

GPU Graphics Pipeline (Part II)

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

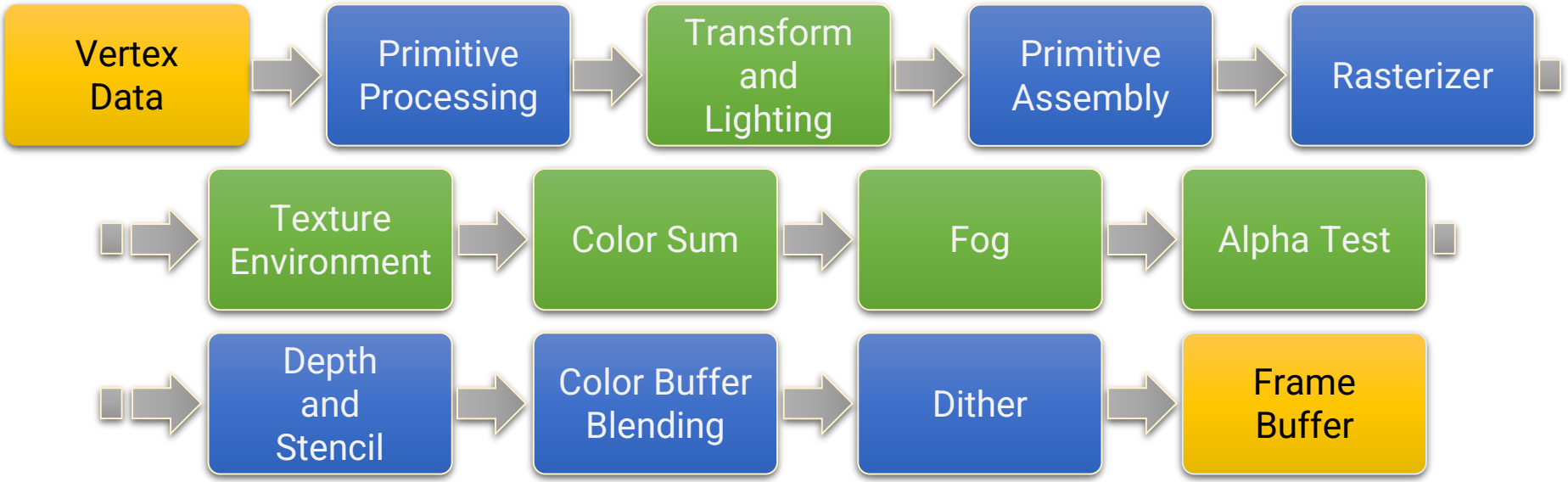
(Part II)

Outline

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

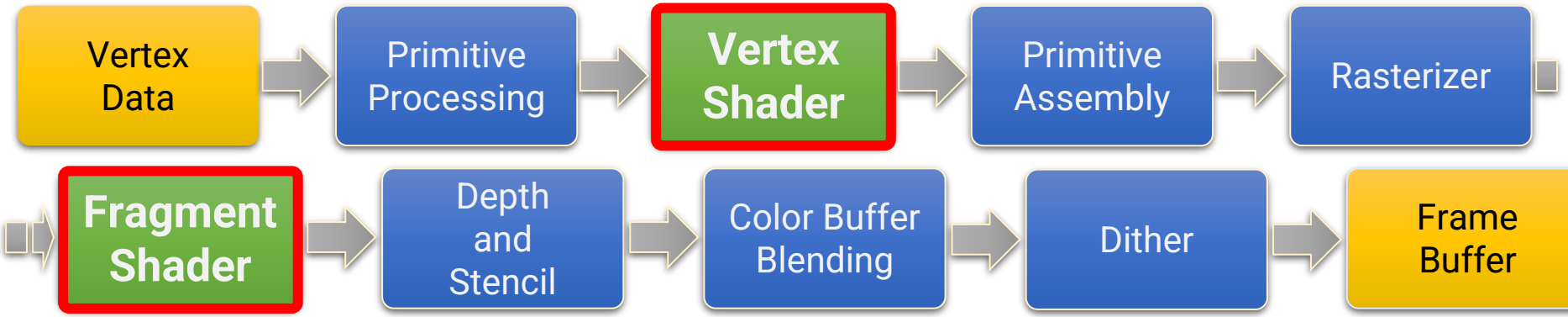
Recap: OpenGL (1.x) Fixed Function Pipeline

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- The stages shown in green have been replaced by **shaders**



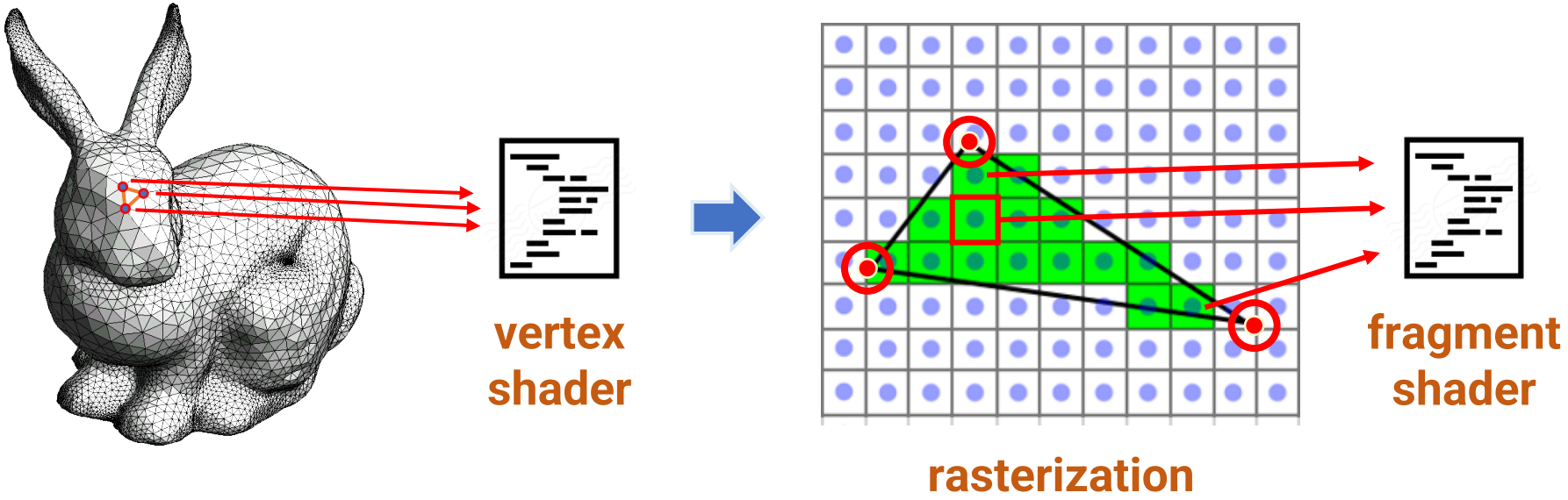
OpenGL (2.0) Graphics Pipeline

- 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)

