / View Reference → VRC

—x—

∴ ORC → WC → VRC
Modelling Viewing

—x—

1) Note: * Plane transformations: (Always follow this order)

(Modelling)

$$P_{WC} = M_1 M_2 M_3 M_4 P_{ORC}$$

M_1: Translate
M_2: Yaw (Z axis)
M_3: Pitch (X axis)
M_4: Roll (Y axis)

↓ Else we will get different results

—x—

Scene Graph:

- A hierarchy of nodes where the leaves correspond to models & inner nodes correspond to transformations.

—x—

2) Viewing: *

- Camera location → Look point } Look vector
  Vector

  will give us 2 of the rotations, since we align the z-axis of the camera along this look vector.

- Up vector is used to determine rotation of camera along the look vector.

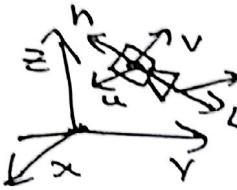
⇒ Up should not be parallel to Look, since we need them to define a plane together. (This plane is the vertical plane in our captured image)

* $P_{WC} = M_1 M_2 \cdot P_{VRC}$ } We need to find this, so we do the inverse

  - M_1: Translate camera origin to center
  - M_2: Rotation to align the world with the camera.

* Look

$\bar{I} = \bar{L}/|\bar{L}|$, $\bar{d} = \bar{u}/|\bar{u}|$, $\bar{n} = -\bar{I}$, $\bar{u} = \bar{d} \times \bar{n} / |\bar{d} \times \bar{n}|$, $\bar{v} = \bar{n} \times \bar{u}$ are what axes



$$M_2 = \begin{bmatrix} \bar{u} & \bar{v} & \bar{n} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$, Columns get aligned to respectively, after transformation

$$\therefore M_1 \cdot M_2 = \begin{bmatrix} I_{3\times3} & \begin{matrix} x \\ y \\ z \end{matrix} \\ 0_3 & 1 \end{bmatrix} \begin{bmatrix} \bar{u} & \bar{v} & \bar{n} & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* * $$M_1 \cdot M_2 = A = \begin{bmatrix} \bar{u} & \bar{v} & \bar{n} & \begin{matrix} x \\ y \\ z \end{matrix} \\ 0 & 0 & 0 & \end{bmatrix}$$

$\therefore P_{VRC} = A^{-1} P_{WC}$ } We want this since we want VRC for each object from WC

* * $A^{-1} = M_2^T \cdot M_1(-C)$

Rotation property. $-x, -y, -z$ translation

———x———

Note: For a given scene & fixed camera P, V matrices are fixed for all objects but M for each object can change. (MVP Matrix)

———x———

Note: Right hand rule for cross product

— END HERE —

*3) Projection

a) Parallel Projection (All rays parallel to each other)

* b) Perspective Projection → (Similar to pinhole camera)
(All rays pass through a single point, COP)
↓
Center of Projection

— x —

Note: Finite aperture models are like pinhole cameras with larger apertures, so the images can be blurred → real cameras

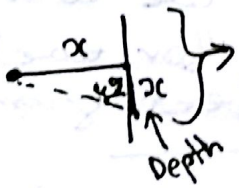
— x —

I) Parallel Projection:

* i) Orthographic:

- Projection plane is perpendicular to parallel rays of projection.
- If Direction of Projection = $(\pm 1, \pm 1, \pm 1)$, its isometric projection.

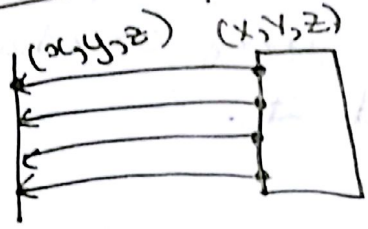
ii) Oblique:



- Cabinet → projection plane forms 45° angle with rays. (Length along depth axis is same as other axis)
- Cavalier → Angle b/w plane & rays is $\tan^{-1}(2)$ (Length along depth axis is halved)

— x —

*Orthographic Projection Matrix:



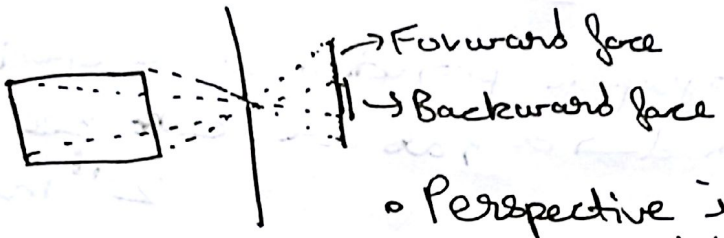
- We project onto the XY plane, so we make $z=0$.

