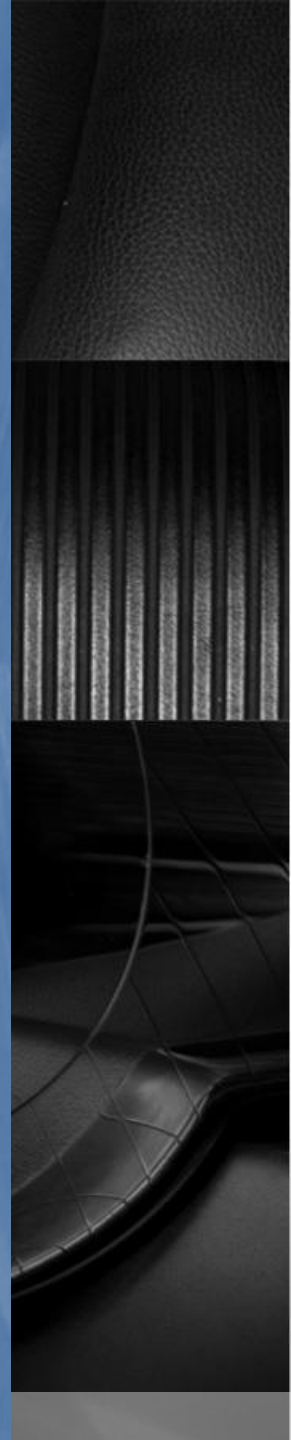
# Web Based Graphics & Virtual Reality Systems

## Rendering Pipeline, Hidden Surface and Surface Mesh Modeling

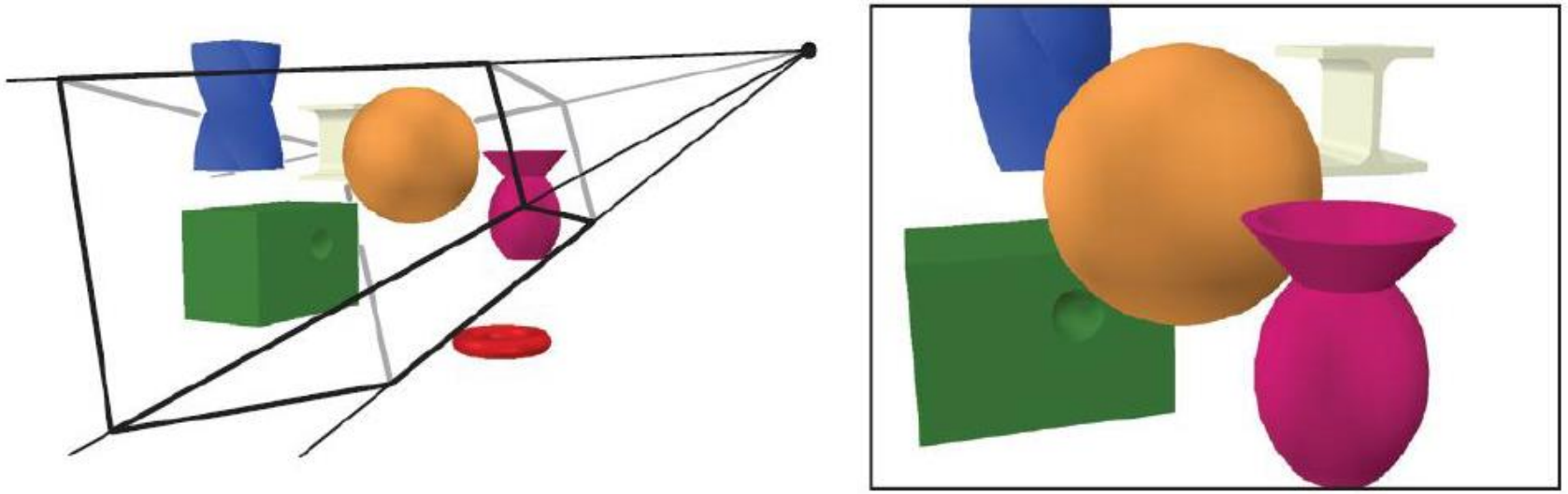


# Recap

- In the previous lectures, we studied many basic components available in graphics rendering engine and the related concept and computations.
- This lecture, we take a look of the full picture how these components are achieved via the graphics rendering pipeline.

# Rendering Pipeline

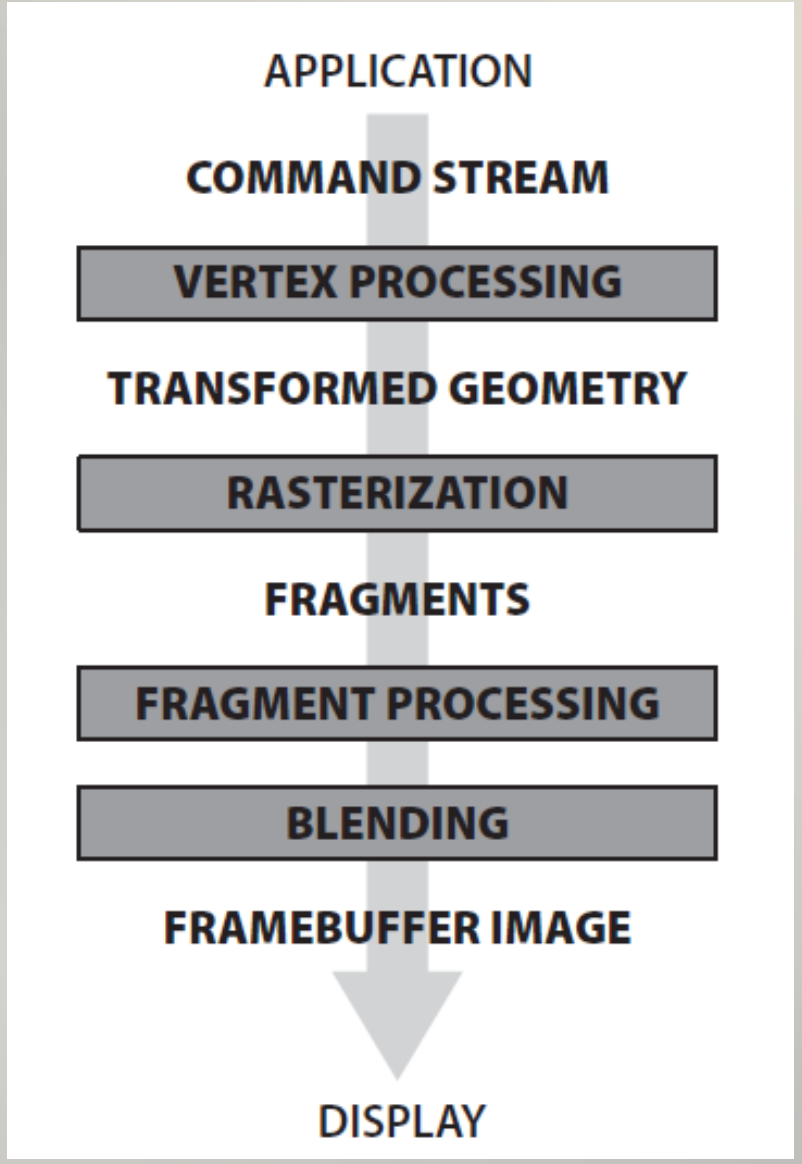
- Flow and steps in rendering
  - From geometric primitives to the output image of the screen



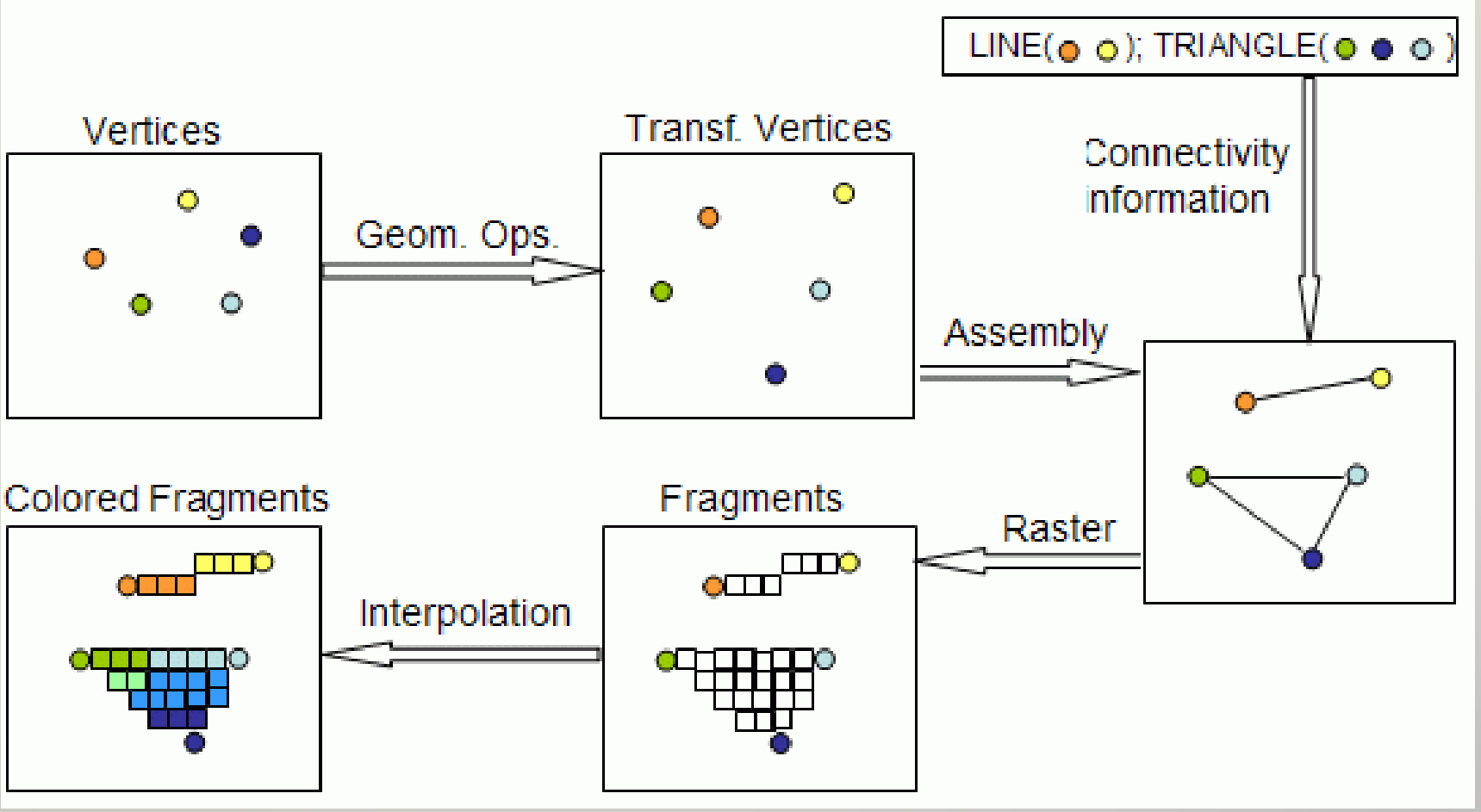
[Akenine-Moeller et al.: Real-time Rendering]

# Rendering Pipeline

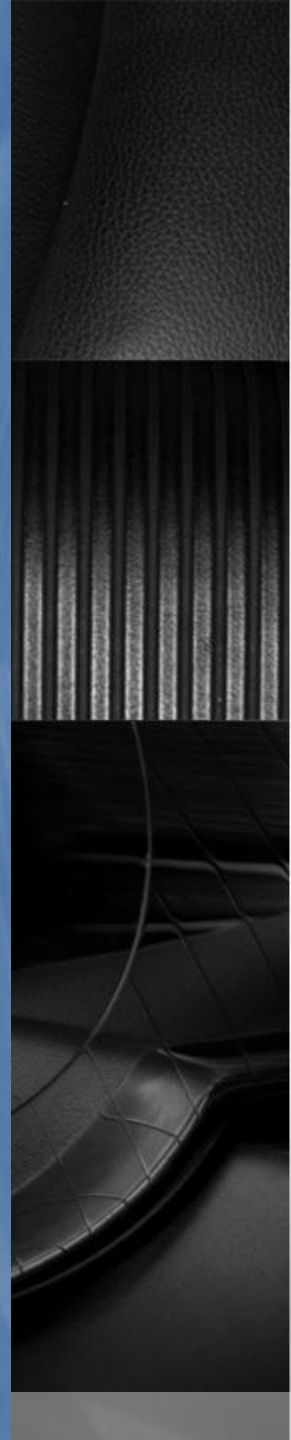
- Most real-time rendering engine follow the pipeline standard
  - Graphics hardware is hardwired
- Major Steps
  - Vertex Processing
  - Rasterization
  - Fragment Processing
  - Blending



# Rendering Pipeline



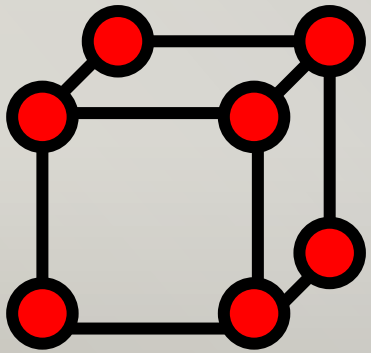
# Vertex Processing

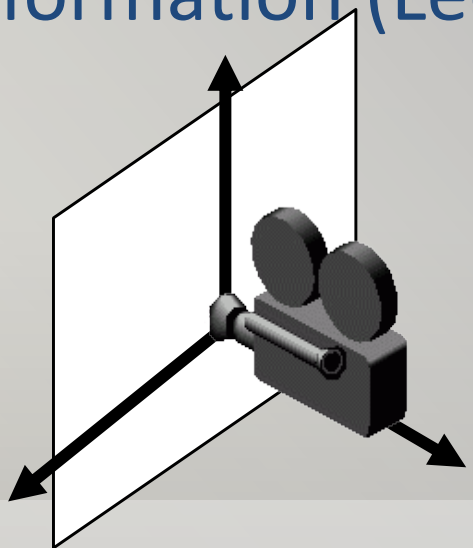


# Vertex Processing

Apply to all vertices independently

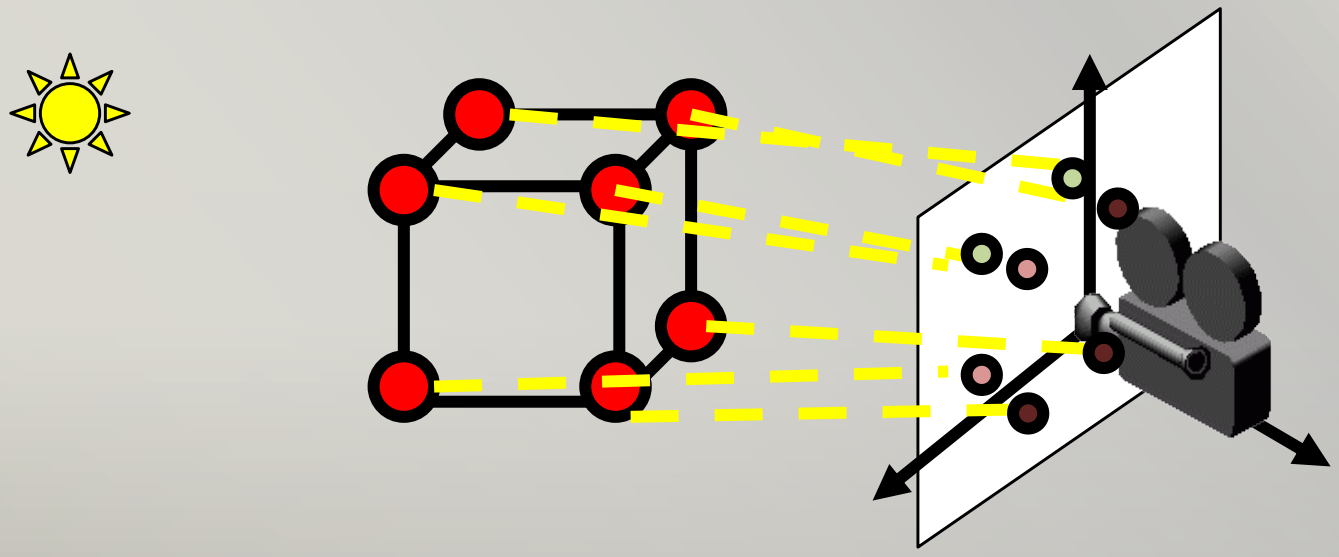
1. The modelview transformation matrix (Lec. 2)
2. The lighting model (Lec. 4)
3. The projection matrix (Lec 3)
4. Clipping and view transformation (Lec 3)





