GPU Graphics Pipeline (Part I)

Computer Graphics

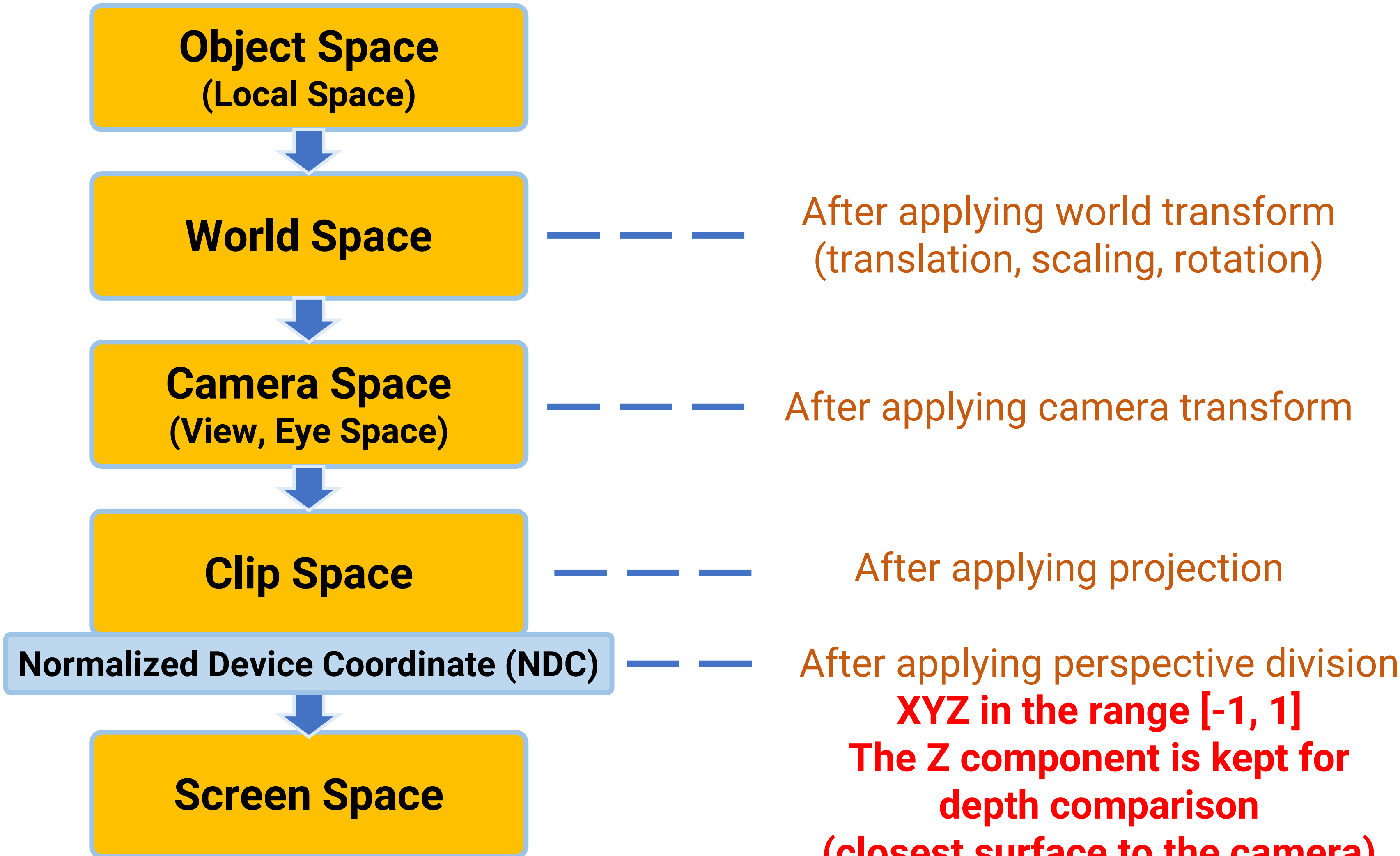
Yu-Ting Wu

Outline

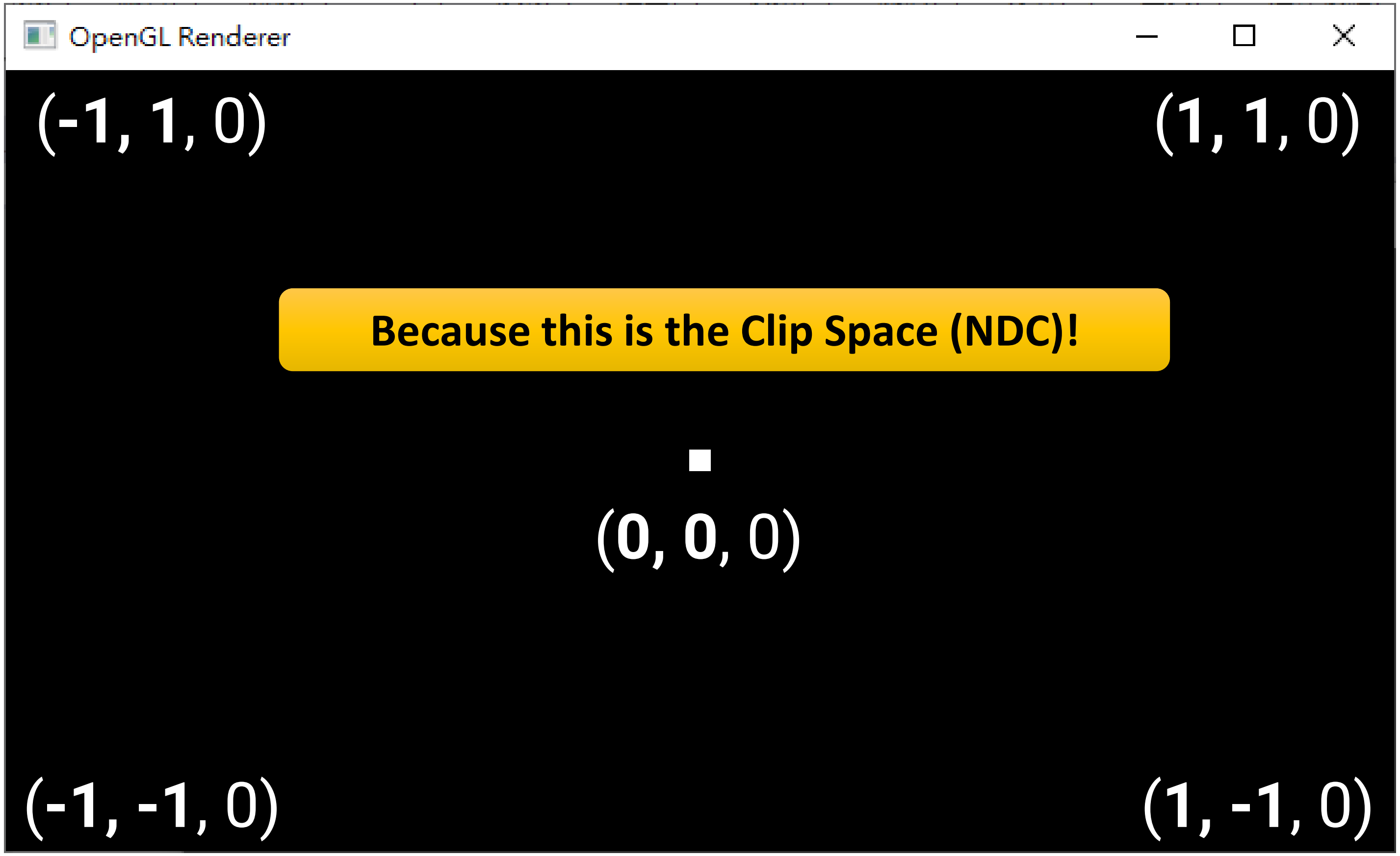
- GPU graphics pipeline
- OpenGL graphics pipeline 1.x (Part I)
- OpenGL graphics pipeline 2.0

- OpenGL and shader implementation (Part II)

Recap.

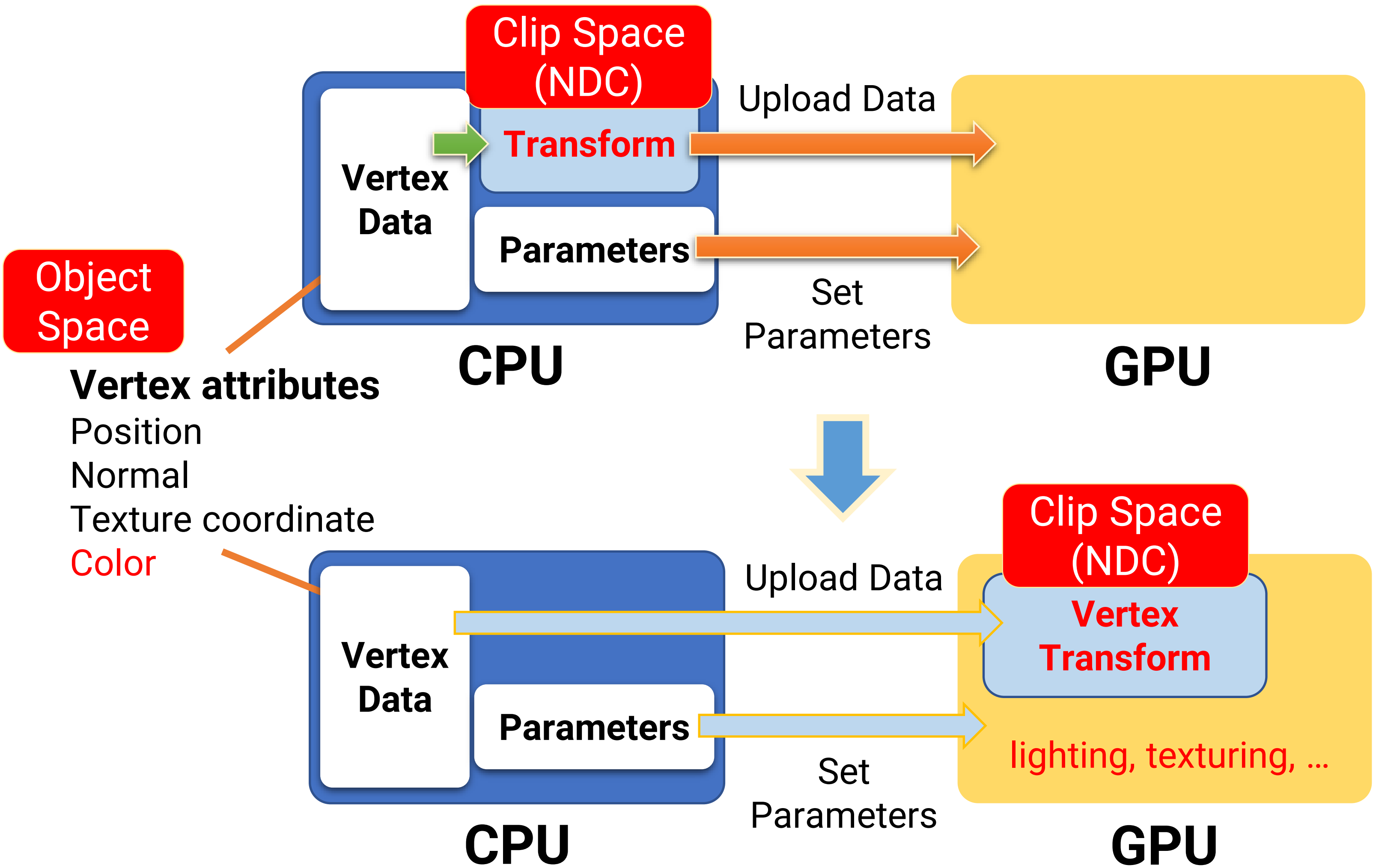

Object Space
(Local Space)
World Space
After applying world transform
(translation, scaling, rotation)
Camera Space
(View, Eye Space)
After applying camera transform
Clip Space
After applying projection
Normalized Device Coordinate (NDC)
After applying perspective division
XYZ in the range [-1, 1]
The Z component is kept for
depth comparison
(closest surface to the camera)
Screen Space

Recap. (cont.)



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

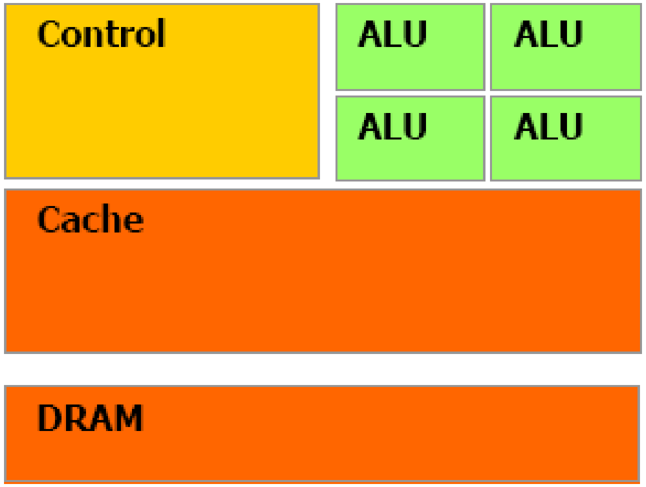
Recap. (cont.)


Clip Space
(NDC)
Upload Data
Transform
Vertex
Data
Parameters
Object
Space
Set
Parameters
CPU
GPU
Vertex attributes
Position
Normal
Texture coordinate
Color
Upload Data
Clip Space
(NDC)
Vertex
Transform
Vertex
Data
Parameters
lighting, texturing, …
Set
Parameters
CPU
GPU

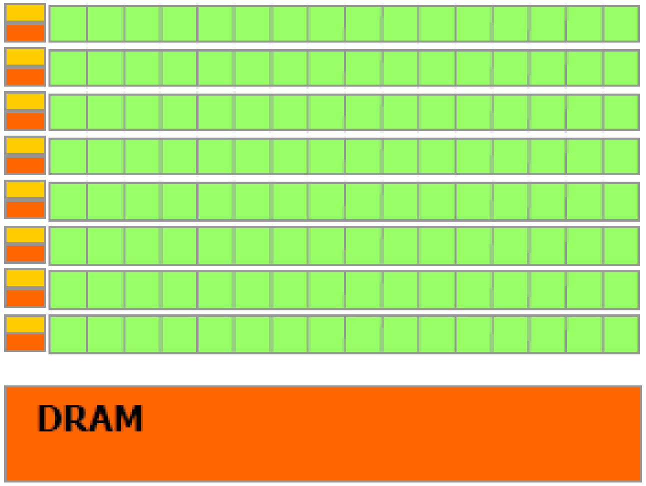
Outline

- **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

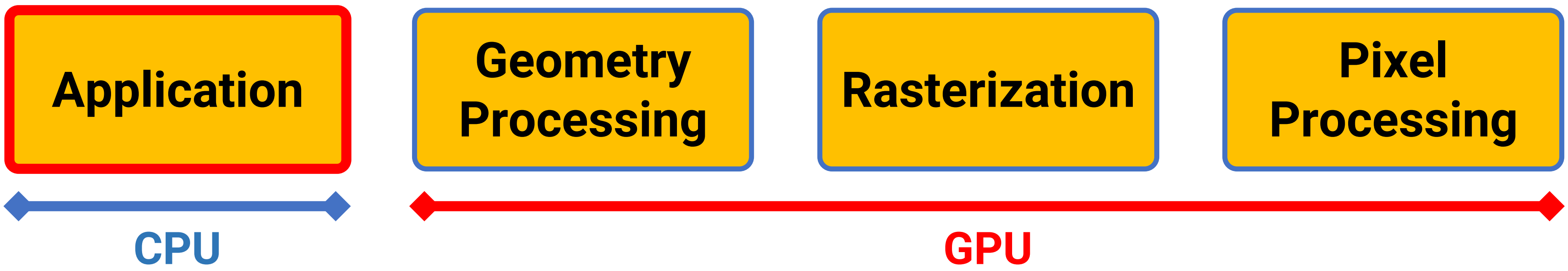
Good at

- Parallel processing
- SIMD
- Higher throughput

Pipeline

GPU Graphics Pipeline Overview

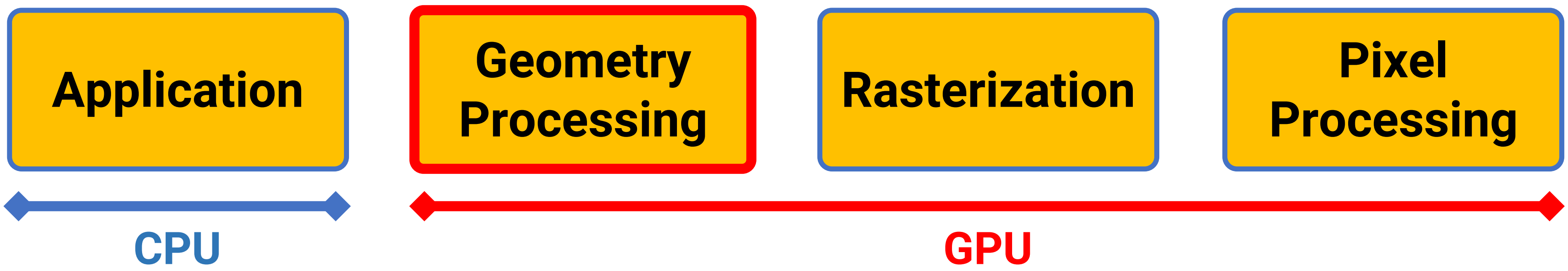
- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 3 stages



