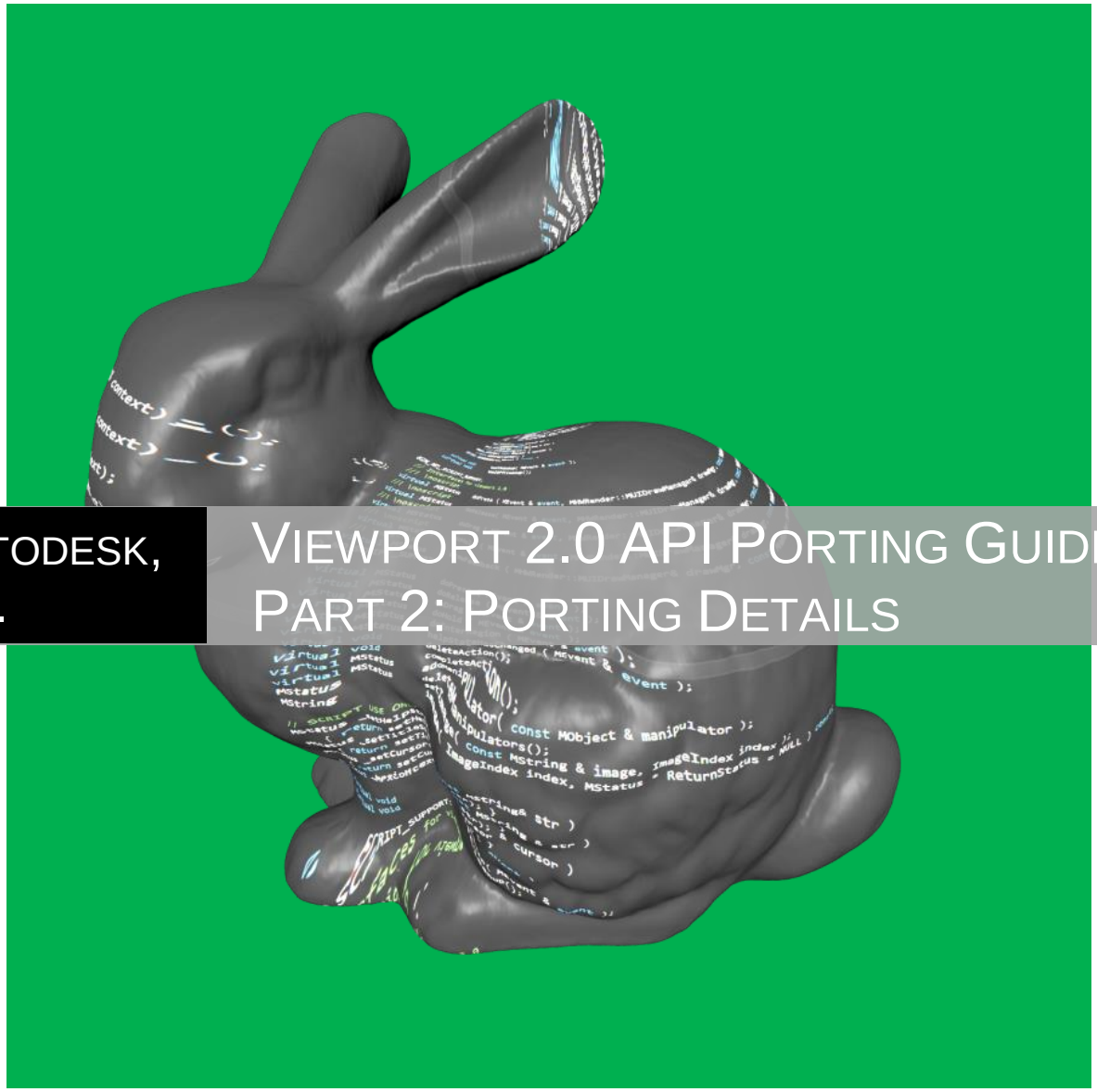
6/1/2017



AUTODESK, INC.

VIEWPORT 2.0 API PORTING GUIDE
PART 2: PORTING DETAILS

Getting Started Guide for Programmers Porting Plug-ins from VP1 to VP2 (Part 2) in Autodesk® Maya®

Revision History

4/1/2015	Initial version for Maya 2016
6/1/2017	Updates for Maya 2018: 6.3 Porting "Simple" Single Objects - Merged “VP2 API Porting Guide for Locators". 6.5 Porting Surface Shapes using MPxSubSceneOverride - Focused on complex parts because 6.3 has covered the basics.

1. Table of Contents

2. Introduction

This document provides details for porting a plug-in from using VP1 interfaces to using VP2 interfaces in Autodesk® Maya®. Various scenarios outlined in the *Viewport 2.0 API Porting guide, Part 1: Basics* are covered in detail in this document. This document is a continuation of *Part 1: Basics*, as reflected by its section numbering. Therefore, it should only be read after completing the reading of *Part 1*. The SDK examples/sample plug-ins referred to in this document are part of the Maya Developer Kit, which is available online at Autodesk Exchange at https://apps.exchange.autodesk.com/MAYA/en/Home/Index.

For more information on all the API interfaces mentioned in this document, see the *C++ API Reference* section of the Maya Developer Help at http://help.autodesk.com/view/MAYAUL/2017/ENU.

For more details on the Maya Viewport 2.0 API, see the *Viewport 2.0 API* section of the Maya Developer Help at http://help.autodesk.com/view/MAYAUL/2017/ENU. Topics from this section of the Maya Developer Help are referenced throughout this document.

For brevity, the Legacy Default Viewport / Viewport 1 will be denoted as VP1 and Viewport 2.0 denoted as VP2 throughout this document.

6. Porting Details

Throughout this document, for all provided code examples, important method calls are highlighted in **bold**, and specific enumerations in italics.

6.1 Porting Manipulators

For VP2 the key interfaces are:

- virtual void **preDrawUI**(const M3dView &view);
- virtual void **drawUI**(MHWRender::MUIDrawManager& drawManager, const MHWRender::MFrameContext& frameContext) const;

These should be used instead of the following VP1 interface:

- virtual void **draw**(M3dView & view, const MDagPath & path, M3dView::DisplayStyle style,M3dView::DisplayStatus status);

Examining the Developer Kit SDK example *footPrintManip* reveals that the basic DG translation is performed in preDrawUI(), and the drawing is done using the UI manager in drawUI().

```
Void footPrintLocatorManip::preDrawUI( const M3dView &view )
{
	// Update text drawing position
	fTextPosition = <Compute the text position>;
}
```

The draw code adds two drawables: one for 3d text and one for 2d text. These will be queued as transient items for drawing.

```
void footPrintLocatorManip::drawUI(
	MHWRender::MUIDrawManager& drawManager,
	const MHWRender::MFrameContext& frameContext ) const
{
	drawManager.beginDrawable();
	drawManager.setColor( MColor( 0.0f, 1.0f, 0.1f ) );
	// Use the text position to draw
	drawManager.text( fTextPosition, "3d text", MHWRender::MUIDrawManager::kLeft );
	drawManager.text2d( MPoint(100,100), "2d text", MHWRender::MUIDrawManager::kLeft);
	drawManager.endDrawable();
}
```

The SDK *lineManip* example demonstrates VP2 selection support for parts of a manipulator.

To indicate the parts of the manipulator that can be selected, the beginDrawable() method uses a unique name (lineName). To check if a drawable with lineName has been selected, a call into the convenience method *MPxManipulatorNode::shouldDrawHandleAsSelected*() is performed. In the sample, the color used for the drawable is changed to a “selected” color.

```
void	lineManip::drawUI(
	MHWRender::MUIDrawManager& drawManager,
	const MHWRender::MFrameContext& frameContext ) const
{
	bool drawAsSelected = false;
	shouldDrawHandleAsSelected(lineName, drawAsSelected);

	// Set this next drawable to be selectable, and adjust color if selected.
	drawManager.beginDrawable(lineName, true);
	drawManager.setColorIndex(drawAsSelected ? fSelectedColorIndex : fColorIndex );
	drawManager.line( fLineStart, fLineEnd );
	drawManager.endDrawable();

	// Set this next drawable not to be selectable
	drawManager.beginDrawable();
	drawManager.setColorIndex( fColorIndex );
	drawManager.text( fLineStart, MString("line manip"));
	drawManager.endDrawable();

	// Set this next drawable to be selectable, and adjust color if selected.
	drawManager.beginDrawable(lineName, true);
	drawManager.setColorIndex(drawAsSelected ? fSelectedColorIndex : fColorIndex );
	drawManager.line2d(MPoint(100, 100), MPoint(200, 100));
	drawManager.setLineWidth(5.0f);
	drawManager.endDrawable();

	// Set this next drawable to not be selectable
	drawManager.beginDrawable();
	drawManager.setColorIndex( fColorIndex );
	drawManager.setLineWidth(5.0f);
	drawManager.text2d(MPoint(100, 105), MString("line manip 2D"));
	drawManager.endDrawable();
}
```

The method *MPxManipulatorNode::glActiveName*() can be used to test for a selection of a given named drawable. Despite its name, this method is device independent in VP2. In the example, below, the manipulator tests the name within a drag event method:

```
MStatus lineManip::doDrag( M3dView& view )
{
	unsigned int activeName = 0;
	if ( glActiveName(activeName ) )
	{
	    // do work
	}
	return MS::kSuccess;
}
```

6.2 Porting Contexts

The basic VP1 interfaces for tools context (MPxContext, MPxTexContext) are:

- virtual MStatus doPress (MEvent & event);
- virtual MStatus doRelease (MEvent & event);
- virtual MStatus doDrag (MEvent & event);
- virtual MStatus doHold (MEvent & event);

The VP2 interfaces attempt to match the old interface signatures, but branch off to allow access to a MUIDrawManager and a frame context (MFrameContext):

- virtual MStatus **doPress** (MEvent & event, MHWRender::MUIDrawManager& drawMgr, const MHWRender::MFrameContext& context);
- virtual MStatus **doRelease**(MEvent & event, MHWRender::MUIDrawManager& drawMgr, const MHWRender::MFrameContext& context);
- virtual MStatus **doDrag** (MEvent & event, MHWRender::MUIDrawManager& drawMgr, const MHWRender::MFrameContext& context);
- virtual MStatus **doHold** (MEvent & event, MHWRender::MUIDrawManager& drawMgr, const MHWRender::MFrameContext& context);
- virtual MStatus **drawFeedback** (MHWRender::MUIDrawManager& drawMgr, const MHWRender::MFrameContext& context);

Note that drawFeedback() exists to allow for occasions when continuous update is required, such as for always drawing a 3d cursor.

The following sample code from the *marqueeTool* SDK example demonstrates the use of an instance of an MUIDrawManager to perform drawing during the drag event (doDrag()). As with VP1, the input event can be used to drive what to draw. The press and release non-drawing functionality can be shared between VP1 and VP2.

```
Mstatus marqueeContext::doDrag (
	MEvent & event,
	MHWRender::MUIDrawManager& drawManager,
	const MHWRender::MFrameContext& context)
{
	// Get the marquee's new end position.
	event.getPosition( last_x, last_y );
```

```
        // Draw the marquee at its new position.
        // Always drawn last so no need for complicated XOR
        drawManager.beginDrawable();
        drawManager.setColor( MColor(1.0f, 1.0f, 0.0f) );
        drawManager.line2d( MPoint( start_x, start_y), MPoint(last_x, start_y) );
        drawManager.line2d( MPoint( last_x, start_y), MPoint(last_x, last_y) );
        drawManager.line2d( MPoint( last_x, last_y), MPoint(start_x, last_y) );
        drawManager.line2d( MPoint( start_x, last_y), MPoint(start_x, start_y) );
        drawManager.endDrawable();
        return MS::kSuccess;
}
```

The main difference between the VP1 and VP2 implementations is that VP2 does not require special code on press and release (and draw) to perform XOR drawing.

This is because all context drawing can be thought of as being drawn as an “overlay”. Thus, for each event, only the current draw needs to be performed and the previous draw does not need to be “erased” using XOR drawing. (M3dView::beginXorDrawing()/endXorDrawing() should not be used for VP2 drawing.)

6.3 Porting “Simple” Single Objects

This section covers the possible approaches to porting over DAG objects that represent single objects. The term “simple” denotes that the complexity of the data to be drawn is small. To illustrate this, variations on four implementations of the footPrintNode locator will be examined:

1. footPrintNode_GeometryOverride: This plug-in adds VP2 support via an MPxGeometryOverride.
2. footPrintNode_SubSceneOverride: This plug-in adds VP2 support via an MPxSubSceneOverride.
3. footPrintNode: This plug-in adds VP2 support via MUIDrawManager.
4. rawfootPrintNode: This plug-in adds VP2 support via an MPxDrawOverride.

After examining the implementation, each approach will be analyzed based on the following factors. Pros will be marked with ✓ and cons are marked with ✗.

- Portability: For example, how easy is the implementation, how much work is required?
- Scalability: For example, how does it perform as the number of objects increases?
- Compatibility/Flexibility: For example, which drawing APIs/platforms is it compatible with, does it interact with other features, and which scenarios are suitable?

Note: *It is possible to reuse legacy viewport drawing and picking when using Viewport 2.0 for plug-in locators. However, this is only available since Maya 2017 and should only be considered a temporary solution to allow for an incremental migration to Viewport 2.0.* Fore more details

see *Use MPxLocatorNode legacy fixed draw code and selection in Viewport 2.0* in Maya Developer Help.

6.3.1 Porting Using a Geometry Override

This example uses an MPxGeometryOverride to handle drawing for a locator (MPxLocatorNode). The example code listed below is taken from the *footPrintNode_GeometryOverride* sample plug-in. Within the plug-in, the life-cycle of one render item that uses a simple stock shader will be examined.

The basic configuration for a geometry override appears as follows, where the override is associated with the DAG object, and the render items are associated with the override.

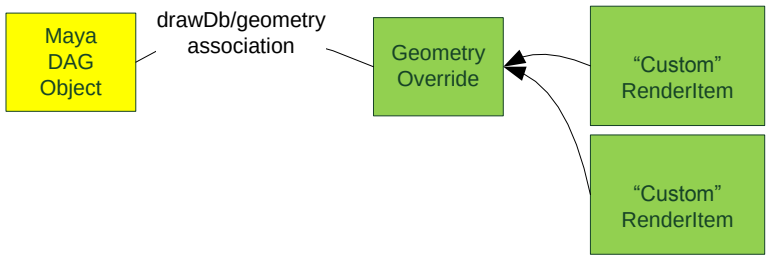


Figure 1: Sample drawDB/geometry association to a geometry override which will produce custom render items.

6.3.1.1 MPxGeometryOverride Registration

To allow the geometry override to be associated with the locator, the same drawdb classification should be used during registration.

First, the Maya node must have a classification string specified. This is the last argument of *MFnPlugin*::*registerNode*() as shown below. In the classification string:

- "geometry" designates it as a geometry evaluator.
- "footPrint" designates a specific geometry, in this case, the "footPrint" evaluator.

Each plug-in would have an appropriate unique identifier instead of "footPrint".

```
MString  footPrint::drawDbClassification("drawdb/geometry/footPrint");

MStatus status = plugin.registerNode(
                     "footPrint",
                     footPrint::id,
                     &footPrint::creator,
                     &footPrint::initialize,
                     MPxNode::kLocatorNode, // Is a locator
                     &footPrint::drawDbClassification); // Use the VP2 classification
```

The geometry override needs to use the exact same classification string in order to form an association. An additional unique id is required to identify the registration as well as creator function which will create a new instance of the override class.

```
status = MHWRender::MDrawRegistry::registerGeometryOverrideCreator(
       footPrint::drawDbClassification, // Use same classification as the node
```

```
        footPrint::drawRegistrantId, // Registration id
        FootPrintGeometryOverride::Creator); // Instance creator function
```

At plug-in unitialization time, a corresponding deregistration is performed:

```
status = MHWRender::MDrawRegistry::deregisterGeometryOverrideCreator(
              footPrint::drawDbClassification, // Use same classification as registration
              footPrint::drawRegistrantId); // Use same registration id
```

6.3.1.2 MPxGeometryOverride Interfaces

In addition to registration via MDrawRegistry, there are a few key interfaces that a geometry override (evaluator) must implement. These interfaces are called in a specific order and reflect the internal separation of computation:

- DG Evaluation: The **updateDG**() method should be the only location where DG evaluation occurs.
- Render Item Handling: The **updateRenderItems**() method is responsible for the updating the parameters on an existing render item, or creating new ones.
- Geometry Update: The **populateGeometry**() method will update both data and indexing streams. Data streams are added to the geometry container for the override. Indexing will be referenced by render items associated with the override.

The following diagram shows this graphically. In the locator case we will ignore the surface shader update since no surface shaders can be associated with locators:

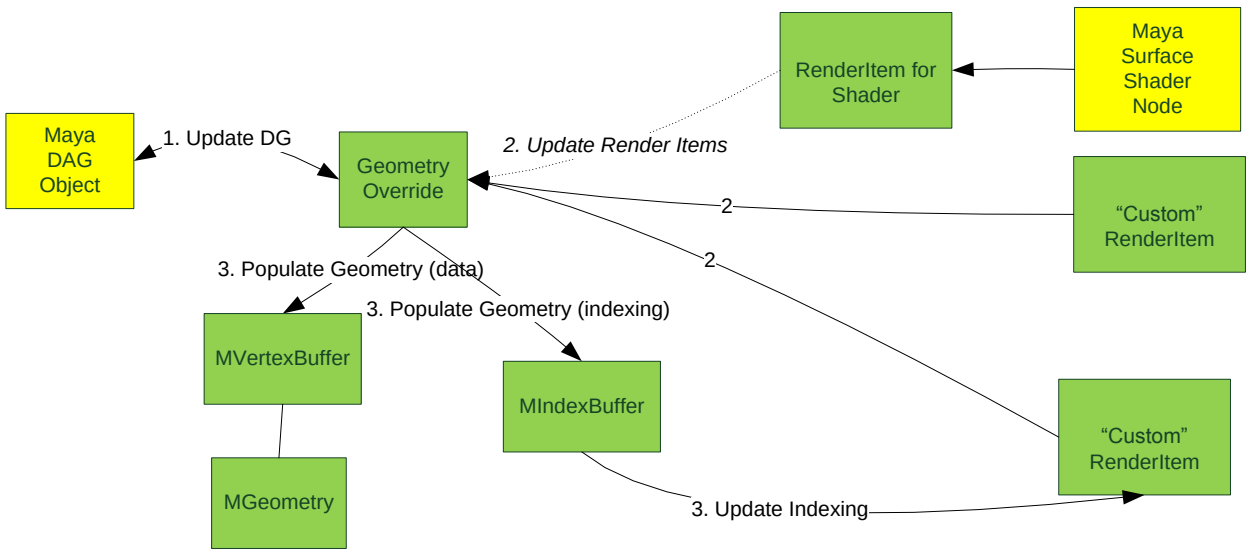


Figure 2: Basic steps enumerated to perform node data extraction, process render items, and then update geometry for those items.

6.3.1.3 DG Update

In the example, the DG evaluation is fairly simple but demonstrates the separation of DG evaluation from draw. Compare this code in VP1 which performs DG evaluation during draw:

```
void footPrint::draw( M3dView & view, const MDagPath & /*path*/,
                                                    M3dView::DisplayStyle style,
                                                    M3dView::DisplayStatus status )
{
       // Get the size
       //
       MObject thisNode = thisMObject();
       MPlug plug( thisNode, size ); // ← Pull data on the size attribute
       MDistance sizeVal;
       plug.getValue( sizeVal );

       float multiplier = (float) sizeVal.asCentimeters();
```

to this VP2 code which performs the same action during **updatedDG**() for the geometry override. The scale is cached locally on the override class instance for reuse during geometry update.

```
void FootPrintGeometryOverride::updateDG()
{
       MPlug plug(mLocatorNode, footPrint::size);
       float newScale = 1.0f;
       if (!plug.isNull())
       {
              MDistance sizeVal;
              if (plug.getValue(sizeVal)) // ← Pull data on the size attribute
              {
                     newScale = (float)sizeVal.asCentimeters();
              }
       }
       if (newScale != mMultiplier)
       {
              mMultiplier = newScale;
              mMultiplierChanged = true;
       }
}
```

6.3.1.4 Render Item Update

Because a locator has no inherent shader assignment, at render item update time, updating of shaded items is not required. (Handling of surface shapes which do have shaded items provided is discussed in "Porting Surface Shapes").

Instead, the plug-in requires additional "UI" items to be explicitly created and updated for the drawing of the "heel" and "sole" part of the footprint shape.

The basic logic behind the update process can be summarized as follows:

- Check if a named persistent render item already exists using **MRenderItemList::indexOf**().
- If it does not exist, then create it using **MRenderItem::Create**() and append it to the list of persistent render items using **MRenderItemList::append**(). At creation time, the following parameters must be appropriately set:
    - Rendering type
    - Topology for indexing
    - Draw mode using **MRenderItem::setDrawMode**().
- Render item or associated shader can then be updated.
    - Render item is associated with shader using **MRenderItem::setShader**().

The following code snippet shows this logic for one render item which is identified by the *wireframeItemName* variable. This code draws a wireframe of the foot print. It is designated as a wireframe "decoration" or UI item and will only draw when wireframe mode is enabled.

```
void FootPrintGeometryOverride::updateRenderItems( const MDagPath& path,
                                                    MHWRender::MRenderItemList& list )
{
    // The MGeometryUtilities class is used here as a convenience to get the
    // appropriate color for a given DAG path. A mini cache is maintained in
    // the plug-in to retrieve the corresponding shader instance of a given
    // color and allow shader instances to be reused, so that VP2 performance
    // optimizations such as consolidation and hardware instancing can be taken
    // advantage of for duplicates.
    MColor color = MHWRender::MGeometryUtilities::wireframeColor(path);
    MHWRender::MShaderInstance* shader = get3dSolidShader(color);
    if (!shader) return;

    // Pointer to the wireframe render item of interest
    MHWRender::MRenderItem* wireframeItem = NULL;

    // Check the persistent render list to see if we have a render item
    // with a given name
    int index = list.indexOf(wireframeItemName);

    // If the render item cannot be found then we need to create a new one
    if (index < 0)
    {
        // Create the new render item with the given name.
        // We designate this item as a UI "decoration" and will not be
        // involved in rendering aspects such as casting shadows
        // The "topology" for the render item is that it's a line list.
        wireframeItem = MHWRender::MRenderItem::Create(
            wireframeItemName,
            MHWRender::MRenderItem::DecorationItem,
            MHWRender::MGeometry::kLines);

        // We want this render item only show up when in "wireframe mode".
        wireframeItem->setDrawMode(MHWRender::MGeometry::kWireframe);

        // The item must be added to the persistent list to be considered
        // for update / rendering
        list.append(wireframeItem);
    }
```

```
    // If the item already exists then just find it in the persistent render
    // item list.
    else
    {
        wireframeItem = list.itemAt(index);
    }

    // When we have a render item, then we need to assign the shader and make
    // sure the render item is enabled.
    if (wireframeItem)
    {
        // Assign the shader to the render item. This adds a reference to that
        // shader.
        wireframeItem->setShader(shader);

        // We want to enable this render item. Disabling basically is equivalent
        // to making it invisible.
        wireframeItem->enable(true);
    }
```

6.3.1.5 Render Item Shader Instances

In order to allow for performance optimizations such as consolidation and hardware instancing, we want to minimize the number of unique shader instances used by all render items. We can do this by implementing a small shader instance cache and reusing the same shader instance for more than one render item, especially for the case of duplicated/instanced objects. See *Section 3.6 Categorization and Consolidation* for more information about how render items can be considered for consolidation.

Note: *To determine whether consolidation or hardware instancing is used, enable the debug tracing of VP2 render pipeline.*

```
// Maintain a mini cache for 3d solid shaders in order to reuse the shader
// instance whenever possible. This can allow Viewport 2.0 optimization e.g.
// consolidation and hardware instancing to be used.
struct MColorHash
{
    std::size_t operator()(const MColor& color) const
    {
        std::size_t seed = 0;
        CombineHashCode(seed, color.r);
        CombineHashCode(seed, color.g);
        CombineHashCode(seed, color.b);
        CombineHashCode(seed, color.a);
        return seed;
    }

    void CombineHashCode(std::size_t& seed, float v) const
    {
        std::hash<float> hasher;
        seed ^= hasher(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
    }
};
```

```
std::unordered_map<MColor, MHWRender::MShaderInstance*, MColorHash> the3dSolidShaders;

// Get a shader instance from the mini cache regarding to a given solid color.
MHWRender::MShaderInstance* get3dSolidShader(const MColor& color)
{
    // Return the shader instance if exists.
    auto it = the3dSolidShaders.find(color);
    if (it != the3dSolidShaders.end()) return it->second;

    // Otherwise, create a new shader instance for the given solid color and
    // store it into the cache. In this case, an instance of "stock 3d solid
    // shader" provided by the shader manager can be used to draw render items
    // with the required solid color.
    MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
    const MHWRender::MShaderManager* shaderMgr = renderer->getShaderManager();
    MHWRender::MShaderInstance* shader = shaderMgr->getStockShader(
        MHWRender::MShaderManager::k3dSolidShader);
    if (shader)
    {
        float solidColor[] = { color.r, color.g, color.b, 1.0f };
        shader->setParameter("solidColor", solidColor);
        the3dSolidShaders[color] = shader;
    }

    // Return the new shader instance.
    return shader;
}
```

At render item update time, a new shader instance may or may not be created, depending on whether the solid color required by the given DAG path has been used. Render items using the same solid color will be associated with the same shader instance, and therefore may be consolidated.

When the shader instances are no longer required, they should be released. As all the shader instances are reusable for the entire lifetime of the plug-in, they are released together after deregistration of the geometry override at plug-in uninitialization time:

```
MStatus releaseShaders()
{
    MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
    const MHWRender::MShaderManager* shaderMgr = renderer->getShaderManager();
    for (auto it = the3dSolidShaders.begin(); it != the3dSolidShaders.end(); it++)
    {
        // Release the reference to shader instances
        shaderMgr->releaseShader(it->second);
    }

    the3dSolidShaders.clear();
    return MS::kSuccess;
}
```

6.3.1.6 Updating Data Streams and Index Streams

The render items created or updated here are not processed by this class as it is only a geometry evaluator. When appropriate, this evaluator will be called back to update the

geometric data streams used as well as the indexing stream for each render item via a call to **populateGeometry**().

To re-iterate, the data streams are associated with the total requirements for the override, while the index streams are associated with specific render items for the override.

Two methods can specify whether streams and indexing are required to be updated:

- **bool MPxGeometryOverride::isStreamDirty()**: This can be used to indicate whether a specific stream requires an update. As an example, if the geometry is always static, it is possible to return a false value here so that the streams are regarded as never requiring an update. In this example, the frequency of update depends on the frequency that the "multiplier" attribute changes on the object.
- **bool MPxGeometryOverride::isIndexingDirty()**: At the render item level, this method can return true if the item requires an indexing update when an object attribute changes. In this example, the indexing per render item never changes; so, a "false" value is always returned.

The net result is that, if no streams nor indexing updates are required, then populateGeometry() may not be called.

6.3.1.6.1 Data Stream Update

In this example:

- A requirements structure (**MGeometryRequirements**) is passed in along with the render items that are to be updated (**MRenderItemList**), and the geometry container that is to be updated (**MGeometry**).
- The drawing of colored lines requires only one data stream (**MVertexBuffer**), which is a position stream.
- The requirements structure will provide a list of descriptions (**MVertexBufferDescriptor**) to be fulfilled in an **MVertexBufferDescriptionList**.
- Each description is checked for its semantic. In this case, the code will encounter a "position" semantic (*MGeometry::kPosition*).
- Thus, an MVertexBuffer needs to be created that matches the description and the data updated via the **MVertexBuffer::acquire()** method.
- The update of the data is done on CPU side and is plug-in dependent. When the CPU side data has been updated, it must be "committed" or transferred to the GPU via the **MVertexBuffer::commit()** method. The CPU side data is not required afterwards and hence can be discarded after the commit.

```
MHWRender::MVertexBuffer* verticesBuffer = NULL;
float* vertices = NULL;

const MHWRender::MVertexBufferDescriptorList& vertexBufferDescriptorList =
    requirements.vertexRequirements();

MHWRender::MVertexBufferDescriptor vertexBufferDescriptor;
for (int i = 0; i < vertexBufferDescriptorList.length(); ++i)
```

```
{
    // Both the wireframe render item (lines) and the shaded render item (triangles)
    // require a position stream only.
    if (vertexBufferDescriptorList.getDescriptor(i, vertexBufferDescriptor) &&
        vertexBufferDescriptor.semantic() == MHWRender::MGeometry::kPosition)
    {
        verticesBuffer = data.createVertexBuffer(vertexBufferDescriptor);
        vertices = (float*)verticesBuffer->acquire(soleCount+heelCount);
        break;
    }
}

// The postions of the heel and the sole are concatenated into a vertex buffer, to
// reduce the draw overhead due to the “small batch problem”. The merged buffer will
// be shared by the wireframe render item (lines) and the shaded item (triangles),
// to reduce memory usage and transfer overhead. Index buffers will specify which
// vertices are assembled for each render item.
for (int i = 0; i < soleCount+heelCount; ++i)
{
    if (i < heelCount)
    {
        int heelVtx = i;
        vertices[i*3]   = heel[heelVtx][0] * mMultiplier;
        vertices[i*3+1] = heel[heelVtx][1] * mMultiplier;
        vertices[i*3+2] = heel[heelVtx][2] * mMultiplier;
    }
    else
    {
        int soleVtx = i - heelCount;
        vertices[i*3]   = sole[soleVtx][0] * mMultiplier;
        vertices[i*3+1] = sole[soleVtx][1] * mMultiplier;
        vertices[i*3+2] = sole[soleVtx][2] * mMultiplier;
    }
}

// Commit: Transfer from CPU to GPU memory.
if (verticesBuffer && vertices)
{
    verticesBuffer->commit(vertices);
}
```

6.3.1.6.2

6.3.1.6.2 Index Stream Update

The updating for indexing is performed per render item. The basic logic behind this process is:

- Loop through the supplied render items.
- Create an index buffer as required, and get access to the raw CPU data using **MGeometry::createIndexBuffer**().
- Update the CPU data as required based on the indexing topology required.
- Like data streams, the CPU data for index streams must be committed using **MIndexBuffer::commit**().
- Associate the index stream with the render item using **MRenderItem::associateWithIndexBuffer**().

The following code snippet examines again only one render item whose name is stored in the variable *wireframeItemName*:

```
// Iterate through the render items for the override
for (int i=0; i < renderItems.length(); ++i)
{
    // Examine the current render item
    const MHWRender::MRenderItem* item = renderItems.itemAt(i);

    // Find the render item with the same name as the one that was created/updated
    // in updateRenderItems()
    if (item->name() == wireframeItemName)
    {
        // Acquire a new 32-bit index buffer.
        MHWRender::MIndexBuffer* indexBuffer =
            data.createIndexBuffer(MHWRender::MGeometry::kUnsignedInt32);

        // The indexing of the heel and the sole are concatenated into an index buffer.
        int numPrimitive = heelCount + soleCount - 2;
        int numIndex = numPrimitive * 2;

        // Gain access to the raw CPU data.
        unsigned int* indices = (unsigned int*)indexBuffer->acquire(numIndex);

        // Update the raw CPU data with indexing of the heel and the sole.
        for (int i = 0; i < numIndex; )
        {
            int startIndex = 0;
            int primitiveIndex = 0;
            if (i < (heelCount - 2) * 3)
            {
                startIndex = 0;
                primitiveIndex = i / 3;
            }
            else
            {
                startIndex = heelCount;
                primitiveIndex = i / 3 - heelCount + 2;
            }
            indices[i++] = startIndex;
            indices[i++] = startIndex + primitiveIndex + 1;
            indices[i++] = startIndex + primitiveIndex + 2;
        }

        // Commit: Transfer from CPU to GPU memory.
        indexBuffer->commit(indices);

        // Assign the index stream to the render item.
        item->associateWithIndexBuffer(indexBuffer);
    }
}
```

6.3.1.7 Analysis

The MPxGeometryOverride class is quite similar to VP2 internal classes used to support native DAG objects, like polygonal meshes and NURBS surfaces. By using this class, plug-ins can take advantage of all internal optimizations such as consolidation and hardware instancing.

6.3.1.7.1 Portability

- ✓ The implementation operates using the MRenderItem interface, and requires only one set of code which can be reused across all supported drawing APIs.
- ✓ Relatively simple to work with. When porting locators which require a stock shader, the required workload can be minimized by using the footPrintNode_GeometryOverride implementation as a template and adding only minimal changes, such as updates for custom geometry data or a different stock shader.

6.3.1.7.2 Scalability

- ✓ A well optimized implementation of MPxGeometryOverride provides predictably better performance over legacy viewport or other porting options. Here are general optimization guidelines as illustrated in the plug-in implementation.
    - Plug-ins should allow render items to use consolidation or hardware instancing by minimizing the number of unique shader instances used by the render items. For example, shader instance caches can be implemented to allow render items produced by various DAG objects to reuse the same shader instances.
    - When appropriate, the number of unique render items regarding categorization, display properties and shader etc. should be minimized by merging data/index streams explicitly for the same type of render items, to improve the drawing performance in case consolidation or hardware instancing cannot be used.
    - When appropriate, a single set of data streams should be shared among render items, with index streams used to assemble the geometries for associated render items. This reduces memory usage and transfer overhead.
- ✓ Multi-draw consolidation is the preferred performance optimization to improve the drawing performance for both matrix-animated and static objects, as long as it is supported. See *Section 3.6 Categorization & Consolidation* for information about its availability on various platforms. Note that, *it is incompatible with a* geometry *override which has UI drawables.* If scalability is important, avoid any use of MUIDrawManager.
- ✓ Hardware instancing is the alternative in case multi-draw consolidation is not supported. It is supported on a wide range of platforms and requires DAG objects to be instanced and “GPU Instancing” option to be enabled via Hardware Renderer 2.0 Settings.

6.3.1.7.3 Compatibility/Flexibility

- ✓ Drawing API agnostic.
- ✓ Picking is handled automatically, although it is possible to override selection via the refineSelectionPath() method.
- ✓ Participation in viewport draw modes, post effects, and advanced transparency algorithms can be specified on render items.
- ✓ Render items can be assigned with a stock shader, a shader translated from a Maya shading group (using MRenderItem::setShaderFromNode()), or a custom shader.

6.3.2 Porting Using a Subscene Override

This example uses an MPxSubSceneOverride to handle drawing for a locator. The example code listed below is taken from the *footPrintNode_SubSceneOverride* sample plug-in, in which an implementation of MPxSubSceneOverride is provided in **FootPrintSubSceneOverride**.

The basic logic for managing an MPxSubSceneOverride includes:

- Registering an association between the override and a dag object
- Maintaining a MSubSceneContainer, which is basically a collection of MRenderItems

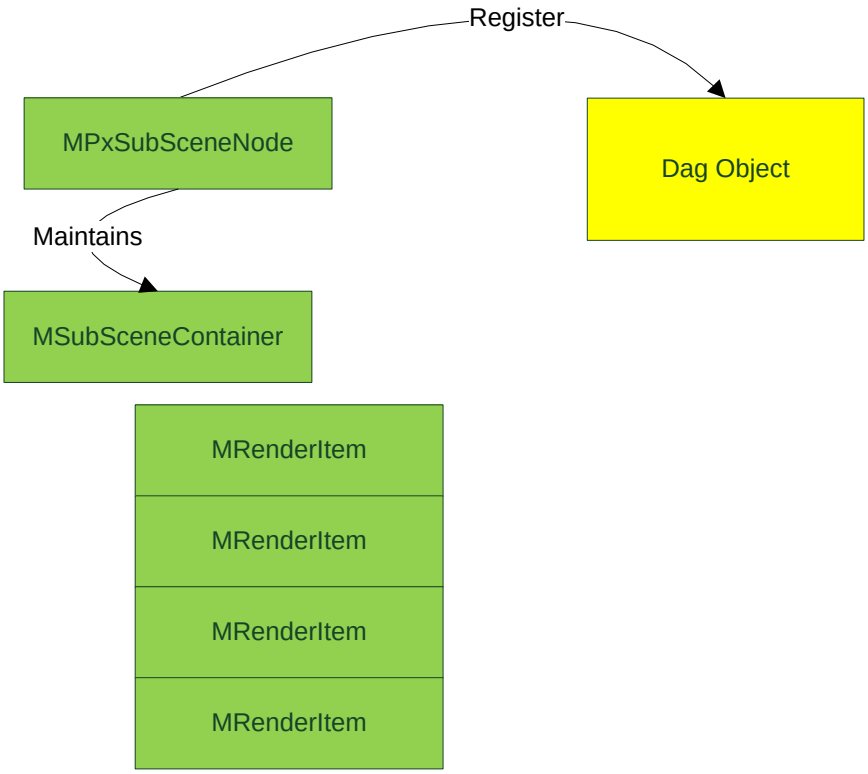


Figure 3: Simple interface connections shown for MPxSubSceneOverride. The override needs to be associated with an object, and the override needs to maintain a list of render items.

6.3.2.1 MPxSubSceneOverride Registration

The required classification string for subscene overrides must start with "**drawdb/subscene**" versus "drawdb/geometry". In *footPrintNode_SubSceneOverride* the full classification is "drawdb/subscene/footPrint_SubSceneOverride". **registerSubSceneOverrideCreator()** and **deregisterSubSceneOverrideCreator()** are the corresponding pair of registration / deregistration interfaces on **MDrawRegistry**.

The plug-in node itself requires a classification string, and the same string must be used when registering the subscene override. At plug-in initialization time, a registration id and a creator function must be provided along with the classification string.

```
static MString sDrawDbClassification("drawdb/subscene/footPrint_SubSceneOverride");
static MString sDrawRegistrantId("FootprintNode_SubSceneOverridePlugin");

plugin.registerNode(
    "footPrint_SubSceneOverride",
    footPrint::id,
    &footPrint::creator,
    &footPrint::initialize,
    MPxNode::kLocatorNode,
    &sDrawDbClassification); // VP2 drawdb classicification string

MHWRender::MDrawRegistry::registerSubSceneOverrideCreator(
    sDrawDbClassification, // Use the same drawdb classification string
    sDrawRegistrantId, // Registration id
    FootPrintSubSceneOverride::Creator); // Instance creator function
```

At plug-in unitialization time, a corresponding deregistration is performed.

```
MHWRender::MDrawRegistry::deregisterGeometryOverrideCreator(
    sDrawDbClassification, // Use same classification as registration
    sDrawRegistrantId); // Use same registration id
```

6.3.2.2 MPxSubSceneOverride Interfaces

The MSubSceneContainer can be thought of as the equivalent of the MRenderItemList, except that the container is optimized to manage and store a large number of render items. While MPxGeometryOverride is passed a render item list, MPxSubSceneOverride is passed a container.