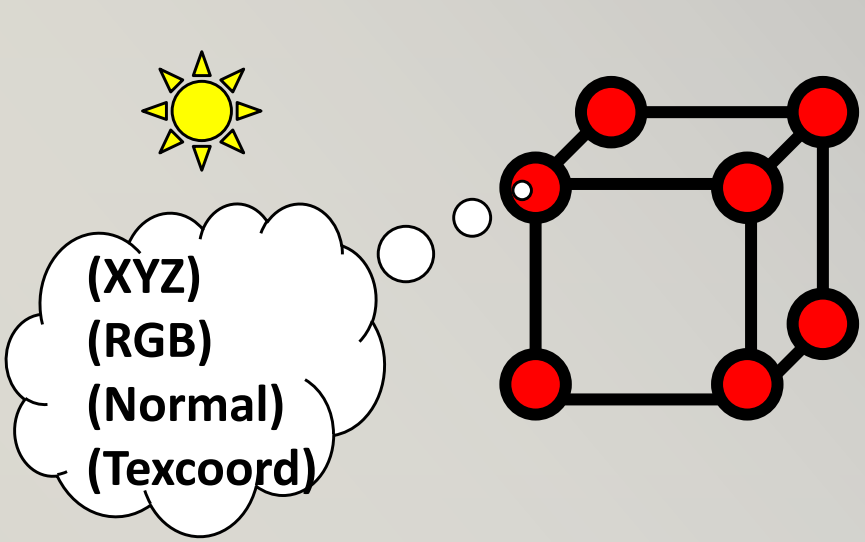
# Vertex Processing

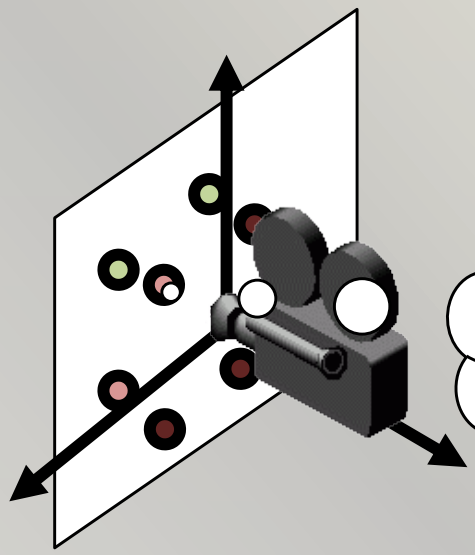
- The outcome of this process is a set of projected vertices in screen space
- The color of the vertices are modified by lighting model



# Vertex Processing

- Within the pipeline, the associated info of vertices are under processing



(X” Y”)
(RGB)
(Normal)
(Texcoord)

Model View Matrix

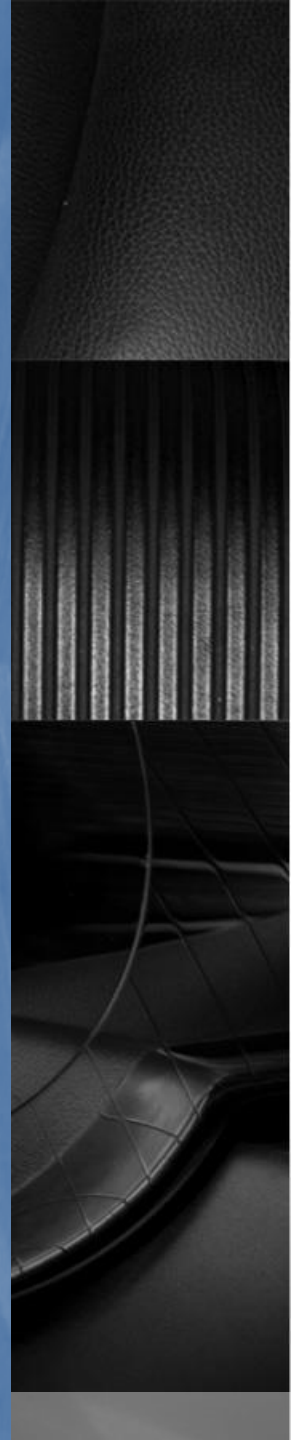
(X’Y’Z’)
(RGB)
(Normal)
(Texcoord)

Light Model

(X’Y’Z’)
(RGB)
(Normal)
(Texcoord)

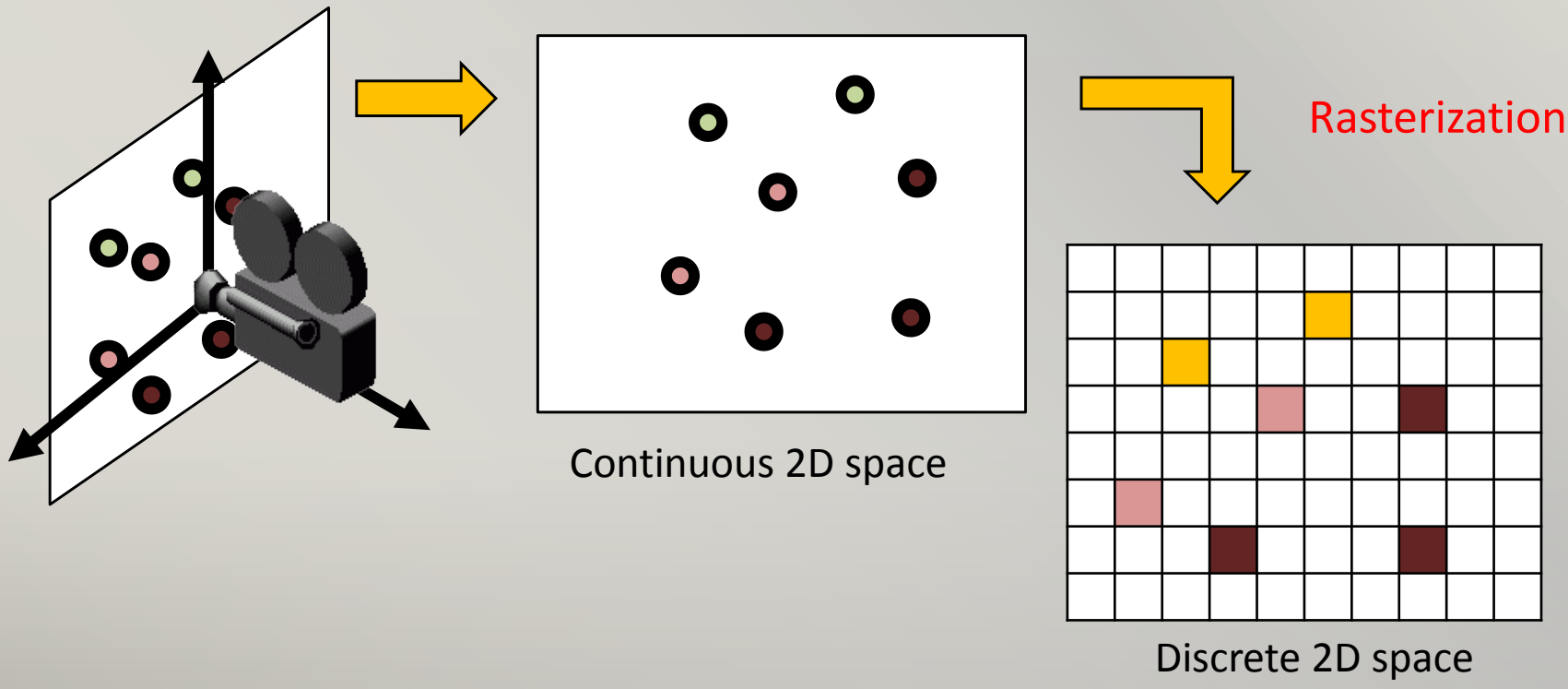
Projection Matrix

# Rasterization



# Rasterization

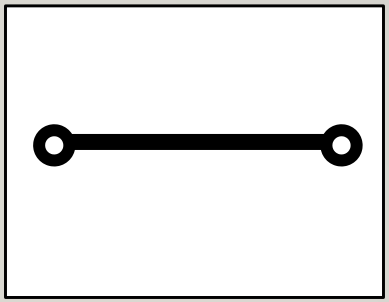
- The process of transforming geometric primitives (e.g. lines and polygon) into raster 2D image representations (i.e. pixels)

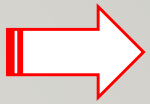


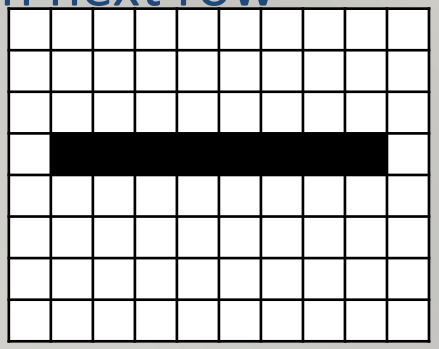
# Rasterizing Line

- Rasterizing a point is relatively easy
  - Convert floating point to integer coordinate
- Rasterizing a line may be a bit complicated
  - The decision on where to draw on next row

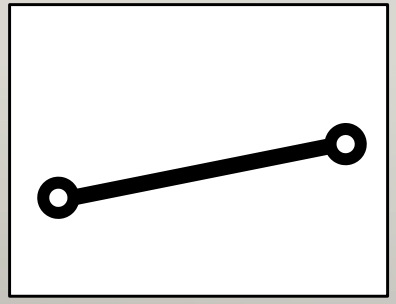
Exact horizontal line

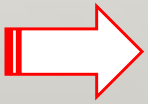


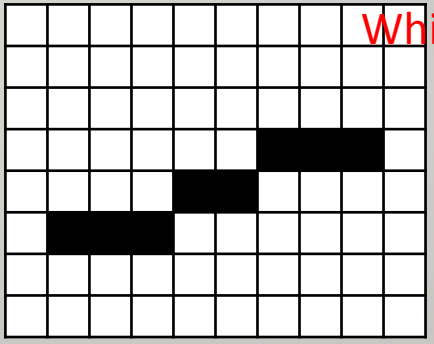




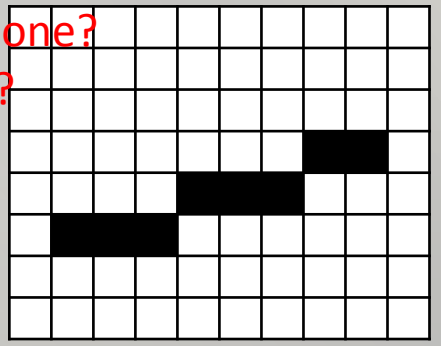
Line with slope







Which one?
???



# A Simple Solution

- Accumulate the error which is defined by

  Derr = abs( x1- x0 / y1- y0)

- We start with y = y0 and error =0
- Loop from x = x0 to x1

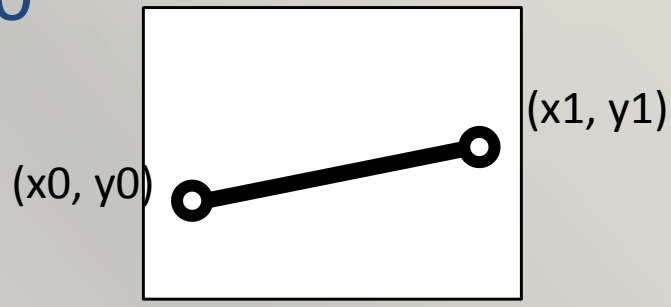
```
draw(x,y)
error += Derr
if (error >= 0.5)
    y++ or y--
    error--
```

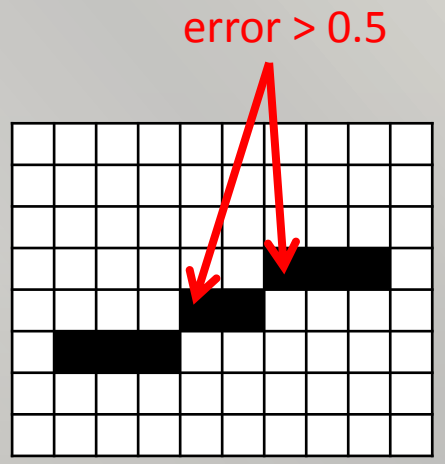
Draw at pixel (x,y)

Accumulate the error

Increment or decrement y

Reduce the error



# Bresenham algorithm

- The previous algorithm requires the use of floating point numbers, (e.g. error and Derr)
  - Very expensive for computers in old days
- Bresenham algorithm uses only integer operations which are fast
- The method is trying to replace the use of current error computation
  - i.e. Derr = abs( x1- x0 / y1- y0)
  - Which is always a floating point

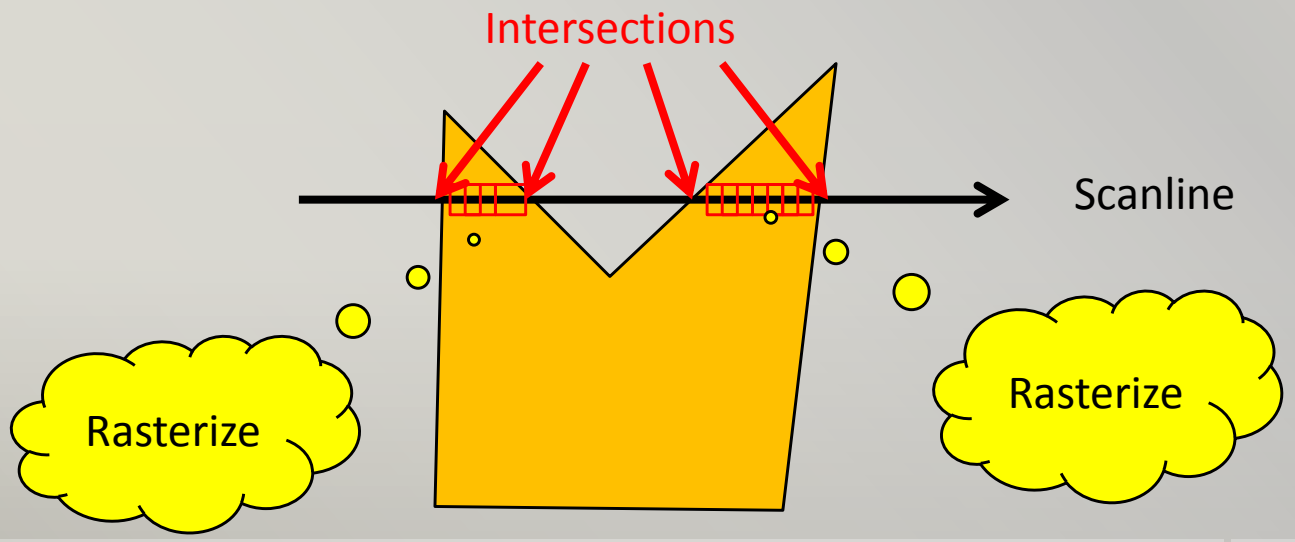
# Bresenham algorithm

- We have integer Error = Dy – Dx
  - Increment Error by Dy (integer) at each step
  - Decrement by Dx (integer) if Error becomes positive

```
Dx = abs(x1-x0)
Dy = abs(y1-y0)
Error = Dy – Dx
loop  from x = x0 to x1
   draw(x,y)
   if Error >= 0
      y++/ y--          Increment  or decrement y
      Error -= Dx       Reduce the error
   Error += Dy          Accumulate the error
```

