Introduction to the Dependency Graph for Maya Programmers

Autodesk Maya

Introduction

This document is an introduction to Maya's Dependency Graph (DG) that is especially relevant to programmers using the Maya Embedded Language (MEL) and/or Maya's C++ Application Programming Interface (API). It will cover the entities that make up the DG: nodes, attributes, and connections, as well as plugs, datablocks and datahandles. It will also explain the DG evaluation process in detail. Finally, it will cover how data is passed within the DG and the Directed Acyclic Graph (DAG) subset of the DG.

The Dependency Graph

At the heart of Maya's architecture is the Dependency Graph, or the DG. It is the control and data storage system of Maya. The DG is what allows seemingly disparate operations to work together seamlessly to produce the final output. Those operations are implemented as nodes, which store data through their attributes and pass those data using connections.

Nodes

Nodes are the fundamental building blocks of the DG. A node stores its data through its attributes. Typically, a node's job is to take data from its input attributes to produce data for its output attributes. This is done via a compute() function. Also, a node's attributes can be connected to those of other nodes to pass on data.

Almost everything in Maya is implemented as a node, e.g. meshes, surfaces, curves, locators, transforms, geometric operations, time, animation, dynamics, lighting, shading, etc.

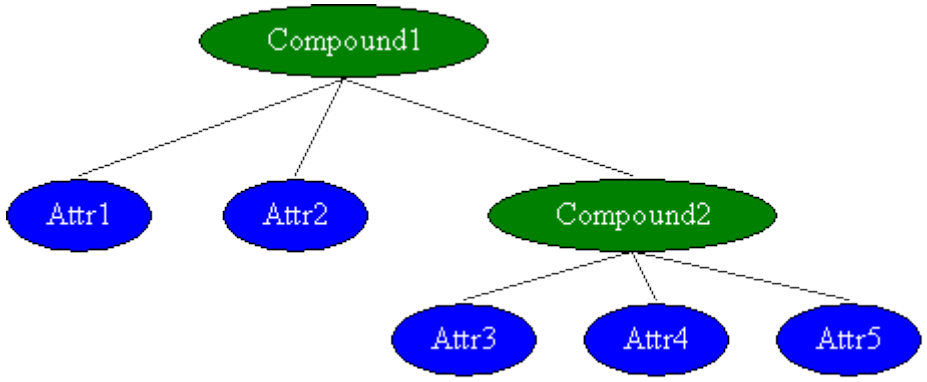
Nodes can also come from Maya plug-ins, implemented using the Maya API. The DG does not distinguish between nodes built into Maya or those coming from plug-ins, adding to the power and flexibility of Maya.

Attributes

Nodes store their data through attributes. An attribute can represent simple data types like int, float, double, string, boolean, etc., or complex types like mesh, curve or surface data, etc.

An attribute can be designated as an input or an output. An output attribute is computed from one or more inputs, so it is not normally stored in a Maya scene file and is not allowed to have incoming connections from other attributes. Input and output attributes also have a relationship that indicates which inputs are used to compute which outputs. This is called an "attributeAffects" relationship.

Attributes can be grouped together in a tree-like structure. Compound attributes allow one or more attributes, either of identical or different types, to be parented underneath it.



Attributes can be made into arrays, sometimes known as multis. An array attribute will not have a single value, but will instead have a list of values with identical types. Array attributes, or multis, are true arrays in that you can access the individual elements of the array.

Maya also has types that contain array data, like intArray, stringArray, or vectorArray, etc. However, unlike array attributes or multis, the values of these types cannot be accessed individually using the [] array indexing operator.

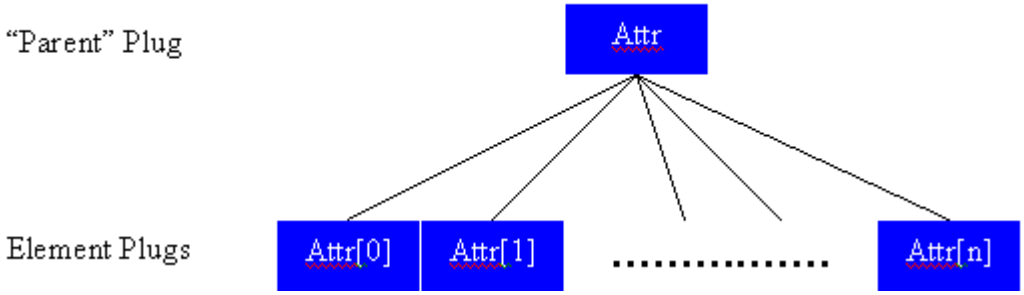
Most attributes are static, meaning they are shared among nodes of a given type. An attribute can also be dynamic, which means it exists only in the node it was added to and is not shared among others, even those with the same node type as the one with the attribute. The DG makes no distinction between static and dynamic attributes during computation and data manipulation.

