

2.0 trueSpace Scripting Fundamentals

Introduction

In trueSpace, scripting is handled by the Windows Scripting Host and adheres to the fundamental concepts and rules outlined at: <http://msdn2.microsoft.com/en-us/library/ms950396.aspx> .

There are important sections at the above link for:

- [VBScript User Guide](#)
- [VBScript Language Reference](#)
- [JScript User Guide](#)
- [JScript Language Reference](#)

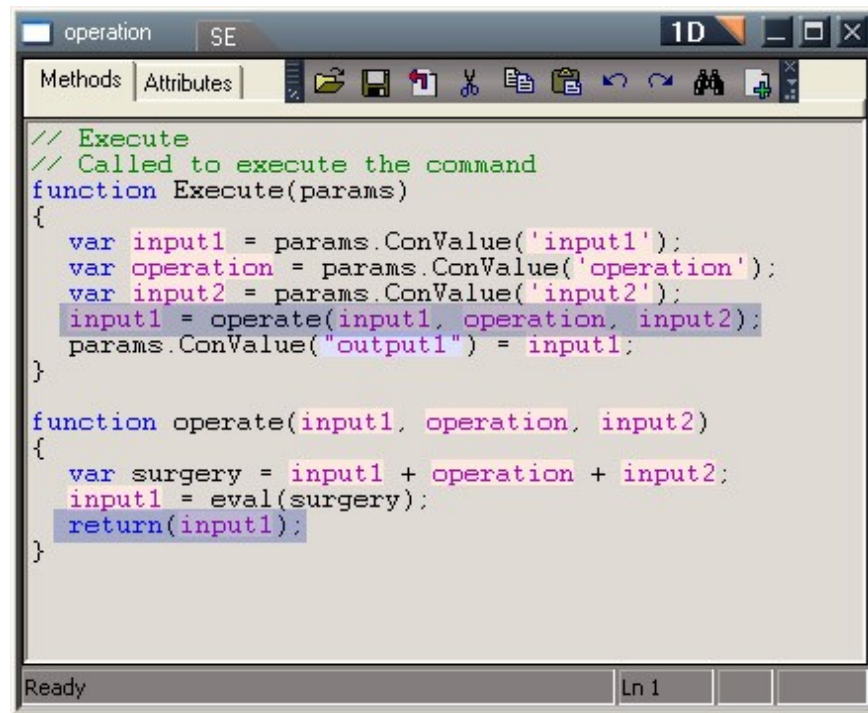
It is highly recommended that you familiarize yourself with these links for future reference. There are differences between the two languages and the references above will allow you to adapt your scripts from one language to the other if desired.

Variable Scope

Variables are used within the trueSpace Scripting environment just as they are used elsewhere. They have what is termed as “scope”. They may have Local or Global scope. A variable is considered local if it is declared within a procedure(VBScript)* or function(Jscript). Global variables are declared outside any procedure or function. The global variable is considered to have a life as long as the script is running. The local variable has a life as long as a procedure or function is being used. If you wish to carry a value for longer than even the script is alive, you would set the value of the variable to an input or output attribute. This way that value has an indefinite lifetime. The purpose behind these life-times for variables is to allow flexibility for the variables as desired.

In addition to the lifetime of a variable, a special scenario can exist, where a variable can be passed to another function with a specific line of code. This extends the life of the variable in a way. The Jscript command object shown below does this.

* In VBScript, “functions” do exist, however they are special cases where it is necessary to return a value to a procedure.



Passing variables between functions in jScript

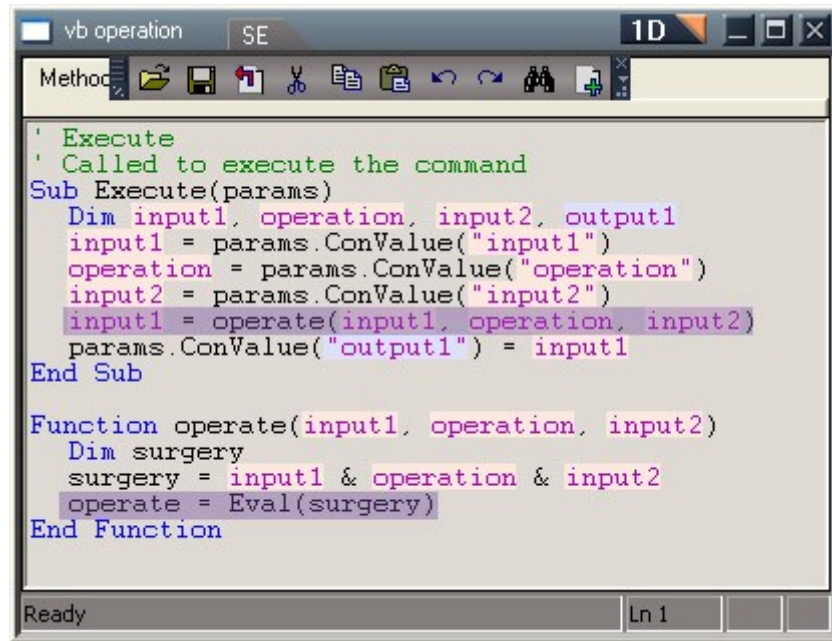
The highlighted code above shows:

```
input1 = operate(input1, operation, input2);
```

The variable input1 is equal to the result of the function “operate” and three variables are being sent to the “operate” function; input1, operation and input2. Down below in the actual “operate” function, some calculations are done on the variables that the function received and the result is returned with following line of code:

```
return(input1);
```

At this point in time the variable input1 is passed back from the “operate” function and returned to the right side of the equation from the original Execute function. The Execute function “paused” execution until it received the value back from the “operate” function. Once received, it continues to execute until it finishes. At that time the variable would have passed to an output connector called “output1”, where it will remain indefinitely (remember that a variable can be set to an input or an output connector for a lifetime that spans beyond the life of the function or script for that matter. Anytime you “set” an input or output connector via script, you are preserving the value for future use if desired.



Passing variables between a procedure and function in VBScript

In VBScript, to reproduce the same script, we have to create a Function, which allows for the return of a value to the calling Procedure. In our script image above, The Sub Execute sends the “operate” function some values:

```
input1 = operate(input1, operation, input2)
```

The “operate” function receives the values and using the concatenate character (&), strings together the 3 values into what is termed a “string expression” that we called “surgery”:

```
surgery = input1 & operation & input2
```

The next line of code actually places a result value into the function “operate”:

```
operate = Eval(surgery)
```

This is the only way to pass the value back to the original procedure. Although different from the jScript ability to pass variables/values between functions, in VBScript, functions have a more explicit meaning. The following line of code is actually a function as defined by VBScript reference documents:

```
Eval(surgery)
```

There are many of these types of functions in VBScript. Each of these functions will pass the result of the function to the procedure it is located in. In order to leave the original procedure and calculate a value outside that procedure then return the result, you have to create a function, which acts like any other function in VBScript. The example code in the “vb operation” script is a good example to remember for such cases.