- 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 4 stages



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

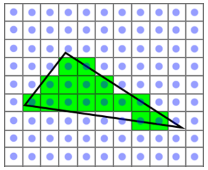
GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages

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

CPU: Application
GPU: Geometry Processing, Rasterization, Pixel Processing

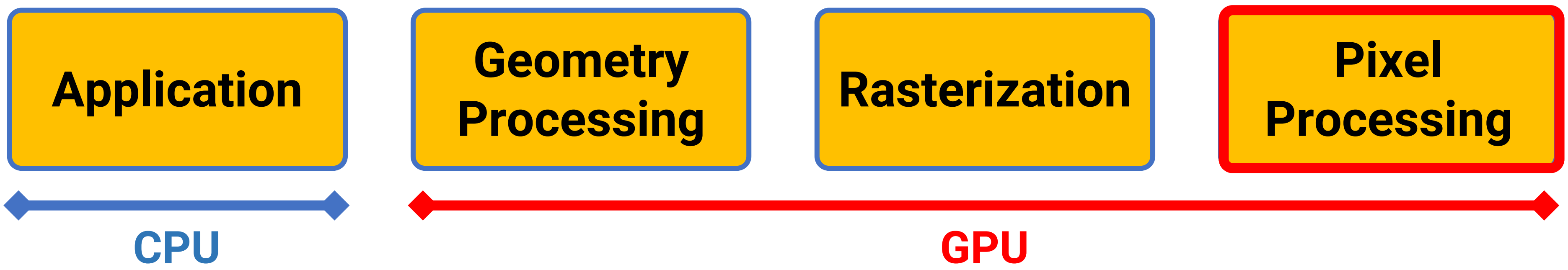
- Triangle setup
- Fragments (pixels) generation
  - Attribute interpolation



from continuous to discrete

GPU Graphics Pipeline Overview (cont.)

- Responsible for the fixed routines of bringing triangles to pixels
- Can be roughly categorized into 4 stages



- Pixel shading / Texturing
- Depth testing
- Alpha blending

GPU Graphics Pipeline Overview (cont.)

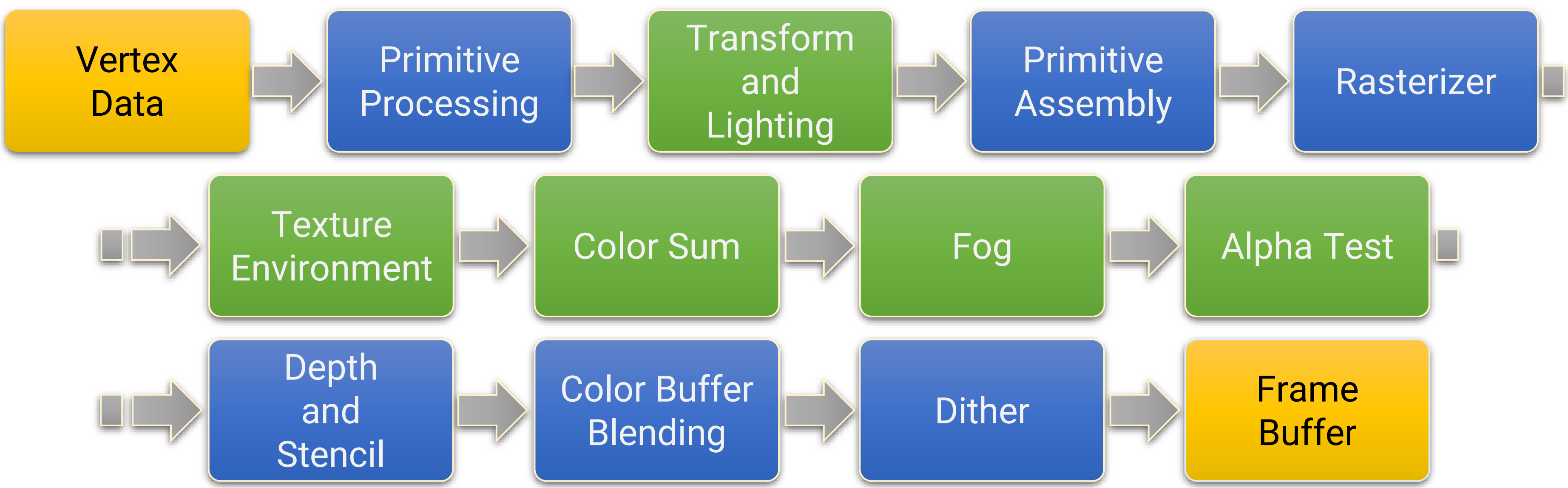
- In the slides, we will first introduce the GPU rendering pipeline revealed in **OpenGL 1.x**
- After that, we will show why (and how) some stages become **programmable** in **OpenGL 2.0**

Outline

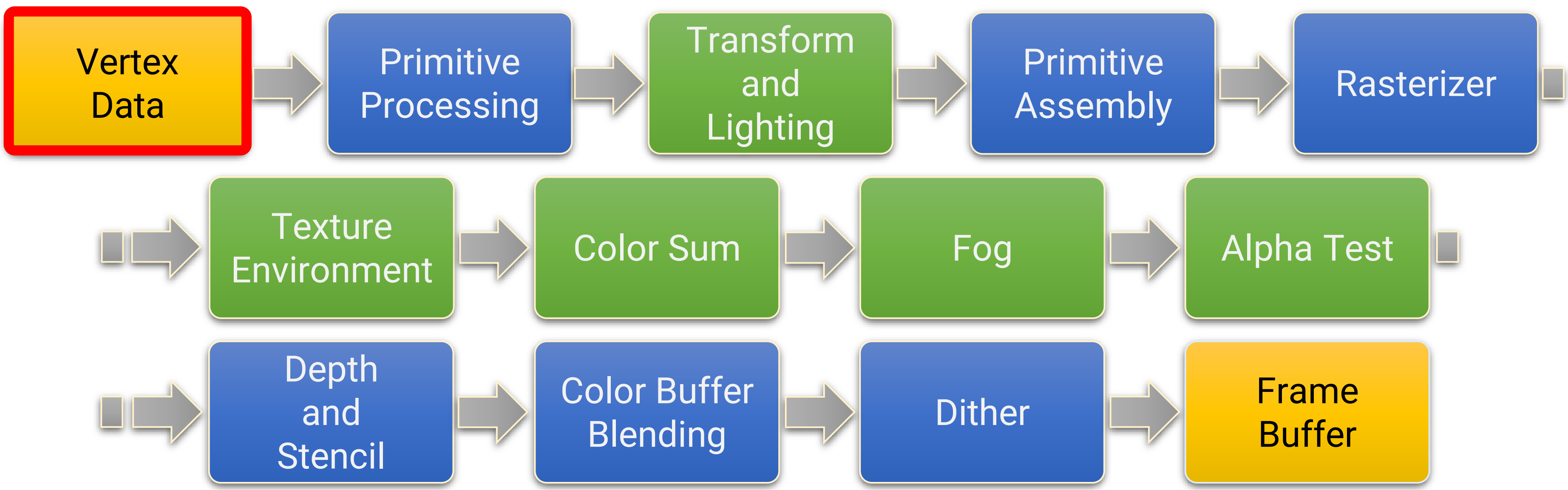
- GPU graphics pipeline
- **OpenGL graphics pipeline 1.x**
- OpenGL graphics pipeline 2.0
- OpenGL and shader implementation

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**



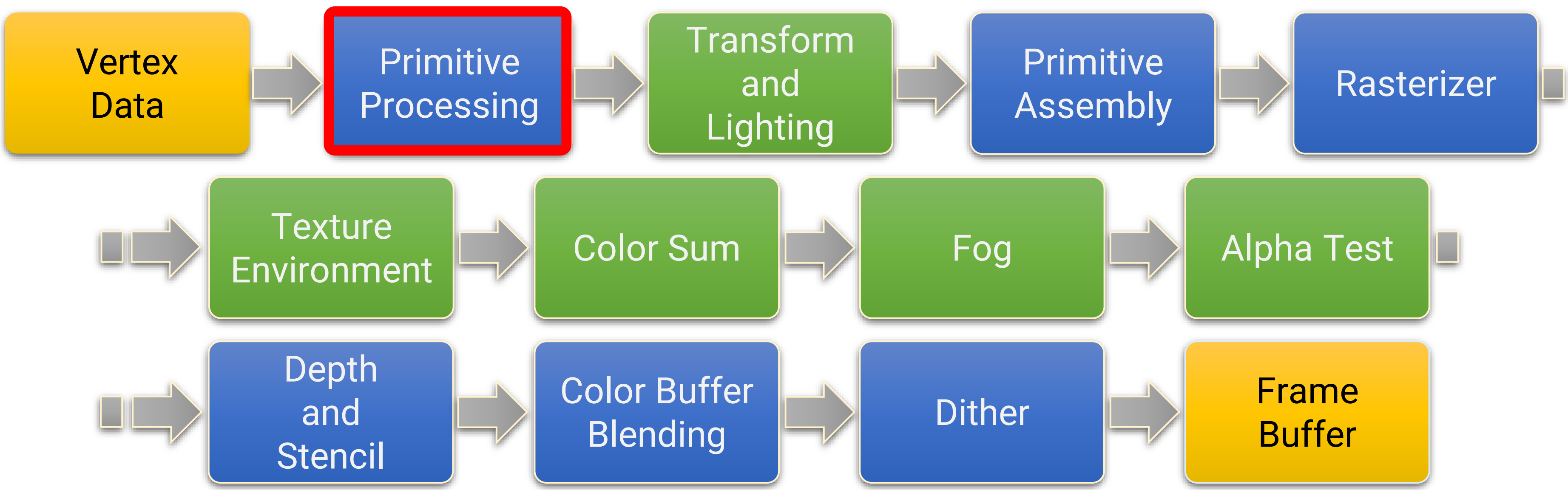
OpenGL (1.x) Fixed Function Pipeline



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**

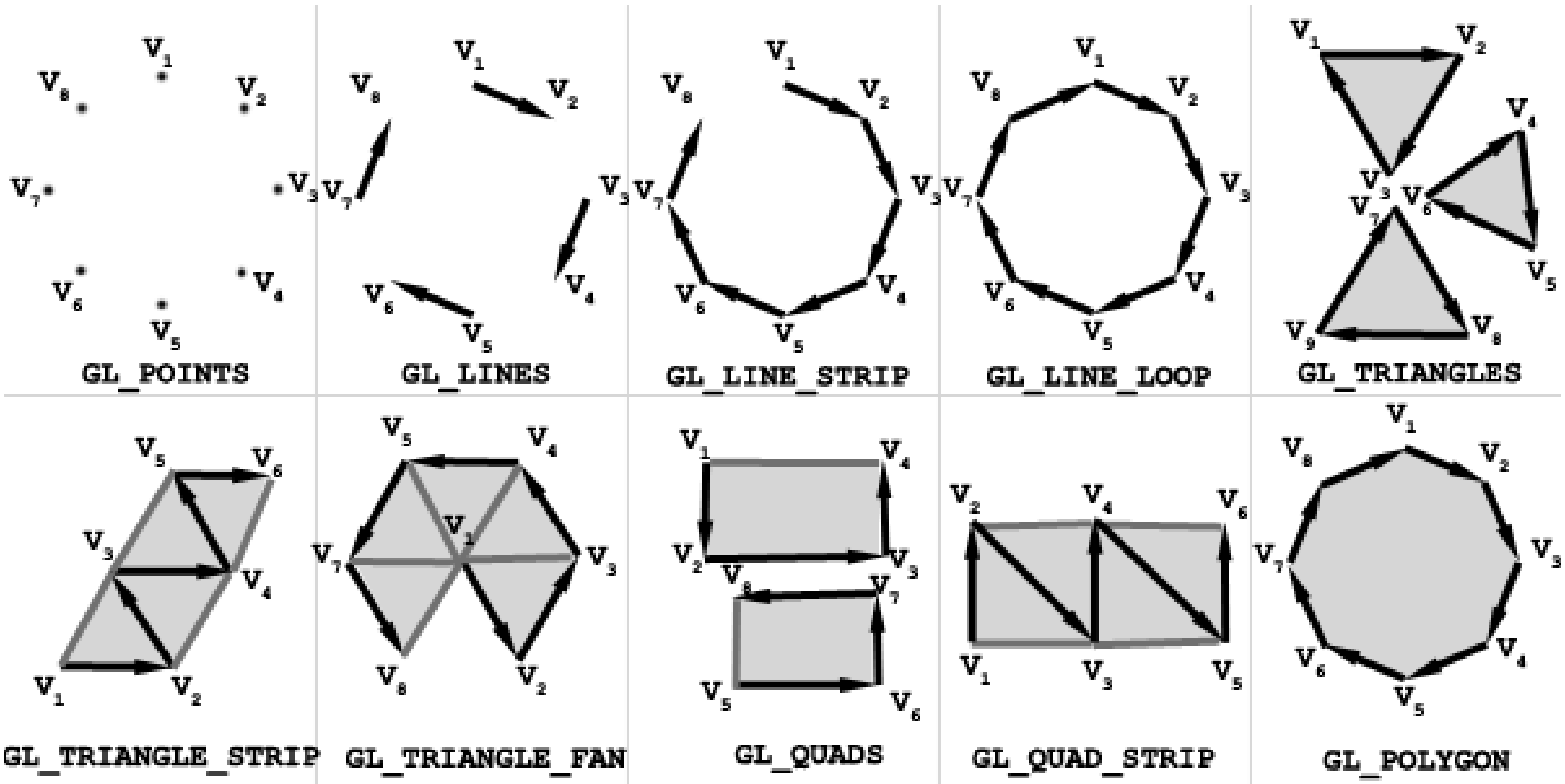
OpenGL (1.x) Fixed Function Pipeline



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
```

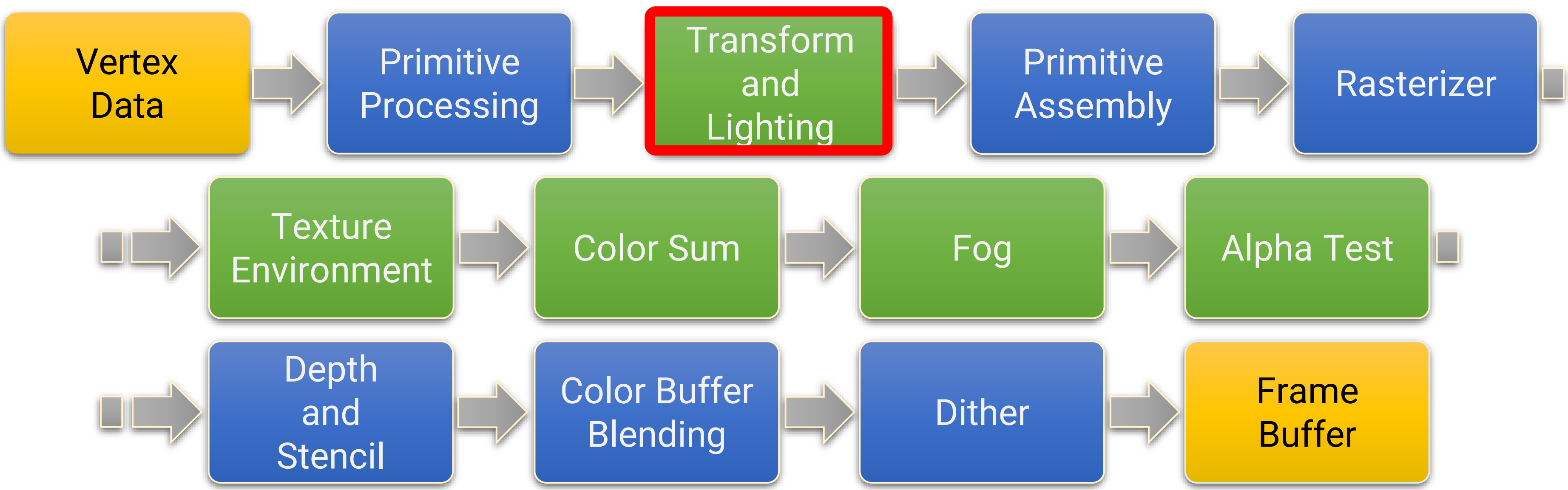
```
glBegin(GL_TRIANGLES);//start drawing triangles
  glVertex3f(-1.0f,-0.25f,0.0f);//triangle one first vertex
  glVertex3f(-0.5f,-0.25f,0.0f);//triangle one second vertex
  glVertex3f(-0.75f,0.25f,0.0f);//triangle one third vertex
  //drawing a new triangle
  glVertex3f(0.5f,-0.25f,0.0f);//triangle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

