



WHAT'S NEXT
GDC:06

Advanced Tool Writing for Character TD's

Judd Simantov
Naughty Dog Inc.

Game Developers
Conference

Table of Contents:

- **Tool Writing Fundamentals.**
- **Maya specific development:**
 - **MEL .vs API**
- **MEL:**
 - **Custom Viewport Pop-ups.**
 - **OptionVar.**
 - **Recursive Functions.**
 - **Custom Pick-Walk.**
 - **ScriptJobs.**
 - **Arrays passed by reference.**
- **MEL UI:**
 - **Implementing Drag and Drop.**
 - **Defining and Using Templates.**
- **API:**
 - **Introduction to the Maya API.**
 - **API Commands:**
 - **Closest Point on Mesh.**
 - **Shape Snap.**
 - **API Nodes:**
 - **Poly Face Node.**
 - **API Locators:**
 - **Poly Face Locator.**
- **Conclusion.**

Tool Writing Fundamentals:

Planning and Design:

Often when people start out developing tools, they tend to overlook the importance of planning before actually starting any coding. As the project or tool becomes bigger and more complex, planning becomes increasingly important. Here are some pointers to take into consideration before actually coding.

Write things down:

It's very important to write things out on paper first. It's impossible to maintain all the dependencies and possible scenarios in your head. Also by writing things down and seeing the bigger picture, it makes it easier to anticipate potential problems. You'll also find the actual coding part is a lot easier and faster when you have a blueprint to follow.

Get input from user:

Before writing a tool always acknowledge whom you are writing the tool for and get their opinion on how they intend to use it and if they will even use it at all. User input is even important for certain things that would almost seem trivial. Such things can include the naming and placement of interface elements or the look of manipulators and custom locators.

Work responsibly:

Always keep in mind that you are working in a production environment and the longer it takes you to get a tool up and running, the more it's going to cost the company. It's important that you build a solid tool that fulfills all the necessary expectations, however you should be wary of trying to implement too many bells and whistles. Also sometimes adding too many features and making a tool too "technical" can be overwhelming for user(s). Tools also tend to go through several changes during its lifespan and making them too complex can make change increasingly difficult.

It's important that you first lock down the essentials and allow people to start testing the tool. Then at a later stage revisit the "bells and whistles". Getting input from the user(s) and finding out what are the essentials comes into play here once again. In production, saving time is incredibly important and writing tools that cater to every possibility can definitely come back to haunt you.



Coding Essentials:

There are certain universal “coding” rules that you should try to abide by. For the most part they will hopefully make your life as well as your colleague’s lives a lot easier both in the short term and long term. I will outline the few that I think are of the highest priority.

Commenting:

Commenting code at first seems like a complete waste of time. You wrote the code and therefore you will have no problem understanding it. Wrong! Spending several minutes a day to comment your code appropriately can save both you and your colleagues days of troubleshooting and aggravation. If you are going to change someone else’s code make sure you comment your changes and specify your name. This way if there are any problems or modifications needed, the other developers can consult with you.

Try to make comments practical and useful. Translating the code into a somewhat grammatical English format is pretty useless to other developers, as they would be able to gather the same information from just reading the code. Use comments in areas that are complex and would require an overall breakdown as well as to explain things that could be potentially ambiguous.

```
.  
//transform point back into local space.   
resultPt = resultPt * geomMatrix.inverse();  
  
.   
//let resultPt equal itself multiplied by the .  
//the geometry's inverse matrix. .  
resultPt = resultPt * geomMatrix.inverse();
```

Modularity:

When writing tools, try to break things up into functions that might be called several times or that is a big chunk of the overall operation. There are several major advantages in modularity, some include not having to rewrite code unnecessarily, your code becomes a lot more manageable when trying to modify it, when changes are made in one place they will propagate through the rest of the program if implemented correctly and it also helps to understand the code when revisiting it at a later stage. Keeping your code modular is important, but it can also be overdone. Try to find a good medium between modularity and practicality.

Maya Specific Development:

Maya has two primary tool development languages available to the user. Maya Embedded Language (MEL), which is Maya's native scripting language, as well as the Maya Application Programming Interface (API), which allows you to develop your own custom plug-ins. The next section will discuss both these languages and their application in relation to tool development.

Having the tools to write the tools:

Writing efficient tools not only requires a problem-solving mind but also requires that the developer knows what tools the language has to offer and also understands how to use these tools to their benefit. If you don't know any of the MEL commands and don't take the time to browse the documentation, it will make your life a nightmare. I can't emphasize enough how much you can learn from reading the documentation.

Maya's API is a C++ API. If you want to develop plug-ins with the API, you must have a good knowledge of C++ and object oriented programming first and foremost. The API is just a set of C++ classes that gives you a whole bunch of function calls that you can use as you choose to. It's important to understand that the API does not give you access to Maya's core implementation; rather it is just an interface to the implementation and therefore you only have access to that which Maya grants you.

Primary differences between MEL and the API:

First and foremost, MEL is a scripting language and the API is a C++ interface to Maya functionality. As most people already know, MEL is a lot easier to learn and apply than the API is. MEL is more procedural (a series of procedures/functions that are in the scope of a program) and the API is object oriented (program is comprised of objects/units that make up the program and it's functionality). Both explanations given above are very concise and these concepts are out of the scope of this talk, however it would definitely be worthwhile looking up more elaborate definitions on both of these programming paradigms and familiarizing yourself with their differences.

MEL does not need to be compiled and plug-ins do. Therefore, MEL is executed directly in Maya and the API requires an external compiler such as Microsoft Visual Studio .NET which will then compile a ".dll" plug-in file that is loaded into Maya. The API also offers you more access to internal Maya functionality. One of the other major advantages to using a compiled plug-in versus an executable MEL script is that plug-ins are significantly faster, especially when dealing with large iterative processes and complex algorithms. One major disadvantage of using the API is that it is operating system dependant and version dependant. MEL however, for the most part is not and although sometimes there might be some discrepancies it usually holds up fine between

versions and operating systems.

MEL can be called from the API using the **MGlobal** class and the **executeCommand()** function. The function does return the results from the MEL command that you can then use in your plug-in. A quick example is shown below.

```
MString sphereName; .
MGlobal::executeCommand("sphere -n \"mySphere\";", sphereName);
cout << "sphere name: " << sphereName.asChar() << endl;|.
```

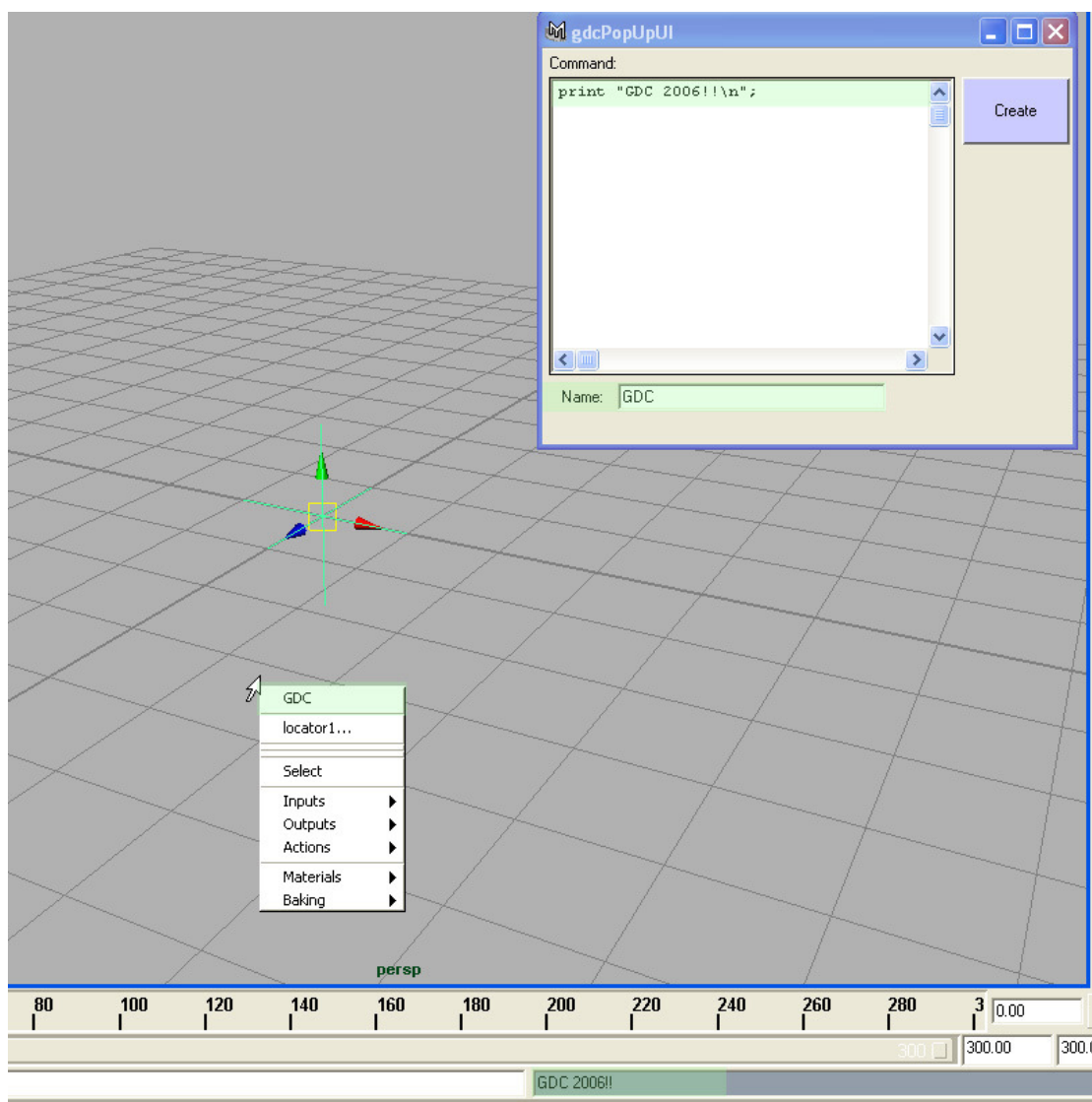
Conclusion:

Developing good tools is more than just having a great idea and writing it down on paper. You need to have the appropriate knowledge of certain aspects in order to write the tools in the most efficient and practical manner. Having a solid programming foundation as well as a good understanding of the respective language is essential. What loops does MEL support? Can you return arrays? What data types does it support...etc? Once you have gained a solid foundation in MEL and C++ you will start to delve into more complex areas such as data structures, algorithms...etc. The entire process is progressive, so it's not to say that you shouldn't get started until you've mastered a 900 page C++ book, but just keep in mind that the more you understand about the language and the basics, the more capable you will be of writing solid tools.

Maya Embedded Language (MEL):

Custom Viewport Pop-ups

Creating pop-ups for UI elements is pretty straightforward; however, creating them in the viewport and eliciting different results depending on the object under the mouse pointer is slightly more complicated. We will also then tag objects with an attribute that will hold a command. This way you can give each object a specific command that is called when the menu item is clicked.



First off you will need to edit one of Maya's MEL scripts, which is called **dagMenuProc.mel**. This file can be found in:

../Alias/Maya6.5/scripts/others/dagMenuProc.mel - Maya 6.5

../Alias/Maya7.0/scripts/others/dagMenuProc.mel - Maya 7.0

In this file you will find a global procedure called **dagMenuProc(string \$parent, string \$object);**

This procedure is called when you right click in the viewport and it is passed two very important arguments. The first argument is named **\$parent** and the second is named **\$object**. **\$parent** is basically the parent menu you that you will be attaching your **menuItems** to and **\$object** is the object that is under the mouse pointer.

