

Programming Deco Pattern Fill Scripts in Photoshop CS6 – User Guide

Radomír Měch

Advanced Technology Lab, Adobe Systems Incorporated

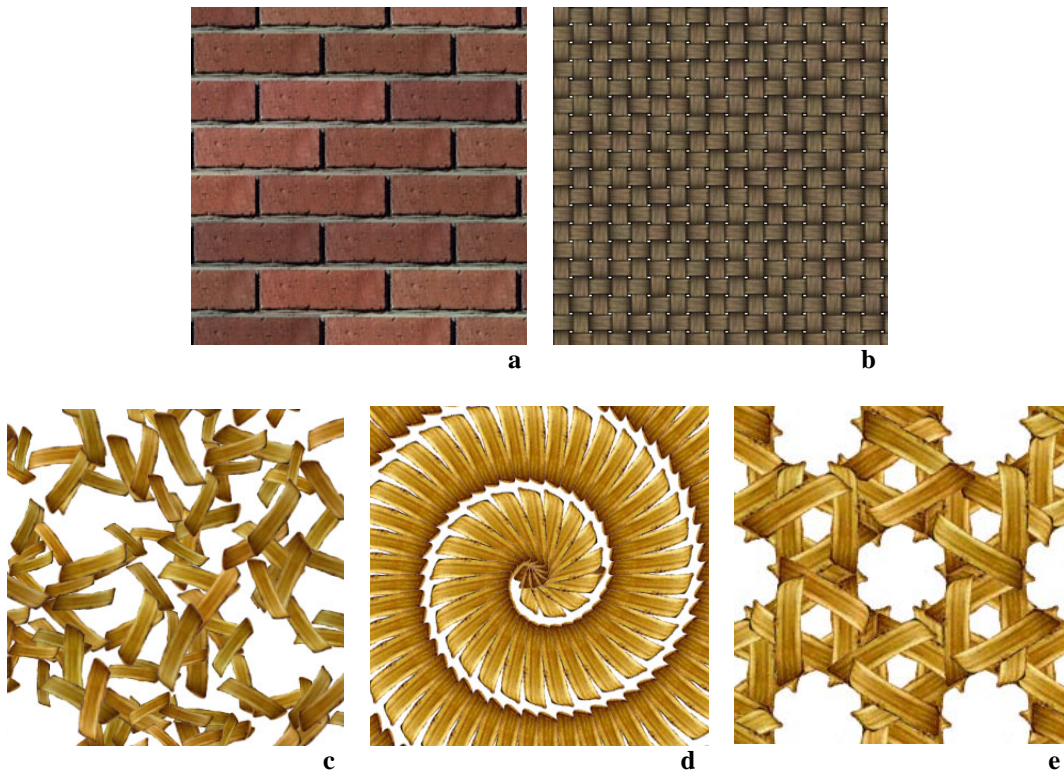
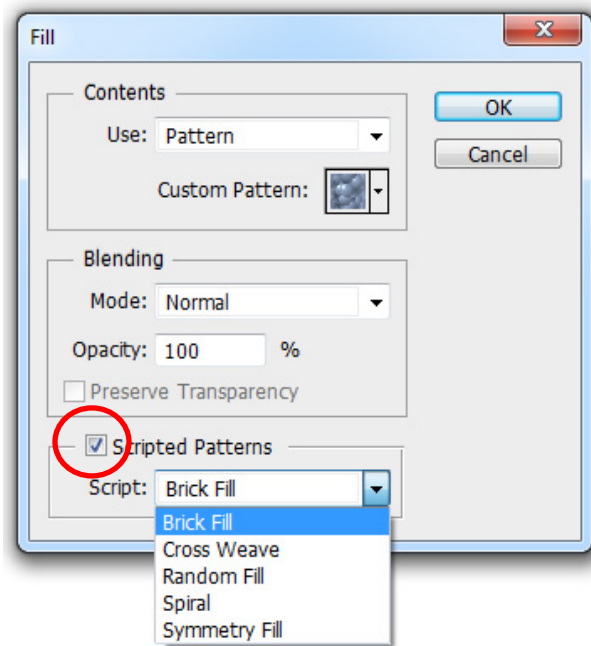


Figure 1: Five patterns generated by scripted Deco pattern fills in Photoshop CS6: Brick Fill (a), Cross Weave (b), Random Fill (c), Spiral (d), and Symmetry Fill (e).

1 Deco Pattern Fills in Photoshop CS6

Photoshop CS6 contains five scripted patterns that can be accessed by right clicking on a path, selecting *Fill*, or by selecting fill from the Edit Menu. Once you have the fill dialog box you choose *Pattern* in the *Use* selection box, and check the checkbox *Scripted Patterns*:



Photoshop CS6 includes five JavaScript files that define five distinct patterns. These patterns are executed by the Deco framework, which is a scriptable environment that is tailored for creating procedural patterns. These five JavaScript files are located in the following directory:

Windows 32 bit:

Program Files (x86)\Adobe\Adobe Photoshop CS6\Presets\Deco

Windows 64 bit:

Program Files\Adobe\Adobe Photoshop CS6 (64 Bit)\Presets\Deco

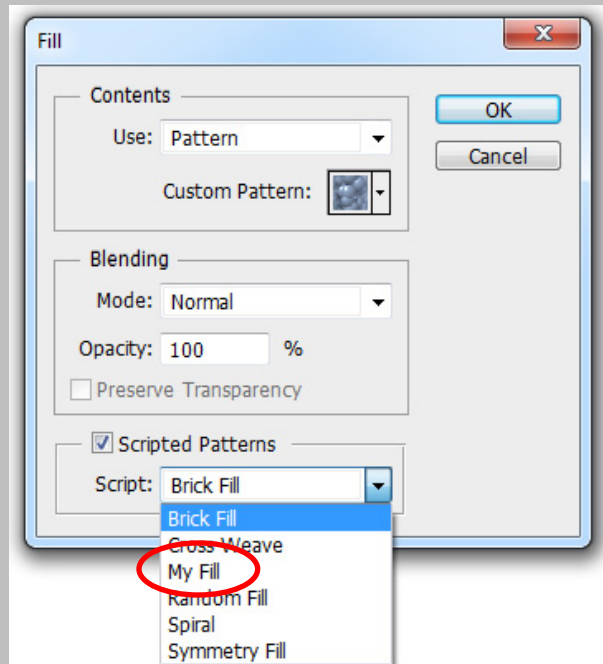
Mac:

/Applications/Adobe Photoshop CS6/Presets/Deco

In this document you will find out more about the Deco framework, you will learn how to modify existing tools and create new ones. Throughout the document you will see simple tasks that will help you to get familiar with the framework and its integration in Photoshop CS6. Here is the first task:

Task 1: Add a new tool

1. Go to the directory where the scripts are located.
2. Copy the *Brick Fill.jsx* to the same directory and rename to *My Fill.jsx*.
3. Open the Fill Dialog
4. Select an area, right click, select *Fill* and *Pattern*. Click on *Scripted Pattern*.
5. A new scripted pattern appears in the pull-down menu – the patterns are ordered alphabetically. Since we copied the *Brick Fill* script, the functionality would be exactly the same.



2 Deco Framework Overview

The core of the Deco framework is a procedural engine. The procedural engine is connected to Photoshop CS6 via an app-specific interface (see Figure 2). After the user invokes a specific scripted pattern, Photoshop informs the

procedural engine about what script to load and what custom pattern (an image patch) to use in the fill.

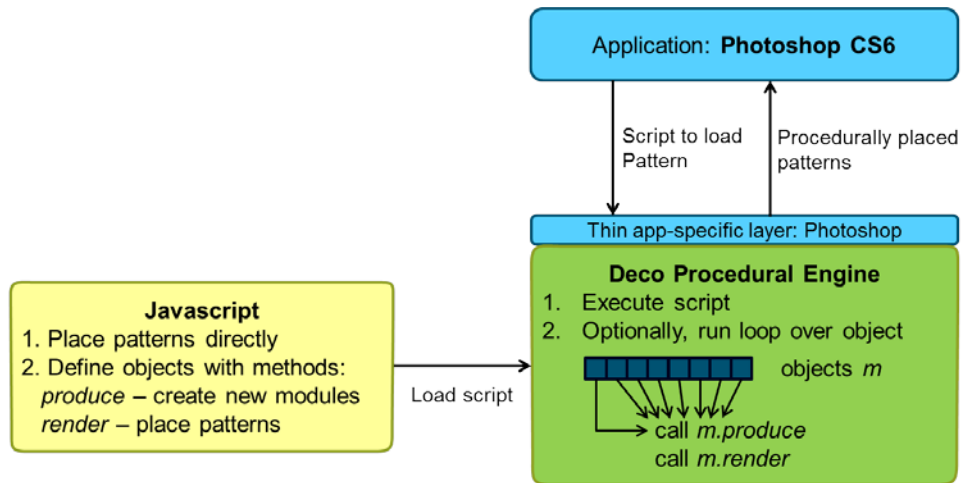


Figure 2: Schematic model of the Deco framework.