Plugs

At a high level, attributes and plugs appear to refer to the same thing. However, when programming with Maya, careful distinction needs to be made between them. An attribute merely describes the data that belongs to nodes of a given type, i.e. the type of data it can store and other properties, like whether it's an input or an output, etc.

On the other hand, a plug points to a specific attribute on a specific node, and is used for getting/setting data or making/breaking/validating connections.

Also, an array attribute not only has one plug for each of its elements, but also a "parent" plug representing the entire attribute:



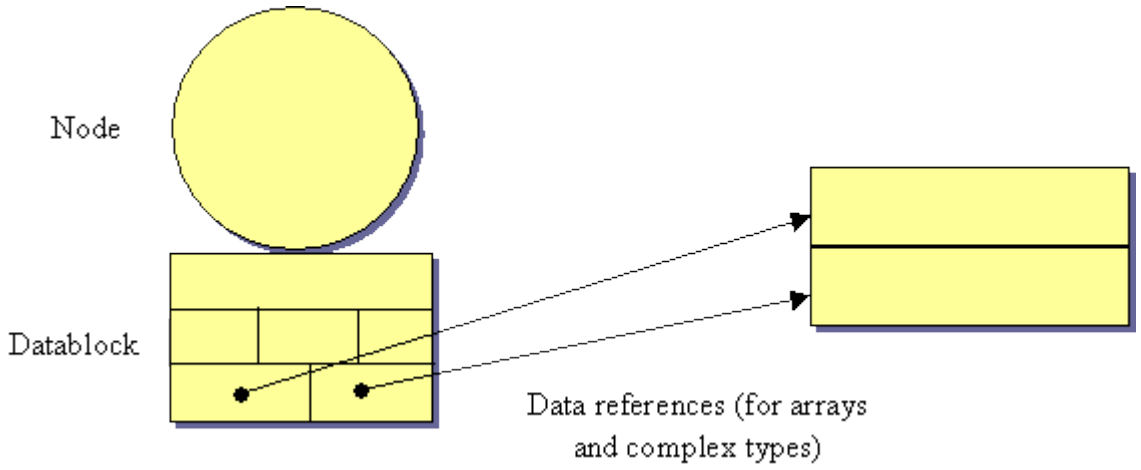
Connections

The power of the DG comes from being able to connect nodes together to accomplish a complex operation. Nodes are connected to each other through their attributes to pass on data. For connections in Maya, data flow is one way only (a source attribute passes its data to a destination but never the other way around). Also, a source attribute can be connected to one or more destination attributes, but a destination can have a connection from only one source.

Datablock

The datablock is the central location where all the node's data from its attributes is stored. A node will have one datablock for the current time context. When needed, the node can also have other transient datablocks to represent other time contexts, for example when the node is animated.

For non-array attributes, the datablock not only stores the value but also the dirty/clean status, which is used during the DG's evaluation process and determines whether or not the value needs to be recomputed. For array attributes and those of a variable size complex type, the datablock stores a reference to another area of memory. This allows complex types to implement their own efficient storage methods, and keeps the size of the datablock constant for faster allocation.



Datahandles

Datahandles are pointers to the data residing in the datablock. A datahandle is similar to a plug in that they are both used to set or retrieve data. However, there are crucial differences:

- A datahandle is used within a node's compute() function, while a plug is used outside.
- A datahandle's get/set methods are more efficient since it intimately knows the layout of the datablock.
- Setting data via a plug propagates the dirty/clean status, while doing so via a datahandle does not.

Since data for array attributes are stored separately, they are accessed using more specialized array datahandles.

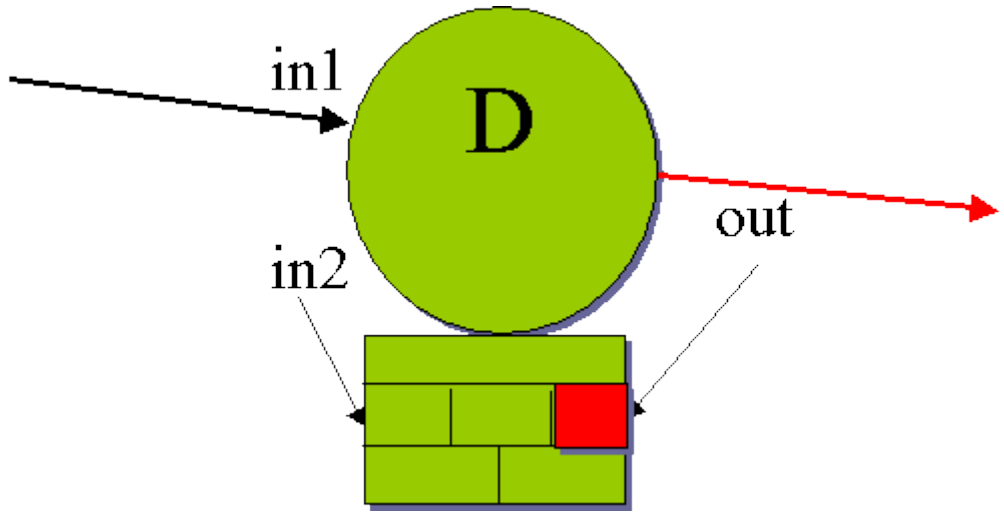
DG Evaluation

The DG's evaluation process can be characterized as a lazy, 2-stage push-pull model. It is lazy in that it only evaluates when necessary.