All we are going to do is add one of our procedures to the **dagMenuProc()** and pass our procedure the **\$parent** and **\$object** arguments, so it can process it with whatever operations we decide.

```
// This has been called because a menu press happened on a DAG object.
// So find the Object submenu and add the DAG specific items to it..
//.
global proc dagMenuProc(string $parent, string $object).
{.
    string $mode = "All";.

    if (`optionVar -exists currentMenuBarTab`) {.
        $mode = `optionVar -q currentMenuBarTab`;
    } else {.
        optionVar -sv currentMenuBarTab $mode;.
    }.

    if (`popupMenu -e -exists $parent`) {.
        setParent -m $parent;.

        gdcAddPopUpMenu $parent $object;.

        // label the object.
        string $shortName = `substitute ".*" $object ""`;
        menuItem -l ($shortName + "...") -c ("showEditor "+$object);.
        menuItem -divider true;.
        menuItem -divider true;.

        // Create the list of selection masks.
        createSelectMenuItems($parent, $object);.
    }.
}
```

The **gdcAddPopUpMenu()** procedure will basically add the **menuItem** to the **\$parent** menu. The **menuItem** added will contain a specific command **gdcReadPopUpCmd()** that will read a string off the attribute on the object and execute the command. The overall idea is to give each object it's own specific

command that is attached to the object, so when the **dagMenuProc()** procedure gives us the object name under the mouse pointer, we can check that object for it's command attribute and execute whichever command is associated with the object.

```
global proc gdcReadPopUpCmd(string $name, string $object)
{
    //list attributes with _gdcCmd suffix
    //these attributes are read in and executed as commands
    string $attrs[] = `listAttr -st "_gdcCmd" $object`;
    string $attr;

    if (!size($attrs))
        return;

    for ($attr in $attrs)
    {
        //tokenize out the suffix, so name of command
        //can be isolated
        string $token[];
        tokenize $attr "_" $token;

        //if the name of the attribute
        //doesn't match the name passed
        //to the procedure move to the
        //next attribute.
        if ($token[0] != $name)
            continue;

        //get the string attribute on the object
        //and evaluate it.
        eval(`getAttr ($object+"."+ $attr)`);
    }
}
```

As you'll see in the code below, the first portion of the code list's all the attributes with a **_gdcCmd** suffix. This is a naming convention giving to the attribute and used to detect popup commands on an object, therefore preventing clashes with any other custom attributes. It's not important what suffix you use here as long as it's unique and consistent. The prefix part of the attribute name will then be used as the menu name. So when you tokenize (split up) the attribute name, the first part will be passed as the name of the command and the second part will be used to identify the attribute as a popup command attribute.

```

global proc gdcAddPopUpMenu(string $parent, string $object)
{
    //list attributes with _gdcCmd suffix
    //these attributes are read in and executed as commands
    string $attrs[] = `listAttr -st "+_gdcCmd" $object`;
    string $attr;

    for ($attr in $attrs)
    {
        string $token[];

        //tokenize out the suffix, so name of command
        //can be isolated
        tokenize $attr "_" $token;

        //pass the name and the object to the gdcReadPopUpCmd
        menuItem -p $parent -l ($token[0])
            -c ("gdcReadPopUpCmd " + $token[0] + " " + $object);
    }

    //create a divider to make things more readable
    if (size($attrs))
        menuItem -p $parent -d 1;
}

```

Because of the function used to derive the name of the command, the name must only contain alphabetical characters and cannot contain any spaces or underscores. This is a pretty easily solvable problem, however I will be keeping examples as simple as possible.

Below is the basic procedure that adds the attribute and the command to the object. This process is made a lot easier through the interface I have supplied, but it gives you a look under the hood at what is going on. The full MEL script for this is called ***gdcCreatePopUpAttr.mel*** and is supplied with this paper.

```

global proc gdcAddPopUpAttrProc(string $object, string $name, string $command)
{
    //If the attribute does not exist
    //add the attribute and set it's
    //string to the command string.
    string $attr = ($name+"_gdcCmd");
    if (!`attributeExists $attr $object`)
        addAttr -dt "string" -ln $attr $object;

    setAttr -type "string" ($object + "." + $attr) $command;
}

```

Using customized pop-ups that are associated with objects can have many advantages and allow animators to access MEL functions without the need of a bulky interface that takes up screen real estate. Some common examples of pop-ups that have proven to be useful, is for toggling visibility of object groups or to launch interfaces that are associated with specific objects. The possibilities are really endless.

What is an optionVar and how to use it:

The **optionVar** is something that I'm sure most people know about, but it's so useful that it's worth mentioning for those that might not. When wanting to store information on a per-user basis that is consistent through Maya sessions and scenes, **optionVars** are ideal. Without **optionVars** you will most likely write the information out to ASCII files and read them in accordingly. An **optionVar** works similarly to global variables, with the exception that they are consistent through Maya sessions. So if you close Maya and re-open it, your **optionVar** is maintained. An **optionVar** can also be used as an array, so you can pass one **optionVar** multiple entries and return them to an array data type.

Below are some examples of **optionVar** being used. Check the Maya Documentation for all its supported commands.

```
//creates optionVar named "gdcVar"
//and sets optionVar string to "entry"
optionVar -sv "gdcVar" "entry";

//queries the optionVar
optionVar -q "gdcVar";
// Result: entry //

optionVar -rm "gdcVar";

//now if you query the optionVar
//after removing it, it will return 0
optionVar -q "gdcVar";
// Result: 0 //
```

The most useful place for **optionVars** is to store UI preferences, although with time you'll probably find hundreds of creative uses for it.

Recursive Directory Search:

Often when writing tools that need to search a directory or it's sub-directories for files, recursively searching through directories can be challenging. This is a very short section explaining how to accomplish this.

The following procedure will take a path and file extension as arguments and return a string array that contains the full path for any file that has the given extension.

```
global proc string[] gdcFindFiles(string $path, string $ext)
{
    //make sure path is valid
    if (!`filetest -d $path`)
        error "no valid path passed to procedure!";
    //make sure extension is valid
    if ($ext == "")
        error "no valid file extension passed to procedure!";

    //intialize folders array with
    //the $path variable passed to the
    //procedure
    string $folders[] = { $path };

    int $i = 0;
    //loop the path in the $folders array
    //if a folder is found in the path
    //add it to the $folders array and
    //keep looping until there are no more
    //folders
    for (; $i < size($folders); $i++)
    {
        string $files[] = `getFileList -fld $folders[$i]`;
        string $file;
        for ($file in $files)
            if (`filetest -d ($folders[$i] + $file)`)
                $folders[size($folders)] = ($folders[$i] + $file + "/");
    }

    //variable to hold all the files that will be returned
    string $return_files[];
    string $folder;
    for ($folder in $folders)
    {
        string $files[] = `getFileList -fld $folder -fs ("*. " + $ext)`;
        string $file;
        for ($file in $files)
            if (!`filetest -d ($folder + $file)`)
                $return_files[size($return_files)] = ($folder + $file);
    }

    return $return_files;
}
```

The most important piece of code here is the first loop that keeps adding folders. This loop is basically saying start looping the **\$folders** array and keep looping it until the loop reaches the end. On every iteration of the loop, get all the files and folders in the directory, then filter out files by testing for directories only and if a directory is found append the found directory string with the current directory string and add it to the **\$folders** array. Now the loop will repeat this step until there are no more directories added. This is the foundation for any recursive looping and as you can see once you've written it and grasped the concept, it's pretty straightforward.

Custom Pick Walk:

Maya by default has built in pick walk that walks the hierarchy of nodes when using the up and down keys or cycles through the DAG (Directed Acyclic Graph) when using the left and right keys. Both of these really don't serve much purpose to animators, especially when you consider the complexity of modern rig setups and the hierarchies. The solution is to establish your own relationship between objects and define your own pick walk script. The best way to handle the relationship between the object and its target is to use message attributes and then read the connections between the objects. This way if the names change the connection is maintained and there are no hard-coded names.

The example I'll give will be pretty basic and will serve as the foundation for the implementation. Take this and extend on it to create a pick walk system that works best for your pipeline.

We will only implement "up" and "down" which will walk up and down a defined control hierarchy, however you should also implement "left" and "right" which will walk between left and right of corresponding controls. This is my suggestion on how to use it, but you are free to define whatever relationship works best for you.

The procedure below basically takes three arguments; the direction, the source object and the target object. As you can see there are a lot of error checks and existence checks. These checks are time consuming initially but will save you huge amounts of time in the long run when things error out and you are trying to debug the problem.

```

global proc gdcSetupPickWalk( string $direction, string $source, string $target)
{
    //validate all arguments
    if ((!`objExists $source`) ||
        (!`objExists $target`) ||
        ($direction == ""))
        error "one of the arguments is either invalid or does not exist!";

    string $source_attr;
    string $target_attr;

    //set the attribute based on the
    //direction passed
    switch ($direction)
    {
        case "up":
            $source_attr = "gdc_pickUp";
            $target_attr = "gdc_pickUpInput";
            break;
        case "down":
            $source_attr = "gdc_pickDown";
            $target_attr = "gdc_pickDownInput";
            break;
    }

    if ($source_attr == "")
        error "check to make sure direction argument is a valid direction!";

    if (!`attributeExists $source_attr $source`)
        addAttr -ln $source_attr -at "message" $source;

    if (!`attributeExists $target_attr $target`)
        addAttr -ln $target_attr -at "message" $target;

    string $source_conn = ($source + "." + $source_attr);
    string $target_conn = ($target + "." + $target_attr);

    if (!`isConnected -iuc $source_conn $target_conn`)
        connectAttr -f $source_conn $target_conn;

}

```

The next procedure is what actually reads the connection between objects and walks the connection to pick the corresponding object. This function will be placed under a hotkey with the direction as the argument for that specific hotkey. E.g. **gdcReadPickWalk "up"**;

```

global proc gdcReadPickWalk( string $direction )
{

    string $sl[] = `ls -sl`;
    string $obj = $sl[0];
    string $attr;

    switch ($direction)
    {
        case "up":
            $attr = "gdc_pickUp";
            break;
        case "down":
            $attr = "gdc_pickDown";
            break;
    }

    if (!`attributeExists $attr $obj`)
        return;

    string $conns[] = `listConnections -d 1 -s 0 ($obj + "." + $attr)`;
    if (size($conns) > 1)
        error ($obj + "." + $attr + " is output to more than 1 object.");

    //select the associated pick target
    select -r $conns[0];

}

```

This is the general foundation for pick walk and will be used to demonstrate script jobs in the next example. I urge you to take this as a “base” for developing something that works best for you. For my own full pledged system I’ve integrated additive pick walk selection, destination switchers to allow for multiple targets that are switched between based on a condition or set driven key, as well as a few other nice features.

Script Jobs:

Script Jobs are becoming more commonly used and primarily serve as a monitoring mechanism that will execute a script based on a certain condition. The list of conditions is extremely extensive, but can be found in the documentation. An example where a script job is often used is to execute a command when the selection changes. This way you can execute different scripts depending on the object selected. As with the pop-up example, the best way to tag objects for different results is using custom defined attributes.

This example used will make use of the pick walk implementation. As you pick walk through objects the control curve for that specific object will become visible and the rest will become invisible. I am not recommending this as a way to set up your rigs, but merely using it as the foundation for an example of a script job implementation. Hopefully the concept will spark some much more useful and applicable ideas.

The way the script works is you select a group of controls that you wish to apply the functionality of the script job to. The way the script job will identify these controls is by tagging each control with a Boolean attribute called “**gdc_sjVisCtrl**”. Once all the objects have been tagged, the script job will be initialized. Script jobs return an integer value, which serves as an identification number known as a “job number” and can be used to edit or delete existing script jobs. The first flag we pass is the **-event** flag. This flag takes two arguments; the first is the event that should trigger the script. In our example that event is called “**SelectionChanged**”. You can get a complete list of events either by checking the MEL command documentation or by executing the following script in the command line:

scriptJob -listEvents;

The second argument passed to the “**-event**” flag is the name of the procedure that should be executed when the selection changes. We also use the “**-killWithScene**” flag, which will basically delete the script job when the scene closes. If you do not specify this flag the script job will continue to run in memory until Maya is closed. There are many cases where you want the script job to not be scene specific, so you would just remove the “**-killWithScene**” flag.

```
global proc gdcInitVisScriptJob()
{
    //add attribute to all selected objects
    //so toggleVisibility recognizes control
    string $sl[] = `ls -sl;
    string $s;
    for ($s in $sl)
        if (!attributeExists "gdc_sjVisCtrl" $s)
            addAttr -at "bool" -ln "gdc_sjVisCtrl" $s;

    //start scriptJob
    int $id = `scriptJob -event "SelectionChanged" "gdcToggleVisibility" -killWithScene`;
}
```

Next we’ll review the procedure **gdcToggleVisibility()** that is called by the script job whenever the user’s selection changes.