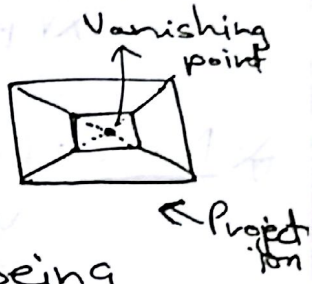
$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

- We can add scaling if required
- For multiple objects we display the nearest one.

---

* II) Perspective Projection:

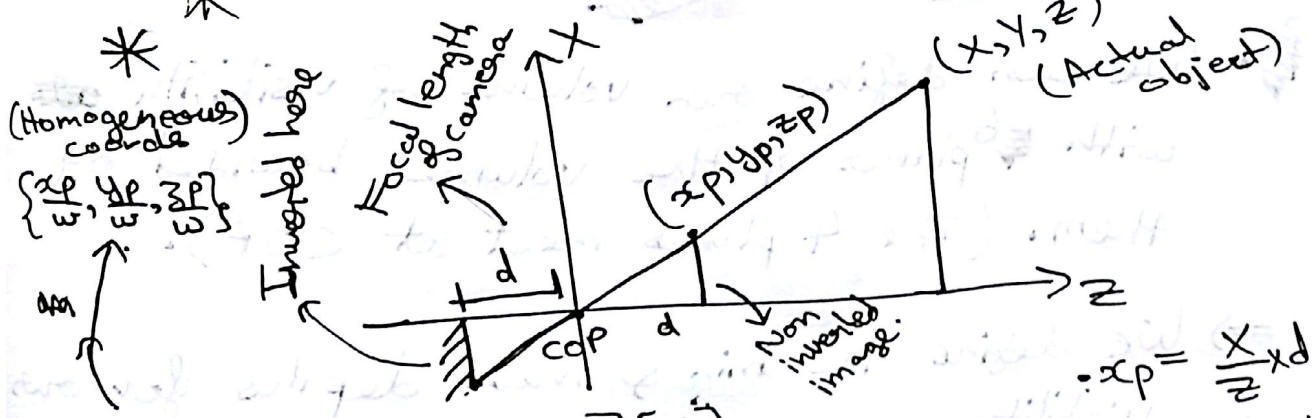




- Perspective is being introduced.

? ⇒ The point where these rays meet (for each edge) is called the vanishing point. ~~Not parallel to projection plane~~.

⇒ Perspective projection can be characterized by the number of vanishing points.



$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

→ So now $w = z$, so we get correct coordinates

- $x_p = \frac{X}{Z} \times d$
- $y_p = \frac{Y}{Z} \times d$
- $z_p = d$

• Now, we can see that multiplying entire transformation matrix by any constant non-zero will not effect it.

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

↳ Perspective projection matrix

z_p is always d since we project onto XY parallel plane.

• $$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

⇒ We get this assuming we shift origin to $(0,0,d)$

—x—

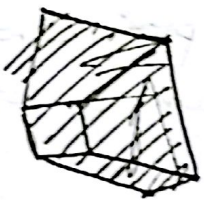
* Note: Orthographic projection is a special case when $d \to \infty$, as we can see from the 2nd matrix.

—x—

* Volume of Visibility: (First do this, then Canonical view volume, then project based (Perspective):

• We assume our view is a rectangle x

• We assume our view is a cone.



→ Frustum (cut of a pyramid)

• We can define our volume of visibility with 6 planes & the volume bounded by them. (The 4 planes meet at COP).

⇒ We define the min & max depths for our visibility region. (Near & Far planes)

—x—

We are advised to use as large as possible, closest view of interest plane & smallest, farthest view of interest plane) (Near & Far plane)

2) Orthographic:

- Just a cuboid formed with 6 planes.

