Adding Programmable HW Shaders to VTK

Gary Templet
Sandia National Laboratories
Visualization and Scientific Computing - 8963
November 15, 2004

Abstract

VTK and ParaView architecture extensions and changes are proposed to allow the incorporation of programmable hardware shader technologies. Although the current target hardware libraries are NVidia's Cg and OpenGL2.0, the approach is general enough to allow incorporation of additional hardware libraries as they become available and are requested by VTK and ParaView developers and users. The architecture proposed here considers the needs of the application developer as well as the expectations of the end-user. Sandia is currenlty prototyping these changes for inclusion into VTK and ParaView.

1 Motivation

Modern graphics cards allow developers to create custom fragment and vertex shaders that replace the standard OpenGL vertex and fragment operations. Various shaders can be loaded by the application at any time during rendering, allowing each object in the scene to be rendered with a different shader, or, multiple objects can be rendered with the same shader. This new approach to graphics programming will have a great impact on the techniques used to produce rendered images and the quality of those images. At Sandia, the most immediate impact will be in the areas of Scientific Visualization and CAD (Computer-Aided Design) visualization.

1.1 Scientific Visualization

Sandia has done a large amount of work in developing hardware-based unstructured volume rendering techniques. The first versions of these renderers were written with extensions to the OpenGL library. The availability of programmable graphics hardware technologies in VTK and ParaView will make these techniques easier to use and develop.

1.2 CAD Visualization

Programmable graphics hardware will impact the CAD visualization world in two major areas.

First - Adding realism to interactive visualization applications. CAD users have long grumbled about unrealistic colors in visual representations of their models. While this certainly helps to distinguish between objects, there are many instances where it is desireable to have more realistic renderings.

Second - Fostering the development and use of Non-Photorealistic rendering.

1.3 Custom Visualization

Any new technology has an investigatory phase where all manner of unforeseen applications are applied to it. Much work has been done with GPUs (Graphics Processing Units) to leverage their computational capabilities to perform more than traditional redering techniques. It is hoped that the work proposed here will facilitate new and exciting applications of GPUs.

2 Very Brief HW Shader Overview

Programmable Vertex and Fragment shaders are designed to replace the default OpenGL fragment and vertex programs. At any given time a rendering pipeline may use a combination of programable hardware shaders and the default OpenGL shaders with the restriction that only one vertex and one fragment shader are active at any one time. The input parameters to hardware shaders are of basic two types, *Uniform Parameters* and *Varying Parameters*. Typically, uniform parameters are meant to be constant while a shader processes a set of uninterrupted primitives. Varying parameters correspond to specific hardware registers associated with each vertex or fragment. Their values are specified relative to and ther are processed with a specific vertex and fragment.

Modern programmable hardware shader technologies have been designed to facilitate incorporation with OpenGL applications. Once a shader has been defined, the mechanics of using it in the rendering pipeline can be generalized into a few steps:

Load the shader into application memory
Compile the shader either in the cpu or on the gpu
Bind the shader to the hardware rendering pipeline
Intialize Uniform parameters
Render Shader

Here 'Render Shader' implies sending graphics primitives to hardware as well as sending their corresponding varying parameters.

An application designed to render using hardware shaders bears the responsibility to perform the above steps. Since VTK's architecture lends itself to encapsulation and delegation of functionality, it is possible to provide an intuitive and flexible interface to working with hardware shader libraries. When considering how the steps above should be achieved in VTK and ParaView, two groups should be considered, shader developers, and end users of VTK-based applications, include ParaView.

2.1 Shader Developer's Interface

The first goal of the architecture changes proposed here is to isolate the shader developer from the mechanics of loading, compiling, binding, setting variables, etc. for their hardware shaders. It should be enough for a developer to completely specify a consistent shader in a specific shader language along with it's varying and uniform paramters, and have VTK handle the rest. If a problem arises with a shader, VTK report the error through it's error macros and default to use the standard OpenGL

pipeline or provide some other reasonable default.

There are two main hurdles for those wishing to incorporate specific shader techniques into their applications. The first is understanding the shading language itself and the second is the mechanics of integrating a hardware shader with an application. There is no way to avoid learning a new language if you wish to program with it, but through a few extensions of the VTK architecture, the mechanics of inserting a shader into the hardware pipeline can be encapsulated and delegated, thus freeing the application developer to focus on shader development.

