Programming Scripted Patterns in Photoshop CC, Version 2 – Programming Guide

Radomír Měch

Principal Scientist
Imagination Lab, Adobe Research

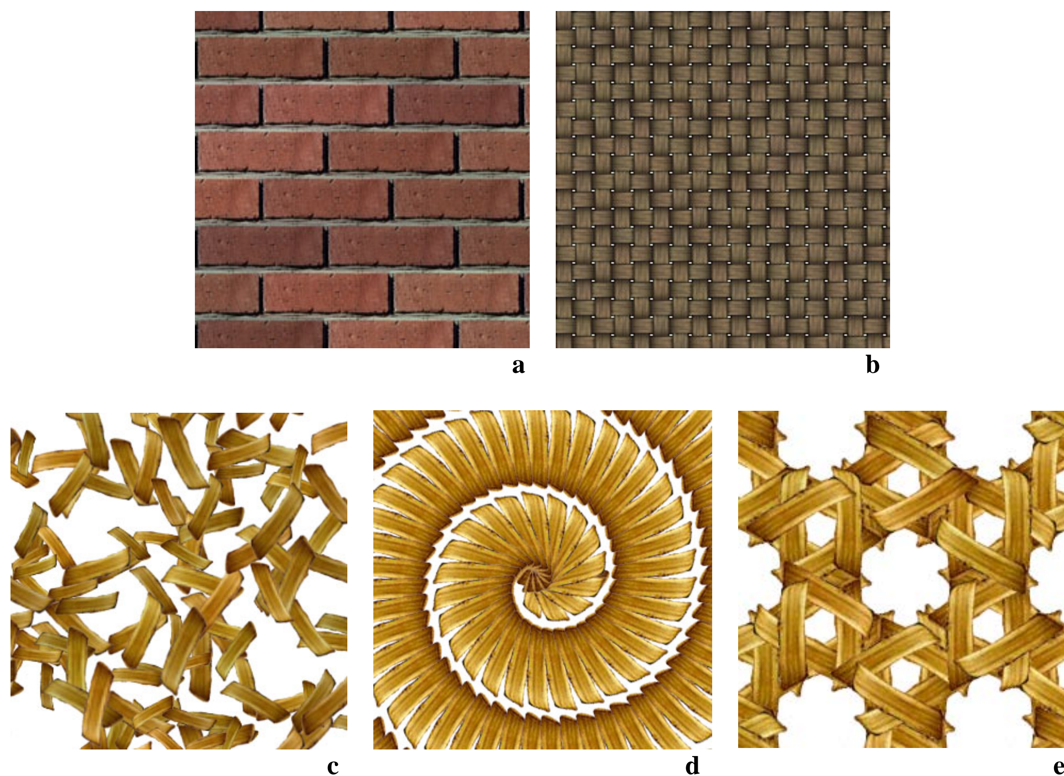


Figure 1: Five patterns generated by scripted pattern fills in Photoshop CC, with default settings: Brick Fill (a), Cross Weave (b), Random Fill (c), Spiral (d), and Symmetry Fill (e).

