

Viewport 2.0 API

Introduction

Maya 2012 introduces a new set of API classes for defining custom drawing, shading and effects in both Viewport 2.0 and Hardware Renderer 2.0 (the new batch/render view version of Viewport 2.0).

Maya's Viewport 2.0 represents a brand-new rendering architecture in Maya that has been written from the ground up to deliver high-performance on large scenes on top of a programmable shader system offering high-quality per-pixel lighting and effects. This, unfortunately, makes it incompatible with many of the existing Maya API classes, due to their dependence on system memory geometry data, C++ driven rendering, and high state switching overheads. As a result, most existing plugins will need to be updated to support the new API classes in order to work correctly in Viewport 2.0.

This document introduces the new classes and highlights the migration paths from the old API to the new one. In most cases, it should be possible to write plugin nodes that simultaneously implement both the new and old APIs where required, thus maintaining a single plugin that will work in both viewports while artists, tools and pipelines are migrated to Viewport 2.0.

This discussion of the new API is broken down into the following sections:

- Data Classes - the objects which describe and provide access to the geometry and shaders in the scene
- Scene Overrides - the interfaces used to define custom drawing, shading and effects within the scene
- Rendering Overrides - the objects used to define custom rendering, multi-pass effects, and filtering
- Framework Classes - the objects that provide services and device state information to plugins

Following this, there are two sections to help accelerate your transition to the new API:

- Transition Guide - tips on moving from legacy interfaces to the new Viewport 2.0 API
- Devkit Samples - a brief overview of updated devkit samples using the new Viewport 2.0 API

All of the new classes reside in a new namespace within the Maya API called MHWRender (all references to class names in this document assume the namespace). This makes it easy to locate all the classes related to viewport and batch hardware rendering and also makes it easy to differentiate them from the older hardware API classes which are not supported in Viewport 2.0 and Hardware Renderer 2.0.

1. Data Classes

These objects describe the renderable geometry in the scene, and present a thin (but obvious) abstraction over the underlying GPU resources they represent. They're used by shapes to describe the renderable geometry (this applies to both Maya's internal shapes and plugin shapes). And they're used by shaders to bind and render the geometry (again, this applies to both Maya's internal shaders and

plugin shaders). This clean separation means that custom shaders can work with Maya shapes, Maya shaders can work with custom shapes, and custom shaders and custom shapes can also work together.

The basic "handshake" between shapes and shaders is as follows:

1. The shape is queried for the list of render items it needs to render (including multiple material sub-geometries, wireframe selection, component display, etc).
2. The shader is queried for its geometry requirements (e.g. "I need positions and UV set foo").
3. Maya works out the super-set of all the geometry requirements on each shape (based on all the render items that use it).
4. The shape populates all the geometry buffers based on its current state.
5. The shader receives the list of render items it needs to render, and it can then pull out the geometry buffers it needs for each geometry item.

However, unlike previous Maya viewports, Viewport 2.0 tries to cache and re-use as much data and information as possible - so unless the scene state is changing, only the final render call will be used to render the cached state.

MGeometry

MGeometry represents the vertex and index data for any renderable geometric entity in Maya. These are most commonly shapes (meshes, NURBS surfaces, subdivision surfaces), but it may also be used to represent any other in-viewport elements (including the grid, manipulators, tool feedback, etc).

In the case of DAG objects, each MGeometry instance holds all the renderable data for all instances of a single object. This includes all of the vertex and index buffer data that describes the shape's control points.

MGeometry is used in several places in the Viewport 2.0 API. In some cases it is simply meant to provide read-only access to existing geometry data, and in others the user is required to fill the MGeometry object with vertex and index data needed to draw a particular object.

MVertexBuffer

MVertexBuffer is a thin wrapper around a graphics card vertex buffer. Each vertex buffer has a name, a semantic (position, normal, uv, etc), and a type (e.g. float3) and this data is encapsulated by an instance of the MVertexBufferDescriptor class. Other than hardware memory and shader interpolant limits, there are no Maya limits to the number of vertex buffers that can be added to an MGeometry object (e.g. an object can have multiple position streams, multiple normal streams, etc).