Loops

When creating scripts, one script/code tool that is used quite a bit is called a Loop.

```
// Name:      Stitches
// Purpose:   demonstrate a simple loop.
//           Hook the "operation's" output1 connector,
//           to the reqStitches input connector on this script.

function OnComputeOutputs(params)
{
    var reqStitches = params.ConValue('reqStitches');

    for(i=0; i<reqStitches; i++)
    {
        params.ConValue('completeStitches') = i;
    }
}
```

The code above is a simple loop scenario. Using the previously discussed “operation” script, the output1 connector from the operation script, is connected to the reqStitches input connector on this new “stitches” script. This value is used to provide a limit to the loop; number of times the loop will execute.

Execution of the loop begins with the line: `for(i=0; i<reqStitches; i++)`

The loop statement as it is called, sets the scenario up with some rules to follow. Most loops will use the variable “i” as a placeholder/counter for the loops. You should start your loop counter variable as 0 to begin with. The semicolon represents end of a line of code, but we string together the entire loop rules into one physical line of code. Next rule the loop has to follow is an upper limit to count to. The value for reqStitches is our upper limit. Since we started at 0, we can utilize the “less than” operator to tell the loop to exit or finish when the value of “i” reaches the reqStitches value(upper limit). The last rule of the loop statement tells the script to increment (++) the value of “i” each time the loop begins. Once the ground rules for how the loop will execute, the actual business end of the loop is created. In our simple example, the output connector named completeStitches is set to equal the value of “i”. Not so exciting an example, but now when you see a loop in script, you should be able to review the “rules” and determine what the loop is doing.

VBScript handles loops much the same way as JavaScript. The same code looks like this in VBScript:

```
' Name:      VBScript stitches
' Purpose:   demonstrate a simple loop in VBScript.
'           Hook the "vb operation's" output1 connector
'           to the reqStitches input connector on this script.
Sub OnComputeOutputs(params)
    Dim reqStitches, completeStitches
    reqStitches = params.ConValue("reqStitches")

    For i = 0 To reqStitches
        params.ConValue("completeStitches") = i
    Next
End Sub
```

If you examine the VBScript version carefully, you can calculate that by the way in which the loop rules work, it will actually loop 1 extra time compared to the JavaScript version. Although VBScript will usually start a loop counter in ordinal fashion (begin with 0), you may at times wish to use 1 as an initial counter value. Doing so would ensure you get the same results as a JavaScript loop would get.

If statement

Another code statement that you will see often is the “if” statement. This statement is used as a “test”, that results in a true or false condition. A typical use is illustrated below:

```
for(i=0; i<reqStitches; i++)
{
    params.ConValue('completeStitches') = i;
    if(i<10)
    {
        System.Alert("This is the " + i + "th iteration");
    }
}
```

By embedding the “if” statement inside the loop, the script will now arrive at this statement each time the loop iterates another value to “i”. The “if” statement has “rules” to follow. In this case, the rule is that the value of “i” must be less-than 10. If the condition is “true”, the “if” statement is executed. In this case, a System Alert pops up on screen and tells us "This is the " + i + "th iteration". We build text using the quotation marks and insert the value for “i” using the “+” signs. Test it out. You can see the iterations for first 10 cycles of the loop. Once the first 10 cycles are up, the “if” statement will have a condition that is false (i is greater than 10). When the condition is false, the “if” statement is ignored and the loop continues.

In VBScript, the same functionality is achieved by using the following code:

```
For i = 0 To reqStitches
    If i < 10 Then System.Alert("This is the " & i & "th iteration")
    params.ConValue("completeStitches") = i
Next
```

There are other varieties of the “if” statement. If you visit the links provided at beginning of this document, you can research and investigate some of them if desired.

Case Statement

The case statement is used to evaluate a variable against a defined set of conditions. The following example demonstrates the case statement:

```
// Purpose: demonstrate the case statement
// Called to execute the command
function Execute(params)
{
    var date = new Date();
    today = date.getDay();
    switch (today)
    {
        case 1:
            params.ConValue("TodayIs") = "Monday";
    }
}
```

```

        break;
    case 2:
        params.ConValue( "TodayIs" ) = "Tuesday";
        break;
    case 3:
        params.ConValue( "TodayIs" ) = "Wednesday";
        break;
    case 4:
        params.ConValue( "TodayIs" ) = "Thursday";
        break;
    case 5:
        params.ConValue( "TodayIs" ) = "Friday";
        break;
    case 6:
        params.ConValue( "TodayIs" ) = "Saturday";
        break;
    case 7:
        params.ConValue( "TodayIs" ) = "Sunday";
        break;
    default:
        params.ConValue( "TodayIs" ) = "Not a valid day.";
        break;
    }
}

```

As you can see from the beginning of the code in the script, a new variable is declared as “date”. This statement calls the computer’s clock and gathers the date from there. Second line below evaluates the date variable and extracts the number of day from it:

```
var date = new Date();
```

```
today = date.getDay();
```

Once the number of the day is known, the case statement begins by:

```

switch (today)
{
    case 1:
        params.ConValue( "TodayIs" ) = "Monday";
        break;

```

...

The switch as it is also referred to contains the predefined conditions and any lines of code that are to be executed if the condition is “true”. The cases are stacked and in our example run from the number 1 to the number 7. This covers all the days in a week. Which ever number value the variable “today” holds, will determine which case statement is “true”. It is considered good practice to also include a “default” case statement, which will cover any condition that is not covered by any of the other case statements.

In VBScripting, this scenario is called Select Case and the code looks like this:

```

' Name:      VBScript Case Example
' Purpose:   demonstrate the select case statement in VBScript.
Sub Execute(params)
    Dim TodayIs, a, b
    a = weekday(date())
    Select Case a
        Case 1
            params.ConValue( "TodayIs" ) = "Sunday"
        Case 2
            params.ConValue( "TodayIs" ) = "Monday"

```

```

Case 3
params.ConValue("TodayIs") = "Tuesday"
Case 4
params.ConValue("TodayIs") = "Wednesday"
Case 5
params.ConValue("TodayIs") = "Thursday"
Case 6
params.ConValue("TodayIs") = "Friday"
Case 7
params.ConValue("TodayIs") = "Saturday"
Case Else
params.ConValue("TodayIs") = "Not a valid day"
End Select
End Sub

```

One other important item to note in this case and indeed in other cases of differences between scripting languages, VBScript begins on a Sunday, where jScript begins on a Monday. When you run into differences in expected values, it may very well be a difference in how the script languages handle the same scenario. You should be able to narrow down the problem and change your code to get the same result as you would in the other language.

