GPU Graphics Pipeline (Part I)

Computer Graphics

Yu-Ting Wu

Outline

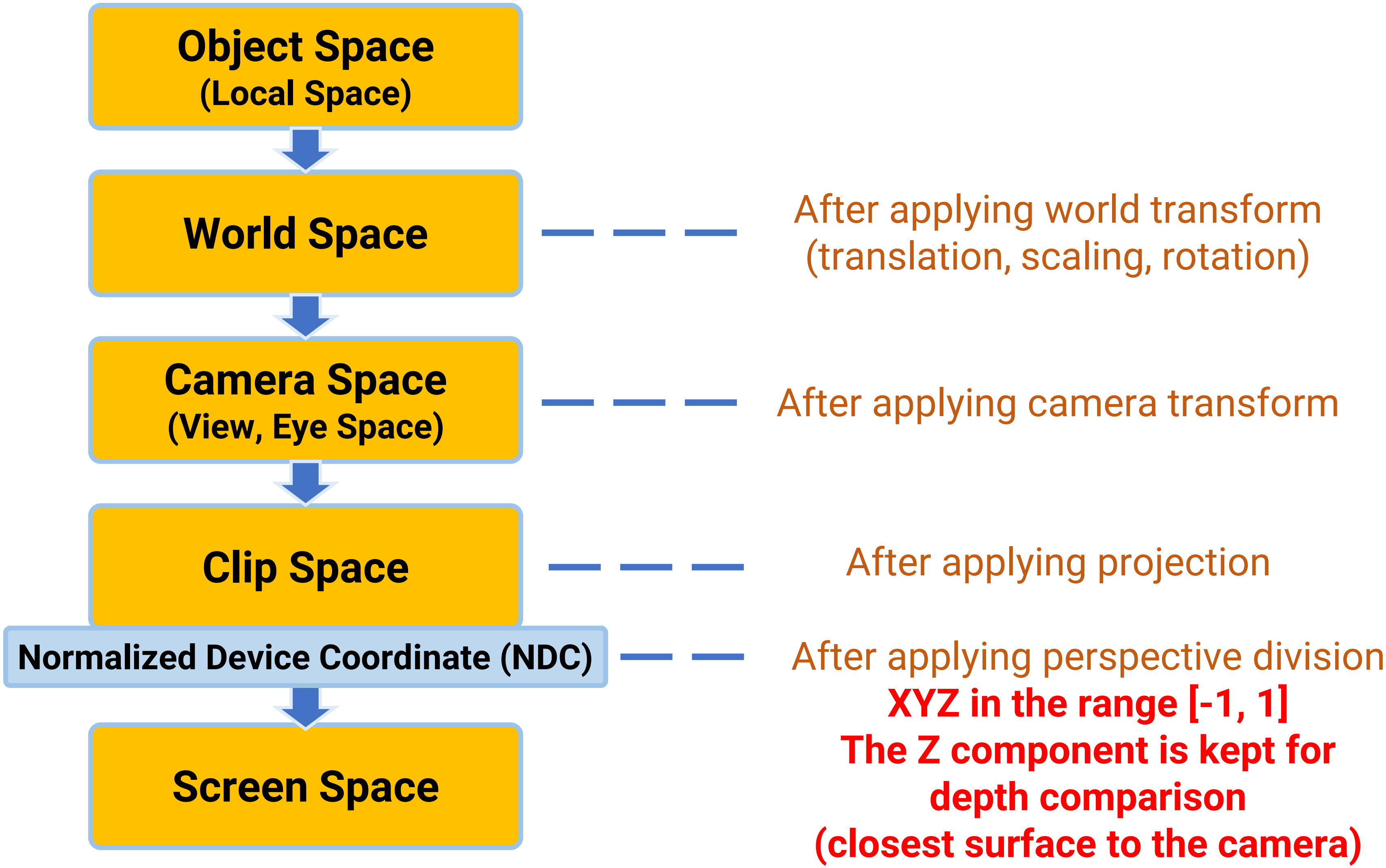
- GPU graphics pipeline
- OpenGL graphics pipeline 1.x

(Part I)

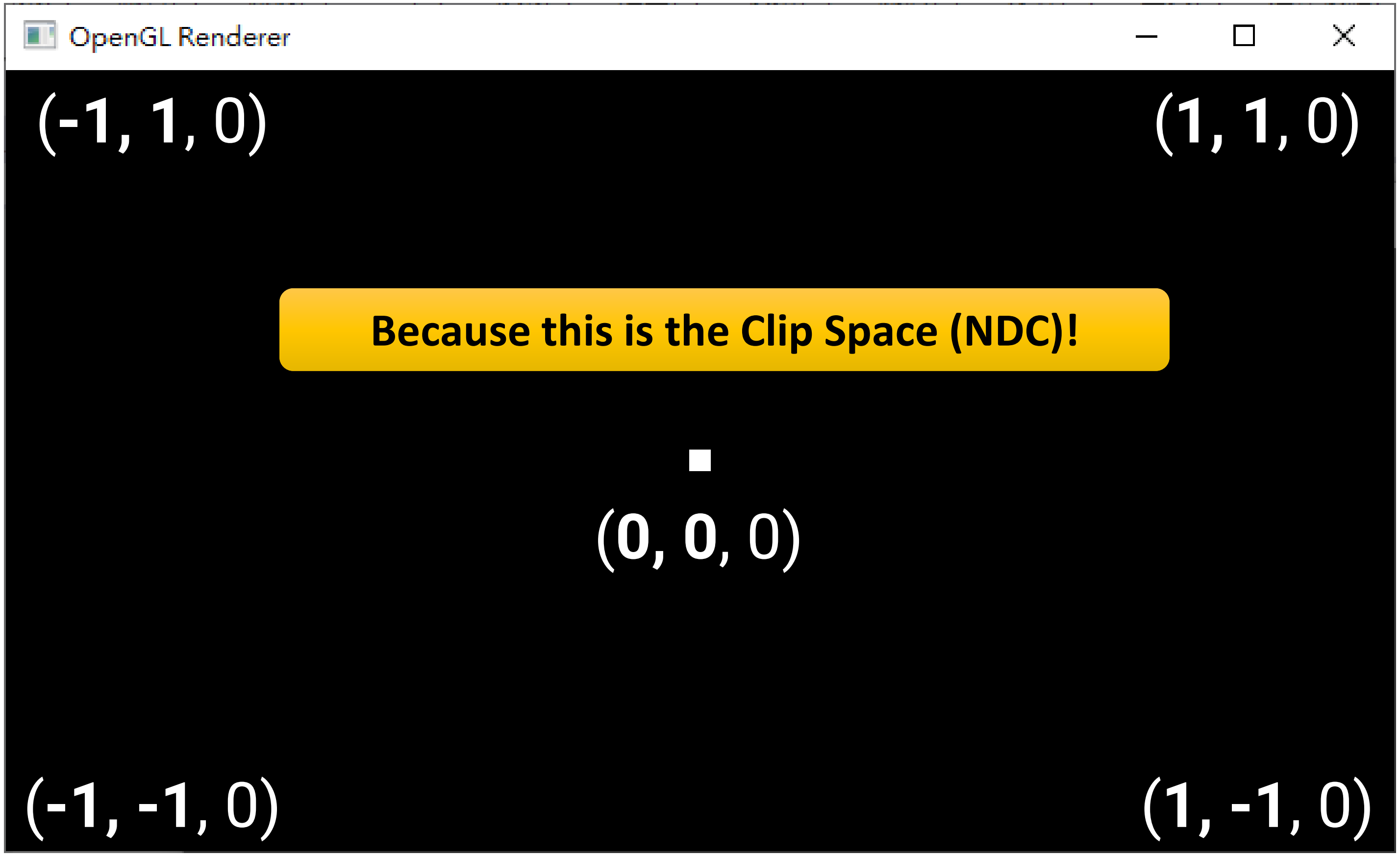
- OpenGL graphics pipeline 2.0
- OpenGL and shader implementation

(Part II)

Recap: Transformations



Recap: NDC

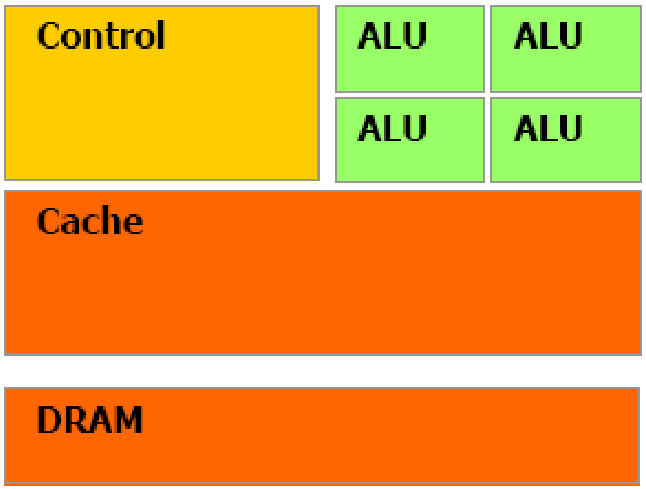


What about the *z* coordinate? You can find the point will only be visible if its *z* value is within **[-1, 1]**

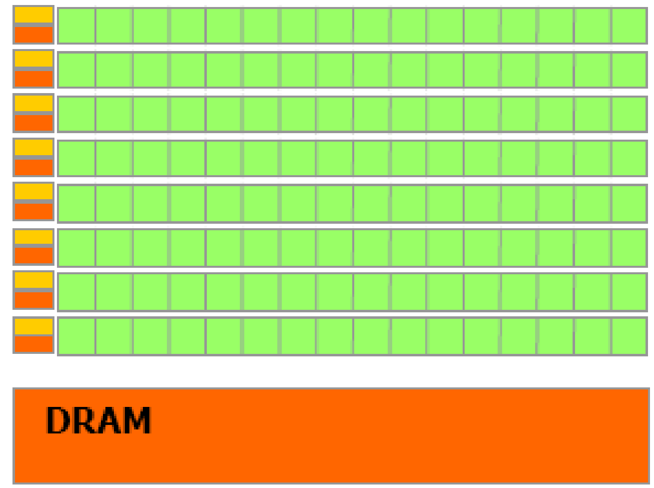
Outline

- **GPU graphics pipeline**
- OpenGL graphics pipeline 1.x
- OpenGL graphics pipeline 2.0
- OpenGL and shader implementation

CPU v.s. GPU



CPU



GPU

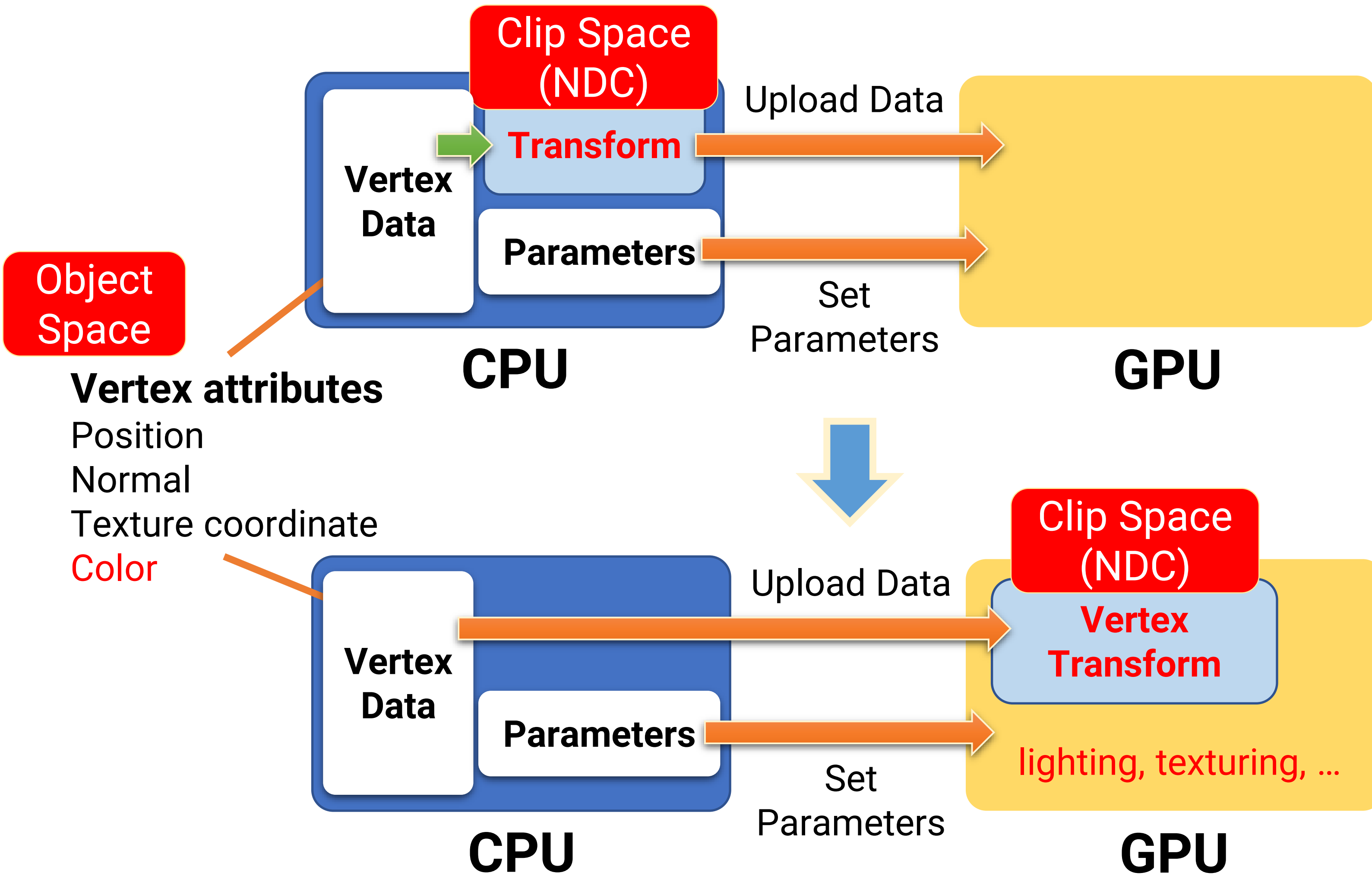
Good at

- Serial processing
- Control (branching)
- Larger cache

Good at

- Parallel processing
- SIMD
- Higher throughput

CPU v.s. GPU (cont.)



