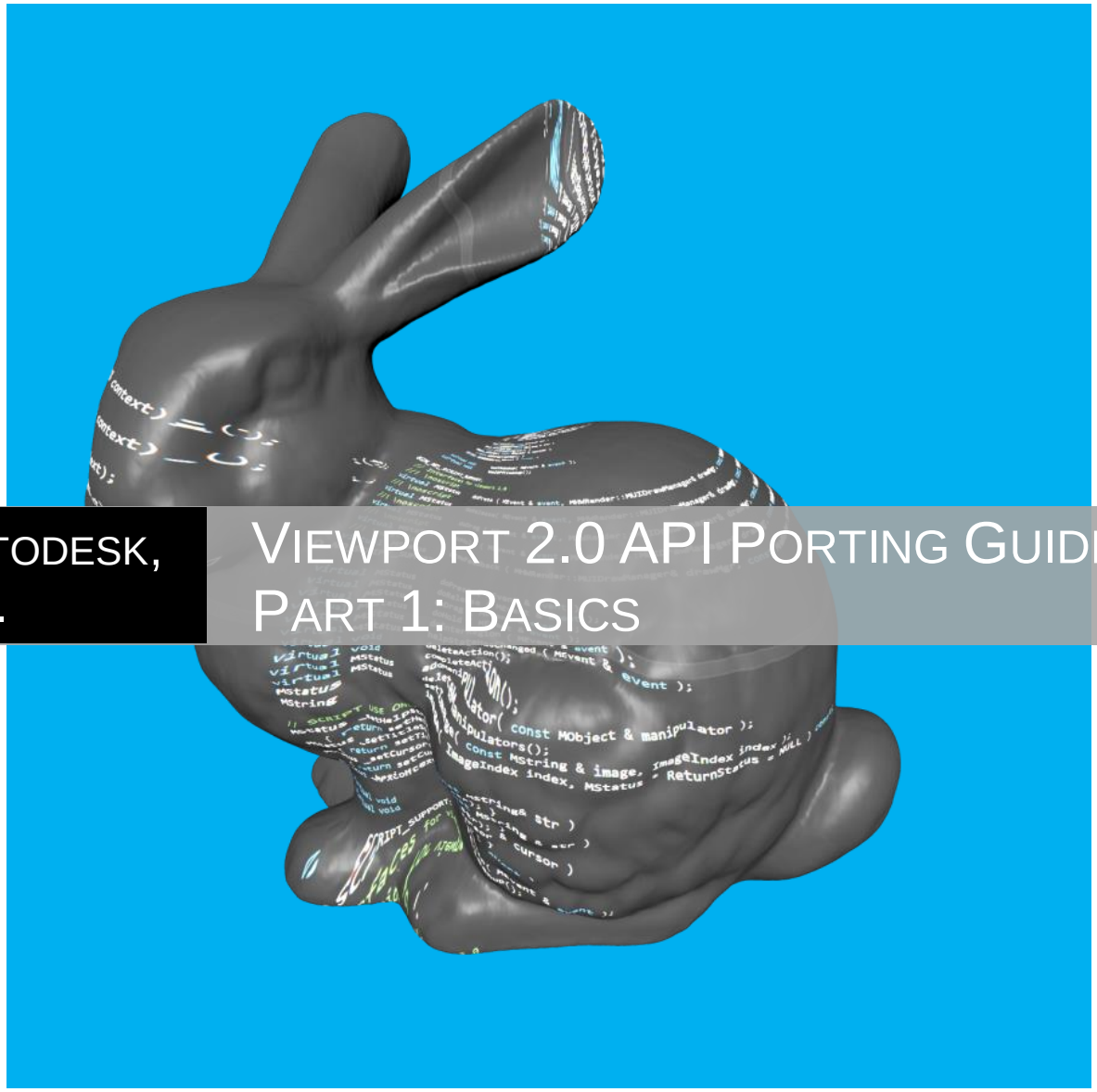
6/1/2017



AUTODESK, INC.

VIEWPORT 2.0 API PORTING GUIDE PART 1: BASICS

Getting Started Guide for Programmers Porting Plug-ins from VP1 to VP2 (Part 1) in Autodesk® Maya®

Revision History

4/1/2015	Initial version for Maya 2016
6/1/2017	Updates for Maya 2018: 3.6 Categorization and Consolidation - Added introduction for consolidation modes and an inspection approach. 4.2 Geometry Evaluator Interfaces - Updated introduction for the interfaces.

1. Table of Contents

2. Background

The main goal of this document is to describe, compare and contrast key differences between the Viewport 2.0 (VP2) and the Legacy Default Viewport / Viewport 1 (VP1) systems in Autodesk® Maya®.

Viewport 1 can be viewed as a collection of draw calls controlled per object, while Viewport 2.0 introduces a rendering framework layer upon which various renderers in Maya are built.

VP2 uses a single consistent and extensible backend system, which is cross-platform and draw-API agnostic. This backend is used in various Autodesk products company wide, including products such as AutoCAD® and 3ds Max®.

The main issues that VP2 attempts to address are VP1 scalability and fidelity.

This guide is divided into the following sections:

Section 3: This section provides a high level comparison between the rendering frameworks of the old viewport versus Viewport 2.0.

Section 4: This section outlines the main Viewport 2.0 API interfaces that expose entry points for geometric object and shader evaluation, as well as rendering pipeline and resource management.

Section 5: This section provides recommendations on how to port various categories of Viewport 1 items to the Viewport 2.0 API.

Part 2: Refer to *Viewport 2.0 API Porting Guide, Part 2: Porting Details* for this document. This document provides more detailed descriptions and code examples for porting plug-ins to the Viewport 2.0 API.

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

Underlining will be used to indicate definitions, and *italics* for points of interest. **Boldface** will be used to indicate important API classes.

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

3. VP1 versus VP2

3.1 Introducing a Renderer

VP2 is not a viewport. Instead, it is a full rendering framework that comprises GPU resources plus a rendering pipeline. There is a persistent rendering database with an internal “render node” hierarchy.

In contrast**,** VP1 is a series of CPU resources plus a draw request queue.

3.2 Attachment Model

VP1 implicitly creates heavy-weight UI nodes (shapes) in a 1:1 correspondence with leaf level DAG nodes. The association is immutable and the UI nodes are always responsible for handling all the drawing, selection and evaluation for all path instances. There is no correlation between UI nodes and shading nodes and no formal shader concept.

VP2 avoids introducing new DAG / DG nodes by using explicit “attachments” to light-weight constructs called sync objects. These constructs represent a render node that is attached to (associated with) a DAG item or a DG shader node.

Sync objects are *not* self-evaluating and self-rendering as VP1 UI shapes are.

Sync objects are part of the internal rendering “database” for VP2 and are not exposed to the user, although the attachment mechanism is exposed.

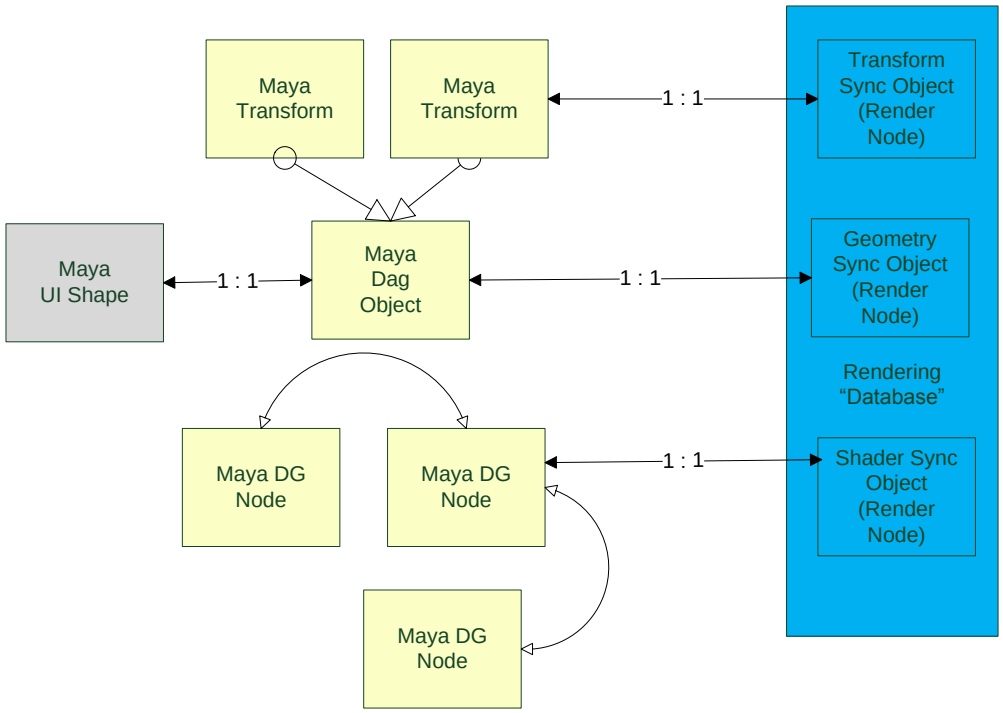


Figure 1 : This diagram illustrates a sample Maya DAG hierarchy and a sample shader graph and the different attachment models used by VP1 versus VP2. In VP1, a UI shape node (grey box) is always automatically defined for each leaf DAG node. The association is immutable. In VP2, the sync object (blue box) is dynamically set. This example demonstrates both a transform level as well as a leaf level DAG association. Each shading node in the graph has a corresponding sync object.

