Autodesk, Inc.

Using Parallel Maya

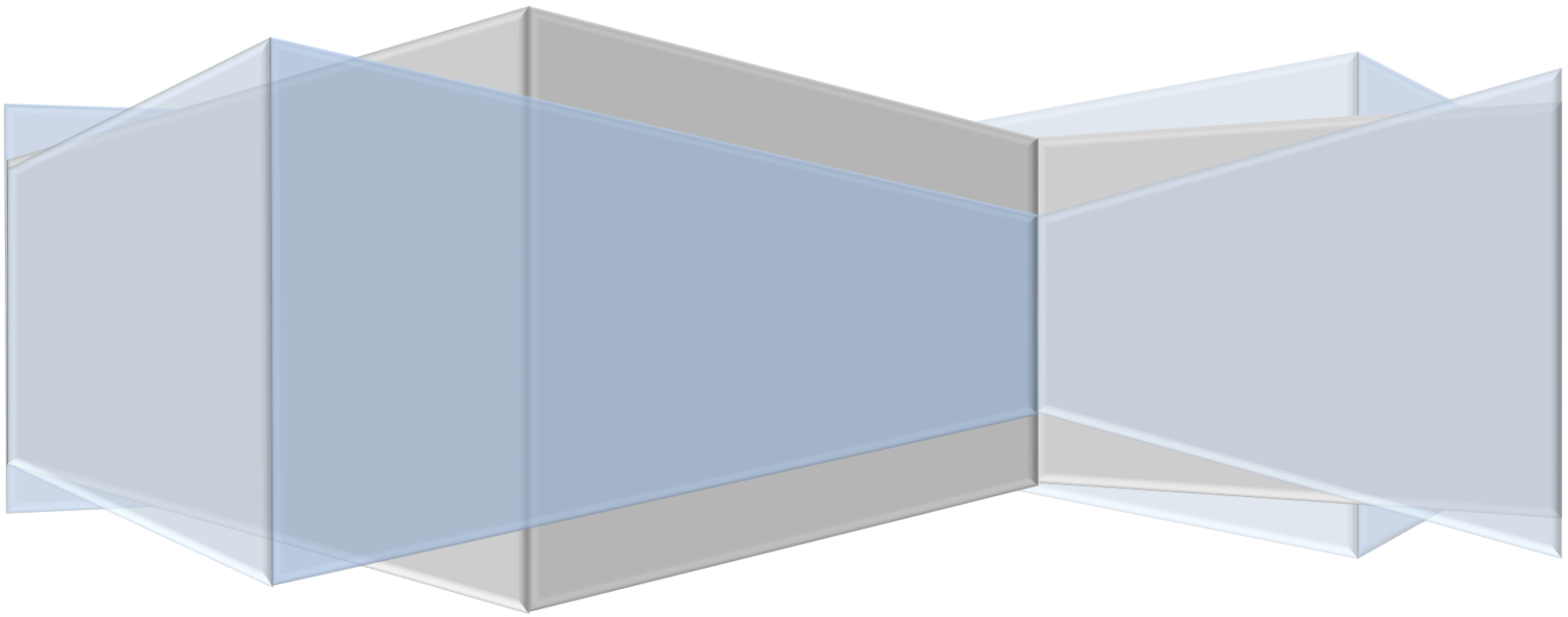


Table of Contents

Overview

This guide describes the new Maya 2016 features for accelerating playback and manipulation of animated scenes. It covers key concepts, shares best practices/usage tips, and lists the known limitations that we'll aim to address in subsequent versions of Maya.

This guide will be of interest to riggers, TDs, and plug-in authors wishing to take advantage of speed enhancements in Maya 2016.

If you'd like an overview of related topics, prior to reading this document, check out the **Supercharged Animation Performance in Maya 2016** video at: https://www.youtube.com/watch?v=KKC7A9bbUuk.

Key Concepts

Maya 2016 accelerates existing scenes by taking better advantage of your hardware. Unlike previous versions of Maya, which were limited to parallelizing individual nodes, Maya 2016 includes a mechanism for scene-level graph analysis and parallelization. For example, if your scene contains different characters that aren't constrained to one another, Maya recognizes this and evaluates each character at the same time.

Similarly, if your scene has a single complex character, it may be possible to evaluate sub-sections of the rig simultaneously. As you can imagine, the amount of parallelism depends on how your scene has been constructed. We'll get back to this later. For now, let's focus on understanding key Maya 2016 evaluation concepts.

At the heart of Maya 2016's new evaluation architecture is an **Evaluation Manager (EM)** responsible for creating a *parallel-friendly* description of your scene, called the **Evaluation Graph (EG)**. The EM schedules EG nodes across available compute resources.

Prior to evaluating your scene, the EM checks if a valid EG graph exists. The EG is a simplification of the **Dependency Graph (DG)**, consisting of DG nodes and connections. Connections in the EG state that a destination node requires values from source node(s) to correctly evaluate its state. A valid EG may not exist for various reasons. For example, you may have loaded a new scene and no EG may have been built yet, or you may have changed your scene, invalidating a prior EG.

Maya 2016 uses the DG's **dirty propagation** mechanism to build the EG. Dirty propagation is the process of walking through the DG, from animation curves to renderable objects, and marking the attributes on DG nodes as needing to be re-evaluated (i.e., dirty). Unlike previous versions of Maya that propagated dirty on every frame, Maya 2016 disables dirty propagation once the EG is built, and reuses the existing EG until it becomes invalid.

With dirty propagation disabled, computing your scene at a given frame involves walking the EG, scheduling, and evaluating EG nodes. Because the EG encodes node-level dependencies, when evaluating a given EG node, you know that all inputs coming from dependent nodes have already been calculated. This further enables pipelining of some operations. Specifically, when we get to EG nodes

without dependents, we can initiate additional processing (e.g., rendering) since we are guaranteed that no downstream nodes will require computed results.

> **Note:** If your scene contains expression nodes that use the `getAttr` command the DG graph will be missing explicit dependencies, resulting in an incomplete EG. In addition to impacting correctness, expression nodes will also reduce the amount of parallelism in your scenes (see Scheduling Types for details).

Depending on how you've built your scene, the EG may contain circular node-level dependencies. If this is the case, the EM creates node **clusters.** At scene evaluation time, nodes in clusters are evaluated serially, before continuing with other parallel parts of the EG. Multiple clusters may be evaluated at the same time. As with previous versions of Maya, you should avoid building scenes with attribute-level cycles as this is unsupported, and leads to unspecified behavior.

By default, the EM schedules node evaluation on available CPU resources. However the EM also provides the ability to override evaluation for sub-sections of the EG, targeting computation to specific runtimes and/or hardware. One example of this is **GPU override** feature included in Maya 2016, which uses your graphics card's graphics processing unit (GPU) to accelerate deformations.