There are two ways of specifying procedural patterns in Photoshop CS6 using a script, also called *scriptal*. In many cases, the selected area is filled by the procedural engine loading and executing the given scriptal. In such cases, the scriptal contains a sequence of commands that repeatedly place the input custom pattern, each at a different pixel location, optionally with a specific rotation or scaling. This approach is used in all shipped tools except in *Symmetry Fill*.

In the second approach, as the procedural engine executes the scriptal, several objects (modules) are created and sent to the engine. After the scriptal is executed, the engine loops over these objects and it calls their *produce* and *render* method. The *produce* method is used to modify parameters of the object or create new objects. The *render* method is used to place the input pattern according to the object's parameters. Refer to section 3.3 for more details on this mode. This approach should be used when a simulation loop is needed for creating more complex patterns or when you want to apply symmetry to your pattern, such as in *Symmetry Fill*.

The patterns placed during the initial script execution or during the simulation loop over the defined objects form the resulting fill. Both approaches can be combined. For example, a part of the pattern can be specified during the execution and part by running the simulation loop.

3 Defining Scripted Pattern Fills

Scripted Pattern Fills in Photoshop CS6 are defined using scripts, called scriptals. These scriptals use ExtendScript engine, with some additional predefined objects.

3.1 *Predefined Objects*

The Deco's JavaScript engine exposes a set of predefined objects that are used by the scriptal to communicate with the procedural engine or to perform various tasks that are often needed in a procedural pattern specification:

- *Engine*: facilitates communication with the procedural engine;
- *RenderAPI*: used to place the input pattern at a certain position, with a possible rotation or scaling. This object is also used to obtain an *Image* object that represents the input pattern.
- *Image*: a container for the input custom pattern – received from the *RenderAPI* object;
- *Vector2*, *Vector3*, and *Vector4*: vector objects with overloaded arithmetic operators;
- *Frame2*, and *Frame3*: two and three dimensional frames specifying the position and orientation of a reference coordinate system. Note that Deco is internally a 3D engine, but the pattern fills in Photoshop CS6 use only the first two dimensions for display.
- *Symmetry*: used to create various symmetries.

These objects are described in more details in Appendices.

3.2 *Direct Specification*

Direct specification refers to the mode when the input pattern is placed during the initial execution of a scriptal. This mode may be best explained by following a simple example.

3.2.1 Debugging a Scripted Pattern

Before we begin detailing an example of a scripted pattern, it will be useful for you to know how to obtain debugging information from Photoshop CS6 and the Deco Engine

Please refer to Task **Error! Reference source not found.** to see how a value from the scriptal can be displayed during debugging of your scripts.

Task 2: Print out the input pattern size

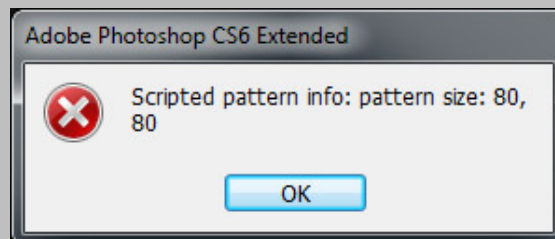
1. Open the file *My Fill.jsx* that you created in Task 1, preferably in the *ExtendScript Toolkit*, but any text editor will suffice.
2. Scroll past the initial comments and past the *function rand*.
3. Check out the four commands defining *outputSize*, *outputOrigin*, *pattern*, and *patternSize*. We will talk about *outputOrigin* just before Task 5.
4. Add the following line just after the line defining *patternSize*.

```
Engine.message("pattern size: ", patternSize.x, " , ", patternSize.y)
```

Alternatively, you can also replace the commas with +

```
Engine.message("pattern size: " + patternSize.x + " , " + patternSize.y)
```

5. Go to pattern fill, and select *My Fill* on the first default input pattern.
Before the selected area is filled, you will receive the following message.



You can see the message sent by the scriptal after the text *Scripted pattern info:*.

Please note that this functionality is intended only for purposes of learning about script writing and for debugging. It is not intended to be used during the normal operation of the scripted patterns.

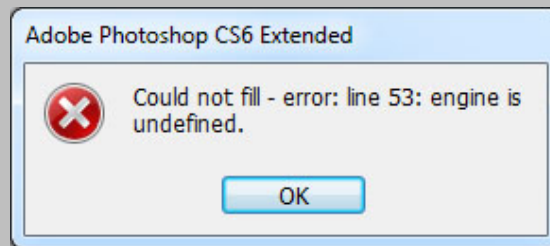
A similar message is printed when there is a parsing error while executing the script, see Task **Error! Reference source not found.**

Task 3: Learn about error messages

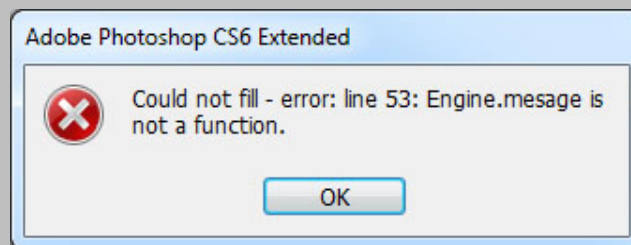
1. Open the file *My Fill.jsx* that you created in Task 1.
2. Let us introduce a typo to the line you just added in Task **Error! Reference source not found.** Change *Engine* to *engine*.

`engine.message("pattern size: ", patternSize.x, " x ", patternSize.y)`

3. If you apply the *My Fill* now, you will see the following error message:



4. Change back *engine* to *Engine* and remove one *s* from *message*:



5. Change back to *message* and remove one comma from inside the message:



Now we are prepared to attempt our first scripted pattern.

3.2.2 Default Grid Fill as Scripted Pattern

Let us try to reproduce the Photoshop's regular pattern fill using a script. The input custom pattern will be placed one after another in a grid layout. We can achieve this pattern by determining the size of the selected area, the size of the input pattern and looping over the *x* and *y* coordinate to fill the given area.

First, we query the size of the selected area from the *RenderAPI* object. We actually get the size of a rectangular area that tightly bounds the current selection – in case it is not a rectangular selection.

```
var outputSize = RenderAPI.getParameter(kpsSize)
```

Note that the parameter *kpsSize* is not a string, it is a predefined variable defining the value. The returned value is a *Vector3* object, where *outputSize.x* is the width of the selected area in pixels and *outputSize.y* is the height.

Then we get the input pattern as an *Image* object and query its size.

```
var pattern = RenderAPI.getParameter(kpsPattern)
var patternSize = pattern.getParameter(kpsSize)
```

The variable *patternSize* is a *Vector3* object, where *patternSize.x* and *patternSize.y* are the width and height of the input pattern in pixels, respectively.

Once we know the sizes, we can write the loop placing the input pattern. A pattern is placed by calling *pattern.render(RenderAPI)* where *pattern* is the *Image* object obtained above. By default the pattern center is at pixel (0,0), which is at the top left corner of the bounding rectangle of the selected area. To move the default location, you can use *RenderAPI.translate* command. As you apply the translation, the transformation matrix stored in the *RenderAPI* object is updated. Thus the consequent translations will be combined. For example, if you translate twice by a distance *d* along the *x* axis it will result in a translation by *2d*. You can also use *RenderAPI.pushMatrix* and *RenderAPI.popMatrix* commands to store and restore the transformation matrix.