During the first stage, when an attribute's value is set, a "dirty" message instead of actual data is "pushed" to every affected attribute via attributeAffects relationships or connections.

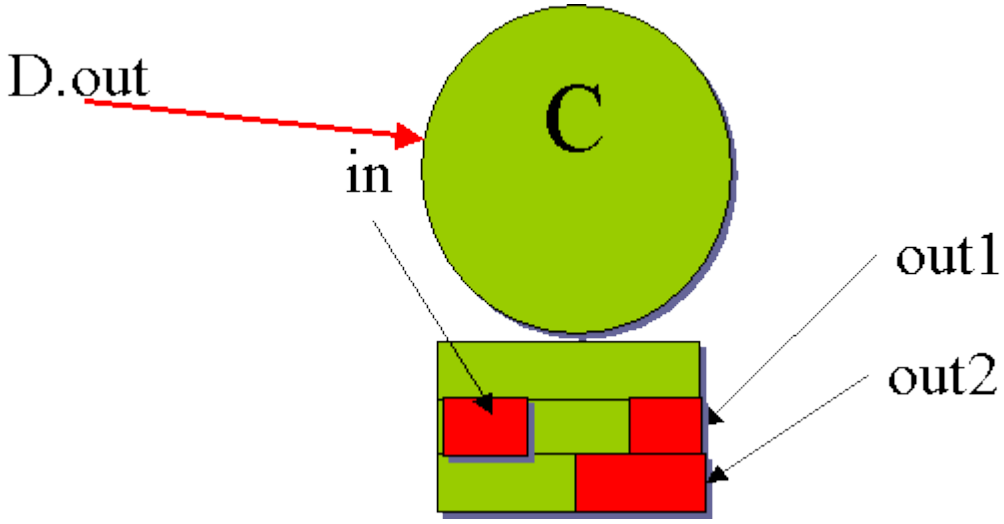
During the second stage, only when a dirty attribute's value is requested will the data be "pulled" by causing the necessary nodes to compute their dirty attributes. Note that at this stage, not all dirty attributes will be generally evaluated at the same time.

We will explain the process using an example. First we will look at some nodes close-up during the first, or "push", stage. Suppose we have a node called "D", with two input attributes "in1" (with an incoming connection) and "in2", one output attribute "out" (with an outgoing connection), and an attributeAffects relation between in2 and out. Then, we set the value of in2:

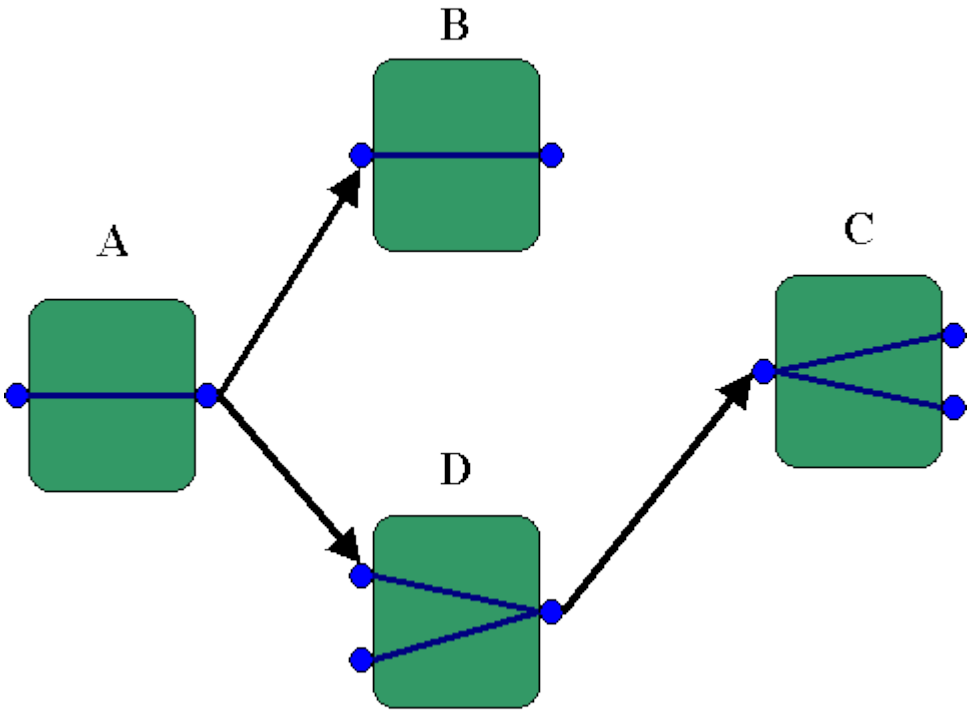


The dirty message (shown in red) propagates through the attributeAffects to the area of the datablock for the out attribute. At this point, out's value won't be computed until in the second stage, when something requests it. The dirty message is then transmitted through out's connection. Note that in2 is not dirty since it already has an up-to-date value.

