

Maya Hardware Shader Interface

White Paper

This document is freely available and does not require a non-disclosure. Readers should be conversant with concepts of 3D computer graphics, 3D modeling/animation packages, and C++ object oriented programming.

1.0 INTRODUCTION.....	3
2.0 BASIC CONCEPTS	3
2.1 Maya Refresh Mechanism.....	3
2.2 Draw Requests.....	4
3.0 THE MAYA 4.0-5.0 HARDWARE SHADER INTERFACE.....	5
3.1 MPxHwShaderNode.....	5
3.2 Hardware Rendering.....	5
3.2.1 Hardware Display Interface	7
3.2.2 Specifying Attribute Data Requirements	12
3.2.4 Drawing Geometry	14
3.2.3 Handling Transparent Objects	15
4.0 THE MAYA 6.0 HARDWARE SHADER INTERFACE.....	15
4.1 The glBind Method.....	16
4.2 The glGeometry Method.....	17
4.3 The glUnbind Method.....	19
5.0 HARDWARE SHADER INTERFACE CHANGES IN MAYA 6.5	20
5.1 Dirty Masks	20
5.2 Current Path	20
5.3 Images for the Texture Editor	21
6.0 HARDWARE SHADER INTERFACE CHANGES IN MAYA 8.0 AND 8.5	21

- 6.1 Batch shading of shapes with the same material (Maya 8.0) 21**
- 6.2 More control over texture coordinates and transparency (Maya 8.5) 22**
- 6.3 More control over the Maya viewport 23**
 - 6.3.1 MViewportRenderer 23
 - 6.3.2 MDrawTraversal 24
 - 6.3.3 MGeometryManager 24
 - 6.3.4 MGeometryRequirements 24
 - 6.3.5 MGeometry 24
 - 6.3.6 MGeometryPrimitive 24
 - 6.3.7 MRenderTarget 24
 - 6.3.8 MRenderingInfo 25
 - 6.3.9 MHwTextureManager 25
 - 6.3.A MGLFunctionTable 25
- 7.0 SETUP AND ACCESS TO EFFECTS DATA 25**
- 7.1 Geometric Coordinate Spaces 25**
- 7.2 Access to Camera and Light Information 26**
- 7.3 Access to Textures 28**
- 8.0 OTHER CONSIDERATIONS 29**
- 8.1 Software Rendering 29**
- 8.2 Artist Considerations 30**
- 8.3 OpenGL Details 32**
 - 8.3.1 Binding OpenGL Extensions 32
- 9.0 SUMMARY 34**

1.0 Introduction

In this document, we discuss the hardware shader plug-in interface available in the OpenMaya API to the Maya® software product.

The refresh architecture of Maya allows users to view a fixed set of visual parameters (for example, Lambert or Phong shader color) in its interactive viewing panels. A Maya hardware shader plug-in allows a programmer to create customized visual effects that seamlessly integrate with the Maya environment. It allows:

1. Full control over the display pipeline for programmers.
2. Full control for programmers to describe the artistic parameters via an interface familiar to artists.

The most important advantage of the hardware shader plug-in is the ability to visualize precisely what you want within the Maya framework, with a high degree of interactivity. This can satisfy requirements such as pre-visualization for a wide range of market segments including film, television, video game, or web production.

A major advantage is the ability to utilize the current set of programmable hardware "transform and lighting" solutions which are available at the commodity and professional graphics board level.

As an overview, we will start with a brief explanation of the Maya refresh architecture in various versions of Maya, and then go on to explain how this interface hooks into the architecture. Currently, the Maya API supports multiple interfaces for hardware shading. The original interface was preserved to maintain compile compatibility with existing hardware shader plug-ins. The developer may choose whether he/she wishes to upgrade to the latest interface to take advantage of its features. We will follow the interface descriptions with a brief example demonstrating its implementation and usage.

Hardware shading is an area of our product that is constantly being updated and improved. This should be taken into account when writing a custom shader so that future upgrades do not require a great deal of code rewrite. For example, you may wish to isolate the plug-in's OpenGL® specific draw code into a method so that it can be called from different interfaces. This document is applicable up to the **Maya 8.5** version of OpenMaya API.

2.0 Basic Concepts

2.1 Maya Refresh Mechanism

The basis of the Maya refresh architecture is the notion of "*parceling*". Parceling means that when a viewing panel (*view* for short) needs to be refreshed, a Maya scene traversal is performed with the appropriate culling and visibility testing. During this traversal, if an object is deemed to be visible from the camera and needs to be drawn, then the object will be asked to determine what it needs to draw and the object will pass back one or more autonomous "parcels" to be drawn. The following algorithm is used for drawing shapes or geometry:

1. Maya visits each DAG node (parent of a shape) collecting draw requests.
2. Draw requests are sorted into opaque and transparent queues.
3. Maya performs the opaque draws (unsorted).
4. Maya depth sorts the transparent queue and performs the transparent draws.

When drawing each shape, Maya collects the geometry to be drawn (vertices, normals, faces) and passes this information to the shader(s) assigned to the shape for rendering into the view. In effect, the shader is entirely responsible for rendering each shape. The hardware shader API allows you to define a new shader node that receives geometry from Maya and is then free to render it to the viewport in any way it chooses.

2.2 Draw Requests

The original hardware shader interface (Maya 4.0 to Maya 5.0) made heavy use of an API object called MDrawRequest. An **MDrawRequest** contains some basic information such as which part of an object is to be drawn, the object space matrix, shaded mode material attributes and/or wire frame display attributes, and the display state of the view to draw into.

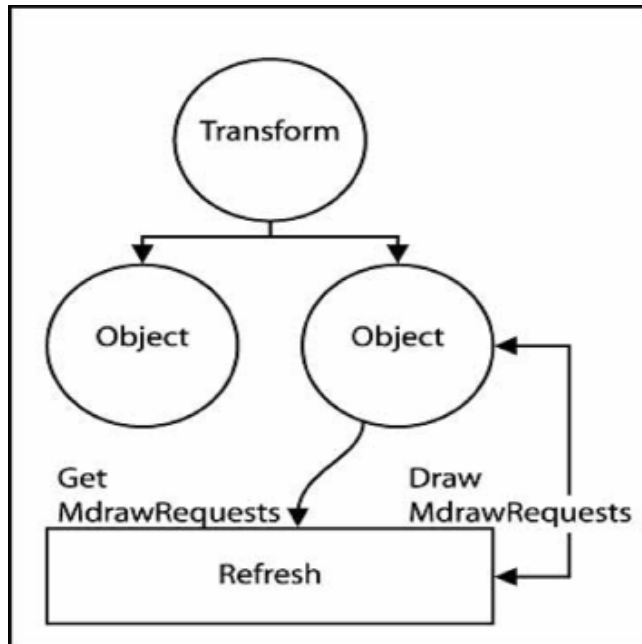
Maya internal object types currently generate a variety of **MDrawRequests** in which custom data is attached as **MDrawData**. The most important piece of data that is supplied on a per object basis is the geometry to draw.

Users of Maya are already accustomed to the notion of a shader, and the ability to define either simple or complicated shading networks by connecting shading nodes via the Maya interactive interface. The HyperShade or MultiLister interfaces are the access points for this interaction.

Typically, these shaders have a number of attributes that can be visualized via software rendering. For 3D interactive view ports, a graphics hardware evaluation is made, and the derived interpretation is passed on as parameters in an **MDrawRequest**.

When the geometric and material attributes have been collected by the refresh mechanism, the object is asked to draw the **MDrawRequest** appropriately in the desired view.

The following flow chart illustrates the creation and drawing of **MDrawRequests**. It shows the act of getting and drawing **MDrawRequests** for one object in the scene hierarchy. The handling of the **MDrawRequest** by the Maya "refresh" mechanism is left as a black box, as plug-in writers do not need to concern themselves with its details.



It should be noted that the current hardware shader interface (Maya 6.0 to Maya 6.5) does not make use of **MDrawRequests**. This interface is provided for integrating plug-in hardware shaders into the new High Quality Interactive viewport and the Hardware Renderer. **MDrawRequests** are not needed for this type of drawing since it is independent of the Maya modeling views.

3.0 The Maya 4.0-5.0 Hardware Shader Interface

A number of access points are provided via the Maya hardware shader mechanism. They provide:

1. The ability to define a custom plug-in material node (**MPxHwShaderNode**).
2. The ability for the custom material node to intercept **MDrawRequests**, and take control of the hardware drawing.
3. The ability to implement the software rendering interpretation of the custom material node.
4. The ability for artists to manipulate the custom material nodes like internal or "normal" Maya shader nodes.

Let's first look at the class and structure of an **MPxHwShaderNode** plug-in node.

3.1 MPxHwShaderNode

This node (C++ class) can be considered as the basis on which plug-in writers can derive their own plug-in hardware shader nodes. If certain methods on this node are not implemented, the default implementation will be used. By default, the shading will compute black (0,0,0) shading values for software and hardware rendering.