The “if” and “case” statements are also considered ways to control a program/script’s flow. Anytime you can “trap” a condition that happens as the script is running, you can control what will happen via script code. Hard to imagine this example as being useful from a trueSpace scripting point-of-view? What if we were to add 2 new attributes to the jScript command object called “Case Example”?

```

// Name:      Case Example with color.
// Purpose:   Show case statement in a trueSpace
//            related scenario.
//            Each day of week has a different
//            color associated with it. Each
//            case assigns a different color for different day.
function Execute(params)
{
    var dayColor = params.ConValue('dayColor');
    var date = new Date();
    today = date.getDay();
    switch (today)
    {
        case 1:
            params.ConValue("Today Is") = "Monday";
            dayColor.SetRed(0.0);
            dayColor.SetGreen(0.0);
            dayColor.SetBlue(1.0);
            break;
        case 2:
            params.ConValue("Today Is") = "Tuesday";
            dayColor.SetRed(1.0);
            dayColor.SetGreen(1.0);
            dayColor.SetBlue(0.0);
            break;
        case 3:
            params.ConValue("Today Is") = "Wednesday";
            dayColor.SetRed(1.0);

```

```

        dayColor.SetGreen(0.0);
        dayColor.SetBlue(0.0);
        break;
    case 4:
        params.ConValue("Today Is") = "Thursday";
        dayColor.SetRed(0.5);
        dayColor.SetGreen(0.0);
        dayColor.SetBlue(1.0);
        break;
    case 5:
        params.ConValue("Today Is") = "Friday";
        dayColor.SetRed(0.0);
        dayColor.SetGreen(1.0);
        dayColor.SetBlue(0.0);
        break;
    case 6:
        params.ConValue("Today Is") = "Saturday";
        dayColor.SetRed(1.0);
        dayColor.SetGreen(0.5);
        dayColor.SetBlue(0.0);
        break;
    case 7:
        params.ConValue("Today Is") = "Sunday";
        dayColor.SetRed(0.0);
        dayColor.SetGreen(0.0);
        dayColor.SetBlue(0.0);
        break;
    default:
        params.ConValue("Today Is") = "Not a valid day.";
        dayColor.SetRed(1.0);
        dayColor.SetGreen(1.0);
        dayColor.SetBlue(1.0);
        break;
    }
    params.ConValue('outColor') = dayColor;
}

```

With VBScript example code looks like this:

```

' Name:      VBScript Case Example
' Purpose:   demonstrate the select case statement in VBScript.
Sub Execute(params)
    Dim dayColor, TodayIs, a, b, c, outColor
    Set dayColor = params.ConValue("dayColor")
    a = weekday(date())
    Select Case a
        Case 1
            params.ConValue("TodayIs") = "Sunday"
            dayColor.SetRed(0.0)
            dayColor.SetGreen(0.0)
            dayColor.SetBlue(1.0)
        Case 2
            params.ConValue("TodayIs") = "Monday"
            dayColor.SetRed(1.0)
            dayColor.SetGreen(1.0)
            dayColor.SetBlue(0.0)

```



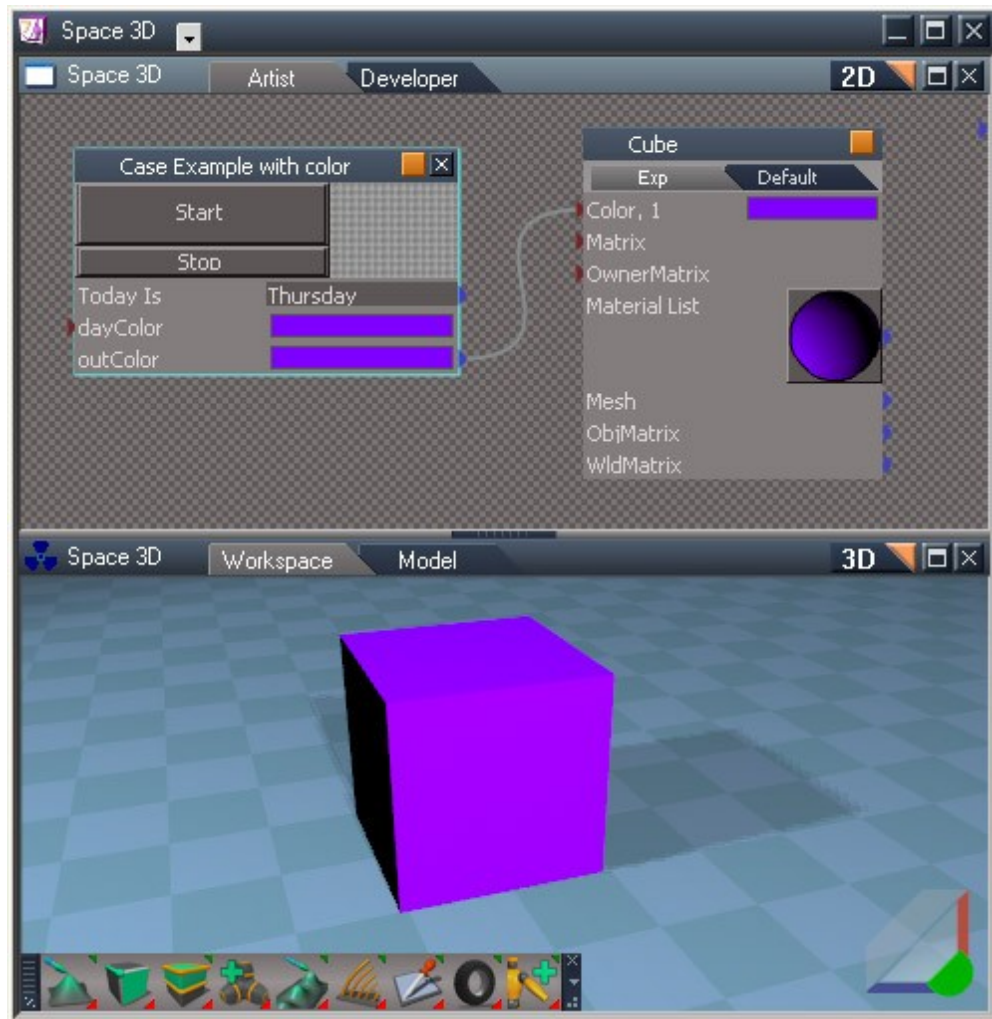
```

Case 3
params.ConValue("TodayIs") = "Tuesday"
dayColor.SetRed(0.0)
dayColor.SetGreen(0.0)
dayColor.SetBlue(0.0)
Case 4
params.ConValue("TodayIs") = "Wednesday"
dayColor.SetRed(1.0)
dayColor.SetGreen(0.0)
dayColor.SetBlue(0.0)
Case 5
params.ConValue("TodayIs") = "Thursday"
dayColor.SetRed(0.5)
dayColor.SetGreen(0.0)
dayColor.SetBlue(1.0)
Case 6
params.ConValue("TodayIs") = "Friday"
dayColor.SetRed(0.0)
dayColor.SetGreen(1.0)
dayColor.SetBlue(0.0)
Case 7
params.ConValue("TodayIs") = "Saturday"
dayColor.SetRed(1.0)
dayColor.SetGreen(0.5)
dayColor.SetBlue(0.0)
Case Else
params.ConValue("TodayIs") = "Not a valid day"
dayColor.SetRed(1.0)
dayColor.SetGreen(1.0)
dayColor.SetBlue(1.0)
End Select
params.ConValue("outColor") = dayColor
End Sub

```

You should notice a few subtle differences in the VBScript version when compared to the jScript version. It is expected that the different languages are slightly different in how they do the same tasks. It is important that you realize that there are these differences. You may try to convert scripts from one language to another and knowing that there may be differences that must be considered, will make your successes increase.

