
GPU Programming using GLSL and VTK

John O'Brien

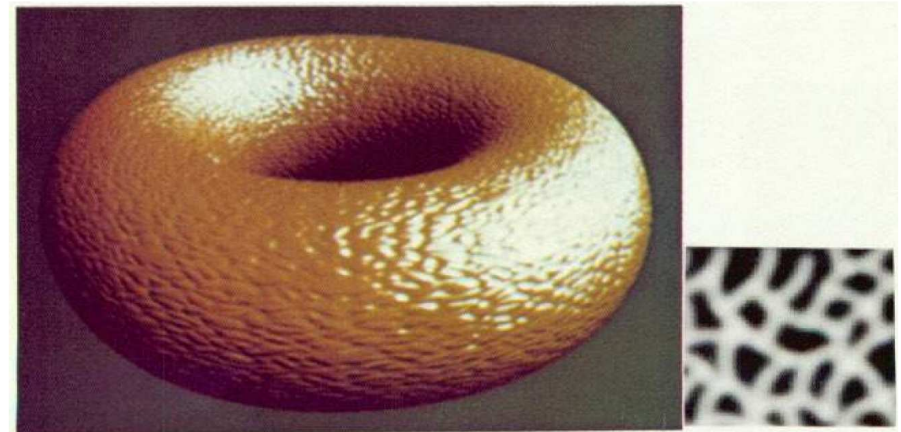
University of Loughborough

Outline

- Why use shaders?
- A short history of GPU development and procedural shaders
- The OpenGL Shader Language (GLSL)
- Integrating GLSL with VTK
- Some examples

Graphics shaders

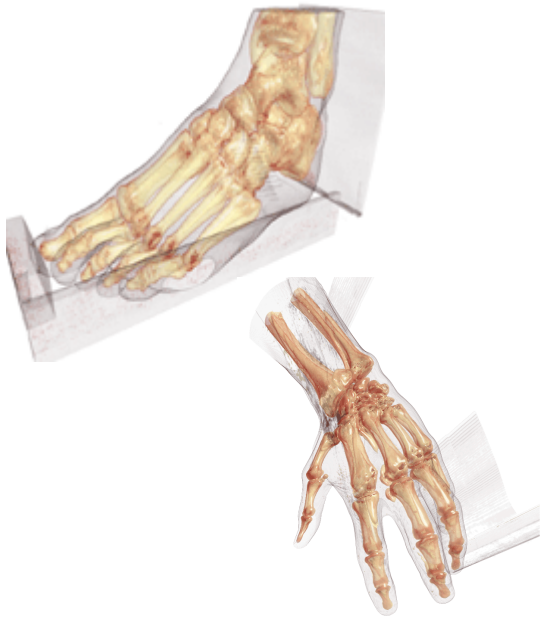
- Procedural graphics shaders have been around since the early days of computing.
- Developed by scientists with a good understanding of the maths underpinning physical environments such as Jim Blinn.
 - This led to the photo realism CG seen in the movies
 - Designed to mimic reality without the complexity (e.g. bump mapping)
- Initially all CPU based



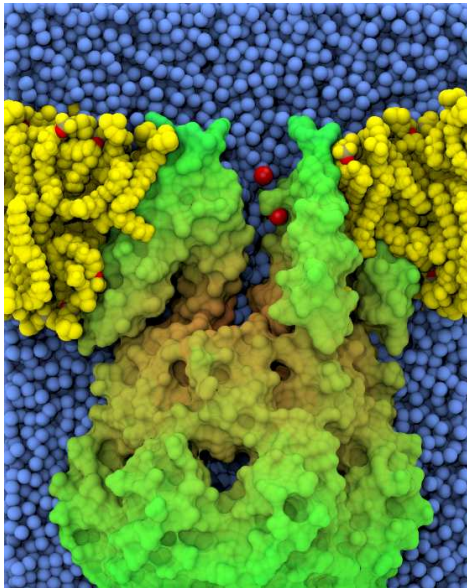
Simulation of wrinkled surfaces, Jim Blinn, 1978

Why use shaders?