---x---

Note: glm::frustum or glm::ortho allow 6 values for inputs to display visibility volume.
glm::perspective only supports θ_h & θ_v → Horizontal & Vertical field of views.

---x---

* Canonical View Volume: (Gives normalized coordinates by projection)

- A normalizing matrix converts our view volume to bounded $-1 \leq x,y,z \leq 1$.
- In both perspective & orthographic volumes of visibility, l, r, t, b are scaled distances. (f, n are +ve)

⇒ Orthographic, Perspective → Canonical View Volume

a) ⇒ Orthographic Normalizing Matrix =

$$\begin{bmatrix} \frac{2}{(r-l)} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

But we need to position this at origin as needed, } Centroid must be shifted to origin.

⇒ * Orthographic Normalizing Matrix =

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

∴ Normalizing Matrix × $[\text{Coords}]_{VRC}$ will give us the normalized coordinates, and we can ignore if coordinates are outside our volume (canonical view volume).

↳ sometimes, we use clipping for this usually.

b) ⇒ Perspective Normalization:

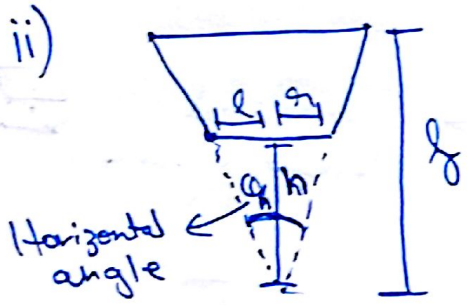
* Matrix,

i) $(l, b, -n) \rightarrow (-1, -1, 1)$
$(l, t, -n) \rightarrow (-1, 1, 1)$
⋮

• Map each point manually & solve.

We have 8 points, so we get 24 equations & 16 variables, so we can solve for the matrix.

* * ii)



• Consider l, r first, then t, b.

a) ⇒ We want to make $\Theta_h = 90°$, so $l = n$ must be made, so scale x-axis by n/l

$\Rightarrow S_x = n/l$, $S_y = n/t$, $S_z = 1$ currently

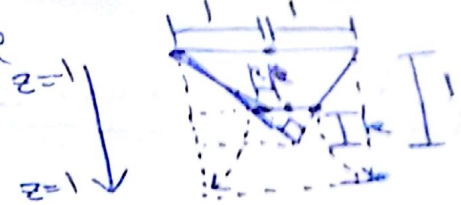
$\Rightarrow S_x = \cot(\Theta_h/2)$

$S_y = \cot(\Theta_v/2)$

) • We want to make $f = 1$, so

$S_x' = S_y' = S_z = 1/f$ (Far plane is scaled)

Now, we have



c) ⇒ Now we want to move the $2k \times 2k$ near square to become the other face of 2×2 of the canonical view volume.

∴ The matrix to do this,

$$M_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1+k}{1-k} & \frac{2k}{1-k} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(We can see $(x, y, -k)$ goes to $(x/k, y/k, 1)$)

$$M_2 \cdot M_1 = \begin{bmatrix} \frac{\cot(\theta_h/2)}{f} & 0 & 0 & 0 \\ 0 & \frac{\cot(\theta_v/2)}{f} & 0 & 0 \\ 0 & 0 & 1/f & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$M_3 \cdot M_2 \cdot M_1 \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_{VRC}$ ⇒ Gives us normalized coordinates.

⇒ Now from normalized coordinates, we can ignore z-values to get screen coordinates of objects.

(But we do preserve z-coords for depth & visibility based reasons).

———×———

*4) Viewport Transformation:

$$M = T\left(\frac{W}{2}, \frac{H}{2}\right) \times S\left(\frac{W}{2}, \frac{H}{2}\right)$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} W/2 & 0 & W/2 \\ 0 & H/2 & H/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

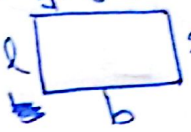
↓ Window coordinates (Screen)

Projected coordinates (Normalized)

- Find X, Y coordinates on the screen

—x—

Note: (At times we can specify t, b, l, r to transform into a rectangle; t, l, r, b signed values)



↓ glViewport command

—x—

Note: Graphics simulate ideal pinhole cameras, to get blur etc we need to do more computation.