Included in the Dev Guide library, you will find an object named Cube. Drag the Cube into the Link Editor. Connect the outColor attribute on the script object to the Color, 1 input connector as shown in the image below.



Case Example controls Cube's color

Once you set the scenario up as shown above, every day of the week will yield a new color for the Cube object. If you check out the script and review the Attributes tab of the Script Editor, you will notice that 2 new attributes were added; dayColor and outColor. Both are of the same type, namely “Common Data Package/Color Data”. Basically, you see how to “set” color in the script. By setting values for Red, Green and Blue, we can create the colors we want. If you check carefully the color values set for each day, you will see that the changes are very simple with values of 0.0, 0.5 and 1.0 mixed together to get distinctive colors. Last task in the new script is to set the output value outColor to the same value as dayColor. We are then able to use this output connector to hookup to the Cube's Color, 1 input connector.

Arrays

Arrays are a very powerful feature in trueSpace scripting environment. There is an assortment of predefined array types in the Common Data Package:

- String Enum Data: special array to populate a Combo Box.
 - [http://msdn2.microsoft.com/en-us/library/bb775796\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/bb775796(VS.85).aspx)
 - Alternately can be used as a single dimensional string array.
- Universal Array Data: special array type that can contain any type of variable.

- Consider using only when necessary.
 - Not as efficient as specialized array types. More memory consumption and slower as well.
- Int Array Data: special array to hold integers.
- Number Array Data: special array to hold numbers.
- String Array Data: special array to hold string values.
- Boolean Array Data: special array to hold Boolean values.

In trueSpace, there are two different flavors of arrays. The first type of array would be termed as trueSpace specific arrays and the second flavor would be regular jScript or VBScript arrays. The reason you would use one or the other, is if you wish to have the array “live” in an input/output connector. The same theory applies to variables as well; if you wish to place value of a variable as a connector, you can tailor the variable using one of trueSpace’s built-in data types. If the variable is never going to be placed on a connector, there is no need to use the trueSpace specific flavor. Just a regular flavor variable or array will suffice.

trueSpace special functions for arrays:

- Clear(): clear (initialize) the data object.
- SetDim(): Set dimension of the array (default is 1).
- GetDim(): Get dimension of the array.
- SetSize(): Set array size
 - One number for each dimension.
 - Note: Add() method automatically increases the size of a one-dimensional array.
 - Array.SetDim(3) can be sized using Array.SetSize(10, 5, 10)
- GetSize(): Get array size
 - One number for each dimension.
 - Array.GetSize(0) gets size of first dimension. Array index is zero based.
 - Array.GetSize(1) gets size of the second dimension.
- SetAt(): sets one item to the array at the given position.
 - Remember that array indices are zero based.
 - Array.SetAt(0, “first string”) would place item at first element in one-dimensional array.
 - Array.SetAt(1, 0, “second string”) sets item in second dimension, first element.
- GetAt(): gets the item at the given position.
 - myItem = Array.GetAt(5) would retrieve the item at location 5(6th position) of one-dimensional array.
 - myOtherItem = Array.GetAt(1, 4) retrieves the item in second dimension location 4 (5th position) of a multi-dimensional array.
- Add(): used for one-dimensional array only.
 - Will increase the size of the array to add the item, otherwise you should use SetAt() to overwrite a value in array.
 - By using Add(), you append the array at the end.
- InsertAt(): used for one-dimensional array only.
 - Will increase the size of the array and insert the item at the specified location.
 - Different from Add(), which will append to end of array. InsertAt() allows you to place the item in other than end element (appended) position.

- RemoveAt(): used for one-dimensional array only.
 - Removes the specified element, decreasing the size of the array.

Single dimensional Arrays

In this first example below, an array “a” is created in memory as a trueSpace specific flavor:

```
// Create simple array by adding items
// Execute
// Called to execute the command
function Execute(params)
{
    var a = System.CreateDO("Common Data Package/Universal Array Data");
    a.Add("first string");
    a.Add("second string");
    a.Add(6.56);
    System.Alert(a.GetSize());
    for(i = 0; i < a.GetSize(); i++)
    {
        System.Alert(a.GetAt(i));
    }
}
```

Once the array is declared on that first line of code using System.CreateDO, the array may be accessed using trueSpace special functions.

The same script scenario using regular flavor jScript array, would look like:

```
// Execute
// Called to execute the command
function Execute(params)
{
    var a = new Array();
    a[0] = "first string";
    a[1] = "second string";
    a[2] = 6.56;
    System.Alert(a.length);
    for(i = 0; i < a.length; i++)
    {
        System.Alert(a[i]);
    }
}
```

You can see the differences between the two examples. As a script author in trueSpace, you have to be aware that you are working with possibility of having one foot in each world (trueSpace arrays or reg jScript or VBScript arrays). You can use both in your scripts. Just remember there are different ways to access each flavor of array!

Here are the VBScript examples of this simple “add” type scenario:

```
' Create simple array by adding items
' Execute
' Called to execute the command
```

```

Sub Execute(params)
    Dim a
    Set a = System.CreateDO("Common Data Package/Universal Array Data")
    a.Add("first string")
    a.Add("second string")
    a.Add(6.56)
    System.Alert(a.GetSize())
    size = a.GetSize()
    For i = 0 To Cint(size)-1
        System.Alert(a.GetAt(i))
    Next
End Sub

```

Notice that jScript and VBScript examples using trueSpace methods look very similar. Now in this next VBScript example we delve into the non-trueSpace flavor of array:

```

' Execute
' Called to execute the command
Sub Execute(params)
    Dim a
    a = Array("first string", "second string", 6.56)
    System.Alert(UBound(a))
    For i = 0 To UBound(a)
        System.Alert(a(i))
    Next
End Sub

```

In VBScript, you could also author the script like so: difference is the initial Dim a(2), setting size of array first:

```

' Execute
' Called to execute the command
Sub Execute(params)
    Dim a(2)
    a(0) = "first string"
    a(1) = "second string"
    a(2) = 6.56
    System.Alert(UBound(a))
    For i = 0 To UBound(a)
        System.Alert(a(i))
    Next
End Sub

```

Again there is a difference in how VBScript handles its own native arrays. If you feel comfortable working with one foot in each world, you can research and figure out how to script it all. The important concept to understand is how both flavors of array can co-exist if you script accordingly.