The loop placing the patterns in a grid would be as follows:

```
RenderAPI.translate (patternSize.x/2, patternSize.y/2)
for (var y = 0; y < outputSize.y + patternSize.y; y+= patternSize.y)
{
  RenderAPI.pushMatrix()
  for (var x = 0; x < outputSize.x + patternSize.x; x+= patternSize.x)
  {
    pattern.render(RenderAPI)
    RenderAPI.translate(patternSize.x, 0)
  }
  RenderAPI.popMatrix()
  RenderAPI.translate(0, patternSize.y)
}
```

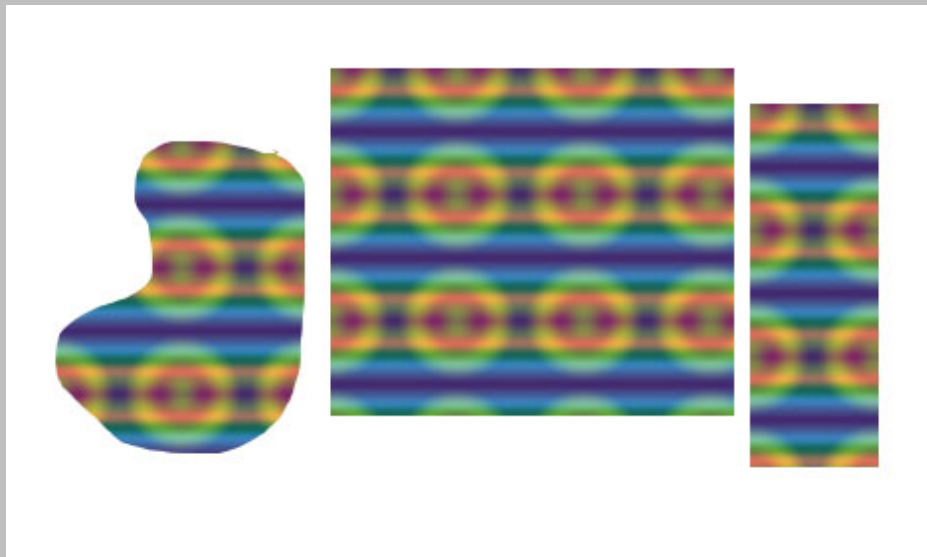
The first translate commands assures that the top left corner of the pattern is aligned with the top left corner of the bounding box of the selected area. Notice the use of *pushMatrix* and *popMatrix* to restore the position after the whole row is place so that we can translate only along the *y* axis. See Task 4 below.

Task 4: Implement default pattern fill (almost)

1. Create a new file *Grid Fill.jsx* in the script directory.
2. Type in the script:

```
var outputSize = RenderAPI.getParameter(kpsSize)
var pattern = RenderAPI.getParameter(kpsPattern)
var patternSize = pattern.getParameter(kpsSize)

RenderAPI.translate (patternSize.x/2, patternSize.y/2)
for (var y = 0; y < outputSize.y + patternSize.y; y+= patternSize.y)
{
  RenderAPI.pushMatrix()
  for (var x = 0; x < outputSize.x + patternSize.x; x+= patternSize.x)
  {
    pattern.render(RenderAPI)
    RenderAPI.translate(patternSize.x, 0)
  }
  RenderAPI.popMatrix()
  RenderAPI.translate(0, patternSize.y)
}
```
3. Go to pattern fill, and select the new *Grid Fill* using the second default input pattern. Depending on your selections you may see something like this:



Notice that the patterns in the neighboring areas are not aligned as they would be if you used Photoshop's default pattern fill. Thus our work is not done yet.

See the text below and Task 5 how to fix that.

When you align the top left pattern with the bounding box of each selected area, the patterns are not aligned in neighboring selections (see the result in Task 4). This is inconsistent with the behavior of Photoshop's default pattern fill. To fix this you can query the position of the top left corner of the selected area in the image coordinates:

```
var origin = RenderAPI.getParameter(kpsOrigin)
```

and use the value to shift the placed patterns:

```
RenderAPI.translate ( -(origin.x % patternSize.x), -(origin.y % patternSize.y))
```

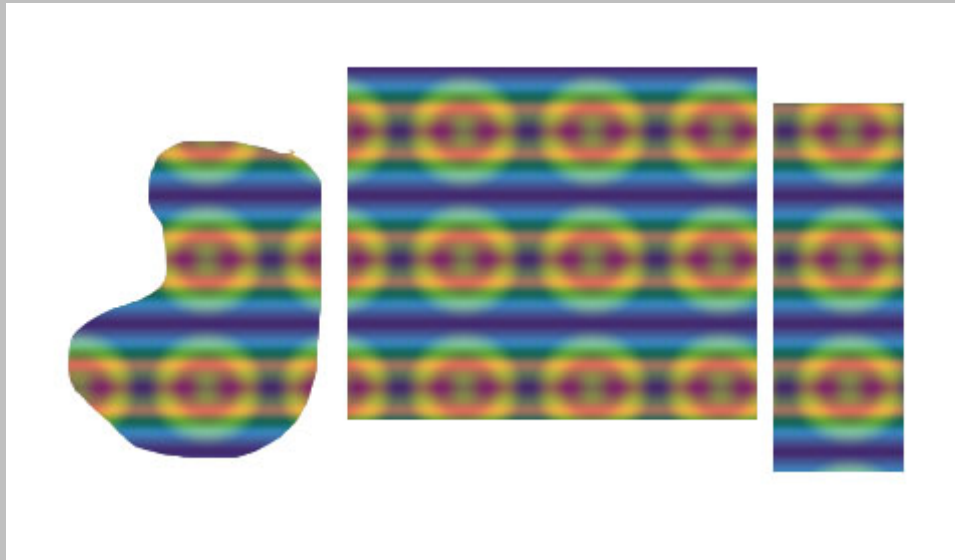
See the result in Task 5.

Task 5: Improve the default pattern fill

1. Open the script *Grid Fill.jsx* you created in Task 4.
2. Add the following two lines just before the first *RenderAPI.translate*:

```
var origin = RenderAPI.getParameter(kpsOrigin)  
RenderAPI.translate ( -(origin.x % patternSize.x), -(origin.y % patternSize.y))
```
3. After you edit the script, you can try to fill your areas again.

This time we get:



Notice the difference between this result and the one from Task 4. Now the patterns are aligned between selected areas.

3.2.3 Adding Rotation to the Pattern

You can use the transformations exposed in the *RenderAPI* object to rotate or scale the placed patterns. You can add rotation to the current transformation matrix used by the *RenderAPI* object by calling

```
RenderAPI.rotate(angleDegrees)
```

where the angle is specified in degrees. See Task 6 for an example.

Task 6: Add rotation

1. Open the script *Grid Fill.jsx*. Save it as *Grid Rotate Fill.jsx*
2. Replace the line *pattern.render(RenderAPI)* with:

```
RenderAPI.pushMatrix()  
RenderAPI.rotate(45)  
pattern.render(RenderAPI)  
RenderAPI.popMatrix()
```
3. After you edit the script, you can apply the new fill



As you can see, the patterns in Task 6 overlap each other based on the order in which they were placed. The Fill started in the first row, left to right, then the second row, etc. This corresponds to the Photoshop's *Normal* blend mode. If the pattern is transparent, the Deco engine will use the transparency even if the layer is not transparent. You can control the blend mode when each individual pattern is placed (see the following section).

Let us make one more adjustment to the fill from Task 6.

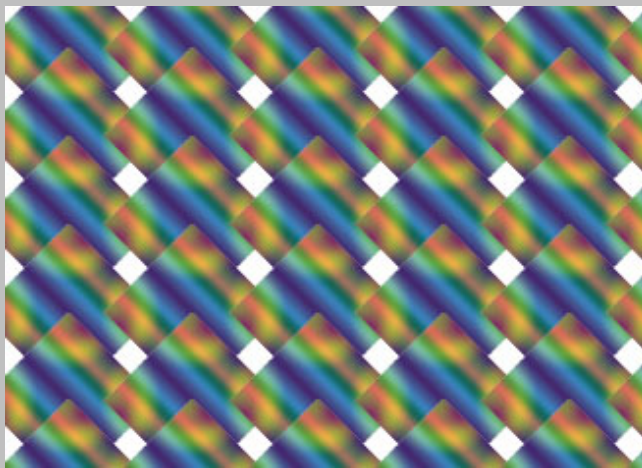
Task 7: Move patterns closer

In this task we will reduce the step between rows to make them overlap more:

1. Open the script *Grid Rotate Fill.jsx* from Task 6.
2. Add `*0.7` to the following three lines:
`RenderAPI.translateRel (patternSize.x/2, patternSize.y/2* 0.7)`
`for (var y = 0; y < outputSize.y + patternSize.y; y+= patternSize.y * 0.7)`
`RenderAPI.translateRel(0, patternSize.y * 0.7)`

The first change moves the first pattern a bit up, the second and third decrease the step in y coordinate.