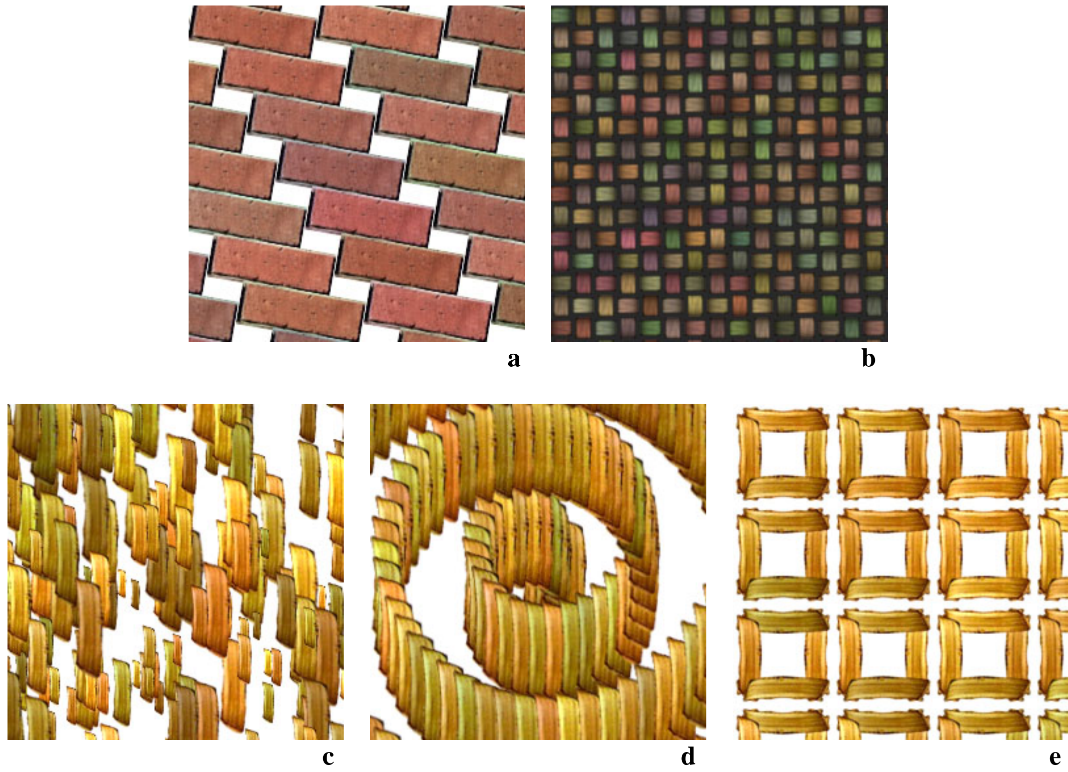


Figure 2: The same five patterns as in Figure 1 with non-default user settings: Brick Fill (a), Cross Weave (b), Random Fill (c), Spiral (d), and Symmetry Fill (e).

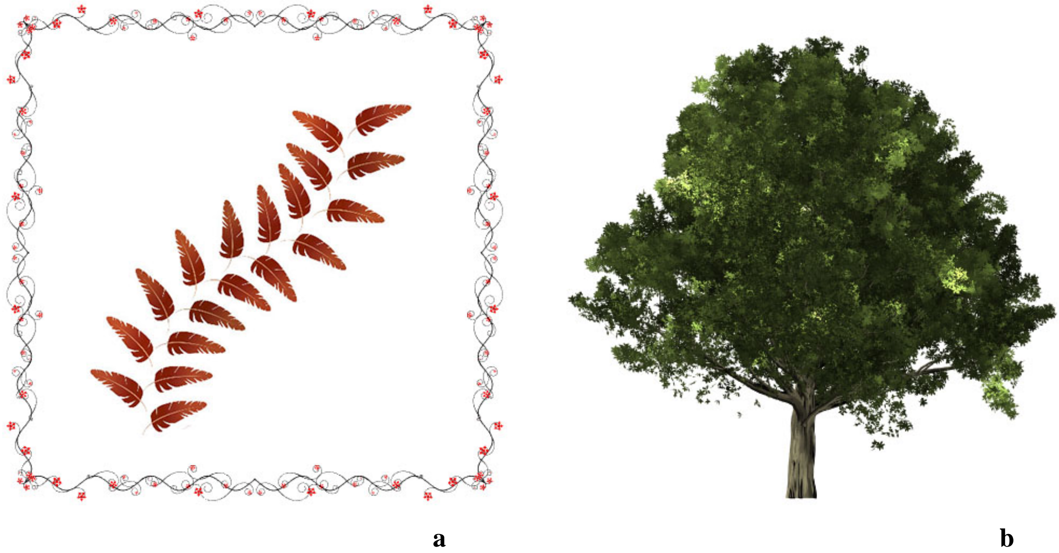
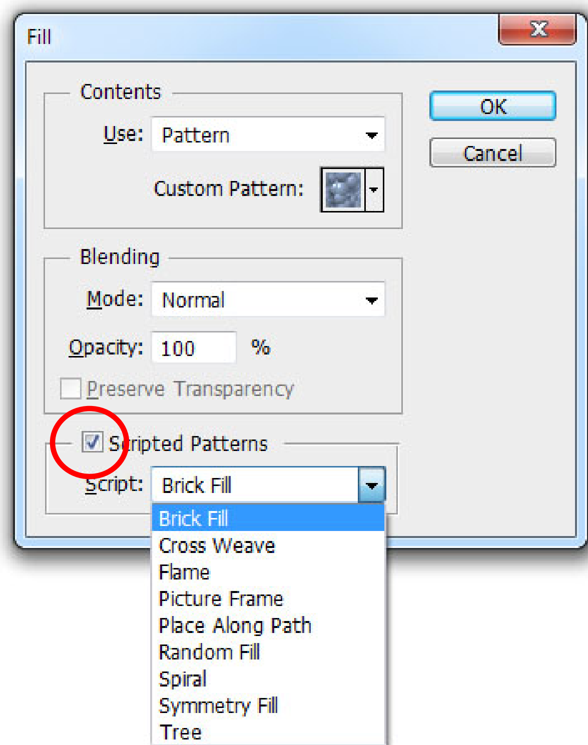


