

Algorithms and Data Structures



COMP261 **Tutorial Week 6**

Yi Mei

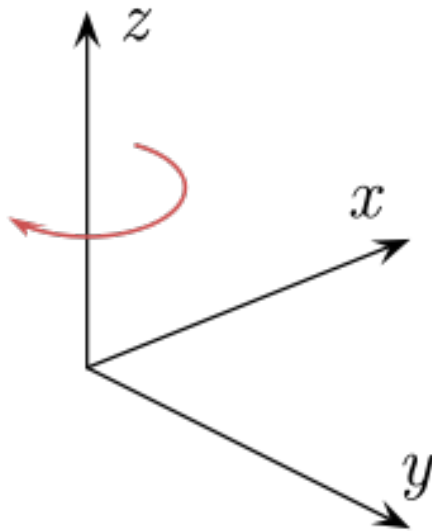
yi.mei@ecs.vuw.ac.nz

3D Rendering

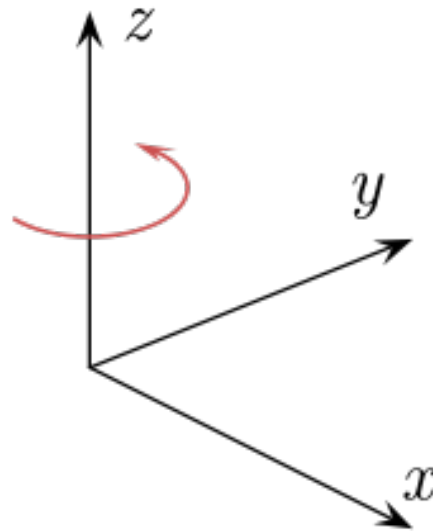
- Given a **3D object**
 - A set of **triangle polygons**, each polygon has **three vertices** (3D points)
- **Step 1: identify which polygons are visible**
 - Calculate **normal**: visible if the **z-values of the normal is negative**
- **Step 2: calculate shading color**
 - **Ambient light + incident light**
- **Step 3: render**
 - Line-by-line, calculate **edge list**, and **z-buffer**
- Whenever **transformed (rotated, scaled, translated)**
 - Calculate new coordinates of the vertices using the **unified transformation operator** (4x4 matrix)
 - Repeat Steps 1-3

3D Coordinate System

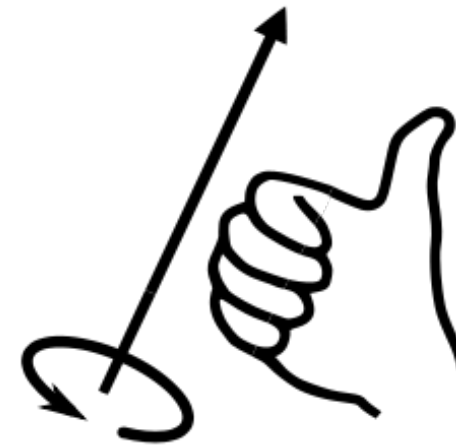
- A standard 3D coordinate system based on **right-hand rule**
 - Order x, y and z
 - Use **right hand**, **thumb** point to the **z** axis
 - Curl of other fingers from x to y (**anti-clockwise viewing from z axis**)



Left-hand rule



Right-hand rule



Right-hand rule

Unified Transformation Operator

Input: original point = $\{x, y, z\}$, 4x4 transformation matrix T

Output: new point $\{x', y', z'\}$ after transformation

Initialise newpoint[4] = $\{0, 0, 0, 1\}$;

//multiply (x, y, z, 1) with the 4x4 transformation matrix T on the left

for (row = 0 to 3) {

for (col = 0 to 3) {

 newpoint[row] += $T[\text{row}][\text{col}] * \text{point}[\text{col}]$;

 }

}

// only keep the first 3 elements

newpoint = newpoint[0:2];

return newpoint;

Transformation Matrices

Translation:

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \cdot s_x \\ y \cdot s_y \\ z \cdot s_z \\ 1 \end{bmatrix}$$

Rotation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{bmatrix}$$

Composite Transformation

- Apply the transformation matrix from right to left
 - First transformation to the right most, last to the left most

// Calculate the composite transformation matrix

Input: a sequence of 4x4 transformation matrices (M_1, \dots, M_k)