Pipeline

GPU Graphics Pipeline Overview

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 3 stages

Application | **Geometry Processing** | **Rasterization** | **Pixel Processing**

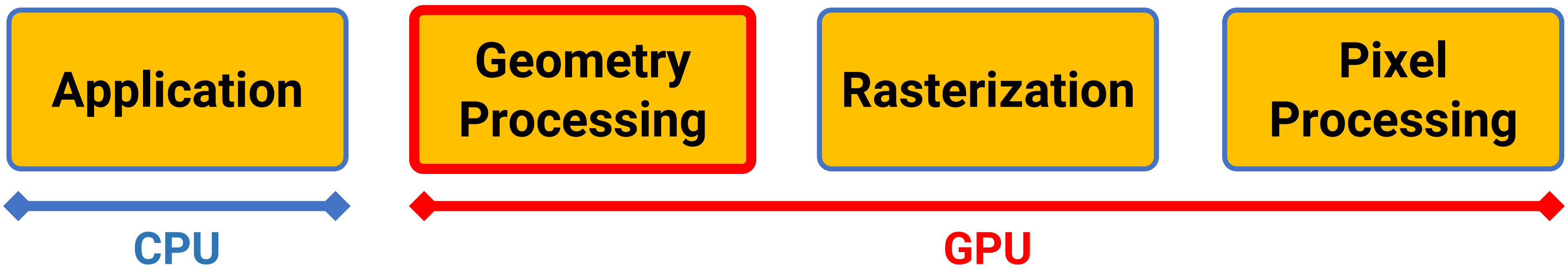
CPU: Application

GPU: Geometry Processing, Rasterization, Pixel Processing

- Physical simulation
- Animation
- Collision detection
- Global acceleration
- etc.

GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 3 stages



- Vertex transform and projection
- Vertex lighting and shading (rarely used now)
- Geometry assembly
- Clipping
- Culling

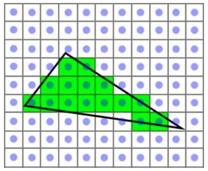
GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 3 stages

Application | **Geometry Processing** | **Rasterization** | **Pixel Processing**

CPU | **GPU**

- Fragments (pixels) generation
  - Attribute interpolation



from continuous to discrete

GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 3 stages

Application → Geometry Processing → Rasterization → **Pixel Processing**

CPU: Application

GPU: Geometry Processing, Rasterization, Pixel Processing

- Pixel shading / Texturing
- Depth testing
- Alpha blending

GPU Graphics Pipeline Overview (cont.)

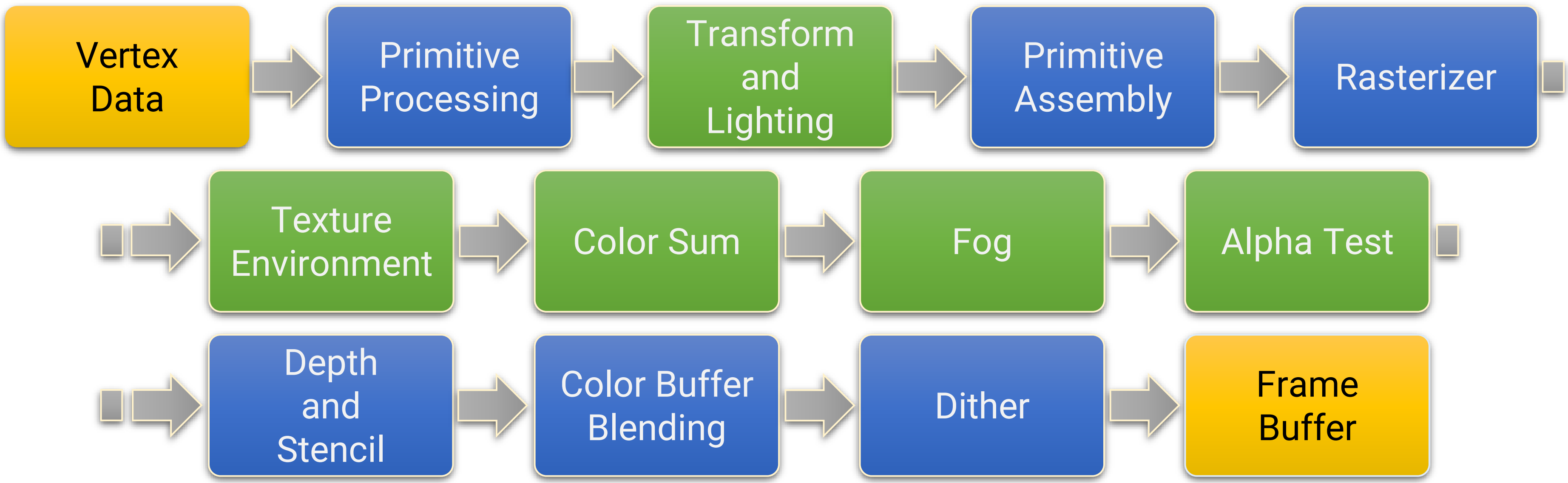
- In this topic, we will first introduce the GPU rendering pipeline revealed in **OpenGL 1.x**
- After that, we will show why (and how) some stages become **programmable** in **OpenGL 2.0**

Outline

- GPU graphics pipeline
- **OpenGL graphics pipeline 1.x**
- OpenGL graphics pipeline 2.0
- OpenGL and shader implementation

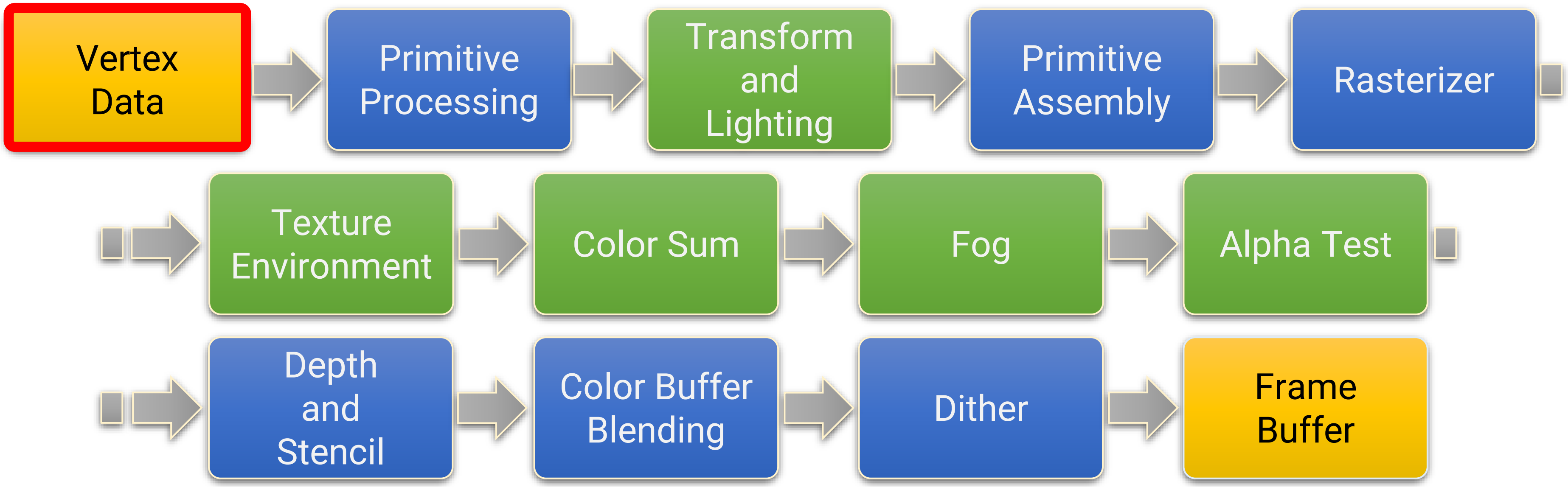
OpenGL (1.x) Fixed Function Pipeline

- Used when OpenGL was first introduced
- All the functions performed by OpenGL are **fixed** and **could not** be modified except through the manipulation of the **rendering states**



- The stages shown in green have been replaced by **shaders**

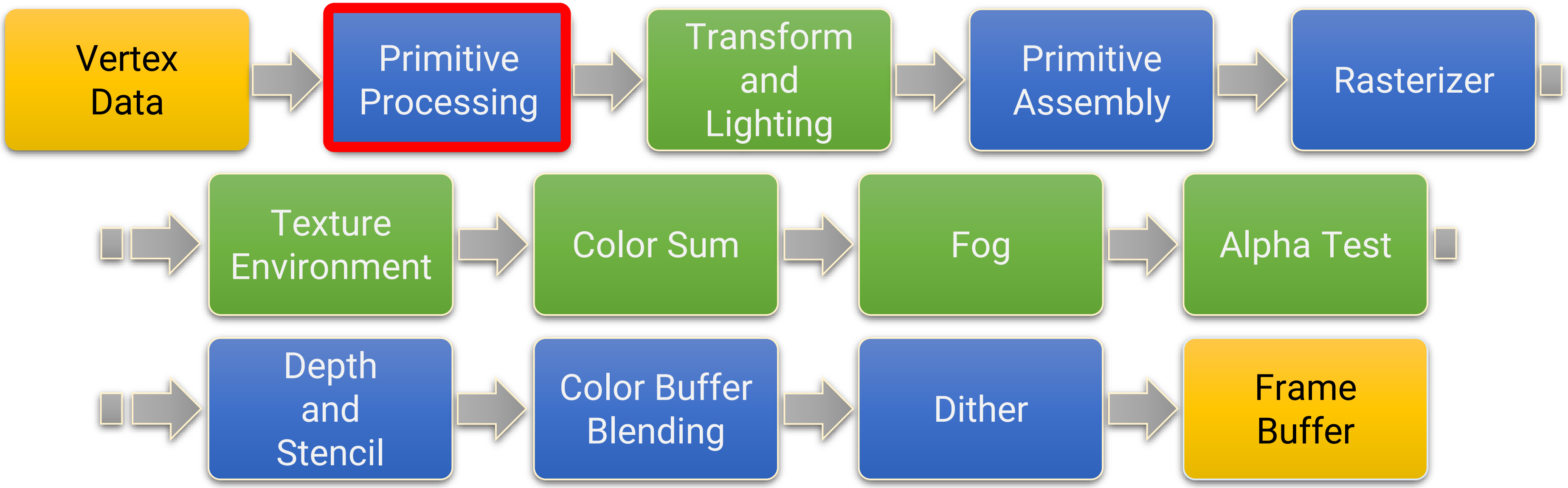
OpenGL (1.x) Fixed Function Pipeline



Vertex Data

- Send the vertex data to the GPU
- **Vertex attributes** include vertex position, vertex normal, texture coordinate, vertex color, fog coordinate, etc.
- The vertex data processed by the GPU is referred to as the **vertex stream**

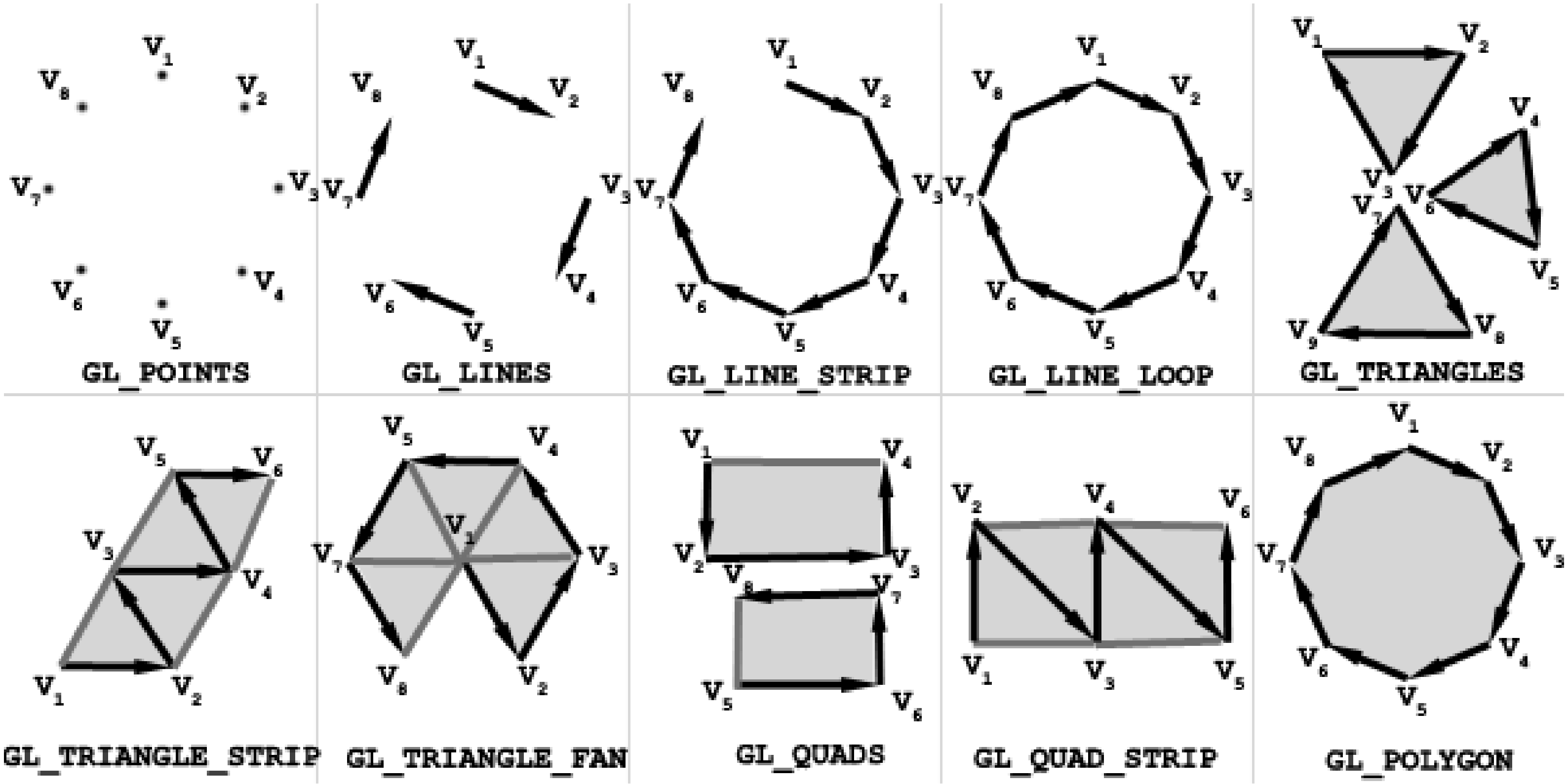
OpenGL (1.x) Fixed Function Pipeline



Primitive Processing

- Vertex stream is processed **per primitive**
- OpenGL supports several types of primitives, including points, lines, triangles, ~~quads~~, and ~~polygons~~ (**deprecated** after OpenGL 3.1)

Primitive Processing (cont.)



primitive types in OpenGL 1.1

Primitive Processing (cont.)

