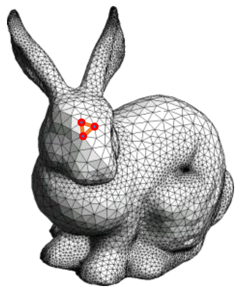
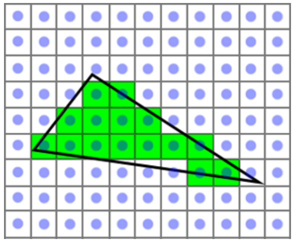
Deferred Shading

Computer Graphics

Yu-Ting Wu

Forward Rendering





vertex transform

Vertex Data → Primitive Processing → **Vertex Shader** → Primitive Assembly → Rasterizer →

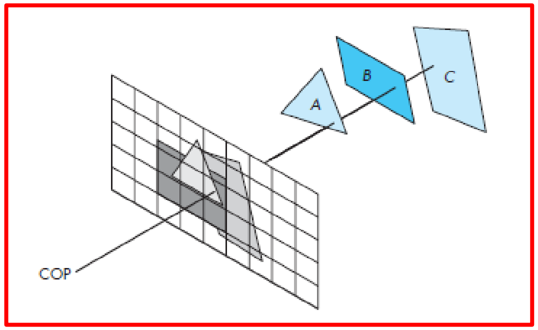
→ **Fragment Shader** → Depth and Stencil → Color Buffer Blending → Dither → Frame Buffer

lighting
texturing

fragments are discarded if they failed in the depth test!

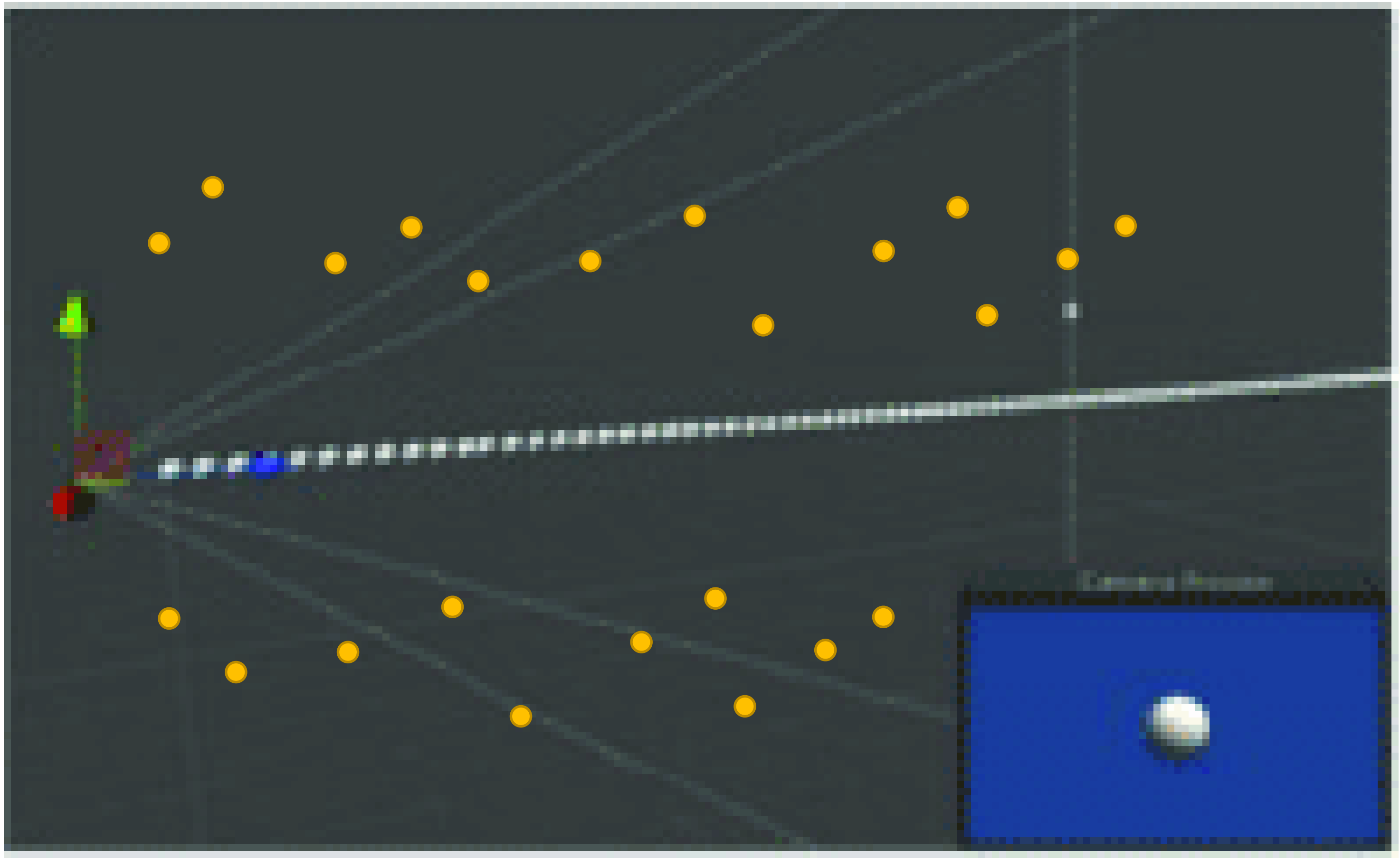
Problem of Forward Rendering

- In scenes with **many** lights and **complex** layouts, lots of computation resources are wasted on shading the **occluded** surfaces that will finally be discarded!



Problem of Forward Rendering (cont.)

