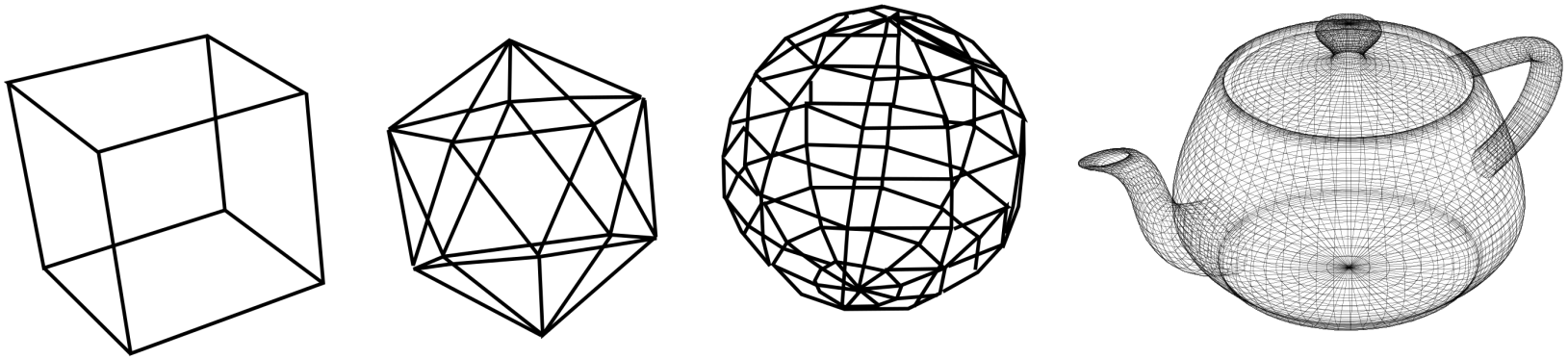
VIETNAM ACADEMY OF SCIENCE AND TECHNOLOGY
UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

COMPUTER GRAPHICS

Lecture 5: Rasterization, Fragment Processing, and Output Merging

Lecturer: Dr. NGUYEN Hoang Ha

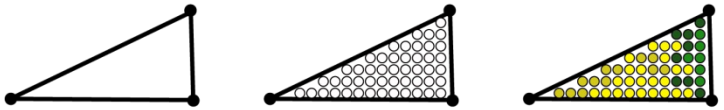


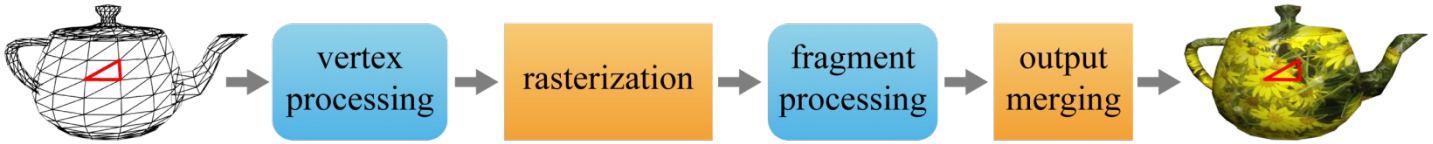
Reference:JungHyun Han. 2011. 3D Graphics for Game Programming (1st ed.), Chapman & Hall/CRC.Chapter 3,4

RASTERIZATION

To build fragment data from given vertices

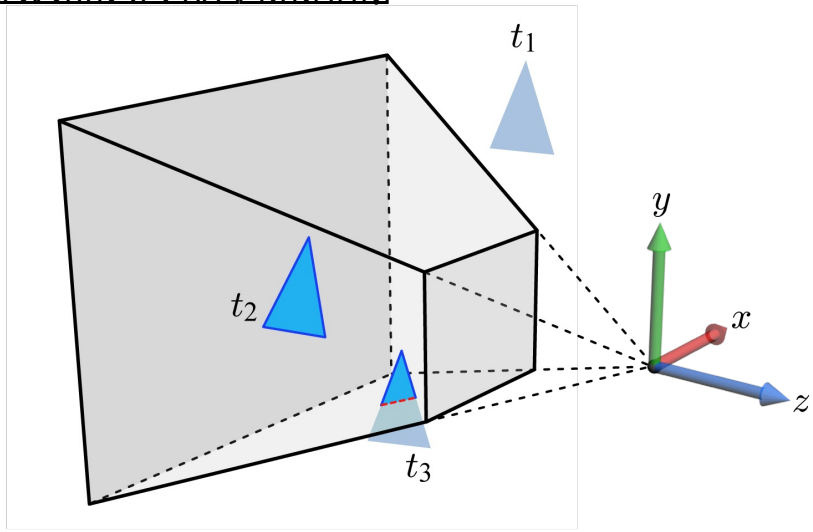
- The vertices processed by the vertex program enter a hard-wired stage and are assembled to build *primitives* (polygons, lines). Primitives are further processed to determine its 2D form on the screen, and are rasterized into a set of fragments.





- The hard-wired rasterization stage performs the following:
  - C
  -
  - B
  -
  -

- Clipping is performed in the clip space, but the following figure presents its concept in the camera space, for the sake of intuitive understanding.
  - [illegible]
  - [illegible]
  - [illegible]



- As a result of clipping, vertices may be added to and deleted from the triangle.

- Unlike affine transforms, the last row of M_{proj} is not (0 0 0 1) but (0 0 -1 0). When M_{proj} is applied to (x,y,z,1), the w-coordinate of the transformed vertex is $-z$.

$$M_{proj} = \begin{pmatrix} \frac{cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} m_{11} & 0 & 0 & 0 \\ 0 & m_{22} & 0 & 0 \\ 0 & 0 & m_{33} & m_{34} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}x \\ m_{22}y \\ m_{33}z + m_{34} \\ -z \end{pmatrix} \rightarrow \begin{pmatrix} -\frac{m_{11}x}{z} \\ -\frac{m_{22}y}{z} \\ -m_{33} - \frac{m_{34}}{z} \\ 1 \end{pmatrix}$$

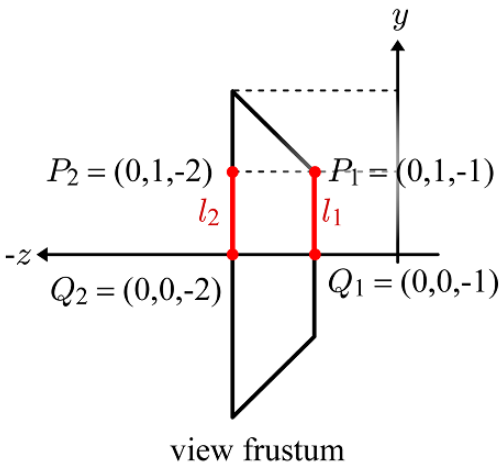
- In order to convert from the homogeneous (clip) space to the Cartesian space, each vertex should be divided by its w-coordinate (which equals $-z$).

- Note that $-z$ is a positive value representing the *distance* from the *xy*-plane of the camera space. Division by $-z$ makes distant objects smaller. It is *perspective division*.

$$M_{proj} = \begin{pmatrix} \frac{\cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

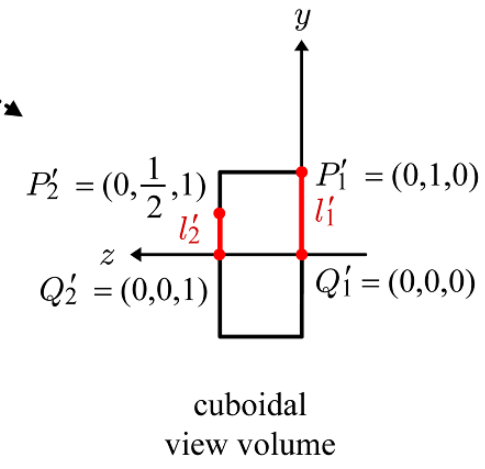
$$fovy = \frac{\pi}{2}$$
$$aspect = 1$$
$$n = 1$$
$$f = 2$$

$$M_{proj} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



view frustum

projection transform M_{proj}



cuboidal view volume

$$M_{proj}P_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = P_1'$$

$$M_{proj}P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 2 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ \frac{1}{2} \\ 1 \\ 1 \end{pmatrix} = P_2'$$

$$M_{proj}Q_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = Q_1'$$

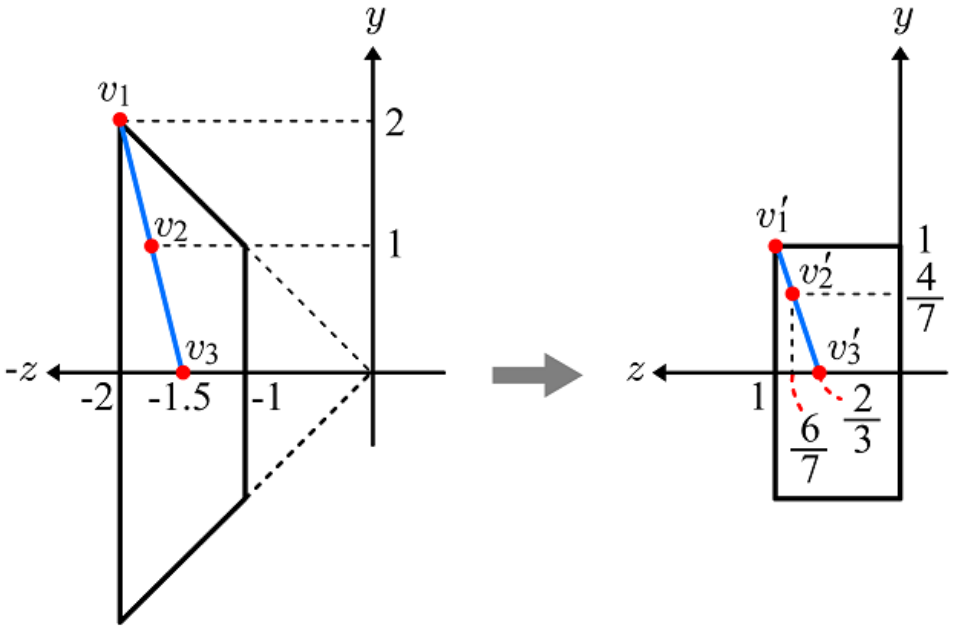
$$M_{proj}Q_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 2 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = Q_2'$$

$$fovy = \frac{\pi}{2}, \quad aspect = 1, \quad n = 1, \quad f = 2 \rightarrow M_{proj} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$M_{proj}\, v_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 2 \\ 2 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = v_1'$$

$$M_{proj}\, v_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ -\frac{7}{4} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \frac{3}{2} \\ \frac{7}{4} \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ \frac{4}{7} \\ \frac{6}{7} \\ 1 \end{pmatrix} = v_2'$$

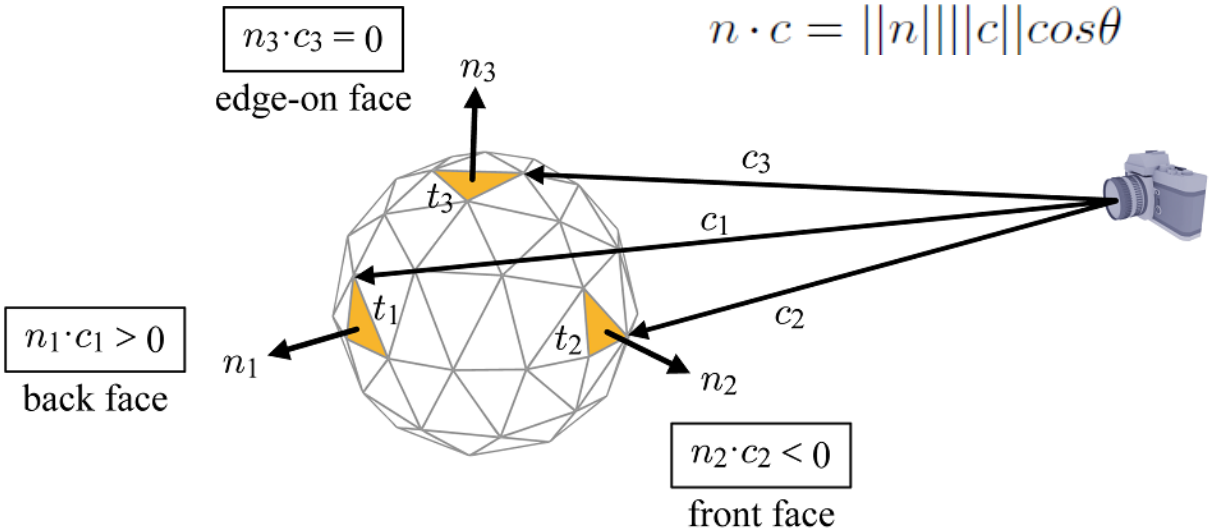
$$M_{proj}\, v_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -\frac{3}{2} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ \frac{3}{2} \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ \frac{2}{3} \\ 1 \end{pmatrix} = v_3'$$