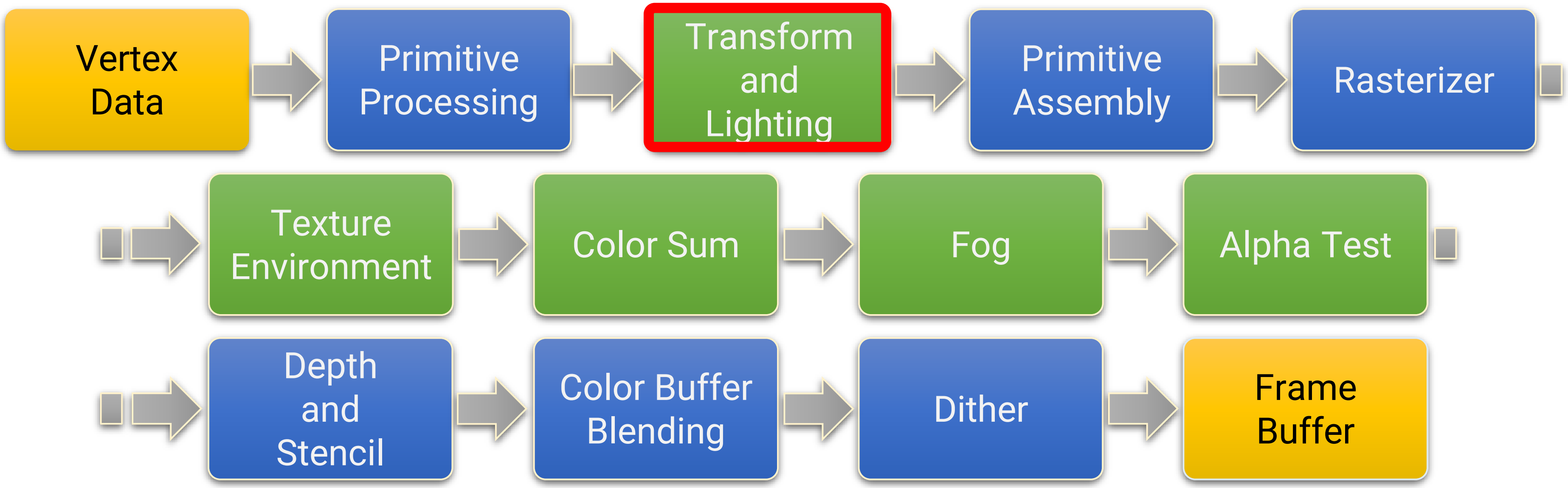
```
glBegin(GL_POINTS); //starts drawing of points
  glVertex3f(1.0f,1.0f,0.0f);//upper-right corner
  glVertex3f(-1.0f,-1.0f,0.0f);//lower-left corner
glEnd();//end drawing of points
```

```
glBegin(GL_TRIANGLES);//start drawing triangles
  glVertex3f(-1.0f,-0.25f,0.0f);//triangle one first vertex
  glVertex3f(-0.5f,-0.25f,0.0f);//triangle one second vertex
  glVertex3f(-0.75f,0.25f,0.0f);//triangle one third vertex
  //drawing a new triangle
  glVertex3f(0.5f,-0.25f,0.0f);//triangle two first vertex
  glVertex3f(1.0f,-0.25f,0.0f);//triangle two second vertex
  glVertex3f(0.75f,0.25f,0.0f);//triangle two third vertex
glEnd();//end drawing of triangles
```

```
glBegin(GL_POLYGON);//begin drawing of polygon
  glVertex3f(-0.5f,0.5f,0.0f);//first vertex
  glVertex3f(0.5f,0.5f,0.0f);//second vertex
  glVertex3f(1.0f,0.0f,0.0f);//third vertex
  glVertex3f(0.5f,-0.5f,0.0f);//fourth vertex
  glVertex3f(-0.5f,-0.5f,0.0f);//fifth vertex
  glVertex3f(-1.0f,0.0f,0.0f);//sixth vertex
glEnd();//end drawing of polygon
```

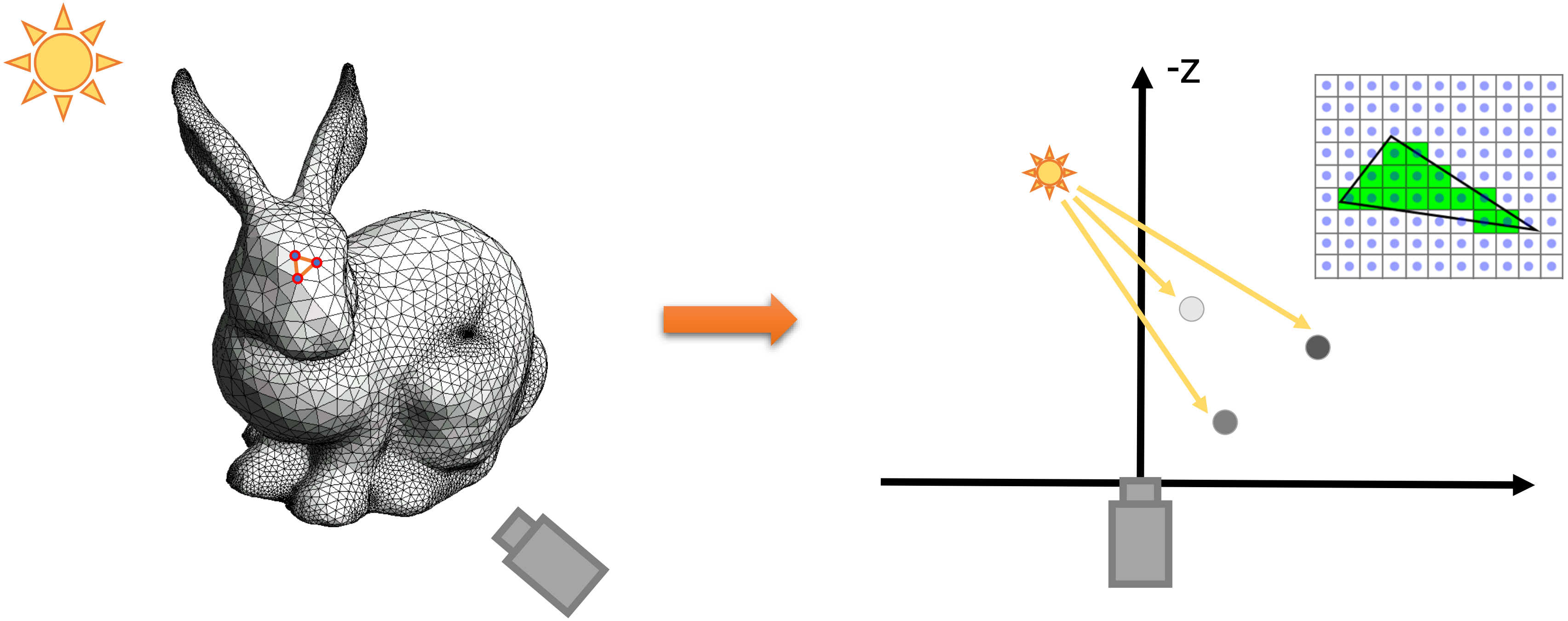
primitive drawing in OpenGL 1.1
(deprecated, DO NOT USE!)

OpenGL (1.x) Fixed Function Pipeline



Transform and Lighting

- Vertex is transformed to **camera space** by the current **ModelView** matrix
- Lighting is computed at each vertex (Gouraud shading)

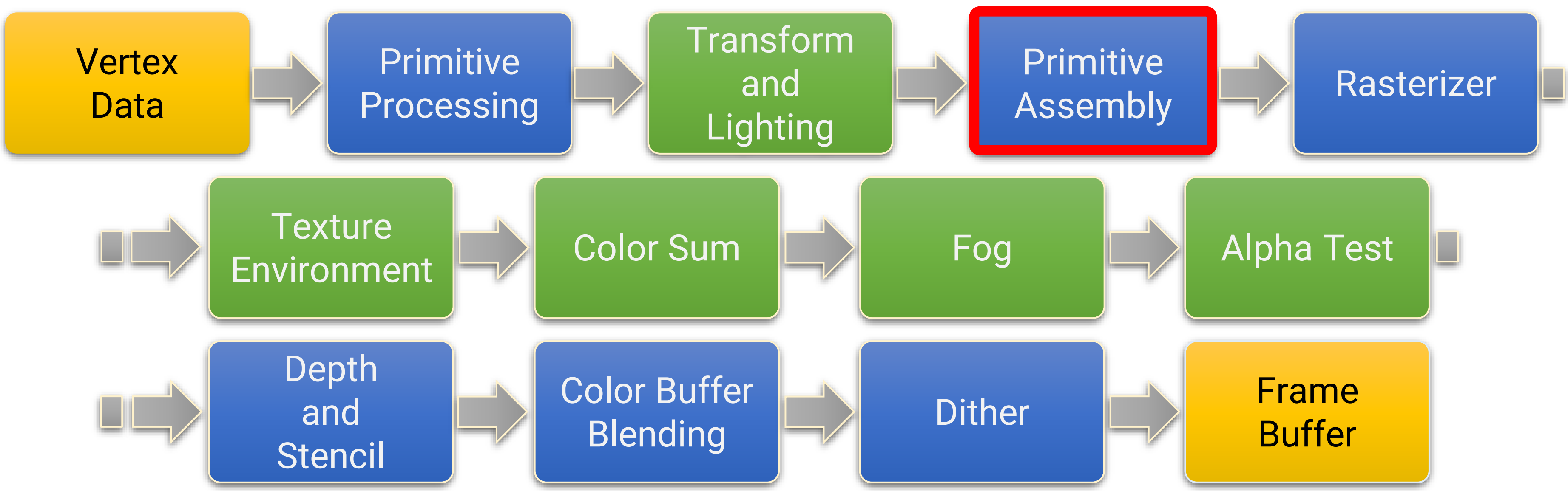


Transform and Lighting (cont.)

- **Transform in OpenGL 1.x** **(deprecated, DO NOT USE!)**

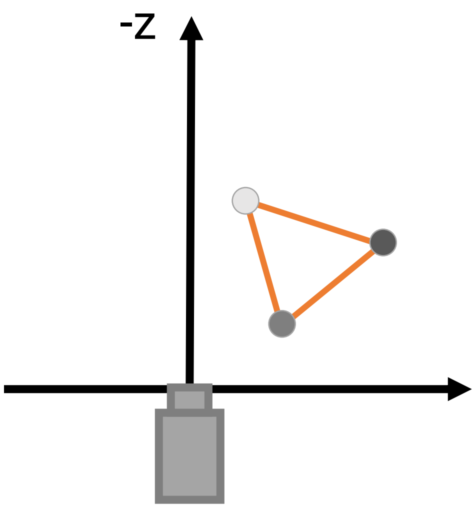
```
void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glLoadIdentity ();              /* clear the matrix */
           /* viewing transformation  */
V  gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
M  glScalef (1.0, 2.0, 1.0);      /* modeling transformation */
   glutWireCube (1.0);
   glFlush ();
}

void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
P  glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);
}
```

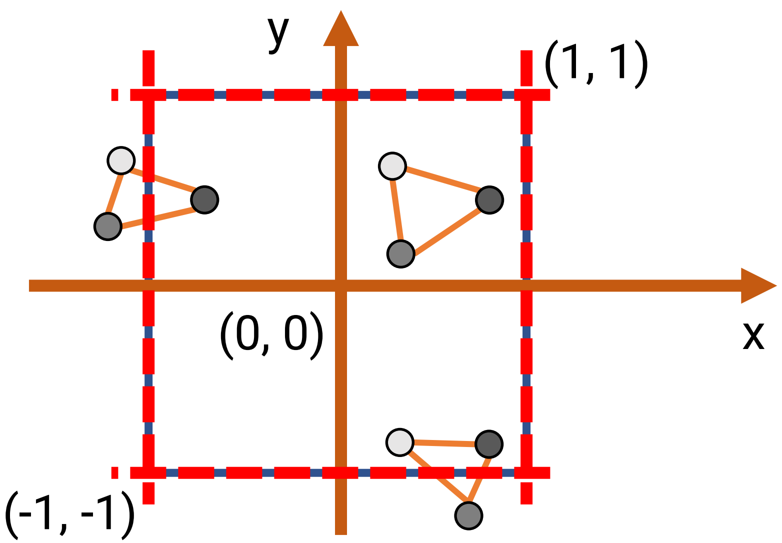
OpenGL (1.x) Fixed Function Pipeline



Primitive Assembly

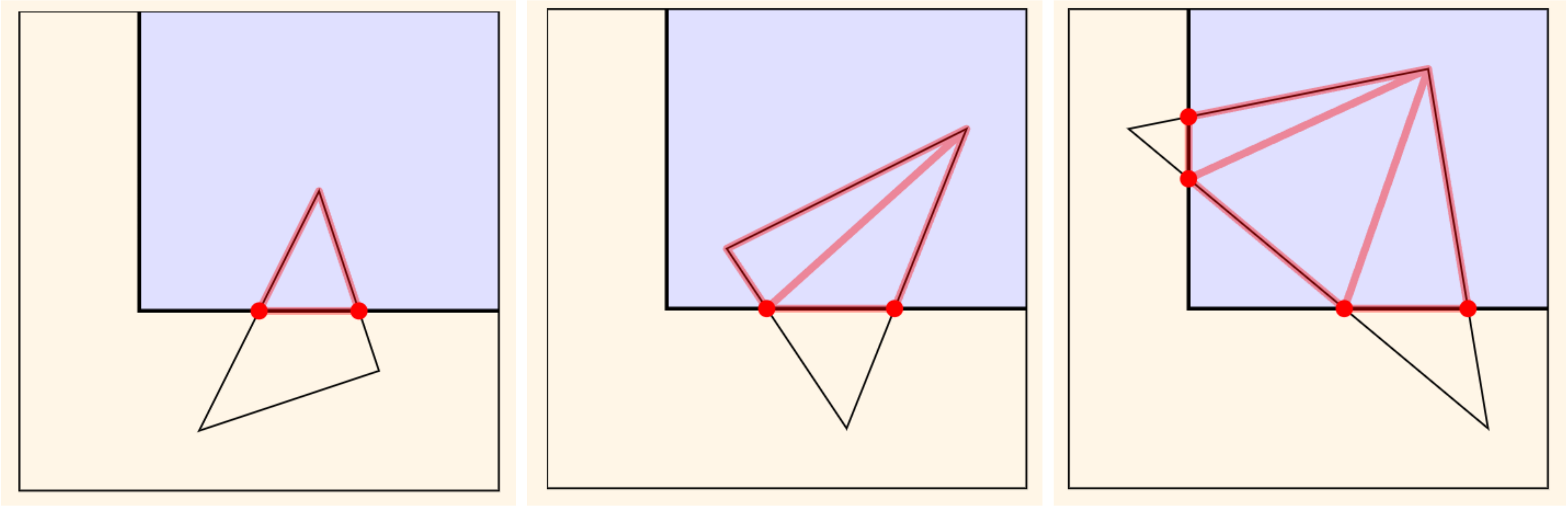
- Convert primitives from the basic primitive types (e.g., triangle strip) into triangles
- Triangles are transformed to NDC and got **clipped** to fit within the viewport boundaries





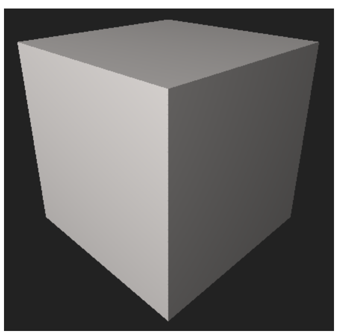
Primitive Assembly (cont.)