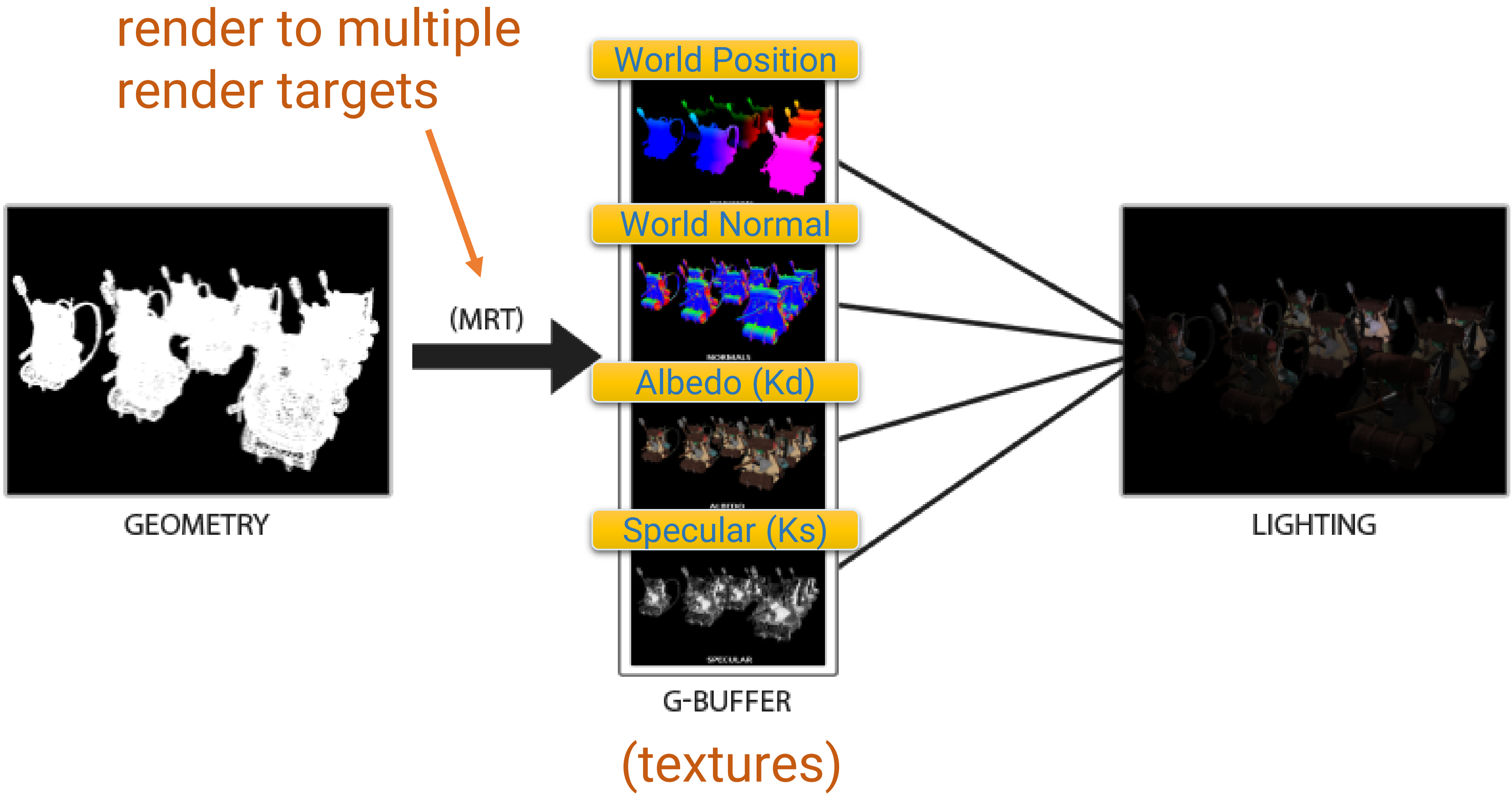
- Overdraw per pixel!



Deferred Shading

- A **Two-pass** rendering algorithm
- In the first pass, recognize all **visible** surfaces from the camera, store their **geometry** and **material** properties in **geometry buffers (G-buffers)**
- In the second pass, only compute lighting on the visible surfaces based on the G-buffers

Deferred Shading (cont.)

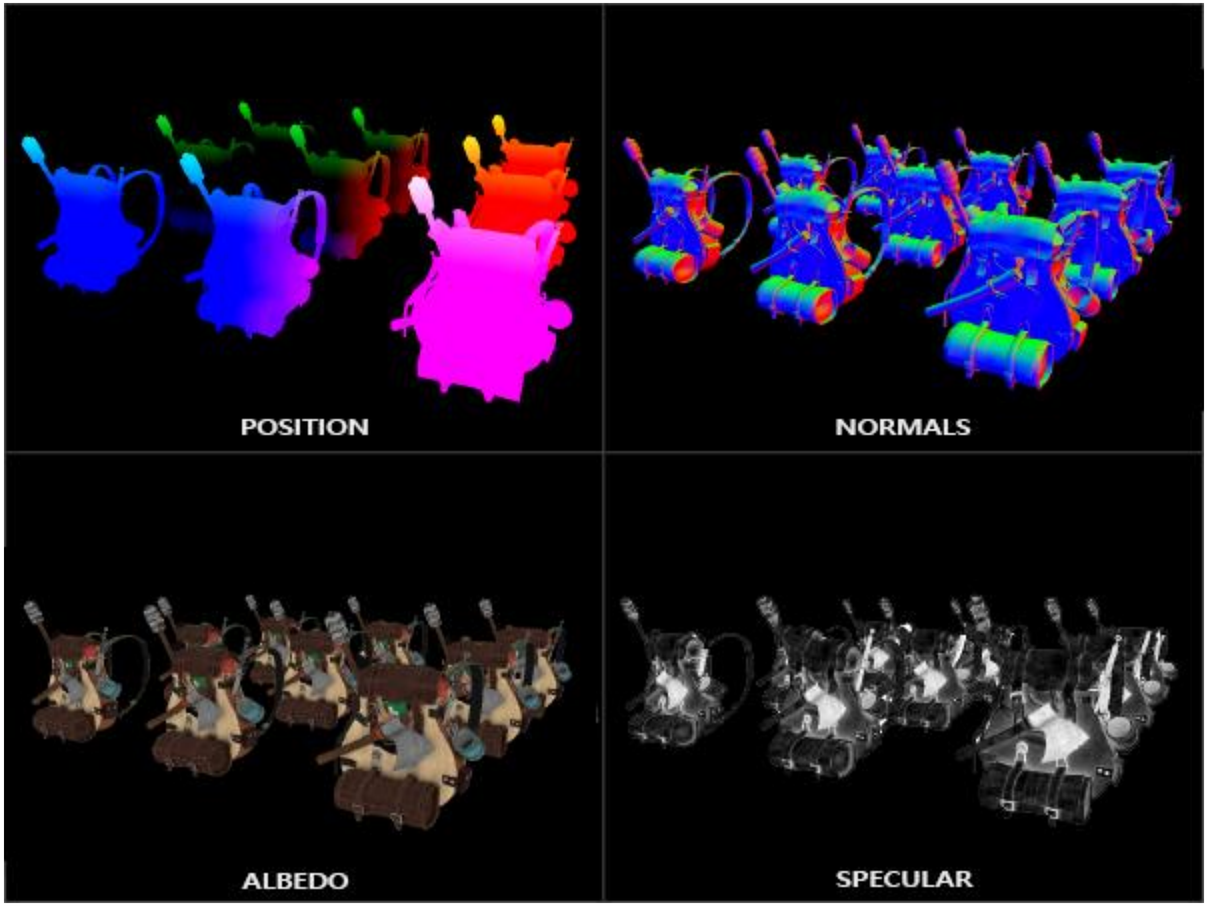


First Pass: Geometry Buffer Creation

- Observation: the surfaces shown on the screen are the visible surfaces from the camera
- We can obtain the geometry and material data of visible surfaces by **rendering** the scene **into textures**
  - **Z buffer** will keep the closest surfaces to the camera for us
- During rendering, the fragment shader outputs the surfaces' **geometry data** (**world-space position** and **normal**, **texture coordinate**) and **material data** (coefficients of diffuse and specular shading) as **color**
  - Current graphics hardware allows us for creating **multiple render targets** (possible to render multiple textures in a render pass)

First Pass: Geometry Buffer Creation (cont.)