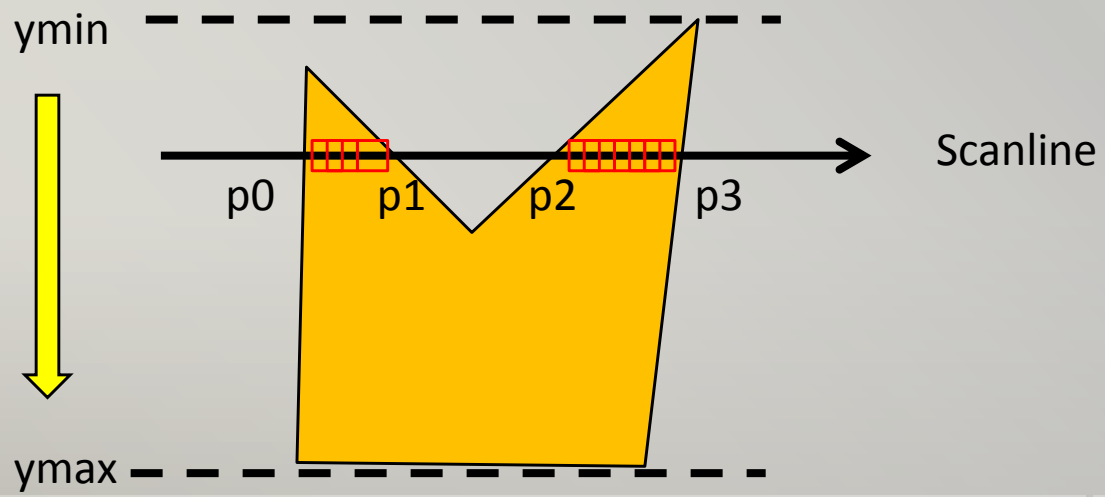
# Rasterizing Polygon

- Rasterizing triangle or polygon can be done quickly with in a scanline like manner
- The scan-conversion method
  - Intersect scanline with polygon edges and fill between pairs of intersections

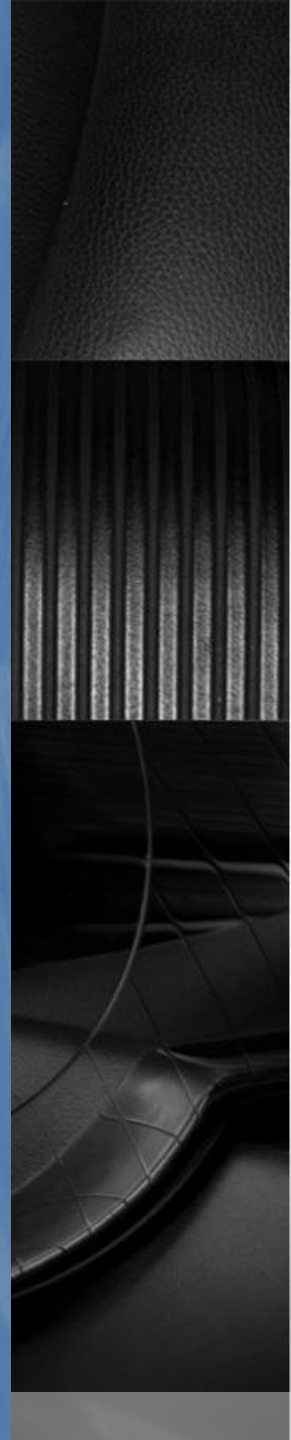


# Scan Conversion Algorithm

- For y = ymin to ymax
  1) intersect scanline y with each edge
  2) sort intersections by increasing x
     E.g. [p0,p1,p2,p3]
  3) draw pixel between pair of intersections (p0 -> p1, p2 -> p3, ....)



# Hidden Surfaces



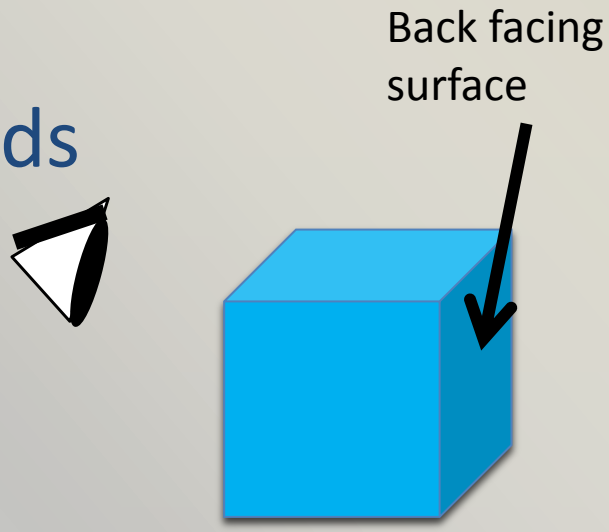
# Hidden Surfaces

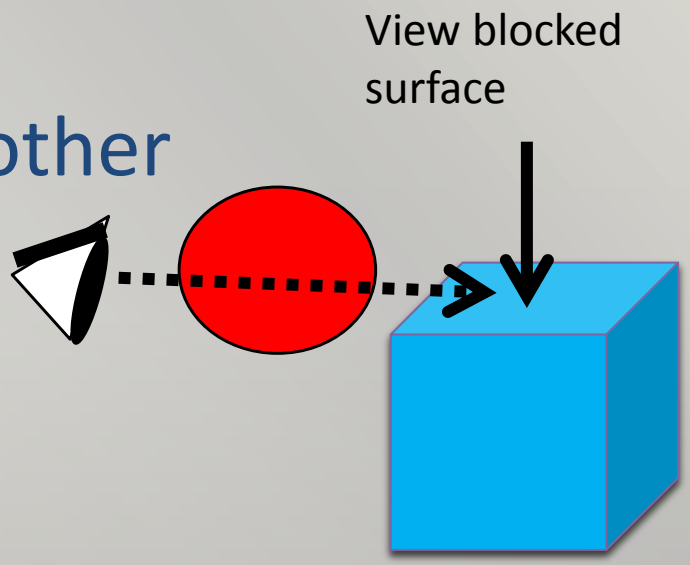
- We found that rasterization is not cheap
  - The more surfaces we draw, the slower we will be
- A very obvious case to reduce what we draw is to ignore surfaces that are not seen from the camera
  - Save lots of unnecessary computations
- We commonly refers to these surfaces as **hidden surfaces**

# Hidden Surfaces

For opaque object, there are 2 kinds of hidden surfaces:

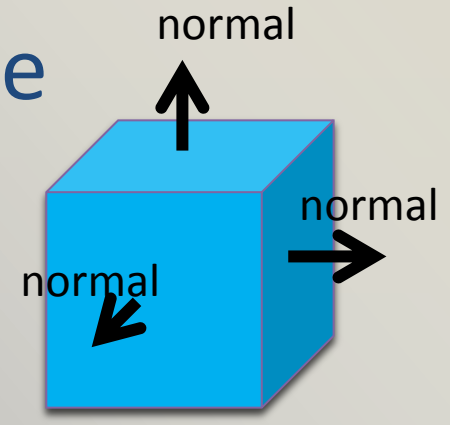
- Surfaces that facing backward towards you
  - Back face culling
  - Based on vertex normals
- Surfaces that are blocked by other surfaces
  - Hidden surface removal
  - Using depth buffer



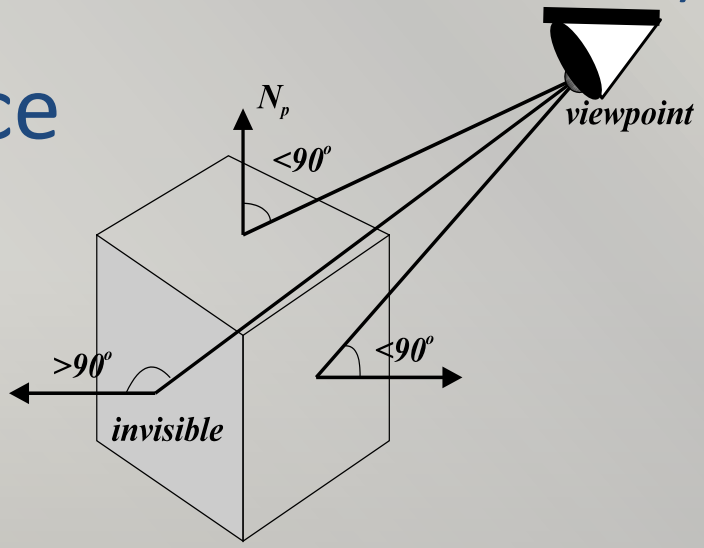



# Back Face Culling

- To define the front face of a surface
  - Need to use normal of surface
  - Normal represents its facing direction
  - It is defined by order of the vertices of the face (will be further discussed next lecture)

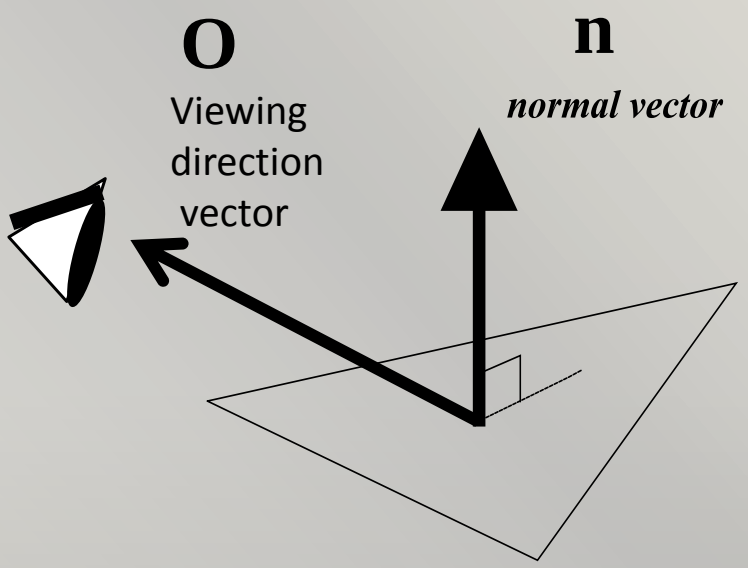



- To check if it is back face
  - Normal and view angle is larger than 90 degree



# Back Face Culling

- A more convenient method to compute using vector math
- To detect if it is back facing
  - Normal vector of face $\mathbf{n}$
  - Viewing direction vector $\mathbf{O}$
- Dot product between $\mathbf{O} \cdot \mathbf{n}$
  - Front face : $\mathbf{O} \cdot \mathbf{n} > 0$
  - Back face : $\mathbf{O} \cdot \mathbf{n} < 0$

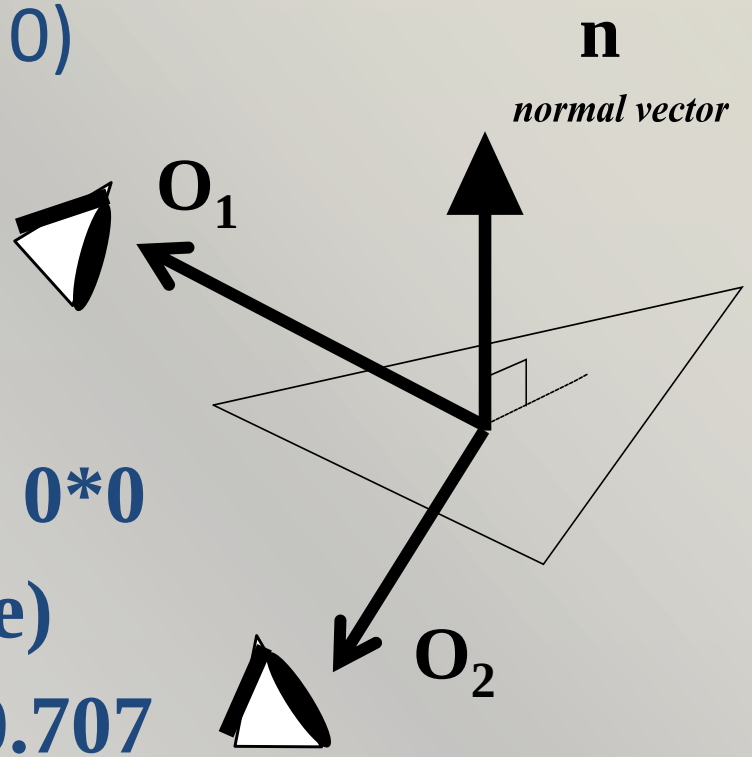


# A Numerical Example

- E.g. surface normal **n** = (0, 1 , 0)

$O_1$ = (0.707, 0.707 , 0)

$O_2$ = (0, -0.707 , 0.707)




- $\mathbf{O}_1 \cdot \mathbf{n} = 0*0.707 + 1*0.707 + 0*0$

  $= 0.707 > 0$ **(front face)**

- $\mathbf{O}_2 \cdot \mathbf{n} = 0*0 + 1*-0.707 + 0*0.707$

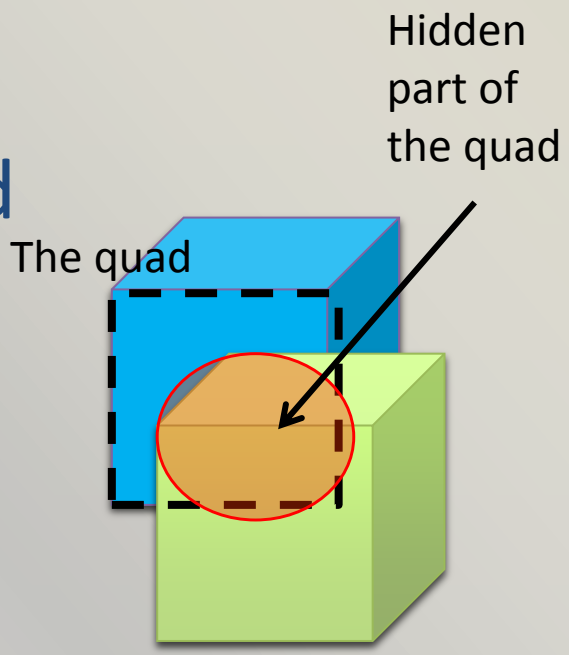
  $= -0.707 < 0$ **(back face)**

# Back Face Culling in OpenGL

- Back face culling is a build in feature in OGL
- We can simply invoke glEnable to activate back face culling in OpenGL
  - glEnable(GL_CULL_FACE)
- To choose the kind of faces to ignore:
  - glCullFace(GL_BACK)
  - This is the default which tries to remove back faces
  - Other possible values: GL_FRONT or GL_FRONT_AND_BACK

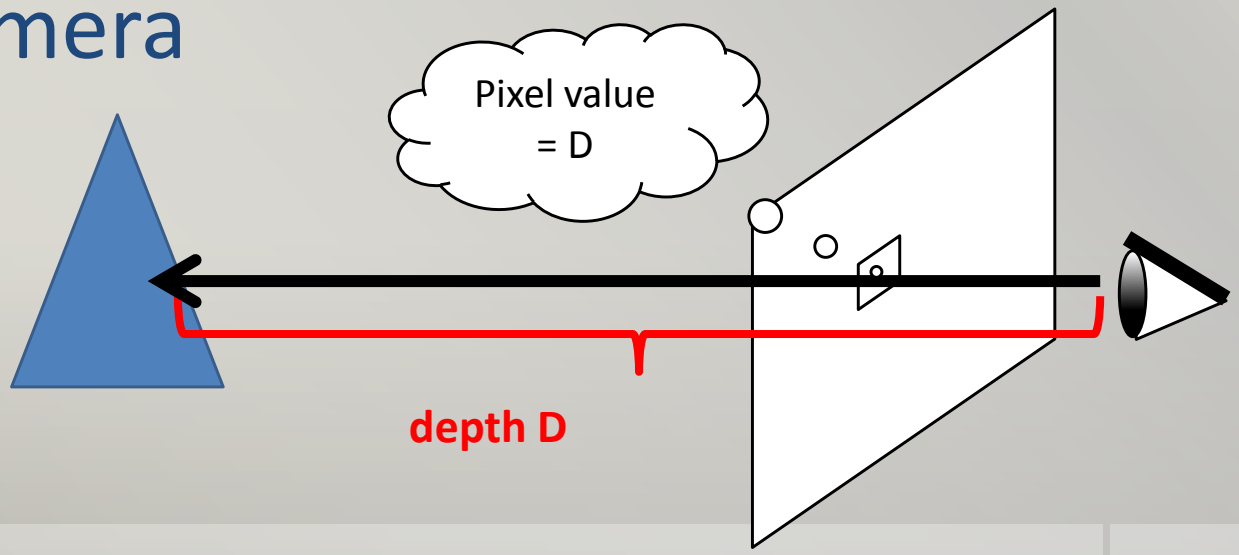
# Hidden Surface Removal

- Happen when a surface is blocked by another surface
- However, we can not perform the removal in vertex or surface level
- Since there are cases that the polygon is only partially hidden
  - If we remove whole surface, the unhidden part will also be removed