- **Clipping**
  - In OpenGL, clipping is performed by adding new vertices and triangulation

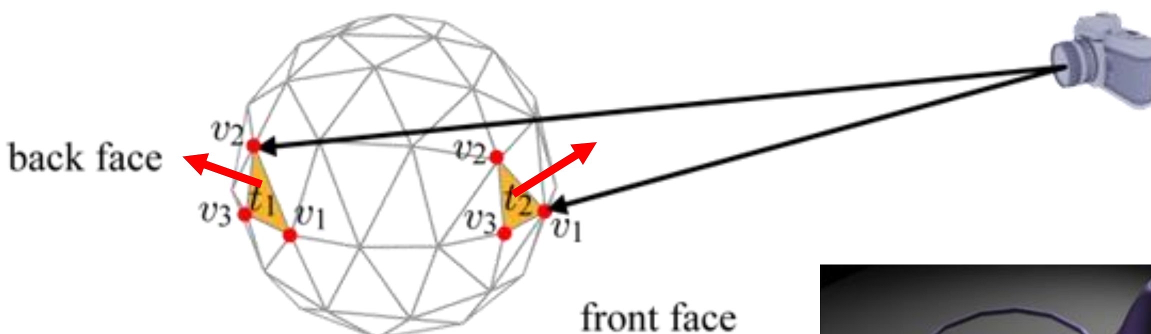


Primitive Assembly (cont.)

- **Back-face culling**
  - If a triangle is facing away from the camera, it will never be seen
  - We can cull these back-facing triangles for saving unnecessary computation



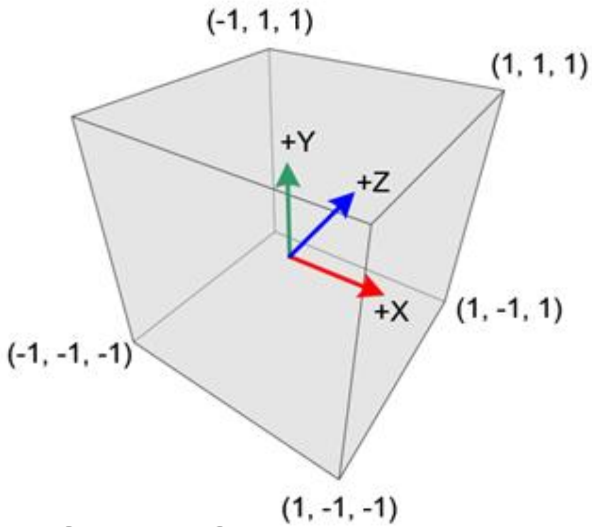
we can only see three faces from six!



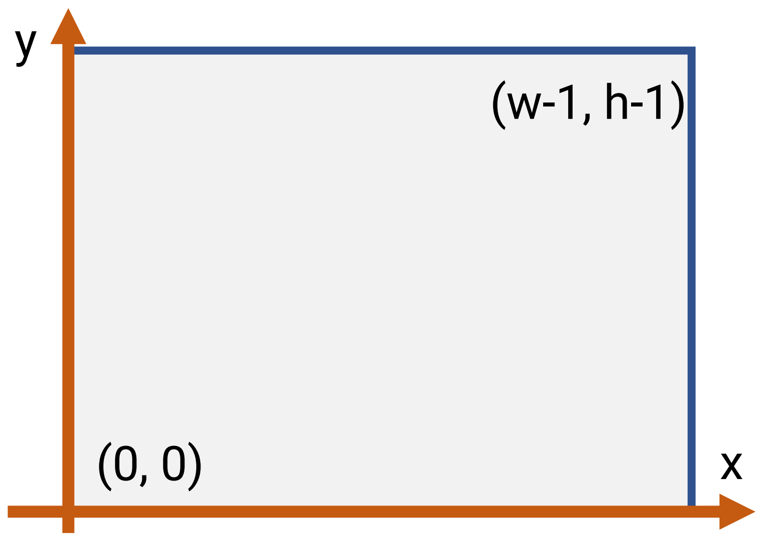
How about this case?

Primitive Assembly (cont.)

- **Screen mapping (OpenGL will handle this!)**



OpenGL NDC

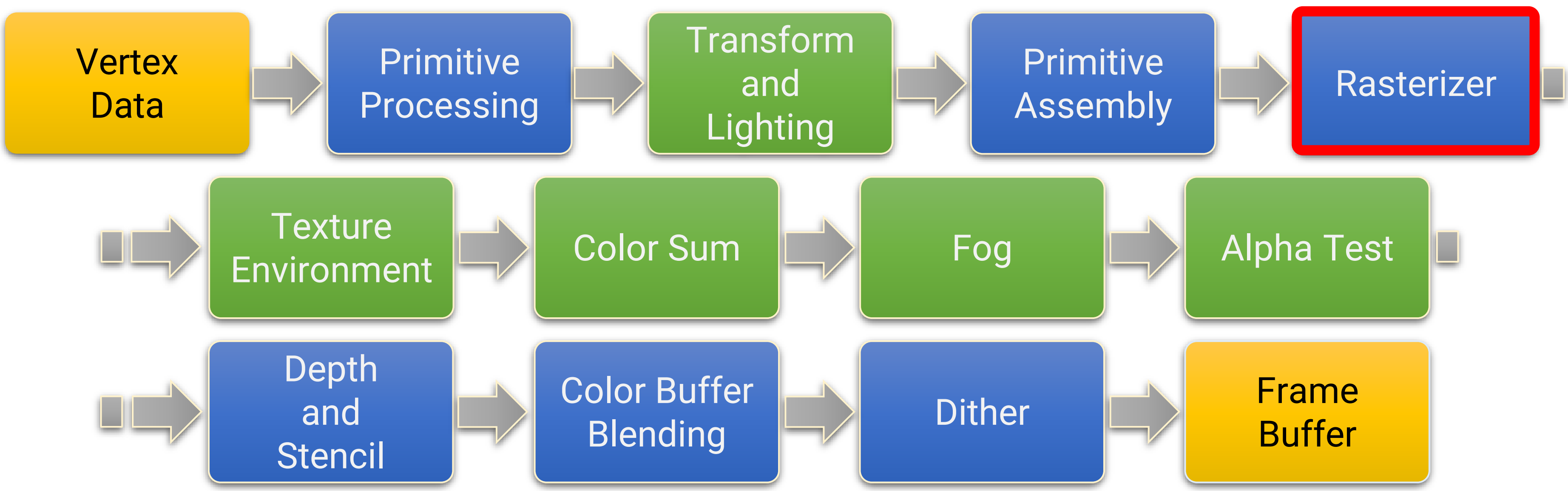


OpenGL Screen Space

$$x_s = w(x_{ndc} + 1)/2$$
$$y_s = h(y_{ndc} + 1)/2$$
$$z_s = (z_{ndc} + 1)/2$$
$$w_s = w_{ndc}$$

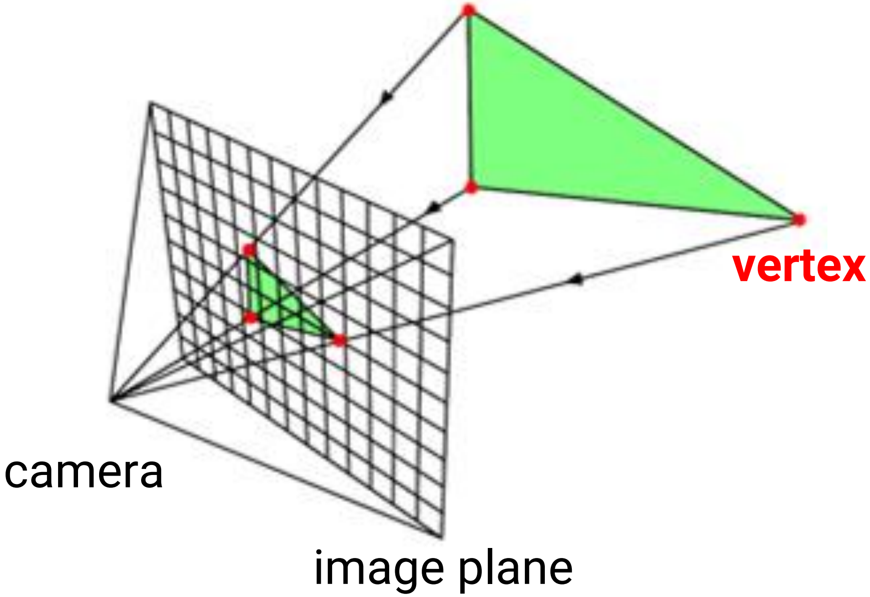
+ screen location

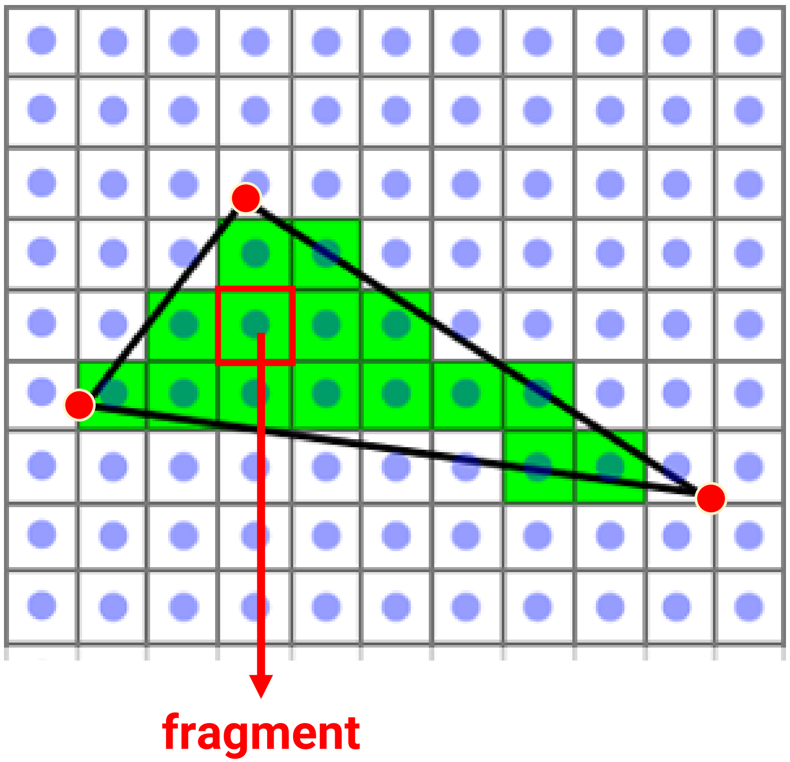
OpenGL (1.x) Fixed Function Pipeline



Rasterization

- Convert **triangles** (**continuous**) into **fragments** (**discrete**)
    - For each pixel that is inside the triangle in the screen space, generate a **fragment**

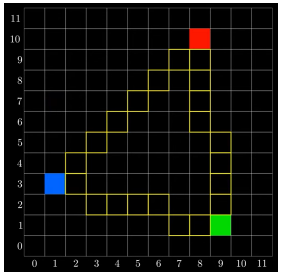


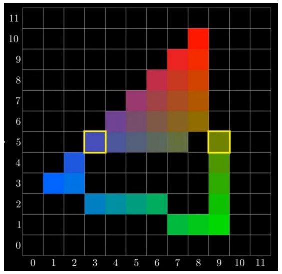


Rasterization (cont.)

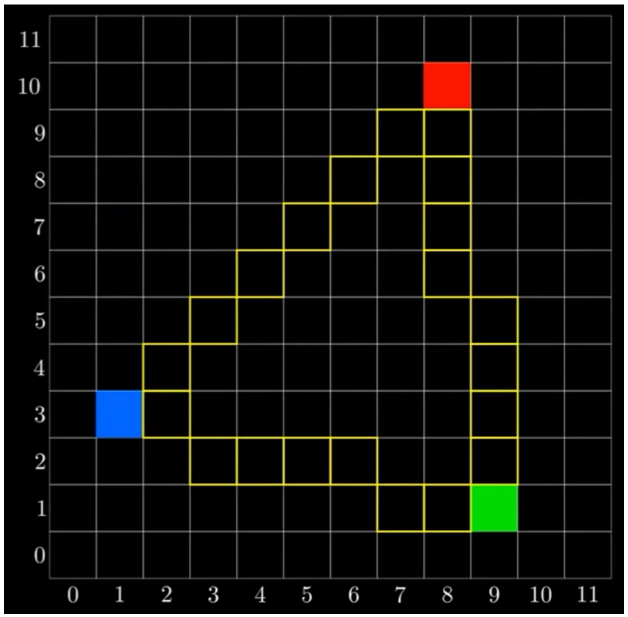
- Convert **triangles** (**continuous**) into **fragments** (**discrete**)
  - For each pixel that is inside the triangle in the screen space, generate a **fragment**
- Obtain **per-fragment data** using **interpolation**: vertex attributes are **interpolated** across the face, such as
  - (Lighting) color — used for per-vertex lighting
  - Texture coordinate
  - Position
  - Normal

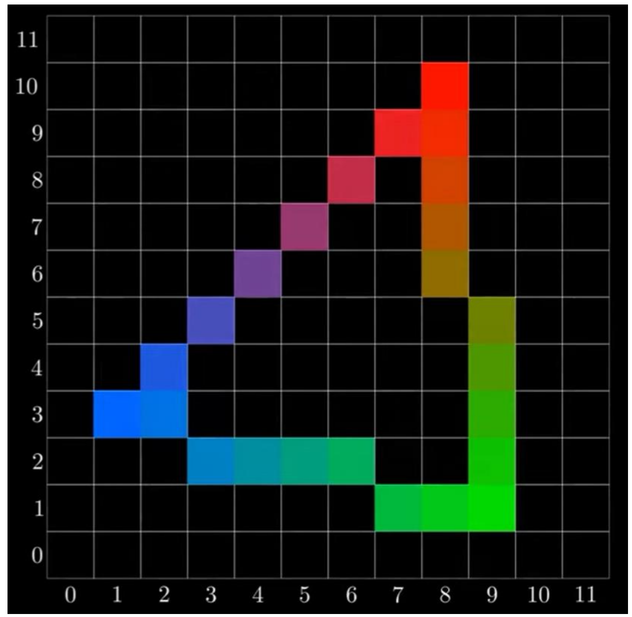
  used for per-fragment lighting (after OpenGL 2.0)
  - Anything you want to interpolate