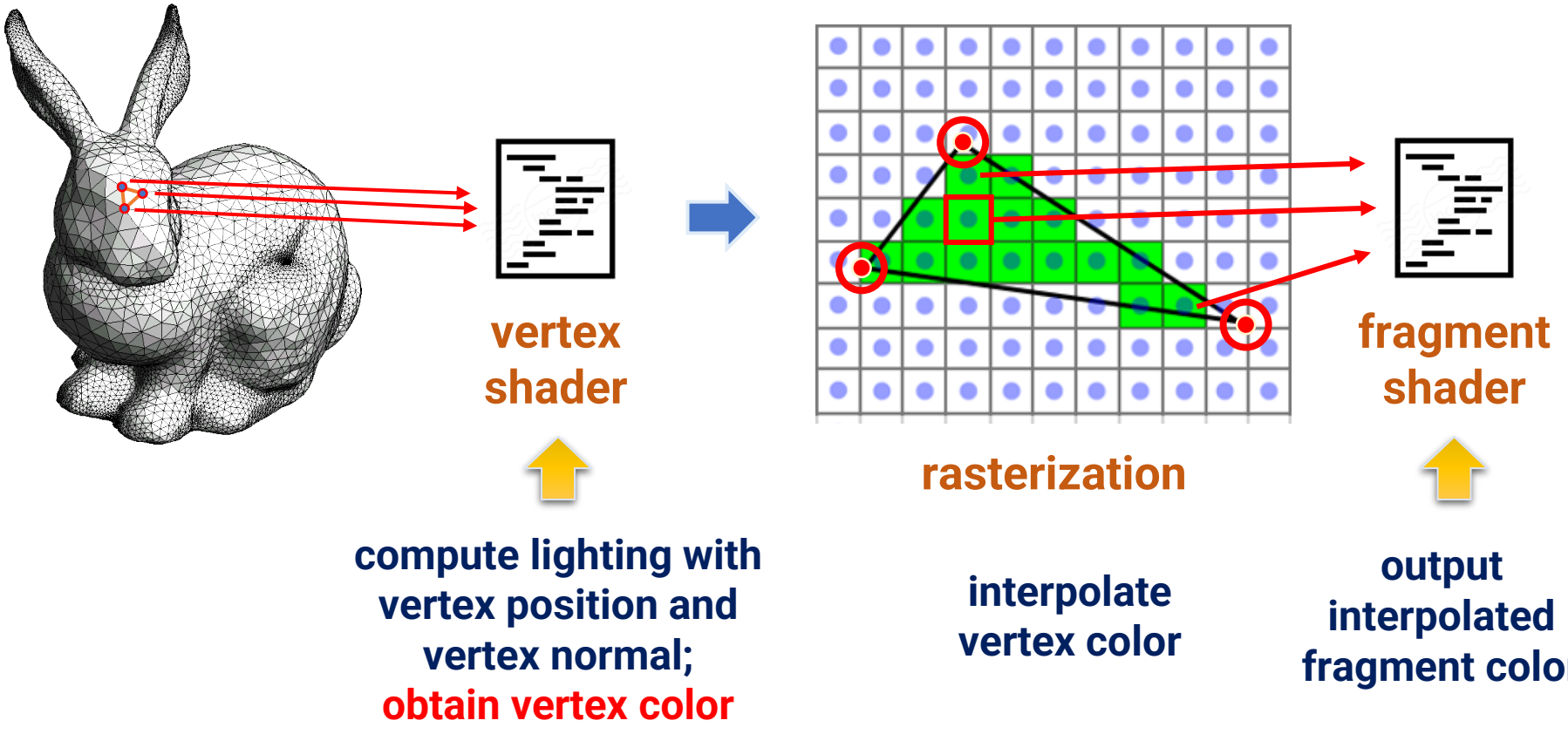
- Provide the programmers more **flexibility** regarding
  - **How the vertices are transformed**
    - We can also choose not to transform the vertices at all
  - **How the lighting is computed**
    - We can also choose to compute lighting in the fragment shader (per-fragment lighting)
- However, ***with 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 for different lighting and fog models, as well as an arbitrary combination of lighting and texture
- Allow for techniques such as per-pixel lighting, bump, normal mapping, etc.

Per-Vertex Shading

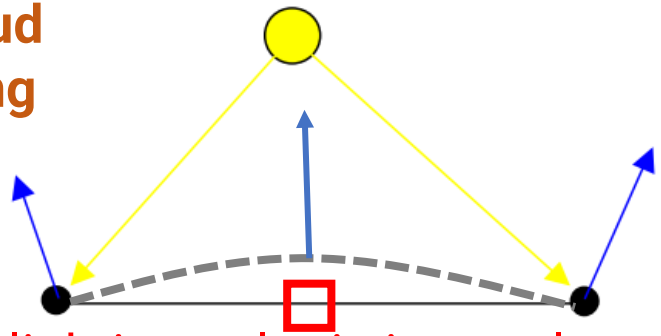
- OpenGL 2.0 can still implement per-vertex shading (Gouraud shading)

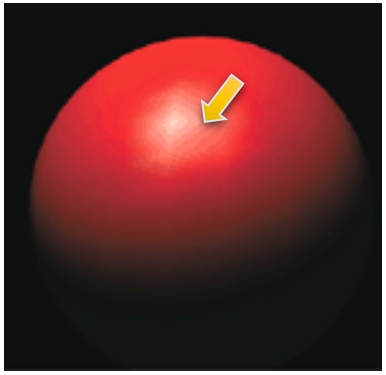


Per-Fragment Shading

- Problem with Gouraud shading

Gouraud shading

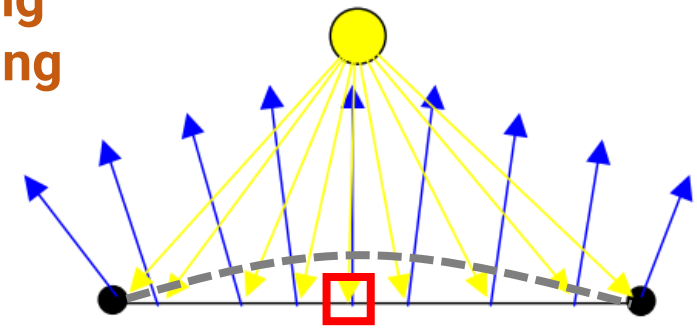


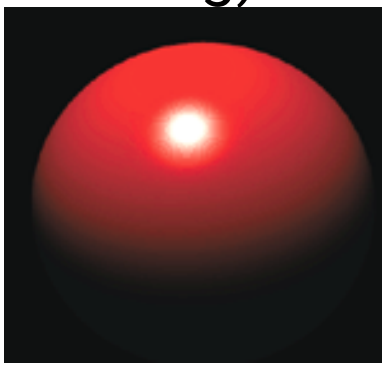


lighting color is interpolated

- Phong shading (instead of Gouraud shading)

