A shader is a small program developed by you that lives in the GPU. A shader is written in a special graphics language called **OpenGL Shading Language** (GLSL). A shader takes the place of two important stages in the OpenGL pipeline: "Per-Vertex Processing" and "Per-Fragment Processing" stage.  
  
Long ago, these stages were provided by OpenGL as static-configurable stages. However, this prevented graphical effects creativity. OpenGL solved this problem by creating the concept of shaders.  
  
There is one shader for each stage. The shader processed in the "Per-Vertex" stage is called **Vertex Shader**. The shader processed in the "Per-Fragment" stage is called **Fragment Shader**.  
The ultimate goal of the *Vertex Shader* is to provide the final transformation of the character’s vertices to the rendering pipeline. The goal of the *Fragment Shader* is to provide coloring and texture data to each pixel heading to the framebuffer.  
  
**Passing data to Shaders**  
Data is passed down from your application to the GPU through OpenGL Objects. This data is first received by the **Vertex Shader**. Inside the vertex shader, this data is defined as an **Attribute**. Unlike the **Vertex Shader**, the **Fragment Shader** can not receive data directly. Any data that is required by the *Fragment Shader* is passed down from the *Vertex Shader* and is defined as a **Varying** data type in both shaders. The only exception, is data defined as **Uniform**. *Uniform* data can be received by the *vertex shader* and *fragment shader*.  
  
**Attributes, Varying and Uniforms**  
  
GPU data received by a *vertex shader* is considered an **Attribute**, for example, vertex, normal and UV-coordinates. A **Uniform** is a global variable that can be received by both *vertex* and *fragment* shader.  
  
A *fragment shader* can not receive attribute data from the GPU directly. Thus, attributes can not defined in fragment shader. If a fragment shader requires the data of an attribute, a new variable of type **Varying** is defined in the vertex shader.  
The attribute data is copied onto the *varying* variable. The *Varying* variable is then sent from the vertex shader to the fragment shader for further processing.  
  
**Shaders Output**  
Although shaders allows you to be creative in their implementation, shaders require data for two output built-in variables: **gl\_Position** and **gl\_FragColor**.  
  
*gl\_Position*  
The gl\_Position is used by the vertex shader to hand over the *transformed* vertices to the rendering pipeline.  
  
*gl\_FragColor*  
The gl\_FragColor is used by the fragment shader to provide coloring and texture data to each pixel heading to the framebuffer.

Generally speaking lighting calculations can be performed in any coordinate space object space, world space, eye/view space. If you are calculating lighting calculations in the eye/view space then you have to know the necessary steps that have to be taken when implementing this type of lighting. You'd have to multiply the current vertex and light position with the modelview matrix, and then transform the per vertex normals to eye space using the inverse transpose of the modelview matrix. Then obtain the vector from the position of the light in eye space to the position of the vertex in eye space, and do a dot product(vector do product from first year vector mechanics) of this vector with the eye space normal. This gives us the diffuse component.  
  
Then you have to calculate the view vector and the halfway vector between the light and the view vector.  
  
These vectors are used for specular component calculation in the Blinn Phong lighting model (used in fixed function pipeline). The specular component is then obtained using pow( dot( N, H), σ), where σ is the shininess value; the larger the shininess, the more focused the specular component is.  
  
The final color is then obtained by multiplying the diffuse value with the diffuse color and the specular value with the specular color.  
  
Then the calculations can be moved from the vertex shader to the fragment shader to create a different visual effect.  
  
 Generally speaking there are many different algorithms that may be implemented using GLSL to produce different visual effects and you generally have to pick these up as you work, as you study, as you research.

To help you get things started, here's a boilerplate:  
  
**Vertex Shader** (executed per vertex)

1. **void** main() {
2. gl\_Position = gl\_ModelViewProjectionMatrix\*gl\_Vertex;
3. }

**Fragment Shader**(executed per fragment (pixel))

1. **void** main() {
2. gl\_FragColor = vec4(1.0, 0.0, 0.0, 1.0); //RGBA
3. }

That's it. What does the code above do? It renders your geometry to screen in red.  
  
What's next? Why not stretch the "height" of your geometry by a factor of 2?  
**Vertex Shader**

1. **void** main() {
2. vec4 stretched = gl\_Vertex\*vec4(1.0, 2.0, 1.0, 1.0); //multiply component-wise
3. gl\_Position = gl\_ModelViewProjectionMatrix\*stretched;
4. }

How about making your geometry changes color according to where it is?  
**Vertex Shader**

1. **out** vec4 interpolatedPosition;
2. **void** main() {
3. gl\_Position = gl\_ModelViewProjectionMatrix\*gl\_Vertex;
4. interpolatedPosition = gl\_ModelViewMatrix\*gl\_Vertex;
5. }

**Fragment Shader**

1. **in** vec4 interpolatedPosition
2. **void** main() {
3. gl\_FragColor = vec4(interpolatedPosition.xyz, 1.0);
4. }

At this point you probably ask what those in/out variables mean. There are also attribute and uniform variables. Once you master vertex and fragment shader and the three types of variables then you should move on to geometry shader and tessellation shader.