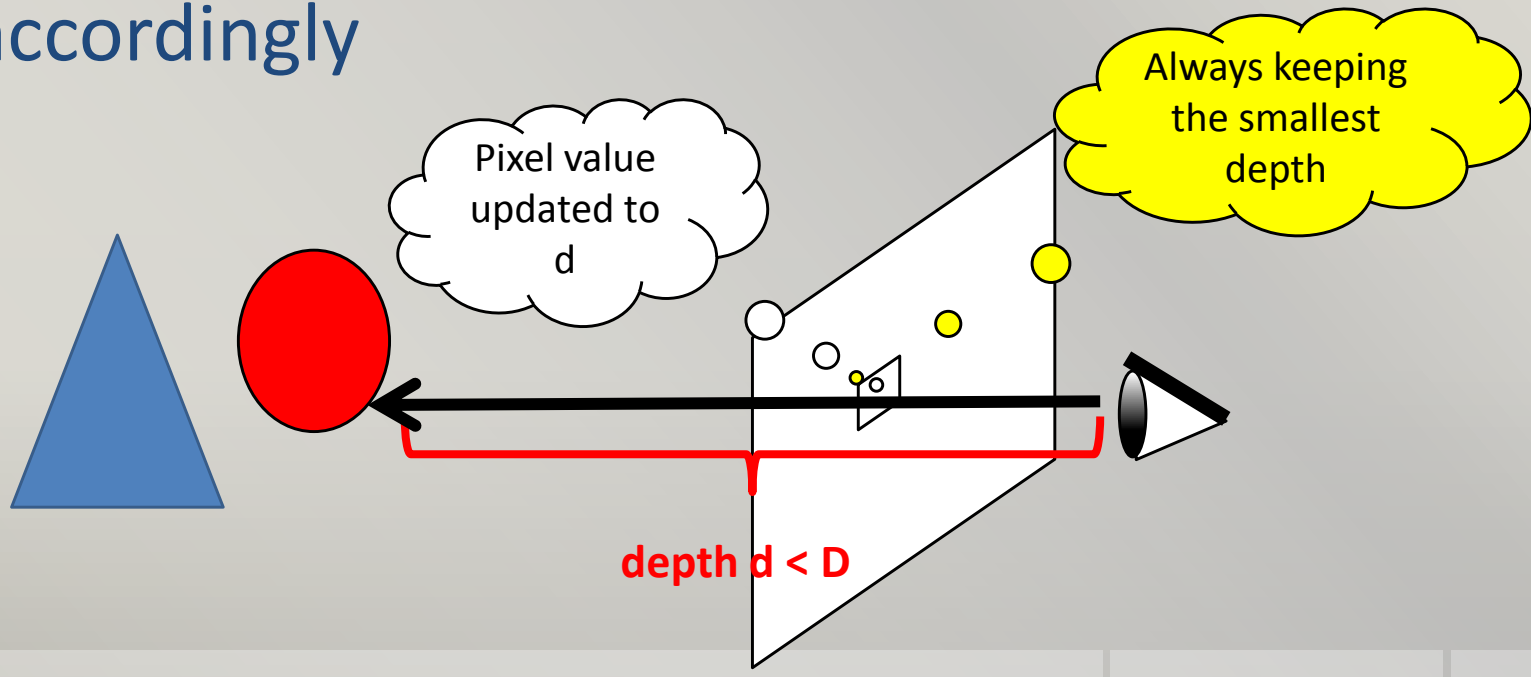
# Z-Buffer for Hidden Surface Removal

- To solve this problem, a special frame buffer is used – **Z-Buffer** (or "Depth Buffer")
- Similar to the color buffer (screen), but it stores the closest surface and its depth to the camera



# Z-Buffer for Hidden Surface Removal

- Whenever a new surface is drawn on the screen, we check with the Z-buffer if the pixels had already drawn with closer surfaces or not
- If not, we can render on screen and update the Z-buffer accordingly



# Z-Buffer for Hidden Surface Removal

- E.g. 2 triangles are drawn, red one is closer and cover partly the green one

Drawing:

| ∞ | ∞ | ∞ | ∞ |
|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ |

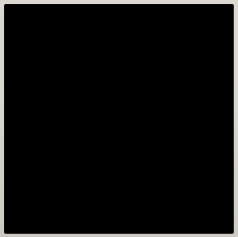
Z-buffer

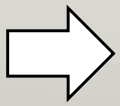
| 9 | ∞ | ∞ | ∞ |
|---|---|---|---|
| 9 | 9 | ∞ | ∞ |
| 9 | 9 | 9 | ∞ |
| 9 | 9 | 9 | 9 |

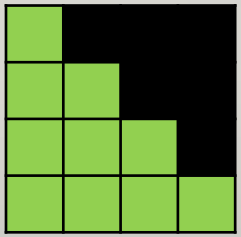
All smaller than ∞

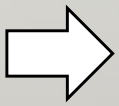
| 9 | ∞ | ∞ | 3 |
|---|---|---|---|
| 9 | 9 | 3 | 3 |
| 9 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 |

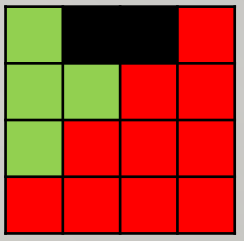
All smaller than ∞ and 9







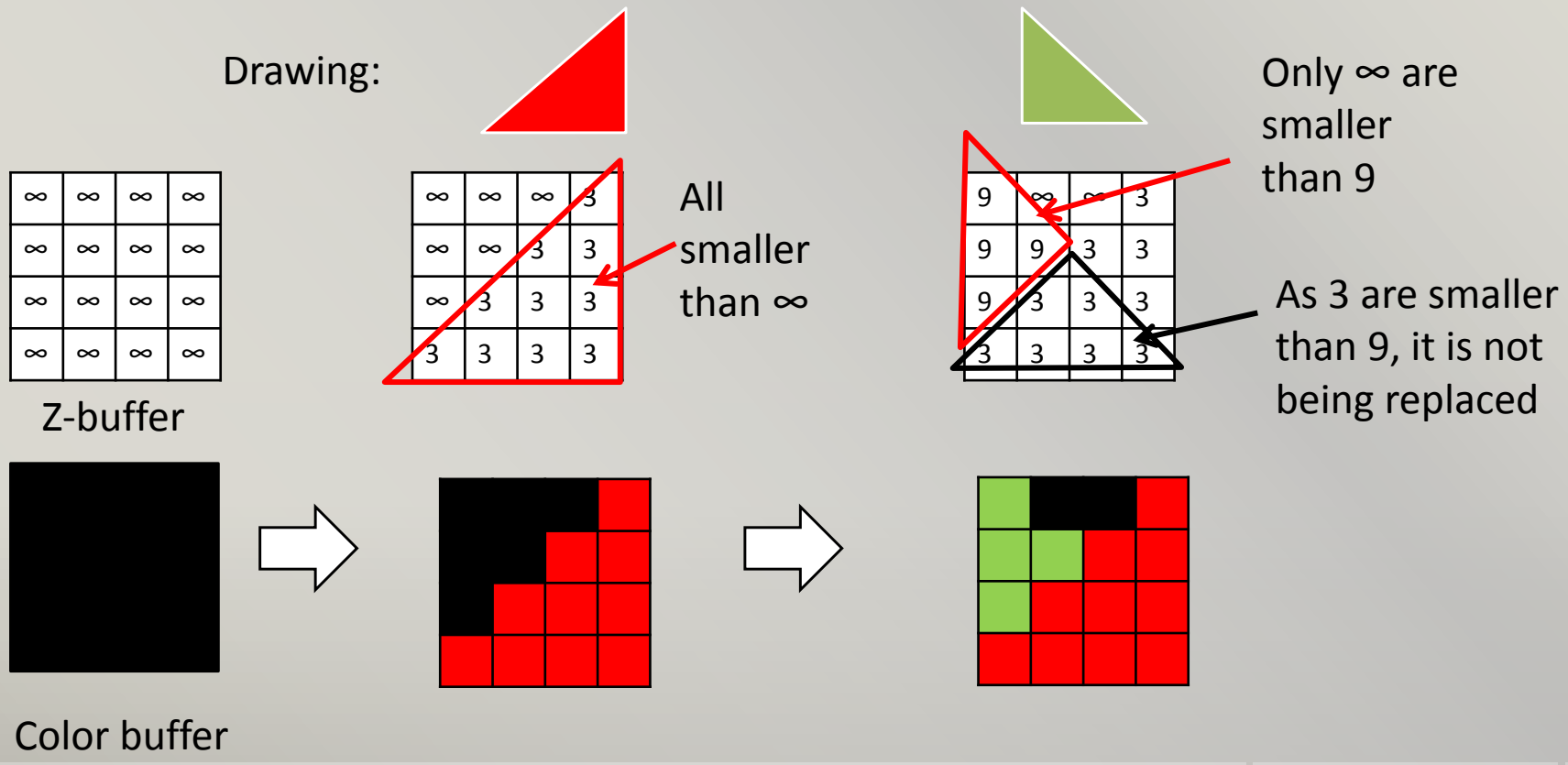




Color buffer

# Z-Buffer for hidden surface removal

- Notice that even the drawing order changed, the hidden surfaces are drawn correctly



# Hidden Surface Removal in OpenGL

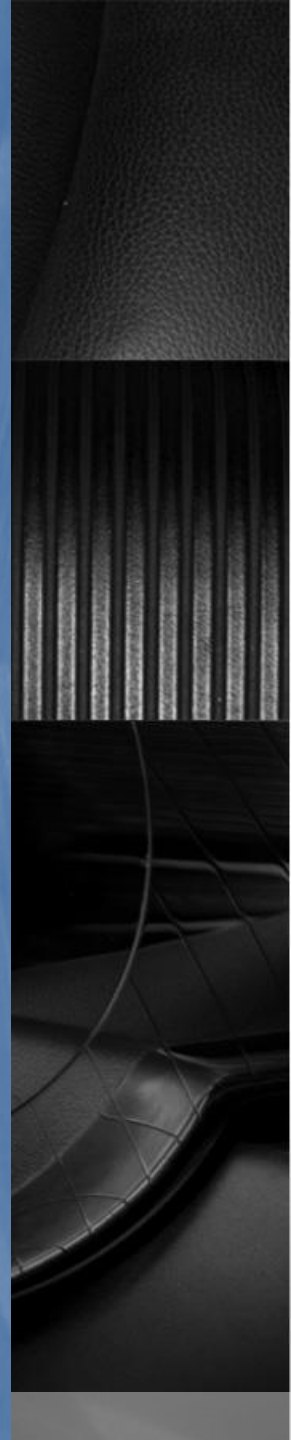
- Hidden surface removal is also build-in in OGL
- We can invoke glEnable to activate it

  glEnable(GL_DEPTH_TEST);
- Worth to note that if hidden surface removal is not being enabled
- The color buffer are drawn purely based on the drawing order of the surfaces

# Back Face Culling and Hidden Surface Removal

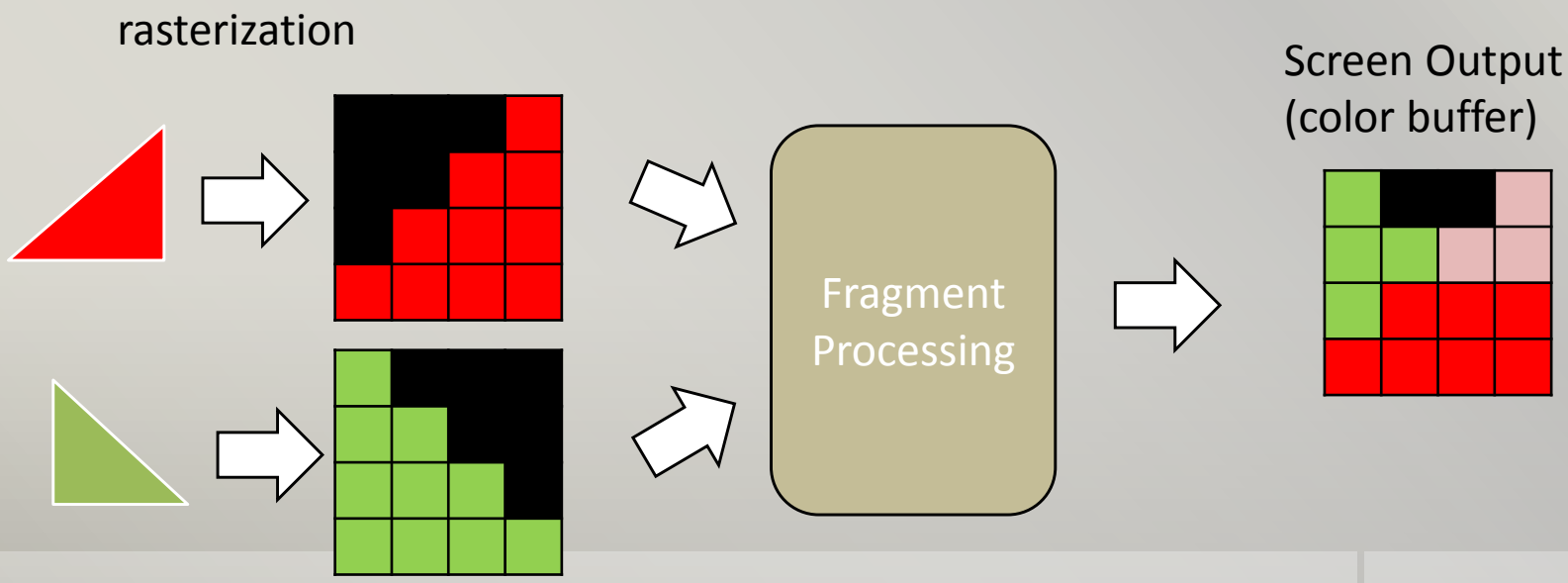
- A short summary on the 2 methods for hidden surfaces
- Back face culling is applied on **vertices and surfaces**
  - Whole surface will be removed if it is back-facing
- Hidden surface removal is applied on **fragments (or rastered pixels)**
  - Z-buffer is used to keep track of the closest fragment's depth to the camera

# Fragment Processing



# Fragment Processing

- The result of rasterization will be a set of fragments for each surface geometry
- Before output to the color buffer for screen display,
- We go through further processes to determine the final output colors in color buffer



# Fragment Processing

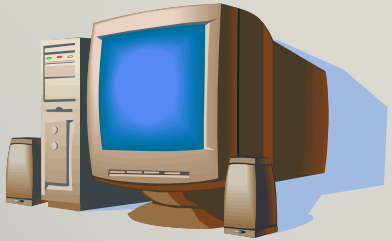
Fragment processing may include,

- shading (Lec 4),
  - E.g. Flat, Gouraud shading
- texture mapping (Lec 4),
- depth test,
- blending (Lec 5) and etc.