When manipulating your rig, you may notice that performance improves once you've added at least 2 keys on a controller. By default, only animated nodes are included in the EG. This helps keep the EG compact, making it fast to build, schedule, and evaluate. Hence, if you're manipulating a controller that hasn't yet been keyed, Maya relies on legacy DG evaluation. When 2 or more keys are added, the EG is rebuilt to include the newly keyed nodes, permitting parallel evaluation via the EM.

Supported Evaluation Modes

When you start Maya 2016, you'll automatically be in Parallel evaluation mode. This is the new default evaluation mode, replacing the legacy DG-based evaluation. Maya 2016 supports 3 evaluation modes:

Mode Type	Purpose
DG	Uses the legacy **Dependency Graph**-based evaluation of your scene. This was the default evaluation mode prior to Maya 2016.
Serial	**Evaluation Manager Serial** mode. Uses the EG but limits scheduling to a single core. Serial mode is a troubleshooting mode to pinpoint the source of evaluation errors.
Parallel	**Evaluation Manager Parallel** mode. Uses the EG and schedules evaluation across all available cores. This mode is the new Maya 2016 default.

When using either Serial or Parallel EM modes, you can also activate the **GPU Override**, which accelerates deformations on your GPU. You must be in Viewport 2.0 to use this feature (see Custom Evaluators).

To switch between different modes, go to the Preferences window (**Windows > Settings/Preferences > Preferences > Animation**). You can also use the **evaluationManager** MEL/Python command; see documentation for supported options.

To see the evaluation options that apply to your scene, turn on the Heads Up Display Evaluation options (**Display > Heads Up Display > Evaluation**).

First Make it Right Then Make it Fast

Before focusing on understanding how to make your scene fast in Maya 2016, it's important to ensure evaluation in DG and EM modes generate the same results.

When you start Maya 2016, if you observe evaluation errors (i.e., what you see in the viewport differs from previous versions of Maya), determine the source of these errors. Errors may be due to an incorrect EG, threading related problems, or other issues. In the sections that follow we'll review 2 important concepts related to errors: **Evaluation Graph Correctness** and **Thread Safety**

Evaluation Graph Correctness

In the event that you see evaluation errors, first try to test your scene in **Serial** evaluation mode (see Supported Evaluation Modes). Serial evaluation mode uses the EM to build an EG of your scene, but limits evaluation to a single core. This eliminates threading as the possible source of differences. Note that since Serial evaluation mode is provided for debugging, it has not been optimized for speed and scenes may run more slowly in Serial than in DG evaluation mode. This is expected.

If transitioning to Serial evaluation eliminates evaluation errors, this indicates that the errors in your scene are likely due to a threading related problem. However, if errors persist even after transitioning to Serial evaluation this indicates that the EM is building an incorrect EG for your scene. There are a few possible reasons for this:

Custom Plugins. If your scene uses custom plug-ins that rely on the MPxNode::setDependentsDirty function to manage attribute dirtying, this may be the source of problems. Plug-in authors sometimes use setDependentsDirty to avoid expensive calculations every time MPxNode::compute is called. Using this approach results from previous evaluations are typically cached and MPxNode::setDependentsDirty is used to trigger re-computation.

Since the EM relies on dirty propagation to create the EG, any custom logic in plug-ins that alter dependencies may interfere with the construction of a correct EG. Furthermore, since the EM evaluation does not propagate dirty messages any custom caching or computation in MPxNode::setDependentsDirty will not be called while the EM is evaluating.

If you suspect that evaluation errors are related to custom plug-ins, temporarily remove the associated nodes from your scene and validate that the DG and Serial evaluation modes generate the same result. Once you've made sure this is the case, you'll need to revisit the plug-in logic. The API Extensions section covers Maya 2016 SDK changes that will help you adapt plug-ins to parallel evaluation.

Errors in Autodesk Nodes. Although we've done our best to ensure that all out-of-the-box Autodesk Maya nodes correctly express dependencies, it's possible your scene is using nodes in an unexpected manner. If this is the case, we ask you make us aware of scenes where you encounter problems. We'll do our best to address problems as quickly as possible.

Thread Safety

Prior to Maya 2016, evaluation was single-threaded and developers did not need to worry about making their code thread-safe. At every frame, developers were guaranteed that evaluation would proceed serially; computation would finish for one node prior to moving onto another. This approach made it possible to cache intermediate results in global memory and to use external libraries without considering their ability to work correctly when called simultaneously from multiple threads.

These guarantees no longer apply for Parallel Maya. Developers working in Maya 2016 will need to update plug-ins to ensure correct behavior during multi-core evaluation. Although individual calls to nodes don't need to be thread-safe (since we evaluate a node's dirty plugs sequentially from a single thread), plugs values may be read concurrently once evaluated. Hence, nodes will now need to handle those situations. This can be done by ensuring stateless behavior or by caching data on plugs.

Since we're aware that making legacy code thread-safe will require time, we've added new scheduling types that instruct the EM regarding how to schedule nodes. Scheduling types provide a straightforward migration path, so you don't have to pass up opportunities for parallelizing some parts of your scenes, because a few nodes still need work.

There are 4 scheduling types:

Scheduling Type	What are you telling the scheduler?
Parallel	Asserts that the node and that all third-party libraries used by the node are thread-safe. The scheduler may evaluate any instances of this node at the same time as instances of other nodes without restriction.
Serial	Asserts it is safe to run this node with instances of other nodes. However, all nodes with this scheduling type should be executed sequentially within the same evaluation chain.
GloballySerial	Asserts it is safe to run this node with instances of other nodes but only a single instance of this node should be run at a time. This should be used if the node relies on static state, which could lead to unpredictable results if multiple node instances are simultaneously evaluated. The same restriction may apply if third-party libraries store state.
Untrusted	Asserts this node is not thread-safe and that no other nodes should be evaluated while an instance of this node is evaluated. Untrusted nodes are deferred to the end of the evaluation schedule and can introduce costly synchronization.

By default nodes are scheduled as **Serial**, which provides a middle ground between performance and stability/safety. In some cases this is too permissive and nodes must be downgraded to **GloballySerial** or

Untrusted. In other cases, it's possible to promote nodes to **Parallel**. As you can imagine, the more parallelism supported by nodes in your graph, the higher level of concurrency you're likely obtain.

When testing your plug-ins with parallel Maya, a simple strategy is to schedule nodes with the most restrictive scheduling type (i.e., **Untrusted**), validate that the evaluation produces correct results, raise individual nodes to the next scheduling level, and repeat the experiment.

Scheduling behavior can also be altered dynamically at runtime. For example, Maya currently defaults to scheduling expression nodes as untrusted, since it's unclear ahead of time what actions an expression will perform. However, if Maya detects an expression node is limited to arithmetic and outputs are purely a function of inputs, we can safely promote scheduling of that expression to GloballySerial. We cannot schedule expressions as Parallel since the Maya command interpreter is not thread-safe, as it must store state to provide useful logging and error reporting.