The key overrides that must be implemented are:

- bool **requiresUpdate**(const MSubSceneContainer& container,
  const MFrameContext& frameContext) const = 0;
- void **update**(MSubSceneContainer& container,
  const MFrameContext& frameContext) = 0;

requiresUpdate() is called *every* refresh and indicates whether to call update(). This is a different update logic from that of geometry overrides, which are only called when a state has

changed internally that, requires a render item or geometry update, or an explicit request is made to indicate a change.

Note also that there are no specific update interfaces related to DG evaluation, render item or geometry update. It is up to the plug-in to determine when any of these are required.

The class **MSubSceneContainer** has utility methods that allow the plug-in to manage render items. Persistence of render items is determined by whether they reside within the container. Items may be arbitrarily added/removed or enabled/disabled within the update() call.

As this interface is meant to handle “scene” level objects, it is also responsible for handling all render items for all instances (transform) of the associated DAG object. For example, if different instances require different display properties, then the update mechanism needs to manage per instance render items, or instance data for render items. There are a set of instancing interfaces which can be used to vary per instance information and allow explicit control of hardware instancing. Beside per-instance transform, addition per-instance data may be bound to specified parameters on the shader.

- MStatus **setInstanceTransformArray**(MRenderItem& renderItem,
  const MMatrixArray& matrixArray);
- MStatus **setExtraInstanceData**(MRenderItem& renderItem,
  const MString& parameterName,
  const MFloatArray& data);

If the associated DAG object is not instanced, render items in the container can use “subscene consolidation”. However, it is in nature a simplified version of traditional static consolidation and doesn’t consider the spatial proximity of render items. Thus, it might be easier to trigger reconsolidation and thus performance scalability might be unstable.

- void **setWantSubSceneConsolidation**(bool state);

Note: *To determine whether hardware instancing or consolidation is used, enable the debug tracing of VP2 render pipeline.*

If “incremental update” is required, it is possible to override the following method:

- bool **furtherUpdateRequired**(const MFrameContext& frameContext);

This can indicate that, during a single update, there was not enough time to complete the action, and to call back to the override again on the next idle event.

6.3.2.3 Container Update

The actions listed below are very specific to the individual plug-in. In the example footPrintNode_SubSceneOverride, shaders are updated, geometry is rebuilt, per-instance data are extracted, and render items in the container are updated as required.

A rough series of steps is outlined in the diagram:

1) Define render items and shaders for the render items.
2) Determine the required data streams and the vertex descriptors create them.
3) Determine the indexing per render item and create index streams.
4) Set the indexing and data streams on the appropriate render items.
5) Determine per instance data required for the render items.

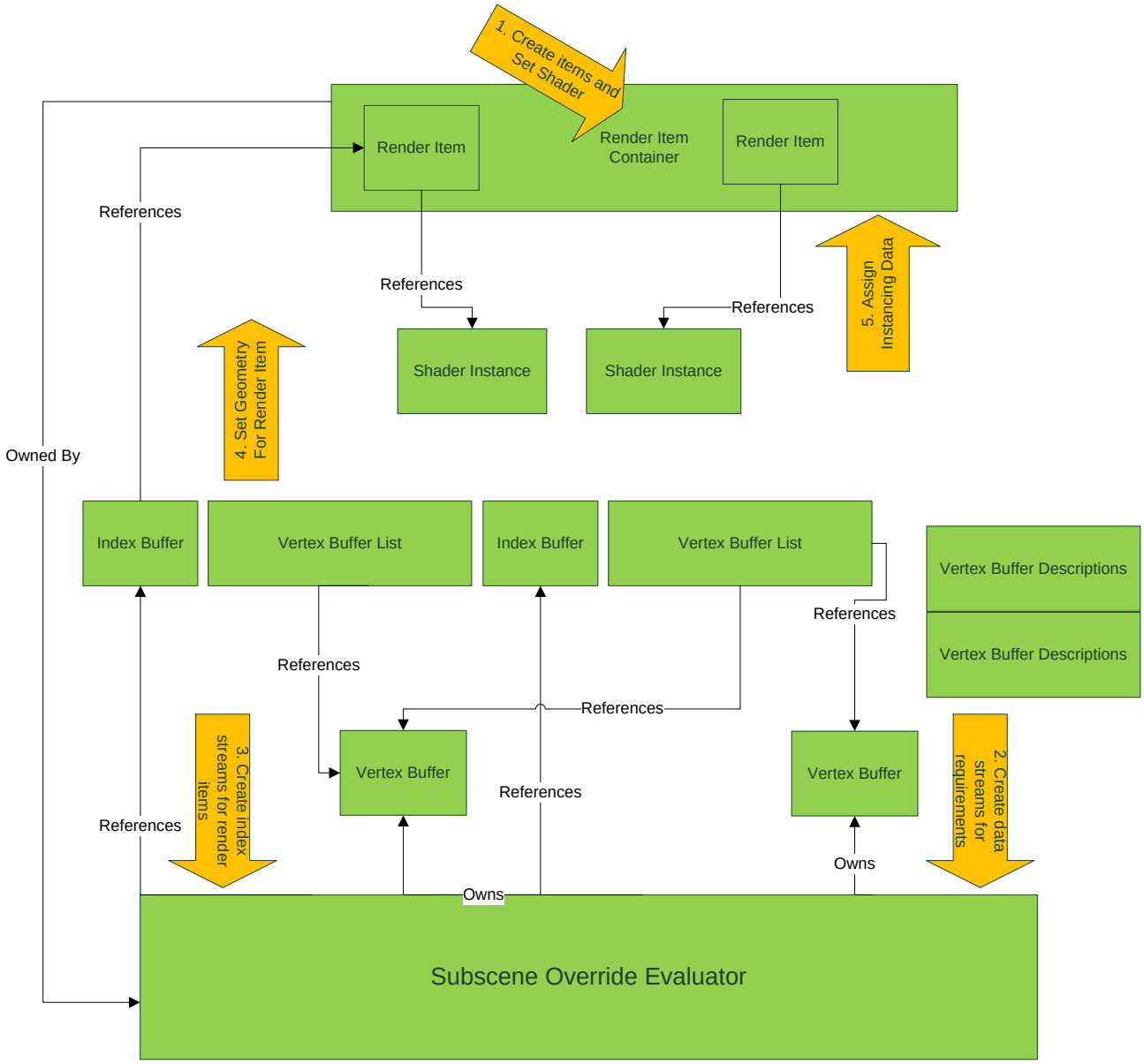


Figure 4: The basic "process" to perform a subscene update. The vertex buffer descriptions are transient and are determined by the plug-in. The data and index streams are owned by the override, as are the shader instances. The render items are owned by the container, which is owned by the override. The vertex buffer lists used to update geometry on render items are also transient. Any per instance data would be owned by the override.

6.3.2.3.1 Shaders

The same as the MPxGeometryOverride example, the plug-in maintains a mini shader instance cache to allow render items to reuse shader instances whenever possible, so that render items can use sub-scene consolidation when the associated DAG object is not instanced.

6.3.2.3.2 Data and Index Streams

To allow reuse, all data (MVertexBuffer) and index streams (MIndexBuffer) are owned by the override. It is up to the plug-in to determine whether to create new streams or to reuse existing streams.

In the footPrintNode_SubSceneOverride implementation, the rebuildGeometryBuffers() routine manages this data and rebuilds all buffers.

As there is no inherent support for assigned shaders, all shaders are created by the override and it is up to the override to determine the total geometry requirements for data streams. There is, in fact, no explicit geometry requirements structure. Instead, the override will create the data streams it requires, and directly set a reference to the data streams on the appropriate render items.

In the example, positions for both wireframe and shaded render item are concatenated into a vertex buffer. The vertex description is determined by the override, and new buffer is explicitly created.

```
// VB for positions. We concatenate heel and sole positions into a single VB
// shared by the wireframe item and the shaded item. The index buffer will
// will decide which positions should be used for each render item.
const MHWRender::MVertexBufferDescriptor vbDesc("",
    MHWRender::MGeometry::kPosition, MHWRender::MGeometry::kFloat, 3);
fPositionBuffer = new MHWRender::MVertexBuffer(vbDesc);
if (fPositionBuffer)
{
    // Allocate CPU memory for writing position data.
    float* positions = (float*)fPositionBuffer->acquire(soleCount+heelCount, true);

    // Write position data and commit
    …
}
```

Index streams are maintained by the override. Two index streams are created to handle the drawing of wireframe and shaded triangles.

```
fWireIndexBuffer = new MHWRender::MIndexBuffer(MHWRender::MGeometry::kUnsignedInt32);
if (fWireIndexBuffer)
{
    // Allocate CPU memory for writing indexing data.
    int numPrimitive = heelCount + soleCount - 2;
    int numIndex = numPrimitive * 2;
    unsigned int* indices = (unsigned int*)fWireIndexBuffer->acquire(numIndex, true);

    // Write indexing data and commit
```

```
    …
}

fShadedIndexBuffer = new MHWRender::MIndexBuffer(MHWRender::MGeometry::kUnsignedInt32);
if (fShadedIndexBuffer)
{
    // Allocate CPU memory for writing indexing data.
    int numPrimitive = heelCount + soleCount - 4;
    int numIndex = numPrimitive * 3;
    unsigned int* indices = (unsigned int*)fShadedIndexBuffer->acquire(numIndex, true);

    // Write indexing data and commit
    …
}
```

The code to fill in the buffer is not shown. The commit() still needs to occur for all data and index streams.

6.3.2.3.3 Render Items

The example demonstrates that, to update render items, a name search can be performed to see if a render item already exists. If not, it can be created. An alternative method would be to override the properties (including geometry) on an existing render item.

As with the examples shown for MPxGeometryOverride, each item needs an item type, a draw mode, and an indexing topology.

```
// The wireframe render item
MHWRender::MRenderItem* wireItem = container.find(wireframeItemName);
if (!wireItem)
{
    wireItem = MHWRender::MRenderItem::Create(wireframeItemName,
        MHWRender::MRenderItem::DecorationItem,
        MHWRender::MGeometry::kLines);
    wireItem->setDrawMode(MHWRender::MGeometry::kWireframe);
    container.add(wireItem);
    itemsChanged = true;
}

// The shaded render item
MHWRender::MRenderItem* shadedItem = container.find(shadedItemName);
if (!shadedItem)
{
    shadedItem = MHWRender::MRenderItem::Create(shadedItemName,
        MHWRender::MRenderItem::DecorationItem,
        MHWRender::MGeometry::kTriangles);
    shadedItem->setDrawMode((MHWRender::MGeometry::DrawMode)
        (MHWRender::MGeometry::kShaded|MHWRender::MGeometry::kTextured));
    container.add(shadedItem);
    itemsChanged = true;
}
```

6.3.2.3.4 Setting Data on Render Items

To explicitly set data and index streams for a render item the **setGeometryForRenderItem**() method can be used. An optional object space bounding box may also be specified.

The sample code below demonstrates the update of wireframe and shaded render items. Each adds the appropriate data streams to an array, and then set that array along with the appropriate index stream reference and bounding box.

The data is still owned by the override, but the render items now reference the data.

```
footPrint* fp = dynamic_cast<footPrint*>(node.userNode());
MBoundingBox *bounds = fp ? new MBoundingBox(fp->boundingBox()) : NULL;

MHWRender::MVertexBufferArray vertexBuffers;
vertexBuffers.addBuffer("positions", fPositionBuffer);

setGeometryForRenderItem(*wireItem, vertexBuffers, *fWireIndexBuffer, bounds);
setGeometryForRenderItem(*shadedItem, vertexBuffers, *fShadedIndexBuffer, bounds);

if (bounds) delete bounds;
```

6.3.2.3.5 Hardware Instancing

MPxSubSceneOverride allows for explicit hardware instancing control which is independent of the global “GPU Instancing” option in Hardware Renderer 2.0 Settings. This means that all instances can be packed into a single render item and drawn by a special draw call, which allows for the shader and per-instance data to be specified once, thus increasing performance.

Hardware instancing is supported on a wide range of platforms, but in case it is not supported, then the render item will be drawn multiple times (versus one time), and refreshing the view will naturally be slower.

One way of implementing hardware instancing is to supply a list of object-to-world transforms for a render item via the **MPxSubsceneOverride::setInstanceTransformArray**() method. Each instance will be drawn in world space based on the matrices provided. After per-instance transform is set, it is also possible to bind other uniform shader parameters with per instance data using **MPxSubsceneOverride::setExtraInstanceData**(). In this case, per-instance data can be used for a given shader parameter on an MShaderInstance per render item instance. In this example, there is a “solidColor” parameter on the 3d solid shader assigned to render items, the color values stored in the color array will be used.

Note: *Per-instance data can be bound to uniform parameters of a shader which is created from a shader fragment or fragment graph. Shaders created from an effects buffer or file cannot use this feature because Maya cannot insert a pass-through shader fragment that converts uniform parameters to varying parameters, i.e. streams to fill with per-instance data.*

```
unsigned int numInstances = fInstanceDagPaths.length(); // Number of all instances
unsigned int numVisibleInstances = 0; // Number of visible instances.
```

```
unsigned int colorChannels = 4; // RGBA

MMatrixArray matrices(numInstances); // Array to store per-instance transform
MFloatArray colors(numInstances * colorChannels); // Array to store per-instance color

for (unsigned int i=0; i<numInstances; i++)
{
    const MDagPath& path = fInstanceDagPaths[i];
    if (path.isValid() && path.isVisible())
    {
        // Get the inclusive object-to-world matrix of the DAG path.
        matrices[numVisibleInstances] = path.inclusiveMatrix();

        // Get the appropriate color of the instance regarding the selection status
        // and attribute settings of the DAG path.
        MColor color = MHWRender::MGeometryUtilities::wireframeColor(path);
        colors[numVisibleInstances*colorChannels]   = color.r;
        colors[numVisibleInstances*colorChannels+1] = color.g;
        colors[numVisibleInstances*colorChannels+2] = color.b;
        colors[numVisibleInstances*colorChannels+3] = color.a;

        numVisibleInstances++;
    }
}

// Shrink to fit.
matrices.setLength(numVisibleInstances);
colors.setLength(numVisibleInstances * colorChannels);

// Conslidation and hardware instancing are incompatible with each other. If
// the associated DAG object is instanced, we need to reset the flag so that
// hardware instancing can be used.
wireItem->setWantSubSceneConsolidation(false);
shadedItem->setWantSubSceneConsolidation(false);

// Set up instance copies of render items to take advantage of hardware instancing.
// It is faster than creating render items for each DAG instance, especially for a
// large number of instances.
//
// Note that these methods should be called after geometry and shader are set,
// otherwise it would fail.
//
setInstanceTransformArray(*wireItem, matrices);
setInstanceTransformArray(*shadedItem, matrices);

// Bind other uniform shader parameters with per-instance data.
//
setExtraInstanceData(*wireItem, "solidColor", colors);
setExtraInstanceData(*shadedItem, "solidColor", colors);
```

6.3.2.4 Additional UI

MUIDrawManager can be used to add in additional UI drawables. The interfaces are the same as for geometry and draw overrides, except for one key interface: **areUIDrawablesDirty**(). This interface allows for explicit control of persistence of the UI drawables between frames, by

allowing for the drawables to remain cached while the override exists (while the associated DAG object exists).

```
bool FootPrintSubSceneOverride::areUIDrawablesDirty() const
{
    // The flag will be set when matrix transformation of any instances
    // are changed, and reset after UI drawables are recreated.
    return fAreUIDrawablesDirty;
}
```

If the associated DAG object is instanced, UI drawables for all its instances should be created in the addUIDrawables() method. In this example, each text should be drawn at the origin in the object space of its belonging instance but MUIDrawManager assumes to use the object space of the first DAG instance, so we should transform coordinates between two object spaces, from each instance to the first instance.

```
void FootPrintSubSceneOverride::addUIDrawables(
    MHWRender::MUIDrawManager& drawManager,
    const MHWRender::MFrameContext& frameContext)
{
    MPoint pos( 0.0, 0.0, 0.0 );
    MColor textColor( 0.1f, 0.8f, 0.8f, 1.0f );
    MString text( "Footprint" );

    drawManager.beginDrawable();
    drawManager.setColor( textColor );
    drawManager.setFontSize( MHWRender::MUIDrawManager::kSmallFontSize );

    // The instance info cache has stored object-to-world matrix of each instance
    // in the update() method called before the addUIDrawables() method.
    MMatrix worldInverse0 = fInstanceInfoCache[0].fMatrix.inverse();
    for (auto it = fInstanceInfoCache.begin(); it != fInstanceInfoCache.end(); it++)
    {
        drawManager.text((pos * it->second.fMatrix) * worldInverse0,
                         text, MHWRender::MUIDrawManager::kCenter);
    }

    drawManager.endDrawable();

    // Reset the flag after UI drawables are updated.
    fAreUIDrawablesDirty = false;
}
```

Note that, selection picking via UI drawables is supported only when the associated DAG object is not instanced. When the object is instanced, only the first instance can be picked; for complete picking support, MRenderItem should be used instead. For detailed example and analysis about UI drawables, see Section 6.3.4.

6.3.2.5 Analysis

Although the MPxSubSceneOverride class is primarily designed for “scene-cache” style nodes that manage a large set of objects, it can be used for any type of DAG object, especially you

can use this class for a DAG object which has multiple per-instance data and needs hardware instancing to improve performance.

6.3.2.5.1 Portability

- ✓ Operates using the MRenderItem interface and requires only one set of code which can be reused across all supported drawing APIs.
- ✓ Gains direct access and explicit control of hardware instancing.
- × Relatively more code due to the amount of the control given to the implementation when compared to a geometry override.
    - Required to handle the update logic explicitly. It is totally up to the implementation to determine whether updates are needed.
    - Required to manage all render items and UI drawables to draw all instances of the associated DAG object.

6.3.2.5.2 Scalability

- ✓ When the associated DAG shape is instanced with additional per-instance data streams, a subscene override can explicitly control hardware instancing by setExtraInstanceData(). In this case, subscene override may outperform geometry override.
- ✓ When the associated DAG object is not instanced, it is possible to allow render items to use “subscene consolidation” by MRenderItem::setWantSubSceneConsolidation().

6.3.2.5.3 Compatibility/Flexibility

- ✓ Drawing API agnostic.
- ✓ Picking is handled automatically, while it is possible to override selection via getInstancedSelectionPath() or getSelectionPath().
- ✓ Participation in viewport display modes, post effects and advanced transparency algorithms can be specified on render items.
- ✓ Render items can be assigned with a stock shader, a shader translated from a Maya shading group (via the setShaderFromNode() method), or a custom shader.

6.3.3 **Porting Using UI Draw Manager**

MUIDrawManager is available to queue transient UI drawing on various override classes. It is possible to queue transient UI as well as maintain persistent render items on an MPxGeometryOverride or an MPxSubSceneOverride.

The following example demonstrates the use of a MUIDrawManager with an MPxDrawOverride where persistent items are not required. The basic configuration is shown on the left below:

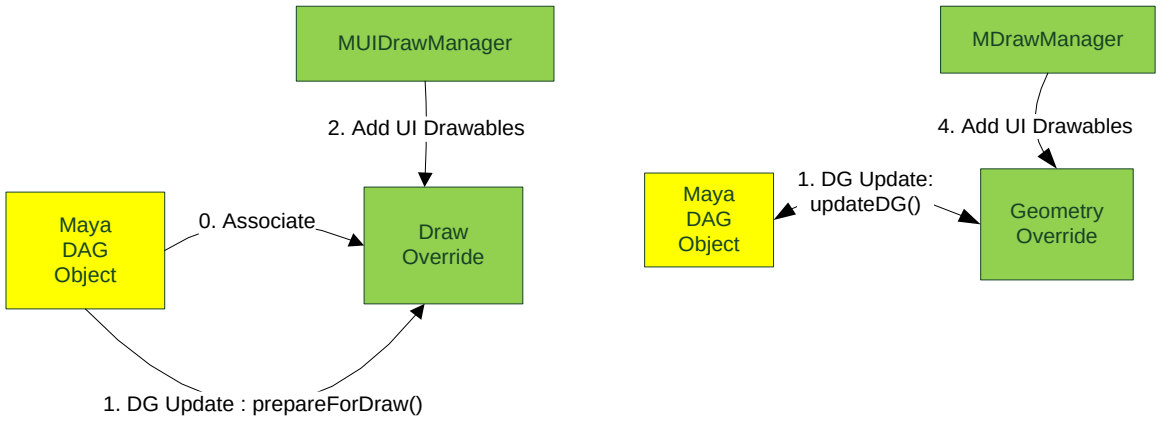


Figure 5: A draw override (left) is associated with a DAG object. A data preparation phase occurs and a MUIDrawManager is used to queue simple drawing. The same basic UI draw manager interface exists for geometry overrides (right) albeit with a different node evaluation interface and association. UI drawable update occurs last.

The code examples shown below are from the *footPrintNode* SDK example. This developer kit example is available in a C++ implementation version and a Python API 2.0 implementation version. The latter is shown in the code examples below. For more information, see the Maya Developer Kit.

Note: For all examples that have both a C++ implementation version and a Python API 2.0 implementation version, the latter can be found in the \plug-ins\scripted directory of the Maya Developer Kit. All Python API 2.0 examples are named with the prefix *py*.

As noted under the *Performance Considerations* section in Part I of the guide this is only recommended for very simplistic drawing.

6.3.3.1 MPxDrawOverride Registration

As with a geometry override, a draw override must set up an association with a Maya node. In this example, the footprint node adds in a drawdb classification as the last argument to **registerNode**(). That same classification must be used when using the **MDrawRegistry::registerDrawOverriderCreator**() method. As with geometry override registration, draw override registration also requires a unique registration identifier and a "creator" function to create an instance of an MPxDrawOverride.

```
import maya.api.OpenMaya as om
import maya.api.OpenMayaRender as omr

plugin.registerNode("footPrint", footPrint.id, footPrint.creator, footPrint.initialize,
om.MPxNode.kLocatorNode, footPrint.drawDbClassification)

omr.MDrawRegistry.registerDrawOverrideCreator(footPrint.drawDbClassification,
footPrint.drawRegistrantId, footPrintDrawOverride.creator)
```

The corresponding deregistration would exist within the uninitializePlugin() method.

```
omr.MDrawRegistry.deregisterDrawOverrideCreator(footPrint.drawDbClassification,
footPrint.drawRegistrantId)
```

6.3.3.2 MUIDrawManager Usage

Instead of using the VP1 locator interface:

- **MPxLocatorNode::draw**(M3dView & view, const MDagPath & path, M3dView::DisplayStyle style, M3dView::DisplayStatus status);

it is possible to use the addUIDrawables() method on the attached MPxDrawOverride to draw:

- **MPxDrawOverride:: addUIDrawables**(const MDagPath& objPath, MHWRender::MUIDrawManager& drawManager, const MHWRender::MFrameContext& frameContext, const MUserData* data);

Note that there is no longer any access to an M3dView, a display style or display status. The required drawing information should instead be extracted from the supplied MFrameContext or via the MGeometryUtilitities class.

Any custom data can be accessed via the **MUserData** instance that is passed in to addUIDrawables(). This includes any data evaluated on the node itself. For this example, the custom data (*footPrintData*) is declared to hold additional color, and geometry information. No node references are kept here.

```
class footPrintData(om.MUserData):
    def __init__(self):
        om.MUserData.__init__(self, False) ## don't delete after draw
        self.fColor = om.MColor()
        self.fLineList = om.MPointArray()
        self.fTriangleList = om.MPointArray()
```

For VP1 the draw and node evaluation are both done within draw:

```
def draw(self, view, path, style, status):
    ## Get the size
    ##
    thisNode = self.thisMObject()
    plug = om.MPlug( thisNode, footPrint.size )
    sizeVal = plug.asMDistance()
    multiplier = sizeVal.asCentimeters()
```

For VP2, DG evaluation is separated out and performed by overriding the **MPxDrawOverride::prepareForDraw**() method. When this method is invoked, a copy of any node data to be used at draw time is copied to the MUserData -- instead of a keeping a reference. This is to decouple any dependency on node data being overridden before the user data contents are used for drawing.

In the example, node data extracted out and cache in a custom MUserData instance (footPrintData). The snippet below focuses on showing the data update required for drawing the wireframe of “footPrint” (which is stored as the fLineList data member of the footPrintData instance).

```
def getMultiplier(self, objPath):
    ## Retrieve value of the size attribute from the node
    footprintNode = objPath.node()
    plug = om.MPlug(footprintNode, footPrint.size)
    if not plug.isNull:
        sizeVal = plug.asMDistance()
        return sizeVal.asCentimeters()

def prepareForDraw(self, objPath, cameraPath, frameContext, oldData):
    ## Retrieve data cache (create if does not exist)
    ## Need to make sure that this is an instance of our data (footPrintData)
    data = oldData
    if not isinstance(data, footPrintData):
        data = footPrintData()

    ## This data is predefined globally to be used for both VP1 and VP2
    global soleCount, sole
    global heelCount, heel

    ## Compute data and cache it on the MUserData. This includes the scaling
    ## factor, the scaled data for the "footPrint" line list as well as the
    ## color. The "sole" portion and the "heel" portion are concatenated into
    ## a line list, to reduce the overhead due to the number of UI drawables
    ## or avoid internal repetitive batching for the same set of UI drawables.
    fMultiplier = self.getMultiplier(objPath)

    data.fLineList.clear()

    for i in range(soleCount-1):
        data.fLineList.append( om.MPoint(sole[i][0] * fMultiplier,
                                         sole[i][1] * fMultiplier,
                                         sole[i][2] * fMultiplier) )
        data.fLineList.append( om.MPoint(sole[i+1][0] * fMultiplier,
                                         sole[i+1][1] * fMultiplier,
                                         sole[i+1][2] * fMultiplier) )

    for i in range(heelCount-1):
        data.fLineList.append( om.MPoint(heel[i][0] * fMultiplier,
                                         heel[i][1] * fMultiplier,
                                         heel[i][2] * fMultiplier) )
        data.fLineList.append( om.MPoint(heel[i+1][0] * fMultiplier,
                                         heel[i+1][1] * fMultiplier,
                                         heel[i+1][2] * fMultiplier) )

    data.fColor = omr.MGeometryUtilities.wireframeColor(objPath)

    return data
```

The actual drawing is fairly straightforward. The following snippet shows the access to the frame context that determines the items that are to be drawn. In particular, the display style is being queried to determine whether to draw a filled drawable:

```
def addUIDrawables(self, objPath, drawManager, frameContext, data):
    locatordata = data
    if not isinstance(locatordata, footPrintData):
        return
```

```
    drawManager.beginDrawable()

    ## Draw the foot print solid/wireframe
    drawManager.setColor( locatordata.fColor )
    drawManager.setPriority(5)

    ## Check the display style to determine if filled drawing is required
    if (frameContext.getDisplayStyle() & omr.MFrameContext.kGouraudShaded):
        drawManager.mesh(omr.MGeometry.kTriangles, locatordata.fTriangleList)

    drawManager.mesh(omr.MUIDrawManager.kLines, locatordata.fLineList)

    ## Draw a text "Footprint". Font and text drawing are only available via
    ## MUIDrawManager class.
    pos = om.MPoint( 0.0, 0.0, 0.0 )  ## Position of the text
    textColor = om.MColor( (0.1, 0.8, 0.8, 1.0) )  ## Text color

    drawManager.setColor( textColor )
    drawManager.setFontSize( omr.MUIDrawManager.kSmallFontSize )
    drawManager.text(pos, "Footprint", omr.MUIDrawManager.kCenter )

    drawManager.endDrawable()
```

Note that what is done here for transient drawables is similar to what would be done for persistent render items but is done **on every refresh by default**. This includes determining the display style (versus adding a render item per style), re-specifying the data as well as the shading parameters.

To avoid the above repetitive overhead, the MPxDrawOverride constructor argument **isAlwaysDirty** should be set to false. In this case prepareForDraw() and addUIDrawables() will only be called when the DAG object t is marked dirty via DG evaluation or dirty messages. Additional callbacks might need to be added to explicitly mark the node as being dirty using MRenderer::setGeometryDrawDirty() for certain cases, e.g. on viewport mode changes.

```
class footPrintDrawOverride(omr.MPxDrawOverride):
    @staticmethod
    def creator(obj):
        return footPrintDrawOverride(obj)

    ## By setting isAlwaysDirty to false in MPxDrawOverride constructor, the
    ## draw override will be updated only when the node is marked dirty via
    ## DG evaluation or dirty propagation.
    def __init__(self, obj):
        omr.MPxDrawOverride.__init__(self, obj, None, False)
```

Note also instead of being assigned with a stub method, the constructor argument **callback** is set to **None**. Internally a proxy render item is created for each associated DAG object and attached with the user-defined draw callback. When the callback is set to null, the proxy render item can be disabled and filtered out earlier in VP2 render pipeline to reduce traversal overhead.

6.3.3.3 Analysis

The MUIDrawManager class provides a straight forward way to draw basic UI elements when porting simple DAG objects to Viewport 2.0. The decision as to using this class is mostly based on a choice for simplicity while flexibility and scalability is not required.

As noted in API documentation, MUIDrawManager is not designed for accessing arbitrary times or arbitrary places. To get the access, plug-ins must associate a custom locator with an implementation of MPxDrawOverride/ MPxGeometryOverride/MPxSubSceneOverride and override the addUIDrawables() method. But for discussion purpose, it's better to treat MUIDrawManager as an independent porting choice.

6.3.3.3.1 Portability

MUIDrawManager avoids the complexity of using MRenderItem. Each UI drawable uses the appropriate geometry, shader and matrix transformation automatically based on the type. For example, text has an appropriate "text shader" and 2D drawing has an appropriate "2D" matrix transformation. This knowledge is hidden from the user for simplicity.

- ✓ Suitable for simple UI, where simple means a small amount of UI, or a when a small number of objects are drawing the UI.
- ✓ Supports text, icons, lines, circles, basic 2D and 3D primitives, and arbitrary meshes.
- ✓ Easier to port legacy draw code as it looks like fixed function drawing.
- ✓ Requires only one set of code which can be reused across all supported drawing APIs.

6.3.3.3.2 Scalability

- ✓ As noted in the API documentation for MUIDrawManager, VP2 may batch the same type of UI drawables which are created by the same call to the addUIDrawables() method. To determine whether UI drawables are batched, enable the debug tracing of VP2 render pipeline.
    - If the associated DAG object is instanced, a subscene override should create UI drawables for all instances in the addUIDrawables() method. This allows VP2 batching optimization to be used, but it is up to the plug-in to transform each UI drawable between object spaces of two DAG instances. For cases where the cost of matrix transformation outweighs the batching optimization, e.g. complex meshes, the MRenderItem interface should be used instead.
    - Plug-ins can consolidate the geometries and reduce the number of UI drawables in their code to avoid the internal repetitive batching for the same set of UI drawables.

- × UI drawables are transient render items in nature, so they cannot use consolidation or hardware instancing, and they may not scale well due to the potential cost of reallocation. To avoid any unnecessary overhead due to the frequency of recreating UI drawables:
    - A draw override should be constructed with the MPxDrawOverride constructor argument isAlwaysDirty set to false whenever possible.
    - A subscene override should override the areUIDrawablesDirty() method to return false when UI drawables doesn't need to be updated.

6.3.3.3.3 Compatibility/Flexibility

- ✓ Drawing API agnostic.
- ✓ *Selectability* of UI drawables can be specified in the **beginDrawable**() method. If a UI drawable is set selectable, picking is handled automatically except the following case.
    - As noted in Section 6.3.2.4, selection picking via UI drawables in a subscene override is not supported, when the associated DAG object is instanced.
- × Only a fixed set of shading options and single-textured drawing are provided.
- × The geometry attributes that can be specified are fixed.
- × No inherent concept of participating based on viewport display modes. Requires the implementation to track these modes manually.
- × No override options for participating in post effects and advanced transparency algorithms.

6.3.4 Porting Using a Draw Override

This example uses an MPxDrawOverride to access low-level graphics APIs, e.g. OpenGL Core Profile and DirectX 11, for drawing a locator. The example code listed in this section is taken from the raw*footPrintNode* sample plug-in, where an implementation of MPxDrawOverride is provided by **RawFootPrintDrawOverride**.

6.3.4.1 MPxDrawOverride (Revisited)

The required classification string for draw overrides must start with "**drawdb/geometry**". In raw*footPrintNode* the full classification is "drawdb/geometry/rawfootPrint". **registerDrawOverrideCreator()** and **deregisterDrawOverrideCreator()** on **MDrawRegistry** are the corresponding interfaces to associate a Maya DAG object with a draw overrides.

The whole configuration for a draw override appears as follows. In addition to the MUIDrawManager interface which queues UI drawables at DG update time, a proxy render item is maintained internally to allow user-defined callback to be invoked at draw time. The callback is the only place where low-level graphics APIs can be accessed. Note that it should be set to null when only the MUIDrawManager interface is used, as demonstrated in Section 6.3.3,

because VP2 may attempt to disable and filter out the proxy render item earlier in the rendering pipeline to skip unnecessary traversal.

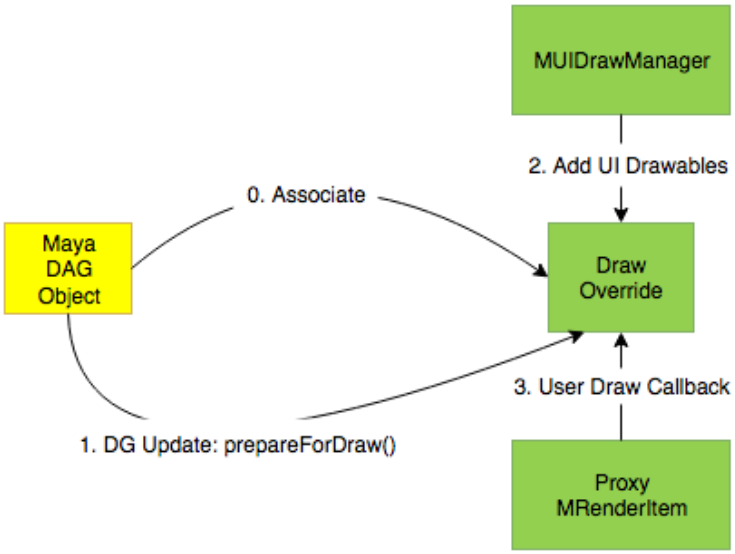


Figure 6: At DG update time user data is prepared and UI drawables are queued. At draw time callback is invoked.

6.3.4.2 MPxDrawOverride Interfaces

By default, a draw override is always updated via prepareForDraw() and addUIDrawables() without checking the dirty state of the DAG object. To avoid unnecessary overhead, the constructor argument **isAlwaysDirty** should be set to false, then the update methods will only be called when the DAG object is marked dirty via DG evaluation or dirty messages.

- **MPxDrawOverride**(const MObject& obj,
  GeometryDrawOverrideCb callback, bool **isAlwaysDirty** = true);

Object-space bounding box and world transformation are updated at an early pipeline phase. The information will then be used for visibility test; if the object is all outside the view frustum, it will stop traversing later phases and thus the update and draw methods will not be called.

- bool **isBounded**(const MDagPath& obj, const MDagPath& camera) const;
- MBoundingBox **boundingBox**(const MDagPath& obj, const MDagPath& camera) const;
- MMatrix **transform**(const MDagPath& obj, const MDagPath& camera) const;

After visibility test, the prepareForDraw() method will be called for “dirty” objects during DG update phase, to pull from Maya any data that will be required when drawing. Display properties, including participation in post effects and transparency, can also be updated here and will then be queried afterwards.

- MUserData* **prepareForDraw**(const MDagPath& obj, const MDagPath& camera,
  const MFrameContext& frameContext, MUserData* oldData) = 0;
- bool **excludedFromPostEffects**() const;
- bool **isTransparent**() const;

At draw time, the user-defined callback may be invoked multiple times, once for each render pass in which the object should participate as per its display properties.

- typedef void (***GeometryDrawOverrideCb**)(const **MDrawContext**&, const **MUserData***);

At selection time, the following method is called to determine whether a user-customized selection behavior or the VP2 default selection behavior is needed.

- bool **wantUserSelection**() const;

If it is overridden to return true, the following method is called to determine the selection status.

- bool **userSelect**(**MSelectionInfo**& selectInfo,
  const **MDrawContext**& context,
  MPoint& hitPoint,
  const **MUserData*** data);

The method should return true if the object should be selected regarding the provided information. In this case, **hitPoint** may be used by VP2 to sort selected objects for correct selection result based on viewport visibility. It uses the origin of the object's local frame in world space as the default value, but can be set to a precise intersection position in world space if implemented by the plug-in.

