Title: Skinning in Maya at Industrial Light & Magic
Presenter: Andrea Maiolo, Industrial Light & Magic

1 Introduction

Industrial Light & Magic is a big structure with a lot of resources and a lot of employees. The energies dedicated to R&D and tool writing are big compared to most companies. This gives us sometime the ability to have more than one solution to solve a problem.

This document describes one of the toolset we developed in house. We choose to present it here because it covers few topics hopefully relevant and useful to most people. It will allow me to cover subjects I never seen discussed in any document and touch on some of the current standing Maya issues.
The toolset in question is a replacement to Smooth Bind and one of skinning solutions we have at ILM. We called it iSkin.

This presentation is divided it three parts.
First we will discuss the reasons and the goals we had while rewriting an existing tool.
Then we'll cover implementation, problems and solutions.
Finally we will go over in detail one of the biggest obstacle we found in this project, namely the generation of fast and efficient structures to store our work.

1.0 Rewriting an existing tool.

The challenges of rewriting an existing tool are several. It's not like implementing a new feature. It's more like building a car from scratch because that one we have has a flat tire and the steering wheel on the right side.

One thing I have learned in my tools writer career is that people are more willing to accept changes if features are initially only added to a toolset instead of been replaced. This way each artist can choose his own pace of discovery and acceptance of the new features while still leveraging his own skills and experience. Removing old functionality make them fill private of their safety net and forced to explore a world of quick sands.

For this reason the new skinning system needed to have at least the same features, speed, memory foot print and reliability of Maya original Smooth Bind. At the same time it had to incorporate some of the tools and functionalities artists were used to from older in-house toolsets and new functionalities production had requested. As it turned out, it hasn't been trivial to replicate some of Maya native component's characteristics and behaviors through the API.

1.1 Why we did it

As everybody can imagine we have been using geometry deformations for a while now (we started with “Terminator 2”). As anybody that worked for a company that has been around for a while can tell you, there are traditions, legacy code and standards that are hard to modify. Few years ago we started transitioning from our old pipeline that took us from “Dragonheart” to “Star Wars EpIII” to a new pipeline that, beside other things, was employing subdivision surfaces. At that time we didn't have good tools to envelope polygons and so we started to explore Maya native enveloping to see how far it would have been able to take us. Unfortunately Smooth Bind quickly collapsed under the weight of our history and its shortcoming.
This is the list of problems we identified been of greatest relevance and that we targeted to have fixed with iSkin. Some have been incorporated in Smooth Bind in the latest versions of Maya; others are still not present even in the most recent incarnation of the package:

- *Not particular methodology has to be used to envelope to general transformations.*

Maya skinning wants joints to envelope to. Only after an initial joint skinning is possible to add other nodes and remove bones. iSkin lets you envelope to any transformation at any time like other in house enveloping tools do.

- *Non destructive unbinding*

In Maya, unbinding the envelope is destructive. If you detach the skeleton, the enveloping is going to be lost if you do not save the weights first. In iSkin you can always attach and detach the skeleton by turning on and off the status of an enveloping flag. This makes re-adjust the position of the

enveloping nodes/bones pretty simple compared to the original Maya tool.

- *Non destructive normalization*

In Maya normalization is a destructive process. When the weights are normalized, the original work done by the artists is modified and then discarded. The outcome, because mostly automatic, rarely brings to the expected result and makes balancing weights really complicated. In iSkin, normalization is done on a copy of the data and only during transformations. This ensures that the original weight assignment doesn't get destroyed.

- *Visual feedback of enveloping decays (cylindrical manipulators)*

Maya enveloping has a unique parameter to control the influences decay called drop-off rate. It's a not really intuitive value that works in bone space and has not visual representation. iSkin implement cylindrical manipulators that have a visual representation. Eventually more elaborated manipulator types can be added.

- *Visual feedback of influence weights (colors)*

Maya Smooth Bind shows weight influence as shaded gray scale intensity or as color chosen from an 8 colors palette, but only one bone at the time. iSkin uses a 24bit shading mode independently by the status of the iSkin node.

- *Muscle and skin sliding*

iSkin supports a primitive version of muscle/skin sliding. Maya already has some of the functionality needed to implement such rig. Muscle and sliding can be realized using a combination of enveloped geometries with jiggle and sculpt deformers applied. The lack of control on the normalization and decay function though makes this approach slow and tedious. Using similar techniques with the advanced control offered by iSkin gives us the ability to envelope to manipulators that can jiggle, slide and maintain their volume.

- *Volume preservation*

iSkin incorporates a volume conservation algorithm. This is a clever extension of the polar coordinates system that we had in previous tools and it's used to preserve the volume of the enveloped mesh by preventing "candy wrapping".

- *Rest pose guarantee*

iSkin doesn't try to store meaningful information about the position the object was before to be enveloped (Bind Pose). The geometry will always go to the rest pose when the tool is set to "Rest Mode". It's responsibility of the user to reconfigure to rig to be appropriate for the geometry. This give the ability to switch and change the rest pose of the rig as many time as required. Rigs element can be added and removed without living behind old Bind Pose information not longer up to date.