2.2 End User's Interface

The second goal of this proposal is to present an inuitive interface to the end-user where the focus is on the end-result of the visualization and not specific shader techniques. In this vein, the XML material file also provides a consistent interface for shader developers; they simply define a shader and assign it to a vtkActor. Since it is applied at runtime, it also facilites changing hardware shaders 'on-the-fly'. Although it's not a focus of this proposal, the material file could also be extended to define members of vtkRIBProperty or vtkMESAProperty.

While the VTK/ParaView developer has a focus on developing any manner of hardware shaders, the end-user will not want to know how a particular rendering is achieved, or even what their fancy new graphics card might be doing. Instead, their focus will be on the stunning images produced by their application. From their perspective, these results are achieved by simply assigning the correct visual properties to each actor in the scene through an intuitive interface. To that end, the visual representation should be selected in a single operation from a set of predefined materials that are labeled with familiar terms that detail familiar idioms such as material composition, finishing techniques, processes, etc.

Many CAD applications achieve this intuitive interface by presenting to the end-user the concept of a materials library. These libraries typically contain enough information to fully specify the visual properties of an object, as defined by the application.

It is proposed that VTK and ParaView implement the concept of a material library, which consists of an extensible set of xml material files. When assigned to an vtkActor/vtkPVActor in a VTK/ParaView scene, the materials library can be used

to do the following in any combination:

Set some or all values for vtkProperty members
Define a Vertex Shader and it's parameters
Define a Fragment Shader and it's parameters

Of course, each application field would have it's own set of favorite and familiar materials, and developers would create new materials as needed. Just as VTK/ParaView developers contribute code they would also contribute materials libraries which would be available in Kitware's repository.

3 Extending VTK

Here is a brief summary of the proposed extensions to VTK that would realize these objectives. It should be noted that 'HW' refers to 'Hardware'. It is abbreviated in this document to facilitate the production of images and charts which would be quite cumbersome with 'Hardware' spelled out. The purpose of this document is to propose the architecture presented, the names of the objects used are simply working names and can be finalized later.

vtkProperty	extended to access a vtkXMLMaterialParser
vtkXMLMaterialParser	provides access to material library files
vtkHWShaderProperty	SubClass vtkProperty, manages vtkHWshaders
vtkHWShader	Base class for handling shader mechanics
vtkHWCgShader	Concrete vtkHWShader to handle NVidia's Cg
vtkHWGLSLShader	Concrete vtkHWShader to handle OpenGL20
vtkHWShaderChooser	Proxy vtkHWShader
vtkCgContext	Manage access to Cg run-time libraries
vtkGLSLContext	Manage access to OpenGL20 run-time libraries
vtkPainter et. al.	Alternative to vtkPolyDataMapper

Figure 1 is an informal representation of the data flow from the XML Material File to the uniform variable registers in a vertex and fragment program when using Cg (VP20 profile) shaders. Other hardware shader libraries would simply have their own instance of vtkHWShader in place of vtkHWCgShader. In this figure, as in Figure 2, arrows represent data flow and interior boxes indicate a 'Has A' relationship.

Similarly,Figure 2 shows the data flow for varying parameters for a vertex program. Here point fields are mapped as varying parameters to specific vertex attributes.