Multi-dimensional Arrays

In addition to single dimensional arrays, trueSpace also allows for multidimensional arrays.

```

// Create 2 dimensional integer array
// Execute
// Called to execute the command

```

```

function Execute(params)
{
    a = System.CreateDO("Common Data Package/Int Array Data");
    a.SetDim(2);
    a.SetSize(5,5);
    for(i = 0; i < a.GetSize(0); i++)
    {
        for(j = 0; j < a.GetSize(1); j++)
        {
            b = i+j;
            a.SetAt(i,j,b);
            System.Alert("Array:  " + i + "\nElement:  " + j + "\nValue:  "
+ a.GetAt(i, j));
        }
    }
}

```

Using a trueSpace flavor array, the scenario is easy enough to understand the script code above. Because we are moving from single dimension, we SetDim() telling trueSpace how many dimensions this array will have. Using the SetSize() function, the size of both dimensions are set to 5.

Using two “for” loops, the script first sets the array column using the variable “i”. Once inside the first loop, another “for” loop is used to move through the second dimension using the variable “j”. Once inside the second dimension, the variable “b = i + j” creates a value we will store in that element. The value is then stored in the element using the SetAt() function. Notice it is easy to reference each element of the array. “a.SetAt(i, j, b). The first two values in parenthesis point to the array element, while the last value is the actual value to be stored.

In order to test the multidimensional array, the System.Alert() function was used with some creative formatting, to send you an alert with the information as illustrated below. Each time the loop executes you get another alert until the looping starts. You can see the progress of the script and note that each element is being populated.

Here is version of the same script, using jScript native multidimensional array (array of arrays):

```

// Execute
// Called to execute the command
function Execute(params)
{
    a = new Array();
    for(i = 0; i < 5; i++)
    {
        for(j = 0; j < 5; j++)
        {
            a[i] = new Array(a.length);
            b = i + j;
            a[i][j] = b;
            System.Alert("Array a:  " + i + "\nArray b:  " + j + "\nValue:
" + a[i][j]);
        }
    }
}

```

```
}
```

With JavaScript native array, you have to create arrays within arrays in order to accomplish the same functionality. Notice in the second for loop, `a[i] = new Array(a.length);`. This is a declaration of a new array inside an existing array. It can be a bit confusing if you are creating a multidimensional array larger than say 3 or 4! Each new dimension you add, requires creation of new arrays. The same scenario using trueSpace flavor arrays makes for much easier control and understanding of the array creation and use.

VBScript Multidimensional array using a trueSpace Data object:

```
' Execute
' Called to execute the command
Sub Execute(params)
    Dim a,c,b
    Set a = System.CreateDO("Common Data Package/Int Array Data")
    a.Clear()
    a.SetDim(2)
    a.SetSize(5),(5)
    c=0
    For i = 0 To Cint(a.GetSize([1]))-1
        For j = 0 To Cint(a.GetSize([2]))-1
            b = i + j
            a.SetAt(i),(j),(b)
            System.Alert("1st Dim Size: " & Cint(a.GetSize([1])) &_
                Chr(13) & Chr(10) &_
                "2nd Dim Size: " & Cint(a.GetSize([2])) &_
                Chr(13) & Chr(10) &_
                "cells = " & c &_
                Chr(13) & Chr(10) &_
                "1st Dim = " & i &_
                Chr(13) & Chr(10) &_
                "2nd Dim = " & j &_
                Chr(13) & Chr(10) &_
                "Cell Value = " & a.GetAt(i,j))
            c=c+1
        Next
    Next
End Sub
```

In the script scenario above, the array variable “a” is “Set” to a trueSpace specific data object; Common Data Package/Int Array Data. By creating the array in this fashion, we are able to use trueSpace specific calling methods like `GetSize()`, which if you notice, uses square brackets [] to signify that a memory array is being referenced. It is perhaps a more difficult to understand, however it is possible and sometimes necessary to format data as a trueSpace specific data object, as is discussed below under Arrays on Connectors.

Although there is a little extra work to creating and using a VBScript multidimensional array in this fashion, it does work and should prove useful where such a memory array is required. The `System.Alert()` is formatted for easy reading in script code. The “&_” tells the script engine to include next line and the “`Chr(13) & Chr(10)`” tells the alert box printout to add a character return and a line-feed combination.

Arrays on Connectors

Up until this point in time we have been using only “memory” resident arrays. Both memory and persistent arrays serve useful purposes. There will be scenarios when data in an array is required to be warehoused/stored. Just as other trueSpace data objects (input/output connectors) can store values, so can an input/output connector that is set-up as an array. This is most useful when you wish to make data available to other scripts or to persist (remain alive) when the script is saved to a library or perhaps shared with friends.

In order to set up such a scenario, some initial work is required. You plan out your script and what it will do like so:

Purpose: gather Red, Green and Blue color data (RGB) from a Common Data Package/Color Data object (actually an array as we show later).

Required Attributes:

ColorOut: output connector: Common Data Package/Color Data.

GetRed: input connector: number: value of Red in ColorOut.

GetGreen: input connector: number: value of Green in ColorOut.

GetBlue: input connector: number: value of Blue in ColorOut.

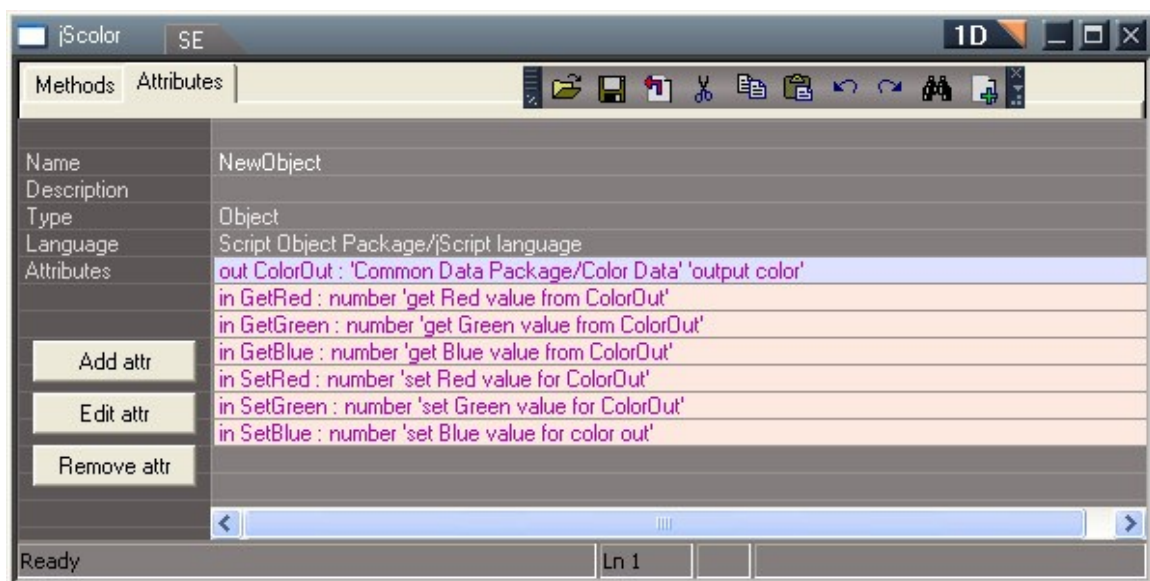
SetRed: input connector: number: set value of Red in ColorOut.