Output: the 4x4 composite transformation matrix CM

$CM = M_2 * M_1;$

for ($i = 3:k$) {

$CM = M_i * CM;$

}

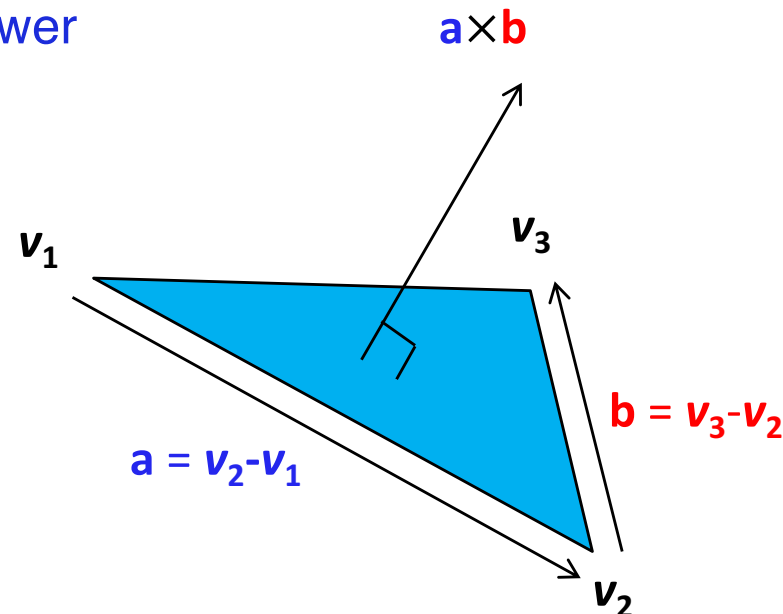
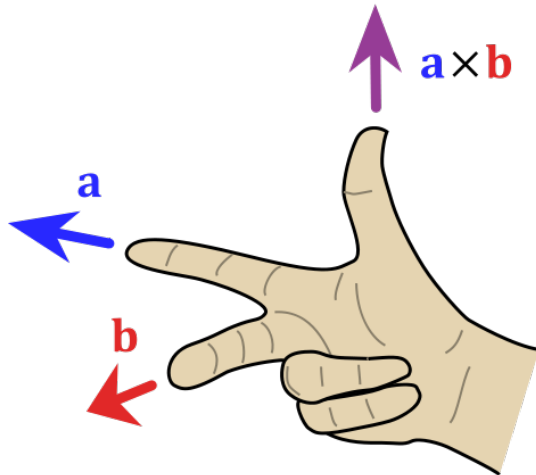
- Order of transformations
 - Rotation -> Scaling -> Translation

Cross Product

- **Cross product** is an operation between vectors. It returns another vector that is **perpendicular** with the input vectors.

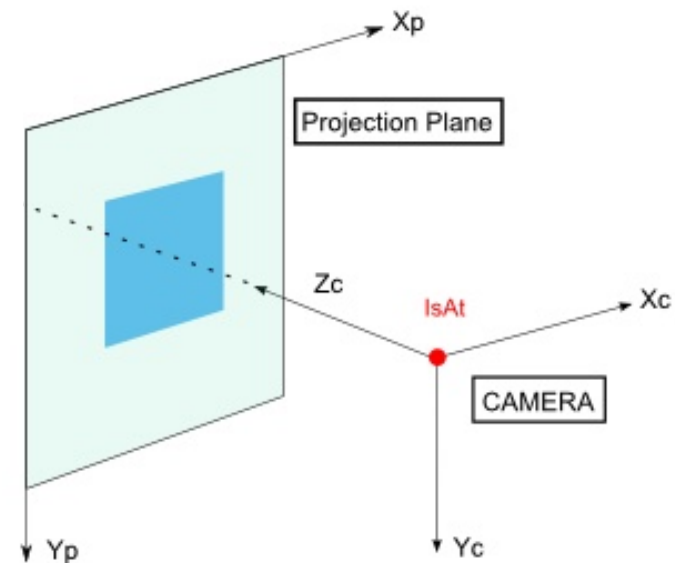
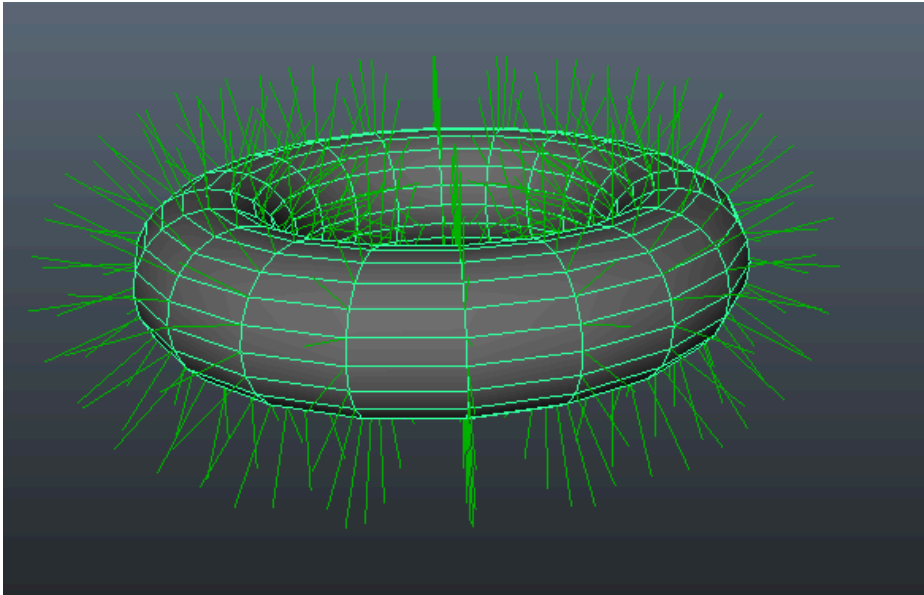
$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{bmatrix}$$