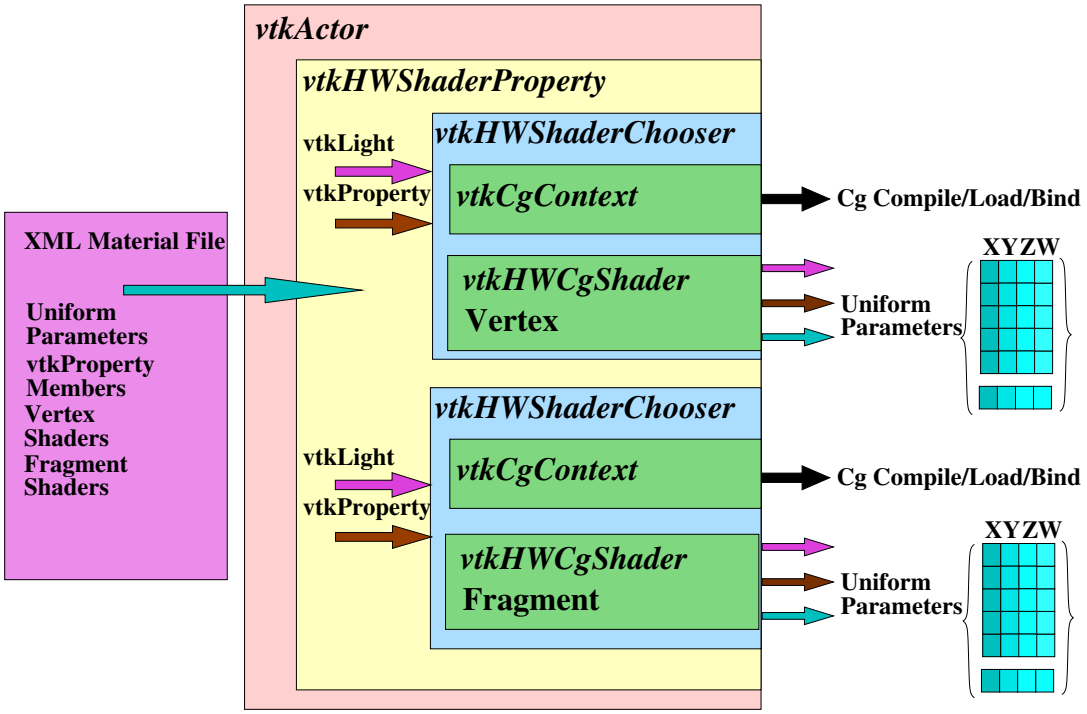

Figure 1: Data Flow Diagrm for Uniform Variables, Cg

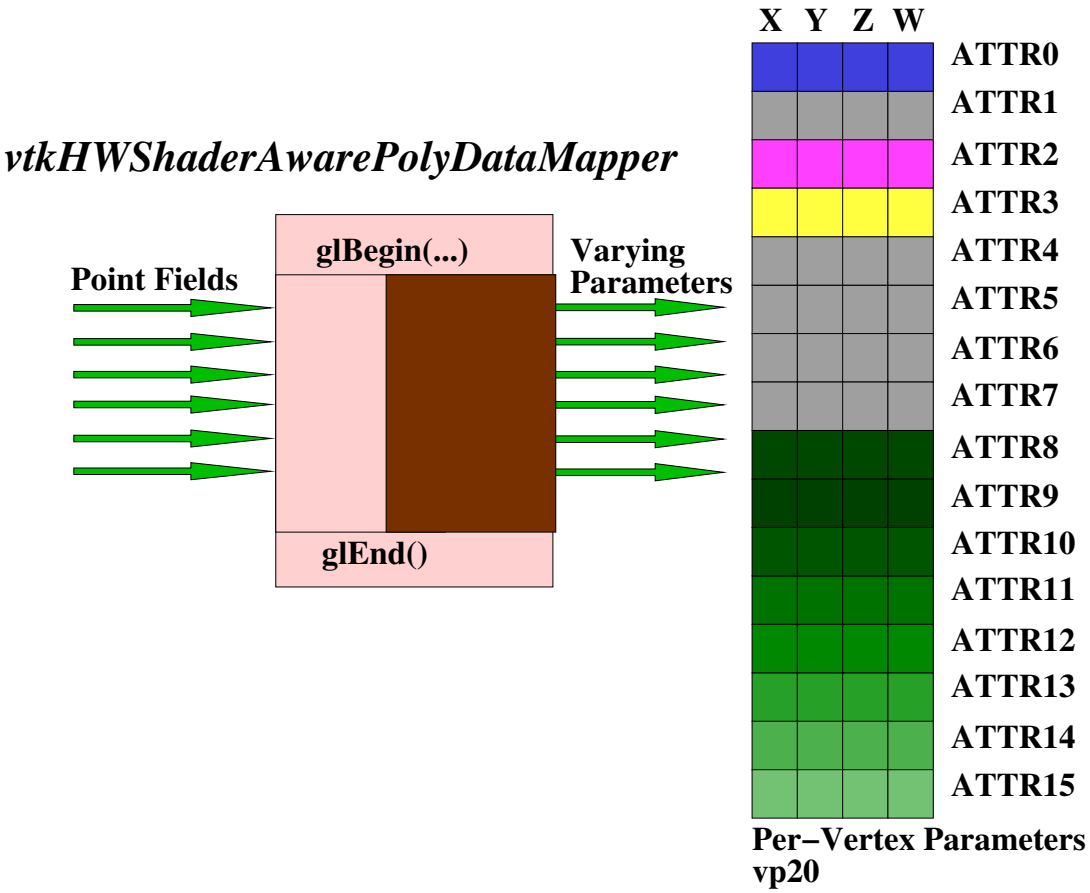

Figure 2: Data Flow Diagrm for Uniform Variables, Cg

3.1 vtkXMLMaterialParser: Access to a Material Definition

Access to the aforementioned XML material file will be provided by a new class, vtkXMLMaterialParser, which will parse the material file and provide accessors to the data contained therein. vtkProperty and vtkHWShader both hold references to a vtkXMLMaterialParser and use it to populate their members. To minimize disk access, any given vtkXMLMaterialParser may be assigned to multiple instances of vtkProperty. The result is that vtkActors that are defined to be of the same material will not only use the same material library but can reference the same instance of vtkXMLMaterialParser. This reduces the number of times a material file is accessed from the number of vtkActors in a scene to the number of unique materials used in the scene.

It should be noted that the materials library can be used to specify visual properties of vtkProperty even without the added functionality of hardware shader. As Brian Wylie would say, the two issues are somewhat orthogonal.

3.2 Extending vtkProperty: Assigning a Material Definition

Since the manner is which an object is rendered is essentially a visual property, it's logical to assign hardware shaders on a per-vtkActor basis. This can best be done by extending vtkProperty to reference a vtkXMLMaterialParser, which vtkProperty will use to populate it's members. Where the XML file defines new values for vtkProperty members, vtkProperty will override it's default value for those members with the values defined in the XML file.

While having the developer assign a specific material for each vtkActor is a good option for this prototype, it's worth investigating an object that manages mapping vtkActors to specific materials libraries.

It seems tempting to task vtkMapper with loading particular shader. But this is not in line with the design intent of VTK where vtkProperty sets the OpenGL rendering state and vtkMapper send graphics primitives to the hardware. The main assumption here is that hardware shaders should be considered part of the OpenGL state and are not part of set of graphics primitives. Just as each vtkProperty currently changes the OpenGL state to suite it's actor, it would also specify a specific hardware renderer as part of the OpenGL state. A descendent of vtkMapper would still be responsible for mapping graphics primitives to hardware. However, the current implementation of vtkOpenGLPolyDataMapper severely limits the ability to use hardware shaders.

