# 实验6：Phong光照模型

# Phong Illumination Model

华东师范大学计算机科学与技术学院

李晨  副研究员

cli@cs.ecnu.edu.cn

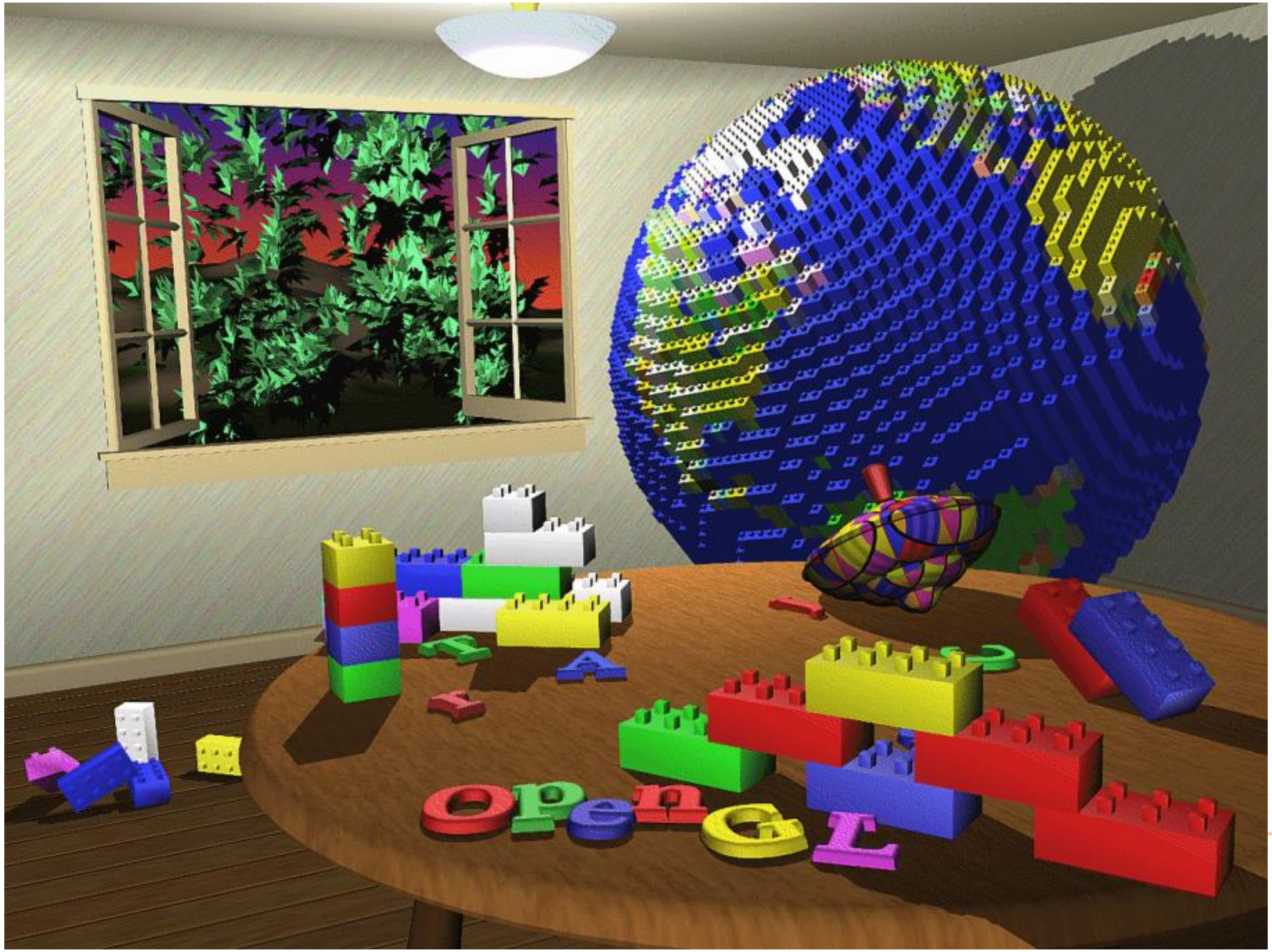
华东师范大学计算机科学与技术学院
School of Computer Science and Technology