- An example of G-buffers

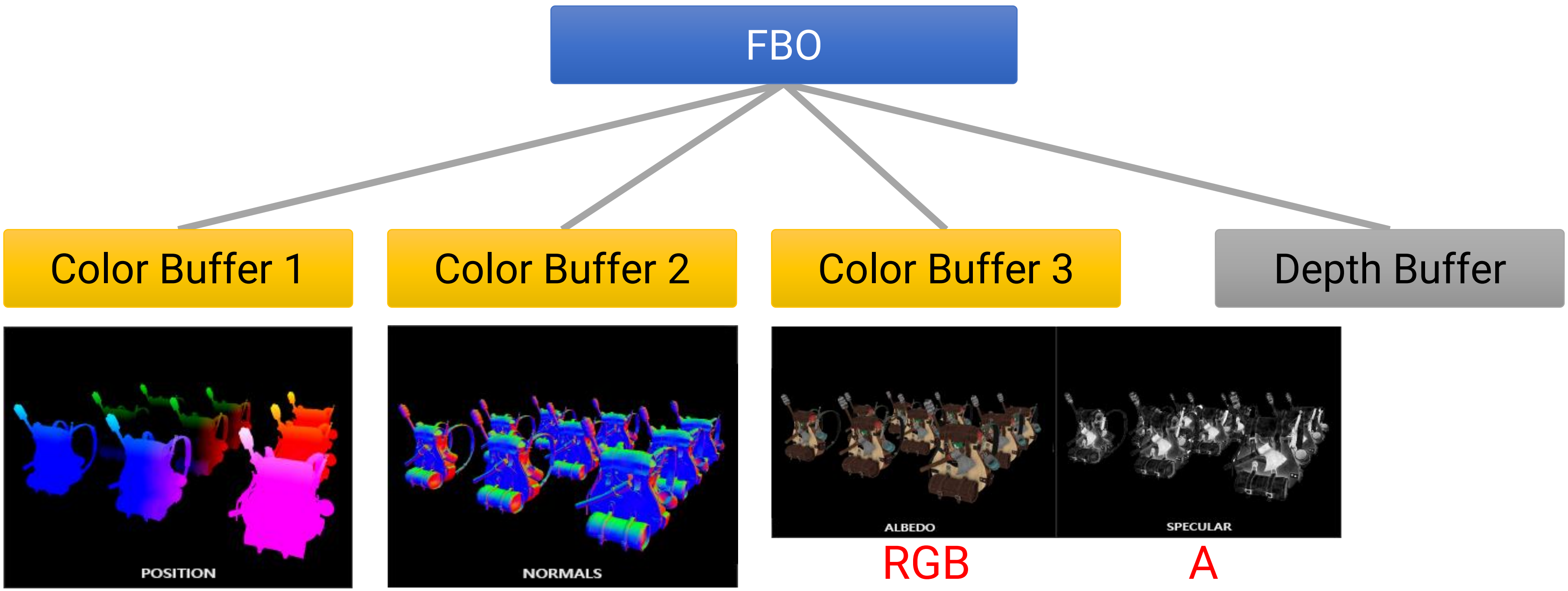


First Pass: Geometry Buffer Creation (cont.)

- Implementation
  - **Frame Buffer Objects (FBO)**
    - The results of the 3D pipeline in OpenGL end up in something which is called a frame buffer object (FBO)
    - When **glutInitDisplayMode()** is called, it creates the default frame buffer using the specified parameters. This framebuffer is managed by the windowing system and cannot be deleted by OpenGL
    - Programmers can create additional FBOs of their own, and render content into the buffers
    - Like the default frame buffer, an FBO consists of **color** and **depth** attachment

First Pass: Geometry Buffer Creation (cont.)

- Implementation
  - **Frame Buffer Objects (FBO)**



Multiple Render Target

draw 3 color images and 1 depth image in one rendering pass

First Pass: Geometry Buffer Creation (cont.)

- Implementation
  - https://learnopengl.com/Advanced-Lighting/Deferred-Shading

```
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
unsigned int gPosition, gNormal, gColorSpec;

// - position color buffer
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, gPosition, 0);

// - normal color buffer
glGenTextures(1, &gNormal);
glBindTexture(GL_TEXTURE_2D, gNormal);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, gNormal, 0);
```

create FBO

Generate textures for storing position, normal, and material data

attach a texture to an FBO

void glTexImage2D(target, level, internalformat, width, height, border, format, type, data);

First Pass: Geometry Buffer Creation (cont.)

- Implementation
  - https://learnopengl.com/Advanced-Lighting/Deferred-Shading

```
// color + specular color buffer
glGenTextures(1, &gAlbedoSpec);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, gAlbedoSpec, 0);
// tell OpenGL which color attachments we'll use (of this framebuffer) for rendering
unsigned int attachments[3] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, attachments);
// create and attach depth buffer
unsigned int rboDepth;
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, SCR_WIDTH, SCR_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboDepth);
// finally check if framebuffer is complete
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "Framebuffer not complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

specifies a list of color buffers to be drawn into

create a depth buffer for the FBO

First Pass: Geometry Buffer Creation (cont.)

- **Vertex Shader:** transform vertex and pass interpolated data
- **Fragment Shader:**

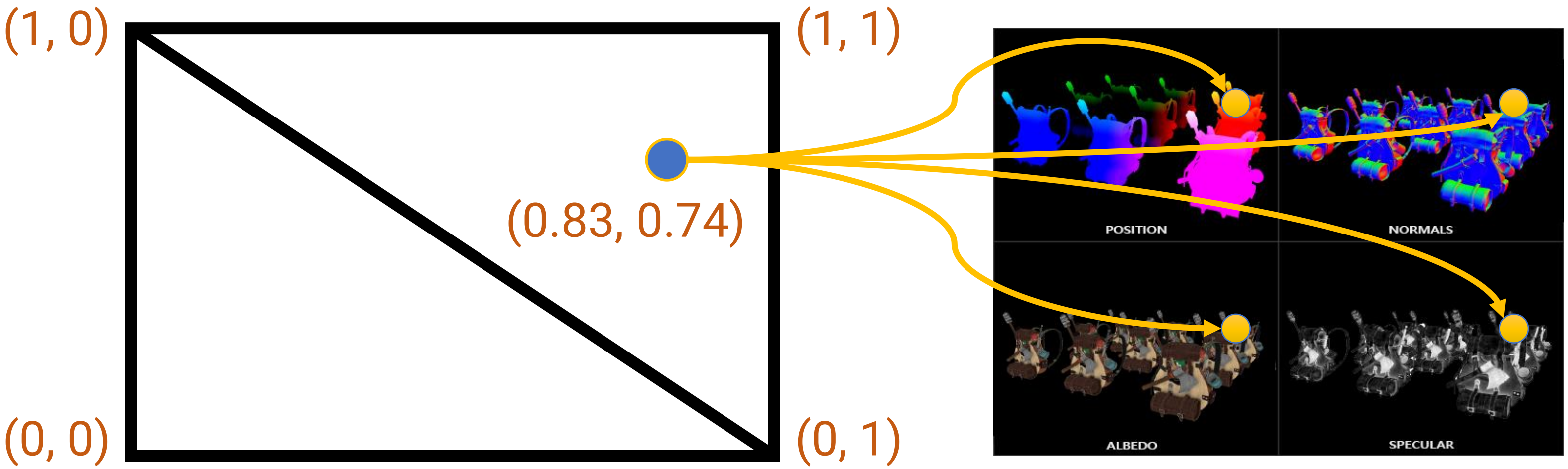
```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

uniform sampler2D texture_diffuse1;
uniform sampler2D texture_specular1;

void main()
{
    // store the fragment position vector in the first gbuffer texture
    gPosition = FragPos;
    // also store the per-fragment normals into the gbuffer
    gNormal = normalize(Normal);
    // and the diffuse per-fragment color
    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
    // store specular intensity in gAlbedoSpec's alpha component
    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

output three images in one pass

interpolated data from Vertex Shader

Second Pass: Compute Lighting

- Render a screen-sized quad
- Pass all lights using uniform variables or textures to the fragment shader
- In the fragment shader, lookup the G-buffers for per-pixel geometry and material data
- Compute lighting with all lights



Second Pass: Compute Lighting (cont.)