As you can see the procedure is fairly self-explanatory, so I won’t go into too much detail. The procedure will query the current selection and check if it is tagged with the “**gdc_sjVisCtrl**” attribute. If it is, the control’s visibility will be set to **1**. The procedure will then loop all **DAG** nodes in the scene that contain the “**gdc_sjVisCtrl**” attribute and set their visibility to **0**.

```

global proc gdcToggleVisibility()
{

    //get the selected object
    string $sl[] = `ls -sl`;
    if (!size($sl))
        return;

    //intialize objects
    string $obj = $sl[0];
    string $nodes[] = `ls -dag`;

    string $attr = "gdc_sjVisCtrl";

    //if the attribute has the visibility attribute
    if (`attributeExists $attr $obj`)
        setAttr ($obj + ".v") 1;

    //loop all those not selected and hide them
    string $node;
    for ($node in $nodes)
    {
        if ($node == $obj)
            continue;

        if (!`attributeExists $attr $node`)
            continue;
        setAttr ($node + ".v") 0;
    }
}

```

Arrays are passed by reference in MEL:

This title might seem a little confusing to people that don't have much programming experience and don't really deal with the memory management of their declared variables in a programming language such as C++.

Essentially there are two ways to pass a variable to a function, the first is by value and the second is by reference. When you pass a variable by value, a copy of the variable is created when you enter the scope of the function and then deleted when you exit the scope of the function. When you pass a variable by reference, you are passing the variable's memory address and therefore the actual variable that you are passing to the function will be used in the function. What this also means is that if the value is modified in the function it will stay modified when exiting the function. If passed by value, the copied variable will be modified, so your original variable will not change. So what is the advantage of this you

might ask? Well the most obvious advantages are:

- A. If you want to pass big arrays of data, it's a lot cheaper to pass an address than to make a duplicate of all the data.
- B. You can pass multiple data types to a function and modify them inside the function instead of having to return different data types in different function calls. Or returning everything is a string and casting it back to a numerical data type.

Picture a scenario where you want to query a group of object's names, translate values and visibility values in one function and return the results. You could store all this information in one big string array and loop in large increments, type casting each string, but this makes the code that much more convoluted and slower by having to cast each string to a float and integer. So the next example will tackle this problem by passing arrays as reference. The real big advantage here is that a name is a string, translate values are floats and visibility values are integers, we are able to return all these data types in one function call.

First we declare our array variables. At this point they are obviously empty and don't contain any values. We call the **gdcPrintInfo()** function and pass all the arrays. This function basically just prints out the array values to the script editor output panel. When you execute the script you'll notice all the arrays are empty.

```
global proc gdcQueryObjects()
{
    //initialize arrays that vary in data type
    string $name[];
    float $translateX[], $translateY[], $translateZ[];
    int $visibility[];

    //print the arrays before they have been passed to gdcQueryInfo function
    print "Before: \n";
    gdcPrintInfo( $name, $translateX, $translateY, $translateZ, $visibility );

    //function that queries object info
    gdcQueryInfo( $name, $translateX, $translateY, $translateZ, $visibility );

    //print the arrays after they have been passed to gdcQueryInfo function
    print "After: \n";
    gdcPrintInfo( $name, $translateX, $translateY, $translateZ, $visibility );
}
```

The arrays are printed right after their declaration and are shown to be empty, then they are passed to the **gdcQueryInfo()** function. The **gdcQueryInfo()** function will query the selected objects for their names, translate values and visibility values. Each array is appended with the corresponding name, translates and visibility. By passing the size of the array as the array index you can dynamically append an array. Once this function is called the results are passed to the **gdcPrintInfo()** function to display that the arrays have been filled with the appropriate object information.

```

global proc gdcQueryInfo( string $name[], float $translateX[],
                        float $translateY[], float $translateZ[], int $visibility[] )
{
    //get a list of selected objects
    string $objects[] = `ls -sl`;
    string $object;

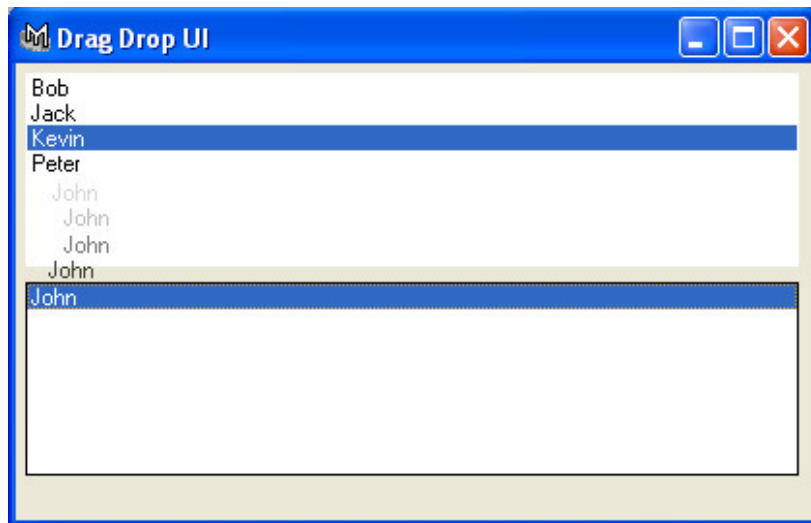
    //loop the objects and pass each objects
    //values to the arrays
    for ($object in $objects)
    {
        $name[size($name)] = $object;
        $translateX[size($translateX)] = `getAttr ($object + ".translateX")`;
        $translateY[size($translateY)] = `getAttr ($object + ".translateY")`;
        $translateZ[size($translateZ)] = `getAttr ($object + ".translateZ")`;
        $visibility[size($visibility)] = `getAttr ($object + ".visibility")`;
    }
}

```

Being able to populate arrays inside of functions has some very important advantages and can really keep code more manageable and increase speed and efficiency during script execution.

MEL User interfaces:

Implementing “Drag and Drop” support:



Drag and Drop support is something that is implemented in most modern applications. Most MEL UI elements support drag and drop callbacks. Often people deal with the issue of transferring objects in a list by having buttons that transfer it from one side to another. Although this approach works, dragging and dropping is more natural and intuitive.

In order to implement drag and drop support you need to register a callback function, this function will be executed when an element either receives a drag or drop on it, depending on which flag you pass the callback to. The element will inherently pass the function certain arguments and you can then use these arguments in your function to do as you please.

The example will include a window with two lists; the first list will be populated with some names that can be dragged back and forth between the two lists. This will demonstrate the functionality and implementation for drag and drop. **Note: Maya uses the “middle” mouse button to drag items.**

The first step is to set the **–dragCallback** flag and **–dropCallback** flag of your element control. You can check the documentation to verify whether or not an element control supports these flags. In the example below we implement both flags in two **textScrollList** elements. The flag takes one argument, which is the name of the function. In our case the drag function is called **gdc_DragCB** and the drop function is called **gdc_DropCB**. Next we will cover the implementation of these functions.

```

//create lists to store names and pass flags for
//drag and drop callbacks
textScrollList -ams 1 -dcc "gdcFOMOpenProc" -dragCallback "gdc_DragCB"
                    -dropCallback "gdc_DropCB"
                    -w 400 -h 100 gdc_A_TSL;
textScrollList -e -a "Bob" -a "John" -a "Jack" -a "Kevin" -a "Peter" gdc_A_TSL;

textScrollList -ams 1 -dcc "gdcFOMOpenProc" -dragCallback "gdc_DragCB"
                    -dropCallback "gdc_DropCB"
                    -w 400 -h 100 gdc_B_TSL;

```

The callback functions will take certain arguments that are by default passed to the function by the element control. For the drag callback it will be passed the name of the element control and the x and y position of the mouse pointer relative to the UI when the middle mouse button was first clicked. So if you click in the top-left corner of your UI, these values should be 0, 0.

The last argument is an integer called **\$mods**, this integer will determine whether or not a button modifier was pressed, such as “ctrl” and “shift”. The **\$mods** is represented as: **0 = No modifiers, 1 = SHIFT, 2 = CTRL, 3 = CTRL + SHIFT**. This will allow you to add things if shift is pressed and remove things if control is pressed, or any other functionality that you would like to incorporate. The drag callback returns a string array. This string array is passed to the drop callback function so you can essentially pass any information from the drag callback to the drop callback.

The drop callback will take a few more arguments. The first is the control from which the objects were dragged. The second argument is the control that objects are being dropped on. The third is a string array called **\$msgs[]**. This array basically holds the strings that you returned in the drag function as mentioned above. At the end of the drag function we returned the **\$list[]** array, this array is now passed to the drop function as the **\$msgs[]** array. We will then use this array to determine the list of selected objects at the time of dragging. The **\$x** and **\$y** variables are the UI position of the mouse at the time of dropping the objects.

The drop function will basically go through the list of objects passed in the **\$msgs[]** array and remove it from the “drag control” and append it to the “drop control”, resulting in a transfer of the selected items.

```

//this is the drag callback and will just print
//that currently dragged content
global proc string[] gdc_DragCB( string $dragControl,
                                int $x, int $y, int $mods )
{
    string $list[] = `textScrollList -q -si $dragControl`;
    string $item;
    for ($item in $list)
        print ("dragging: " + $item + "\n");

    return $list;
}

//this is the drop callback and will
//remove the dragged content from one
//list to another
global proc gdc_DropCB(string $dragControl, string $dropControl,
                      string $msgs[], int $x, int $y, int $type )
{
    string $item;
    for ($item in $msgs)
    {
        textScrollList -e -ri $item $dragControl;
        textScrollList -e -a $item $dropControl;
    }
}

```

By allowing the ability to inherently pass information and return information between the drag and drop function you are able to pretty much handle the implementation in any way you please. This makes for very intuitive workflows with respect to moving objects in UI's. Almost all elements have support for drag and drop flags.

Defining and Using Templates:

Often when developing MEL user interfaces, certain control elements have consistent attributes that can become tedious when coding complex and large interfaces that re-use the same element type. Defining and using templates helps make this process slightly less painful. The first part of this process is obviously to define the template.

Most UI elements have a “**-defineTemplate**” flag that is used to define which **uiTemplate** should hold the template information. We start by initializing the **uiTemplate** that will hold the template information. In our case the name of the **uiTemplate** is “**gdcTemplate**”. Once we have initialized the **uiTemplate** we start defining templates for specific control elements. In order to define the template information, you make a call to the control and use the “**-defineTemplate**” flag. You then pass all the necessary information that should be stored. We define a button template which aligns the text to the left, adds a “**gdcButton**” annotation, sets the width to 100, sets the height to 20 and finally sets the background color to a light red color.

```
global proc gdcTemplateUI()
{
    /*****
    //define template

    if (!`uiTemplate -exists gdcTemplate`)
        uiTemplate gdcTemplate;

    button -defineTemplate "gdcTemplate" -al "left" -ann "gdcButton" -w 100 -h 20 -bgc 0.9 0.6 0.6;

    //end template definition
    *****/

    //intialize window
    string $window = "gdcTemplateUI";
    if (`window -ex $window`)
        deleteUI $window;
    window -t "gdc Template UI" -w 123 -h 135 $window;

    //intialize layout to hold elements
    columnLayout -co "left" 7 -rs 3 gdcTemplate_main_CL;

    //create button and use dgcTemplate for each button except the last one
    button -l "Select" -useTemplate "gdcTemplate" gdcTemplate_select_B;
    button -l "Delete" -useTemplate "gdcTemplate" gdcTemplate_delete_B;
    button -l "Quit" -useTemplate "gdcTemplate" gdcTemplate_quit_B;
    button -l "Default" gdcTemplate_default_B;

    showWindow $window;
}
```

Once the template has been defined we create our control elements in the same manner as usual, however the “**-useTemplate**” flag is added and passed the “**gdcTemplate**” uiTemplate that we defined earlier in the script. Essentially what this does is tells the button to use the defined characteristics of the button in the “**gdcTemplate**”. We create three buttons that use the template and a fourth button that doesn’t. The result is three buttons that are red, wide and aligned to the left, while the fourth one is gray, fits exactly to the size of the text and the text is aligned in the center.