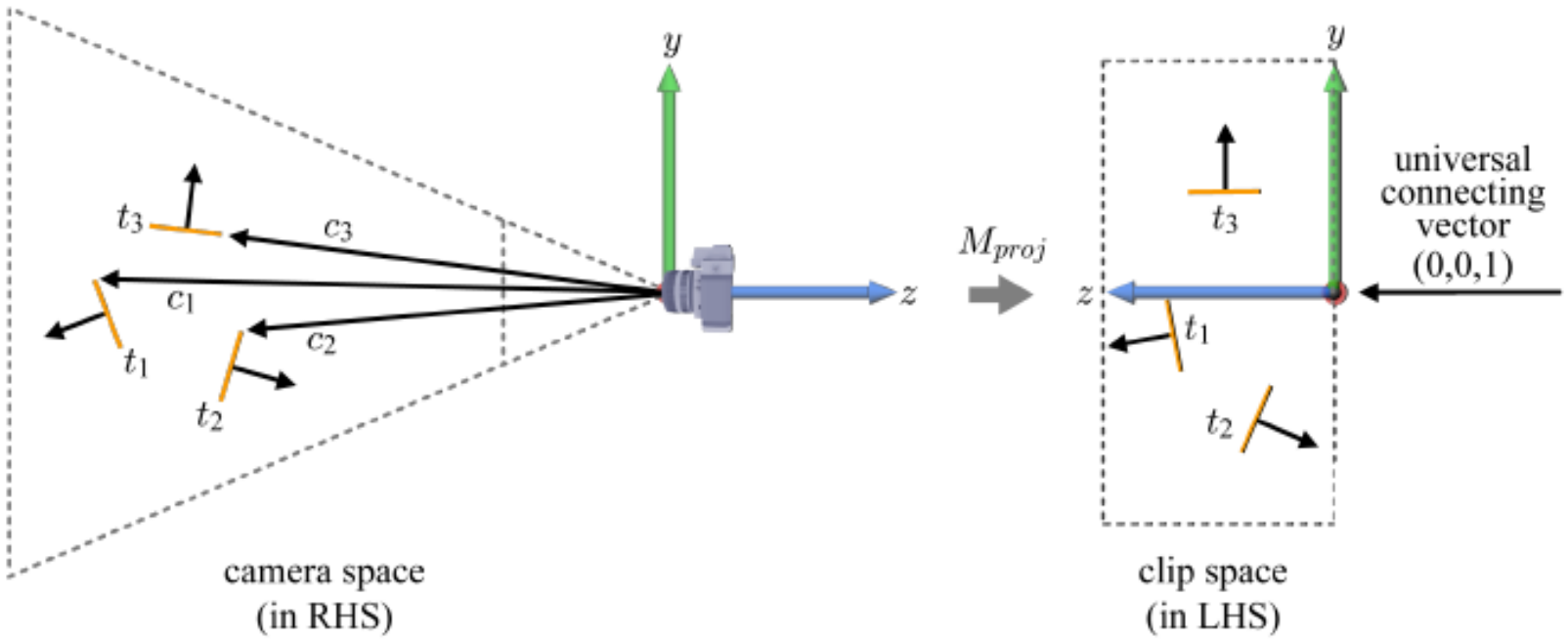
B□□□-□□□□C□□□□□□

- The polygons facing away from the viewpoint of the camera are discarded. Such polygons are called *back-faces*. (The polygons facing the camera are called *front-faces*.)
- In the camera space, the normal of a triangle can be used to determine whether the triangle is a front-face or a back-face. A triangle is taken as a back-face if the camera (**EYE**) is in the opposite side of the triangle's normal. Otherwise, it is a front-face.
- For the purpose, compute the *dot product* of the triangle normal *n* and the vector *c* connecting the camera position and a vertex of the triangle.

$$n \cdot c = ||n||||c||cos\theta$$



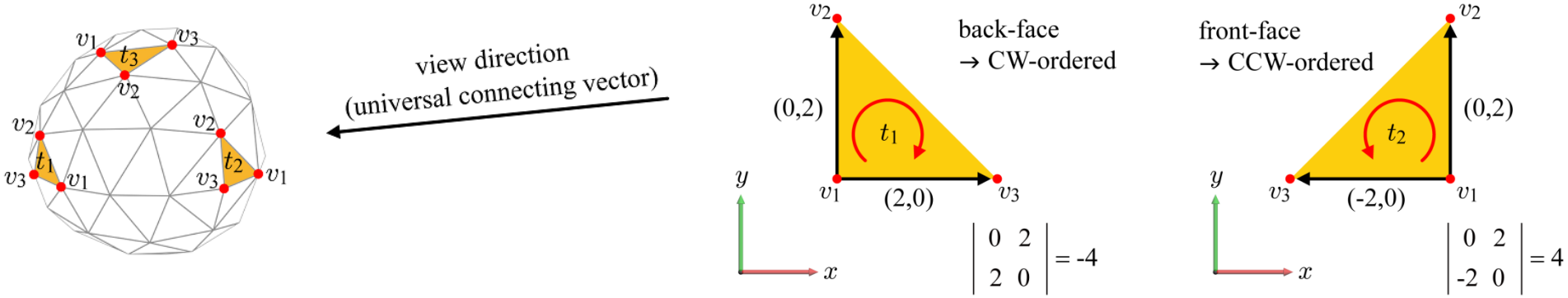
B□□□-□□□□C □□□□□□(□□□□□)



- Back-face culling in the camera space is expensive.
- Fortunately, The projection transform makes all the connecting vectors parallel to the *z*-axis. The universal connecting vector represents the parallelized projection lines. Then, by viewing the triangles along the *universal* connecting vector, we can distinguish the back-faces from the front-faces.

B□□□-□□□□C □□□□□□□(□□□□□□)

- Viewing a triangle along the universal connecting vector is equivalent to orthographically projecting the triangle onto the *xy*-plane.
- A 2D triangle with CW-ordered vertices is a back-face, and a 2D triangle with CCW-ordered vertices is a front-face.



- Compute the following determinant, where the first row represents the 2D vector connecting v_1 and v_2, and the second row represents the 2D vector connecting v_1 and v_3. If it is positive, CCW. If negative, CW. If 0, edge-on face.

$$\begin{vmatrix} (x_2 - x_1) & (y_2 - y_1) \\ (x_3 - x_1) & (y_3 - y_1) \end{vmatrix}$$

- Note that, if the vertices are ordered CW in the clip space, the reverse holds, i.e., the front-face has CW-ordered vertices in 2D.

- OpenGL and Direct3D allow us to control the face culling mechanism based on vertex ordering.

[Note: Face culling in OpenGL]

In OpenGL, face culling is disabled by default, and both front- and back-faces are rendered. When it is enabled, `glCullFace()` specifies whether front- or back-facing polygons are culled. It accepts the following symbolic constants:

- `GL_FRONT`
- `GL_BACK`
- `GL_FRONT_AND_BACK`

The default value is `GL_BACK`, and back-faces are culled.

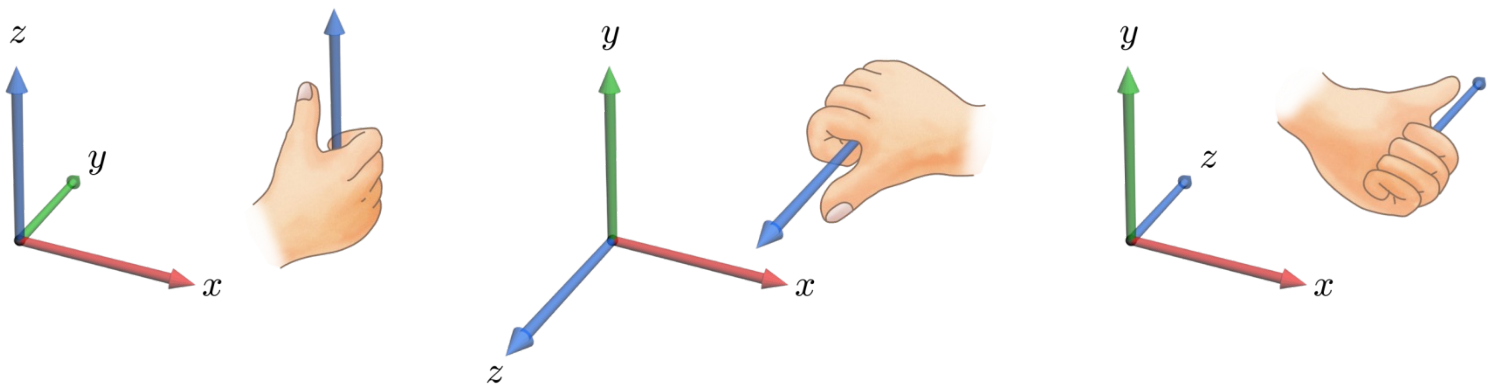
Then, `glFrontFace()` specifies the vertex order of front-facing polygons. It accepts the following:

- `GL_CW`
- `GL_CCW`

The default value is `GL_CCW`.

C[illegible]H[illegible].[illegible]H[illegible]

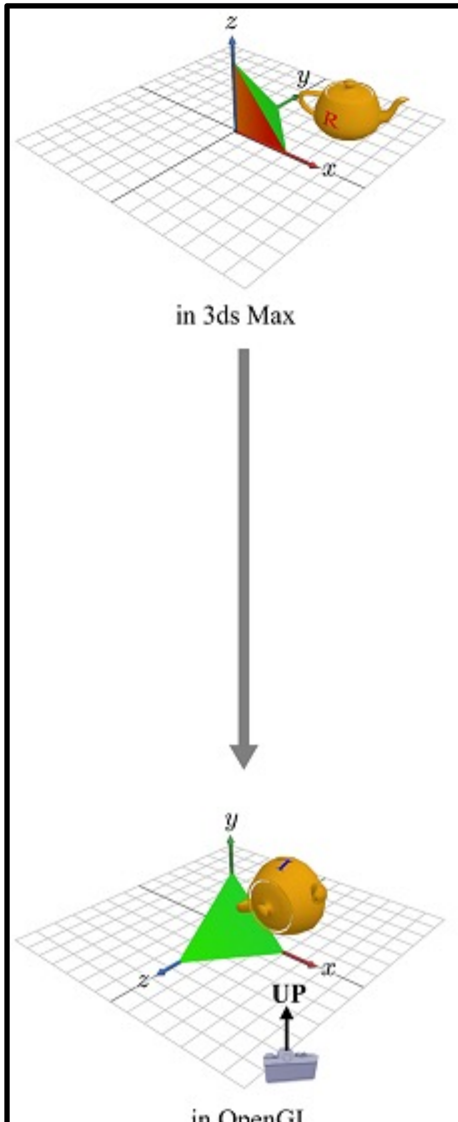
- RHS vs. LHS



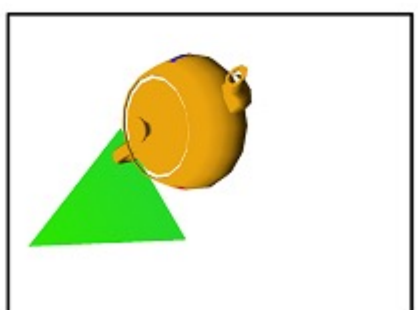
RHS in 3ds Max　　RHS in OpenGL　　LHS in Direct3D

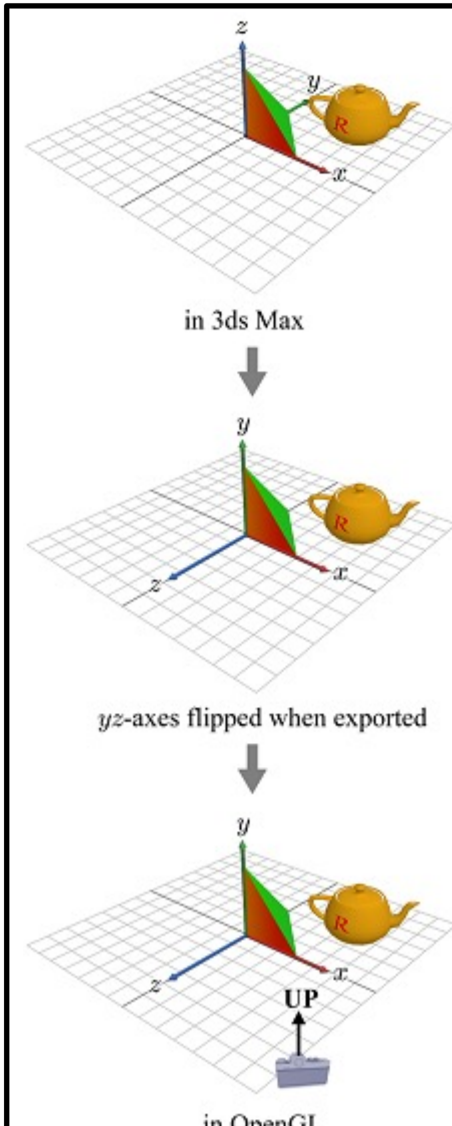
- Notice the difference between 3ds Max and OpenGL: The vertical axis is the *z*-axis in 3ds Max, but is the *y*-axis in OpenGL.

C[illegible] 3[illegible] [illegible]G[illegible]

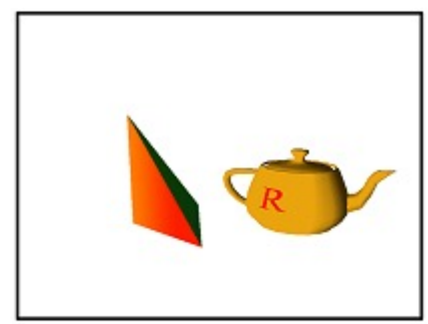


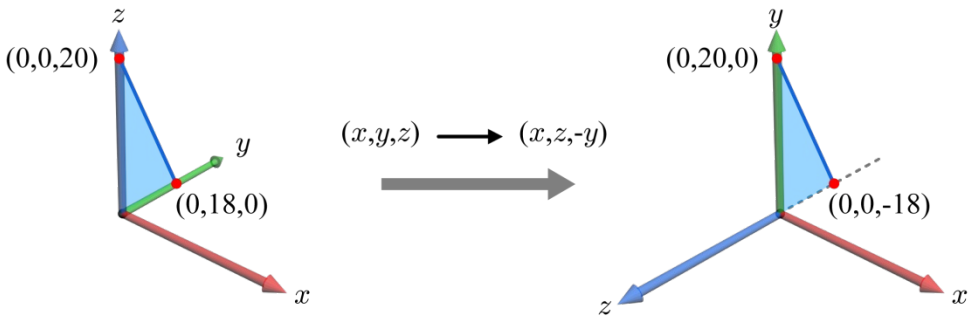
If the scene is exported *as is* to OpenGL, the objects appear flipped.