3.3 Change Monitoring

3.3.1 VP1

VP1 is based on various nodes broadcasting or “pushing” dirty messages through the DG via dirty propagation.

A refresh, or other plug evaluation, causes the scene DAG hierarchy and shading graphs to be re-traversed to perform an update. Data on nodes are “pulled” at this time. This traversal *always* occurs even if there is no change to the scene, as all node data is always assumed to be “dirty” and thus requires evaluation.

There is no concept of a separate evaluation system. All processing is handled by the node or its associated UI shape.

3.3.2 VP2: Change Management System

In order to allow nodes to indicate that a change has occurred, a change management system exists.

- This allows Maya nodes to “push” change messages (*not* dirty messages).
- These change messages are queued for a later independent interpretation.
- Only when this interpretation occurs is new data “pulled” from nodes (See next section.)

Note that change monitoring *does not* necessarily force a refresh or invoke evaluation.

3.4 Evaluation

3.4.1 VP2: Evaluators

Each sync object is associated with a class that performs Maya node evaluation (an evaluator).

The association is defined by using the same classification string on a DAG/DG node as for an evaluator.

“drawdb” is the root classification string. Specialization can be defined by using further classification separators; for example “drawdb/geometry” for geometric shapes, and “drawdb/shader” for shaders.

By default, all internally supported VP2 nodes have a “drawdb” classification and a 1:1 association with a registered evaluator. Thus, mesh shapes have a mesh evaluator; each shader node has a unique shader evaluator; joints have their own evaluators and so forth.

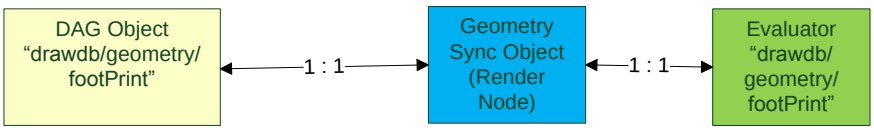


Figure 2 : A DAG object is with associated with a sync object and an evaluator. The association is provided by using the same classification string for the dag object as for the evaluator. The classification is “geometry”, and in particular, the sub-classification: “footprint” geometry.

Note: There are various classes matching the name pattern **M*Override** that allow for API evaluator definition. **MDrawRegistry** is used to define node/evaluator associations based on classification.

Depending on the evaluator type, each can have distinct entry points designated for:

- DG evaluation update: This can include bounding box updates as well as various CPU geometry data or display property updates.
- GPU device update: This can include creating and updating GPU resources such as textures, shaders, states, data streams, and so forth.
- Drawing: Internally defined evaluators *never* draw themselves.[1]

These entry points are called as appropriate for data or rendering updates.

Note: API evaluators expose different configurations of entry points. For (1) it is possible for API evaluators (MPxDrawOverrides) to provide an explicit drawing override.

3.4.2 VP2: Tasks

Work during a frame update can be broken down into: “update tasks” and “rendering tasks”.

- Update tasks: Can include visibility testing, bounding box evaluation and camera volume culling, as well as scheduling the execution of DG interfaces on evaluators.
- Render tasks: Includes GPU shader generation from shade trees, texture and lighting updates, as well as the actual draw for rendering and selection.

Update tasks perform the majority of Maya node evaluation by examining change messages and calling into evaluators to perform the appropriate update. For shaders, shader building and texture evaluation may be queued to be handled by a render task. Lighting updates generally occur at render time.

When enabled, the evaluation manager (EM) can disable dirty propagation for time changes and manipulation, and also work independently of change messages. The EM will attempt to traverse forward through a dependency graph and compute plugs and cache node data. When an evaluator attempts to pull data, it will find that the data is already complete and cached for the aforementioned cases.

3.4.3 VP1

Draw requests are returned from UI objects and queued by a request handler. There is no communication between the UI objects and the request handler to ensure separation of DG versus non-DG updates, device access, and drawing. The API reflects this internal disconnect.

3.5 Render items

3.5.1 VP2

Evaluators that draw produce “render items**”**.

A render item is the smallest atomic unit that can be drawn, and roughly comprises geometry and an instance of a shader (shader instance).

- Render items flow through a rendering pipeline.
- Update tasks and rendering tasks may modify the render item’s data.

- Render items may be persistent (owned by the sync object) or transient depending on the use case. For example, temporary UI items are transient, while render items for mesh shapes are retained.

Global lifetime management of associated GPU resources is linked to render items that reference these resources (such as: textures, shaders, geometry, and so forth).

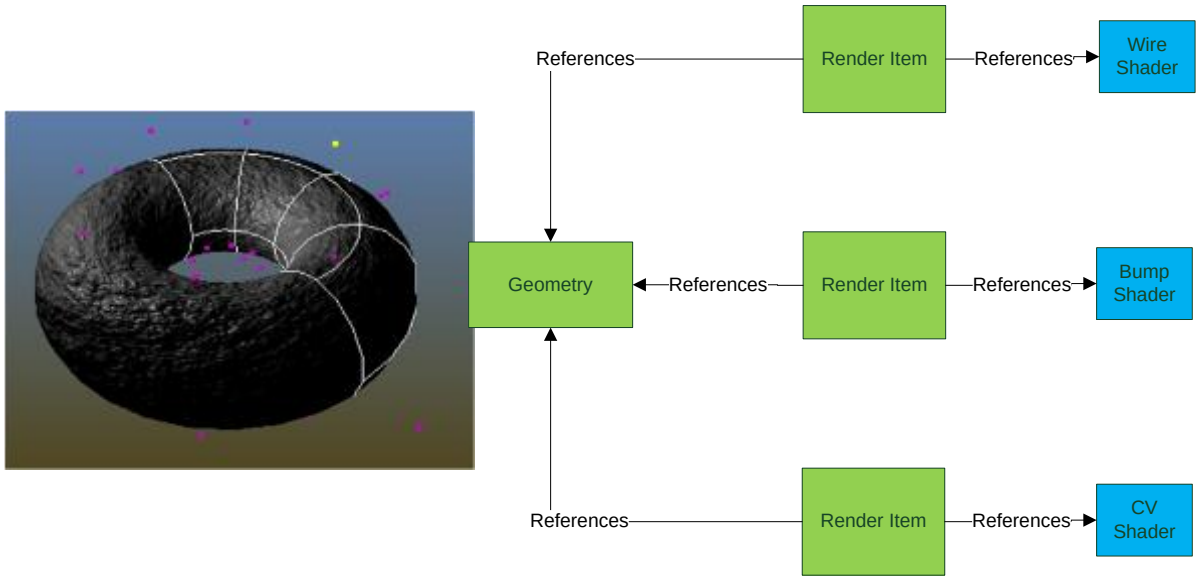


Figure 3: In this example, three persistent render items are used to draw different parts of the torus: one for the wireframe, one for the bump map, and one for the CVs. The items reference a set of data streams on a geometry container. Although not shown here, the shaders and texture used for bump mapping are also persistent resources. The bump shader would reference the bump texture, and the render items reference the appropriate shaders.

Tip: See **MRenderItem** for API exposure of render items. Refer to the GPU "*manager*" classes (such as MShaderManager) in the API for resource management interfaces.

