Transparency

Computer Graphics

Yu-Ting Wu

Transparency

- So far, the objects we render are all opaque
  - Z-buffer will keep the closest objects from the camera for us
- What if the scene contains transparent objects
  - We would like to see occluded objects through the transparent one!

Alpha Value

- The transparency of an object is defined by its color's **alpha** value (the 4th component of a color vector)
  - Previously, we set this to a fixed value of **1.0**, giving the object zero transparency (fully **opaque**)

```
void main()
{
    FragColor = vec4(fillColor, 1.0);
}
```

RGB A

in Shader example

```
void main()
{
    vec3 texColor = texture2D(mapKd, iTexCoord).rgb;
    FragColor = vec4(iLightingColor * texColor, 1.0);
}
```

return vec4 RGBA

RGB A

in Texture example

  - On the other hand, an alpha value of **0.0** results in the object having complete transparency
  - The values in between mean **semi-transparency**

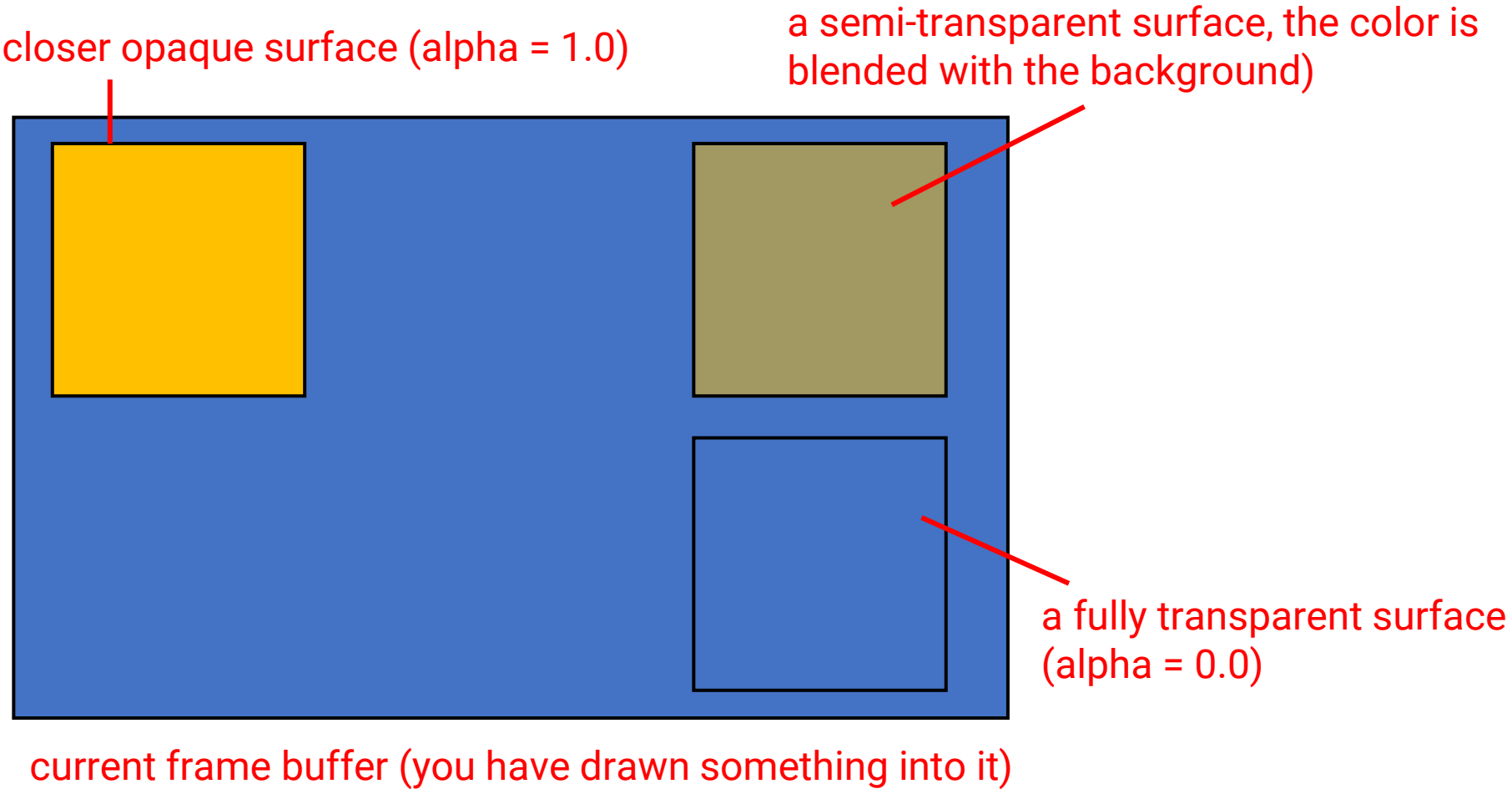
Alpha Blending

- For rasterization, transparency is difficult to resolve **because each polygon only has its own information**
  - It does not know which triangle locates behind, so it cannot determine the pixel color in its fragment shader

- **Major idea**
  - Render **opaque** objects into the fragment buffer first
  - Render **transparent** objects **in an order w.r.t their distance to the camera** (farther objects first)
  - When rendering transparent objects, **blend** the surface color with the previous results in the color buffer

Alpha Blending (cont.)

- Concept of transparency in rasterization



Alpha Blending in OpenGL (cont.)

- OpenGL provides flexibility to composite the fragment color when rendering transparent objects