SetGreen: input connector: number: set value of Green in ColorOut.

SetBlue: input connector: number: set value of Green in ColorOut.

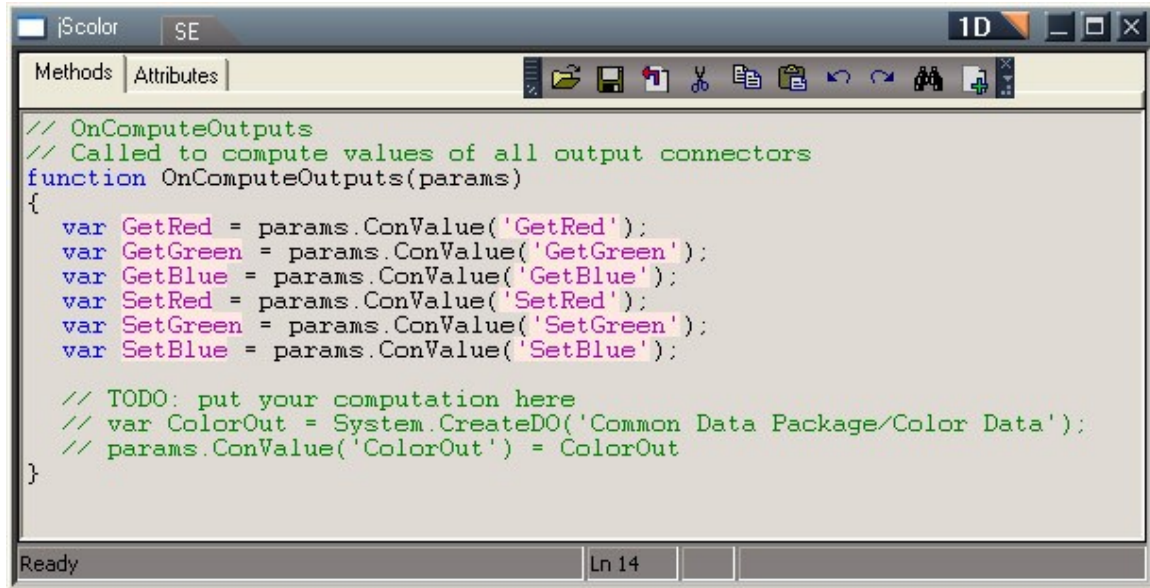
Earlier we mentioned that the Common Data Package/Color Data was actually an array of data. How do we know this? Included with the Development SDK, there are several reference documents, which outline trueSpace Script and Data objects. These html based references allow you to for instance look up the Color Data. In the Class Hierarchy, you would find IRdColorDisp. The IRdColorDisp Struct Reference shows a list of Public Member Functions, which is where the information was gathered to decide which attributes we could use in this little project. To keep the scenario relatively simple, we are only getting and setting values for RGB. If you were to look at the IRdColorDisp Struct Reference, you would find much more than RGB is actually stored in the Color Data object! It is easy enough to determine that the Color Data object is a single dimension array of color data.

Once a plan is organized for a script, the script is created and the Attributes-tab of the Script Editor gets populated by the planned attributes. Once all the attributes have been added, the Attributes-tab of the Script Editor looks like this:



jsColor Attributes-tab

Once the attributes have been added, the Methods-tab of the jsColor script is populated with trueSpace generated script and looks like this:



jsColor Methods-tab

The first task is to edit the default script to look like this:

```
// OnComputeOutputs
// Called to compute values of all output connectors
function OnComputeOutputs(params)
{
    var ColorOut = System.CreateDO('Common Data Package/Color Data');
    var GetRed = params.ConValue('GetRed');
    var GetGreen = params.ConValue('GetGreen');
    var GetBlue = params.ConValue('GetBlue');
    var SetRed = params.ConValue('SetRed');
    var SetGreen = params.ConValue('SetGreen');
    var SetBlue = params.ConValue('SetBlue');

    // TODO: put your computation here

    params.ConValue('ColorOut') = ColorOut;
}
```

The declaration of ColorOut and creation of a memory array is moved to the top of the function:

```
ColorOut = System.CreateDO('Common Data Package/Color Data');
```

The comment characters // were removed to make the line of code execute as desired. The second change made involved removing comment characters from the last line of code:

```
params.ConValue('ColorOut') = ColorOut;
```

After removing the comment characters, this line of code will also now execute as desired. With the OnComputeOutputs function (default), the script only executes if an output connector must be calculated because of a change that occurs in script. In order to instantiate(initialize) this script, we must create such a condition in script, to cause a change in our only output connector ColorOut. We do this by creating code:

```
// OnComputeOutputs
// Called to compute values of all output connectors
function OnComputeOutputs(params)
{
    var ColorOut = System.CreateDO('Common Data Package/Color Data');
    var GetRed = params.ConValue('GetRed');
    var GetGreen = params.ConValue('GetGreen');
    var GetBlue = params.ConValue('GetBlue');
    var SetRed = params.ConValue('SetRed');
    var SetGreen = params.ConValue('SetGreen');
    var SetBlue = params.ConValue('SetBlue');

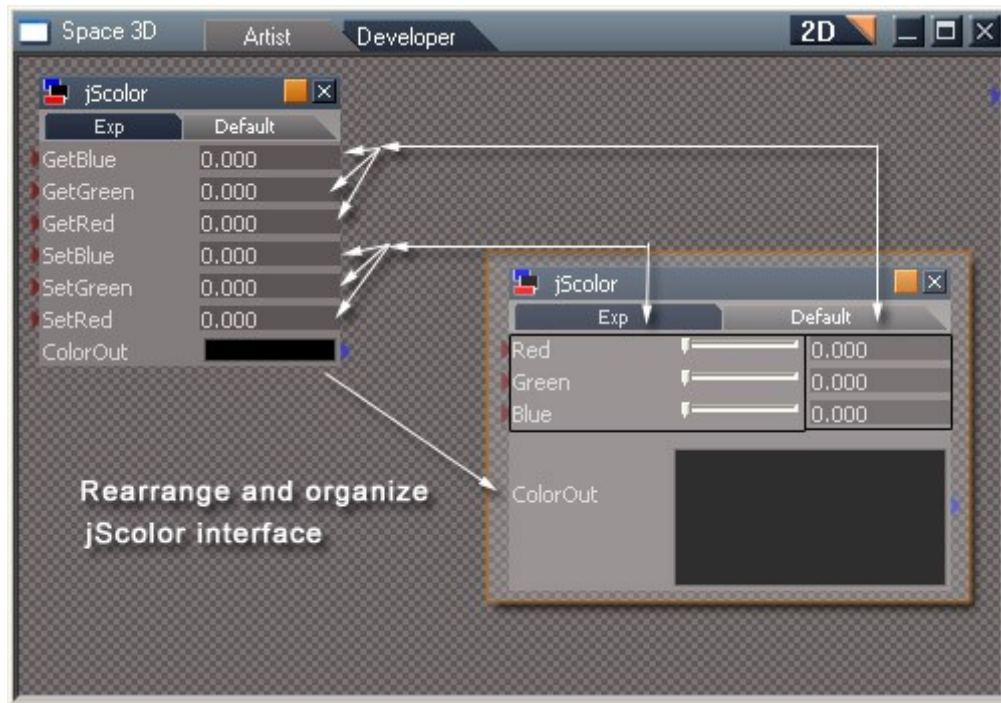
    // Change values for ColorOut:
    ColorOut.SetRed(SetRed);
    ColorOut.SetGreen(SetGreen);
    ColorOut.SetBlue(SetBlue);

    // Pass the ColorOut value to output connector:
    params.ConValue('ColorOut') = ColorOut;

    // Once output connector has its value,
    // we can get values and pass them to input connectors:
    params.ConValue('GetRed') = ColorOut.GetRed();
    params.ConValue('GetGreen') = ColorOut.GetGreen();
    params.ConValue('GetBlue') = ColorOut.GetBlue();
}
```

