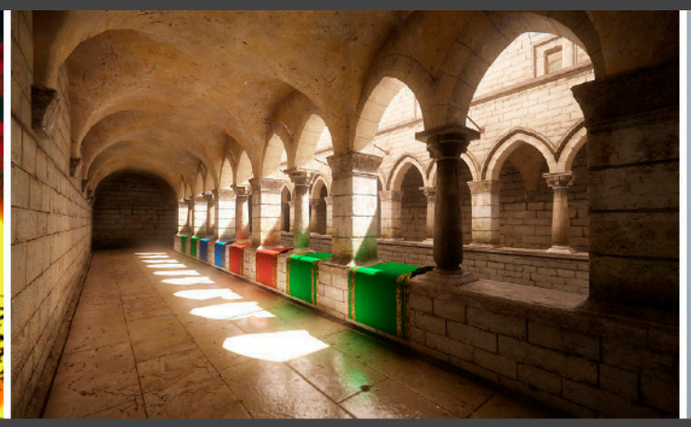
Real-Time High Quality Rendering

GAMES202, Lingqi Yan, UC Santa Barbara

Lecture 2:
Recap of CG Basics

Announcement

- Assignment 0 has been released!
  - Try to play with the code
  - Not required to submit

- Course website is online
  - The PDF slides of lecture 1 has been released!

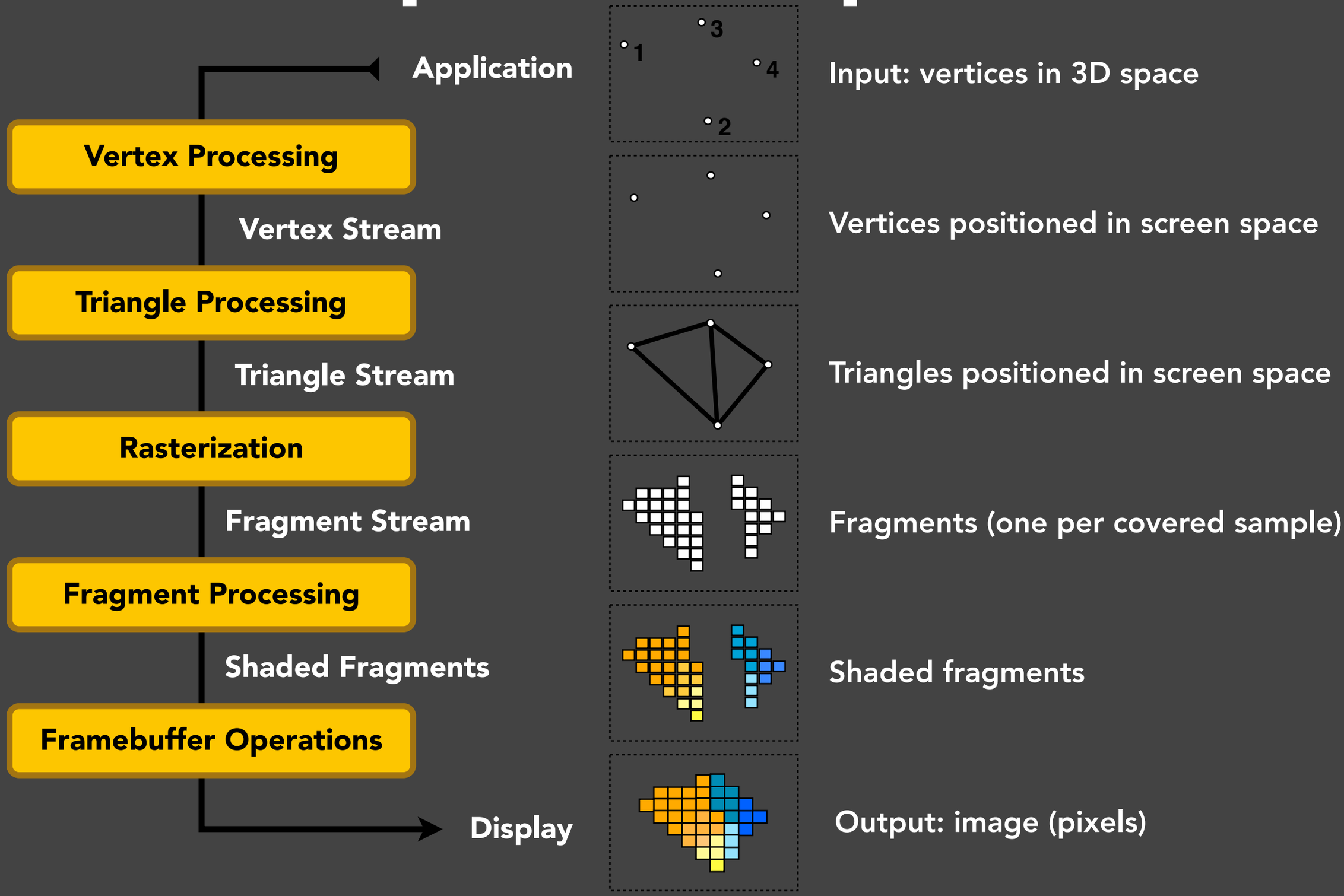
- Week 4 (Apr 6 & 10) might have to be recorded only

