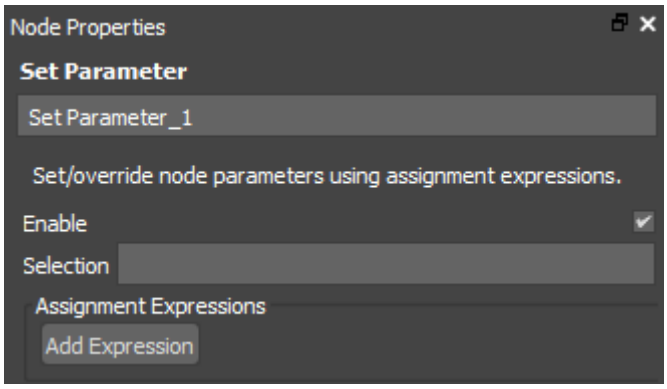
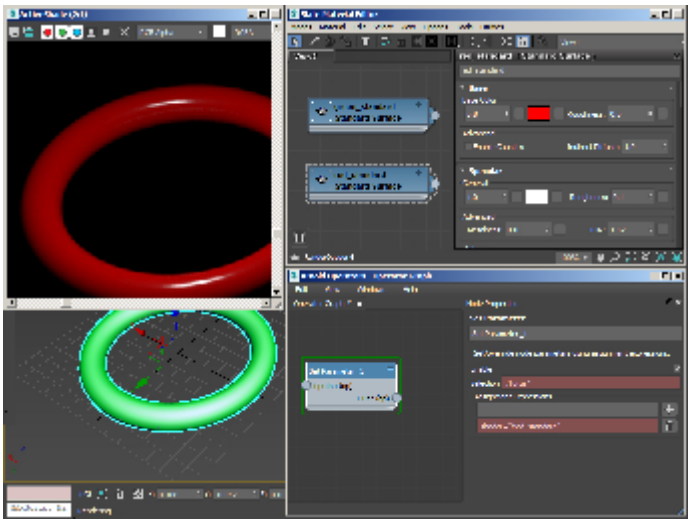
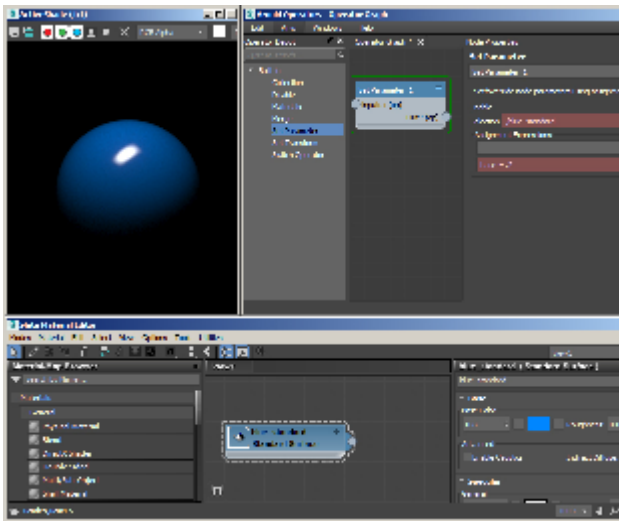
Set Parameter



Set/override node parameters using assignment expressions.



Green shader replaced with red shader



Blue Base set to 0.2

Enable

Enable/disable the *operator*. Disabled operators are bypassed when the *operator* graph is evaluated.

Selection

An expression to select which nodes this operator will affect. The expression syntax is described in the selection expression documentation, with some examples. Note that if the *operator* is connected to a *procedural* the selections are assumed to be relative to the procedural's namespace.

Assignment Expressions

An expression to select which nodes this operator will affect. The expression syntax is described in the selection expression documentation, with some examples. Note that if the *operator* is connected to a *procedural* the selections are assumed to be relative to the procedural's namespace.

The *set_parameter* operator takes advantage of assignment expressions to set and override node parameters. An assignment expression is made up of:

Assignment Expression Format

```
[type declaration (optional)] [parameter name]
[assignment operation] [value]
```

The type declaration is only necessary when specifying a new user parameter, where the available types are:

- `bool, byte, int, uint, float, rgb, rgba, vector, vector2, string, matrix, node`

Examples:

- `subdiv_type = 'catclark'`
- `shader = 'my_shader_node'`
- `shader = '' (Setting an empty string will remove the shader assignment)`
- `float a = 1.5`

Defining Arrays

Arrays are declared using square brackets, e.g. `float[3]`

It's not necessary to define the array size, where `float[] = [1 2 3]` will automatically declare a float array with 3 entries.

Value and numerical operators

A value is made up of one or more operands and operators as well as optional parenthesis scoping, which together forms a numerical expression such as:

- `3 + 4 * 2 / (1 - 5) ^ 2 ^ 3`

The supported operators work on numerical types but they can't mix types. There are two exceptions:

- Every numerical value can be multiplied/added/etc. with a single value (see division and power examples in the table below)
- Modulus only works with integer types

Description	Operator	Examples
Addition	`+`	`float: 3.0 + 4.0, vector: [1 2 3] + ([8 5 9] + [3 2 1])`
Subtraction	`-`	`int: -4 - (-2)`
Multiplication	`*`	`vector2[]: [[1 2][3 4]] * [[2 3]]`
Division	`/`	`matrix: [1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6] / 2`
Power	`^`	`rgb: [1 0.5 0.2] ^ 2`
Modulus	`%`	`int: 5 % 3`

Assignment Operations

Apart from the standard assignment operator '=', every supported numerical operator has a corresponding assignment operator, forming the set of supported operations:

- `=, +=, -=, *=, /=, ^=, %=`

Literal Strings

The following parameter types are treated as literal strings and need to be scoped using single quotes or escaped double quotes:

- `string/string[], enum/enum[], node/node[]`

This is to distinguish them from parameter references which are shown in 'Referring to other Parameters' below.