3. After you edit the script, you can apply the fill:



3.2.4 Changing Pattern Blend Mode

You can change the default normal blend mode before a pattern is placed using the following command:

```
pattern.setParameter(kpsPatternBlendMode, mode)
```

where *mode* is one of:

- kpsBlendNormal
- kpsBlendDarken
- kpsBlendLighten
- kpsBlendHue
- kpsBlendSaturation
- kpsBlendColor
- kpsBlendLuminosity
- kpsBlendMultiply
- kpsBlendScreen
- kpsBlendDissolve
- kpsBlendOverlay

kpsBlendHardLight
kpsBlendSoftLight
kpsBlendDifference
kpsBlendExclusion
kpsBlendColorDodge
kpsBlendColorBurn
kpsBlendLinearDodge
kpsBlendLinearBurn
kpsBlendLinearLight
kpsBlendVividLight
kpsBlendPinLight
kpsBlendHardMix
kpsBlendLighterColor
kpsBlendDarkerColor
kpsBlendSubtraction
kpsBlendDivide

These values and the blend behavior correspond to the Photoshop blend modes. Keep in mind that these blend modes affect only the blending between patterns that are placed into a scratch buffer before the buffer is added to the selected area. Once you change a blend mode on a pattern it will stay set until you change it again.

Task 8: Change pattern blend mode

1. Open the script *Grid Rotate Fill.jsx* from Task 7.
2. Remove the **0.7* from the three lines where you added them in Task 7.
3. Add the following line before the first for loop:
`pattern.setParameter (kpsPatternBlendMode, kpsBlendLighterColor)`
4. After you edit the script, you can apply the fill:



Compare the result with the one from Task 6.

.

3.2.5 Modifying Color – Color Blend Mode

You may notice that the Deco scripted fill patterns shipped with Photoshop CS6 randomly modify the color of the pattern. By default the pattern is placed with its original color. To change the color you can send a color value to the *RenderAPI* object and use various Photoshop blend modes to modify the color of the input pattern. You can use a different fill color and different blend mode for each placed pattern.

First, you set a fill color is using the command:

```
RenderAPI.Color(kFillColor,r,g,b)
```

specifying the value for the red, green, and blue channels (values of *r*, *g*, and *b* range from 0 to 1). Then you can set a blend mode for the placed pattern using

```
pattern.setParameter(kpsColorBlendMode,mode)
```

where *mode* is one of Photoshop blend modes defined in Section 3.2.4.

The default color blend mode is multiply, thus the pattern's red green and blue channel are multiplied by the fill color values of *r*, *g*, and *b*, respectively.

If you want to not only darken the color but also lighten it, you can use *kpsBlendLinearLight* blend mode. Values of *r*, *g*, and *b* below 0.5 will darken the pattern's color while values over 0.5 will lighten it.

3.2.6 Scaling the Patterns

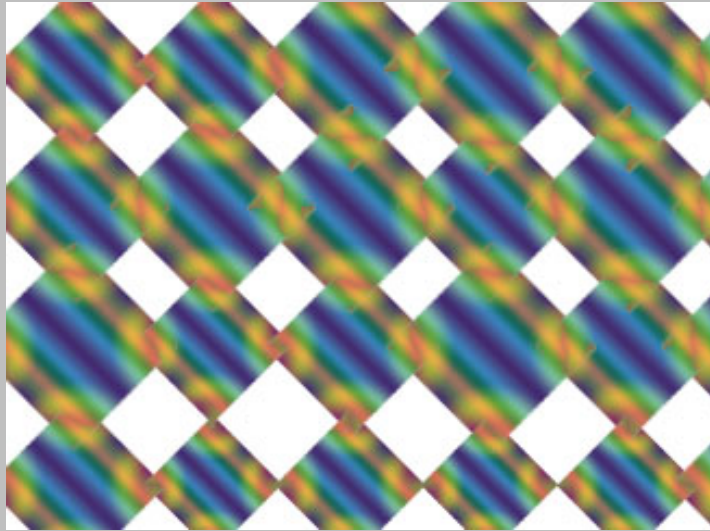
We can not only specify the location or orientation of the placed patterns but we can also scale them using the command:

```
RenderAPI.scale(scaleFactor)
```

Keep in mind that the commands *translate*, *rotate*, and *scale* are additive, thus if you rotate twice by 20 degrees, for example, the resulting rotation will be 40 degrees.

Task 9: Add Scale to Grid Rotate Fill

1. Open the script *Grid Rotate Fill.jsx* from Task 8.
2. Add the following line before the *pattern.render* command:
`RenderAPI.scale(0.7 + Math.random()*0.4)`
3. Apply the fill:



3.2.7 Preserving Randomness across Selections

As you saw in Task 4 and Task 5, some care is needed to ensure that the pattern is consistent for neighboring or overlapping selections. If such behavior is desired you should not use the method *Math.random* as you did in Task 9 because you cannot control its seed. Instead, you should use your own random generator, for which you can set the seed.

If you check out the shipped script *Brick Fill.jsx* you will see that we determine the row and column index of the top left element in the selection using commands:

```
var row = Math.floor( outputOrigin.y / patternSize.y )  
var column = Math.floor( outputOrigin.x / patternSize.x )
```

We update these values in the loop placing the patterns and then we seed the random number generator for each position using:

```
rand.seed = row * 1234567 + column * 7654321
```

This assures that the same random values will be used for the pattern even if it is part of a different selection.

The example in Task 10 places randomly rotated and scaled patterns in a grid, where each position is slightly modified (jittered) by +/- quarter of the pattern width and height. The rotations are selected so that there are only 30 distinct values between 0 and 360 degrees. The reason for this is related to performance, because rotated patterns are stored in a cache to speed up subsequent rotations. See Section 4 for more information.

Task 10: Add Random Rotation and Scale to Brick Fill

1. Open the script *Brick Fill.jsx*.
2. Run the script with the following pattern (you will need to copy and paste it from this document to Photoshop and make it a pattern):



3. Run the fill and you will receive the result from Figure 3, on the left.
4. Add the following two lines just before *pattern.render*:

```
RenderAPI.scale(rand()*0.1 + 1)  
RenderAPI.rotate(-4 + Math.floor(rand()*60) / 7.5) // 60 distinct rotations
```
5. Run the fill again and you will get the result from Figure 3, on the right.

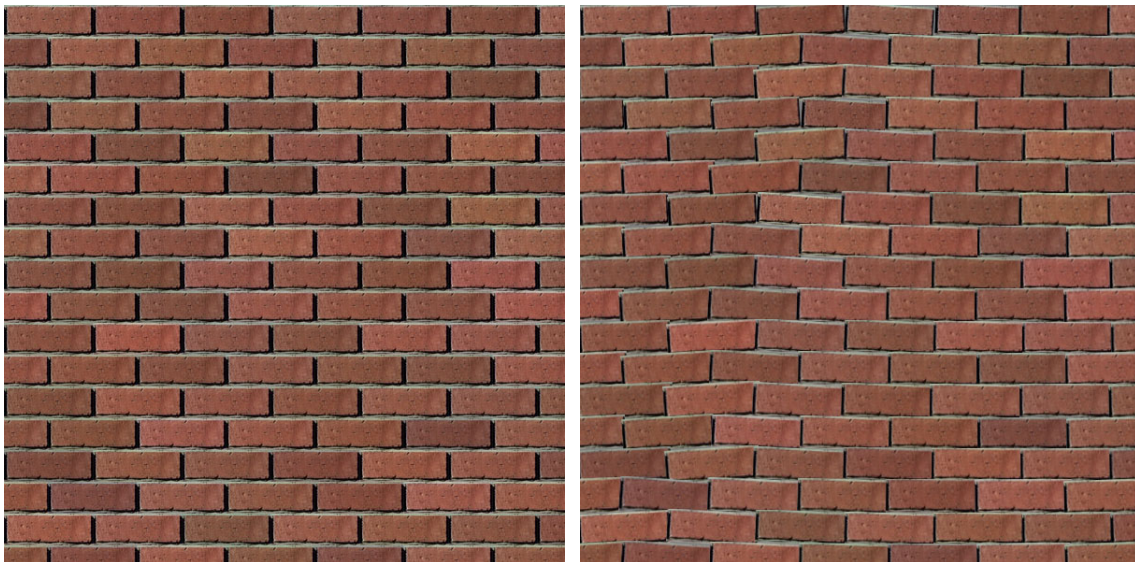


Figure 3: An effect of random scaling and rotation (on the left) applied to the default *Brick Fill* (on the right).

The following section will explain the simulation mode of the Deco framework.

3.3 Simulation Loop

There are cases when direct specification of pattern fills during the execution of the script is not sufficient. An example could be the use of symmetries or when creating more complex procedural pattern.