Phong shading

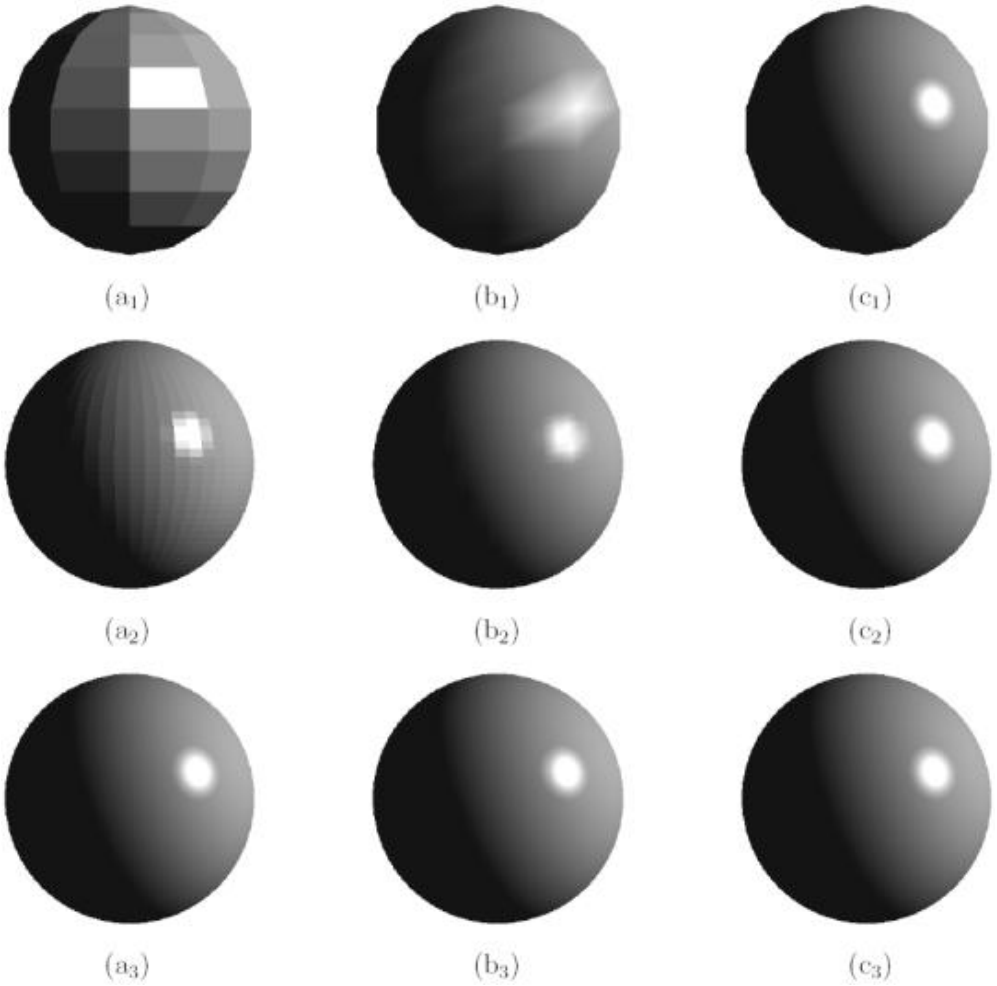




surface normal is interpolated (how? Rasterization!)

Per-Fragment Shading (cont.)

flat shading Gouraud shading Phong shading

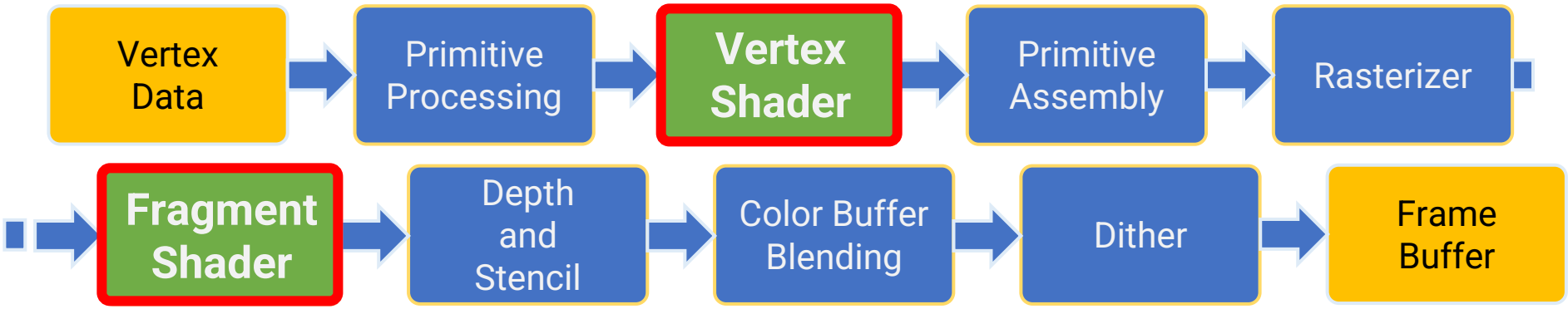


Outline

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

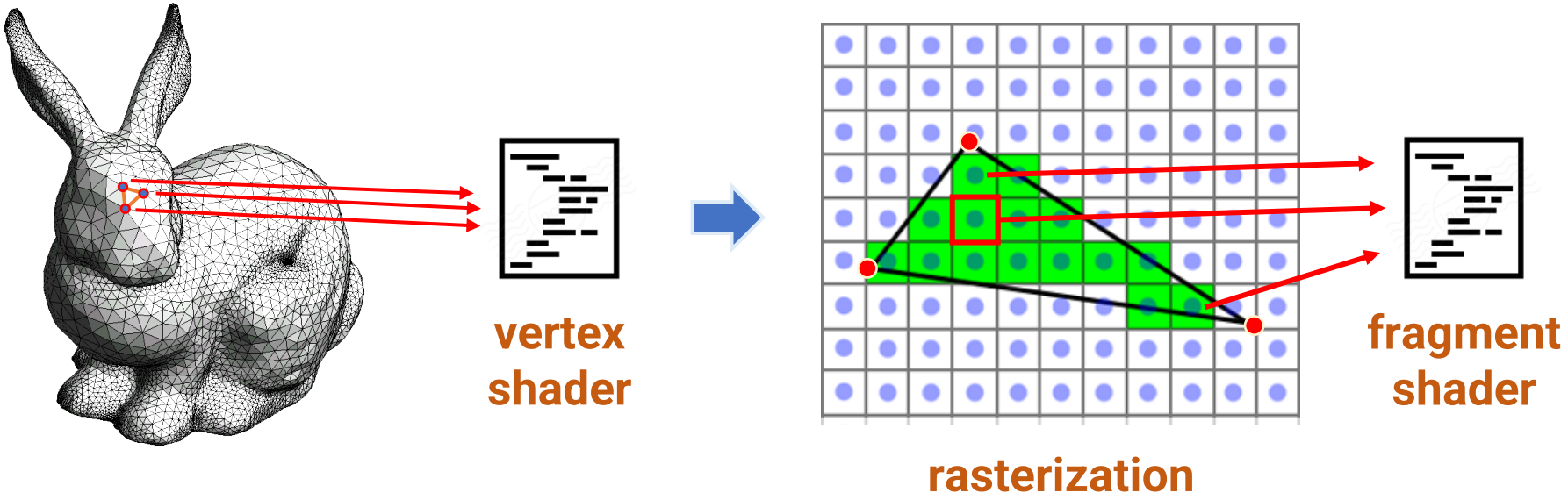
Recap: OpenGL 2.0 Graphics Pipeline

- Programmers need to provide the two shader programs
- Other stages maintain the same (set OpenGL states)