Now that the logic behind the script is set-in-place, it is time to exit the Script Editor and examine the script interface. By default, the jScolor interface looks like the upper-left interface in the image below. Using the existing elements of the interface, it is organized and rearranged for ease of use.

- The slider controls were originally numeric values for SetRed, SetGreen and SetBlue. Just changed the type to slider on existing elements.
 - The slider-controls can be edited to set minimum and maximum values for the sliders.
 - Because the color values we are accessing are number values between 0.000 and 1.000, these were used to control min/max slider values.
 - Renamed SetRed, SetGreen and SetBlue elements to: Red, Green and Blue respectively.
- Numeric values for GetRed, GetGreen and GetBlue were moved to their respective positions on right-side of Red, Green and Blue sliders.

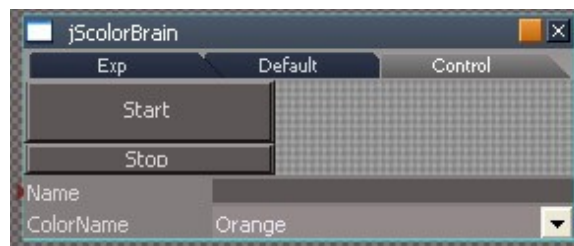


Organize the jSColor interface

Although this is a very basic scenario, it does illustrate how to manipulate data in a single dimensional array as well as how to retrieve values from the array on the connector. The GetRed, GetGreen and GetBlue values are being populated by actual data warehoused in the array, on the ColorOut output connector.

This script is included in the Dev Guide library. As you move the RGB sliders, you change the color swatch and as the color swatch changes color, the numeric values for RGB change. So the script actually tests the script and ensures that the data exchange is working properly.

Using this jSColor object, we can expand this scenario to one that will utilize a multidimensional array and also add in a ComboBox control that serves as a drop-down list (single dim array).



jSColorBrain panel/interface

The jSColorBrain performs a task. By breaking away to a separate command script object, this scenario will show you how to access data from the jSColor script object. The task that the jSColorBrain performs is simple enough to understand, yet opens up a whole new area of possibilities. Here is script code (heavily commented):

```

// Execute
// Called to execute the command
function Execute(params)
{
    //Instantiate/initialize the arrays: remove after initial use.
    //var ColorArray = System.CreateDO("Common Data Package/String Array
Data");
    //var ColorName = System.CreateDO("Common Data Package/String Enum
Data");
    //ColorArray.SetDim(2);

    // Once instantiated, we can use the arrays and associated vari-
ables:
    var ColorArray = params.ConValue("ColorArray"); // array for name
and other values.
    var ColorName = params.ConValue("ColorName"); // drop-down list.

    var Name = params.ConValue('Name'); // new name for new color.
    // TODO: put your action code here
    //If user clicks on Start button with an empty Name:
    if(Name == "") // name is empty.
    {
        System.Alert("Please enter a name for the color"); // alert the
user.
        params.SetTerminationFlag(); // terminate the execution of code.
        return; // go to beginning of function and wait for start button to
get pressed.
    }
    else // user has entered a new name. Does name already exist?:
    {
        nSize = ColorName.GetSize(); // get size of array and store it in
variable nSize.
        for(i = 0; i < nSize; i++) // loop through array, comparing names
that exist:
        {
            if(Name == ColorName.GetStringAt(i)) // if name exists:
            {
                params.ConValue('Name') = ""; // set name field on interface
to empty.
                System.Alert("Name already exists,\nPlease choose another.");
// alert the user.
                params.SetTerminationFlag(); // terminate the execution of
code.
                return; // go to beginning of function and wait for start but-
ton to get pressed.
            }
        }
    }
    // If the script runs to this point, the name is not found in the
existing array.
    // Begin to add this new color to the array and its name to the
drop-down list:
    nSize = nSize + 1; // increase array by 1.
    ColorName.SetStringAt(nSize - 1, Name); // add Name to drop-down
list.