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

blending equation

- $\bar{C}_{source}$: the source color vector (**the color output by the fragment shader**)
- $\bar{C}_{destination}$: the destination color vector (**the color vector currently stored in the color buffer**)
- F_{source} : the source factor value (set the impact of the **alpha value** on the source color)
- $F_{destination}$: the destination factor value (set the impact of the **alpha value** on the destination color)

Alpha Blending in OpenGL (cont.)

- Implementation transparency in OpenGL

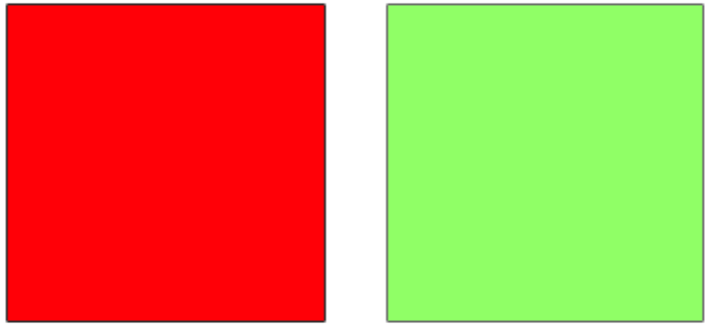
Option	Value
GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector: $1 - \bar{C}_{source}$.
GL_DST_COLOR	Factor is equal to the destination color vector $\bar{C}_{destination}$
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector: $1 - \bar{C}_{destination}$.
GL_SRC_ALPHA	Factor is equal to the $alpha$ component of the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to $1 - alpha$ of the source color vector $\bar{C}_{source}$.
GL_DST_ALPHA	Factor is equal to the $alpha$ component of the destination color vector $\bar{C}_{destination}$.
GL_ONE_MINUS_DST_ALPHA	Factor is equal to $1 - alpha$ of the destination color vector $\bar{C}_{destination}$.
GL_CONSTANT_COLOR	Factor is equal to the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_COLOR	Factor is equal to 1 - the constant color vector $\bar{C}_{constant}$.
GL_CONSTANT_ALPHA	Factor is equal to the $alpha$ component of the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_ALPHA	Factor is equal to $1 - alpha$ of the constant color vector $\bar{C}_{constant}$.

Alpha Blending in OpenGL (cont.)