6.3.4.3 MPxDrawOverride Usage

As with the MUIDrawManager example, the draw override constructor argument **isAlwaysDirty** is set to false, to avoid repetitive performance overhead of calling the update methods. The draw override can still be updated when the associated DAG object is marked as being dirty via DG evaluation or dirty messages, however, for global state changes which don't trigger dirty propagation, e.g. switching display appearance of a model editor, additional callbacks should be registered and the MRenderer::**setGeometryDrawDirty**() method should be called in these callbacks to explicitly mark the object as being dirty.

```
class RawFootPrintDrawOverride : public MHWRender::MPxDrawOverride
{
public:
    RawFootPrintDrawOverride(const MObject& obj)
    : MHWRender::MPxDrawOverride(obj, RawFootPrintDrawOverride::draw, false)
    , fRawFootPrint(obj)
    {
        fModelEditorChangedCbId = MEventMessage::addEventCallback(
            "modelEditorChanged", OnModelEditorChanged, this);
    }

    ~RawFootPrintDrawOverride() override
    {
        if (fModelEditorChangedCbId != 0)
        {
            MMessage::removeCallback(fModelEditorChangedCbId);
            fModelEditorChangedCbId = 0;
        }
```

```
    }

    void OnModelEditorChanged(void *clientData)
    {
        RawFootPrintDrawOverride *ovr =
            static_cast<RawFootPrintDrawOverride*>(clientData);
        if (ovr) MHWRender::MRenderer::setGeometryDrawDirty(ovr->fRawFootPrint);
    }
}
```

In the prepareForDraw() method, DG evaluation is performed. Any required information can be pulled from Maya DG and stored in a user data structure. The user data will be cached internally and passed to the draw callback as it is invalid to trigger DG evaluation at draw time. The implementation also determines display properties which will later be returned by the excludedFromPostEffects() and isTransparent() method.

```
MUserData* RawFootPrintDrawOverride::prepareForDraw(
    const MDagPath& objPath,
    const MDagPath& cameraPath,
    const MHWRender::MFrameContext& frameContext,
    MUserData* oldData)
{
    // Update the display properties that will then be queried afterwards.
    float transparency = getTransparency(objPath);
    mIsTransparent = (transparency < 1.0);
    mExcludedFromPostEffects = (frameContext.getDisplayStyle() &
        (MHWRender::MFrameContext::kGouraudShaded |
         MHWRender::MFrameContext::kFlatShaded) == 0);

    // Retrieve the old user data if any, create otherwise.
    RawFootPrintData* data = dynamic_cast<RawFootPrintData*>(oldData);
    if (!data) data = new RawFootPrintData();

    // Store any information required by the draw callback in the user data.
    // DG evaluation should only be triggered in this method.
    if (MHWRender::MGeometryUtilities::displayStatus(objPath) == MHWRender::kActive)
    {
        MColor color = MHWRender::MGeometryUtilities::wireframeColor(objPath);
        data->fColor[0] = color.r;
        data->fColor[1] = color.g;
        data->fColor[2] = color.b;
    }
    else
    {
        data->fColor[0] = 0.6f;
        data->fColor[1] = 0.6f;
        data->fColor[2] = 0.6f;
    }

    data->fColor[3] = transparency;

    return data;
}

bool RawFootPrintDrawOverride::excludedFromPostEffects() const
{
```

```
    return mExcludedFromPostEffects;
}

bool RawFootPrintDrawOverride::isTransparent() const
{
    return mIsTransparent;
}
```

Depending on visibility test, the user-defined callback may or may not be invoked at draw time. An **MDrawContext** and optionally an **MUserData** will be supplied for the callback. In comparison to the VP1 draw interface on MPxLocatorNode:

- **MPxLocatorNode::draw**(M3dView & view,
  const MDagPath & path, M3dView::DisplayStyle style, M3dView::DisplayStatus status);

There is no longer any access to an M3dView, a display style or display status. Instead, the required information should be extracted from the draw context, or retrieved from the user data.

```
/* static */
void RawFootPrintDrawOverride::draw(
     const MHWRender::MDrawContext& context,
     const MUserData* data)
{
    const RawFootPrintData* footData = dynamic_cast<const RawFootPrintData*>(data);
    if (!footData) return;

    // Retrieve any required information from the user data.
    float color[4] = { footData->fColor[0] * footData->fColor[3],
                       footData->fColor[1] * footData->fColor[3],
                       footData->fColor[2] * footData->fColor[3],
                       footData->fColor[3] };

    // Access draw context to retrieve viewport interaction states.
    if (context.inUserInteraction() || context.userChangingViewContext())
    {
    }

    // Access draw context to retrieve the state manager.
    MHWRender::MStateManager* stateMgr = context.getStateManager();

    // Access draw context to retrieve various information.
    footPrint_DebugCameraInformation(context);
    footPrint_DebugDisplayStyle(context);
    footPrint_DebugRenderOverride(context);
    footPrint_DebugPassInformation(context);
    footPrint_DebugPostEffects(context);
    footPrint_DebugFogStatus(context);
    footPrint_DebugObjectTypeExclusions(context);
    footPrint_DebugBackground(context);
    footPrint_DebugDestination(context);
```

Any DG evaluation update should be avoided in the callback. For this example, the custom data (*RawFootPrintData*) is declared to hold a color regarding the display status; checking display status and retrieving color might invoke DG evaluation and thus should only be done in the prepareForDraw() phase.

By default, the callback is invoked once for the opaque pass and each shadow map pass as required. If isTransparent() returns true, the callback will not be invoked by the opaque pass, but once for the front-culling pass and once for the back-culling pass. If excludedFromPostEffects() is overridden to return false, the render passes for post effects can also invoke the callback several times. It is totally up to the plug-in to query pass semantic from the draw context and draw the object correctly for each pass. See *Effects Interfaces* section in *Maya Developer Help* for more information.

Note: If the *deleteAfterUse* flag is true, an MUserData will be deleted by VP2 immediately after the callback returns. So, if the callback is expected to be invoked for multiple times in a refresh, the flag should be set to false. In this case, the user data will be cached until it is explicitly deleted by the prepareForDraw() implementation or the owning object is deleted.

In this example, the actual draw code is wrapped as a base class RawFootPrintDrawAgent and a set of its derived classes to support all draw APIs as required by the plug-in. Note that if a plug-in only needs to support a certain draw API, only one derived class, i.e. one set of codes, is required to be implemented.

```
    // Each draw agent is derived from the base class for each draw API.
    // It is a singleton responsible for managing GPU resources and the
    // actual drawing, thus geometries and shaders are shared among all
    // copies/instances of the associated DAG object.
    RawFootPrintDrawAgent* drawAgent = NULL;
    if (theRenderer->drawAPI() == MHWRender::kOpenGLCoreProfile)
        drawAgent = &RawFootPrintDrawAgentCoreProfile::getDrawAgent();
    #ifdef _WIN32
    else if (theRenderer->drawAPI() == MHWRender::kDirectX11)
        drawAgent = &RawFootPrintDrawAgentDX::getDrawAgent();
    #endif
    else
        drawAgent = &RawFootPrintDrawAgentGL::getDrawAgent();

    // Store parameter values which will later be passed to a custom shader
    // used to draw the object for color pass.
    drawAgent->setColor(MColor(color[0], color[1], color[2], color[3]));
    drawAgent->setMatrix(context);

    // Low-level draw APIs are called here for GPU resource management and
    // drawing.
    drawAgent->beginDraw(context, passShaderOverride);
    if (requireShaded) drawAgent->drawShaded();
    if (requireWireframe) drawAgent->drawWireframe();
    if (requireBoundingBox) drawAgent->drawBoundingBox();
    drawAgent->endDraw();
```

The draw callback can be invoked by a render pass which has a shader instance override, e.g. SSAO has a normal-depth pass which uses a specific shader instance override to output normal and depth values, Motion Blur has a motion-vector pass which uses another specific shader instance override to output motion vectors. Shader instance override can be retrieved using the **MPassContext::shaderOverrideInstance()** method and be bound/unbound as appropriate.

```
/*virtual*/
```

```
void RawFootPrintDrawAgent::beginDraw(
    const MHWRender::MDrawContext& context,
    MHWRender::MShaderInstance* passShaderOverride)
{
    mDrawContext = &context;
    mShaderOverride = passShaderOverride;

    if (mShaderOverride)
    {
        mShaderOverride->bind(context);
        mShaderOverride->updateParameters(context);
        mShaderOverride->activatePass(context, 0);
    }
}

/*virtual*/
void RawFootPrintDrawAgent::endDraw()
{
    if (mShaderOverride)
    {
        mShaderOverride->unbind(*mDrawContext);
        mShaderOverride = NULL;
    }
    mDrawContext = NULL;
}
```

Note that shader instance overrides only fit basic requirements for most common objects. In case there is any custom per-object requirement, a shader can be defined and bound using either the MShaderManager / MShaderInstance interface or low-level draw APIs. See *Effects Interfaces* on the *Maya Developer Help* for more information.

```
/*virtual*/
void RawFootPrintDrawAgentCoreProfile::beginDraw(
    const MHWRender::MDrawContext& context,
    MHWRender::MShaderInstance* passShaderOverride)
{
    // One-time initialization.
    if (!mInitialized)
    {
        GLCP::initGLFunctionsCoreProfile();
        mValid = initShadersCoreProfile() && initBuffersCoreProfile();
        mInitialized = true;
    }

    if (!mValid) return;

    RawFootPrintDrawAgent::beginDraw(context, passShaderOverride);

    // Bind a custom shader if there is no shader instance override for the
    // current pass, e.g. color pass.
    if (!mShaderOverride)
    {
        GLCP::UseProgram(mShaderProgram);
    }
}

/*virtual*/
```

```
void RawFootPrintDrawAgentCoreProfile::endDraw()
{
    if (!mValid) return;

    GLCP::BindVertexArray(0);

    // Unbind the custom shader.
    if (!mShaderOverride)
    {
        GLCP::UseProgram(0);
    }

    RawFootPrintDrawAgent::endDraw();
}

/*virtual*/
void RawFootPrintDrawAgentCoreProfile::drawShaded()
{
    if (!mValid) return;

    // Set parameter values for the custom shader.
    if (!mShaderOverride)
    {
        GLCP::UniformMatrix4fv(mWVPIndex, 1, GL_FALSE, (float*)mMVPMatrix);
        GLCP::Uniform4f(mColorIndex, mColor.r, mColor.g, mColor.b, mColor.a);
    }

    // Bind geometry streams and draw. Streams can be further merged when the
    // performance is bound on # of draw calls (a.k.a. small batch problem).
    GLCP::BindVertexArray(mSoleShadedVAO);
    glDrawElements(GL_TRIANGLES, 3 * (soleCount-2), GL_UNSIGNED_SHORT, 0);
    GLCP::BindVertexArray(mHeelShadedVAO);
    glDrawElements(GL_TRIANGLES, 3 * (heelCount-2), GL_UNSIGNED_SHORT, 0);
}
```

6.3.4.4 Analysis

The MPxDrawOverride class is designed for accessing low-level drawing APIs which gives complete and total control (and responsibility) for drawing the associated DAG object.

6.3.4.4.1 Portability

- Required to implement multiple sets of codes if required to support multiple drawing APIs.
    - OpenGL Legacy mode and OpenGL Core Profile with compatibility mode may reuse the legacy fixed draw code.
    - OpenGL Core Profile strict mode and DirectX 11 need new sets of codes if required.

6.3.4.4.2 Scalability

- Due to the wide-open nature of the interface, it is up to the implementation to use its own performance schemes as Viewport 2.0 performance schemes will not be used, otherwise

equivalent performance cannot be expected when compared to a geometry or subscene override, although it should still outperform MUIDrawManager.

6.3.4.4.3 Compatibility/Flexibility

- ✓ A draw override is free to act as necessary in the draw callback to draw the DAG object (apart from triggering evaluation of the Maya dependency graph).
    - If the override needs to modify the state (in any manner), it must be sure to restore that state before completing execution to avoid state corruption.
- ✓ A draw override can participate in post effects by overriding excludedFromPostEffects(). It is up to the implementation to draw correctly for each render pass based on the context information.
- ✓ Transparency is supported; however a draw override cannot participate in advanced transparency algorithms when isTransparent() is set to true.
- ✓ Selection is fully supported. A draw override can either choose the default VP2 selection behavior, or provide a customized selection implementation. The default VP2 selection behavior fits well for WYSIWYG (What You See Is What You Get) selection, and works no matter whether camera-based selection is on or off as of 2017 Update 4. A customized selection implementation may be required when the plug-in wants to use proxy geometry (like explicit or implicit surfaces represented in analytical forms) for selection test. In this case, it is up to the implementation to perform selection test correctly, e.g. by the OpenGL pick mode, or a CPU geometry intersection method.

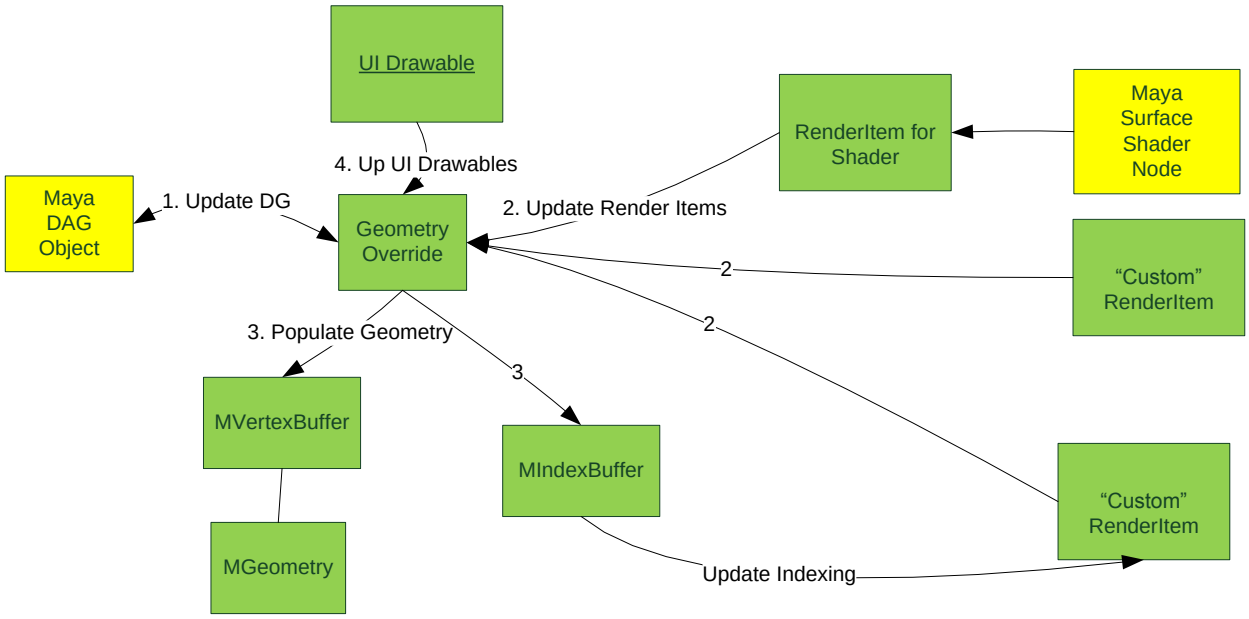
6.4 Porting Surface Shapes using MPxGeometryOverride

This section discusses the added complexity required to support a surface shape (MPxSurfaceShape or MPxComponentShape) as opposed to a locator. This complexity will be examined with respect to an MPxGeometryOverride implementation of the *apiMeshShape* sample plugin. This includes:

- the handling of render items created due to Maya shader node association
- the handling of components
- the handling of VP2 selection

For comparison, a version which is implemented using MPxSubsceneOverride will also be examined.

6.4.1 MPxGeometryOverride (Revisited)



This section will revisit the logic for MPxGeometryOverride, and add render items for shaders. The UI drawable interface has already been examined and will not be looked at again. The "custom" render items are those created by the override. This section will also examine the association between Maya node components and how they are supported using these render items.

6.4.2 Render Items Assigned Shaders for MPxGeometryOverride

Maya automatically adds render items with the appropriate shader instance for each surface shader assigned to an object.

If additional custom render items are required, then the **updateRenderItems()** method should be overridden. Within this method, it is possible to disable any or all existing shaded mode render items and provide custom ones instead. These render items need not be associated with any actual Maya shader assignment.

The sample code below shows how to find existing shading items. Note that there can be shaded items for textured and non-texture display modes as indicated by the render items draw mode (**drawMode**()).

```
// Scan through MRenderItemList passed in via updateRenderItems()
for (int i=0; i<list.length(); i++)
{
       MHWRender::MRenderItem *item = list.itemAt(i);
       if (!item)
          continue;

       // By default the only shaded or textured items are ones provided
       // by Maya.
       MHWRender::MGeometry::DrawMode drawMode = item->drawMode();
```

```
        if (drawMode == MHWRender::MGeometry::kShaded ||
            drawMode == MHWRender::MGeometry::kTextured)
        {
            // Found an internally provided shaded render item.
        }
}
```

Even though these render items are not created by the geometry override, their requirements will be merged with any requirements for shaders explicitly used on render items created by the override.

6.4.3 Wireframe Render Item Example

This section will cover the support required to draw the wireframe for the apiMeshShape plug-in. As the draw modes in which the dormant / template wireframe are displayed differ from active wireframe, separate render items are required.

6.4.3.1 Depth Priority for "UI" Items

If an active render item does not currently exist, one must be created. It must also take into account "depth priority". When drawing more than one render item, depth fighting should be avoided if the render items overlap in depth when drawing. In order to do this, a "depth priority" should be specified for each render item.

The following diagram demonstrates how depth priority can place an object either closer or further away from the current camera used for drawing. Hard-coded values are provided in MRenderItem, but plug-ins can also specify their own priority number as desired.

It is not necessary to provide a “material” depth priority because, unlike in VP1, non-material render items are pulled closer to the camera as opposed to being pushed further away.

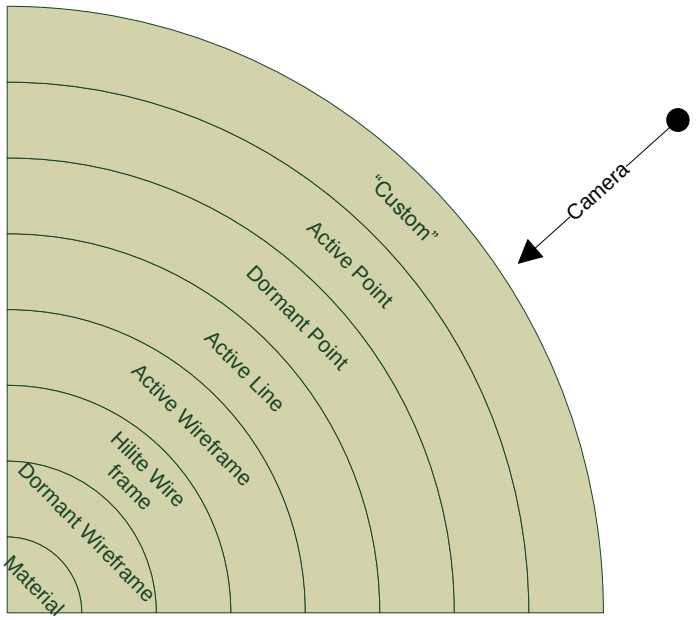


Figure 7: This figure demonstrates how depth priority determines proximity to the camera and hence help avoid depth fighting. In this case, the dormant wireframe priority is lower than that of the active wireframe, but above that of the shaded (material) value.

The render item creation is shown below. Persistent storage for the shader instance on the override is not required, as assignment to a render item will add an additional reference. MShaderInstance::releaseShader() is thus called after assignment to return a reference.

```
// Look for the active wireframe item indicated by name by
// the variable “sSelectedWireframeItemName”
MHWRender::MRenderItem* selectItem = NULL;
int index = list.indexOf(sSelectedWireframeItemName);
if (index < 0)
{
	// This item does not exist yet so create a new one
	selectItem = MHWRender::MRenderItem::Create(
		sSelectedWireframeItemName,
		// Item is a decoration (UI)
		MHWRender::MRenderItem::DecorationItem,
		// Drawing lines
		MHWRender::MGeometry::kLines);
	// Active wireframe shows up in all display modes (shaded,wire,textured etc)
	selectItem->setDrawMode(MHWRender::MGeometry::kAll);
	// Set the depth priority to that used internally for active wireframe
	selectItem->depthPriority( MHWRender::MRenderItem::sActiveWireDepthPriority);
```

```
        // Add the new item to the render item list
        list.append(selectItem);

        // For active wireframe we will require a shader to draw a single color
        //
        MHWRender::MShaderInstance* shader =
                shaderMgr->getStockShader( MHWRender::MShaderManager::k3dSolidShader );
        if (shader)
        {
                // Assign shader to the render item
                selectItem->setShader(shader);
                // Once assigned, no need to hold on to shader instance
                shaderMgr->releaseShader(shader);
        }
}
else
{
        selectItem = list.itemAt(index);
}
```

6.4.3.2 Handling Display State for "UI"

To support the display status coloring for a node, the convenience method MGeometryUtilities::wireframeColor() can be used to return the appropriate color for a given MDagPath (path).

It is also possible to explicitly check the display status using the **MGeometryUtilities::displayStatus()** method. Based on the return status, custom colors may be set.

The first option is shown here. The custom color sample code is available as part of the *apiMeshShape* example in the developer kit.

The display status can also be used to determine whether to enable or disable a render item (show / hide). For active wireframe, this logic takes into account display status.

```
// Get the display status for the dag path
MHWRender::DisplayStatus displayStatus =
        MHWRender::MGeometryUtilities::displayStatus(path);

// Get the color for the given display state
MColor wireColor = MHWRender::MGeometryUtilities::wireframeColor(path);

// Check the display status. If it is not lead, active, hilite, or active
// component then hide the active wireframe render item
switch (displayStatus)
{
  // Only want to show the item when these node states are set
  case MHWRender::kLead:
  case MHWRender::kActive:
  case MHWRender::kHilite:
  case MHWRender::kActiveComponent:
        MHWRender::MShaderInstance* shader = selectItem->getShader();
        if (shader)
```

```
        {
            // Set the shader color parameter
            const MString colorParameterName = "solidColor";
            shader->setParameter(colorParameterName, &(wireColor.r));
        }
        selectItem->enable(true);
        break;

    default:
        // Hide / disable the item otherwise
        selectItem->enable(false);
        break;
};
```

The dormant and template render items have similar logic to the active render item for creation and update. The main differences would be that those items would only be enabled for "wireframe" draw mode, have different depth priority, and use different colors.

It is important to note that the dormant render item has a different name identifier to allow different options to be set for different render items.

The following code creates the dormant / template render item:

```
        wireframeItem = MHWRender::MRenderItem::Create(
                        sWireframeItemName, // Dormant/template wireframe render item name
                        MHWRender::MRenderItem::DecorationItem, // UI item
                        MHWRender::MGeometry::kLines); // Draw lines
        wireframeItem->setDrawMode(MHWRender::MGeometry::kWireframe);

        // Set dormant wireframe with appropriate priority to not clash with
        // any active wireframe which may overlap in depth.
        wireframeItem->depthPriority( MHWRender::MRenderItem::sDormantWireDepthPriority );
        list.append(wireframeItem);

        MHWRender::DisplayStatus displayStatus =
                MHWRender::MGeometryUtilities::displayStatus(path);

        MColor wireColor = MHWRender::MGeometryUtilities::wireframeColor(path);

        // Enable / disable dormant / template wireframe item and
        // update the shader parameters. Note that this differs from active checks.
        if (wireframeItem)
        {
                MHWRender::MShaderInstance* shader = wireframeItem->getShader();

                switch (displayStatus) {
                case MHWRender::kTemplate:
                case MHWRender::kActiveTemplate:
                case MHWRender::kDormant:
                case MHWRender::kActiveAffected:
                        MHWRender::MShaderInstance* shader = selectItem->getShader();
                        // Set the shader color parameter
                        const MString colorParameterName = "solidColor";
                        shader->setParameter(colorParameterName, &(wireColor.r));
```

```
                        // Enable the item
                        wireframeItem->enable(true);
                        break;
                default:
                        // Otherwise disable
                        wireframeItem->enable(false);
                        break;
                }
        }
```

6.4.4 Populating Geometry for Wireframe and Shaded Render Items

Similar to the locator example using MPxGeometryOverride, two parts are involved in the update that is required when populateGeometry() is called: data stream update based on requirements and index stream update for render items.

This section examines an additional update: the shaded render items update.

This example assumes that the total requirement for all assigned shaders comprises: positions, normals and texture coordinates. If there is a stream that is required but not supplied, the internal update system will attempt to create “stand-in” streams just so that there is data specified. As the internal update knows nothing about the specific override’s object geometry, the same logic will always be applied. For example, if a texture coordinate stream is missing, then the values (0,0) could be used for all values.

For data streams, the requirements are scanned (MVertexBufferDescriptionList) and the semantic required for each requirement checked (MVertexBufferDescriptor::semantic()). The steps are the same for each requirement:

- Create a data stream (MVertexBuffer) via MGeometry::createVertexBuffer().
- Acquire a reference / pointer to the CPU memory in the stream. (MVertexBuffer::acquire()).
- Fill in the data as appropriate for the object.
- Commit the data to the GPU via MVertexBuffer::commit().

The code is very similar for all data streams, with the key difference being that the vertex buffer description is used to create the data. When filling in the data, this description should be checked to ensure that the data provided is in the correct format. For example, one texture coordinate requirement could be for a 4-coordinate value while another for a 2-coordinate value. The onus is on the plug-in to provide the correct data.

```
// Buffers for each data stream this geometry supports
MHWRender::MVertexBuffer* positionBuffer = NULL;
MHWRender::MVertexBuffer* normalBuffer = NULL;
MHWRender::MVertexBuffer* cpvBuffer = NULL;
MHWRender::MVertexBuffer* uvBuffer = NULL;
// Pointers to access the CPU data in the buffers
float* positions = NULL;
float* normals = NULL;
float* cpv = NULL;
```

```
float* uvs = NULL;

unsigned int totalVerts = <total number of vertices in object>;

// Scan through the requirements one requirement at a time
//
const MHWRender::MVertexBufferDescriptorList& descList =
       requirements.vertexRequirements();

int numVertexReqs = descList.length();
MHWRender::MVertexBufferDescriptor desc;

for (int reqNum=0; reqNum<numVertexReqs; reqNum++)
{
       if (!descList.getDescriptor(reqNum, desc))
       {
              continue;
       }

       // Check the semantic for the description to determine the stream
       // that requires update…

       // Fill vertex stream data used for dormant vertex, wireframe and shaded drawing.
       switch (desc.semantic())
       {
              case MHWRender::MGeometry::kPosition:
              {
                     if (!positionBuffer)
                     {
                            // Create the buffer and acquire a CPU pointer to it
                            positionBuffer = data.createVertexBuffer(desc);
                            if (positionBuffer)
                            {
                                   positions = (float*)positionBuffer->acquire(totalVerts,
                                                 true /*writeOnly */);
                            }
                     }

              }
              break;

              case MHWRender::MGeometry::kNormal:
              {
                     if (!normalBuffer)
                     {
                            normalBuffer = data.createVertexBuffer(desc);
                            if (normalBuffer)
                            {
                                   // Create the buffer and acquire a CPU pointer to it
                                   normals = (float*)normalBuffer->acquire(totalVerts,
                                                 true /*writeOnly */);
                            }
                     }
              }
              break;
              case MHWRender::MGeometry::kTexture:
              {
                     // Fill in uv values
```

```
                        if (!uvBuffer)
                        {
                                // Create the buffer and acquire a CPU pointer to it
                                uvBuffer = data.createVertexBuffer(desc);
                                if (uvBuffer)
                                {
                                        uvs = (float*)uvBuffer->acquire(totalVerts,
                                                true /*writeOnly */);
                                }
                        }
                }
                break;
                case MHWRender::MGeometry::kColor:
                {
                        if (!cpvBuffer)
                        {
                                cpvBuffer = data.createVertexBuffer(desc);
                                if (cpvBuffer)
                                {
                                        // Create the buffer and acquire a CPU pointer to it
                                        cpv = (float*)cpvBuffer->acquire(totalVerts,
                                                true /*writeOnly */);
                                }
                        }
                }
                break;
                default:
                        // do nothing for streams we do not understand
                        break;
        }
}
```

The actual filling in of the CPU data is not shown here as it is purely data filling. When the data has been filled in, the streams need to be committed (transferred to the GPU). This is done for all streams (positions, normals, texture coordinates and color per vertex).

```
if (positions)
        positionBuffer->commit(positions);

if (normals)
        normalBuffer->commit(normals);

if (uvs)
        uvBuffer->commit(uvs);
}
if (cpv)
        cpvBuffer->commit(cpv);
```

Note in the code below (for the *apiMeshShape* developer kit example), the position stream data is the same for drawing dormant vertices, wireframe as well as shaded drawing (render items). The data organization and the amount of reuse are up to the plug-in.

In this case, an attempt is made to reduce GPU data size by reusing the same data streams for different render items.

The indexing that is used for each render item differs as vertex drawing requires point indexing, wireframe: line indexing, and shaded: triangle indexing.

The *apiMeshShape* has a set of support methods used to fill in the indexing for each of these render items. The basic update logic is the same for each:

- Create a new index buffer using MGeometry::createIndexBuffer().
- Get access to the CPU data using MIndexBuffer::acquire() (vs MVertexBuffer::acquire for data streams)
- Fill in the appropriate indexing for the topology desired.
- Commit the data to the GPU using MIndexBuffer::commit().
- Set the indexing for the appropriate render item using MRenderItem::associateWithIndexBuffer().

The code snippet below demonstrates the indexing update for shaded triangles. The code here has the number of triangles passed in (*numTriangles*). *fMeshGeom* is the mesh data being examined to pull out the indexing.

```
void apiMeshGeometryOverride::updateIndexingForShadedTriangles(const
                              MHWRender::MRenderItem* item,
                             MHWRender::MGeometry& data,
                             unsigned int numTriangles)
{
       // Create an index buffer. We use 32-bit indexing
       MHWRender::MIndexBuffer* indexBuffer =
              data.createIndexBuffer(MHWRender::MGeometry::kUnsignedInt32);
       if (indexBuffer)
       {
             // Get access to CPU data buffer. We are drawing triangles so
             // we require 3 indices per triangle. Thus 3 * number of triangles
             // index data is allocated.
              unsigned int* buffer = (unsigned int*)indexBuffer->acquire(3*numTriangles,
                                    true /*writeOnly */);
              if (buffer)
              {
                    // Compute index data for triangulated convex polygons sharing
                    // poly vertex data among triangles.
                    unsigned int base = 0;
                    unsigned int idx = 0;
                    for (int faceIdx=0; faceIdx<fMeshGeom->faceCount; faceIdx++)
                    {
                           // Ignore degenerate faces
                           int numVerts = fMeshGeom->face_counts[faceIdx];
                           if (numVerts > 2)
                           {
                                  for (int v=1; v<numVerts-1; v++)
                                  {
                                         buffer[idx++] = base;
                                         buffer[idx++] = base+v;
                                         buffer[idx++] = base+v+1;
```

```
                                    }
                                    base += numVerts;
                                }
                            }

                            // Commit the data to the GPU and set the indexing on the
                            // render item.
                            indexBuffer->commit(buffer);
                            item->associateWithIndexBuffer(indexBuffer);
                        }
                    }
}
```

The update for wireframe is very similar, except that, since lines are being drawn, the acquire() would allocate (2 * the number of vertices) and hence the data filling code would also differ.

6.4.5 Component Handling (MPxGeometryOverride)

If a shape supports components, then additional render items are required for them to display properly.

This section will only discuss vertex component drawing, although the *apiMeshShape* developer kit example also shows how edge and face components can be handled. There are also other render items that draw other UI "decorations" such as face centers in the example.

The example points out the changes in the drawing calls if VP2.0 selection support is required. In summary, additional classes are required to handle intersection testing as well as remapping from rendering geometry to component indexing. For more information on component selection, see *Porting Selection from Viewport 1 to 2* in the *Maya Developer Help*.

6.4.5.1 Vertex Render Item Update

The method in the example that updates dormant vertex render items is *updateDormantVertexItem*(). This method creates an item as required for drawing, but includes additional code for VP2 selection as follows:

- The render item is always marked as being selectable if MGeometry::kAll is used for the draw mode.
- The render item, by default, uses the same selection mask as the shape. A vertex component selection mask is enabled by calling **MRenderItem::setSelectionMask**().
- A reference to the internal geometry is part of custom user data placed on the render item to allow for remapping of indexing from the internal geometry to a component. Refer to the *MPxComponentConverter C++ API Reference documentation*, and *Porting Selection from Viewport 1 to 2* in the *Maya Developer Help* for more information. .

```
void apiMeshGeometryOverride::updateDormantVerticesItem(const MDagPath& path,
                        MHWRender::MRenderItemList& list,
                        const MHWRender::MShaderManager* shaderMgr)
{
        MHWRender::MRenderItem* vertexItem = NULL;
        int index = list.indexOf(sVertexItemName);
```

```
	if (index < 0)
	{
		// Create new UI item which draws points
		vertexItem = MHWRender::MRenderItem::Create(
			sVertexItemName,
			MHWRender::MRenderItem::DecorationItem,
			MHWRender::MGeometry::kPoints);

		// Set draw mode to kAll indicating it will be visible in the
		// viewport and also during viewport 2.0 selection
		vertexItem->setDrawMode(
		    MHWRender::MGeometry::DrawMode)(MHWRender::MGeometry::kAll );

		// VP2 Selection: Set selection mask to kSelectMeshVerts indicating
		// we want the render item to be used for "Vertex Component" selection
		vertexItem->setSelectionMask( MSelectionMask::kSelectMeshVerts );

		// Set depth priority higher than wireframe and shaded render items,
		// but lower than active points. Raising higher than wireframe will make
		// them not seem embedded into the surface
		vertexItem->depthPriority(
				MHWRender::MRenderItem::sDormantPointDepthPriority );

		// We want a shader which can draw "fat" points
		MHWRender::MShaderInstance* shader = shaderMgr->getStockShader(
			MHWRender::MShaderManager::k3dFatPointShader );
		if (shader)
		{
			// Set the point size parameter
			static const float pointSize = 3.0f;
			setSolidPointSize( shader, pointSize );

			// Assign shader to the render item
			vertexItem->setShader(shader);

			// Once assigned there is no need to hold on to the shader instance
			shaderMgr->releaseShader(shader);
		}

		// Add the item to the render item list
		list.append(vertexItem);
	}
	else
	{
		vertexItem = list.itemAt(index);
	}
```

After the appropriate render item has been created or found, its color is set, and its enable status is set based on its display status. For example, the example checks to see if the object has been templated, and if so, vertex display is disabled accordingly.

```
	if (vertexItem)
```

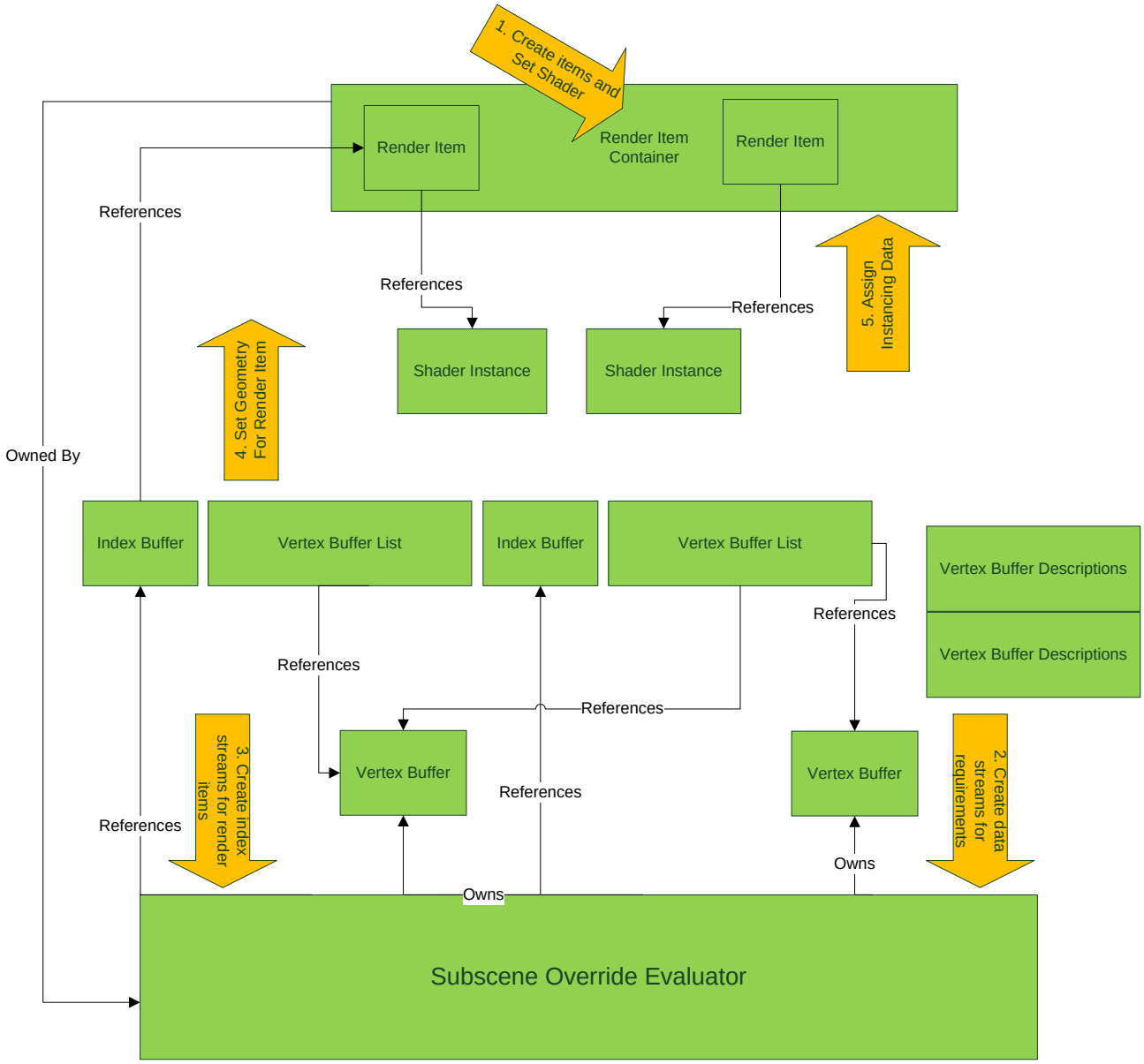
```
        {
                MHWRender::MShaderInstance* shader = vertexItem->getShader();
                if (shader)
                {
                        // Set color for the vertices
                        static const float theColor[] = { 0.0f, 0.0f, 1.0f, 1.0f };
                        setSolidColor( shader, theColor);
                }

                MHWRender::DisplayStatus displayStatus =
                        MHWRender::MGeometryUtilities::displayStatus(path);

                // Generally if the display status is hilite then we
                // draw components.
                if (displayStatus == MHWRender::kHilite)
                {
                        // In case the object is templated
                        // we will hide the components to be consistent
                        // with how internal objects behave.
                        if (path.isTemplated())
                                vertexItem->enable(false);
                        else
                                vertexItem->enable(true);
                }
                else
                {
                        vertexItem->enable(false);
                }

        }
}
```

The updating of the position data stream for dormant vertices is covered above. See the *updateIndexingForDormantVertices*() method in this example for index stream updating. This code is very similar to the wireframe or shaded render item update.

6.5 Porting Surface Shapes using MPxSubSceneOverride

This section will revisit the logic for MPxSubSceneOverride. The *apiMeshShape* example also serves as a simple example of implementing a subscene override for surface shapes. In this example, an alternate implementation to MPxGeometryOverride is provided in the **apiMeshSubSceneOverride** class which is derived from MPxSubSceneOverride. Note that multiple override implementations cannot be associated with the same object and the subscene override implementation is chosen only when the environment variable MAYA_APIMESHSHAPE_USE_SUBSCENEOVERRIDE is set.