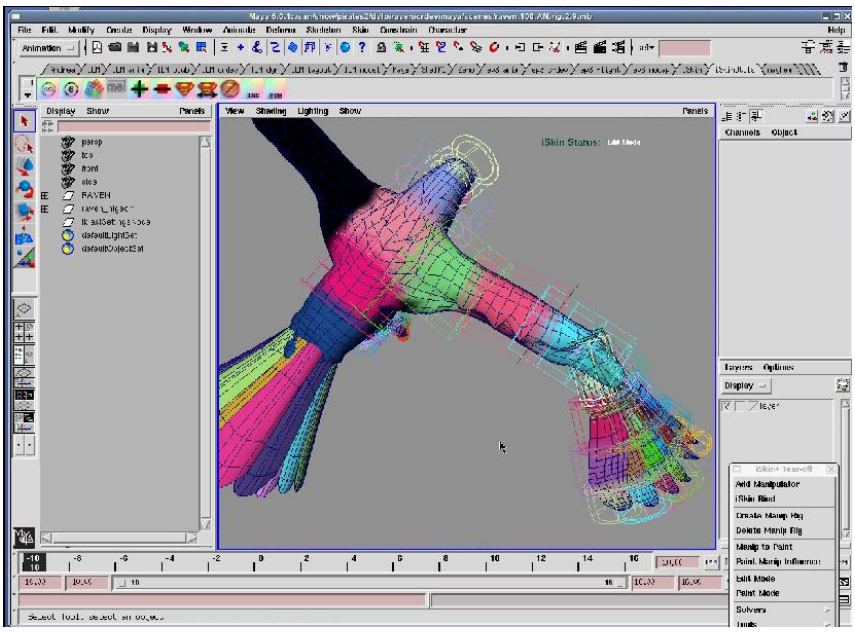

Figure1: *bird enveloped in iSkin.*

2 Implementation, Problems and Solutions

Like many production tools iSkin grew in an organic way following the needs of the artists. Priorities are continuously shifting to accommodate the requirements of shows, people and departments.

At ILM there are few groups entitled of writing, maintaining and designing tools. One is R&D and they generally do the very heavy lifting when it comes to software engineering. They are for instance the authors and support group for our in house 3D software Zeno. We have other teams of coders that are called the pipeline groups. They are composed by artists with very strong technical skills or in some case, by programmers with some strong artistic skill. They take care of the more production driven needs and tools. Enveloping felt quite suited for the creature development pipeline group because it didn't really require to come out with the next great thing or implement a new algorithm... even though we ended up doing just that.

2.1 Lots of MEL

One of the first things we were able to identify is that we had much more people able to recognize, write and debug MEL than Maya API. So we decided from the beginning to go against current. A

commercial package is mostly written in C++, with the few parts of MEL code embedded directly in the C executable. The main goal is to have an atomic module that can be easily shipped and installed.

We wrote as much MEL as possible. This allowed us to have more people able to identify scripts, modify them, extract part from them, fix bugs, and generally participate in the development of the toolset, especially technical directors. On top of that in our pipeline is much more immediate to distribute scripts than Plug-ins for production. Finally, scripts require less support. They travel between platforms and Maya versions with little or not support.

2.1.1 Flexible procedures with arguments

The basics of good code writing may sound obvious even to the less experienced software engineer, but for a lot of technical directors and people focused in production is not. Very often the only goal of productions is to complete a shot in less time possible. Tool writing is intended as a last resource to complete a specific task and it gets realized in a very rough and inflexible way. While we understand that is reasonable for some tool to follow this kind of paradigm, we quickly realized that modularity rimes often with flexibility. We wrote or rewrote all 50+ MEL commands to accept input, manage a standard error format, and provide outputs.

Finally, we incorporated a system to automatically generate documentation from the code using “Natural Docs”. This creates a web page that acts like an extension of Maya MEL and Node Reference docs. Anybody developing, expanding and modifying one of the iSkin tools has the ability to quickly search the list of existing commands, nodes, get their input output etc.

2.1.2 UI calling a main core engine

The interfaces as well got written as an input gatherer for a computing engine that could be used by other scripts or stand alone. Here are few examples.

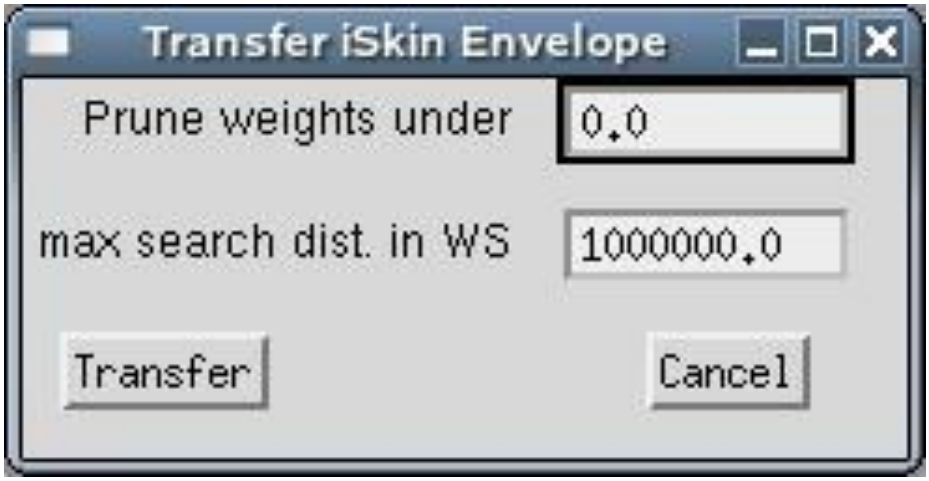

Figure2: *Transfer Envelope is just a UI to the transferiSkinEnvelope command Plug-in.*