# Today

- OpenGL and GLSL
- Phong Illumination Model

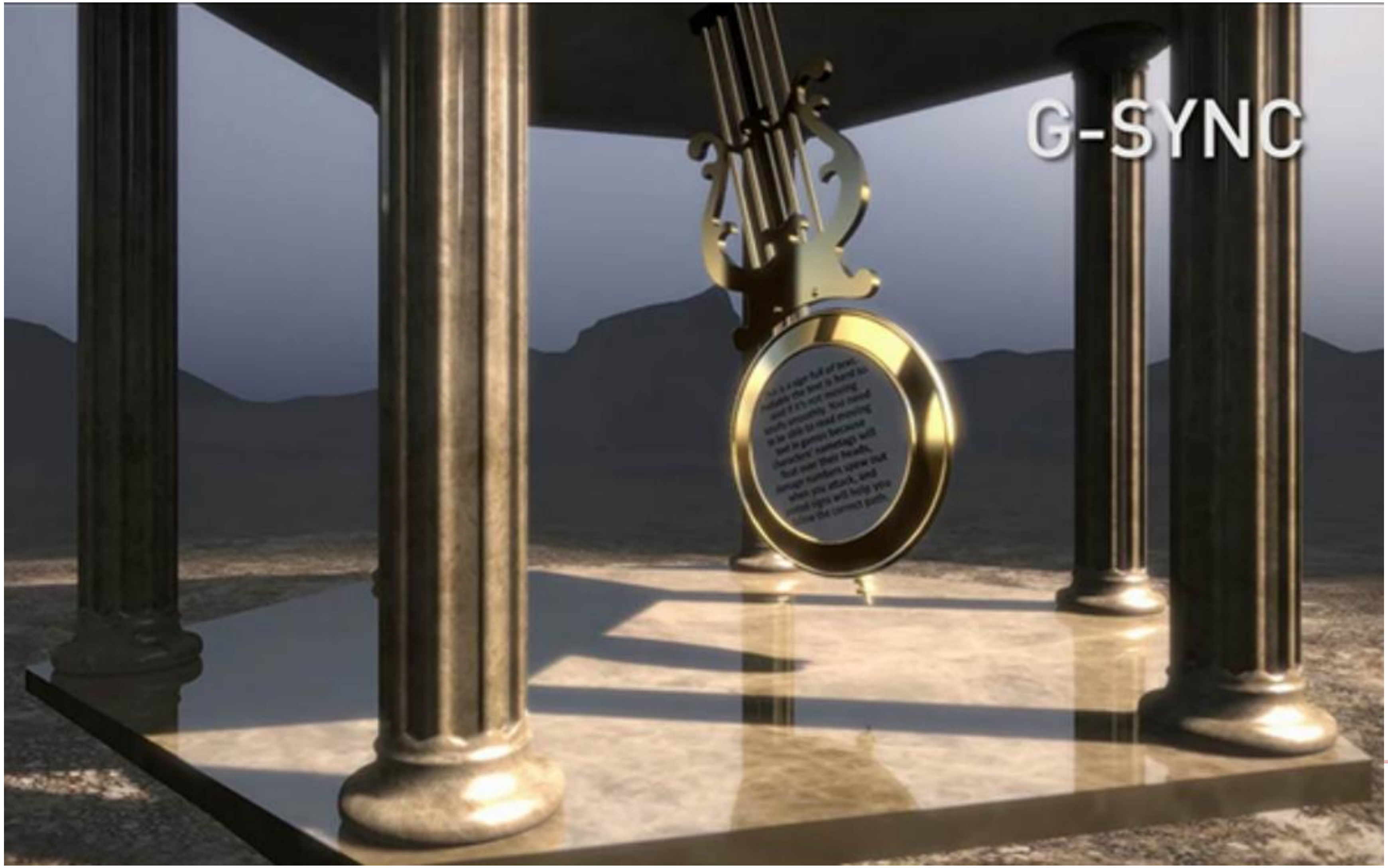
Bui Tuong Phong
**裴祥风**
**1942-1975**

# OpenGL 30 Years ago




华东师范大学计算机科学与技术学院
School of Computer Science and Technology

# Modern OpenGL



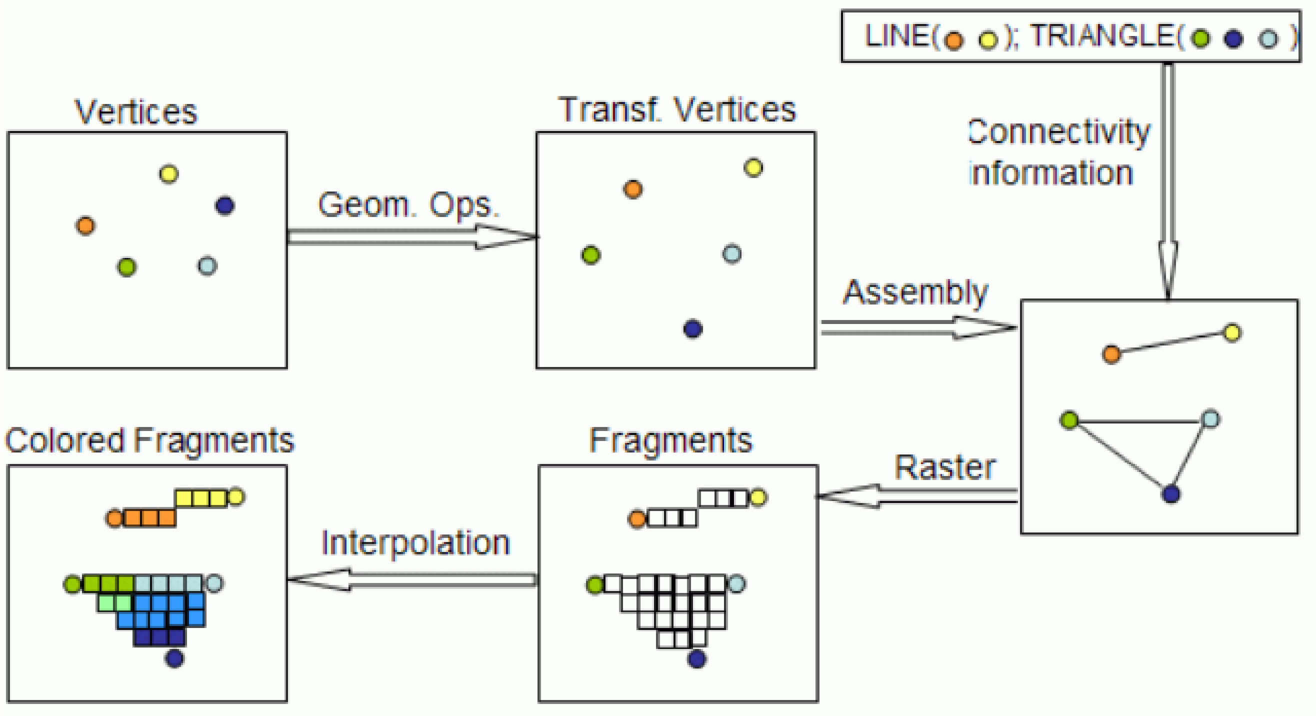
# Modern OpenGL

# What Changed?

- 30 years ago
  - Vertex transformation/fragment shading hardcoded into GPUs
  - Transform vertices with modelview/projection matrices
  - Shade with Phong lighting model only
- Now
  - More parts of the GPU are programmable (but not all)
  - Custom vertex transformation
  - Custom lighting model
  - More complicated visual effects
  - Shadows
  - Displaced and detailed surfaces