Compared to “simple” objects (Section 6.3.2), surface shapes require more complex implementation for handling material assignment, viewport draw modes, component display and selection. UI drawables and hardware instancing in this example are the same and will not be looked at again.



6.5.1 Shaders

To allow for reuse, all shaders are created once and reused. For example, the dormant and active wireframe and vertex/edge/face component shaders are shown. A reference of each shader is held by the override and will be released when the subscene override is destroyed, i.e. the associated shape is deleted.

```
// Set up shared shaders if needed
if (!fWireShader)
{
```

```
    fWireShader = shaderMgr->getStockShader(
        MShaderManager::k3dSolidShader);
    fWireShader->setParameter("solidColor", sRed);
}

if (!fSelectShader)
{
    fSelectShader = shaderMgr->getStockShader(
        MShaderManager::k3dSolidShader);
    fSelectShader->setParameter("solidColor", sGreen);
}

if (!fVertexComponentShader)
{
    fVertexComponentShader = shaderMgr->getStockShader(
        MShaderManager::k3dFatPointShader);
    fVertexComponentShader->setParameter("solidColor", sWhite);
    const float pointSize[2] = {5.0, 5.0};
    fVertexComponentShader->setParameter("pointSize", pointSize);
}

if (!fEdgeComponentShader)
{
    fEdgeComponentShader = shaderMgr->getStockShader(
        MShaderManager::k3dThickLineShader);
    fEdgeComponentShader->setParameter("solidColor", sWhite);
    const float lineWidth[2] = {2.0, 2.0};
    fEdgeComponentShader->setParameter("lineWidth", lineWidth);
}

if (!fFaceComponentShader)
{
    fFaceComponentShader = shaderMgr->getStockShader(
        MShaderManager::k3dSolidShader);
    fFaceComponentShader->setParameter("solidColor", sWhite);
}
```

In the case where it is desired to use a shader that is defined for a node (such as using an MPxSurfaceShadingNodeOverride), then it is possible to find that node and associate it with a given render item. Note that it is solely the responsibility of the plug-in to handle notifications about life-time management of the node.

The **MRenderItem::setShaderForNode**() is the key interface used in the following code. This code explicitly searches for connected shaders to the current DAG object. If it finds a surface shader, then it associates that shader with the render item. Note that a *shadedItemLinkLost*() function has been added to support the scenario when the node lifetime changes.

```
// Find the shaded item if exists, create one otherwise.
MRenderItem* shadedItem = container.find(sShadedName);
if (!shadedItem)
{
    shadedItem = MRenderItem::Create(
        sShadedName,
        MRenderItem::MaterialSceneItem,
        MGeometry::kTriangles);
```

```
    shadedItem->setDrawMode(MGeometry::kShaded);
    shadedItem->setExcludedFromPostEffects(false);
    shadedItem->castsShadows(true);
    shadedItem->receivesShadows(true);
    container.add(shadedItem);
}

// Get the shading node associated with the first component of the first DAG instance
// and use it to get an MShaderInstance. This could be extended to support all DAG
// instances and all components as required.
MPlugArray connectedPlugs;
MObjectArray sets, comps;
if (node.getConnectedSetsAndMembers(0, sets, comps, true))
{
    for (unsigned int i=0; i<sets.length(); i++)
    {
        MFnDependencyNode fnSet(sets[i], &status);
        if (status)
        {
            MPlug shaderPlug = fnSet.findPlug("surfaceShader");
            if (!shaderPlug.isNull())
            {
                shaderPlug.connectedTo(connectedPlugs, true, false);
                if (connectedPlugs.length() > 0) break;
            }
        }
    }
}

// Update shader for the shaded item.
if (fMesh->materialDirty() || !shadedItem->isShaderFromNode())
{
    ShadedItemUserData *userData = new ShadedItemUserData(this);

    if (connectedPlugs.length() > 0 &&
        shadedItem->setShaderFromNode(connectedPlugs[0].node(),
                                      instances[0],
                                      shadedItemLinkLost,
                                      userData,
                                      true))
    {
        fLinkLostCallbackData.push_back(userData);
    }
    else
    {
        shadedItem->setShader(fShadedShader);
        delete userData;
    }
}

// Reset the dirty flag since shader has been updated.
fMesh->setMaterialDirty(false);
```

6.5.2 Data and Index Streams

In this example, positions and normals for mesh display are created, as well as addtional positions for bounding box. The vertex descriptions are determined by the override, and new

buffers are explicitly created. The value of *totalVerts* (total number of vertices) is determined elsewhere.

```
// Acquire vertex buffers.
// We build 2 buffers which can be shared by different render items: position, normal
// and one which is used for bounding box positions (which don't reside on the mesh)
const MVertexBufferDescriptor posDesc("", MGeometry::kPosition, MGeometry::kFloat, 3);
const MVertexBufferDescriptor normalDesc("", MGeometry::kNormal, MGeometry::kFloat, 3);

fPositionBuffer = new MVertexBuffer(posDesc);
float* positions = (float*)fPositionBuffer->acquire(totalVerts, true);

fNormalBuffer = new MVertexBuffer(normalDesc);
float* normals = (float*)fNormalBuffer->acquire(totalVerts, true);

// A bounding box has only 8 positions.
fBoxPositionBuffer = new MVertexBuffer(posDesc);
float* boxPositions = (float*)fBoxPositionBuffer->acquire(8, true);
```

Note that MVertexBuffer::acquire() is still used to allocate CPU memory.

Index streams are maintained by the override. Three index streams are created to handle the drawing of wireframe, shaded triangles and bounding box.

```
// Acquire index buffers.
fWireIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt32);
unsigned int* wireBuffer = (unsigned int*)fWireIndexBuffer->acquire(2*totalVerts, true);

fShadedIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt32);
unsigned int* shadedBuffer = (unsigned int*)fShadedIndexBuffer->acquire(3*numTriangles,
true);

// Short index is used because a bounding box has only 8 positions.
fBoxIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt16);
unsigned short* boxBuffer = (unsigned short*)fBoxIndexBuffer->acquire(24, true);
```

In addition, three index streams are created to handle the drawing of active vertex/edge/face components if needed.

```
// Acquire and fill index buffer for active (selected) vertices
fActiveVerticesIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt32);
unsigned int* buffer = (unsigned int*)fActiveVerticesIndexBuffer->acquire(
    numActiveVertices, true);

// Acquire and fill index buffer for active (selected) edges
fActiveEdgesIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt32);
unsigned int* buffer = (unsigned int*)fActiveEdgesIndexBuffer->acquire(
    2*numActiveEdges, true);

// Acquire and fill index buffer for active (selected) faces
fActiveFacesIndexBuffer = new MIndexBuffer(MGeometry::kUnsignedInt32);
unsigned int* buffer = (unsigned int*)fActiveFacesIndexBuffer->acquire(
    3*numActiveFacesTriangles, true);
```

The code to fill in the buffer is not shown. The commit() still needs to occur for all data and index streams.

6.5.2.1 Advanced: Client Side Buffers

It is possible to assign user created GPU buffers (called "**client side buffers**") to either data or index streams. This is sometimes useful to avoid duplication of data if that data needs to be used outside of Maya. As long as context sharing is performed, then the data can be referenced by Maya.

The **resourceHandle()** methods for MVertexBuffer and MIndexBuffer can be used to assign data as opposed to using the corresponding acquire() methods.

The example code demonstrates the use of resourceHandle() for a simple case of the bounding box data, for both OpenGL Core Profile and OpenGL Legacy.

```
static MGLFunctionTable* gGLFT = NULL;
if (!gGLFT)
{
    MHardwareRenderer* pRenderer = MHardwareRenderer::theRenderer();
    if (pRenderer)
    {
        gGLFT = pRenderer->glFunctionTable();
    }
}

if (gGLFT)
{
    // Position buffer
    gGLFT->glGenBuffersARB(1, &fBoxPositionBufferId);
    if (fBoxPositionBufferId != 0)
    {
        gGLFT->glBindBufferARB(MGL_ARRAY_BUFFER_ARB, fBoxPositionBufferId);
        gGLFT->glBufferDataARB(MGL_ARRAY_BUFFER_ARB, 8*3*sizeof(float),
                               boxPositions, MGL_STATIC_DRAW_ARB);
        gGLFT->glBindBufferARB(MGL_ARRAY_BUFFER_ARB, 0);
        fBoxPositionBuffer->resourceHandle(&fBoxPositionBufferId, 8*3);
    }

    // Index buffer (short index)
    gGLFT->glGenBuffersARB(1, &fBoxIndexBufferId);
    if (fBoxIndexBufferId != 0)
    {
        gGLFT->glBindBufferARB(MGL_ELEMENT_ARRAY_BUFFER_ARB, fBoxIndexBufferId);
        gGLFT->glBufferDataARB(MGL_ELEMENT_ARRAY_BUFFER_ARB, 24*sizeof(unsigned short),
                               boxIndices, MGL_STATIC_DRAW_ARB);
        gGLFT->glBindBufferARB(MGL_ELEMENT_ARRAY_BUFFER_ARB, 0);
        fBoxIndexBuffer->resourceHandle(&fBoxIndexBufferId, 24);
    }
}
```

and DirectX11:

```
// Get the current device
ID3D11Device* pDevice = (ID3D11Device*)renderer->GPUDeviceHandle();

// Fill in a data buffer description.
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.Usage            = D3D11_USAGE_DEFAULT;
```

```
bufferDesc.ByteWidth        = sizeof(float) * 3 * 8;
bufferDesc.BindFlags        = D3D11_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags   = 0;
bufferDesc.MiscFlags        = 0;

// Fill in the sub-resource data.
D3D11_SUBRESOURCE_DATA InitData;
InitData.pSysMem = boxPositions;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;

if (pDevice)
{
    pDevice->CreateBuffer( &bufferDesc, &InitData, &fBoxPositionBufferDX );
    if (fBoxPositionBufferDX)
        fBoxPositionBuffer->resourceHandle((void*)fBoxPositionBufferDX, 8*3);
}

// Index buffer (short index)
bufferDesc.ByteWidth = sizeof(unsigned short) * 24;
bufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
InitData.pSysMem = boxIndices;

if (pDevice)
{
    pDevice->CreateBuffer( &bufferDesc, &InitData, &fBoxIndexBufferDX);
    if (fBoxIndexBufferDX)
        fBoxIndexBuffer->resourceHandle((void*)fBoxIndexBufferDX, 24);
}
```

6.5.3 Render Items

In this example, a set of render items are created as required to handle the drawing for shaded triangles, wireframe, bounding box and active components. They will be stored and managed by an instance of **MSubSceneContainer** associated with the subscene override.

Also shown in the example is that unused items are removed from the container. As noted in the API documentation, the container assumes ownership of those render items. Implementations of MPxSubSceneOverride are free to maintain separate pointers to render items stored in the container, but those pointers will become invalid as soon as the associated render item is removed from the container. This example shows a very simple case of only a few render items. If there are many items, then, in general, it is more efficient to disable them rather than remove them from the container depending on the frequency of reuse.

```
// Always create a render item to handle the drawing of shaded triangles.
MRenderItem* shadedItem = container.find(sShadedName);
if (!shadedItem)
{
    shadedItem = MRenderItem::Create(
        sShadedName,
        MRenderItem::MaterialSceneItem,
        MGeometry::kTriangles);
    shadedItem->setDrawMode(MGeometry::kShaded);
```

```
    container.add(shadedItem);
}

// Always create a render item to handle the drawing of textured triangles.
MRenderItem* texturedItem = container.find(sTexturedName);
if (!texturedItem)
{
    texturedItem = MRenderItem::Create(
        sTexturedName,
        MRenderItem::MaterialSceneItem,
        MGeometry::kTriangles);
    texturedItem->setDrawMode(MGeometry::kTextured);
    container.add(texturedItem);
}

// If needed, create a render item to handle the drawing of wireframe.
MRenderItem* wireItem = container.find(sWireName);
if (!wireItem && anyInstanceUnselected)
{
    wireItem = MRenderItem::Create(
        sWireName,
        MRenderItem::DecorationItem,
        MGeometry::kLines);
    wireItem->setDrawMode(MGeometry::kWireframe);
    wireItem->depthPriority(MRenderItem::sActiveWireDepthPriority);
    wireItem->setShader(fWireShader);
    container.add(wireItem);
}
else if (wireItem && !anyInstanceUnselected)
{
    container.remove(sWireName);
    wireItem = NULL;
}

// If needed, create a render item to handle the drawing of bounding box.
MRenderItem* boxItem = container.find(sBoxName);
if (!boxItem && anyInstanceUnselected)
{
    boxItem = MRenderItem::Create(
        sBoxName,
        MRenderItem::NonMaterialSceneItem,
        MGeometry::kLines);
    boxItem->setDrawMode(MGeometry::kBoundingBox);
    boxItem->setShader(fWireShader);
    container.add(boxItem);
}
else if (boxItem && !anyInstanceUnselected)
{
    container.remove(sBoxName);
    boxItem = NULL;
}

// If needed, create a render item to handle the drawing of active objects.
MRenderItem* selectItem = container.find(sSelectName);
if (!selectItem && anyInstanceSelected)
{
    selectItem = MRenderItem::Create(
        sSelectName,
```

```
        MRenderItem::DecorationItem,
        MGeometry::kLines);
    selectItem->setDrawMode((MGeometry::DrawMode)
        (MGeometry::kWireframe | MGeometry::kShaded | MGeometry::kTextured));
    selectItem->depthPriority(MRenderItem::sActiveWireDepthPriority);
    selectItem->setShader(fSelectShader);
    container.add(selectItem);
}
else if (selectItem && !anyInstanceSelected)
{
    container.remove(sSelectName);
    selectItem = NULL;
}

// If needed, create a render item to handle the drawing of active bounding box.
MRenderItem* selectedBoxItem = container.find(sSelectedBoxName);
if (!selectedBoxItem && anyInstanceSelected)
{
    selectedBoxItem = MRenderItem::Create(
        sSelectedBoxName,
        MRenderItem::NonMaterialSceneItem,
        MGeometry::kLines);
    selectedBoxItem->setDrawMode(MGeometry::kBoundingBox);
    selectedBoxItem->setShader(fSelectShader);
    container.add(selectedBoxItem);
}
else if (selectedBoxItem && !anyInstanceSelected)
{
    container.remove(sSelectedBoxName);
    selectedBoxItem = NULL;
}

// If needed, create a render item to handle the drawing of active vertices.
MRenderItem* activeVertexItem = container.find(sActiveVertexName);
if (!activeVertexItem && anyVertexSelected)
{
    activeVertexItem = MRenderItem::Create(
        sActiveVertexName,
        MRenderItem::DecorationItem,
        MGeometry::kPoints);
    activeVertexItem->setDrawMode(MGeometry::kAll);
    activeVertexItem->depthPriority(MRenderItem::sActivePointDepthPriority);
    activeVertexItem->setShader(fVertexComponentShader);
    container.add(activeVertexItem);
}
else if (activeVertexItem && !anyVertexSelected)
{
    container.remove(sActiveVertexName);
    activeVertexItem = NULL;
}

// If needed, create a render item to handle the drawing of active edges.
MRenderItem* activeEdgeItem = container.find(sActiveEdgeName);
if (!activeEdgeItem && anyEdgeSelected)
{
    activeEdgeItem = MRenderItem::Create(
        sActiveEdgeName,
        MRenderItem::DecorationItem,
```

```
        MGeometry::kLines);
    activeEdgeItem->setDrawMode(MGeometry::kAll);
    activeEdgeItem->depthPriority(MRenderItem::sActiveLineDepthPriority);
    activeEdgeItem->setShader(fEdgeComponentShader);
    container.add(activeEdgeItem);
}
else if (activeEdgeItem && !anyEdgeSelected)
{
    container.remove(sActiveEdgeName);
    activeEdgeItem = NULL;
}

// If needed, create a render item to handle the drawing of active faces.
MRenderItem* activeFaceItem = container.find(sActiveFaceName);
if (!activeFaceItem && anyFaceSelected)
{
    activeFaceItem = MRenderItem::Create(
        sActiveFaceName,
        MRenderItem::DecorationItem,
        MGeometry::kTriangles);
    activeFaceItem->setDrawMode(MGeometry::kAll);
    activeFaceItem->depthPriority(MRenderItem::sActiveLineDepthPriority);
    activeFaceItem->setShader(fFaceComponentShader);
    container.add(activeFaceItem);
}
else if (activeFaceItem && !anyFaceSelected)
{
    container.remove(sActiveFaceName);
    activeFaceItem = NULL;
}
```

6.5.4 Setting Data on Render Items

The sample code below demonstrates the geometry update for the drawing of shaded triangles, bounding box, wireframe and active components. Each adds the appropriate data streams to an array, and then set that array along with the appropriate index stream reference and bounding box. Streams are still owned by the override, but they are now referenced and can be shared by render items.

```
// Get the bounding box for the geometry to draw
MBoundingBox bounds = fMesh->boundingBox();

// Update geometry for shaded and textured triangles
MVertexBufferArray shadedBuffers;
shadedBuffers.addBuffer("positions", fPositionBuffer);
shadedBuffers.addBuffer("normals", fNormalBuffer);
setGeometryForRenderItem(*shadedItem, shadedBuffers, *fShadedIndexBuffer, &bounds);
setGeometryForRenderItem(*texturedItem, shadedBuffers, *fShadedIndexBuffer, &bounds);

// Update geometry for bounding box
MVertexBufferArray boxBuffers;
boxBuffers.addBuffer("positions", fBoxPositionBuffer);
if (boxItem)
    setGeometryForRenderItem(*boxItem, boxBuffers, *fBoxIndexBuffer, &bounds);
if (selectedBoxItem)
```

```
    setGeometryForRenderItem(*selectedBoxItem, boxBuffers, *fBoxIndexBuffer, &bounds);

// Update geometry for wireframe
MVertexBufferArray vertexBuffer;
vertexBuffer.addBuffer("positions", fPositionBuffer);
if (wireItem)
    setGeometryForRenderItem(*wireItem, vertexBuffer, *fWireIndexBuffer, &bounds);
if (selectItem)
    setGeometryForRenderItem(*selectItem, vertexBuffer, *fWireIndexBuffer, &bounds);

// Update geometry for active components
if (activeVertexItem)
    setGeometryForRenderItem(*activeVertexItem, vertexBuffer,
        *fActiveVerticesIndexBuffer, &bounds);
if (activeEdgeItem)
    setGeometryForRenderItem(*activeEdgeItem, vertexBuffer,
        *fActiveEdgesIndexBuffer, &bounds);
if (activeFaceItem)
    setGeometryForRenderItem(*activeFaceItem, vertexBuffer,
        *fActiveFacesIndexBuffer, &bounds);
```

6.5.5
6.5.5

6.5.5 Selection

In addition to object-level selection, component-level selection is also demonstrated in this example. Maya calls the **updateSelectionGranularity**() function during the pre-filtering phase of VP2 selection pipeline, to determine selection level for a given instance of the associated DAG object. A subscene override can override this method to specify an appropriate selection level, otherwise object-level selection will be performed. In the sample code, the selection context is modified to component-level for several cases. For all the other cases object-level selection will still be used although not specified explicitly.

```
void apiMeshSubSceneOverride::updateSelectionGranularity(
    const MDagPath& path, MHWRender::MSelectionContext& selectionContext)
{
    if (MHWRender::MGeometryUtilities::displayStatus(path) == MHWRender::kHilite)
    {
        // The global selection mode is being checked to update the selection context
        // which is used to specify an appropriate selection level.
        MSelectionMask globalComponentMask =
            MGlobal::selectionMode() == MGlobal::kSelectComponentMode ?
            MGlobal::componentSelectionMask() : MGlobal::objectSelectionMask();
        MSelectionMask supportedComponents(MSelectionMask::kSelectMeshVerts);
        supportedComponents.addMask(MSelectionMask::kSelectMeshEdges);
        supportedComponents.addMask(MSelectionMask::kSelectMeshFaces);
        supportedComponents.addMask(MSelectionMask::kSelectPointsForGravity);

        if (globalComponentMask.intersects(supportedComponents))
        {
            selectionContext.setSelectionLevel(MHWRender::MSelectionContext::kComponent);
        }
    }
    else if (pointSnappingActive())
    {
        // For snapping to points, the selection context should be updated to use
```

```
        // component-level selection.
        selectionContext.setSelectionLevel(MHWRender::MSelectionContext::kComponent);
    }
}
```

For more information, see *Porting Selection from Viewport 1 to 2* in the *Maya Developer Help*.

The render items created for the drawing of wireframe, bounding box and shaded triangles can be used for picking automatically. We also create render items for the drawing of active components, but to allow selection picking for all components, we must create additional "selection-only" render items which include geometry data of all components and participate in the selection pipeline only (but not the rendering pipeline as we don't need to draw them). Selection mask should be set as appropriate so that these render items can be picked at the desired selection level, such as vertex, edge or face selection. The code for vertex and edge selection is demonstrated, selection for other component types, e.g. faces, are similar.

```
MRenderItem* vertexSelectionItem = container.find(sVertexSelectionName);
if (!vertexSelectionItem)
{
    // Create render item for vertex selection
    vertexSelectionItem = MRenderItem::Create(sVertexSelectionName,
        MRenderItem::DecorationItem, MGeometry::kPoints);
    // Use for selection only, not visible in viewport
    vertexSelectionItem->setDrawMode(MHWRender::MGeometry::kSelectionOnly);
    // Set selection mask: to be used for vertex selection
    MSelectionMask mask(MSelectionMask::kSelectMeshVerts);
    mask.addMask(MSelectionMask::kSelectPointsForGravity);
    vertexSelectionItem->setSelectionMask( mask );
    // Set selection priority: on top
    vertexSelectionItem->depthPriority(MRenderItem::sSelectionDepthPriority);
    vertexSelectionItem->setShader(fVertexComponentShader);
    container.add(vertexSelectionItem);
}

MRenderItem* edgeSelectionItem = container.find(sEdgeSelectionName);
if (!edgeSelectionItem)
{
    // Create render item for edge selection
    edgeSelectionItem = MRenderItem::Create(
        sEdgeSelectionName,
        MRenderItem::DecorationItem,
        MGeometry::kLines);
    // Use for selection only, not visible in viewport
    edgeSelectionItem->setDrawMode(MHWRender::MGeometry::kSelectionOnly);
    // Set selection mask: to be used for edge selection
    edgeSelectionItem->setSelectionMask(MSelectionMask::kSelectMeshEdges);
    // Set selection priority: on top
    edgeSelectionItem->depthPriority(MRenderItem::sSelectionDepthPriority);
    edgeSelectionItem->setShader(fWireShader);
    container.add(edgeSelectionItem);
}

// Point vertex buffer is fully sequential, use an empty index buffer
// for non-indexed draw.
setGeometryForRenderItem(*vertexSelectionItem,
```

```
    wireBuffers, MIndexBuffer(MGeometry::kUnsignedInt32), &bounds);
// Indexing data for both active and inactive edges are included.
setGeometryForRenderItem(*edgeSelectionItem,
    wireBuffers, *fWireIndexBuffer, &bounds);

if (!fIsInstanceMode && numInstances == 1)
{
    // Set object-to-world matrix.
    vertexSelectionItem->setMatrix(&objToWorld);
    edgeSelectionItem->setMatrix(&objToWorld);
}
else
{
    // Set the object-to-world matrix array and use hardware instancing.
    setInstanceTransformArray(*vertexSelectionItem, instanceMatrixArray);
    setInstanceTransformArray(*edgeSelectionItem, instanceMatrixArray);
}
```

When component-level selection is used, a registered component converter can be called for each associated render item that intersects the selection frustum to convert the intersection data to a list of object components, by mapping the index buffer positions to valid component ids. Refer to the *MPxComponentConverter C++ API Reference* documentation for more information.

```
void simpleComponentConverter::addIntersection(MHWRender::MIntersection& intersection)
{
    // Convert the intersection index, which represent the primitive position in
    // the index buffer, to the correct component id.
    // For vertex and edge: the primitive id is the same as the component id
    // For face: a lookup table has been set up at initialization and can be
    // used to get the component id from the primitive (triangle) id.
    int idx = intersection.index();
    if (fComponentType == MFn::kMeshPolygonComponent)
        if (idx >= 0 && idx < (int)fLookupTable.size())
            idx = fLookupTable[idx];
    fComponent.addElement(idx);
}
```

The pair of the getInstancedSelectionPath() and getSelectionPath() method is called once for each render item that intersects the selection frustum to specify the selected DAG path. Note that getSelectionPath() is maintained for backward compatibility only; it doesn't need to be overridden if a getInstancedSelectionPath() implementation handles both instancing and non-instancing cases.

```
bool apiMeshSubSceneOverride::getInstancedSelectionPath(
    const MHWRender::MRenderItem& renderItem,
    const MHWRender::MIntersection& intersection,
    MDagPath& dagPath) const
{
    MStatus status;
    MFnDagNode node(fObject, &status);
    if (!status) return false;

    MDagPathArray instances;
    if (!node.getAllPaths(instances)) return false;
```

```
    unsigned int numInstances = instances.length();
    if (numInstances == 0) return false;

    int instanceId = intersection.instanceID();

    // The instance ID starts from 1 for the first DAG instance. (instanceID-1)
    // is used as index to MDAGPathArray returned by MFnDagNode:: getAllPaths().
    // In case of one instance or nested instancing, return the first instance.
    if (numInstances == 1 || instanceId == -1 || instanceId > (int)numInstances)
        instanceId = 1;

    dagPath = instances[instanceId - 1];
    return true;
}
```

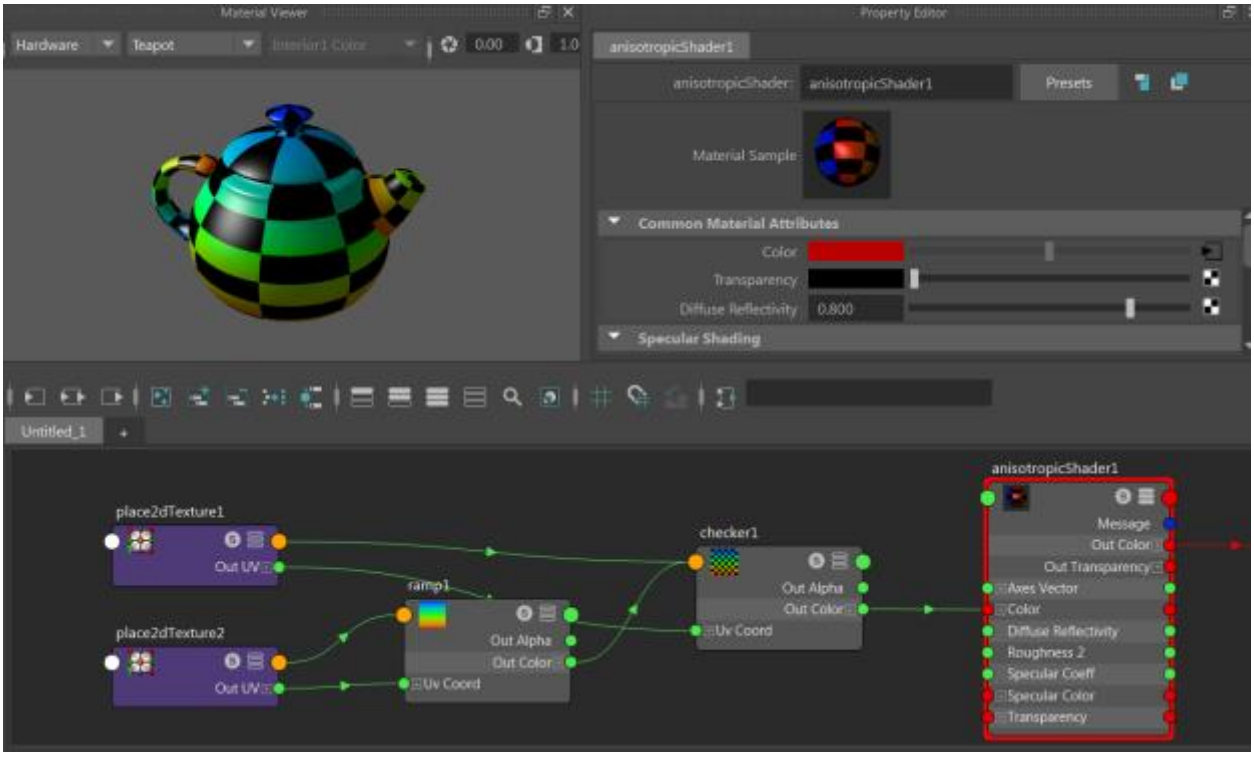
The selected DAG path, combining with the components returned by component converter if component-level selection is used, will form a selection item that is added to the active selection list.

6.6 Porting Shaders

6.6.1 Software Shading Node Attribute Matching

If the minimum requirement for any existing software shader node or new shading node is basic shading, then all that is required is that the names of the attributes on a given node match the names of the parameters on an internal shader.

As an example, the *anisotropicShader* shader plug-in is shown below. The attribute with the name “color” is recognized as a match to the name of an internal shader input parameter, and thus its value as well as its DG connections is tracked. The upstream checker and ramp textures are natively supported in hardware. Internal examination of the connections results in these nodes being supported in the exact same way as they would be if they were connected to an internally provided surface shader.



Internally, the name mapping for “color” can be illustrated as follows. In addition, “Incandescence” is another attribute that is recognized as a match.

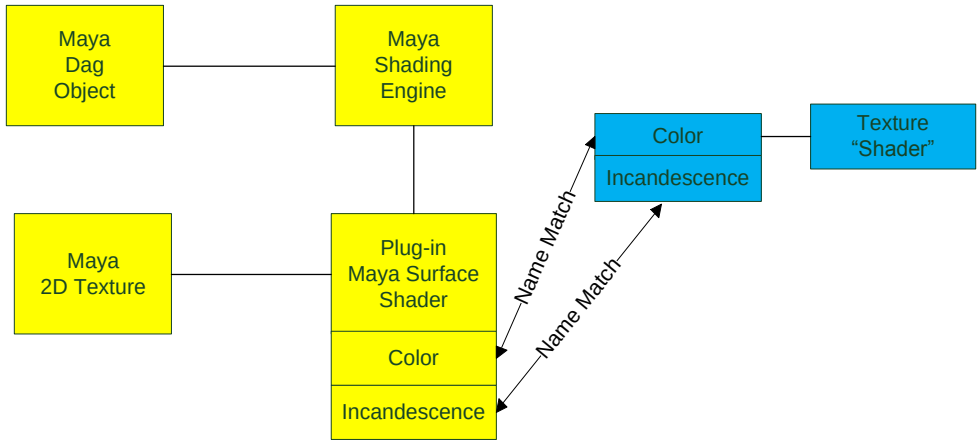


Figure 8 : Plug-in attribute name “Color” matches the internal shader parameter name “Color”. The same is true for “Incandescence” or any other attribute names. The Maya 2D Texture maps internally to a texture “shader”. The connected internal shader graph is examined to produce the final shader instance used for rendering.

Within the plug-in code these are the attribute definitions:

```
MFnNumericAttribute nAttr;

aColor = nAttr.createColor( "color", "c" ); // “color” name is recognized
aIncandescence = nAttr.createColor( "incandescence","ic" ); // “incandescence” name is
recognized
```

6.6.2 Phong Fragment Description

The internal fragments used for VP2 can be accessed via the MFragmentManager class.

The Maya Phong shader fragment is used for default support (as discussed in the previous section). The parameters for this fragment can be examined by dumping out the XML using the *fragmentDumper* sample plug-in (*dumpFragment* command).

In the XML code, the properties section provides a list of parameters of interest. The Phong shader is the fragment named "*phong*_1". The parameters are written as data members.

The following snippet shows parameter names in bold. For example, for "*phong_1.specularColor*", the fragment is "*phong_1*" and the parameter is "*specularColor*".

```
<float  name="phong_1diffuse" ref="phong_1.diffuse" />
<float  name="phong_1translucence" ref="phong_1.translucence" />
<float  name="phong_1translucenceDepth" ref="phong_1.translucenceDepth" />
<float  name="phong_1translucenceFocus" ref="phong_1.translucenceFocus" />
<float3  name="phong_1specularColor" ref="phong_1.specularColor" />
<float  name="phong_1cosinePower" ref="phong_1.cosinePower" />
<float3  name="phong_1color" ref="phong_1.color" />
<float3  name="phong_1transparency" ref="phong_1.transparency" />
<float3  name="phong_1ambientColor" ref="phong_1.ambientColor" />
<float3  name="phong_1incandescence" ref="phong_1.incandescence" />
<float  name="phong_1reflectivity" ref="phong_1.reflectivity" />
<float3  name="phong_1reflectedColor" ref="phong_1.reflectedColor" />
```

6.6.3 Software Shading Node Fragments

The next logical step for greater control is to specify an explicit shader fragment as opposed to the fixed internal one.

The two possible scenarios are:

1) Providing a fragment for a node that is upstream of the surface shader node, regardless of whether or not the surface shader is a plug-in.
2) Providing a fragment for a plug-in surface shader node.

The **MPxShadingNodeOverride** interface can be used for the first scenario and its subclass **MPxSurfaceShadingNodeOverride** for the second.

The following examples will be used as reference to describe how this is accomplished for the two scenarios:

- *brickShader*: Provides an implementation of a brick texture (MPxShadingNodeOverride)
- *onbShader*: Provides an implementation of an Oren-Nayer shader (MPxSurfaceShadingNodeOverride)

The following sections will focus on the VP2 additions required and not the implementation of the software node or its corresponding computation.

6.6.4 Intermediate Nodes (Brick Texture Example)

The first step is to add registration code in the *initializePlugin*() and *uninitializePlugin*() functions respectively. Find the following code in the *brickShader* developer kit example:

```
static const MString sRegistrantId("brickTexturePlugin");

// In initializePlugin()
// To form an association the classification string used for the shading node
// override must match the one added to the software shading node.
// In this case it is “drawdb/shader/texture/2d/brickTexture”
//
// Register software node
const MString UserClassify( "texture/2d:drawdb/shader/texture/2d/brickTexture" );
CHECK_MSTATUS( plugin.registerNode("brickTexture", brickTextureNode::id,
                                        &brickTextureNode::creator,
                                        &brickTextureNode::initialize,
                                        MPxNode::kDependNode,
                                        &UserClassify ) );

// Register shading node override
MHWRender::MDrawRegistry::registerShadingNodeOverrideCreator(
                     "drawdb/shader/texture/2d/brickTexture",
                     sRegistrantId,
                     brickTextureNodeOverride::creator));

…

// In uninitializePlugin()
// Deregister shading node override
MHWRender::MDrawRegistry::deregisterShadingNodeOverrideCreator(
                     "drawdb/shader/texture/2d/brickTexture",
                     sRegistrantId));
```

For the override to be recognized as an evaluator for a shading node, the “***drawdb/shader***” string must be specified. The “*brickTexture*” string is added to distinguish this particular shading node. The optional classification “*texture/2d*” string is added to make it consistent with Maya’s internal naming scheme and to provide a unique string, although it is not strictly required. That is, a “drawdb/shader/brickTexture” classification would also work but may result in accidental reuse of the same classification string. For duplicate classifications, only the last registration will be used.

To associate the evaluator / override with the plug-in shader node, the same classification string must exist as one of the classifications when *registerNode*() is called.

The class *brickTextureNodeOverride* derives from the override class MPxShadingNodeOverride, with a *creator*() method which will instantiate one instance of the override per Maya shading node.

The code for an MPxShadingNodeOverride which will provide the fragment definition with default handling of input parameters and output parameters are:

1. virtual MString **fragmentName**() const = 0;
2. virtual MHWRender::DrawAPI **supportedDrawAPIs**() const;

Fragment registration is via the MFragmentManager class.

6.6.4.1 Fragment Definition

For the brick example, the definition is embedded as a string within the code (as opposed to a separate XML file).

First, the fragment name must be available so that it can be returned. In the example, the name "*brickTextureNodePluginFragment*" is specified.

```
// Define fragments needed for VP2 version of shader, this could also be
// defined in a separate XML file
//
// 1. We require a fragment name
static const MString sFragmentName("brickTextureNodePluginFragment");
```

The fragment body must be specified next. This is composed of the following:

- The input parameter definitions
- The input parameter default values
- The output parameter definition
- The implementation(s) for each supported shader language

Within the body definition, the fragment name must be specified, in this case, "brickTextureNodePluginFragment". The "class" value is a "**ShaderFragment**", as a shader fragment is defined. In the *onbShader* example a different class value is used to indicate a fragment graph.

```
// 2. The fragment body is defined in one string
//
static const char* sFragmentBody =
"<fragment uiName=\"brickTextureNodePluginFragment\" name=\"brickTextureNodePluginFragment\"
type=\"plumbing\" class=\"ShadeFragment\" version=\"1.0\">"
"       <description><![CDATA[Brick procedural texture fragment]]></description>"
```

The <properties>, <values> and <outputs> sections define the input and output parameters and their default values. The <properties> section lists the parameters one at a time as well as their type. A semantic can be applied to allow for binding of various supported parameters. [See *Shading Node Overrides* in the *Maya Developer Help* for more information.]

```
// 3a. The properties define the input parameters to the shader
"       <properties>"
"              <float3 name=\"brickColor\" />"
"              <float3 name=\"jointColor\" />"
```

```
"              <float name=\"blurFactor\" />"
"              <float2 name=\"uvCoord\" semantic=\"mayaUvCoordSemantic\"
flags=\"varyingInputParam\" />"
"              <float2 name=\"uvFilterSize\" />"
"       </properties>"

// 3b. The values define the default values for the parameters
"       <values>"
"              <float3 name=\"brickColor\" value=\"0.75,0.3,0.1\" />"
"              <float3 name=\"jointColor\" value=\"0.75,0.75,0.75\" />"
"       </values>"
```

For this fragment, there is one 3-float output which is called “outColor”:

```
// 4. The output section defines the output parameter for the shader
"       <outputs>"
"              <float3 name=\"outColor\" />"
"       </outputs>"
```

After the description of the interfaces to the fragment has been defined, all that is left is to define the actual implementation. Currently, three possible shader languages are supported: Cg, GLSL and HLSL.