—x—

Note: After getting window coordinates, Clipping ~~normalized~~ lines to the viewport can be done by using line equations to find intersections.

—x—

Note: Triangles are guarenteed to be planar, so they are used as 2D primitives.

—x—

*Primitive Pipeline:

- Vertex Processing → Primitive Assembly → Rasterization (Scan Conversion) (Vertex Shader) → Pixel Processing (Fragment Shader) → Frame Buffer

⇒ In the rasterization step, each pixel is given a colour, depth & texture coord and we get a pixel colour as output, which is set to the pixel in pixel processing.

—— x ——

Note: (GPGPU) → General Processing on GPUs

—— x ——

*Note: Scissoring → Clipping + Rasterization

—— x ——

Vision:

*Visible Surface Determination:

- Sometimes ray tracing is done which help determine which objects we see first etc.

⇒ If we use the hierarchial model while constructing our scene. We can do a few things to determine visible surfaces.

*i) View Frustum Culling: (selective deletion) (VFC) (Large scale culling)

- If an object's bounding box is out of the view volume, then we can eliminate all the objects which are within this bounding box. (View Hierarchy → Children of this node + this node)

- If the bounding box is completely inside the view volume, then we will consider all the children + this node.
- If its partially intersecting the view volume; we recursively go down the tree.

—x—

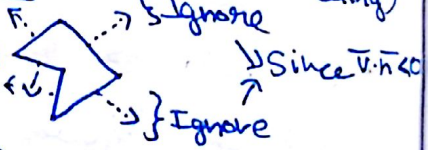
Note: AABB ⇒ Axis Aligned Bounding box.
OBB ⇒ Oriented Bounding box. } Minimum volume (Hard to compute) (Rarely used)

—x—

Note: To check if bounding box is within view volume, we can check if any x points of object lie within opposite planes. (All pairs of x opposite planes of view frustum).

—x—

*2) Viewing a solid object: (Medium scale culling)

* (Back Face Culling) ⇒ 

- For every triangle, we have normal vectors which are facing outside the object.
- Suppose our view direction = $\bar{V}$

⇒ Normal of each Δle = $\bar{n}$

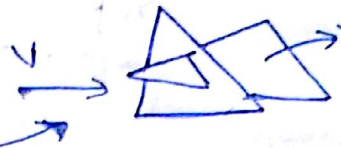
⇒ If $\bar{V}\cdot\bar{n} < 0$ we draw it, else we discard.

(If $\bar{V}\cdot\bar{n} < 0$, Angle b/w them > 90 & < 180)
So its a part of the front which can be displayed.

—x—

*3) Pixel Culling: (Pixel scale culling)

- Back to front order drawing might work, but it fails in some cases



This part is not visible since its behind the other Δ'le.

i) Object Precision Algorithm:

- For every object check if any other object is obstructing the selected object. $O(n^2)$

ii) Image Precision Algorithm:

- For every pixel check each object, $O(m^2 \times n)$ } $m \times m$ pixel screen, n objects

* iii) Z-Buffer | Depth Buffer: (Generally used)

**

- Frame Buffer (Nowadays) = Colour Buffer + Depth Buffer (Z-Buffer)

$(-1 \leq z \leq 1)$ (Larger z → (closer) (to camera)

(Assume $(-1 \leq z \leq 0)$ → Since others are culled.

⇒ Write to frame buffer ~~[illegible]~~ only if the z of the object is larger than the previously stored value.

- For this algorithm we need to be able to calculate 'z' of any internal point of the Δle, given the 'z' of the 3 points that form the Δ'le. (We will get these 3 'z' values from the values stored in the projection of VRC step).