The two basic shading requirements a plug-in writer needs to handle are:

1. Implementation of shading logic for hardware rendering
2. Implementation of shading logic for software rendering

Either one or the other of these aspects can be implemented in the derived node. There is no required interdependence between the two.

3.2 Hardware Rendering

The hardware rendering interface controls how the shaded shapes appear in the viewport. It does not affect the software renderer, the UV Texture Editor or swatch rendering. It can be thought of as the computation of the hardware rendering effect (similar to the software rendering computation) using an explicit interface to the geometry of an object. This interface hooks directly into the Maya hardware refresh mechanism by allowing for:

1. Geometry assigned to the hardware shader to be passed to it.
2. Hardware shader nodes to have complete control of display state.

When the material evaluation (described in **Section 2.2**) is performed by an object, a check is made internally to see if either:

1. A hardware shader (node) has been assigned to an object.
2. A hardware shader (node) has been connected as the input texture to a

"standard" shader node. A "standard" shader node is one that is supplied with Maya (Blinn, Phong, PhongE, Lambert etc).

If one of these conditions is met, then the refresh mechanism no longer attempts to perform its default evaluation. Instead, the refresh mechanism acknowledges that no software-to-hardware interpretation is required since the plug-in hardware shader node will determine the display setup. The existence of a plug-in hardware shader is kept as part of the **MDrawData** for the **MDrawRequest**.

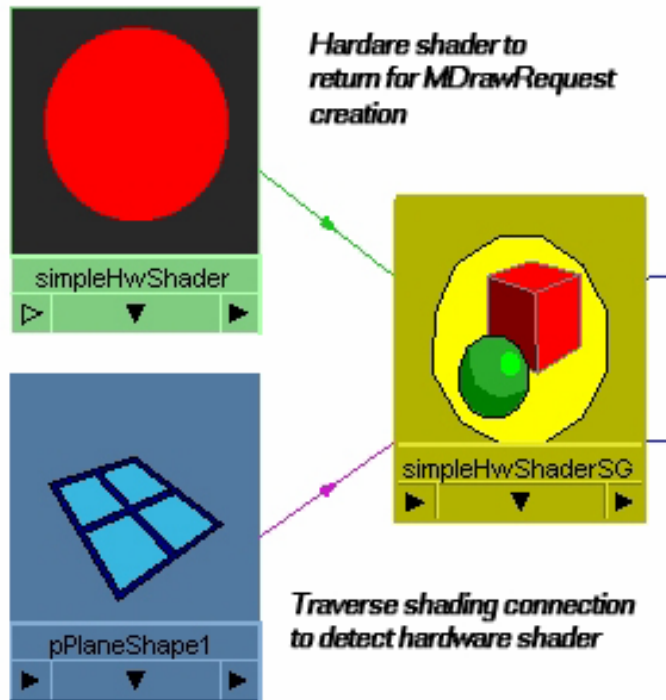
At this stage, the geometry used for drawing is *not* evaluated, except if there is component level shader assignment for determining the parts of the object that belong to the shader. The indicator of parts membership is also kept as part of the **MDrawData** for the **MDrawRequest**.

In general, one **MDrawRequest** will be created for each material assignment. Thus, if an object has N hardware shaders assigned to different parts of the object, then N **MDrawRequests** will be created, one for each unique shader.

The rough pseudo logic for this is as follows:

```
getDrawRequests( Mobject &object, M3dView &view )
{
    shaderList = getShadersConnectedToObject( object );
    for each ( shader s in the shaderList )
    {
        shaderProperties = getMaterialProperties( s );
        triangles = getTrianglesAssociatedWith( s );
        MDrawRequest dr = createDrawRequest();
        setGeometryOnRequest( dr, triangle );
        if (shaderTypeOf(shaderProperties) == hardware shader)
        {
            markAsUsingHardwareShader( dr );
        }
    }
}
```

The following is an example hardware shader called "simpleHwShader" which has been assigned to an object called "pPlaneShape1" within Maya. The shader has been assigned directly to the object. pPlaneShape1 will detect this assignment and build an **MDrawRequest** which keeps track of the fact that there is a hardware shader present. Only one **MDrawRequest** will be created since the whole of the object is assigned to a single shader.



3.2.1 Hardware Display Interface

When the refresh mechanism in Maya encounters an **MDrawRequest** which uses a hardware shader, the interface to the hardware shader node is used. The interface is described by three methods on the shader node:

1. *bind()*
2. *geometry()*
3. *unbind()*

Each hardware shader node is responsible for setting up the display state during *bind()*, drawing the geometry during the *geometry()* call, and cleaning up the display state during *unbind()*. The default implementation of these methods does not perform any action. With this plug-in shader interface, Maya uses the following algorithm to shade geometry:

```

For every mesh shape
    bind()
    geometry()
    unbind()
    
```

The pseudo logic for the usage of this API is as follows:

```

handleDrawRequests( Mobject &object, MDrawRequest &dr, M3dView &view )
{
    if ( isUsingHardwareShader( dr ) )
    {
        MPxHwShaderNode hardwareShader = dr.hardwareShader();
        // Let hardware shader set up display state
    }
}
    
```

```

hardwareShader.bind( ... );

// Code to fill in geometric information to send
// to hardware shader ...

// Draw using hardware shader
hardwareShader.geometry( ... );

// Let hardware shader clean up display state
hardwareShader.unbind();
}
else
{
    normalDrawingOf( dr );
}
}

```

3.2.1.1 The Bind() and Unbind() Methods

The `bind()` and `unbind()` methods are straightforward, and should be used to prepare for the drawing which occurs during the `geometry()` method. The interfaces are specified as follows:

```

virtual MStatus bind( const MDrawRequest& request, M3dView& view );
virtual MStatus unbind( const MDrawRequest& request, M3dView& view );

```

The arguments are, respectively, an **MDrawRequest** and the 3D view (**M3dView**) that is being drawn to. If there are multiple views (modeling panels) that are currently visible in Maya, then there is one **MDrawRequest** per view to refresh. The methods `bind()` and `unbind()` will be called for each draw request.

Internally, the Maya refresh mechanism keeps track of the display state in a layer above the display API level (*OpenGL*). During normal evaluation, Maya will set and restore the state as needed between **MDrawRequests**. For example, the display state set up is performed to accommodate one **MDrawRequest** that draws the shaded surface of an object, followed by another **MDrawRequest** that will draw the wire frame version of the same surface.

Maya does not know which display state has been modified by your shader, so it is the responsibility of the plug-in shader node to save and restore the display state during `bind()` and `unbind()` respectively. This can be done in two ways: via the **M3dView** which is passed in, and/or by making direct calls at the display API level (*OpenGL*).

The following is a very simple example that will set up an object to draw shaded with a blue material if the object is selected, and red otherwise.

First of all, note the usage of the `beginGL()` and `endGL()` methods on **M3dView**. All *OpenGL* calls made within the plug-in should be bracketed by these two method calls. This sets the correct display context for drawing, and also indicates to Maya that the *OpenGL* state is no longer being kept track of by the Maya refresh mechanism.

Secondly, note that the *GL* state is *pushed* in `bind` and *restored* in `unbind`. Failure to correctly restore the display state to its original value on leaving `unbind` can result in the corruption or crashing of Maya. In the example below, we have pushed all *OpenGL* attributes to save state. However, your shader will be more efficient if it only pushes and pops the state it actually modifies.

```

/* virtual */
MStatus simpleHwShader::bind(const MDrawRequest& request,
                             M3dView& view)
{

```

```

// Check to see if this draw request says to draw the geometry
// active/selected
M3dView::DisplayStyle displayStatus = request.displayStatus();
bool objectIsActive = ( displayStatus == M3dView::kActive );
view.beginGL(); // Tell Maya not to keep track of display state

// Push attributes
glPushAttrib( GL_ALL_ATTRIB_BITS );
glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);

// Override ambient and diffuse material colors
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);

if (objectIsActive)
{
    glColor4f( 0, 0, 1, 1 ); // Set the color to be blue
}
else
{
    glColor4f( 1, 0, 0, 1 ); // Set the color to be red
}
view.endGL(); // Tell Maya to keep track of display state
return MS::kSuccess;
}

/* virtual */
MStatus simpleHwShader::unbind(const MDrawRequest& request,
                               M3dView& view)
{
    view.beginGL();
    glPopClientAttrib();
    glPopAttrib();
    view.endGL();
    return MS::kSuccess;
}

```

For performance reasons and for proper node evaluation, values which are used to set up the display state or draw should be calculated in the `compute()` method for the object. For example, you may wish to set the color or the shader in the `compute()` method. The `bind()` and `unbind()` methods are called for each refresh request so that it is more efficient to have this precalculated in `compute()`. In the case above, the check for the active display state is appropriate as it is view port and draw request dependent.

3.2.1.2 The geometry() Method

The `geometry()` method performs the actual drawing. This method is called between the `bind()` and `unbind()` calls. The display state is guaranteed not to change between a `bind()` and `geometry()` call.