The brick texture plug-in includes all 3 implementations, which is reflected in the overridden method: **supportedDrawAPIs**(). **kOpenGLCoreProfile** indicates that an OpenGL core profile is supported via an GLSL implementation.

```
MHWRender::DrawAPI brickTextureNodeOverride::supportedDrawAPIs() const
{
       return MHWRender::kOpenGL | MHWRender::kDirectX11 | MHWRender::kOpenGLCoreProfile;
}
```

The implementation section starts with the <implementation> tag.

```
// 5. The output section defines the output parameter for the shader
// We have one for each of the supported languages: Cg, GLSL and HLSL
// Not all are required but this must be reflected in the
// MPxShadingNodeOverride::supportedDrawAPIs() method
//
"       <implementation>"
```

The Cg implementation is indicated via the language option on the <implementation> tag: “*Cg*”. The function name provided within the <function_name> tag should match the name of the fragment so that the correct function can be used. Again, this is “*brickTextureNodePluginFragment*”. The actual source is specified within the <source></source> tags. This code is only interpreted when the shader is compiled.

It is worth noting that the actual procedure argument names do not have to match the property name, but the ordering and type needs to match. For example shader argument “*color1*” will

match the "*brickColor*" property. In general it is recommended that the property names be re-used for clarity.

```
// 5a. Cg implementation

"<implementation render=\"OGSRenderer\" language=\"Cg\" lang_version=\"2.1\">"

"<function_name val=\"brickTextureNodePluginFragment\" />"
"<source><![CDATA["
"float btnplinearstep(float t, float a, float b) \n"
"{ \n"
"       if (t < a) return 0.0f; \n"
"       if (t > b) return 1.0f; \n"
"       return (t - a)/(b - a); \n"
"} \n"
// Note that the color1 argument matches brickColor property which are both 3-float
"float3 brickTextureNodePluginFragment(float3 color1, float3 color2, float blur, float2 uv,
float2 fs) \n"
"{ \n"
"       uv -= floor(uv); \n"
"       float v1 = 0.05f; float v2 = 0.45f; float v3 = 0.55f; float v4 = 0.95f; \n"
"       float u1 = 0.05f; float u2 = 0.45f; float u3 = 0.55f; float u4 = 0.95f; \n"
"       float du = blur*fs.x/2.0f; \n"

"       float dv = blur*fs.y/2.0f; \n"
"       float t = max( \n"
"               min(btnplinearstep(uv.y, v1 - dv, v1 + dv) - btnplinearstep(uv.y, v2 - dv, v2 +
dv), \n"
"                       max(btnplinearstep(uv.x, u3 - du, u3 + du), 1.0f - btnplinearstep(uv.x,
u2 - du, u2 + du))), \n"
"               min(btnplinearstep(uv.y, v3 - dv, v3 + dv) - btnplinearstep(uv.y, v4 - dv, v4 +
dv), \n"
"                       btnplinearstep(uv.x, u1 - du, u1 + du) - btnplinearstep(uv.x, u4 - du,
u4 + du))); \n"

"       return t*color1 + (1.0f - t)*color2; \n"
"} \n]]>"
"               </source>"
"       </implementation>"
```

The HLSL implementation is very similar to the Cg implementation, except that a proper language version "*lang_version*" must be defined. This must be "**11.0**" to indicate specific support for DirectX version **11**.

```
// 5b. HLSL implementation

//

"       <implementation render=\"OGSRenderer\" language=\"HLSL\" lang_version=\"11.0\">"

"               <function_name val=\"brickTextureNodePluginFragment\" />"
"               <source><![CDATA["
"float btnplinearstep(float t, float a, float b) \n"
"{ \n"
"       if (t < a) return 0.0f; \n"
"       if (t > b) return 1.0f; \n"
"       return (t - a)/(b - a); \n"
"} \n"
```

```
"float3 brickTextureNodePluginFragment(float3 color1, float3 color2, float blur, float2 uv,
float2 fs) \n"
"{ \n"
"       uv -= floor(uv); \n"
"       float v1 = 0.05f; float v2 = 0.45f; float v3 = 0.55f; float v4 = 0.95f; \n"
"       float u1 = 0.05f; float u2 = 0.45f; float u3 = 0.55f; float u4 = 0.95f; \n"
"       float du = blur*fs.x/2.0f; \n"

"       float dv = blur*fs.y/2.0f; \n"
"       float t = max( \n"
"               min(btnplinearstep(uv.y, v1 - dv, v1 + dv) - btnplinearstep(uv.y, v2 - dv, v2 +
dv), \n"
"               max(btnplinearstep(uv.x, u3 - du, u3 + du), 1.0f - btnplinearstep(uv.x, u2 - du,
u2 + du))), \n"
"               min(btnplinearstep(uv.y, v3 - dv, v3 + dv) - btnplinearstep(uv.y, v4 - dv, v4 +
dv), \n"
"               btnplinearstep(uv.x, u1 - du, u1 + du) - btnplinearstep(uv.x, u4 - du, u4 +
du))); \n"

"       return t*color1 + (1.0f - t)*color2; \n"
"} \n]]>"
"               </source>"
"       </implementation>"
```

The GLSL implementation is also very similar, but with the language option set to “GLSL”. As different video cards may support different GLSL versions, ensure that your code matches the version supported. Again, the source code will be checked at shader compile time.

```
// 5c. GLSL implementation
//
"       <implementation render=\"OGSRenderer\" language=\"GLSL\" lang_version=\"3.0\">"
"               <function_name val=\"brickTextureNodePluginFragment\" />"
"               <source><![CDATA["
"float btnplinearstep(float t, float a, float b) \n"
"{ \n"
"       if (t < a) return 0.0f; \n"
"       if (t > b) return 1.0f; \n"
"       return (t - a)/(b - a); \n"
"} \n"
"vec3 brickTextureNodePluginFragment(vec3 color1, vec3 color2, float blur, vec2 uv, vec2 fs)
\n"

"{ \n"
"       uv -= floor(uv); \n"
"       float v1 = 0.05f; float v2 = 0.45f; float v3 = 0.55f; float v4 = 0.95f; \n"
"       float u1 = 0.05f; float u2 = 0.45f; float u3 = 0.55f; float u4 = 0.95f; \n"
"       float du = blur*fs.x/2.0f; \n"

"       float dv = blur*fs.y/2.0f; \n"
"       float t = max( \n"
"               min(btnplinearstep(uv.y, v1 - dv, v1 + dv) - btnplinearstep(uv.y, v2 - dv, v2 +
dv), \n"
"               max(btnplinearstep(uv.x, u3 - du, u3 + du), 1.0f - btnplinearstep(uv.x, u2 - du,
u2 + du))), \n"
"               min(btnplinearstep(uv.y, v3 - dv, v3 + dv) - btnplinearstep(uv.y, v4 - dv, v4 +
dv), \n"
"               btnplinearstep(uv.x, u1 - du, u1 + du) - btnplinearstep(uv.x, u4 - du, u4 +
```

```
du))); \n"

"	return t*color1 + (1.0f - t)*color2; \n"
"} \n]]>"
"		</source>"
"	</implementation>"
```

We close off the string by indicating the end of the implementations and the end of the fragment definition.

```
"	</implementation>"
"</fragment>";
```

6.6.4.2 Fragment Registration

In order to use this fragment, it must be added to the internal fragment manager via the *MFragmentManager* class. Because the fragment is a string buffer in this example, the *MFragmentManager::addShaderFragmentFromBuffer*() method is used.

sFragmentName is the fragment name and *sFragmentBody* is the string containing the XML description.

```
// Register fragment with the manager
MHWRender::MRenderer* theRenderer = MHWRender::MRenderer::theRenderer();
if (theRenderer)
{
	MHWRender::MFragmentManager* fragmentMgr =
		theRenderer->getFragmentManager();
	if (fragmentMgr)
	{
		// Add fragment if it has not already been added.
		bool fragAdded = fragmentMgr->hasFragment(sFragmentName);
		if (!fragAdded)
		{
			fragAdded = (sFragmentName ==
			     fragmentMgr->addShadeFragmentFromBuffer(sFragmentBody, false));
		}

		// Use the fragment on successful add
		if (fragAdded)
		{
			fFragmentName = sFragmentName;
		}
	}
}
```

In order for the override to indicate the fragment name used, the **fragmentName**() method is overridden. Note that, at the end of the registration code (above), the member *fFragmentName* is set to be the name of the fragment if it is successfully added to the fragment manager.

```
MString brickTextureNodeOverride::fragmentName() const
{
	return fFragmentName;
```

```
}
```

If the fragment is correctly registered, it is possible to retrieve the fragments XML description from the fragment manager. In this case, the XML for the fragment graph wrapper would appear as follows (as retrieved via the *fragmentDumper* plug-in). Note that the values for each of the parameters are exposed. Also note that, when created, a unique name has been given for the particular instance of the fragment. (As a fragment graph is shown, the implementation is not described).

```
<fragment_graph  name="MFragmentManager_getFragmentXML_TempGraph"
ref="MFragmentManager_getFragmentXML_TempGraph" class="FragmentGraph" version="1.0" feature_level="0" >
   <fragments>
         <fragment_ref name="brickTexture_1" ref="brickTexture_1" />
   </fragments>
   <connections>
   </connections>
   <properties>
      <float3  name="brickTexture_1brickColor" ref="brickTexture_1.brickColor" />
      <float3  name="brickTexture_1jointColor" ref="brickTexture_1.jointColor" />
      <float  name="brickTexture_1blurFactor" ref="brickTexture_1.blurFactor" />
      <float2  name="uvCoord" ref="brickTexture_1.uvCoord" semantic="mayaUvCoordSemantic"
flags="varyingInputParam" />
      <float2  name="brickTexture_1uvFilterSize" ref="brickTexture_1.uvFilterSize" />
   </properties>
   <values>
<float3 name="brickTexture_1brickColor" value="0.750000,0.300000,0.100000"  />
<float3 name="brickTexture_1jointColor" value="0.750000,0.750000,0.750000"  />
   </values>
   <outputs>
      <float3  name="outColor" ref="brickTexture_1.outColor" />
   </outputs>
</fragment_graph>
```

6.6.4.3 Attribute-Parameter Mappings (Part 1)

In most cases, the values for the parameters on the shading effect are automatically driven by the attributes of the Maya nodes that were used to create the effect. This is done by matching the attributes on each Maya node to the parameters of the corresponding fragment. The name and type of the attributes must match the name and type of the parameters for this automatic relationship to be established.

In the sample code, the input parameters (or properties) of the brick texture fragment have the same name and data type as the input attributes defined for the *brickTexture* plug-in node. Thus, Maya automatically sets the values for those parameters on the final shading effect using the values from the attributes on the brickTexture node. No further work is required by the shading node override.

6.6.4.3.1 Example in Use

As previously mentioned Maya combines the fragments for each node in a shading network and turns the overall fragment graph into a shading effect. In the example above

- The brick texture node is connected to a Blinn shader via the “outColor” attribute. No additional code has been added to map the attribute to the parameter since “outColor” is the name of the output parameter on the fragment.
- A checker node has been connected to the “brickColor” attribute as input. Again no additional code is required since the attribute name matches a corresponding parameter name.
- A place2dTexture node (2d texture placement) is connected to the “uvCoord” attribute and again, automatic mapping occurs as the parameter name is also called “uvCoord”. The number of repeats on the placed2Dtexture node has been increased from its default value of 1 to 5.

The result of the shader compiled from the fragment graph is shown on the right on a 2d plane.

6.6.4.4 Attribute-Parameter Mappings (Part 2: fileTexture Example)

To add custom handling of attribute-parameter mappings, use the following methods:

1. virtual void **updateDG**();
2. virtual void **updateShader**(MShaderInstance& shader, const MAttributeParameterMappingList& mappings);
3. virtual void **getCustomMappings**(MAttributeParameterMappingList& mappings);

The *fileTexture* developer kit example demonstrates use of these methods.

Plug-ins may specify associations between attributes and parameters of the same type, but with different names, by implementing **MPxShadingNodeOverride::getCustomMappings**(). This method is called immediately after the fragment is created, but before the automatic mappings

are done. No automatic mapping is performed for any parameter on the fragment that already has a custom mapping.

Any attribute on the node that has no mapping to a parameter on the fragment is ignored. Similarly, any parameters on the fragment without a mapping to an attribute on the node is ignored (unless custom parameter setting is done by **MPxShadingNodeOverride::updateShader**()).

In the *fileTexture* example, method mappings are added for the "*map*" and "*textureSampler*" parameters.

```
void FileNodeOverride::getCustomMappings(
	MHWRender::MAttributeParameterMappingList& mappings)
{
	// Set up some mappings for the parameters on the file texture fragment,
	// there is no correspondence to attributes on the node for the texture
	// parameters.
	MHWRender::MAttributeParameterMapping mapMapping(
		"map", "", false, true);
	mappings.append(mapMapping);

	MHWRender::MAttributeParameterMapping textureSamplerMapping(
		"textureSampler", "", false, true);
	mappings.append(textureSamplerMapping);
}
```

The mappings should be handled by deriving from the MPxShadingNodeOverride::**updateShader**() method. If this is not done, then no remapping will be performed.

Note that all functionality is driven through these attribute parameter mappings. When Maya is traversing the shading network and building and connecting fragments, it only traverses connections where the input attribute on the node has a defined mapping (custom or automatic). Also, as fragments are combined for all the nodes in the Maya shading graph, their **parameters are renamed** (**resolved**) in order to avoid "collisions" (allowing the same fragment type to be used multiple times in a graph). Only parameters with mappings are renamed; all others may suffer name collisions which produce unpredictable results.

For the *fileTexture* node example the updateShader() method will receive the name of the parameter that is created after the fragment graph has had its names resolved / renamed.

```
void FileNodeOverride::updateShader(
	MHWRender::MShaderInstance& shader,
	const MHWRender::MAttributeParameterMappingList& mappings)
{
	// Handle resolved name caching
	if (fResolvedMapName.length() == 0)
	{
		const MHWRender::MAttributeParameterMapping* mapping =
			mappings.findByParameterName("map");
		if (mapping)
```

```
		{
			fResolvedMapName = mapping->resolvedParameterName();
		}
	}
	if (fResolvedSamplerName.length() == 0)
	{
		const MHWRender::MAttributeParameterMapping* mapping =
			mappings.findByParameterName("textureSampler");
		if (mapping)
		{
			fResolvedSamplerName = mapping->resolvedParameterName();
		}
	}
```

.

The plug-in retrieves the resolved names using the MAttributeParameterMapping::resolvedParameterName() method. The names are then used to find the parameters on the shader instance. In this example the code will set the appropriate hardware texture and hardware sampler for two parameters using the appropriate **MShaderInstance::setParameter()** method variant. The example also shows how to use the texture manager (MTextureManager) to acquire a hardware texture based on a file texture name that is stored in a data member called *fFileName*.

```
	// Set the parameters on the shader
	if (fResolvedMapName.length() > 0 && fResolvedSamplerName.length() > 0)
	{
		// Set sampler to linear-wrap
		if (!fSamplerState)
		{
			MHWRender::MSamplerStateDesc desc;
			desc.filter = MHWRender::MSamplerState::kAnisotropic;
			desc.maxAnisotropy = 16;
			fSamplerState = MHWRender::MStateManager::acquireSamplerState(desc);
		}
		if (fSamplerState)
		{
			shader.setParameter(fResolvedSamplerName, *fSamplerState);
		}

		// Set texture if we can acquire a texture from the texture
		// manager via MTextureManager::acquireTexture().
		MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
		if (renderer)
		{
			MHWRender::MTextureManager* textureManager =
				renderer->getTextureManager();
			if (textureManager)
			{
				MHWRender::MTexture* texture =
					textureManager->acquireTexture(fFileName);
				if (texture)
				{
					MHWRender::MTextureAssignment textureAssignment;
					textureAssignment.texture = texture;
					shader.setParameter(fResolvedMapName,
```

```
                                        textureAssignment);

                                    // Release our reference now that it is set on the
                                     // shader
                                    textureManager->releaseTexture(texture);
                                }
                            }
                        }
                    }
}
```

The **updateDG**() method is overridden in the example to cache data that is used for the **updateShader**() method. In this case, its purpose is to cache the file texture name into the fFileTexture member variable.

```
void FileNodeOverride::updateDG()
{
        // Pull the file name from the DG for use in updateShader
        MStatus status;
        MFnDependencyNode node(fObject, &status);
        if (status)
        {
                MString name;
                node.findPlug("fileName").getValue(name);
                MRenderUtil::exactFileTextureName(name, false, "", fFileName);
        }
}
```

6.6.4.5 Output Parameter Handling

Like input parameters, if the output parameters of the fragment specified by an override match the name and type of the output attributes on the associated shading node, Maya automatically forms connections between the output of the fragment and the inputs of other fragments as required by the shading network.

Currently, the fragment system only supports one output parameter per fragment for normal shading fragments. To create a fragment with multiple outputs (to match multiple outputs on a Maya shading node), such a fragment must define its single output parameter as a “struct” output. A separate fragment must be created to define the **struct** type and the main fragment must be connected to this new struct definition fragment in a fragment graph. The graph can then be used by Maya and Maya automatically matches the names and types of the struct members to the output attributes of the shading node where required.

In the *fileTexture* example, the following structure is created to support an output structure that can return a 3-channel output color as well as a 3-channel output alpha. The **type** in this case is a “**structure**” and a unique id (*fileTexturePluginFragmentOutput*) is used for the structure name (**struct_name**). (Only the GLSL implementation is shown below for brevity, although all three implementations exist in the sample code).

```
static const MString sFragmentOutputName("fileTexturePluginFragmentOutput");
static const char* sFragmentOutputBody =
// Fragment is defined as a structure
"<fragment uiName=\"fileTexturePluginFragmentOutput\"
name=\"fileTexturePluginFragmentOutput\" type=\"structure\" class=\"ShadeFragment\"
version=\"1.0\">"
"      <description><![CDATA[Struct output for simple file texture
fragment]]></description>"
"      <properties>"

// Define the structure property
"            <struct name=\"fileTexturePluginFragmentOutput\"
struct_name=\"fileTexturePluginFragmentOutput\" />"
"      </properties>"
"      <values>"
"      </values>"
"      <outputs>"
"            <alias name=\"fileTexturePluginFragmentOutput\"
struct_name=\"fileTexturePluginFragmentOutput\" />"
"            <float3 name=\"outColor\" />" // Output color is part of the struct
"            <float name=\"outAlpha\" />"  // Output alpha is part of the struct
"      </outputs>"

// Add shader source code implementation(s) for the structure
"      <implementation>"
"      <implementation render=\"OGSRenderer\" language=\"GLSL\" lang_version=\"3.0\">"
"            <function_name val=\"\" />"
"            <declaration name=\"fileTexturePluginFragmentOutput\"><![CDATA["

// Output structure GLSL source code
"struct fileTexturePluginFragmentOutput \n"
"{ \n"
"      vec3 outColor; \n"
"      float outAlpha; \n"
"}; \n]]>"
"            </declaration>"
"      </implementation>"
"      </implementation>"
"</fragment>";
```

Within the body definition, the <outputs> section must reference the defined output structure. Note how the **struct** keyword is used as the return type and the previously specified structure name (*fileTexturePluginFragmentOutput*) is used for the **struct_name** keyword, which differs from the previous example for the brick shader which just returns a single float3 value.

```
"<outputs>"
<struct name=\"output\" struct_name=\"fileTexturePluginFragmentOutput\" />"
"</outputs>"
```

As with the body, the output structure must be added to the fragment manager using **MFragmentManager::addShaderFragmentFromBuffer**().

In the *fileTexture* example, the names of the output parameters on the fragment match the names of the output attributes on the node (and the types match also).

As with input parameters, the plug-in can define custom mappings for outputs. The **MPxShadingNodeOverride::outputForConnection**() needs to be overridden to handle custom output mappings. The following sample code show a sample scenario:

```
MString sampleOverride::outputForConnection(
	const MPlug& sourcePlug,
	const MPlug& destinationPlug)
{
	// Map attribute names outValue/outValueA to parameter name outColor/outAlpha
	MFnAttribute attribute(sourcePlug.attribute());
	if (attribute.name() == "outValue")
	{
		return "outColor";
	}
	else if (attribute.name() == "outValueA")
	{
		return "outAlpha";
	}
	else
	{
		return attribute.name();
	}
}
```

Here, the nodes attributes are called outValue and outValueA. The code maps them to return outColor and outAlpha respectively.

Whenever Maya detects a change in an attribute value, the method **valueChangeRequiresFragmentRebuild()** is called. If this method returns true, then Maya assumes that a new configuration of the fragment graph is required, which will trigger a rebuild of the shading effect. For example:

```
bool sampleOverride::valueChangeRequiresFragmentRebuild(
	const MPlug* plug) const
{
	MStatus status;
	if (plug)
	{
		if (*plug == sampleNode::aMyAttribute)
		{
			return true;
		}
	}
	return false;
}
```

Here the attribute sampleNode::aMyAttribute is checked when a value change occurs and will return true indicating that a graph rebuild is required.

6.6.4.6 XML file example

The *checkerShader* plug-in is an example of using a shader description which is written as an XML file on disk. Like the fileTexture node example, it also has an output structure description.

The main difference is the usage of the **MFragmentManager::addShadeFragmentFromFile()** interface to add fragments to the fragment manager, and the usage of **MFragmentManager::addFragmentPath()** to allow for an additional search path to be specified to search for XML files.

In the sample code, a fragment to define the checker logic, an output structure definition, and a graph to connect the two are stored in three different XML files:

- checkerNodePluginFragment.xml
- checkerNodePluginFragmentOutput.xml
- checkerNodePluginGraph.xml

The XML files are assumed to be stored in the “devkit/plug-ins/checkShader” relative folder.

```
// Fragments are defined in separate XML files, add the checker node
// directory to the search path and load from the files.
static const MString sFragmentName("checkerNodePluginFragment");
static const MString sFragmentOutputName("checkerNodePluginFragmentOutput");
static const MString sFragmentGraphName("checkerNodePluginGraph");

// Add search path (once only)
static bool sAdded = false;
if (!sAdded)
{
	MString location;
	if( ! MGlobal::executeCommand(MString("getModulePath -moduleName \"devkit\""),
location, false) ) {
		location = MString(getenv("MAYA_LOCATION")) + MString("/devkit");
	}
	location += "/plug-ins/checkerShader";
	fragmentMgr->addFragmentPath(location);
	sAdded = true;
}

// Add fragments if needed
bool fragAdded = fragmentMgr->hasFragment(sFragmentName);
bool structAdded = fragmentMgr->hasFragment(sFragmentOutputName);
bool graphAdded = fragmentMgr->hasFragment(sFragmentGraphName);
if (!fragAdded)
{
	fragAdded = (sFragmentName == fragmentMgr->addShadeFragmentFromFile(sFragmentName
+ ".xml", false));
}
if (!structAdded)
{
	structAdded = (sFragmentOutputName == fragmentMgr-
>addShadeFragmentFromFile(sFragmentOutputName + ".xml", false));
}
if (!graphAdded)
{
	graphAdded = (sFragmentGraphName == fragmentMgr-
>addFragmentGraphFromFile(sFragmentGraphName + ".xml"));
}
```

6.6.5 Surface Shader Node Example (Oren-Nayer)

The shader example *onbShader* shows how a surface shader can be implemented with a series of fragments using the **MPxSurfaceShadingNodeOverride** override.

6.6.5.1 Registration

The registration interfaces on MDrawRegistry are:

- registerSurfaceShadingNodeOverrideCreator()
- deregisterSurfaceShadingNodeOverrideCreator()

The classification string must start with the string "**drawdb/shader/surface**" for it to be recognized as a surface shader. As shown in the registration code, the same classification string used to register the override must exist on the shader node to form an association.

```
MString OnbShader::drawDbClassification("drawdb/shader/surface/onbShader");
// Include the override classification as part of the shader node classification
MString OnbShader::classification("shader/surface:" + drawDbClassification);

plugin.registerNode(
               OnbShader::nodeName,
               OnbShader::id,
               OnbShader::creator,
               OnbShader::initialize,
               MPxNode::kDependNode,
               &OnbShader::classification));

MHWRender::MDrawRegistry::registerSurfaceShadingNodeOverrideCreator(
               OnbShader::drawDbClassification,
               sRegistrantId,
               onbShaderOverride::creator));
```

6.6.5.2 Fragment Definition

For a surface shader that handles lights, there are a number of inputs that can be automatically bound by Maya in order to "pass" internal information to the shader. These are referred to as "pass through" fragments.

In the *onbShader* example, an "onbFloat3PassThrough" fragment is declared to pass through 3-float values.

Additionally, a fragment is added that computes various dot products are required to compute specular and diffuse contributions. This fragment is called "onbShaderGeom". Example products include: normal and light vector, half-angle, view and half-angle.

The diffuse and specular computations are a simplified version of the Oren-Nayer model ("onbDiffuse" and "onbSpecular").

To handle **light binding,** a "light accumulator" fragment called "onb16LightAccum" exists in the example. This fragment takes as input an automatically generated Maya "light selector" fragment. The accumulator fragment indicates to Maya that an internal light loop must be performed, and to select a given light using the "selector" for each loop iteration. For example,

there is a selector called “mayaLightSelector16” that allows a maximum of 16 lights to be looped through

In the code below, the diffuse and specular irradiance is accumulated over a maximum of 16 lights. Note that the type is “**accum**” versus “**plumbing**”.

```
fragmentName = "onb16LightAccum";
fragmentBody =
"<fragment uiName=\"onb16LightAccum\" name=\"onb16LightAccum\" type=\"accum\"
class=\"ShadeFragment\" version=\"1.0\"> \r\n"
"      <description><![CDATA[Accumulates specular & diffuse irradiance for 16
lights]]></description> \r\n"
"      <properties> \r\n"
"             <float3 name=\"scaledDiffuse\" /> \r\n"
"             <float3 name=\"scaledSpecular\" /> \r\n"
"             <string name=\"selector\" /> \r\n"
"      </properties> \r\n"
"      <values> \r\n"
"             <string name=\"selector\" value=\"mayaLightSelector16\" /> \r\n"
"      </values> \r\n"
"      <outputs> \r\n"
"             <alias name=\"scaledDiffuse\" /> \r\n"
"             <alias name=\"scaledSpecular\" /> \r\n"
"      </outputs> \r\n"

"      <implementation> \r\n"

"      </implementation> \r\n"

"</fragment> \r\n";
```

An output structure is specified in order to support multiple outputs. This fragment is called “*onbShaderOutput*”. The output structure has parameters to output color, transparency, and surface (color with alpha).

The “glue” which connects the light contribution to the shading is a “combiner” fragment called “onbCombiner”. It takes in the computed diffuse and specular results and combines them with other shader fragments to compute the values for the output structure. Any ambient light contribution is also factored as an input parameter.

6.6.5.3 Fragment Graph Building

With the appropriate fragments specified, it is possible to form a surface shader fragment graph. The class type for this fragment is a “FragmentGraph”. The final graph can be described as follows:

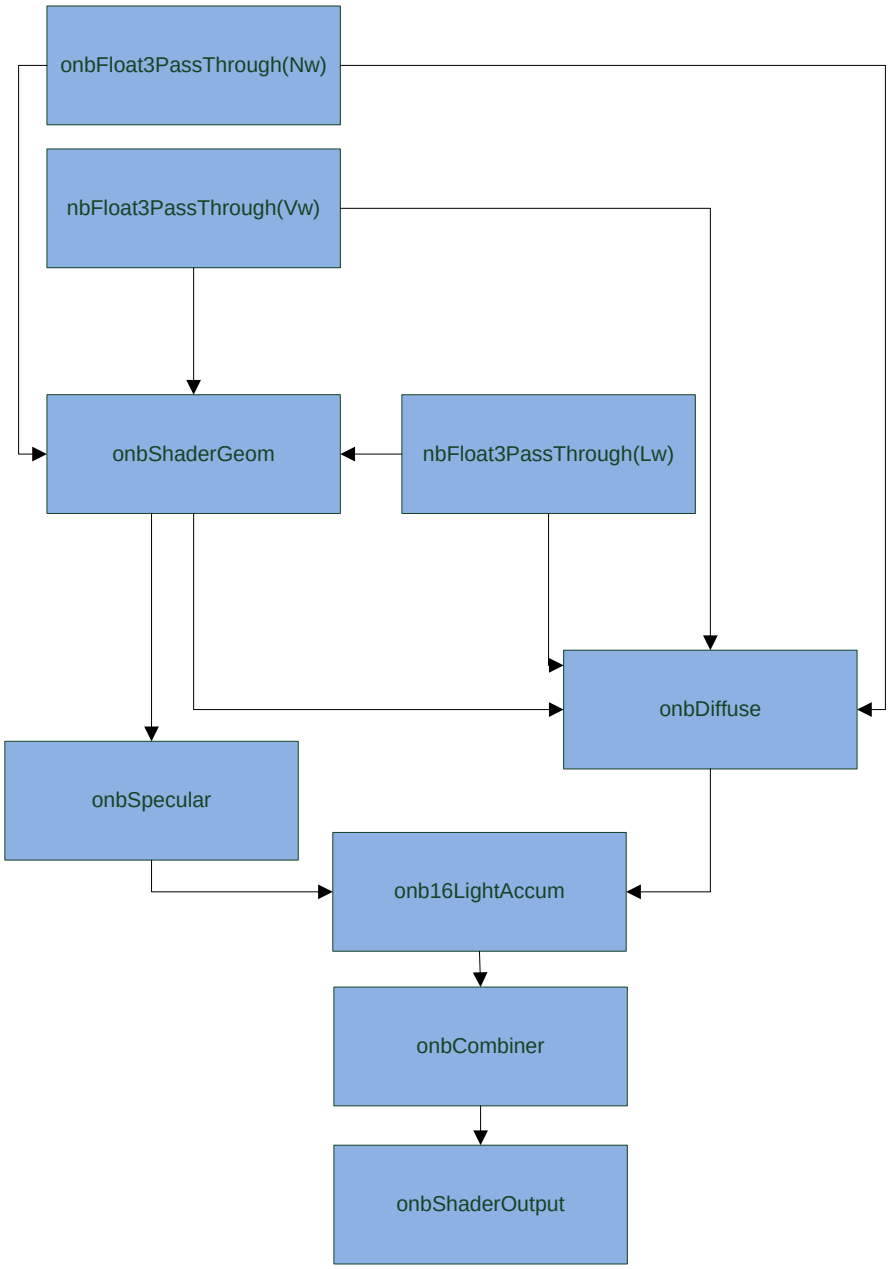


Figure 9: Fragment Graph for the onbShader example.

The fragment graph definition begins with specifying the referenced fragments. All of the previously mentioned fragments are referenced using the *<fragment ref name = "" />* notation within the <fragments> </fragments> section.

```
"<fragment_graph name=\"onbShaderSurface\" ref=\"onbShaderSurface\"
class=\"FragmentGraph\" version=\"1.0\"> \r\n"
"      <fragments> \r\n"
"            <fragment_ref name=\"nwPassThrough\" ref=\"onbFloat3PassThrough\" /> \r\n"
"            <fragment_ref name=\"vwPassThrough\" ref=\"onbFloat3PassThrough\" /> \r\n"
"            <fragment_ref name=\"lwPassThrough\" ref=\"onbFloat3PassThrough\" /> \r\n"
"            <fragment_ref name=\"onbShaderGeom\" ref=\"onbShaderGeom\" /> \r\n"
```

```
"            <fragment_ref name=\"onbDiffuse\" ref=\"onbDiffuse\" /> \r\n"

"            <fragment_ref name=\"onbSpecular\" ref=\"onbSpecular\" /> \r\n"
"            <fragment_ref name=\"onb16LightAccum\" ref=\"onb16LightAccum\" /> \r\n"
"            <fragment_ref name=\"onbCombiner\" ref=\"onbCombiner\" /> \r\n"
"            <fragment_ref name=\"onbShaderOutput\" ref=\"onbShaderOutput\" /> \r\n"
"      </fragments> \r\n"
```

The names used for the references can be chosen as desired.

These names can be used to form **connections** by specifying the name and the parameter name used for the connection. For example, the nwPassThrough fragments “output” parameter is connected to the “onbDiffuse” fragments “Nw” (normal world) parameter using the notation:

<connect from = “fragmentName.parameterName” to “fragmentName2.parameterName” name = “connection name”

This is done within the *<connections></connections>* section.

The following code builds the fragment graph shown in the Figure 15.

Of note is the connection between the light accumulator (“onb16LightAccum”) and the combiner (“onbCombiner”). There are actually no explicit connections to lights defined. These connections are done by the renderer internally based on the current set of active lights specified. This is a key difference between fragment based interfaces and effects based interface (such as MPxShaderOverride) in **that light binding is all handled automatically by the renderer**.

```
"      <connections> \r\n"
"            <connect from=\"nwPassThrough.output\" to=\"onbDiffuse.Nw\" name=\"Nw\" />
\r\n"
"            <connect from=\"vwPassThrough.output\" to=\"onbDiffuse.Vw\" name=\"Vw\" />
\r\n"
"            <connect from=\"lwPassThrough.output\" to=\"onbDiffuse.Lw\" name=\"Lw\" />
\r\n"
"            <connect from=\"nwPassThrough.output\" to=\"onbShaderGeom.Nw\" name=\"Nw\"
/> \r\n"
"            <connect from=\"vwPassThrough.output\" to=\"onbShaderGeom.Vw\" name=\"Vw\"
/> \r\n"
"            <connect from=\"lwPassThrough.output\" to=\"onbShaderGeom.Lw\" name=\"Lw\"
/> \r\n"

"            <connect from=\"onbShaderGeom.onbShaderGeom\"
to=\"onbDiffuse.shaderGeomInput\" name=\"shaderGeomInput\" /> \r\n"
"            <connect from=\"onbShaderGeom.onbShaderGeom\"
to=\"onbSpecular.shaderGeomInput\" name=\"shaderGeomInput\" /> \r\n"

"            <connect from=\"onbDiffuse.onbDiffuse\"
to=\"onb16LightAccum.scaledDiffuse\" name=\"scaledDiffuse\" /> \r\n"
"            <connect from=\"onbSpecular.onbSpecular\"
to=\"onb16LightAccum.scaledSpecular\" name=\"scaledSpecular\" /> \r\n"

"            <connect from=\"onb16LightAccum.scaledDiffuse\"
```

```
to=\"onbCombiner.diffuseIrradIn\" name=\"diffuseIrradIn\" /> \r\n"
"             <connect from=\"onb16LightAccum.scaledSpecular\"
to=\"onbCombiner.specularIrradIn\" name=\"specularIrradIn\" /> \r\n"
"             <connect from=\"onbCombiner.onbCombiner\"

to=\"onbShaderOutput.onbShaderOutput\" name=\"onbShaderOutput\" /> \r\n"
"      </connections> \r\n"
```

Properties are specified to expose graph parameters with specific names. For example “nwPassThrough.input” is exposed as “Nw”. Additionaly, automatic binding can be indicated using the “semantic” keyword. In the example the “Nw” semantic indicates to auto-bind world space normals and the “flags=varyingInputParam” indicates it varies per vertex.

Values specify initial values. Of note here is that the exposed “selector” parameter is set to the value: “mayaLightSelector16” (16 light selector).

The final output structure name for the graph (“onbShaderOutput”) is also specified to indicate which one of all the possible outputs returns the final shaded values.

```
"      <properties> \r\n"
"             <float3 name=\"Nw\" ref=\"nwPassThrough.input\" semantic=\"Nw\"
flags=\"varyingInputParam\" /> \r\n"
"             <float3 name=\"Vw\" ref=\"vwPassThrough.input\" semantic=\"Vw\"
flags=\"varyingInputParam\" /> \r\n"
"             <float3 name=\"Lw\" ref=\"lwPassThrough.input\" /> \r\n"
"             <float3 name=\"HLw\" ref=\"onbShaderGeom.HLw\" /> \r\n"
"             <float3 name=\"diffuseI\" ref=\"onbDiffuse.diffuseI\" /> \r\n"

"             <float name=\"roughness\" ref=\"onbDiffuse.roughness\" /> \r\n"
"             <float3 name=\"specularI\" ref=\"onbSpecular.specularI\" /> \r\n"
"             <float3 name=\"specularColor\" ref=\"onbSpecular.specularColor\" /> \r\n"
"             <float name=\"eccentricity\" ref=\"onbSpecular.eccentricity\" /> \r\n"
"             <float name=\"specularRollOff\" ref=\"onbSpecular.specularRollOff\" />
\r\n"
"             <string name=\"selector\" ref=\"onb16LightAccum.selector\" /> \r\n"
"             <float3 name=\"color\" ref=\"onbCombiner.color\" /> \r\n"
"             <float3 name=\"transparency\" ref=\"onbCombiner.transparency\" /> \r\n"
"             <float3 name=\"ambientColor\" ref=\"onbCombiner.ambientColor\" /> \r\n"
"             <float3 name=\"ambientIn\" ref=\"onbCombiner.ambientIn\" /> \r\n"
"             <float3 name=\"incandescence\" ref=\"onbCombiner.incandescence\" /> \r\n"
"      </properties> \r\n"
"      <values> \r\n"
"             <float3 name=\"Lw\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float3 name=\"HLw\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float3 name=\"diffuseI\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float name=\"roughness\" value=\"0.3\" /> \r\n"
"             <float3 name=\"specularI\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float3 name=\"specularColor\" value=\"1.0,1.0,1.0\" /> \r\n"
"             <float name=\"eccentricity\" value=\"0.1\" /> \r\n"
"             <float name=\"specularRollOff\" value=\"0.7\" /> \r\n"
"             <string name=\"selector\" value=\"mayaLightSelector16\" /> \r\n"
"             <float3 name=\"color\" value=\"0.5,0.5,0.5\" /> \r\n"
"             <float3 name=\"transparency\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float3 name=\"ambientColor\" value=\"0.0,0.0,0.0\" /> \r\n"
```

```
"             <float3 name=\"ambientIn\" value=\"0.0,0.0,0.0\" /> \r\n"
"             <float3 name=\"incandescence\" value=\"0.0,0.0,0.0\" /> \r\n"
"      </values> \r\n"
"      <outputs> \r\n"
"             <struct name=\"onbShaderOutput\" ref=\"onbShaderOutput.onbShaderOutput\" />
\r\n"
"      </outputs> \r\n"

"</fragment_graph> \r\n";
```

In the example, all input parameter names match attribute names, and as such, no mappings are required.

6.6.6 Custom Effect Nodes (MPxShaderOverride)

The amount of work required to add support for shaders using an **MPxShaderOverride** is a reflection of the complexity required to support custom shading nodes. In addition to handling drawing, the override may potentially have to include code for parsing, compiling and binding shaders, as well as handling all associated input resources, state management and integration with the internal pipeline. It can basically be a renderer in itself.