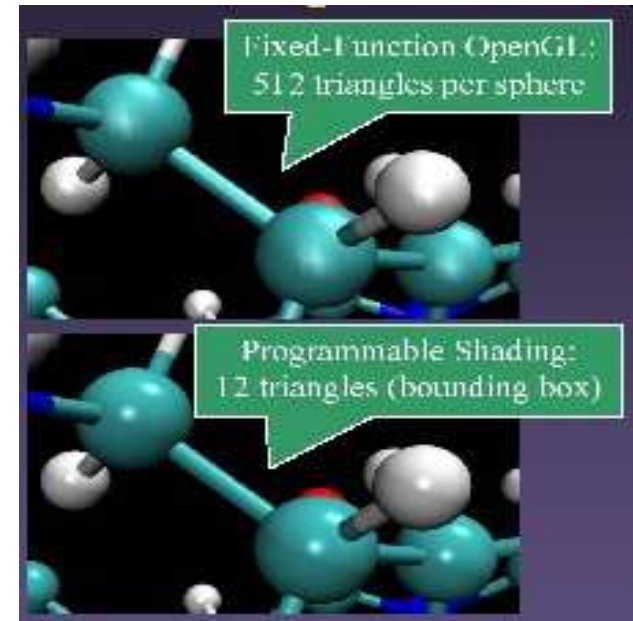
- Procedural shaders allow us to achieve a much more tailored visualization than with the fixed OpenGL API



*Improve Realism
and Accuracy
(ray tracing)*



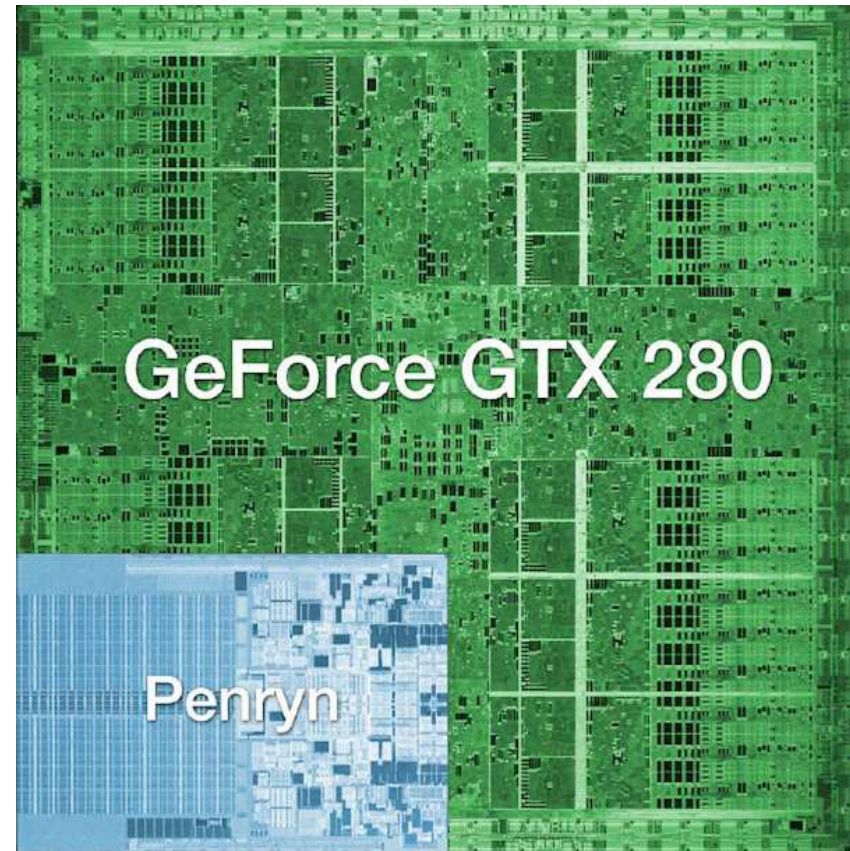
*Improve Visual Relationships
(ambient occlusion)*



Improve Throughput

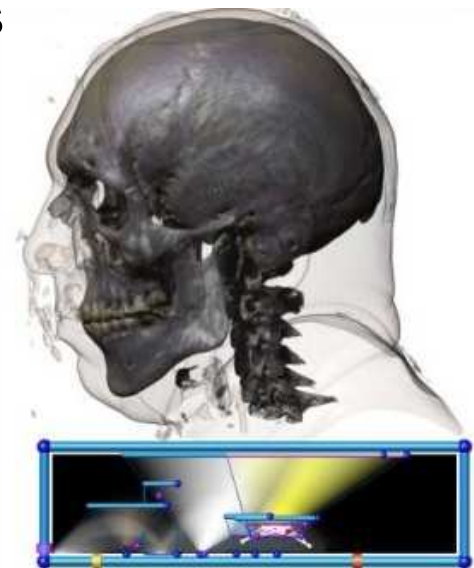
The Rise of Graphics Processors (GPUs)