Attributes for the geometry are passed from the object to the `geometry()` method of the hardware shader node for display. The interface for this method is as follows:

```

virtual MStatus geometry( const MDrawRequest& request,
                          M3dView& view,
                          int prim,
                          unsigned int writable,

```

```

int indexCount,
const unsigned int * indexArray,
int vertexCount,
const int * vertexIDs,
const float * vertexArray,
int normalCount,
const float ** normalArrays,
int colorCount,
const float ** colorArrays,
int texCoordCount,
const float ** texCoordArrays);

```

As with the `bind()` and `unbind()` methods, an **MDrawRequest** and an **M3dView** are passed as arguments. All OpenGL calls must be bracketed with **M3dView** `beginGL()` and `endGL()` calls.

In addition, the following important geometric information is provided:

1. *prim* : The type of display primitive. This corresponds to the OpenGL primitive type. Currently only `GL_TRIANGLES` is supported, and will be the only value passed for this parameter. This may change in the future so plug-in writers should check the primitive type passed via this method.

2. *writable* : Depending on the requirements of the geometry, it is possible that the attribute data passed is read-only or writable. Thus, a bit mask is passed to the geometry that indicates, for each attribute, which attribute array can be modified. For the current revision, all data can be overwritten. The bit mask flags to check with are specified via the *Writable* enum in **MPxHwShaderNode**.

3. *indexCount*, *indexArray* : By default, this interface provides a single indexing structure to one or more sets of arrays of attribute data. *indexArray* contains an array of such indices, and *indexCount* is the size of this array.

This level of indirection has been provided to allow for geometric pipeline optimization using OpenGL calls such as `glDrawElements()`. Note, also, that this is very similar to how Maya itself stores its internal geometry attributes, and thus allows plug-in writers to take advantage of any internal representation performance enhancements immediately and *seamlessly*.

4. *vertexCount* : The number of vertices. This is the attribute count for position, normal, color, and texture coordinate data.

5. *vertexIDs* : The vertex component IDs that correspond to the *vertexArray* parameter. This information allows for the lookup of per vertex information such as blend data. (Added in Maya 5.0.)

6. *vertexArray*: This is a list of homogenous **object space** positions (x,y,z).

The positions are sent in object space, since the Maya refresh mechanism has already set the model view transformation *before* making any calls to this method. Maya objects keep positions in object space, and, for performance reasons, this interface maintains this space.

The user has access to the object-to-world matrix via the **MDagPath** available in the **MDrawRequest** object passed in via this method. This is the correct path for displaying the current instance of object (if the object was instanced).

7. *normalArrays*, *normalCount* : *normalArrays* contains the normalized normal vector data for the object.

normalCount represents the number of sets of "normals". Surface tangent and surface bi-normal information (the cross-product of the normal and tangent) may

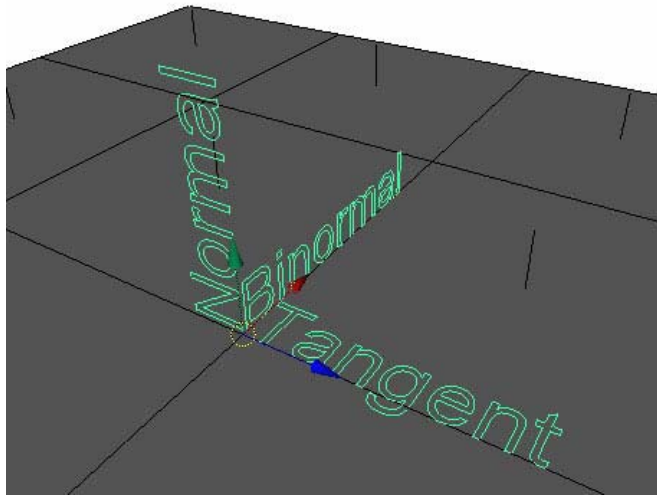
be included if the *normalCount* is greater than one. The first normal is the surface normal.

The tangents and bi-normals are computed internally using the texture coordinate data on the object. If there is no texture coordinate information, then no tangents nor binormals can be passed as data. The texture coordinates from the *first* UV set are used for the tangent and bi-normal computation.

This set of vectors defines the **tangent space** orthonormal basis either per vertex or per vertex-face pair. That is, the basis may be shared at a vertex if the 3D topology and UV topology are both shared. Maya surface display options can be used to view either of these geometric or UV discontinuities or "borders".

The first array of data in *normalArrays* contains the normals (i.e. *normalArrays[0]*). The second is the optional tangents and the third the optional bi-normals. Since surface objects in Maya always compute normals, the value of *normalCount* is, by default, 1.

Below is a visual example of one such basis on a surface of quadrilaterals in a grid. In this example, the u direction in UV-space is the tangent direction, and the v direction is the direction of the binormal. In general, the tangent lies along the u axis.

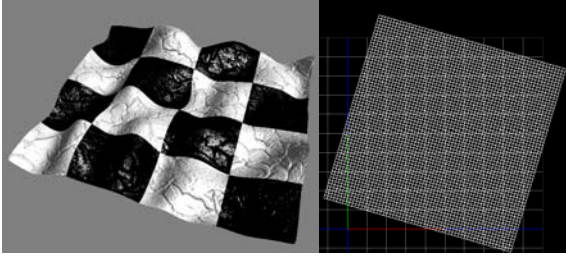


8. *colorArrays*, *colorCount* : *colorArrays* contains the RGBA color data for the object. *colorCount* indicates the number of sets of colors there are being used. This may be a subset of the number of color sets on an object. In Maya, polygonal objects can have multiple set of colors. By default, *colorCount* is 0.
9. *texCoordsArray*, *texCoordCount* : *texCoordsArray* contains the coordinate (u,v) data for the object. *texCoordCount* indicates the number of sets of texture coordinates that are being used. In Maya, polygonal objects can have 0 or more sets of texture coordinates. By default, *texCoordCount* is 0. The texture coordinates passed to your shader are controlled by some additional virtual

methods that will be described shortly.

The following snapshot from Maya shows a simple polygon grid. Shown on the left is the object in 3D, and on the right, the first set of texture coordinates. In this case:

1. The *texCoordsArray* would contain one set of texture coordinate values, used for both the color decal and the bump map.
2. The shader does not use color-per-vertex, so the *colorCount* would be 0, and the *colorArrays* argument would be empty.
3. The *normalCount* would be 3, since the bump-mapping effect requires tangent-space information. There would be three arrays of "normals" in the *normalArrays* argument (containing normal, tangent, and bi-normal information).
4. The tangent space computation would use the texture coordinates shown on the right (snapshot of the *UV Texture Editor* in Maya).



3.2.2 Specifying Attribute Data Requirements

MPxHwShaderNode contains additional interfaces that allow the plug-in writer to control exactly which data is required to achieve the shading effect. This directly affects the data that gets passed back through the *geometry()* call.

1. *normalsPerVertex()* : This method can be overridden in the derived hardware shader implementation to represent the number of normal sets that are required for display. By default, this is one set.
 - If no normals are required, a value of zero can be returned from this method.
 - If only the normals are required, a value of one can be returned from this method.
 - If tangents are required, a value of two can be returned from this method.
 - If tangents and bi-normals are required, a value of three can be returned from this method.
2. *colorsPerVertex()* : This method can be overridden in the derived hardware shader implementation to represent the number of sets of colors that are required for display. By default, this is 0.
 - i. By default, when an object is first created in Maya, it contains no color sets. The *geometry()* call will return the minimum of the number of existing color sets, and the number of color sets requested. The *colorCount* argument in the *geometry()* call should be checked to see exactly how many color sets have been returned.

3. `texCoordsPerVertex()` : This method can be overridden in the derived hardware shader implementation to represent the number of sets of texture coordinates (uvs) that are required for display. By default, no texture coordinates are provided.
 - i. The number of UV sets is restricted by how many the artist creates on the given object. If there are fewer texture coordinate sets created than requested by the hardware shader, the `geometry()` call will return the minimum number. The `texCoordCount` argument in the `geometry()` call should be checked to see exactly how many UV sets have been returned. A limitation of this method is that if a UV set is missing, you will not know which one it is. As a result, the next method is provided.
4. `getTexCoordSetNames()` : This method can be overridden in the derived hardware shader implementation to represent, by name, the UV sets that are requested. If the UV set name does not exist for the object to be drawn, then the `geometry()` call will return a subset of those UV sets requested. The `texCoordCount` argument in the `geometry()` call should be checked to see exactly how many UV sets have been returned.