- The three vectors follow the **right-hand rule**.
- Use **cross product** to get **which direction the polygon is facing**
 - $\mathbf{v}_1 \rightarrow \mathbf{v}_2 \rightarrow \mathbf{v}_3$ is anti-clockwise
 - $(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_2)$ is facing the viewer



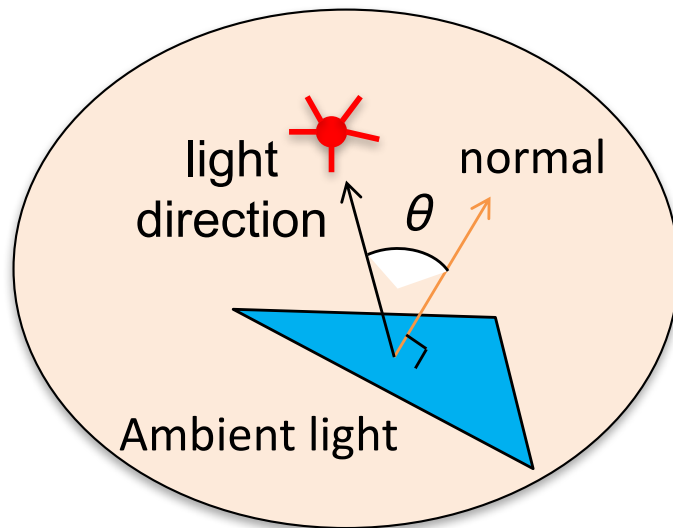
Visible/Invisible Polygons

- The cross product $(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_2)$ is called the **normal** of the polygon
- A polygon is **visible** to viewer, if its **normal has negative z coordinate value**



Shading

- A simple method:
 - Uniform reflectance for red, green, blue
 - Ambient light: intensity (0, 1]
 - $\text{ambientlight} = \text{ambient light intensity} * \text{reflectance}$
 - Incident light source: intensity (0, 1], and its direction
 - $\text{incidentlight} = \text{incident light intensity} * \text{reflectance} * \cos(\theta)$
 - Light (shading color = ambientlight + incidentlight)



$$\cos(\theta) = \frac{\text{lightDirection} \cdot \text{normal}}{|\text{lightDirection}| \times |\text{normal}|}$$

Shading Computation

Input:

- three vertices ordered anti-clockwise when facing the viewer: $v_i, i = 1, 2, 3$
- Ambient light intensity $AL = (AL.r, AL.g, AL.b)$, each color is within the range $(0, 1]$
- Incident light intensity $IL = (IL.r, IL.g, IL.b)$, each color is within the range $(0, 1]$
- Incident light direction $D = (D.x, D.y, D.z)$
- Reflectance $R = (R.r, R.g, R.b)$, each color within the range $[0, 255]$