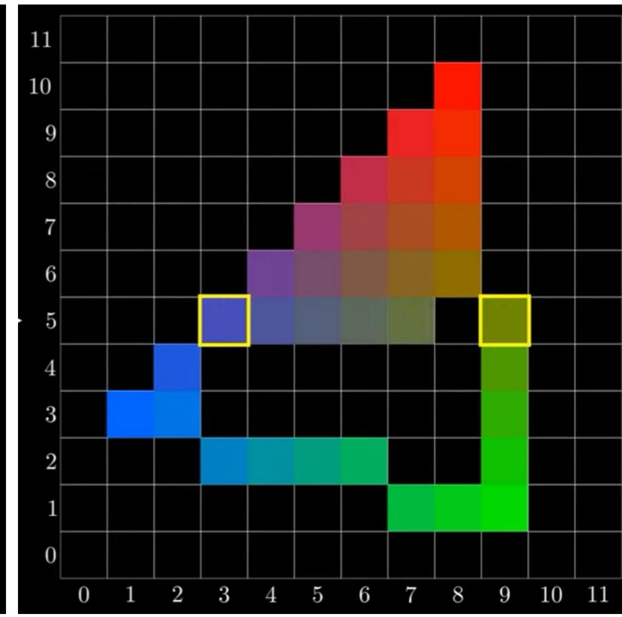




Scanline Rasterization







Find edge pixels
- DDA [link]
- Bresenham [link]

Attributes interpolation of edge pixels using vertices

Attributes interpolation of inner pixels using edge points

Digital Differential Analyzer (DDA)

- Draw a line segment passing through $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (7, 5)$

$$y = mx + b$$

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

slope

$$\Delta y = m\Delta x = m$$ (if Δx = 1)

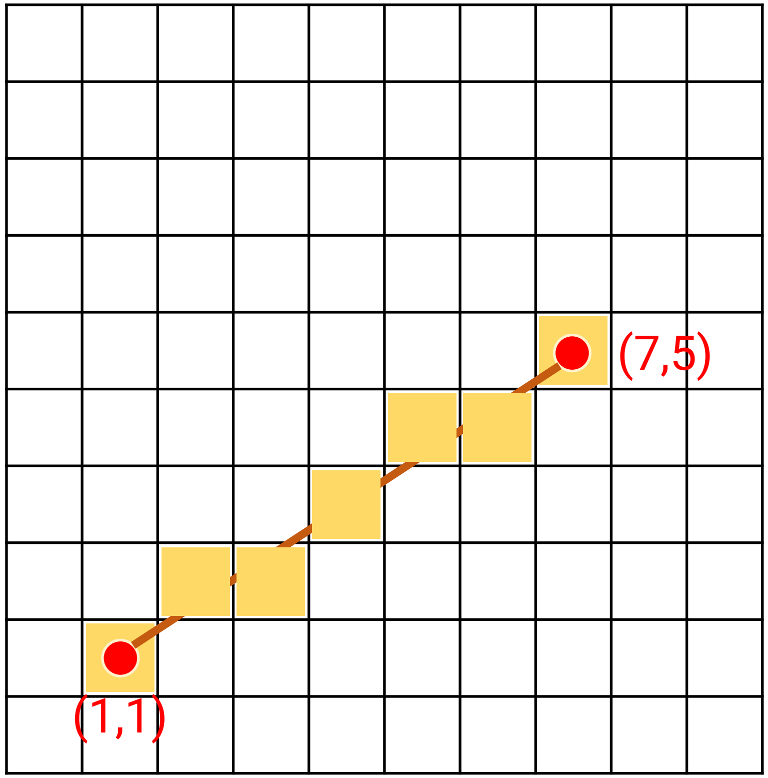
$x_a = 2$ ➔ $y_a = y_1 + m = 1.667$ ➔ (2, 1.667) (2, 2)

$x_b = 3$ ➔ $y_b = y_a + m = 2.333$ ➔ (3, 2.333) (3, 2)

$x_c = 4$ ➔ $y_c = y_b + m = 3.000$ ➔ (4, 3.000) (4, 3)

$x_d = 5$ ➔ $y_d = y_c + m = 3.667$ ➔ (5, 3.667) (5, 4)

$x_e = 6$ ➔ $y_e = y_d + m = 4.333$ ➔ (6, 4.333) (6, 4)



floating-point addition / comparison

Bresenham Algorithm

- Draw a line segment passing through $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (7, 5)$

$$y = mx + b$$

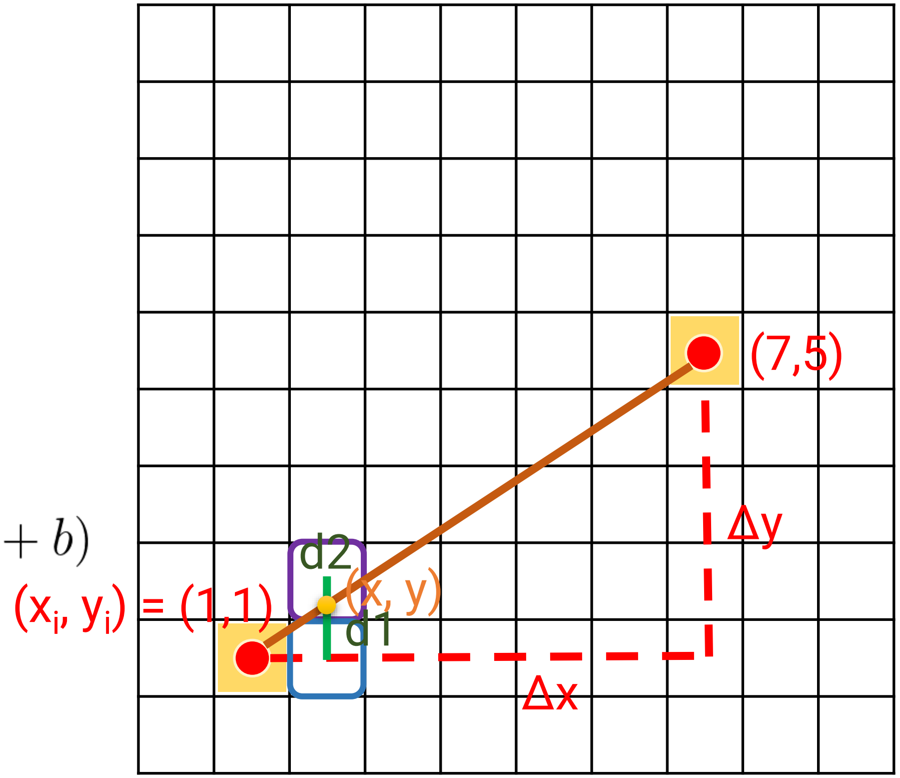
$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

$$d1 = y - y_i = (m(x_i + 1) + b) - y_i$$

$$d2 = (y_i + 1) - y \ = y_i + 1 - (m(x_i + 1) + b)$$

$$d1 - d2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

$$\Delta x(d1 - d2) = 2\Delta y x_i - 2\Delta x y_i + c$$



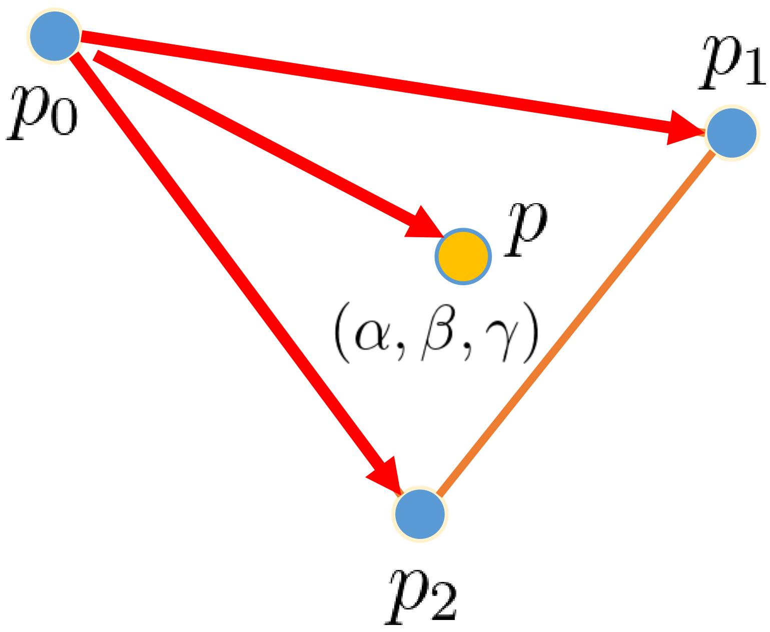
integer multiplication / comparison

Scanline Rasterization (cont.)

- https://www.youtube.com/watch?v=t7Ztio8cwqM

Barycentric Coordinates

- Barycentric coordinates inside a triangle



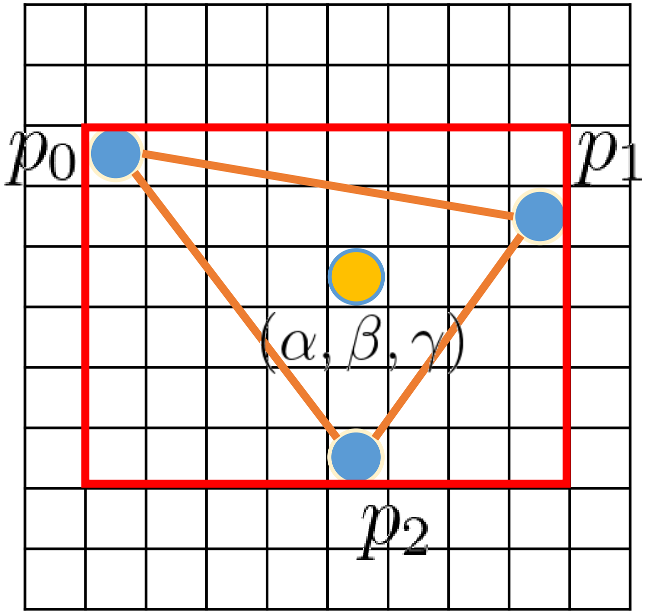
$$
\begin{aligned}
p &= p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0) \\
&= (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2 \\
&= \alpha p_0 + \beta p_1 + \gamma p_2
\end{aligned}
$$

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

The values $\alpha, \beta, \gamma \in [0, 1]$ **if and only if *p* is inside the triangle**

Barycentric Coordinates (cont.)

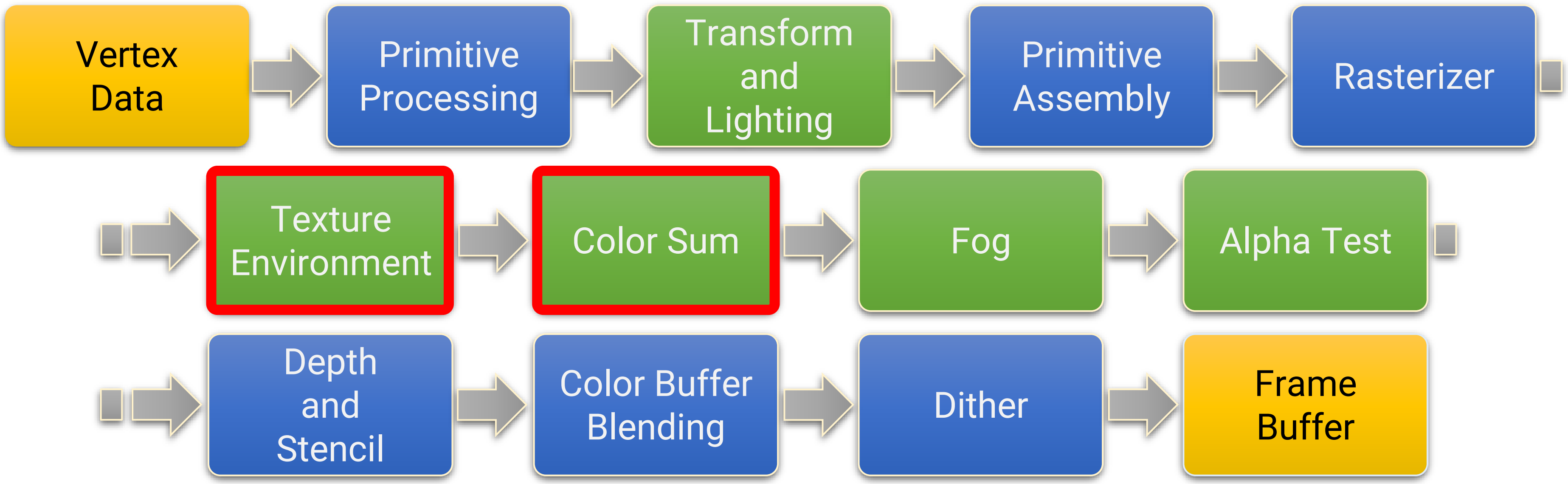
- Compute the 2D bounding box of the 2D triangle
- For each pixel inside the bounding box, compute its barycentric coordinates [link]
- If the coordinates are all ≥ 0 and ≤ 1, the pixel is covered by the triangle



The barycentric coordinates *α, β, γ* can be used to interpolate vertex attributes directly