There are two ways to alter the scheduling level of your nodes:

Mel/Python Commands. You can use the evaluationManager command to change the scheduling type of nodes at runtime. Below, we illustrate how to change the scheduling of scene transform nodes:

Scheduling Type	Command
Parallel	`evaluationManager -nodeTypeParallel on "transform";`
Serial	`evaluationManager -nodeTypeSerialize on "transform";`
GloballySerial	`evaluationManager -nodeTypeGloballySerialize on "transform";`
Untrusted	`evaluationManager -nodeTypeUntrusted on "transform";`

C++/Python API methods. Individual nodes can also be scheduled at compile time by overriding the MPxNode::schedulingType function. Functions should return one of the enumerated values specified by MPxNode::schedulingType. See the Maya 2016 MPxNode class reference for more details.

Safe Mode

On rare occasions you may notice that during manipulation or playback, Maya changes from Parallel to Serial evaluation. This is due to **Safe Mode,** which attempts to trap errors leading to instabilities, such as crashes. If Maya detects that multiple threads are attempting to simultaneously access a single node instance at the same time, the evaluation is forced to Serial execution to prevent problems.

While Safe Mode catches many problems, it cannot catch them all. Therefore, we've also developed a special **Analysis Mode** that performs a more thorough and costly checks of your scene. Analysis mode is designed for riggers and TDs who wish to troubleshoot evaluation problems when creating new rigs.

If Safe Mode forces your scene into Serial mode, the EM may not produce the incorrect results when manipulating. In such cases you can either disable the EM:

```
evaluationManager -mode "off";
```

or disable EM-accelerated manipulation:

```
evaluationManager -man 0;
```

You should avoid using Analysis Mode during animation since it will slow down your scene. See Analysis Mode for details.

Custom Evaluators

Once the EG has been created, Maya can target evaluation of node sub-graphs. In this section, we'll review how we've used custom evaluators to accelerate deformations and trap evaluation errors on specific scenes. Currently it's not possible for users to author new custom evaluators. In the future, we may extend OpenMaya to support such extensions.

Tip: Use the `evaluator` command to query the available evaluators and query or modify evaluators that are currently active.

```
import maya.cmds as cmds

# Returns a list of all evaluators currently available
cmds.evaluator( query=True )
# Result: [u'dynamics', u'pruneRoots', u'deformer', u'cache',
u'null'] #

# Returns a list of all evaluators currently enabled.
cmds.evaluator( query=True, enable=True )
# Result: [u'dynamics', u'pruneRoots'] #
```

GPU Override

To accelerate deformations in Viewport 2.0, Maya 2016 contains a custom **deformer evaluator** that targets mesh deformations on the GPU using OpenCL. The massively parallel nature of modern GPUs makes them ideal for problems such as deformations that must perform the same operations on streams of data, such as mesh vertices and normals. We've included GPU implementations for 6 of the most commonly-used deformers in animated scenes: **skinCluster**, **blendShape**, **cluster**, **tweak**, **groupParts**, and **softMod**.

Unlike Maya's previous deformer stack, that performed deformations on the CPU and subsequently sent deformed geometry to the graphics card for rendering, the GPU override sends *undeformed* geometry to the graphics card, performs deformations in OpenCL and hands-off data to Viewport 2.0 for rendering without read-back overhead. We've observed substantial speed improvements from this approach for many scenes with dense geometry.

Even if your scene uses only supported deformers, it's possible the GPU override may not be enabled due to unsupported node features. For example, with the exception of softMod, deformers must currently apply to all vertices and there is no support for incomplete group components. Additional deformer specific limitations are listed below:

Deformer	Limitation
skinCluster	Value of the following attributes are ignored: • bindMethod • basePoints • bindPose • bindVolume • dropOff • driverPoints • envelope • heatmapFalloff • influenceColor • lockWeights • maintainMaxInfluences • maxInfluences • nurbsSamples • paintWeights • paintTrans • smoothness • useComponents • weightDistribution • weightList
blendShape	Value of the following attributes are ignored: • baseWeights • baseOrigin • icon • inputTarget • inputTargetGroup • inputTargetItem • targetWeights • normalizationId • normalizationGroup • origin • parallelBlender • supportNegativeWeights • targetOrigin • topologyCheck • useTargetCompWeights
cluster	n/a
tweak	Only relative mode is supported. relativeTweak must be set to 1.
groupParts	n/a
softMod	• Only volume falloff is supported when distance cache is disabled • Falloff must occur in all axes • Partial resolution must be disabled

There are a few other reasons that will prevent GPU override from accelerating your scene:

- **Meshes are not sufficiently dense.** Unless meshes have a large enough number of vertices, it's still faster to perform deformations on the legacy CPU path. This is due to driver-specific overhead incurred when sending data to the GPU for processing. For deformations to happen on the GPU, your mesh needs over 500/2000 vertices, on AMD/NVIDIA hardware respectively. You can change the threshold by using the `MAYA_OPENCL_DEFORMER_MIN_VERTS` environment variable. Setting the value to 0 will make it possible for all meshes connected to supported deformation chains to be processed on the GPU.

- **Downstream nodes in your graph read deformed mesh results.** No node, script, or viewport can read the mesh data computed by the GPU override. This means that GPU override won't be able to accelerate portions of the deformation chain upstream of nodes, such as follicle or

pointOnPolyConstraint, as it requires information about the deformed mesh. GPU data read-back is a known bottleneck in the area of GPGPU. Evolving standards and hardware aim to remove some of these inefficiencies. We'll re-examine this limitation as software/hardware capabilities mature. When diagnosing GPU Override problems this situation may be reported as an unsupported fan-out pattern. See below details on the `deformerEvaluator` command for details.

- **Meshes have animated topology changes.** If your scene animates the number of mesh edges, vertices, and/or faces during playback, corresponding deformation chains will be removed from the GPU deformation path.

- **Maya Catmull-Clark Smooth Mesh Preview is used.** We've included acceleration for OpenSubDiv (OSD)-based smooth mesh preview but there is currently no support for Maya's legacy Catmull-Clark. To take advantage of OSD OpenCL acceleration, select OpenSubDiv Catmull-Clark as the subdivision method and make sure that OpenCL Acceleration is selected in the OpenSubDiv controls.

- **Unsupported streams are found.** Depending on the drawing mode you've selected for your geometry (e.g., shrunken faces, hedge-hog normals, etc) and the material assigned to your geometry Maya will need to send different information to the graphics card. This requires that Maya allocate different streams. Since we've focused our efforts on the most common settings used in production settings, Maya doesn't currently handle all streams. If you determine that your meshes are failing to accelerate due to unsupported streams, try changing the display mode and/or updating the material used by the geometry.