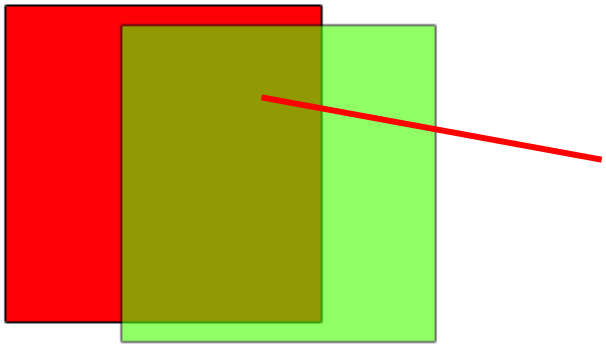
- Example

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

GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA



RGBA = (1.0, 0.0, 0.0, 1.0) (0.0, 1.0, 0.0, 0.6)



(0.0, 1.0, 0.0, 0.6) * **0.6** +
(1.0, 0.0, 0.0, 1.0) * **(1 – 0.6)**

= (0.4, 0.6, 0.0, 0.76)

Alpha Blending in OpenGL

- Implementation
  - In the CPU (OpenGL) program, turn on the following setting if you want to render a transparent object

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

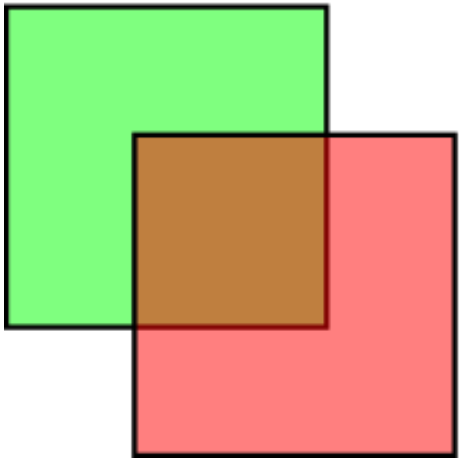
source factor destination factor

  - In the shader, set the transparency when outputting color

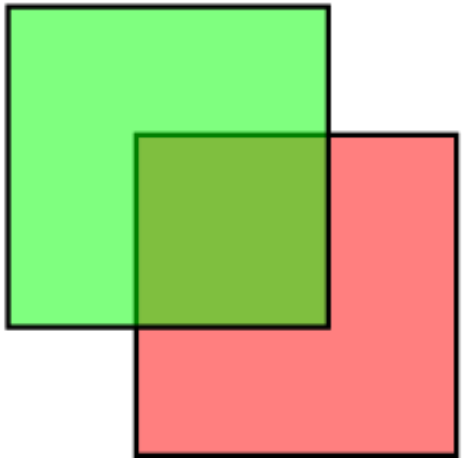
```
void main()
{
    vec3 texColor = texture2D(mapKd, iTexCoord).rgb;
    FragColor = vec4(iLightingColor * texColor, 0.5);
}
```

Rendering Algorithm for Transparency

- Render **opaque** objects first **in any order**
- Render **transparent** objects **in an order w.r.t their distance to the camera** (farther objects first)
- **The rendering order does matter!**