Recap: Vertex Shader and Fragment Shader

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



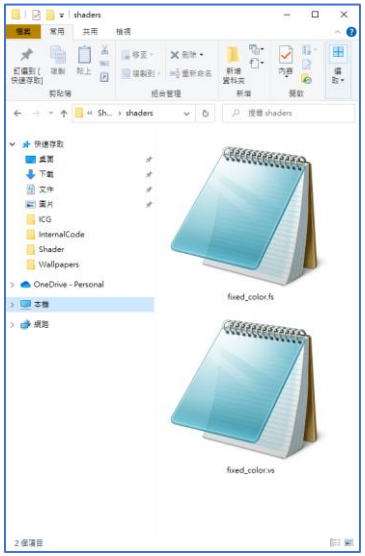
Sample Project

- You can find the sample code in the project, **Shader**

Prepare Shaders

- Shaders are just text files written in a **special shader language**, such as
  - OpenGL Shading Language (GLSL)
  - High-Level Shading Language (HLSL) for DirectX
  - Nvidia Cg (used by Unity)

the file extension does not matter!



fixed_color.vs - 記事本

```
#version 330 core

layout (location = 0) in vec3 Position;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;
// uniform mat4 MVP;

void main()
{
    gl_Position = projMatrix * viewMatrix * modelMatrix * vec4(Position, 1.0);
    // gl_Position = MVP * vec4(Position, 1.0);
}
```

vertex shader

fixed_color.fs - 記事本

```
#version 330 core

uniform vec3 fillColor;

out vec4 FragColor;

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

fragment shader

Load and Create an OpenGL Shader

```
// Shader.
GLuint shaderProgId;
GLint locM, locV, locP, locMVP;
GLint locFillColor;

void CreateShader(const std::string vsFilePath, const std::string fsFilePath)
{
    // Create OpenGL shader program.
    shaderProgId = glCreateProgram();
    if (shaderProgId == 0) {
        std::cerr << "[ERROR] Failed to create shader program" << std::endl;
        exit(1);
    }

    // Load the vertex shader from a source file and attach it to the shader program.
    std::string vs, fs;
    if (!LoadShaderTextFromFile(vsFilePath, vs)) {
        std::cerr << "[ERROR] Failed to load vertex shader source: " << vsFilePath << std::endl;
        exit(1);
    }
    GLuint vsId = AddShader(shaderProgId, vs, GL_VERTEX_SHADER);

    // Load the fragment shader from a source file and attach it to the shader program.
    if (!LoadShaderTextFromFile(fsFilePath, fs)) {
        std::cerr << "[ERROR] Failed to load vertex shader source: " << fsFilePath << std::endl;
        exit(1);
    };
    GLuint fsId = AddShader(shaderProgId, fs, GL_FRAGMENT_SHADER);
```

Create OpenGL shader program (ID)

in our case, a shader program consists of a vertex shader and a fragment shader

Load vertex shader source

Create, compile the vertex shader and attach it to the shader program

Load fragment shader source

Create, compile the fragment shader and attach it to the shader program

Load and Create an OpenGL Shader (cont.)

```
// Link and compile shader programs.
GLint success = 0;
GLchar errorLog[MAX_BUFFER_SIZE] = { 0 };
glLinkProgram(shaderProgId);
glGetProgramiv(shaderProgId, GL_LINK_STATUS, &success);
if (success == 0) {
    glGetProgramInfoLog(shaderProgId, sizeof(errorLog), NULL, errorLog);
    std::cerr << "[ERROR] Failed to link shader program: " <<  errorLog << std::endl;
    exit(1);
}

// Now the program already has all stage information, we can delete the shaders now.
glDeleteShader(vsId);
glDeleteShader(fsId);

// Validate program.
glValidateProgram(shaderProgId);
glGetProgramiv(shaderProgId, GL_VALIDATE_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgId, sizeof(errorLog), NULL, errorLog);
    std::cerr << "[ERROR] Invalid shader program: " << errorLog << std::endl;
    exit(1);
}

// Get the location of uniform variables.
// Discuss later
```

Link all **attached** shaders to the program

Delete (free memory) vertex/fragment shader object

Validate your shader program

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;
void main() {
    gl_Position = projMatrix * viewMatrix *
                  modelMatrix * vec4(Position, 1.0);
}
```

Vertex attribute
- **glEnableVertexAttribArray(0)**

uniform variables communicated with the CPU
- Get location by **glGetUniformLocation**
- Set value by **glUniformXXX**

the main program **executed per vertex**

a built-in variable for the Clip Space coordinate

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 MVP;
```

Input: vertex attribute
- **glEnableVertexAttribArray(0)**

uniform variables communicated with the CPU
- Get location by **glGetUniformLocation**
- Set value by **glUniformXXX**

the main program **executed per vertex**

```
void main() {
    gl_Position = MVP * vec4(Position, 1.0);
}
```

a built-in variable for the Clip Space coordinate

Fragment Shader

```
#version 330 core
uniform vec3 fillColor;
```

uniform variables communicated with the CPU

- Get location by **glGetUniformLocation**
- Set value by **glUniformXXX**

```
out vec4 FragColor;
```

Output: fragment data

the main program **executed per fragment**

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

Connect the Program with Shaders

- Get the location of uniform variables in the shader

```
// Get the location of uniform variables.
locM = glGetUniformLocation(shaderProgId, "modelMatrix");
locV = glGetUniformLocation(shaderProgId, "viewMatrix");
locP = glGetUniformLocation(shaderProgId, "projMatrix");
locMVP = glGetUniformLocation(shaderProgId, "MVP");
locFillColor = glGetUniformLocation(shaderProgId, "fillColor");
```

- Assign values to the uniform variables in shaders

```
// Bind shader and set parameters.
glUseProgram(shaderProgId);
glUniformMatrix4fv(locM, 1, GL_FALSE, glm::value_ptr(M));
glUniformMatrix4fv(locV, 1, GL_FALSE, glm::value_ptr(camera->GetViewMatrix()));
glUniformMatrix4fv(locP, 1, GL_FALSE, glm::value_ptr(camera->GetProjMatrix()));
// glUniformMatrix4fv(locMVP, 1, GL_FALSE, glm::value_ptr(MVP));
glUniform3fv(locFillColor, 1, glm::value_ptr(fillColor));

// Render the mesh.
if (mesh ≠ nullptr)
    mesh->Draw();

// Unbind shader.
glUseProgram(0);
```

bind (there might be several shaders in your program)

unbind

Connect the Program with Shaders (cont.)

- Bind and unbind to a shader program

the shader program you created

```
void glUseProgram( GLuint program );
```

```
glUseProgram(shaderProgId);
// set parameters
// render something
glUseProgram(0);
```

Connect the Program with Shaders (cont.)

- Get the location of uniform variables in the shader

```
GLint glGetUniformLocation(
        GLuint program ,   the shader program you created
        const GLchar *name
);                         the uniform variable in the shader
```

```
// Get the location of uniform variables.
locM = glGetUniformLocation(shaderProgId, "modelMatrix");
locV = glGetUniformLocation(shaderProgId, "viewMatrix");
locP = glGetUniformLocation(shaderProgId, "projMatrix");
locMVP = glGetUniformLocation(shaderProgId, "MVP");
locFillColor = glGetUniformLocation(shaderProgId, "fillColor");
```

Connect the Program with Shaders (cont.)

- Assign values to the uniform variables
- Lots of variants depending on the variable type, please refer to https://registry.khronos.org/OpenGL-Refpages/gl4/html/glUniform.xhtml