- Implementation
  - https://learnopengl.com/Advanced-Lighting/Deferred-Shading

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gPosition);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
// also send light relevant uniforms
shaderLightingPass.use();
SendAllLightUniformsToShader(shaderLightingPass);
shaderLightingPass.setVec3("viewPos", camera.Position);
RenderQuad();
```

Second Pass: Compute Lighting (cont.)

- **Vertex Shader:** transform vertex (quad) and pass interpolated data
- **Fragment Shader:**

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;
};
const int NR_LIGHTS = 32;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;
```

we only need interpolated texture coordinates because position and normal are stored in G-buffers

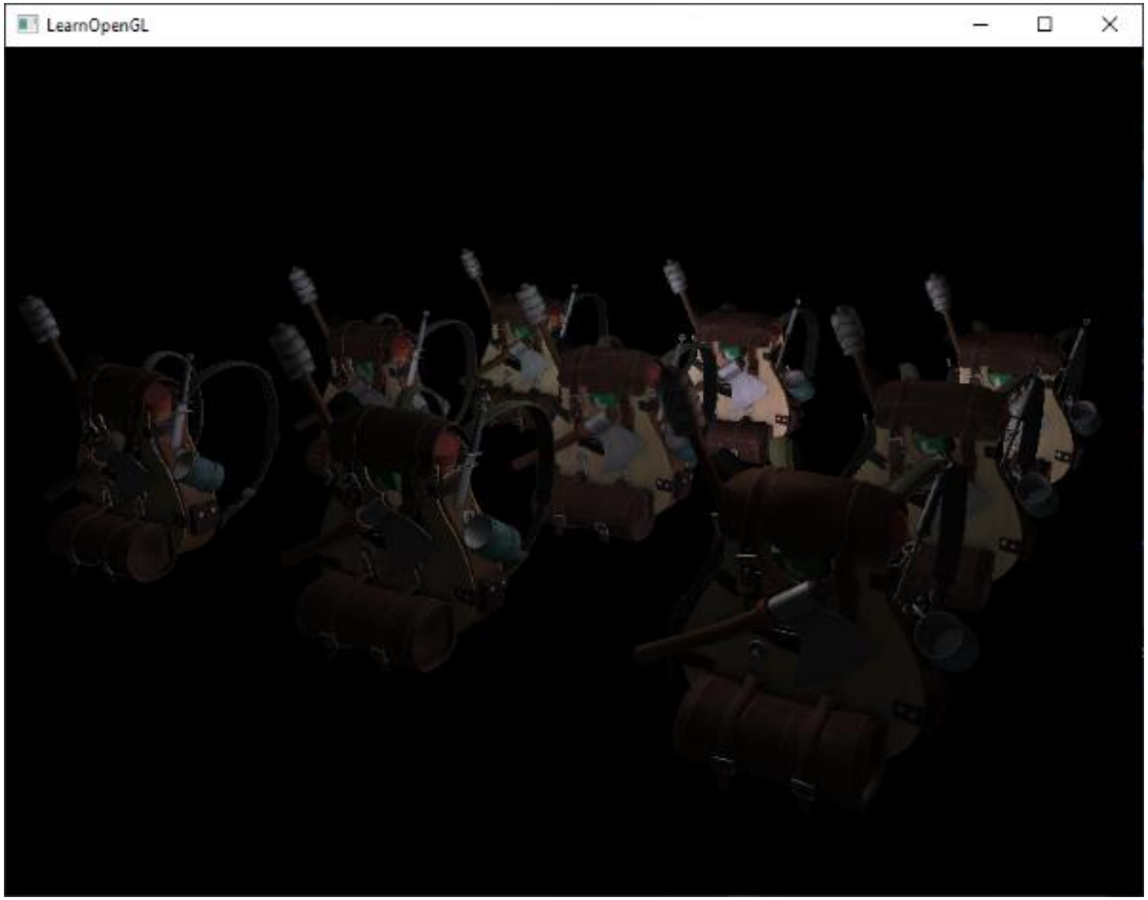
the generated G-buffers as textures

Second Pass: Compute Lighting (cont.)

```
void main()
{
    // retrieve data from G-buffer
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Albedo = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;

    // then calculate lighting as usual
    vec3 lighting = Albedo * 0.1; // hard-coded ambient component
    vec3 viewDir = normalize(viewPos - FragPos);
    for(int i = 0; i < NR_LIGHTS; ++i)
    {
        // diffuse
        vec3 lightDir = normalize(lights[i].Position - FragPos);
        vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Albedo * lights[i].Color;
        lighting += diffuse;
    }

    FragColor = vec4(lighting, 1.0);
}
```

Second Pass: Compute Lighting (cont.)

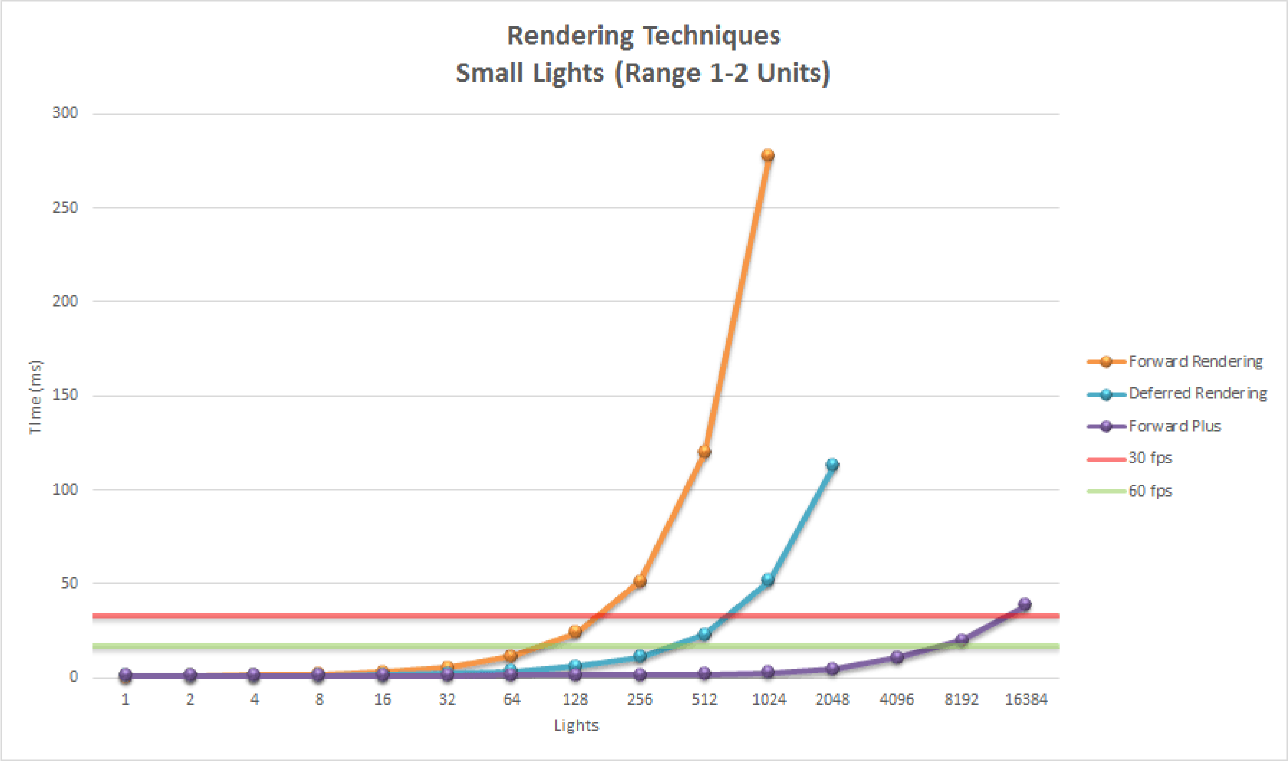
- Render a scene with 32 lights



Deferred Shading in Unreal Engine 4

Discussion: Pros

- **Reduce unnecessary lighting computation**
  - Can achieve significant performance improvement in complex scenes with massive lights