MIndexBuffer

MIndexBuffer is the index equivalent of MVertexBuffer. An MGeometry object can include 0, 1, or multiple index buffers depending on how many renderable objects (render items, see below) the MGeometry object represents.

MGeometryRequirements

MGeometryRequirements describes the geometry streams and index buffers that are required to draw all render items associated with a specific object. This class is passed to implementations of MPxGeometryOverride to indicate what geometry needs to be produced.

The list of vertex buffers stored on this object is formed by taking the union of the requirements from all of the render items stored on the object. Any one render item can be examined to determine the requirements for that item; however, all data is shared where possible.

MRenderItem

MRenderItem describes a renderable primitive, something that is actually rendered on the graphics card. Each MRenderItem includes:

- The render primitive to use (e.g. indexed triangles, points, lines, triangle strips, etc.)
- An index buffer to use when rendering that primitive
- A list of which vertex buffers to render
- The shader to use (may be overridden in some cases)
- A set of display mode/options to control which render passes and viewports this render item is visible in (e.g. so one viewport can display wireframe while another displays shaded mode)

The render item is deliberately a light weight object to make it easy and efficient to share heavy vertex/index data across multiple renders of the same underlying geometry (e.g. to draw wireframe over shaded mode re-using the same position data).

Some render passes (e.g. shadow pass, depth pass) will use the display mode/option to select which render items to display, but will override the material in order to render other surface properties.

The shader associated with a render item is what drives the geometry requirements that need to be fulfilled to draw the object. These requirements can be retrieved from MRenderItem and will be non-empty if a shader is assigned. Shaders can be acquired through MShaderManager.

Render items may be enabled or disabled to allow or prevent them to draw without needing to delete the render item and recreate it later.

Render items are created in two places. First, Maya automatically creates one render item for each shader assignment to each instance of each object. These render items may be disabled but not removed and are automatically associated with the shader described by the Maya shading assignment. Second, users may add additional render items per instance for any custom purpose.

MVertexBufferDescriptor

This class is used to describe the properties of either an existing vertex buffer or one that needs to be created (i.e. to specify geometry requirements).

MVertexBufferDescriptorList

A simple list of `MVertexBufferDescriptor` objects.

MRenderItemList

A simple list of `MRenderItem` objects. Each item is owned by the list.

2. Scene Overrides

The scene override classes allow plugins to override elements of the 3d scene graph, including geometry, shaders, and other renderable elements. These classes provide new APIs to replace the old rendering code in `MPxLocator`, `MPxSurfaceShape`, `MPxComponentShape`, `MPxHwShader`, and `MPxHardwareShader`. User plugins can continue to derive off these existing base classes; however, you'll need to register one of the following overrides to define your custom rendering logic in the new viewport. This also means you can leave your existing rendering code in place, allowing your plugin to work simultaneously in both the old and new viewports as you transition your pipeline across. The old viewport will continue to call into the old code, and the new viewport will use the overrides defined below.

While many of the old interfaces provided raw C++ interfaces at render time, this approach does not provide the best performance, particularly when you have large numbers of plugin elements in the scene. The new overrides use the geometry data objects above to support a cached, retained mode rendering interface that more clearly separates shaders and shapes, and also allows them both to benefit from many of the performance optimizations in the new viewport (such as geometry consolidation). Having said this, there are still times where you'll want total control over what is rendered, so we still provide an immediate mode interface to succeed the interfaces provided in `MPxLocator` and `MPxSurfaceShapeUI`.

MPxDrawOverride

`MPxDrawOverride` formalizes the draw override functionality provided by the old `MPxLocator` class. It provides a simple interface that allows users to make arbitrary OpenGL calls whenever a specific shape type needs to be drawn by Maya. When using this interface, your plugin assumes full control over all the geometry and material resources and setup required to draw. This also means your plugin will not benefit from the resource management and resource optimizations provided by the new viewport. However for light-UI or large proxy geometry, this can still be a good approach.

Implementations of `MPxDrawOverride` must be registered with the `MDrawRegistry` class against a specific type of Maya DAG object (either plugin or standard). Registration is by classification string. This class can be used to provide drawing for plugin shapes or to completely override the draw of standard shapes like meshes or NURBS surfaces. For a DAG object type to be recognized by Viewport 2.0, it must be registered with a classification string that begins with "drawdb/geometry/" and this is the same classification string that is used to register the implementation of `MPxDrawOverride`. For example, the standard Maya mesh type is registered as "drawdb/geometry/mesh" and so to override

the standard draw with an implementation of `MPxDrawOverride` one would register that implementation with the classification string "drawdb/geometry/mesh".