- **Back face culling is enabled**

- **Driver-related issues.** We're aware of various hardware issues related to driver support/stability for OpenCL. To maximize Maya's stability, we've disabled GPU Override in the specific cases that will lead to problems. For example, currently if you are using VP2 OpenGL Core Profile on OSX, we'll disable GPU override. We expect to eliminate this restriction in the future and are actively working with hardware vendors to address detected driver problems.

You can also increase support for new custom/proprietary deformers using new API extensions (refer to Custom GPU Deformers for details).

If you've enabled GPU Override and the HUD reports *Enabled (0 k)*, this indicates that no deformations are happening on the GPU. There could be a number of reasons, such as those mentioned above.

To troubleshoot factors limiting use of GPU override for your particular scene, use the `deformerEvaluator` command. Supported options include:

Command	What does it do?
`deformerEvaluator;`	For each selected node print the chain or a reason it is not supported
`deformerEvaluator -chains;`	Print all active deformation chains
`deformerEvaluator -meshes;`	Print a chain for each mesh or a reason it is not supported

Dynamics Evaluator

Maya 2016 has limited support for animated dynamics. **Although scenes with Bullet rigid bodies and Bifrost fluids should evaluate correctly, attempting to playback or manipulate scenes with animated legacy (particles, fluids) or nucleus (nCloth, nHair, nParticles) nodes will disable the EM and revert to DG-based evaluation.** This is due to the fact that legacy dynamics relies on special evaluation rules, to generate repeatable and stable results, which may violate DG evaluation best practices. We're actively working to remove some of these restrictions, broadening the set of scenes that can take advantage of Parallel evaluation.

To handle dynamics, we've created a special dynamics evaluator responsible for detecting animated dynamics in your scene and disabling the evaluation manager. This is another example of how custom evaluators are used in Maya 2016 to alter the way evaluation is handled.

If you wish to test the impact of the new evaluation modes with scenes that contain dynamics, you can manually turn off the dynamics evaluator using the following command:

```
evaluator -en off -name dynamics
```

Note: Disabling the dynamics evaluator may cause evaluation problem and/or application crashes; this is unsupported behavior. Proceed with caution.

Other Evaluators

In addition to the GPU override and dynamics evaluators, additional evaluators exist for specialized tasks:

Evaluator	What does it do?
pruneRoots	Testing revealed that scenes with several thousand paramCurves can become bogged down scheduling resulting EG nodes, losing potential gains from increased parallelism. To handle this situation, special clusters are created that group paramCurves into a small number of evaluation tasks, thus reducing overhead.

Custom evaluator names are subject to change as we introduce new evaluators and expand these funcationalities.

Evaluator Conflicts

There is currently no mechanism to handle situations where multiple evaluators conflict with one another and "claim responsibility" for the same node. This may negatively impact performance, since an evaluator with a relatively small speedup, such as *pruneRoots*, may take control of a node that would otherwise go to an evaluator with a larger speedup (e.g., *deformerEvaluator*). For now, there are few enough evaluators that this is not a cause for concern. In the future we expect to have a more formal mechanism to manage multiple evaluators and resolve conflicts.

API Extensions

Maya 2016 includes API extensions to help your pipeline and tools make the most of the new evaluation capabilities. This section reviews API extensions for **Parallel Evaluation, Custom GPU Deformers**, and **Profiling Plug-ins**.

Parallel Evaluation

If your plug-in plays by the DG rules, you are likely to need very few changes for your plug-in to work in Parallel mode. Porting your plug-in to Maya 2016 may be as simple as recompiling it against the latest version of OpenMaya!

If the EM is generating different results, make sure that your plug-in:

- **Overrides MPxNode::compute().** This is especially true of classes extending MPxTransform which previously relied on asMatrix(). See the rockingTransform SDK sample. For classes deriving from **MPxDeformerNode** and **MPxGeometryFilter**, override the deform() method.
- **Handle requests for evaluation at all levels of the plug tree.** While the DG can request plug values at any level, the EM always requests the root plug. For example, for plug N.gp[0].p[1] your compute() method must handle requests for evaluation of N.gp, N.gp[0], N.gp[0].p, and N.gp[0].p[1].

If your plug-in relies on custom dependency management, you'll need to use new API extensions to ensure correct results. As described earlier, Maya 2016 requires a correct EG to generate correct evaluation results. Since the EG is created using the legacy dirty-propagation mechanism, you'll need to make sure that the custom logic in MPxNode::setDependentsDirty override methods are accounted for. Additionally, plug-in specific optimization to limit dirty propagation should be disabled during EG graph rebuilding. **MEvaluationManager::graphConstructionActive()** can be used to detect if this is occurring.

There are new virtual methods you'll want to consider implementing:

MPxNode::preEvaluation. To avoid performing expensive calculations every time MPxNode::compute is called, one strategy plug-in authors use is to store results from previous evaluations and then rely on MPxNode::setDependentsDirty to trigger re-computation. As discussed previously, once the EG has been built, dirty propagation is disabled and the EG is re-used. Threrefore, any custom logic in your plug-in that depends on setDependentsDirty no longer applies.

MPxNode::preEvaluation allows your plug-in to determine which plugs/attributes are dirty and if any action is needed. The new MEvaluationNode class can be used to determine what has been dirtied.

Refer to the simpleEvaluationNode.cpp devkit example for an illustration of how to use MPxNode::preEvaluation.

MPxNode::postEvaluation. Until now it was difficult to determine when all processing for a particular node instance was complete. Users sometimes had to resort to complex bookkeeping/callbacks schemes to detect this situation and perform additional work, such as custom rendering. This mechanism was cumbersome and error-prone.

A new method, MPxNode::postEvaluation, is called once all computations have been performed on a specific node instance. Since this method is called from a worker thread, it can perform calculations for downstream graph operations, without blocking other Maya processing tasks of non-dependent nodes.

See the simpleEvaluationDraw devkit example to understand how to use this method. If you run this example in regular evaluation, Maya slows down, since evaluation is blocked whenever expensive calculations are performed. When you run in Parallel Evaluation Mode, a worker thread calls the postEvaluation method and prepares data for subsequent drawing operations. When testing, you will see higher frame rates in parallel evaluation versus regular or serial evaluation. Please note that code in postEvaluation should be thread-safe.

Some other recommended best practices include:

Avoid storing state in static variables. Store node state/settings in attributes. This has the additional benefit of automatically saving/restoring the plug-in state when Maya files are written/read.

Node computation shouldn't have any dependencies beyond input values. Maya nodes should be like functions. Output values should be computed from input state and node-specific internal logic. Your node should never walk the graph or try to circumvent the DG.

Custom GPU Deformers

To make GPU Override work on scenes containing custom deformers, Maya 2016 provides new API classes that allow the creation of fast OpenCL deformer back-ends.

Though you'll still need to have a CPU implementation for the times when it's not possible to target deformations on the GPU (see GPU Override), you can augment this with an alternate deformer implementation inheriting from MPxGPUDeformer. This applies to your own nodes as well as to standard Maya nodes.