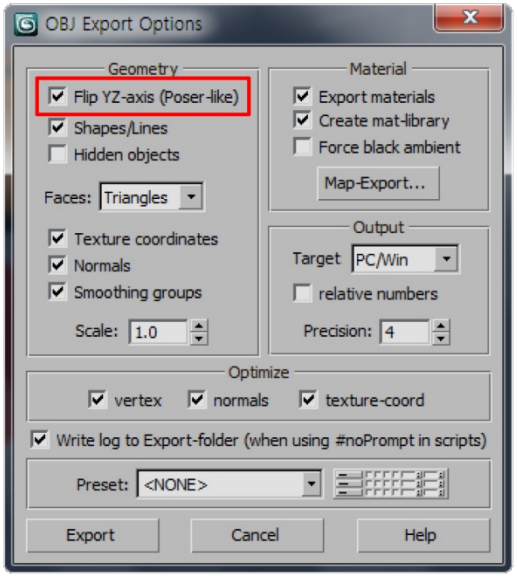


At the time of export, flip the *yz*-axes while making the objects immovable.




z
(0,0,20)
y
(0,18,0)
x
(x,y,z) → (x,z,-y)
y
(0,20,0)
(0,0,-18)
z
x

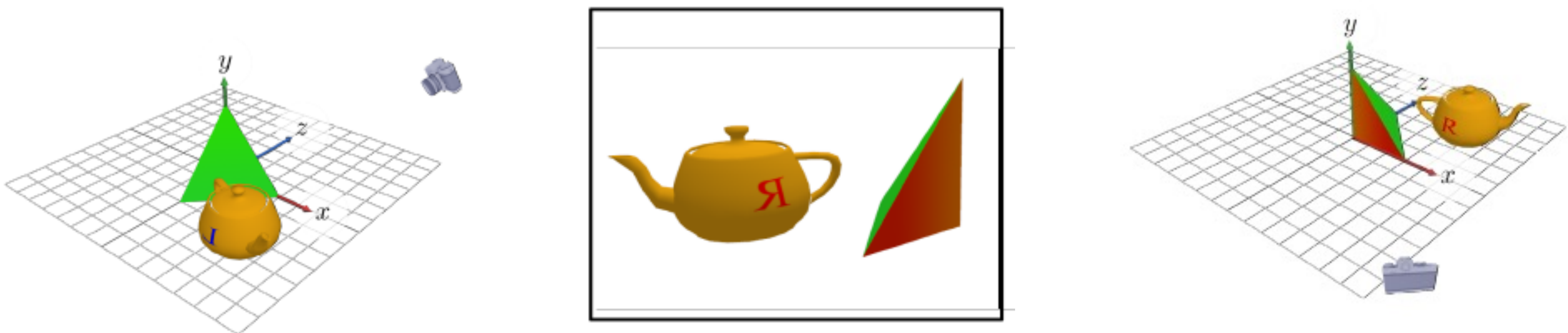
(a)


OBJ Export Options
Geometry
Flip YZ-axis (Poser-like)
Shapes/Lines
Hidden objects
Faces: Triangles
Texture coordinates
Normals
Smoothing groups
Scale: 1.0
Material
Export materials
Create mat-library
Force black ambient
Map-Export...
Output
Target PC/Win
relative numbers
Precision: 4
Optimize
vertex
normals
texture-coord
Write log to Export-folder (when using #noPrompt in scripts)
Preset: <NONE>
Export
Cancel
Help

(b)

C[illegible] 3[illegible]D [illegible]3D

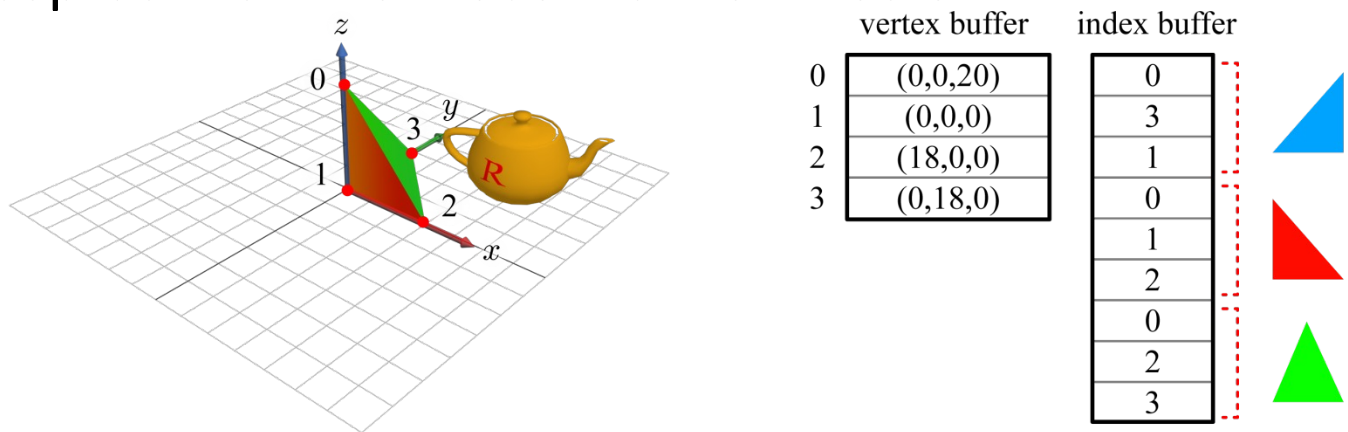
- Assume that the *yz*-axes have been flipped. Then, ‘3ds Max to Direct3D’ problem is reduced to ‘OpenGL to Direct3D.’
- Placing an RHS-based model into an LHS (or vice versa) has the effect of making the model reflected by the *xy*-plane mirror, as shown in (a).
- The problem can be easily resolved if we enforce one more reflection, as shown in (b). Reflecting the reflected returns to the original!
- Reflection with respect to the *xy*-plane is equivalent to negating the *z*-coordinates.



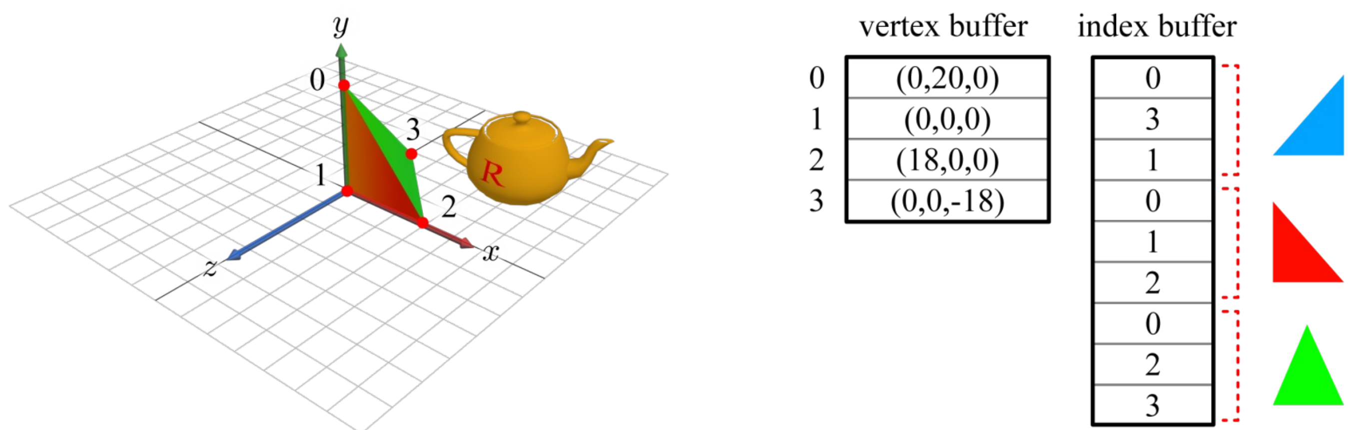
(a) (b)

C[illegible] [illegible]3[illegible]D [illegible]3D ([illegible])

- Conceptual flow from 3ds Max to Direct3D



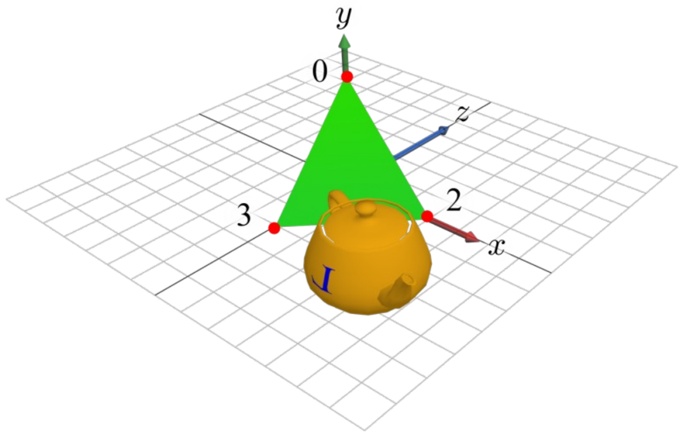
(a) Original (in RHS)



(b) Flipped yz-axes (in RHS)

C[illegible] 3[illegible]D [illegible]3D ([illegible])

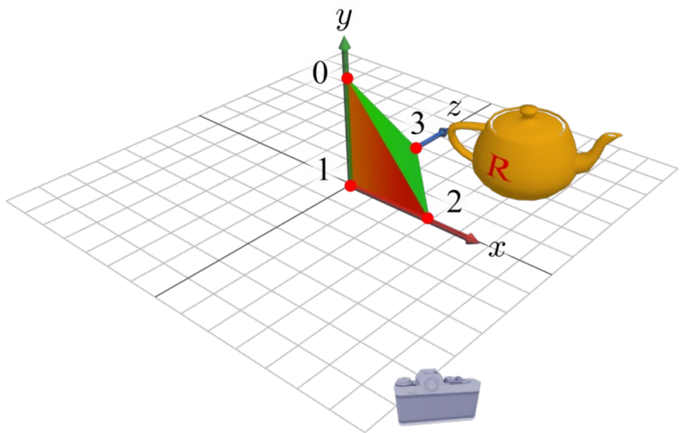
- Conceptual flow from 3ds Max to Direct3D



	vertex buffer
0	(0,20,0)
1	(0,0,0)
2	(18,0,0)
3	(0,0,-18)

index buffer
0
3
1
0
1
2
0
2
3

(c) Imported (in LHS)

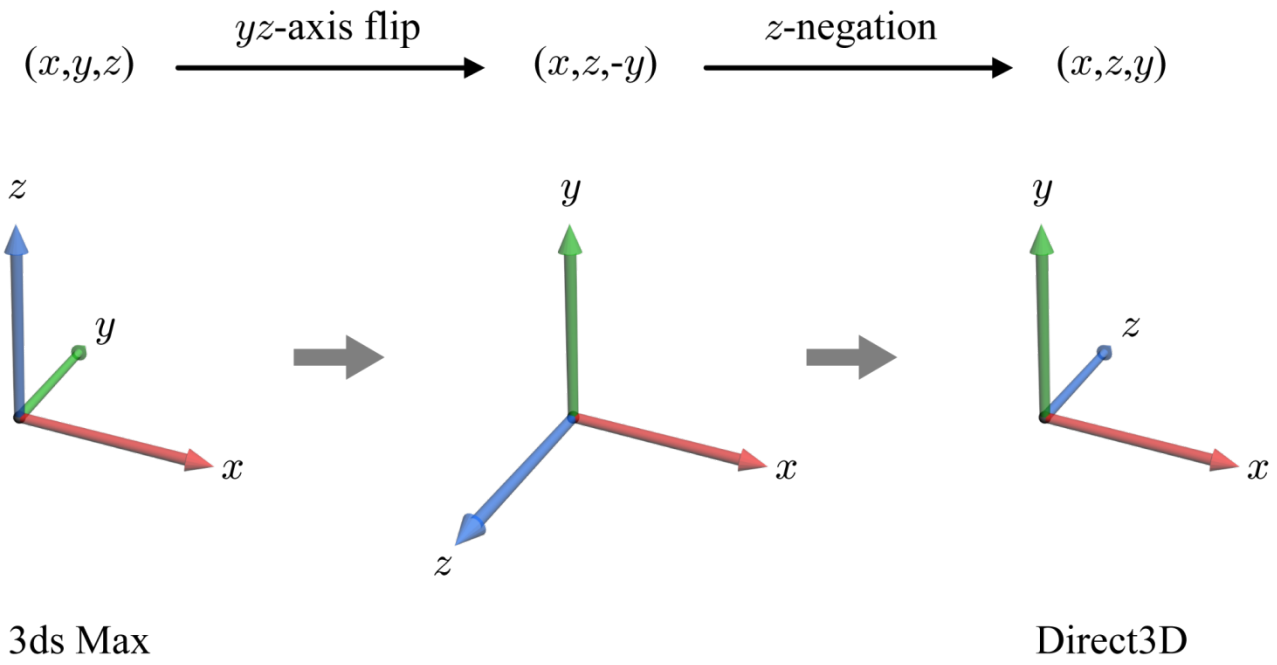


	vertex buffer
0	(0,20,0)
1	(0,0,0)
2	(18,0,0)
3	(0,0,18)

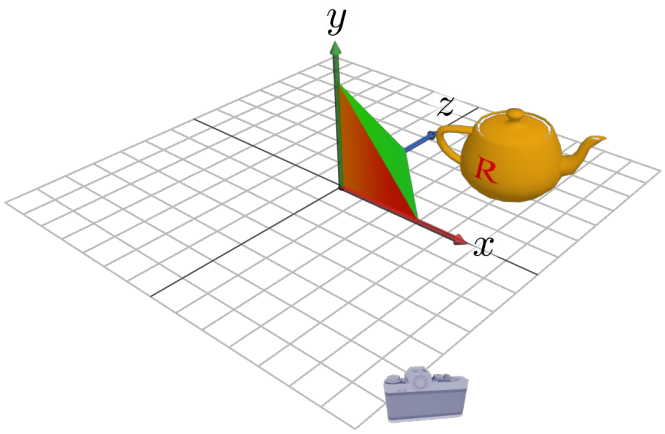
index buffer
0
3
1
0
1
2
0
2
3

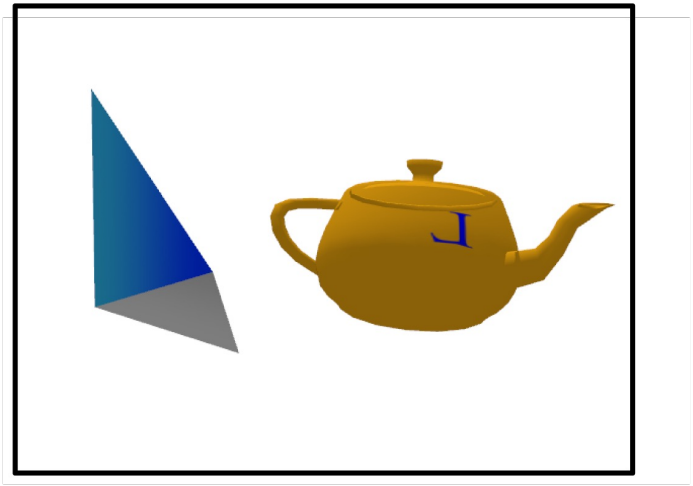
(d) Reflected (in LHS)

- Conversion from 3ds Max to Direct3D requires *yz*-axis flip followed by *z*-negation. The combination is simply *yz*-swap.