This class provides very low-level access to the draw. It is necessary to manually set up all shaders and do proper state management in order to get correct drawing and to avoid corrupting the draw of other objects. A new utility class called `MDrawContext` has been added to assist with state management. An instance of this object is always passed into the draw call of `MPxDrawOverride`.

For a higher level interface to geometry, see the new `MPxGeometryOverride` class.

MPxGeometryOverride

This class represents one of the larger changes from the old API. Where `MPxSurfaceShapeUI` gave you C++ draw control, `MPxGeometryOverride` is purely an interface for defining geometry and render items. All control of the actual rendering is left to the shader. By doing this, this class can work with external shaders and it can leverage performance optimization such as geometry consolidation.

`MPxGeometryOverride` is a high-level interface that allows the user to provide geometry buffers, index buffers and custom render items that will be used by the Viewport 2.0 system to draw a specific DAG object. Data is primarily transferred through the `MGeometry` class described above.

Implementations of `MPxGeometryOverride` must be registered against the DAG object type just like `MPxDrawOverride`.

An implementation of `MPxGeometryOverride` is invoked when an associated DAG object changes. The implementation is expected to provide updated geometry buffers and indexing information which will be packed by Viewport 2.0 and stored on the graphics card (if possible). `MPxGeometryOverride` is only invoked when something needs updating and not on every frame.

Similar to `MPxDrawOverride`, `MPxGeometryOverride` can be used to provide geometry for either plugin shapes or to override the behaviour of standard Maya shapes. The biggest advantage to using this higher level class is that objects which use it will automatically work with any shader supported by Viewport 2.0. This class also requires no direct OpenGL knowledge as it is device independent.

MPxShaderOverride

`MPxShaderOverride` allows the user to create a custom override for associating a "full shading effect" with a shading node (custom or standard) in Maya. Its primary use is for associating hardware effects with pre-existing plugin shaders.

A "full shading effect" defines the complete shading and lighting involved to render a given object. Input resources for shading such as geometry, textures, and lights are defined and bound to the shading effect via the override as required. The override is fully responsible for these tasks. As an example, for hardware shading, this can be thought of as implementing a CgFx or HLSL effect file renderer which can use the resources defined within a Maya scene.

Like `MPxDrawOverride`, this is a low-level class. Any object using an instance of the associated shader to draw will trigger the `MPxShaderOverride` draw callback at draw time. The implementation is responsible for setting up all shading information. It can then either query the geometry itself through the `MDrawContext` to do all binding and drawing; or, it can make a call back into Maya to allow Viewport 2.0 to handle the draw using the current state.

Like the other two new interfaces, `MPxShaderOverride` must be registered with `MDrawRegistry` using a classification string. This means it can be associated with either a plugin shader type or with an existing Maya shader type. Viewport 2.0 shader classification strings must begin with “drawdb/shader/” (for example, the standard lambert is “drawdb/shader/surface/lambert”).

3. Rendering Overrides

This section describes the new rendering interface. The term pass is not used on purpose to avoid confusion with the existing Mental Ray based “render passes” system. This is not a wrapper for that system. This is instead intended to subsume the existing hardware “multi-pass” interface which is exposed in `MPx3dModelView`.

Main Features of the New Interface

- To allow for control of the render loop at the start of a refresh or batch render.
- To be able to have explicit rendering “operations” (e.g. a scene render) and to be able to use an arbitrary number of such render operations.
- To be able to have explicit “render targets” (e.g. color, depth stencil buffers).
- To allow for a sense of “context” as to where the rendering will occur (e.g. which viewport).
- To allow for render operation “overrides” similar to what is currently available, and allow for future override enhancements.
- To have a logical separation of drawing the scene versus the UI elements.
- To have the ability to access hardware resources.