— x —

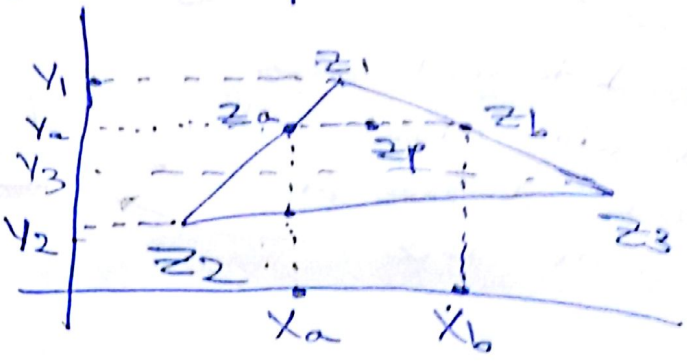
*Computing z- for inner points:

*

- We can get z_a & z_b

*(Since z varies linearly)

- $z_a = ?$
  $z_b = ?$



- $\frac{z_2 - z_1}{y_2 - y_1} = \frac{z_a - z_1}{y_a - y_1}$ (Linear interpolation)

$$\Rightarrow z_a = \frac{(y_a - y_1)}{(y_2 - y_1)} \times (z_2 - z_1) + z_1 \longrightarrow (1)$$

$$\Rightarrow \frac{z_3 - z_1}{y_3 - y_1} = \frac{z_b - z_1}{y_b - y_1}$$

$$\Rightarrow z_b = \frac{(y_b - y_1)}{(y_3 - y_1)} \times (z_3 - z_1) + z_1 \longrightarrow (2)$$

- $\frac{z_p - z_a}{x_p - x_a} = \frac{z_b - z_a}{x_b - x_a}$

$$\Rightarrow \boxed{z_p = \frac{(x_p - x_a)}{(x_b - x_a)} \times (z_b - z_a) + z_a}$$

(We know x_p, For x_a & x_b we know $y_a = y_b = y_p$, we know the line equations of $z_1 z_2$ & $z_1 z_3$, so we can get x_a & x_b).

— x —

Note: List Priority algorithms were used earlier. When we have tricky cases of triangles & back to front ordering cannot be done, then we split the Δ'les into smaller ones as needed.

———x———

* Binary Space Partitioning Trees: (Old Method) (slides)

- To find the back to front ordering of objects to be drawn.

? - We create trees where internal nodes → planes & leaf nodes → objects / triangles.

⇒ Each plane will put the view point on either one side of the plane & this algo tells us to draw the objects on the other side of the plane first & then the objects on this side.

- For these planes, just select the planes of our polygons / triangles. (And this polygon is drawn after the backside polygons and then the front are drawn)

⇒ If the plane cuts a polygon split it into 2.

(Pseudocode in slides)

~~(One of the leaves of the tree~~

* (All leaves on one side of the node are objects on one side of the plane & the others are on the other side, so when we add our view point, if its on one side of the plane, we can draw the other part of the tree first).

- Large preprocessing time, but redrawing is O(n).
- Back face culling can be integrated in this easily.

Note: With ray tracing we can do things like transparency, reflectivity etc. But highly computational.

—— x ——

Lighting & Shading:

- Lighting → Corners of the shape concerned
  Shading → Entire shape concerned.

—— x ——

*Additive Primaries: RGB representation of colours. (Red, Green, Blue)
(Origin at black) (0,0,0) → Black

—— x ——

*Subtractive Primaries: CMY representation of colours (Cyan, Magenta, Yellow)
(Origin at white) (0,0,0) → White.

→ Printers use this since the paper we print on is initially white.

→ Also we use CMYK where K corresponds to black/ greyscale

$K = \min(C, M, Y)$ ↑ Cheap , $C -= K$, $M -= K$, $Y -= K$

—— x ——

Note: * i) Diffuse reflection → Dull/Rough surface
Specular reflection → Smooth/Shiny surface

—— x ——

*Lambert's Law & Diffuse Reflection:

*. Normal component of light is absorbed by the object & reflected back completely.

$$\Rightarrow I_d = I_p \times k_d \times \cos\theta = I_p K_d (N \cdot L)$$

$(\cos\theta = N \cdot L)$
(Dot Product)

$\Rightarrow I_p$ = Light falling on surface
K_d = Diffuse reflection coefficient.
$\theta \in [0, 90°]$

* $\Rightarrow \boxed{I_{d\lambda} = I_{p\lambda} \cdot (K_d \cdot O_{d\lambda})(N \cdot L)}$ } For each wavelength λ.