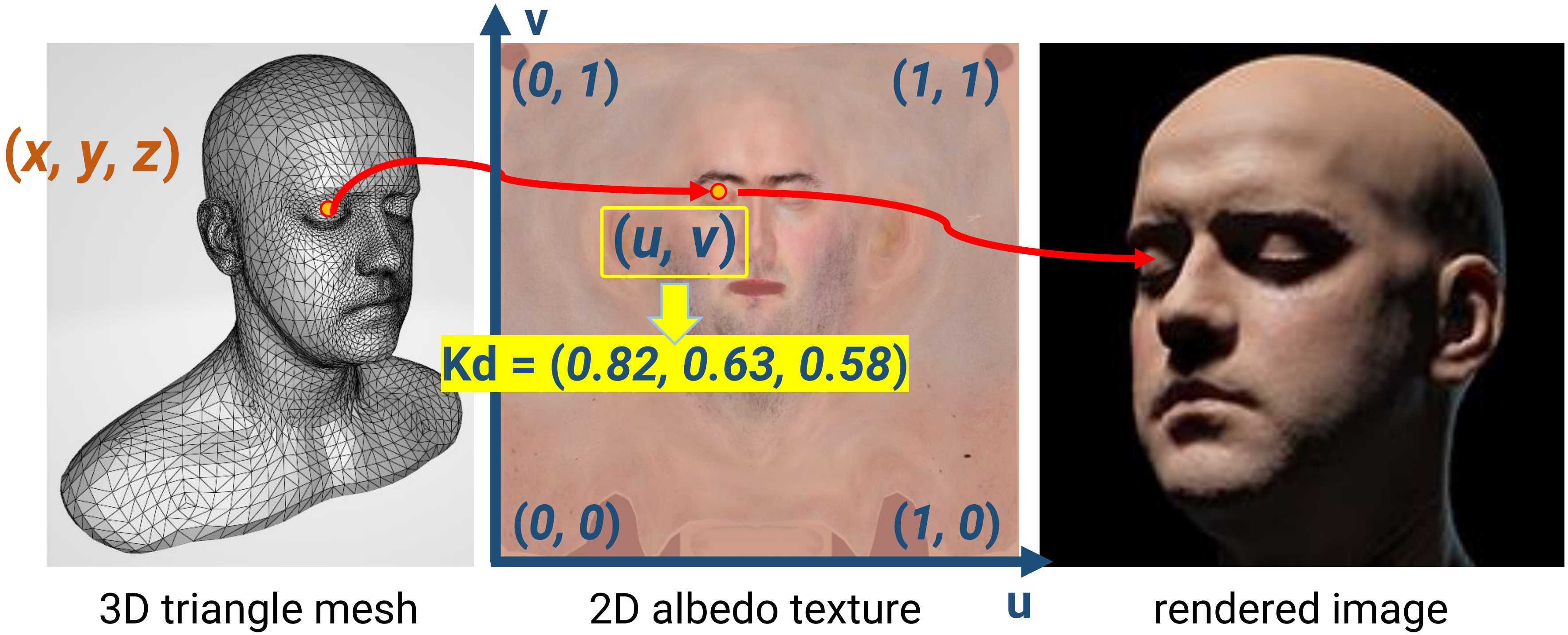
$$\begin{aligned} p &= p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0) \\ &= \alpha p_0 + \beta p_1 + \gamma p_2 \end{aligned}$$

OpenGL (1.x) Fixed Function Pipeline



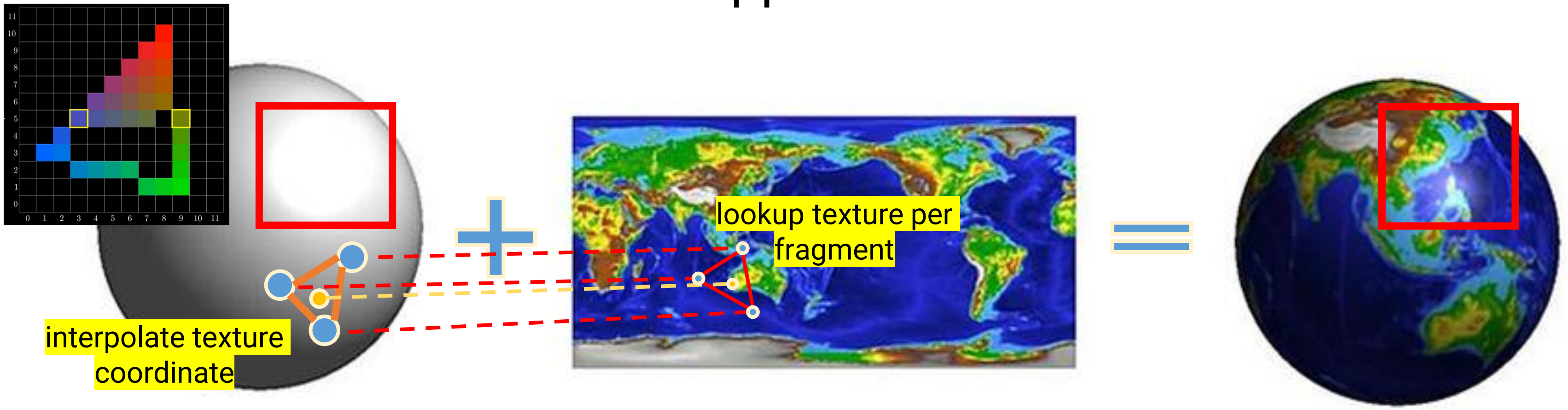
Recap: Texturing

- Textures are used to describe complex materials
- A (vertex) texture coordinate is used to look up the texture
  - The way to map a point on the 3D surface to a pixel (texel) on a 2D image texture



Texture Environment and Color Sum

- **Texture Environment**
  - Apply the textures to the fragments
- **Color Sum**
  - Used to add-in a secondary color to the geometry after the textures have been applied

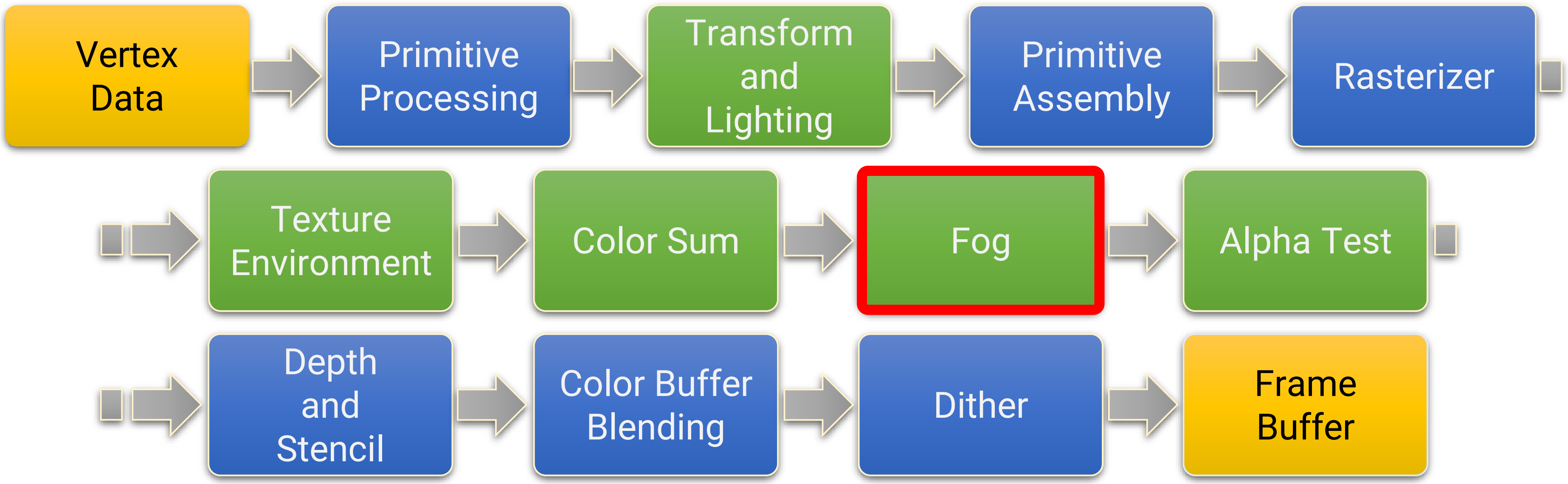


3D object

texture

texture mapped object
(color = lighting × texture)

OpenGL (1.x) Fixed Function Pipeline



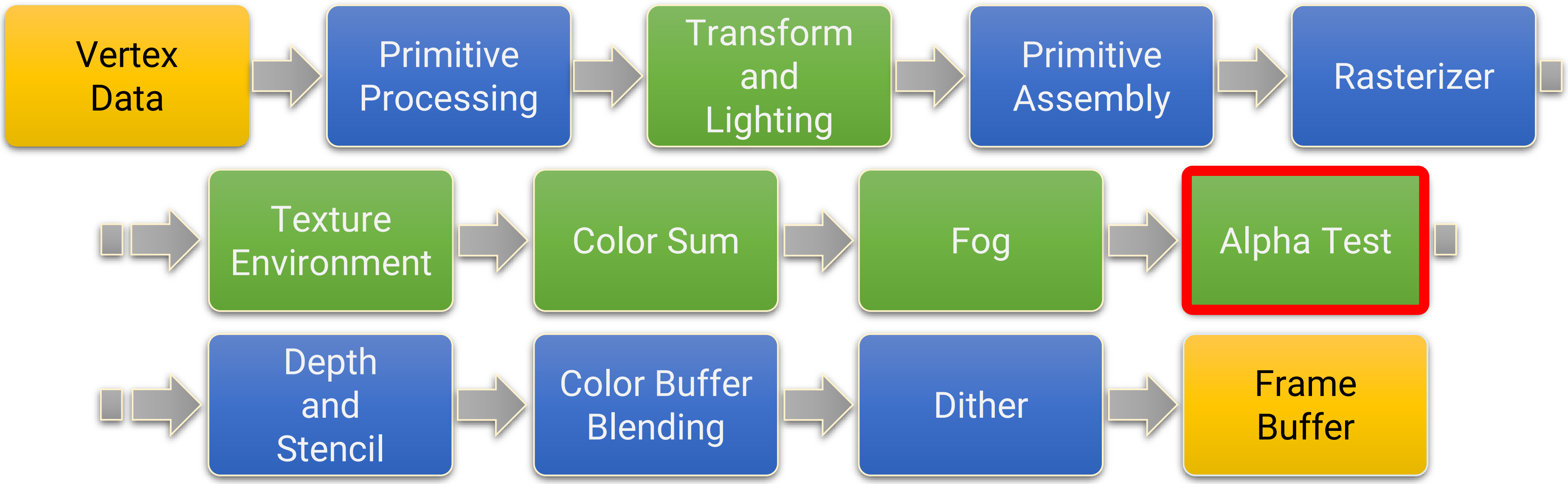
Fog

- Simulate the effect of geometry fadeout as dimmed by fog
- Blend the **fragment color** with the **fog color** according to **object distance**

Fog in Games

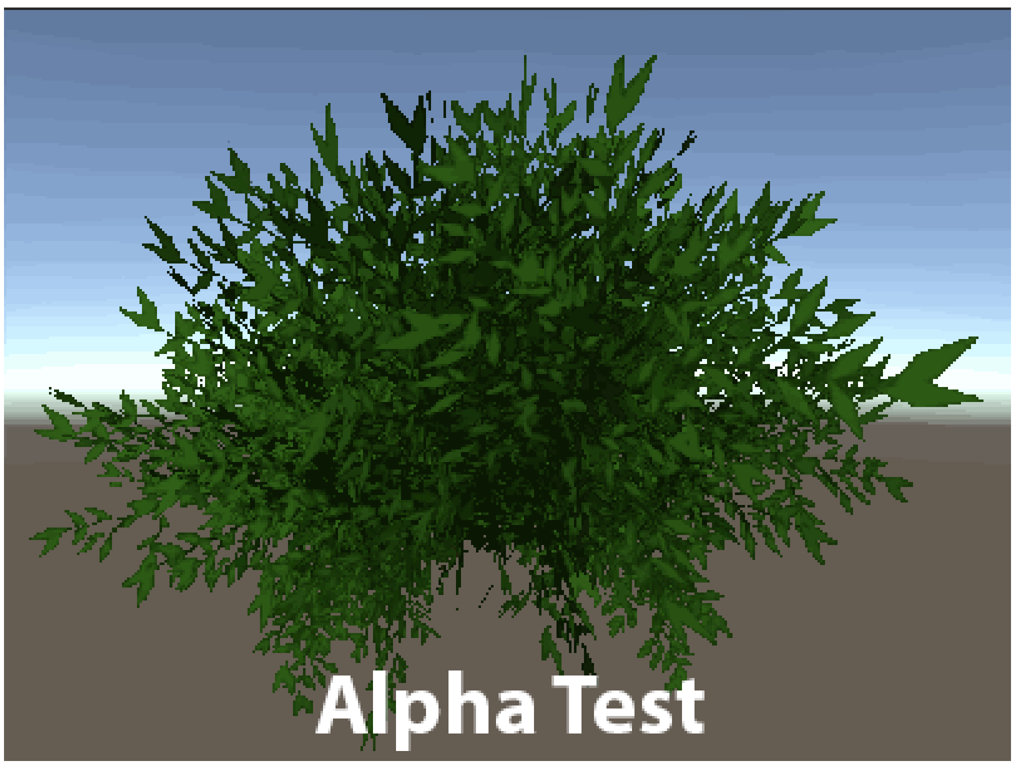
Fog in Games (cont.)