Reserved Strings

Reserved strings don't need quotes like literal strings and they supersede parameter references which are discussed in the next section.

- Boolean values are set using `True` or `False`.
- random() is a function which generates one or more uniform random values between 0 and 1 depending on how many values are needed for a given parameter we're assigning to, e.g.:
  - `float a = random()` (produces a single random value)
  - `rgb mix = random()` (produces an RGB value with 3 random numbers)
  - `matrix m = random()` (produces a matrix value with 16 random numbers)

- Different ranges can be generated by multiplying the random value. To make this more useful we would need to add at least a way to control the seed.

Referring to other Parameters

Any operand in a value can refer to any parameter that exists on the node, where the rules of mixing parameters discussed above apply. If a given string is not a literal string or a reserved string we look for a parameter on the node with the given name, and substitute the value with the corresponding parameter value, e.g. imagine we have an RGB user parameter 'mix' available to us:

- RGB: `[1 0 0] + mix * random()`

A parameter can be set/overridden using a referenced parameter of the same type as the RGB example above. However, we can also set a value like an RGB using one or more referenced float parameters. For instance, let's say the node has 2 float parameters 'a' and 'b'. Then an RGB parameter can be set using e.g.:

- RGB: `[a 0.5 b]`

We can also reference connected and linked parameters (not components) on the node we are cooking or on another node, wherein the latter case a #node_name is used, e.g.:

- `rgb ref_rgb = #some_shader.base_color`
- `int ref_int = #some_node.param_node_array[1].some_int`
- `matrix[] ref_matrix = #light.matrix`

Assignment Expression Examples

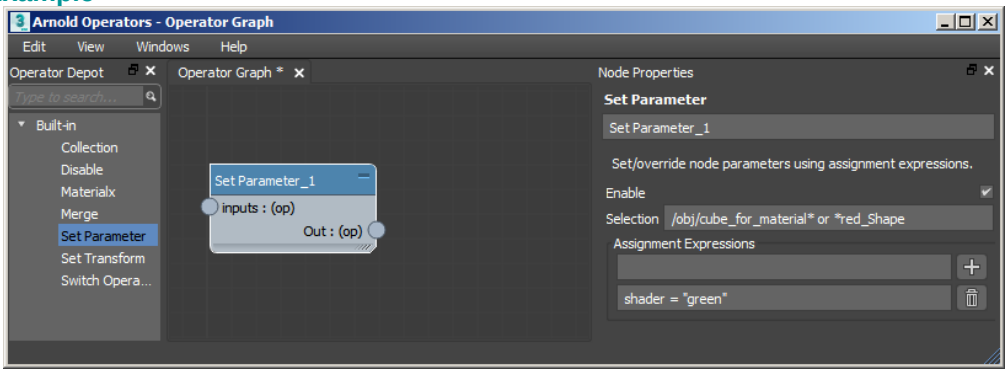
Built-in and predefined parameters

```
radius = 0.1
radius += 0.5
radius *= some_base_width / (random + 0.2)^2

matrix = [[1 0 0 0 0 1 0 0 0 0 1 0 0 -2 0 1]]

shader = 'some_shader_name'
transform_type = 'rotate_about_center'

opaque = False

ref_floats[2] = 100.0
ref_floats = [100.0 200 300]
float_array[3] = 10.0
vec2_array[0] = [10 10]
rgb_array[1] = [0.5 0.2 0.8]
desc[1] = 'replaced with something else'
```

Declaring new parameters

```
rgb white = 1
rgb green = [0 1 0]

string model = 'Cessna'
string[2] choices = ['penguin knees' 'balaclava stand']
string[] choices = ['penguin knees' 'balaclava stand']
string[] choices = [\"penguin knees\" \"balaclava
stand\"]

vector2[] limp = [[1 2] [3 4]] * 2
vector2[2] limp = [[1 2] [3 4]] * [[2 3]]

vector vel = [1 2 3] + ([8 5 9] * [3 2 1])^2

bool lasers = True
bool[] flags = [True False True False]

int look_variant_idx = counter % num_values

float[3] margins = [0.4 0.2 0.1] + [0.1 0.2 0.3]

matrix my_matrix = [1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6] +
(10 * random)
```

Procedural Example



In this example, we want to assign a shader used inside a *Procedural (Stand-in Maya/C4D)* to objects in the scene. In the *Procedural*, there is a shader called 'green' that works for overrides on objects inside the ass file since this green shader is in the same *Procedural.* The full path for the shader is used inside the *Procedural* for it to work for both the shapes in the ass file and the ones coming from the scene:

```
shader = "^/obj/arnold_procedural1/procedural^green"
```

You could choose a namespace for the *Procedural* by setting the namespace parameter on the Arnold *Procedural* object node in the Arnold parameters (to foo for example). Then you can use the following shader override which is more intuitive and has the same effect.

```
shader = "^foo^green"
```

Green shader from Procedural assigned to the cube in the scene