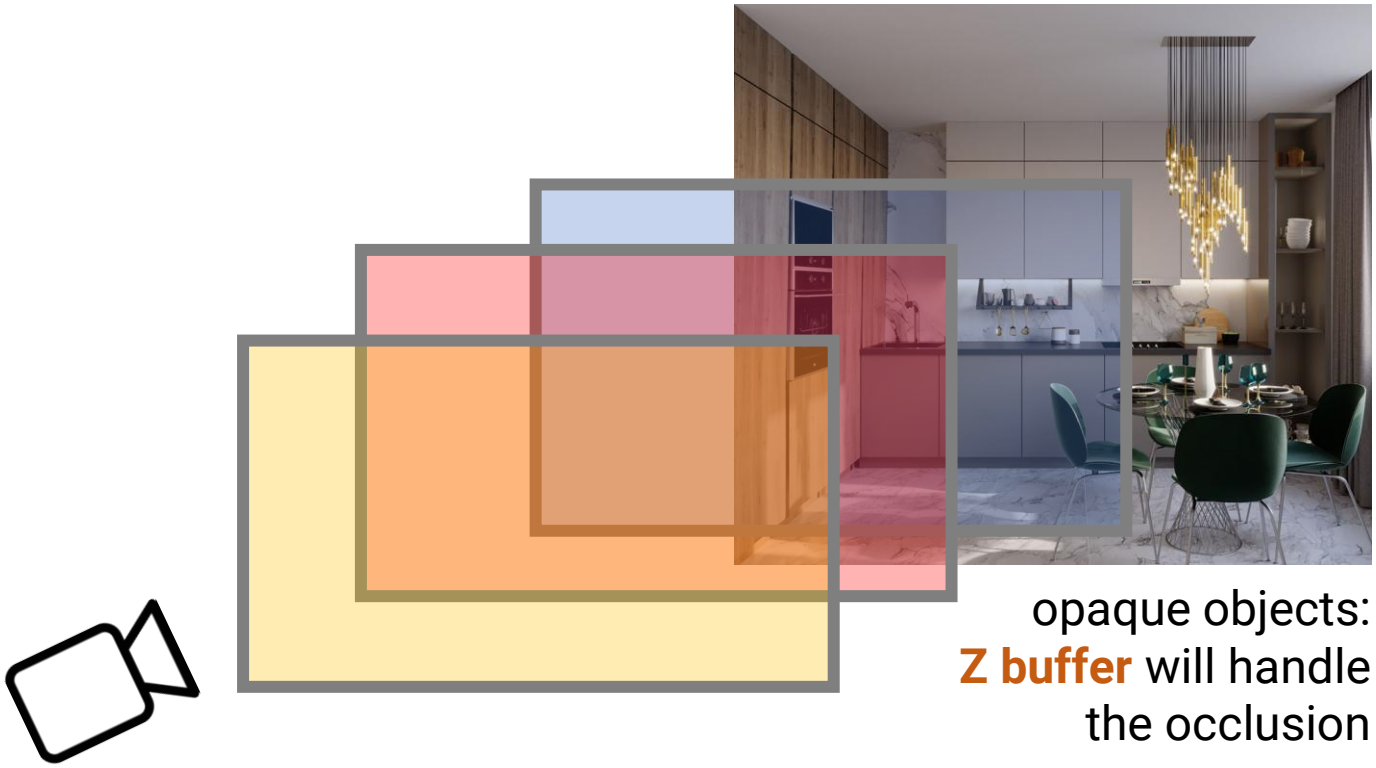
red on top



green on top

Rendering Algorithm for Transparency (cont.)