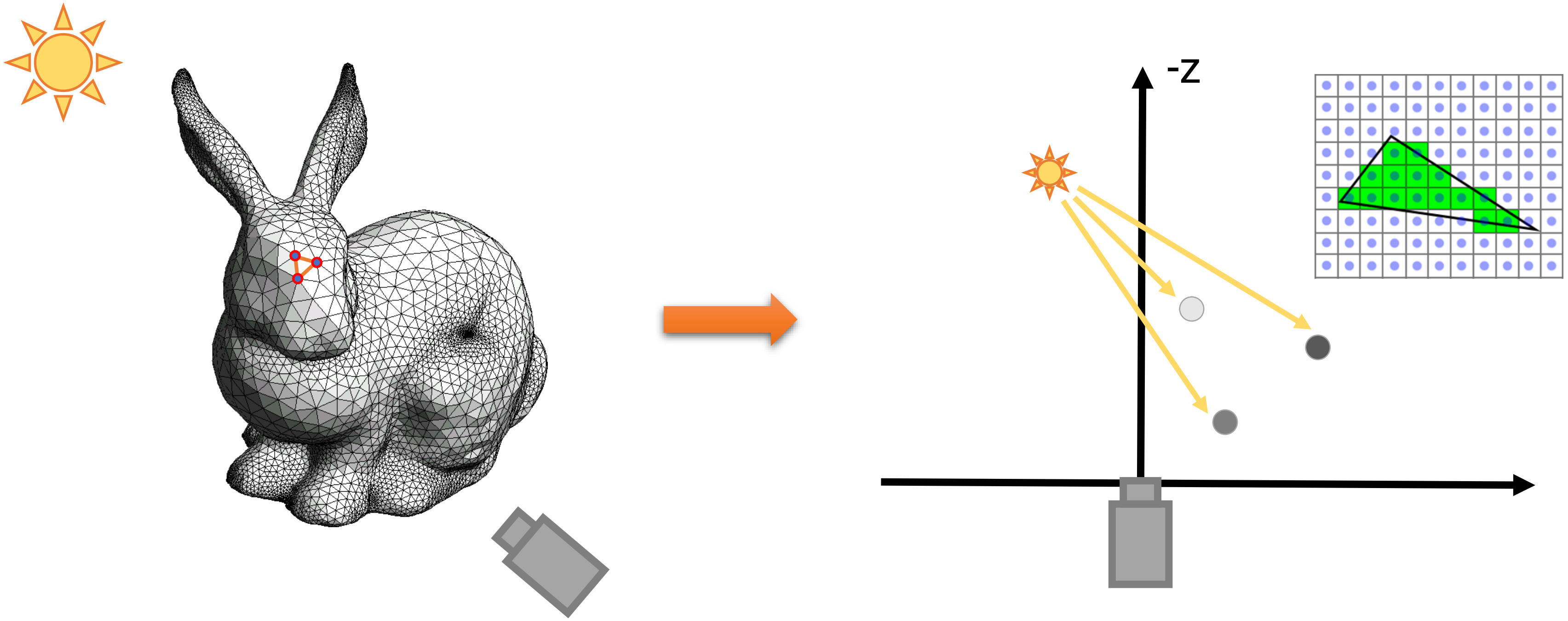
(deprecated, DO NOT USE!)

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  */
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   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 ();
   glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);
}
```

Transform and Lighting (cont.)

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

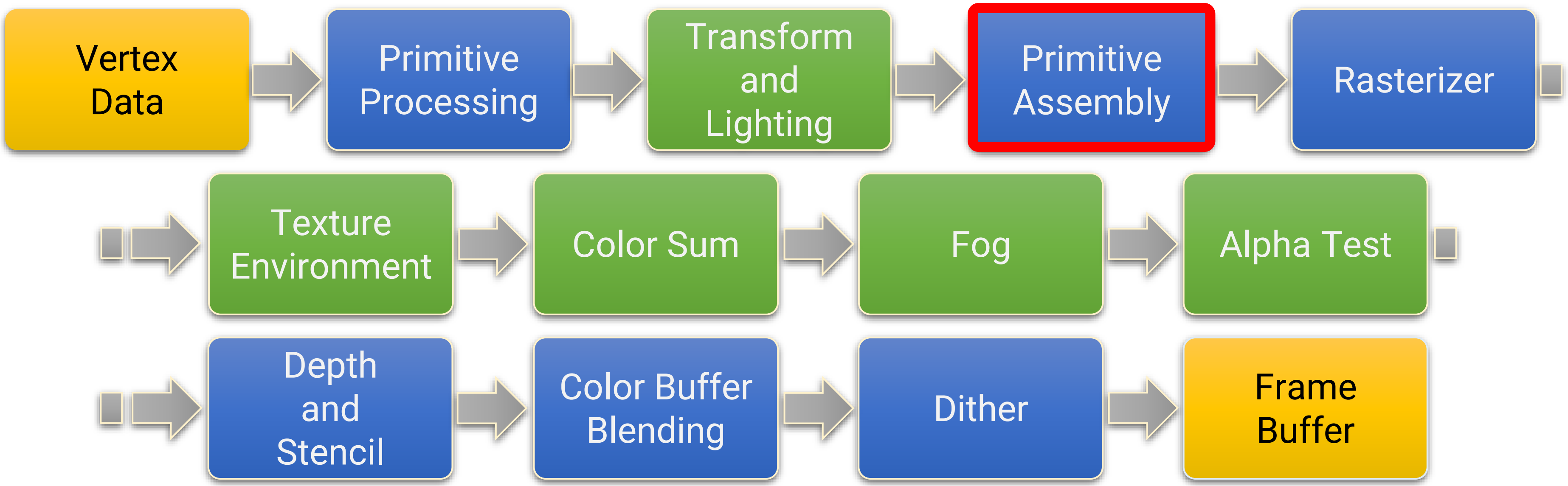
```
void init(void)
{
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);

   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

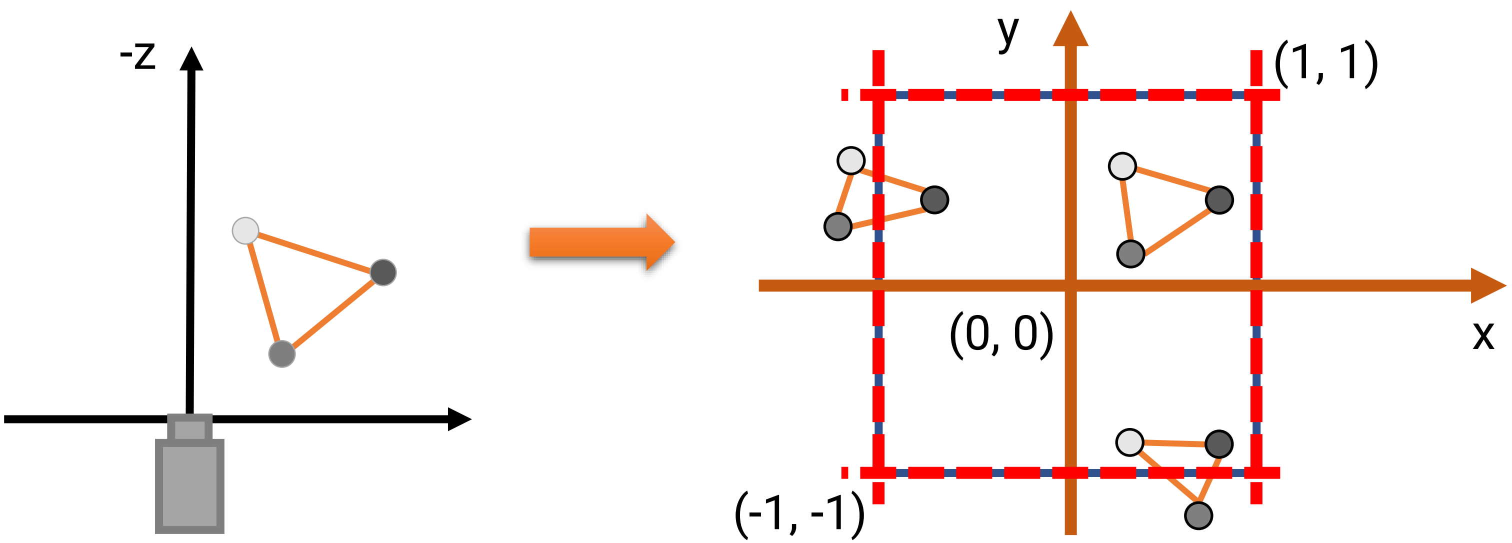
Support at most 8 lights:
GL_LIGHT0 to GL_LIGHT7

OpenGL (1.x) Fixed Function Pipeline



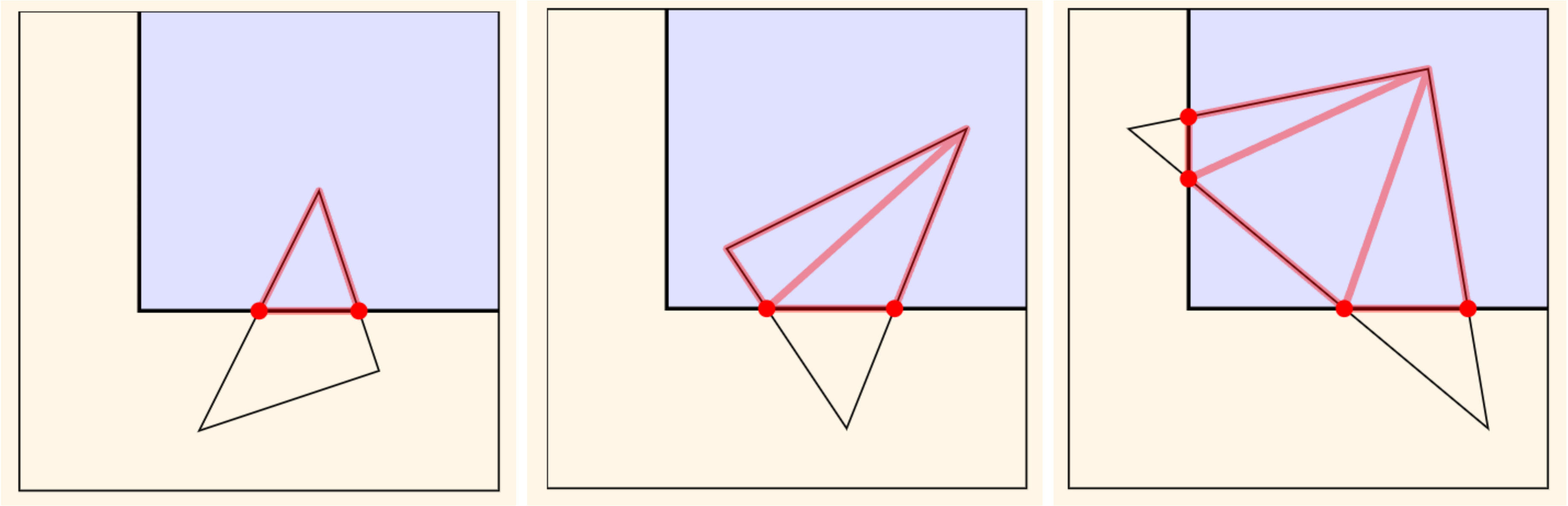
Primitive Assembly

- 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



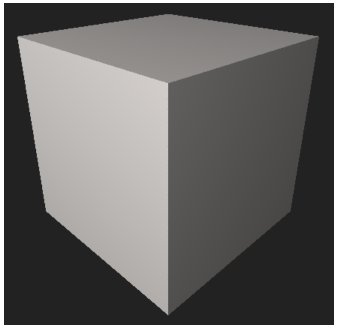
Primitive Assembly (cont.)

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

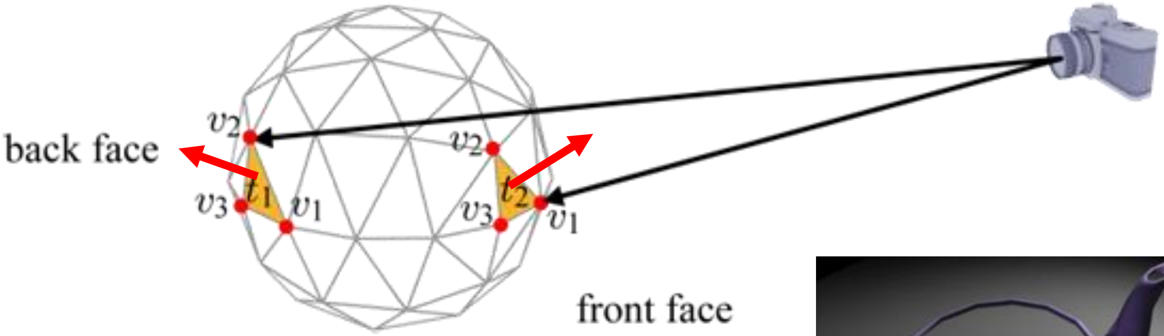


Primitive Assembly (cont.)

- **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



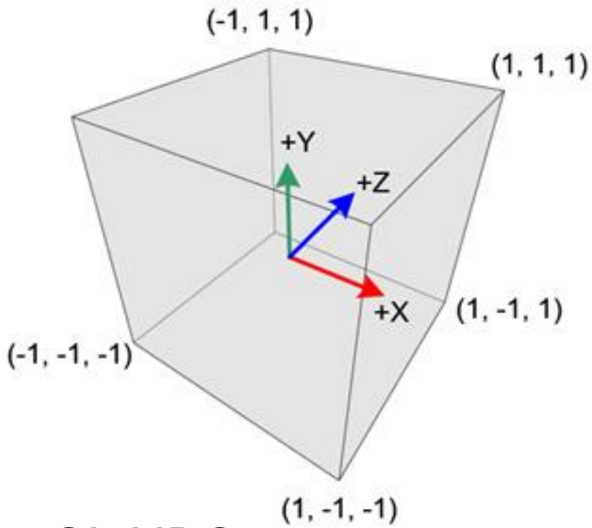
we can only see three faces from six!



