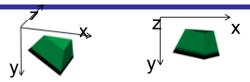
# COMP 261 Lecture 16

3D Rendering

## Simple Z-buffer Rendering Pipeline




**input**: set of polygons
viewing direction
direction of light source(s)
size of window.

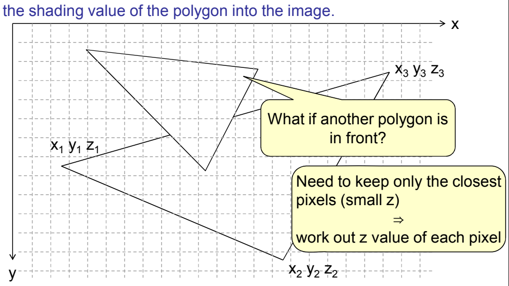
**output**: an image

**Actions**

- rotate polygons and light source so viewing along z axis
- translate & scale to fit window / clip polygons out of view
- remove any polygons facing away from viewer (normal$_z$ > 0)
- for each polygon
  - compute shading
  - work out which image pixels it will affect
  - for each pixel write shading and depth to z-buffer (retains only the shading of the closest surface)
- convert z-buffer to image

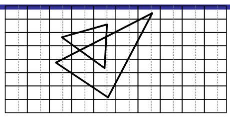
## Projecting onto the image

- View along the z axis – project to the x-y plane
- Find all the pixels covered by the polygon
- Put the shading value of the polygon into the image.



## Z-buffer

- 2D array of
  - pixel value (shading)
  - z value (depth of pixel)



- Only copy shading value for a pixel into z-buffer if it is closer than current value.
  ⇒ hidden parts of image automatically disappear

  **for each** polygon:
    **for each** pixel on polygon
      compute its z value
      insert shading value into z-buffer if z less than current entry

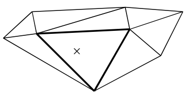
## Problem:

- Polygons specified by their vertices only.
- We need to work out
  - each pixel on the polygon
  - the depth of each pixel.

- Solution:
  Interpolate between the vertices

## Interpolation possibilities

- Light reflected from a polygon:
  - could be uniform (if assume each polygon is a flat, uniform surface)
    ⇒ compute once for whole polygon
  - could vary across surface (if polygons approximate a curved surface)

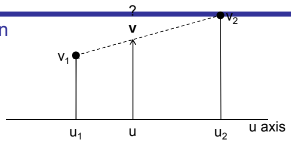


  - Can interpolate from the vertices:
    - use "vertex normals" (average of surfaces at vertex)
    - either interpolate shading from vertices
    - or interpolate normals from vertices and compute shading

- What about shadows and reflected light from other sources
  - ray tracing!! expensive, we will ignore it

## Interpolation

- Linear interpolation




- $v = v_1 + \underbrace{(v_2 - v_1) / (u_2 - u_1)}_{\text{slope}} * (u - u_1)$
- If repeatedly interpolating in steps ($\Delta u$), can be more efficient:

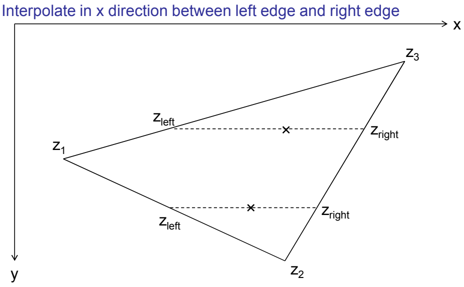
  $\Delta m \leftarrow (v_2 - v_1) / (u_2 - u_1) * \Delta u$
  $u \leftarrow u_{1}, v \leftarrow v_1$
  **while** $u < u_2$
  $u \leftarrow u + \Delta u, \quad v \leftarrow v + \Delta m$
  output $\langle u, v \rangle$

  only 2 additions per point

## Interpolation on polygon

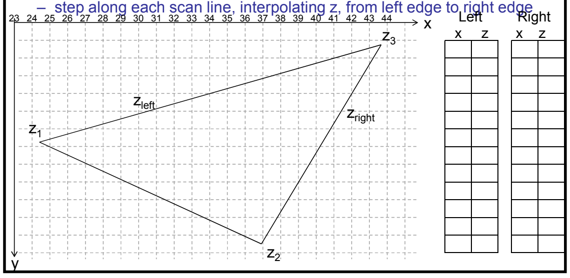
Which two edges?

- Interpolate in two stages:
  - Interpolate between vertices to get left and right edge
  - Interpolate in x direction between left edge and right edge

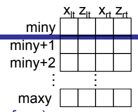



## Rasterisation and EdgeLists

- Finding the image pixels (and depths) on a polygon:
  - step along each edge, in y direction, by pixel, recording x and z
    → "EdgeLists" left and right values of x and any values to interpolate
  - step along each scan line, interpolating z, from left edge to right edge



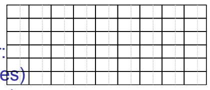
## Compute EdgeLists




**input**: polygon (triangle) (vertices $v_1,v_2,v_3$)
**output**: array of EdgeList entries
starting row of edge list
**actions**:
miny ← round($\min_i$ of $v_i$.y), maxy ← round($\max_i$ of $v_i$.y)
initialise array EdgeList[*height of image* ] (init with [∞ ,∞, -∞, ∞])
**for each** edge in polygon **do**
$v_a$← vertex with smallest y value, $v_b$← other vertex
mx = ($v_b$.x-$v_a$.x)/($v_b$.y-$v_a$.y), mz = ($v_b$.z-$v_a$.z)/($v_b$.y-$v_a$.y),
x ← $v_a$.x, z ← $v_a$.z, *// and any other values to interpolate*
i ← round($v_a$.y), maxi ← round($v_b$.y)
**repeat**
put x and z into EdgeList[i] ( as left or right, depending on x)
i++, x ← x + mx, z ← z + mz
**until** i >= maxi
put $v_b$.x and $v_b$.y into EdgeList[maxi]
**return** edgelist array