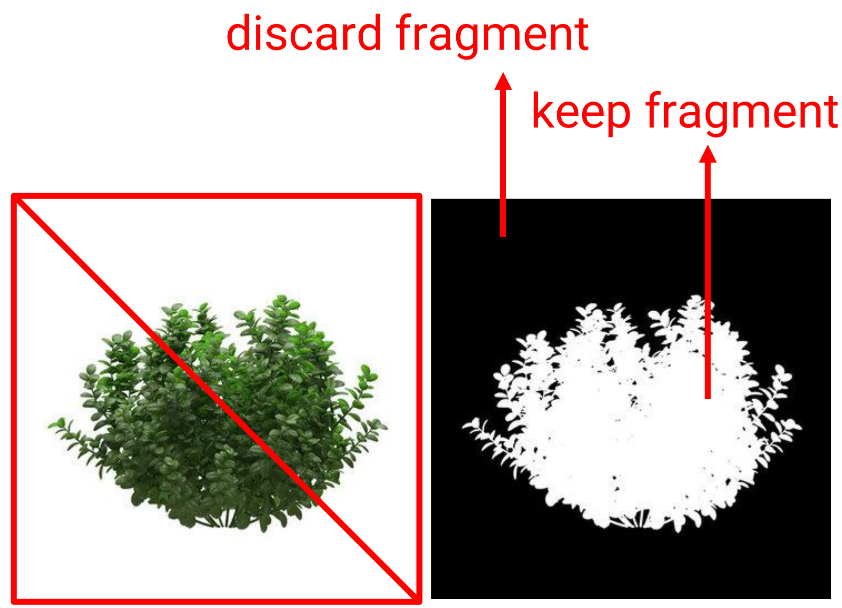
OpenGL (1.x) Fixed Function Pipeline



Alpha Test

- Discard fragments if their alpha values are below a certain threshold



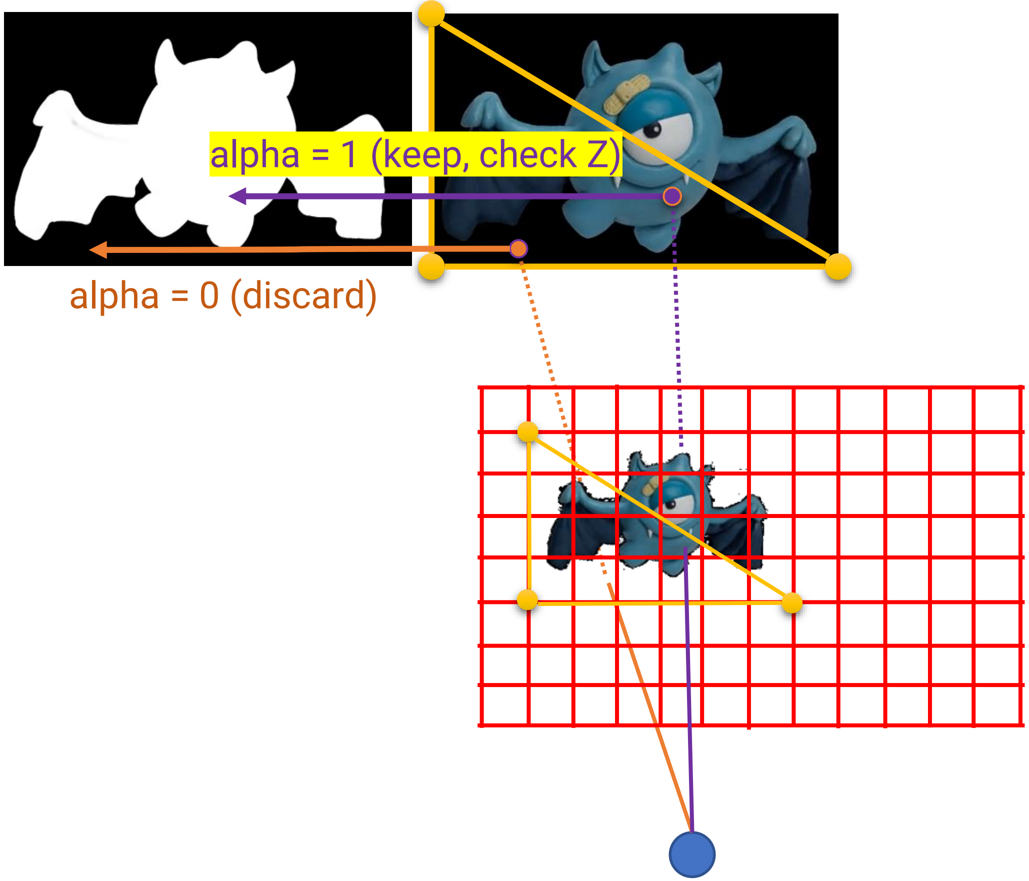


albedo texture alpha mask

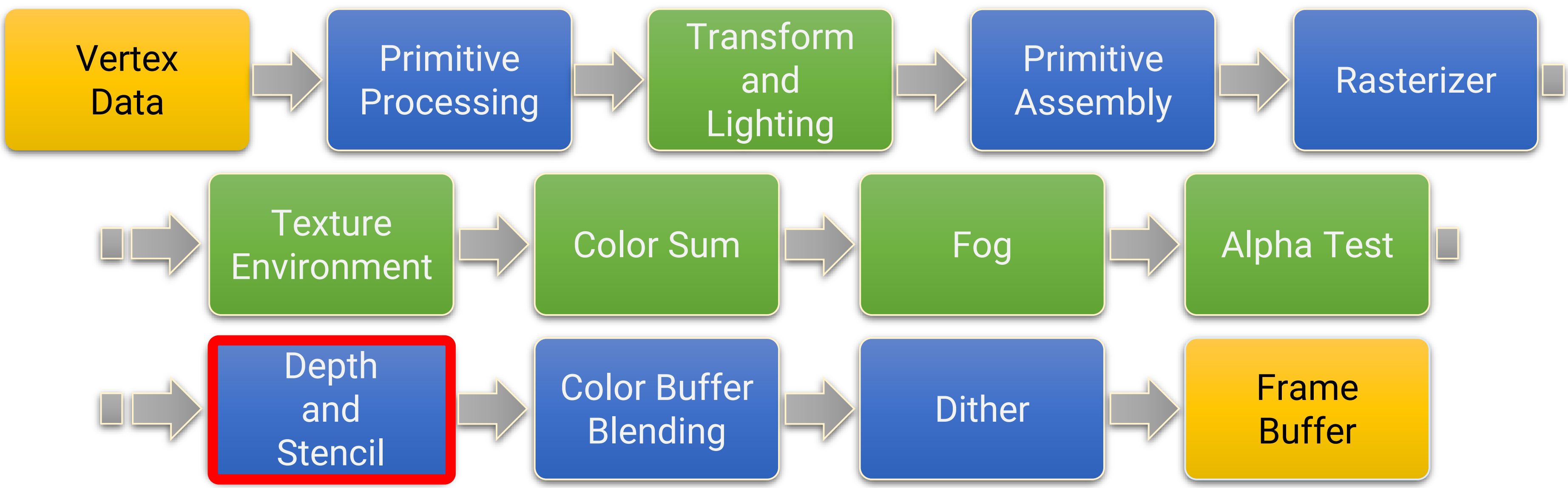
(usually combined in an RGBA texture)

Alpha Test (cont.)

- Discard fragments if their alpha values are below a certain threshold

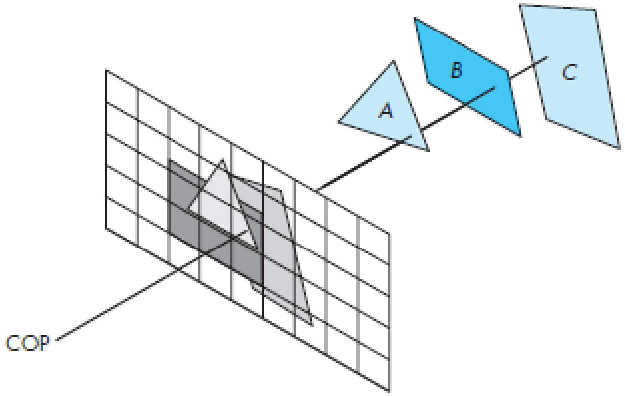


OpenGL (1.x) Fixed Function Pipeline



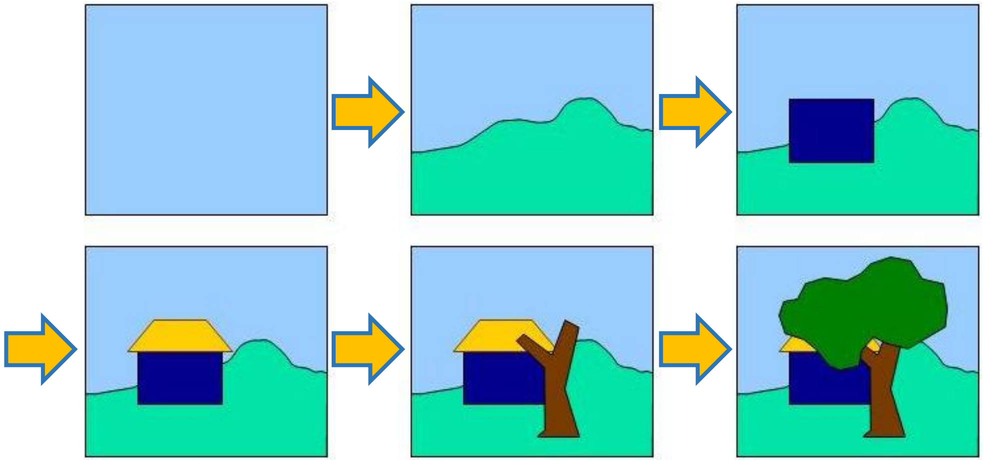
Depth Test

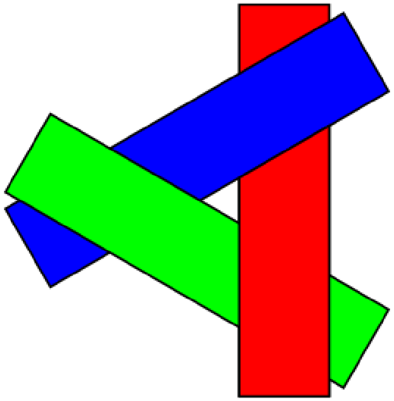
- Used for **hidden surface removal**
  - Only show the **closest** surfaces to the camera at each pixel



Depth Test (cont.)

- Used for **hidden surface removal**
  - Only show the **closest** surfaces to the camera at each pixel
- Earlier approach: painter's algorithm

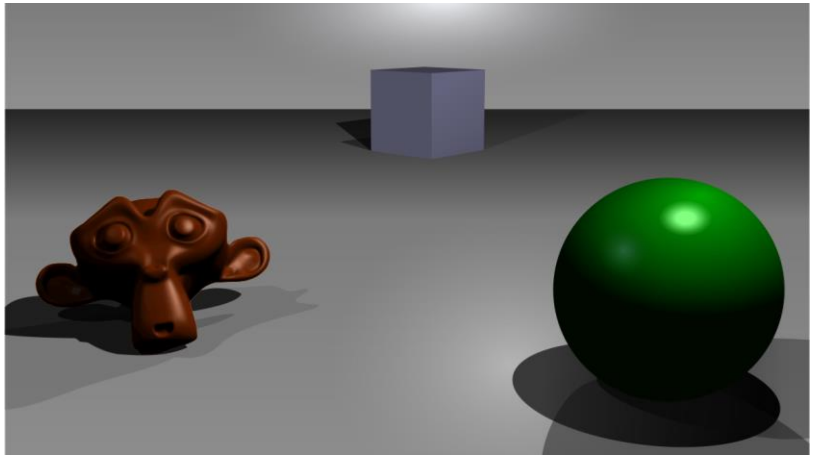




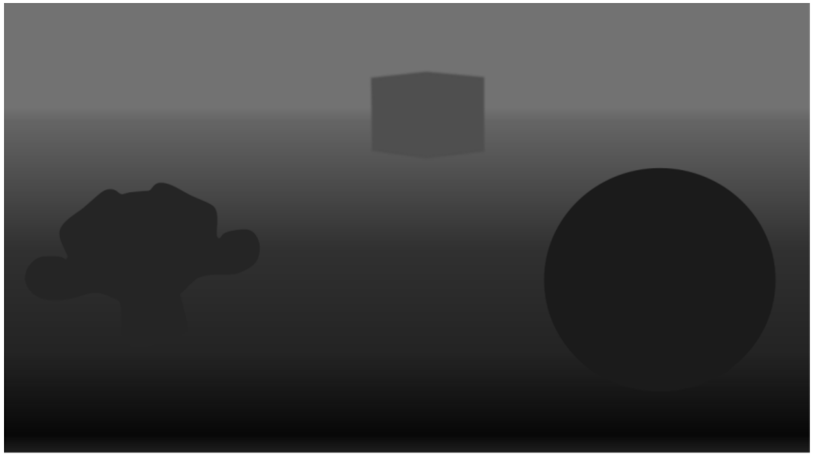
How about this?
(cyclical overlapping)

Depth Test (cont.)

- **Z-buffer**
  - An additional buffer used to maintain the z value of the closest surface to a pixel
  - Discard fragments if they have larger depth values than the ones stored in their corresponding positions in the Z buffer



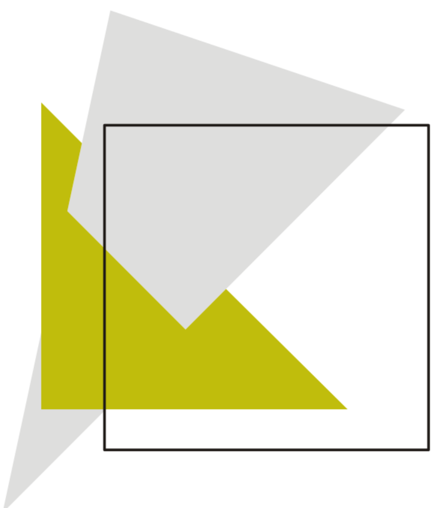
color frame buffer



Z (depth) buffer

Z-Buffer

- Z-buffer update



∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

+

5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

=

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

+

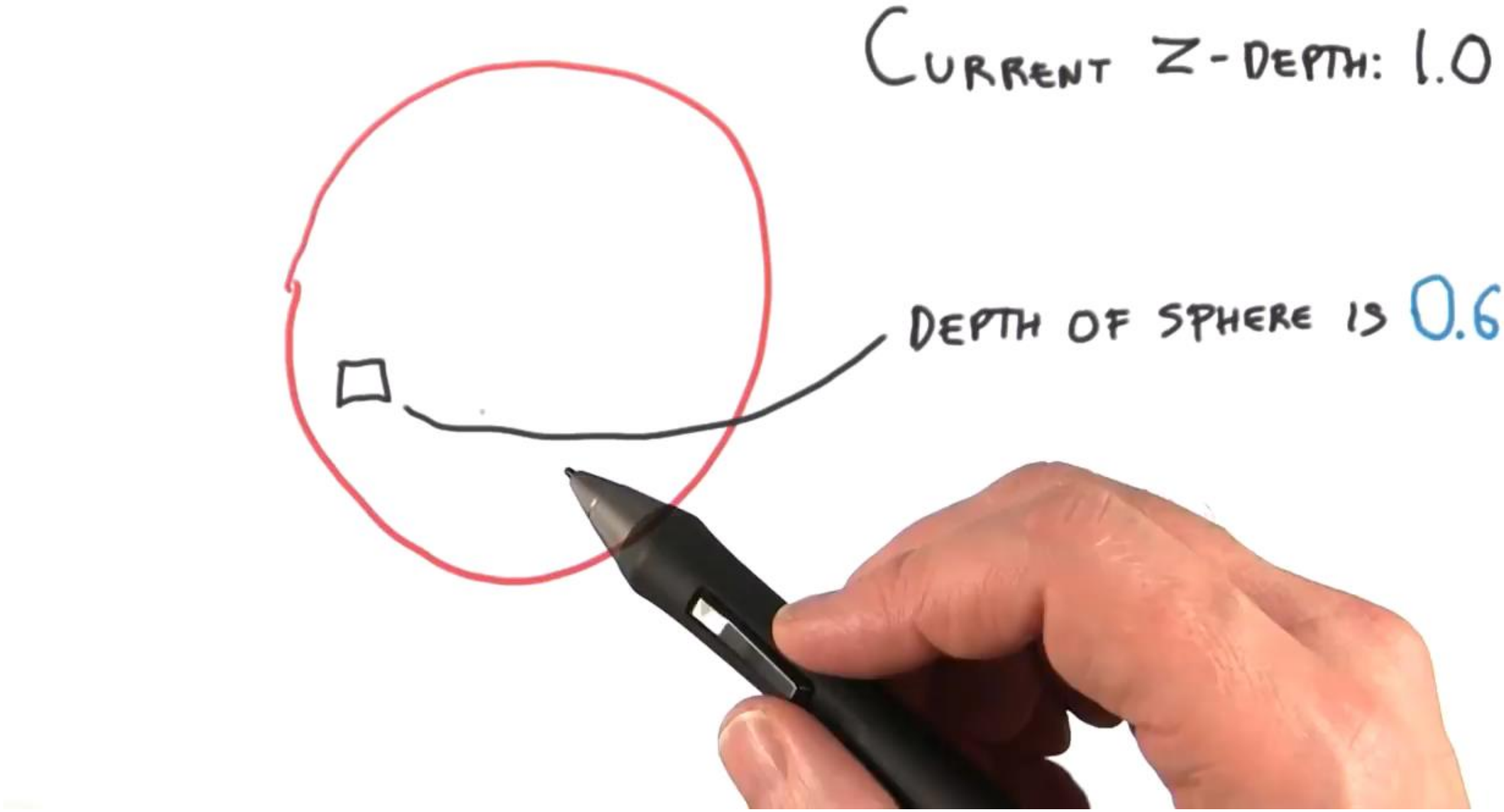
7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

=

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

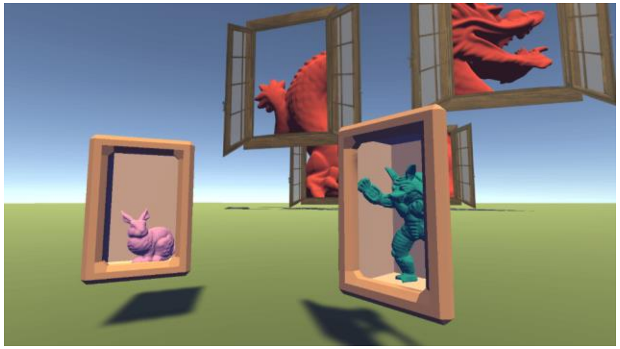
Z-Buffer (cont.)