3.5.2 VP1

Draw requests appear similar to render items; however, they are quite different – requests are only transient descriptions of what to draw, and are arbitrarily managed by UI objects (not the renderer).

Locally, it is up to each UI object to retain any associated persistent resource data. Without GPU resource handler interfaces, performance issues may occur during the constant retransfer of CPU to GPU resources (such as geometry buffers).

3.6 Categorization and Consolidation

Render items are categorized into a series of lists or "buckets" based on whether they are persistent or transient, as well as various display criteria. In the API, this is exposed as render item "types" (such as UI versus non-UI), display modes (wireframe versus shaded), and properties such as "transparency". Categorization may be used to group items with similar drawing properties that can help with performance by avoiding unnecessary device state changes.

Consolidation is a VP2 feature used to mitigate the "small batch problem": that is, when the amount of overhead required to draw a small amount of geometry results in undue performance loss. To help alleviate this problem, render items may be consolidated (merged). *Note that, as a result, the original render items created may not be the ones drawn.* By default, a hybrid mode combining traditional static consolidation and multi-draw consolidation is used by VP2.

- Traditional static consolidation improves the drawing performance for static shapes only.
- Multi-draw consolidation improves the drawing performance for matrix-animated (i.e. non-deforming) and static shapes, however, it requires OpenGL Core Profile and most recent platform configurations. E.g. it is supported by most recent GPU architectures and graphics drivers on Windows and Linux, but not supported on Mac OS X due to missing graphic driver support.
- The hybrid mode allows certain render items to switch dynamically between the two schemes, e.g. when a static shape starts non-deforming animation, its render items will be pulled out of traditional static consolidation and re-examined for multi-draw consolidation.

Render items may be considered for consolidation at a higher level based on its classification string, categorization or display properties, as well as at a finer level based on shader and geometry "signature" (number of data streams and format of each stream). Consolidation also takes into account the spatial proximity of the render items.

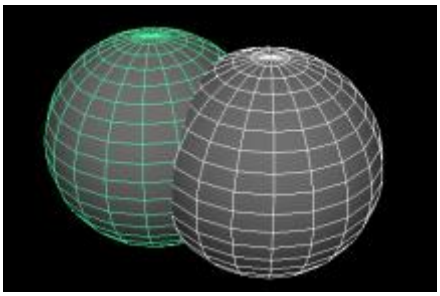


Figure 4: There are two shapes in this example. Each produces a wireframe and a shaded render item. The two shaded render items can be consolidated as they share the same shader which requires the same data streams (position and normals). The two wireframe items cannot be consolidated because, even though they share the same shader definition (which requires a position stream), the color parameter used for each instance of the shader differs. The wireframe and shaded items have different display mode categorization as well as different shaders.

Note: See the **MRenderItem** class description in the C++ API Reference which mentions that custom user data is also considered part of the consolidation criteria.

Selection affects consolidation at the render node level; therefore, all render items for the selected node must to be re-examined for consolidation.

Transient items (such as UI) are not consolidated.

Note: Different API interfaces provide different levels of support for consolidation. Per interface specifics are described in more detail in Sections 4, and 5.

To inspect categorization and consolidation, debug tracing of VP2 render pipeline can be enabled by MEL.

```
ogs -traceRenderPipeline on;
```

The count of render items categorized into each list will be displayed in the Script Editor per every refresh. The following statistics are from the example shown in Figure 4, where the [Opaque UI] list contains the wireframe render items and the [Opaque] list contains the shaded render items.

```
// -- VP2 Trace[modelPanel4][3d Beauty Pass] //
// Count for list: [Opaque UI] = 2 //
// Count for list: [Opaque] = 2 //
```

After consolidation is performed, the number of render items in the “Opaque” list is reduced from 2 to 1.

```
// -- VP2 Trace[modelPanel4][3d Beauty Pass] //
// Count for list: [Opaque UI] = 2 //
// Count for list: [Opaque] = 1 //
```

VP1 does not support consolidation.

3.7 Render Pipeline

VP2:

For simplicity, the complete VP2 pipeline can be summarized as follows:

- The update and render tasks (section 3.4)
- Categorization and consolidation (section 3.6)
- A series of rendering operations presented in graph form is a render graph. The render graph configuration is flexible and is adjusted based on output destination as well as destination properties. For example:
    - There are different configurations for 3d viewports, the UV Editor and Render View.
    - Configuration changes can occur based on viewport or rendering display options.

Within the pipeline, the key data elements that flow through the pipeline are render items.

Groupings for logical parts of the pipeline are called phases. Semantic pass labels can be associated with different phases, or sub-graphs within the render graph.

The pipeline runs with underlying context information. Different levels of context include:

- Frame Context: per frame information. This can include information such as output target size, current camera, viewport display options, and so forth.
- Draw Context: draw time information such as GPU device state.
- Pass Context: pass information. Mostly this is a series of nested names. Each nesting represents a semantic pass within a semantic pass; for example, a “transparency” pass invoked within a “beauty” pass.

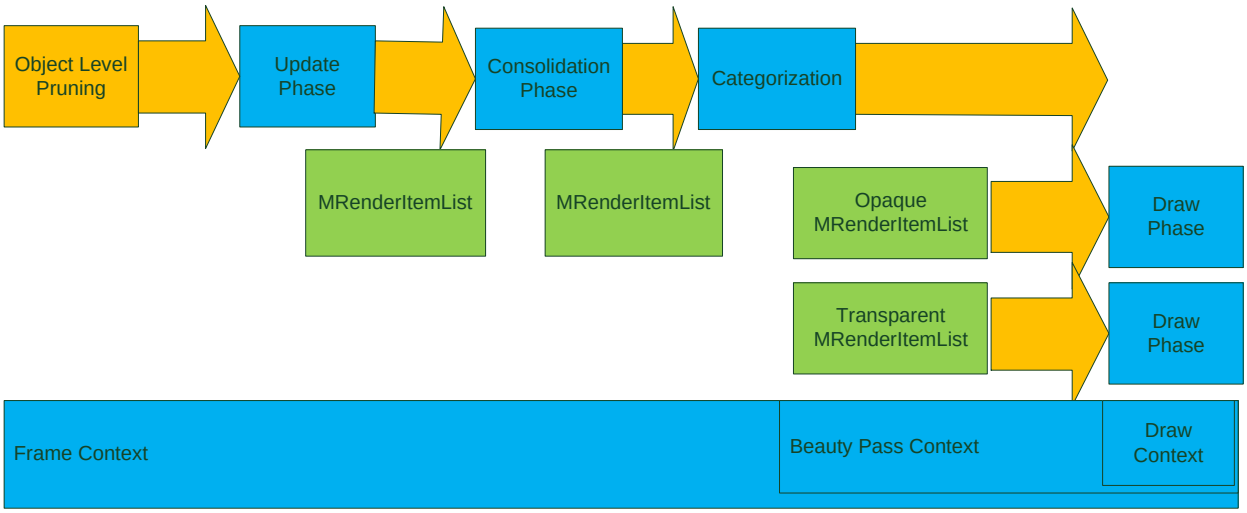


Figure 5: This example begins first with basic object pruning, followed by the DAG/DG node update to produce render items. These items may be consolidated to form a new set of items. Categorization can form new lists. In this example, both an opaque and a transparent object list are produced. These lists are then passed through a render graph. Finally, render items are drawn. In terms of context, for simplicity, the render graph's semantic pass is a "beauty pass", and the drawing within this beauty pass occurs within a draw context. The entire frame rendering occurs within a frame context.

Context management helps to reduce issues such as redundant state execution as well as pipeline stalls due to direct device access.

Note: Custom pipeline configurations are possible in the API via "render overrides". See **MRenderOverrides** for "render overrides", and **MFrameContext** / **MDrawContext** / **MPassContext** for context interfaces.

The association between lighting and shaders on render items occurs "late" in the pipeline at draw time. The lighting information is thus available via the current draw context. All lights within the scene are available, but depending on the lighting mode requirements, only some may be active.

Light parameter updates may require additional passes in the render graph, such as shadow passes with the appropriate "shadow pass" semantic exposed.

Note: Access to lighting information can be obtained via the **MLightParameterInformation** class.

VP1:

There is hard-coded custom code per output destination. Only 3d viewports allow for an enumerated "multi-pass" or a single free-form drawing pass via viewport render options in the API.