The GPU implementation will need to:

- Declare when it's valid to use the GPU-based backend (e.g., you may want to limit you GPU version to cases where various attributes are fixed, omit usage for specific attribute values, etc)
- Extract MDataBlock input values and upload values to the GPU
- Define and call the OpenCL kernel to perform needed computation
- Register itself with the MGPUDeformerRegistry system. This will tell the system which deformers you are claiming responsibility for.

When you've done this, don't forget to load your plug-in at startup. Two working devkit examples (offsetNode and identityNode) have been provided to get you started.

Tip: To get a sense for the maximum speed increase you can expect by providing a GPU backend for a specific deformer, tell Maya to treat specific nodes as passthrough. Here's an example applied to polySoftEdge:

```
GPUBuiltInDeformerControl
      -name polySoftEdge
      -inputAttribute inputPolymesh -outputAttribute output
      -passthrough;
```

Although results will be incorrect, this test can confirm if it's worth investing time in implementing an OpenCL version of your node.

Profiling Plug-ins

To visualize how long custom plug-ins are taking in the new profiling tools (see Profiling Your Scene) you'll need to instrument your code. Maya 2016 provides C++, Python, and Mel interface for you to do this. Refer to the Profiling using MEL or Python or the API technical docs for more details.

Profiling Your Scene

In the past, it could be challenging to understand where Maya was spending time. To remove the guess work out of performance diagnosis, Maya 2016 includes a new integrated profiler that lets you see exactly how long different tasks are taking.

You can open the Profiler by selecting:

- **Windows > General Editors > Profiler** from the Maya menu
- **Persp/Graph Layout** from the Quick Layout buttons and choosing **Panel Layout > Profiler**.

Once the Profiler window is visible:

1. Load your scene and start playback
2. Click **Start** in the Profiler to record information in the pre-allocated record buffer.
3. Wait until the record buffer becomes full or click **Stop** in the Profiler to stop recording. The Profiler shows a graph demonstrating the processing time for your animation.
4. Try recording the scene in **DG**, **Serial**, **Parallel**, and **GPU Override** modes.

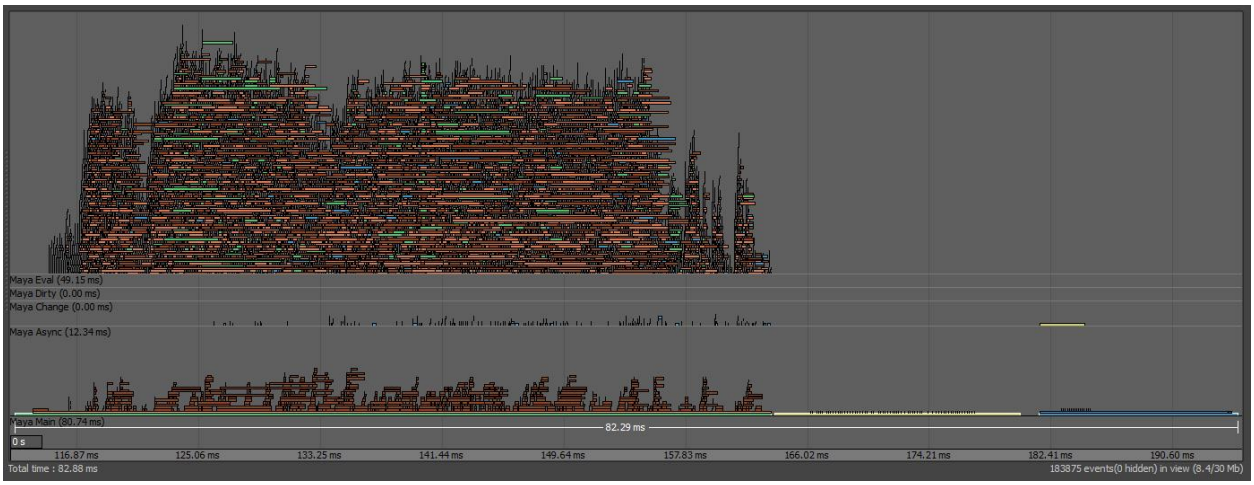
The Profiler includes information for all instrumented code, including playback, manipulation, authoring tasks, and UI/Qt events. When profiling your scene, make sure to capture several frames of data to ensure gathered results are representative of scene bottlenecks.

Tip: By default the profiler allocates a 10MB buffer to store results. The record buffer can be expanded via the UI or using the command:

```
profiler -b value
```

where `value` is the desired size in MB. This may be needed for more complex scenes.

The Profiler supports several views depending on the task you wish to perform. The default **Category View**, shown below, classifies events by type (e.g., dirty, VP1, VP2, Evaluation, etc). The **Thread** and **CPU** views show how function chains are subdivided amongst available compute resources. Currently the Profiler does not support visualization of GPU-based activity.



Now that you have a general sense of what the Profiler tool does, let's discuss key phases involved in computing results for your scene and how these are displayed. By understanding why scenes are slow, you can target scene optimizations.

Every time Maya updates a frame, it must compute and draw the elements in your scene. Hence, computation can be split into one of two main categories:

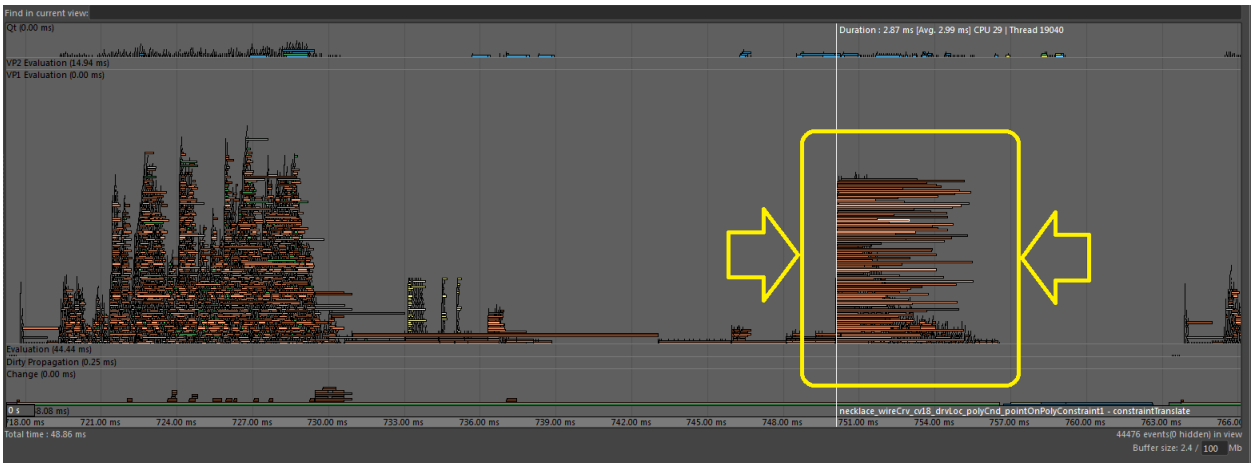
1) Evaluation (i.e., doing the math that determines the most up-to-date values for scene elements)
2) Rendering (i.e., doing the work that draws your scene in the viewport).