Using templates can definitely save you lots of time and keep your code a lot more manageable. If for any reason the color, width, height, font...etc of a UI element has to change, you will only need to change it in one place as apposed to a making the change for every occurrence of the button.

Maya's API (Application Programming Interface):

Many technical Maya users are proficient with MEL and are able to accomplish and solve many production tasks on an everyday basis only using MEL. As powerful and capable as MEL is, it does have certain drawbacks and depending on the problem at hand, it may not be as capable or ideal as the Maya API. When I first started delving into the API, I was really struggling to figure out what the advantages were and where it would be applicable. I'm going to try and demonstrate both the advantages and applications for the API. The API is a C++ API and therefore does require that you have a fairly good understanding of C++ and object oriented programming. Obviously the better your C++ knowledge the more stable, optimized and extensive your plug-ins will be.

One of the noticeable advantages of using the API over MEL is that in most cases the API is faster. One reason for plug-ins being faster is that they are compiled and linked and therefore optimized for the target platform. The result of this is that your plug-ins will be platform dependant and version dependant. Because the API is a C++ API, you can include external C++ libraries and exploit their functionality within your plug-ins. Also, for things such as Deformers, Custom openGL Locators and Custom Manipulators, you have no access to these features through MEL.

There are several different types of plug-ins you can write using Maya's API and some are used in conjunction with others. The plug-in types available are:

- **Commands**
- **Nodes**
- **Deformers**
- **Locators**
- **Manipulators**
- **Contexts**
- **Solvers**
- **Shaders**
- **Fields**
- **Emitters**
- **File Translators**
- **Shapes**

Due to time constraints I will not cover all of the above and primarily focus on commands, nodes and locators.

API Introduction:

I will start off with an overview of some of the fundamental concepts that can help you get started with writing plug-ins. A lot of the information will be sparse and concise to fit with in the time limits of the talk and also to outline broader concepts without getting lost in the details. I urge you to explore these concepts further and try to gain a better understanding of them in your own time.

All the classes in the API have prefixes that help distinguish their core application. I will go over these prefixes so that when you are looking through the API class documentation, the functionality of the classes make more sense. All handles to data and wrappers have an **M** prefix. E.g. **MObject**, **MDataHandle**, **MDataBlock**. All function sets that allow you to work on top of the data have an **MFn** prefix. E.g. **MFnSkinCluster**, **MFnDependencyNode**, **MFnSingleIndexedComponent**. All proxy classes that are derived from, have a **MPx** prefix. E.g. **MPxNode**, **MPxCommand**, **MPxDeformerNode**. Lastly, all iterators have a **MIt** prefix. E.g. **MItSelectionList**, **MItMeshVertex**, **MItGeometry**. Being aware of this will help you when trying to find classes and functions to assist in your plug-in development.

Error Checking:

Error checking is an important aspect of any programming or scripting language. Maya offers the **MStatus** class to handle error checking within the API and for the sake of consistency you should use **MStatus** for error checking in your own functions and classes. Most function calls in the API either take a pointer to an instance of **MStatus** or they return an instance of **MStatus**. After you have called a function and either passed or returned an **MStatus**, you can check its status code to determine whether the function call was successful or a failure. There are several other status codes that I will not cover but they are documented. The two most common are **kSuccess** and **kFailure**. Based on the status code, you can determine the rest of your program flow and easily report errors and successes. The more error checking you implement the less time you spend debugging simple mistakes.

MGlobal:

MGlobal is a very commonly used class and for this reason deserves some special attention. **MGlobal** allows you to perform common tasks that you will find yourself needing to do fairly often. Some examples of this are: Getting a selection list by name, setting the current frame, executing a MEL command from within an API function call and outputting messages and errors to the script editor. These are just a few of the useful functions that **MGlobal** offers you.

MGlobal will be used throughout the examples, however you should take the time to look through the documentation and just familiarize yourself with the functions that **MGlobal** has to offer. You will probably find yourself using this class more often than not.

Commands:

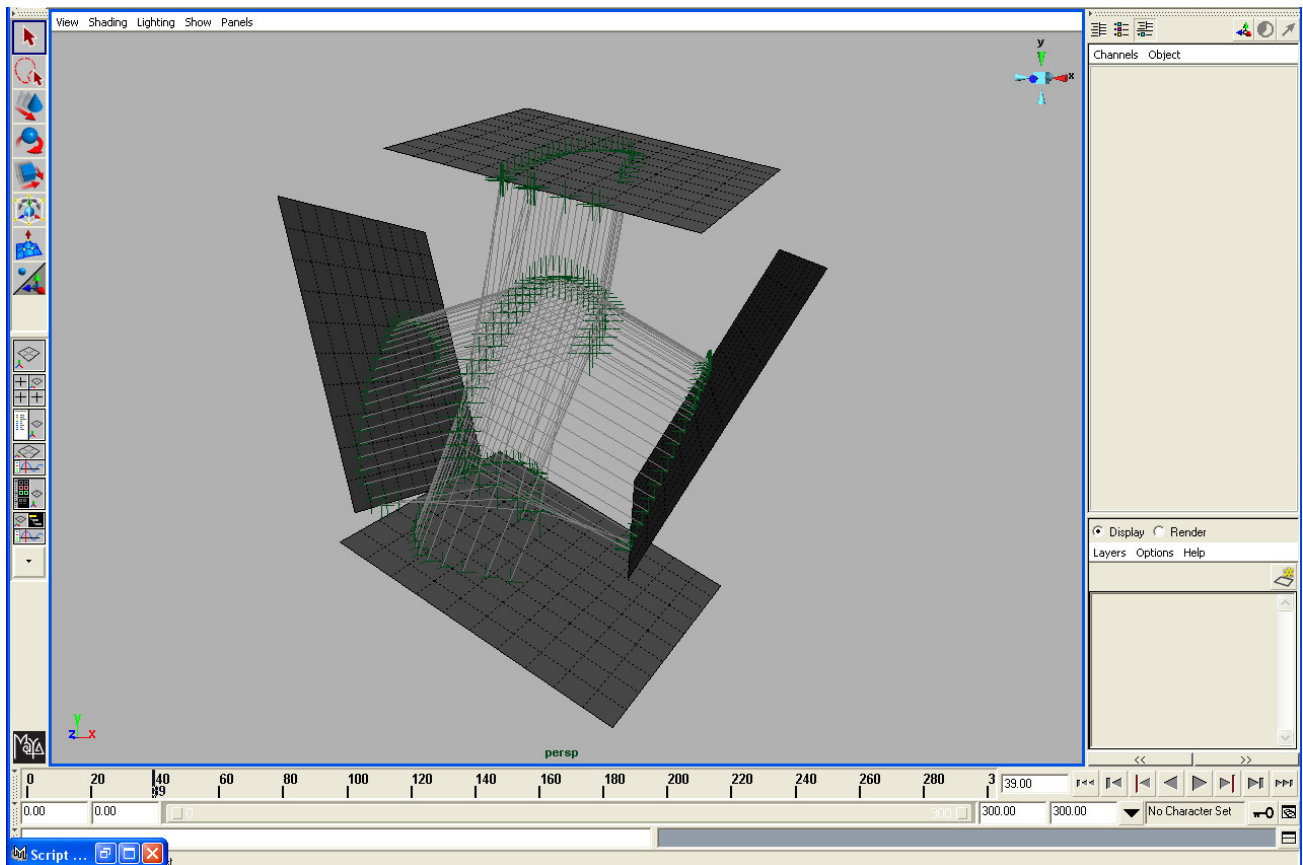
Writing your own MEL Command Plug-ins:

Every MEL command that you use when writing MEL procedures is essentially an API command. Maya has an exhaustive list of MEL commands that it supplies to you, however there are many times that you will need to write your own commands to perform operations that aren't available in the supplied MEL commands. Although you can often combine MEL commands and create procedures to accomplish these tasks, speed as well as not having the ability to derive off external libraries such as extensive math libraries can become an issue.

When writing commands you will derive off of a base class called **MPxCommand**. There are many member functions associated with this class that you don't need to pay attention to, but in order for your command to function correctly, the one function you do need is the **dolt()** function. This function is essentially where all your command data will be stored and executed. If you wish to add undo support to your plug-in you will need to look into the **redolt()** and **undolt()** functions. When adding undo support the design of your plugin will need careful pre-planning and will require more work. Unfortunately this is unavoidable. When adding undo support you will structure your plugin in such a way that you query and store all your initial information in the **dolt()** function and then perform the actual operations in the **redolt()** function. You will also want to store any information necessary for undoing in the **dolt()** function. When you call **undolt()**, you are essentially restoring the scene to it's state prior to the execution of your command, therefore you will have to store the appropriate initial state information in order to revert to it. Undo is only implemented when you change something in the scene and want to revert the changes, there is no need to implement undo in a command that is going to query something in the scene.

An example that uses undo and redo will follow the closest point on mesh command.

Closest Point on Mesh Command:



The first command we are going to look at is a command that will find the closest point on a polygon mesh to the point passed to the command. The idea behind this command is to illustrate the overall workflow in writing a command, I will keep it simple and to the point. The next example will be a command that is more applicable and will incorporate undo functionality.

Before getting into the details of the command, included in each project will be a **pluginMain.cpp** file. This file is where we initialize the plug-in and register all the necessary commands, nodes, deformers...etc.

This file will need to include the headers for the command, node or anything else you wish to register; hence we have included the “**gdcClosestPointCmd.h**” file. If you don’t include this header, then when the **plugin.registerCommand()** function is called and passed **CClosestPoint::creator** and **CClosestPoint::newSyntax** the compiler will have no idea what these classes or functions are.

An instance of **MfnPlugin** is created, which is what will show up in the plugin manager in Maya and the command is registered with the plugin using **MFnPlugin::registerCommand()**; If you were to include a node in this plugin as well, you would then have a **plugin.registerNode()** call in the **initializePlugin**

function as well as the **plugin.registerCommand()**. You will see an instance of this in later examples. Also note that if you do not have any flags associated with your command or you are not handling flags and arguments through **MSyntax**, you do not have to register the **newSyntax()** function with your plugin.

```
#include "gdcClosestPointCmd.h"

#include <maya/MFnPlugin.h>

MStatus initializePlugin( MObject obj )
{
    MStatus    status;
    MFnPlugin plugin( obj, "Judd Simantov", "1.0", "Any");

    status = plugin.registerCommand( "gdcClosestPoint", CClosestPoint::creator, CClosestPoint::newSyntax );
    if (!status) {
        status.perror("registerCommand");
        return status;
    }

    return status;
}
```

The next function in the file is **uninitializePlugin()** and an instance of **MFnPlugin** is initialized again. The **deregisterCommand** function is called and passed the name of the command that was the first argument in the **registerCommand** function.

```
MStatus uninitializePlugin( MObject obj )
{
    MStatus    status;
    MFnPlugin plugin( obj );

    status = plugin.deregisterCommand( "gdcClosestPoint" );
    if (!status) {
        status.perror("deregisterCommand");
        return status;
    }

    return status;
}
```

When it comes to writing a simple command that will not incorporate undo and redo functionality, but will incorporate flags with arguments, there is a few functions that are necessary for the command to operate. These functions are the **creator()** function, the **dolt()** function and the **newSyntax()** function. The creator

function will create an instance of your class and return it. The **newSyntax()** function will add all the flags to an instance of **MSyntax** specifying the long name, short name and the data type. Both the **creator()** and **newSyntax()** functions need to be registered with the plug-in as shown earlier.

```
void* CClosestPoint::creator()
{
    return new CClosestPoint();
}

//function handles the adding of flags to the command
MSyntax CClosestPoint::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag( pointXFlag, pointXLongFlag, MSyntax::kDouble );
    syntax.addFlag( pointYFlag, pointYLongFlag, MSyntax::kDouble );
    syntax.addFlag( pointZFlag, pointZLongFlag, MSyntax::kDouble );

    return syntax;
}
```

The flag names are passed to the **syntax.addFlag()** function as a string of characters. They are declared as global variables at the top of the .cpp file. The syntax looks like this:

```
const char *pointXFlag = "-px";
```

Check the source file for a more applicable example.