Suppose we have another node "C", with one input "in" (with an incoming connection from D's out attribute), two outputs "out1" and "out2", and attributeAffects relationships between in and out1, and in and out2.

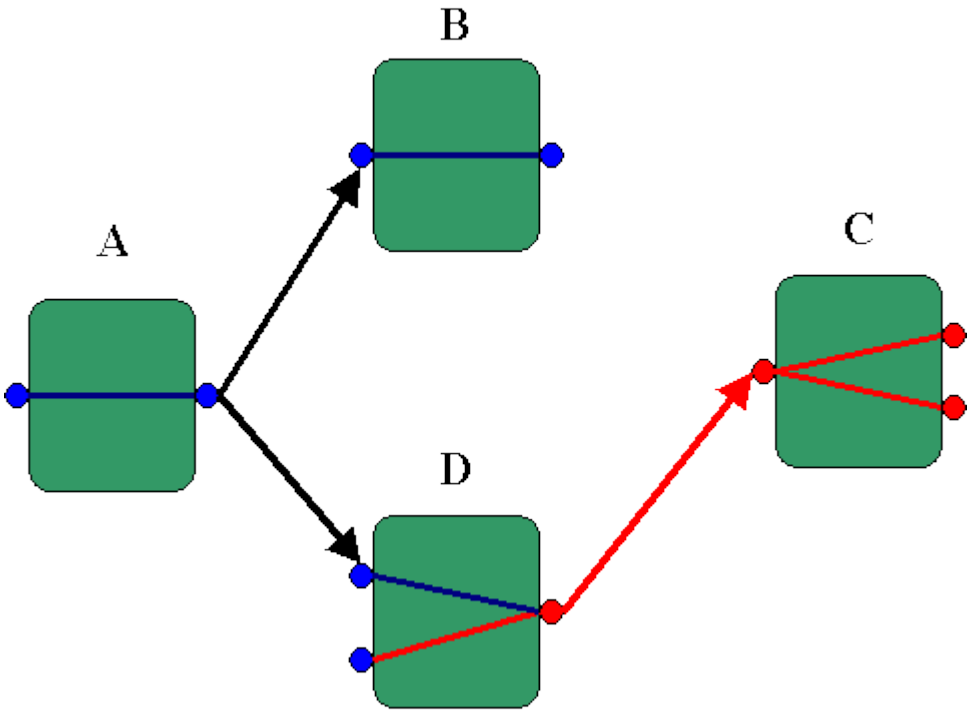


The dirty message from D.out propagates across the connection and sets C.in dirty. The message then propagates across the two attributeAffects relationships, setting both out1 and out2 dirty. Again, at this point, the affected attributes remain dirty and their values are not computed until something requests their values.

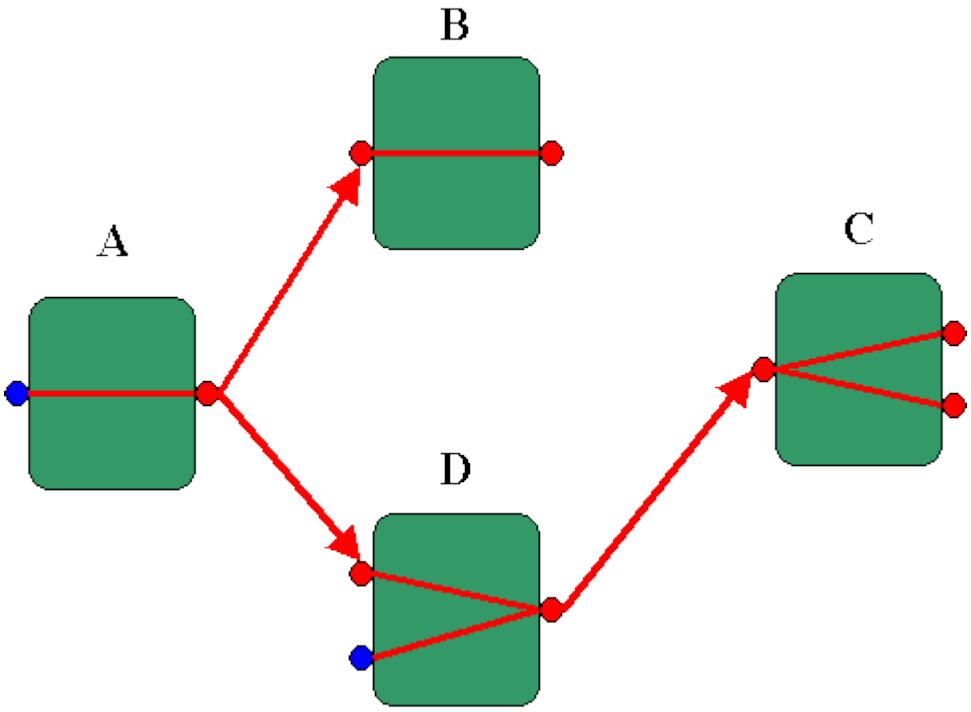


Now, let's look at a network example, consisting of 4 nodes (the blue lines inside the nodes represent attributeAffects relationships).