```
void glUniform3fv(
        GLint location,
        GLsizei count,
        const GLfloat *value
);
```

the variable location get by glGetUniformLocation

the number of the vectors (1 if not an array)

the values of the parameters

```
glm::vec3 fillColor = glm::vec3(1.0f, 1.0f, 0.0f);
glUniform3fv(locFillColor, 1, glm::value_ptr(fillColor));
```

Connect the Program with Shaders (cont.)

- Assign values to the uniform variables

```
void glUniformMatrix4fv(
        GLint location ,
        GLsizei count ,
        GLboolean transpose ,
        const GLfloat *value
);
```

should the matrix be accessed in a transpose way **(since both OpenGL and GLM use column-major, we set it to FALSE)**

```
glUniformMatrix4fv(locM, 1, GL_FALSE, glm::value_ptr(M));
glUniformMatrix4fv(locV, 1, GL_FALSE, glm::value_ptr(camera->GetViewMatrix()));
glUniformMatrix4fv(locP, 1, GL_FALSE, glm::value_ptr(camera->GetProjMatrix()));
```

Resize Window

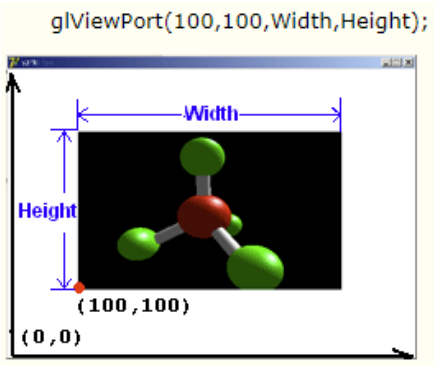
```
glutReshapeFunc(ReshapeCB);