Key Differences in the New Interface

- To override or extend how rendering is performed, the level of interjection need not be at the view or panel level. That is, a new API view and API panel are not required. The amount of overhead to introduce an override has been greatly reduced.
- There is no longer any fixed looping logic where N iterations of a refresh with pre and post callbacks are required.
- Instead of intercepting callbacks from the internal render loop logic, the logic now allows the API writer to queue a set of rendering operations.
- Render target or output buffers need no longer be externally defined and are presented as formal rendering resources.
- The mechanism has been standardized to work both for interactive as well as batch rendering.
- A render logic description is formally defined by the user for the renderer to execute. This can be thought of as “retained” versus “immediate” mode execution.

Key Concepts and Constructs

- **Render Override:** There is a formal notion of a rendering override. An override will override all rendering per refresh for interactive rendering or per frame output for batch rendering. An override is composed of a set of rendering operations. The key class for this is `MRenderOverride`.
- **Rendering Operation:** An operation can be loosely thought of as being equivalent to a “pass” in the `MPx3dModelView` pass based system, except that they are explicitly defined. The key class for this is `MRenderOperation`. There can be predefined operations as well as custom user operations. The basic operation set includes:
 - A background clear operation (`MClearOperation`)
 - A 3D scene render operation (`MSceneRender`)
 - A 2D HUD (heads up display operation) (`MHUDRender`)
 - A user defined operation (`MUserRenderOperation`)
 - A quad blit operation (`MQuadRender`) which renders a 2D screen space quad.
 - A target presentation operation (`MPresentTarget`)
- **Render Target Overrides:** Render targets are now formalized. This is so that there is a managed and recognized set of resources that both the renderer and the plugin writer can use. Being formalized also means that the resource can be defined in a draw API agnostic way (e.g. for OpenGL and DirectX). The interface presented is a render target description, `MRenderTargetDescription`. The descriptions are used to describe desired render target inputs or outputs for rendering operations.
 - An access to the device level resource is provided so that users can write their own custom code which accesses targets.
 - It is still possible to use user defined render targets but device state must be restored or the renderer’s render target state will be corrupted. This is not a new restriction.
- **Shader Overrides:** The `MShaderManager` interface allows access to some simple stock shaders as well as access to file based effects. These shaders can then be applied as overrides in one of two places:
 - As per object shader overrides for 3d scene renders.
 - As per quad render shader for quad blits.
- **State Overrides:** State setting has been formalized and is managed by the renderer. State setting can be performed per render operation.
- **General Scene and User Operation Overrides:** Basic overrides formally presented include:
 - Camera specification
 - World or scene filtering
 - Object type filtering
 - Draw mode filtering (e.g. filled, wire, component)
 - Access to `M3dView` for interactive rendering which will allow the user to use other existing overrides.
 - For batch rendering, the user also has access to hardware and common render globals.

- As custom user operations are available, users are still free to perform their own direct drawing with the restriction that they restore the previous device level state. The same restriction currently applies for all existing hardware rendering interfaces, including `MPx3dModelView` overrides.

Render Logic Building

The main steps for defining a render override are:

- Derive a custom render override.
- Register the override with the renderer. Any number of overrides can be registered.
- Set the active override for a viewport or for batch rendering.
- Set up a set of render operations and overrides as an ordered list. If the render logic is a graph, then this can be broken down into a series of render operations.
- Provide the list of operations when queried and update parameters per operation as required.
- Custom operations and custom overrides have the ability to intercept intermediate and final render targets and may do so to use the resources as required (e.g. to save to disk, or to route to custom rendering code).

Key Restrictions

- Scene modifications may not occur during rendering. This is not a new restriction.
- There is no formal mechanism as yet to provide feedback (looping) override outputs back into specific places in a scene render as those interfaces do yet exist. For example looping shadow maps back to custom lights as a custom light interface does not currently exist.

4. Framework Classes

The framework classes give plugins access to Maya's internal rendering framework objects, including state and resource management, rendering context and override management.

`MDrawRegistry`

This is where implementations of proxy override classes must be registered against a specific node type in Maya. The same proxy may be registered against multiple node types, but each node type may only have one associated proxy class. Additional registrations override existing registrations.

`MDrawContext`

`MDrawContext` stores all draw context information for use by classes that override the actual draw (`MPxDrawOverride` and `MPxShaderOverride`). This class can be used to get and set draw state.