Today

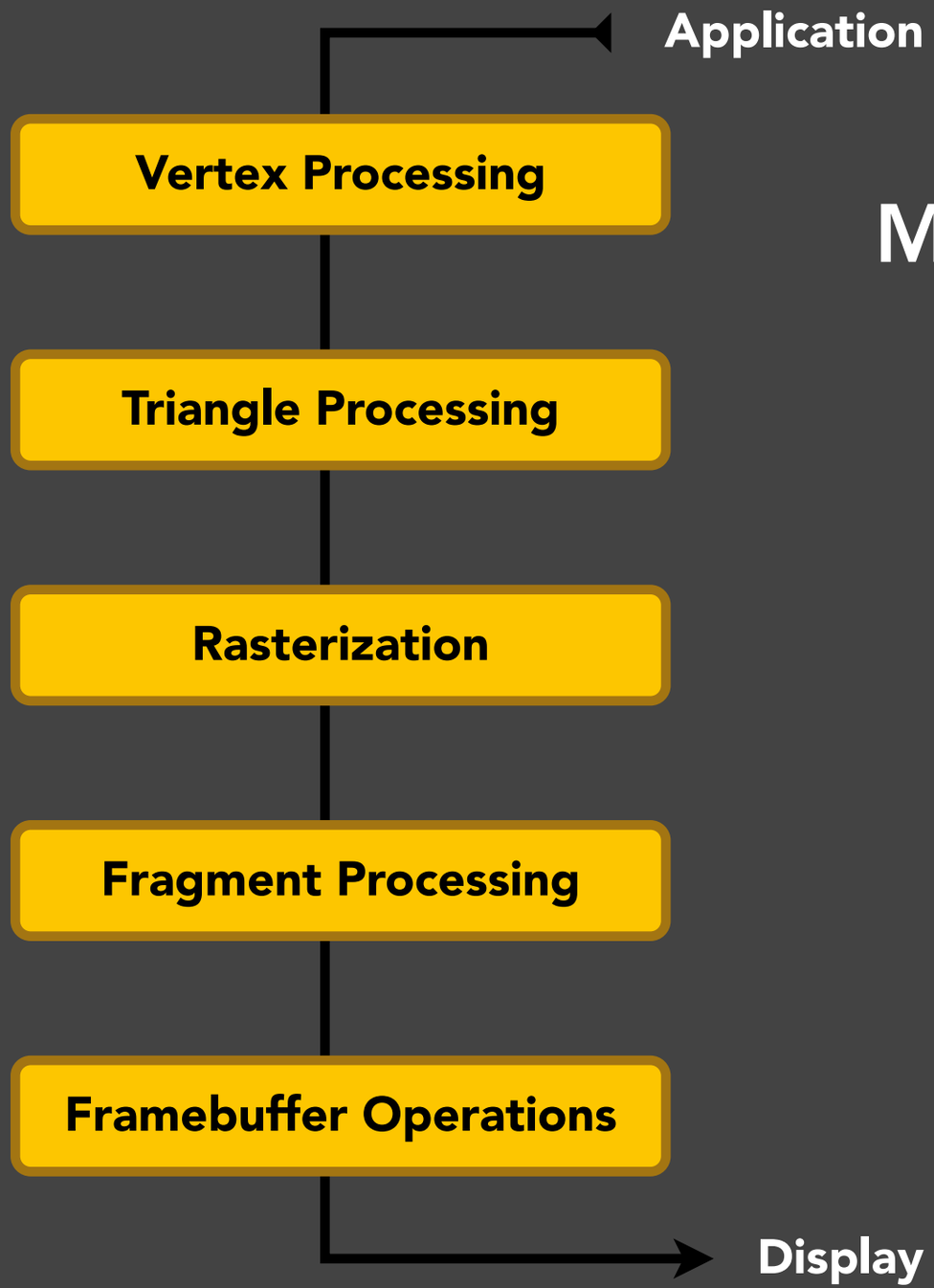
- Recap of CG Basics
  - Basic GPU hardware pipeline
  - OpenGL
  - OpenGL Shading Language (GLSL)
  - The Rendering Equation
  - Calculus