The **dolt()** function in this example is where all the work is done. There is some basic error checking that I will not cover, but I do suggest understanding and implementing error checks where potential errors may occur.

The first part of the **dolt()** function initializes the point that will store the positions passed to the command. It is initialized at the origin as a default, in case no flags are passed. Then an instance of **MArgDatabase** is created called **argData**. **MArgDatabase** is a function set that takes a syntax function as an argument and a **MArgList** as an argument, it then handles the retrieving of flag information for you. You can check if the flag is set, and if it is, you get the flag argument. The last argument of the **getFlagArgument()** function is the ".x" or [0] index element of the **MPoint** point we declared earlier. It will assign the flag argument value to the **point.x** variable and do the same for **point.y** and **point.z**.

```

MStatus CClosestPoint::doIt( const MArgList &args )
{
    MStatus stat = MS::kSuccess;

    //intialize point to origin
    MPoint point( MPoint::origin );

    //retrieve flag arguments and pass to point
    MArgDatabase argData( syntax(), args, &stat );

    if (argData.isFlagSet( pointXFlag ))
        argData.getFlagArgument( pointXFlag, 0, point.x );

    if (argData.isFlagSet( pointYFlag ))
        argData.getFlagArgument( pointYFlag, 0, point.y );

    if (argData.isFlagSet( pointZFlag ))
        argData.getFlagArgument( pointZFlag, 0, point.z );
}

```

The next piece of code is probably something you will use very often. In MEL the line of code: **`string $sl[] = `ls -sl`;`** is probably very familiar to you and something you use all the time. In the API, an **MSelectionList** will store the selection information in a similar way to how the string array in the MEL example stores it. You will pass the selected information to the **MSelectionList** by using **MGlobal::getActiveSelectionList()**. This will get any selected objects in the scene and pass them to the list. We then create a **MItSelectionList** object which is an iterator that will give us added flexibility over standard “for loops” in looping the **MSelectionList** container. It is not 100% necessary that you use an iterator to loop the list. You could achieve the same results using a “for loop”, but it is definitely more convenient and practical as well as being faster. The iterator also allows you to filter out any objects that are unwanted. In our instance we are filtering out anything that is not a poly mesh.

```

//get the active selection - this will be the selected meshes
MSelectionList selectionList;
stat = MGlobal::getActiveSelectionList(selectionList);
if (!stat)
{
    MGlobal::displayError("Error getting active selection!");
    return MS::kFailure;
}

//intialize iterator to iterate over selection and filter
//anything but poly meshes
MItSelectionList iterList( selectionList, MFn::kMesh, &stat );
if (!stat)
{
    MGlobal::displayError("Error intializing selection iterator!");
    return MS::kFailure;
}

```

Finally we get to the actual core functionality of the command. In this next piece of code the iterator loop is entered and this allows us to apply the same functionality to as many objects as the user has selected. Through the iterator **iterList** we receive a handle to the selected objects dag path. In Maya the dag path is the full path to the object including the hierarchy in the path. The reason for this is that Maya does allow objects of the same name to exist in a scene provided they are in different hierarchies. Using full dag path names will always ensure you are not selecting the wrong object or causing your code to error out.

We then initialize two variables, a point called **closestPoint** and an integer called **polygonID**. The point will hold the closest point on the mesh that we are querying and the integer will hold the id of the face that the vertex is associated with. The dag path is then passed to the **MFnMesh** function set. The **MFnMesh** class is used to operate on poly meshes. The naming of the classes is self-explanatory so if you were looking for a function set to work on NURBs surfaces, the obvious name would be **MFnNurbsSurface**...etc.

Once we have created an **MFnMesh** object called **meshFn** and passed it the necessary dag path, now we can make use of all its functions and operate on the selected mesh. We are looking for the closest point on the mesh, relative to the point we passed as an argument. So the function we are looking for is **getClosestPoint()**; This function takes four arguments, the point that we passed to the command as an argument, the variable that will hold the resulting position of the closest point, the space co-ordinates in which to do the calculation and the variable to hold the face id.

The output code is pretty self explanatory and just prints the acquired information to the script editor using the **MGlobal::displayInfo()**. At the end of each iteration we append a double array with the three points acquired and then at the end of the **dolt()** function we **setResult** to the double array. The **setResult()** function returns a data type that can be stored in a MEL variable. So in our example when we call our command through MEL, it would look like this:

```
float $pt_array[] = `gdcClosestPoint -px 0.1 -py 0.3 -pz 2.5`;
```

It's important to note that you cannot return multiple data types at the same time. You would need to return different results based on a flag or condition. Check the Maya documentation for further information on which data types can be passed to the **setResult()** function.

```
//for every mesh selected
MDoubleArray returnPoints;

for ( ; !iterList.isDone(); iterList.next() )
{
    MDagPath geoPath;
    iterList.getDagPath( geoPath );

    //declare variables
    MPoint closestPoint;
    int polygonID;

    //pass mesh path to MFnMesh function class
    MFnMesh meshFn( geoPath );

    //pass the point from arguments and find the closest point on the specified mesh
    //this is done in world space but can also be done in local space
    //this function will also return the faceID that the point belongs to
    meshFn.getClosestPoint( point, closestPoint, MSpace::kWorld, &polygonID );

    //output all the information retrieved to the script editor
    MString outputStr = "\n-----Closest Vertex-----\n\n";
    outputStr += "\tMesh Full Path: " + geoPath.fullPathName() + "\n";
    outputStr += MString("\tFace ID: ") + polygonID + "\n";
    outputStr += "\tClosest Vertex Position: \n";
    outputStr += MString("\t\tX: ") + closestPoint.x + "\n";
    outputStr += MString("\t\tY: ") + closestPoint.y + "\n";
    outputStr += MString("\t\tZ: ") + closestPoint.z + "\n\n";
    outputStr += "-----END-----\n";
    MGlobal::displayInfo( outputStr );

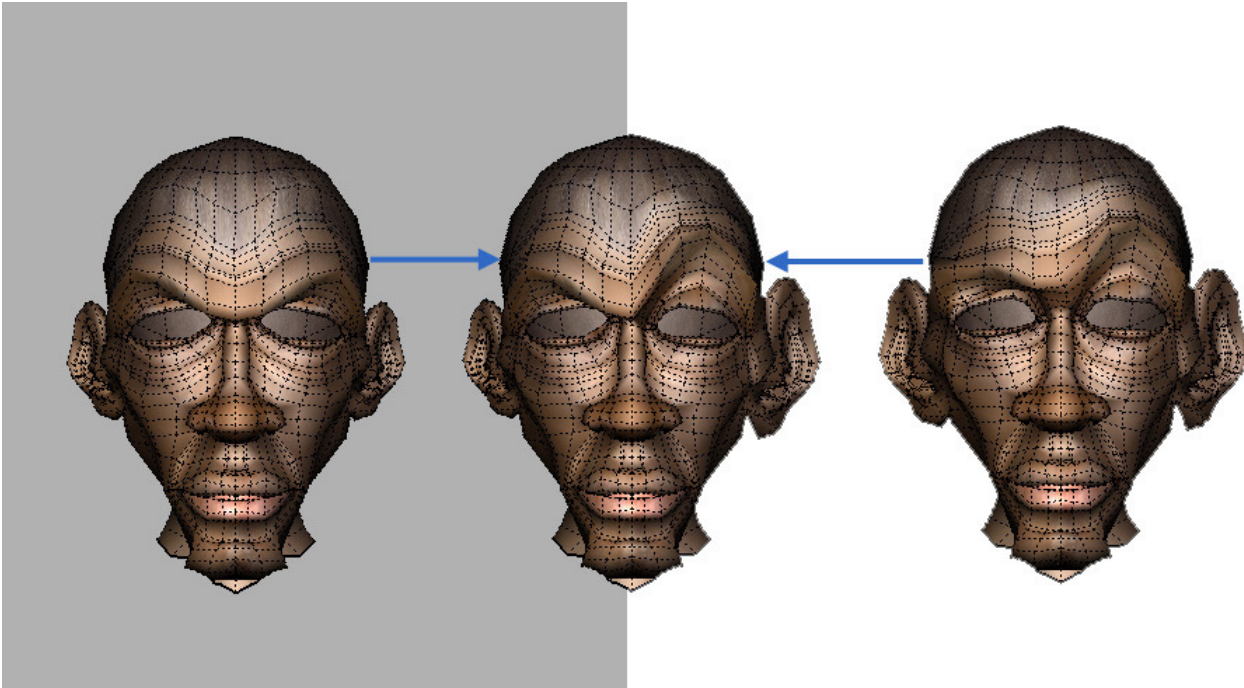
    //appended the returnPoints array with the retrieved closest points
    returnPoints.append( closestPoint.x );
    returnPoints.append( closestPoint.y );
    returnPoints.append( closestPoint.z );

}

//return the array
setResult( returnPoints );
return MS::kSuccess;
}
```

That is the final step in our first command. It is a very simple command and as a stand-alone tool, doesn't serve too much of a purpose, but the idea is really to introduce the basics of Maya's API and writing a custom MEL command. As an exercise to further your skills trying implementing the ability to handle NURBs surfaces as well as handling each type by detecting it automatically. In general commands are definitely the easier of plug-ins to write, especially when there is no undo and redo support incorporated. The next example will incorporate undo and redo support.

Shape Snake – Undo/Redo functionality



The next command will introduce the concept of undo and redo. Implementing undo and redo capabilities require that you take them into account before finalizing the design for your command. When you implement undo and redo you essentially add an **undolt()** and **redolt()** function to your plugin, instead of just a **dolt()** function as in the previous example. The first question that comes to mind is what is the difference between **dolt()** and **redolt()** and aren't they redundant? To answer this question I will describe the process of what will occur in your command. The basic explanation will potentially give you more insight into how to design your plug-in with undo and redo in mind.

The general flow of a command is, first you gather the information you need in order to perform the operation. Then you gather the information of the current state of the object(s) that you will be applying the operation to. This will be used to return the operation to its previous state if an undo is requested. Finally you execute the operation on the object(s) using the acquired information. Now if at this point the user requests an undo, the command will take the previous state information you recorded and apply that to the object(s). Basically returning it to its state before the command execution. If the user was to perform a redo, it would be redundant to retrieve all the operation information and undo information again. So at this point all you really need to do is just perform the operation again with the information you acquired the first time.

This description above basically defines the difference between **dolt()** and **redolt()**. The **dolt()** function gathers the information that is only required once. The **redolt()** function performs the actual work.

I will try not to repeat repetitive information from that described in the point on mesh command, so please refer to the source for a more detailed breakdown of the command.

The first few things you need to do are to declare all your needed functions in the header file. The new functions to pay attention to are the **redolt()**, **undolt()** and **isUndoable()** functions. The **isUndoable()** function is concise and simple, all it does is return true or false. This Boolean value, basically tells Maya that the command should or shouldn't be added to the undo queue when executed.

```
class CShapeSnap : public MPxCommand
{

public:

                                CShapeSnap();

    virtual                    ~CShapeSnap();

    MStatus                    doIt( const MArgList& );
    MStatus                    redoIt();
    MStatus                    undoIt();
    bool                      isUndoable() const;

    static                    void* creator();

};
```

First we'll take a look at the **doIt()** function. As mentioned earlier in this example the **doIt()** function will serve to retrieve information that is important to redoing and undoing the command. With that said, the first portion of the function gets the current selection list and checks to make sure there are only two objects selected. It is also a good idea to check to make sure that the object types selected are polygon meshes. I've omitted this step once again for the sake of keeping things simplified.

```
MStatus CShapeSnap::doIt( const MArgList& )
{
    MStatus stat = MS::kSuccess;

    //get selectionList to currently selected objects
    MSelectionList selectionList;
    MGlobal::getActiveSelectionList(selectionList);

    //check to make sure only two objects are selected
    if (selectionList.length() != 2)
    {
        MGlobal::displayError("Please make sure you have two objects selected!\n");
        return MS::kFailure;
    }
}
```


Now that we have our selection list, we can get handles to the actual objects. Essentially there will be two objects in this command, the original and the modified object. The modified shape will be the one that we want our original to become. We declare an **MDagPath** to the modified object and a **MObject** to point to the component data of the modified mesh, however we don't declare the original **MDagPath** or **MObject** anywhere in the **dolt()**. The reason for this is that we only need the modified information once off in the **dolt()** function where we retrieve the information. The original mesh will need to be queried in the **dolt()** for it's position before modification and then modified in the **redolt()** and **undolt()** functions. Because the original needs to be accessible in several different functions, we declare all of its handles as members of the **CShapeSnap** class. This way we can have access to it, from any **CShapeSnap** function.