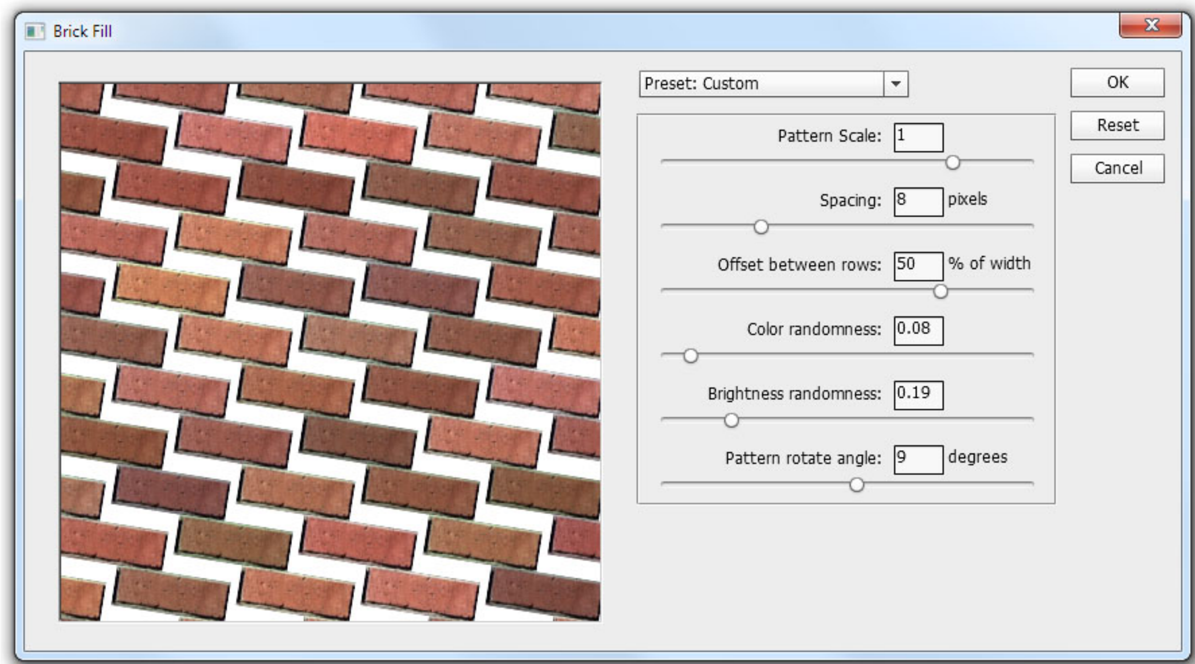
Figure 3: Patterns generated by 3 new scripts in Photoshop CC:
Place Along Path inside a Picture Frame (a), Tree (b).