Graphics (Hardware) Pipeline

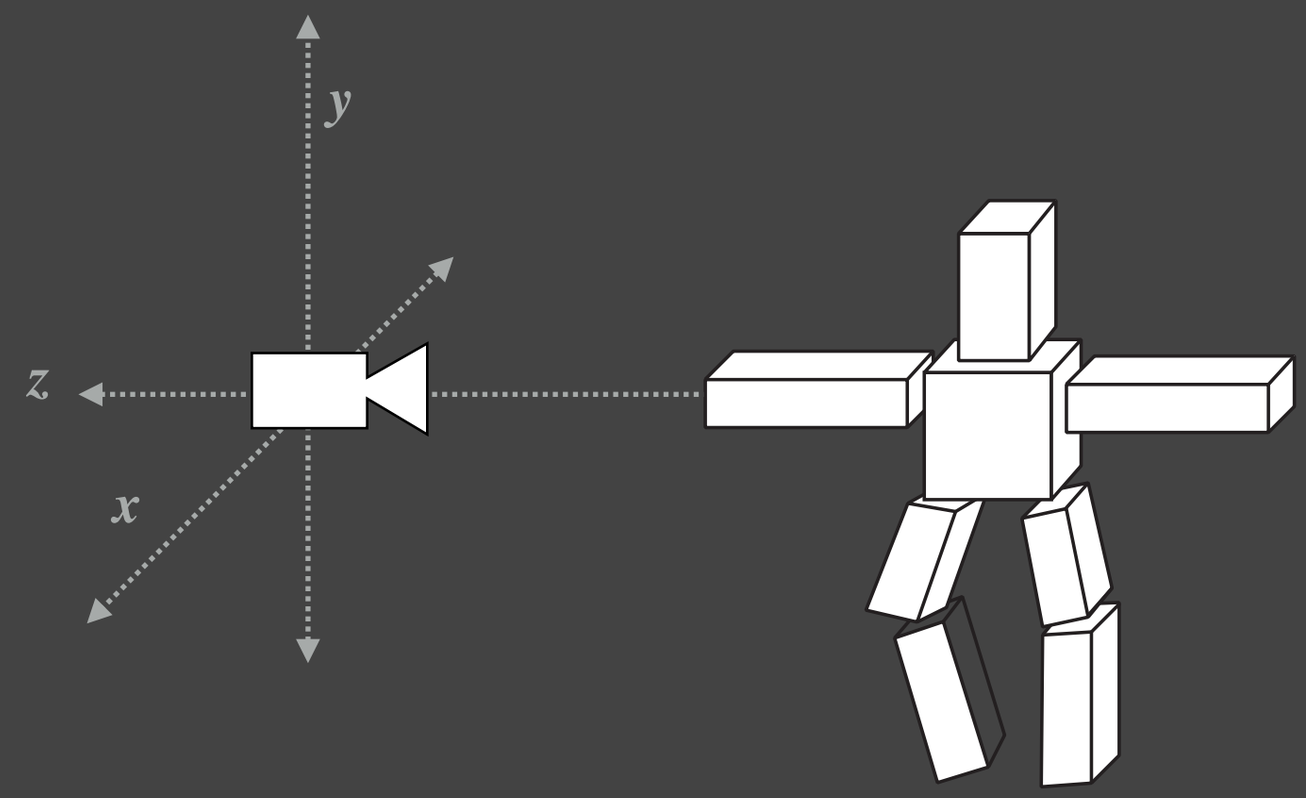
Graphics Pipeline



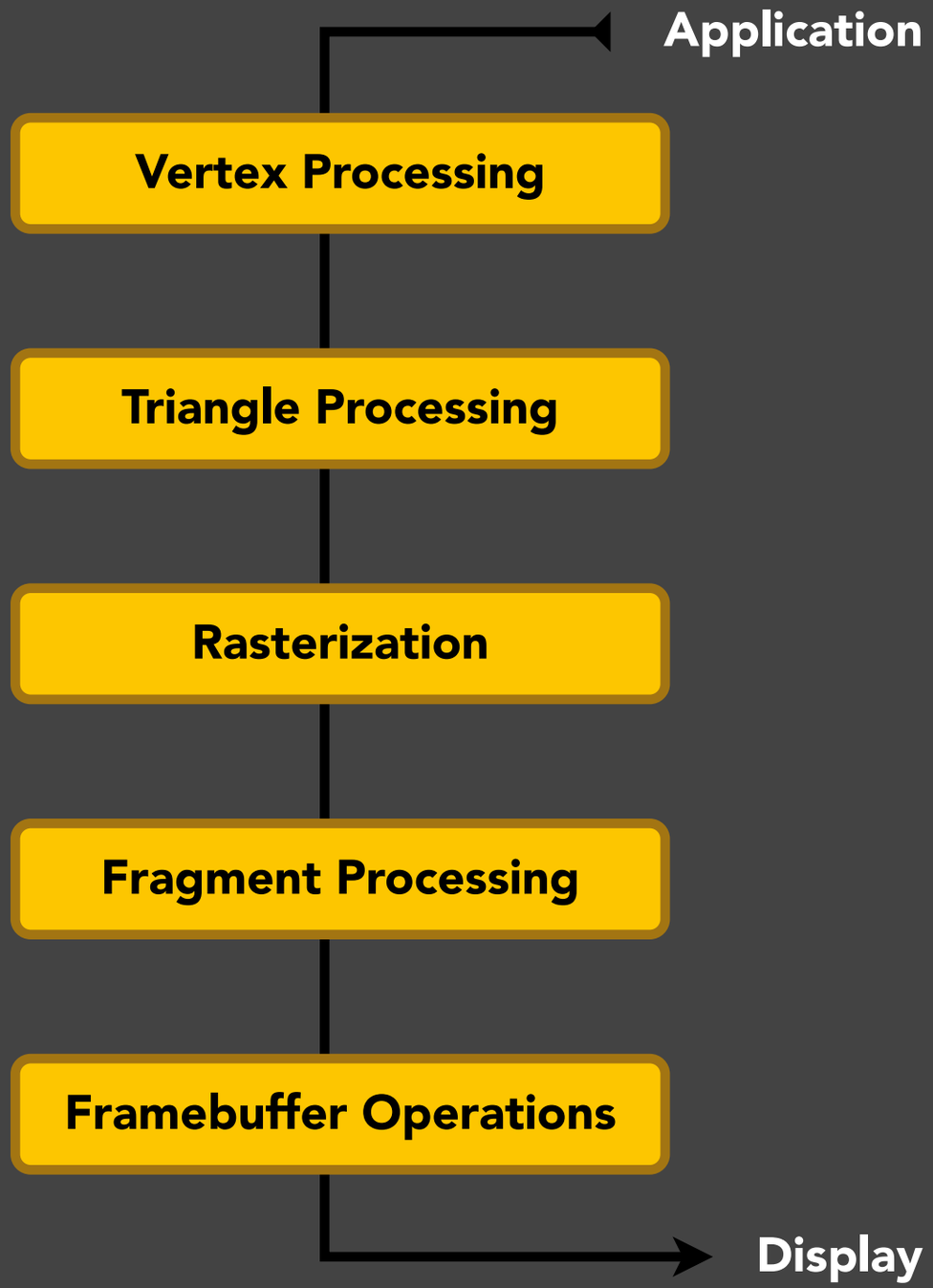
Graphics Pipeline



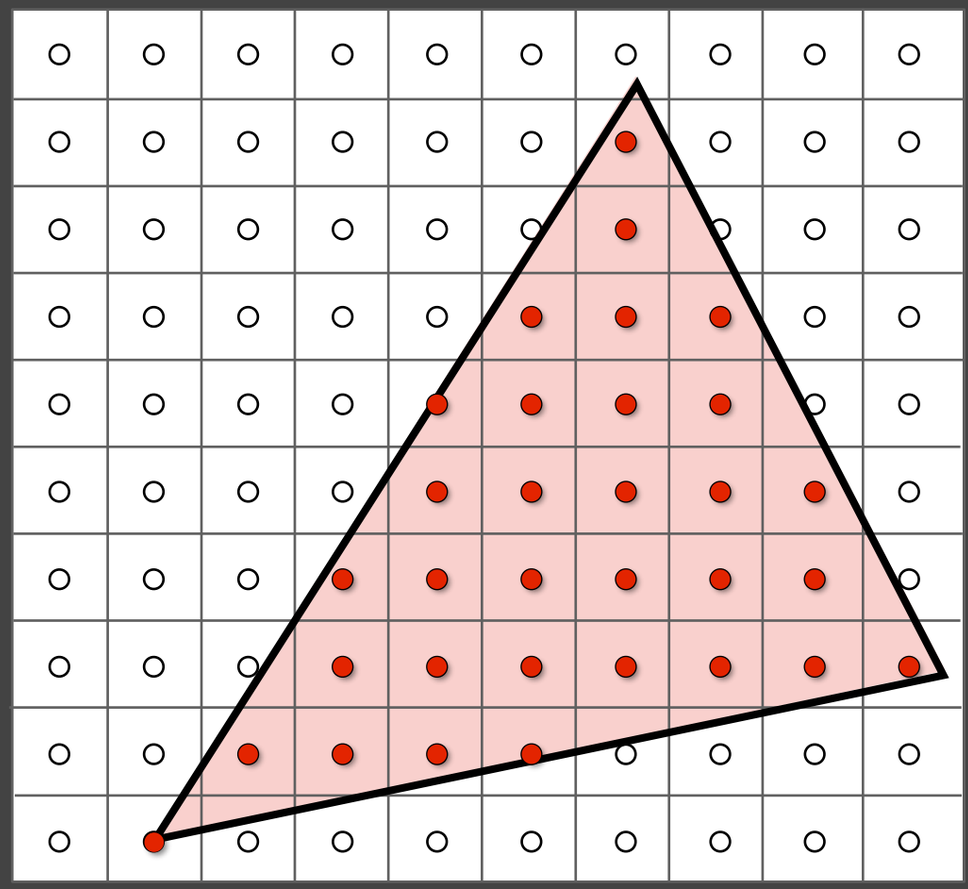
Model, View, Projection transforms



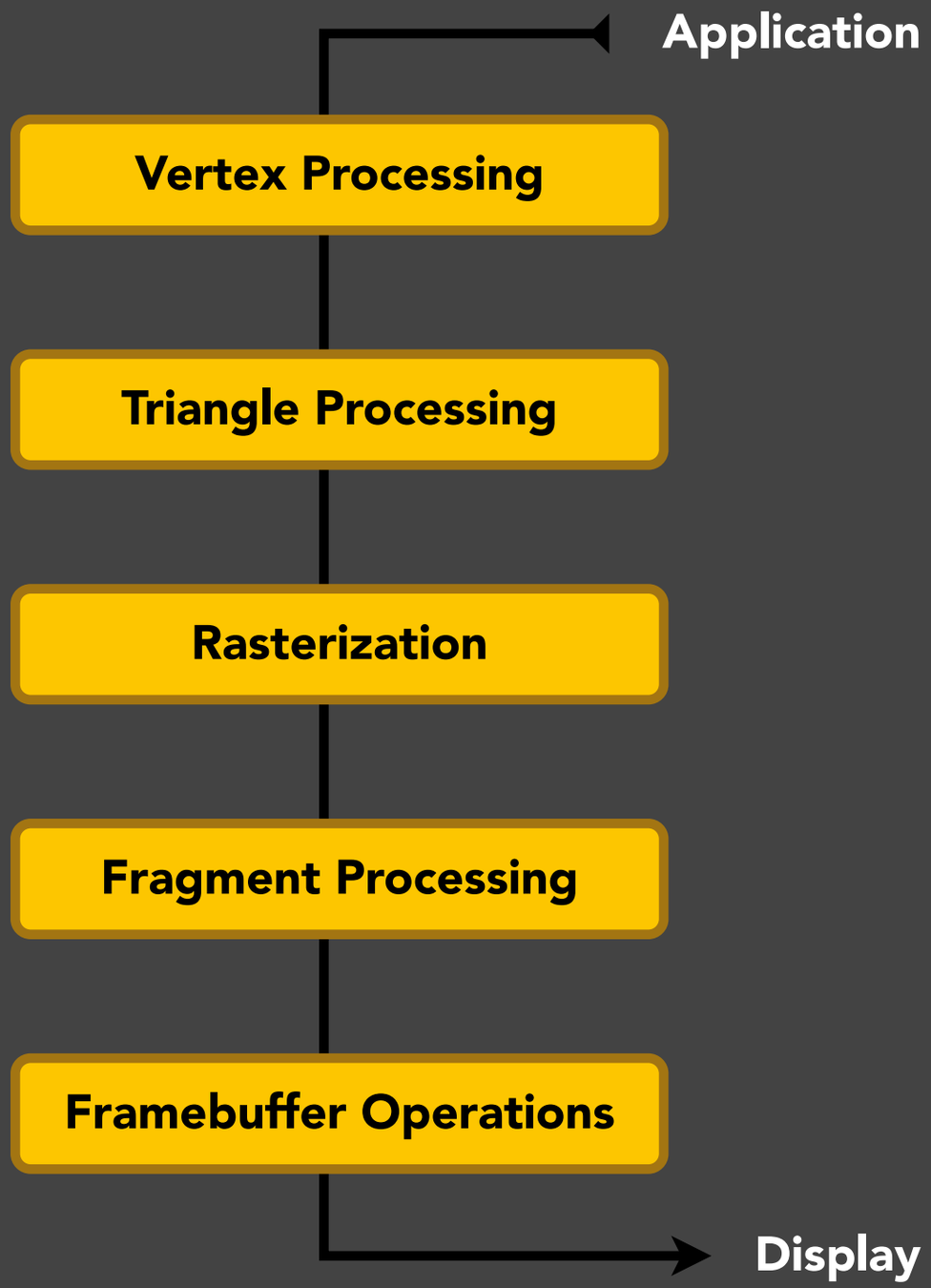
Graphics Pipeline