Suppose we set the value of D.in2 (like in the close-up examples). The graph will now look like:



The dirty message propagates through connections and attributeAffects relations. Again, the attribute that was set is not dirty since it already has an up-to-date value.



Now, suppose we set the value of A.in. As usual, the dirty message will propagate through connections and attributeAffects. However, attributes that are already dirty will not retransmit the message, for example C will not get another dirty message from D. This helps in the DG's performance.

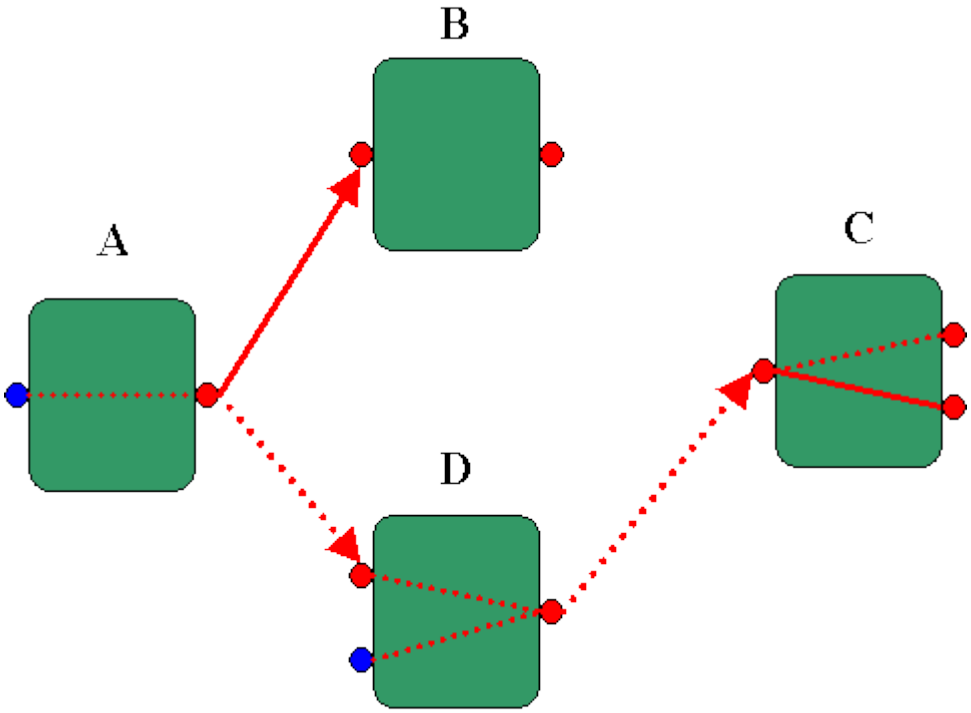
Let us now look at the examples during the second, or "pull", stage. This happens on demend, when something in Maya wants to request the value of an attribute that is dirty. Only requested values will be computed while non-requested ones are left dirty.

One of the following can trigger an evaluation:

- Draw refresh
- UI that is displaying the attribute's value (e.g. Attribute Editor, Channel Box, etc.)
- Rendering
- getAttr MEL command
- MPlug::getValue()

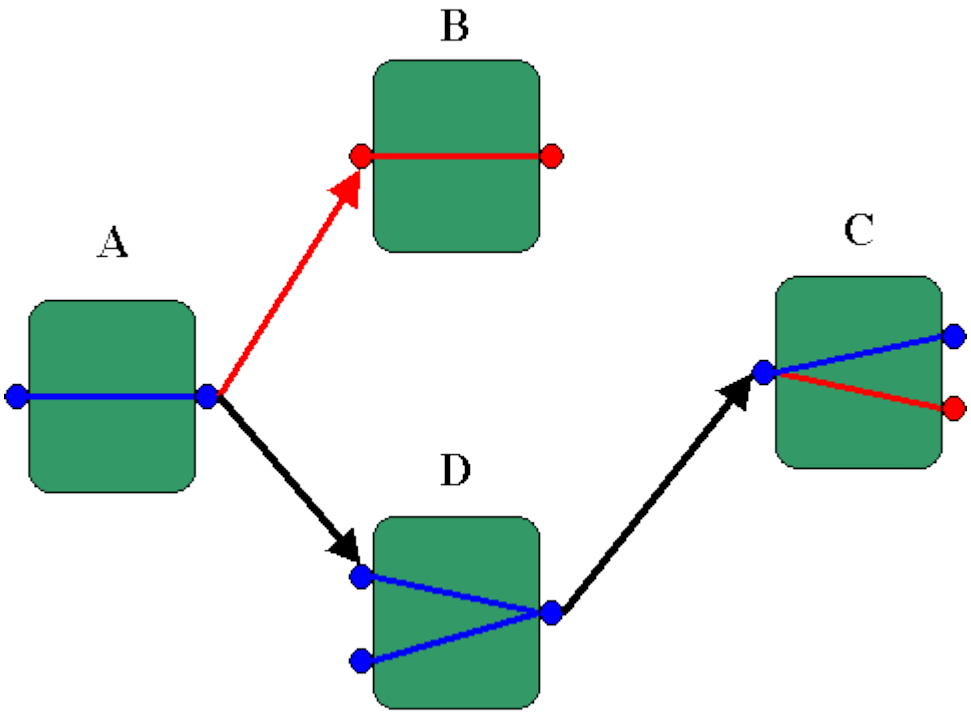
Suppose we retrieve the value of C.out1 using getAttr. The following are the sequence of events:

- C's compute() function executes, and requests the value of its input from the connection.
- D's compute() executes, and requests the values of its inputs
- A's compute() executes, and requests the value of its input



Since A's input is already up-to-date from setting it directly, A can directly use the input's value to compute its output. A's output is then set clean and its value is passed forward through the connection to D's first input.

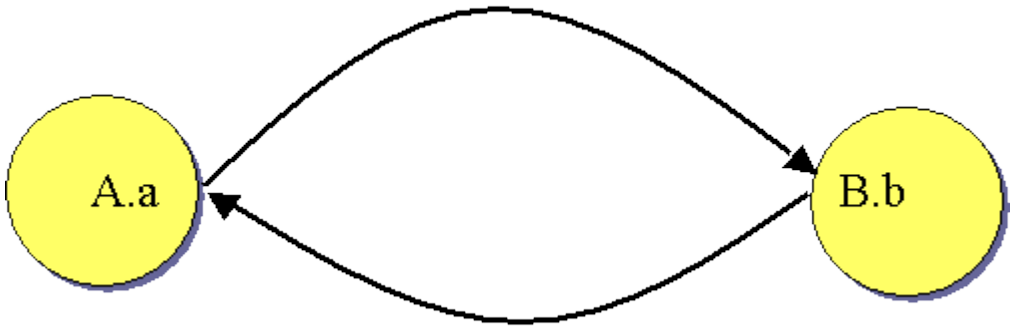
Then, D finishes computing its output from its inputs, sets both its first input and output clean, and passes the value forward through its connection to C's input. C finally finishes computing out1, and sets its input and out1 clean.



Note that non-requested values, like C's second output and B's output, stay dirty and uncomputed, since they were not part of the evaluation chain for C's first output. As long as they are not requested, they will continue to be dirty.

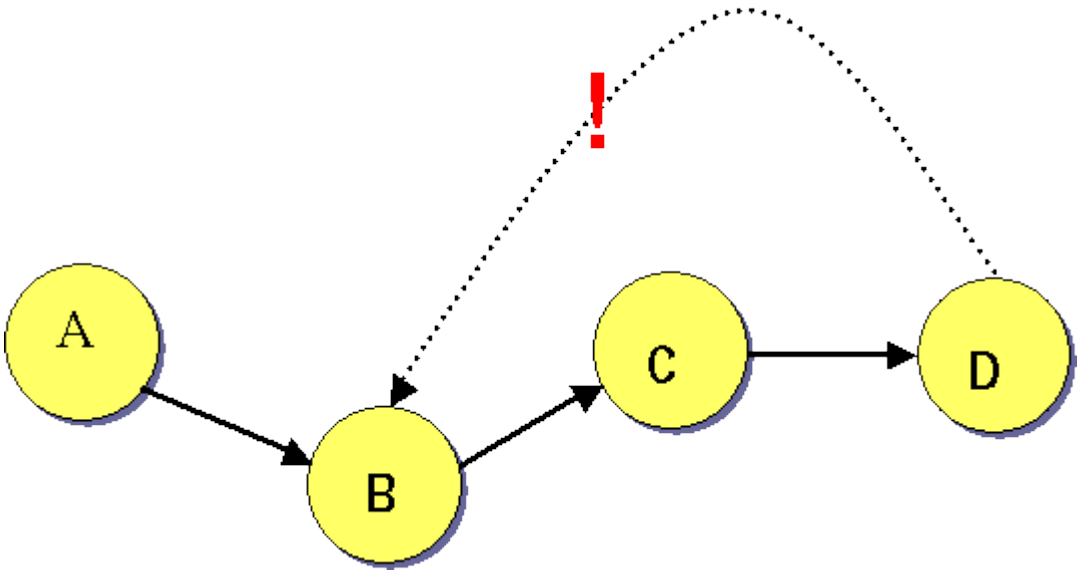
Cycles

Cycles are detected by Maya, but are avoided. Maya does not attempt to resolve them as the DG is not designed as a constraint system. If dependent attributes are involved, the results are not guaranteed to be consistent.