Diffuse colour of the object. (corresponding to λ)

$\Rightarrow$ In graphics $\lambda \rightarrow R, G, B$. So 3 equations.

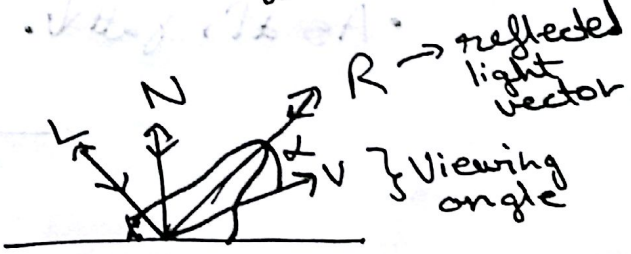
---

*Specular Reflection:

- On shiny objects, a spot is present where we get a highlight.
- Highlight changes with ~~moving~~ watching angle.

$\Rightarrow$ Highlight does not depend on diffuse colour of object.

- The intensity generally varies as $(\cos\alpha)^n$.

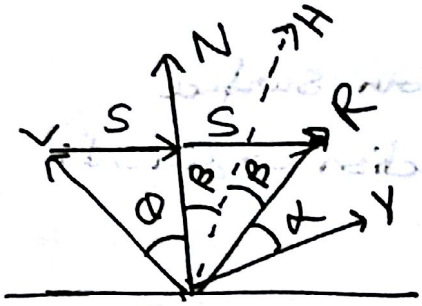


- $I_{s\lambda} = I_{p\lambda} \cdot k_s \cdot \cos^n \alpha = I_{p\lambda} k_s \cdot (V \cdot R)^n$

↑ Specular reflection coefficient.

(OpenGL allows us to specify a specular colour for the objects, although its not technically correct).

??



$\bar{R} = L + 2S = 2L + 2S$?

$\bar{R} = 2\bar{N}\cos\theta - \bar{L}$

- $H = ?$

$(|\bar{L}| = |\bar{N}| = |\bar{R}|)$, So $|\bar{L}|\cos\theta = |\bar{N}|\cos\theta$ ✓

---

*Ambient Light: (Independent of light source I_d)

- Some light that is always present.

$I_{p\lambda} = I_{a\lambda} k_a O_{a\lambda}$ } Ambient light

---

*Atmospheric Effects:

- Attenuation effects of lights,

$$I_{p\lambda} = \underbrace{I_{a\lambda} k_a O_{a\lambda}}_{\text{a}\lambda} + f_{att}(I_{d\lambda}) + f_{att}(I_{s\lambda})$$

↑ Diffuse component

? $\Rightarrow f_{att} = 1/(a_1 + a_2 d + a_3 d^2)$

- As $d \uparrow$, $f_{att} \downarrow$.

---

*Light Sources:

1) Point Light: & more.
2) Spot Light:

(Total Illumination Equation) ($\lambda \rightarrow R, G, B$)

Note: $I_{p\lambda} = I_{a\lambda} k_a O_{a\lambda} + \sum_i f_{att_i} \left(I_{\lambda i} \left(k_d O_{d\lambda} (N \cdot L_i) + k_s O_{s\lambda} (V \cdot R_i)^n \right) \right)$

** ↓ Final point colour.

↑ Summation here for multiple light sources

↑ Incoming light

Note: * Emissive colours for objects that emit light on their own.

*Shading:

- Evaluating illumination equation only once per polygon surface? what else?

I) Constant/Flat Shading:

- For each surface pick a point (ex: center) and evaluate illumination equation there and use it for the entire surface.
- We get a lot of sharp edges (drastic colour change b/w adjacent faces of same object.

*II) Interpolated Shading: (Gouraud) (Interpolate final illumination)

- Calculate the illumination equation once for the 3 corners of the triangle & then use them to interpolate.

- This ~~intrapolation~~ interpolation can be done along with the interpolation of z-values.

⇒ Gouraud Shading:

- Colour interpolation shading.
- Similar to z-value interpolation.

⇒ The problem here is the highlights. Specular highlights are localized bright spots. These can be missed & interpolated across polygon making the highlight bigger.