## Hidden surface removal

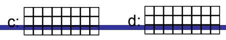
- Which (bits of) polygons are in front and which are behind?
- Z-buffer rendering:
  - render each polygon into a z-buffer:
    - an image (2D array of pixel values)
    - with z value (depth) for each pixel



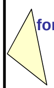
  - Only copy pixels into z-buffer if they are closer than current pixel
  - hidden parts of image automatically disappear

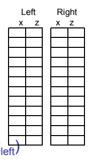
**for** each polygon:
step along each edge from the edge lists of the polygon
interpolating z (and shading and…) as you go
insert shading value into z-buffer if z less than current entry

## Rendering with Edgelists & Z-buffer




**initialise** :
Zbuffer.c ← array [0..imageWidth] [0..imageHeight] of Color
Zbuffer.d ← array [0..imageWidth] [0..imageHeight] of ∞





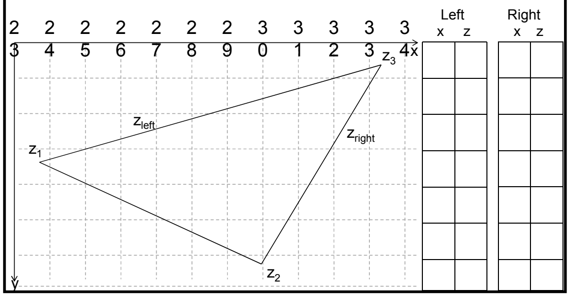

**for** each polygon **do**
compute edge lists EL, and shading
(EL is an array of ⟨$x_{left}$, $z_{left}$, $x_{right}$, $z_{right}$⟩ )
**for** y ← 0 to edgelists.length -1 **do**
x ← round(EL[y] . $x_{left}$ ), z ← EL[y] . $z_{left}$
mz ← (EL[y]. $z_{right}$– EL[y]. $z_{left}$) / (EL[y]. $x_{right}$– EL[y]. $x_{left}$)
**while** x <= round(EL[y]. $x_{right}$ ) **do**
**if** z < Zbuffer.d[x][y] **then**
Zbuffer.d[x][y] ← z, Zbuffer.c[x][y] ← shading
x++, z ← z + mz

check if x is in bounds!

copy Zbuffer.c to image

# Building the EdgeLists

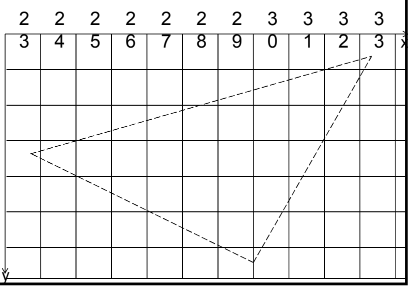
- step along each edge, by scan line, recording x and z
  → "EdgeLists" left and right values of x and any values to interpolate




# Filling the z-buffer

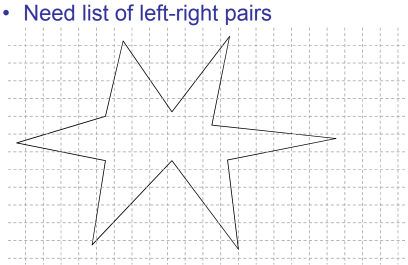
- step along each scan line, from left to right
- interpolating the z values
- put z and colour into z-buffer if smaller than current z

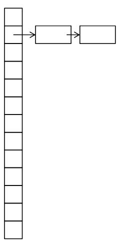
| Left x | Left z | Right x | Right z |
|---|---|---|---|
| 33.2 | 15 | 33.2 | 15 |
| 30.0 | 17.2 | 32.6 | 14.5 |
| 26.8 | 19.4 | 32.1 | 14 |
| 23.7 | 21 | 31.4 | 13.5 |
| 25.8 | 18 | 30.8 | 13 |
| 27.9 | 15 | 30.2 | 12.5 |
| 30 | 12 | 30 | 12 |




# Non triangle Polygons

- Need list of left-right pairs





## Costs and efficiency issues

- Cost:
  - fixed # operations for each pixel along each edge of each polygon
  - fixed # operations for each pixel in area of each polygon.
  - ⇒ scales with the number and size of the polygons

- Efficiencies:
  - minimise multiplications and divisions
  - remove recalculations
    - (eg, calculate bounds of loops at start, not each time)
  - integer vs floating point
    - use integer where possible, BUT
    - GPU on modern video cards optimised for floating point
  - memory access and minimising pointers.
    - eg, use 32 bit integer for colour, not a Color object
    - use arrays, not arrayLists (use C, not Java)

## GPUs

- High end video cards have GPU and own memory
  - optimised for floating point operations
  - high parallelism
  - own memory, image buffers and Z-buffers
  - efficient operations for lots of graphics processing, including
    - rotations, translations, etc
    - computing shading and texture mapping
    - Z-buffer operations

- Also useful for non-image applications, eg scientific calculation!
  - ⇒ specialised programming for GPU's