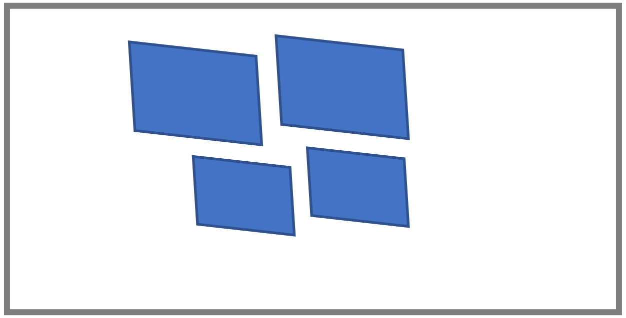
- https://www.youtube.com/watch?v=yhwg_O5HBwQ



Stencil Test

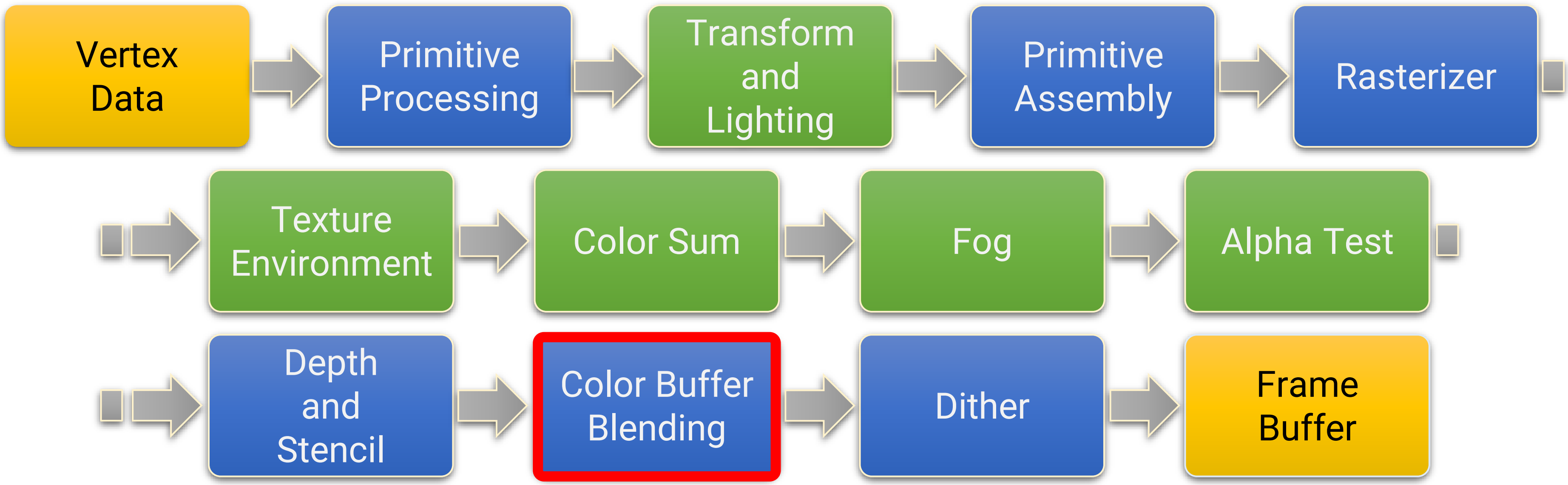
- Used to discard fragments that fail a stencil comparison, based on the content of the stencil buffer





stencil buffer

OpenGL (1.x) Fixed Function Pipeline



Color Buffer Blending

- Blend the color of fragments with the previous results in the frame buffer based on the **alpha values** of the current fragments, as well as the **blend function** and the **blend equations**

Color Buffer Blending (cont.)

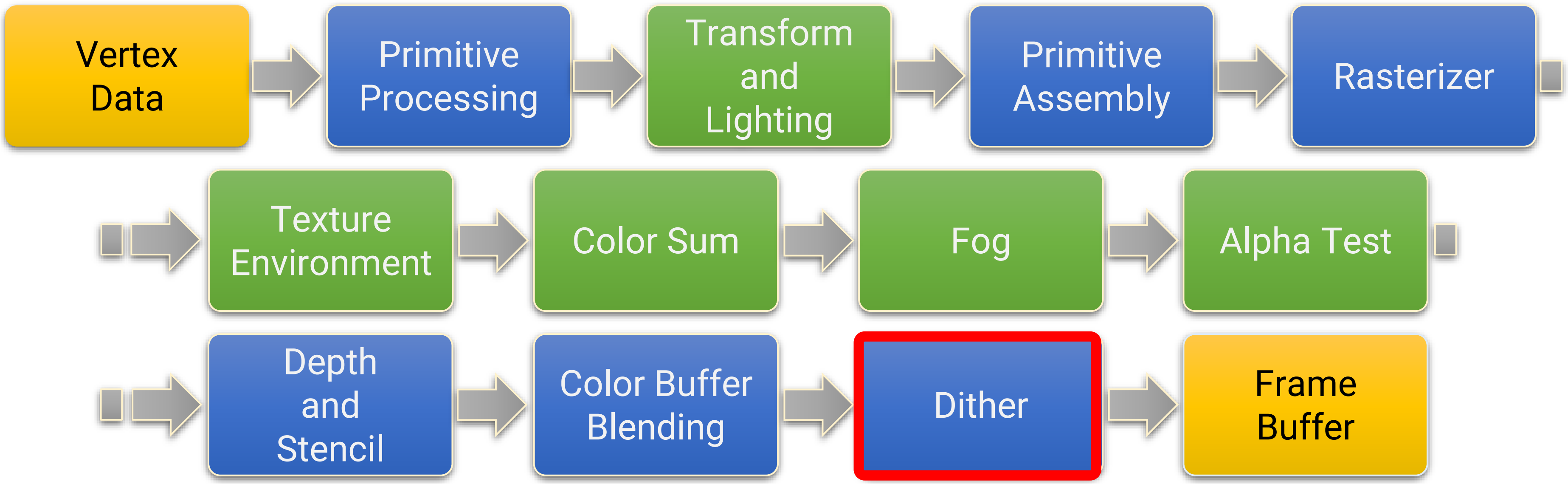
- Blend the color of fragments with the previous results in the frame buffer based on the **alpha values** of the current fragments, as well as the **blend function** and the **blend equations**

$$\bar{C}_{result} = \bar{C}_{source} * \boxed{F_{source}} + \bar{C}_{destination} * \boxed{F_{destination}}$$

GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA

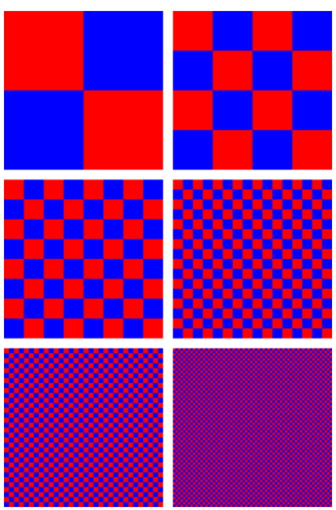
opaque objects:
Z buffer will handle the occlusion

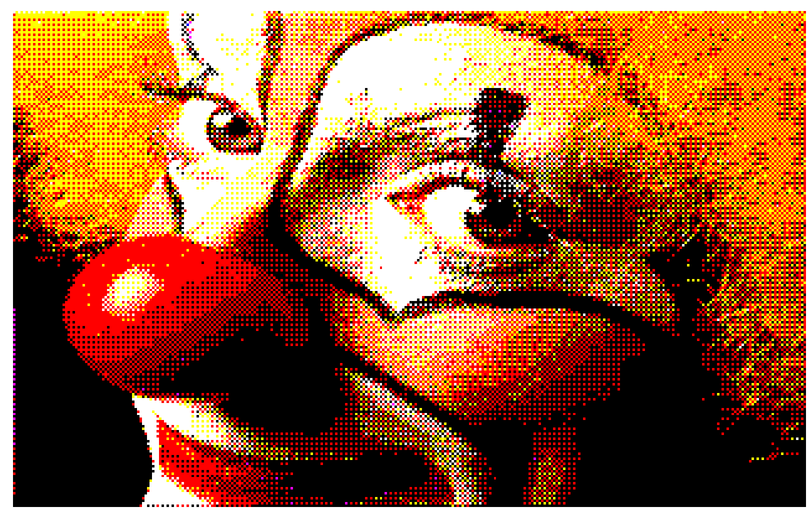
OpenGL (1.x) Fixed Function Pipeline



Dither

- If a color palette is used, OpenGL will try to simulate a larger color palette by mixing colors in close proximity
- Areas of a single color are replaced by a pattern of dots of several different colors, in such a way that optical mixing in the eye produces a color close to the desired one





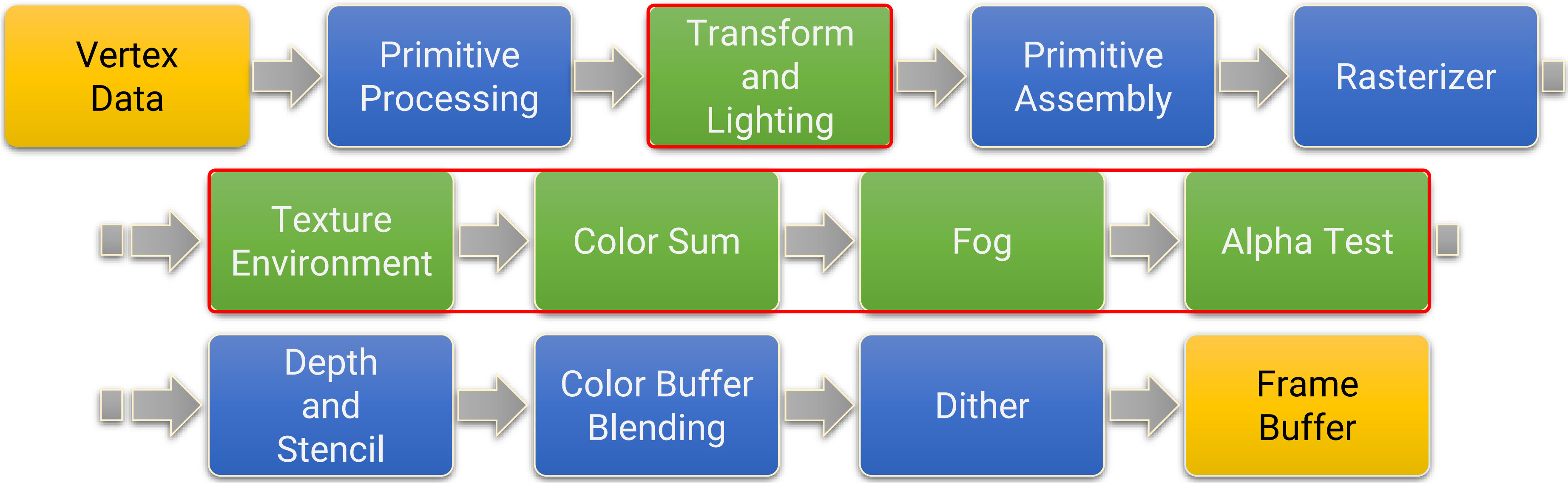
Summary of Fixed Function Pipeline

- An 3D object will come to the screen with a series of "fixed" steps
  - Fixed transformation (MVP matrix)
    - How about user interface on the screen?
  - **Fixed (Phong) lighting model on vertices**
  - **Fixed modulation of lighting color and texture color**
    - Color = Lighting $\times$ Texture

- **We would like more flexibility!**
- In Part II, we will introduce how to add **flexibility** by using **shaders**

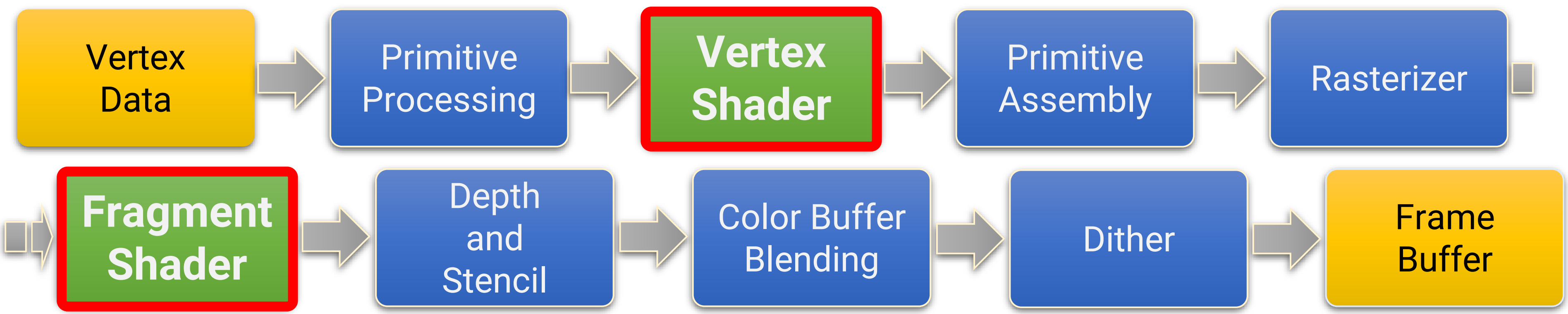
Spoiler: OpenGL 1.x to OpenGL 2.0

- All the functions performed by OpenGL are **fixed** and **could not** be modified except through the manipulation of the **rendering states**
- The stages shown in green have been replaced by **shaders**



Spoiler: OpenGL 1.x to OpenGL 2.0 (cont.)

- Released in 2004
- Provide the ability to **programmatically** define the vertex transformation and lighting and the fragment operations (with small GPU programs called **shaders**)




END OF PRESENTATION
ANY QUESTIONS?