Output: the shading color $(S.r, S.g, S.b)$

// calculate normal

$$a = v_2 - v_1, b = v_3 - v_2$$

$$n = a \times b$$

Cross product

// calculate $\cos(\theta)$

$$\cos(\theta) = \frac{n \cdot D}{|n| \times |D|}$$

Dot product

// calculate the shading

for (c in {r, g, b}) {

$$S.c = AL.c \times R.c + IL.c \times R.c \times \cos(\theta);$$

}

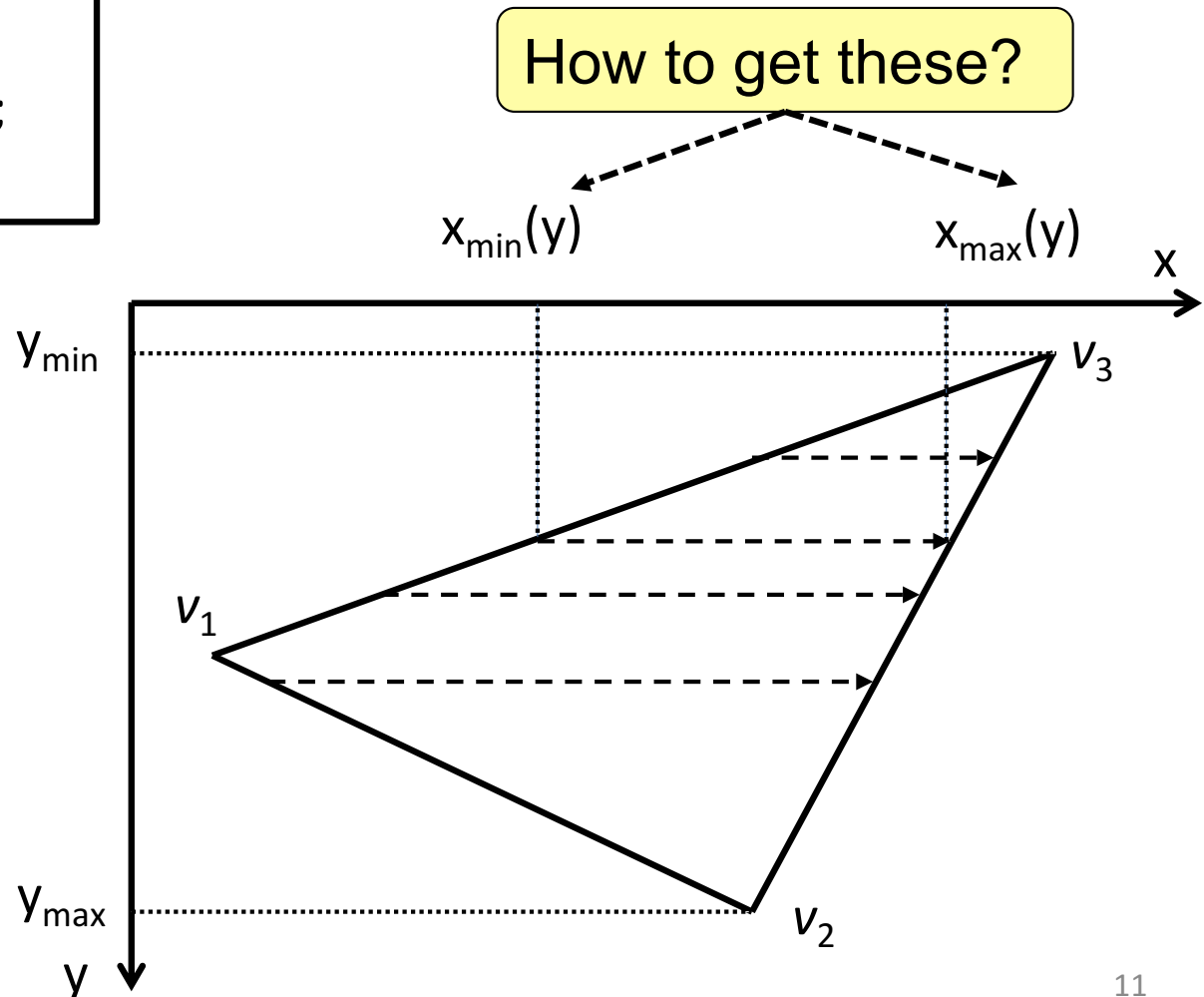
Polygon Rendering

- z-axis is the viewing direction, so the screen is x-y plane
 - Render pixels **line by line**

```
for (y = ymin to ymax) {  
  for (x = xmin(y) to xmax(y))  
    pixel(x,y) = shading color;  
}
```