———x———

III) Interpolated Shading: (Phong) (Interpolate normals & calculate illumination)

- Suppose we know to which surface a triangle belongs to, then if we can calculate actual normals for the vertices instead of assuming they are ⊥ar to the face always.

⇒ Once we have these 3 normals, we can interpolate the normals along the surface.

- This is however calculating for each pixel point, so its more expensive compared to ~~other~~ flat shading methods.

———x———

* Texture Mapping:

- Textures use <u,v> coordinates.

⇒ 3D points of the game correspond to 2D texture coordinate corners.

- We can use these to make our games a lot more realistic.

———x———

* Interpolated Transparency:

- $I_\lambda = (1 - k_{t1}) \times I_{\lambda 1} + k_{t1} \times I_{\lambda 2}$

- k = Transmission coefficient of a polygon.

($k = 0 \rightarrow$ ~~Transparent~~ Opaque)
($1 \rightarrow$ ~~Opaque~~ Transparent)

⇒ $(1 - k)$ = Opacity.

- If two objects overlap each other, then we can use the above equation to consider the transparency of t_1 object.

———x———

* Shadows:

- Shadow Map → Bitmap (0's or 1's)

⇒ If $S_i = 0$ } That part is a shadow

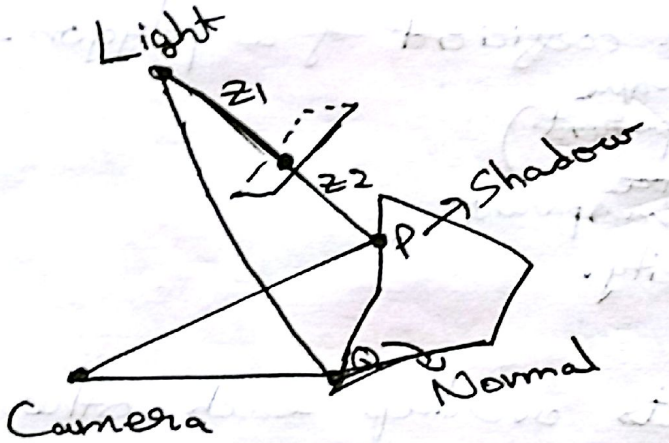
- $I_{p\lambda} = I_{a\lambda} k_a O_{a\lambda} + \sum_i S_i f_{atti} I_{l\lambda i}$ (Diffuse + Specular)

⇒ Calculating shadow map is hard.

*2-Pass z-buffer Shadow:

⇒ Get the z-coordinate map when we draw the scene from the lightsource & then from our actual camera & on transforming the camera to that light source.

- We then compare this z-depth with the z-depth we got assuming camera is at light source. If our new z-depth is greater, then it means originally something was blocking this point from the light, so its a shadow.



z-depth

- Case 1 : z_1

Case 2 : $z_1 + z_2$ (This is greater)

↓

∴ P is a shadow

---

*Ray Tracing:

- Ray casting → Ray tracing at a depth of 1. → No reflection etc.
(Similar to rasterization results).
- Send out a ray for each pixel, and determine the color to assign the pixel.

⇒ Ray Equation:

- Camera (CoP) $= P_0 = (x_0, y_0, z_0)$
  Pixel $(P_1) = (x_1, y_1, z_1)$

$\Rightarrow P = (x_0 + t\Delta x,\ y_0 + t\Delta y,\ z_0 + t\Delta z)$

(t is the only variable) ($t \geq 0$, It's a ray)

* * To find intersections, (with planes).
↳ $ax + by + cz + d = 0$

I) a) Find if the ray intersects the entire plane or not. (∞ plane),

$\rightarrow a(x + t\Delta x) + b(y + t\Delta y) + c(z + t\Delta z) + d = 0$

$\rightarrow t = \dfrac{-(ax + by + cz + d)}{a\Delta x + b\Delta y + c\Delta z}$, If $t \geq 0$ then our ray intersects the plane.

b) Find if ray intersection is within a specified triangle on the plane,

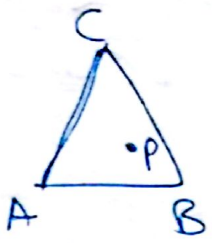
→ One of the methods is to use transforms (rotations) to align plane with XY etc plane & then use standard methods (point lies on right of all 3 sides (oriented) of the triangle).