By comparison, a shader instance (MShaderInstance) can take advantage of the internal shader manager, as well as take advantage of fragments and the fragment manager. It is still possible for MPxShaderOverride to take advantage of the internal shader mechanisms to a certain extent if it uses MShaderInstances. This interface can not take direct advantage of the fragment system nor the internal fragment building logic to automatically evaluate any connected nodes.

A simplified comparison between the two strategies is shown below:

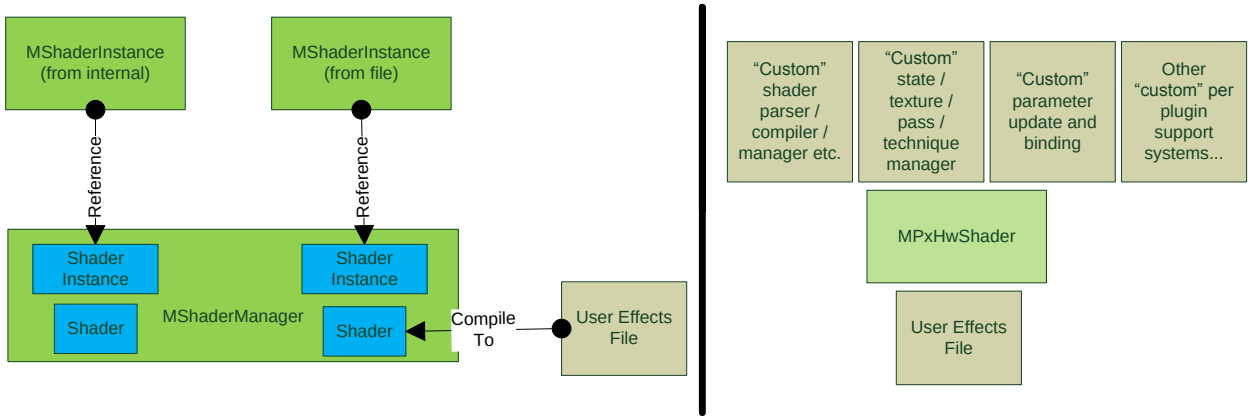


Figure 10

In some cases, it may be required to take full control of rendering. Some possible reasons include:

- Requirement for custom light types
- Requirement for custom geometry or tessellation shaders

- Requirement for custom semantics or annotations
- Requirement for custom packing of data (data or indexing streams)
- Requirement for custom shader building
- Requirement to draw API specific features

To illustrate the complexity of this interface, the following sections will discuss two implementations: One that uses an internal shader instance and one that writes an independent shader system.

6.6.6.1 Basic Connections

The following section discusses the basic layout of how a shader override interacts with Maya nodes, render items and geometry requirements.

It continues to use the model wherein an override is associated with a DG node by **classification**. In this case, the DG node is a plug-in hardware shader. Assume that there is no difference between the two existing VP1 interfaces. The details of how these interfaces work will not be covered here.

Shader overrides will "*produce*" a shader instance. This can either be an explicit one returned by the plug-in (MShaderInstance) or an internal one which is used to allow association with a render item (MRenderItem) for an object. The association between a shader instance and a render item is generally determined by the shader assignments specified via DG connections between a shading engine and a given object.

As with any shader, a series of geometry stream or indexing requirements must be specified when an object needs to update the data required for its associated render items.

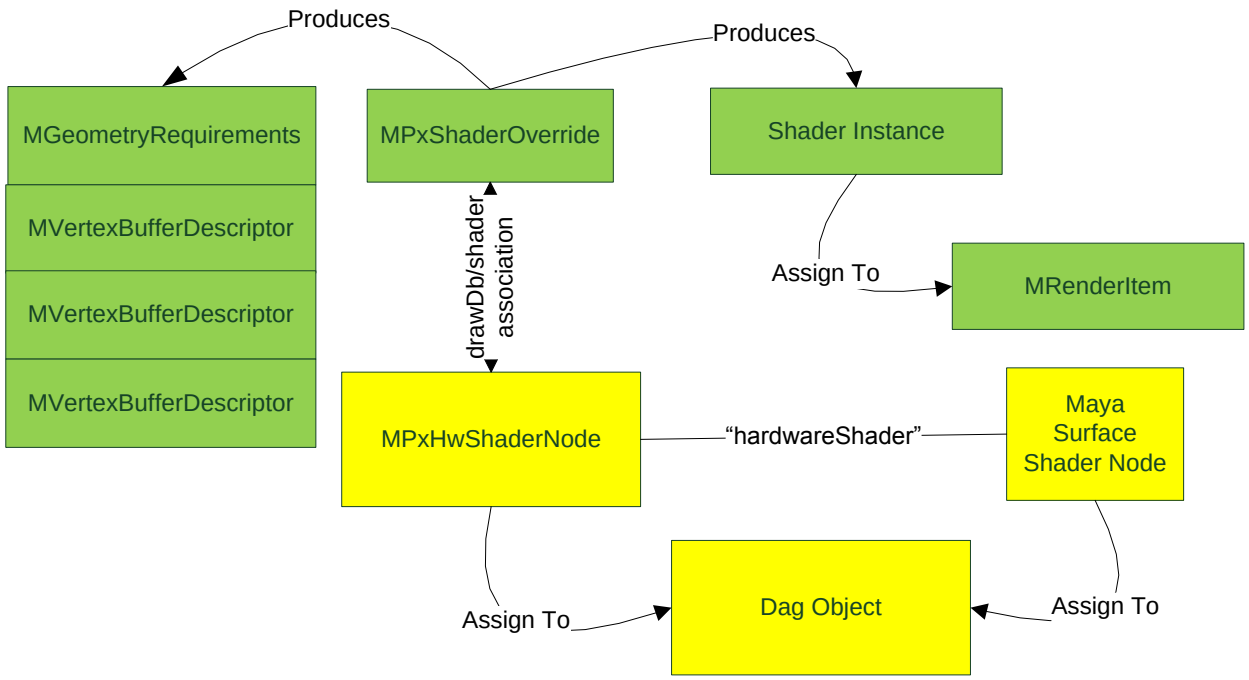


Figure 11: Basic associations for an MPxShaderOverrde. Usage for one render item is shown, but there can be 0 or more render items that use a shader based on Maya shading node assignments. 0 is when the shader is not assigned to any geometry. Only stream (vertex buffer requirements) are shown, but index buffers could also be required.

6.6.6.2 Starting Example

The developer kit *vp2BlinnShader* plug-in example demonstrates basic support. To reduce complexity, the plug-in uses internal utilities while still attempting to show a complete override implementation. The greater complexity inherent with more customization will be covered via different examples.

6.6.6.3 Attribute Specification

Unlike an MPxShadingNodeOverride or MPxSurfaceShadingNodeOverride, there is no built in monitoring for attributes on a hardware shader node that performs name matching and automated update. It is up to the plug-in writer to determine the attributes that are appropriate and map them accordingly. Any mapped (upstream) node connections, whether to an internal node or a plug-in node, will never be automatically evaluated for the MPxShaderOverride as part of internal shader evaluation.

The *vp2BlinnShader* example adds a few *static* attributes for simple mapping for diffuse color and specular color and transparency. This is done as part of the static initialization method for the hardware shader node (**MPxNode**::**initialize**()).

Dynamic attribute creation in VP2 follows the same timing as it does for VP1. The **MUniformParameter** utility classes can be created and used in VP2 in the same manner as for VP1. (For further details on MUniformParameter usage for VP2, see Uniform Data Handling.)

As the logic for the node's **compute**() method is based on software rendering and is performed independently of any hardware rendering, no further details will be provided here.

6.6.6.3.1 Override Registration

The first requirement is registration. This requires that a "**drawdb/shader**" classification be specified for both the node (*vp2BlinnShader*) and the shader override (*vp2BlinnShaderOverride*) to form the proper association.

In this example, the classification "*drawdb/shader/surface/vp2BlinnShader*" is used.

A shader override is registered using the **MDrawRegistry::registerShaderOverrideCreator**() method.

Note that the "*drawdb*" classification is not used for the purposes of filtering any UI as the node classification does ("*shader/surface/utility*" here). The registration id, as with other overrides, simply provides a unique identifier. Each time a new hardware shader node is created, the "**Creator**()" method will create a new vp2BlinnShaderOverride instance.

```
static const MString svp2BlinnShaderRegistrantId("vp2BlinnShaderRegistrantId");
const MString UserClassify(
"shader/surface/utility/:drawdb/shader/surface/vp2BlinnShader");

MFnPlugin plugin( obj, PLUGIN_COMPANY, "1.0", "Any");
plugin.registerNode( "vp2BlinnShader", vp2BlinnShader::id,
                                vp2BlinnShader::creator, vp2BlinnShader::initialize,
                                MPxNode::kHardwareShader, &UserClassify );

// Register a shader override for this node
MHWRender::MDrawRegistry::registerShaderOverrideCreator(
       "drawdb/shader/surface/vp2BlinnShader",
       svp2BlinnShaderRegistrantId,
       vp2BlinnShaderOverride::Creator);
```

The corresponding deregistration code uses the same classification string and registration id.

```
// Deregister a shader override for this node
MHWRender::MDrawRegistry::deregisterShaderOverrideCreator(
              "drawdb/shader/surface/vp2BlinnShader", svp2BlinnShaderRegistrantId);
```

6.6.6.3.2 Override "Phases"

A shader override has a series of "**phases**" that will be executed. For each of these phases, there are some key interfaces that are required. For a more information on phases, see *Effect Overrides* in the *Maya Developer Help*.

As an overview, the phases are summarized here. Key interfaces are in **bold**, with minimally required methods being underlined.

1. **Initialization**: This phase occurs whenever a shader is deemed as requiring a "rebuild". This could be for a number of reasons, including shader assignment change, and attribute value change. This is the phase responsible for:
    a. Geometry requirements specification

b. Shader “Key” generation, where a key is used to uniquely identify an instance of a shader. Unique keys are used when attempting to group render items to amortize shader setup / clean-up cost.
c. User data may be specified, but for the simple use case, it is not recommended that data be added.

Interfaces involved in this phase include:

1. virtual MString **initialize**(MObject shader);
2. virtual MString **initialize**(const MInitContext& initContext, MInitFeedback& initFeedback);
3. virtual bool overridesDrawState();
4. virtual bool supportsAdvancedTransparency() const;

- virtual MHWRender::MShaderInstance* nonTexturedShaderInstance(bool &monitorNode) const;

2. **Data Update**: This can be described in two parts, a DG data update and a device data update.
   a. DG node evaluation should always occur within the DG data update interface for thread-safety.
   b. The device update would generally be hardware resource update and shader parameter updates.

   Not all interfaces for this phase need to be overridden. If the shader is independent of any associated node attributes, the DG update method does not need to be overridden. If the shader parameters are static, then the device update method does not need to be overridden. Both methods have empty implementations by default.

   Interfaces involved in this phase include the following. Note that none have to be overridden for a static shader.

   1. virtual bool rebuildAlways();
   2. virtual void **updateDG**(MObject object);
   3. virtual void **updateDevice**();
   4. virtual bool isTransparent();
   5. virtual void **endUpdate**();

3. **Drawing**: Shader binding and unbinding are the responsibilities of the interfaces in this phase. This corresponds to the concept of “activating” the shader “key” provided at initialization time and “deactivating” the same key. Between a bind/unbind pair, one or more render items geometry may be drawn. Unless required, geometry drawing for a given render item should be performed via a provided utility interface.

   Interfaces involved in this phase include:

1. virtual bool handlesDraw(MDrawContext& context);
2. virtual void **activateKey**(MDrawContext& context, const MString& key);
3. virtual bool **draw**(MDrawContext& context,
   const MRenderItemList& renderItemList) const = 0;
4. virtual void **terminateKey**(MDrawContext& context, const MString& key);

Geometry requirements handling can be thought of as a "phase" that occurs sometime between shader update and drawing. The shader override itself does not know what render items it will be associated with and hence which DAG object's evaluator will be handling the update. The evaluator could be for either plug-in geometry or internal geometry or both.

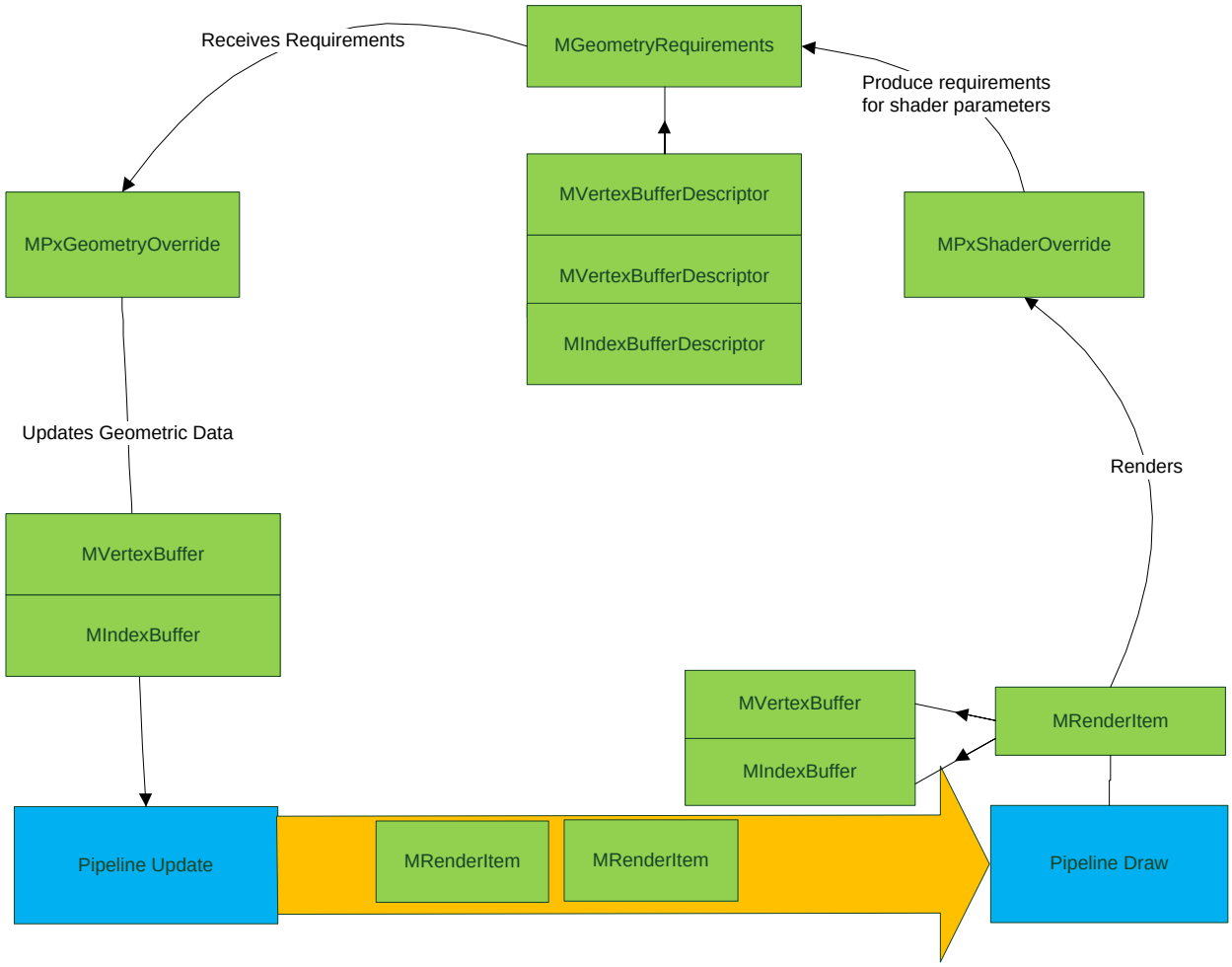


Figure 12: Diagram shows how the shader override specifies the requirements (descriptions), which are handled by an DAG object evaluator (a MPxGeometryOverride in this case). It will update the data for the appropriate render items. These items are shown "flowing" down the pipeline until the "draw phase" is reached for the shader override. At this time, the properly updated render items are sent back to the shader override.

More complex plug-in examples will be examined to discuss the complexities of each phase.

6.6.6.3.3 "Phase" Support in VP1

By comparison, the interfaces for VP1 hardware shaders are similar to those of VP2, but have different execution time and frequency of execution. Of note is that, almost all interfaces are

called at draw time, and hence the lack of any explicit DG, device update and draw only interfaces. The concept of explicit shader identification (key) also does not exist. VP1 hardware shaders can basically be thought of as providing custom draw request handling when a VP1 UI shape is asked to draw itself.

The concept of “batching” geometry calls between shader bind/unbind is similar, although the granularity in VP1 is for blocks of CPU geometry instead of render items (and GPU geometry). VP1 data blocks mostly have a 1:1 correspondence with per object or per object component shader assignments. As previously noted, VP2 a render item does not necessarily require this correspondence (for example, a user generated render item, or a render item after consolidation).

The VP1 **MPxHardwareShader** interface can be described as follows:

- Initialization:
  - No concept and no shader key concept.
- Data Update:
  - Part of draw.
- Drawing: Has no concept of device and DG separation. DG evaluation generally ends up occurring at arbitrary places, including at draw time.
  - Get geometry requirements from the shader via **populateRequirements**().
  - Check for transparency via **transparencyOptions**() [roughly a combination if MPxShaderOverride isTransparent() and overridesDrawState()].
  - Draw using the **render**(MGeometryList& iterator) method. Iterator is for the batching concept.

The older **MPxHwShaderNode** interface, which is no longer recommended for use with VP1, can be described as follows:

- Initialization:
  - No concept and no shader key concept.
- Data Update:
  - Part of draw.
- Drawing: Has no concept of device and DG separation. DG evaluation generally ends up occurring at arbitrary places including at draw time.
  - Get geometry requirements, which are limited to asking for normal spaces, color and UV set counts: **normalsPerVertex**(), **getColorSetNames**(), **getTexCoordSetNames**().
  - Check for transparency via **transparencyOptions**().
  - Draw using **bind**(), **unbind**() and **geometry**() and OpenGL specific equivalent methods. There is no batching concept here.

6.6.6.3.4 Basic “Phase” Support: Initialization

In the *vp2BlinnShader* example, the **MPxShaderOverride::initialize**() method uses a single MShaderInstance class member called *fColorShaderInstance*. The shader key is a constant

value, as the number and type of shader parameters as well as geometry stream requirements never change.

```
if (fColorShaderInstance)
{
	// This plugin is using the utility method
	// MPxShaderOverride::drawGeometry(). For DX11 drawing,
	// a shader signature is required. We use
	// the signature from the same MShaderInstance used to
	// set the geometry requirements so that the signature
	// will match the requirements.
	//
	addShaderSignature( *fColorShaderInstance );

	// Set the geometry requirements based on the shader instance
	setGeometryRequirements( *fColorShaderInstance );
}
// Return constant shader key
return MString("Autodesk Maya vp2 Blinn Shader Override");
```

The geometry requirements can be obtained from the shader instance using the **MPxShaderOverride::setGeometryRequirements**() utility method. However, due to the nature of DirectX11 shading binding, a specific shader signature must be specified. The utility method **MPxShaderOverride**::**addShaderSignature**() is used to obtain this from the shader instance.

6.6.6.3.5 Base “Phase” Support: Update

Using the attributes defined for the hardware shader node for the vp2BlinnShader example, the values need to be cached during DG update within the **MPxShaderOverride::updateDG**() methods, and then bound during the device update method **MPxShaderOverride::updateDevice**(). As stated previously, if this shader had static parameter values, then these virtual methods would not need to be overridden.

The code for the DG update is a straight forward evaluation of the color, transparency and specular attributes to cache to the *fDiffuse*, *fTransparency* and *fSpecular* data members on the override.

```
virtual void updateDG(MObject object)
{
	// Get the hardware shader node from the MObject.
	vp2BlinnShader *shaderNode = (vp2BlinnShader *)
			MPxHardwareShader::getHardwareShaderPtr( object );

	// Cache any data from the node to local data members.
	MStatus status;
	MFnDependencyNode node(object, &status);
	if (status)
	{
		node.findPlug("colorR").getValue(fDiffuse[0]);
		node.findPlug("colorG").getValue(fDiffuse[1]);
		node.findPlug("colorB").getValue(fDiffuse[2]);
		node.findPlug("transparency").getValue(fTransparency);
		fDiffuse[3] = 1.0f - fTransparency;
```

```
                node.findPlug("specularColorR").getValue(fSpecular[0]);
                node.findPlug("specularColorG").getValue(fSpecular[1]);
                node.findPlug("specularColorB").getValue(fSpecular[2]);
        }
}
```

The shader update is also straight forward, using MShaderInstance parameter setting methods to update from values cached during a DG update. To allow for transparent drawing, the shader instance is set as transparent.

```
virtual void updateDevice()
{
        // Update shader to mark it as drawing with transparency or not.
        fColorShaderInstance->setIsTransparent( isTransparent() );
        fColorShaderInstance->setParameter("diffuseColor", &fDiffuse[0] );
        fColorShaderInstance->setParameter("specularColor", &fSpecular[0] );
}
```

After the updates have been processed, the override can provide a transparency “hint” by overriding the MPxShaderOverride::**isTransparent**() method. The example overrides this and uses the cached fTransparency value as an indicator:

```
virtual bool isTransparent()
{
        return (fTransparency > 0.0f);
}
```

6.6.6.3.6 Base “Phase” Support: Draw

Assuming that the DAG objects have updated the geometry for the render items using this shader at draw time, the draw interfaces on the override are called.

Since the vp2BlinnShader example is using a MShaderInstance, it should return that instance by overriding the MPxShaderOverride::shaderInstance() method.

```
virtual MHWRender::MShaderInstance* shaderInstance() const
{
        return fColorShaderInstance;
}
```

This will trigger the appropriate setup for an internal shader instance.

MPxShaderOverride::**activateKey**(), **MPxShaderOverride::terminateKey**() will be called to allow for the plug-in to bind and unbind the shader instance. The MShaderInstance::bind() method is used to set up the shader instance.

```
virtual void activateKey(MHWRender::MDrawContext& context, const MString& key)
{
        // Bind the shader
        fColorShaderInstance->bind( context );
}
```

The **terminateKey**() method will unbind the shader within the **MShaderInstance::unbind()** method.

```
virtual void terminateKey(MHWRender::MDrawContext& context, const MString& key)
{
	// Unbind the shader
	fColorShaderInstance->unbind( context );
}
```

MPxShaderOverride::draw() can allow custom drawing. In this example, the utility method **MPxShaderOverride::drawGeometry**() can simply be used. For completeness, all passes on the shader instance are looped through, even though, in this case the shader has only one pass. The drawGeometry() method does not need to loop through the render item list as it can access this data internally.

```
virtual bool draw(MHWRender::MDrawContext& context,
                          const MHWRender::MRenderItemList& renderItemList) const
{
	// Activate all the shader passes and draw using internal draw methods.
	unsigned int passCount = fColorShaderInstance->getPassCount( context );
	for (unsigned int i=0; i<passCount; i++)
	{
		fColorShaderInstance->activatePass( context, i );
		MHWRender::MPxShaderOverride::drawGeometry(context);
	}
	return true;
}
```

6.6.6.3.7 Override Swatch Rendering

Swatch rendering still resides within either MPxHwShaderNode::renderSwatchImage() or MPxHardwareShader::renderSwatchImage().

Non-VP2 based swatch rendering relies on the use of the *MHardwareRenderer* class which is based on the legacy hardware renderer. Utilities are provided to get swatch parameters such as light and background parameters and the ability to extract CPU stock geometry. All drawing must be done explicitly by the plug-in, and if the utility context is used, then it must be drawn using OpenGL.

The recommended interface for VP2 swatch rendering is **MRenderUtilities::renderMaterialViewerGeometry**(). The capabilities of this interface are based on the Material Viewer in the Hypershade. The interface only requires the specification of the environment, removing any requirement for data handling and drawing.

In the *vp2BlinnShader* example, the “shader ball” geometry is used if the size of the swatch is large enough; otherwise a stock sphere is used. The swatch camera and single directional light are also used.

```
MStatus vp2BlinnShader::renderSwatchImage( MImage & outImage )
```

```
{
	MString meshSphere("meshSphere");
	MString meshShaderball("meshShaderball");

	// Find out the size of the swatch to render
	unsigned int targetW, targetH;
	outImage.getSize(targetW, targetH);

	// Render the swatch
	return MHWRender::MRenderUtilities::renderMaterialViewerGeometry(
			targetW > 128 ? meshShaderball : meshSphere,
			thisMObject(),
			outImage,
			MHWRender::MRenderUtilities::kPerspectiveCamera,
			MHWRender::MRenderUtilities::kSwatchLight);
}
```

The following snapshot shows an example of the geometry in the scene (on the left) and the swatch render image (on the right).

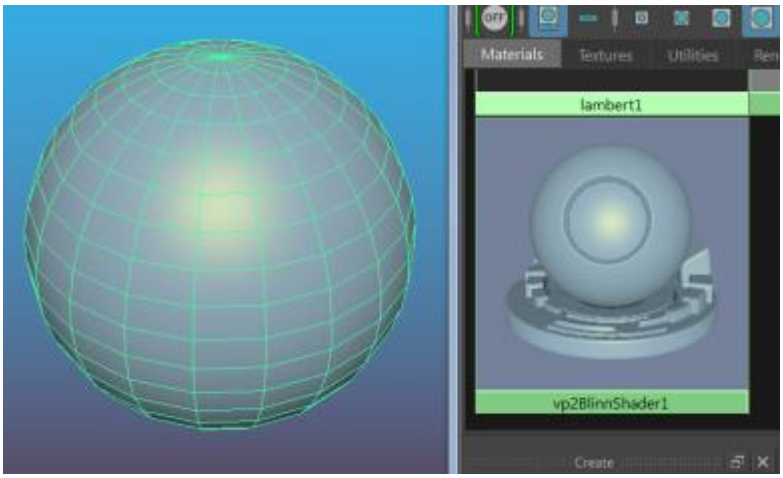


Figure 13: Swatch for vp2BlinnShader instance on the right. Material attributes are from the shader instance but camera, lighting and geometry are internally provided.

For reference, the *hwPhongShader* plug-in example can be examined to compare the usage of the VP2 interface versus the raw OpenGL drawing that was used previously in VP1.

6.6.6.4 Initialization: Shader Uniqueness and Rebuild Logic

The first example used an MShaderInstance that was essentially a static parameter description internally. Thus, a static shader key was used.

This will not be true if the override is meant to handle a generic set of shaders. This is the case with the sample *cgFxShader*, *dx11Shader*, and *glslShader* plug-ins which support CgFx, FX and **OGSFX** effects frameworks respectively. Depending on the flexibility a given plug-in wishes to provide, a unique key should be generated. Assuming that the plug-in supports an effects framework, this key could be based on the following factors:

- Differing effects file name (if from disk), string content (if from buffer), or binary identifier (if from precompiled Effect)

- Differing effect “techniques”
- Differing number of effect “passes”

More generally, this can include:

- Variations in uniform or varying parameter inputs
- Variations in usage and combination of code for various shader stages (vertex, geometry, tessellation, pixel)

The Autodesk defined OGSFX effect format supports multiple shader language implementations, and thus differences in language will also be a variation.

For example, for the *glslShader* plug-in, the key definition includes the shader name, effect file name, and technique. The effect file extension is used to indicate shading language. The same key combination is used for the dx11Shader.

```
// Build key string, note that if any attribute on the node changes that
// would affect the value of this string, then we must trigger rebuild of
// the shader
MString result = MString("Autodesk Maya GLSLShaderOverride, nodeName=");
result += fShaderNode ? fShaderNode->name() : MString("null");
result += MString(", effectFileName=");
result += fShaderNode ? fShaderNode->effectName() : MString("null");
result += MString(", technique=");
result += (fShaderNode) ? fShaderNode->techniqueName() : MString("null");
return result;
```

For all shaders mentioned above, an effect change means that a new key is required and hence re-initialization is required. As such, the **MPxShaderOverride::rebuildAlways**() method will return “true” to indicate that the initialization phase must be re-invoked.

Rebuild is also required when attribute / parameter changes occur during the update phase, which means a different shader is required. For example, changing a “technique” attribute for an effect means that the shader will become different. This type of check is similar to the **MPxShadingNodeOverrides::valueChangeRequiresFragmentRebuild()** indicator which forces the re-initialization of fragment based shaders.

Care should be taken to determine the minimal frequency for rebuilding, as this is roughly equivalent to reassigning the shader to all affected render items, and hence can trigger geometry rebuilding on all affected items (DAG object evaluator re-evaluation).

If a rebuild/re-initialization is not required, then attribute changes can continue to only invoke update and draw phase execution.

6.6.6.5 Initialization: Explicitly Specification of Varying Data

In the case where the geometry requirements cannot be determined based on MShaderInstance usage, it is up to the plug-in to determine the varying parameter data requirements and use **MPxShaderOverride::addGeometryRequirement**() for each varying parameter using a suitable **MVertexBufferDescriptor** description instance each time.

See the *hwColorPerVertexShader* plug-in for a simple example, where a sample position description is being added as a requirement.

```
// Set position requirement
MString reqName;
addGeometryRequirement(
       MHWRender::MVertexBufferDescriptor(
              reqName,
              MHWRender::MGeometry::kPosition,
              MHWRender::MGeometry::kFloat,
              3));
```

When multiple streams of the same type are not required, an empty name should be specified for the description. This example requests the default position stream with data returned as 3 floating point (x,y,z) values.

Not all combinations of semantic, data type and dimension are available, and the **addGeometryRequirement**() C++ API Reference documentation should be referenced for the available options. Unless custom data is available, it is advisable to *reuse the basic formats provided by internal geometry*. This is the same for both VP1 and VP2 interfaces. For example, texture coordinates are a property of Maya's polygonal object representation, and are always 2-float (u,v) values regardless of which interfaces are used to access the data.

A name specification makes the shader more geometry dependent, and as such a named set may or may not exist for all geometry instances that use the shader. In the same example, usage of a color set name (as reqName) does not guarantee that the data will be available.

```
addGeometryRequirement(
       MHWRender::MVertexBufferDescriptor(
              reqName,
              MHWRender::MGeometry::kColor,
              MHWRender::MGeometry::kFloat,
              4));
```

For internally defined DAG object evaluators, an attempt will be made to either use a "default" set or some fallback data but this is not guaranteed. There is also no guarantee that a plug-in shape will provide the appropriate data. It is thus up to the plug-in shader to handle situations where data may not be available. This same logic applies for VP1 interfaces.

The previously mentioned usage of a DX11 shader signature specification (MPxShaderOverride::addShaderSignature() is not necessary for the *dx11Shader* as it does not use an MShaderInstance and the input layout is handled at draw time by the shader. The *glslShader* (like the *vp2BlinnShader*) plug-in uses an MShaderInstance and hence requires a specification if reading in a DX11 FX files.

6.6.6.6 Initialization: Custom Geometry Specification

For more details on how to handle advanced use cases, such as:

- Allowing for customized data streams to be provided.
- Allowing for custom data formats.
- Allowing for data repackaging.

See *Customizing Geometry Data for Shaders* in the *Maya Developer Help*. All additional requirements must be specified at initialization time, with any support classes such as mutators registered during plug-in initialization.

The *hwPhongShader* plug-in is an example that shows sample code custom primitives (using the *customPrimitiveGenerator* plug-in), as well as custom data repackaging (using the *vertexBufferMutator* plug-in).

The *dx11Shader* and *glslShader* plug-ins each use custom primitive generators which are supplied as part of the plug-in to create the appropriate data for hardware tessellation.

Previously, Figure 18 demonstrated the requirements for passing to evaluators. The next figure adds the locations data repackaging (**MPxVertexBufferMutator**), and custom data formats (**MPxPrimitiveGenerator** for indexing + **MPxVertexBufferGenerator** for data). Instances of each class can be registered via MDrawRegistry. The name used for registration can be used as the *semantic name* for data or index descriptors when filling in geometry requirements.

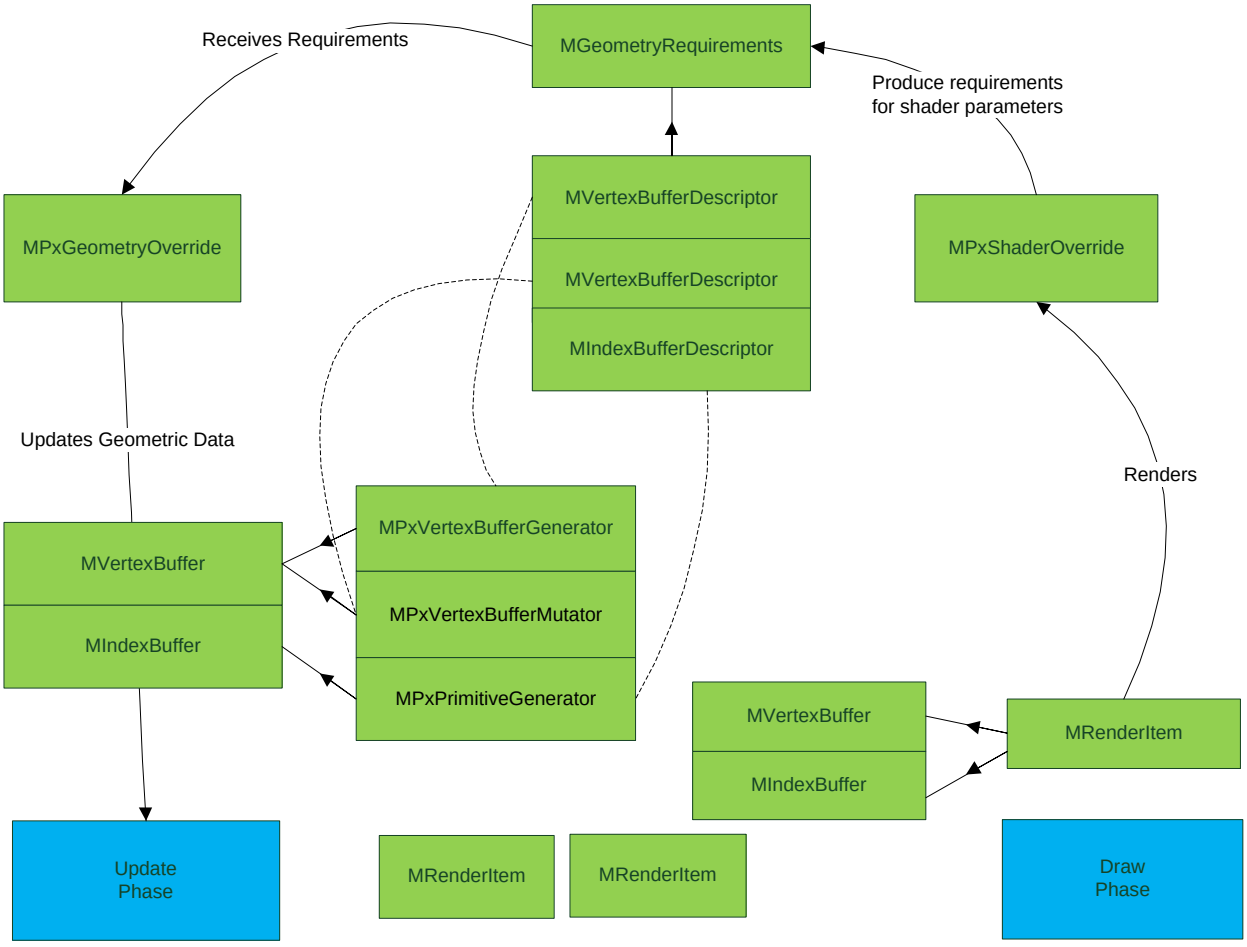


Figure 14: An MPxVertexBufferGenerator class instance can fill in data for a data stream (MVertexBuffer). It is registered and can be specified as the semantic name for an MVertexBufferDescriptor. An MPxPrimitiveGenerator class instance can fill in data for an indexing stream (MIndexBuffer). It is registered and can be specified as the semantic name for a MIndexBufferDescriptor. Data repackaging can be performed on an existing data stream using

an MPxVertexBufferMutator. Once registered, it can be specified as the semantic name for an MVertexBufferDescriptor.

In the *hwPhongShader* example a custom primitive type (indexing requirement) called "*customPrimitiveTest*" is created, which is the name of a registered primitive generator. The "*customPositionStream"* and "*customNormalStream"* semantic names are used to reference custom stream generators.

```
MString customPrimitiveName("customPrimitiveTest");
MHWRender::MIndexBufferDescriptor indexingRequirement(
       MHWRender::MIndexBufferDescriptor::kCustom,
       customPrimitiveName,
       MHWRender::MGeometry::kTriangles);

addIndexingRequirement(indexingRequirement);

MHWRender::MVertexBufferDescriptor positionDesc(
       empty,
       MHWRender::MGeometry::kPosition,
       MHWRender::MGeometry::kFloat,
       3);
positionDesc.setSemanticName("customPositionStream");

MHWRender::MVertexBufferDescriptor normalDesc(
       empty,
       MHWRender::MGeometry::kNormal,
       MHWRender::MGeometry::kFloat,
       3);
normalDesc.setSemanticName("customNormalStream");

addGeometryRequirement(positionDesc);
addGeometryRequirement(normalDesc);
```

The example also demonstrates an alternate specification for data by adding a mutator to swizzle the position data. If the mutator is not found, then no mutation is performed. The normal stream is not affected by the mutator, as no semantic name is specified.

```
MHWRender::MVertexBufferDescriptor positionDesc(
       empty,
       MHWRender::MGeometry::kPosition,
       MHWRender::MGeometry::kFloat,
       3);
// Use the custom semantic name "swizzlePosition"
// When the vertexBufferMutator plugin is loaded,
// this will swap the x,y and z values of the vertex buffer.
positionDesc.setSemanticName("swizzlePosition");

MHWRender::MVertexBufferDescriptor normalDesc(
       empty,
       MHWRender::MGeometry::kNormal,
       MHWRender::MGeometry::kFloat,
       3);

addGeometryRequirement(positionDesc);
addGeometryRequirement(normalDesc);
```

6.6.6.7 Initialization: Using Effect Annotations

The dx11Shader and glslShader plug-ins both have implementations that are based on the contents of a shader effect.

At initialization time, both plug-ins make use of technique level annotations to determine certain characteristics of the shader. This includes: the type of indexing (tessellation requirement), transparency algorithm supported, and state override indication. As the glslShader plug-in uses an MShaderInstance to represent the shader, various utilities are available for annotation parsing at the parameter, pass and technique level.