Now that we have handles to all our data, we initialize an **MFnMesh** function set, by passing it our new acquired **MDagPath** handles.

All our handles and function sets are initialized and ready to go, we can now start querying and editing our meshes.

```
//modified object's path
MDagPath modifiedPath;
//component handle to modified object's vertices
MObject modifiedComponent;

//get dag path to both original and modified
//also get handles to components
selectionList.getDagPath(0, modifiedPath, modifiedComponent );
selectionList.getDagPath(1, m_originalPath );

//get a function set for polygons for both objects
MFnMesh modifiedFn( modifiedPath );
MFnMesh originalFn( m_originalPath );
```

We want the command to be able to work on whole object selections or component selections. That is the reason for getting a handle to the object's components in the first place. In order to check whether or not we have an object or it's components selected, we will validate the data stored in the component handle. If the component handle is NULL, then we know that the whole object was selected and if it is not NULL then we can retrieve the indices of the selected components.

We use the **MFnSingleIndexedComponent** function set to allows us to query or edit the component data. In this instance we just want the indices of the selected components so we call the **getElements()** function and pass it a member variable of type **MIntArray** called **m_vertexIds**. As with the original object handles, the vertex indices are also going to be needed in all the functions and therefore are also declared as a member of the **CShapeSnap** class. You can tell member variables by the "m_" prefix used. This is another good example of why naming conventions are so important!

If the component handle is NULL then we just populate the **m_vertexIds** array with all the vertex ids of the mesh.

```

if (!modifiedComponent.isNull())
{
    //get the selected components
    MFnSingleIndexedComponent modifiedComponentFn( modifiedComponent);
    modifiedComponentFn.getElements( m_vertexIds );
}
else
{
    //if there is no component selection - get all vertex ids
    int num = originalFn.numVertices();
    for ( int i = 0; i < num; i++ )
        m_vertexIds.append( i );
}

```

Now that everything is initialized, we can query the positions of the modified vertices. We loop the **m_vertexIds** and using the **MFnMesh::getPoint()** function we pass the index of the vertex we want to query the **MPoint** that the position will be assigned to and the space in which we are querying. In our case, we want the position to be based in object space. This way we can move the transforms around and snap the vertices relative to it's transform position. We retrieve the positions of both the modified mesh and the original mesh and append the **m_pointArray** and the **m_undoPoints** respectively.

Just to re-iterate, the reason we are storing the original mesh's vertex position is so that when the user executes an undo operation we know how to return the object to it's previous state before being snapped to the modified mesh position. This is where implementing undo can become a huge inconvenience; you always have to save the previous state if an object is to be modified.

*Maya does provide some classes to help with undoing, depending on the type of operation. If you are modifying animation curves there is a **MAnimCurveChange** class that will store all changes and then has it's own **redolt()** and **undolt()** functions that can be called. Similarly If you are executing MEL from the API, creating nodes, deleting nodes, creating connections or deleting connections, there is a **MDGModifier** class which will help handle the storing of previous states and the ability to then undo and redo them.*

The last part of the **dolt()** function is to return the result of the **redolt()** function. This way when you execute the command for the first time the **dolt()** will execute and because it returns the results of the **redolt()**, obviously the **redolt()** function is also called. Now from this point on, if the user toggles between undo and redo, only the **undolt()** and **redolt()** functions are called. Essentially this just ensures that the execution part (the **redolt()**) of your command is performed the first time the command is executed.

```

//loop the verts retrieved and store original position information and
//new position information
for (unsigned int i = 0; i < m_vertexIds.length(); i++)
{
    //get the modified objects points and store them in an array
    MPoint modPt;
    stat = modifiedFn.getPoint( m_vertexIds[i], modPt, MSpace::kObject );
    if (!stat)
    {
        MGlobal::displayError( "error getting modified objects point position!" );
        return MS::kFailure;
    }
    m_pointArray.append( modPt );

    //get the original objects points to use for undo purposes
    MPoint origPt;
    stat = originalFn.getPoint( m_vertexIds[i], origPt, MSpace::kObject );
    if (!stat)
    {
        MGlobal::displayError( "error getting original objects point position!" );
        return MS::kFailure;
    }
    m_undoPoints.append( origPt );
}

//this will not execute the redoIt() function and return it's
//MStatus result as the doIt() functions return.
return redoIt();

```

Lets take a look at the **redoIt()** function now. We initialize an **MFnMesh** function set on the **MDagPath** of the original mesh. We then loop the previously stored indices of either the entire mesh or just the selected components. Finally we set the new position of the original mesh vertices to match that of the stored points from the modified mesh vertices. We use the **MFnMesh::setPoint()** function which works in the same way as the **getPoint()** function, with the exception that it takes a **MPoint** as a new position to set the vertex to.

That's all there is to the **redoIt()** function. As you can see, if we were to execute an undo and redo now, the entire retrieval process in the **doIt()** function is bypassed. Depending on the density of the mesh in question, if everything was implemented in the **redoIt()**, including the retrieval process, things could slow down drastically when calling a redo after an undo. This is also a pretty simplistic scenario, but commands can get a lot more complex and the retrieval or initializing aspect could take much longer, in which case this becomes even more important to be aware of.

```

MStatus CShapeSnap::redoIt()
{
    MStatus stat;

    //intialize function set for original mesh and appy points position
    //retrieved in the doIt() function
    MFnMesh originalFn( m_originalPath );
    for (unsigned int i = 0; i < m_vertexIds.length(); i++)
    {
        stat = originalFn.setPoint( m_vertexIds[i], m_pointArray[i], MSpace::kObject );
        if (!stat)
        {
            MGlobal::displayError( "error setting point position!" );
            return stat;
        }
    }

    return MS::kSuccess;
}

```

I wont go into too much detail with the **undolt()** function, as it is pretty much the exact same thing as the **redolt()**, except that we set the original mesh's vertices back to their original state by using the **m_undoPoints** instead of the **m_pointArray** that is used in the **redolt()**.

```

MStatus CShapeSnap::undoIt()
{

    MStatus stat;

    //intialize function set for original mesh and appy undo points position
    //retrieved in the doIt() function
    MFnMesh originalFn( m_originalPath );

    for (unsigned int i = 0; i < m_vertexIds.length(); i++)
    {
        stat = originalFn.setPoint( m_vertexIds[i], m_undoPoints[i], MSpace::kObject );
        if (!stat)
        {
            MGlobal::displayError( "error setting point position in undo!" );
            return stat;
        }
    }

    return MS::kSuccess;
}

```

That concludes the shape snap command and the command section of the API. The registering of the command is the same for the shape snap command as it was for the point on mesh command. Always figure out if you need to be able to undo your operation before starting to write or plan your command. A good rule of thumb is if it is going to modify something in the DG and not just query information then implementing undo is usually a good idea.

Nodes:

Writing your own Custom DG Nodes

In this section I will be taking you through the basics of writing your own custom DG nodes. The basis for a node is, it takes an input, does some computation with this input in the **compute()** function and then outputs the result. In your node you will setup dependencies so that the output will be recomputed when an input is changed.

Nodes are derived from the **MPxNode** class. When you derive from **MPxNode** there will be a function called **compute()**. This is the function where you will be doing all your computations. In the **compute()**, you get your input values and pass them to variables, once they are stored in variables you perform some operation with them and then pass the result of the operations performed to the output attribute. The final step is to set the attribute as being “clean”. This basically tells Maya that the attribute has been computed and holds the new value.

One thing to remember about a node is that it should never know about anything outside of itself. What this basically means is that if there is something in your scene that should affect your node or be used in the compute function, you should never retrieve the information without connecting the appropriate attribute as an input. Disregarding this can result in dependency cycles and cause un-wanted results.

In order for the dependency graph and Maya’s file formats to recognize a custom node, each node will need a unique id assigned to it. This is done through the use of a **MTypeId**, which is a class Maya supplies you with. A valid range for the id number is between **0x00000000** and **0x0007ffff**. It’s very important that this number is set correctly from the start and not changed too often, if at all. Whenever a Maya Binary file is opened the **MTypeId** is used to identify the node type and what information to load from disk. If this number changes, then opening existing Maya Binary files that contain the node will not load the correct information. Maya ASCII files use the node’s type to identify the information so this problem is not as much of a concern with the respect to changing the id number. If you plan on making 3rd party plug-ins that will be commercialized, it’s important that you contact Alias for a unique id so that there is no conflict with any other commercial plug-ins currently available.

When your node is created the **initialize()** function is called. This function will create all your attributes and setup the dependencies between them. Once all the attributes are initialized and added to your node, you can access them using an **MDataHandle**. An **MDataHandle** basically just gives you a handle to the attribute’s data and allows you to pass the data to a variable of the appropriate data type.

Poly Face Custom DG Node:

Attaching objects to polygon faces is definitely something that can be very useful. I figured this would make for a good custom DG node example. All the source code will be supplied, so you can also make modifications if you please. Our goal for this node will be to create a node that will output the position,

orientation and normal of any specified face on a polygon mesh.

First off is an example of what the class declaration for our node will look like. As mentioned above you will derive from the **MPxNode** class. We then declare our functions and members of the class. The full header file is available with the notes.

```
class CPolyFaceNode : public MPxNode
{
public:

    CPolyFaceNode();
    ~CPolyFaceNode();

    static void*      creator();
    static MStatus    initialize();

    virtual MStatus compute(const MPlug &plug, MDataBlock &dataBlock);
```

Next we have to define all the static **MObjects** that will be the handles to the attributes. We also define the node's unique **MTypeId** that will then be registered with the plug-in.

```
MObject CPolyFaceNode::mIn_polyShape;

MObject CPolyFaceNode::mIn_faceId;

MObject CPolyFaceNode::mIn_upVector;

MObject CPolyFaceNode::mOut_normal;
MObject CPolyFaceNode::mOut_rotation;
MObject CPolyFaceNode::mOut_position;

const MTypeId CPolyFaceNode::m_typeId( 0x000033ff );
```

The first real implementation of the node is the **initialize()** function that will get called when the node is created. In the initialize for the poly face node, we will start by adding all the attributes that our node needs in order to compute the necessary information. There are several attribute function sets that allow you to create different types of attributes. The most commonly used will definitely be **MFnNumericAttribute**, however there are several more that you will find yourself using frequently. In our case we are going to need numeric and typed attributes, so we will declare function sets for these two. The image illustrates numeric attributes of type **k3Double** being defined and initialized as well as a typed

attribute of **kMesh**. For further examples of the rest, refer to the supplied source code. You create the attribute and pass the data to the earlier declared **MObject** that will hold the data for each specific attribute. From this point on, anytime you need to pass the data of the attribute to a function set, you will be using the attributes **MObject** handle. Once you have created the attribute data you can define whether the attribute should be keyable, hidden, storable...etc. Finally you call the **addAttribute()** function which will add the attribute to the node.

```
MStatus CPolyFaceNode::initialize()
{

    //initialize all attributes and attribute dependencies
    MStatus stat;

    MFnNumericAttribute nAttr;
    MFnTypedAttribute tAttr;

    //input shape to be processed
    mIn_polyShape = tAttr.create( "polyShape", "pshp", MFnData::kMesh, &stat );
    if (!stat)
        MGlobal::displayError("error creating polyShape attribute!");
    nAttr.setKeyable(false);
    nAttr.setStorable(false);
    stat = addAttribute( mIn_polyShape );
    if (!stat)
        MGlobal::displayError("error adding polyShape attribute!");

    //id of the face to read
    mIn_faceId = nAttr.create( "faceId", "fid", MFnNumericData::kInt, 0, &stat );
    if (!stat)
        MGlobal::displayError("error creating faceId attribute!");
    nAttr.setKeyable(true);
    nAttr.setStorable(true);
    stat = addAttribute( mIn_faceId );
    if (!stat)
        MGlobal::displayError("error adding faceId attribute!");

    //up vector used to compute orientation and output vector
    mIn_upVector = nAttr.create( "upVector", "upv", MFnNumericData::k3Double, 0.0, &stat );
    if (!stat)
        MGlobal::displayError("error creating upVector attribute!");
    nAttr.setKeyable(true);
    nAttr.setStorable(true);
    stat = addAttribute( mIn_upVector );
    if (!stat)
        MGlobal::displayError("error adding upVector attribute!");
```

