# OpenGL Shading Language

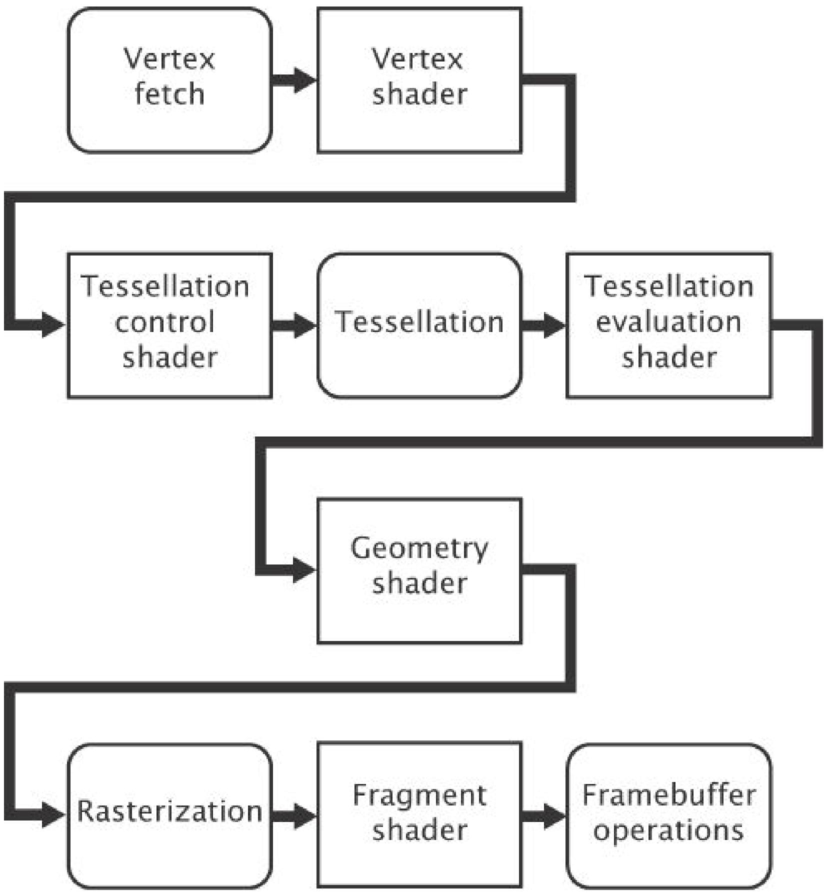
# Intended Learning Outcomes

- Briefly introduce programmable shader and OpenGL Shading Language

# Limitation of OpenGL Pipeline

- It is a fixed function pipeline
- No longer matches the way that modern graphics hardware operates. It is unable to take full advantage of the power available in GPU, with the result that rendering performance suffers

# Programmable-function OpenGL Pipeline




The following are programmable

- Vertex shader
- Tessellation control shader
- Tessellation evaluation shader
- Geometry shader
- Fragment shader

# OpenGL Shading Language (GLSL)

- GLSL is a C-like language designed to directly support the development of shader
- High level, easy to use programming language that works well with OpenGL, and as hardware independent as possible
- Only looks like C or C++, but not exactly (differences in how function parameters are handled, much stricter type checking, many familiar C and C++ data types and language constructs intentionally not included)

# Example: Phong shader

- Cannot implement using OpenGL because of the fixed function pipeline
- Can implement using GLSL by programming the vertex shader and the fragment shader