Black-Box Operation

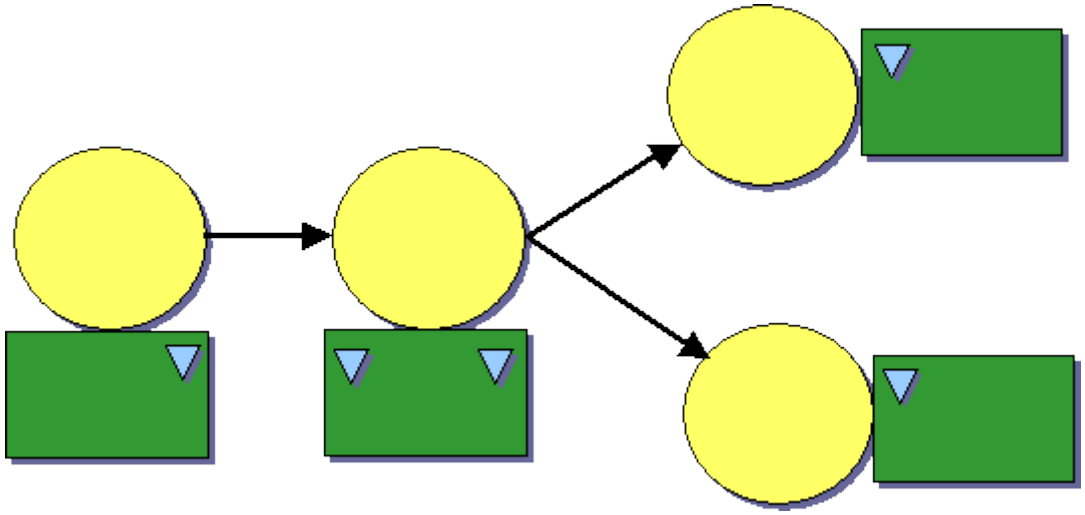
The most important tenet that must be followed by nodes is what's sometimes called the "black-box" principle: a node must compute its outputs using ONLY data from its inputs or incoming connections, i.e. it must NOT directly grab data from outside sources, like other nodes.



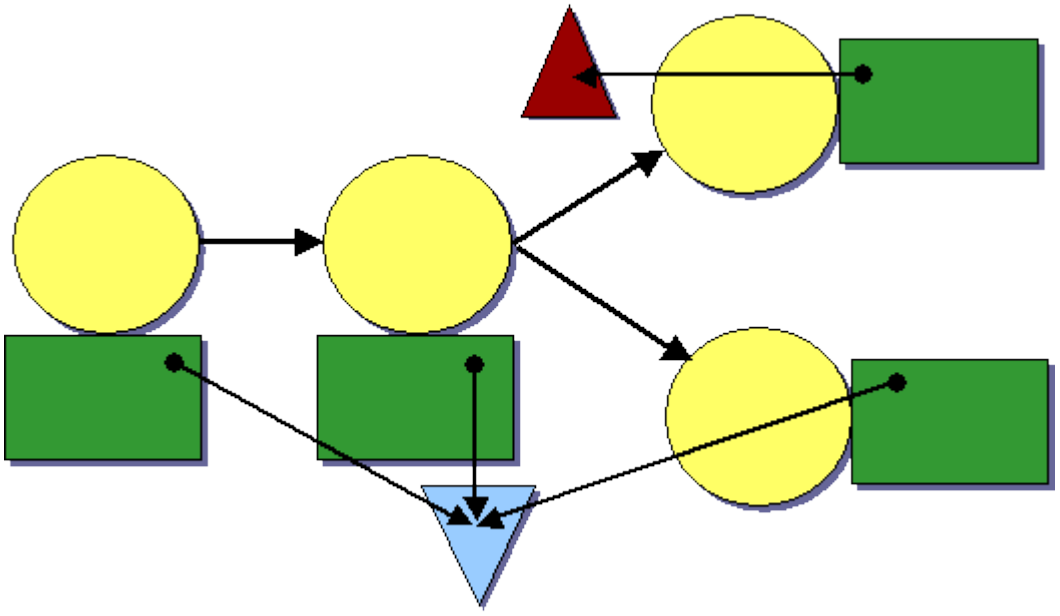
Suppose we have the above example, in which B tries to break this rule by getting data from D. When B's compute() is executing, D is not clean yet. By breaking the black-box principle, B created a cycle without the DG's knowledge, modified the state of data without waiting for the state of the graph itself to update, and finally used invalid data since D has not fully computed yet.

DG Data

Data in the DG is handled depending on the type. Light or numerical data is duplicated in each node's datablock and can be converted as needed (e.g. int to float).