`MStateManager`

The state manager can be retrieved from the draw context instance in order to access and/or modify GPU state in an efficient and controlled manner. Although it is, of course, possible for users to modify state directly, it is recommended that all state management go through this interface. Not only does

Maya attempt to optimize access to device state in order to avoid expensive read-backs and redundant state changes, using this interface also helps Maya keep track of the current state to avoid corruption.

MRenderer

This is the main interface for the Viewport 2.0 renderer. It presents interfaces for resource management as well render logic control. The renderer is a singleton object which may be accessed through `MRenderer::theRenderer()`.

Texture Management

Textures are resources that can be acquired and released from the renderer. The main interfaces allow for acquiring a file texture from disk or creating a texture from a block of system memory. All textures are managed by the renderer. A texture is described by `MTextureDescription` and the list of available raster formats is described by `MRasterFormat`. The supported texture types are described by `MTextureType` and the environment mapping types by `MEnvironmentMapType`. It is also possible to access the device level handle of a texture to allow users to directly use the resource.

Shader Management

Some simple stock shaders, as well as CgFX shaders loaded stored in files on disk, can be acquired from the renderer through the shader manager. Access the `MShaderManager` class from the renderer to get access to `MShaderInstance` objects which can be used by `MPxGeometryOverride` or as a complete shader override on a render.

Target Management

`MRenderTargetManager` can be used to acquire render targets from the renderer for use as overrides on render operations.

5. Transition Guide

MPxLocator

Support for Maya nodes implementing the `MPxLocator` interface can be added by providing an implementation of `MPxDrawOverride` and registering it with `MDrawRegistry` against a classification string for the Maya plugin node type. See the devkit example “footPrintNode” for more details.

MPxSurfaceShape / MPxComponentShape

Support for Maya nodes implementing these interfaces can be added by providing an implementation of `MPxGeometryOverride` and registering it with `MDrawRegistry` against a classification string for the Maya plugin node type. See the devkit example “apiMeshShape” for more details.

MPxHwShaderNode / MPxHardwareShader

Support for Maya nodes implementing these interfaces can be added by providing an implementation of `MPxShaderOverride` and registering it with `MDrawRegistry` against a classification string for the Maya plugin node type. See the devkit example “hwPhongShader” for more details.

M3dView/MPx3dModelView

Custom renderer definitions for Viewport 2.0 can be provided using the new rendering override interfaces `MRenderOverride` and `MRenderOperation`. See the devkit example “viewRenderOverride” for more details.

6. Devkit Samples

footPrintNode

The `footPrintNode` plugin is an example of using `MPxLocator`. It has been updated to provide a simple Viewport 2.0 implementation as well as an example of using `MPxDrawOverride`. Not all draw modes are supported, but it shows how `MPxDrawOverride` is meant to be used.

apiMeshShape

The `apiMeshShape` plugin is an example of how to implement a custom Maya shape using `MPxSurfaceShape`. It has been updated to include how to use `MPxGeometryOverride` in order to provide vertex data, index data, and custom render items to Viewport 2.0 for drawing a custom Maya shape with arbitrary shader assignments.

hwPhongShader

`hwPhongShader` is an example of how to implement a custom hardware shader using `MPxHwShaderNode`. It has been updated to include how to use `MPxShaderOverride` to implement a complete custom shading effect for Viewport 2.0. In its primary mode, it uses the simple `drawGeometry` interface to allow Viewport 2.0 to handle geometry drawing after it sets up the shader. However, it also has an alternate implementation that shows how to query the actual geometry resource handles through the `MGeometry` interface in order to do custom binding and drawing.

cgFx

Maya’s `CgFX` plugin has been updated to support Viewport 2.0 through the `MPxShaderOverride` interface. Unlike `hwPhongShader` above, this is a non-trivial example of using the new API to produce fast and interesting custom shading. The plugin makes extensive use of `MDrawContext` in order to set and maintain state as the plugin renders the custom `CgFX` file shaders.

viewRenderOverride

The `viewRenderOverride` plugin is an example of how to override the viewport drawing to produce various effects on top of the standard draw. This makes use of many new API classes including: `MRenderer`, `MRenderOverride`, `MRenderOperation` (and several of its derived classes), `MCameraOverride` and `MRenderTarget`.