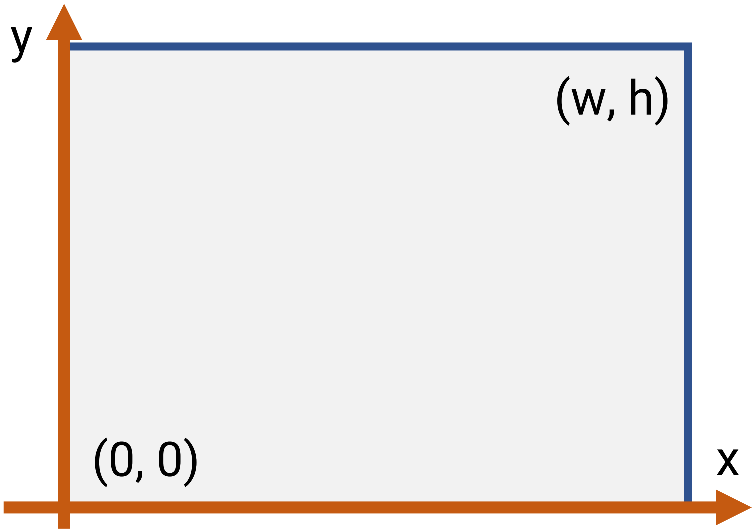
How about this case?

Primitive Assembly (cont.)

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



OpenGL NDC

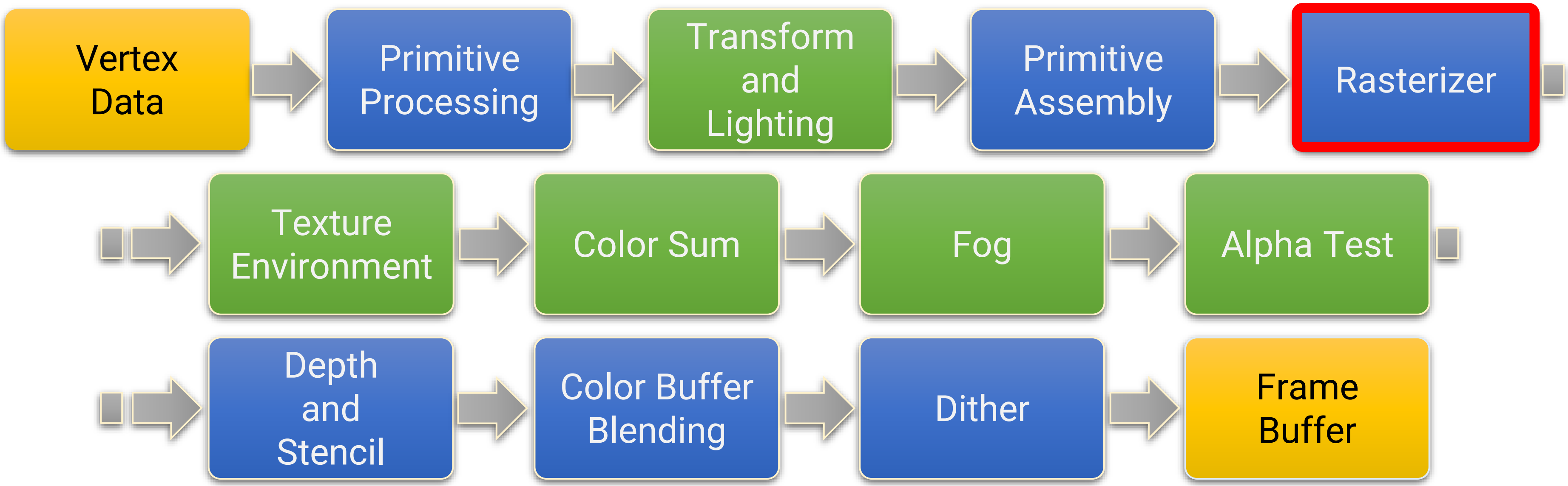


OpenGL Screen Space

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

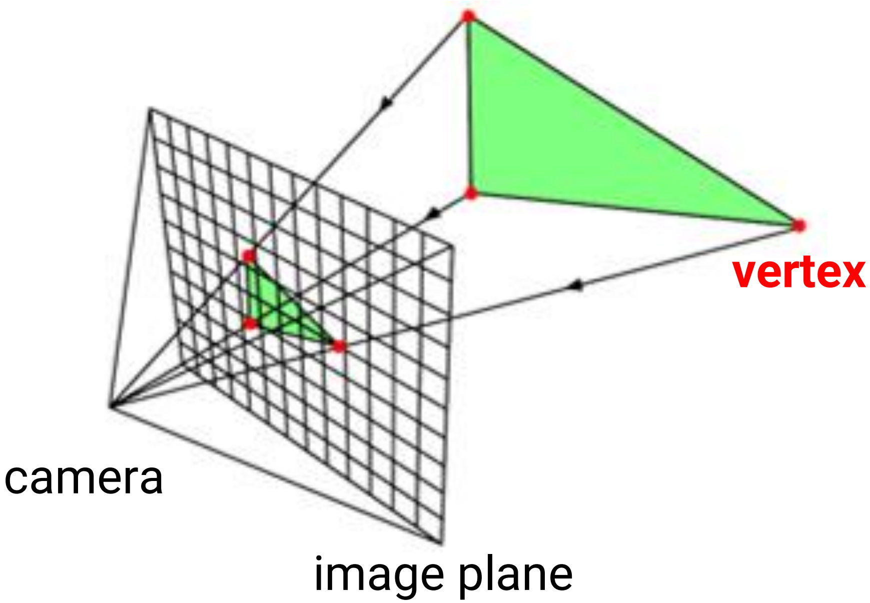
+ screen location

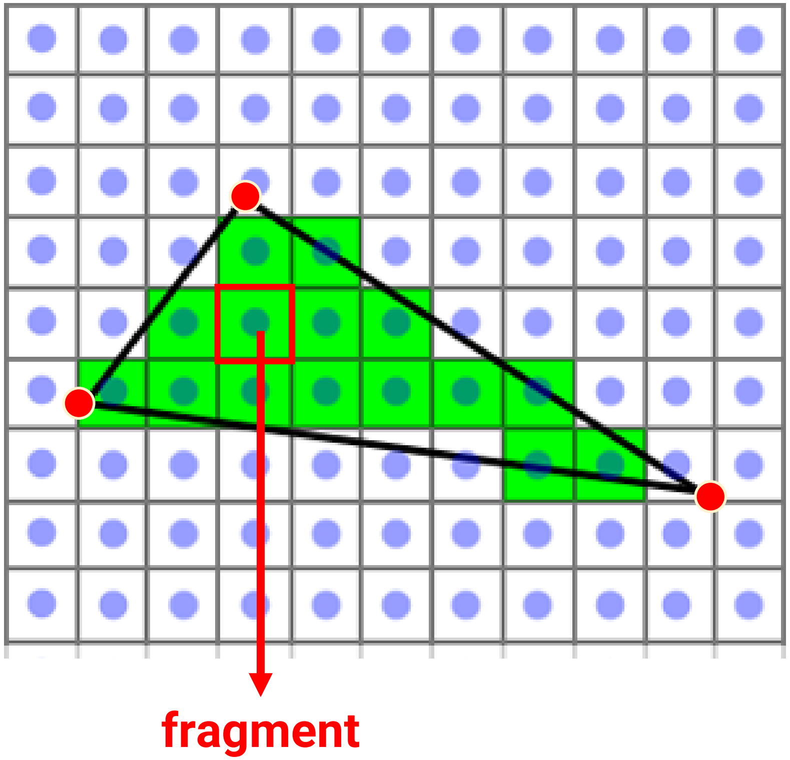
OpenGL (1.x) Fixed Function Pipeline



Rasterization

- The task of taking an **image described in vector graphics format (shapes)** and converting it into a **bitmapped/raster image (pixels)**

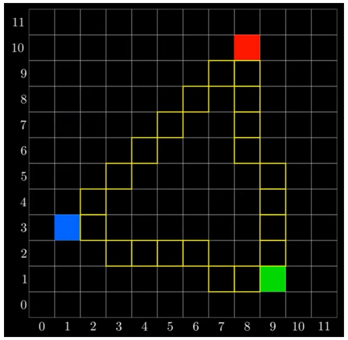


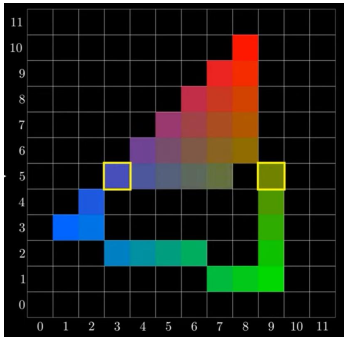


Rasterization (cont.)

- Convert **triangles** (**continuous**) into **fragments** (**discrete**, which eventually become the individual **screen pixels**)
- Vertex attributes are **interpolated** across the face, including
  - (Lighting) color used for per-vertex lighting
  - Texture coordinate
  - Position
  - Normal

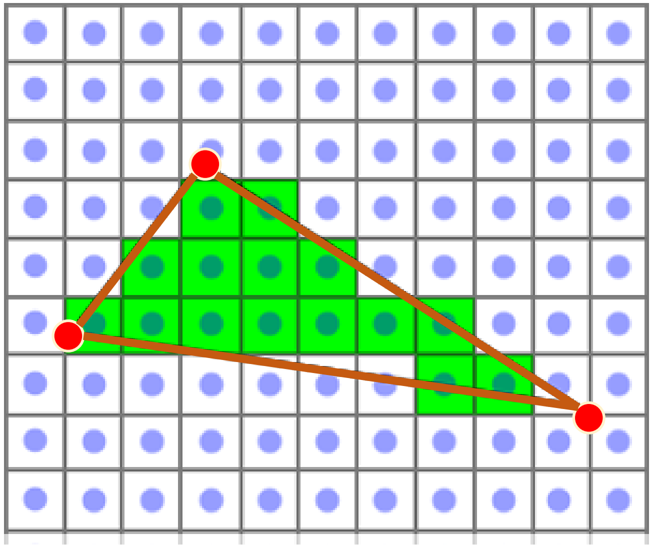
  used for per-fragment lighting (after OpenGL 2.0)
  - Anything you want to interpolate





Rasterization (cont.)

- The task of taking an **image described in vector graphics format (shapes)** and converting it into a **bitmapped/raster image (pixels)**
- **Triangle setup**
  - Setup the properties of a triangle using the vertices data
    - E.g., the equations of edges

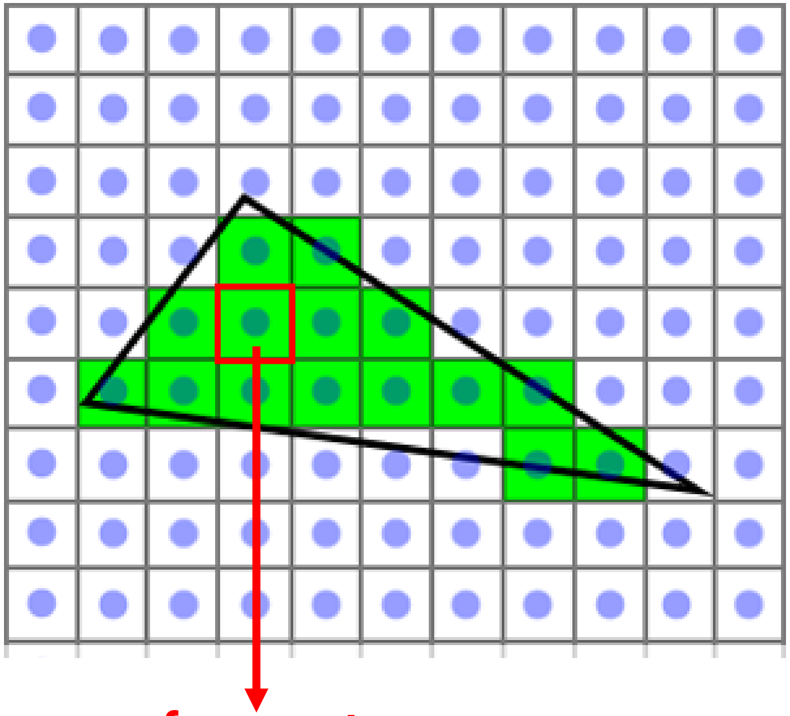


Rasterization (cont.)

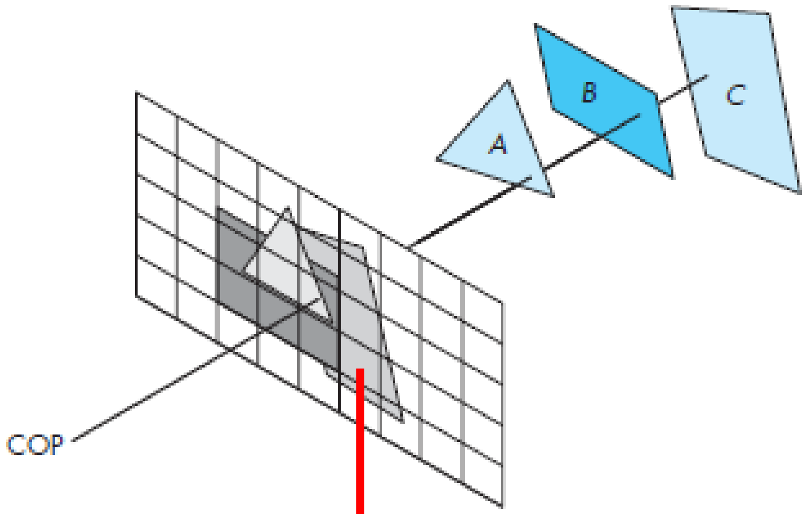
- The task of taking an **image described in vector graphics format (shapes)** and converting it into a **bitmapped/raster image (pixels)**
- **Triangle setup**
  - Setup the properties of a triangle using the vertices data
    - E.g., the equations of edges
- **Fragment generation**
  - For each pixel that is inside the triangle in the screen space, generate a **fragments**
  - Obtain **per-fragment data** using **interpolation**

Rasterization (cont.)

- Fragment != Pixel



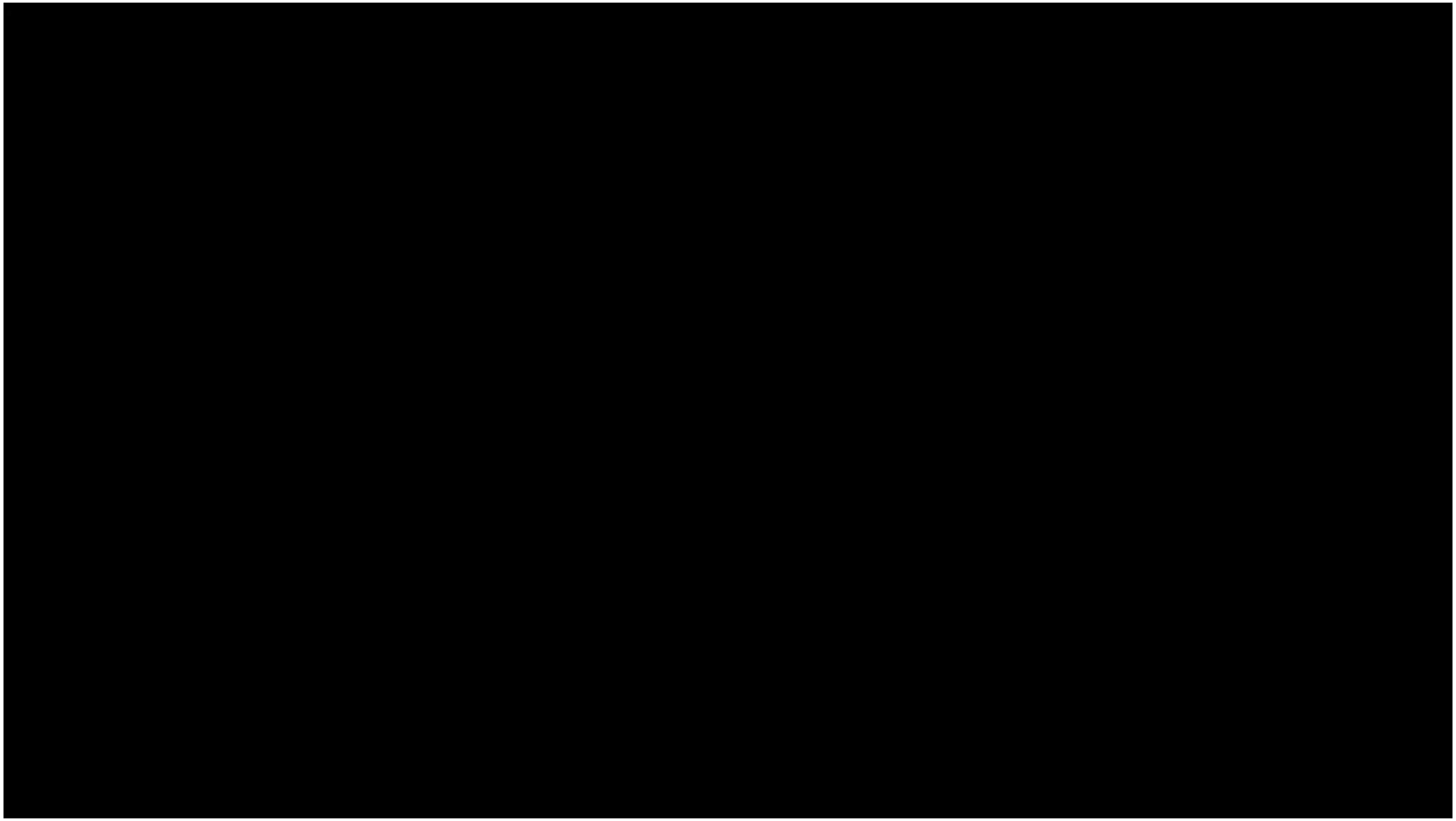
fragment



pixel

Rasterization (cont.)