Thus, the pseudo code previously presented on handling `MDrawRequests` would now resemble the following:

```

handleDrawRequests( Mobject &object, MdrawRequest &dr, M3dView &view )
{
    if (isUsingHardwareShader( dr ))
    {
        MPxHwShaderNode hardwareShader = dr.hardwareShader();
        // Let hardware shader set up display state
        hardwareShader.bind( ... );
        // Get object space positions
        positionArray = positions(object);
        // Get object space normals
        normals normalArray[3]

        if (hardwareShader.normalsPerVertex() > 0)
        {
            normalArray[0] = computeNormals(object);
            // Get object space tangents
            if (hardwareShader.normalsPerVertex() > 1)
                normalArray[1] = computeTangents( object );
            // Get object space bi-normals
            if (hardwareShader.normalsPerVertex() > 2)
                normalArray[2] = computeBiNormals( object );
        }

        // Get colors
        if (hardwareShader.colorsPerVertex() > 0)
            colorArray = getColors( object );

        // Get texture coordinates
        MStringArray names;
        getTexCoordSetNames(names);
        for (i=0; i< hardwareShader.texCoordsPerVertex(); i++)
            texCoordArray[i] = getTextureCoordinates( object );

        // Draw using hardware shader and geometric information
    }
}

```

```

        hardwareShader.geometry( positionArray, normalArray,
                                colorArray, texCoordArray );

        // Let hardware shader clean up display state
        hardwareShader.unbind();
    }
    else
    {
        normalDrawingOf( dr );
    }
}

```

If, for instance, the method `normalsPerVertex()` was overridden with a value of 3, then both tangents and bi-normals would be computed and returned via the geometry call.

3.2.4 Drawing Geometry

What the plug-in chooses to do with the data passed to it via the `geometry()` call can vary widely. In general, no state changes should be made within this method, but otherwise it is up to the plug-in writer to decide how to best use the geometric data for drawing.

The following is an example implementation of the `geometry()` call. It draws and shades all of the geometry passed in using the positions and possibly the normals, tangents and binormals (if passed in). For this shader, the `normalsPerVertex()` method is overridden to return a value of 3, and the `texCoordsPerVertex()` method is overridden to return a value of 1.

As with the `bind()` and `unbind()` methods, a call to `beginGL()` and `endGL()` on **M3dView** is used to bracket any OpenGL calls made.

```

/* virtual */ int simpleHwShader::normalsPerVertex()
{ return 3; }

/* virtual */ int simpleHwShader::texCoordsPerVertex()
{ return 1; }

/* virtual */
MStatus simpleHwShader::geometry(const MDrawRequest& request,
    M3dView& view,
    int prim,
    unsigned int writable,
    int indexCount,
    const unsigned int * indexArray,
    int vertexCount,
    const int * vertexIDs,
    const float * vertexArray,
    int normalCount,
    const float ** normalArrays,
    int colorCount,
    const float ** colorArrays,
    int texCoordCount,
    const float ** texCoordArrays)
{
    view.beginGL()

    // Set up positions to draw with
    glVertexPointer(3, GL_FLOAT, 0, vertexArray);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Set up the 2D texture coordinates in texture unit 0

```

```

if (texCoordCount)
{
    glActiveTextureARB(GL_TEXTURE0_ARB);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoordArrays[0]);
}

// Set up any normals to draw with in normalArrays[0]
// and send the tangent and binormal as u,v values
if (normalCount == 3)
{
    glNormalPointer(GL_FLOAT, 0, normalArrays[0]);
    glEnableClientState(GL_NORMAL_ARRAY);
    glActiveTextureARB (GL_TEXTURE1_ARB);
    glTexCoordPointer(3, GL_FLOAT, 0, normalArrays[1]);
    glActiveTextureARB (GL_TEXTURE1_ARB);
    glTexCoordPointer(3, GL_FLOAT, 0, normalArrays[2]);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glEnable(GL_TEXTURE_COORD_ARRAY);
}

// Draw the elements
glDrawElements(prim, indexCount, GL_UNSIGNED_INT, indexArray);
view.endGL();
return MS::kSuccess;
}

```

3.2.3 Handling Transparent Objects

The Maya refresh mechanism handles draw requests that display transparent or semi-transparent objects in a special way. Maya refresh handles the following cases:

1. The output transparency on a shader is non-opaque.
2. The output color is mapped to a four-channel texture which has non-opaque pixel values (not equal to 1.0).
3. The view port is in X-ray mode.
4. The object is displayed using color-per-vertex color, where one or more colors have non-opaque values.

In these cases, the refresh mechanism draws those objects or parts of objects last, after all of the other objects have been drawn. The draw handling logic of the `MDrawRequest` only needs to know that there will be two calls made to the handler:

1. One to draw with the front faces culled
2. One to draw with the back faces culled

Since Maya cannot know whether a plug-in hardware shader wants to draw something with transparency, the plug-in writer needs to tell the refresh mechanism. This can be accomplished by overriding the `MPxHwShaderNode` method `hasTransparency()`. By default, the value returned from the parent class method is `FALSE`.

Formatted: Bullets and Numbering

4.0 The Maya 6.0 Hardware Shader Interface

In addition to supporting the original hardware shader interface described in the previous section, Maya 6.0 supports a supplementary interface using the `glBind()`, `glGeometry()` and `glUnbind()` methods on **MPxHwShaderNode**. These new methods add support for hardware rendering with shadows for meshes and NURBs surfaces, but only in the hardware rendering and high quality interactive shading modes. Maya draws using this interface with the following algorithm:

```

For every shader
    glBind
For every shape (meshes/NURBs)
    For every pass
        glGeometry
For every shader
    glUnbind

```

These virtual methods on the **MPxHwShaderNode** class are implemented to provide the shading functionality you require for an object. In comparison to a software shader, these methods allow you to perform OpenGL operations within the context of Maya rather than having to update the `outColor` attribute of a software shader. You are free to choose which set of interfaces you want to implement based on the features of each. In addition, you can implement both interfaces in order to define the behavior in the normal viewport, high quality interactive shading and the hardware renderer. Or, you may choose to implement the `glBind/glGeometry/glUnbind` methods alone since, by default, these will be called by the `bind/geometry/unbind` methods. (In this case, NURBs and shadow information will not be shown in the viewport.) The next section describes how the methods of the interface are used.

4.1 The glBind Method

The `glBind` method is used for setting up state, for example, texture loading and caching, effect loading and caching and the processing of node attribute as required

```

void hwPhongShader::init_Phong_texture ( void )
{
    GLubyte * texture_data;

    // Always release the old texture id before getting a
    // new one.
    if ( phong_map_id != 0 )
        glDeleteTextures( 1, &phong_map_id );
    glGenTextures ( 1, &phong_map_id );

    glEnable ( GL_TEXTURE_CUBE_MAP_EXT );
    glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
    glPixelStorei ( GL_UNPACK_ROW_LENGTH, 0 );
    glBindTexture ( GL_TEXTURE_CUBE_MAP_EXT, phong_map_id );

    texture_data = new GLubyte [3*PHONG_TEXTURE_RES*PHONG_TEXTURE_RES];

    for ( int face=0 ; face<6 ; face++ )
    {
        int index = 0;
        for ( int j=0 ; j<PHONG_TEXTURE_RES ; j++ )
        {
            // -1 to 1
            double t = 2*double(j)/(PHONG_TEXTURE_RES - 1) - 1;

```

```

        for ( int i=0 ; i<PHONG_TEXTURE_RES ; i++ )
        {
            // -1 to 1
            double s = 2*double(i)/(PHONG_TEXTURE_RES - 1) - 1;
            double x, y, z;
            cubeToDir ( face, s, t, x, y, z );

            MFloatVector intensity = Phong ( z );

            texture_data[index++] = (int)(255*intensity[0]);
            texture_data[index++] = (int)(255*intensity[1]);
            texture_data[index++] = (int)(255*intensity[2]);
        }
    }

    glTexImage2D ( faceTarget[face], 0, GL_RGB,
                  PHONG_TEXTURE_RES, PHONG_TEXTURE_RES,
                  0, GL_RGB, GL_UNSIGNED_BYTE, texture_data );
}

glDisable ( GL_TEXTURE_CUBE_MAP_EXT );

delete [] texture_data;

// Make sure to mark attributes "clean".
mAttributesChanged = false;
}

MStatus hwPhongShader::glBind(const MDagPath&)
{
    if ( mAttributesChanged || (phong_map_id == 0) )
    {
        init_Phong_texture ();
    }

    return MS::kSuccess;
}

```

4.2 The glGeometry Method

The `glGeometry()` method is core to the Maya 6.0 hardware shader interface. It is implemented to do all of the drawing. The first parameter passed to this method is the path to the object that is being drawn. Other information passed to the method includes vertices, normals and color information.

```

MStatus hwPhongShader::glGeometry(const MDagPath & path,
                                  int prim,
                                  unsigned int writable,
                                  int indexCount,
                                  const unsigned int * indexArray,
                                  int vertexCount,
                                  const int * vertexIDs,
                                  const float * vertexArray,
                                  int normalCount,

```

```

        const float ** normalArrays,
        int colorCount,
        const float ** colorArrays,
        int texCoordCount,
        const float ** texCoordArrays)
    {
        return draw( prim, writable, indexCount,
                    indexArray, vertexCount,
                    vertexIDs, vertexArray, normalCount,
                    normalArrays, colorCount,
                    colorArrays, texCoordCount, texCoordArrays);
    }