1 Scripted Patterns in Photoshop CC

Scripted patterns are invoked from Fill dialog in Photoshop CC. There are 8 scripted pattern fills in the Fill dialog, accessed by right clicking on a selection and choosing *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*. Many of these scripts place the selected Custom Pattern in a variety of ways. Two scripts, Picture Frame and Tree generate geometry without using the custom pattern.



Before a scripted pattern is created, you can modify the various parameters for each fill in a pop-up dialog. The dialog also contains a preview that is automatically scaled to show only a local part of the resulting pattern – so that the style of the pattern is understood (see below).



Photoshop CC includes eight JavaScript files that define eight distinct Fill patterns. These scripts are executed by the Deco framework, which is a scriptable environment that is tailored for creating procedural patterns. All these JavaScript files are located in the following directory:

Windows 32 bit:
Program Files (x86)\Adobe\Adobe Photoshop CC\Presets\Deco

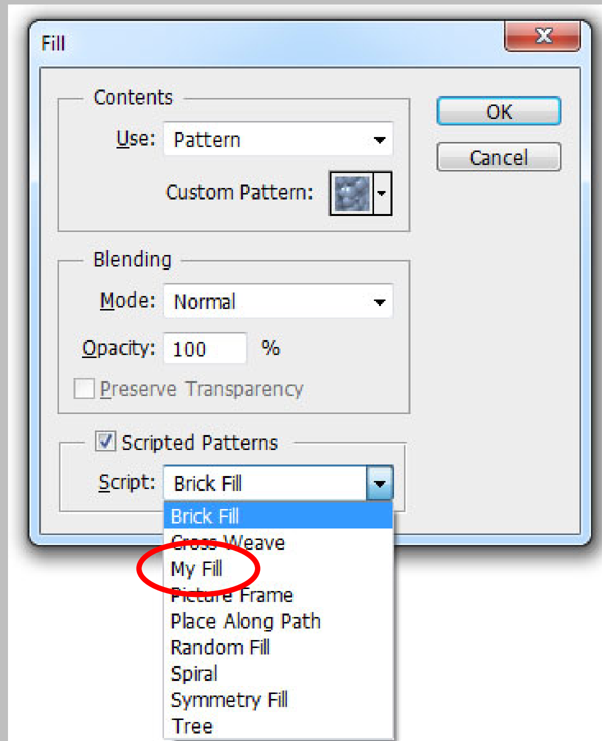
Windows 64 bit:
Program Files\Adobe\Adobe Photoshop CC (64 Bit)\Presets\Deco

Mac:
/Applications/Adobe Photoshop CC/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 CC. Here is the first task:

Task 1: Add a new tool

1. Go to the directory where the Fill 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.



Note that you can also place your scripts to a user directory C:\Users\yourUserName\AppData\Roaming\Adobe\Adobe Photoshop CC\Presets\Deco

On mac the directory is
~Library/Application Support/Adobe/Adobe Photoshop CC/Presets/Deco.
Keep in mind that unless you run
chflags nohidden ~/Library
in a terminal, you will not see the Library folder in Finder.

2 Deco Framework Overview

The core of the Deco framework is a procedural engine. The procedural engine is basically a C++ class that is connected to Photoshop via an app-specific interface (see Figure 4). After the user invokes a specific scripted pattern, Photoshop informs the engine about what script to load and what custom pattern (an image patch) to use in the fill.