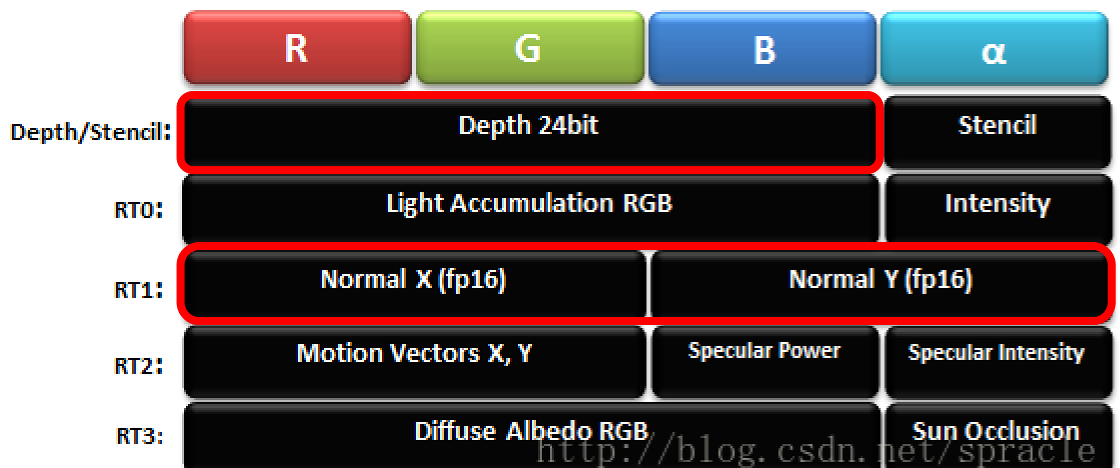


Discussion: Cons

- **Larger memory bandwidth**
  - The storage of G-buffers takes lots of GPU memory
  - Laborious for **mobile devices**
  - Assume 10 textures are used (assume RGBA16F)

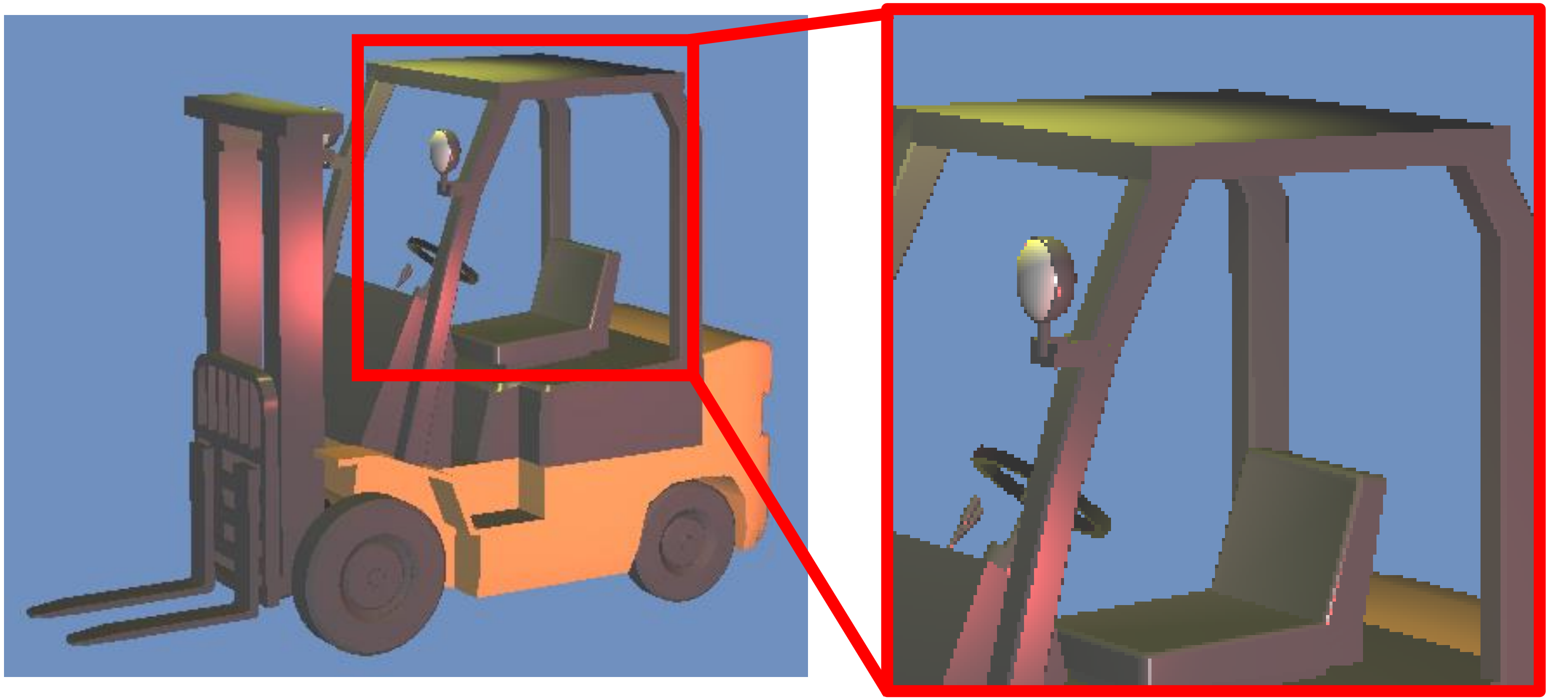
$$10 \times 1920 \times 1080 \times 4 * 16 \text{ bits} = \mathbf{158\ MB}$$

  - Solution: use compact G-buffers
    - Killzone 2



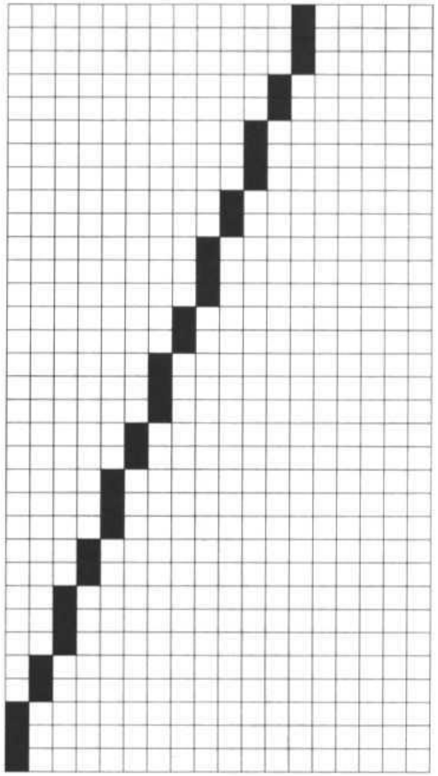
Discussion: Cons (cont.)