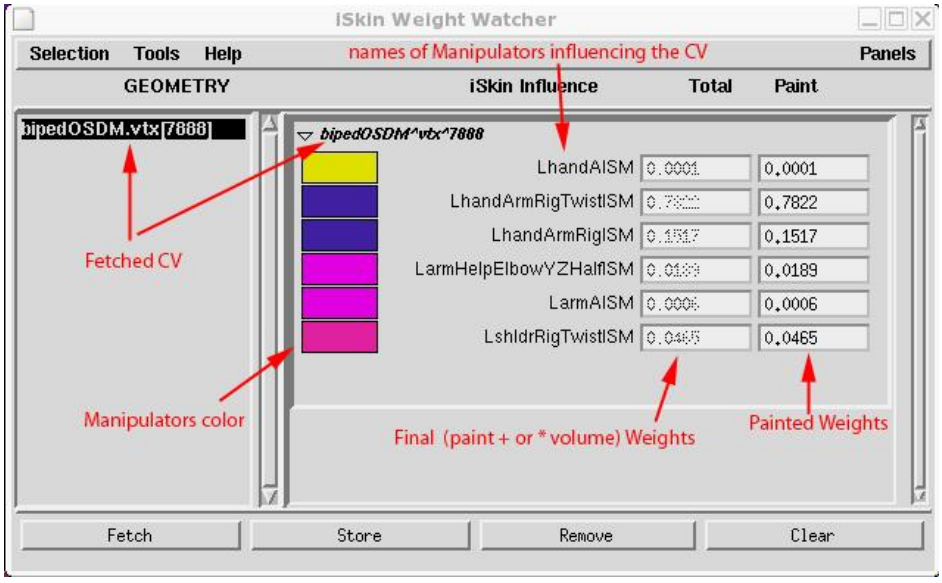

Figure3: *Weight watcher display weight gathered and set by the getiskinweight and setiskinweight command Plug-ins.*

Functions that turned out to be invoked in heavy for loops or be computationally intensive got written as a C++ commands. An example of the first situation are the set and get weights commands. Transferring enveloping is a good example of the second case.

2.2 Separated Plug-ins

One of the unconventional decisions we made on this project was not to package functionalities together. Mayatomr is a great example of the opposite scenario. Dozen of commands, nodes and other functionalities are grouped in one executable file.

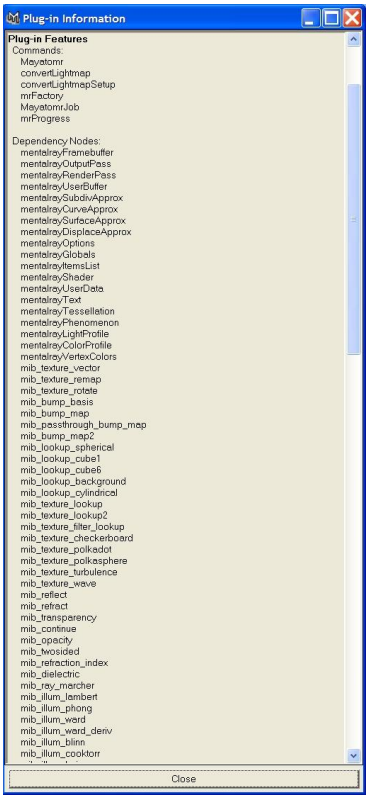

Figure3: *the info window of Mayatomr shows the list of functionalities that are packaged in this Plug-in.*

We have instead a command/node/translator per executable. The reason for this is just contingent to our pipeline. Because we don't have a need to transport files easily we had no problem to keep things separated.

The MEL commands that control the nodes creation, use commands or invoke translators make sure

that the required Plug-ins are loaded at the right time.
For instance these lines:

```
// load the set and get iskinweight Plug-in if it's not loaded
if (`pluginInfo -q -l setiskinweight` == 0) loadPlugin
"setiskinweight.so";
if (`pluginInfo -q -l getiskinweight` == 0) loadPlugin
"getiskinweight.so";
```

are presents in the Weight Watcher UI MEL code (Figure 3) to ensure that the Plug-ins to set and get weights are loaded.

This gave us for instance the flexibility to upgrade in stages or very easily roll back just certain parts of the package in case a problem was found.
The only annoying thing with this approach was that the command Plug-in has to be eval in the code because Maya can not use a command in the same scope where it got defined.
Luckily the limitation has been removed in Maya 8.5 and commands can be used directly even from the procedure that loaded them.