Sampling triangle coverage

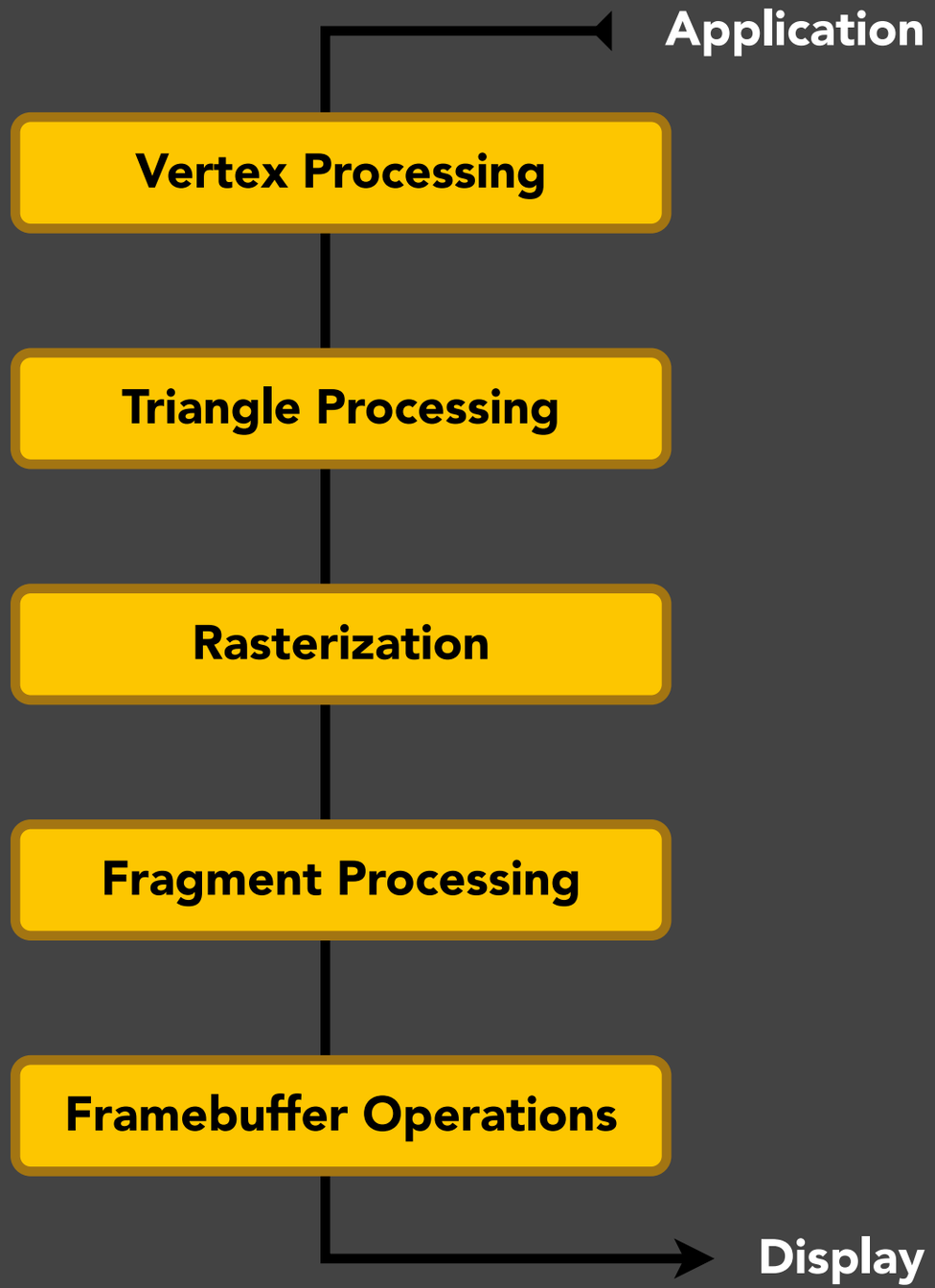


Graphics Pipeline

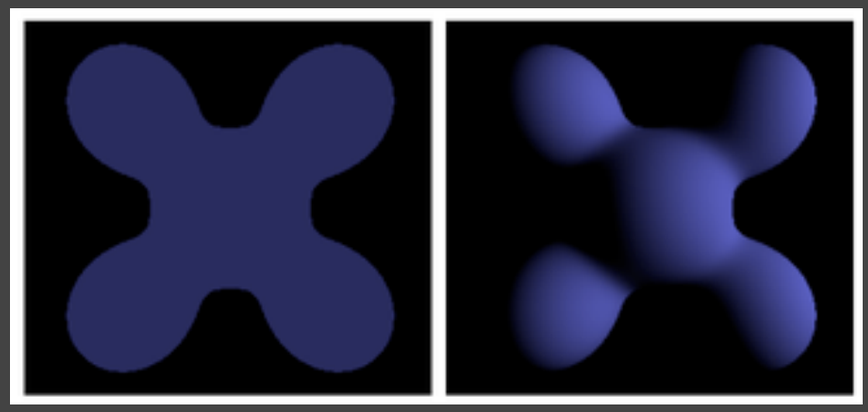


Z-Buffer Visibility Tests

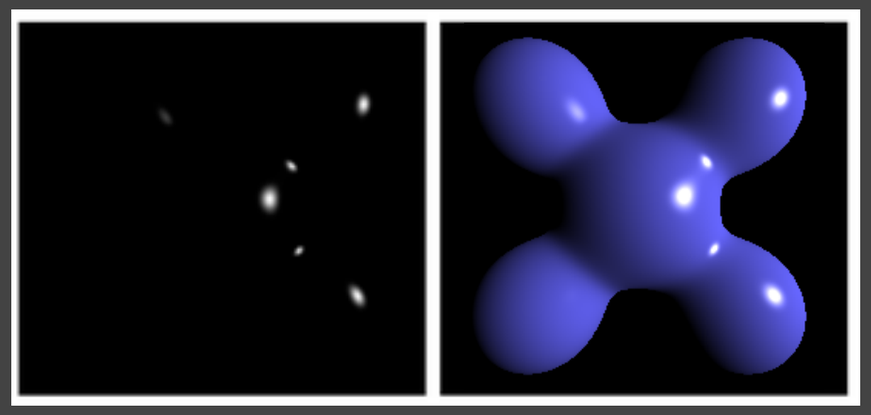
Graphics Pipeline



Shading

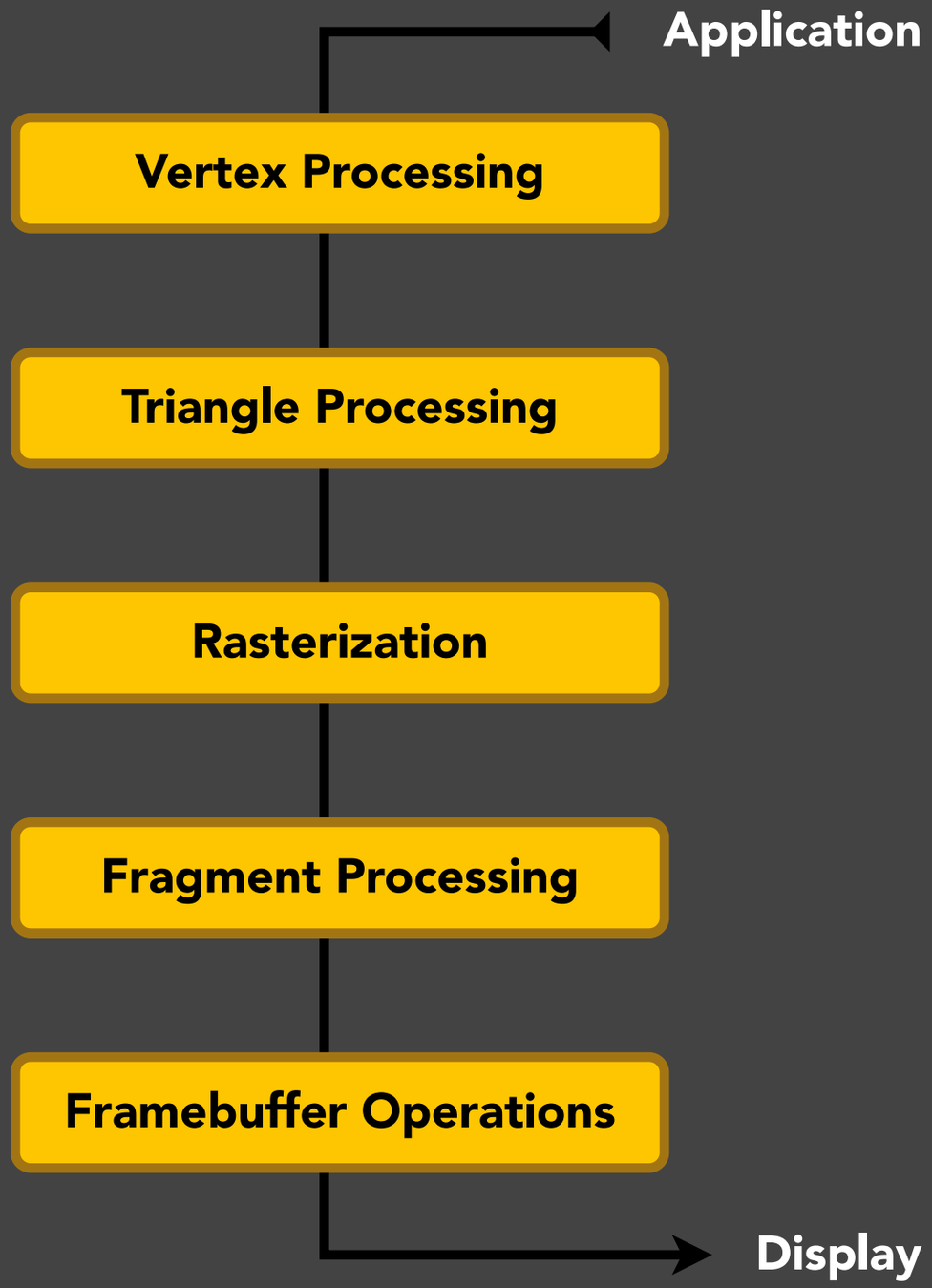