```

```

    ColorArray.SetSize(nSize, 4); // increase the ColorArray by 1 in
first dimension.
    ColorArray.SetAt(nSize-1, 0, Name); // add name to first position in
array.
    // harvest RGB values from the jSColor script object:
    owner = System.ThisOwner(); // path of the encapsulator object: Col-
orArray
    target = owner + "/jSColor"; // path of the jSColor in relation to
ColorArray
    ColorArray.SetAt(nSize-1, 1, Node.Value(target, 'GetRed')); // set
Red value found to specified position.
    ColorArray.SetAt(nSize-1, 2, Node.Value(target, 'GetGreen')); // same
here for Green.
    ColorArray.SetAt(nSize-1, 3, Node.Value(target, 'GetBlue')); // last
do same for Blue.

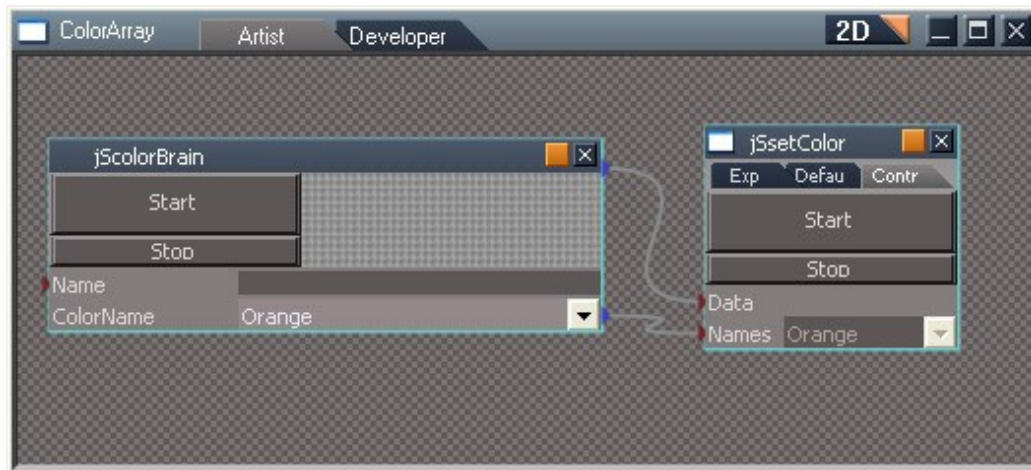
    params.ConValue("ColorArray") = ColorArray; // updated array put back
to output connector.
    params.ConValue("ColorName") = ColorName; // updated drop-down list
to output connector.
    var test = ColorName.GetSelectedString();
    System.Alert(ColorName.GetSelectedString());
    params.ConValue("ColorArray") = ColorArray;
    params.ConValue("ColorName") = ColorName;
    // Set the selected color name in ComboBox to new color name:
    for(i = 0; i < nSize; i++)
    {
        if(ColorName.GetStringAt(i) == Name) // if same then do this:
        {
            ColorName.SetSelectedString(ColorName.GetStringAt(i))
        }
    }
    params.ConValue('Name') = ""; // set name field to empty.
}

```

By reviewing the code, you should be able to tell that the script was prepared for use of arrays and then commented out. Once the script was set up for arrays, the logic was added so that the user will enter a name after setting RGB values on the jSColor script object.

User hits start button. The name is checked for emptiness and alerts the user if no name entered. If the name field is filled in with a name, then the name is checked against the data in the ColorName array to see if the name exists. If the name exists, the user receives an alert, the name field is emptied and the user tries again. If the name is verified as being “new”, it is added to the ColorName array and values for RGB as well as name of new color, are added to the ColorArray, which holds 4 pieces of data for each color: name and RGB values.

At this stage, single and multidimensional arrays have been created, covered and then used. The scenario at this point is limited in what it can do. A final piece of the scenario must be created to round-out the functionality of this concept with color values and names. We can set colors and give them names, populate a drop-down list with the names of the colors and also store RGB values to an array. Now what if user was able to select a name in the drop-down list and have the color swatch and RGB values show up on the jSColor script object? That would round-out nicely this example of scripting.

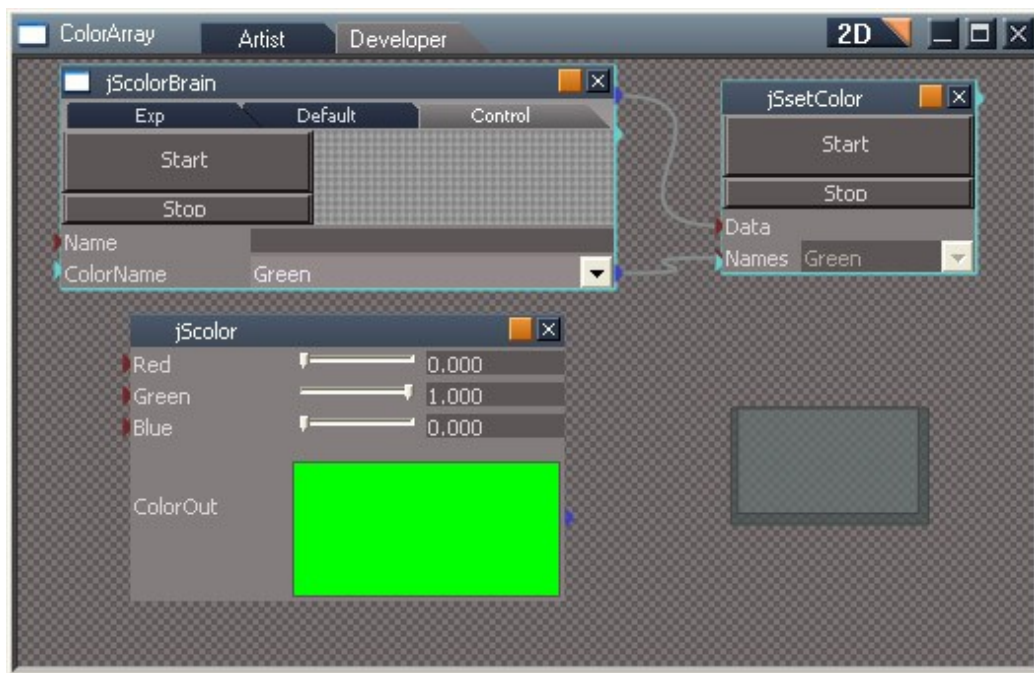


jSsetColor panel/interface on right

Again we are breaking off with a new script command object to handle this task. The jSsetColor has two input connectors, which are set up as arrays to take data from the jSColorBrain's existing arrays. Once the connections are made, the manipulation of data can begin! In this last script command, you will note we did not have to instantiate the arrays. Connecting them to the already established arrays on the jSColorBrain script does this for us. Basically, the script finds the name that is selected in the drop-down list, in the Data array. When the name matches, the RGB values from the Data array are passed directly to the jSColor attributes for getting and setting color values.

```
// Execute
// Called to execute the command
function Execute(params)
{
    var Names = params.ConValue('Names');
    var Data = params.ConValue('Data');
    var owner = System.ThisOwner();
    var target = owner + "/jSColor";
    for(i=0; i < Names.GetSize(); i++)
    {
        // find name of color in Data array:
        if(Data.GetAt(i,0) == Names.GetSelectedString())
        {
            // Set values on jSColor
            Node.Value(target, 'SetRed') = Data.GetAt(i,1);
            Node.Value(target, 'SetGreen') = Data.GetAt(i,2);
            Node.Value(target, 'SetBlue') = Data.GetAt(i,3);
            Node.Value(target, 'GetRed') = Data.GetAt(i,1);
            Node.Value(target, 'GetGreen') = Data.GetAt(i,2);
            Node.Value(target, 'GetBlue') = Data.GetAt(i,3);
        }
    }
}
```

There you have a great round-trip through trueSpace arrays. There is a tremendous amount of power to be harnessed by using arrays in trueSpace scripting. The scripts for this latest example all located in the Dev Guide library in a single object called ColorArray. Just drag and drop the ColorArray object into link editor and enter it.



ColorArray encapsulates the 3 scripts