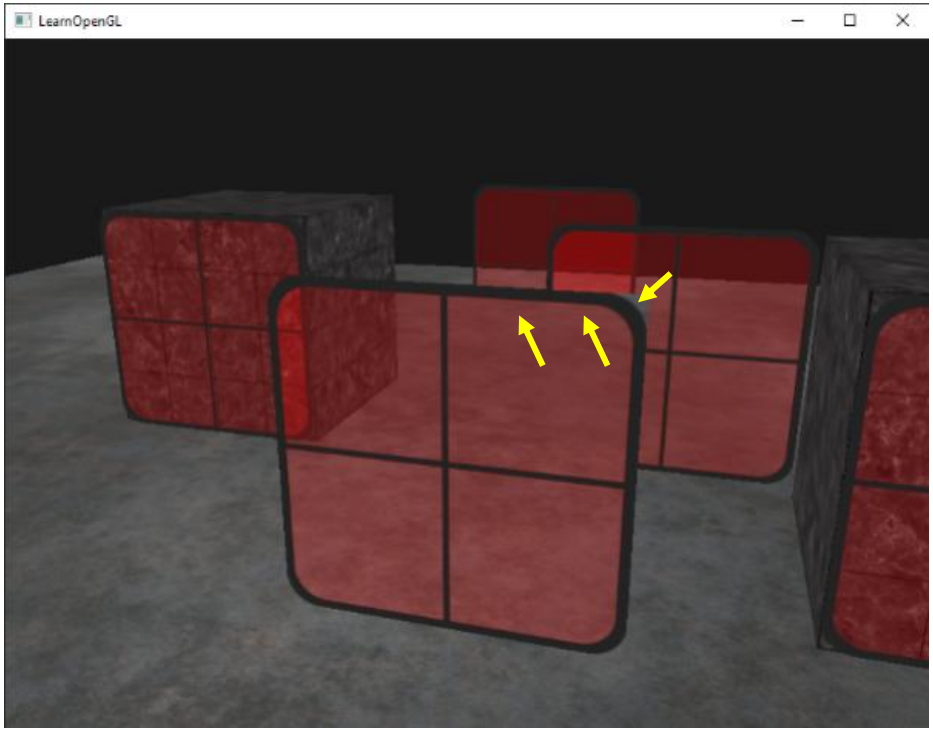
- Render **opaque** objects first **in any order**
- Render **transparent** objects **in an order w.r.t their distance to the camera** (farther objects first)

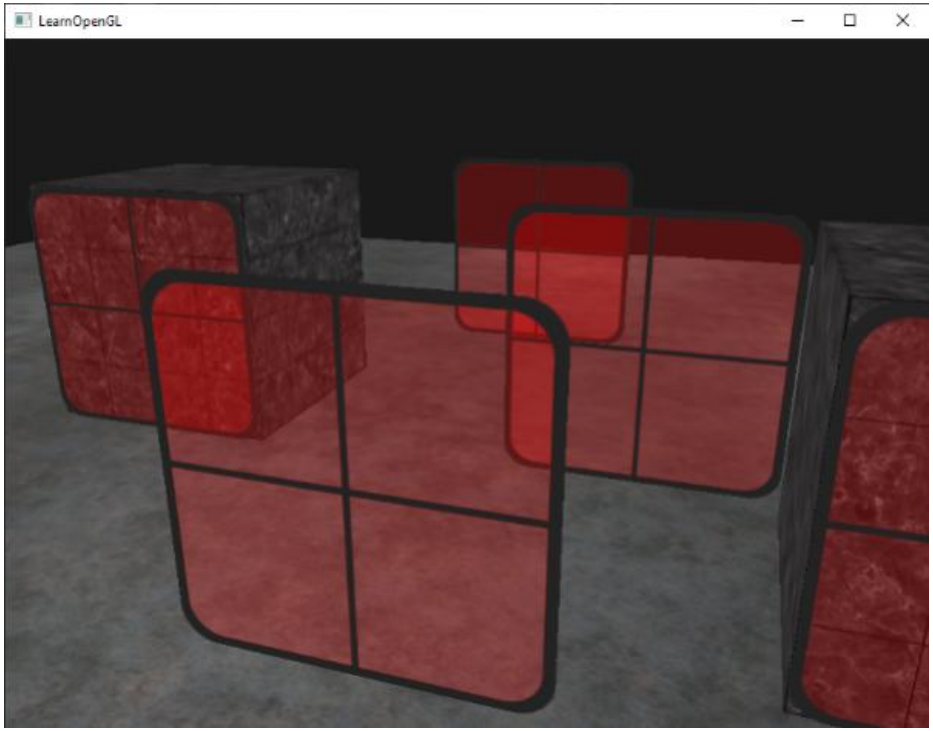


opaque objects:
Z buffer will handle
the occlusion

Rendering Algorithm for Transparency (cont.)

- An example demonstrating incorrect rendering orders





Rendering Algorithm for Transparency (cont.)