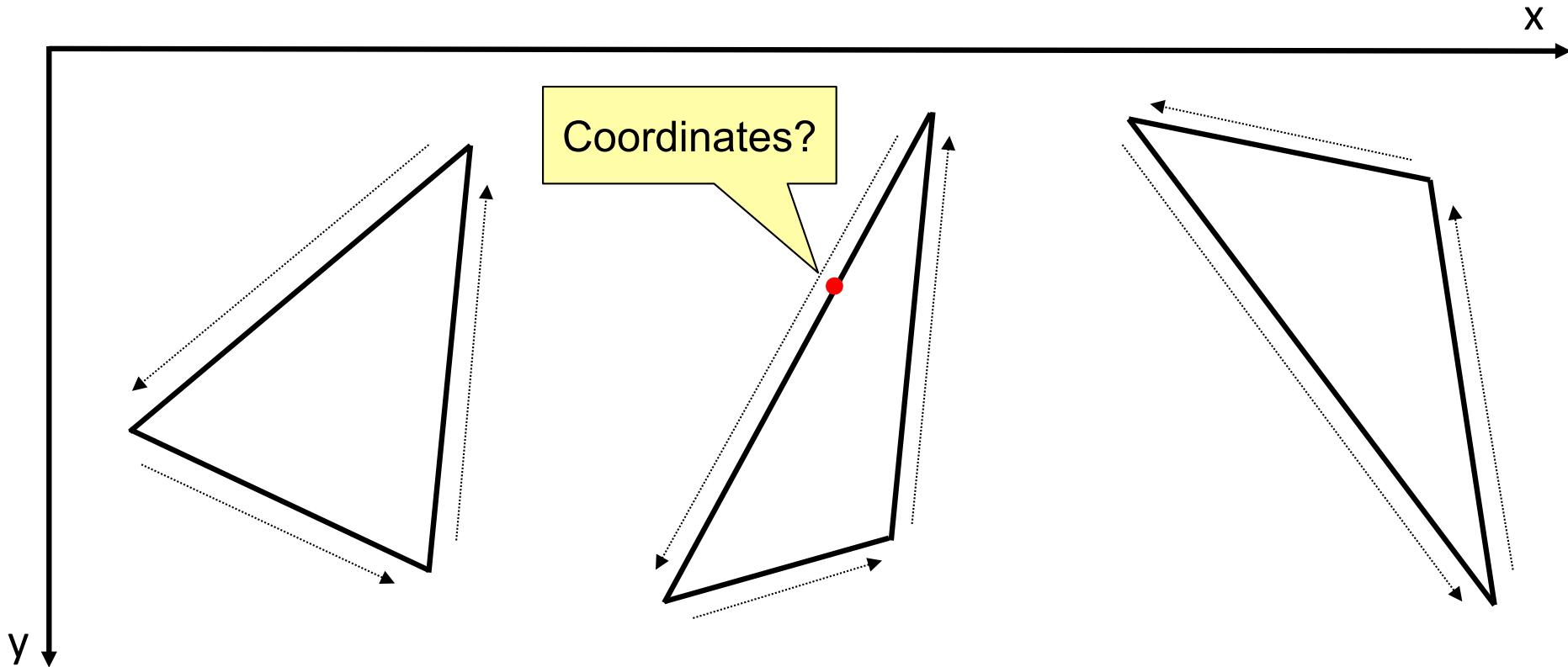
$$y_{\min} = v_3 \cdot y;$$

$$y_{\max} = v_2 \cdot y;$$



Polygon Rendering

- For any y value, get $x_{\min}(y)$ and $x_{\max}(y)$
 - All the $x_{\min}(y)$ and $x_{\max}(y)$ are on the edges of the polygon
 - If scanning the edges **anti-clockwise**, then
 - When the scan is **going down**, then visit $x_{\min}(y)$
 - When the scan is **going up**, then visit $x_{\max}(y)$



Edge List

- Scan **each of the three edges**, and update **2-column EdgeList**
 - If scanning **up**, then update **$x_{\max}(y)$** column
 - If scanning **down**, then update **$x_{\min}(y)$** column
 - Use liner interpolation for each edge