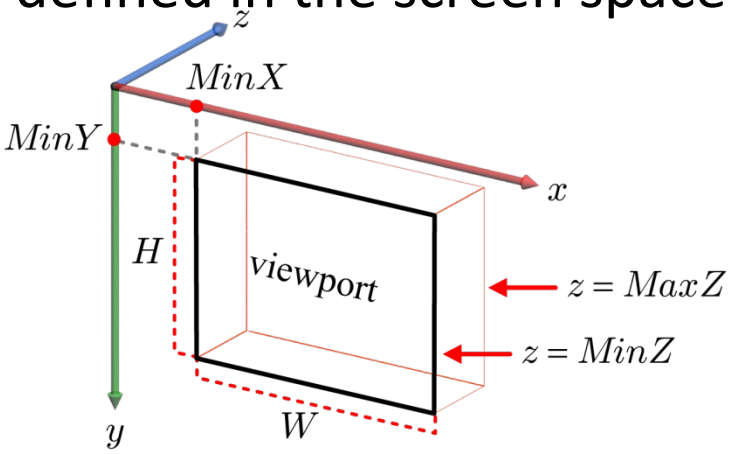
- When only *yz*-swap is done, we have the following image. Instead of back-faces, front-faces are culled.





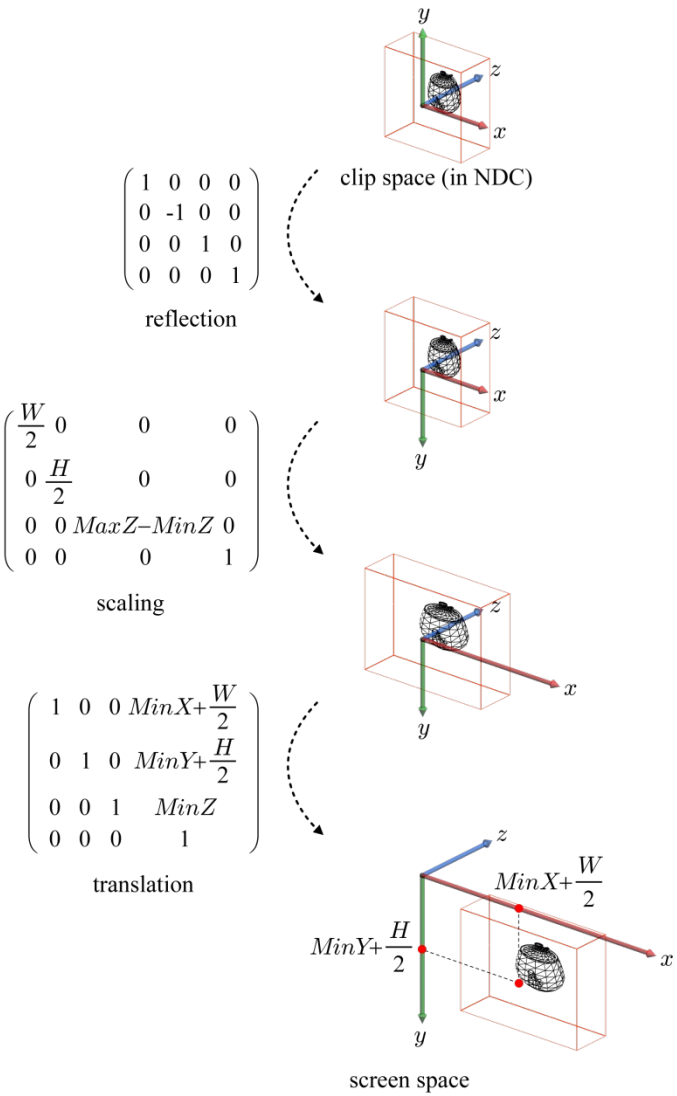
- The *yz*-swap does not change the CCW order of vertices, and therefore the front-faces have CCW-ordered vertices in 2D. In Direct3D, the vertices of a back-face are assumed to appear CCW-ordered in 2D, and the default is to cull the faces with the CCW-ordered vertices.
- The solution is to change the Direct3D culling mode such that the faces with CW-ordered vertices are culled.

- A window at the computer screen is associated with its own *screen space*. It is a 3D space and right-handed.
- A *viewport* is defined in the screen space.




```
typedef struct _D3DVIEWPORT9 {
DWORD X;
DWORD Y;
DWORD Width;
DWORD Height;
float MinZ;
float MaxZ;
} D3DVIEWPORT9;
```

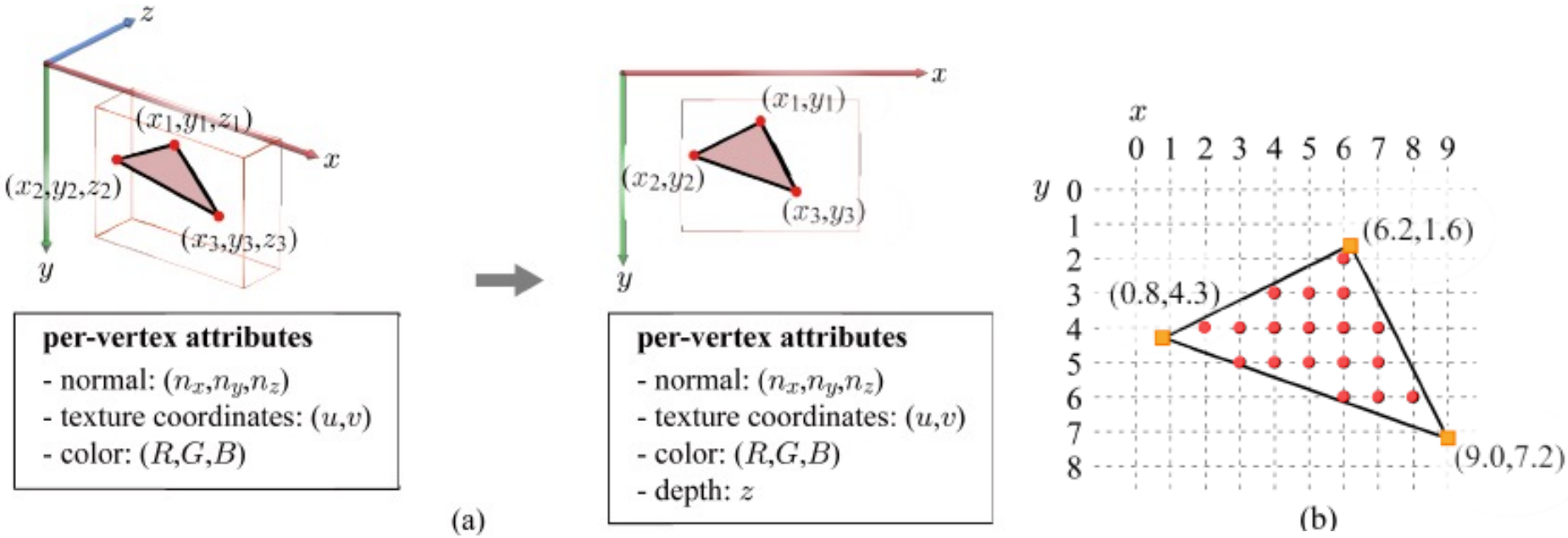
```
typedef struct D3D10_VIEWPORT {
INT TopLeftX;
INT TopLeftY;
UINT Width;
UINT Height;
FLOAT MinDepth;
FLOAT MaxDepth;
} D3D10_VIEWPORT;
```



$$\begin{pmatrix} \frac{W}{2} & 0 & 0 & MinX + \frac{W}{2} \\ 0 & -\frac{H}{2} & 0 & MinY + \frac{H}{2} \\ 0 & 0 & MaxZ - MinZ & MinZ \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In most applications, *MinZ* and *MaxZ* are set to 0.0 and 1.0, respectively, and both of *MinX* and *MinY* are zero.

$$\begin{pmatrix} \frac{W}{2} & 0 & 0 & \frac{W}{2} \\ 0 & -\frac{H}{2} & 0 & \frac{H}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



- Defines the screen-space pixel locations covered by the primitive and
- Interpolates the per-vertex attributes to determine the per-fragment attributes

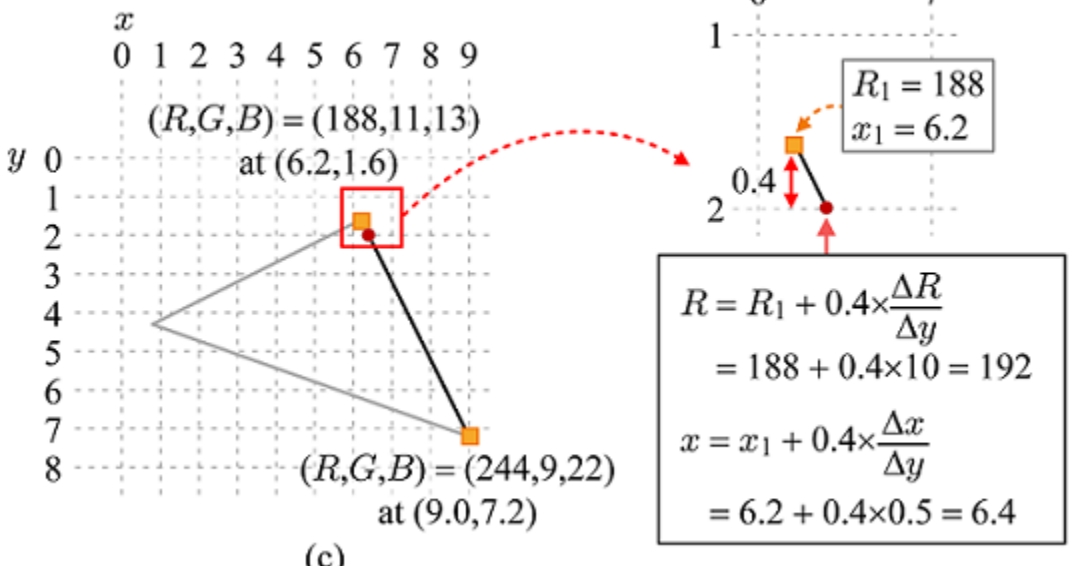
$\Delta y = 7.2 - 1.6 = 5.6$

$\Delta R = 244 - 188 = 56$

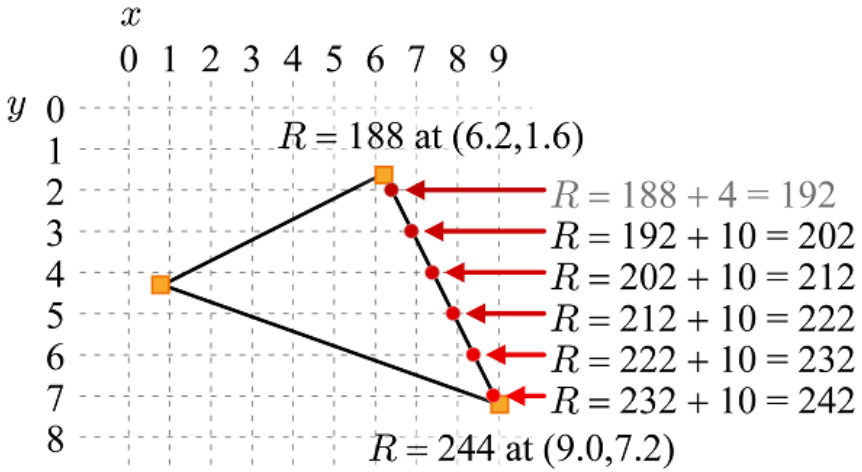
$\frac{\Delta R}{\Delta y} = \frac{56}{5.6} = 10$