There is no formal context state that can lead to pulling data directly from a device. All states must be "pushed" and "popped" to ensure state safety.

VP1 has fixed function lighting as part of the GPU device context, but this information is not directly associated with the actual light objects in the scene, thus requiring direct node evaluation, or extraction from the hardware device itself.

3.8 Core Shader / Geometry Update Mechanism

The following figure demonstrates the logic behind the shader/geometry update phase that occurs during the production or update of a render item in Viewport 2.0.

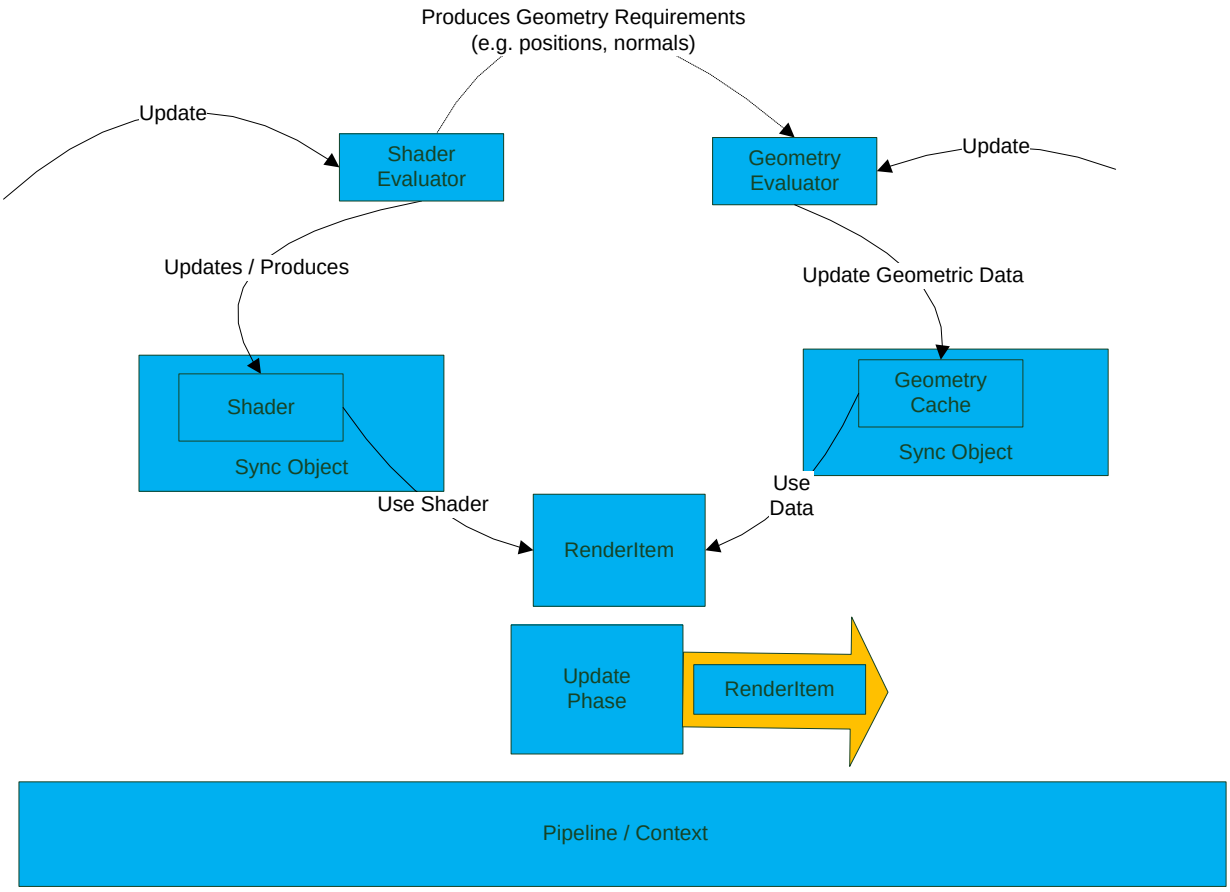


Figure 6: The basic dependencies for updating a render item for a shader assigned to a DAG object is shown above. Both the shader instance and geometry elements of the render item must be updated respectively. A shader instance update can trigger a geometry update. The corresponding evaluators are shown.

Every Maya shading engine assigned to a DAG object results in the requirement for a render item for the associated DAG object. A shader evaluator is required to produce the shader instance part of the render item, and a geometry evaluator is required for the DAG object to produce the appropriate geometry to match the shader. Once an item is updated, it will flow down the pipeline.

As the production of render items is a core element of the update system exposed via the API, the following sections will delve more deeply into its internal details.

3.8.1 VP2: Shader Definition

All internally produced render items use programmable shader instances.

For the most part, this can be interpreted as having an effects system where one shader can be thought of as one effect, and a shader instance an effect instance.

- Fragment Effects: Shaders that are created based on the shader "fragments" (procedures) associated with each Maya shading node.
- Non-Fragment Effects: Shaders that are created from files on disk or buffers.

Logic for shaders is currently written to fit a forward renderer that can have multiple passes.

Both shader and light evaluators exist to perform DG evaluation and to create appropriate fragments and assign appropriate parameters. Internally, graph evaluation produces fragment effects.

Features that affect the render graph (such as screen space ambient occlusion, depth of field, and color management) "inject" extra passes and use 2D custom shaders. These shaders generally use non-fragment effects.

Note: See **MShaderInstance**, **MPxShaderOverride**, **MPxShadingNodeOverride** in the C++ API Reference.

3.8.1.1 VP2: Fragment Effects

Each internally created shading node's shader evaluator defines a shading fragment. The connections within the Maya shading graph are used to derive the connections between the corresponding shading fragment parameters.

When translating a shading network, Maya traverses upstream from a surface shader. For each node visited, the fragment used for the node is obtained from the evaluator and connections are made which approximate the DG connections of the nodes in the shading network.

Following traversal, lighting and geometry information are attached and compilation produces the final shading effect.

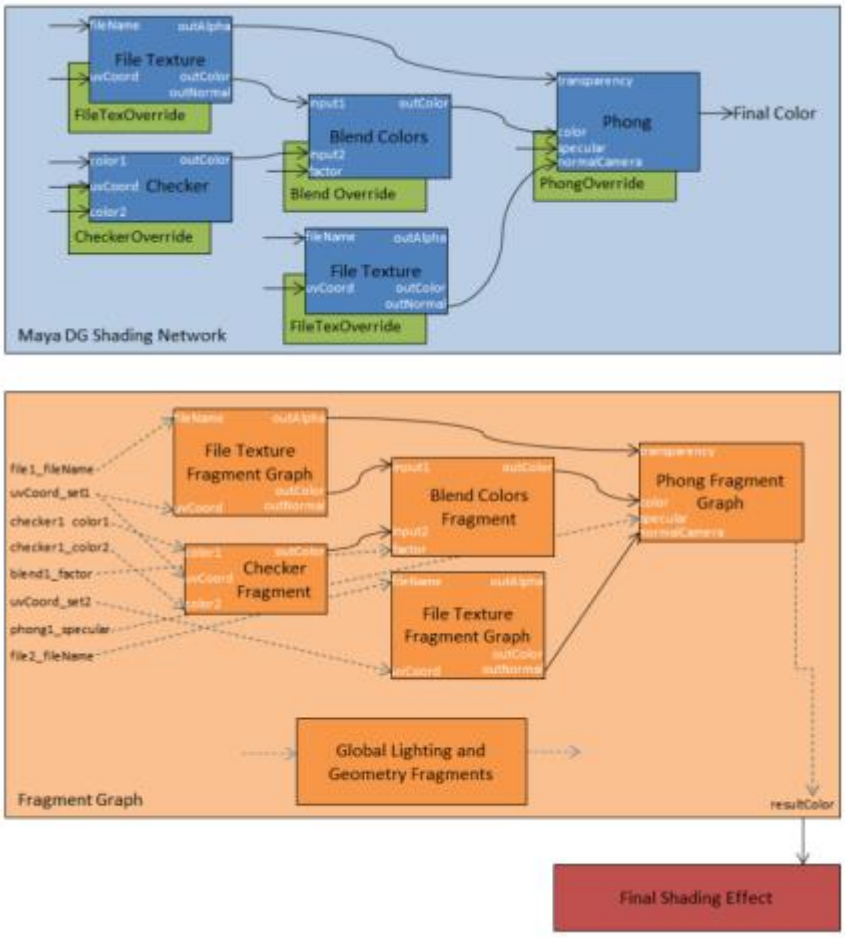


Figure 7: Shows a sample Maya shading network and its corresponding hardware fragment graph which is used to create the final shading effect. Each shading node has an associated evaluator.