Ambient + Diffuse

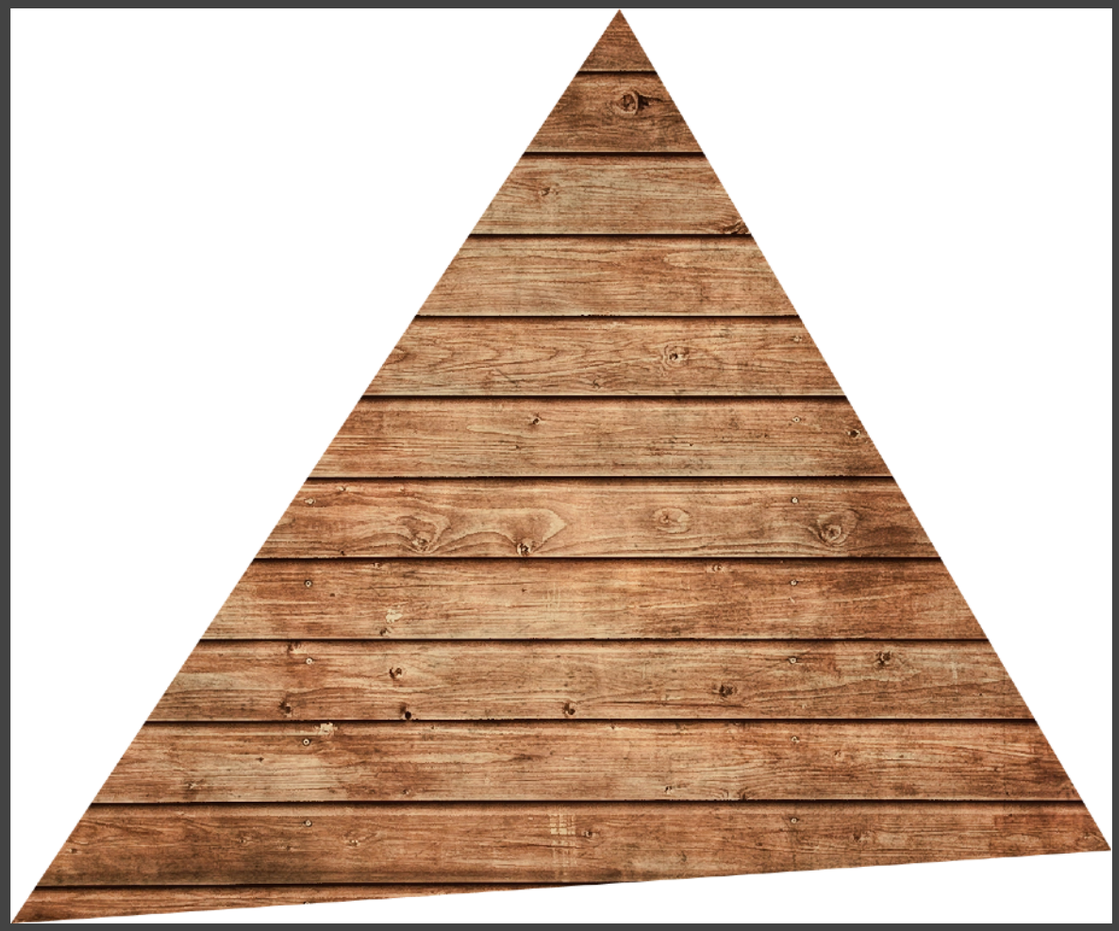


+ Specular = Blinn-Phong Reflectance Model

Graphics Pipeline



Texture mapping & interpolation



Questions?

Today

- Recap of CG Basics
  - Basic GPU hardware pipeline
  - OpenGL
  - OpenGL Shading Language (GLSL)
  - The Rendering Equation
  - Calculus

OpenGL

- Is a set of APIs that call the GPU pipeline from CPU
  - Therefore, language does not matter!
  - Cross platform
  - Alternatives (DirectX, Vulkan, etc.)

- Cons
  - Fragmented: lots of different versions
  - C style, not easy to use
  - Cannot debug (?)