void ReshapeCB(int w, int h)
{
    // Update viewport.
    screenWidth = w;
    screenHeight = h;
    glViewport(0, 0, screenWidth, screenHeight);
    // Adjust camera and projection.
    float aspectRatio = (float)screenWidth / (float)screenHeight;
    camera->UpdateProjection(fovy, aspectRatio, zNear, zFar);
    MVP = camera->GetProjMatrix() * camera->GetViewMatrix() * M;
}
```

remember to reset the range of rendering in an OpenGL window

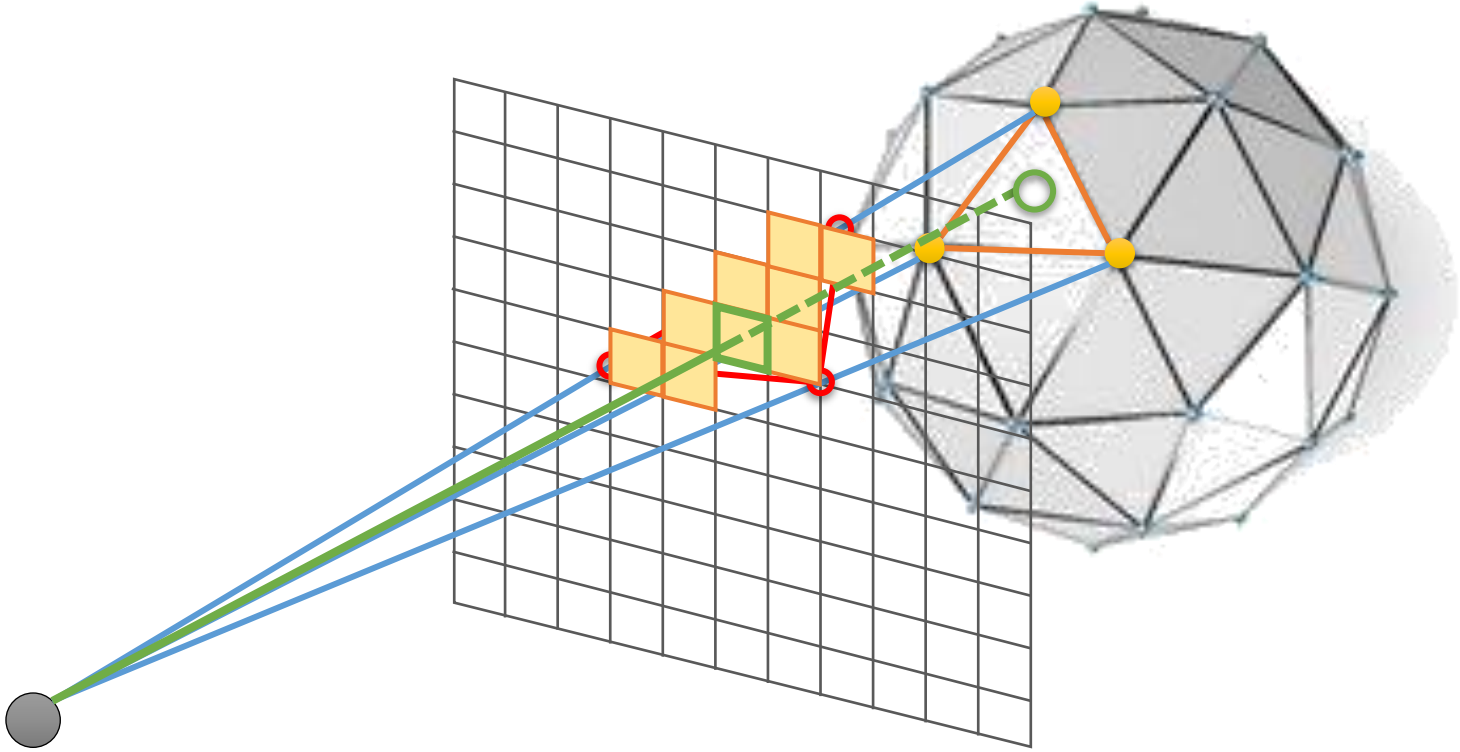






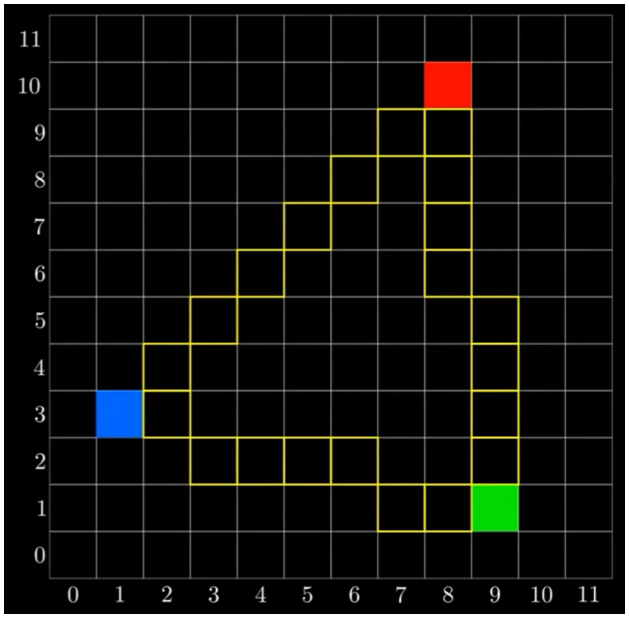
Vertex Attribute Interpolation

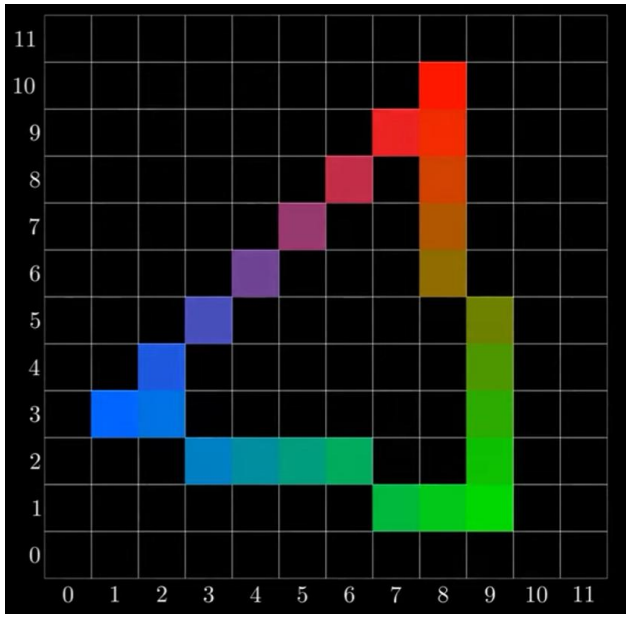
- Generate **fragments** for each triangle
- Interpolate vertex attributes at each fragment



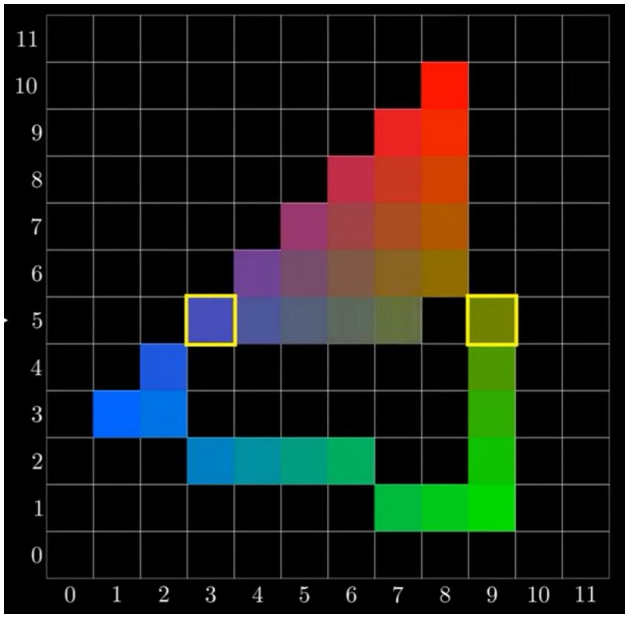
Vertex Attribute Interpolation (cont.)

- **Interpolate vertex color**





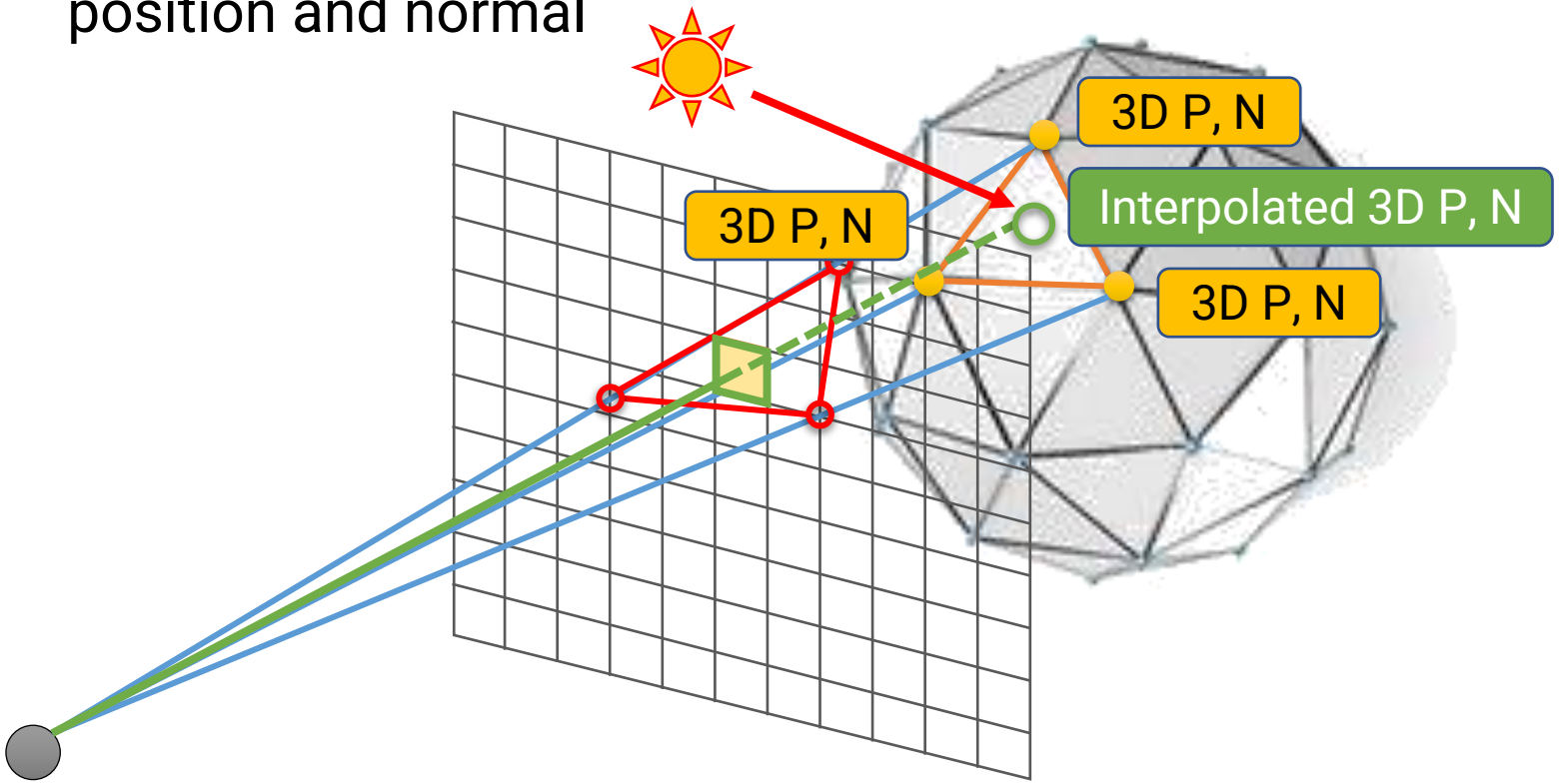
Attributes interpolation of edge pixels using vertices



Attributes interpolation of inner pixels using edge points

Vertex Attribute Interpolation (cont.)

- **Interpolate geometry attributes**
  - Compute lighting at each fragment (in the fragment shader) requires per-fragment geometry attributes such as 3D position and normal



Vertex Attribute Interpolation (cont.)

- Example: interpolate **world-space vertex position** and **world-space vertex normal**

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 Position;
layout (location = 1) in vec3 Normal;

// Transformation matrix.
uniform mat4 worldMatrix;
uniform mat4 normalMatrix;
uniform mat4 MVP;

// Data pass to fragment shader.
out vec3 iPosWorld;
out vec3 iNormalWorld;

void main()
{
    gl_Position = MVP * vec4(Position, 1.0);

    // Pass vertex attributes.
    vec4 positionTmp = worldMatrix * vec4(Position, 1.0);
    iPosWorld = positionTmp.xyz / positionTmp.w;

    iNormalWorld = (normalMatrix * vec4(Normal, 0.0)).xyz;
}
```