- However, in practice you will find it might not work correctly non-planar objects

Why?

- Non-planar objects have **back faces**
- The triangles **in an object** can be rendered in any order (in parallel)
- If a front face is rendered before the back faces behind it, **it will blend with the background**
- If a front face is rendered after the back faces behind it, **it will blend with the back faces**

using back-face culling can help

Rendering Algorithm for Transparency (cont.)

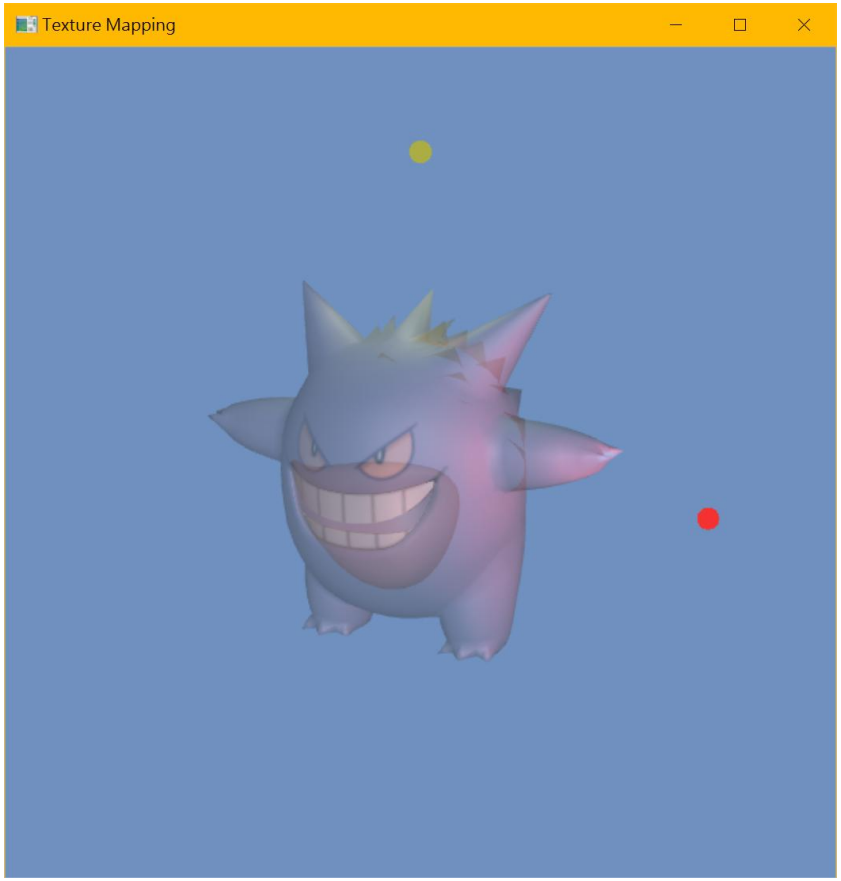
- However, in practice you will find it might not work correctly non-planar objects