- Difficult for **Multi Sampled Anti Aliasing (MSAA)**
- Recap: aliasing



Recap: Aliasing

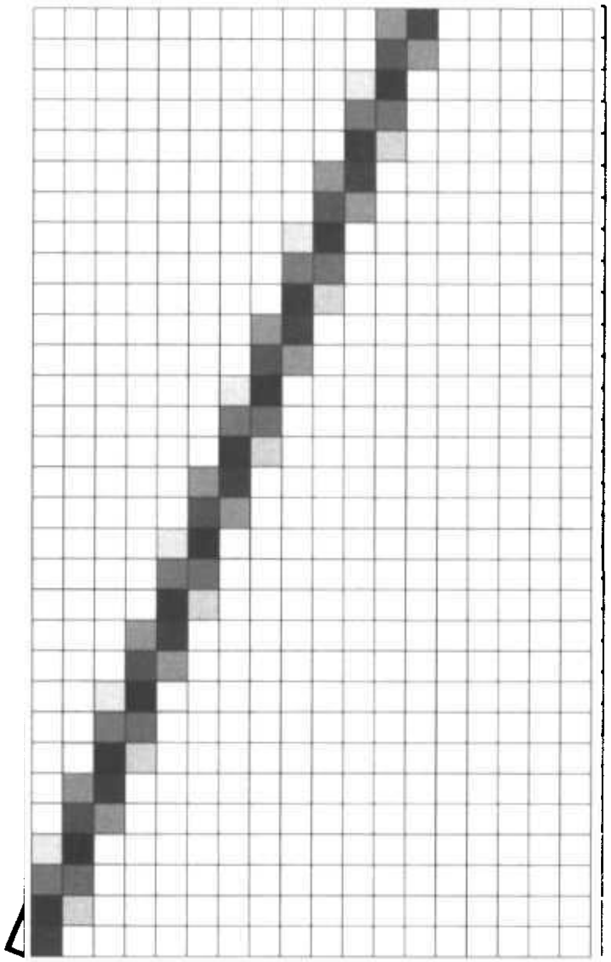
- Rendering a continuous function (e.g., lines, curves) with a discrete representation (pixels) will encounter the aliasing problem
  - Example: $y = 5x/2 + 1$



- Jaggedness is inevitable!
  - Due to the use of a grid of discrete pixels

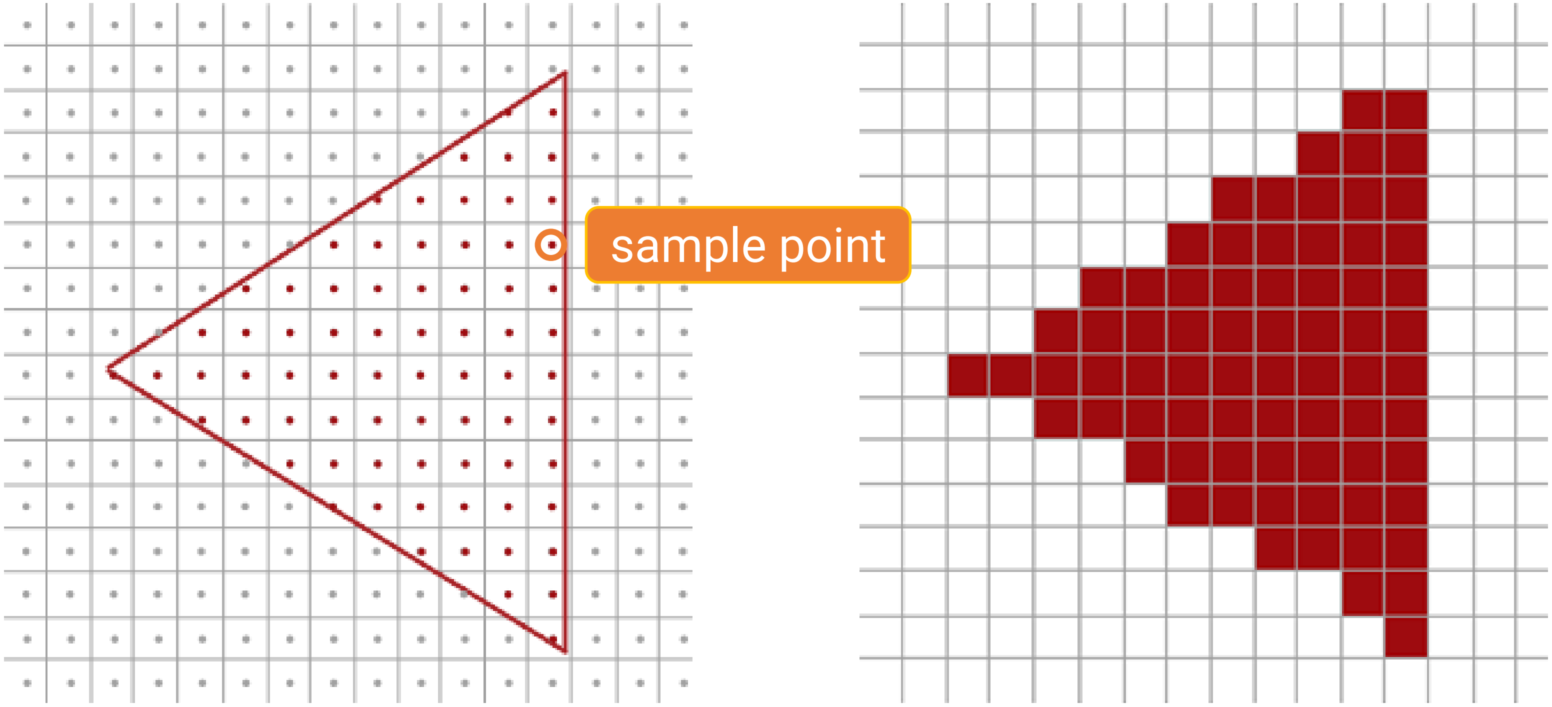
Recap: Anti-aliasing

- Anti-aliasing is a **practical** technique to reduce the jaggies
- Use intermediate grey values
  - In the frequency domain, it relates to reducing the frequency of the signal
- Coloring each pixel in a shade of grey whose **brightness is proportional to the area** of the intersection between the pixels and a "**one-pixel-wide**" line