- CPUs do lots of different tasks fairly well - GPUs need to do one specific task very well.
 - SGI and others developed hardware that implemented most of the functionality in fixed hardware
 - Driven by demand for interactive graphics in simulators and games.
- Development of shaders continues but is restricted to high precision software solutions such as RenderMan



OpenGL

- The OpenGL API is defined to provide a hardware abstraction between vendors. Defines a largely fixed pipeline for rendering.
- Over time vendors are driven to utilise adhoc programmable firmware in order to keep up with changes to the API and extensions.
 - Researchers use these extensions and in novel ways to achieve effects seen in procedural shaders
 - Managing extensions and updating assembly based programs is complex.
 - OpenGL architecture board agrees the GLSL standard to reduce these issues.
- NVIDIA's Cg and Microsofts High-Level Shader Language are broadly similar to GLSL



Utah's Simian volume renderer

Programmable Units

The OpenGL Rendering Process

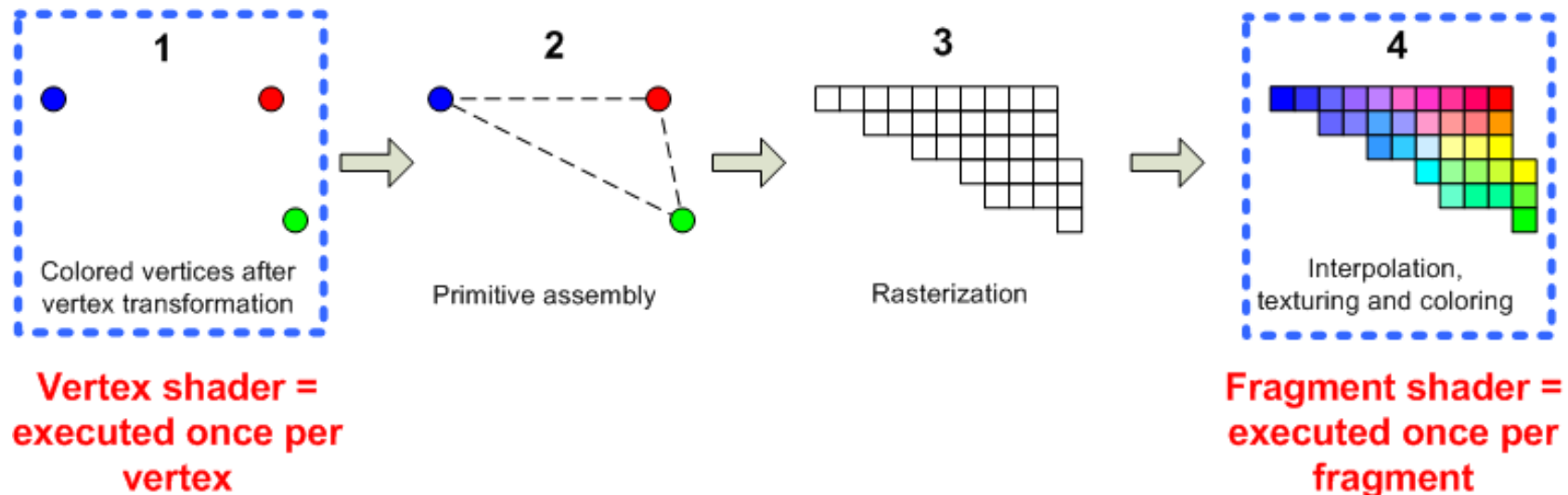


Diagram courtesy of Ralph Brecheisen, *Real-time volume rendering with hardware-accelerated raycasting*