When the main bottleneck in your scene is evaluation, we say the scene is **evaluation-bound**. When the main bottleneck in your scene is rendering, we say the scene is **render-bound**.

Evaluation-Bound Performance

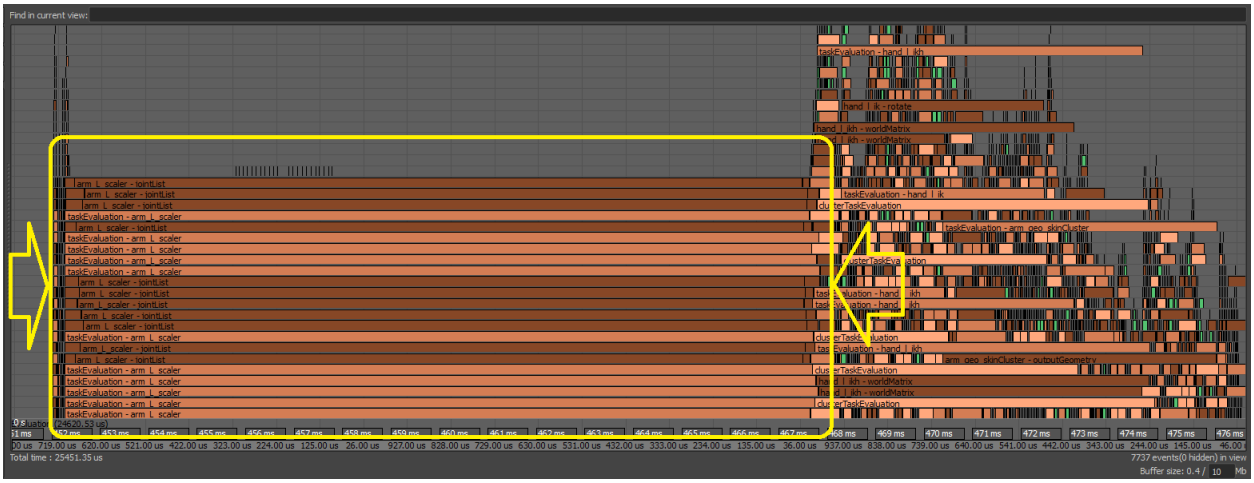
There are several different problems that may lead to evaluation-bound performance.

Lock Contention. When many threads try to access a shared resource you may experience Lock Contention, due to lock management overhead. One clue that this may be happening is that evaluation takes roughly the same duration regardless of which evaluation mode you use. This occurs since threads cannot proceed until other threads are finished using the shared resource.



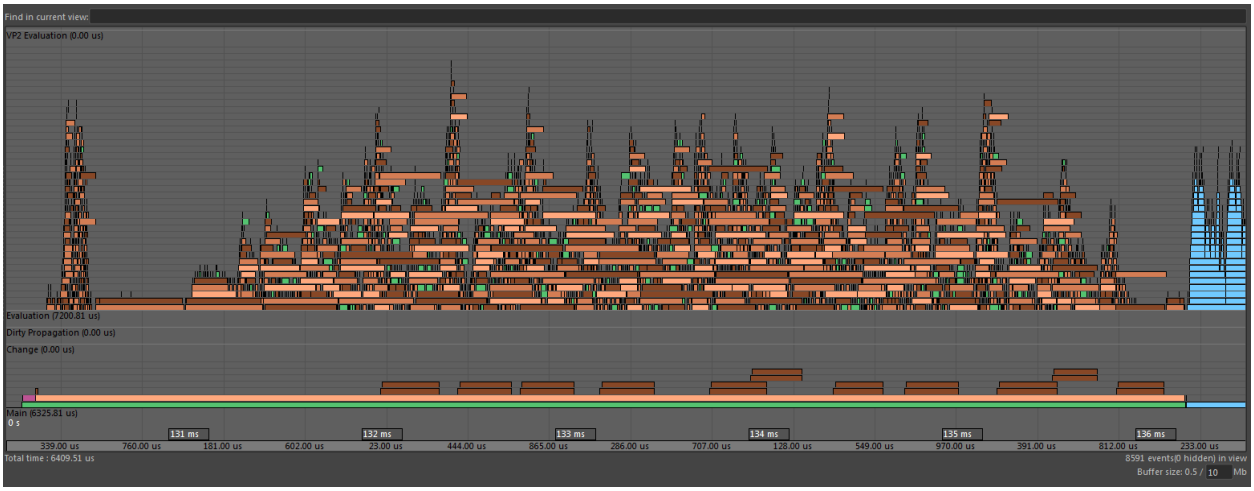
Here the Profiler shows many separate identical tasks that start at nearly the same time on different threads, each finishing at different times. This type of profile offers a clue that there might be some shared resource that many threads need to access simultaneously.

Below is another image showing a similar problem.