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



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)

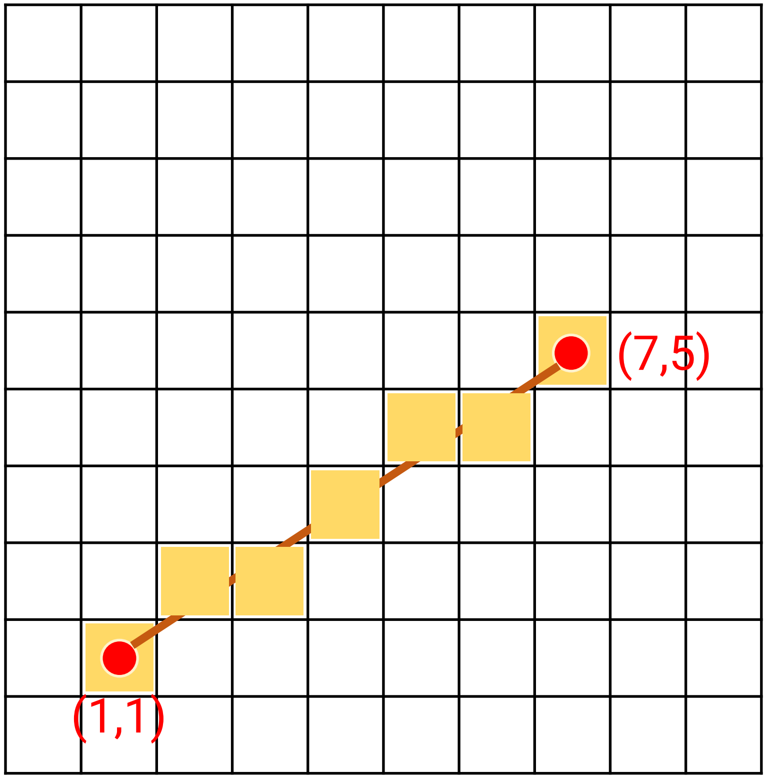
$x_a = 2 \rightarrow y_a = y_1 + m = 1.667 \rightarrow (2, 1.667)$ (2, 2)

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

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

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

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



floating-point addition / comparison

Bresenham Algorithm

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

$$y = mx + b$$

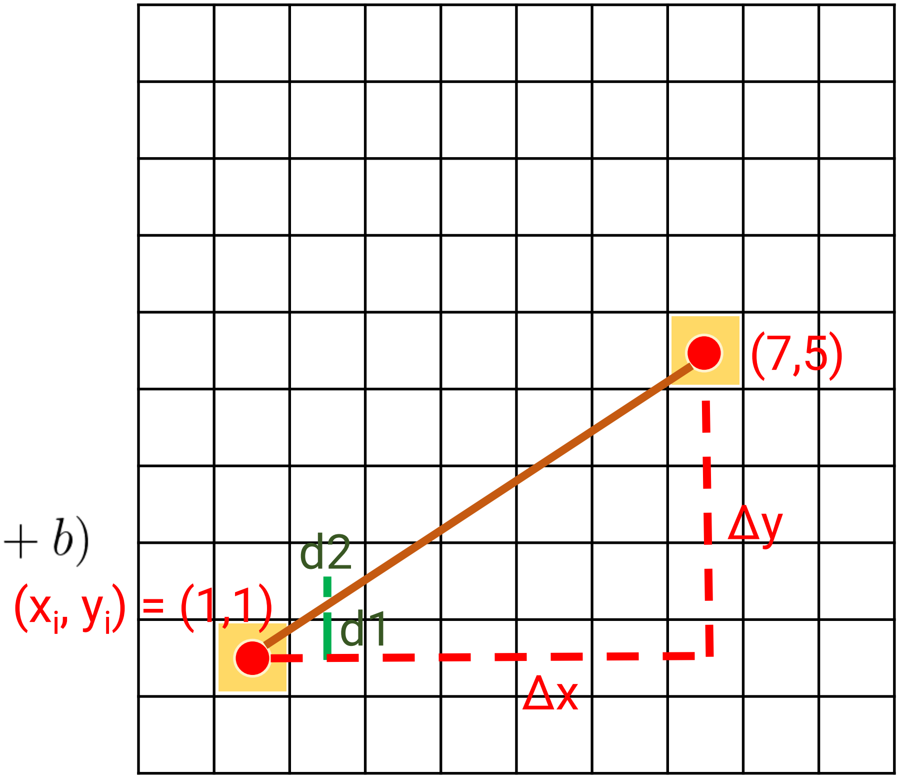
$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

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

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

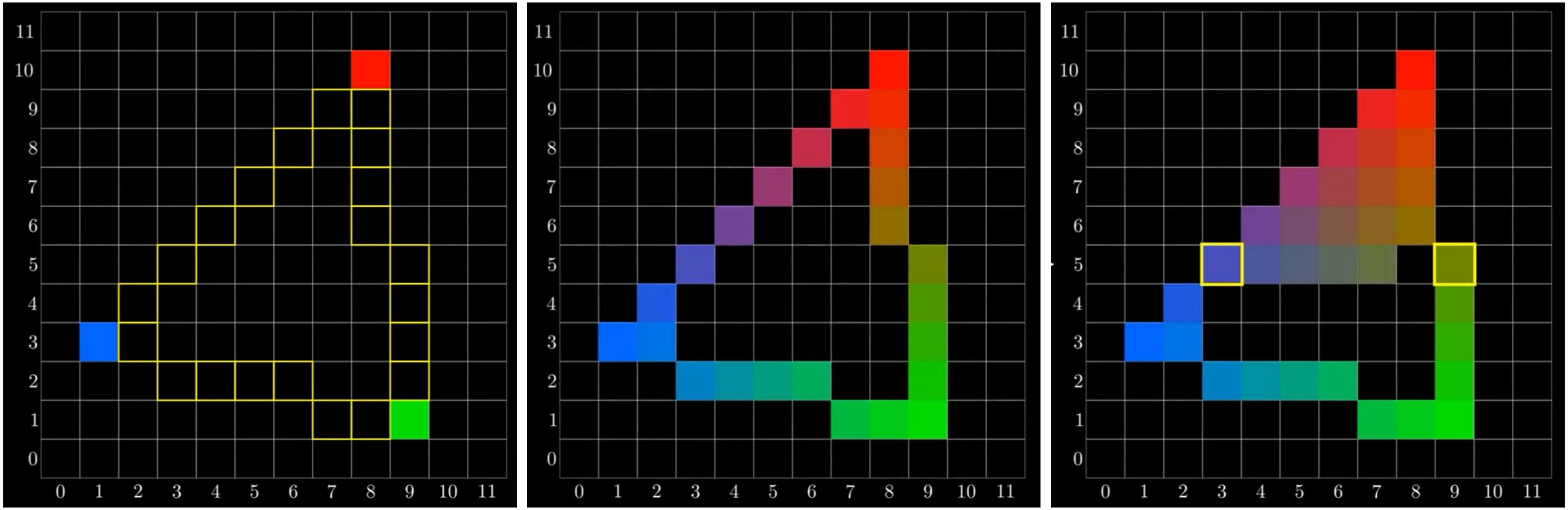
$$d1 - d2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

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



integer multiplication / comparison

Scanline Rasterization



Attributes interpolation of edge pixels using vertices

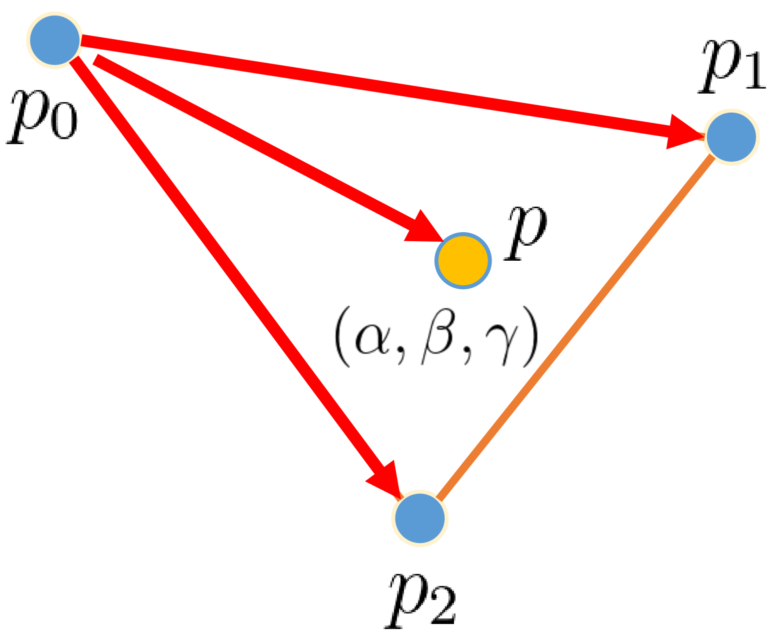
(interpolate y dir.)

Attributes interpolation of inner pixels using edge points

(interpolate x dir.)

Barycentric Coordinates

- Barycentric coordinates inside a triangle



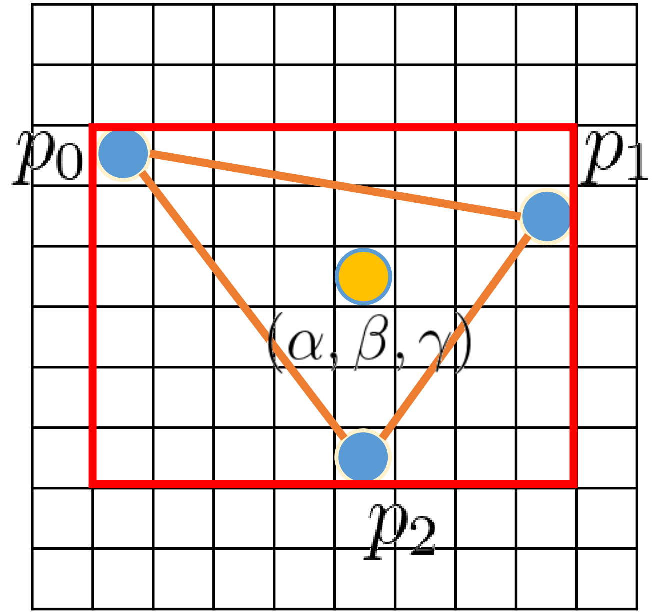
$$p = p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0)$$
$$= (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2$$
$$= \alpha p_0 + \beta p_1 + \gamma p_2$$
$$\alpha + \beta + \gamma = 1$$

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

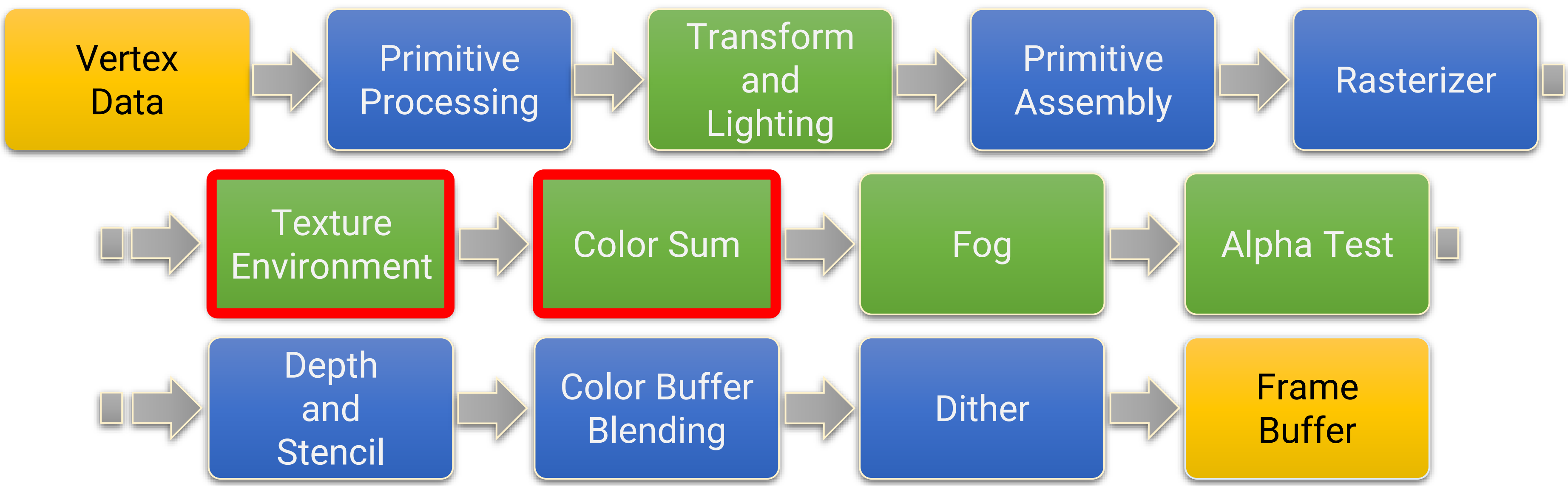
Barycentric Coordinates (cont.)