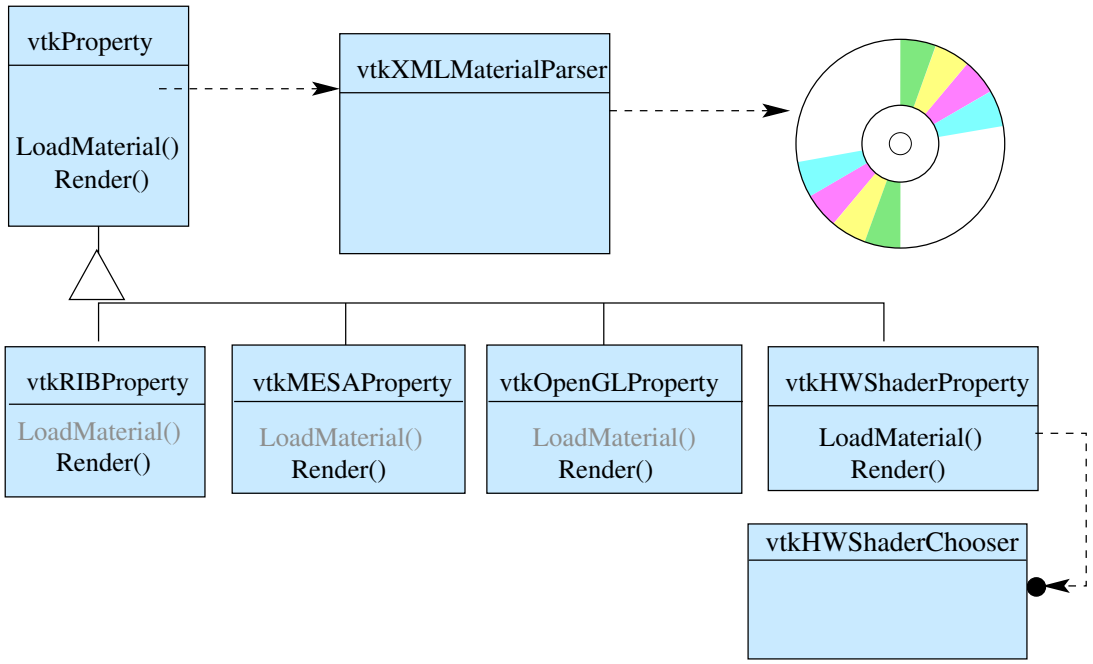

Figure 3: vtkProperty Class Diagram

3.3 vtkOpenGLPolyDataMapper: Sending Graphics Primitives to HW

HW shader programs have access to the 16 (as defined by the vp20 Cg profile) or so hardware registers associated with each vertex as defined in the OpenGL standard. Since these hardware registers map to the complete set of possible varying parameters for a given hardware shader, an application must be able to populate all registers in any combination to take full advantage of the hardware shaders.

The current implementation of vtkOpenGLPolyDataMapper only allows a VTK application to populate 4 of these registers, ATTR0, ATTR2, ATTR3, and one texture coordinate, ATTR8. These correspond to the OpenGL aliases 'Position', 'Normal', 'Color0', and 'TexCoord0' respectively and are set by calls to glVertex*, glNormal*, glColor*, and glTexCoord*. Extending vtkOpenGLPolyDataMapper to cover all possible comibinations of the 16 hardware registers is nearly impossible since it's use of macros require each possible case to be enumerated. Figure 4 shows the hardware registers provided for varying and uniform parameters for vertex and fragment programs. An application will typically set the varying vertex parameters and the vertex shader will typicall set the varying shader parameters. The 'grey' vertex

registers are those that VTK cannot access through vtkOpenGLPolyDataMapper, or any other class.

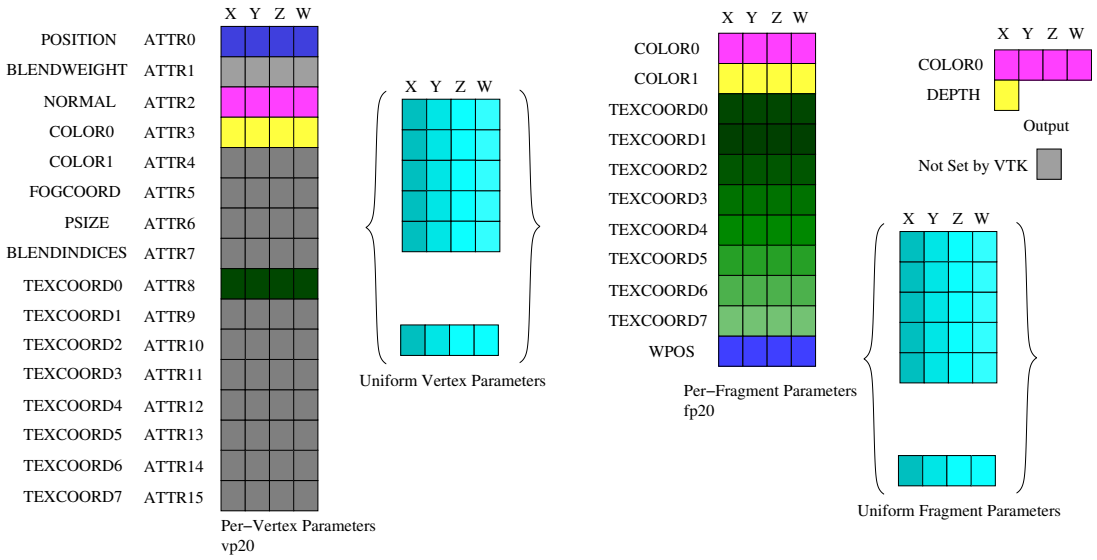

Figure 4: vtkOpenGLPolyData HW Mappings

Sandia has prototyped a direct extension of vtkOpenGLPolyDataMapper that allows use of POSITION, NORMAL, COLOR0 and all available TEXCOORD* hardware registers but limits these to combinations that monotonically increase in register id from ATTR8 to ATTRN, where N in not greater than 15. This exploratory class enumerates this limited set of ATTR* combinations but still does not have the flexibility required to fully implement all hardware shaders.