3.8.1.2 VP2: Fragments and Fragment Graph Management

Shader fragments and fragments graphs are managed by a fragment manager, where a fragment graph is a connected set of fragments.

XML formats for fragments and fragment graphs are fully defined by XML Schema Documents (XSDs). The format currently allows the definition of code for a Cg, and/or GLSL and/or HLSL function.

API interfaces allow for custom fragments and the reuse of existing internal fragments.

Note: See **MFragmentManager, MPxShadingNodeOverride** in the C++ API reference. Supported XSD files can be found in the bin/ShadeFragment folder of the Maya installation.

3.8.1.3 VP2: Non-Fragment Effects

There are various ways to directly create a shader effect that does not involve building fragment graphs. This includes text and pre-compiled effects on disk and effects stored in strings. Preprocessing macros can be accepted as part of the compilation process.

Cross product usage of the same effect is possible. For example 3ds Max and Maya each provide a custom macro to identify compilation within the respective products.

In all cases, because these are standalone effects, no additional shader fragments can be attached to them automatically. For example, light fragments cannot be added.

3.8.2 VP2: Connection between Shaders and Geometry

A geometry evaluator is associated with each DAG object via its sync object. The geometry evaluator manages render items as well as geometric data that are owned by the corresponding sync object. Geometric data consists of vertex and index buffers. (Note: See **MGeometry**, **MVertexBuffer** and **MIndexBuffer** in the C++ API reference).

Each shader assigned to a DAG object requires a render item to be created and managed by the geometry evaluator. Each shader evaluator update produces a set of geometry requirements (data stream descriptions). (Note: See **MGeometryRequirements** in the C++ API reference).

The vertex buffer requirements are determined by merging the individual requirements for each render item associated with a DAG object.

Render items reference a single index buffer. Index buffer data is determined by the topology used for drawing (for example, a line versus a triangle), as well as component filtering. Additional index buffer requirements can be specified for non-drawing purposes, such as topology data extraction for tessellation shaders.

It is important to note that a geometry evaluator update is independent of shader update as the geometry evaluator does not need to know how or where data is used.

The following diagram demonstrates both the shader evaluator generation of requirements as well as the geometry evaluator handling of requirements to fill in the appropriate data and index buffers:

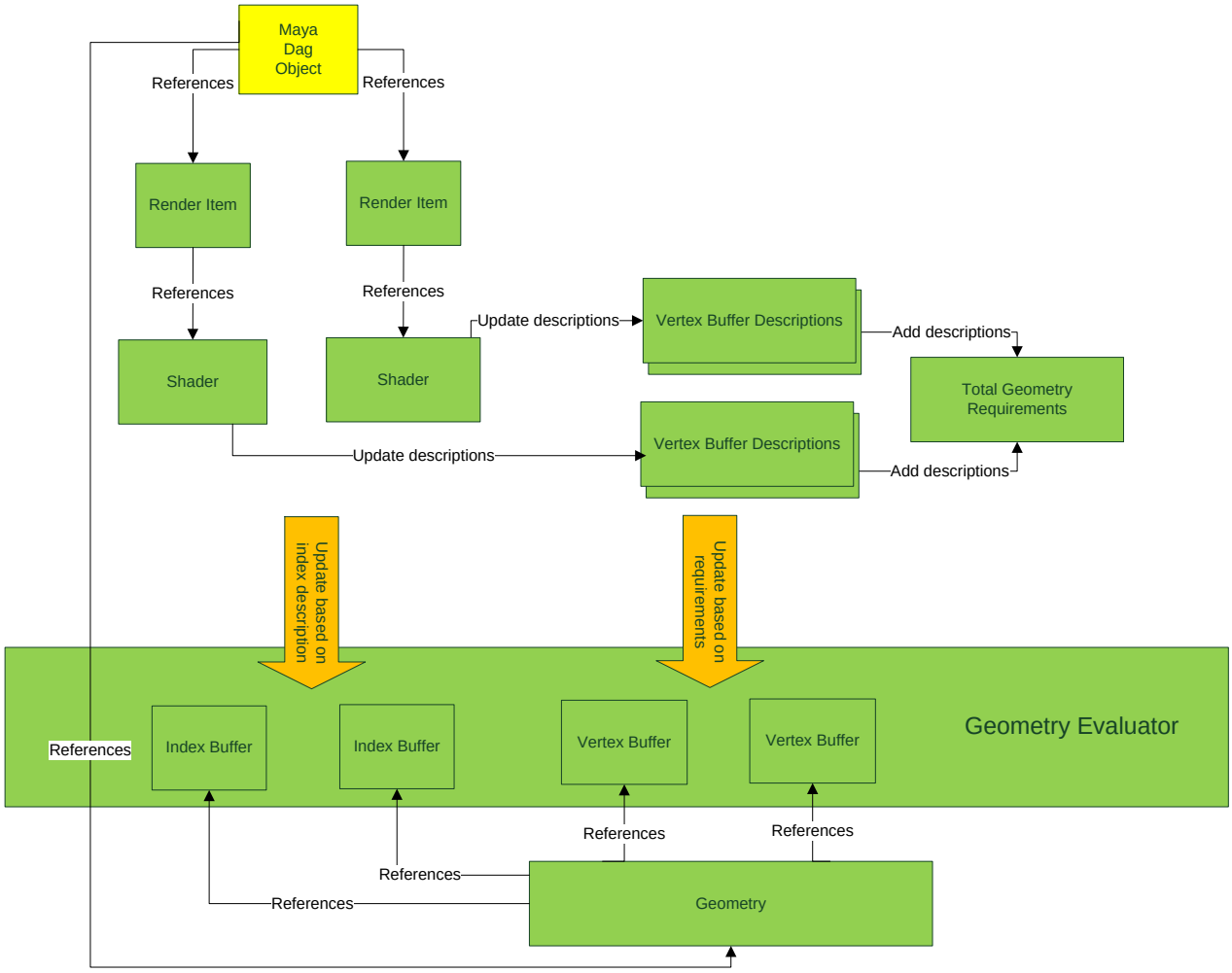


Figure 8: In this example, there is one dag object with two render items. Each render item has a shader instance associated with it. During shader evaluation, each shader instance will update its vertex buffer requirements (descriptions). When the geometry evaluator performs an update, it will update the vertex buffer and index buffers (per render item).

3.8.3 VP2: "Shader Overrides"

Vertex buffer requirements may result from more than just the shader assigned to a given DAG object. Often, there can be additional passes required to compute intermediate buffers. The shaders used for these passes may differ from the shader assigned to an object for beauty pass drawing, and such shaders are thus called shader overrides. Each of these shaders has geometry requirements that can contribute to the total requirement.

The evaluation of light nodes is done via internal light evaluators. These light evaluators may require the computation of shadow maps and hence require shadow map passes to be executed. Each of these passes can have a "shadow shader" which has a set of geometry requirements.

For post processing, motion blur may require normals to be drawn to a buffer. In this case, the shader for the normal pass will have a set of geometry requirements.

As the render loop can be defined via the API, any shader overrides used for render loop overrides will also affect the total geometry requirement.

3.8.4 VP1

Options are limited to internal fixed function shader or API shader nodes, with no formal concept of a fragment, a shader, or shader separation from geometry.

Handling of shader geometry requirements is performed by pulling directly from shapes, as *there is no concept of a separate shader evaluator and geometry evaluator*.

All handling is generally performed within the “draw” call for a UI shape, or via plug-in hardware shader “draw” logic. There is no concept of customized vertex or index buffers.

3.9 Selection Picking

In Maya 2016, picking can be processed through a VP2 “picking pipeline”. This can be thought of as a custom render pipeline, which includes a custom render graph.

“Pick items” do not exist, although render items can be marked as being used for picking for API geometry evaluators as well as manipulator handles.

It should be noted that picking is a separate mechanism that is utilized by the higher level logic for selection. The selection logic has not changed and is shared between VP1 and VP2.