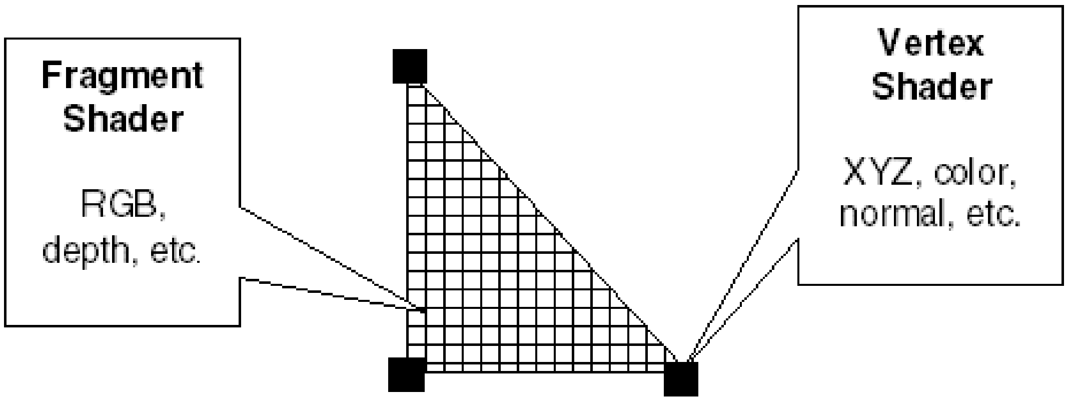
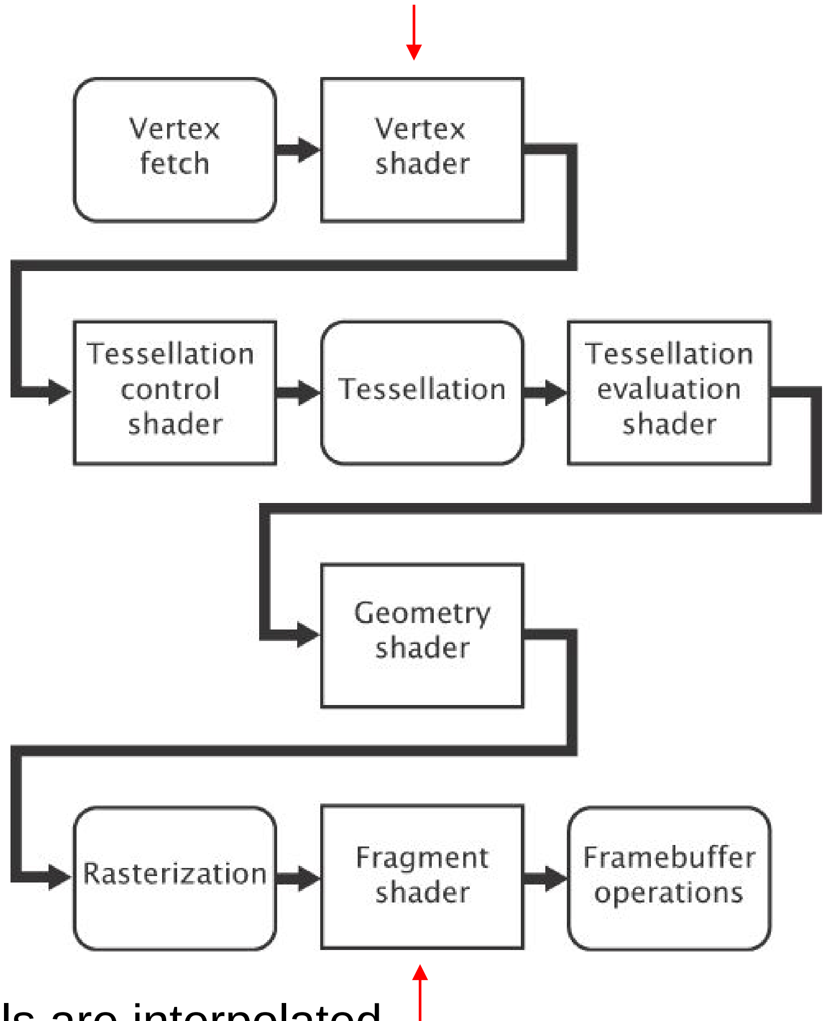



Figure from
http://climserv.ipsl.polytechnique.fr/documentation/idl_help/About_Shader_Programs.html




The normals are interpolated by the rasterizer

# Vertex shader code

```
in vec4 vPosition;        // input vertex array object
in vec4 Normal;           // input normal at each vertex

uniform mat4  ModelView;
uniform vec4  LightPosition;
uniform mat4  Projection;

out vec3  N;
out vec3  L;
out vec3  E;
```

```
void main ()
{
      gl_Position = Projection∗ModelView∗vPosition;
                              // vPosition is the position of the viewpoint
      N = Normal.xyz;
      L  = LightPosition.xyz – vPosition.xyz
      if  (LightPosition.w == 0.0)   L = LightPosition.xyz;
                              // take care of the situation of lighting direction
      E = vPosition.xyz;
}
```

The rasterizer will interpolate the normal N

# Fragment shader code

```
uniform vec4  AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4  ModelView;
uniform vec4  LightPosition;
uniform float   Shininess;

in vec3  N;
in vec3  L;
In vec3  E;
```

```
void main ()
{
    vec3   NN = normalize(N);
    vec3   EE  = normalize(E);
    vec3   LL   = normalize(L);
    vec4   ambient, diffuse, specular;
    vec3   H = normalize(LL+EE);
    float    Kd = max(dot(LL, NN), 0.0);    // N · L
    float    Ks = pow (max(dot(NN, H), 0.0), Shininess);
    ambient = AmbientProduct;                     // k_a I_a
    diffuse   = Kd∗DiffuseProduct;               // k_d I_l (N · L)
    if (dot(LL,NN) < 0.0)   specular = vec4 (0.0, 0.0, 0.0, 1.0);
    else specular = Ks∗SpecularProduct;
        // Use the approx. formula for specular reflection k_s I_l (N · H)^{n_s}
        // H = normalize(L + V)
    gl_FragColor = vec4 ((ambient + diffuse + specular).xyz, 1.0);
}
```

# Some other techniques implemented using shader

- shadow mapping
- deferred shading
- toon shading

Refer to the demo programs

# References


- Text. Ch. 22
- E. Angel, D. Shreiner, Interactive Computer Graphics, A Top-down Approach with Shader-based OpenGL, 6th Ed., Ch. 5., pp. 295-296
- G. Sellers, R.S. Wright, N. Haemel, OpenGL SuperBible, 7th Ed., 2016, Ch. 1