A viable solution is the proposed vtkPainter classes detailed in Ken Moreland's "vtkPainter: An Improved Poly Data Mapper" proposal. These classes leverage the latest versions of OpenGL and some OpenGL extensions to allow a vtkMapper to populate in any combination the vertex program hardware registers of a video card.

The only step required by the application is to flag vtk cell or point fields that are to be sent to hardware and to specfify what registers they should occupy. These fields can be specified through the shader definition in the material file.

3.4 vtkHWShader: Rendering With a HW Shader

The mechanics of rendering a vtkActor with a hardware shader are encapsulated in descendants of vtkHWShader. Figure 5 depicts a class diagram that highlights vtkHWShader and it's descendants.

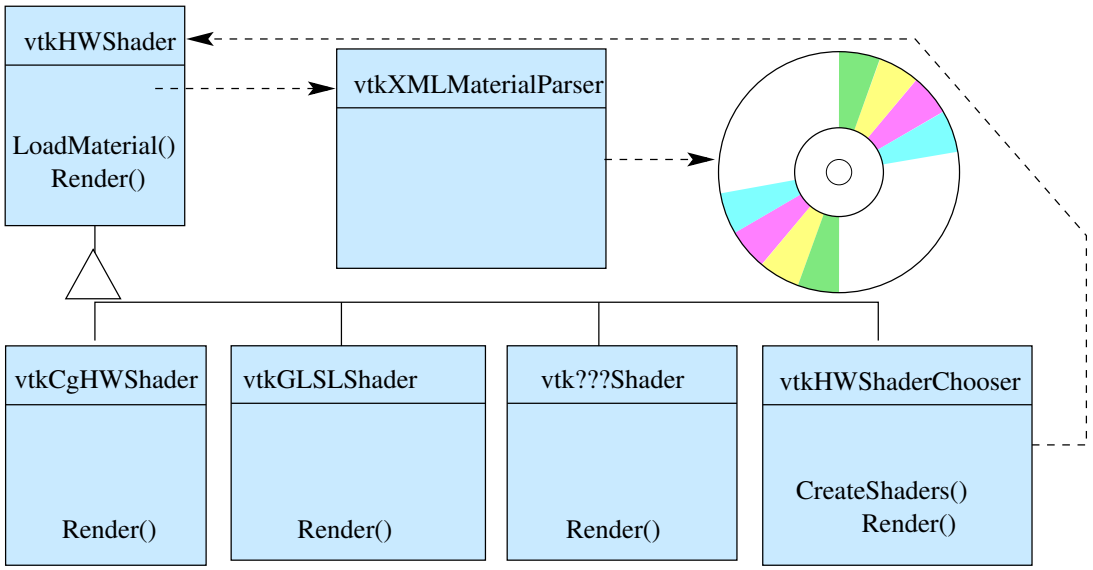

Figure 5: vtkHWShaders

vtkHWShader is a base class that provides an interface for rendering with a hardware shader. It holds a pointer to a vtkXMLMaterialParser (passed to it by vtkHWShaderProperty) from which it obtains the hardware shader and parameter specifications to be use when it is rendered.

Two concrete implementations of vtkHWShader, vtkHWCgShader and vtkHWGLSLShader, will render with NVidia's Cg library and OpenGL2.0. Other concrete implementations of vtkHWShader will be developed as users and developer request and or contribute them.

vtkHWShaderChooser is a proxy class [GoF] that delegates the mechanics of rendering with a hardware shader to the correct subclass of vtkHWShader. The main advantage of this approach is that it delays until runtime the instantiation of a particular concrete implementation of vtkHWShader. This is important since we can't be sure until the material file has been read which shader library will be required. It's also possible that a machine could have multiple hardware shader libraries installed, any of which could be used to render given the correct shader. Considering all this, VTK's standard 'New' methods will not have enough information to select the proper hardware library.

3.5 vtkHWShaderProperty: HW Shader Management

vtkHWShaderProperty is a descendant of vtkProperty that holds a reference to two vtkHWShaderChoosers. The first manages any vertex shaders defined in the material file while the second manages any fragment shaders defined in the material file.