Let us review how a procedural model operates. A procedural model is specified by a set of modules and a set of rules specifying the behavior of these modules over time. The modeling process starts with an initial module or modules. Then in a simulation loop, the rule for each active module is applied, controlling the development of the model. At any stage, or after the simulation loop is completed, the model can be converted into a graphical representation. Some or all modules have certain graphical meaning – they represent parts of the modeled structure.

Procedural modeling takes advantage of the fact that from a set of simple rules applied repetitively to various parts of the model – captured as modules – a complex pattern may emerge – as in the case of examples from Section 5.

A key task in developing a procedural model is to determine the rules that control the local behavior of the model and the modules, to which these rules are applied. In Deco framework, the modules are expressed as JavaScript objects and the rules are captured in the object methods:

```
produce (engine)
render (renderAPI)
```

The first method is called by the procedural engine during each simulation step and the second method is called when the structure is being displayed. Note that Photoshop keeps a buffer for the current fill pattern thus the *render* method is in fact adding patterns to this buffer, which is then merged with the current layer.

The *produce* method can create new modules or it can just modify the behavior of the existing module. The parameter *engine* is an object that represents the procedural engine. The parameter *renderAPI* stores the predefined *RenderAPI* object that contains methods for placing the input pattern.

Example 1: This example will assume that the input pattern is a square pattern. It will divide the selected area into bins of size equal to the pattern size, and create an array that stores a flag for each bin marking whether the bin is occupied or not.

The following is a definition of an object *ModuleSeek* that will place patterns in a straight line until it reaches an occupied bin. Then it turns right and tries to continue. If even the bin to the right is occupied, it removes itself from the engine.

```

function ModuleSeek(frame)
{
    this.frame = frame
    markOccupied (frame)
}

ModuleSeek.prototype.produce = function (engine)
{
    // test if we can move forward
    this.frame.advance(patternSize.x)
    if (positionOccupied (this.frame))
    {
        // try to turn right
        this.frame.advance(-patternSize.x) // move back first
        this.frame.rotateDeg(-90)
        this.frame.advance(patternSize.x)
        if (positionOccupied(this.frame))
        {
            Engine.removeModule(this)
            return kDontCallAgain
        }
    }
    markOccupied (this.frame)
    return kCallAgain
}

ModuleSeek.prototype.render = function (renderapi)
{
    pattern.render (renderapi)
}

```

This code defines the module and its methods. It needs to create the first module (the initial state). That can be done by the following code:

```

// Initial module
var frame = new Frame2()
frame.rotateDeg(90)
frame.setPosition (patternSize.x/2, patternSize.y/2)
Engine.addModule (new ModuleSeek (frame))

```

The array used to mark which place is occupied is defined as follows:

```

// Get the size of the selected area and of the input pattern
var outputSize = RenderAPI.getParameter(kpsSize)
var pattern = RenderAPI.getParameter(kpsPattern)
var patternSize = pattern.getParameter(kpsSize)

// get the size of the selected area in multiples of pattern size
var sizex = Math.floor((outputSize.x + patternSize.x-1) / patternSize.x)
var sizey = Math.floor((outputSize.y + patternSize.y-1) / patternSize.y)

// define the array and initialize to false
var occupied = new Array(sizex*sizey)

```

```

for (var i = 0; i < sizex*sizey; i++)
    occupied[i] = false;

function positionOccupied (frame)
{
    var x = frame.position().x
    var y = frame.position().y
    // first test whether we are inside the selected area
    if (x < 0 || x >= sizex * patternSize.x || y < 0 || y >= sizey * patternSize.y)
        return true

    return occupied[Math.floor(x / patternSize.x+1) + sizex * Math.floor(y /
patternSize.y+1) ]
}

function markOccupied (frame)
{
    var x = Math.floor((frame.position().x ) / patternSize.x + 1)
    var y = Math.floor((frame.position().y ) / patternSize.y + 1)
    occupied[x + sizex * y ] = true
}

```

Before finishing the script, you need to set the output bounding box and make sure that the simulation will run for a certain number of steps:

```

Engine.setSceneBBox (0, outputSize.x, 0, outputSize.y)
Engine.setParameter (kRunSimulation, 1)
Engine.setParameter (kNumSimulationSteps, 1000)

```

The number of steps can be a very large number because the simulation stops automatically when a no new module is added to the engine and no pattern is placed in a simulation step.

These three pieces of code define the whole script, which can produce the result in Figure 4 (on the left). Note that it looks best when the input pattern has a clear single direction. The fill on the right has been obtained by defining a second input module:

```

// second initial module
var frame2 = new Frame2()
frame2.setPosition (patternSize.x*(Math.floor(sizex/2) - 0.5), patternSize.y*1.5)
Engine.addModule (new ModuleSeek (frame2, 1 /* delay */)

```

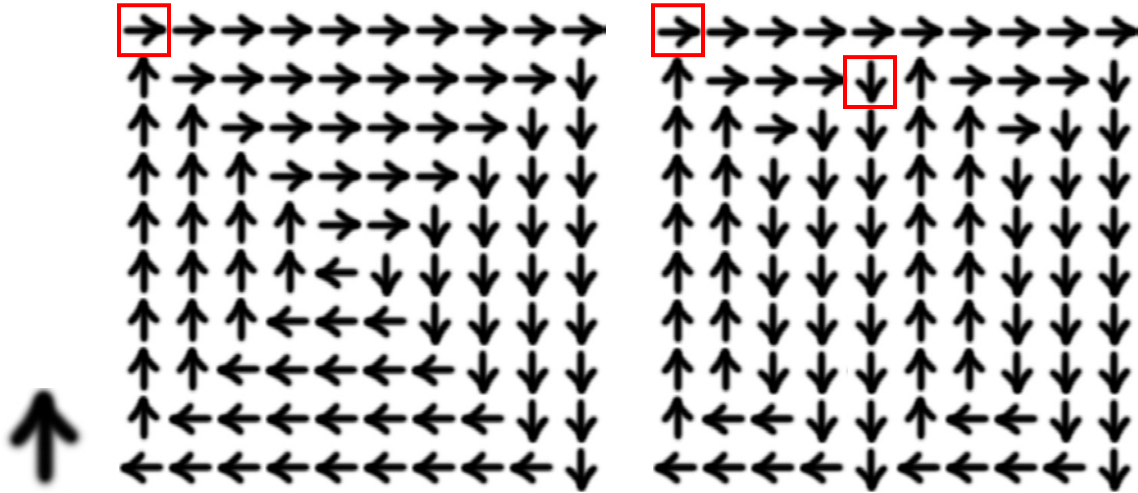


Figure 4: A more complicated pattern defined by custom scripts (not shipped with CS6). A simulation loop is executed on modules that place the input pattern (the arrow) in a row until an occupied element is reached and then they turn right. The pattern on the left started with one module and the pattern on the right started with two initial modules (marked in red).

This is just a very basic example of using the simulation loop for creating procedural pattern fills. It is possible to extend the basic functionality of this scriptal. You could add a random change of direction, even if there is no occupied element in the path of the module. You could decide whether to turn right or left if the place ahead is occupied or you could spawn new modules on the side of the straight rows of elements at certain distances – forming branches. In those cases the result may not be a completely filled area. In that case an additional step may be performed - if no module can move forward, the array of occupied elements is scanned and if there is an empty element, a new module is placed there.

The following section reviews the operation of the Deco pattern fills.

3.4 Operation of Deco Pattern Fills

The Deco procedural engine is implemented as a C++ class that is exposed in the scriptal as the *Engine* object. When a Deco pattern fill is applied to a selected area, the scriptal that defines the model is loaded and executed (see the long brown arrow along the scriptal in Figure 5). A pattern fill can be defined in that stage (Section 3.2). Optionally, a set of objects can be defined to create a pattern during the simulation loop (Section 3.3).

When the method *Engine.addModule* is invoked in the scriptal, the C++ method *Engine::addModule* is called (see the control going to the C++ class and back). The C++ implementation of the method stores the module in a list kept by the *Engine* class. After executing the scriptal, a JavaScript runtime is populated with the modules defined by the scriptal.

The engine then runs the simulation by repeatedly performing a *Produce* and *Render* pass over the modules stored in the list. During each pass, the corresponding method *produce* or *render* defined on each module is executed. Figure 5 illustrates the flow of control in the Deco framework.