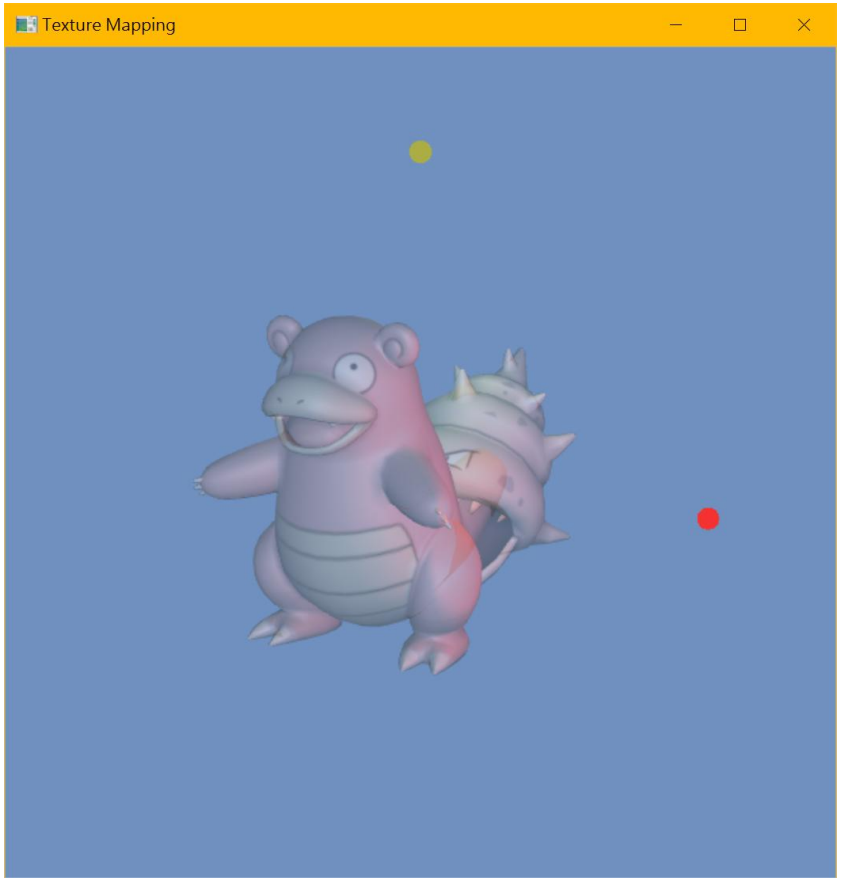
enable back-face culling

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CCW);
```

Rendering Algorithm for Transparency (cont.)

- Cannot avoid rendering inner geometry



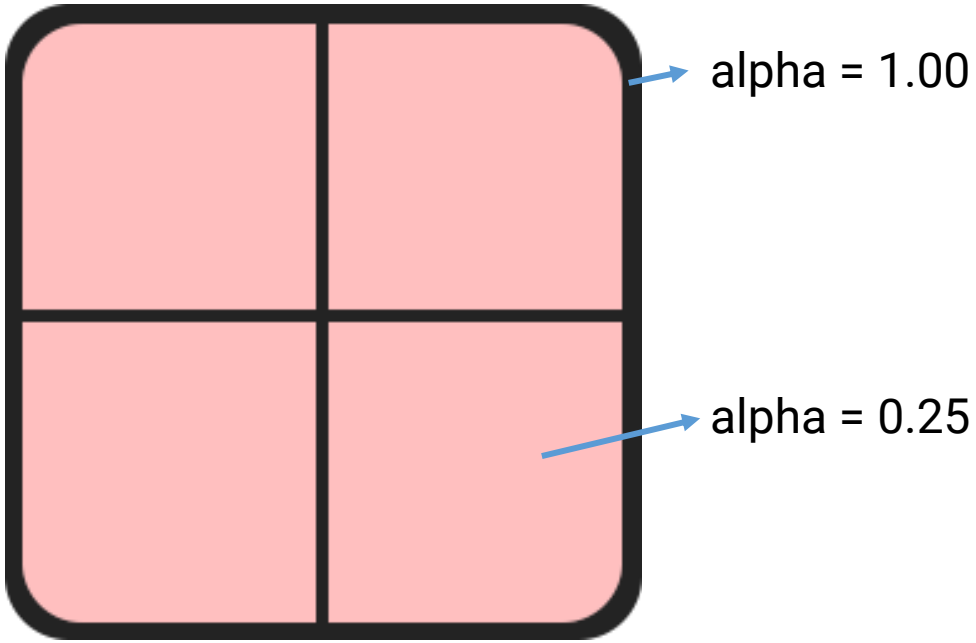


Rendering Algorithm for Transparency (cont.)

- Transparency for arbitrary objects
  - Per-frame objects/triangles sorting is expensive (especially when the scene has many transparent objects)
  - There are some papers addressing this issue; however, with a **large overhead**
    - *Order-Independent transparency for Programmable deferred shading Pipelines* (Pacific Graphics 2015)
    - Techniques using ***depth peeling***
  - In practice, game designers will limit the maximum number of transparent objects in a scene

Alpha Value in Texture