$\Delta x = 9.0 - 6.2 = 2.8$

$\frac{\Delta x}{\Delta y} = \frac{2.8}{5.6} = 0.5$

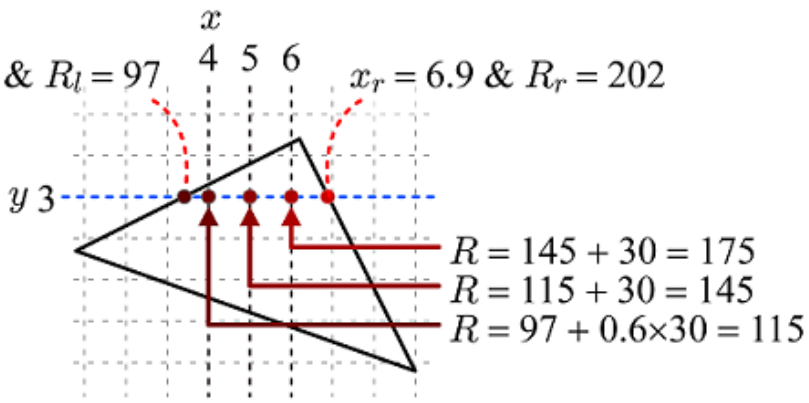


(c)

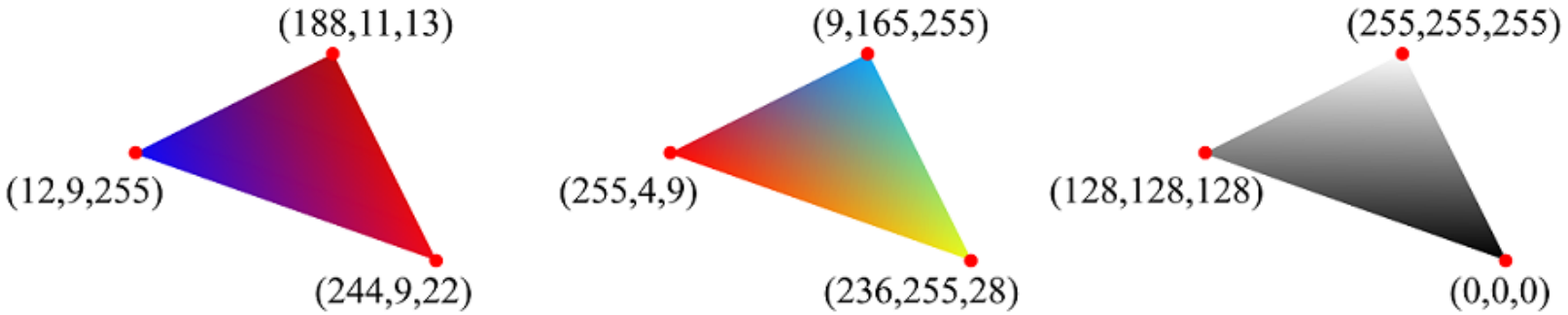


(d)

$$\Delta x = 6.9 - 3.4 = 3.5$$
$$\Delta R = 202 - 97 = 105$$
$$\frac{\Delta R}{\Delta x} = \frac{105}{3.5} = 30$$

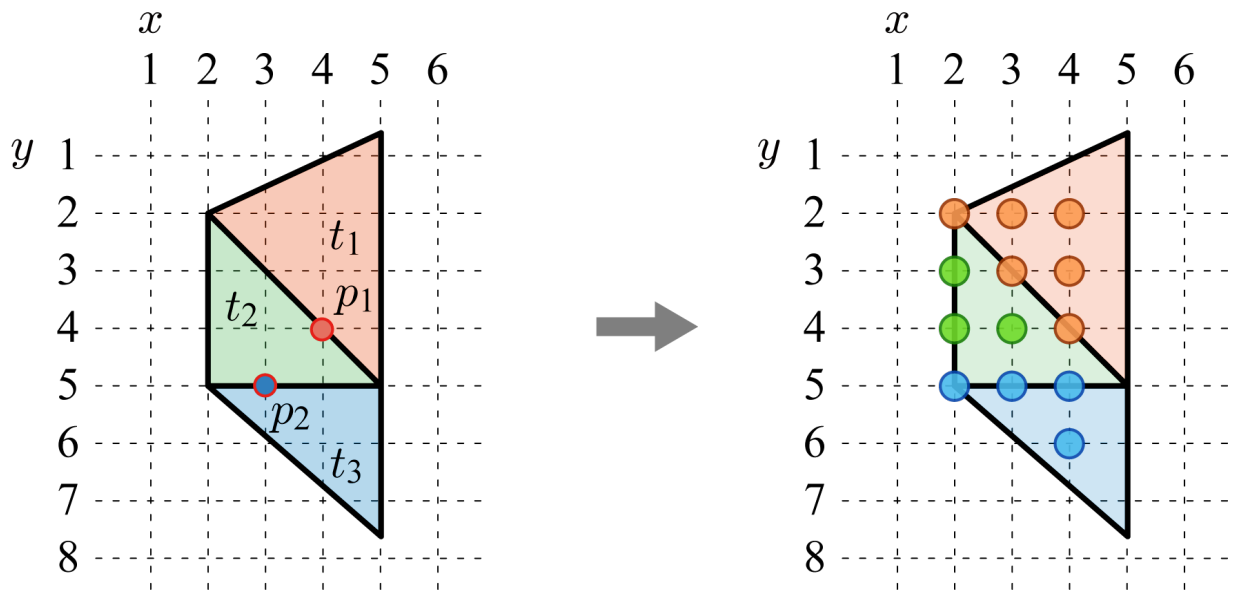


(e)

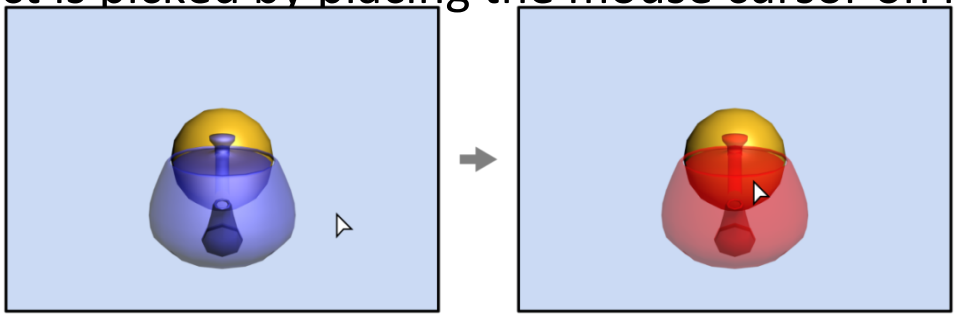


(f)

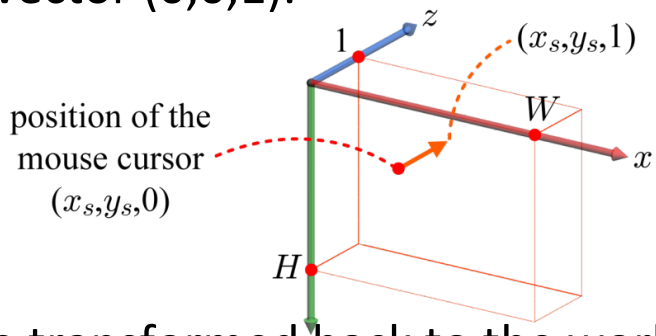
- When a pixel is on the edge shared by two triangles, we have to decide which triangle it belongs to.
- A triangle may have right, left, top or bottom edges.
- A pixel belongs to a triangle if it lies on the top or left edge of the triangle.



- An object is picked by placing the mouse cursor on it or clicking it.



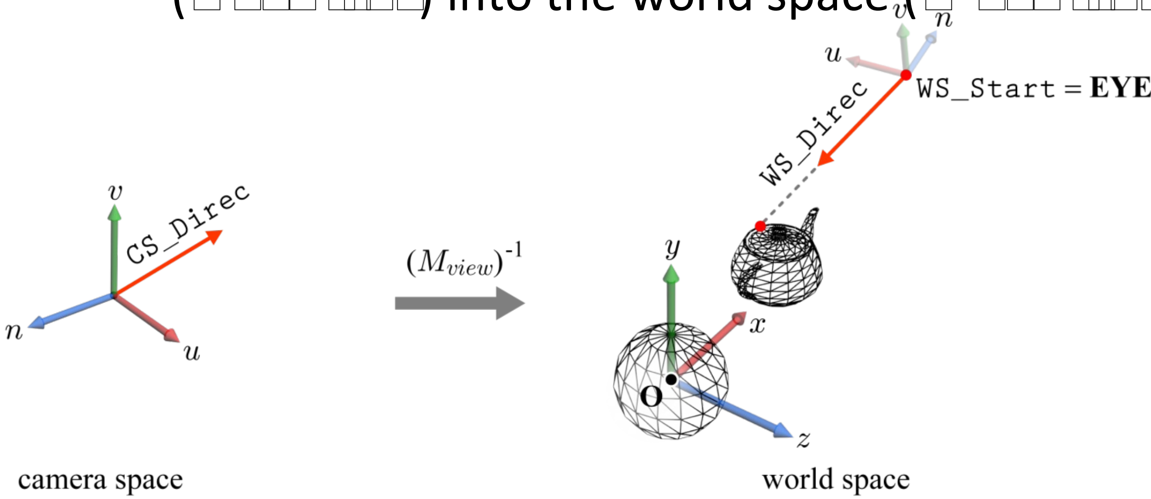
- Mouse clicking simply returns the 2D pixel coordinates (x_s,y_s). Given (x_s,y_s), we can consider a *ray* described by the start point $(x_s,y_s,0)$ and the direction vector (0,0,1).



- The ray will be transformed back to the world or object space, and then ray-object intersection test will be done. For now, let's transform the ray to the camera space

Object Picking (con)

- Let us transform the direction vector of the camera-space ray (CS_Direc) into the world space (WS_Direc).



camera space

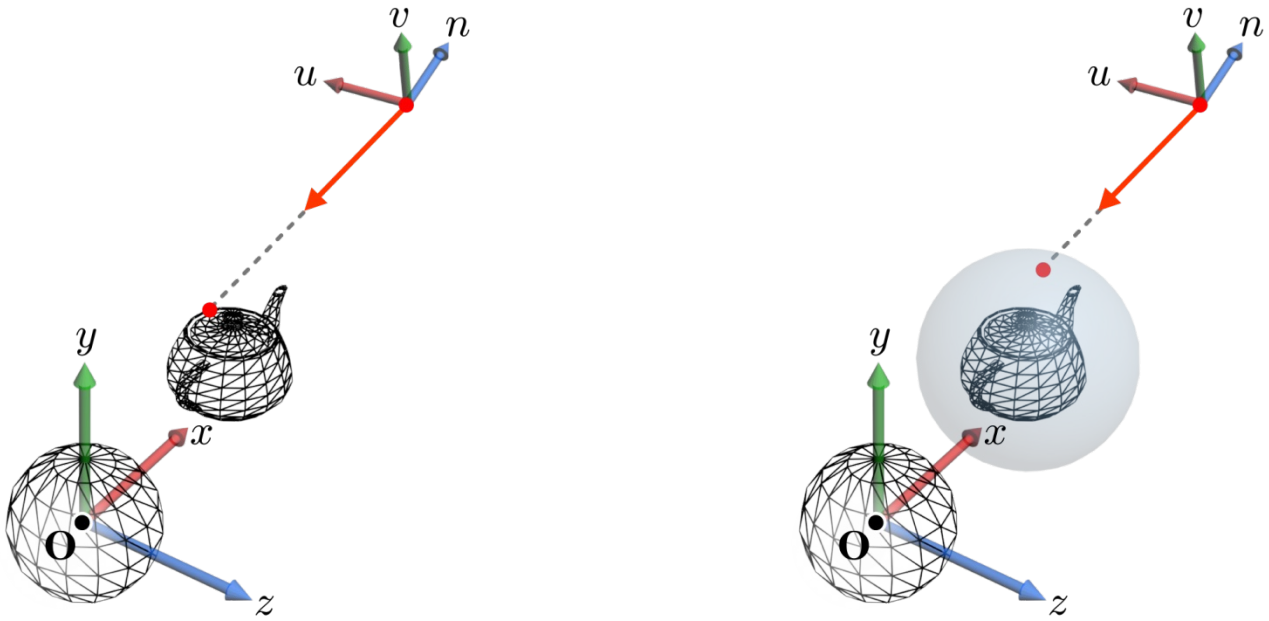
world space

$$
\begin{aligned}
M_{view}^{-1} &= (RT)^{-1} \\
&= T^{-1}R^{-1} \\
&= \begin{pmatrix} 1 & 0 & 0 & \mathbf{EYE}_x \\ 0 & 1 & 0 & \mathbf{EYE}_y \\ 0 & 0 & 1 & \mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} u_x & v_x & n_x & \mathbf{EYE}_x \\ u_y & v_y & n_y & \mathbf{EYE}_y \\ u_z & v_z & n_z & \mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

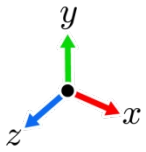
$$
\begin{pmatrix} u_x & v_x & n_x & \mathbf{EYE}_x \\ u_y & v_y & n_y & \mathbf{EYE}_y \\ u_z & v_z & n_z & \mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}
\begin{pmatrix} \frac{\frac{2x_s}{W}-1}{m_{11}} \\ \frac{-\frac{2y_s}{H}+1}{m_{22}} \\ -1 \\ 0 \end{pmatrix}
=
\begin{pmatrix} u_x\frac{\frac{2x_s}{W}-1}{m_{11}} + v_x\frac{-\frac{2y_s}{H}+1}{m_{22}} - n_x \\ u_y\frac{\frac{2x_s}{W}-1}{m_{11}} + v_y\frac{-\frac{2y_s}{H}+1}{m_{22}} - n_y \\ u_z\frac{\frac{2x_s}{W}-1}{m_{11}} + v_z\frac{-\frac{2y_s}{H}+1}{m_{22}} - n_z \\ 0 \end{pmatrix}
$$

- For now, assume that the start point of the camera-space ray is the origin. Then, the start point of the world-space ray (WS_Star) is simply **EYE**.