  - Simple reflections and refractions

# OpenGL Rendering Pipeline

- Although OpenGL pipeline is fast it is limited due to its fixed functionality

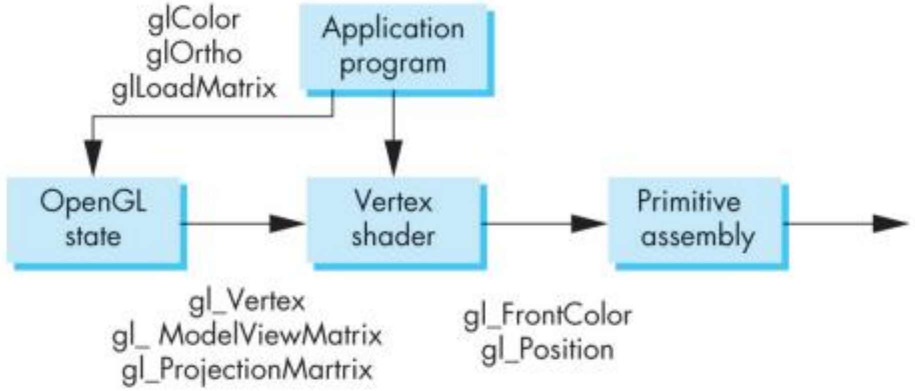


# From Fixed to Programmable: GLSL

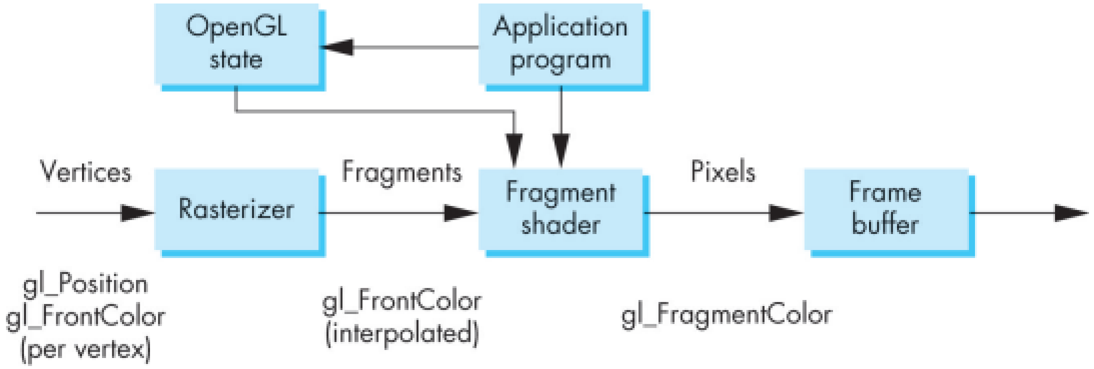
- GLSL : **G**raphics **L**ibrary **S**hading **L**anguage
  - Syntax similar to C/C++
  - Language used to write shaders
  - Vertex, tessellation, geometry, fragment, compute
  - We only cover vertex and fragment shaders today
- Based on OpenGL
- Alternatives: Nvidia Cg and Microsoft HLSL

# From Fixed to Programmable

- Vertex Shader: Replace for vertex transformation stage




- Fragment Shader: Replace fragment texturing and coloring stage.