- Understanding
  - 1-to-1 mapping to our software rasterizer in GAMES101

OpenGL

- Important analogy: oil painting
  A. Place objects/models
  B. Set position of an easel
  C. Attach a canvas to the easel
  D. Paint to the canvas
  E. (Attach other canvases to the easel and continue painting)
  F. (Use previous paintings for reference)

OpenGL

- A. Place objects/models
  - Model specification
  - Model transformation

- User specifies an object's vertices, normals, texture coords and send them to GPU as a Vertex buffer object (VBO)
  - Very similar to .obj files

- Use OpenGL functions to obtain matrices
  - e.g., glTranslate, glMultMatrix, etc.
  - No need to write anything on your own

OpenGL

- B. Set up an easel
  - View transformation
  - Create / use a framebuffer

- Set camera (the viewing transformation matrix) by simply calling, e.g., gluPerspective

```
void gluPerspective( GLdouble fovy,
                     GLdouble aspect,
                     GLdouble zNear,
                     GLdouble zFar );
```

OpenGL

- C. Attach a canvas to the easel

- Analogy of oil painting:
  - E. you can also paint multiple pictures using the same easel

- One rendering pass in OpenGL
  - A framebuffer is specified to use
  - Specify one or more textures as output (shading, depth, etc.)
  - Render (fragment shader specifies the content on each texture)

OpenGL

- D. Paint to the canvas
  - i.e., how to perform shading
  - This is when vertex / fragment shaders will be used

- For each vertex in parallel
  - OpenGL calls user-specified vertex shader:
    Transform vertex (ModelView, Projection), other ops

- For each primitive, OpenGL rasterizes
  - Generates a *fragment* for each pixel the fragment covers

OpenGL

- For each fragment in parallel
  - OpenGL calls user-specified fragment shader: Shading and lighting calculations
  - OpenGL handles z-buffer depth test unless overwritten

- This is the “Real” action that we care about the most: user-defined vertex, fragment shaders
  - Other operations are mostly encapsulated
  - Even in the form of GUI-s

OpenGL

- Summary: in each pass
  - Specify objects, camera, MVP, etc.
  - Specify framebuffer and input/output textures
  - Specify vertex / fragment shaders
  - (When you have everything specified on the GPU) Render!

- Now what's left?
  - F. Multiple passes!
    (Use your own previous paintings for reference)

Questions?

Today

- Recap of CG Basics
  - Basic GPU hardware pipeline
  - OpenGL
  - OpenGL Shading Language (GLSL)
  - The Rendering Equation
  - Calculus

Shading Languages

- Vertex / Fragment shading described by small program
- Written in language similar to C but with restrictions
- Long history. Cook's paper on Shade Trees, Renderman for offline rendering
  - In ancient times: assembly on GPUs!
  - Stanford Real-Time Shading Language, work at SGI
  - Still long ago: Cg from NVIDIA
  - HLSL in DirectX (vertex + pixel)
  - GLSL in OpenGL (vertex + fragment)

Shader Setup

- Initializing (shader itself discussed later)
  - Create shader (Vertex and Fragment)
  - Compile shader
  - Attach shader to program
  - Link program
  - Use program
- Shader source is just sequence of strings
- Similar steps to compile a normal program

Shader Initialization Code (FYI)

```
GLuint initshaders (GLenum type, const char *filename) {
  // Using GLSL shaders, OpenGL book, page 679
  GLuint shader = glCreateShader(type) ;
  GLint compiled ;
  string str = textFileRead (filename) ;
  GLchar * cstr = new GLchar[str.size()+1] ;
  const GLchar * cstr2 = cstr ; // Weirdness to get a const char
  strcpy(cstr,str.c_str()) ;
  glShaderSource (shader, 1, &cstr2, NULL) ;
  glCompileShader (shader) ;
  glGetShaderiv (shader, GL_COMPILE_STATUS, &compiled) ;
  if (!compiled) {
    shadererrors (shader) ;
    throw 3 ;
  }
  return shader ;
}
```

Linking Shader Program (FYI)

```
GLuint initprogram (GLuint vertexshader, GLuint fragmentshader)
{
  GLuint program = glCreateProgram() ;
  GLint linked ;
  glAttachShader(program, vertexshader) ;
  glAttachShader(program, fragmentshader) ;
  glLinkProgram(program) ;
  glGetProgramiv(program, GL_LINK_STATUS, &linked) ;
  if (linked) glUseProgram(program) ;
  else {
    programerrors(program) ;
    throw 4 ;
  }
  return program ;
}
```

Phong Shader in Assignment 0

- Vertex Shader

```
attribute vec3 aVertexPosition;
attribute vec3 aNormalPosition;
attribute vec2 aTextureCoord;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;

varying highp vec2 vTextureCoord;
varying highp vec3 vFragPos;
varying highp vec3 vNormal;


void main(void) {

  vFragPos = aVertexPosition;
  vNormal = aNormalPosition;

  gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(aVertexPosition, 1.0);

  vTextureCoord = aTextureCoord;

}
```

Phong Shader in Assignment 0

- Fragment Shader

```
4   uniform sampler2D uSampler;
5   uniform vec3 uKd;
6   uniform vec3 uKs;
7   uniform vec3 uLightPos;
8   uniform vec3 uCameraPos;
9   uniform float uLightIntensity;
10  uniform int uTextureSample;
11
12  varying highp vec2 vTextureCoord;
13  varying highp vec3 vFragPos;
14  varying highp vec3 vNormal;
15
16  void main(void) {
17    vec3 color;
18    if (uTextureSample == 1) {
19      color = pow(texture2D(uSampler, vTextureCoord).rgb, vec3(2.2));
20    } else {
21      color = uKd;
22    }
```

Phong Shader in Assignment 0

- Fragment Shader (cont.)

```
24    vec3 ambient = 0.05 * color;
25
26    vec3 lightDir = normalize(uLightPos - vFragPos);
27    vec3 normal = normalize(vNormal);
28    float diff = max(dot(lightDir, normal), 0.0);
29    float light_atten_coff = uLightIntensity / length(uLightPos - vFragPos);
30    vec3 diffuse =  diff * light_atten_coff * color;
31
32    vec3 viewDir = normalize(uCameraPos - vFragPos);
33    float spec = 0.0;
34    vec3 reflectDir = reflect(-lightDir, normal);
35    spec = pow (max(dot(viewDir, reflectDir), 0.0), 35.0);
36    vec3 specular = uKs * light_atten_coff * spec;
37
38    gl_FragColor = vec4(pow((ambient + diffuse + specular), vec3(1.0/2.2)), 1.0);
39
40  }
```