Tell OpenGL you want to interpolate these attributes

world matrix for transforming normal (intro. in next lecture)

Fragment Shader

```
#version 330 core

// Data from vertex shader.
in vec3 iPosWorld;
in vec3 iNormalWorld;

out vec4 FragColor;

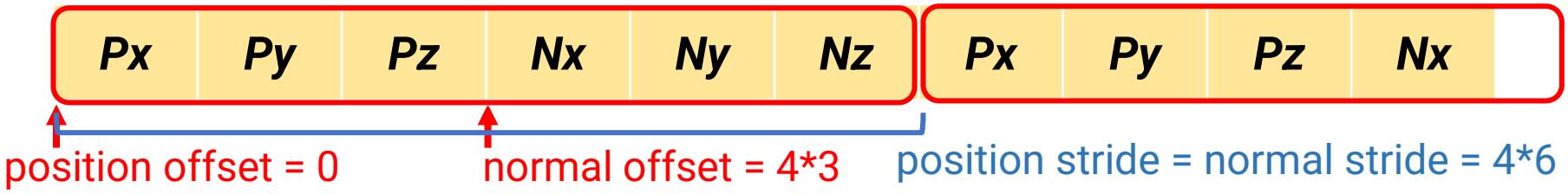

void main()
{
    vec3 N = normalize(iNormalWorld);
    FragColor = vec4(N, 1.0);
}
```

interpolated fragment data

Ensure the interpolated normal has a unit length

Vertex Attribute Interpolation (cont.)

- Note: when you do this, you need to pass in the vertex normal attribute




```
glEnableVertexAttribArray(0);  the first attribute: in this case, position
glEnableVertexAttribArray(1);  the second attribute: in this case, normal

glBindBuffer(GL_ARRAY_BUFFER, vboId);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPN), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPN), (const GLvoid*)12);

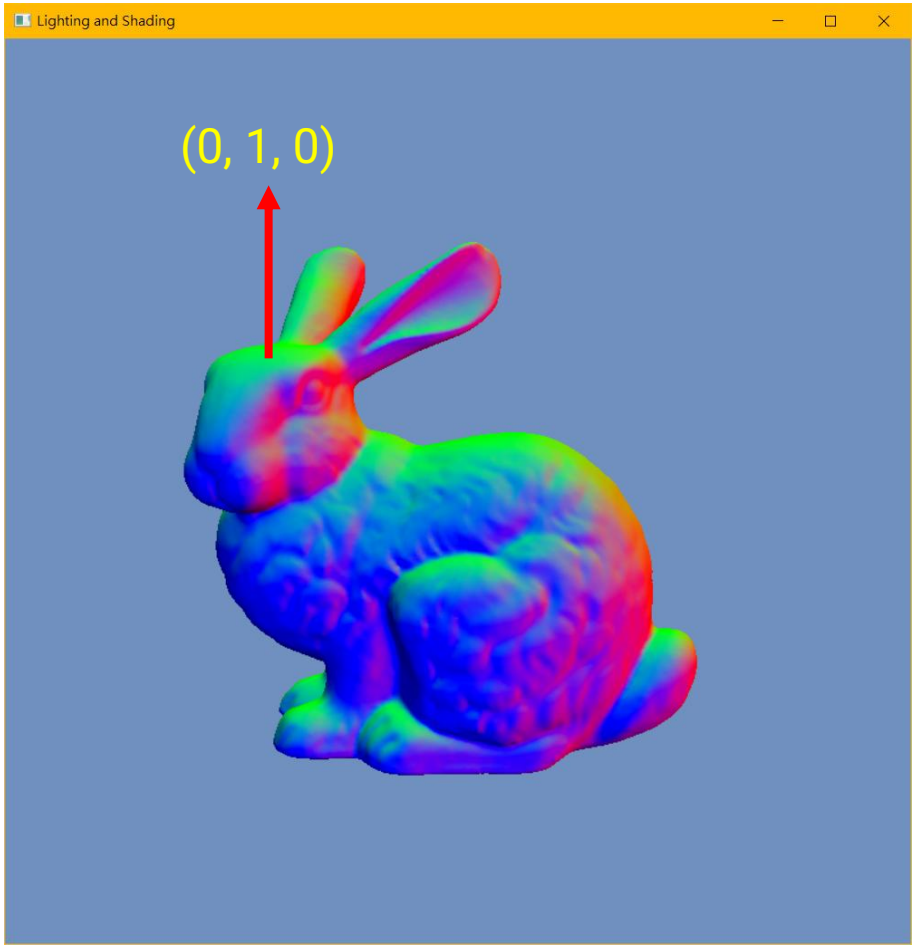
 // … draw primitives

glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
```

Vertex Attribute Interpolation (cont.)



visualize world-space position as color



visualize world-space normal as color

Vertex Attribute Interpolation (cont.)

- Remember the homogeneous coordinate for a 3D point *(x, y, z)* is *(x, y ,z, **1**)*
  - Why? To enable the combination of a **translation** matrix with other transformation matrices

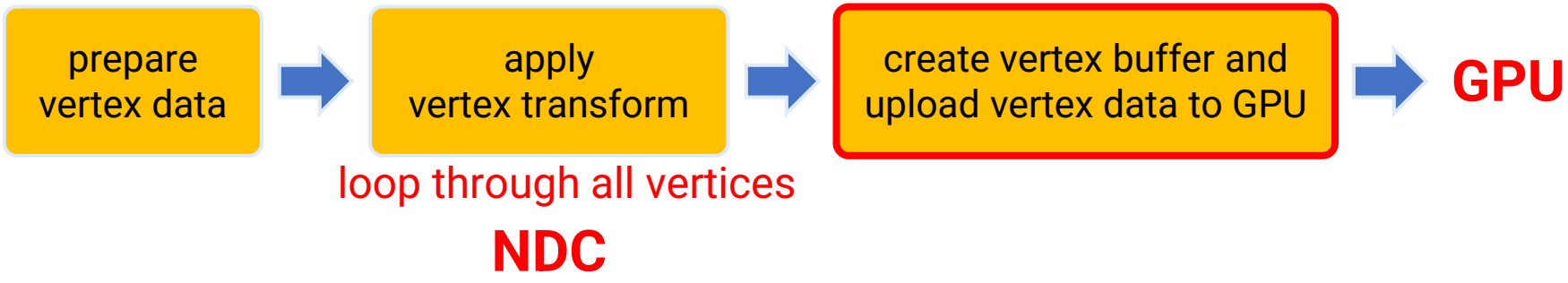
$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \\ z' &= z + t_z \end{aligned}$$

- When transforming a vector, we represent a 3D direction *(dx, dy, dz)* by *(dx, dy, dz, **0**)* because we do not want a translation for “direction”
  - Otherwise, the direction (0.578, 0.578, 0.578) will become (3.578, 4.578, 5.578) after a translation of (3, 4, 5)

CPU v.s. GPU

- **CPU (what we do in HW1)**




```
void ApplyTransformCPU(std::vector<glm::vec3>& vertexPositions, const glm::mat4x4& mvpMatrix)
{
    for (unsigned int i = 0 ; i < vertexPositions.size(); ++i) {
        glm::vec4 p = mvpMatrix * glm::vec4(vertexPositions[i], 1.0f);
        if (p.w != 0.0f) {
            float inv = 1.0f / p.w;
            vertexPositions[i].x = p.x * inv;
            vertexPositions[i].y = p.y * inv;
            vertexPositions[i].z = p.z * inv;
        }
    }
}
```

CPU v.s. GPU (cont.)