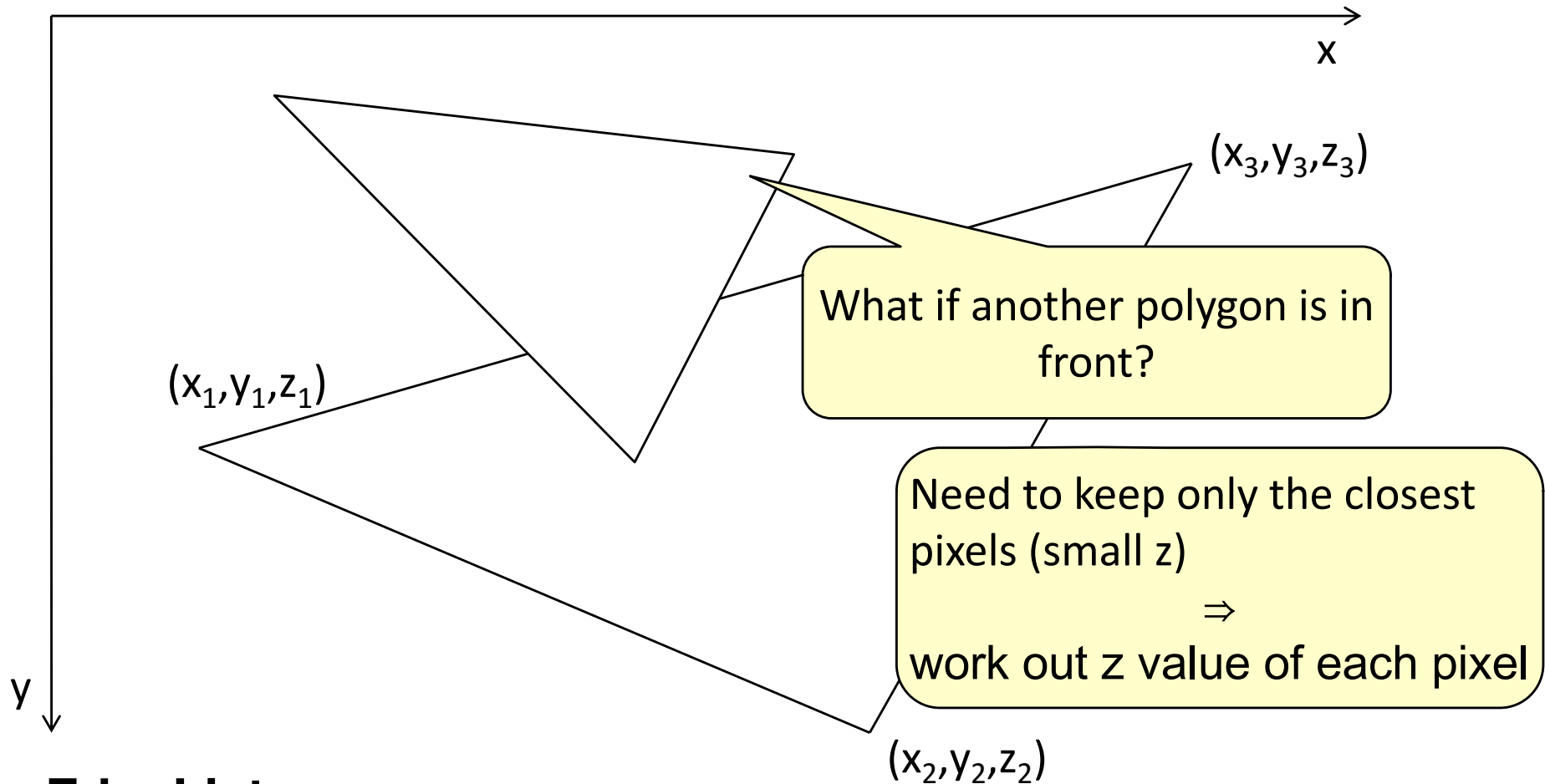
```

for (edge (a, b) in  $\{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$ ) {
    slope = (b.x - a.x) / (b.y - a.y); Anti-clockwise ordered
    x = a.x, y = round(a.y);
    if (a.y < b.y) // going down, update  $x_{\min}(y)$ 
        while (y <= round(b.y))
             $x_{\min}(y) = x$ ,  $x = x + \text{slope}$ , y++;
    else // going up, update  $x_{\max}(y)$ 
        while (y >= round(b.y))
             $x_{\max}(y) \leftarrow x$ ,  $x \leftarrow x - \text{slope}$ , y--;
    }