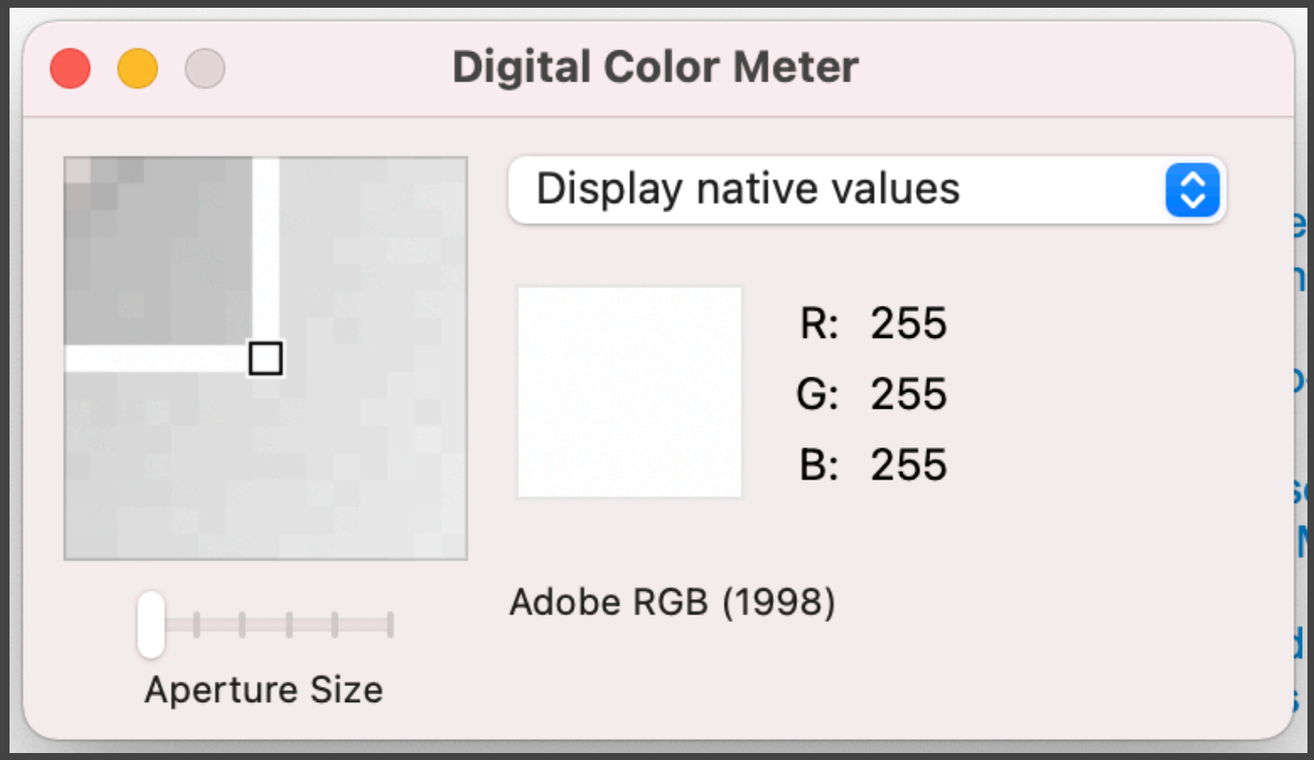
Debugging Shaders

- Years ago: NVIDIA Nsight with Visual Studio
  - Needed multiple GPUs for debugging GLSL
  - Had to run in software simulation mode in HLSL

- Now
  - Nsight Graphics (cross platform, NVIDIA GPUs only)
  - RenderDoc (cross platform, no limitations on GPUs)
  - Unfortunately I don’t know if they can be used for WebGL

Debugging Shaders

- Personal advice
  - Print it out!
  - But how?
  - Show values as colors!



Questions?

Today

- Recap of CG Basics
  - Basic GPU hardware pipeline
  - OpenGL
  - OpenGL Shading Language (GLSL)
  - The Rendering Equation
  - Calculus

The Rendering Equation

- Most important equation in rendering
  - Describing light transport

$$L_o(\mathrm{p}, \omega_o) = L_e(\mathrm{p}, \omega_o) + \int_{H^2} f_r(\mathrm{p}, \omega_i \rightarrow \omega_o)\, L_i(\mathrm{p}, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

outgoing radiance

emission

BRDF

incident radiance

The Rendering Equation

- In real-time rendering (RTR)
  - Visibility is often explicitly considered
  - BRDF is often considered together with the cosine term

$$L_o(\mathrm{p}, \omega_o) = \int_{\Omega+} L_i(\mathrm{p}, \omega_i) f_r(\mathrm{p}, \omega_i, \omega_o) \cos\theta_i V(\mathrm{p}, \omega_i) \mathrm{d}\omega_i$$

outgoing lighting

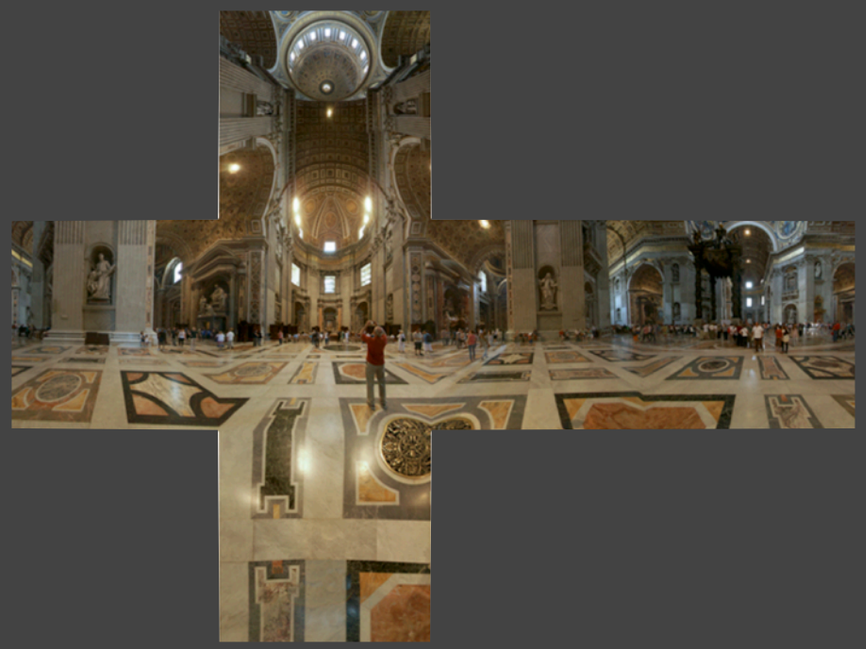
incident lighting (from source)

(cosine-weighted) BRDF

visibility

Environment Lighting

- Representing incident lighting from all directions
  - Usually represented as a cube map or a sphere map (texture)
  - We'll introduce a new representation in this course



Direct illumination

One-bounce global illumination

Two-bounce global illumination

Next Lecture

- Shadow mapping
  - Basic shadow mapping technique
  - Advanced soft shadow techniques
    - Variance shadow maps (VSM)
    - Moment shadow maps (MSM)

Zelda: Breath of the Wild

Thank you!