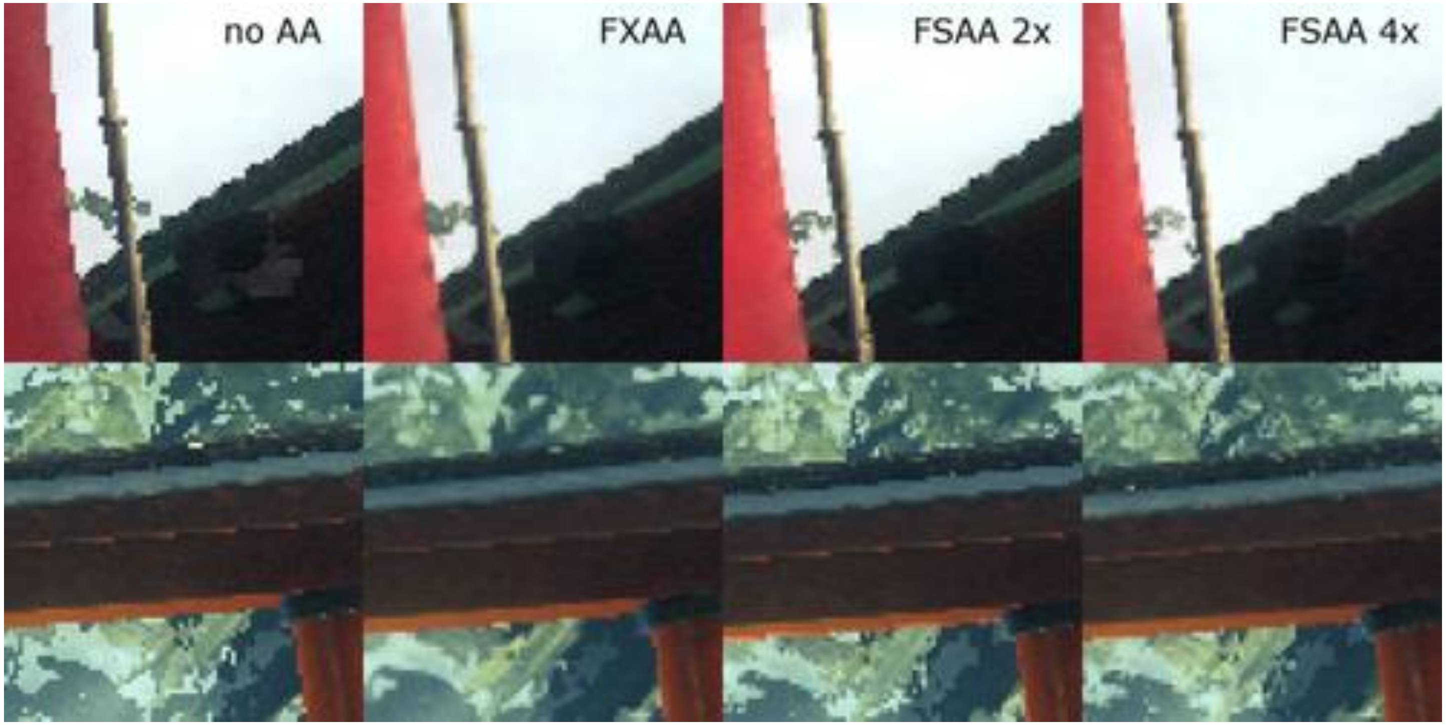
Recap: Aliasing (cont.)

- Aliasing in rasterization
  - Using discrete representation (pixel) to represent continuous signal (triangle)



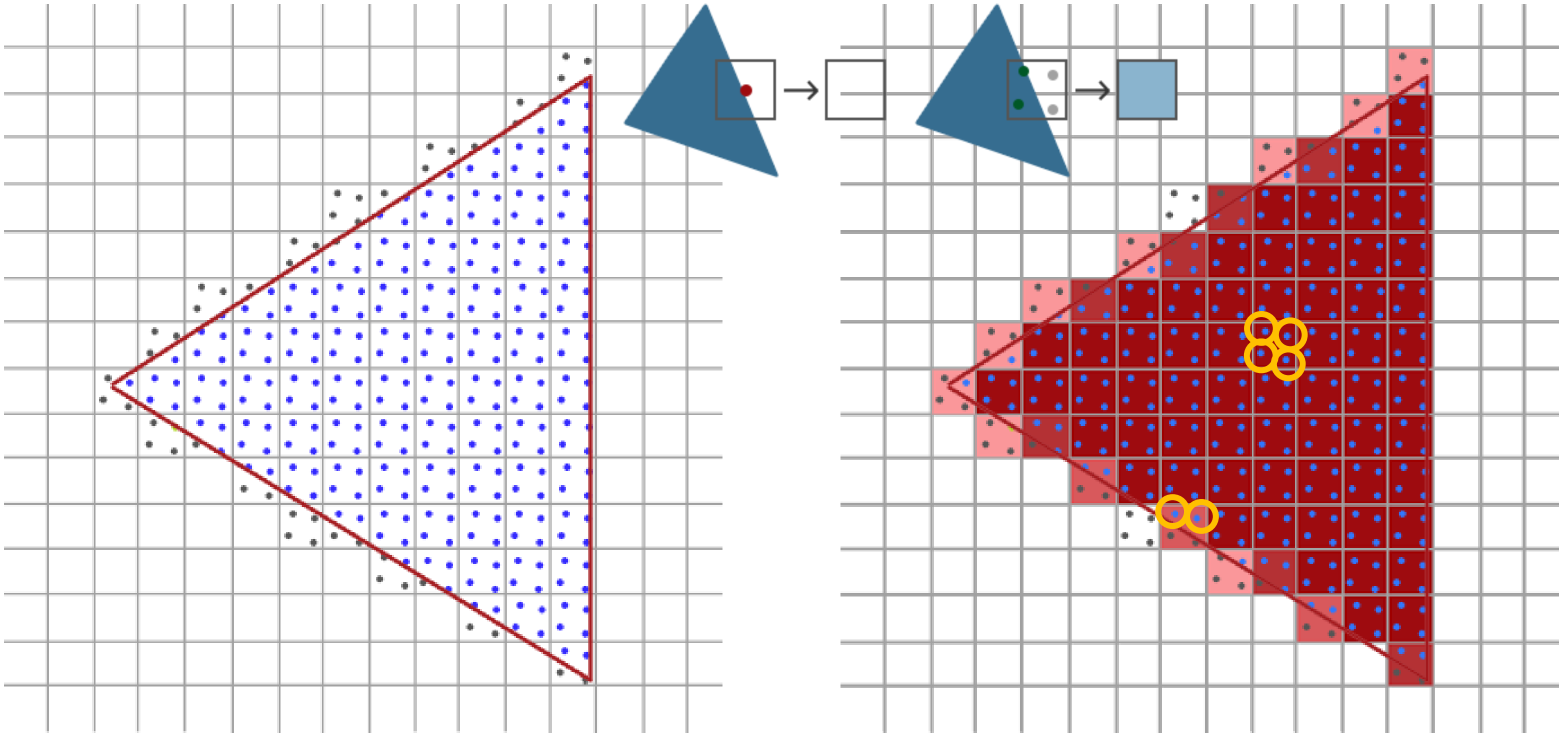
Anti-aliasing

- **Full Scene Anti Aliasing (FSAA)**
  - Render a higher resolution image and do down-sampling
  - Very expensive