- To represent **spatially varying** transparency, some textures have an embedded **alpha channel** in addition to the red, green, and blue channels



Alpha Testing

- A special case in that we only have two types of alpha values in a texture
  - Fully opaque (alpha = 1.0)
  - Fully transparent (alpha = 0.0)

Alpha Testing in OpenGL

- In the fragment shader

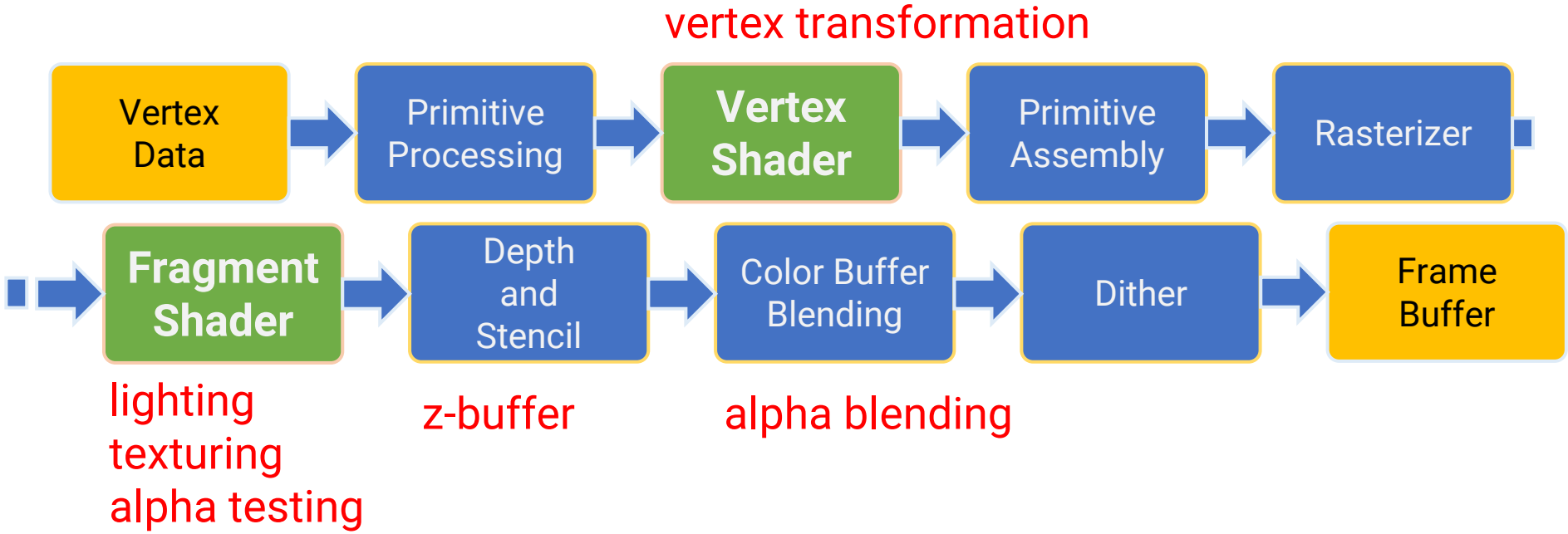
```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

- **Cut follow-up computation for this fragment**
- **Avoid writing Z-buffer in fully-transparent part**

Alpha Testing in OpenGL (cont.)

Review: GPU Graphics Pipeline

- We have introduced the most parts of the GPU rendering pipeline



- In the next slides, we will start to introduce some effects that need multi-pass rendering


END OF PRESENTATION
ANY QUESTIONS?