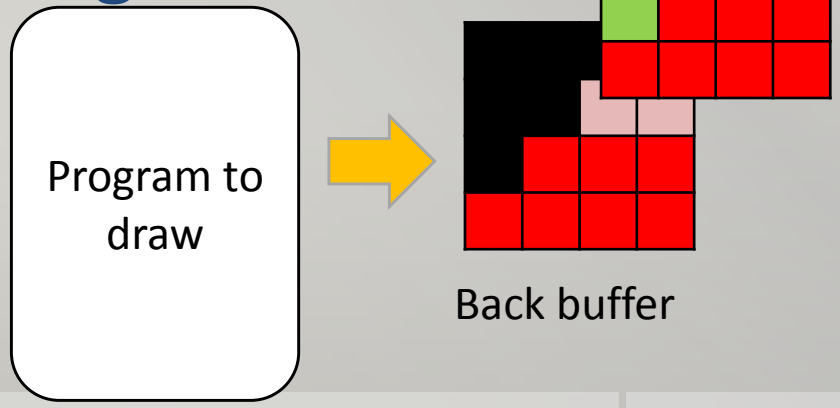
The final result of all the fragment process will be output to the **color buffer (or frame buffer)** for display

# COLOR Buffers and Double Buffering

- Two color buffers are commonly used
- Front buffer
  - for display
- Back buffer
  - draw in background
- Called **“Double buffering”**
-

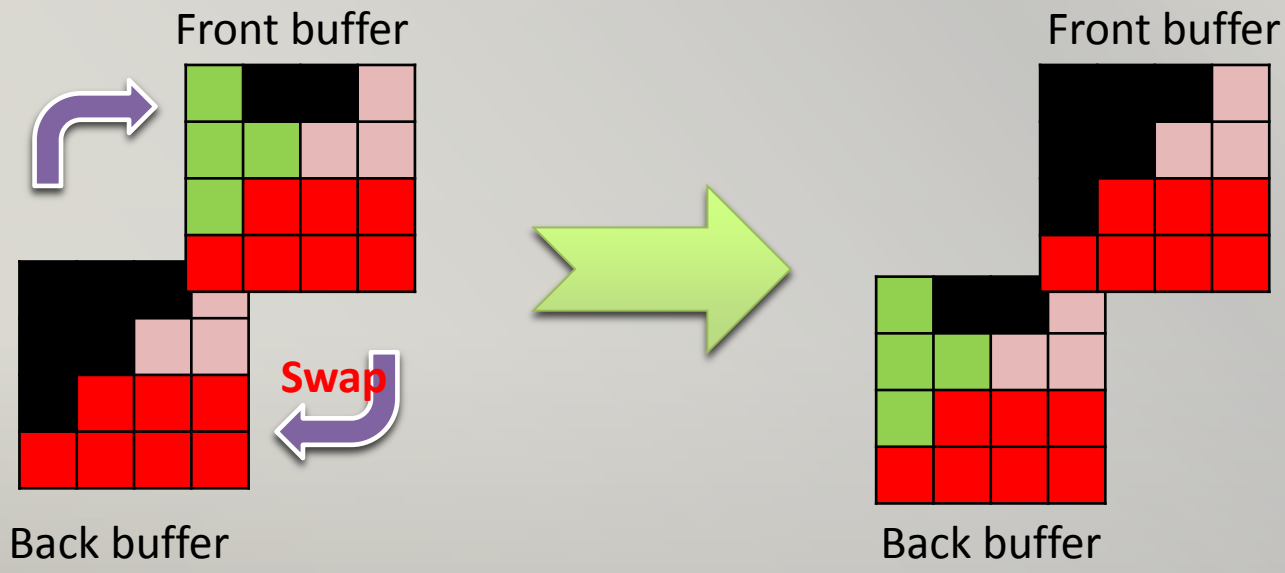


Front buffer



# Double Buffering

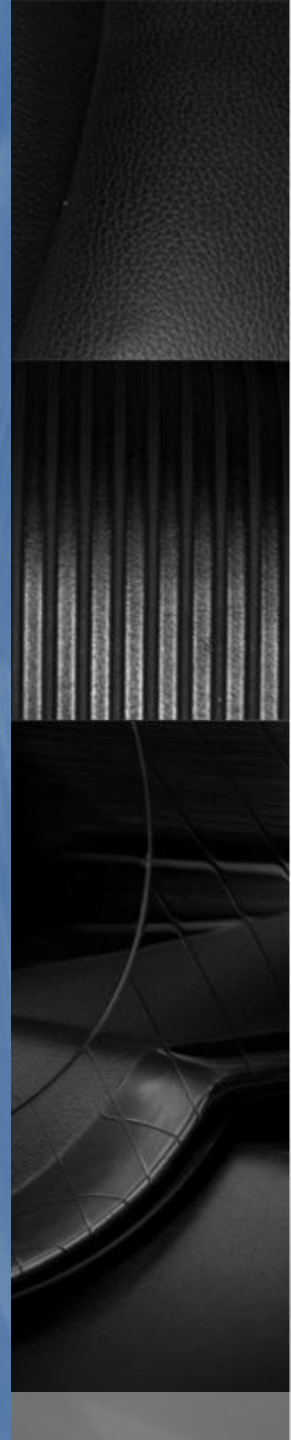
- Swap happen when the back buffer is done
- Ensure all drawing are done before display
- To avoid flicker



# Summary

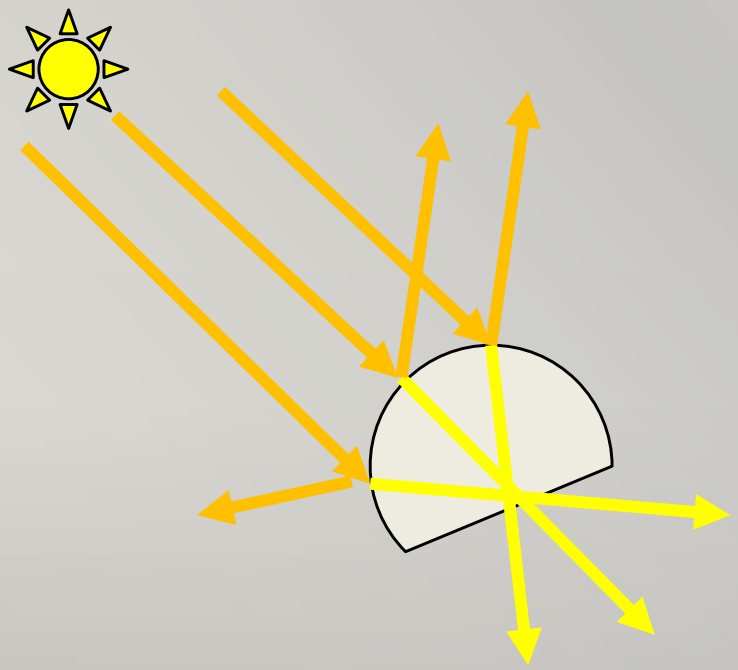
- We have a more comprehensive view on the whole rendering process in real-time rendering engines
- Major steps in rendering pipeline includes
  - Vertex processing
  - Rasterization
  - Fragment processing
- Back face culling and Hidden surface removal are common methods to ignore hidden surfaces in rendering

# Surface Modeling, Mesh and Scene Graph



# Surface Representations

- When referring to 3D models, usually we are talking about the **surface model**
  - Most of the light interaction happens at surface of 3D model

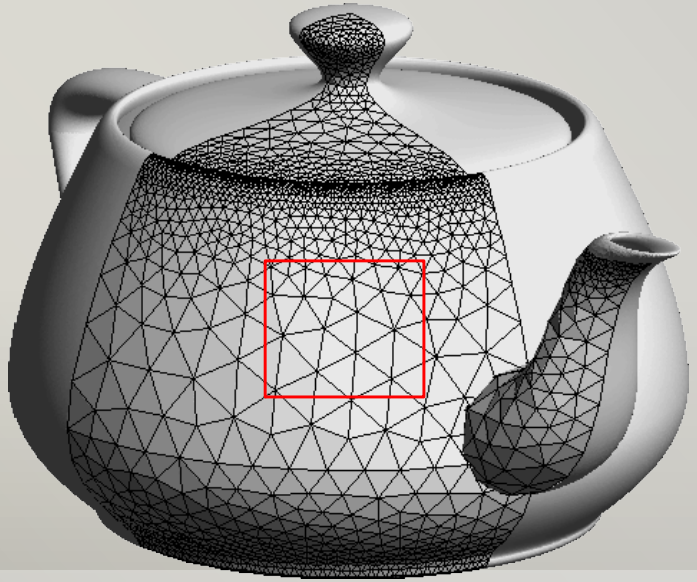


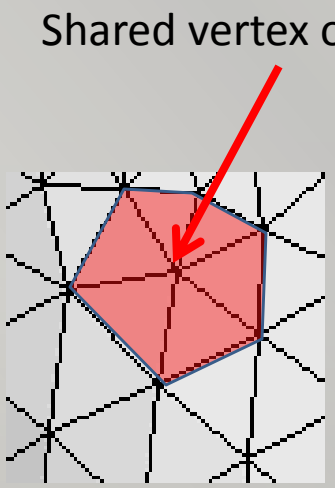
# Surface Representations

- To represent the surface in computer
  - Surface representation
    - Triangular mesh
    - Parametric surfaces
  - Volumetric representation
    - CSG
    - Iso-surface

# Triangular Mesh

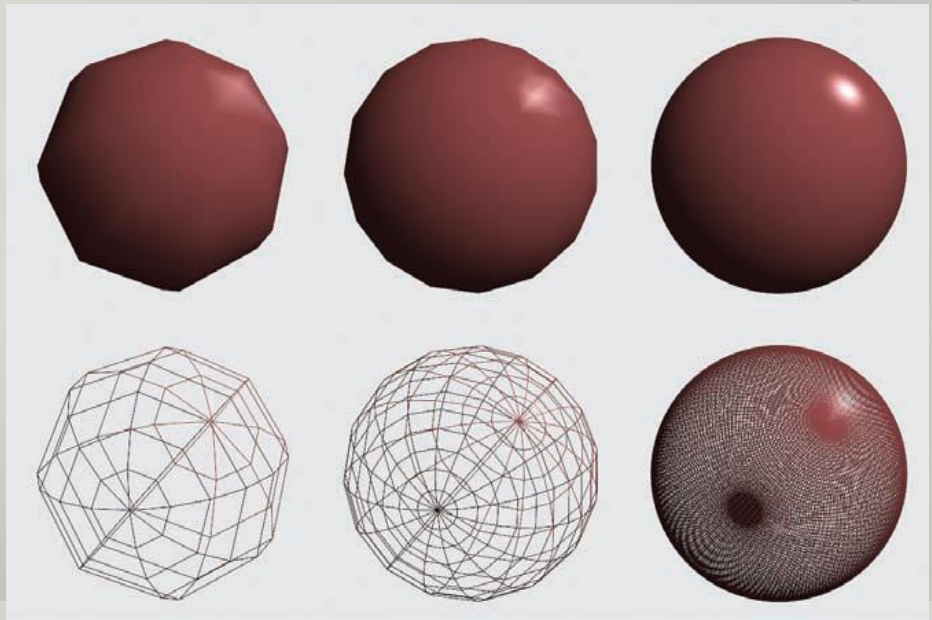
- Composed of complexes of triangles with shared vertices
- Like cutting a surface into a network of triangles
  - Interconnected triangles





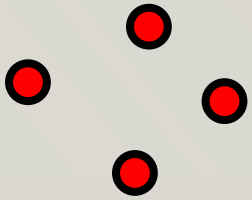
# Triangular Mesh

- Each triangle is a planar surface
  - Difficult to model smoothly changing surface with little number of triangles
- The more the number of small triangles in a mesh, the closer it models the original surface

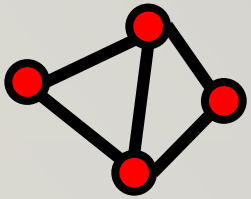


# Vertices, Edges, Triangles and Mesh

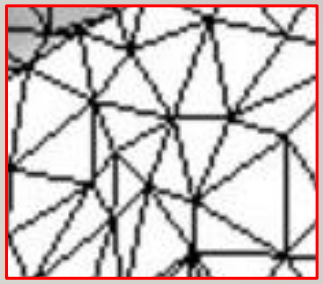
- Vertices + Edges/connectivity = Triangle
- Many connected Triangles = Mesh



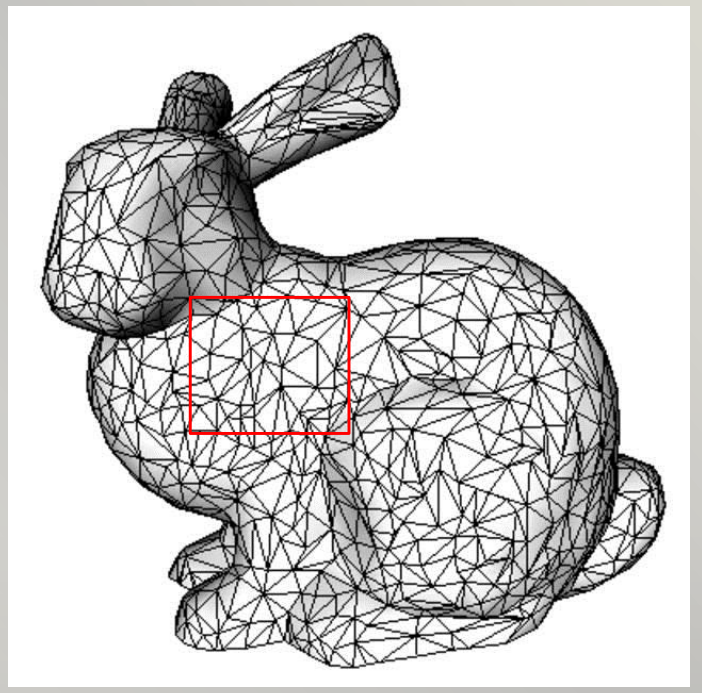
vertices



Edges/
connectivity



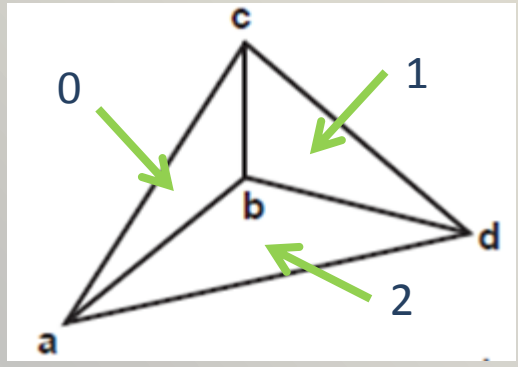
Triangles



Mesh

# Data Structure for a Mesh

- Ways to store or represent data in a mesh



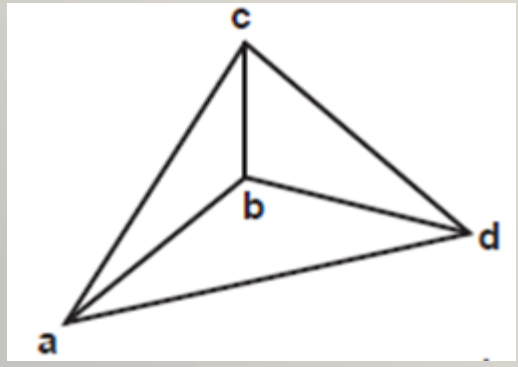

- Store separate triangles
  - Duplicated vertices positions (all in floating point)
  - Waste storage

separate triangles:

| # | vertex 0 | vertex 1 | vertex 2 |
|---|---|---|---|
| 0 | $(a_x, a_y, a_z)$ | $(b_x, b_y, b_z)$ | $(c_x, c_y, c_z)$ |
| 1 | $(b_x, b_y, b_z)$ | $(d_x, d_y, d_z)$ | $(c_x, c_y, c_z)$ |
| 2 | $(a_x, a_y, a_z)$ | $(d_x, d_y, d_z)$ | $(b_x, b_y, b_z)$ |

# Data Structure for a Mesh

- Shared vertices
- Triangles formed by referring to the vertex indices
- Save storage
  - No duplicate vertex positions
  - Indices are integers




separate triangles:

| # | vertex 0 | vertex 1 | vertex 2 |
|---|---|---|---|
| 0 | $(a_x, a_y, a_z)$ | $(b_x, b_y, b_z)$ | $(c_x, c_y, c_z)$ |
| 1 | $(b_x, b_y, b_z)$ | $(d_x, d_y, d_z)$ | $(c_x, c_y, c_z)$ |
| 2 | $(a_x, a_y, a_z)$ | $(d_x, d_y, d_z)$ | $(b_x, b_y, b_z)$ |

shared vertices:

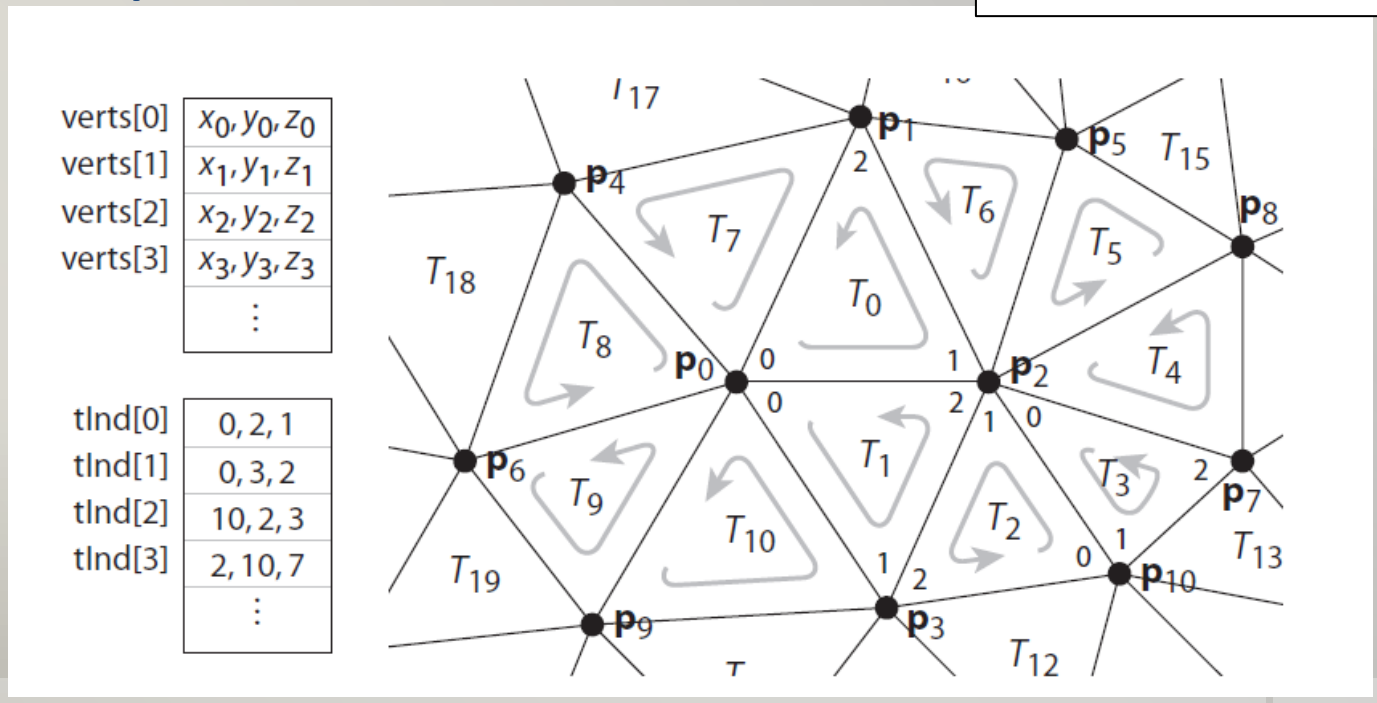
| triangles | | vertices | |
|---|---|---|---|
| # | vertices | # | position |
| 0 | (0, 1, 2) | 0 | $(a_x, a_y, a_z)$ |
| 1 | (1, 3, 2) | 1 | $(b_x, b_y, b_z)$ |
| 2 | (0, 3, 1) | 2 | $(c_x, c_y, c_z)$ |
| | | 3 | $(d_x, d_y, d_z)$ |

Reference using index

# Data Structure for a Mesh

- A mesh stored in this way is also called **Indexed triangle mesh**
- Arrays are normally used in an implementation

```
IndexedMesh {
    int tInd[nt][3]
    vector3 verts[nv]
}
```



# File Formats for Triangular Mesh

- Data of triangular mesh are stored for data exchange and reuse
- Commonly seen 3D file formats:
  - OBJ, STL, DXF, 3ds, Collada
- Most 3D editing software use their own format
- However, the way and structure to store mesh data are very similar
  - Commonly Index Triangle Mesh

# STL and OBJ file Format

- Let’s take STL and OBJ as examples
  - Human readable

    ASCII

    also a binary version
  - STL stores only geometry information
  - OBJ support also

    texture coordinates

    Color and texture in separate file

    No compression

# 3D Systems STL (ASCII)

All data are stored in markup format

- solid (define an object)
- facet (a triangle)
- normal (facet normal)
- vertex (vertex position)
  - Stored in floating point of the x,y,z coordinates

```
solid cube_corner
  facet normal 0.0 -1.0 0.0
    outer loop
      vertex 0.0 0.0 0.0
      vertex 1.0 0.0 0.0
      vertex 0.0 0.0 1.0
  endloop
   endfacet

   facet normal 0.0 0.0 -1.0
      outer loop
        vertex 0.0 0.0 0.0
       ….
      endloop
     endfacet
……
 endsolid
```

# Wavefront OBJ file Format

All data are in plain text

- Vertex (v)
- Texture coordinate (vt)
- Vertex normals (vn)
- Faces (f)
  - Indices reference to vertices
  - Vertices are associating with texture coordinate and vertex normal in the form of **( v/vt/vn )**

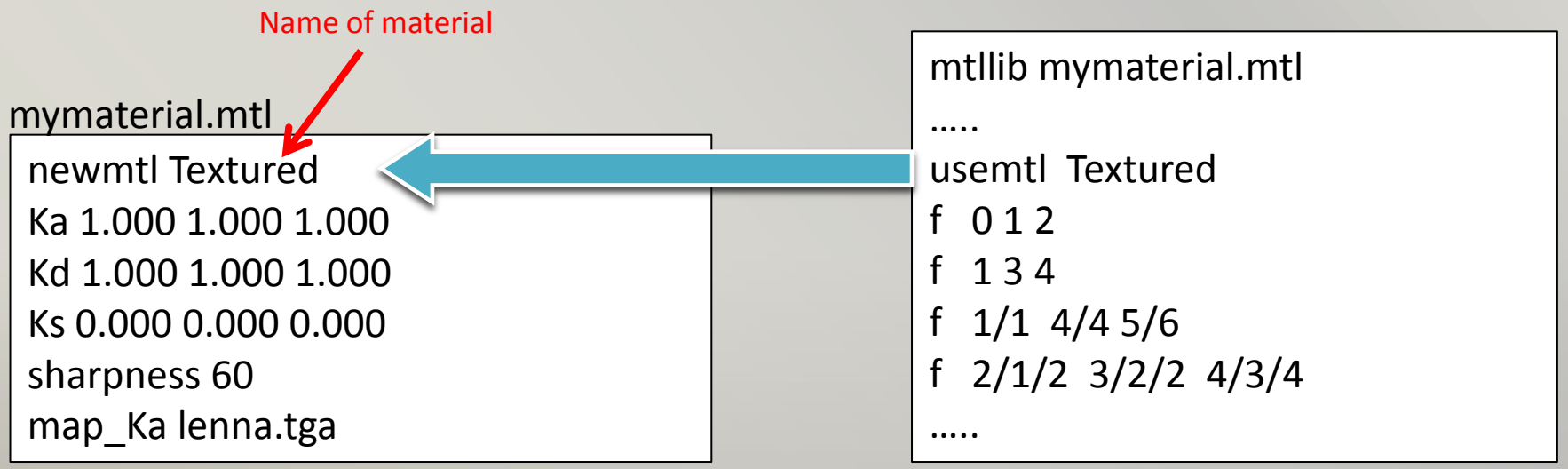
    You can omit the last 2

```
v 0.122  0.534  0.774
v 0.23  0.977  0.408
v .....
.....
.....
vt 0.0 0.0
vt 1.0 0.0
vt .....
.....
vn 0.0 1.0 0.0
vn 0.707 0.707 0.0
.....
.....
f   0 1 2
f   1 3 4
f   1/1  4/4 5/6
f   2/1/2  3/2/2  4/3/4
.....
```

# Wavefront OBJ file Format

- Materials are stored separately
  - in .mtl file (Material Template Library format )
  - Texture map are also defined

    Link to the texture file
- "usemtl" to refer to a defined material name



# Drawing Mesh in OpenGL

- Vertex Array
  - An alternative way to pass vertices to OpenGL
  - Vertices are stored in a large array

```
GLfloat vertices[] = {0.122 , 0.534 ,0.774, 0.23,  0.977,  0.408, ……};
…..

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);…..
```

# Drawing Mesh in OpenGL

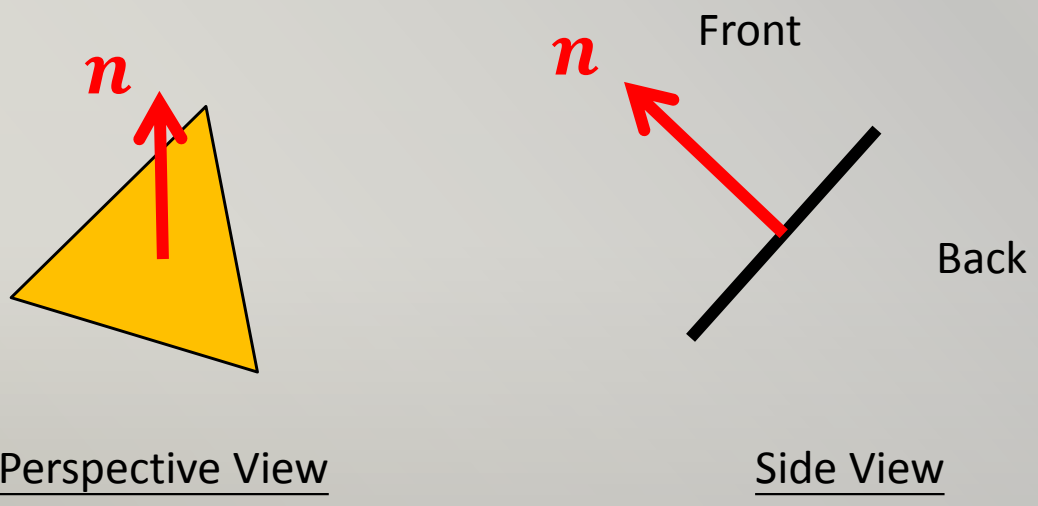
- Similarly, indices of vertex are stored in an array to form the triangles
- Use glDrawElements to tell OpenGL to draw triangles based on the vertex array and given vertex indices

```
GLfloat vertices[] = {0.122 , 0.534 ,0.774, 0.23,  0.977,  0.408, ……};
GLubyte indices[] = {0, 1, 2,  1, 3 ,4 ……}; // 36 indices
…..

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
…..
// draw the array
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);
glDisableClientState(GL_VERTEX_ARRAY);
```

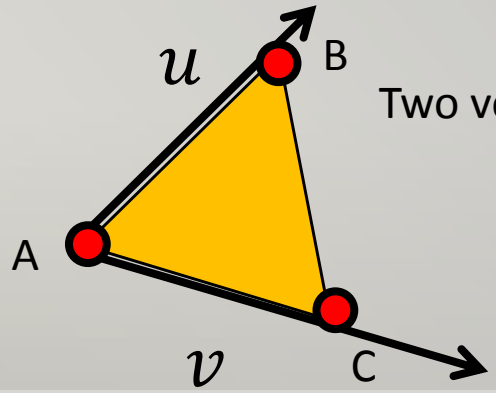
# Normal Computation

- Normal is a commonly used feature to define
  - The orientation of the triangle/surface
  - Front facing direction
- Important in lighting and backface culling process



# Normal Computation

- Face normal can be computed based on the vertices of a triangle
- Choose one vertex, e.g. A
  - Vector $u$ = (B – A)
  - Vector $v$ = (C – A)

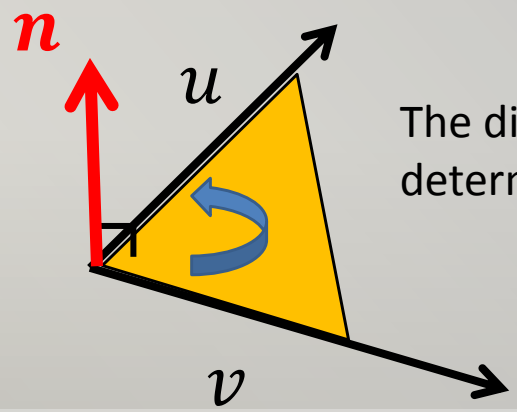



Two vectors can be formed

# Normal Computation

- As discussed in lecture 2, we can use **cross product** on these 2 vectors to compute the normal which is perpendicular to the triangle
  - $n = v \times u$

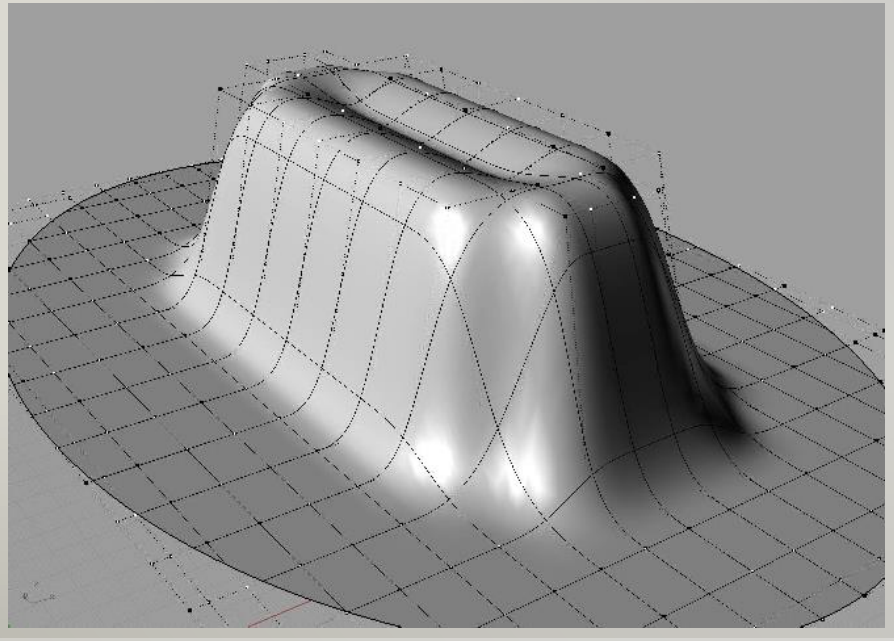
    Further details, please see lecture notes 2




The direction of the vector can be determined by right hand rule

# Parametric Surfaces

- Apart from triangular mesh, many modeling software allow the use of smooth parametric surfaces in 3D objects

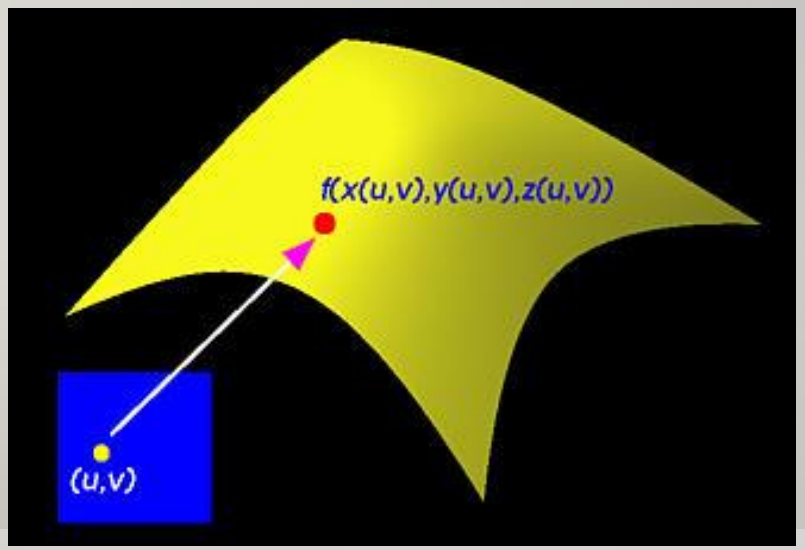


# Parametric Surfaces

- Parametric surfaces are defined by a set of three functions, one for each coordinate