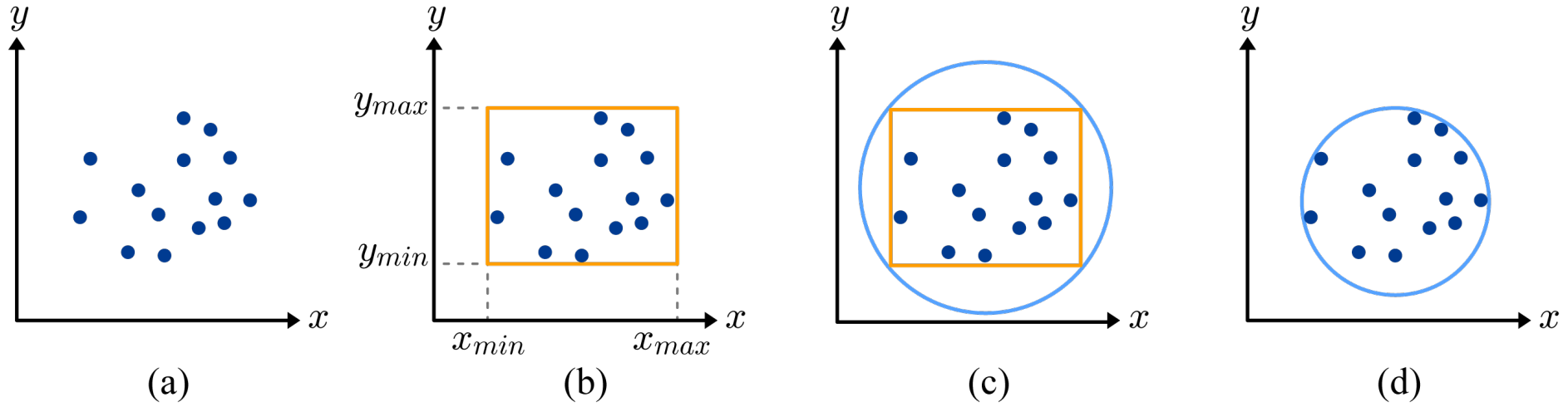
- In principle, we have to perform the ray intersection test with every triangle in the triangle list. A faster but less accurate method is to approximate each mesh with a bounding sphere that completely contains the mesh, and then do the ray-sphere intersection test.



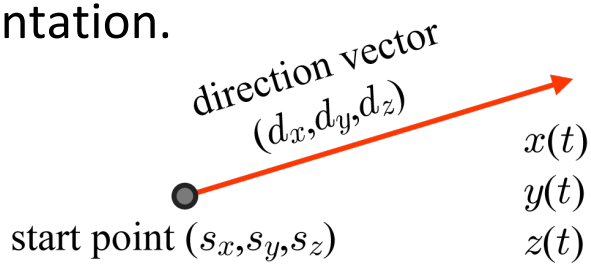
- Bounding volumes



- Bounding volume creation



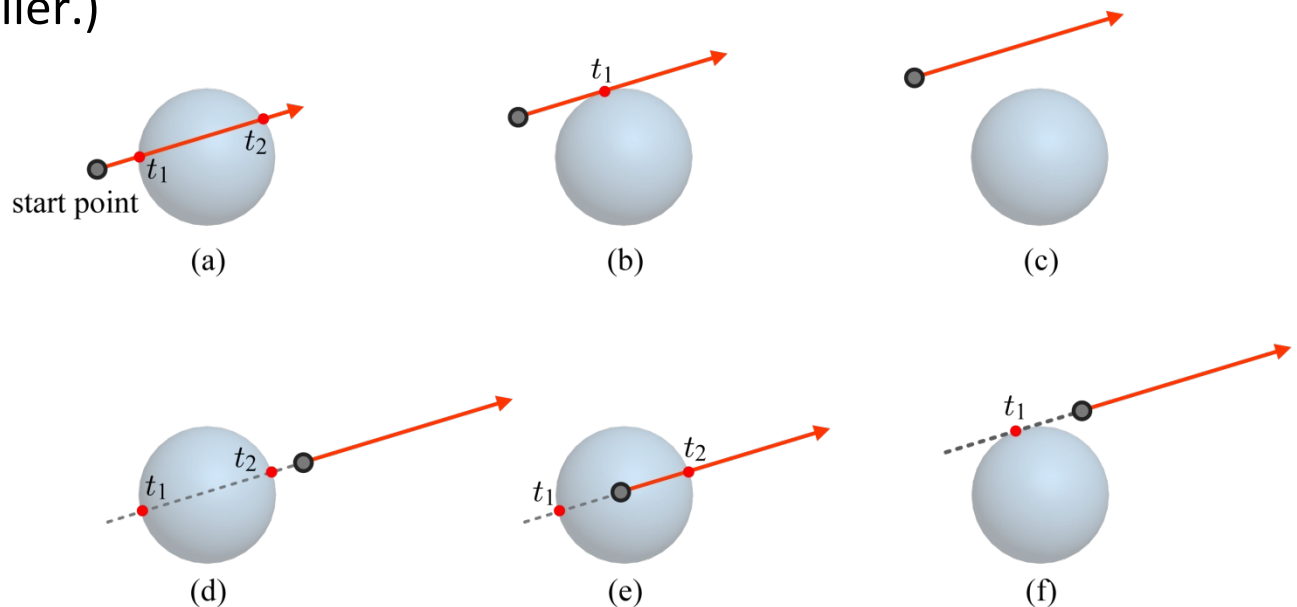
- For the ray-sphere intersection test, let us represent the ray in a parametric representation.



$$x(t) = s_x + td_x$$
$$y(t) = s_y + td_y$$
$$z(t) = s_z + td_z$$

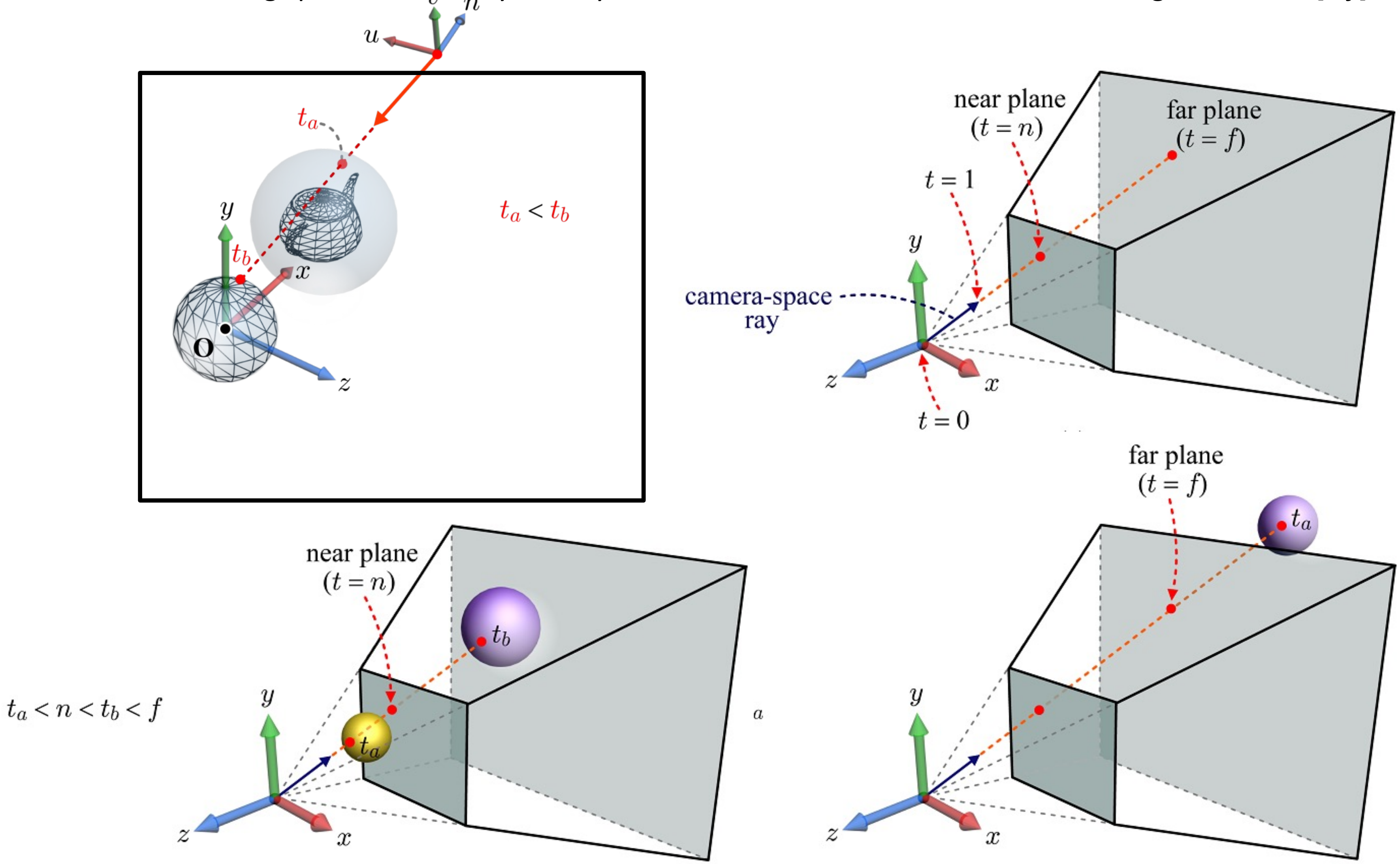
$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \Rightarrow at^2 + bt + c = 0 \Rightarrow t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Collect only positive *t*s. (Given two positive *t*s for a sphere, choose the smaller.)



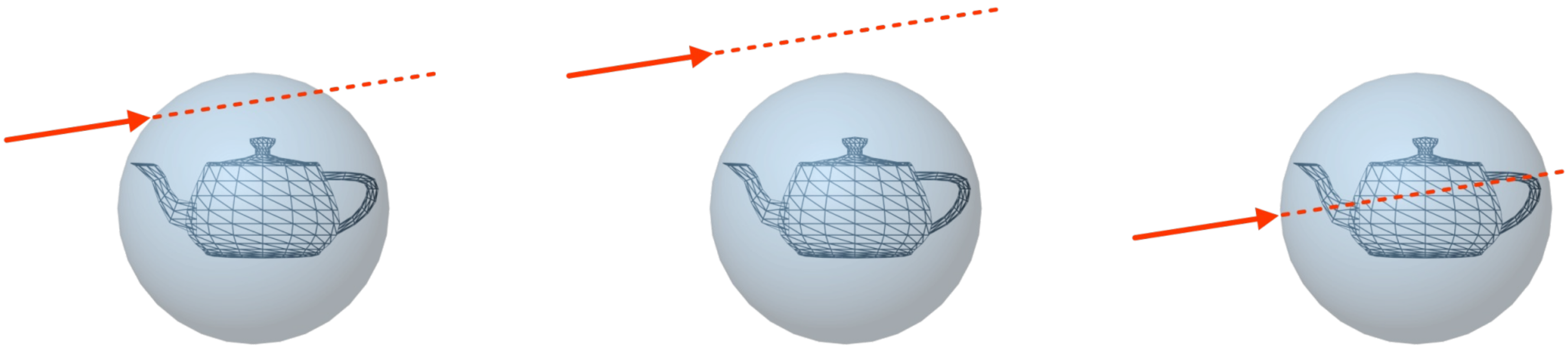
Object Picking (con)

- The bounding sphere hit first by the ray is the one with the smallest *t* with the range constrain [*n*,*f*].



Object Picking (cont')

- Ray-sphere intersection test is often performed at the preprocessing step, and discards the polygon mesh that is guaranteed not to intersect the ray.



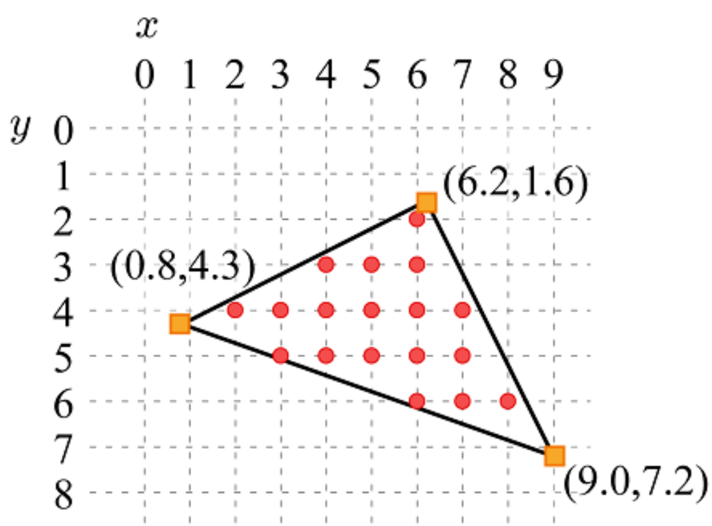
(a) (b) (c)

FRAGMENT PROCESSING

To determine final color of each fragment

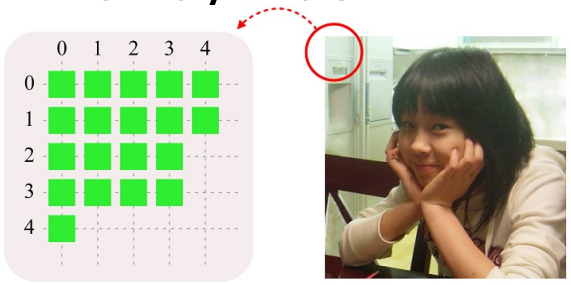
Fragment Processing

- Per-fragment attributes: normal vector, texture coordinates, color values, depth…