Creating replacement shaders for each unit

- GLSL is based on the syntax of ANSI C:
 - with a dash of C++ constructors to enable function overloading for different types.
 - Has enhanced types to make vector and matrix arithmetic very succinct and encourage parallelism. For example
 - `vec2 a = vec2(1.0,2.0); vec3 b = vec3(4.0,5.0,6.0); vec2 c = a + b.xy;`
 - Defines special reserved variable names to access OpenGL state, prefixed with `gl_`
- Each shader must define a single main entry point:

```
void main() {  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

The Vertex Processor

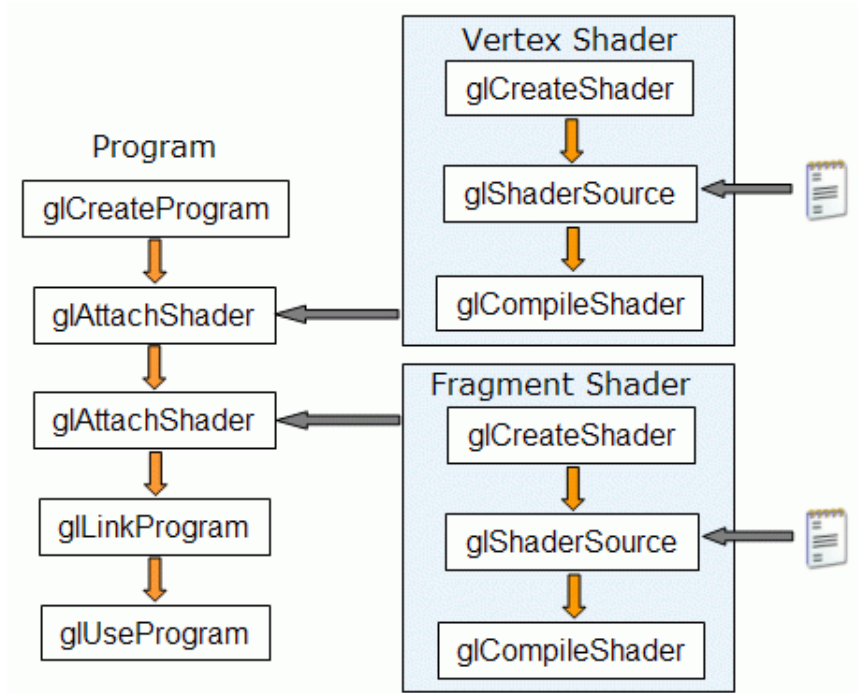
- Performs traditional graphics operations:
 - Vertex transformation
 - Normal transformation and normalization
 - Texture coordinate generation and transformation
 - Lighting and colour material application
- When a custom vertex shader is loaded, the shader is responsible for performing all these operations!
 - Don't Panic – Chapter 9 of the GLSL guide (*Orange Book*) provides reference implementation for all these operations
 - The most crucial operation is the transformation from a local coordinate system to global system of each vertex and so very straight forward perform:
 - `gl_Position = ftransform();`

The Fragment Processor

- The fragment processor operations include:
 - Texture access and applying texture values
 - Colouring
 - Fog
- The processor is passed variables from the vertex processor, and also the location of the pixel in world space computed during primitive assembly.
 - *Making per pixel shading possible.*
- The output of these operations is placed in `gl_FragColor` and `gl_FragDepth`
 - These values are then composited into the framebuffer through fixed hardware for speed.
 - *However, unlike the vertex shader, the fragment processor is free to discard the result all together!*