Once the attributes have been added and initialized the next step is to set the dependencies between the attributes. If you want an input attribute to trigger a compute on a specific output attribute, it's imperative that you define a relationship between the attributes. The way to do this is by calling the **attributeAffects()** function. This function takes the input attribute as the first argument and the output attribute as the second attribute. This is essentially saying whenever the specified input attribute changes, compute the specified output attribute.

```
//setup dependencies between attributes
stat = attributeAffects(mIn_polyShape, mOut_position);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");
stat = attributeAffects(mIn_faceId, mOut_position);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");

stat = attributeAffects(mIn_polyShape, mOut_rotation);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");
stat = attributeAffects(mIn_faceId, mOut_rotation);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");
stat = attributeAffects(mIn_upVector, mOut_rotation);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");

stat = attributeAffects(mIn_polyShape, mOut_normal);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");
stat = attributeAffects(mIn_faceId, mOut_normal);
    if (!stat) MGlobal::displayError("error setting attribute dependency!");
```

That's all we are going to define in our initialize. If at this point that were all you implemented in your node and you then created the node in your scene, you would have a node with all the attributes added that have no functionality but are defined. The next section will give the node some purpose, by defining the nodes implementation and functionality. The function that handles all of this is the **compute()** function. The **compute()** is really the brain of the node. In the compute function we will get the values of all the attributes, use them to do some sort of calculations and finally output the result.

The compute function is inherently passed two arguments, an **MPlug** and an **MDataBlock**. A plug can be thought of as an attribute and an **MPlug** gives you a set of functions to query and modify the plug/attribute. An **MDataBlock** is a convenient way of storing all the attribute data so that you can easily access the information through the **compute()**. Maya provides you with an **MDataHandle** and **MArrayDataHandle** to access the attributes in an **MDataBlock**. As mentioned above the compute will inherently be passed the **MPlug** for the attribute that is being marked dirty and the **MDataBlock** for the specific node. You can use the plug passed to the compute to verify which operations should be performed for which outputs and also to verify if the plug is a valid attribute. In the example below the operation will only take place if any of the output attributes (**mOut_position**, **mOut_rotation** and **mOut_normal**) are to be evaluated.

First off we create an **MDataHandle** for each of our input attributes. We get the data by calling the **MDataBlock::inputValue()** function and passing the name of the attribute in question. This function will return an **MDataHandle**. We then use the **MDataHandle** to get the attribute value and pass it to a variable.

```
MStatus CPolyFaceNode::compute(const MPlug &plug, MDataBlock &dataBlock)
{
    MStatus stat;

    //for any of the plugs, execute the function
    if ( mOut_position == plug || mOut_normal == plug || mOut_rotation == plug )
    {

        MDataHandle polyShapeData = dataBlock.inputValue( mIn_polyShape, &stat );
        if (!stat) MGlobal::displayError("error getting polyShape data!");

        //intialize shape function set
        MObject polyShapeObj = polyShapeData.asMesh();
        MFnMesh polyShapeFn( polyShapeObj );

        //get face ID to identify polygonId
        MDataHandle faceIdData = dataBlock.inputValue( mIn_faceId, &stat );
        if (!stat) MGlobal::displayError("error getting faceId data!");

        int faceId = faceIdData.asInt();

        //get up vector attribute data
        MDataHandle upVectorData = dataBlock.inputValue( mIn_upVector, &stat );
        if (!stat) MGlobal::displayError("error getting upVector data!");

        MVector upVector = upVectorData.asDouble3();

        //declare vector to hold the face normal
        MVector faceNormal;
        stat = polyShapeFn.getPolygonNormal( faceId, faceNormal, MSpace::kWorld );
        if (!stat) MGlobal::displayError("error getting face normal!");

        //set the out normal
        MDataHandle outNormalData = dataBlock.outputValue( mOut_normal, &stat );
        if (!stat) MGlobal::displayError("error getting outNormal data!");
        outNormalData.set( faceNormal );
    }
}
```

Once we have retrieved all the attribute data and passed them to variables, we start the actual implementation. For the **polyShape** input attribute, instead of passing the result to a standard data type such as an integer or float – which would clearly not be feasible – we pass it to an **MObject** which will have a handle to the **polyShape** data so that polygon function sets such as **MFnMesh()** can be used on the **polyShape** data.

Now that we have a function set that will operate on the mesh we can start to query the necessary information we care about. We call the **getPolygonNormal()** function that will return us the normal of the specified face in world space. Now that we have the normal of the face we can define an objects orientation. Using the normal and the input up-vector we construct a **Quaternion** that will give us the rotation amount from the up-vector to the normal. Objects are rotated in Maya using **Eulers** so we convert the **Quaternion** rotation into an **Euler** rotation. Finally we pass the **Euler** rotation to a vector (3 doubles) and multiply the vector by 57.29578, which will convert from radians to degrees. Now that we have our rotation in a format that can be passed to our output attribute, we set the **m_outRotation** attribute with the **outRotation** vector. For the position we follow very similar steps to that of the rotation, except that in order to calculate the position of a face we do an average of all the vertices that make up that face. We use the **getPolygonVertices()** function to get a list of the vertices that make up the specified face. Once we have the vertices, we loop the list of vertex ids and get the position of each vertex in world space. While retrieving each point, we add the points up by appending the **sumOfPoints** vector.

```

//declare vector to hold the face normal
MVector faceNormal;
stat = polyShapeFn.getPolygonNormal( faceId, faceNormal, MSpace::kWorld );
    if (!stat) MGlobal::displayError("error getting face normal!");

//set the out normal
MDataHandle outNormalData = dataBlock.outputValue( mOut_normal, &stat );
    if (!stat) MGlobal::displayError("error getting outNormal data!");
outNormalData.set( faceNormal );

//initialize a quaternion rotation based off the 2 vectors
MQuaternion quaternion( upVector, faceNormal );
MEulerRotation euler = quaternion.asEulerRotation();

//convert euler to vector and convert radians to degrees
MVector outRotation = euler.asVector();
outRotation *= 57.29578; //radian to degrees

//set out rotation
MDataHandle outRotationData = dataBlock.outputValue( mOut_rotation, &stat );
    if (!stat) MGlobal::displayError("error getting outRotation data!");
outRotationData.set( outRotation );

//get an id list of all the connected vertices
MIntArray vertexList;
stat = polyShapeFn.getPolygonVertices( faceId, vertexList );
    if (!stat) MGlobal::displayError("error getting poly vertex list for face!");

//loop all the vertices connected to the face and get an average of their position
MVector sumOfPoints;
unsigned int numPts = vertexList.length();
for ( unsigned int i = 0; i < numPts; i++ )
{
    MPoint pts;
    stat = polyShapeFn.getPoint( vertexList[i], pts, MSpace::kWorld );
        if (!stat) MGlobal::displayError("error getting point position!");
    sumOfPoints += pts;
}

```

Finally we divide the sum of all the points by the number of points and that is our final position. Now that we have completed all position calculations, we move onto setting the output attribute with the newly calculated data. Earlier we got a **MDataHandle** from the **MDataBlock** using the **inputValue()** function and this time we receive the **MDataHandle** through using the **outputValue()** function. The

inputValue() function guarantees that the attribute data will be valid and therefore is ideal for reading data from the attributes. The **outputValue()** function on the other hand does not guarantee the data will be valid and is therefore used for writing data to the node.

The **MDataHandle** has a **set()** function that is used to set the new data. Once the data has been set, the **MDataBlock::setClean()** function is called which basically just tells Maya that the attribute is set and does not need to be evaluated.

```
//calculate the average of all the points
MPoint outPoint = { sumOfPoints / (double)numPts };

//set the out position
MDataHandle outPositionData = dataBlock.outputValue( mOut_position, &stat );
    if (!stat) MGlobal::displayError("error getting outPosition data!");
outPositionData.set( outPoint );

//let Maya know that all the data in this node is set and is correct.
dataBlock.setClean( plug );
```

The last aspect we will look at for the node is the registering and de-registering of the node. This registration will take place in the nodes **initializePlugin()** function and the de-registration will take place in the nodes **uninitializePlugin()** function exactly like in the command examples. To register the node you basically call the **MFnPlugin::registerNode()** function which takes the node name as the first argument, the unique **MTypeId** as the second, the node's **creator()** function as the third and the node's **initialize()** function as the last. The **uninitializePlugin()** function has a **deregisterNode()** function call that only takes the unique **MTypeId** to de-register the node. This part is definitely the easiest to forget when writing plug-ins. So if you find that for some odd reason your plug-in is not loading or you cannot create your nodes, check this portion of your code first!

```
MStatus stat;
MFnPlugin plugin( obj, "Judd Simantov", "1.0", "Any" );

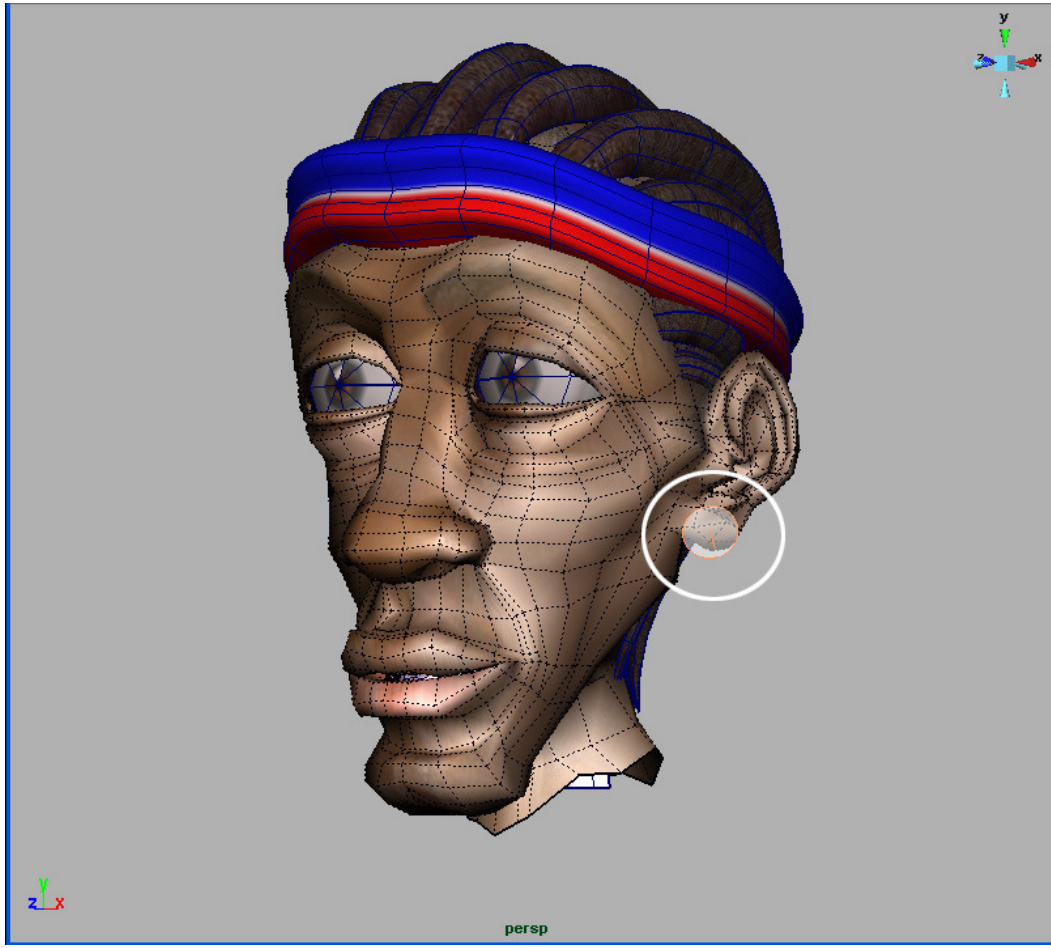
//register polyFace node
stat = plugin.registerNode( "gdcPolyFaceNode", CPolyFaceNode::m_typeId,
                           CPolyFaceNode::creator, CPolyFaceNode::initialize );

if (!stat)
{
    stat.perror("registerNode");
    return stat;
}
```

```
stat = plugin.deregisterNode( CPolyFaceNode::m_typeId );  
if (!stat)  
{  
    stat.perror("deregisterNode");  
    return stat;  
}
```

That's all there is to writing your own custom DG node. There is a MEL script that accompanies this talk called **gdcCreatePolyFaceNode.mel**, which will setup the node for all the selected faces on a mesh. I would suggest trying to do it manually as well, so that you can get a feel for setting up the connections and in turn gain a clearer understanding of exactly what is going on under the hood. The full source code is also supplied with these notes for a detailed example of the implementation. I would suggest definitely looking through those as well. There is a good amount of stuff that I opted to omit from this document for the sake of sticking to the fundamental concepts that are necessary, so make a point of reading through as much of the accompanied source files as possible.