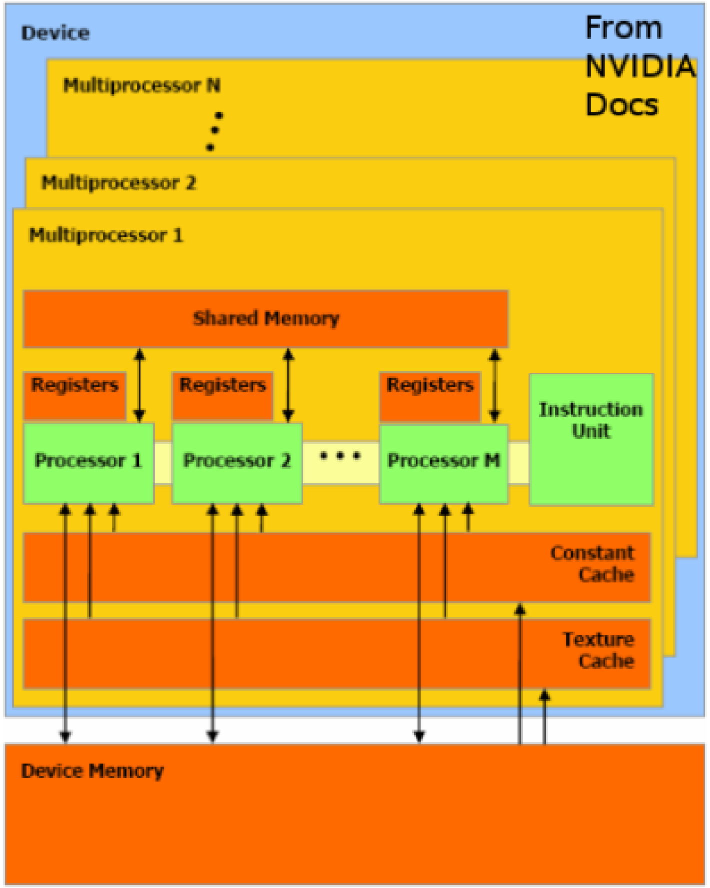


# From Fixed to Programmable

- Vertex Shader
  - Vertex position using ModelView and Projection matrices.
  - Lighting per vertex or per pixel.
  - Color and texture computation.
  - Must at least set gl_Position variable.
- Fragment Shader
  - Computing colors, texture coordinates per pixel.
  - Texture application.
  - Fog computation.
  - Output result by setting gl_FragColor

# From Fixed to Programmable

- Data Parallelism
  - Each data item can be processed independently
- SPMD
  - Single Program Multiple Data
- Stream Processors:
  - Many simple processors attached to fixed computational units



# GLSL: Data Types

- Scalar: Floating point (float), integer (int), boolean (bool).
- Vectors: One dimensional arrays with swizzling operator.
- Matrices: Square two-dimensional arrays with overloaded operators.
- Arrays and Structs: Same as in C.

```
void main()
{
    float f[16];
    int i = 0;
    vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
    mat4 m = gl_ProjectionMatrix;
    float r, g, b;

    r = red.r;
    g = red.g;
    b = red.b;
    red.rgb = vec3(0.5, 0.0, 0.0);
}
```

# GLSL: Data Qualifiers

- Const: A compile-time constant, or a function parameter that is read-only
- Attribute: Linkage between a vertex shader and OpenGL for per-vertex data
- Uniform: Value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application.
- Varying: Linkage between a vertex shader and a fragment shader for interpolated data.