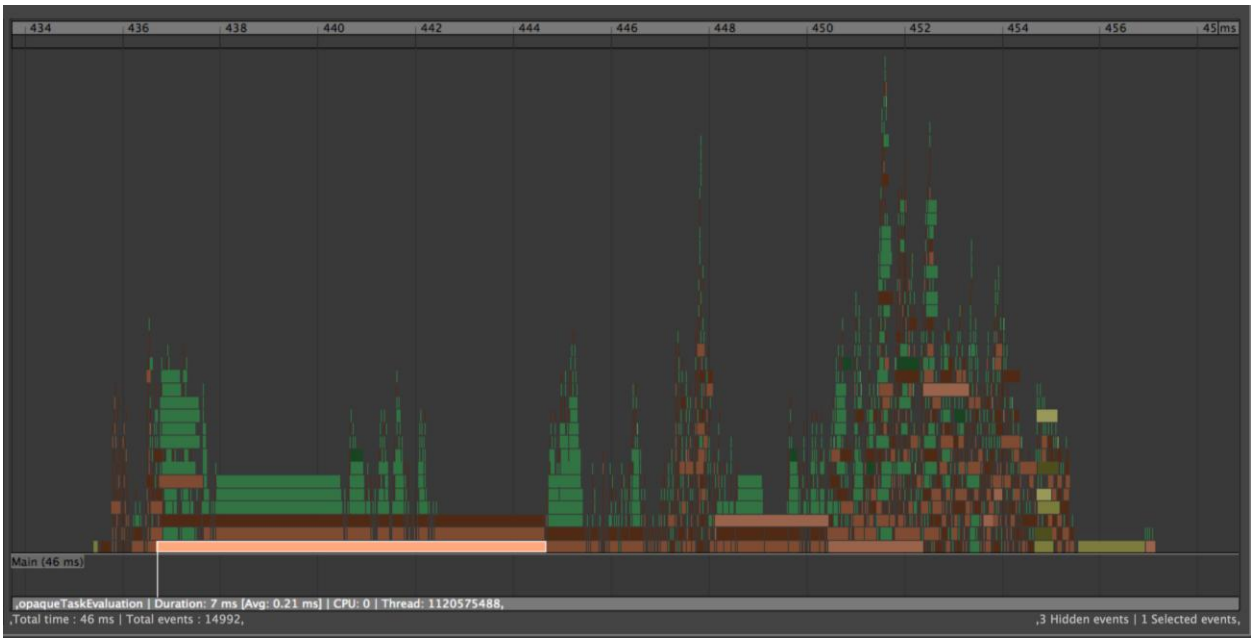


In this case, since several threads were executing Python code, they all had to wait for the Global Interpreter Lock (GIL) to become available. Bottlenecks and performance loses caused by contention issues may be more noticeable when there is a high concurrency level, such as when your computer has many cores.

If you encounter contention issues, try to fix the code in question. If you are using many Python plug-ins, consider scheduling these as Globally Serial. For the above example, changing node scheduling converted the above profile to the following one, providing a nice performance gain.



Clusters. As mentioned earlier, if the EG contains node-level circular dependencies, those nodes will be grouped into a **cluster** which represents a single unit of work to be scheduled serially. Although multiple clusters may be evaluated at the same time, large clusters limit the amount of work that can be performed simultaneously. Clusters can be identified in the Profiler as bars with the **opaqueTaskEvaluation** label, shown below.

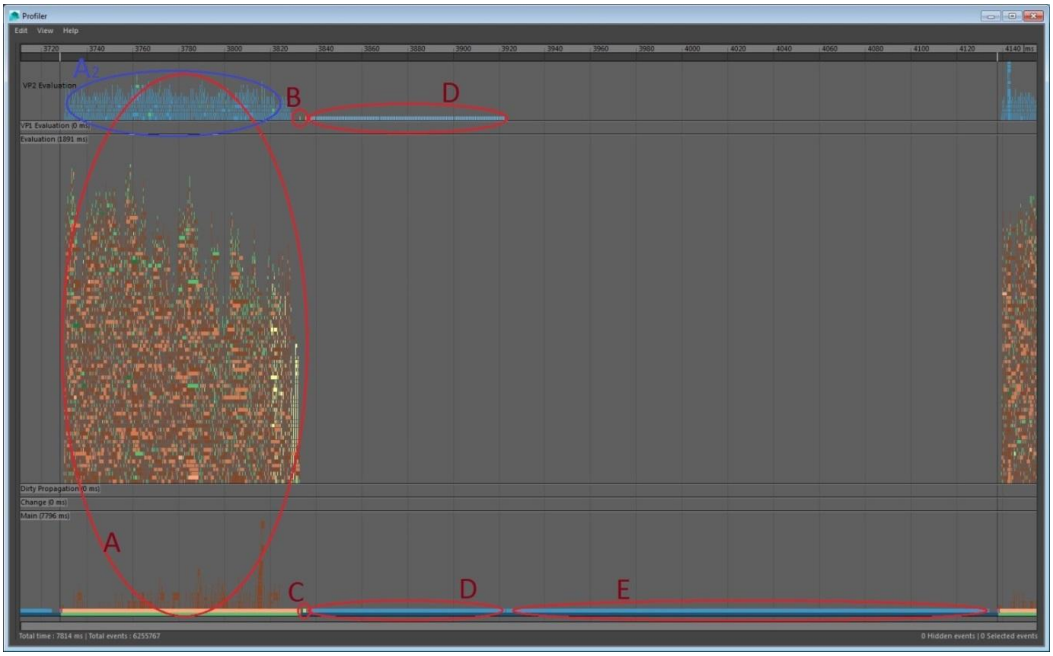


If your scene contains clusters, analyze your rig's structure to understand why circularities exist. Ideally, you should strive to remove coupling between parts of your rig, so rig sections (e.g., head, body, etc.) can be evaluated independently.

Tip: When troubleshooting scene performance issues, you can temporarily disable costly nodes using the per-node **frozen** attribute. This removes specific nodes from the EG. Although the result you see will change, it's a simple way to check that you've found the bottleneck for your scene.

Render-Bound Performance

Below, is an illustration of a sample scene in the Maya Profiler made up of many small, rigidly bound meshes on a complex rig. Textures are enabled and the scene uses the GL Core Profile version of Viewport 2.0. Evaluation is done in Parallel mode. The scene runs at 2.6 fps (approx. 400 ms per frame). Because of the nearly 12,000 visible VP2 nodes this scene is render-bound.



The attached profile has five main areas:

- Evaluation
- Vp2UpdateVisibilityTask
- Vp2BaseRendererUpdateHUD
- Vp2BuildRenderLists
- Vp2DrawLists

With the exception of Evaluation, all other tasks are related to VP2 rendering.

Evaluation. Area A depicts time spent computing the state of the Maya scene. In this case, the scene is nicely parallelized. There is tight packing of events in the Evaluation category, illustrating the high level of concurrency.

Area A2 (above area A), shows the work that VP2 does in parallel to the evaluation. In this scene, there are a large number of transform updates, run from the EM. Scenes with many deformation would also require work to prepare geometry for rendering that would appear as larger blue tasks in A2.

Vp2UpdateVisibilityTask. Area B is the first of the three primary render phases.

VP2 maintains an internal copy of the world and uses it to determine what to draw in the viewport. When it's time to render the scene, we must first ensure that the objects in the VP2 database have been modified to reflect changes in the Maya scene. For example, objects may have become visible or hidden, their position or their topology may have changed, etc. This is done by VP2 UpdateVisibilityTask.

UpdateVisibilityTask can be slow when there are shape topology, material, or object visibility changes. In this particular example, UpdateVisibilityTask is quick since little processing required on scene objects.

Vp2BaseRendererUpdateHUD. Area C corresponds to the work needed to update the Heads Up Display (HUD) messages. This is usually not a significant bottleneck. In extreme cases, the HUD may require additional (slow) DG evaluation, resulting in conflicts with GPU override and introducing major performance issues. One example of this is when the Poly Count HUD is enabled during playback.

Vp2BuildRenderList. Once visibility and HUD elements have been updated, VP2 must next build the list of objects to render. This can be seen in area D in the profiler (see main thread category); instrumented functions can also be seen doing some computation in the "Viewport 2.0 Evaluation" profiler category.