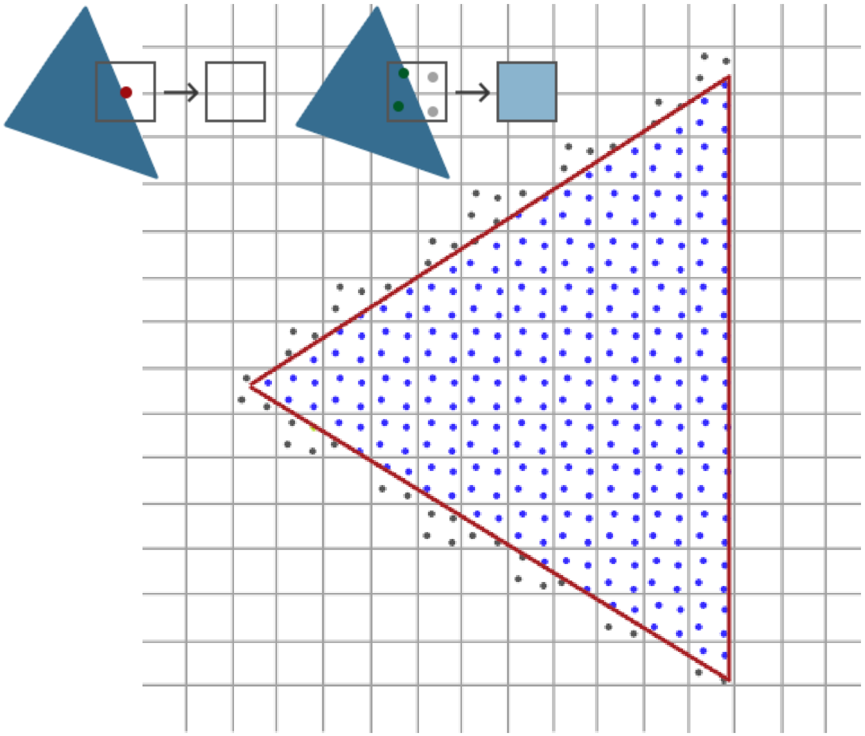
Anti-aliasing (cont.)

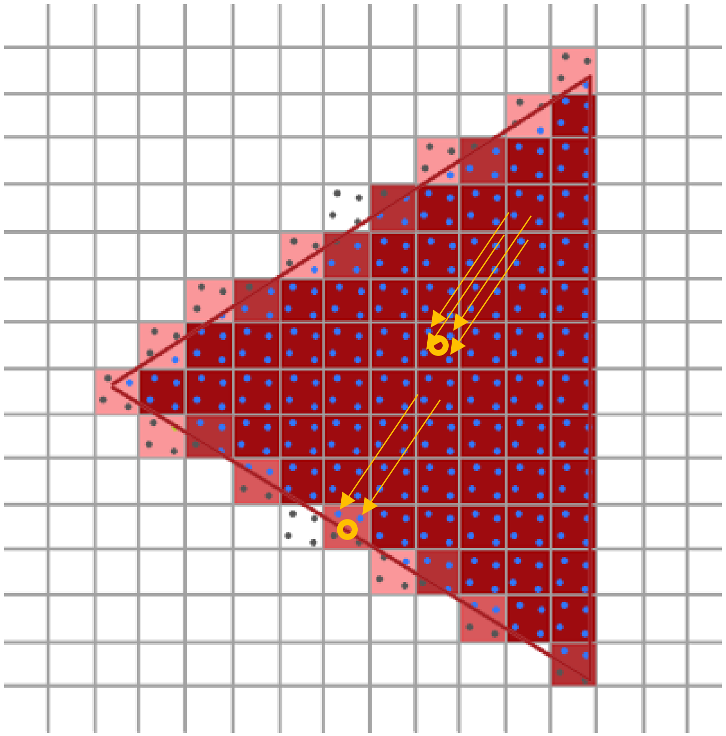
- **Super Sample Anti Aliasing (SSAA)**
  - Multiple locations are sampled within every pixel
  - Also expensive (4× SSAA means 4× fragment computation)



Anti-aliasing (cont.)

- **Multi Sample Anti Aliasing (MSAA)**
  - Multi-samples are only used for determining visibility
  - For each triangle, remain one fragment shader per pixel

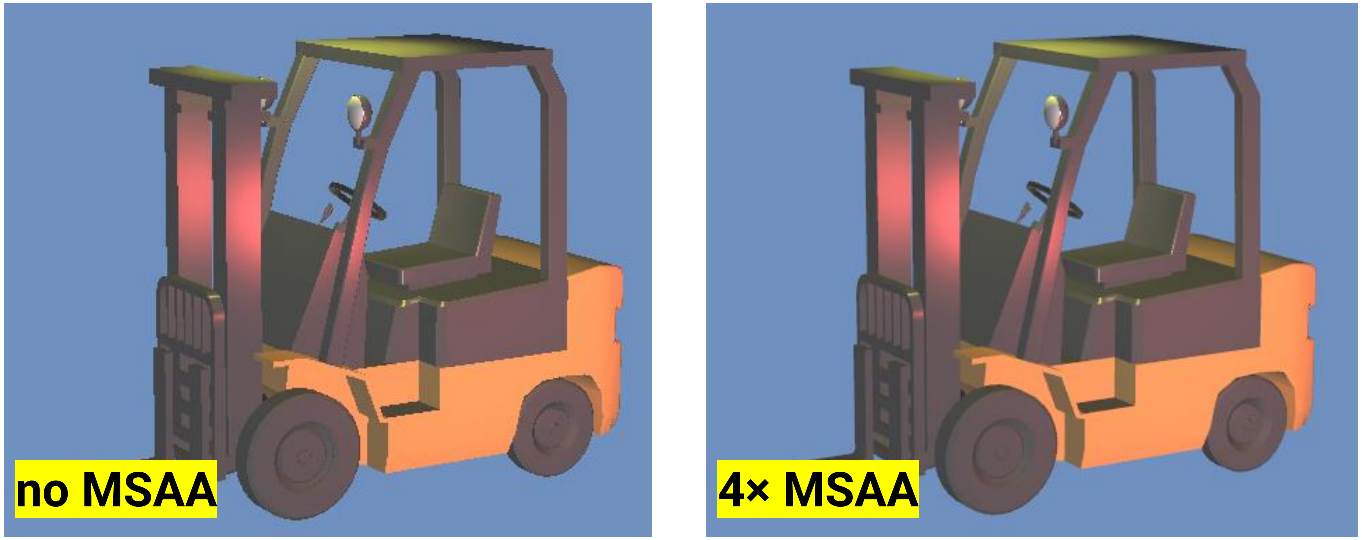




Anti-aliasing (cont.)

- Multi Sample Anti Aliasing (MSAA) in OpenGL
  - Enable MSAA in your FreeGlut project

```
int main(int argc, char** argv)
{
    // Setting window properties.
    glutInit(&argc, argv);
    glutSetOption(GLUT_MULTISAMPLE, 4);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_MULTISAMPLE);
```



Discussion: Cons (cont.)

- MSAA is difficult for deferred shading
    - **Deferred shading decouples geometry process and shading process**
        - Only the closest surface is kept in the G-buffers
    - MSAA requires multiple subpixels information; however, each pixel can store only one value
    - Significantly increase rendering cost if you want to keep more information within the pixel
        - Render and compute lighting with respect to larger-resolution G-buffers

Discussion: Cons (cont.)

- Solution: turn to software algorithms, such as Fast Approximate Anti Aliasing (FXAA)
  - https://www.youtube.com/watch?v=jz_po-QcreU

Discussion: Cons (cont.)

- Cannot handle **transparent** objects
  - Standard G-buffers only store the closest **opaque** surface
  - In practice, the transparent objects are rendered using forward rendering in an alternative pass


END OF PRESENTATION
ANY QUESTIONS?