VP1 picking is supported using raw OpenGL draw code and software pick buffers. No DirectX or OpenGL core profile equivalent support exists.

4. The VP2 API at a Glance

This section outlines the various interfaces available for use with VP2.

Guidelines for choosing a suitable interface for a given object type or porting scenario is given in the next section.

4.1 General

All interfaces use the namespace *MHWRender* and files are in the *OpenMayaRender* SDK include area. The namespace can be examined in the Maya Developer Help.

- M*Override: These classes are evaluators.
- MDrawRegistry: This class allows registration of evaluators with a dag object or a shader node.
- MRenderer *(MViewport2Renderer* file): This class provides global renderer interfaces and render loop overrides.
- M*Manager: These classes manage GPU resources
- M*Context: These classes expose pipeline context.

4.2 Geometry Evaluator Interfaces

- MPxGeometryOverride: evaluator that provides geometry for a given object. It is similar to internal evaluators used to support native Maya objects in VP2 and can thus use all internal performance schemes. “Dirty” must be forced on geometry changes.
- MPxSubSceneOverride: evaluator that controls all data management. Updates can be controlled explicitly. A simplified version of traditional static consolidation can be used for specified render items.
- MPxDrawOverride: a special evaluator that controls its own drawing via a draw callback. Update is called every frame by default. There is no integration with any shading effects.

4.3 Shader Evaluator Interfaces

1. MPxShaderOverride: evaluator that supports drawing for old MPxHwShaders. This is the most complex interface for adding shader support if the plug-in manages its own shaders and drawing.
2. MPxShadingNodeOverride: provides a fragment to an evaluator that allows the fragment to be used when creating a shader from a shade tree.
3. “Pure attributes”: There is no real interface here. The attribute names are checked to see if they match the parameters on a “stock” shader. This is roughly akin to providing parameter values to the fixed function shader in VP1. The underlying evaluator uses the attribute values to update parameters on a fragment used when creating a shader from a shade tree.

Note that there is no concept of a separate light evaluator in the API. Light access is, however, available explicitly for MPxShaderOverrides by querying light information from MDrawContexts, and are implicitly calculated for internally generated shaders (bullet points 2 and 3).

4.4 Renderer Interfaces

Instances of an MRenderOverride can be used to replace the VP1 classes: MPx3dModelView or MViewportRenderer.

These are not evaluators but render graph overrides. You can create an override using a series of render operations:

- MSceneRender: Render a single pass of the scene. It is possible to have various overrides.
- MUserOperation: User defined drawing. These are generally not required if all drawing is done using Maya.
- MQuadRender: 2D quad blit with a custom shader instance.
- MPresentOperation: On screen presentation.

4.5 Manipulator Entry Points

The underlying evaluators will queue transient UI for drawing (See Section 4.7). Support is provided via VP2 methods on existing API classes: MPxManipulatorNode, MPxManipContainer.

4.6 Tool Context Entry Points

Contexts are not evaluators since they have no associated Maya nodes.

Drawing of transient UI (See Section 4.7) as “overlays” is supported via additional methods on existing API classes: MPxContext, MPxTexContext.

4.7 UI Drawing

For transient UI drawing, a MUIDrawManager can be used**.** This class does not directly expose render items to the user, but instead provides an interface that looks like fixed function drawing, but is in fact creating transient render items (with corresponding shaders and geometry) for drawing.

Items are executed at the appropriate place in the pipeline based on the display characteristics defined for the drawables.

MUIDrawManager is accessible for these evaluators: MPxDrawOverride, MPxGeometryOverride, MPxSubSceneOverride, MRenderOverride, as well as manipulator and tool context entry points.

4.8 Resource Classes

Resource manager classes are generally used in conjunction with the provided VP2 interfaces, but are made generally available to allow external rendering to take advantage of the resource management capabilities of each manager. Some resources can be self-binding, while others allow access to draw API native GPU handles.

4.8.1 MShaderManager

The manager handles instances of programmable shaders (MShaderInstance).

MShaderInstances can be used with MRenderItems and rendering operations (on MRenderOverrides) as a way to use shaders without writing the underlying system to support a hardware shading language.

The provided “stock” shaders can save time and eliminate the need for plug-ins to handle draw API specifics, as implementations for all currently supported back-ends exist. For example, a “fat point” shader is provided which uses geometry shaders. All stock shaders are OpenGL legacy, OpenGL core profile and DX11 compliant.

MShaderInstances can also be used with raw geometry routing in MDrawOverrides and MUserOperations, as they can be self-bound / unbound.

4.8.2 MFragmentManager

MFragmentManager provides the functionality for managing Viewport 2.0 shade fragments (procedures) and fragment graphs (connected procedures). Both are defined using an XML format and can be specified using either a file or a string buffer.

Internally existing fragments can be accessed in addition to the definition of new fragments.

Fragments can be used directly for shading graph evaluation (per node), or to create specific shading instances (MShaderInstance) for use in other interfaces.

4.8.3 MTextureManager

The manager handles hardware textures (MTexture), instead of maintaining additional software rasters (as in VP1).

MTexture instances can be used where textures are required. This includes binding to parameters on an MShaderInstance.

4.8.4 Geometry Classes

The classes for geometry storage that are referenced from render items are:

- MGeometry: Vertex and index buffer (stream) container (MVertexBuffer, MIndexBuffer). These constructs are associated with a sync object and cannot be created nor destroyed.
- MVertexBuffer, MVertexBufferArray: A GPU data stream, and a list of data streams
- MIndexBuffer: A GPU index stream.

The classes that allow specification of requirements to geometry evaluators are:

- MVertexBufferDescriptor, MVertexBufferDescriptorList: A vertex buffer description and a list of descriptions. Shaders are responsible for supplying the appropriate descriptions. MIndexBufferDescriptor, MIndexBufferDescriptorList: An index buffer description and a list of descriptions. Shaders that require custom tessellation, as well as geometry extraction interfaces, specify index requirements.
- MGeometryRequirements: A logical description of index and vertex streams that a shader requires (MVertexBufferDescriptorList, MIndexBufferDescriptorList).

4.8.5 MGeometryExtractor

Interfaces such as MFnMesh and MfnNurbsCurve provide access to topologically complex data structures that are best suited for data editing, and in general have multi-indexed data streams; for example, different indexing to indicate sharing for normals, texture coordinates, colors, and so forth.

MGeometryExtractor, in contrast, compresses the data to be suitable for use by a single index stream. Sharing information is not lost. This data can then be used directly for render purposes or used to create GPU data (MVertexBuffers, MIndexBuffers).

4.8.6 MRenderTargetManager

Handles “render targets” (MRenderTarget). Render target terminology comes from DirectX. For OpenGL, this can be thought of as off-screen textures. Targets can be used for render loop overrides, but are also accessible during per object drawing via an MDrawContext.

4.8.7 MStateManager

The manager maintains unique GPU state objects that can be used at draw time to reduce undue state changes. This includes raster (MRasterizerState), sampler (MSamplerState), blend (MBlendState), and depth-stencil (MDepthStencilState) states.

4.8.8 Context Information

Unlike resources, context information is only available at certain times during render graph execution:

- **MFrameContext**: Per frame information. Includes transform matrices.
- **MDrawContext**: Draw time information. Includes pass and lighting information and target information.
- **MPassContext**: Information about pass semantics
- **MLightParameterInformation**: Per light Information, including shadow maps.