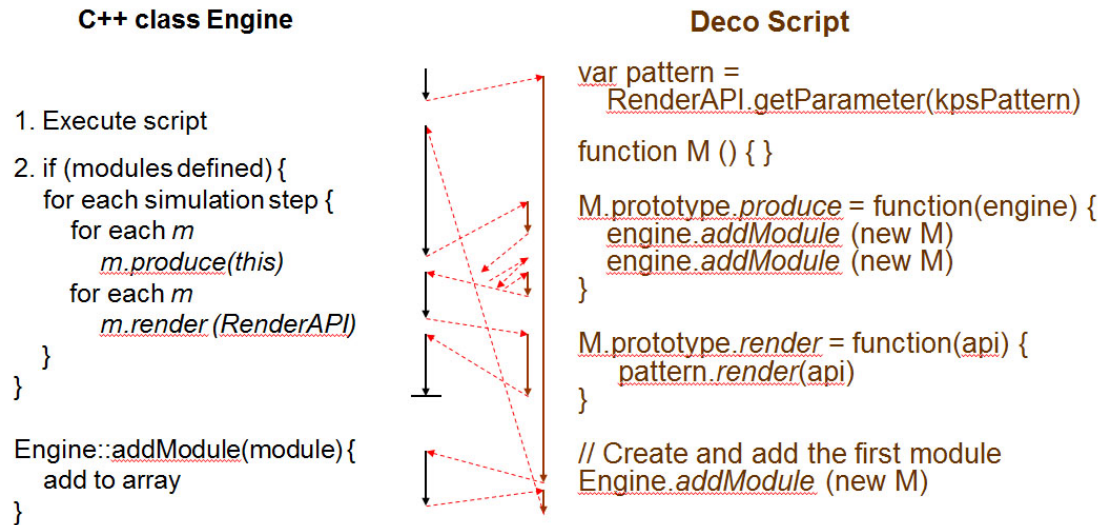


Figure 5: Example of the flow of control in the Deco framework during a simulation.

The functionality of the simulation loop is very simple and the question one may ask is why it is not implemented in the script directly? There are several advantages of having the procedural engine control the simulation loop.

1. The loop execution can be started and stopped depending on changes in the model. If no module is created in a produce pass and no pattern is placed in a render pass the simulation is automatically stopped.
2. The simulation loop can be stopped by the application if the execution of the model is too long.
3. Also, there are various additional mechanisms affecting the simulation loop that would be difficult for the user to implement in the script.

One of the additional mechanisms is an execution of *Start* method. Before the simulation starts (but after the scriptal is loaded), a method *Start* is called, if present. Note that the simulation is not started by default and the user needs to set the *Engine's* parameter *kRunSimulation* to 1 (see Appendix A).

The following sections describe the individual passes in more detail.

3.4.1 Produce Pass

The *Produce* pass is very straightforward. At the beginning of each pass, the optional method *StartEach*, which may be defined in the scriptal, is called. Then the list of

modules stored in the procedural engine is parsed and the *produce* method of each object is called, if it is defined. New objects created by the pass are added to the end of the list, but their *produce* methods are not executed until the next *Produce* pass.

A method *EndEach*, if defined, is executed at the end of each *Produce* pass.

3.4.2 Rendering Pass

Rendering of the procedural model is done by calling the engine's *render* method, which in turn calls each module's *render*. Similarly to the *Produce* pass, you can define a method *StartEachRender* and *EndEachRender* that will be called at the beginning and at the end of each *Render* pass, respectively.

During the *Render* pass the module's *render* method receives a *RenderAPI* object as a parameter. The *RenderAPI* object is used to place the input pattern to form the desired fill.

3.4.3 Module's Position and Orientation - Frame

A module can contain an optional parameter *frame*, specifying the position and orientation of the module. If the parameter *frame* is present it is applied automatically before the module's *render* call is executed. Thus the primitives can be specified in the local coordinate frame, such as the fill in the Example 1.

4 Performance

The performance of the scripted pattern fills depends on the number of patterns placed and on the type of manipulations done to the pattern. If you do not rotate or scale the input pattern and the pattern is not too small compared to the size of the selected area, the performance will be very good. If you scale each placed pattern, there will be an impact to performance. Rotating the input pattern can be quite costly and that is why Deco internally uses a **cache to store rotated patterns**. If you write your script so that it uses only a limited number of rotations, up to 50 or so, they will be cached. For example, see the way rotations are defined in Task 10.

The size of the cache is by default set to 128 MB, which is sufficient to run the Spiral fill on the default patterns without any rotated pattern being dropped from the cache. If you use a bigger input pattern, you would reach the cache limit before all angles are stored and the performance would drop. For that reasons, you can increase the size of the cache by calling the following command:

```
pattern.setParameter(kpsMaxPatternCacheSize, sizeInBytes)
```

The cache is kept around so that when you place the same input pattern again in the same script – but in a different fill area - or in a different script that uses same rotations, the cached patterns can be reused. You can disable this behavior by setting the following parameter to 1:

```
pattern.setParameter(kpsKeepPatternCache, 1)
```

Note that this will affect the global behavior of the cache, for all scripts and all patterns. If you want to clear cache just in your script you can call the command:

```
pattern.clearCache()
```

If your input pattern is too small, even without rotations, the fill can take a long time. It is important to realize that patterns of constant color get automatically converted to a 1x1 pixel patterns in Photoshop, because original pattern fill did not need to know about the pattern size. If you use such a pattern in any Deco script the fill could be very slow.

5 Motivation for Using Procedural Modeling

The motivation for creating Deco framework came from my previous experience with procedural modeling. In a procedural model, a local behavior or growth of a structure or a pattern is described by simple rules or procedures. These rules are applied in parallel to many parts of the model, resulting in a potentially complex behavior or structure.

Example of a simple branching structure defined by two rules is given in Figure 6.

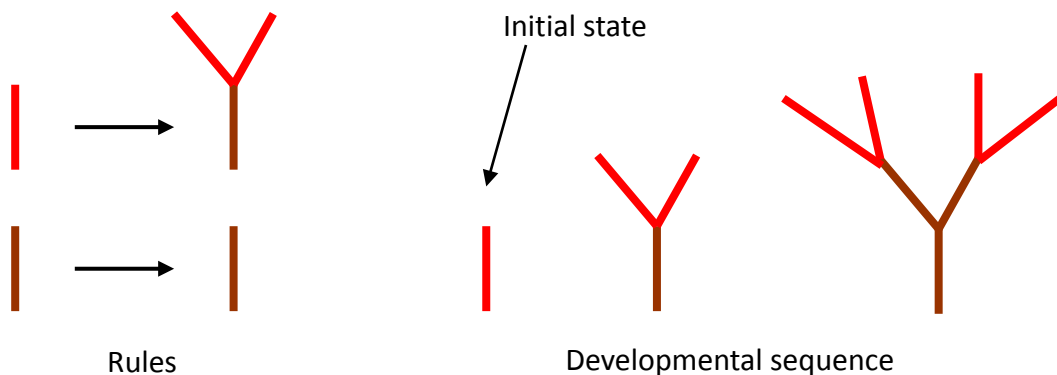


Figure 6: Example of a simple procedural model with two rules

Rules similar to those in Figure 6, with additional clipping when the branch reached out of a predefined shape have been used to generate branching structures in Figure 7. The leaves were added using additional rules.

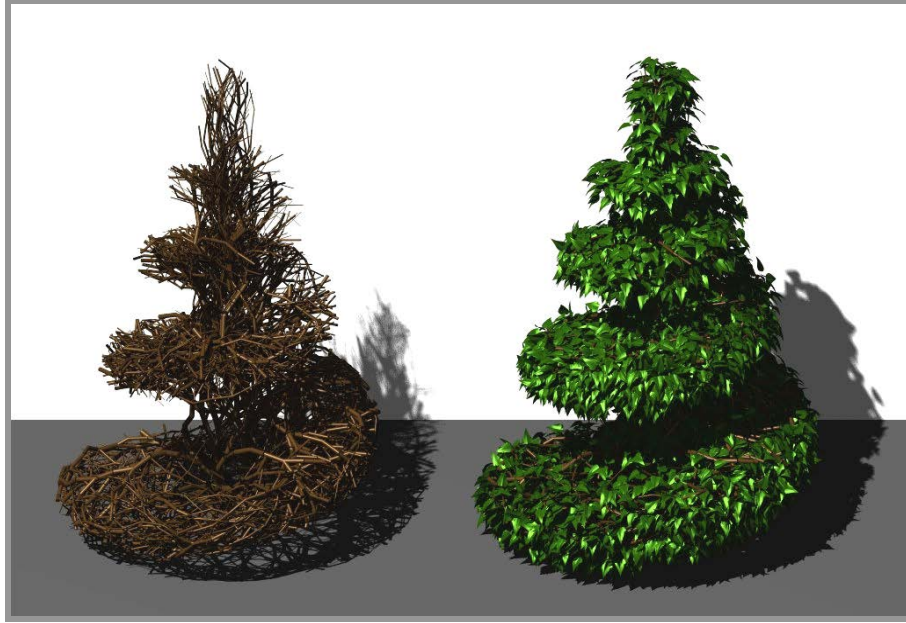


Figure 7: Procedurally generated branching structure clipped to a spiral shape (*from Siggraph '94 paper by Prusinkiewicz et al.*).