- Compute the 2D bounding box of the 2D triangle
- For each pixel inside the bounding box, compute its barycentric coordinates
- 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

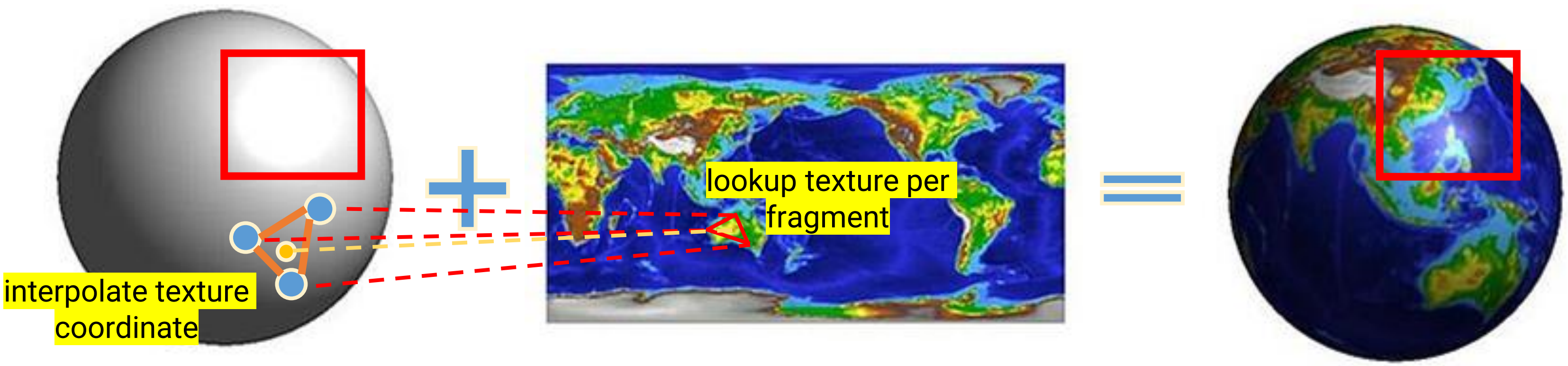


OpenGL (1.x) Fixed Function Pipeline



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

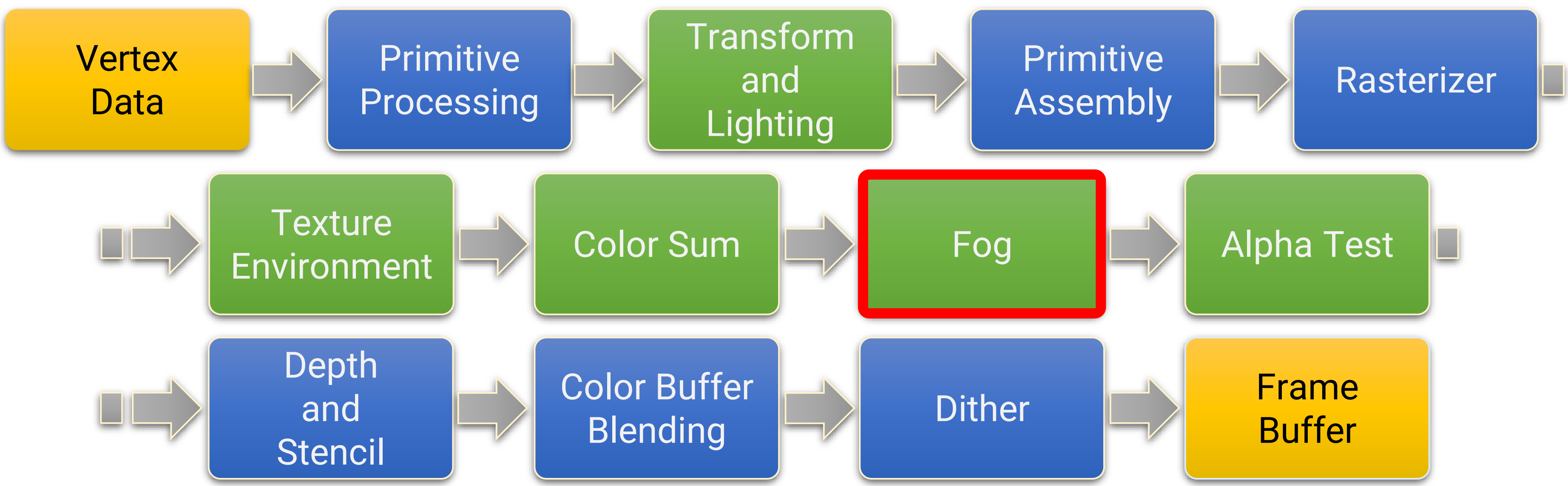


3D object

texture

texture mapped object
(color = lighting × texture)

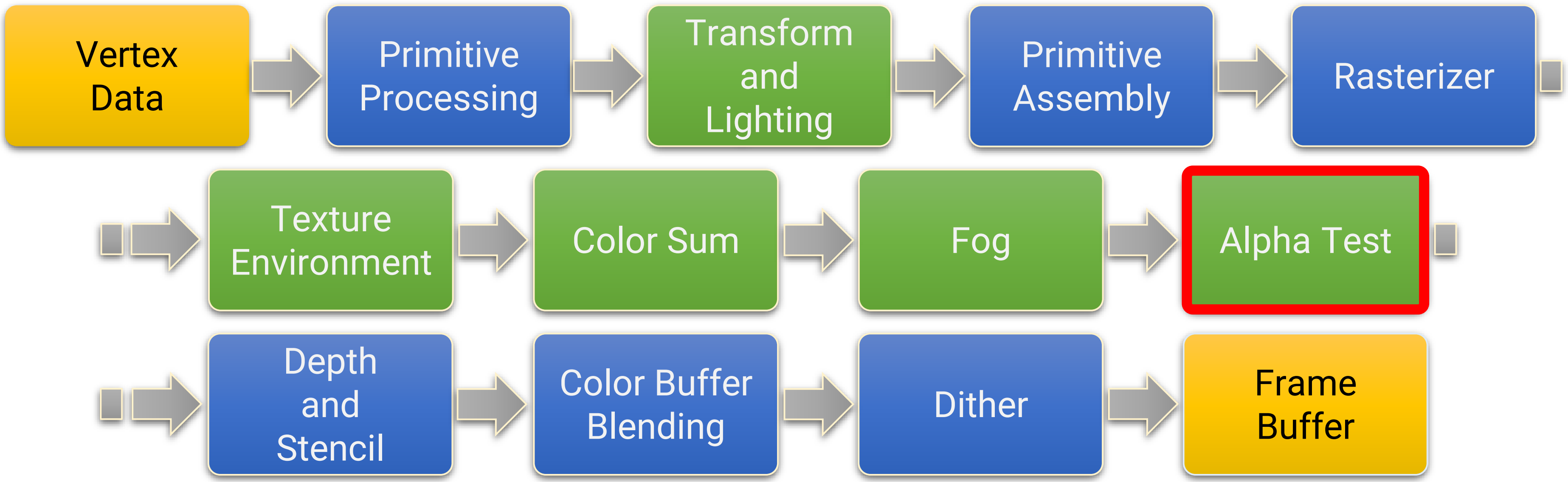
OpenGL (1.x) Fixed Function Pipeline



Fog

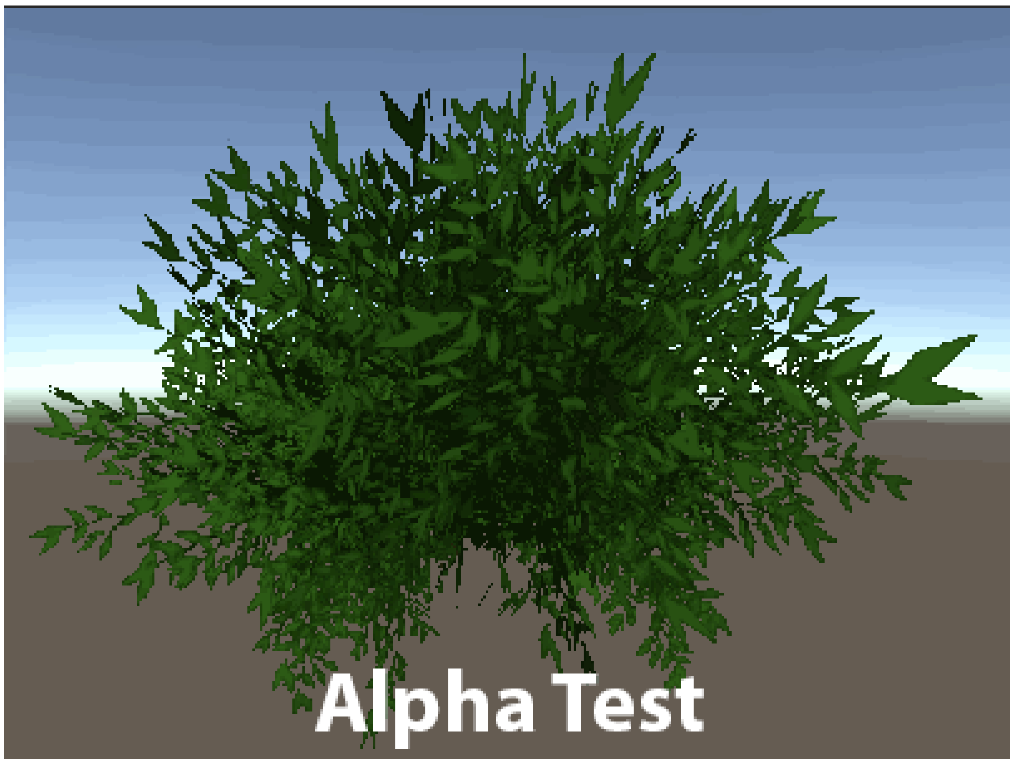
- Simulate the effect of geometry fadeout as dimmed by fog
- Linearly blend the fragment color with the fog color

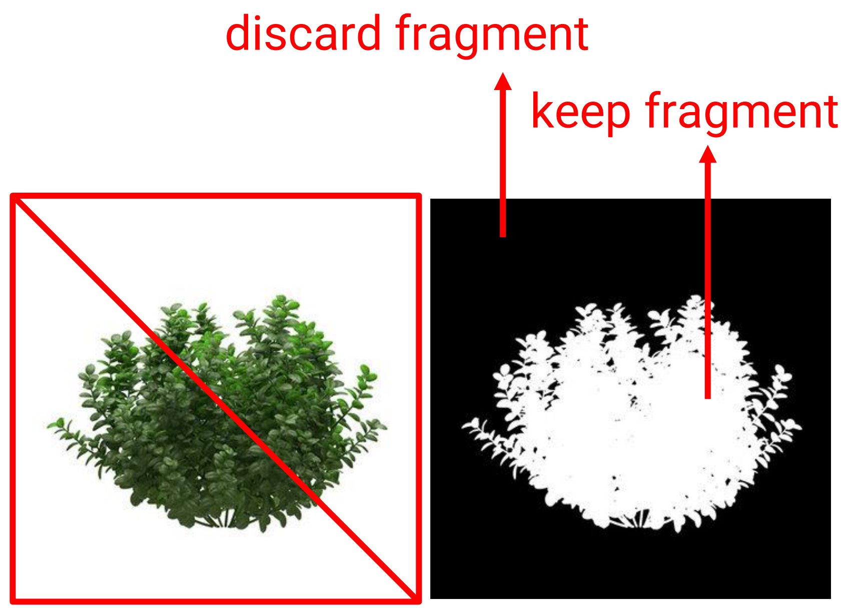
OpenGL (1.x) Fixed Function Pipeline



Alpha Test

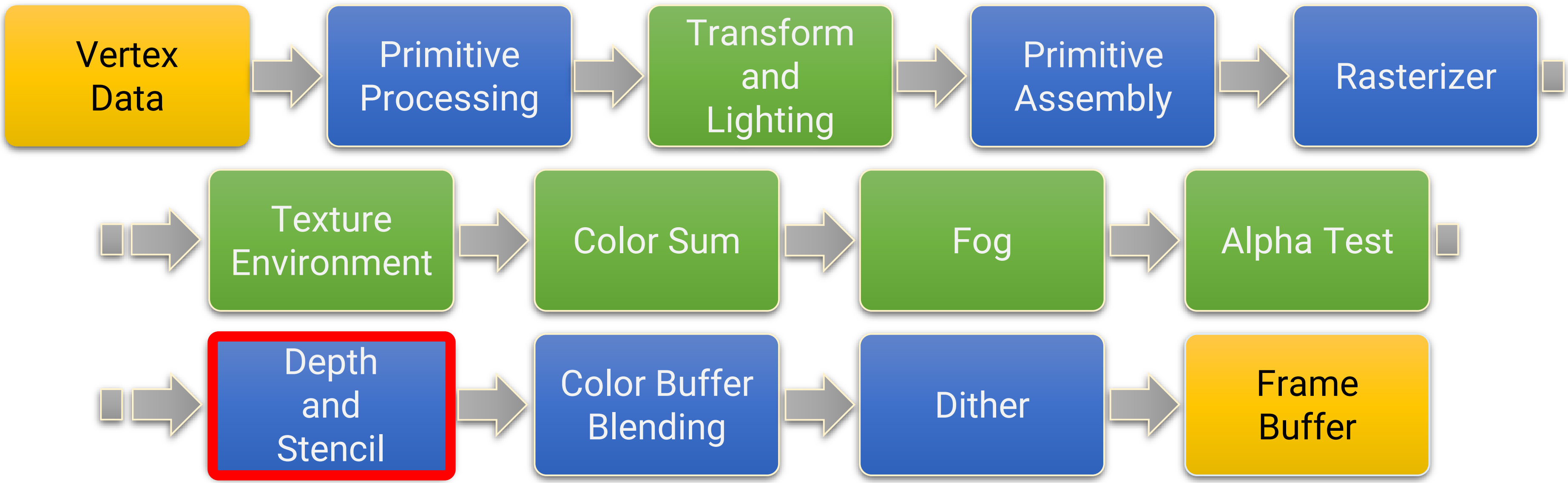
- Discard fragments if their alpha values are below a certain threshold





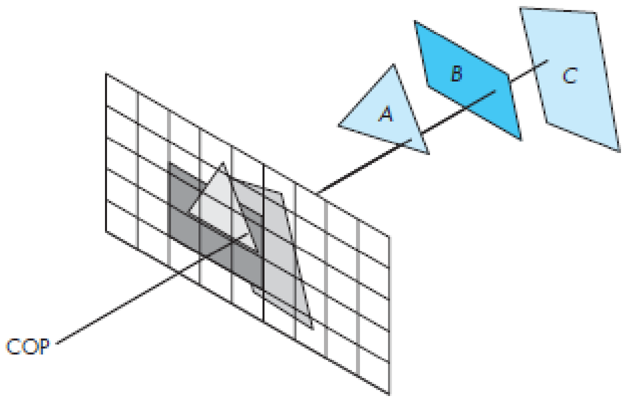
albedo texture alpha mask
(usually combined in an RGBA texture)

OpenGL (1.x) Fixed Function Pipeline



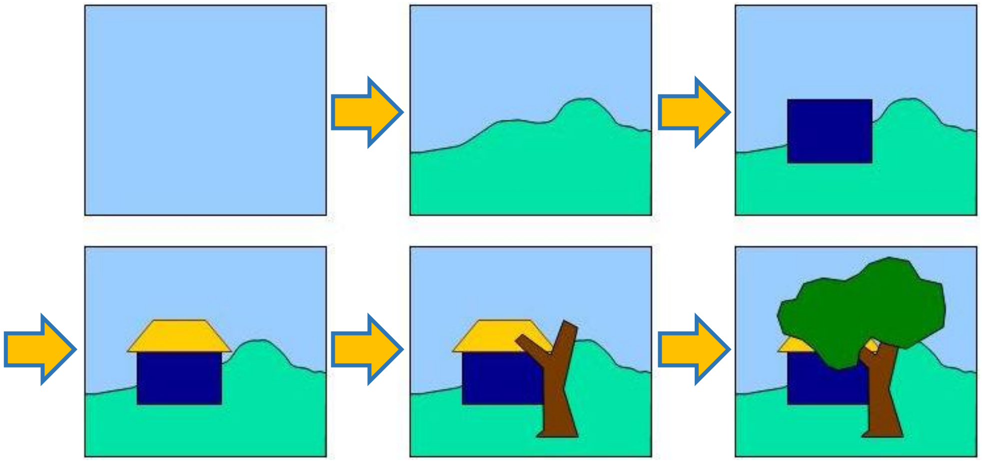
Depth Test

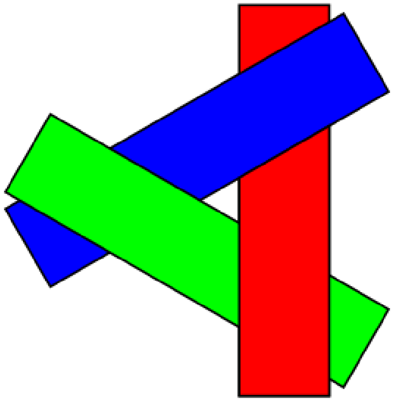
- Used for **hidden surface removal**
  - Only show the **closest** surfaces to the camera at each pixel



Depth Test (cont.)