For example, in the *glslShader* example, after the MShaderInstance has been loaded, **MShaderInstance::techniqueAnnotationAsString()** is used to parse transparency, custom indexing and transparency state override. Note that *glslShaderAnnotationValue* is used to keep track of custom annotations for this shader.

```
// Setup Transparency using technique annotation
//
fTechniqueIsTransparent = false;
const MString transparency = fGLSLShaderInstance-
>techniqueAnnotationAsString(glslShaderAnnotation::kTransparency, opStatus);
if (opStatus == MStatus::kSuccess)
{
	fTechniqueIsTransparent = (strcmp(transparency.asChar(),
glslShaderAnnotationValue::kValueTransparent)==0);
}

// Setup index buffer mutators using annotations
//
fTechniqueIndexBufferType = MString();
const MString indexBufferType = fGLSLShaderInstance-
>techniqueAnnotationAsString(glslShaderAnnotation::kIndexBufferType, opStatus);
if (opStatus == MStatus::kSuccess)
{
	fTechniqueIndexBufferType = indexBufferType;
}

// Query technique if it should follow the Maya transparent object rendering or is self-
// managed (multi-passes)
fTechniqueOverridesDrawState = false;
MString overridesDrawState = fGLSLShaderInstance-
>techniqueAnnotationAsString(glslShaderAnnotation::kOverridesDrawState, opStatus);
if (opStatus == MStatus::kSuccess)
{
	fTechniqueOverridesDrawState = (strcmp(overridesDrawState.toLowerCase().asChar(),
glslShaderAnnotationValue::kValueTrue)==0);
}
```

The dx11Shader uses DX11 code to directly parse its internally kept DX11 effect instance.

The next section examines uniform handling, and discusses how annotations can be used at the parameter level.

6.6.6.8 Uniform Data Handling: Initialization

If an effects file is used, then the uniform parameters need to be parsed to determine if they have any semantics or annotations.

The semantics can be stored to determine if any data supplied by the renderer can be bound during the draw or update phases.

If the uniform is exposed as an attribute, then the annotations can be parsed to determine the formatting for the attribute in the Attribute Editor, such as, to determine the minimum and maximum range values for a slider used to control an attribute.

The dx11Shader and glslShader plug-ins both use **MUniformParameter** instances for attribute handling and to parse semantics and annotations to update these instances upon an effects file change.

If an MShaderInstance is used, the semantic and annotation utility methods can be used.

An example can be found in the *glslShader* implementation. In the code snippet below, the **MShaderInstance::annotationAsInt**() method is used to check for annotations that represent UI minimum and maximum values on the MShaderInstance *fGLSLShaderInstance* , and the result is stored in an MUniformParameter called uniformParam. Other annotations can be extracted as appropriate.

```
MUniformParameter uniformParam;

uniformParam.setUIHidden(false);

// Set UIMin and UIMax on the uniform parameter instance based
// on values extracted out from the shader instance
int uiMin = fGLSLShaderInstance->annotationAsInt(uniformParam.name(),
glslShaderAnnotation::kUIMin, opStatus);
if (opStatus == MStatus::kSuccess)
{
	uniformParam.setRangeMin(uiMin);
}
int uiMax = fGLSLShaderInstance->annotationAsInt(uniformParam.name(),
glslShaderAnnotation::kUIMax, opStatus);
if (opStatus == MStatus::kSuccess)
{
	uniformParam.setRangeMax(uiMax);
}
```

Note that this code sets the name for the uniformParam to be *the same name as the MShaderInstance parameter name*.

This is done as part of the logic to loop over each of the parameters on the MShaderInstance to extract the semantic for a given parameter using **MShaderInstance::semantic()**, as well as any annotation information.

```
// Get the parameter list from the shader
MStringArray params;
fGLSLShaderInstance->parameterList(params);
```

```
unsigned int numParams = params.length();

// Loop through the parameter list
for (unsigned int i=0; i<numParams; i++)
{
       // Create a MuniformParameter for each parameter. Give it a name the same
       // as the shader's parmeter name. UniformDataType and UniformSemantic are
       // are values remapped from MShaderInstance values values that can be
       // set on an MUniformParameter.
       //
       MUniformParameter::DataType  UniformDataType =
            ConvertParameterDataType(
                        fGLSLShaderInstance->parameterType(params[i]));
       MUniformParameter::DataSemantic UniformSemantic
            glslShaderSemantic::ConvertShaderSemantics(
                        fGLSLShaderInstance->semantic(params[i]).asChar());

       MHWRender:: ParameterType paramType = fGLSLShaderInstance->parameterType(params[i]
       UniformDataType = ConvertParameterDataType(paramType);
       MUniformParameter uniParam(params[i].asChar(), UniformDataType, UniformSemantic);
       {
```

MShaderInstance::parameterList() is used to get the names of the parameters, **MShaderInstance::parameterType**() is used to get a type based on a parameter name, and **MShaderInstance::semantic**() is used to get a semantic based on a parameter name.

6.6.6.9 Uniform Data Handling: Drawing

In general, most of the uniform renderer data is bound during the draw phase, when an MDrawContext is available for use. As light binding is more complex, it will be examined in the next section.

Some parameters with internally defined semantics can have parameter values automatically bound at draw time if an MShaderInstance is used. If this is not the case, or there are additional custom semantics, then explicit binding by the plug-in is required. For example, code such as the following can be used to extract view parameters, where *context* is a passed in MDrawContext parameter.

```
MMatrix wMatrix, vMatrix, pMatrix, sMatrix;
MMatrix wvMatrix, wvpMatrix, wvpsMatrix;
MMatrix vpMatrix, vpsMatrix;

wvpMatrix = context.getMatrix(MHWRender::MFrameContext::kWorldViewProjMtx);
wvMatrix = context.getMatrix(MHWRender::MFrameContext::kWorldViewMtx);
wMatrix = context.getMatrix(MHWRender::MFrameContext::kWorldMtx);
vMatrix = context.getMatrix(MHWRender::MFrameContext::kViewMtx);
pMatrix = context.getMatrix(MHWRender::MFrameContext::kProjectionMtx);
vpMatrix = context.getMatrix(MHWRender::MFrameContext::kViewProjMtx)
```

If **MUniformParameter** instances are used to keep track of uniform parameters, then there are a series of VP2 utility interfaces that can be used to extract data, given an MDrawContext

- bool **MUniformParameter::hasChanged**(const MHWRender::MDrawContext& context) const;
- const float* **MUniformParameter:getAsFloatArray**(const MHWRender::MDrawContext& context) const;
- float **MUniformParameter::getAsFloat**(const MHWRender::MDrawContext& context) const;
- MString **MUniformParameter::getAsString**(const MHWRender::MDrawContext& context) const;
- bool **MUniformParameter::getAsBool**(const MHWRender::MDrawContext& context) const;
- int **MUniformParameter::getAsInt**(const MHWRender::MDrawContext& context) const;

The **MUniformParameter::semantic**() method can also be explicitly checked, and direct calls to MDrawContext can be made as shown above.

For a list of internally supported semantics, see *Shader semantics supported by Viewport 2.0* in the *Maya Developer Help*.

6.6.6.10 Draw: Explicit Lighting Handling

Light binding can be explicitly performed regardless of whether an MShaderInstance is used. In this case, a light parameter interface is available on MDrawContext for querying the available light values, including values such as shadow map textures.

In summary, lights can be bound at draw time via the following interfaces:

- unsigned int **MDrawContext::numberOfActiveLights**(LightFilter lightFilter=kFilteredToLightLimit, MStatus* ReturnStatus=NULL) const;
- MLightParameterInformation* **MDrawContext::getLightParameterInformation**(unsigned int lightNumber, LightFilter lightFilter=kFilteredToLightLimit) const;
- MStatus **MDrawContext::getLightInformation**(unsigned int lightNumber, MFloatPointArray& positions, MFloatVector& direction, float& intensity, MColor& color, bool& hasDirection, bool& hasPosition, LightFilter lightFilter=kFilteredToLightLimit) const;

The last interface **getLightInformation**() allows you to obtain a simple set of parameters that roughly matches what would be required for fixed-function lighting. If a plug-in is using raw OpenGL lighting calls, it should be noted that all lighting values are specified in world space.

Note that either the active set or the full list of lights can be returned. It is possible to control which lights affect which shader by selecting the set of lights from the full list.

Sample code and further details can be found in the *Light Interfaces* section of the *Maya Developer Help*.

There is no equivalent interface in VP1, as no light information is ever provided to the plug-in. Thus it is up to the plug-in code to parse the Maya scene and extract light information as desired.

6.6.6.11 Update: Explicit Transparency and State Handling

There are a few methods that allow for transparency control. The level of control is similar to what is available with VP1 hardware shaders. Setting a shader as transparent also marks the render item that uses that shader as transparent if no other override shader is active. In general, the amount of batching is reduced and the affected items will be consolidated.

After the device level update, the shader can update and mark itself as transparent by returning true from **MPxShaderOverrdie::isTransparent**(). The default logic for transparent object drawing is very similar to that of VP1, which is per object sorting relative to the camera. The shader is called to draw twice using a simplistic cull-back / cull-front approach. The method **MPxShaderOverride::overridesDrawState**() can be called to avoid double calls.

In this snippet, the dx11Shader override determines this based on a technique level annotation (as previously mentioned). The logic is left up to the plug-in.

```
bool dx11ShaderOverride::overridesDrawState()
{
	return (fShaderNode && fShaderNode->techniqueOverridesDrawState());
}
```

As VP2 supports more transparency algorithms, a method that returns whether the shader override can support advanced transparency algorithms is the **MPxShaderoverride::supportsAdvancedTransparency**() method. The dx11Shader override again uses a per technique annotation to determine the level of support.

```
bool dx11ShaderOverride::supportsAdvancedTransparency() const
{
	return (fShaderNode && fShaderNode->techniqueSupportsAdvancedTransparency());
}
```

Sample shader code that uses the dx11Shader plug-in and that handles advanced transparency algorithms can be found in the *AutodeskUberShader.fx* shader file (provided in the presets/HLSL11/examples folder of the Maya installation directory). The following code snippet shows the transparency as well as other technique level annotations for a shader that does not support hardware tessellation:

```
technique11 TessellationOFF
<
	bool overridesDrawState = false;   // we do not supply our own render state
settings
	int isTransparent = 3;
	// objects with clipped pixels need to be flagged as isTransparent to avoid the
occluding underlying geometry since Maya renders the object with flat shading when
computing depth
	string transparencyTest = "Opacity < 1.0 || (UseDiffuseTexture &&
UseDiffuseTextureAlpha) || UseOpacityMaskTexture || OpacityFresnelMax > 0 ||
```

```
OpacityFresnelMin > 0";
	// 'VariableNameAsAttributeName = false' can be used to tell Maya's DX11ShaderNode
to use the UIName annotation string for the Maya attribute name instead of the shader
variable name.
	// When changing this option, the attribute names generated for the shader inside
Maya will change and this can have the side effect that older scenes have their shader
attributes reset to default.
	// bool VariableNameAsAttributeName = false;
#ifdef _MAYA_
	// Tells Maya that the effect supports advanced transparency algorithm,
	// otherwise Maya would render the associated objects simply by alpha
	// blending on top of other objects supporting advanced transparency
	// when the viewport transparency algorithm is set to depth-peeling or
	// weighted-average.
	bool supportsAdvancedTransparency = true;
#endif
>
```

Depending on the internal transparency algorithm that is chosen, a different pass is invoked. Continuing the example (technique) above, passes are listed below. Pass *p0* is used for default color pass rendering. Passes *pTransparentPeel*, *pTransparentPeelAndAvg*, and *pTransparentWeightedAvg* are variants used for “advanced” algorithms such as depth peeling.

Note that there is a special annotation *drawContext* used for each pass. This is the pass semantic identifier which is supplied by the renderer as described in the next section (Pass Semantic Handling).

```
pass p0
<
string drawContext = "colorPass";  // tell maya during what draw context this shader
should be active, in this case 'Color'
>
{
// even though overrideDrawState is false, we still set the pre-multiplied alpha state
here in
// case Maya is using 'Depth Peeling' transparency algorithm
// This unfortunately won't solve sorting issues, but at least our object can draw
transparent.
// If we don't set this, the object will always be opaque.
#ifdef _MAYA_
	SetBlendState(PMAlphaBlending, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xFFFFFFFF);
#endif
	SetVertexShader(CompileShader(vs_5_0, v()));
	SetHullShader(NULL);
	SetDomainShader(NULL);
	SetGeometryShader(NULL);
	SetPixelShader(CompileShader(ps_5_0, f()));
}

pass pTransparentPeel
<
	// Depth-peeling pass for depth-peeling transparency algorithm.
	string drawContext = "transparentPeel";
>
```

```
{
	SetVertexShader(CompileShader(vs_5_0, v()));
	SetHullShader(NULL);
	SetDomainShader(NULL);
	SetGeometryShader(NULL);
	SetPixelShader(CompileShader(ps_5_0, fTransparentPeel()));
}

pass pTransparentPeelAndAvg
<
	// Weighted-average pass for depth-peeling transparency algorithm.
	string drawContext = "transparentPeelAndAvg";
>
{
	SetVertexShader(CompileShader(vs_5_0, v()));
	SetHullShader(NULL);
	SetDomainShader(NULL);
	SetGeometryShader(NULL);
	SetPixelShader(CompileShader(ps_5_0, fTransparentPeelAndAvg()));
}

pass pTransparentWeightedAvg
<
	// Weighted-average algorithm. No peeling.
	string drawContext = "transparentWeightedAvg";
>
{
	SetVertexShader(CompileShader(vs_5_0, v()));
	SetHullShader(NULL);
	SetDomainShader(NULL);
	SetGeometryShader(NULL);
	SetPixelShader(CompileShader(ps_5_0, fTransparentWeightedAvg()));
}
```

As the dx11Shader also handles hardware tessellation, a different technique is used with a similar set of pass specifications. In this case hull and domain shaders are required:

```
pass p0
<
	string drawContext = "colorPass";
>
{
	SetBlendState(PMAlphaBlending, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xFFFFFFFF);
	SetVertexShader(CompileShader(vs_5_0, vt()));
	SetHullShader(CompileShader(hs_5_0, HS()));
	SetDomainShader(CompileShader(ds_5_0, DS()));
	SetGeometryShader(NULL); // without geo
	SetPixelShader(CompileShader(ps_5_0, f()));
}

pass pTransparentPeel
<
	// Depth-peeling pass for depth-peeling transparency algorithm.
	string drawContext = "transparentPeel";
>
{
	SetVertexShader(CompileShader(vs_5_0, vt()));
```

```
        SetHullShader(CompileShader(hs_5_0, HS()));
        SetDomainShader(CompileShader(ds_5_0, DS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, fTransparentPeel()));
}
pass pTransparentPeelAndAvg
<
        // Weighted-average pass for depth-peeling transparency algorithm.
        string drawContext = "transparentPeelAndAvg";
>
{
        SetVertexShader(CompileShader(vs_5_0, vt()));
        SetHullShader(CompileShader(hs_5_0, HS()));
        SetDomainShader(CompileShader(ds_5_0, DS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, fTransparentPeelAndAvg()));
}

pass pTransparentWeightedAvg
<
        // Weighted-average algorithm. No peeling.
        string drawContext = "transparentWeightedAvg";
>
{
        SetVertexShader(CompileShader(vs_5_0, vt()));
        SetHullShader(CompileShader(hs_5_0, HS()));
        SetDomainShader(CompileShader(ds_5_0, DS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, fTransparentWeightedAvg()));
}
```

6.6.6.12 Pass Semantic Handling

To disambiguate, within this section, when we mention the term "**pass**" or "**pipeline pass**" this refers to the passes that are performed by the VP2 pipeline and **not** necessarily the passes associated with a given effects file, which will be called a "**technique-pass**".

With the API class MDrawContext we expose where in the pipeline a shader is invoked from using a series of pass semantics and pass identifiers.

The method

MPxShaderOverride::handlesDraw()

can be overridden by the shader override to indicate the passes for which the override will be used. By default, the override is called for color passes (such as "beauty passes"). It is not necessary to override any other passes. If the override indicates that it will not handle a pass, default internal shaders will be used instead during execution of the given pass.

As shown in the previous section, if the override returns that it will handle a given pass, it can switch to use the desired shader before drawing geometry.

For the *dx11Shader* plug-in, the following code indicates handling of color (**MPassContext::kColorPassSemantic**), as well as shadow passes (**MPassContext::kShadowPassSemantic**), transparency passes, as well as passes used for post-effects such as SSAO.

```
bool dx11ShaderOverride::handlesDraw(MHWRender::MDrawContext& context)
{
	const MHWRender::MPassContext & passCtx = context.getPassContext();
	const MStringArray & passSem = passCtx.passSemantics();

	// For color passes, only handle if there isn't already
	// a global override. This is the same as the default
	// logic for this method in MPxShaderOverride
	//
	bool handlePass = false;
	for (unsigned int i=0; i<passSem.length() && !handlePass; i++)
	{
		if (passSem[i] == MHWRender::MPassContext::kColorPassSemantic)
		{
			bool hasOverrideShader = passCtx.hasShaderOverride();
			if (!hasOverrideShader)
				handlePass = true;
		}

		// Handle special pass drawing.
		//
		else if (passSem[i] == MHWRender::MPassContext::kShadowPassSemantic ||
			passSem[i] == MHWRender::MPassContext::kDepthPassSemantic ||
			passSem[i] == MHWRender::MPassContext::kNormalDepthPassSemantic ||
			passSem[i] == MHWRender::MPassContext::kTransparentPeelSemantic ||
			passSem[i] ==
				MHWRender::MPassContext::kTransparentPeelAndAvgSemantic ||
			passSem[i] ==
				MHWRender::MPassContext::kTransparentWeightedAvgSemantic)
		{
			handlePass = fShaderNode->techniqueHandlesContext(passSem[i]);
		}
	}
	return handlePass;
}
```

A test exists for the transparency pass semantic. As part of the implementation we saw the corresponding technique-pass definitions to invoke.

A key area of the code in the dx11Shader plug-in is the activation of a technique-pass. Here, a check is made based on MDrawContext information to determine how to draw. In particular, the pass context MPassContext's list of active semantics (*passSem*) is checked. passId is produced by looping through all the technique-passes for a given technique.

```
dx11ShaderDX11Pass* dx11ShaderNode::activatePass( dx11ShaderDX11Device *dxDevice,
		dx11ShaderDX11DeviceContext *dxContext, dx11ShaderDX11EffectTechnique*
		dxTechnique, unsigned int passId, const MStringArray& passSem, ERenderType
		renderType ) const
{
	// Look for a pass with a given id
```

```
        dx11ShaderDX11Pass* dxPass = dxTechnique->GetPassByIndex(passId);
        if(dxPass == NULL || dxPass->IsValid() == false)
        {
                MStringArray args;
                args.append( MStringFromInt(passId) );
                args.append( fTechniqueName );

                fErrorLog += dx11ShaderStrings::getString(
                    dx11ShaderStrings::kErrorSetPass, args );
                displayErrorAndWarnings();
                return NULL;
        }

        bool canActivate = true;

        // Check if we have a drawContext annotation for the pass (see .fx file)
        MString drawContext;
        getAnnotation(dxPass, "drawContext", drawContext);
        if (drawContext.length())
        {
                // If the shader defines pass contexts, then we must make sure we are in
                // the right one before activating:
                canActivate = false;
                for (unsigned int i=0; i<passSem.length() && !canActivate; i++)
                {
                        if (::_stricmp(passSem[i].asChar(), drawContext.asChar()) == 0)
                        {
                                canActivate = true;
                        }
                }
        }
```

The pass information is extracted from within the **MPxShaderOverride::render**() method using MDrawContext::getPassContext(), and the **MPassContext::passSematics**() to get the list of "active" pass semantics.

```
bool dx11ShaderNode::render(const MHWRender::MDrawContext& context, const
           MHWRender::MRenderItemList& renderItemList)
{
  // Pull out pass information from the draw context:
  const MHWRender::MPassContext & passCtx = context.getPassContext();

  const MStringArray & passSem = passCtx.passSemantics();
```

Note that this code is only an example of how the dx11Shader handles pipeline passes (as technique passes that are annotated using the pipeline pass name). It is not required that a given plug-in work this way.

Further details about specific algorithms for certain passes can be found within the sample code for the dx11Shader and glslShader plug-ins as well as the API Guide (See Advanced Topics->Effect Instances, and Plug-in Entry Points->Effects Overrides sections).

6.6.6.13 Initialization: Non-textured Mode Shader

Non-textured display mode drawing adheres to VP1 logic, where hardware shaders do not affect the rendered outcome. Instead, for better performance, a fixed internal shader instance is used for all render items using shader overrides.

It is, however, possible to specify an override shader instance by overriding the

MPxShaderOverride::nonTexturedShaderInstance()

method to return a non NULL value during the *initialization phase*. This can be useful if some differentiation in rendering is required when in non-textured mode.

The more the shader instance is reused, the better the possibility for geometry batching. However, it can never achieve the same level of reuse as the internally provided shader.

See the *vp2BlinnShader* example for sample code. fNonTexturdColorShaderInstance is a data member kept per override.

```
virtual MHWRender::MShaderInstance* nonTexturedShaderInstance(bool &monitor) const
{
	if (fNonTexturedColorShaderInstance)
	{
		monitor = true;
		return fNonTexturedColorShaderInstance;
	}
	return NULL;
}
```

This frequency of storage is used as the code keeps track of a non-textured shader color attribute which is updated and used at the same time as attributes for the textured shader.

6.7 Porting Renderers

The **MRenderOverride** interface is the avenue for writing logic that will override the entire render loop for drawing. It replaces the legacy VP1 interfaces **MViewportRenderer** and **MPx3dModelView**.

The key elements which provide greater power and flexibility available for an MRenderOverride are: custom and internal operations, and the ability to arbitrarily connect operations into a dependent series of scene and quad operations via shaders.

Some of the key differences among the three interfaces are listed as follows:

	MPx3dModelView	MViewportRenderer	MRenderOverride
Renderer support	VP1, 1 pass VP2	VP1	VP2
“Unit” of rendering	A “pass” = 1 full internal scene render	A user render function	An operation
Render logic control	Number of passes	Single call	Iterator provides list of operations to perform
Formal access to internally defined “units”	None. Scene renders are implicitly tied to a pass number, and	None.	Arbitrary number of scene operations, HUD, and

	internally invoked.		presentation operation can be invoked.
Extensible operations	None	None	Scene, HUD, User, 2d quad blit.
Non 3d viewport support	None	Render view, command line	Render view, command line
Rendering Toggle Exposure	None	Listed as a viewport renderer	Listed as a renderer for viewport and batch rendering
Output target control	None. Linux only MPxGLBuffer deprecated.	None	Explicit targets possible
DAG type filters	Yes	Yes	Yes
Camera override	Yes	No	Yes
Shader override	No	No	Yes
Lighting Override	Yes	No	Yes
Resource management	No	No	Yes
Dependent operation specification.	No	No	Connection via shared targets and shader parameters
Camera overrides	Interactive	Interactive and batch	Interactive and batch
UI drawables	No	No	Yes
Draw API	OpenGL	OpenGL	OpenGL, DirectX
Pipeline Callbacks	Via MUIMessage interfaces (viewport only)	Via MUIMessage interfaces (viewport only)	View MRenderer pre/post-frame, pre/post-scene

Note that callbacks that are invoked from within rendering are not necessarily part of the logic but are included as they are important in scenarios such as image sequence capture. [See "Capturing" Render Targets for more details].

6.7.1 Handling Multi-Pass MPx3dModelView Logic

The preferred approach to porting an MPx3dModelView is to map either single or multi-pass logic to a series of scene operations, followed by HUD and presentation operations. Although not recommended, MPx3dModelView can still be called while using VP2, with the caveat that only one iteration (pass) should ever be executed, and no legacy drawing occurs within that interface. This way, any non-rendering logic previously written using this class can be preserved, while the actual rendering logic is replaced. The relevant MPx3dModelView interfaces are as follows:

```
virtual void         preMultipleDraw();
virtual void         postMultipleDraw();
virtual void         preMultipleDrawPass(unsigned int index);
virtual void         postMultipleDrawPass(unsigned int index);
virtual bool         okForMultipleDraw(const MDagPath &);
virtual unsigned int multipleDrawPassCount();
```

```
bool                multipleDrawEnabled() const;
void                setMultipleDrawEnable(bool enable);
```

If the number of passes is one, then this naturally becomes a single scene operation, followed by a HUD and presentation operation. This is presented as a "simple" sample scenario in the Simple Render Override section. A multi-pass example would be the stereo plug-in.

6.7.2 Handling MViewportRenderer Logic

The porting of an MViewportRenderer plug-in to an MRenderOverride should be straight-forward, as MRenderOverride supports a superset of MViewportRenderer's functionality and both are "renderer" based interfaces.

The recommended approach is to encapsulate the **MViewportRenderer::render**() method logic as a single user operation (MUserRenderOperation) from within a custom MRenderOverride.

There is no explicit concept of a UI (non-beauty) draw pass in MViewportRenderer. The *MViewportRenderer::kOverrideThenUI* option can be supported by adding a scene operation that only draws UI after the user operation.

The *MViewportRenderer::kOverrideThenStandard* option can be supported by a user operation followed by a scene operation. The MRenderOverride logic is flexible, and the ordering of user and scene operations can be tailored to best support the depth compositing workflow.

6.7.3 Override Operations

A set of operations are available for use as atomic units to build rendering logic.

- **MUserRenderOperation**: User defined rendering operation.
- **MHUDRender** : HUD render operation
- **MPresentTarget** : Presentation of rendered result to an on-screen target
- **MClearOperation** : Background clear
- **MSceneRender** : 3d scene render
- **MQuadRender** : 2d quad blit with a shader

A "clear background" operation is not something that can be run standalone, and as such is attached as an option to scene and quad operations. Customization is possible by deriving from the existing classes. HUD and presentation operations generally don't require any overrides while quad and user operations won't perform anything useful unless they are overridden.

Scene operations have a variety of formal overrides which can be set as part of customization. For non-VP2 interfaces, overrides change the *actual state* of a viewport and care must be taken to restore state afterwards. Overrides specified on scene operations only persist while the operation is being executed**.** Take, for example, this possible operation sequence:

- The global viewport filter state allows drawing of all objects
- Scene operation 1 filters out NURBS curves
- Scene operation 2 filters out NURBS shapes

When operation 2 is executed, it does not filter out both NURBS curves and shapes, but instead only filters NURBS shapes.

6.7.4 Simple Render Override

The simplest possible override is one that mimics what is done internally for a single scene render. The operations required would be: a scene, a HUD and a presentation operation, in that order. The *viewOverrideSimple* sample plug-in will be used as reference.

6.7.4.1 Registration

Registration is handled by calling MRenderer::**registerOverride**() during plug-in initialization, and MRenderer::**deregisterOverride**() during plug-in uninitialization. The basic code shown below creates an instance of a class derived from MRenderOverride and uses that instance to register the plug-in. The name of the plug-in is used to find the override for deregistration later on.

```
// Registration in plug-in initialization
MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
viewOverrideSimple *overridePtr = new viewOverrideSimple("viewOverrideSimple");
renderer->registerOverride(overridePtr);

…

// Deregistration in plug-in uninitialization
const MHWRender::MRenderOverride* overridePtr = renderer-
>findRenderOverride("viewOverrideSimple");
if (overridePtr)
{
	renderer->deregisterOverride( overridePtr );
	delete overridePtr;
}
```

6.7.4.2 Creation of Operations

MRenderOverride::setup(), and **MRenderOverride::cleanup()** are logical places in which to create and cleanup operation instances that are used to build rendering logic. As setup() and cleanup() are called for every frame update, operation parameter updates can be performed at this time.

For this *viewOverrideSimple* example, three operations are created as needed during setup(). The example derives from MSceneRender to create the scene operation that sets the background “clear” color.

```
//
// Inside class definition
//
class viewOverrideSimple : public MHWRender::MRenderOverride
{
public:
	…
```

```
        // Basic setup and cleanup
        virtual MStatus setup( const MString & destination );
        virtual MStatus cleanup();

protected:
       …
       // Operations and operation names
       MHWRender::MRenderOperation* mOperations[3];
       MString mOperationNames[3];
};

// Operations and operation names defined as part of the viewOverrideSimple
class simpleViewRenderSceneRender : public MHWRender::MSceneRender
{
public:
    simpleViewRenderSceneRender(const MString &name);
    virtual MHWRender::MClearOperation & clearOperation();
};

----
//
// Inside class definition
// Operations in setup() which creates the operations once
//
MStatus viewOverrideSimple::setup( const MString & destination )
{
        MHWRender::MRenderer *theRenderer = MHWRender::MRenderer::theRenderer();
        // Create a new set of operations as required
        if (!mOperations[0])
        {
                mOperations[0] = (MHWRender::MRenderOperation *) new
                   simpleViewRenderSceneRender( mOperationNames[0] );
                mOperations[1] = (MHWRender::MRenderOperation *) new
                   MHWRender::MHUDRender();
                mOperations[2] = (MHWRender::MRenderOperation *) new
                   MHWRender::MPresentTarget( mOperationNames[2] );
        }
        if (!mOperations[0] ||
                !mOperations[1] ||
                !mOperations[2])
        {
                return MStatus::kFailure;
        }
        return MStatus::kSuccess;
}
```

6.7.4.3 "Building" Render Loop Logic

Building the sample plug-in is straightforward. It will simply return an ordered list of operations; in this case: scene, HUD and presentation.

As the iteration interfaces are called after setup(), per-frame modification of the render loop logic can be executed. This external ordering reflects what is done for internal rendering for render logic updates.

Overrides should be implemented for the following key “iterator” methods on an MRenderOverride:

- virtual bool **startOperationIterator**();
- virtual MHWRender::MRenderOperation * **renderOperation**();
- virtual bool **nextRenderOperation**();

startOperationIterator()allows the override to start iterating, renderOperation() returns the current operation, and nextRenderOperation() iterates to the next operation.

The code from viewOverrideSimple is as follows:

```
// Basic iterator methods which returns a list of operations in order.
// The operations are not executed at this time only queued for execution
//
// - startOperationIterator() : to start iterating
// - renderOperation() : will be called to return the current operation
// - nextRenderOperation() : when this returns false we've returned all operations
//
bool viewOverrideSimple::startOperationIterator()
{
	mCurrentOperation = 0;
	return true;
}

MHWRender::MRenderOperation*
viewOverrideSimple::renderOperation()
{
	if (mCurrentOperation >= 0 && mCurrentOperation < 3)
		if (mOperations[mCurrentOperation])
			// Return an operation
			return mOperations[mCurrentOperation];
	return NULL;
}

bool
viewOverrideSimple::nextRenderOperation()
{
	mCurrentOperation++;
	if (mCurrentOperation < 3)
		return true;
	// Return false to indicate no more operations
	return false;
}
```

The operations are not executed immediately, but are instead queued for execution once the list of operations has been interpreted. Once interpreted, operations are “executed” in the same order as they were specified.

Any hardware resources returned via overridden methods only need to persist during the execution of the operation. Any resources that are shared between operations should be stored

at the override level. This often means resources such as render targets, shaders and state objects.

There is very little additional cost involved with the code that returns a different set of or different ordering of operations per frame. If there is a higher cost on the plug-in side, then it is up to the plug-in to cache the ordering as appropriate.

It is not advisable, however, to rebuild resources every frame unless required. For example, if additional output render targets / textures, shaders, or state objects are required, an attempt to avoid rebuilding every frame is advised. As a simple example, if offscreen render targets are being used to render the scene, generally they only need to updated when the output size is changed, and not between frames.

Maya resources do not need to be cached across the entire set of operations, and can be evaluated at operation execution time on demand. For example, a Maya object list filter can be computed when required.

As there are an uncountable number of render loop variations, the next sections will cover some of the “common” use cases. Additional examples not discussed in this document can be found in the Maya Developer Kit.

6.7.5 Sample Override Options

One very simple example implementation of an override that performs drawing is found in the derived class *simpleViewRenderSceneRender*(). Here, the “clear” operation has an override implemented to track the colors used for the internal renderer. It is possible to use M3dView accessor methods for colors, but they may give incorrect results for command line / render view rendering.

```
// Background color override. We get the current colors from the
// renderer and use them
// MHWRender::MClearOperation &
simpleViewRenderSceneRender::clearOperation()
{
       MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
       bool gradient = renderer->useGradient();
       MColor color1 = renderer->clearColor();
       MColor color2 = renderer->clearColor2();

       float c1[4] = { color1[0], color1[1], color1[2], 1.0f };
       float c2[4] = { color2[0], color2[1], color2[2], 1.0f };

       mClearOperation.setClearColor( c1 );
       mClearOperation.setClearColor2( c2 );
       mClearOperation.setClearGradient( gradient);
    return mClearOperation;
}
```

For more examples, see the *viewMRenderOverride* developer kit example. More of the base classes have been overridden to allow for the writing of more custom override methods.

6.7.5.1 Selection Filter

It is possible that only a subset of objects is to be rendered. In this case, a scene operation can override the objectSetOverride() to return a selection list (MSelectionList). The code in the *viewMrenderOverride* example demonstrates how a set name can be used to populate a selection list for an MSceneRender override called MSceneRenderTester.

```
virtual const MSelectionList* MSceneRenderTester::objectSetOverride()
{
	MSelectionList list;
	list.add( mSetName );

	MObject obj;
	list.getDependNode( 0, obj );

	MFnSet set( obj );
	set.getMembers( mFilterSet, true );

	if (mFilterSet.length())
	{
		return &mFilterSet;
	}
	return NULL;
}
```

.

6.7.5.2 UI Drawables

Each operation, except for the presentation operation, can have additional transient UI drawn per frame render.

To do this, use the interface MUIDrawManager, similar to how per object UI drawing is added. For scene operations, both pre and post UI drawing is supported, and different interfaces are available. Otherwise, the **hasUIDrawables**() method on the appropriate operation will be called to check for additional drawing and **addUIDrawables**() will be called back.

See the following example for code that draws simple UI, specified for a custom user operation called MUserRenderOperationTester. All positions specified are in world or screen coordinates as appropriate.

```
virtual void MUserRenderOperationTester::addUIDrawables( MHWRender::MUIDrawManager&
drawManager, const MHWRender::MFrameContext& frameContext )
{
	drawManager.beginDrawable();

	drawManager.setColor( MColor( 0.95f, 0.5f, 0.1f ) );
	drawManager.setFontSize( MHWRender::MUIDrawManager::kSmallFontSize );
	drawManager.text( MPoint( 0, 2, 0 ), MString("UI draw test in user operation") );
	drawManager.line( MPoint( 0, 0, 0), MPoint( 0, 2, 0 ) );
	drawManager.setColor( MColor( 1.0f, 1.0f, 1.0f ) );
	drawManager.sphere(  MPoint( 0, 2, 0 ), 0.8, false );
	drawManager.setColor( MColor( 0.95f, 0.5f, 0.1f, 0.4f ) );
	drawManager.sphere(  MPoint( 0, 2, 0 ), 0.8, true );

	drawManager.endDrawable();
```

```
}
```

6.7.6 Compositing Externally Rendered Results

Often, it is useful to have a custom external renderer that fills in the pixels for the “beauty” (color) pass, and just have Maya render the remaining items, which includes the UI. There are two approaches to accomplish this. One is to write raw draw code. The recommended way is to use instead a 2d quad blit which does not exist as a construct outside of VP2.

See the viewImageBlitOverride. As the name suggests, it blits images for the color pass.

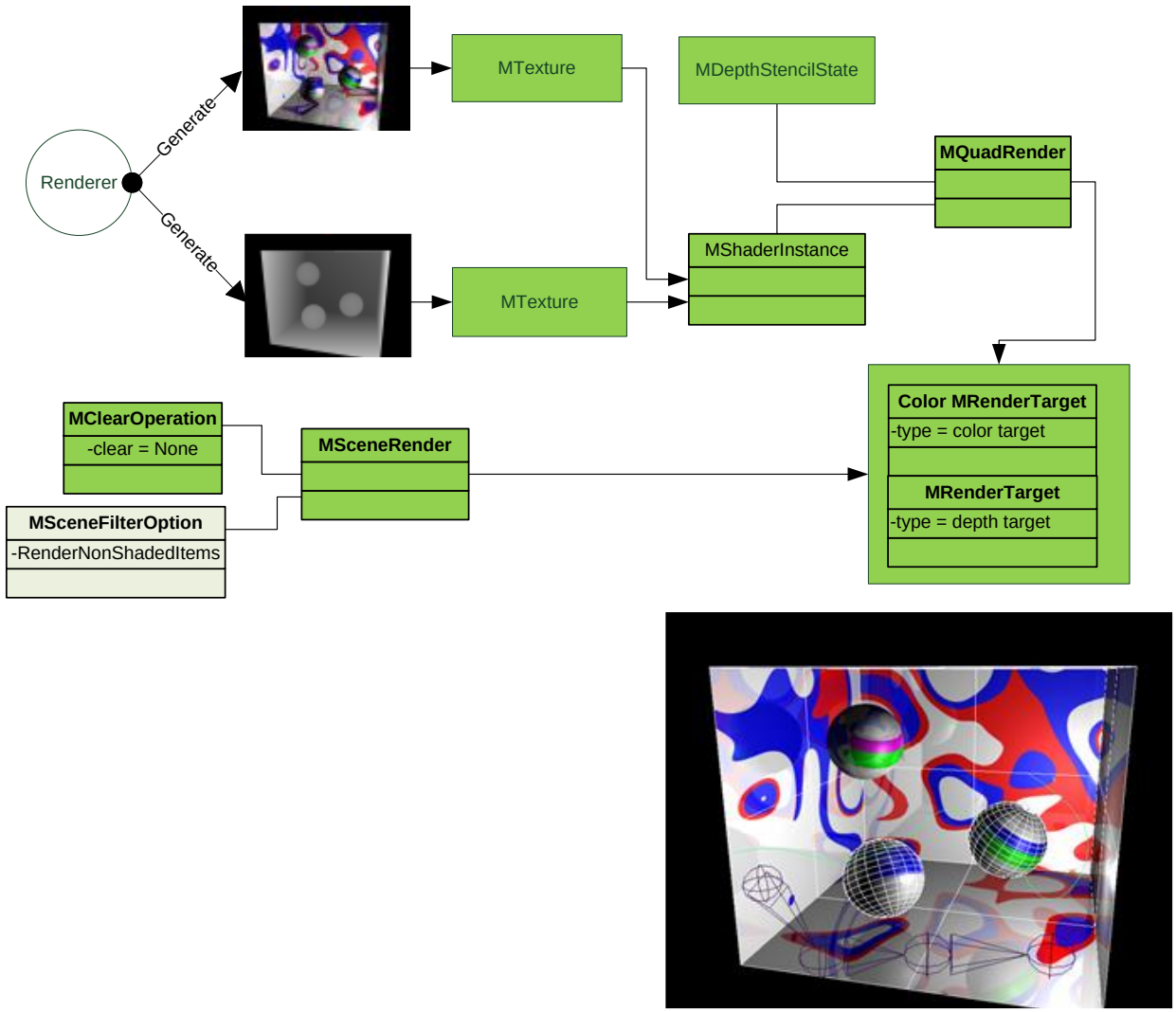


Figure 15: This figure demonstrates how an external renderer (“Renderer”) would generate a color and depth image. The override would load the images as textures (MTexture), use these as input to a 2d Quad blit (MQuadRender). This is followed by a scene render operation which composites additional “UI” on top to produce the final image shown at bottom.