Figure 6 depicts a typical object diagram to illustrate how a vtkHWShaderProperty would render a vtkActor under this proposed architecture. Each object loads it's respective definitions from the XML material file when a call is made to 'LoadMaterial()'. Rendering is delegated in the same manner, with the concrete implementation of vtkHWShader sending it's shader to the graphics hardware.

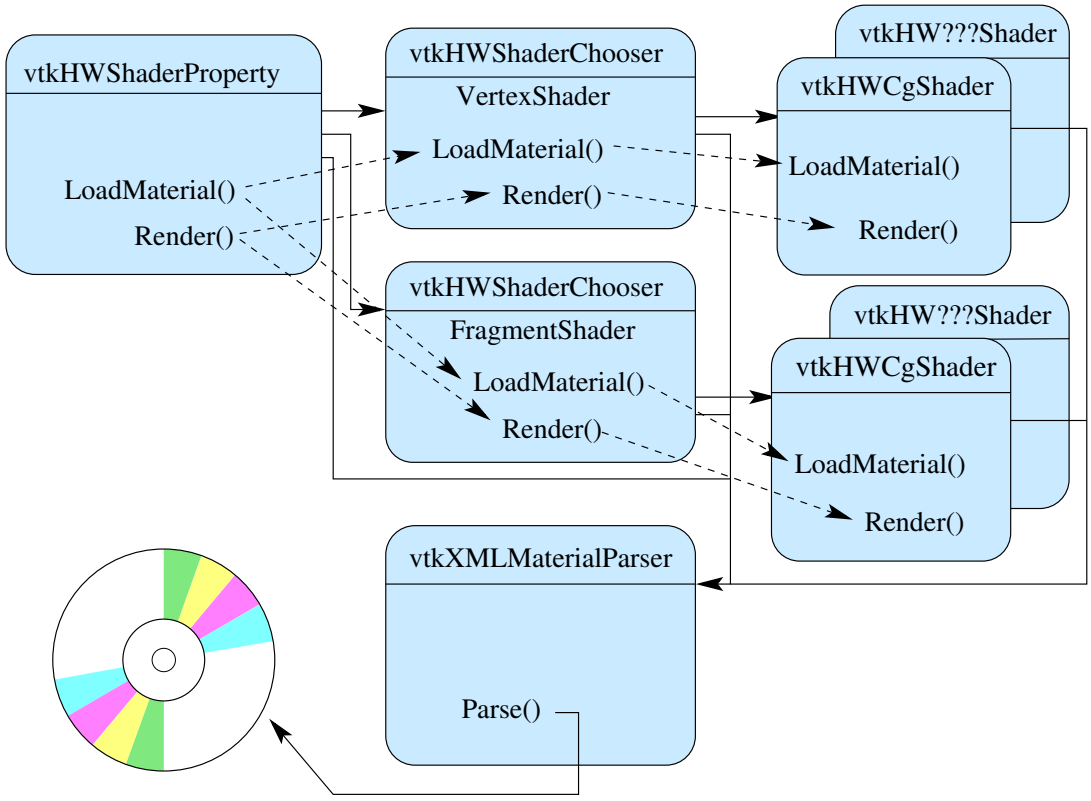

Figure 6: Shader Object Diagram

4 ParaView

Once the functionality described above is implemented in VTK, the next logical step is to add the same capability to ParaView. The concept of a materials library is a

logical extension to the ParaView interface. It would simply be added as a option in the 'Display' tab of a pipeline actor. Just as there is an option for 'Actor Color', 'Actor Material' can be added as another choice. Upon selecting this choice the user would be prompted to select from a set of pre-defined material libraries. The mapping of material file to objects could also be done with a configuration file.

5 Ongoing Issues

A few outstanding issues remain before this proposal is complete.

5.1 Mapping Textures to HW Shaders

While a strategy for mapping textures to hardware shaders in VTK is goal of this proposal, no work has been done in this area.

5.2 XML Schema

An initial XML schema is being developed to represent visual properties stored in vtkProperty and vertex and fragment shaders and their parameters. It is proposed that these be stored in one file and used together to define a material familiar to end-users.

5.3 Acknowledgements

The proposal presented here benefited from discussions with Ken Moreland, Andy Wilson, and Dave Thompson. The initial vtkXMLMaterialParser was prototyped by David Karelitz. LaTeX formatting used to create this document was taken from a model provided by David Thompson.

This work was done at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.