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

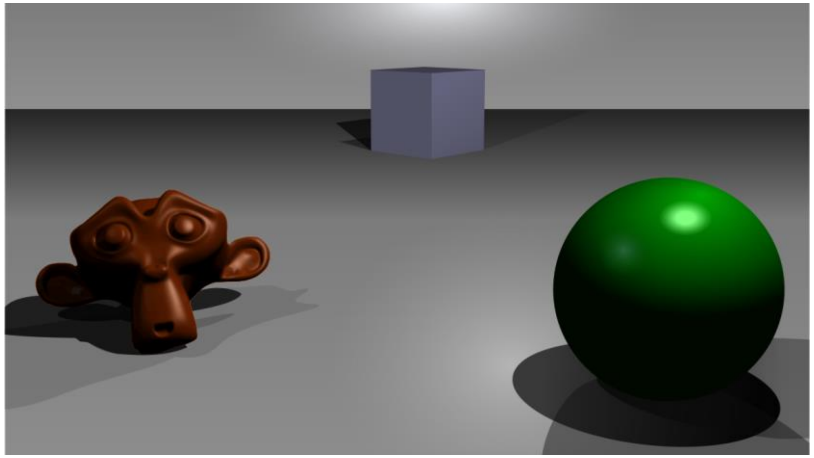




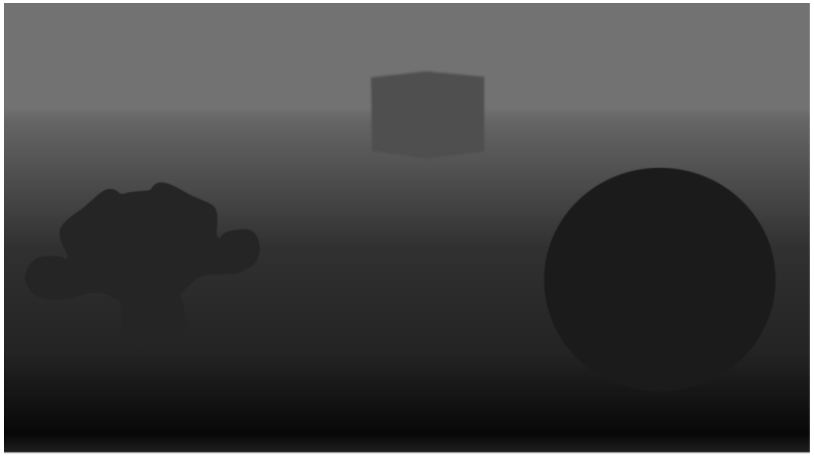
How about this?
(cyclical overlapping)

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



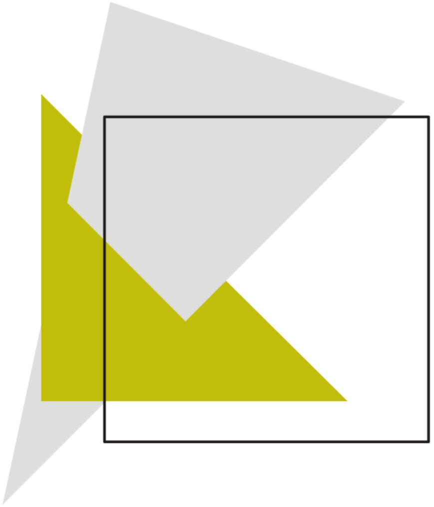
color frame buffer



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	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

+

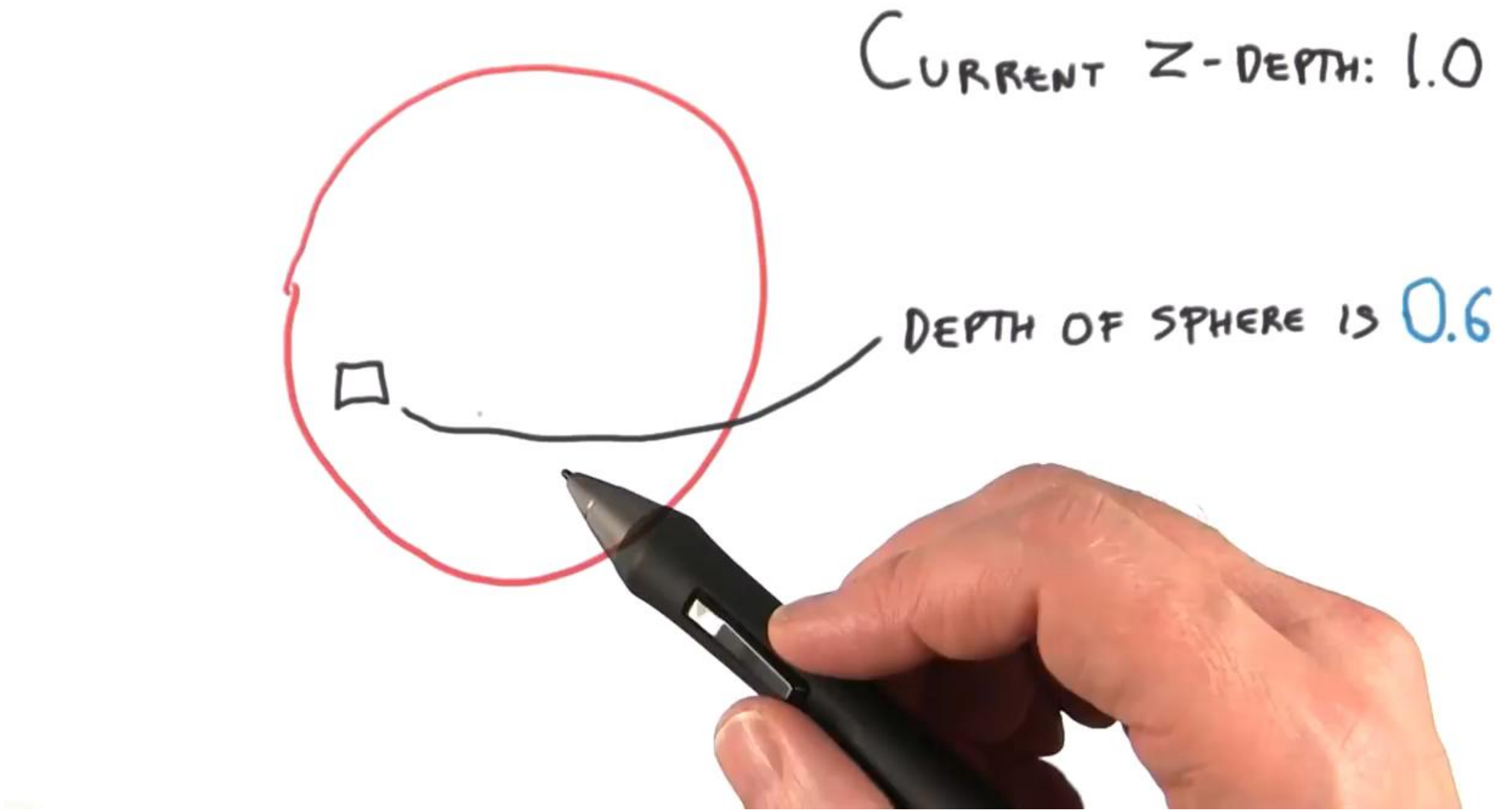
7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

=

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	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

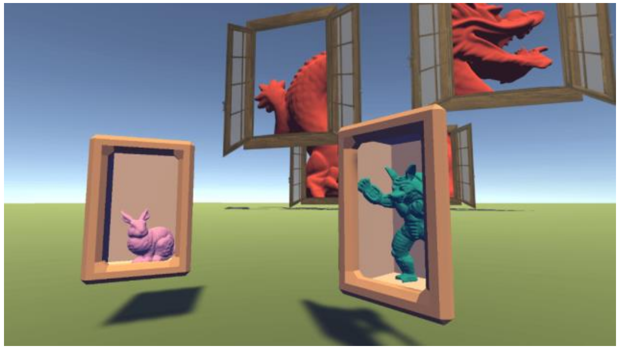
Z-Buffer (cont.)

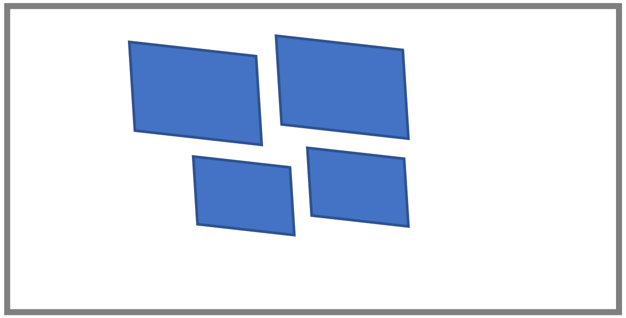
- https://www.youtube.com/watch?v=yhwg_O5HBwQ



Stencil Test

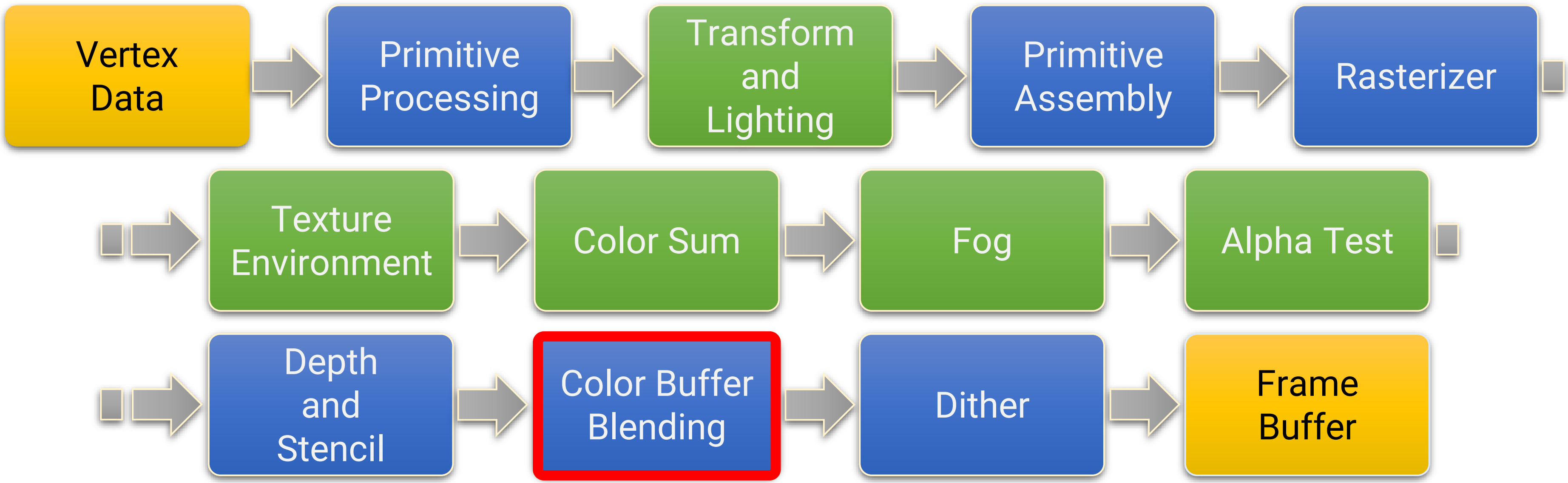
- Used to discard fragments that fail a stencil comparison, based on the content of the stencil buffer





stencil buffer

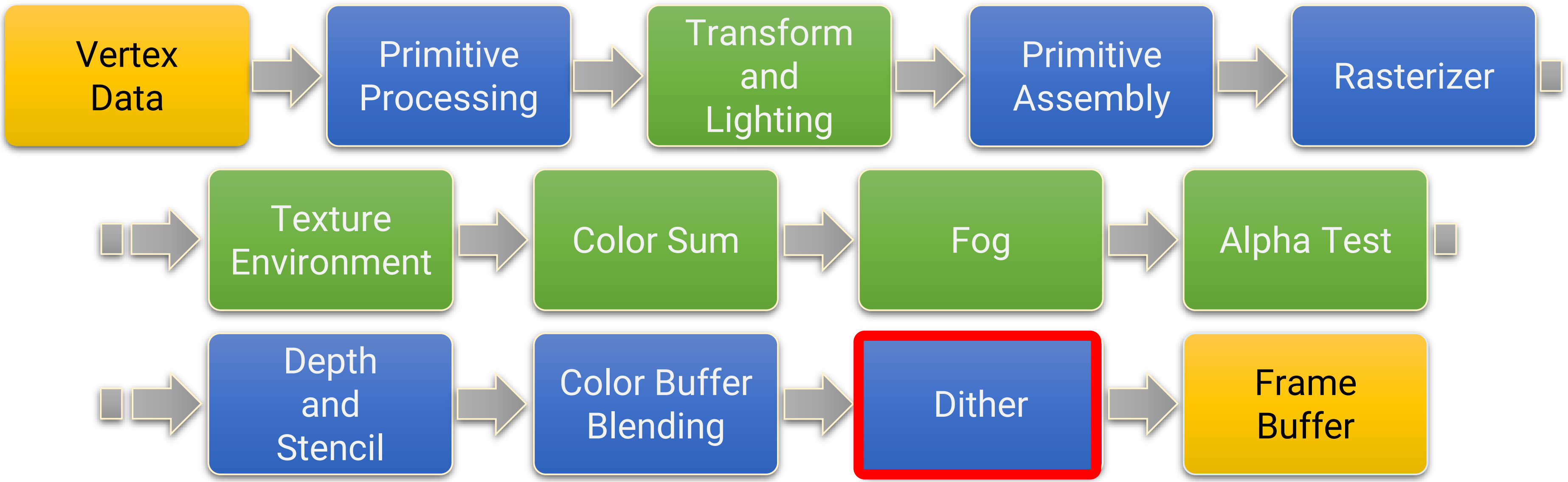
OpenGL (1.x) Fixed Function Pipeline



Color Buffer Blending

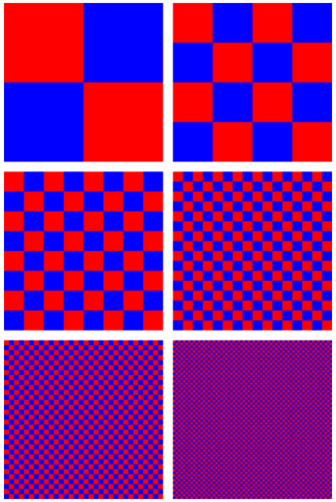
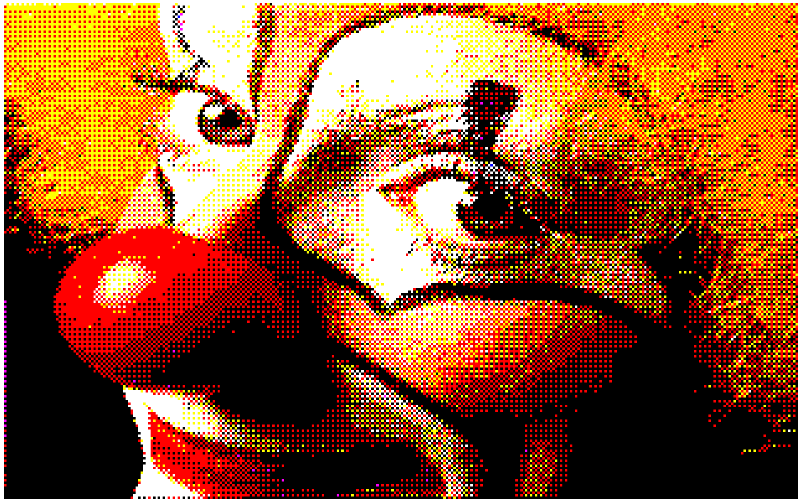
- Blend the color of fragments with the previous results in the frame buffer based on the **alpha values** of the current fragments, as well as the **blend function** and the **blend equations**

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)
  - **Fixed (Phong) lighting model on vertices**
  - **Fixed modulation of lighting color and texture color**
    - Color = Lighting $\times$ Texture