```

```

MStatus hwPhongShader::draw(int prim,
                            unsigned int writable,
                            int indexCount,
                            const unsigned int * indexArray,
                            int vertexCount,
                            const int * vertexIDs,
                            const float * vertexArray,
                            int normalCount,
                            const float ** normalArrays,
                            int colorCount,
                            const float ** colorArrays,
                            int texCoordCount,
                            const float ** texCoordArrays)
{
    if ( prim != GL_TRIANGLES && prim != GL_TRIANGLE_STRIP) {
        return MS::kFailure;
    }

    {
        glPushAttrib ( GL_ENABLE_BIT );

        glDisable ( GL_LIGHTING );
        glDisable ( GL_TEXTURE_1D );
        glDisable ( GL_TEXTURE_2D );

        // Setup cube map generation
        glEnable ( GL_TEXTURE_CUBE_MAP_EXT );
        glBindTexture ( GL_TEXTURE_CUBE_MAP_EXT, phong_map_id );
        glEnable ( GL_TEXTURE_GEN_S );
        glEnable ( GL_TEXTURE_GEN_T );
        glEnable ( GL_TEXTURE_GEN_R );
        glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT );
        glTexGeni ( GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT );
        glTexGeni ( GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT );
        glTexParameterf(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_S,
GL_CLAMP);
        glTexParameterf(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_T,
GL_CLAMP);
        glTexParameterf(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexParameterf(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
    }

```

```

    glTexEnvf ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );

    // Could modify the texture matrix here to do light tracking...
    glMatrixMode ( GL_TEXTURE );
    glPushMatrix ();
    glLoadIdentity ();
    glMatrixMode ( GL_MODELVIEW );
}

// Draw the surface.
//
//
{
    glPushClientAttrib ( GL_CLIENT_VERTEX_ARRAY_BIT );
    // GL_VERTEX_ARRAY does not necessarily need to be
    // enabled, as it should be enabled before this routine
    // is called.
    glEnableClientState( GL_VERTEX_ARRAY );
    glEnableClientState( GL_NORMAL_ARRAY );

    glVertexPointer ( 3, GL_FLOAT, 0, &vertexArray[0] );
    glNormalPointer ( GL_FLOAT, 0, &normalArrays[0][0] );

    glDrawElements ( prim, indexCount, GL_UNSIGNED_INT, indexArray );

    // The client attribute is already being popped. You
    // don't need to reset state here.
    //glDisableClientState( GL_NORMAL_ARRAY );
    //glDisableClientState( GL_VERTEX_ARRAY );
    glPopClientAttrib();
}

{
    glMatrixMode ( GL_TEXTURE );
    glPopMatrix ();
    glMatrixMode ( GL_MODELVIEW );

    glDisable ( GL_TEXTURE_CUBE_MAP_EXT );
    glDisable ( GL_TEXTURE_GEN_S );
    glDisable ( GL_TEXTURE_GEN_T );
    glDisable ( GL_TEXTURE_GEN_R );

    glPopAttrib();
}

return MS::kSuccess;
}

```

4.3 The glUnbind Method

The glUnbind method is used for releasing resources. In the glBind() method, we created a new texture. We could release it here but do not take this approach for efficiency.

```

MStatus hwPhongShader::glUnbind(const MDagPath&)
{
    // Release texture later for efficiency
    return MS::kSuccess;
}

```

```

}

```

5.0 Hardware Shader Interface Changes in Maya 6.5

As of the Maya 6.5 release, the two previously described interfaces exist in the **MPxHwShaderNode** class. In this release, we have added several features that make implementing hardware shaders easier. These features will be described next.

5.1 Dirty Masks

Dirty masks are now available. This information allows you to find out which aspects of a piece of geometry has changed since the last draw. The enum provided in **MPxHwShaderNode** is the following:

```

enum DirtyMask {
    kDirtyNone                = 0x0000,
    kDirtyVertexArray        = 0x0001,
    kDirtyNormalArray        = 0x0002,
    kDirtyColorArrays        = 0x0004,
    kDirtyTexCoordArrays    = 0x0008,
    kDirtyAll                 = 0x000f
};

```

This mask is used in conjunction with the `dirtyMask()` method to find out which geometry items have changed since the last plug-in draw call. The mask is valid only at the time that the `geometry()` or `glGeometry()` methods are called.

5.2 Current Path

It is also possible now to get the Dag Path to the current object being drawn using the **MPxHwShaderNode** class. This allows a plug-in to look up properties on the node. For example, your plug-in may wish to alter the drawing of the node based on blind data settings. The method to call is:

```

const MDagPath & currentPath() const;

```

The path is only valid during calls to any of the following attribute specifying methods:

```

normalsPerVertex()
colorsPerVertex()
texCoordsPerVertex()
getTexCoordSetNames()
hasTransparency()
provideVertexIDs()

```

The path is not guaranteed to be valid at any other time.

5.3 Images for the Texture Editor

Previously, there was no integration between plug-in hardware shaders and the Texture Editor. We now provide a way for a plug-in hardware shader to display a texture in the Texture Editor. The methods of interest on **MPxHwShaderNode** are:

- `virtual MStatus getAvailableImages(const MString& uvSetName, MStringArray& imageNames);`

Override this method to specify the list of images that are associated with the given UV set in this shader. This method is used to determine which texture images are available in the UV Texture Editor.

- `virtual MStatus renderImage(const MString& imageName, const float region[2][2], int& imageWidth, int& imageHeight);`

Override this method to draw an image of this material. This method allows a shader to override the rendering and Maya will only use this method if `getAvailableImages()` returns at least one image name. The `imageWidth` and `imageHeight` parameters should be populated with the native resolution of the input image to allow pixel snapping or other resolution dependent operations.

6.0 Hardware Shader Interface Changes in Maya 8.0 and 8.5

In the Maya 8.0 and 8.5 releases, there were several important changes made to the `MPxHwShaderNode` class. These changes are described next along with a related change to the opening up of the Maya viewport.

6.1 Batch shading of shapes with the same material (Maya 8.0)

In this release, changes have been made to the Maya hardware shaders to support a more efficient draw. Batching of shaders is the new operation that has been introduced. All hardware shaders follow the following algorithm:

```

For every unique shader (MPxHwShaderNode node instance)
{
    glBind
    For all shapes (meshes/NURBs) using the shader
    {
        glGeometry
    }
    glUnbind
}

```

If you are not using batching or the shape has marked itself as being "transparent" then the code looks like the following:

```

For every shape
{
    For the shader (MPxHwShaderNode node instance)
    assigned to the shape
    {
        glBind
    }
}

```

```

        glVertex for the shape(meshes/NURBs) using `
            the shader
    glEnd
}
}

```

Traversal is by shape order and not by shader order.

You can control the batch shading of shapes with the same material by implementing a method derived from the MPxHwShaderNode class. This method is:

```
virtual bool MPxHwShaderNode::supportsBatching( void ) const;
```

In the batching case, the glBind() and glEnd() methods will no longer be called for each shape. As a result, the callee will no longer be able to extract out information about the current shape such as its object to world transform in the bind and unbind stages. This must be done within glVertex().

6.2 More control over texture coordinates and transparency (Maya 8.5)

Some image formats may require the orientation of texture coordinates to be flipped. This can be controlled using the method:

```
virtual bool MPxHwShaderNode::invertTexCoords() const;
```

The default behaviour is to return false which sets the origin of the UV space at the bottom left corner of the texture. If true is returned, the origin of the UV space will be the top-left hand corner of the texture. An example of a file format that requires this behavior is DDS.

More control over transparency can be achieved by using the method and enum:

```
virtual unsigned int MPxHwShaderNode::transparencyOptions();
```

```
enum TransparencyOptions
{
    // When set means draw transparent
    ///
    kIsTransparent = 0x0001,

    // When set means ignore front back cull
    ///
    kNoTransparencyFrontBackCull = 0x0002,

    // When set means ignore polygon sorting
    ///
    kNoTransparencyPolygonSort = 0x0004
};
```

This method supersedes the isTransparent() function if that method returns false.

Maya viewports set a per viewport transparency sorting mode at the object and polygon level. Additionally Maya refresh can draw transparent objects by drawing them twice -- once for the back faces, and once for front faces. An MPxHwShaderNode can set bits to control the desired behaviour. They can:

- enable transparency (specify kIsTransparent)
- disable the two pass drawing for transparency (specify kNoTransparencyFrontBackCull)
- disable polygon sorting mode (kNoTransparencyPolygonSort)

There is no way to individually disable sorting of transparent objects at the object level. It is either on or off for all objects in the scene. Again, control is on a per viewport basis.

6.3 More control over the Maya viewport

Closely related to hardware shaders is the ability to override drawing in the Maya viewport. In Maya 8.0 and 8.5, we have introduced several new classes that allow more control over viewport rendering. These classes are as follows:

6.3.1 MViewportRenderer

MViewportRenderer is a class which represents a hardware viewport renderer. This class allows the overriding of viewport rendering. You need to derive from it to create your own custom renderer. By default you can either override everything, or allow for some "mixed" rendering. Note that "mixed" means that the custom renderer does not change the state that Maya inherently assumes that it's drawing into. At the root level this is an OpenGL rendering context / buffer with colour, depth, stencil, accumulation buffer and overlay planes available.