On top of this we have added a menu in Maya that select from a stable and beta version of the code.

This is the list of Plug-in created for iSkin:

Dependency Nodes:
iSkin
envColor
envShader
cylManipulator
maintainVol

Commands:
getiskinweight
setiskinweight
transferiSkinEnvelope
closestCVOnMesh

Translators:
readiskinenv
saveiskinenv

3 Attributes Data Structure and its mysteries

When we started implementing this toolset (or I should say when I started to implement this toolset, I don't want to blame anybody else for my mistakes...) I had the naive idea that it wasn't going to be a big deal. After all we weren't writing the next big thing. We were just doing a little bit of pipe cleaning, the skinning algorithm is one of the simplest to implement in the field of computer graphic. It turned out, as many of you probably already know, that writing a Maya Plug-in and writing an efficient Maya Plug-in that can handle a production model are two completely different stories. There are so many tricks that can be exploited to improve code efficiency that during every one of my several monitor head-banging session I wished they where actually written somewhere accessible to the common mortal. This and the following sections are mostly about this. All the little things it's good to know to have a better life as API programmer. So let's start with the mother of all evil, Maya attributes.

3.1 How to create attributes for different situations

The purpose of attributes on nodes is to describe the type of data the node owns. An attribute is not data on a node, only a description of the data the node may use. Choosing attributes requires a thorough understanding of what the node will be capable of, and what operations will be most useful for the intended users. Knowing the guru behind the attribute managing can save some desperate moment to the person in charge of creating this attribute. It's about this less documented last part that I would like to focus my attention. I will leave the rest of the attribute description to the excellent literature available in the Maya docs or presented in previous editions of the API Conference.

3.1.1 Compound attributes or not compound attributes

In Maya attributes can be nested in hierarchies or “compounded” so that they can be accessed as a larger entity or as individual components. The purpose of this is not only to allow you to keep your attributes organized but also to give the option of connecting or setting attributes at a higher level. An example of the usage of this is a “color” attribute, which will contain children attributes “red”, “green”, and “blue”. For example, in this case, you would have the option of connecting two “color” attributes together to make them identical, or only connect their “red” channels and let their green and blue channels operate independently.

A necessary addition to attributes is the ability to make them arrays, sometimes referred to as “multi”. Making an attribute into an array type tells the node that the data will not just be a single value, but will instead be a list of values of the given type. Sometimes, nodes display a combination of these two techniques. For instance the weights of a skinCluster node is store in an attribute that is both a compound attribute and an array. The “parent” attribute *weightList* is in fact an array and it has children attribute *weights* as a compound attribute. What gets almost everybody sooner or later is the fact that these kinds of attributes can not be connected with other nodes. So the first decision to make when creating attributes is if their data has to be accessed by some other node. Remember the rule

of the black box. Maya manual, documentation, and programmers insist that poking in other nodes data structure is not safe and make bad Maya programming. I will avoid entering in the debate about the validity of this statement here. A lot of worlds have and can be said about this subject alone. In the case of iSkin we wanted to have a node to computer the weights and a node to display them. For that reason we could not follow the same data structure of the skinCluster to save our weights. The trick we ended up using is the following. Instead of create a bidimensional data structure, we took the "matrix" columns and flatten it on the first row.

	Cv0	Cv1	Cv2
Transformation 0	Cv0-Trans0	Cv1-Trans0	Cv2-Trans0
Transformation 1	Cv0-Trans1	Cv1-Trans1	Cv2-Trans1
Transformation 2	Cv0-Trans2	Cv1-Trans2	Cv2-Trans2

Figure 4*: the image is a schematic representation of how the skinCluster weight organization looks like for a mesh with three vertices and three influencing transformations. The top row where Cv0, Cv1,Cv2 are is the index of the weightList attribute. Each one of the element in columns underneath are the element of the weights attribute.*

[Cv0-Trans0, Cv0-Trans1, Cv0-Trans2, Cv1-Trans0, Cv1-Trans1, Cv1-Trans2, Cv2-Trans0, Cv2-Trans1, Cv2-Trans2]

Figure 5: *This is how the skinCluster weights structure can be "flattened" to be able to pass through nodes.*

3.1.2 `MPlug` VS `MDataHandle`

As I already mentioned before, there is a big difference between writing a Plug-in and writing a GOOD Plug-in. Part of the difference is learning the crouched paths of Maya API secrets.

One of the first one I learned from a web page somewhere is the big efficiency difference between `MPlug` and `MDataHandle`. Both these structures offer access and management of nodes attributes, but there is a big difference in performance between the two. While an `MDataHandle` is a smart pointer into a data block a Plug and an attribute are in simple cases equivalent.

First of all when a Plug is “get” or “set” it always triggers the execution of the `compute()` function if the attribute is dirty. DataHandles can be set to be output or input. The difference between them is that an output handle doesn't require the attribute to be up to date and as a consequence, it doesn't trigger node computation. This can be useful in situation when we already know that the value we have is good and we want to avoid unnecessary computations.