# GLSL: Operators and Functions

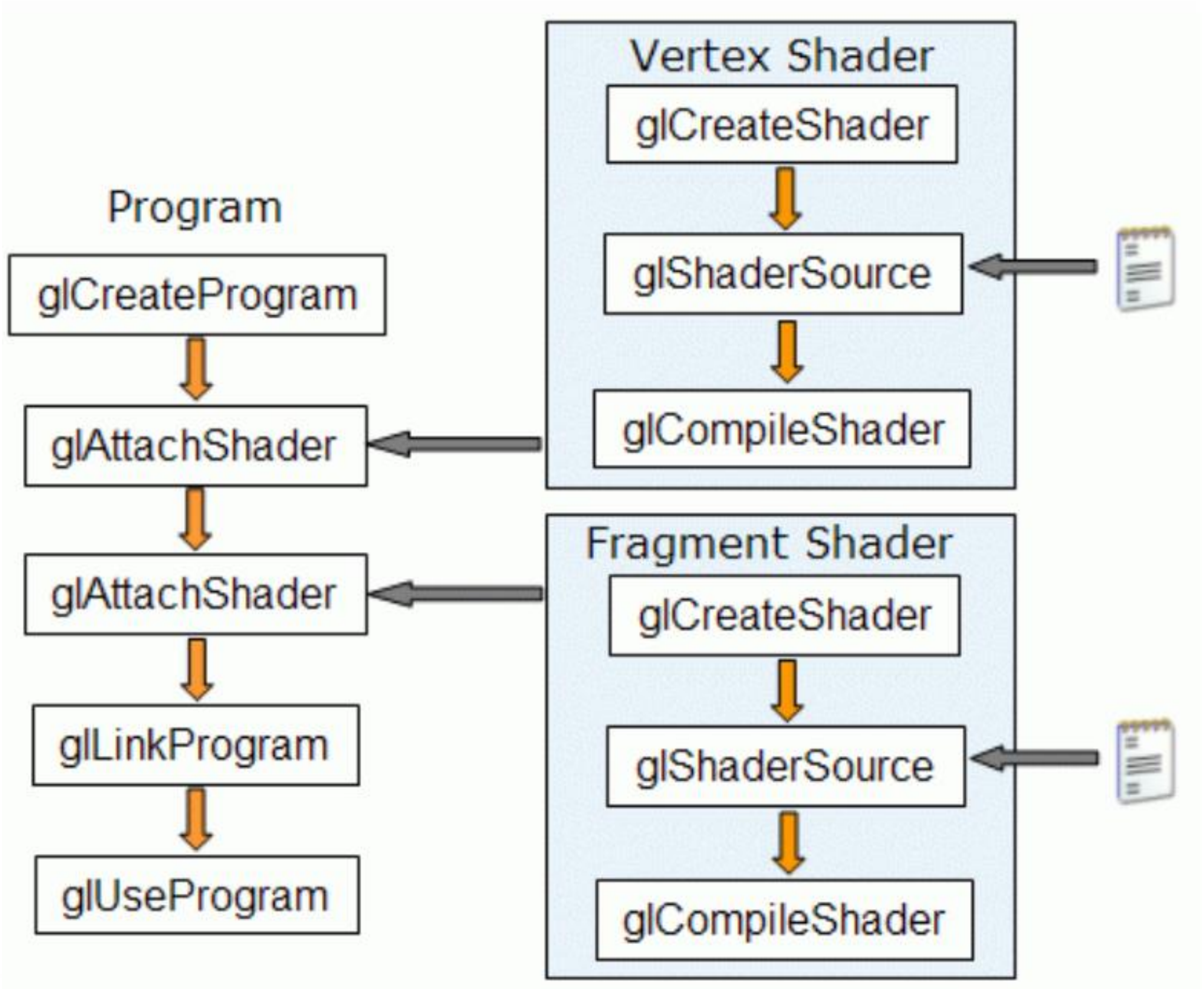
- Matrix-vector operations behave as expected.
- Swizzling operator allows for convenient access to elements of a vector: x,y,z,w; r,g,b,a; s,t,p,q.
- Can access built-in functions:
  - Trigonometric: asin, acos, atan.
  - Mathematical: pow, log2, sqrt, abs, max, min.
  - Geometric: length, distance, dot, normalize, reflect.
- Can make your own functions, but must qualify variables as in, out, inout.

# GLSL Programming in a Nutshell

- Create shader programs
- Create buffer objects and load data into them
- “Connect” data locations with shader variables
- Render

# GLSL: Create Shader Programs

- To use shaders in an OpenGL program, we need to load them into memory, attach them, compile them and link the program.



# GLSL: Create Shader Programs

```
// 1. retrieve the vertex/fragment source code from filePath
std::string vertexCode;
std::string fragmentCode;
std::ifstream vShaderFile;
std::ifstream fShaderFile;
// ensure ifstream objects can throw exceptions:
vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
try { ... }
catch (std::ifstream::failure& e) { ... }
const char* vShaderCode = vertexCode.c_str();
const char* fShaderCode = fragmentCode.c_str();
// 2. compile shaders
unsigned int vertex, fragment;
// vertex shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
checkCompileErrors(vertex, "VERTEX");
// fragment Shader
fragment = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment, 1, &fShaderCode, NULL);
glCompileShader(fragment);
checkCompileErrors(fragment, "FRAGMENT");
// shader Program
ID = glCreateProgram();
glAttachShader(ID, vertex);
glAttachShader(ID, fragment);
glLinkProgram(ID);
checkCompileErrors(ID, "PROGRAM");
// delete the shaders as they're linked into our program now and no longer necessery
glDeleteShader(vertex);
glDeleteShader(fragment);
```

# GLSL: Create Buffer Objects

- Vertex Array Object
    - An object which contains one or more VBOs and is designed to store the information for a complete rendered object
- Vertex Buffer Object
    - A memory buffer in the high speed memory of your video card designed to hold information about vertices.

```
// set up vertex data (and buffer(s)) and configure vertex attributes
// ------------------------------------------------------------------
float vertices[] = { ... }
// first, configure the cube's VAO (and VBO)
unsigned int VBO, cubeVAO;
glGenVertexArrays(1, &cubeVAO);
glGenBuffers(1, &VBO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glBindVertexArray(cubeVAO);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// normal attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

# GLSL: Connect Data with Shader Variables and Render

```
// be sure to activate shader when setting uniforms/drawing objects
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
lightingShader.setVec3("lightPos", lightPos);
lightingShader.setVec3("viewPos", camera.Position);

// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
lightingShader.setMat4("projection", projection);
lightingShader.setMat4("view", view);

// world transformation
glm::mat4 model = glm::mat4(1.0f);
lightingShader.setMat4("model", model);

// render the cube
glBindVertexArray(cubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

# Example: Red Vertex Shading

- Vertex Shader

```
// Color all vertices red
const vec4 RedColor = vec4(1.0, 0.0, 0.0, 1.0);

void main()
{
    gl_Position    = ftransform();
    gl_FrontColor = RedColor;
}
```

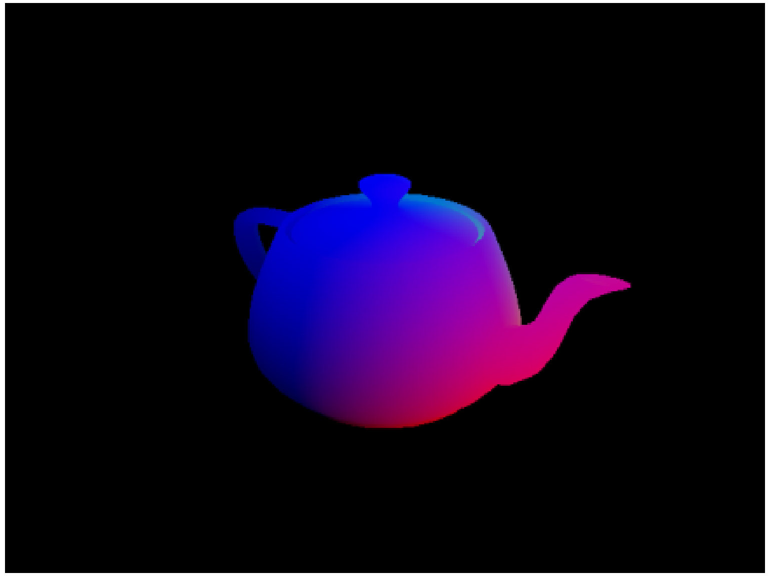
华东师范大学计算机科学与技术学院
School of Computer Science and Technology

# Example: Color based on Vertex

- Vertex Shader

```
// Color all vertices based on normalized vertex coordinates
const vec4 RedColor = vec4(1.0, 0.0, 0.0, 1.0);

void main()
{
    gl_Position   = ftransform();
    gl_FrontColor = normalize(gl_Vertex);
}
```



# Example: Scaling over Time

- Vertex Shader

```
// Scale vertices over time
uniform float ElapsedTime;

void main()
{
    float s;

    s = 1.0 + 0.5*sin(0.005*ElapsedTime);

    gl_Position   = gl_ModelViewProjectionMatrix*(vec4(s, s, s, 1.0)*gl_Vertex);
    gl_FrontColor = gl_Color;
}
```

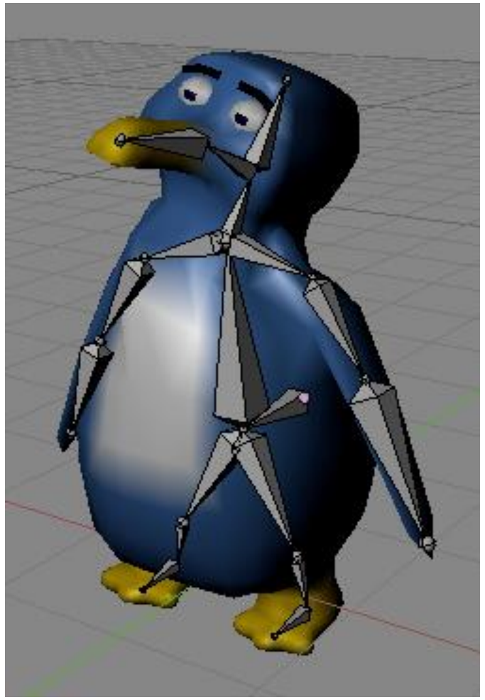
- OpenGL

```
void set_elapsed_time()
{
    GLint etloc;

    /* Set shader uniform variable */
    etloc = glGetUniformLocation(prog_id, "ElapsedTime");
    glUniform1f(etloc, glutGet(GLUT_ELAPSED_TIME));
}
```

# Fun with GLSL

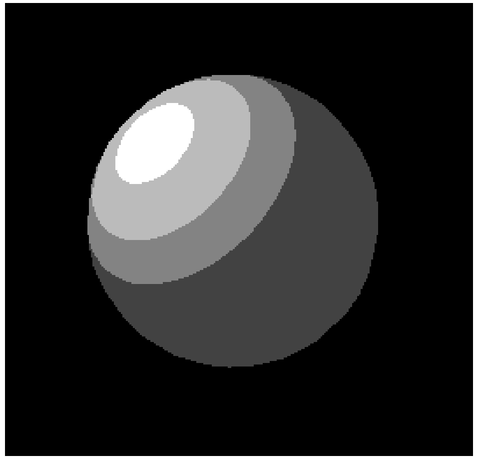
- Vertex shaders: what happens if you vary the coordinates with time and position?