4.9 Entry Point Summary

The following diagram presents an overview of all the main API classes and how they relate to each other:

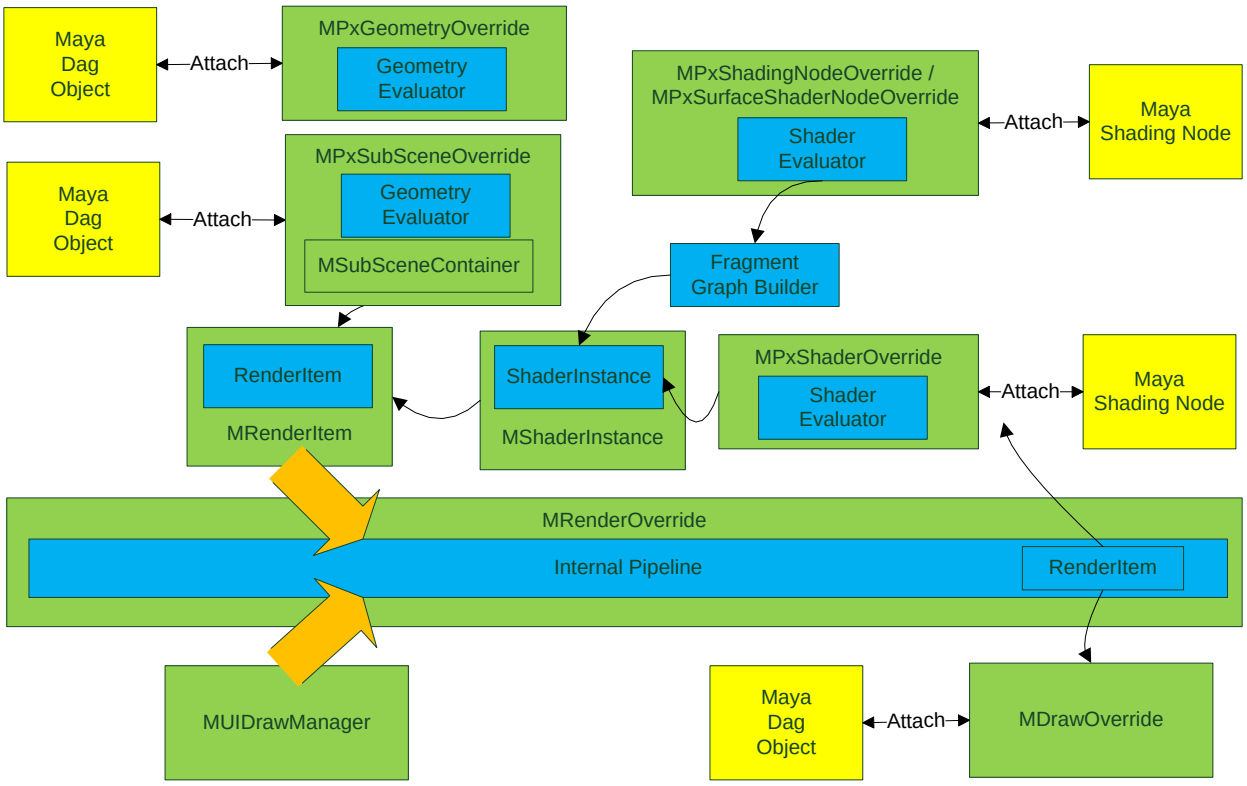


Figure 9: Main API entry points. Filled blue items are internal constructs. Green items are plug-in interfaces. Yellow items are Maya nodes.

Geometry evaluators are listed on the upper-left and shader evaluators on the upper right. The central focus of all object and node based overrides (except MDrawOverride) is to allow MRenderItems to flow down the pipeline. MDrawOverride acts as a custom render item draw

callback. The figure also shows that MPxShaderOverride can still act like a draw callback. Both of these interfaces support VP1 behaviour as “draw time” constructs. MUIDrawManager lives outside the geometry / shader evaluator update block, as it produces transient “drawables” (render items) independently.

5. Choosing VP2 API Interfaces

5.1 Porting Categories

The general categories of items that need to be ported are broken down below.

- Manipulator: Transient drawing for manipulator nodes or container.
- Tool Context: Transient drawing for tool contexts.
- Scene Object: There are a number of options available for the drawing of objects, as VP2 overrides are not directly tied to a particular Maya DAG object type. The following categories exist, each ideal for a different level of *drawing complexity*.
    - Single Object: A Maya object represents a single object. There are two variations
        - Single Object (simple draw): Ideal for drawing a small amount of data per object and per scene, such as a locator (MPxLocatorNode).
        - Single Object (complex draw): Ideal for drawing more complex data, such as a data mesh.
    - Scene Per Object: A special case of a single Maya object actually representing an entire “scene” (but without corresponding Maya nodes).
- Shaders: Shaders can also be implemented in a variety of ways, not all of which need to be node based. For each of the following options, the degree of complexity and control may result in a different option being used:
    - Custom Shading Effect: Shading which takes over all aspects of shading and lighting.
    - Custom Fragment Effect: Shading fragment to be used as part of shade tree evaluation.
    - Shader Node: Custom shading node.
- Render Graph: Custom render loop or custom renderer integration.

5.2 Level of Integration

The choice of interface can be further complicated if there is a required “level of integration”. Possible levels include:

- Device Level:
    - *This level is not recommended in general for new implementations***.**
    - Valid cases exist that require the use of this level of integration. For example, the requirement that “raw” draw API / device level code be maintained.
    - Draw / selection are not integrated at all with any internal systems.
    - Fixed function setup is no longer performed.
    - Raw drawing is done by the plug-in and OpenGL state must be protected as in VP1.

    - Any kind of internal effects / pass system integration is up to the plug-in.
- Resource Level:
    - *This level is not recommended in general for new implementations***.**
    - This is basically a variant of device level integration, except that VP2 managers are used for resources or state handling.
- Render Item Level: Create / use render items along with geometry evaluators and shader evaluators.
- Renderer Level: Device level drawing and/or externally defined drawing can be combined with internal render operations within the same plug-in via user operations.

5.3 Porting Choices

The following table shows the available interface options for various categories versus integration levels.

Even though all options available are shown, items in **bold** indicate suggested options. As noted above, device or resource level integrations are not recommended in general.

Category vs Level	Device Level	Resource Level	Render Item Level	Renderer Level
Single Object	MPxDrawOverride	MPxDrawOverride	**MPxGeometryOverride** or **MPxSubSceneOverride**	MRenderOverride
Scene per Object	MPxDrawOverride	MPxDrawOverride	**MPxSubSceneOverride**	**MPxDrawOverride, MRenderOverride**
Manipulator	N/A	N/A	**MPxManipuatorNode, MPxManipContainer**	N/A
Tool Context	N/A	N/A	**MPxContext**	N/A
Custom Shading Effect	N/A	N/A	**MShaderInstance**	**MShaderInstance**
Custom Shading Fragment	N/A	N/A	**MFragmentManager**	**MFragmentManager**
Shading Node	MPxShaderOverride	MPxShaderOverride	**Attribute name matching.** **MPxShadingNodeOverride** MPxShaderOverride	N/A
Render Graph	MUserOperation	MUserOperation	N/A	**MRenderOverride**

5.3.1 Manipulators and Tool Contexts

Support for VP2 can be accomplished by using the VP2 API equivalents to existing VP1 interfaces.

5.3.2 "Single Objects"

MPxGeometryOverride should be used for drawing a single object. This can be a combination of MRenderItems for persistent data and MUIDrawManager for light-weight transient data. Note

that shaders assigned to a given object are automatically provided with render items, and thus additional items for UI drawing should be all that is required.

For a single complex system, such as a particle system, an MPxSubSceneOverride is preferable. A special classification for volume objects exists: “drawdb/volume”.

MPxSubSceneOverride also allows for the persistence of drawables created via the MUIDrawManager interface. As such, this can greatly decrease overhead incurred via other interfaces, and has better performance scalability.

Although not recommended, it is still possible to use raw calls by using either an MPxDrawOverride or a MUserOperation in an MRenderOverride. However, the latter case can be very wasteful.

5.3.3 “Per-scene” Objects

An **MPxSubSceneOverride** can be used for drawing per-scene objects. One basic caveat is that the plug-in is also responsible for creating render items for any assigned shader nodes. This extends to the management of UI drawables, although the sole logic required is the decision as to whether the drawables are persistent between frame updates.

An MPxSubSceneOverride provides the greatest flexibility, but management is up to the plug-in. It is possible to use an MPxDrawOverride or MUserOperation in an MRenderOverride, wherein the object is simply an attachment point to an external scene renderer.

5.3.4 Custom Shading Effect

MShaderInstances are the simplest level from which to start using the internal shading system with render items. Shader instances can be accessed / created via the shader manager (MShaderManager).

“Stock shaders” provide basic functionality for shaded or flat drawing, as well as basic stipple patterns and numeric drawing. These shaders are made up of fragments (procedures). The equivalent or different custom fragments can be created via the fragment manager interface. [See Custom Shading Fragment.]

Effects on disk can also be loaded via the shader manager (MShaderManager) and will use the internal effects parser. Using this interface is generally the easiest way to reuse pre-existing effects.