Another situation when it's almost imperative to use DataHandles is in the creation of attributes arrays. MPlugs are easier and more intuitive to use. They manage the creation of the elements and the growth of arrays automatically, but they are slower, especially when used with logical indexing. I will go more in dept about array and `MDataHandle` later and discuss how these aren't speed demons neither.

These two methods of attribute management are of course not completely equivalent. The list of their functionality is slightly different but most notability, `MDataHandle` can be only used if the Plug-in has a `compute()` or `deform()` function. This mean that commands Plug-ins have to use Plugs and they can became very slow in case they have to manipulate arrays of attributes.

3.1.3 `MArrayDataBuilder`, what it is and why we need it

I want to spend few worlds to describe the `MArrayDataBuilder` because I noticed that especially beginners in Maya programming get confused about it and its usage in general is not very intuitive.

As I said in section 3.1.2 MPlugs offer a more intuitive and familiar way to create array attributes. For instance if I want to create an attribute that has one element for each cv of a mesh I can just keep calling `MPlug::elementByLogicalIndex(<element index>)` and Maya will create the attribute for me if it doesn't exists, sizing the array if necessary. DataHandles don't operate this way. To explain it in a very simple way they are incapable of creating the data. If data for an array attribute needs to be created internally of a node, then the attribute needs to be notified that a `DataBuilder` is going to create it for it using `MFnAttribute::usesArrayDataBuilder`. For instance:

```
MFnNumericAttribute numericAttr;
numericAttr = numericAttr.create ("SumWeight", "SumW",
MFnNumericData::kDouble, 1.0);
numericAttr.setArray(true);
numericAttr.setUsesArrayDataBuilder(true);
```

instruct Maya to use a DataBuilder to create information for the attribute `SumWeight`.

Then, in the code, new data will be added to the DataBuilder, not the attribute. Finally, when the code has finish with its work the attribute can be set to get the data from the builder. For instance:

```
invMatArrayData.set (invMatArrayBuilder);
```

3.2 How to structure a paintable attribute

How to construct and setup a paintable attributes has been covered already a couple of times during seminars in past API conferences. I want to cover this topic one more time because is another of those things that if you don't hear about around the campfire it is difficult to figure out and can be source of pain and frustration.

Let's start from the structure. Since Maya 6.0, an attribute to be paintable MUST be the child of a compound attribute. I think the reason behind it is the fact that Artisan need to know which surface is painting in case more than one surface is attached to the node the paintable attribute belongs to. The attribute can be `intArray`, `doubleArray`, `vectorArray`, `multiInteger`, `multiFloat`, `multiDouble` or `multiVector`. You will need to use a `MFnTypedAttribute` for the Array (intArray, doubleArray etc.) and a `MFnNumericAttribute` for the "multi" (multiFloat, multiInteger...). If they are "multi" you must use `MFnAttribute::setArray()` to store all vertices weight values. This is an example of a paintable data structure:

```
// add Manip paint weight attribute
MFnNumericAttribute Manp;
MApaint = Manp.create ("MApaint", "manp", MFnNumericData::kFloat, 0.0);
Manp.setHidden(true);
Manp.setStorable(false);
Manp.setArray(true);
Manp.setUsesArrayDataBuilder(true);

 // add the MApaintList Attribute. This is needed for the MApaint in order
to work
MFnCompoundAttribute cAttr;
MApaintList = cAttr.create("MApaintList", "manpl");
cAttr.addChild(MApaint);
cAttr.setHidden(true);
cAttr.setArray(true);
cAttr.setUsesArrayDataBuilder(true);
addAttribute (MapaintList);
```

Note that MApaint has been defined as `MFnNumericAttribute` of type float. If using a little more memory and disk space is not an issue it would be more efficient to use a `MFnTypedAttribute` of type `doubleArray`. It is always faster to use function set instead of iterator to acquire data. `MFnDoubleArrayData` would allow you to copy the entire array in a `MDoubleArray` instead of accessing to the values one element at the time.

Then you need to notify Maya that you indent to paint that attribute. This is done through the MEL command `makePaintable`. This function needs an attribute type, an attribute name and a node name to run. For instance this line makes the attribute MApaint defined above paintable.

```
// make the MApaint attribute of the iSkin node paintable
makePaintable -attrType "multiFloat" -sm "deformer" "iSkin" "MApaint";
```

`makePaintable` can be run in the initialize function of the node itself or from the command that creates the node and sets its connections. We choose the second option.

3.3 `MDataHandle`, a slow way to handle data

As I mentioned in the section 3.1.2 MDataHandles are faster than Plugs, but by no means they are a fast data structure. According to Autodesk support, the reason is the number of dependencies a `MDataHandle` class needs to go through to retrieve the needed information.

If performances are an important factor in your code then, you have to avoid to get data from Handles as much as possible. There are few loopholes you have to jump through to achieve this goal.

The first thing to avoid is to get data from handles in heavy for loop. It has been said in courses of past editions of conference. API wrapper classes can slow down the code. If for instance you need to get the same array attribute for each element of another array attribute, copy the first array in some local variable and then use that one when looping through the second attribute. Using skinning as an example, if for every Cv you need to loop though all the transformation matrices influencing it, loop though all the matrices once first and store them in some other structure, then you can loop through all the Cv and get the copy of the data. It's a kludgey and messy way of doing things, but in our tests it made a big difference.