# Fun with GLSL

- Toon shading: Step the intensity instead of smoothly varying it





```
if (intensity > 0.95)      color = vec4(1.0, 0.5, 0.5, 1.0);
else if (intensity > 0.50) color = vec4(0.6, 0.3, 0.3, 1.0);
else if (intensity > 0.25) color = vec4(0.4, 0.2, 0.2, 1.0);
else                       color = vec4(0.2, 0.1, 0.1, 1.0);
```

# Fun with GLSL

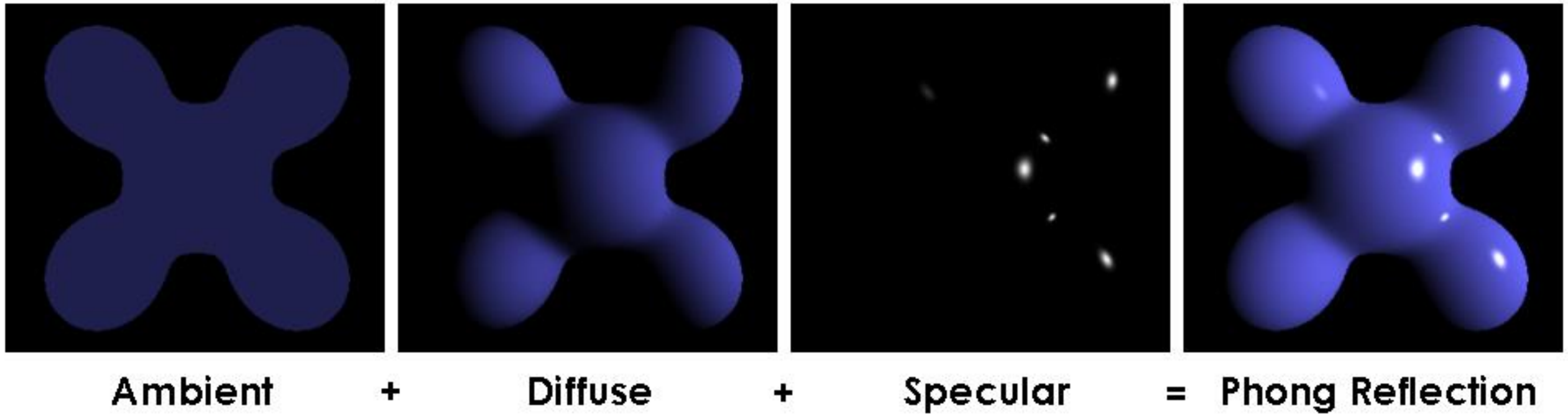
- Glow / bloom effect
  - Render parts of the scene that glow / are bright
  - Blur that texture, and overlay it on the scene

# Resources

- OpenGL Shading Language Specs:
  - http://www.opengl.org/documentation/glsl/
- Lighthouse 3D GLSL Tutorial:
  - http://www.lighthouse3d.com/opengl/glsl/index.php?intro
- Neon Helium GLSL Tutorial:
  - http://nehe.gamedev.net/data/articles/article.asp?article=21
- Clockwork Coders GLSL Tutorial:
  - http://www.clockworkcoders.com/oglsl/tutorials.html
- Swiftless Tutorials
  - http://www.swiftless.com/glsltuts.html

# Phong Illumination

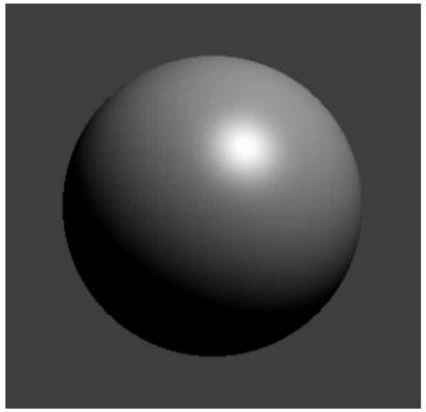
- Empirically divides reflection into 3 components
  - Ambient
  - Diffuse (Lambertian)
  - Specular



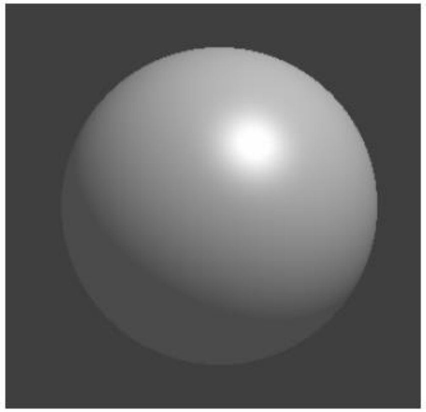
# Ambient Light

- Independent of location of viewer, location of light, and curvature of surface
  - $I = I_a k_a$
  - $I_a$ is intensity of ambient light
  - $k_a$ is ambient coefficient of surface

- Note: this is a total hack, of course



no ambient



ambient

# Diffuse Reflection

- Component of reflection due to even scattering of light by uniform, rough surfaces
- Depends on direction of light and surface normal
- $I_d = I_p k_d max(\mathbf{L} \cdot \mathbf{N}, 0)$
- $I_p$ is intensity of point light