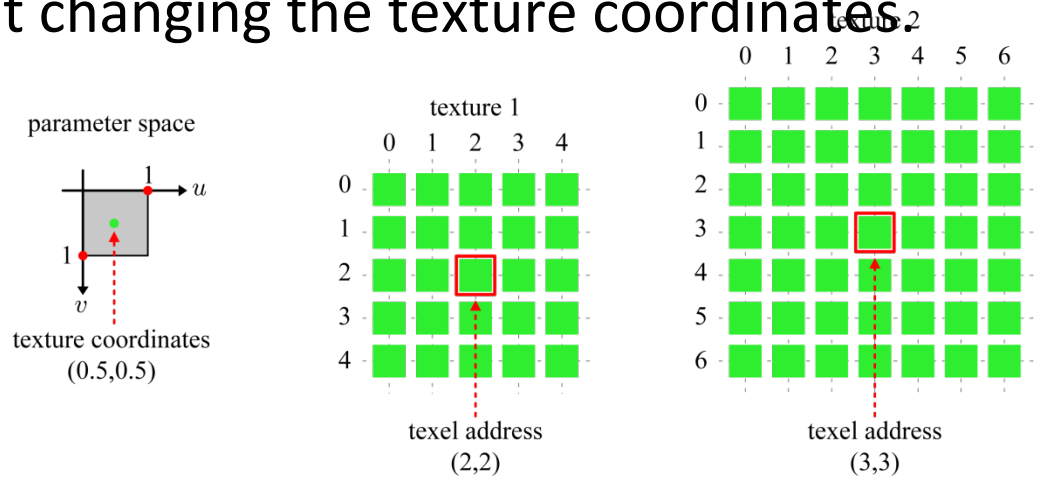


- Fragment processing stage determines the final color of each fragment.
  - Per-fragment lighting
  - Texturing

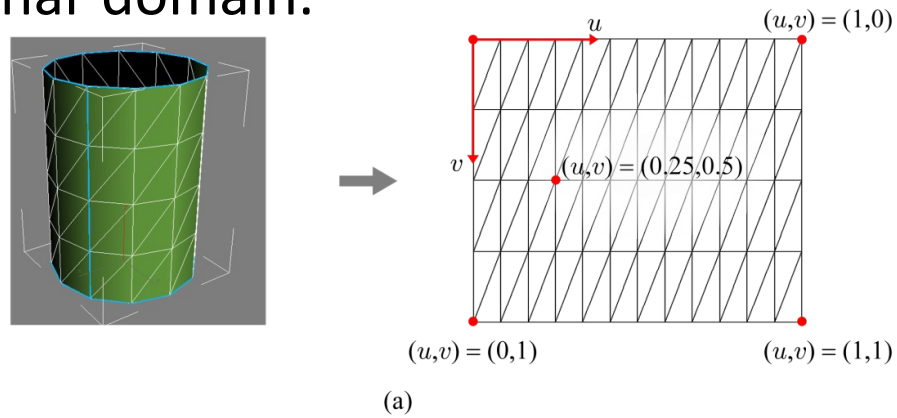
- An image texture is a 2D array of *texels* (texture elements). Each texel has a unique address, i.e., 2D array index.



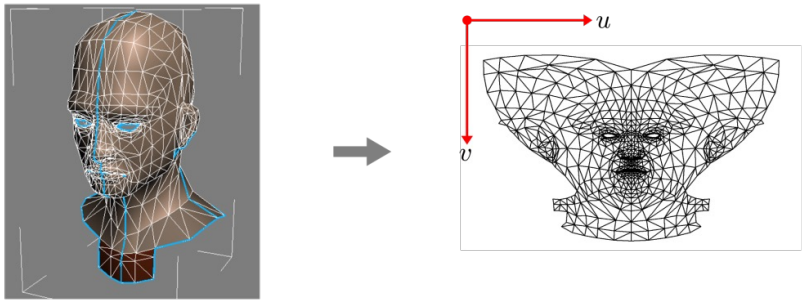
- Use normalized *texture coordinates* (*u*,*v*) in [0,1] → multiple images of different resolutions can be glued to a surface without changing the texture coordinates.



- *Surface parameterization:* texture coordinates are assigned to the vertices of the polygon mesh.
- In general, parameterization requires unfolding a 3D surface onto a 2D planar domain.



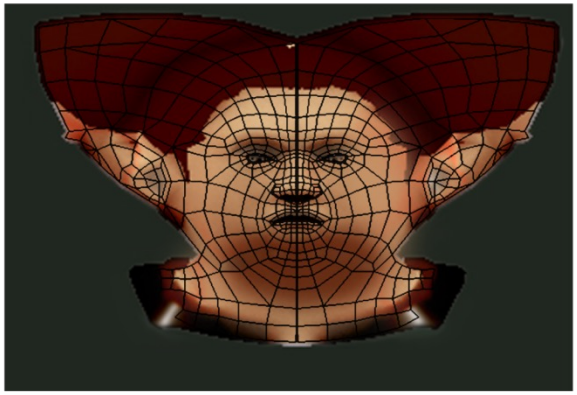
(a)



(b)

C□□□□□□□□A□□□□

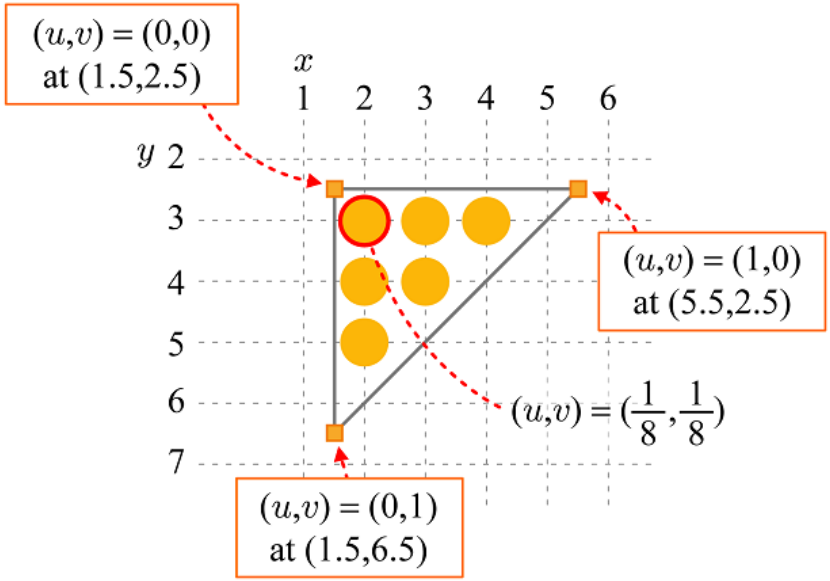
- Surface to be textured is subdivided into patches.
- Each patch is unfolded and parameterized. An image for a patch is called a *chart*.
- Multiple charts are usually packed and arranged in a texture, which is often called an *atlas*.



(a)

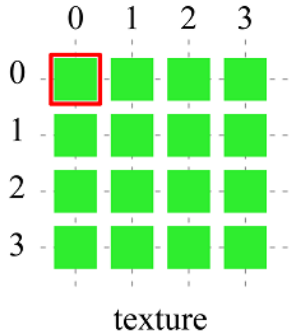
(b)

- Scan conversion is done with the texture coordinates.

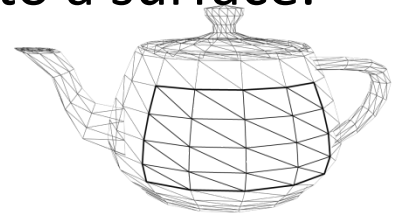


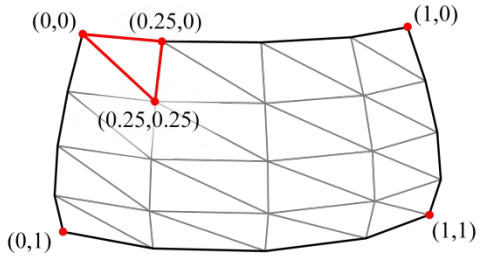
- D3D performs the following computation to convert the texture coordinates to the texel address

$$t_x = (u \times S_x) - 0.5$$
$$t_y = (v \times S_y) - 0.5$$

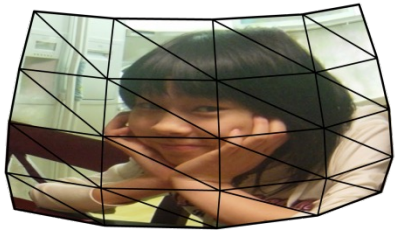


- Observe that multiple images of different resolutions can be glued to a surface.

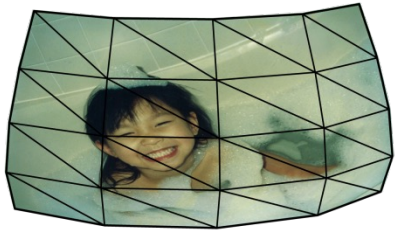




(a)

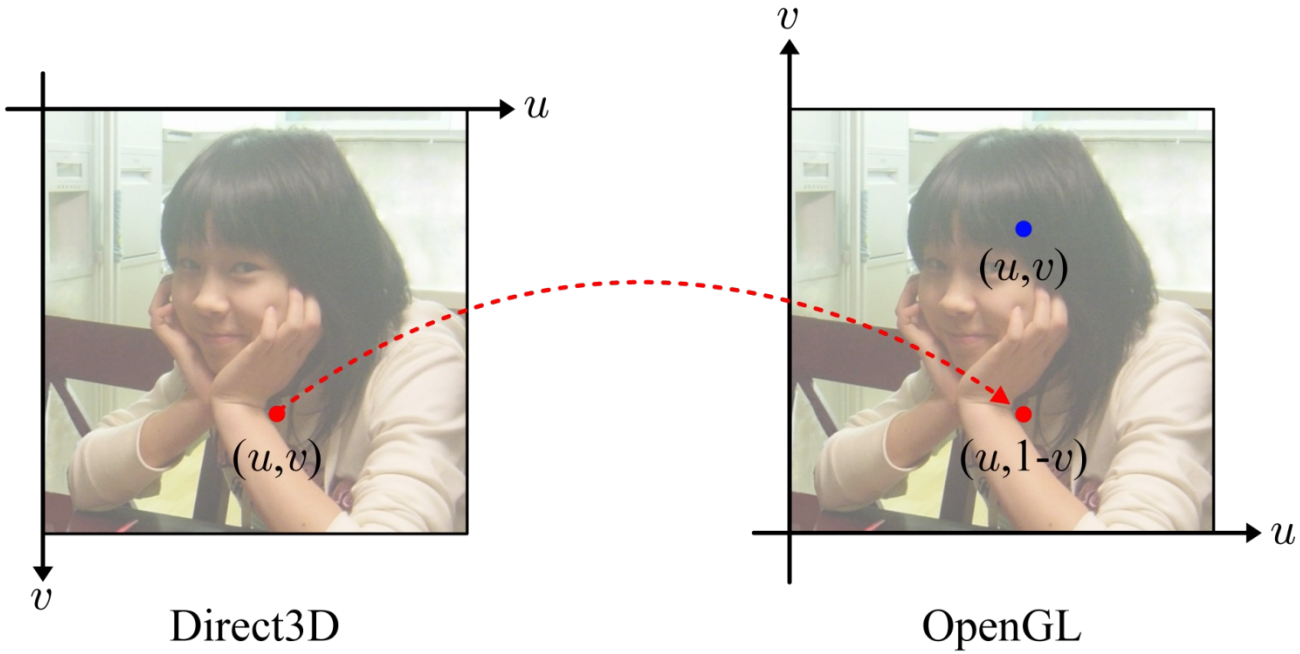


(b)



(c)

- Direct3D and OpenGL adopt different parameter spaces for texture coordinates.
- When a textured model is exported between Direct3D and OpenGL, the *v*-coordinates should be converted into (1-*v*).



OUTPUT MERGING

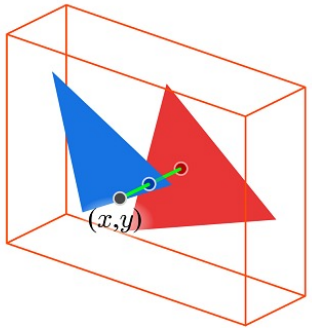
To determine the final color of each pixel from corresponding fragments