Some common cases for which Vp2BuildRenderLists can be slow during Parallel Evaluation are:

- Large numbers of rendered objects (as in this example)
- Mesh topology changes
- Object types, such as image planes, requiring legacy evaluation before rendering
- 3rd party plug-ins that trigger API callbacks

On AMD GPUs: If the GPU Override is loaded with heavy meshes, the main thread may block waiting for OpenCL work to complete. On NVIDIA GPUs this blocking happens earlier (in a different phase).

In this test scene, with 12,000 nodes to render, a lot of time is spent in VP2 BuildRenderLists.

Vp2DrawLists. Once all data has been prepared, it's time to render the scene (Area E). VP2 DrawLists is where the actual OpenGL or DirectX rendering occurs. DrawLists can be slow if your scene:

- **Has Many Objects to Render** (as in this example).
- **Uses Transparency or Many Materials**. In VP2, objects are sorted by material prior to rendering. Hence, having many materials can make this time-consuming. Transparent objects are also costly since objects must be sorted prior to being drawn. In this scene, removing transparency (e.g., using untextured mode) nearly doubles the performance.
- **Uses Viewport Effects.** Many effects such as SSAO (Screen Space Ambient Occlusion), Depth of Field, Motion Blur, Shadow Maps, or Depth Peeling require additional processing.

Other Considerations. Although the key phases described above apply to all scenes, your scene may have different performance characteristics.

For static scenes with limited animation, consolidation is used to improve performance. Consolidation groups objects that share the same material and that are close together. This reduces time spent in both BuildRenderLists and DrawLists, since there are fewer objects to render.

Troubleshooting Your Scene

Analysis Mode

The purpose of Analysis mode is to perform more rigorous inspection of your scene to catch evaluation errors. Since Analysis Mode introduces overhead to your scene, only use this during debugging activities; animators should not enable Analysis Mode during their day-to-day work. Note that Analysis Mode is not thread-safe, so it is limited to Serial; you cannot use analysis mode while in Parallel evaluation.

The key function of Analysis Mode is to:

- **Search for errors at each playback frame.** This is different than Safe Mode, which only tries to identify problems at the start of parallel execution.
- **Monitor read-access to node attributes.** This ensures that nodes have a correct dependency structure in the EG.
- **Return diagnostics to better understand which nodes influence evaluation.** This is currently limited to reporting one destination node at a time.

Tip: To activate Analysis Mode, use the following MEL command:

```
dbtrace -k evalMgrGraphValid;
```

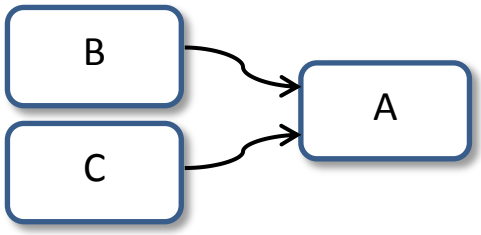
To disable Analysis Mode, type:

```
dbtrace -k evalMgrGraphValid -off;
```

Once active, error detection occurs after each evaluation. Missing dependencies are saved to a file in your machine's temporary folder (e.g., %TEMP%_MayaEvaluationGraphValidation.txt on Windows). The temporary directory on your platform can be determined using the command:

```
internalVar -utd;
```

Let's assume that your scene contains the following three nodes. Because of the dependencies, the evaluation manager must compute the state of nodes B and C prior to calculating the state of A.



Now let's assume Analysis Mode returns the following report:

Detected missing dependencies on frame 56
{
A.output <-x- B
A.output <-x- C [cluster]
}
Detected missing dependencies on frame 57
{
A.output <-x- B
A.output <-x- C [cluster]
}

The "<-x-" symbol indicates the direction of the missing dependency. The "[cluster]" term indicates that the node is inside of a cycle cluster, which means that any nodes from the cycles could be responsible for attribute access outside of evaluation order

In the above example, B accesses the "output" attribute of A, which is incorrect. These types of dependency do not appear in the Evaluation graph and could cause a crash when running an evaluation in Parallel mode.

There are multiple reasons that missing dependencies occur, and how you handle them depends on the cause of the problem. If Analysis Mode discovers errors in your scene from bad dependencies due to:

- **A user plug-in.** Revisit your strategy for managing dirty propagation in your node. Make sure that any attempts to use "clever" dirty propagation dirty the same attributes every time. Avoid using different notification messages to trigger pulling on attributes for computation.
- **A built-in node**. You should communicate this information to us. This may highlight an error that we are unaware of. To help us best diagnose the causes of this bug, we'd appreciate if you can provide us with the scene that caused the problem.

Graph Execution Order

There are two primary methods of displaying the graph execution order.

The simplest is to use the 'compute' trace object to acquire a recording of the computation order. This can only be used in Serial mode, as explained earlier. The goal of compute trace is to compare DG and EM evaluation results and discover any evaluation differences related to a different ordering or missing execution between these two modes.

Tip: To activate graph execution order tracing, use the following MEL command:

```
dbtrace -k compute;
```

To disable analysis mode, type:

```
dbtrace -k compute -off;
```

Keep in mind that there will be many differences between runs since the EM executes the graph from the roots forward, whereas the DG uses values from the leaves. For example in the simple graph shown earlier, the EM guarantees that B and C will be evaluated before A, but provides no information about the relative ordering of B and C. However in the DG, A pulls on the inputs from B and C in a consistent order dictated by the implementation of node A. The EM could show either "B, C, A" or "C, B, A" as their evaluation order and although both might be valid, the user must decide if they are equivalent or not. This ordering of information can be even more useful when debugging issues in cycle computation since in both modes a pull evaluation occurs, which will make the ordering more consistent.

The EM Shelf

The BonusTools has a special shelf specifically aimed at working with the EM that contains features to query and analyze your scene and to toggle various modes on/off. See the accompanying shelf documentation for a complete list of all shelf features.

Known Limitations

This section lists known limitations for the new evaluation system.

- **VP2 Motion Blur will disable Parallel evaluation.** For Motion Blur to work, the scene must be evaluated at different points in time. Currently the EM does not support this.
- **Scenes using FBIK will revert to Serial.** For several years now Autodesk has been deprecating FBIK. We recommend using HIK for full-body retargeting/solving.
- **dbtrace will not work in Parallel mode.** As stated in the Analysis Mode section, the dbtrace command will only work in Serial evaluation. Having traces enabled in Parallel mode will likely cause Maya to crash.
- **The DG Profiler crashes in Parallel Mode.** Unless you are in DG evaluation mode, you'll be unable to use the legacy DG profiler. As time permits, we expect to move features of the DG profiler into the new thread-safe integrated profiler.
- **Batch rendering scenes with XGen may produce incorrect results.**
- **Evaluation manager in both Serial and Parallel mode changes the way attributes are cached.** This is done to allow safe parallel evaluation and prevent re-computation of the same data by multiple threads. This means that some scenes may evaluate differently if multiple computations of the same attribute occur in one evaluation cycle. With the Evaluation Manager the first value will be cached.
- VP2 Direct update doesn't work with polySoftEdge nodes.