Custom Maya Locators (A node with a shape):



A locator is exactly like a node with the exception that it draws a shape in the view-port. This gives a locator the ability to display the results of what the node is doing to the user and therefore make certain aspects of a node that are hard to visualize more clear. The major difference between an MPxNode and an MPxLocatorNode with respect to implementation is that a locator has a **draw()** function that you can implement, so you can make your own native OpenGL calls. Unfortunately OpenGL in itself is out of the scope of this talk, although I will cover it briefly when implementing the **draw()** function for our locator.

Just like a node, a locator has a **compute()** function where you handle your attribute inputs and generate your attribute outputs. A locator is registered as a dependency graph node and therefore will also require that you make sure that your locator has a unique **MTypeId**. Finally when you register your node you have to specify that the node being registered is a locator node. I will cover this later on, when talking about the **initializePlugin()** and **uninitializePlugin()** of a locator.

Converting the Poly Face Node into a Poly Face Locator:

As I mentioned above a locator is very similar to a node with the exception that a locator has drawing abilities. Due to this, I will not cover any of the same implementation discussed above, so I will only be discussing the areas that are unique to a locator. Our **compute()** function will stay almost exactly the same as in the node example. The only difference is that we will be adding a **shapeScale** attribute that will allow us to scale the size of the locator shape. Our goal is to draw a shape that is a circle and a line that is positioned at the center of the face and is pointing in the direction of the face normal.

The first difference to note is that a node is derived from the **MPxNode** class and a locator is derived from the **MPxLocatorNode** class.

Below is the first part of the class declaration, as you can see we are deriving from the **MPxLocatorNode** class and have two new functions, **draw()** and **getCirclePoints()**. Only the **draw** function is absolutely necessary, the **getCirclePoints** function is just used to calculate the points for drawing a circle, this could easily have just been done in the **draw** function, however it comes back to modularity and keeping your code clean.

```
class CPolyFaceLocator : public MPxLocatorNode
{
public:

    CPolyFaceLocator();
    ~CPolyFaceLocator();

    void draw(M3dView &view, const MDagPath &path,
              M3dView::DisplayStyle style, M3dView::DisplayStatus displayStatus);

    bool getCirclePoints( MPointArray &pts, const MVector &normal );

    static void*      creator();
    static MStatus    initialize();

    virtual MStatus compute(const MPlug &plug, MDataBlock &dataBlock);
```

As mentioned previously, the **draw()** function is the heart of a locator. Most of what takes place in the **draw** function is **OpenGL** based. This means that in order to draw complex locators, you will need to gain a basic knowledge of **OpenGL**. As with the compute function there are some inherit arguments that are passed to the draw function. The first is an instance of **M3dView**, this class gives you access to functions that allow you to work in a Maya **OpenGL** window. The second argument is an **MDagPath** to the locator in the DAG. The fourth is an instance of **M3dView::DisplayStyle**. This determines what mode the current OpenGL window is drawing in. This way you can determine whether the current viewport is in wireframe, shaded or texture mode. The last argument is the **M3dView::DisplayStatus**, this determine the selection

status of the object. Using the **DisplayStatus** you can determine if the object is selected, non-selected, highlighted, affected...etc. Depending on all the retrieved display information, you can make decisions on how to draw your object.

The **draw()** function does not take an **MDataBlock** as an argument and therefore if you want to gain access to your node's plugs/attributes, you need to initialize an **MPlug** using the attribute's **MObject** handle and using the **thisMObject()** function call. The **thisMObject()** basically returns an **MObject** handle of the current class that you are calling the function in. You then initialize an **MPlug** by passing those two **MObjects** and you can now query and edit the attribute's values. As you can see below we get the values of all the attributes that will help us draw the locator.

```
void CPolyFaceLocator::draw(M3dView &view, const MDagPath &path,
    M3dView::DisplayStyle style, M3dView::DisplayStatus displayStatus)
{
    MVector position, normal;
    double scale;

    //get scale value from shapeScale attribute
    MPlug scalePlug(thisMObject(), mIn_scale );
    scalePlug.getValue( scale );

    //get position value from position attribute plug
    MPlug positionPlug(thisMObject(), mOut_position );
    MObject positionObj;
    positionPlug.getValue( positionObj );

    MFnNumericData posData( positionObj );
    posData.getData( position.x, position.y, position.z );

    //get normal value from normal attribute plug
    MPlug normalPlug(thisMObject(), mOut_normal );
    MObject normalObj;
    normalPlug.getValue( normalObj );

    MFnNumericData normData( normalObj );
    normData.getData( normal.x, normal.y, normal.z );
```

Once we have passed all the data needed to draw our locator to variables, we can actually start the drawing implementation. The first thing we need to do is prep everything so that we can draw our shape. Using the **M3dView view** that was passed to the draw function, we call the **beginGL()** function. This function will setup a port so that we can make native **OpenGL** calls. Next we call **glPushAttrib()** and **glPushClientAttrib()**. In a nutshell, these functions allow you to “push” or save the current attribute states from the current **OpenGL** view. This way if you modify the line width or transparency of something, all of Maya's existing lines and transparent objects will not be affected. It is very important that you include both these functions before drawing your own objects. The concept of “pushing” and “popping” is commonly used in **OpenGL** and something that you will probably become familiar with in no time.

Enabling and disabling of states is also something commonly used in OpenGL. In order to allow for transparency of polygons we have to enable alpha blending using the **glEnable()** function.

glBlendFunc() determines the source and destination blend factors. Using the **M3dView::DisplayStatus** we can determine if the object is selected or not. Based on whether or not the object is selected, we either display an orange color or a light green color. Lastly we set the line width before drawing the objects. In order to set the line width, we use the **glLineWidth()** function.

```
//prep viewport for custom OpenGL drawing
view.beginGL();
glPushAttrib( GL_ALL_ATTRIB_BITS );
glPushClientAttrib( GL_ALL_ATTRIB_BITS );

//used to enable alpha blending in order to allow for transparency
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

//if object is selected show green - else show orange
if (displayStatus == M3dView::kActive || displayStatus == M3dView::kLead)
    glColor4f( 0.6f, 1.0f, 0.4f, 1.0f );
else
    glColor4f( 1.0f, 0.6f, 0.4f, 1.0f );

//set the line width
glLineWidth( 1.0f );
```

Now that we have initialized all the necessary data for drawing the objects, we can begin defining our shapes. In order to transform objects in OpenGL you must save the current transformation matrix using **glPushMatrix()** and then apply your transformations. Once you've stored the transformation matrix, applied your transformation and drawn your object, you must restore the previous transformation state using the **glPopMatrix()**. Now that we have stored the transformation state, we can translate and scale the object by calling the **glTranslated()** and **glScaled()** functions. We use the position of the face that we calculated in the compute function to translate the object and we use the scale from the shapeScale attribute we created to modify the size in which the shape is displayed.

In order to draw something in OpenGL you must call the **glBegin()** function which will take an argument that describes what you will be drawing. In our case we will start off by just drawing a line to symbolize the normal of the face. After calling **glBegin()** we set the co-ordinates of our vertices using the **glVertex3d()** function. The first point is at the origin and the second is the face normal that we calculated in the **compute()**. When done drawing, we call the **glEnd()** function. The rest of the code that follows in the image below is basically this same format described above. We then also go on to draw a polygon circle that is centered at the center of the face.

```

//push transformation matrix so we can apply our own transformations
glPushMatrix();

//move all drawing below to the face position
glTranslated( position.x, position.y, position.z );
//scale shape based on user attribute
glScaled( scale, scale, scale );

//draw normal line
glBegin(GL_LINES);
    glVertex3d( 0.0, 0.0, 0.0 );
    glVertex3d( normal.x, normal.y, normal.z );
glEnd();

//get array of points to define the circle
MPointArray circlePts;
getCirclePoints( circlePts, normal );

//draw circle wire
glBegin(GL_LINE_STRIP);
for ( unsigned int i = 0; i < circlePts.length(); ++i )
{
    glVertex3d( circlePts[i].x, circlePts[i].y, circlePts[i].z );
}
glEnd();

glColor4f( 1.0f, 1.0f, 1.0f, 0.5f );

//using the same circle points above draw polygon triangle fan
glBegin(GL_TRIANGLE_FAN );
glVertex3d( 0.0, 0.0, 0.0 );
for ( unsigned int i = 0; i < circlePts.length(); ++i )
{
    glVertex3d( circlePts[i].x, circlePts[i].y, circlePts[i].z );
}
glEnd();

```

Once we have done all our drawing, we restore all the previous states and transformation matrices. Lastly we call the **M3dView::endGL()** function, which tells Maya that we are done making native **OpenGL** calls.

```

//pop back up to previous transformation state
glPopMatrix();

//restore all openGL states
glPopClientAttrib();
glPopAttrib();
view.endGL();

```

The **initializePlugin()** function has one small difference to that of a normal **MPxNode**. Below you can see that when we call the **registerNode()**, the last argument we pass is **MPxNode::kLocatorNode**, this defines that our node is registered as a locator node.

```

//register poly face locator node - specify MPxNode type as last argument
stat = plugin.registerNode( "gdcPolyFaceLocator", CPolyFaceLocator::m_typeId,
    CPolyFaceLocator::creator, CPolyFaceLocator::initialize, MPxNode::kLocatorNode );

```

That concludes writing your own custom locators. The biggest hurdle with writing locators is that you are required to learn an entirely new API, **openGL**. While familiarizing yourself with Maya's API and more importantly C++, it's probably a good idea to leave the **openGL**-based plug-ins for a later stage.

Conclusion:

Writing advanced tools can be extremely challenging and overwhelming at first. It's important that you persevere through the complexity and magnitude that is initially presented. The learning curve will soon become exponential and in good time, the concepts that seemed impossible to grasp, will become common usage in your everyday tool writing.

Everything that I have presented in this paper and talk is the basis for much larger and more innovative tools. Although some of the examples presented are production applicable, I urge you to use them as a guide towards coming up with your own creative tools, that are far more extensive than those presented here.

If you have any further questions and need to contact me, please email me at:

judd@cgmuscle.com

Thank you very much for the support by attending this talk. I hope it will prove to be useful to you in your future development.

Special Thanks to: Jacky Simantov, Anna DiDonato, Sammy Simantov, Carlos Gonzalez-Ochoa, Jeremy Lai-Yates, Vitaliy Genkin and everybody at Naughty Dog Inc.