Procedural models can generate other structures than trees and bushes. Figure 8 shows several procedural ornaments, illustrating the potential of using procedural models in design applications.



Figure 8: Procedurally generated ornamental patterns (*from Siggraph'98 paper by Wong et al.*).

In the context of Photoshop CS6, the procedures are placing image patterns, similarly to the original pattern fill. The procedures can be as simple as a nested loop creating a brick fill with randomly varied color of the placed input pattern (see Figure 1a) or a more complicated, creating the pattern in Figure 9.

The pattern in Figure 9 has been created in two layers. The first layer has been filled using the Spiral pattern. For the second layer we modified the Spiral script and added extra logic for creating a sequence of weaved patterns perpendicular that avoiding collision with each other.

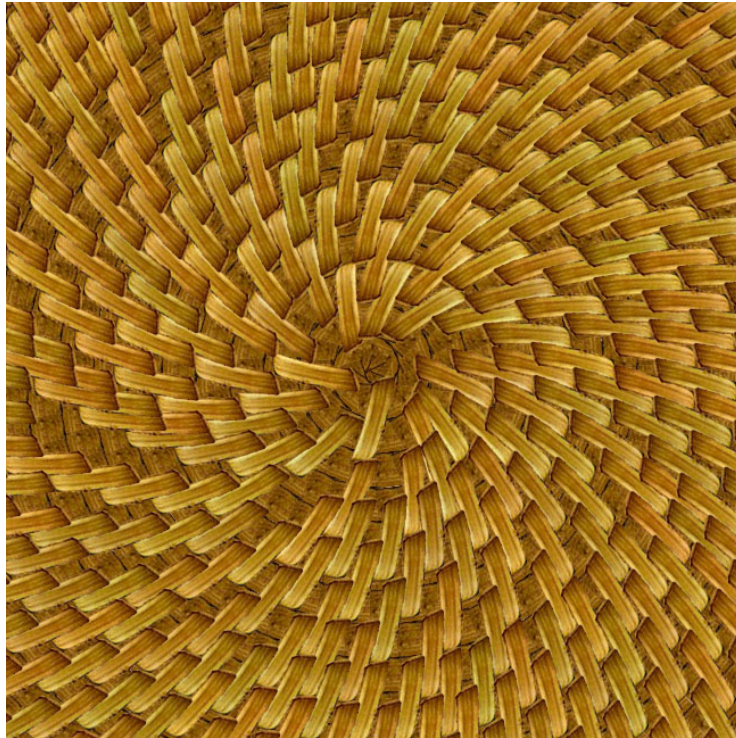


Figure 9: A more complicated pattern defined by a custom script (not shipped with CS6).

6 Conclusions

Deco is a powerful framework for creating procedural pattern fills. The framework was designed so that the fills are easy to specify in a well-known scripting language with various functionalities provided as predefined objects

The framework as it is implemented in Photoshop CS6 is targeting users at several levels:

1. A small group of users who are familiar with scripting can write their own scripts.

2. Another group of users may just modify the existing scriptals and change the parameters of the defined fills.
3. The rest of the users just choose from a set of predefined scriptals provided with the application, exposed as Deco scripted pattern fills.

We hope that some users may also upload scripts created by others to enhance their application.

7 Additional Resources

Feel free to contact me at rmech@adobe.com with any questions.

Appendices

The appendices list the methods associated with predefined objects and give more details about the object's function.

A Object Engine

Procedural engine can be referred to as an object *Engine* in the script. The class exposes the following methods:

```
addModule (object)
removeModule (object)
setInitialObject (object)

setSceneBBox (minx, maxx, miny, maxy)
setParameter (parameterType, value)
getParameter (parameterType)
setModuleParameter (module, parameterType, value)
getModuleParameter (module, parameterType)

evalFile (scriptName [, init func parameters])

stopPass ()
```

The method **addModule** adds an object to the procedural engine. The method **removeModule** removes the module from the engine.

The method **setInitialObject** adds a special module whose produce method is called to create the initial modules. After the first step the module is ignored.

The method **setSceneBBox** is used to define the scene span in x and y coordinates.

The method **setParameter** controls various parameters of the procedural engine. The first parameter of the method is one of the following predefined numbers:

- *kApplyFrame*: if set to 1 (default), the module's frame, if defined, is automatically applied before its *render* method is called.
- *kRunSimulation*: if this parameter is set to 1, the procedural engine informs the application that the simulation should be automatically started. It is **not** set by default in Photoshop CS6.
- *kNumSimulationSteps*: this parameter controls the number of steps in the procedural engine loop. It is 1 by default since all patterns shipped in Photoshop CS6 are defined while the script is executed or require only one step (Symmetry pattern).

The method **getParameter** can query any of the above parameters.

The method **setModuleParameter** controls various parameters of a given module. It is important to first add the module to the procedural engine using the *addModule* function,

before you can call *setModuleParameter*. The second parameter of the method is one of the following predefined numbers or strings:

- *kModuleProcessed*: if this parameter is set to 1, the procedural engine stops further processing of the module. This can be used in case of symmetries when the *render* method can be called multiple times with different symmetry matrices.
- *kModuleApplyFrame*: this parameter overrides the global parameter set in the procedural engine for the given module.
- “*call*”. This string parameter can be followed by one of “produce” or “render” and a value of 1 or 0. If the value is set to 1, the corresponding method of the module will be called in the subsequent simulation step. If it is set to 0, it will not be called unless it is set to 1 again.

The method **evalFile** evaluates the given script. The script can contain an initialization method, whose name is a concatenation of the script name and the word “Initialize”. This method is called after the script is evaluated and any additional parameters of the method *evalFile* are sent as parameters of the initialize method.

The method **stopPass** terminates the current *produce* or *render* pass.

The *Engine* class defines these additional constants that are accessible in the script:

- values returned by methods *produce* and *render*
 - kDontCallAgain*
 - kCallAgain*

B Object Frame

The *Frame* object contains a 4x4 matrix specifying a position and an orientation. There are two frame objects *Frame2* and *Frame3*. They are both represented by the same data structures, but the 2D version (*Frame2*) ignores the third axis and thus some operations are faster. Note that although you can use the third dimension in your *Frame3* objects, the *z* axis is ignored when placing the pattern.

The default frame points up along the positive *y* axis, thus a pattern placed using the default frame will point down, since the point (0,0) is in the top left corner of the bounding box of the selected area.

The frame object has the following methods:

```
translate (x,y,z), translate (vector)
advance (distance)
setPosition (x,y,z), setPosition (vector)
setHeading (x,y,z), setHeading (vector)
setUp (x,y,z), setUp (vector)
setRight (x,y,z), setRight (vector)
setSize (x,y,z), setSize (vector)
position (), position (index)
```

```

heading (), heading (index)
up (), up (index)
right (), right (index)
size (), size (index)

rotateDeg (angle), RotateDeg (angle, point),
rotateDeg (angle, point, vector)
rotatePitch (angle), rotateYaw (angle), rotateRoll (angle)
rotateTowards (point, maxangle)
addToHeading (x,y), addToHeading (vector)

applyToPoint (point)
applyToVector (vector)
toLocalCoords (point)

```

The method **translate** translates the frame's position by the given distance. Keep in mind that the vectors specifying local coordinate system of the frame are scaled by the given scale of the frame. Thus if you set the frame size to (2, 2, 2) and then translate it by (1, 1, 1) the position will be increased by (2, 2, 2). The parameters can be either a two or three numbers or a vector object *Vector2*, *Vector3*, or *Vector4* (see the section below).

The method **advance** is equivalent to *translate (0,distance,0)*.

Methods **setPosition**, **setHeading**, **setRight**, **setUp**, and **setSize** are used to set the frame position, heading vector (*y* axis), right vector (*x* axis), up vector (*z* axis) and the size. The parameters can be either numbers or a vector object *Vector2*, *Vector3*, or *Vector4*. In case of size, we can specify only one value, which is then used for all three axes. Note that you have to make sure that heading, up, and left, vectors are orthonormal.