As such a renderer must specify the drawing API that it is using to avoid possible drawing clashes. OpenGL and Direct3D® are the two main possibilities. Specifying "software" means that there is no hardware component to drawing.

The renderer can be initialized and uninitialized via initialize() and uninitialized() methods. The renderer cannot make any assumptions as to when either of these methods will be called, nor assume that there is some valid graphics state available to them. In general initialize will be called when the renderer is first chosen as the current renderer for a viewport. uninitialized() will generally occur when Maya exits, or when the plug-in is unloaded or "deregistered".

To be able to use the renderer within Maya, it needs to be registered. This can be done using the register() method. deregister() is the corresponding method for unregistering the renderer.

The renderer must have a unique internal name as specified on creation. A renderer is generally created on plug-in initialization, but is not initialized at this point. The name should not be changed while the plug-in is registered. A user interface name can be specified and changed.

When a renderer is registered, it will show up under the "Renderer" menu for each viewport. The UI name will show up as the menu item name. If a MEL script called:

```
<internalName>OptionBox.mel
```

is specified then an option box option will appear with the menu. An example is the Direct3D renderer (D3DViewportRenderer) which is provided as an example in the SDK. The internal name is, and the option box name, is D3DViewportRendererOptionBox.mel with a D3DViewportRendererOptionBox() MEL procedure inside.

There is the concept of a "current" renderer for each viewport. Choosing the menu item for your renderer from the viewport menus will make it the current one. Alternatively, the current renderer can be set via the MEL command modelEditor.

The render() method is called whenever your renderer is the current one. It will be called for each refresh generated by Maya. The plug-in writer cannot, at this time, force more calls to render(). For those renderers which draw using OpenGL, the background will be cleared, and the current camera (perspective matrix) will be set up for them. At the end of the render call, Maya will display the rendered image to screen. For those renderers not using OpenGL, they will need to draw into OpenGL. The easiest way to achieve this is render into the MRenderTarget (the OpenGL render target) by passing an MImage to draw to. See

MRenderTarget for more information. The D3DViewportRenderer has example code which does this.

6.3.2 MDrawTraversal

MDrawTraversal is a new utility class that traverses through the current scene graph and provides a means of accessing a list of visible objects with respect to a given view and view volume specification. Using this class involves setting the frustum of interest and then iterating over the objects that are contained within the frustum.

In relation to the MViewportRenderer, this class can be used to traverse the scene for rendering during a render() call. Currently, this class does not handle per 3D modeling viewport display filters such as display layers, and isolate select.

6.3.3 MGeometryManager

The MGeometryManager class provides access to Maya cached renderable geometry. Internally, Maya has rendering information that may not be in a compatible format for native rendering. Cached renderable geometry converts Maya rendering information to a format that is compatible for rendering. This class makes use of the MGeometryRequirements class and returns MGeometry objects to the code that calls it.

Currently, on polygonal objects can have their data queried. It is possible for a MPxHwShader to ask for all of its geometry via the manager instead of the current geometry querying interface. For MViewportRenderers this is the easiest way to retrieve draw friendly geometry for display.

6.3.4 MGeometryRequirements

The MGeometryRequirements class describes the set of geometry data elements required by a geometry user. For example, a material can specify that it needs position, normal and UV set "a" to render a surface.

6.3.5 MGeometry

MGeometry stores a collection of MGeometryData arrays which describe a Maya surface, including per-component data such as UV mapping and color. Included in this information is a MGeometryPrimitive that is described next.

This class has a uniqueID() which changes whenever new data of a given type on a surface changes. This can be used to determine when geometry has changed from frame to frame (i.e. has become "dirty")

6.3.6 MGeometryPrimitive

MGeometryPrimitive is a class that describes the topology used for accessing MGeometryData. Information such as the primitive type (points, lines, etc.) can be accessed using this class.

6.3.7 MRenderTarget

MRenderTarget is a class that contains information about a given hardware render target. A MRenderTarget describes a region that is being rendered. A class object of this type is returned by the MRenderingInfo class that is described below.

Currently there is no way to create your own custom render targets, and the only targets that are provided are OpenGL base on-screen targets. When rendering with a MViewportRenderer, the MRenderTarget is provided as a data member for a MRenderInfo. There is currently only minimal functionality which allows for setting the current rendering target, and for writing to the color and or depth buffer. An MImage with the appropriate data is required.

6.3.8 MRenderingInfo

MRenderingInfo is a class which holds information about rendering into hardware render targets. Information such as the origin, height and width of the rendering is accessible via this class. MRenderingInfo objects are utilized by the MViewportRenderer class described previously.

6.3.9 MHwTextureManager

The MHwTextureManager class allows access to OpenGL file texture functionality. Using this interface, plug-ins can load and use hardware textures.

This class only supports managing file textures loaded on a file texture node into OpenGL. Additionally, only 2D textures are supported. If the user has added in a custom image format loader (derived from MPxImageFile), then the texture will be loaded via the plug-in loader's glLoad() or load() methods. load() will load data into system memory and then bind it as a hardware texture. glLoad() loads the image on disk directly into a hardware texture. This would be a way to load custom image formats directly into video card memory for efficiency.

6.3.A MGLFunctionTable

Wrapper class for the OpenGL bindings that Maya uses. Core, as well as various OpenGL, extensions are supported. This class can be used to alleviate the dependency on either a 3rd party, or writing your own drawing API class. OpenGL buffer / render context management is not provided with this class. The class is cross platform.

7.0 Setup and Access to Effects Data

When creating effects, it is often not enough to only access the object space geometry and view state. In this section, we will discuss how to access certain types of other useful data, and the handling of different coordinate spaces.

7.1 Geometric Coordinate Spaces

The plug-in writer can transform the object space 3D position and normal information passed to it by using the various matrix methods on the MDagPath for the object. As previously noted, the MDagPath is available via the MDrawRequest which is passed in as an argument to each of the bind(), geometry(), and unbind() calls respectively. (As of Maya 6.0, the dag path of the object being drawn is available directly from MPxHwShaderNode but only during certain calls.)

MDagPath matrix access methods include the following:

```
Mmatrix inclusiveMatrix(MStatus * ReturnStatus = NULL) const;
Mmatrix exclusiveMatrix(MStatus * ReturnStatus = NULL) const;
Mmatrix inclusiveMatrixInverse(MStatus * ReturnStatus = NULL) const;
Mmatrix exclusiveMatrixInverse(MStatus * ReturnStatus = NULL) const;
```

Note that MMatrix is row-order and OpenGL is column-order. There are two ways to handle this. One is to transpose the matrix by hand. Another is to use the GL_ARB_transpose_matrix OpenGL extension if it is available.

The following is an example of deriving a matrix to get to eye-space from object-space:

```
// Get the modelView matrix, and transpose into Maya
// matrix ordering
```

```

// Find the dag path
MDagPath dagPath;
dagPath = request.multiPath();

GLfloat modelViewMatrix[16];
glGetFloatv(GL_MODELVIEW_MATRIX, modelViewMatrix);
float transposeMatrix[4][4];
for (int i=0; i<16; i++)
{
    transposeMatrix[i/4][i%4] = modelViewMatrix[i];
}
MMatrix mvMatrix(transposeMatrix);

// Get the object-to-world matrix (Maya matrix ordering)
MMatrix modelMatrix = dagPath.inclusiveMatrixInverse();

// Calculate the view transpose matrix.
//
MMatrix toEyeMatrix = modelMatrix * mvMatrix;

// Transpose back to OpenGL matrix ordering
toEyeMatrix = mv.transpose();

```

In this example, *request* is the **MDrawRequest** passed to the routine. The first part of the code handles getting the model view matrix. The second part gets the object-to-world transform, and concatenates it with the model view matrix to obtain the final object-to-eye-space transform *toEyeMatrix*.