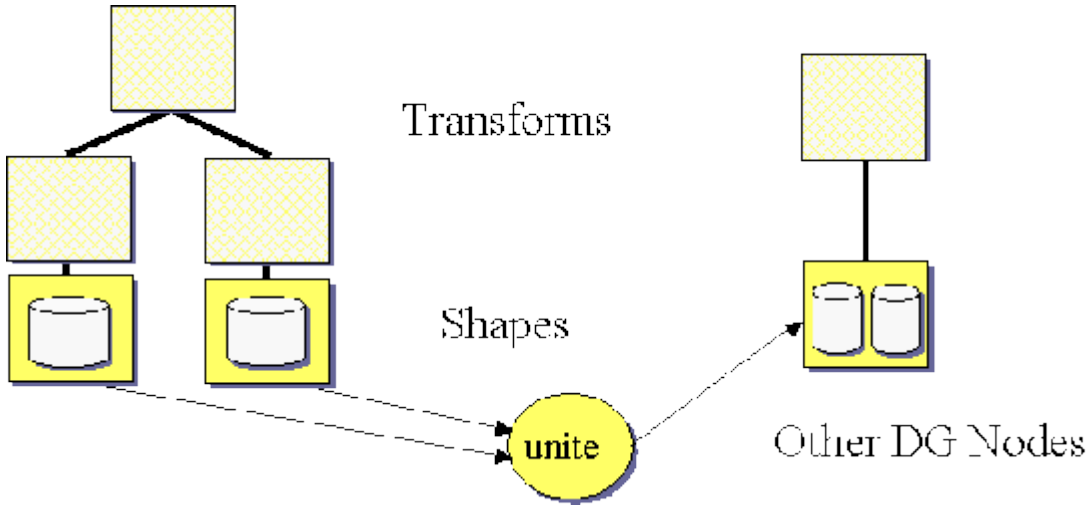
Heavy data, like meshes, curves, matrices and other complex types, is reference counted and duplicated only when one node wants to modify it apart from the rest.



DAG (Directed Acyclic Graph)

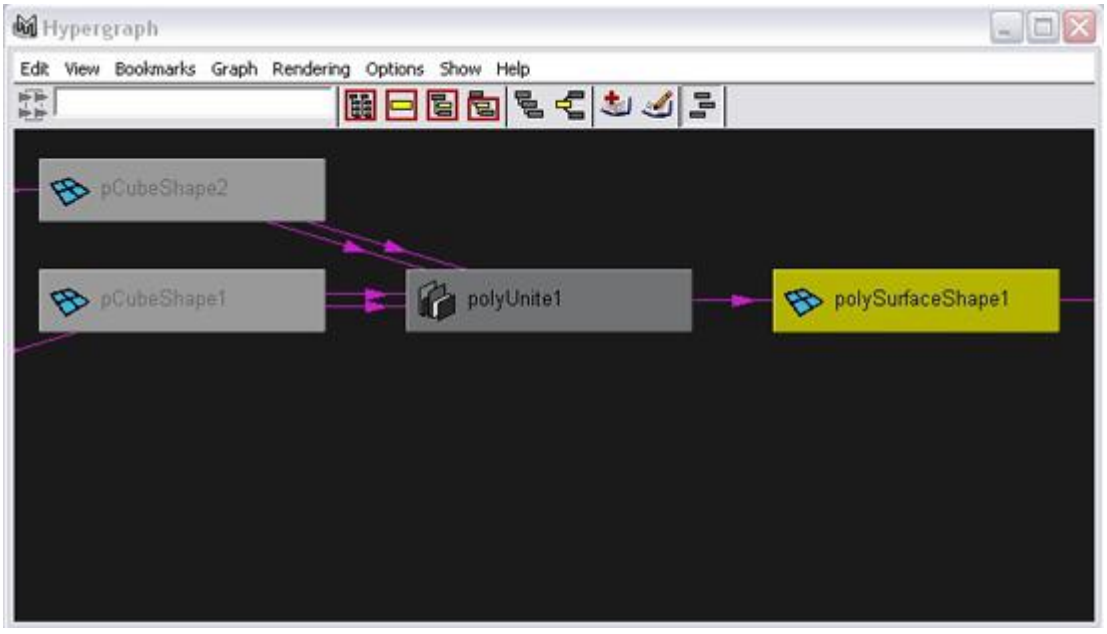
The DAG is a subset of the DG containing nodes that are capable of being visible and need to be in some hierarchy. The DAG consists of transformation nodes (transforms and joints), which help in positioning, orienting, sizing and grouping objects, and "shape" objects, which consist of geometry (polygon, NURBS, subdivision surface), lights, cameras, locators, particle emitters, etc.

DAG nodes have all the properties of regular DG nodes, with the exception of a parent-child relationship, which is special and is not represented as a DG connection.



Maya's Hypergraph

For Maya programmers who want to visualize the DG and the DAG, the Hypergraph can be an invaluable tool during development of scripts or plug-ins that use or manipulate the DG, e.g. make connections between nodes or traverse connections to find certain nodes. With the Hypergraph, you can, for example, observe how Maya nodes are typically connected to each other.



Additional Resources

The following book contains an explanation of the DG that is ideal for those just beginning to program in Maya using MEL or the API:

Gould, David A. D. *Complete Maya Programming: An Extensive Guide to MEL and the C++ API*. Morgan Kaufmann Publishers, 2003.