→ Another is to consider taking projections of plane on XY, YZ & ZX.

* → Barycentric coords is the way we will tackle it.

* * Barycentric Coordinates:

(To find if point lies within △)



$\Rightarrow P = \alpha A + \beta B + \gamma C$

$\Rightarrow \alpha + \beta + \gamma = 1$

- For every point within the △le

$\Rightarrow \alpha, \beta, \gamma > 0$

$\alpha, \beta, \gamma < 1 \rightarrow \leq 1$ if on the △le.

$\Rightarrow P = \alpha A + \beta B + (1 - \alpha - \beta) C$

$$\begin{bmatrix} x - x_3 \\ y - y_3 \end{bmatrix} = \underbrace{\begin{bmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{bmatrix}}_{T} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \text{T.e}$$

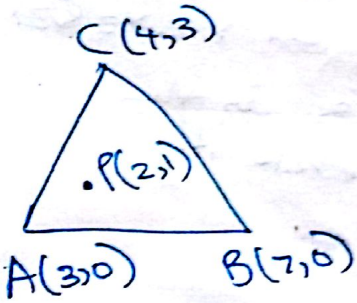
$e = T^{-1}(P - C) \rightarrow$ Barycentric coordinates

$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$, Once we find this, we have the barycentric coords for point $(x, y) \Rightarrow (\alpha, \beta)$

---

* *

ex: $T = \begin{bmatrix} -1 & 3 \\ -3 & -3 \end{bmatrix}$, $T^{-1} = \frac{1}{12}\begin{bmatrix} -3 & -3 \\ 3 & -1 \end{bmatrix}$



$$P \Rightarrow T^{-1}\begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 & -3 \\ 3 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ -2 \end{bmatrix} \times \frac{1}{12}$$

$$= \left[+\frac{1}{2}, +\frac{1}{6}\right]$$

$$\Rightarrow (\alpha, \beta, \gamma) = \left(\frac{1}{2}, \frac{1}{6}, \frac{1}{3}\right)$$

Note: Once you get α, β, if any of them are
* greater than 1 or less than 0, we can say we lie outside the Δ'le.

II) Intersection with Sphere:

- Sphere $\Rightarrow (x-a)^2 + (y-b)^2 + (z-c)^2 - r^2 = 0$

---

*Note: In ray tracing, for each point where the ray hits & stops, we check if a direct ray to a light source exists or not, if it doesn't then this point is a shadow.

---

* *Shading → with Ray Tracing: (Code in slides)

- Once a ray intersects an object at a point, we calculate the color that needs to be assigned.

→ color = ambient
for each light:
- if a ray can be drawn from light to point with no intersection,
- we modify color, otherwise shadow

recursively check reflective & refractive rays

Set a depth limit →

---

* Acceleration Techniques:

- We use various datastructures to organize Δ'les in our scene.
- We then try to remove Δ'les we are sure that won't intersect the ray (pruning) & on the rest of the Δ'les we run the ray tracing & try to find the nearest intersection. (This is the toughest job).

* 1) Grids: (One of the acceleration techniques)

- "Teapot in a stadium" problem, we solve it using non-uniform grid cells. (Lots of Δ'les in a single cell)
- For a given ray we check which cells it passes through & we only consider Δ'les which belong to these cells.

* 2) Quad-Tree: (Octa-tree in 3D)

- Works for the "Teapot in a stadium" problem.
- We subdivide our region into 4 (or 8) if we feel the region has many Δ'les etc.
- Again draw the ray & check which cells it intersects & use Δ'les there.

* 3) KD Tree:

- Adaptive dividing plane so as to minimize the amount of object splits.
- Splits are in XY plane parallel, then YZ, then ZX plane parallel.
  - => Each split try to ensure you have equal no. of objects on each side.

4) BVH Tree :

- Use bounding volumes on various objects to construct tree.

→ If two BV's overlap, we go through both child nodes → not strictly $O(\log N)$

- Creation is very fast, so this is often used with movable objects. (Recreate tree every frame).

Note: Beam tracing → Multiple rays treated at the same time.