7.2 Access to Camera and Light Information

Camera information can be accessed via the **M3dView** which is passed as an argument to *bind()*, *geometry()*, and *unbind()*. For example, to perform texture cube environment mapping, it is necessary to take into account the camera rotation and use it as a means to build a 3D texture transform matrix. An example is as follows:

```

// Get camera information from the 3D view "view", where
// "view" is the M3dView passed into this routine.
//
MDagPath cameraPath;
view.getCamera( cameraPath );

// Get rotation angle and axis
//
MVector camAxis;
double camTheta;
MMatrix mmatrix = cameraPath.inclusiveMatrix( &status );
MTransformationMatrix tmatrix( mmatrix );
m_CameraRotation = tmatrix.rotation();
m_CameraRotation.GetAxisAngle( camAxis, camTheta );

// Convert rotation to degrees from radians
camTheta *= 57.295779513082320876798154814105; // ~= (180 / M_PI)

// Flip from Maya to OpenGL coordinates, and
// rotate the Textures according to the camera orientation

```

```

//
glMatrixMode( GL_TEXTURE );
glPushMatrix();
glLoadIdentity();

// Maya texture space is 180 degrees flipped
glRotated( 180.0, 1.0, 0.0, 0.0 );

// Do camera texture rotation
glRotated( camTheta, camAxis[0], camAxis[1], camAxis[2]);
glMatrixMode( GL_MODELVIEW );

```

For each **M3dView**, Maya will set up the state of the hardware lights based on the light objects that have been created by the user. The plug-in writer must handle two cases:

1. If the hardware shader effect does not perform its own lighting computation, then there is no extra work which needs to be done.
2. If the hardware shader effect requires light information, it can be accessed using:
 - A dag (object hierarchy) iterator such as **MitDag** (for versions of Maya prior to 4.5) or access methods on **M3dView** (for versions of Maya above 4.5).
 - If lighting is not to be used for the effect, then the plug-in writer can allow for lighting to be disabled (e.g., via a `glDisable(GL_LIGHTING)` OpenGL call)

← Formatted: Bullets and Numbering

Maya supports four different lighting modes per 3D view panel:

1. Use all lights (up to the maximum number of OpenGL lights supported).
2. Use only selected lights (up to the maximum number of OpenGL lights supported).
3. Use only active lights (up to the maximum number of OpenGL lights supported).
4. Use a default light (not associated with a light shape).

The Maya 4.5 (and above) methods allow for the following functionality:

1. Getting the number of light shapes in the scene (`getLightCount()`).
2. Getting the lighting mode, as previously described (`getLightingMode()`).
3. Getting the DAG paths to light shapes (`getLightPath()`).
4. Determining whether or not a light is visible (`isLightVisible()`).
5. Getting the OpenGL light index for a given light shape (`getLightIndex()`).

```

MdrawRequest request; // Request is passed into the methods that
need this info

M3dView view = request.view();
boolean onlyVisible = FALSE;
MDagPathArray lightPathArray;
MDagPath lightPath;
MIntArray openGL_lightIndices;
int lightIndex;
boolean lightVisible;

M3dView::LightingMode mode;
view.getLightingMode(mode);

// Only do the check if we are in "all" lighting mode
if (mode == M3dView::kLightAll)
{

```

```

// Find out the number of total lights in the scene
unsigned int numberOfLights = view.getLightCount( onlyVisible
);
for (unsigned int I=0; I<numberOfLights; I++)
{
    // Check if light is visible, if it is than,
    // store the dag path and the OpenGL light index
    //
    view.isLightVisible( I, lightVisible );
    if (lightVisible)
    {
        view.getLightPath( I, lightPath );
        lightPathArray.append( lightPath );

        view.getLightIndex( I, lightIndex );
        openGL_lightIndices.append( lightIndex );
    }
}
}
}

```

7.3 Access to Textures

The OpenMaya API class **MImage** is available as an easy to use file texture loading utility.

In general, it will handle all the file types that Maya "file texture node" can handle, so that the user can use all of the file texture types that they currently familiar with. This class allows for the loading of file textures from disk, and access to the actual block of pixels (**pixmap**) that are created on file load.

This pixmap, along with access to the pixmap's width, height, and depth allows plug-in writers to create and bind OpenGL textures as required. Full control is given to the plug-in writer to determine the specific parameters used for binding, including control over things like mip-map filtering.

The following is an example of reading and binding a 2D RGBA Targa file from disk.

```

MImage texture;
MString fileTextureName("D:/sampleFileTexture.tga");
unsigned int width, height;

// See if we can read in the file texture
MStatus stat;
if (!(stat = texture.readFromFile(fileTextureName)) )
    return MS::kFailure;

// Do any filtering required...

// Create a texture ID
GLuint textureName;
glGenTextures(1, &textureName);

// Bind the 2D texture
glBindTexture( GL_TEXTURE_2D, texNames[0] );
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
            GL_UNSIGNED_BYTE, texture.pixels() );

```

As of Maya version 4.5, MImage has the capability to now write images out to disk. The new method added is called:

- `writeToFile()` : This method allows the option of performing image filtering on file write, as well as to scale or offset the image.

Some limited image filtering (or conversion) is also available in this version. **MImage** supports three basic format options:

- No special format
- Height Field format for bump maps
- Normal Map format for bump maps

The method `filterExists()` returns true if MImage has a filtering support to convert from one format to another. Currently only height field to normal map conversion is supported. This allows the plug-in writer to provide the capability of reading in color maps from file, and pre-filtering for usage in hardware bump mapping.

The method `filter()` performs filtering from one format to another with an optional scale for offset. For example if we wanted to perform normal map conversion on the previously specified MImage example we would add the following code where it says “Do any filtering required...” above:

```
if (MImage::filterExists( MImage::kHeightFieldBumpFormat,
MImage::kNormalMapBumpFormat)
{
    // Just for the example, also do a scale and offset
    double scale = 1.5;
    double offset = 0.5;

    texture.filter(MImage::kHeightFieldBumpFormat,
MImage::kNormalMapBumpFormat, scale, offset);
}
```

8.0 Other Considerations

In this section, we discuss software rendering, the artist and other items to consider when implementing a plug-in shader.

8.1 Software Rendering

The software rendering part of the node controls how shaded shapes appear in the software render. To allow a hardware shader to be software shaded, you must use a number of predefined attributes in this class as well as the `compute()` method which is inherited from **MPxNode**, the parent class of **MPxHwShaderNode**.

This allows for the node to appear like an internal shader that can be accessed and "connected" like any other shading node. Therefore, from the artist's point of view, the plug-in shader will appear as a shader in the standard UI elements such as the *HyperShade* or *MultiLister* panels, as well as in the Attribute Editor and Channel Box.

The important attributes added as part of this class are:

1. **Output Color:** This is the color evaluated by this node. This attribute can be used to connect this node to other nodes that accept three-channel color as input. Normally this connection is to a shading group node (derived from **MFnSet**)
2. **Output Transparency:** This is the transparency evaluated by this node. This attribute can be used to connect this node to other nodes which accept three

channel transparency as input

3. **Matte transparency:** This is the matte transparency evaluated by this node. This attribute can be used to connect this node to other nodes which accept three channel matte transparency as input
4. **Glow color:** This is the glow color evaluated by this node. This attribute can be used to connect this node to other nodes which accept three channel glow as input

The `compute()` method does not necessarily need to be implemented, nor do the base level attributes need to be used during the `compute()` method. This method is only necessary in the cases where the plug-in writer wishes to visualize the computation of the node for "software rendering". Since this software render evaluation is also performed to derive the visual "swatches" used for the attribute editor, HyperShade and MultiLister, the plug-in writer may want to implement the `compute()` method to give a visual indication as to the result of the shader computation.

The following is a very simple example of changing the behavior of the output color to red (1.0, 0.0, 0.0) for a node called *simpleHwShader* that is derived from **MPxHwShaderNode**.

```
MStatus simpleHwShader::compute(const MPlug &plug,
                                MDataBlock& block)
{
    // Check to see if we want to compute the
    output color
    bool k = false;
    k |= (plug==outColor);
    k |= (plug==outColorR);
    k |= (plug==outColorG);
    k |= (plug==outColorB);
    if( !k ) return MS::kUnknownParameter;

    // Set output color attribute to "red"
    MDataHandle outColorHandle = block.outputValue(
outColor );
    MFloatVector& outColor =
outColorHandle.asFloatVector();
    outColor.x = 1.0;
    outColor.y = 0.0;
    outColor.z = 0.0;
    outColorHandle.setClean();
    return MS::kSuccess;
}
```

If necessary, the plug-in writer could mimic the exact computation which is done for hardware rendering to achieve "exact" visual correlation.

8.2 Artist Considerations

To make a hardware shader "appear" as a regular shader, the plug-in writer can add a "classification" when registering the hardware shader node. This classification indicates to

Maya that when building the UI for render editors such as HyperShade and MultiLister that this is a certain type of shading node.