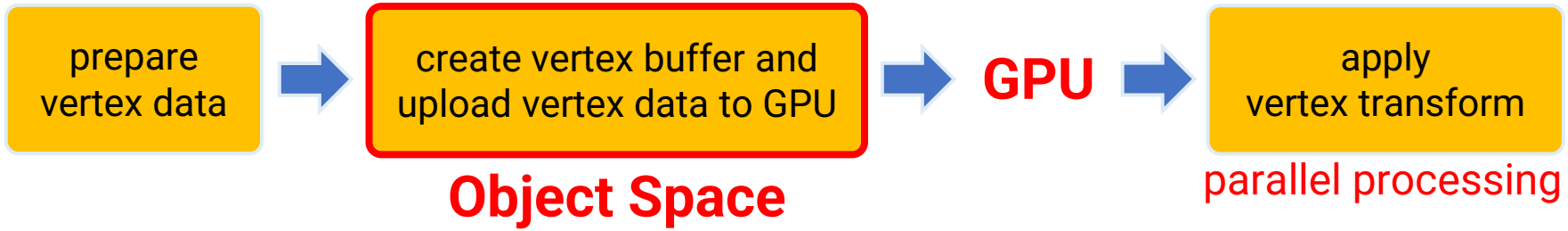
- **GPU: Transform in OpenGL 1.x** **(deprecated, DO NOT USE!)**

```
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);
}
```

CPU v.s. GPU (cont.)

- **GPU (what we do with shader)**




```
locMVP = glGetUniformLocation(shaderProgId, "MVP");
glUniformMatrix4fv(locMVP, 1, GL_FALSE, glm::value_ptr(MVP));
```

CPU

1

2

Vertex Shader

GPU

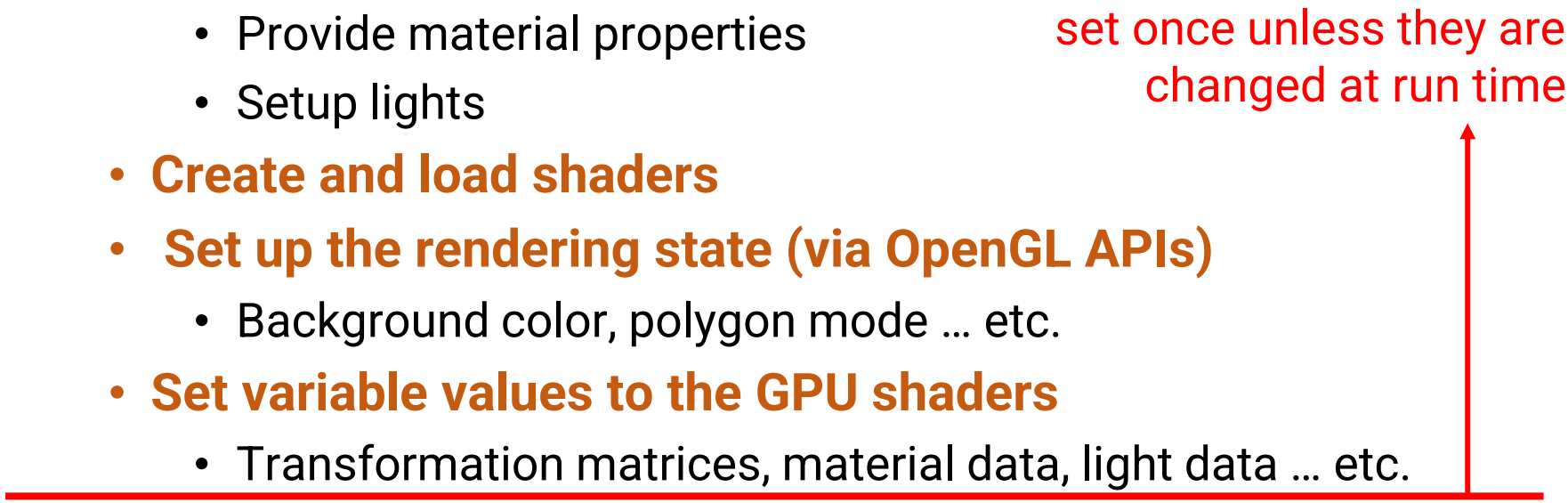
```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 MVP;
void main() {
    gl_Position = MVP * vec4(Position, 1.0);
}
```

a built-in variable for the Clip Space coordinate

No loop because the vertex shader is executed for each vertex **in parallel** by nature

CPU v.s. GPU (cont.)

- In the **CPU** application, we
  - **Load the scene data (from files)**
    - Create vertex and index buffers
    - Provide material properties
    - Setup lights
  - **Create and load shaders**
  - **Set up the rendering state (via OpenGL APIs)**
    - Background color, polygon mode … etc.
  - **Set variable values to the GPU shaders**
    - Transformation matrices, material data, light data … etc.
  - **Call "Draw" functions to render objects (via OpenGL APIs)**
    - Vertex buffer format, primitive type, # of indices



CPU v.s. GPU (cont.)

- On the **GPU**, we
  - Execute the **Vertex Shader** for each vertex that belongs to a triangle
    - Vertex transformation
    - Vertex lighting (optional)
    - Prepare for interpolated vertex attributes (pass to fragment shader)

OpenGL performs **rasterization** by **hardware**

  - Execute the **Fragment Shader** for each fragment generated by the rasterization for each triangle
    - Fragment shading (lighting, texturing … etc.)

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?

Supp: Load and Create an OpenGL Shader

```
GLuint AddShader(GLuint& progId, const std::string& sourceText, GLenum shaderType)
{
    GLuint shaderObj = glCreateShader(shaderType);
    if (shaderObj == 0) {
        std::cerr << "[ERROR] Failed to create shader with type " << shaderType << std::endl;
        exit(0);
    }

    const GLchar* p[1];
    p[0] = sourceText.c_str();
    GLint lengths[1];
    lengths[0] = (GLint)(sourceText.length());
    glShaderSource(shaderObj, 1, p, lengths);
    glCompileShader(shaderObj);

    GLint success;
    glGetShaderiv(shaderObj, GL_COMPILE_STATUS, &success);
    if (!success) {
        GLchar infoLog[MAX_BUFFER_SIZE];
        glGetShaderInfoLog(shaderObj, MAX_BUFFER_SIZE, NULL, infoLog);
        std::cerr << "[ERROR] Failed to compile shader with type: " << shaderType << ". Info: " << infoLog << std::endl;
        exit(1);
    }

    glAttachShader(progId, shaderObj);

    return shaderObj;
}
```

Types:
GL_VERTEX_SHADER /
GL_FRAGMENT_SHADER
GL_GEOMETRY_SHADER
GL_TESS_CONTROL_SHADER,
GL_TESS_EVALUATION_SHADER,
GL_COMPUTE_SHADER