The following example illustrate this technique

```
// loop through the trMatrices
for(unsigned int i = 0; i < num_joint; ++i)
{
  // gets the handle to the tr Matrix in the array and
  // assign it to the matrix var
  m_data = m_array_data.inputValue(&status);
  CHECK_MSTATUS( status );

  // store the value of the handle in a std::vector
  trmatrix = m_data.asMatrix();
```

```
  trMatrixArr.push_back(trmatrix);

  // initialize the arrays for the next loop
  m_array_data.next();
}

// loop through the Cvs
for(!iter.isDone(), iter.next())
{
  // loop through the trMatrices that influence that cv
  for(unsigned int i = 0; i < *manIdxListIter; ++i)
  {
    ...
    // calculate the contribution of the current
    // transformation for the cv
    accum += (cv * trMatrixArr[curr_joint] - cv) * currweight;
    ...
  }
}
```

Another important technique to reduce the amount of calls to `MDataHandle` is by using `setDependantDirty()`. I'm not going to go in detail about the use of this method, but the key idea is that `setDependantDirty()` give you a much finer control/feedback on which Plug got dirty.

In case of multi attributes if the code rely only on `attributeEffect()` to decide when the `compute()` function should be called, as soon as any of the array Plug get marked dirty, the entire `compute()` function needs to be run to recompute the outputs.

`setDependantDirty()` allow you to know exactly which plug go dirtying so that a way smarter and efficient `compute()` function can be written. For instance, in the case of iSkin only the weights related to the modified of painted transformation are updated, not the entire set of transformations.

3.4 Why we do not use `MPxData`

As you probably know, the Proxy Data (`MPxData`) class is the base class for user-defined Data types. All user-defined Data that is to be passed between Nodes in the DG must be derived from `MPxData`. `MPxData` transparently incorporates the common behavior and defines the common interface required for DG Data. Data objects of classes derived from `MPxData` are recognized by Maya the same as built-in DG Data types, but are able to implement user-specified behavior. Data of a user-defined type on a Node is accessed in the same way as intrinsic Data types. The Data is held in the Data Block (`MDataBlock`).

The big win of a `MPxData` is the ability to control the implementation and optimize it for your purpose.

There are two main reasons we are not using it much. One is because when a data structure is wrapped around `MPxData`, it ends up in the Data Block, with all the slow down that this will imply. Secondly, because you are defining a custom type, Maya will have a reference to this Plug-in when you save out the file even if the data has not been created yet. So if you try to unload the plug-in from the file with an unused node employing `MPxData`, you will be told that the plug-in is in use.

According to support, if memory is not an issue, the best performances are achieved by coping the data in the node and use some fast structure to do the calculations.

Finally it should be noted that the behavior of `MPxData` is such that the data ends up getting copied every time it is accessed through a Plug. If you embed all of your data inside `MPxData` this can become very expensive. Also, attributes are by default cached. This means that if you connect two attributes with `MPxData` then each end of the connection will by default hold a copy of the data.

3.5 `MDataHandle` VS STL demo Plug-in

As I said in the previous paragraph the best performances are achieved by coping data coming from the DataBlock in the Plug-in. In case it's needed to store long arrays, probably the best bet for efficiency is to use a `MFnTypedAttribute()` because it has method to retrieve and set data in one go. We will talk about the limitations of this class later.

When we were trying to figure out why our code was always very slow we started to suspect that the access to the DataBlock was the bottle neck, so we came out with a simple test-case to verify our hypothesis.

I wrote a node that first fill up an array with a certain amount of data and then read it back. When it creates the data it stores it in two structure. A `MFnNumericData()` attribute of type `kFloat` and a `std:: vector` of type float. An attribute in the node allow to read the data from the DataBlock or from the `vector`.

The performance increase of the `vector` over the attribute is in order of 10 times. Yes, to read the data from a `std::vector` is 10 times faster than from the DataBlock. The experiment was done without taking advantage of vector iterators. That should have give us a little extra speed boost on the std side.

3.6 The intricate nature of Maya files

The way attributes are saved in Maya file is surprisingly less straight forward than it could be expected. You need to be aware of the difference between binary and ASCII files as well as the differences between different type of attributes to create the most efficient and compact file possible.

All attributes are stored automatically when the file is saved. If the attribute doesn't need to be saved this need to be declared through `MFnAttribute::setStorable(false)`. Declaring attribute not storable can be useful to reduce file size and loading time when that attribute gets constantly recalculated from other attributes.

3.6.1 Maya ASCII files and attributes

Inspecting a Maya ASCII file is very simple. Any kind of test editor will do the trick. There are few things that are worth to be noticed. First of all default attributes don't get saved.

There is a difference on how the attributes are stored depending of the attribute type. The most commonly used attribute types are probably `MFnNumericAttribute()` and `MFnTypedAttribute()`.

If you have an array of attributes with value of 0 and the declared default value for the attribute is 0, nothing of that array will be saved in the Maya file.