6.7.6.1 Custom Quad Rendering

The custom operation that derives from **MQuadRender** is called SceneBlit. For proper depth compositing, it allows both a color and a depth texture (MTexture) as input. In the sample plug-in, these textures are read from disk, but for a "live" render they can also be updated dynamically.

```
class SceneBlit : public MHWRender::MQuadRender
{
public:
	virtual const MHWRender::MShaderInstance * shader();
	virtual const MHWRender::MDepthStencilState *depthStencilStateOverride();
protected:

	// Shader used for the quad render. Owned by operation.
	MHWRender::MShaderInstance *mShaderInstance;
	// Texture(s) used for the quad render. Not owned by operation.
	MHWRender::MTextureAssignment mColorTexture;
	MHWRender::MTextureAssignment mDepthTexture;

	// Depth stencil state used for the blit
	const MHWRender::MDepthStencilState *mDepthStencilState;
```

The class definition shows a shader instance (mShaderInstance) used for drawing, as well as the color and depth textures (mColorTexture, mDepthTexture), and the depth-stencil state used as input parameters to the shader and quad operation respectively.

The **shader**() override will return the shader stored, and the **depthStencilStateOverride**() will return the stored depth state.

The shader() method will create the appropriate shader as needed. In this case, it will use a file based effect shader (*mayaBlitColorDepth10.fx*), which is provided in the \bin\HLSL folder of the Maya installation directory.

```
const MHWRender::MShaderInstance * SceneBlit::shader()
{
	if (!mShaderInstance)
	{
		MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
		const MHWRender::MShaderManager* shaderMgr = renderer->getShaderManager();
		if (shaderMgr)
		{
			// Create the shader if not already created
			//
			// The default shader technique will blit color and depth textures
			// to the output color and depth buffers respectively. The values in
			// the depth texture are expected to be normalized.
			//
			bool showDepthAsColor = false;
			mShaderInstance = shaderMgr->getEffectsFileShader(
                                        "mayaBlitColorDepth", "" );
		}
	}

	…
```

```
}
```

It is important to remember to override the depthStencilStateOverride() method when trying to blit depth, as the default behaviour is to disable depth writes. The override enables the appropriate depth state. A new depth stencil state object is created once for reuse using the method: **MStateManager**::**acquireDepthStencilState**()

```
// We want to have this state override set to override the default
// depth stencil state which disables depth writes.
//
const MHWRender::MDepthStencilState *SceneBlit::depthStencilStateOverride()
{
	if (!mDepthStencilState)
	{
		MHWRender::MDepthStencilStateDesc desc;
		desc.depthEnable = true;
		desc.depthWriteEnable = true;
		desc.depthFunc = MHWRender::MStateManager::kCompareAlways;
		mDepthStencilState =
			MHWRender::MStateManager::acquireDepthStencilState(desc);
	}
	return mDepthStencilState;
}
```

For the shader to be useful, the color and depth textures need to be set up and bound.

The following data members are declared in the override class to keep track of the textures and corresponding texture descriptions:

```
// Texture(s) used for the quad render
MHWRender::MTextureDescription mColorTextureDesc;
MHWRender::MTextureDescription mDepthTextureDesc;
MHWRender::MTextureAssignment mColorTexture;
MHWRender::MTextureAssignment mDepthTexture;
```

The color texture in this example is loaded from disk, but can also be filled in from memory. In the example, the MImage class is used to load an image of size *targetWidth* by *targetHeight* from a Maya iff file. textureManager is the render's texture manager (MTextureManager). The color format for this example is RGBA 8 bits per channel fixed, although any RGBA color format could have been used.

```
// Read the image from disk using MImage and get the raw data
MImage image;
image.readFromFile( colorImageFileName );
image.getSize( targetWidth, targetHeight );
unsigned char* textureData = image.pixels();

// Set up the texture description
mColorTextureDesc.fWidth = targetWidth;
mColorTextureDesc.fHeight = targetHeight;
mColorTextureDesc.fDepth = 1;
mColorTextureDesc.fBytesPerRow = 4*targetWidth;
mColorTextureDesc.fBytesPerSlice = mColorTextureDesc.fBytesPerRow*targetHeight;
```

```
// Acquire a new texture using the description and the raw data
mColorTexture.texture = textureManager->acquireTexture( "", mColorTextureDesc,
                                        textureData );

// Mark texture has changed
mColorTextureChanged = true;
```

The depth texture in this example is also loaded from disk, but can also be filled in from memory. Note that additional normalization can be performed using an MDepthNormalizationDescription instance at texture creation time. In this example, nothing extra is required, since the depth image was generated by Maya itself.

```
// Read the image from disk using MImage and get the raw data
MImage image;
image.create(targetWidth, targetHeight, 4, MImage::kByte);
image.readDepthMap( depthImageFileName );
image.getDepthMapSize( targetWidth, targetHeight );

// Set up the texture description
mDepthTextureDesc.fWidth = targetWidth;
mDepthTextureDesc.fHeight = targetHeight;
mDepthTextureDesc.fDepth = 1;
mDepthTextureDesc.fBytesPerRow = targetWidth;
mDepthTextureDesc.fBytesPerSlice = mDepthTextureDesc.fBytesPerRow*targetHeight;

// Acquire a new texture using the description and the raw data
MHWRender::MDepthNormalizationDescription normalizationDesc;
mDepthTexture.texture = textureManager->acquireDepthTexture("",
       image, false, &normalizationDesc );

// Mark texture has changed
mDepthTextureChanged = true;
```

Textures can now be bound to the shader. . Assuming that the textures can change every frame, the sample code updates the shader parameters when the shader is requested in *SceneBlit::shader*(). The following occurs after the shader is acquired (as shown in the previous code snippet).

```
// If texture changed then bind new textures to the shader
// based on the mColorTextureChanged / mDepthTextureChanged flags
//
status = MStatus::kSuccess;
if (mColorTextureChanged)
{
       status = mShaderInstance->setParameter("gColorTex", mColorTexture);
       mColorTextureChanged = false;
}

if (status == MStatus::kSuccess && mDepthTextureChanged)
{
       status = mShaderInstance->setParameter("gDepthTex", mDepthTexture);
       mDepthTextureChanged = false;
}
```

The shader has a parameter named "*gColorTex*" for the color texture and a parameter called "*gDepthTex*" for the depth texture.

6.7.6.2 Filtered Scene UI Operation

The second part to this example is a second operation that renders the scene UI on top of the existing contents of the render target. Since depth has been specified, appropriate depth testing will occur for the UI elements.

Depending on the types of objects drawn by the external scene render, different filters can be set. In this example, overrides are implemented for the following methods for a custom scene render:

- **MSceneRender::renderFilterOverride**() : To provide render item filtering to indicate that only non-shaded items should be drawn. (*MSceneRender::kRenderNonShadeditems*) . The assumption being that the previous quad blit has placed an image of all shaded items into the render targets.
- **MSceneRender::objectTypeExclusion**() : To exclude drawing the grid and image planes.
- **MSceneRender::clearOperation**() : To clear any previous color or depth target contents by set an empty clear mask (*MClearOperation::kClearNone*).

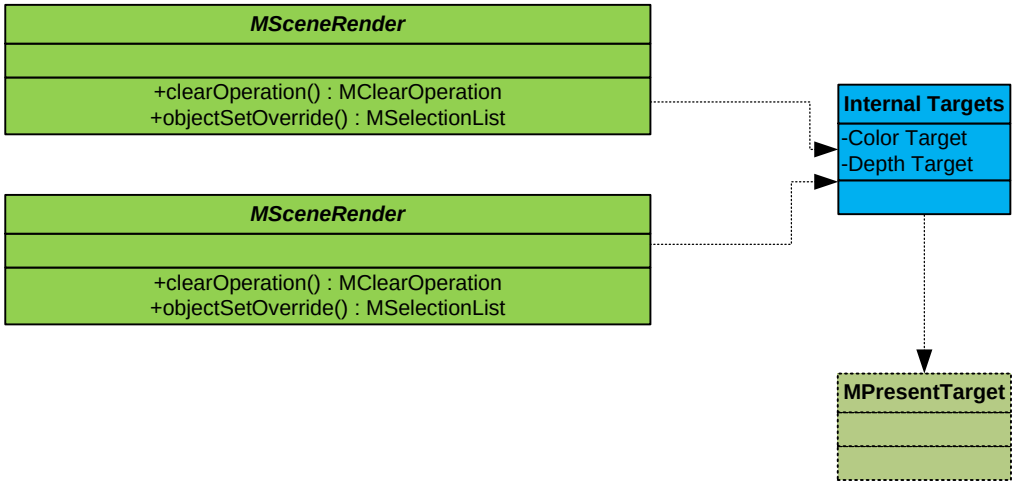
The following is a code snippet for the UIDraw class that is derived from MSceneRender:

```
MHWRender::MSceneRender::MSceneFilterOption
UIDraw::renderFilterOverride()
{
       // Draw only non-shaded items
       return MHWRender::MSceneRender::kRenderNonShadedItems;
}

MHWRender::MSceneRender::MObjectTypeExclusions
UIDraw::objectTypeExclusions()
{
       // Exclude drawing the grid and image planes
       Return(MHWRender::MSceneRender::MObjectTypeExclusions)
          (MHWRender::MSceneRender::kExcludeGrid |
           MHWRender::MSceneRender::kExcludeImagePlane);
}

MHWRender::MClearOperation &
UIDraw::clearOperation()
{
       // Disable clear since we don't want to clobber the scene colour blit.
       mClearOperation.setMask((unsigned int)MHWRender::MClearOperation::kClearNone);
       return mClearOperation;
}
```

6.7.7 Multi-Pass Scene Rendering

An example of multi-pass scene rendering, with logic similar to that provided by MPx3dModelView, can be found in the *viewObjectSetOverride* plug-in.

The example shows how two scene renders can be specified to run one after the other with each scene render using a different object set as a filter.



The key here, as with the previous external render composite, is not to clear the color and depth context for the second scene render. The custom scene operator is as follows. It accepts, at creation time, a set name for the object set filter (that returns the list of objects that should be drawn), and the clear mask that should be used.

```
class ObjectSetSceneRender : public MSceneRender
{
public:
	ObjectSetSceneRender( const MString& name, const MString setName, unsigned int
clearMask )
	: MSceneRender( name )
	, mSetName( setName )
	, mClearMask( clearMask )
	{}

	// Return filtered list of items to draw
	virtual const MSelectionList* objectSetOverride()
	{
		MSelectionList list;
		list.add( mSetName );

		MObject obj;
		list.getDependNode( 0, obj );

		MFnSet set( obj );
		set.getMembers( mFilterSet, true );

		return &mFilterSet;
	}
```

```
        // Return clear operation to perform
        virtual MHWRender::MClearOperation & clearOperation()
        {
                mClearOperation.setMask( mClearMask );
                return mClearOperation;
        }

protected:
        MSelectionList mFilterSet;
        MString mSetName;
        unsigned int mClearMask;

};
```

When operations are created, the appropriate arguments values are passed in. In this example, the two sets to be drawn are, consecutively “set1” and “set”, and the corresponding clear masks are MClearOperation::kClearAll and MClearOperation::kClearNone.

```
// Clear when rendering set 1
mRenderSet1 = new ObjectSetSceneRender( render1Name, set1Name,
            (unsigned int)MHWRender::MClearOperation::kClearAll );

// But don't clear when rendering set 2
mRenderSet2 = new ObjectSetSceneRender( render2Name, set2Name,
             (unsigned int)MHWRender::MClearOperation::kClearNone );
```

6.7.8 “Dependent” Operation Rendering

An operation with a data dependency that determines the ordering of operations is referred to as “**dependent”**. To add these kinds of dependency to the rendering logic, shaders and resources (such as render targets or texture) can be used.

The dependency between two operations A followed by B is defined implicitly by B having a shader parameter that requires data computed by operation A.

If the input data is a render target, and the shader outputs to the same render target (for example, to perform an operation directly on the target), then a “**circular dependency”** is created on the same resource. Although it is best to avoid these situations, the renderer can still handle this case by making a copy of the source targets to break its circular nature.

6.7.9 Stereo Rendering (Scene -> Quad Render Shader Dependency)

An example which is not in the developer’s kit is the plug-in which is used for stereo panel rendering. It will multi-pass using two scene renders. Each render will render into a two separate color render targets and two separate cameras as specified by the scene operators.

These two targets are then used by a third quad render operation to composite the two images together. This forms an operation dependency between the scene renders and quad operation. The complete operation layout is shown below with the dependency show using red arrows.

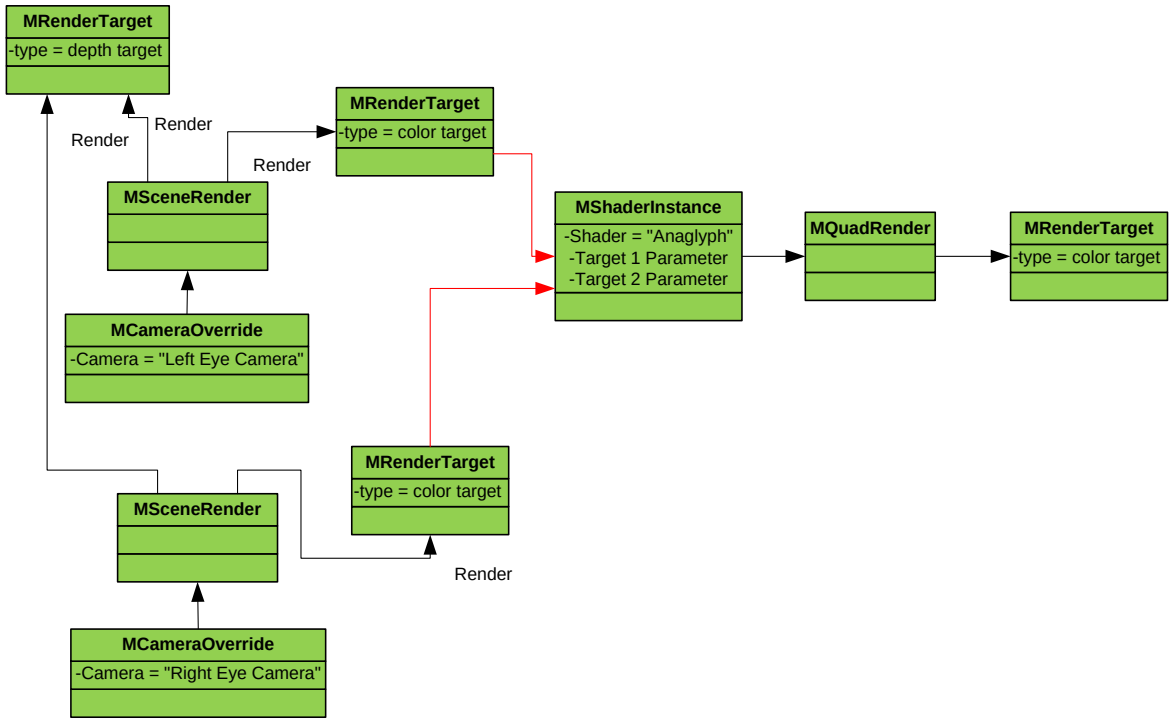


Figure 16: Two scene renders which are used to render a stereo paint(left and right cameras) is shown on the left side of the diagram. The final result is a composite of the pair of color render results. There is no need to composite depth, and the depth target simply acts as a reusable resource.

The camera override is specified by overriding MSceneRender::cameraOverride(). The code finds the appropriate camera in the “rig” and returns an MCameraOverride. The camera path property (mCameraPath) of the MCameraOverride is used specify the camera override.

```
virtual const MHWRender::MCameraOverride * cameraOverride()
{
	// Use a different camera depending on what to render
	if (mDisplayMode==”center”)
		mCameraOverride.mCameraPath = <center camera path>;
	else if (mDisplayMode==”left camera”)
		mCameraOverride.mCameraPath  <left camera path>;
	else
		mCameraOverride.mCameraPath = <right camera path>;
	return &mCameraOverride;
}
```

The actual targets to render to are specified per operation. The **MSceneRender::targetOverrideList**() will return a pointer to a target list. For stereo rendering, we use one color and one depth target per camera.

```
// Return target overrides
virtual MHWRender::MRenderTarget* const* targetOverrideList(unsigned int &listSize)
{
	listSize = 2;
	return &mTargets[0];
```

```
}
…

// Definition of the target overrides as part of the scene render class definition:
protected:
      MHWRender::MRenderTarget* mTargets[2];
```

These targets are maintained at the override level, and are simply cached on the operation to be returned when requested via the targetOverrideList() methods. The targets are specified as follows:

```
mTargetOverrideNames[kColorTarget] = MString("_stereoLeftTarget_");
mTargetOverrideNames[kColorTarget2] = MString("_stereoRightTarget_");
mTargetOverrideNames[kDepthTarget] = MString("_stereoDepthTarget");

MHWRender::MRasterFormat colorFormat = MHWRender::kR8G8B8A8_UNORM;
MHWRender::MRasterFormat depthFormat = MHWRender::kD24S8;
unsigned int sampleCount = 1; // no multi-sampling
mTargetDescriptions[kColorTarget] =
       new MHWRender::MRenderTargetDescription(mTargetOverrideNames[0], 256, 256,
                                               sampleCount, colorFormat, 0, false);
mTargetDescriptions[kColorTarget2] =
       new MHWRender::MRenderTargetDescription(mTargetOverrideNames[1], 256, 256,
                                               sampleCount, colorFormat, 0, false);
mTargetDescriptions[kDepthTarget] =
       new MHWRender::MRenderTargetDescription(mTargetOverrideNames[2], 256, 256,
                                               sampleCount, depthFormat, 0, false);
```

A unique name is used for each target resource. Two unique color targets are required, since they both need to persist during the execution of the operations. Only a single depth target is required, as the depth contents are not required after each scene render.

The targets are set to an initial size, and resized as required based on changes to the output width and height, as specified by the **MRenderer::outputTargetSize()** method. To avoid complicated resize management, **MRenderTarget::updateDescription()** allows for an update of the target characteristics based on a target description (which includes size). As size changes are assumed to only occur per frame, the **MRenderOverride::setup**() method is the appropriate place to update these resources.

```
// Get the current target width and height
MHWRender::MRenderer *theRenderer = MHWRender::MRenderer::theRenderer();
unsigned int targetWidth = 0;
unsigned int targetHeight = 0;
if( theRenderer )
{
       theRenderer->outputTargetSize( targetWidth, targetHeight );
}

// Update the cached target descriptions width and height parameters
for (unsigned int i=0; i<kTargetCount; i++)
{
       mTargetDescriptions[i]->setWidth( targetWidth );
       mTargetDescriptions[i]->setHeight( targetHeight );
```

```
}

// Acquire new targets if never acquired before, OR resize the current targets
// using the updateDescription()method for each target.
const MHWRender::MRenderTargetManager *targetManager =
                  theRenderer->getRenderTargetManager(;
if (targetManager)
{
       for (unsigned int i=0; i<kTargetCount; i++)
       {
              if (!mTargets[i])
              {
                     mTargets[i] = targetManager->acquireRenderTarget( *(
                                                  mTargetDescriptions[i]) );
              }
              else
              {
                     mTargets[i]->updateDescription( *( mTargetDescriptions[i]) );
              }
       }
}
```

As shown in the renderer compositing example, the quad render compositing operation requires that the two color targets be bound to the input parameters of the shader used for the quad operation. In the example, the input target parameters are gSourceTex and gSourceTex2, and the two color targets are mTargets[0] and mTargets[1].

```
// Assign color target 0 as the first input
MHWRender::MRenderTargetAssignment assignment;
assignment.target = mTargets[0];
MStatus status = shaderInstance->setParameter("gSourceTex", assignment);
if (status != MStatus::kSuccess)
{
       return NULL;
}
// Assign color target 0 as the first input
MHWRender::MRenderTargetAssignment assignment2;
assignment2.target = mTargets[1];
status = shaderInstance->setParameter("gSourceTex2", assignment2);
if (status != MStatus::kSuccess)
{
       return NULL;
}
```

6.7.9.1 Color and Depth Targets

The *viewRenderOverrideTargets* plug-in example demonstrates how to render out color and depth to two intermediate targets and then blit that to a third. This is similar to the operation connections used for stereo pair rendering for side by side viewing.

6.7.10 Glow (Quad to Quad Render Dependency)

The example with the most “complex” dependent operation graph is the viewRenderOverride plug-in example. In this case, the algorithm requires a scene operation to a target that is used

by subsequent 2d color operations to achieve a “glow”. The steps are fairly rudimentary, but help to show the scene operation to quad operation dependency as well as quad operation to quad operation dependencies.

An “intensity threshold” shader exists in the first quad operation. This is followed by a 2-pass blur (using 2 quad operations for horizontal and vertical blur), followed by a blend back to the original scene rendered color target.

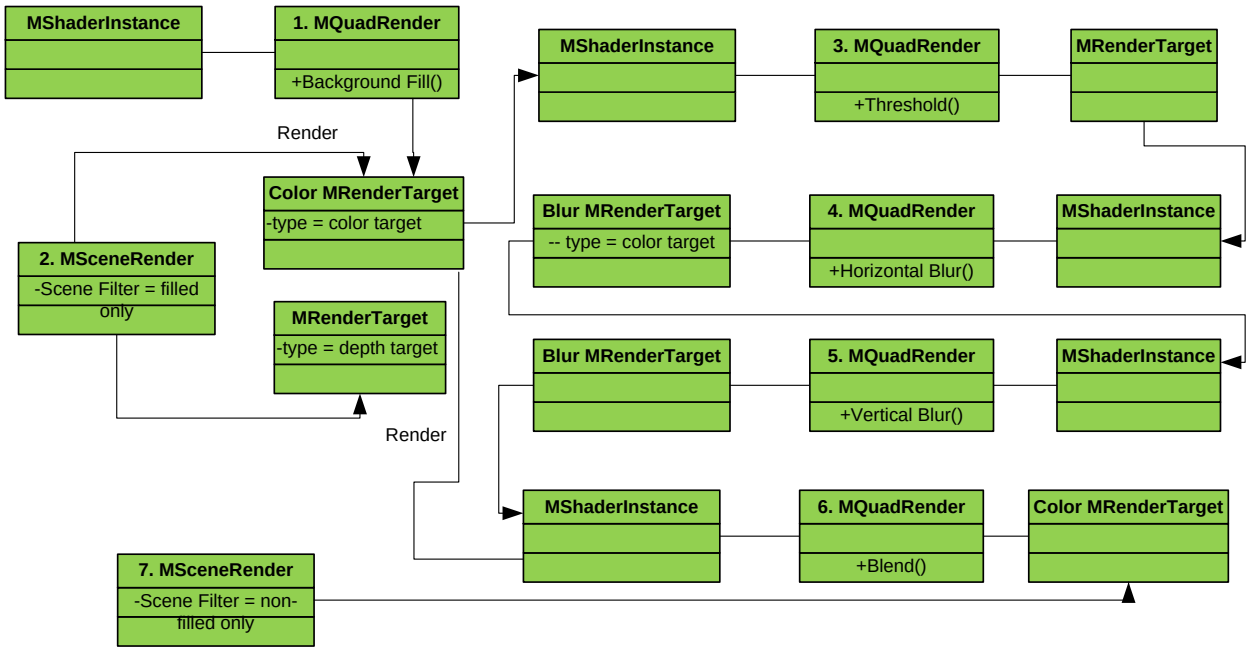


Figure 17: Glow render override example shows dependencies between the various operations used. Note that all dependencies are between targets / textures on “source” operations and shader inputs on “destination” operations.

Within the code example, there is one subclassed quad render called viewRenderQuadRender, which is derived from MQuadRender. Different shaders are used depending on member options, and hence different input render targets are set as input parameters.

As with previous examples, the appropriate shader is determined when required within **MQuadRender::setup**(), and the parameters are updated then. Note that the following code takes advantage of the ability to specify a technique name to retrieve different shader instances (MShaderInstances) within the same “blur” effects file. How shaders are organized is up to the plug-in. The acquired shader is stored in an *mShaderInstance* data member of the viewRenderQuadRender instance.

```
const MHWRender::MShaderInstance *
viewRenderQuadRender::shader()
{
	// Create a new shader instance for this quad render instance
	//
	if (mShaderInstance == NULL)
	{
		MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
```

```
            if (renderer)
            {
                  const MHWRender::MShaderManager* shaderMgr =
                                          renderer->getShaderManager();
                  if (shaderMgr)
                  {
                        // The second argument here is the technique. If desired
                        // an effect on disk can hold different techniques. For each
                        // unique effect + technique a different shader instance is
                        // created.
                        switch (mShader)
                        {
                        case kScene_Threshold:
                              mShaderInstance = shaderMgr->getEffectsFileShader(
                                          "Threshold", "" );
                              break;
                        case kScene_BlurHoriz:
                              mShaderInstance = shaderMgr->getEffectsFileShader(
                                          "Blur", "BlurHoriz" );
                              break;
                        case kScene_BlurVert:
                              mShaderInstance = shaderMgr->getEffectsFileShader(
                                          "Blur", "BlurVert" );
                              break;
                        case kSceneBlur_Blend:
                              mShaderInstance = shaderMgr->getEffectsFileShader(
                                          "Blend", "Add" );
                              break;
                        default:
                              break;
                        }
                  }
            }
      }
```

Continuing on within the **setup**() method, the parameter update logic branches based on shader used. For example, the threshold shader requires one input target and also sets a threshold value, while the blend shader simply sets two input targets. (In this sample code, the targets being used are kept in an array of render targets called mTargets, which is indexed by an enumeration that indicates the location of a given target, such as kMyColorTarget.)

```
      // Set parameters on the shader instance.
      //
      // This is where the input render targets can be specified by binding
      // a render target to the appropriate parameter on the shader instance.
      //
      if (mShaderInstance)
      {
            switch (mShader)
            {
            case kScene_Threshold:
            {
                  // Set the input texture parameter 'gSourceTex' to use
                  // a given color target
                  MHWRender::MRenderTargetAssignment assignment;
                  assignment.target = mTargets[kMyColorTarget];
```

```
                        mShaderInstance->setParameter("gSourceTex", assignment);
                        mShaderInstance->setParameter("gBrightThreshold", 0.7f );
                }
                break;

                case kScene_BlurHoriz:
                {
                        // Set the input texture parameter 'gSourceTex' to use
                        // a given color target
                        MHWRender::MRenderTargetAssignment assignment;
                        assignment.target = mTargets[kMyBlurTarget];
                        mShaderInstance->setParameter("gSourceTex", assignment);
                }
                break;

                case kScene_BlurVert:
                {
                        // Set the input texture parameter 'gSourceTex' to use
                        // a given color target
                        MHWRender::MRenderTargetAssignment assignment;
                        assignment.target = mTargets[kMyBlurTarget];
                        mShaderInstance->setParameter("gSourceTex", assignment);
                }
                break;

                case kSceneBlur_Blend:
                {
                        // Set the first input texture parameter 'gSourceTex' to use
                        // one color target.
                        MHWRender::MRenderTargetAssignment assignment;
                        assignment.target = mTargets[kMyColorTarget];
                        mShaderInstance->setParameter("gSourceTex", assignment);

                        // Set the second input texture parameter 'gSourceTex2' to use
                        // a second color target.
                        MHWRender::MRenderTargetAssignment assignment2;
                        assignment2.target = mTargets[kMyBlurTarget];
                        status = mShaderInstance->setParameter("gSourceTex2", assignment2);

                }
                break;
        }
        return mShaderInstance;
}
```

6.7.11 “Capturing” Render Targets

viewRenderOverrideFrameCache demonstrates one way that a capture can be performed at any given time to a cache. Once cached, the plug-in has additional code that performs playback.

The frame capture is performed by setting a custom user operation as the last operation to be performed. The operation then uses:

- **MDrawContext**::**copyCurrentColorRenderTargetToTexture**() to retrieve an MTexture for caching.
- **MTextureManager::saveTexture**() to optionally save the retrieved texture to disk.

In this plug-in, the override for the **MUserOperation::execute**() method has code similar to the following, where the context is passed in using an MDrawContext.

```
// Retrieve the target as a texture
MHWRender::MRenderer* renderer = MHWRender::MRenderer::theRenderer();
MHWRender::MTextureManager* textureManager = renderer->getTextureManager();
MHWRender::MTexture* colorTexture = context.copyCurrentColorRenderTargetToTexture();

// Save the texture to disk to some desired name (e.g. foo.iff)
MString diskName = "foo.iff";
if (colorTexture)
   textureManager->saveTexture(colorTexture, diskName);
```

Similar capture code can also be used without an MRenderOverride. This is demonstrated in the *blast2Cmd* developer kit example using *pipeline callbacks*. In this case, only a "post frame" callback is required. The key interfaces in MRenderer are:

- A callback function definition that allows for context, as well as user data, to be passed in:
  - typedef void (***NotificationCallback**)(MDrawContext& context, void* clientData)
- The ability to add a named callback at a specific pipeline location. (There is also a corresponding "remove"):
  - MStatus **addNotification**(NotificationCallback notification, const MString& name, const MString semanticLocation, void* clientData);

Optionally, the size of the rendered image (render target) can be controlled via **MRenderer::setOuputTargetOverrideSize**(), while **MRenderer**::**setPresentOnScreen**() controls whether to hide the on-screen presentation of results.

In the following code snippet, the semantic *mPostRenderNotificationSemantic* has been set to **MPassContext::kEndRenderSemantic,** which indicates the end of a frame.

A series of refresh events are queued to capture images over a time sequence from *mStart* to *mEnd* time. For each refresh, the callback function *captureCallback* is executed by the renderer.

```
// Set up notification of end of render. Send the blast command
// to allow accessing data members.
//
MString mPostRenderNotificationSemantic = MPassContext::kEndRenderSemantic;
MString mPostRenderNotificationName = "myCaptureCallback";
renderer->addNotification(captureCallback, mPostRenderNotificationName,
           mPostRenderNotificationSemantic, (void *)this );

// Set override image size.
renderer->setOutputTargetOverrideSize( mWidth, mHeight );

// Temporarily turn off on-screen updates
```

```
renderer->setPresentOnScreen(false);

// Change time and perform a refresh to allow the callbacks to invoked.
for ( mCurrentTime = mStart; mCurrentTime <= mEnd; mCurrentTime++ )
{
       MAnimControl::setCurrentTime( mCurrentTime );
       view.refresh( false /* all */, true /* force */ );
}

// Remove notification of end of render
renderer->removeNotification(mPostRenderNotificationName,
           mPostRenderNotificationSemantic);

// Restore off on-screen updates
renderer->setPresentOnScreen(true);
// Disable target size override
renderer->unsetOutputTargetOverrideSize();
```

The capture code is very similar to the code shown above, as an MDrawContext will be passed in for use by the callback.

6.8 Drawing 2D Elements

There is no predetermined way to draw 2D non-scene elements. In general, they are overlays on a per frame basis. This section will discuss a few possible options.

6.8.1 HUDS using a Locator

One way in which a post draw 2D HUD can be added is by creating a plug-in locator object in the scene.

The *MayaPluginForSpreticle* developer kit example creates a single *spReticleLoc* locator node solely for the purposes of drawing 2D UI for camera based information.

We use the same approach for porting 2d drawing as for 3d drawing. The VP1 MPxLocatorNode::draw() method logic is ported to use MUIDrawManager. There is no inherent difference in the override that the UI drawing is being added from, and hence an MPxDrawOverride was used.

Looking at how a 2d “mask” is drawn shows two differences between VP1 and VP2 approaches:

- State Management:
    - For the VP1 code, a transformation from 3d space to 2d space must be manually set and unset. For example, for OpenGL the GL_MODELVIEW and GL_PROJECTION matrices would need to be set. In addition, if any transparency is involved, the blend states need to be “pushed” and “popped”.
    - For VP2, there is a 2D interface for drawing available on MUIDrawManager which will handle the transformations as required, and will allow interleaved 3D and 2D drawing without explicitly switching matrices to go between the two spaces.
- Depth Location and Order:

- To draw using the VP1 style immediate mode, often a “painters algorithm” is used, resulting in depth testing being disabled and elements being drawn in depth order. The Z location is in world coordinates, and hence must be computed to be at the near clip plane to avoid being obscured by the scene.
- VP2 UI drawables are queued. The appropriate depth priority (**MUIDrawManager::setDepthPriority**()) values can be used to control ordering.

Hence, the VP1 code appears as follows, with depth test disabled, and the use of –ncp incorporated, which is the negative of the near-clip plane plus a “fudge” factor of 0.001 to move it in front of the plane.

```
void spReticleLoc::drawMask( Geom g1, Geom g2, MColor color, bool sides, double ncp )
{
    double z = -ncp;

    // Turn off z-depth test
    glDisable( GL_DEPTH_TEST );
    glDepthMask( GL_FALSE );`

    glBegin( GL_QUADS );

    glColor4f( color.r, color.g, color.b, 1-color.a );

    // Bottom Mask
    glVertex3d( g1.x1, g1.y1, z );
    glVertex3d( g1.x2, g1.y1, z );
    glVertex3d( g2.x2, g2.y1, z );
    glVertex3d( g2.x1, g2.y1, z );
```

The VP2 code simply starts drawing and using the appropriate 2d interface. The Z coordinate in this case is ignored and a default depth priority (which will place the drawing in front of the camera) is used.

```
void spReticleLocDrawOverride::drawMask( Geom g1, Geom g2, MColor color, bool sides,
MHWRender::MUIDrawManager& drawManager )
{
       drawManager.beginDrawable();

       drawManager.setColor(MColor(color.r, color.g, color.b, 1 - color.a));

       MUintArray index;
       index.append(0);
       index.append(1);
       index.append(3);
       index.append(2);

       // Bottom Mask
       MPointArray bottomMask;
       bottomMask.append(MPoint( g1.x1, g1.y1, 0.0));
       bottomMask.append(MPoint( g1.x2, g1.y1, 0.0));
       bottomMask.append(MPoint( g2.x2, g2.y1, 0.0));
       bottomMask.append(MPoint( g2.x1, g2.y1, 0.0));
       drawManager.mesh2d(MHWRender::MUIDrawManager::kTriStrip,bottomMask,NULL,&index);
```

The code for drawing stippled lines is similar, except that the stipple is not set as part of state before immediate mode drawing, but is instead passed in as a line style using MUIDrawManager::setLineStyle().

```
drawManager.beginDrawable();
drawManager.setColor(MColor(color.r,color.g,color.b,1-color.a));
if(stipple) {
   drawManager.setLineStyle(2,0x00FF);
}

drawManager.line2d(MPoint(x1,y1),MPoint(x2,y2));

drawManager.endDrawable();
```

The same VP1 stipple patterns for the most part can used for the VP2 interfaces

```
// VP1 OpenGL Stipple
if (stipple)
{
   glEnable (GL_LINE_STIPPLE);
   glLineStipple(2,0x00FF);
}
```

Overall usage of MUIDrawManager should reduce the dependence on draw API specific code, thus reducing the overhead caching and restoring state.

For a full set of possible options available via MUIDrawManager, see the uiDrawManager developer kit example for code that performs 2d drawing, including 2d and 3d geometric primitives, text with various font options and icons.

6.8.2 Manipulators and In-View Editors

Another option that can be useful for tying attributes to 2D HUDs is to take advantage of in-view editors. These are 2D HUDs that allow for interactive modifications of attribute values. They are drawn within the 3d viewports, and are layered on top of any other UI drawn using MUIDrawManager or internally.

See the footPrintManip plug-in in the developer kit for an example. . Within the MPxManipContainer::**connectToDependNode**() method, a call to **addPlugToInViewEditor**() is used to add attributes of interest. In this case, the scale attribute is added:

```
MStatus footPrintLocatorManip::connectToDependNode(const MObject &node)
{
    MFnDependencyNode nodeFn(node);
    MPlug sizePlug = nodeFn.findPlug("size", &stat);
    // Allow the user to tweak the size via the In-View Editor
    //
     addPlugToInViewEditor( sizePlug );
      …
}
```

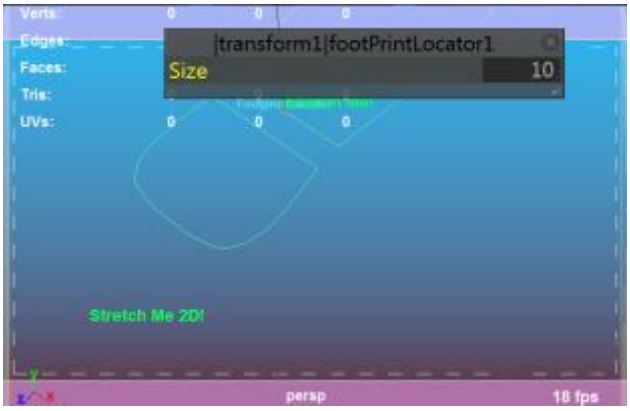


Figure 18: Snapshot shows the footprint manipulator. The “Stretch Me!” and “Stretch Me 2D!” UI are drawn using MUIDrawManager for the manip. The in-view editor is drawn for the plug-in internally. Note that it is layered to be on top of the footprint plug-in and spReticle UI, as well as the internal UI (polygon statistics shown here as an example).

6.8.3 Image Planes

No additional overrides are required for plug-in image planes, as this interface is only an image provider. Plug-in image planes are automatically given the correct internal classification and will hence use internal evaluators instead of explicitly requiring a plug-in override. The snapshot below shows the *customImagePlane* plug-in example (showing an “Alpha Channel” image. As all evaluation is done internally, the depth composition is consistent with internal logic.

Figure 19: Image planes using internal logic to determine depth relative to 3d scene elements and other 2D UI elements. In this case, the default image plane depth places the image behind all other elements.