The following is an example of plug-in initialization. The important piece of information is the *UserClassify* string which, in this case, classifies the node as a surface shader.

```
MStatus initializePlugin( MObject obj )
{
    MStatus status;
    const MString UserClassify( "shader/surface/utility" );

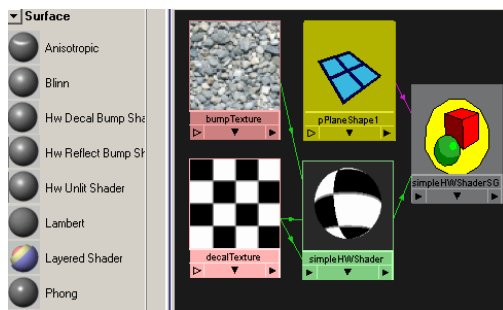
    // Load the hardware shader, specifying the user classification
    //
    MFnPlugin plugin( obj, "Autodesk - Example", "6.5", "Any");
    status = plugin.registerNode("simpleHwShader", simpleHwShader::id,
        simpleHwShader::creator,
        simpleHwShader::initialize,
        MPxNode::kHwShaderNode, &UserClassify );

    if (!status) {
        status.perror("registerNode");
        return status;
    }

    // Make sure the classification exists
    MString command( "if( `window -exists createRenderNodeWindow` )
    {
        refreshCreateRenderNodeWindow(\" \");
    }");
    command += UserClassify;
    command += "\n");
    MGlobal::executeCommand(command);

    return MS::kSuccess;
}
```

The following is a snapshot of a hardware shader called "simpleHwShader" which has been assigned to an object called "pPlaneShape1". The "Create Materials" tab in HyperShade is shown with a few hardware shaders ("HW ..."). When a new hardware shader is created, since the "Include Shading Group with Materials" create option is enabled, the shading group (*simpleHwShaderSG*) will be automatically created and the output color attribute on the hardware shader will be connected to this shading group.



After the node is created, the artist/user is free to perform normal shading network connections to the hardware shader just like any regular shader such as Lambert or Blinn. For example, in the above snapshot, a decal and a bump texture has been assigned

respectively to the color and bump attributes on the simpleHwShader plug-in node. In this case, because the user wants to see the texture appear in the simpleHwShader shader swatch, there is a hook up from decalTexture to the “outColor” attribute. This can be done using drag-and-drop (The “default” option means to hook up to “outColor”).

It is up to the plug-in writer to provide the necessary attributes, and the proper interpretation of the shading network. Since the plug-in writer has full control over what the hardware shader expects as inputs, the writer can easily customize this interpretation.

8.3 OpenGL Details

Maya is available on three operating systems (Windows®, Linux®, and Mac OS® X) and as such is an *OpenGL* based application (because OpenGL is supported on all three of these systems). Maya hardware shader plug-in writers need to use the OpenGL display API.

8.3.1 Binding OpenGL Extensions

As of Maya 8.5, the MGLFunctionTable class can be used instead of writing custom code for loading OpenGL extensions. Previous to this release, a more low level approach was required. We describe this low level approach next.

Unlike the Direct3D API, the plug-in writer may need to bind the OpenGL extensions they use. These may be either EXT, ARB or vendor specific extensions for functionality that they may wish to use in their shader. If programmers are writing for a Windows platform, then, by default only OpenGL v1.1 is supported. Depending on vendor specific driver implementation, higher versions may be supported. For Linux systems, the default OpenGL version is usually v1.2. In both cases, the plug-in writer may need to check for extensions. This usually entails checking a string that lists the available extensions, and then, if the extension has functions associated with it, to bind those functions. For Windows, the procedure for binding is *wglGetProcAddress()* while on Linux and IRIX this would be *glxGetProcAddress*. For Macintosh, only the extension string needs to be checked. No binding of functions is required.

Sometimes the extension definitions have already been defined in the operating system supplied *glext.h* or *gl.h*, or by vendor specific versions of these files. If this is the case the definition is not required, but the binding may still be required.

As a simple example, the following is an example code for binding `GL_EXT_fog_coord` (vertex fog). In the include file for the plug-in you would add code like the following:

```
// Define APIENTRY per platform
#ifndef APIENTRY
#   ifdef _WIN32
#       define WIN32_LEAN_AND_MEAN 1
#       include <windows.h>
#   else
#       define APIENTRY
#   endif
#endif

/* GL_EXT_fog_coord */
#ifndef GL_EXT_fog_coord
    #define GL_EXT_fog_coord 1

    // Extension constants
    #define GL_FOG_COORDINATE_SOURCE_EXT          0x8450
    #define GL_FOG_COORDINATE_EXT                0x8451
    #define GL_FRAGMENT_DEPTH_EXT               0x8452
    #define GL_CURRENT_FOG_COORDINATE_EXT       0x8453
    #define GL_FOG_COORDINATE_ARRAY_TYPE_EXT    0x8454
    #define GL_FOG_COORDINATE_ARRAY_STRIDE_EXT  0x8455
    #define GL_FOG_COORDINATE_ARRAY_POINTER_EXT 0x8456
    #define GL_FOG_COORDINATE_ARRAY_EXT        0x8457

```

```

// Define function prototype type
typedef void (APIENTRY * PFNGLFOGCOORDFEXTPROC) (GLfloat
coord);
typedef void (APIENTRY * PFNGLFOGCOORDFVEXTPROC) (const GLfloat
*coord);
typedef void (APIENTRY * PFNGLFOGCOORDDEXTPROC) (GLdouble
coord);
typedef void (APIENTRY * PFNGLFOGCOORDDVEXTPROC) (const
GLdouble *coord);
typedef void (APIENTRY * PFNGLFOGCOORDPOINTEREXTPROC) (GLenum
type, GLsizei stride, const GLvoid *pointer);

// Test if the prototypes are predefined
#ifdef GL_GLEXT_PROTOTYPES
    extern void APIENTRY glFogCoordfEXT (GLfloat);
    extern void APIENTRY glFogCoordfvEXT (const GLfloat *);
    extern void APIENTRY glFogCoorddEXT (GLdouble);
    extern void APIENTRY glFogCoorddvEXT (const GLdouble *);
    extern void APIENTRY glFogCoordPointerEXT (GLenum,
GLsizei, const GLvoid *);
#else
    // We need to define our own prototypes
    #define BIND_OUR_OWN_FUNCTIONS_FOR_FOG_EXTENSION
    static PFNGLFOGCOORDFEXTPROC glFogCoordfEXT = NULL;
    static PFNGLFOGCOORDFVEXTPROC glFogCoordfvEXT = NULL;
    static PFNGLFOGCOORDDEXTPROC glFogCoorddEXT = NULL;
    static PFNGLFOGCOORDDVEXTPROC glFogCoorddvEXT = NULL;
    static PFNGLFOGCOORDPOINTEREXTPROC glFogCoordPointerEXT
= NULL;

#endif /* GL_GLEXT_PROTOTYPES */
#endif

```

In this example, a number of `#defines` are declared for the constants used by the extension functions. In addition, each function needs to have a `typedef` to specify the function's prototype. Depending on whether the prototypes have already been defined (`GL_GLEXT_PROTOTYPES`), the plug-in may need to define and bind its own local extension functions.

```

// Convenience macro to switch which function to call to bind
// extensions based on operating system..
#ifdef _WIN32
    // Convenience macro for declaring Windows API entries
    #define OGL_GET_PROC ( _type_, _entry_ ) \
        _entry_ = ( _type_ ) wglGetProcAddress( #_entry_ );
#elif LINUX
    // Convenience macro for declaring Linux OGL API entries.
    // As glXGetProcAddressARB may not be defined, do an extra
    // check here.
    #ifdef GLX_ARB_get_proc_address
    #define OGL_GET_PROC ( _type_, _entry_ ) \
        _entry_ = ( _type_ ) ::glXGetProcAddressARB( (const GLubyte *)
#_entry_ );
    #endif
#else
    #define OGL_GET_PROC( _type_, _entry_ )
#endif
. . .
// Get the OpenGL extension strings
const GLubyte *gfx_extensions = ::glGetString(GL_EXTENSIONS);
if (gfx_extensions)
{
    // Check for the "fog coordinate" extension. If it exists, bind
    // the functions.
    //
    if (strstr((char *)gfx_extensions, " GL_EXT_fog_coord"))
    {

```

```

// Set some kind of flag here saying that the extension is
supported..
#ifdef BIND_OUR_OWN_FUNCTIONS_FOR_FOG_EXTENSION
OGL_GET_PROC ( PFNGLFOGCOORDFEXTPROC, glFogCoordfEXT,
kHasFogCoordFlag);
OGL_GET_PROC ( PFNGLFOGCOORDFVEXTPROC, glFogCoordfvEXT,
kHasFogCoordFlag);
OGL_GET_PROC ( PFNGLFOGCOORDDEXTPROC, glFogCoorddEXT,
kHasFogCoordFlag);
OGL_GET_PROC ( PFNGLFOGCOORDDVEXTPROC, glFogCoorddvEXT,
kHasFogCoordFlag);
OGL_GET_PROC ( PFNGLFOGCOORDPOINTEREXTPROC,
glFogCoordPointerEXT, kHasFogCoordFlag);
}
}

```

9.0 Summary

In summary, what has been presented here is a very simple yet powerful API interface that allows plug-in writers to hook directly into the Maya refresh mechanism to allow for full control over the display of arbitrary effects. At the same time, a plug-in writer can provide artist friendly customization of the UI presented to users while still maintaining the familiar UI interface of shaders.



Autodesk and Maya are registered trademarks or trademarks of Autodesk, Inc., in the USA and/or other countries. All other brand names, product names, or trademarks belong to their respective holders. Autodesk reserves the right to alter product offerings and specifications at any time without notice, and is not responsible for typographical or graphical errors that may appear in this document.

© 2007 Autodesk, Inc. All rights reserved.