If a `MFnTypedAttribute()` is used the .ma file will save every single element that is stored in the `MDataBlock` for that attribute. For instance, if you set a `MFnTypedAttribute()` to be of type `MDoubleArray()` and your double array is [0,1,2,3,4] the Maya file will have the [0,1,2,3,4] saved on it.

If the attribute is of type `MFnNumericAttribute()` it's still possible (and widely diffuse) to create an array of double out of it declaring `MFnAttribute::setArray()` for the attribute. If such type of attribute is saved in an ASCII file the values will be packed in group of 500 and for each group, values declared as default, will be skipped. These lines for instance create such type of attribute:

```
MFnNumericAttribute Cvw;
CVweight = CVw.create ("CVweight", "cvW", MFnNumericData::kDouble, 0.0);
Cvw.setArray(true);
```

If we pass the previous array [0,1,2,3,4] to it, only [1,2,3,4] will be saved in the Maya file. Notice though that this nice way to compress data is true only until the first not default data is found in the 500 attributes package. For instance [0,0,0,0,1] gets [1] saved, but [0,1,0,0,0] get [1,0,0,0] and even worst [1,0,0,0,0] gets [1,0,0,0,0] saved!

3.6.2 Maya binary files and attributes

All the complicated “compression rules” described for the ASCII files do not apply to binary. Binary files are intrinsically smaller, but in the case of array attributes saved with a lot of default values, they can became considerably bigger!

3.6.3 How to save attributes efficiently

So what is the best way to save data in the most compacted and efficient way if array attributes are used? The answer is surprisingly simple.

Don't put in the DataBlock what you don't want to appear in the file. This few lines for instance insure that the declared default value of 0 doesn't get put in the DataBlock of the attribute referred by `weightArrayBuilder`:

```
if (wevalue != 0.0)
{
  MDataHandle weightData = weightArrayBuilder.addElement(ElementIndex);
  double  & wevalue = weightData.asDouble();
}
```

A little more convolute alternative could be to overwrite `MFnAttribute::setInternal()` to have the function filter out default value when the `MArrayDataHandle` is set.

3.7 Read attributes from a newly open Maya file

One of the things that often confuse programmers first starting to work with Maya attributes is how they are created.

When an attribute is read from the disk it's default values will be skipped. In the example of the paintable attribute I presented in section 3.2, an attribute MApaint with a default value of 0 was created. If for instance we assume that the node containing this attribute is attached to a mesh with 100 Cvs, comes quite instinctive to assume that the attribute will have 100 elements. This is completely not guaranteed. If for instance the attribute is set as not storable, the array will have 0 elements when created.

If the attribute is of type array you can easily check the attribute size with `getAttr -s <node name>.<attribute name>`. If your code is making assumptions about an attribute to be a certain size, you can get in troubles when the file is saved and reopened. Interesting enough MEL shield the user quite nicely from this complexity. Both `getAttr` and `setAttr` create the attribute for you with the default value if the attributes do not exits.

The are a couple of easy things to do to check that the attribute you are looking for exists in the API as well. If you are using DataHandles the following code offer a solution:

```
// loop though all the possible elements of the array
for (unsigned int idx=0;idx < colorEle; ++idx)
{
  // if the first available element match the current index
  if (plugColorData.elementIndex() == idx)
  {
    //read the data
    MDataHandle colorData = plugColorData.inputValue (&stat);
    CHECK MSTATUS( stat );
    colorValuelist[idx] = colorData.asFloat3();
    // go to the next element of the sparse array
    plugColorData.next();
  }
}
```

In case `MPlug` is used it is a little easier. It is possible for instance to have a list of all the elements of an attribute array using `MPlug::getExistingArrayAttributeIndices()` or read the array elements with `MPlug::elementByLogicalIndex()`. The last one behaves like the `getAttr` MEL command and it will create an attribute with the default value if the array element is missing.

Another alternative would be to use a `setAttr -s <array size> <node name>.<attribute name>`. The array will be grown and all the missing element filled up with default values.

Finally, notice that `MFnTypedAttribute()` get treated by Maya monolithically. If for instance you do a `setAttr -s <array size> <node name>.<attribute name>` on a `MFnTypedAttribute()` of type `MFnDoubleArrayData::kDoubleArray` the result is going to be 1 even if the array is full. Notice also that there is no way to avoid to save default values from a `kDoubleArray`.

It is important to highlight the fact that `MPxData` gets manipulated from a `MFnTypedAttribute()` as well, and from here some of the pro and cons that we have discussed in the section 3.4 dedicated to `MPxData()`.

3.7.1 `jumpToElement()` and `jumpToArrayElement()`

Let's begin this chapter by quickly recap what we have discovered about attributes so far.

If we use a `MFnTypedAttribute()` to store our values we will get every single element of the array. For instance, if we use a `doubleArray` even if we only save one value at index 3000 we need to size the array to have 3000 elements, and this will create default values up to 2999 that are going to be put in the DataBlock and consecutively on file. The data is going to be copied in every node that is connected to the attribute. A lot of attention will be necessary in setting the `MFnAttribute::isCached()` status of the attribute to avoid wasteful data duplications with node connection and attribute querying thought commands.