Methods **position**, **heading**, **right**, **up**, and **size** return either a *Vector3* if no parameter is given or a specific coordinate if parameter *index* is set.

The method **rotateDeg** rotates the frame around its position by the given angle. The axis of rotation is (0,0,1). Positive angles rotate the frame clockwise, negative angles counter clockwise. Optionally, you can specify the point of rotation and the vector around which the frame is rotated. Methods **rotatePitch**, **rotateYaw**, and **rotateRoll**, rotate the frame around its position by the given angle. The vector of rotation is (1,0,0), (0,0,1), and (0,1,0), respectively.

The method **rotateTowards** rotates the heading towards the given point, but not more than the given maximum angle. This operation works only when the frame position and the given point are in the plane $z=0$.

The method **addToHeading** adds a vector to the heading vector. This method adjusts only the *x* and *y* axis, the *z* axis of the frame has to be (0,0,1).

Methods **applyToPoint** and **applyToVector** multiply the given vector or point by the frame and return the transformed vector or point, respectively.

The method **toLocalCoords** converts a given point to the coordinates within the frame.

C Object Vector

There are three classes, *Vector2*, *Vector3*, and *Vector4* exposed in the script, defining two to four-dimensional vectors. The elements of a vector can be accessed using *.x*, *.y*, *.z*, and *.w*.

There are following methods:

```
length ()
lengthSquared ()
dot (vector)
normalize ()
cross (vector)
```

They are self-explanatory.

In addition you can perform the following operations on vectors:

```
vector1 + vector2
vector1 - vector2
vector * scalar
vector / scalar
vector1 == vector2
```

D Object RenderAPI

The *RenderAPI* object is used to place the patterns into the application's current layer.

The object has the following methods:

```
setFrame (frame)
getFrame ()
scale (x)
rotateDeg (deg)
translate(x,y)
translateRel(x,y)
pushMatrix ()
popMatrix ()

setSceneBBox (minx, maxx, miny, maxy)
Color (kFillColor, red, green, blue)

setParameter (type, value(s))
getParameter (type)
```

The method **setFrame** sets the current frame. In fact it multiplies the existing frame with the new one thus you need to use *pushMatrix* and *popMatrix* calls if you do not want this frame to persist. The method **getFrame** gets the current frame. This may be useful when a symmetry is applied to the module.

The method **scale** sets the scale of the subsequently placed pattern. The scale factor is uniform, same in *x* and *y*. The method **rotateDeg** rotates the subsequently placed pattern by the given angle in degrees. Note that a pattern without any rotation is pointing up.

The method **translate** moves the current position or orientation by the given *x* and *y* pixels horizontally and vertically, respectively. Note that initial position is at 0,0, which is the top left corner of the bounding box of the selected area.

The method **translateRel** is a special version of the method **translate**. It translates the current position by *x* and *y* within the current frame, respecting the actual rotation and scale. Thus if you first apply a rotation by 45 degrees, then scale by a factor of 2, *RenderAPI.translate(4,0)* will move the current position diagonally by a distance of 8 pixels.

The method **Color** can be used to multiply the red, green, and blue component of each subsequently placed pattern by the given values (a value of 1 results in no change). The first parameter *kFillColor* has to be specified since the Deco engine internally supports also vector art that uses both stroke and fill color (vector art is not exposed in Photoshop CS6). See Section 3.2.5 for more detail.

The method **setParameter** and **getParameter** are used to set and get specific parameters, respectively. The *RenderAPI* object defined by Photoshop uses the following parameters:

The method *RenderAPI.getParameter* supports these parameters:

- *kRenderAPIName* – returns a string “PS”.
- *kpsPattern* – returns the pattern selected in Photoshop CS6.
- *kpsSize* – returns a *Vector3* object specifying the size of the bounding box around the selected area in pixels.
- *kpsOrigin* – returns a *Vector3* object containing the location of the top left corner of the bounding rectangle around the selected area.
- *kpsAnyPatternPlaced* – returns 1 if there was a pattern placed. Usually, you would set the value to 0 using *RenderAPI.setParameter(kspAnyParameterPlaced,0)* in the function *StartEachRender* and you can get the value in the function *EndEachRender* (see Section 3.4.2).

E Object Image

The object *Image* is used to represent the pattern that is to be placed by the script. The pattern is obtained from the *RenderAPI* object using the following call:

```
pattern = RenderAPI.getParameter(kpsPattern)
```

The object *Image* has the following methods:

```
render (RenderAPI)  
clearCache ()
```

```
getParameter (type)
setParameter (type, value)
```

The method **render** sends the pattern to the given renderer. The position, rotation, and color of the patterns are affected by the parameters set in the *RenderAPI* object..

The method **clearCache** clears the cache of rotated patterns. The whole cache is cleared, not only cache related to the current pattern. See Section 4 for more detail on the use of pattern cache.

The method **getParameter** queries these parameters:

- *kpsSize* – returns a *Vector3*, whose *x* and *y* coordinates contain the pattern size in pixels.
- *kpsMaxPatternCacheSize* – returns the maximum allowed size of the pattern cache in bytes. See Section 4 for more details on the use of the pattern cache.
- *kpsKeepPatternCache* – returns value 0/1 depending on whether the pattern cache is being kept after each pattern fill is completed. See Section 4 for more details on the use of the pattern cache.

The method **setParameter** can be used to set the following values:

- *kpsColorBlendMode* – the value specifies the blend modes used to blend each placed pattern with a color specified in the *RenderAPI* object. The default blend mode is *kpsBlendMultiply* (the list of all blend modes is given in Section 3.2.4).
- *kpsPatternBlendMode* – the value specifies the blend modes used to blend each placed pattern with the previously placed patterns. The default blend mode is *kpsBlendNormal* (the list of all blend modes is given in Section 3.2.4).
- *kpsMaxPatternCacheSize* – the value is the maximum allowed size of the pattern cache in bytes. See Section 4 for more details on the use of the pattern cache.
- *kpsKeepPatternCache* – the value 0 or 1 specifies whether the pattern cache is being kept after each pattern fill is completed. See Section 4 for more details on the use of the pattern cache.

F Object Symmetry

Symmetries are supported by inserting an instance of a built-in object *Symmetry* among modules specifying the structure (using *Engine.addModule*). The module stores a list of matrices to be applied for the symmetry. The matrices are created by the module according to the type of symmetry. The type is set using the method:

```
mySymmetry.setSymmetry (type, parameters)
```

Currently, the type parameter can be one of the following

```
kSymmetryLineReflection
kSymmetryPointReflection
```



```
kSymmetryRotation
kSymmetryTranslation
kSymmetryFriezeTranslation
kSymmetryFriezeGlideReflection
kSymmetryFriezeTranslationLineReflection
kSymmetryFriezeTranslationMirrorReflection
kSymmetryFriezeTranslationPointReflection
kSymmetryFriezeGlideReflectionRotation
kSymmetryFriezeTranslationDoubleReflection
kSymmetryWallpaperP1
kSymmetryWallpaperP2
kSymmetryWallpaperPM
kSymmetryWallpaperPG
kSymmetryWallpaperCM
kSymmetryWallpaperP4
kSymmetryWallpaperP4M
kSymmetryWallpaperP4G
kSymmetryWallpaperPMM
kSymmetryWallpaperPMG
kSymmetryWallpaperPGG
kSymmetryWallpaperCMM
kSymmetryWallpaperP3
kSymmetryWallpaperP3M1
kSymmetryWallpaperP31M
kSymmetryWallpaperP6
kSymmetryWallpaperP6M
kSymmetryTranslationLineReflection
kSymmetryGlideReflection
kSymmetryDilatation
kSymmetryDilativeRotation
kSymmetryInfiniteDilativeRotation
kSymmetryDilativeReflection
kSymmetryRosette
kSymmetryTiling
```

A line reflection is defined by a frame (the line is the y axis) or by a point and an angle from y axis.

A point reflection is defined by a frame or a point.

A rotation is defined by a frame or a point and an angle (defining the space of the first instance), followed by a number of instances around the center (frame's position or the given point).

A translation is defined by a frame or a point and an angle and the number of instances along the given direction.

See the scriptal *Symmetry Fill.jsx* to see how to define each of these symmetries.

After you create a new *Symmetry* object, set its type, and add it to the *Engine* object, you have to add to it those modules that the symmetry will affect using the method **addModule** of the symmetry object.