```

	EdgeList	
	$x_{\min}(y)$	$x_{\max}(y)$
y_{\min}		
$y_{\min}+1$		
\vdots		
y_{\max}		

Multiple Polygons



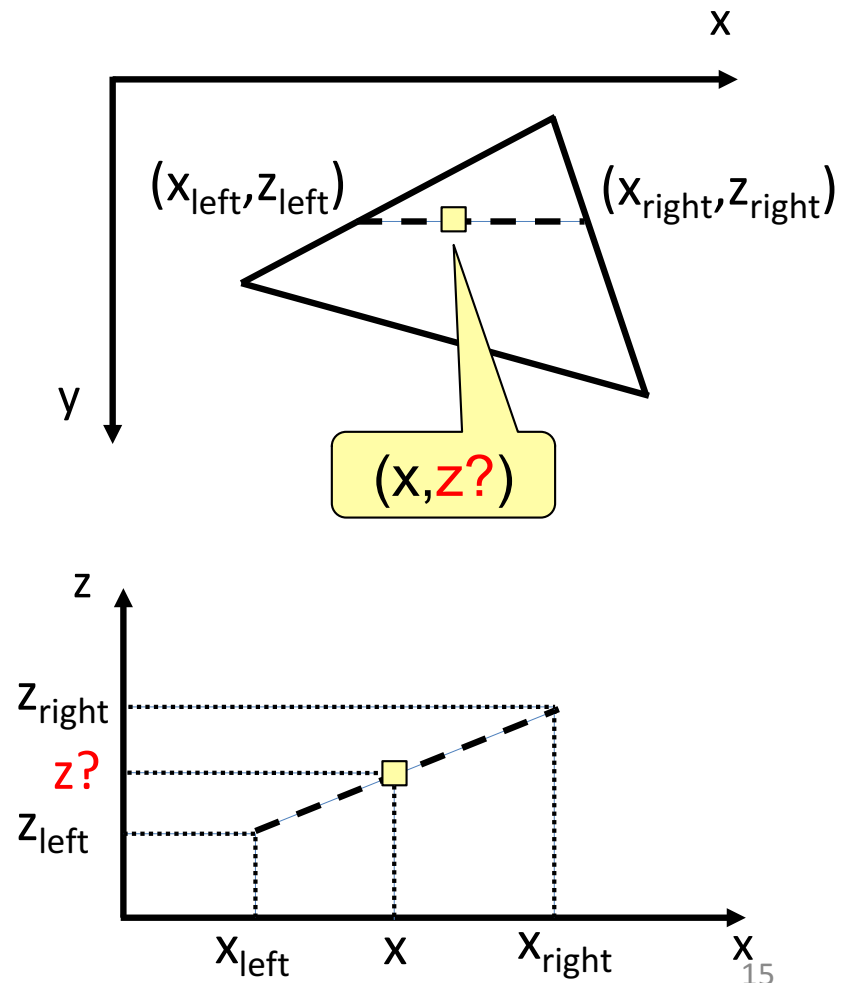
- **EdgeList**

- $x_{\min}(y)$ and $x_{\max}(y)$ for the edges
- $z(x, y)$ for each pixel
 - If a pixel is on multiple polygons, render the polygon where it has the smallest z value

Render with EdgeList and z-buffer

- Compute the EdgeList for both x and z for the vertices on the edges
 - Compute the z value for each pixel inside the polygon using another linear interpolation

	EdgeList			
	Left		Right	
	x	z	x	z
y_{\min}				
$y_{\min}+1$				
\vdots				
y_{\max}				



Render with EdgeList and z-buffer

```
renderedImg = new Color[imageWidth][imageHeight];  
zdepth = new double[imageWidth][imageHeight], initialise all entries to  $\infty$ ;  
for (each polygon) {  
    calculate the x and z EdgeList (EL) of this polygon;  
    for (y from EL.ymin to EL.ymax) {  
        slope = (EL.zright(y) - EL.zleft(y)) / (EL.xright(y) - EL.xleft(y));  
        x = round(EL.xleft(y)), z = EL.zleft(y) + slope * (x - EL.xleft(y));  
        while (x <= round(EL.xright(y))) {  
            if (z < zdepth(x,y)) {  
                renderedImg(x,y) = shading color of this polygon, zdepth(x,y) = z;  
                z  $\leftarrow$  z + slope, x++;  
            }  
        }  
    }  
}  
return renderedImg;
```