If we use array of attributes (multi attributes) and we are careful not to put in the DataBlock default values, we will get array that nicely compress but we will lose the ability to manipulate the array as a big chunk of data.

If we build our own data Type with an `MPxData()` attribute we can get the best of the two worlds, but your Plug-in is going to be hard to unload after a save, even if it hasn't been used. If this is not a problem then this could be a route to follow.

For the skinning toolset we decided to use arrays of attributes, so the problem we had to solve is how to get the best performance out of the very very slow attribute access. One of the worst possible scenario is when a random access is needed.

The `MArrayDataHandle::jumpToElement()` method is the slowest in the `MArrayDataHandle` class and can easily bring your node to its knee. The main reason is that it performs a linear search of the array every time the command is invoked. Using `MArrayDataHandle::jumpToElement()` to go through all the elements of an array is a very bad idea!

There is a new method that has been added since version 8.0 (I like to think because I kept complaining about `jumpToElement()` been slow). It is called `MArrayDataHandle::jumpToArrayElement()` and it doesn't require a search, since the array indices are non-sparse. `MArrayDataHandle::elementIndex()` method can always be used to determine the logical index related to the current array index. This new method makes jumping through attribute arrays a less painful experience.

3.7.2 A faster alternative to get an array element

As I said in the previous paragraph `jumpToElement()` is not fast at all so it is a good rule to stay away from it.

Although, if you are dealing with a sparse array, one of the fist ideas that pop to mind to check if an element exists is to use `jumpToElement()` to jump at every possible element of the array and

check for the status of the method. If the method fails the element is not present, if it success we can proceed reading it.

An alternative would be to use a `MPlug` to get the list of existing indices, and then use `jumpToElement()` to get only the existing value. This would speed things a little because the loop to retrieve the elements is shorted, but the speed increase would be mitigated by the fact that the access to Plugs is quite show.

A solution could be to do what I did in paragraph (3.7) dedicated to read the attribute from a node loaded from a file. If we know the maximum possible size of the array, we can loop through all the elements and check if the current `elementIndex()` of the attribute match the index in the loop. If it does we read it and we use `MArrayDataHandle::next()` to point the handle to the next existing element of the array.

This method is way faster than using `jumpToElement()` to retrieve existing indices and its faster than `MPlug::getExistingArrayAttributeIndices()` + `jumpToElement()` even in case the array is heavily sparse.

3.8 Read and Set attributes from a command

If access to an attribute is required through a command, unfortunately DataHandles are not available. If we have decided to use array of attributes to store our big data, then things became quite slow. The only possible thing to do is to get or set one element at the time using something like this:

```
// assign the weight to CVPAINT
for (unsigned int g=0; g<weightBuffer.length() ; ++g)
{
  MPlugCVpaintElem = DestinCVpaintPlug.elementByLogicalIndex(g,&status);
  CHECK_MSTATUS( status );
  status = MPlugCVpaintElem.setValue(weightBuffer[g]);
  CHECK_MSTATUS( status );
}
```

3.8.1 A way to speed up `MPlug` operations.

There are few things that can be done when forced to set one element at the time and use `MPlug()` at the same time. The first one is to make sure the array that we are setting has been grown to the proper size. There isn't an API command to resize arrays, but `setAttr` called by `MGlobal::ExecuteCommand()` can do the trick. For instance:

```
MString MELsetMApaint = ( "setAttr -s " + MapaintSize + " " +
```

```
iSkinNode.name() + ".MApaintList[0].MApaint");
  status = MGlobal::executeCommand(MELsetMApaint);
  CHECK_MSTATUS( status );
```

set the size of the attribute MApaint to a size of MapaintSize.

In this way Maya doesn't have to continuously grow the array one element at the time.

The other operation that can have an impact on performances is the order in which the attributes get saved. Setting them consecutively instead of jumping around the elements can dramatically improve the performance of the command.

Finally, in case the attribute is getting set on a newly create node that is not connected yet to other nodes, can be beneficial to set the attribute before to make connections to the rest of the graph. The reason for this is that every time an attribute is getting set through a Plug, dirty bit propagation is triggered and the computation of other nodes can trickle down the graph. If for instance 3000 attributes are set individually through a `MPlug::setValue()`, each one of the nodes depending for their values from the set one will be recalculated 3000 times.

3.9 Conclusion

We examined some of the problems related on implementing an already existing tool for production. Artists are expecting all the functionalities and performance they are used to plus all the bonus features they requested.

We discussed the reasons and motivations that convinced us to go in this direction.

Finally, we spend quite a bit of time analyzing the best way to efficiently manage big quantity of data like they could be weights of an skinned creature. We highlighted pros and cons of every solution offered by the package.

For anybody interested to explore more methods and techniques behind the graph, node evaluation and attribute structure I sincerely suggest to start from the intro offered by Barbara Balents in “Introduction to the Maya API” and continue with the excellent “Efficient coding within the Maya API” that Rasmus Tamstorf presented at the API Conference in 2005. Finally, Tracy Narine “Solution Sessions” are always full of tricks and techniques to write a faster and more efficient code.