  $x=fx(u,v),\ \ y=fy(u,v),\ \ z=fz(u,v)$

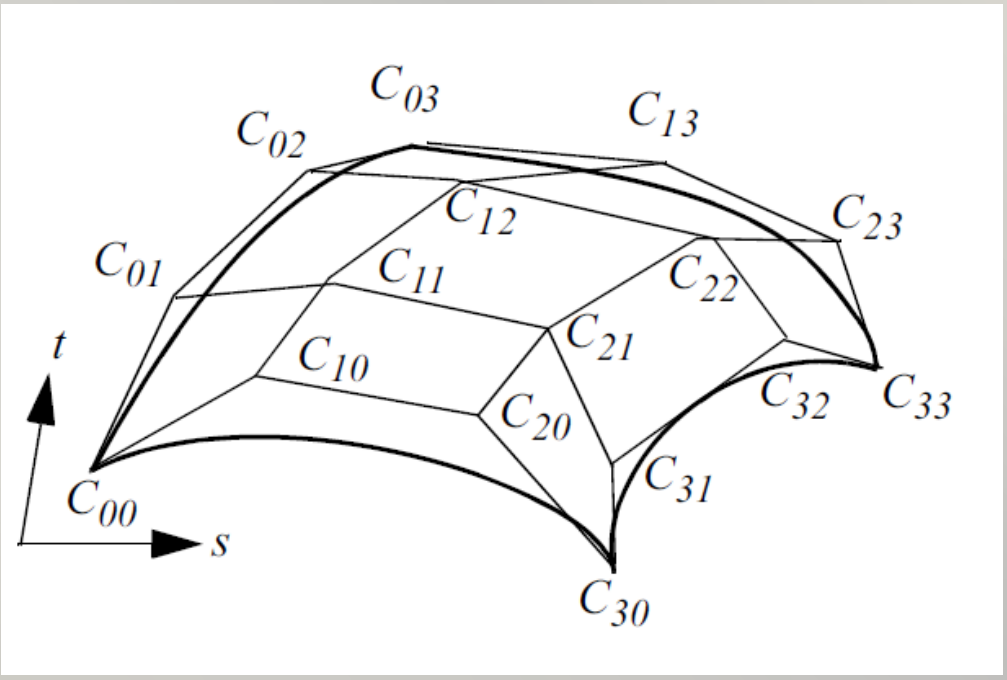
- So a particular surface point can be computed with given a **pair of parameters (u,v)**



# Parametric Surfaces

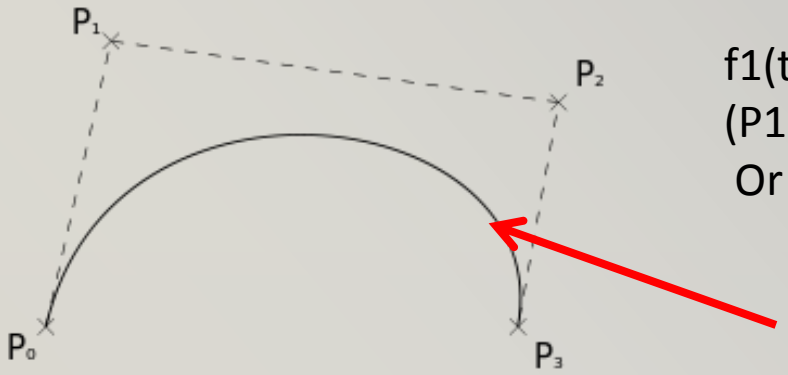
- The functions **f** are commonly defined with **control points**
  - i.e. f (C, u, v)

$$C = \begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix}.$$



# Parametric Curves

- A 2D analogy of parametric 3D surface is the parametric curves
  - Take Bézier Curve as an example
  - Curve point is a function of t  x = f1(t),  y = f2(t);

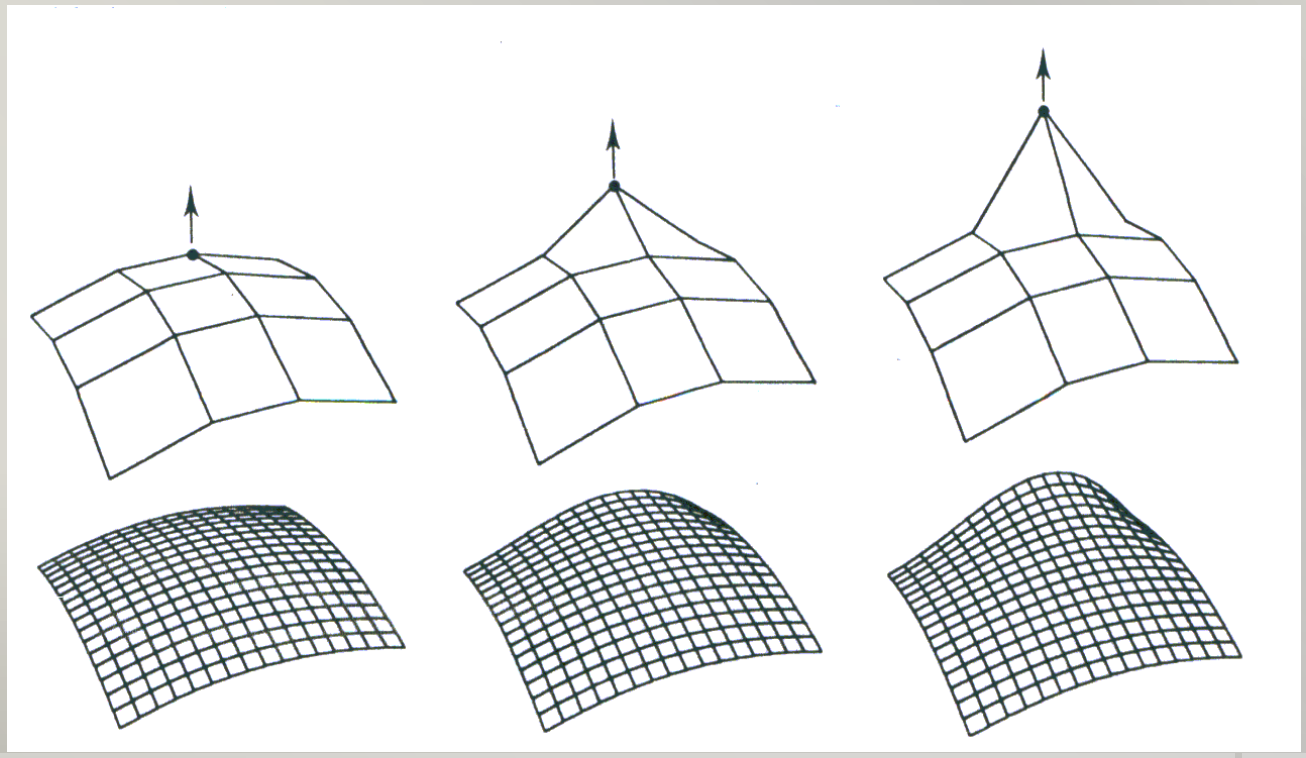



f1(t) and f2(t) are defined by 4 control points (P1- P4 here)
Or 2 point with 2 tangent vectors

- The types and mathematics of parametric curves will be discussed in next lecture

# Modeling of Parametric Surfaces

- We can move the few discrete control points to modify the surface
  - Remember the surface is still a smooth one



# Parametric Surfaces

- Although a parametric surface is defined by the set fo functions **f (C, u, v)**, i.e. involving
  - Control points: C
  - Parameters: u,v
- Many variations are available to define **f (C, u, v)**
- Common examples
  - Non-uniform rational basis spline (NURBS)
  - Bi-cubic Bézier surface
  - Bi-cubic B-spline surface

# Parametric Surfaces

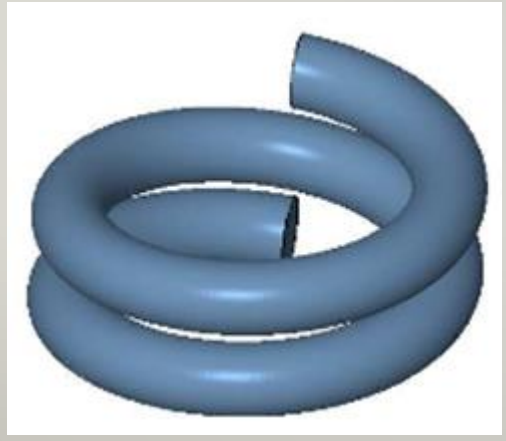
- The degree of freedom for parametric surfaces are limited
- Usually formed by patches

# Parametric Surfaces

- Advantage
  - Smooth surface can be defined with a small number of parameters and control points
  - Modeling of surface can be done by modifying the few control points which is more efficient
- Disadvantage
  - The rendering of parametric surface is less straightforward than a triangle mesh
  - Only smooth surface can be defined within a patch

# Tessellation

- To meet the needs in different 3D systems
- Conversion between parametric surface and mesh is necessary
- The process to convert smooth parametric surface to triangle mesh is called **“Tessellation”**



Parametric surface

Tessellate

Mesh

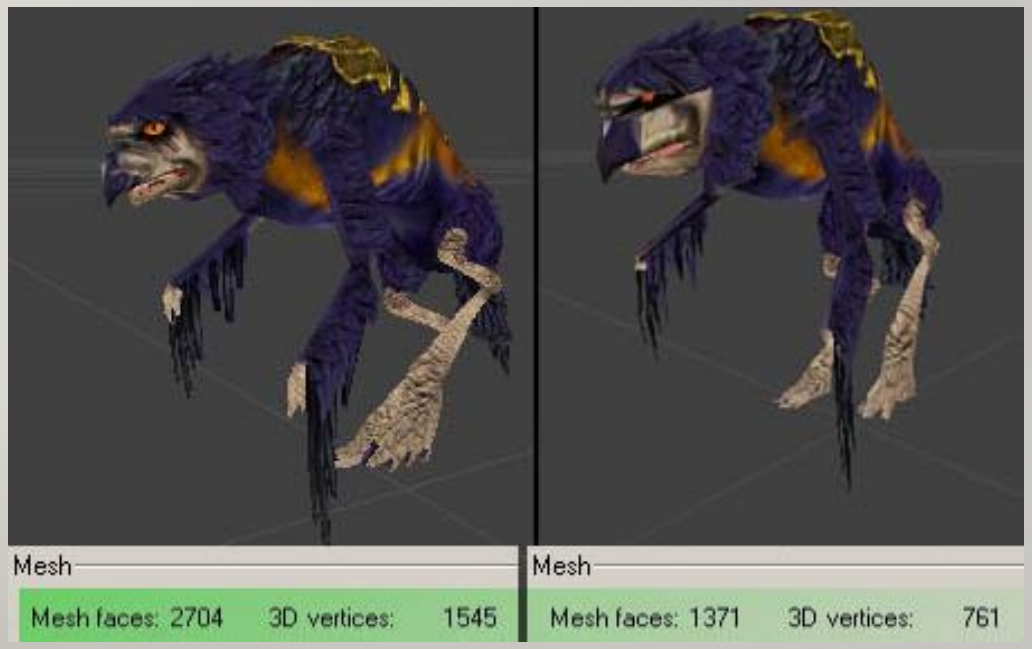
# Tessellation

- Error will be introduced in tessellation
- The more the triangles, the better it approximate the smooth parametric surfaces

http://docs.unity3d.com/

# Decimation of Triangular Mesh

- To meet different requirements like
  - Real-time Rendering
  - High quality animation (e.g. Movie)
- We need triangle mesh in different scales




http://udn.epicgames.com

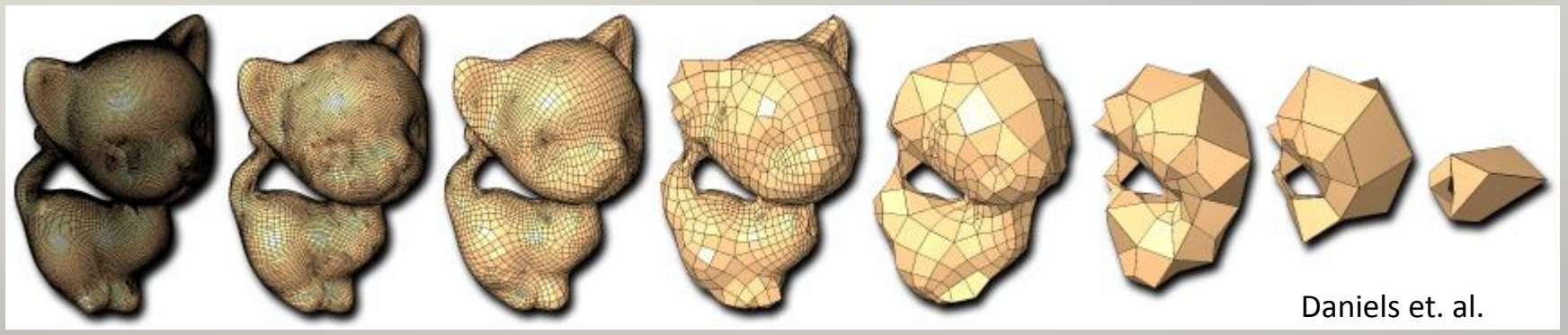
Larger no. of polygons Less no. of polygons

# Decimation of Triangular Mesh

- Decimation is trying the reduce the number of triangles in a mesh used to represent a 3D object
- Useful for object that are over-tessellated

Large no. of triangles/quads

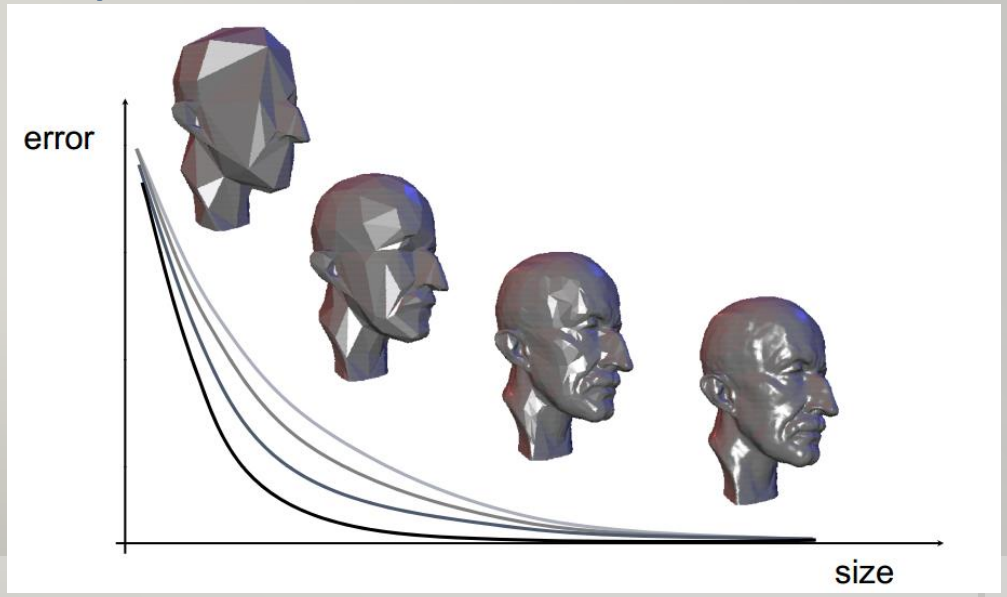
Less no. of triangles/quads



Daniels et. al.