# Specular Reflection

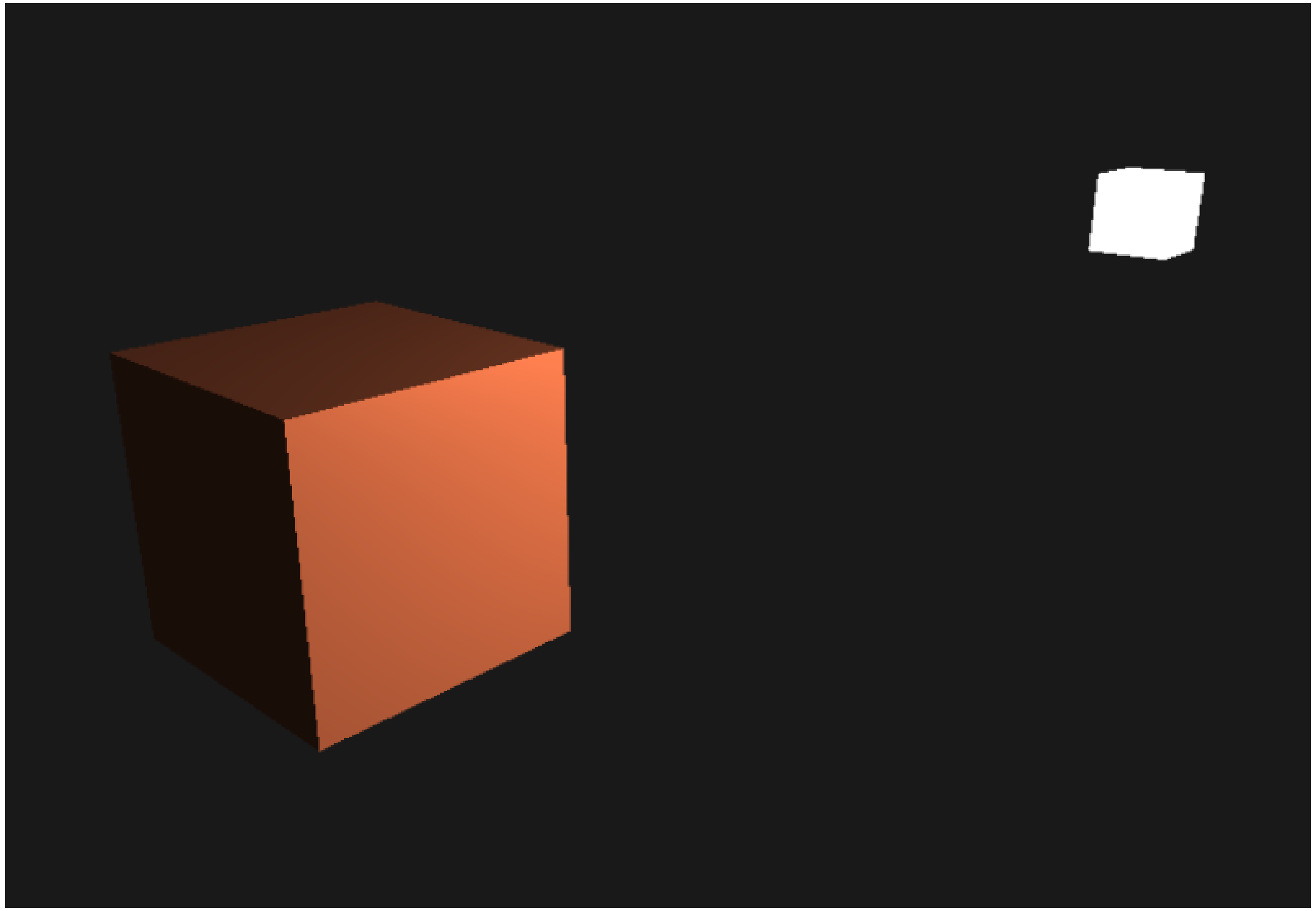
- Component of reflection due to mirror-like reflection off shiny surface
- Depends on perfect reflection direction, viewer direction, and surface normal
- $I_s = I_p k_s (\mathbf{R} \cdot \mathbf{V})^n$
- $n$ is specular exponent, determining falloff rate

# Illumination with Color

- Surface reflection coefficients and light intensity may vary by wavelength
- For RGB color
    - Light intensity specified for R, G, and B
    - Surface reflection coefficients also for R, G, B
    - Compute reflected color for R, G, and B

# Assignment

- 实验编号：6
- 实验名称：Phong光照模型
- 实验内容
  - 阅读GLSL绘制代码
  - 实现Phong光照模型



# Extra Credit

- Could you try to implement different light models?
  - Point lights
  - Directional lights
  - Spot lights
  - Area lights

# Reference


- https://en.wikipedia.org/wiki/Phong_reflection_model
- https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf
- https://graphicscompendium.com/gamedev/15-pbr