Shader instances can be assigned to custom render items (MRenderItems), or used as overrides for operations within a render override (MSceneRender, MQuadRender, MUserOperation); for example, to perform a custom scene pass with a shader override to render geometric normals.

5.3.5 Custom Shading Fragment

Shader fragments (procedures) can be loaded via the fragment manager (MFragmentManager) and used to create an **MShaderInstance**, or used with an **MPxShadingNodeOverride** to return fragments for a given custom shading node.

5.3.6 Shading Nodes

The choice of interface is dependent on the required amount of control for shading and the level of integration desired. The first option, which is the most familiar to those experienced with VP1 implementations, is hardware shader nodes. The next two options are, however, much more integrated and do not require additional integration code beyond shader definition / parameter mapping. The last option requires no or very minimal code changes at most.

5.3.6.1 MPxShaderOverride

This interface allows for the greatest flexibility but can also require the most work. The complexity can be equivalent to the backends for VP1 shader node overrides. This is due to the fact that they are full effects based and take over all aspects of drawing (including lighting, for instance).

Existing VP1 node level interfaces can be used for attribute handling, along with MPxShaderOverride for VP2 support.

If custom effect handling is not required, it is recommended that internally provided MShaderInstances be used instead. If the shader instance is fragment based, then a greater level of pipeline (lighting) and API integration is inherently provided.

If no custom drawing is required, then the internal drawing routines can also reduce the complexity of the code.

Explicit node based shader assignment to arbitrary render items is possible. One general use case is that of using hardware shaders with subscene override render items. It should be noted that dependencies on the lifetime of a node must be explicitly handled by the plug-in. This differs from implicit assignment, which occurs via regular DG connections to DAG objects, and where lifetime management is handled by the renderer.

5.3.6.2 MPxShadingNodeOverride

The most integrated approach is to use MPxShadingNodeOverride to define a fragment and indicate how the input parameters of that fragment are related to the attributes on the Maya node.

This interface is called as part of fragment graph building when interpreting shading networks, and should be familiar to those that write shader fragments for software rendering.

Surface shader building is more complicated and requires understanding of how lighting is bound (MPxSurfaceShadingNodeOverride).

5.3.6.3 Attribute Naming

The workflow that requires the least amount of work is to create a node with attributes that map to the shading parameters of a fixed internal shader. The “root” fragment that is used is the *mayaPhongSurface* fragment.

There is no additional coding required beyond getting the appropriate names on the attributes. For example, to get color support, simply having an attribute with the name “color” is sufficient.

The VP1 interface that mostly close matches this workflow is *MPxMaterialInformation,* but it is much more complicated and requires explicit coding.

5.3.7 Render Graphs

An **MRenderOverride** should be used to replace previous implementations of MPx3dModelViews and MViewportRenderers. It is still possible to keep the MPx3dModelView multi-pass structure, but only one pass is executed for VP2. In that one pass, any MRenderOverride specified is used.

5.3.8 2D Elements

2D HUDs can be drawn using an MRenderOverride. If this is too intrusive, then a locator that uses MUIDrawManager's 2D interfaces can also be used. Per object 2D HUDS can take advantage of the "In View Editor" interfaces available for manipulators. 2D camera based image planes are natively supported by internal evaluators. [See the "Drawing 2D Elements" section in the *Viewport 2.0 API Porting Guide, Part 2: Porting Details* for more information].

5.4 VP2 Selection Considerations

In order to use VP2 selection (introduced in Maya 2016), code that uses interfaces that allow for "raw" draw calls for selection must be ported to use the Viewport 2.0 API.

MUIDrawManager and **MRenderItem** interfaces are recommended.

The mappings to take into consideration are:

a) For manipulators (MPxManipulatorNode):
   a.1. VP1 raw OpenGL "handle" interfaces can be replaced with logical UI draw manager identifiers on MUIDrawManager::beginDrawable ().
b) For simple drawing found in locators (MPxLocatorNode) and surface shapes: (MPxSurfaceShape / MPxComponentShape):
   b.1. MUIDrawManager interfaces should be used for selection.
c) For complex drawing on MPxSurfaceShape/MPxComponentShape:
   c.1. MPxSurfaceShapeUI interfaces should be replaced by render item based interfaces on MPxGeometryOverride and MPxSubsceneOverride:
   c.2. Object level render items can be specified as being selectable.
   c.3. The same can be done for component level render items. Additional work is required to identify component level selection items and support mapping from the indexing used for geometry buffers on render items to shape level components (MFnComponent).
d) For plug-in shaders, it is possible to use MPxShaderOverrides to support custom drawing for selection; for example, to draw wireframe and components for hardware tessellation performed by the shader.

Tool contexts are not affected as they have no selection functionality.

If raw draw calls are being used, and a legacy OpenGL profile is not available, then OpenGL software pick buffers will not work.

5.5 Performance Considerations

MPxDrawOverrides that use raw calls should beware of stalling the graphics pipeline. MFrameContext and MDrawContext can be used to avoid pulling the graphics state directly from the device and hence causing stalls. To avoid any context state bleeding, the VP1 adage of a “pushing” and “popping” state for OpenGL still holds true.

Drawing a large amount of data using MUIDrawManager is not recommended. All data is generally considered to be transient, and hence can result in constant data reallocation. Persistence only occurs via the interfaces on MPxSubSceneOverride. In this case, if the draw is mostly static, then the caching mechanism will reduce the additional cost of deleting and recreating every frame.

Before Maya 2016, use of MPxGeometryOverrides for rendering a very large number of render items was not recommended due to scalability. This is partially the reason for adding the MPxSubSceneOverride class.

If hardware instancing on render items is desired, MPxSubSceneOverride exposes options for transform or shader parameter level instancing which can increase performance. Hardware instancing may not always be available, in which case the internal fallback will be to use software instancing.

Shader “uniqueness”, which includes geometry requirements, returned from a shader, helps determine if consolidation is possible. The more flexible the interface used for the shaders, the greater the chance of differentiating a shader. Fragment based shaders generally do not have this concern. MPxShaderOverrides have the greatest flexibility, including a custom user data setting – which, in general, should be avoided if possible.

Explicit use of state objects is generally not recommended. If a state is overridden, then it should be restored, as internal drawing does not explicitly attempt to reset state for every render item drawn. Specific rendering passes set up particular states, and any explicit state override may result in an incompatible state being used. MPxDrawOverrides and MPxShaderOverrides are generally used for setting states. In these cases, the state should match the internal state for a given pass context.

5.6 Threaded Evaluation / Evaluation Manager Considerations

Usage of MRenderItems is recommended via MPxGeometryOverride or MPxSubSceneOverride. The code should adhere to the interfaces that are designated for DG evaluation, and GPU device access. Any custom / user data should not reference items which may be re-evaluated (such as a DG node).

MPxDrawOverride can still be used for items such as scene-level objects, assuming that the plug-in scene’s interface with the evaluation manager is managed properly.

5.7 Device Considerations

The following sections list the concerns associated with device level implementations. These issues should be reviewed to determine if a device level approach is still an acceptable one; for example, using MPxDrawOverride.

5.7.1 OpenGL Core Profile Contexts

If this option is chosen, then plug-ins should beware of the following caveats.

The biggest of these is that device level code written for legacy profile context may no longer execute properly due to support for given calls being deprecated.

For plug-ins that must run on a core profile context on Mac OS X, any usage of “raw” draw calls is discouraged, as this can result in legacy draw or selection no longer being available.

5.7.2 Device Tips

VP2 no longer sets up the fixed function state for a number of items, including lighting. OpenGL “glGet*” methods are not recommended for extracting data. In general, these calls cause pipeline stalls. Push and pop state must be used if VP2 interfaces are not used for a given state.

Access to M3dView is no longer recommended, and information should be read from MFrameContext or MDrawContext instead. The MGeometryUtilities class is also available for accessing information such as Maya display colors, and display status.

The 2d coordinate convention for VP2 is DirectX or Windows based. Therefore, buffer coordinates start from the upper left instead of lower right (flipped in Y), and texture lookups are also flipped in Y.

The 3d coordinate system is left handed.

Clip space in OpenGL ranges from -1 to +1, whereas it ranges from 0 to 1 for Direct3D.

For OpenGL, FBO overrides cannot be used. If used for local drawing, they must be cached and restored.

notice, and is not responsible for typographical or graphical errors that may appear in this document.