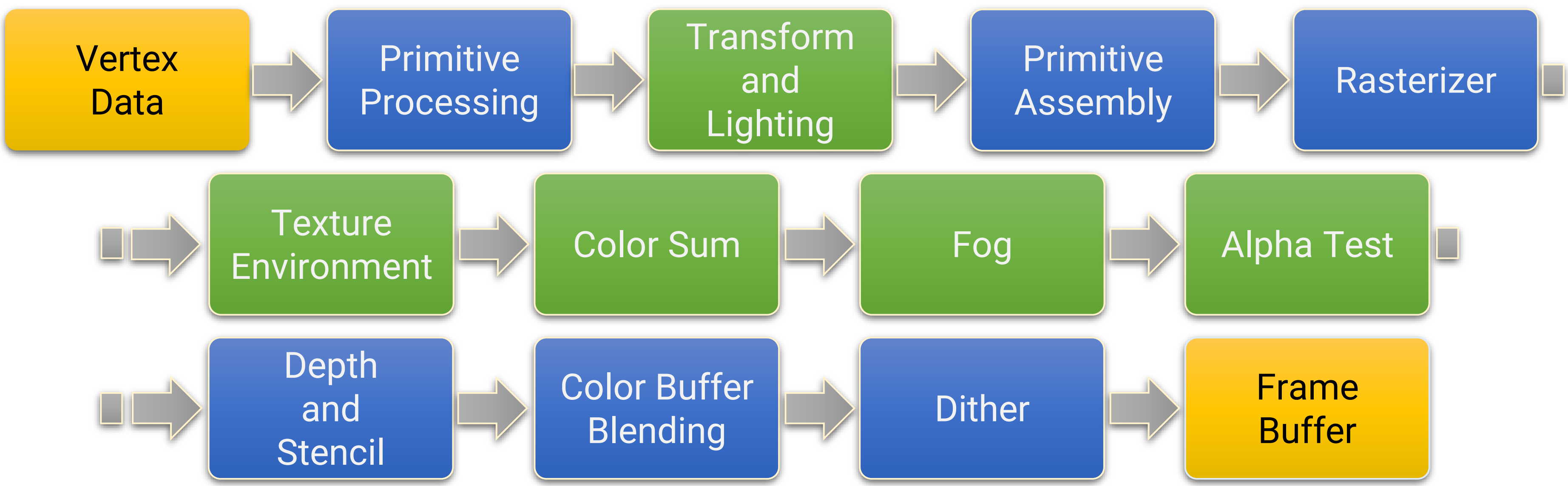
- **We would like more flexibility!**

Outline

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

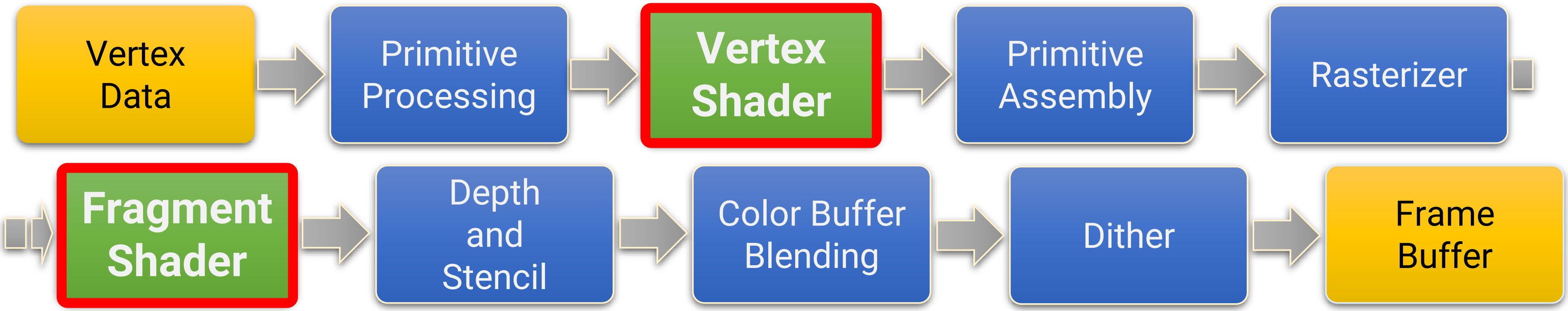
Recap: OpenGL (1.x) Fixed Function Pipeline

- 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**



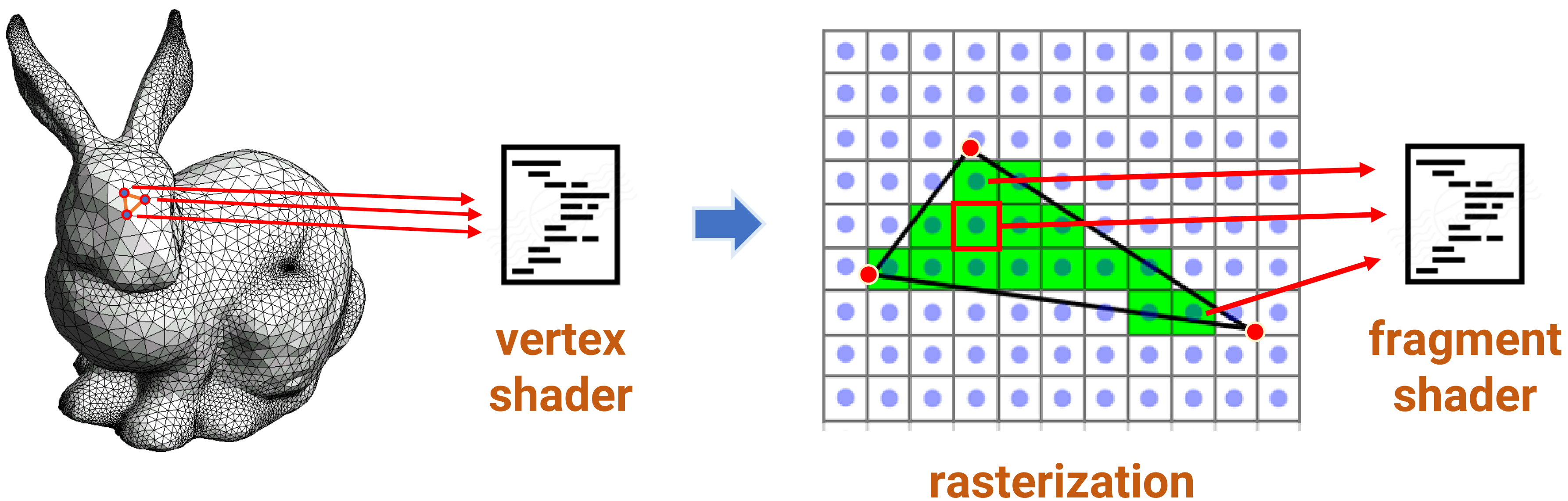
OpenGL (2.0) Graphics Pipeline

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



Vertex Shader and Fragment Shader

- **Important concepts**
  - The vertex shader runs **per vertex**
  - The fragment shader runs **per (rasterized) fragment**



Vertex Shader (Run per Vertex)

- Give the programmers more **flexibility** regarding
  - **How the vertices are transformed**
    - We can also choose not to transform the vertices at all
  - **How the lighting is computed**
    - We can also choose to compute lighting in the fragment shader (per-fragment lighting)
- However, ***with great power, comes great responsibility***
  - Programmers have to implement the functions provided by the fixed pipeline on their own
  - The primary responsibility of the vertex shader program is to **transform the vertex position into Clip Space**
    - Commonly, this is done by multiplying the vertex with the **model-view-projection** matrix

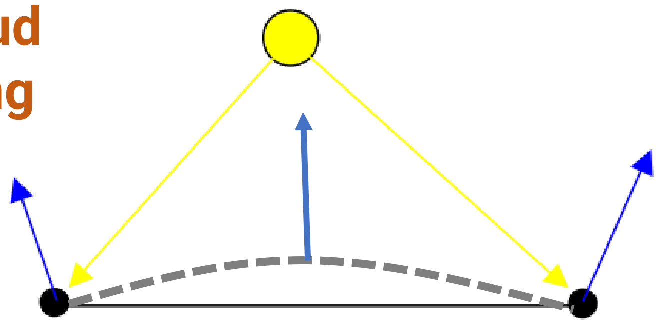
Fragment Shader (Run per Fragment)

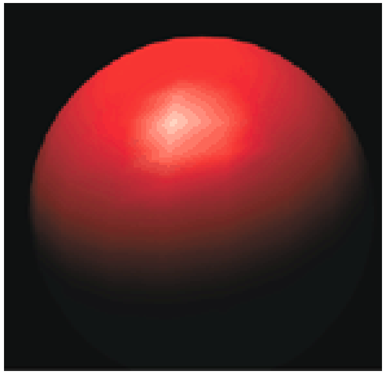
- Replace the texture blending, color sum, fog, and alpha test operations from the fixed function pipeline
- Graphics programmers have to write a fragment program to perform these operations (of course, you can omit them if you do not care!)
- The primary responsibility of the fragment program is to **determine the final color of the fragment**
- Allow different lighting and fog model, as well as an arbitrary combination of lighting and texture
- Allow for techniques such as per-pixel lighting, bump, normal mapping, etc.

Per-Fragment Shading

- Problem with Gouraud shading
- Phong shading (instead of Gouraud shading)

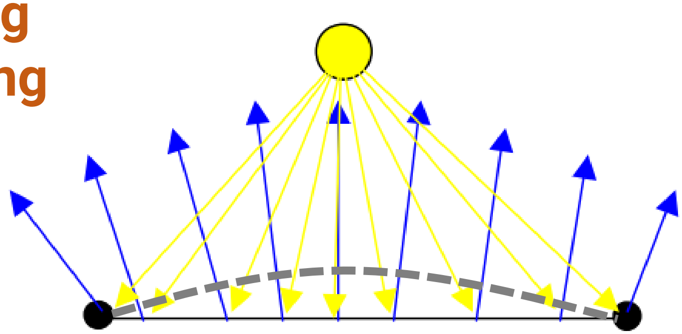
Gouraud shading

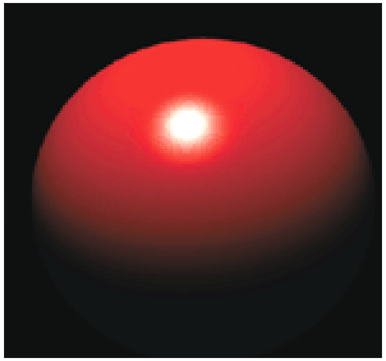




lighting color is interpolated

Phong shading

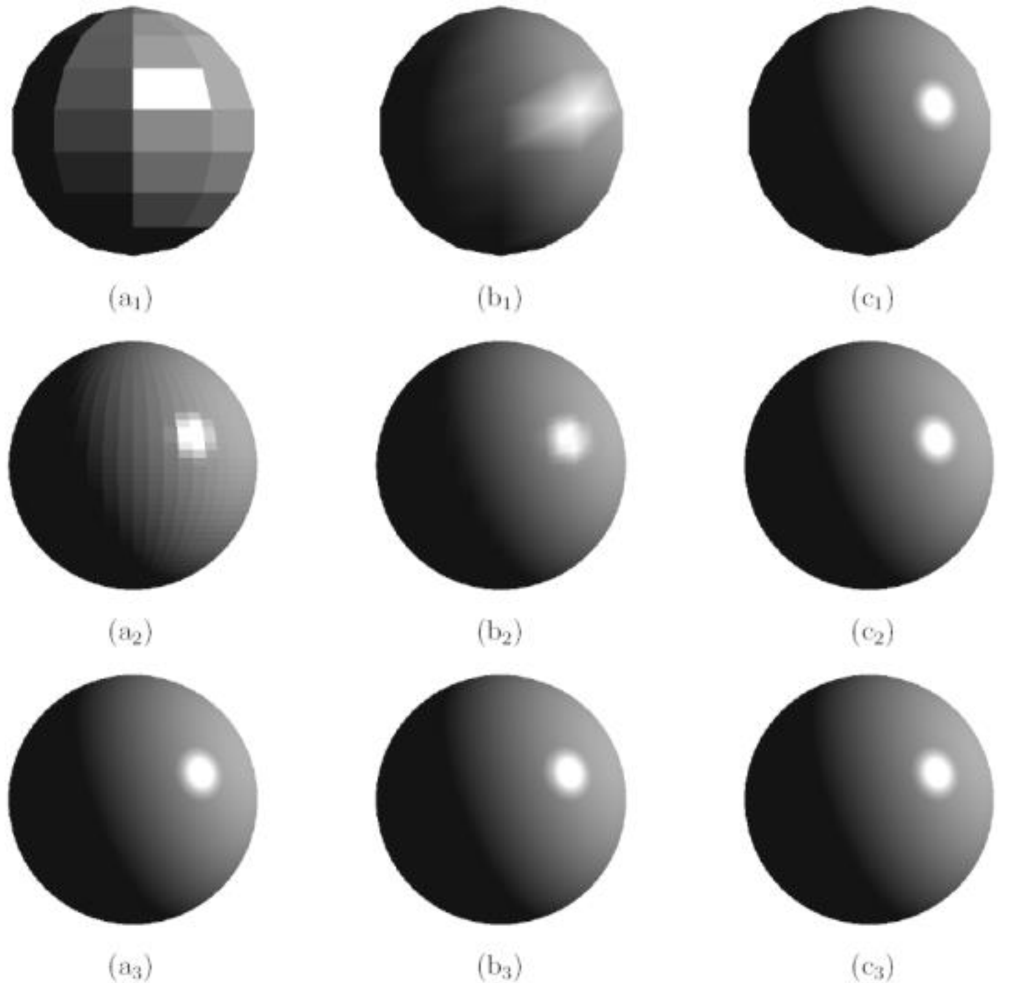




surface normal is interpolated (how? Rasterization!)

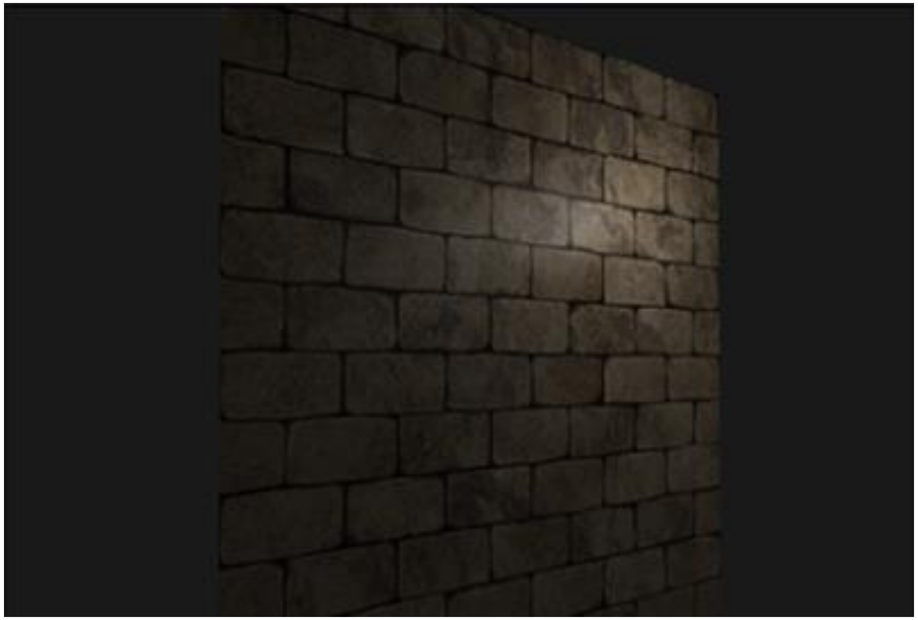
Per-Fragment Shading (cont.)

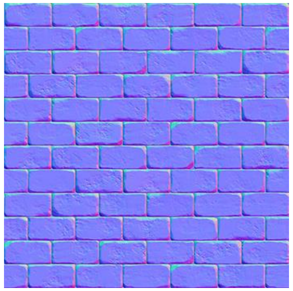
flat shading   Gouraud shading   Phong shading



Per-Fragment Shading (cont.)

- Normal mapping





We will talk about this when introducing Textures

Modern Graphics Pipeline

- Modern graphics pipeline comprised more programmable (shader) stages, such as
  - **Geometry shader** in OpenGL 3.2
  - **Tessellation control shader** and **tessellation evaluation shader** in OpenGL 4.0
  - **Compute shader** in OpenGL 4.3
  - **Mesh shader** in OpenGL ?

- Hopefully, we could have time to introduce these shaders later in this semester


END OF PRESENTATION
ANY QUESTIONS?