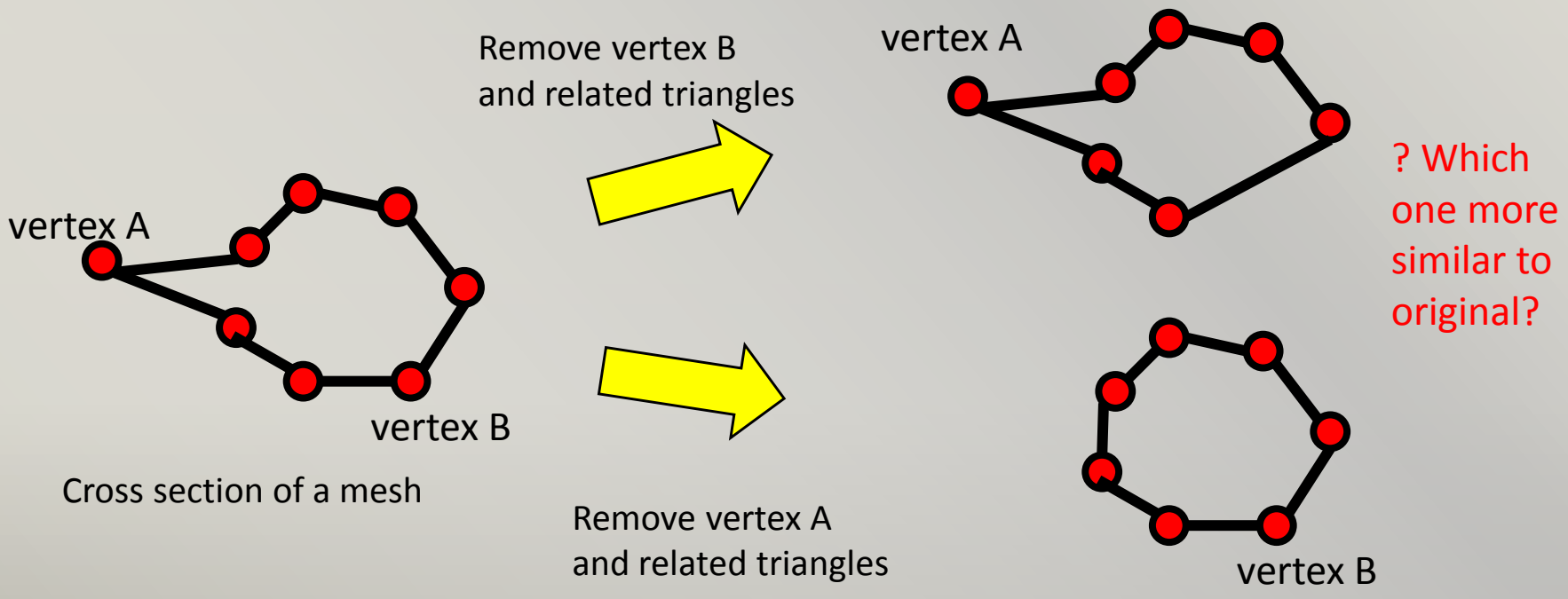
# Decimation of Triangular Mesh

- Always a tradeoff between **Size** and **Quality** of the mesh
  - The lesser the triangles, the poorer the quality
  - The More the triangles, the higher the storage and rendering requirement



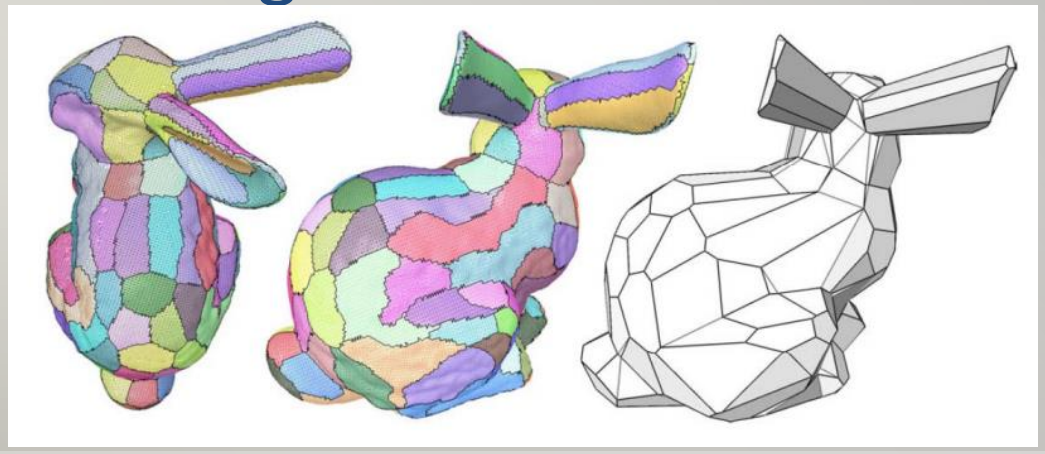
# Decimation of Triangular Mesh

- A basic requirement of decimation algorithm is not to remove **important features** that early
  - features of a mesh may include sharp corners and edges



# Decimation of Triangular Mesh

- Many approaches are being proposed, one is called **Vertex Clustering**
- Group similar triangles into a cluster
  - Similarity is measured in surface normal
- So, triangles in same cluster can be represented with fewer triangles

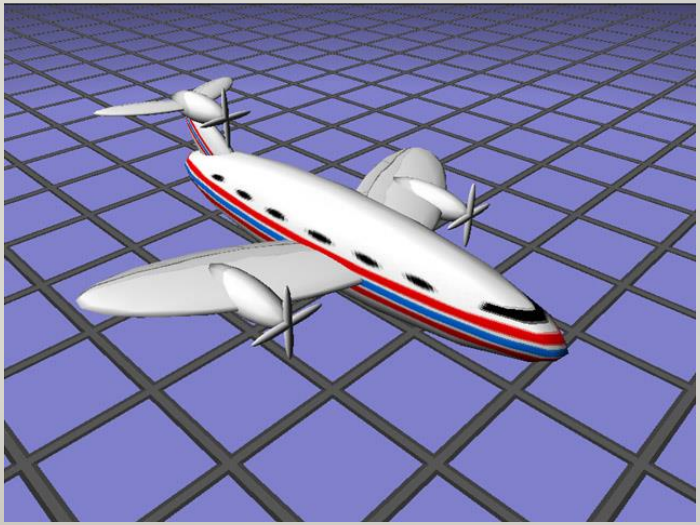


# Scene Graph

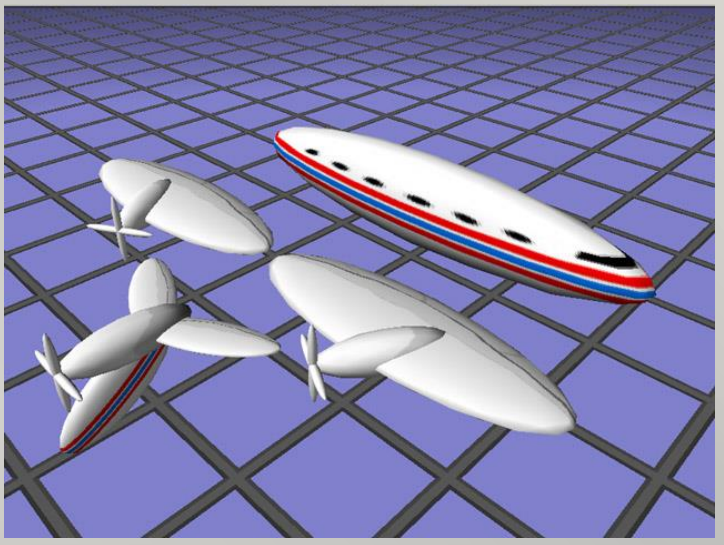
- Widely used in higher level graphics engine
  - E.g. game engine
- A data structure to organize objects/shapes in the scene in a tree
  - May also include some other entities like sound, lighting and etc.
- Root node is the scene itself
- Any object/shape can be inserted into the tree

# Scene Graph : An example

- Air plane
  - Wings
  - Cabin

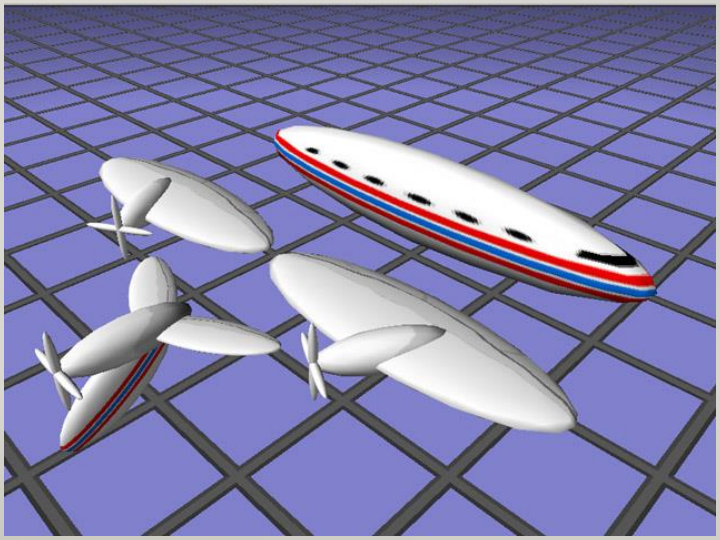


Composed of

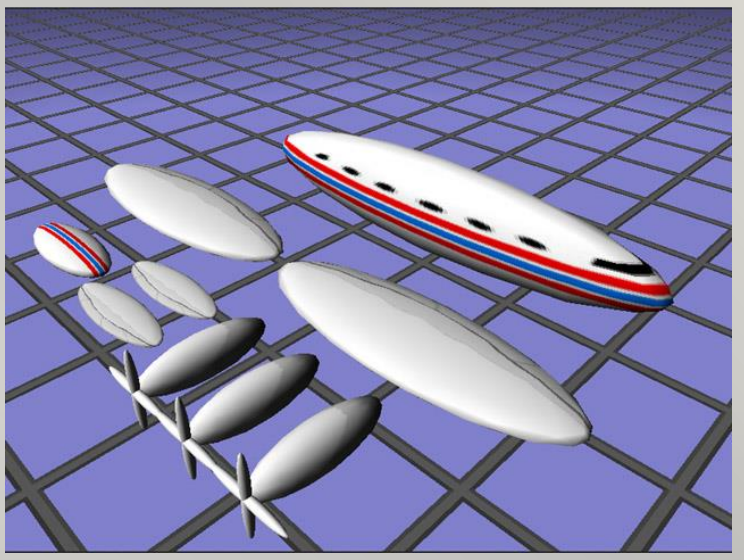


# Scene Graph : An example

- Wings
  - Engines
  - Smaller wings

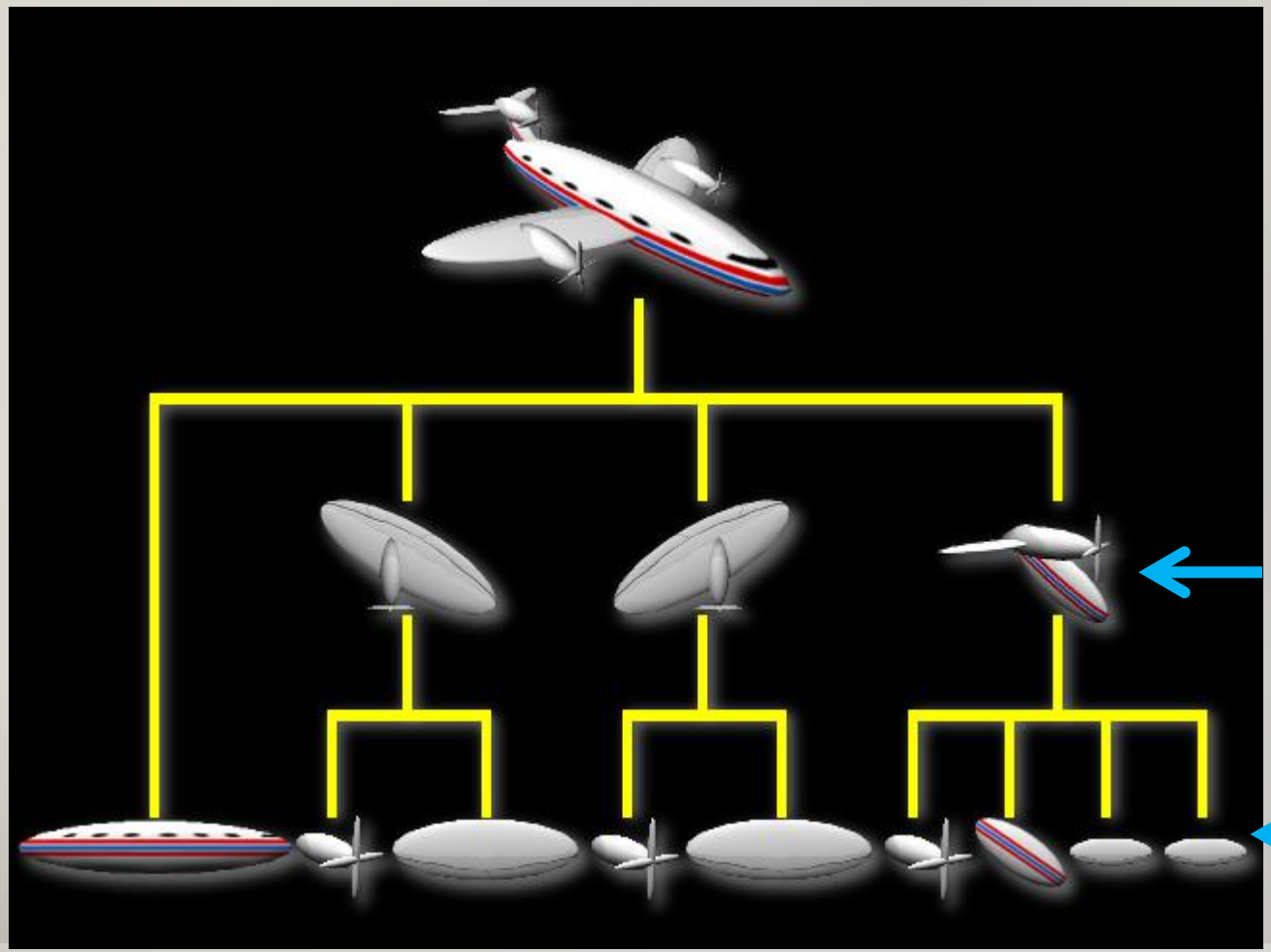


Composed of



# Scene Graph

- So, to represent their relationship in a graph:



# Scene Graph

- Node: The basic element in a scene graph
  - Leaf nodes: nodes with no children

    In case of Java3D, it can be:

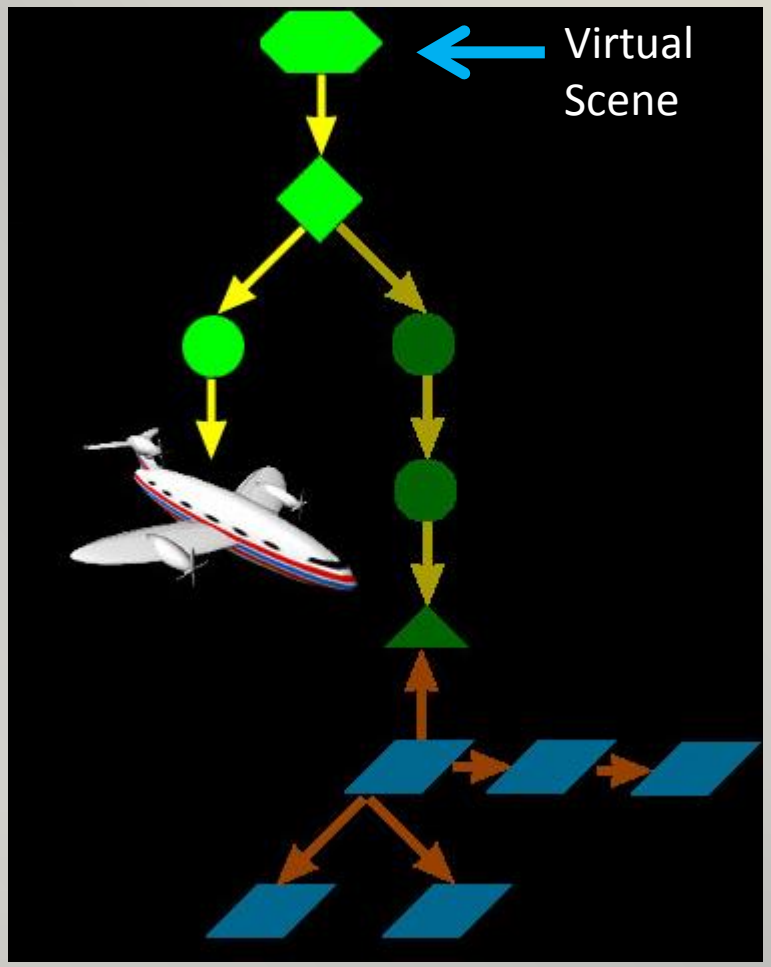
    Shapes, lights, sounds, etc.

    Animation behaviors
  - Group nodes: nodes with children

    Transformation, switches, etc.

# Scene Graph

- The **root node**:
  - commonly representing the virtual scene
- So, to include our object
- We add our object as a child of the root node



# Summary

- Geometric surfaces are mainly represented in
  - Triangle Mesh
  - Parametric Surfaces
- Indexed triangle mesh forms triangles without repeating vertices' coordinates
- Tessellation and Decimation help to produce necessary triangle mesh
- Scene Graph as a data structure to organize shapes in a scene in many high level graphics engines