Loading custom shaders

- New shaders are loaded as strings into a *program* reference
 - A program can contain many shader sources
 - But only one main entry point per vertex / fragment processor



(Courtesy of <http://www.lighthouse3d.com/opengl/glsl/>)

Communicating with loaded shaders

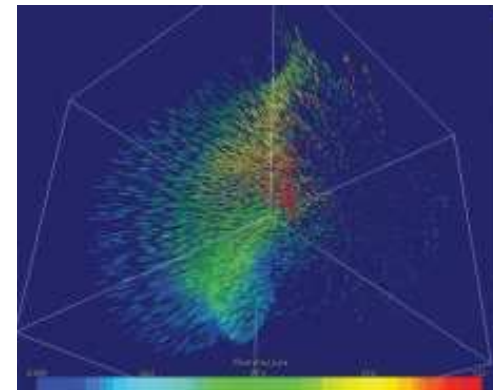
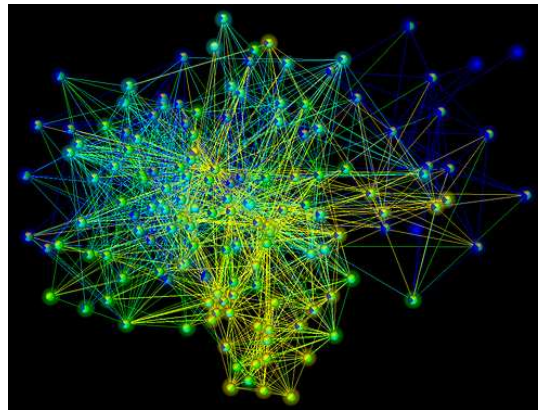
- Applications are able to communicate with a processing unit through several means:
 - Attributes – Frequently changing information from the application to a vertex shader usually set per vertex.
 - Each attribute is stored in a fixed location with an index to that location
 - Uniform – Infrequently changing information from the application to either a vertex or fragment shader.
 - User defined, OpenGL must be queried to retrieve a pointer to the memory.
 - Varying – Variables that are passed from the vertex shader to the fragment shader.
 - Textures – Used for transferring large amounts of the data from the application to fragment shader.

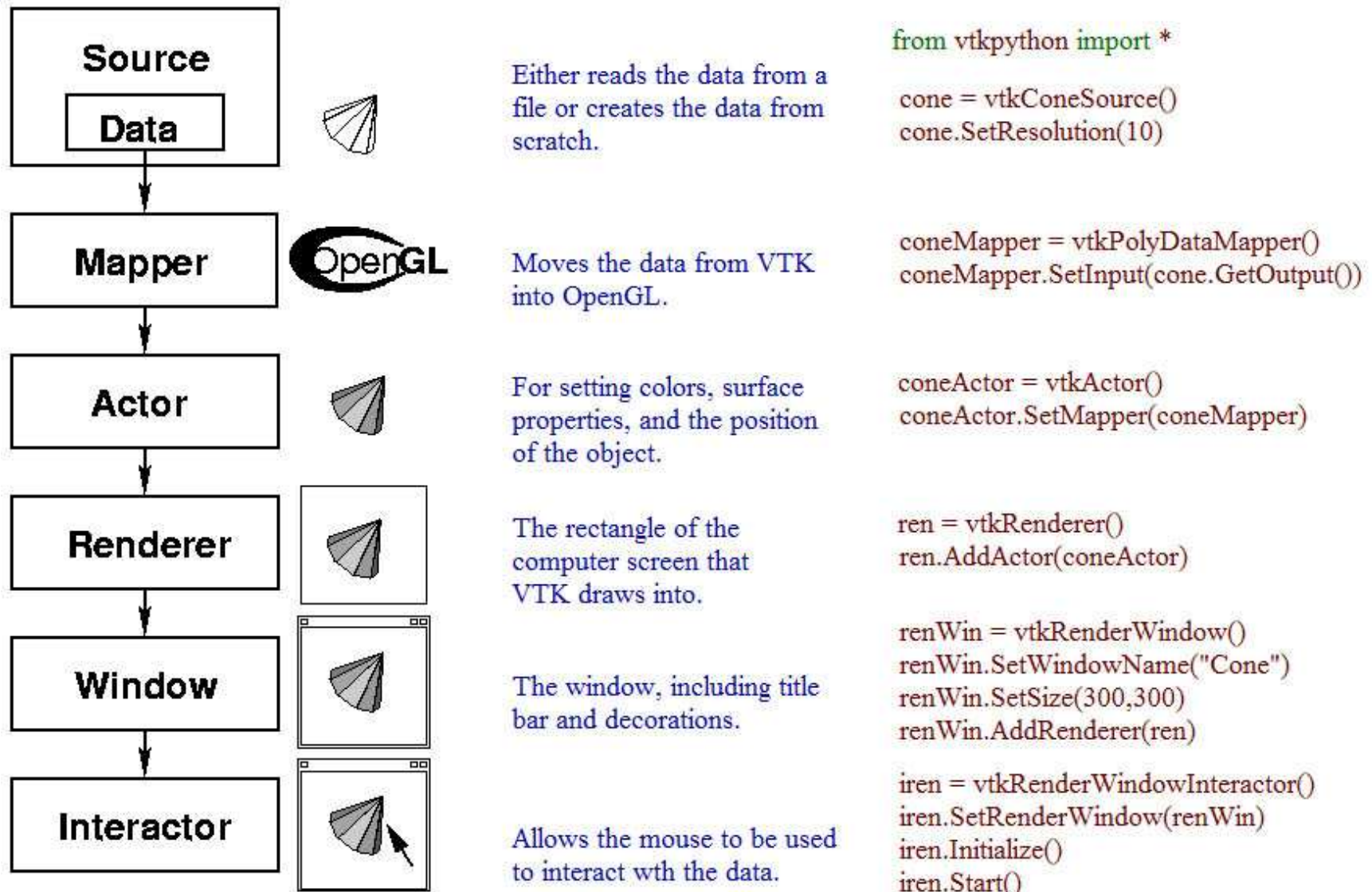
How do we utilise shaders within
visualizations?

*(or using custom shaders without
rebuilding the wheel)*

The Visualization Toolkit

- A flexible open source toolkit for loading a range of different structured or unstructured data.
 - Provides a large number of readers for different data types
- Supports a wide variety of visualization algorithms including: scalar, vector, tensor, texture, and volumetric methods;
-and provides a very convenient method for loading GLSL shaders.





Courtesy of David Gobbi (<http://www.imaging.robarts.ca/~dgobbi/vtk/vtkcourse/index.html>)

Loading an GLSL program

- VTK defines an GLSL program as a material.
 - Specified by an XML file that links to source code file for one or more GLSL shaders.
- Loading this file is straight forward:

```
prop = coneActor.GetProperty()  
prop.LoadMaterial("F:\\MyShaders.xml")  
prop.ShadingOn()  
prop.AddShaderVariable("Rate", 1, 1.0)
```

Format of the XML Material file

```
<Material name="Mat1" NumberOfProperties="1" NumberOfVertexShaders="1" NumberOfFragmentShaders="1">
```

```
  <Property name="Property1">
```

```
    <Member name="AmbientColor" number_of_elements="3" type="Double" value="0.0 0.0 0.15"> </Member>
```

```
    .....
```

```
    <Member name="EdgeVisibility" number_of_elements="1" type="Int" value="0"> </Member>
```

```
  </Property>
```

```
  <Shader scope="Vertex" name="brickVert" location="OrangeBook/Ch06/Ch06BrickVert.glsl" language="GLSL"
entry="main" args="-DVERTEX_PROGRAM">
```

```
    <Uniform type="vec3" name="LightPosition" number_of_elements="3" value="0.0 10.0 4.0"> </Uniform>
```

```
  </Shader>
```

```
  <Shader scope="Fragment" name="brickFrag" location="OrangeBook/Ch06/Ch06BrickFrag.glsl" language="GLSL"
entry="main" args="-DFRAGMENT_PROGRAM">
```

```
    <Uniform type="vec3" name="BrickColor" number_of_elements="3" value="0.75 0.15 0.15"> </Uniform>
```

```
    <Uniform type="vec3" name="MortarColor" number_of_elements="3" value="0.75 0.75 0.75"> </Uniform>
```

```
    <Uniform type="vec2" name="BrickSize" number_of_elements="2" value="0.75 0.75"> </Uniform>
```

```
    <Uniform type="vec2" name="BrickPct" number_of_elements="2" value="0.75 0.75"> </Uniform>
```

```
  </Shader>
```

```
</Material>
```

Procedural Texture Example

Vertex Shader

```
void main(void)
{
    vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec    = normalize(-ecPosition);
    float diffuse   = max(dot(lightVec, tnorm), 0.0);
    float spec      = 0.0;

    if (diffuse > 0.0) {
        spec = max(dot(reflectVec, viewVec), 0.0);
        spec = pow(spec, 16.0);
    }

    // Phong Shading
    LightIntensity =
        DiffuseContribution * diffuse +
        SpecularContribution * spec;

    MCposition     = gl_Vertex.xy;
    gl_Position    = ftransform();
}
```



Fragment Shader

```
void main(void)
{
    vec3 color;
    vec2 position;
    vec2 useBrick;

    position = MCposition / BrickSize;

    if (fract(position.y * 0.5) > 0.5)
        position.x += 0.5;

    position = fract(position);
    useBrick = step(position, BrickPct);

    color = mix(MortarColor, BrickColor,
               useBrick.x * useBrick.y);
    color *= LightIntensity;
    gl_FragColor = vec4(color, 1.0);
}
```