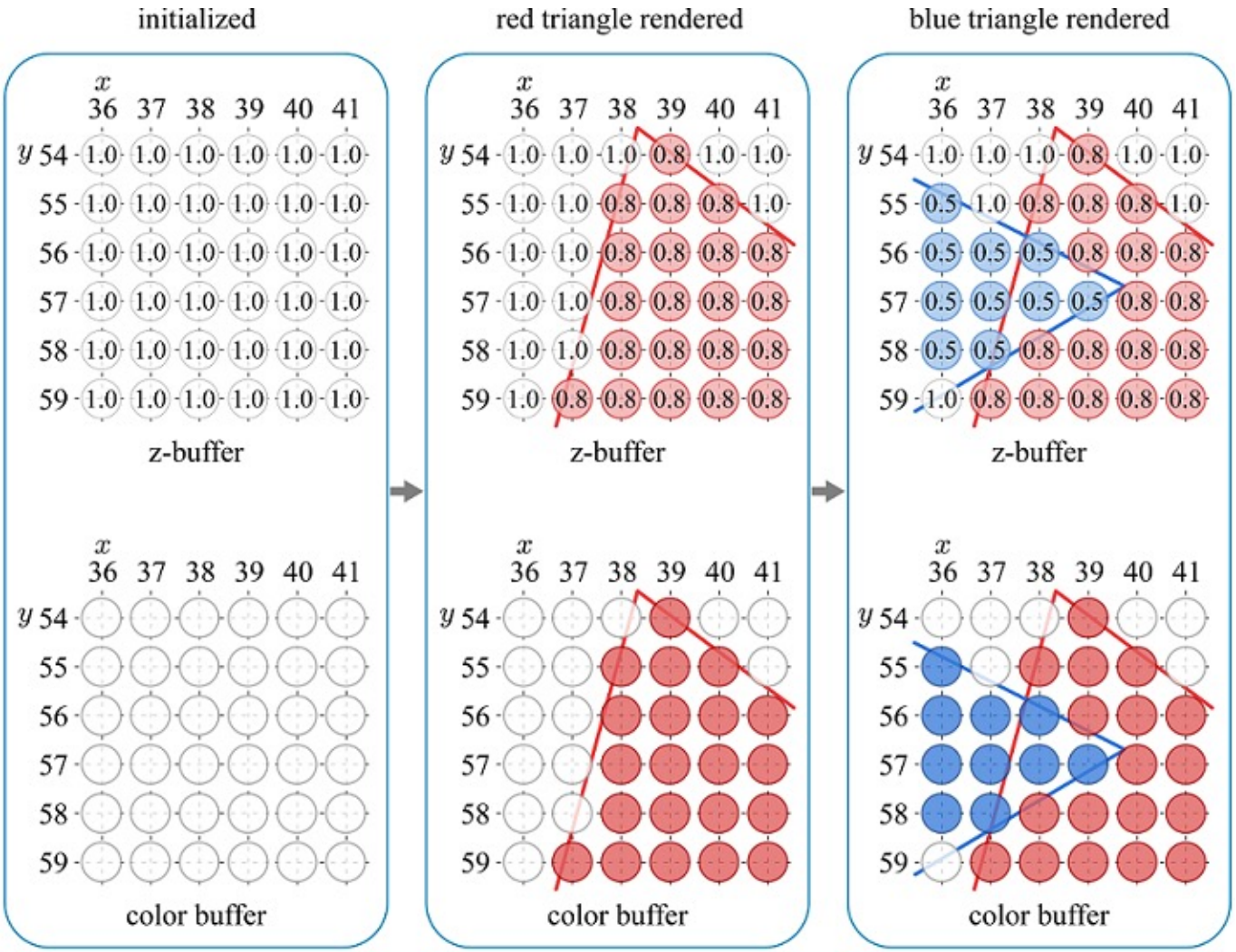
Z-buffering

- The output of the fragment program is often called the RGBAZ fragment.
  - A or alpha for representing the fragment's opacity
  - Z or depth used for -buffering
- Using alpha and depth values, the fragment competes or is merged with the pixel of the color buffer.

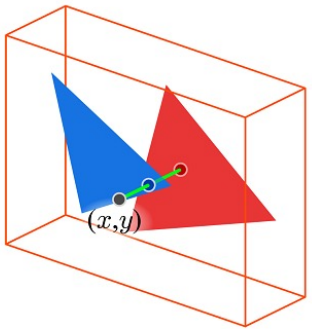
- The z-buffer has the same resolution as the color buffer, and records the *z*-values of the pixels currently stored in the color buffer.
  - When a fragment (,) is passed from the fragment program, its -value is compared with the zbuffer value (,)
  - If the fragment has a smaller zvalue, its color and zvalue are used to update the color buffer and -buffer (,), respectively
  - Otherwise, the fragment is discarded

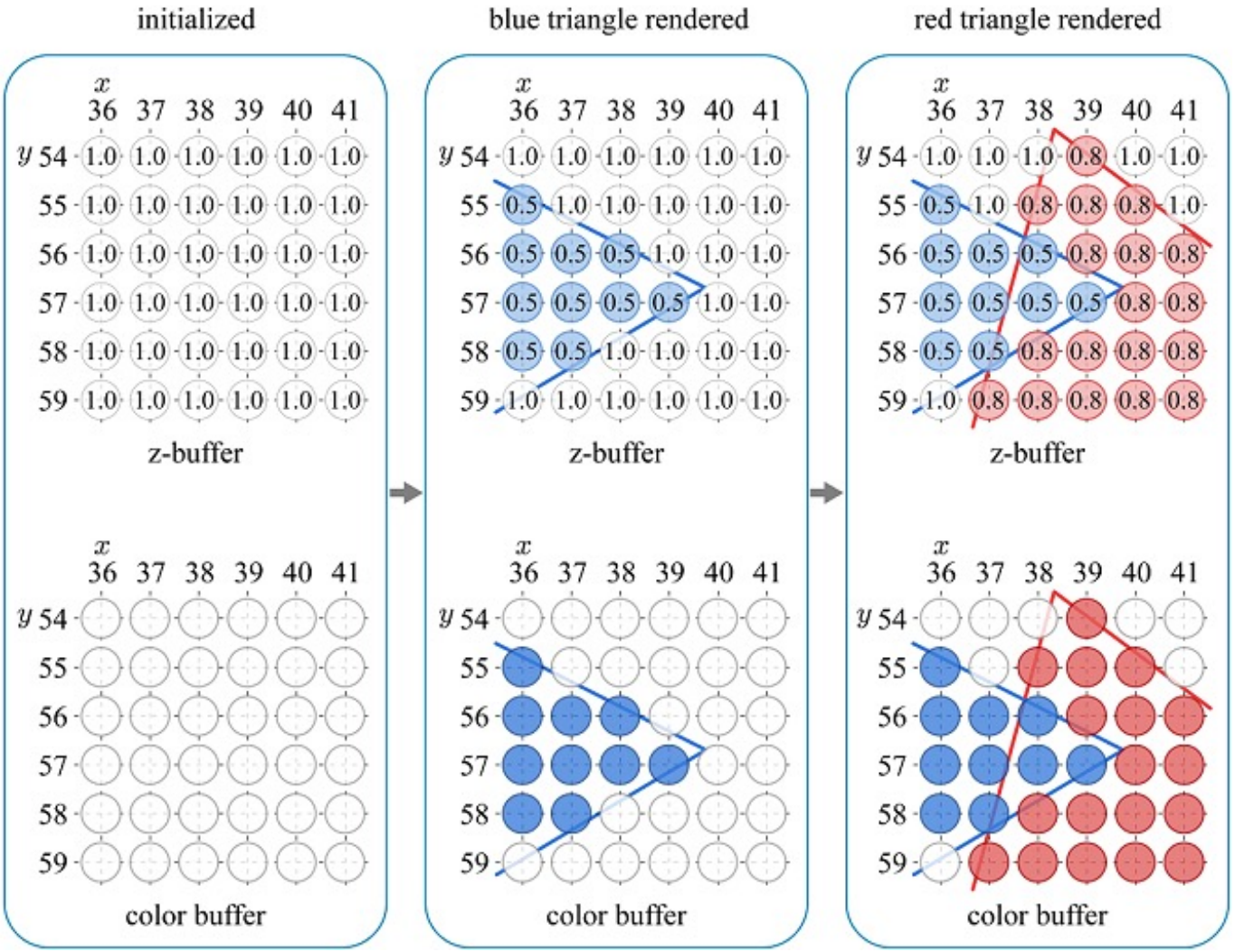
- Assume *MaxZ* is 1.0, red triangle's depth is 0.8, blue triangle's is 0.5.





- Rendering-order independence!!





Ed Catmull

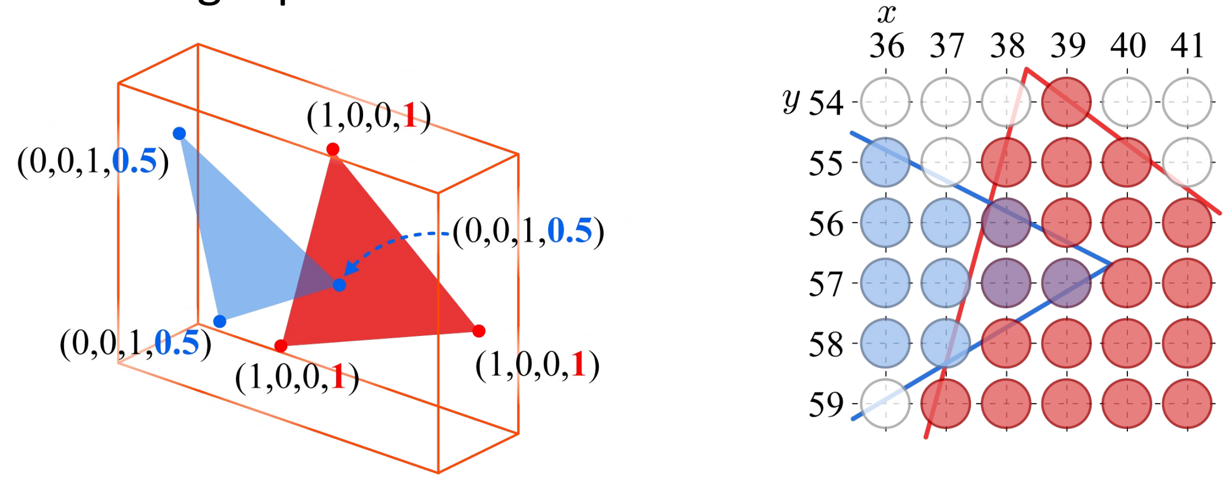
- Founder of Pixar and now president of Walt Disney and Pixar Animation Studios
- Academy awards:
  - 1993 Academy Scientific and Technical Award "for the development of PhotoRealistic RenderMan software which produces images used in motion pictures from 3D computer descriptions of shape and appearance"
  - 1996 Academy Scientific and Technical Award "for pioneering inventions in Digital Image Compositing"
  - 2001 Oscar 'for significant advancements to the field of motion picture rendering as exemplified in Pixar's RenderMan
  - 2008 Gordon E. Sawyer Award “for an individual in the motion picture industry whose technological contributions have brought credit to the industry“
- Ed Catmull’s work
  - Texture mapping
  - Z-buffering
  - Bicubic patches and subdivision surfaces

A[illegible] B[illegible]

- The alpha channel in the range of [0,1]
  - 0 [illegible]
  - 1 [illegible]

$$c = \alpha c_f + (1 - \alpha) c_p$$

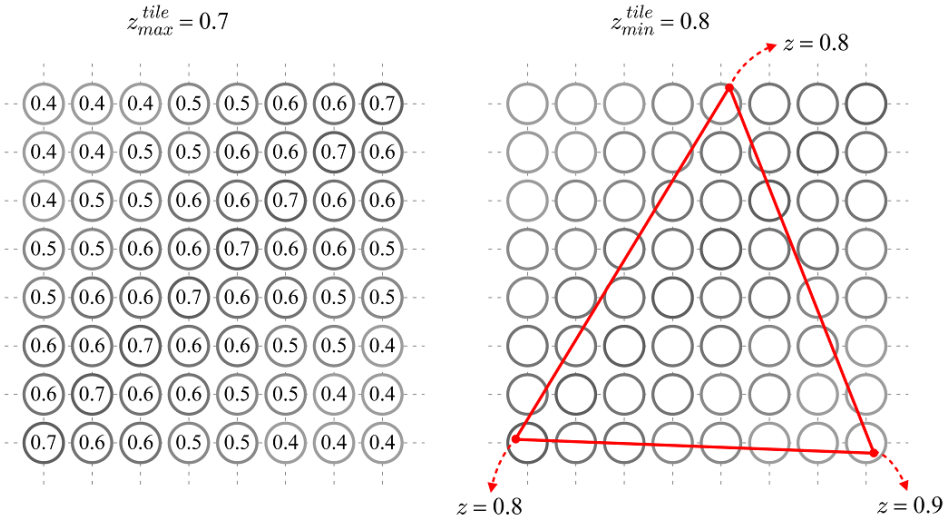
- A typical blending equation is described as follows:



(a) (b)

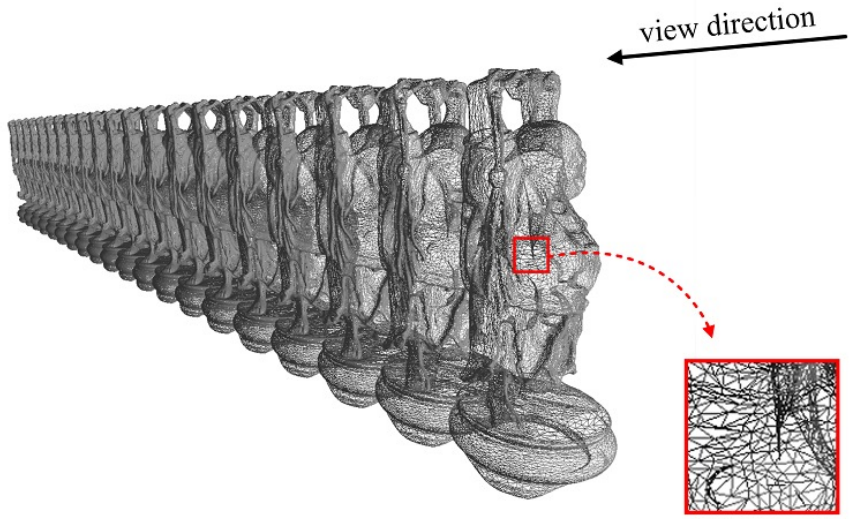
- For alpha blending, the primitives cannot be rendered in an arbitrary order. They must be rendered *after* all opaque primitives, and in *back-to-front order*. Therefore, the partially transparent objects should be *sorted*.

- Let's kill the fragments "before the fragment processing stage" if possible.
  - A [illegible]
  - [illegible]



If $z_{max}^{tile} < z_{min}^{tri}$, i.e., the closest vertex is deeper than the deepest pixel of the tile, the triangle is determined to be invisible and no fragment of the triangle has a chance to enter the fragment processing stage.

- Z-culling is powerful, and is enabled by default.
- Consider the scene, where the first object occludes almost of triangles of the other objects.



- In a test, the average frame rate with the front-to-back ordered objects is 12.75 fps whereas that with the back-to-front ordered objects is 2.71.
- This shows that the triangles may need to be sorted in *front-to-back order* in order to maximize the performance increase brought by z-culling.

- Two-pass algorithm
  - 1
  - 2
- The pre-Z pass algorithm even with back-to-front ordered objects outperforms the single-pass front-to-back rendering.