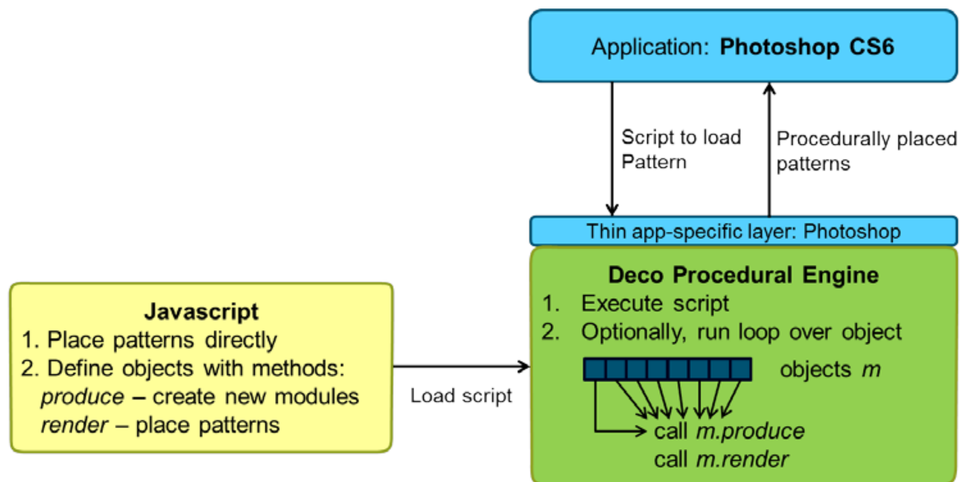


Figure 4: Schematic model of the Deco framework (same in Photoshop CC)

There are two ways of specifying procedural patterns and additional geometry primitives in Photoshop using a Deco 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. Additional geometry can be placed as well (as in the case of Picture Frame or Tree script).

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 to create new objects. The *render* method is used to place the input pattern according to the object's parameters. Refer to section 5.7 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 and geometry primitives 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 another part by running the simulation loop.

3 Defining Scripts

Scripted Pattern Fills in Photoshop CC are defined using scripts, called scriptals. These scriptals are based on an ExtendScript, with some additional predefined objects.

It is important to note that most functionality available in regular Photoshop scripts is available in Deco scripts as well. The exceptions are functions that modify layers or selection (using those will result in undetermined outcome). The preview dialog, for example, was fully built using Photoshop script functionalities.

3.1 Predefined Objects

When Deco scripts are executed, the JavaScript engine (the same as regular Photoshop scripts use) is initialized with a set of predefined objects that are used by the scriptal to communicate with the procedural engine class or to perform various tasks that are often needed in a procedural pattern specification. In Photoshop CC, the following objects are defined:

- *Engine*: facilitates communication with the procedural engine class;
- *RenderAPI*: used to place the input pattern and additional geometry primitives 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 supports full 3D, but the pattern fills in Photoshop CC use only the first two dimensions for display.
- *Symmetry*: used to create various symmetries.
- *DecoGeometry*: stores geometric primitive, such as lines, Bezier curves, or polygons;
- *Curve, GenCylPoint*: used to define cross-section and profile curves and control points of a generalized cylinder (not used by any shipped scripts).

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 Script

Before we begin detailing an example of a script, it will be useful for you to know how to obtain debugging information when running a Deco Script.

Please refer to Task 2 to see how a value from the scriptal can be displayed during debugging of your scripts.

Task 2: Print out the input pattern size

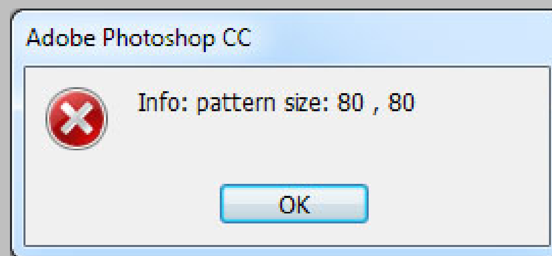
1. Open the file *My Fill.jsx* that you created in Task 1, preferably in the *ExtendScript Toolkit,* but any text editor will suffice.
2. Scroll past the *modelParameters* definition.
3. Check out the two commands defining *pattern*, and *patternSize.*
4. Add the following line just after the line defining *patternSize*.

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

   Alternatively, you can also use the Photoshop script command alert

   ```
   alert("Pattern size: " + patternSize.x + " , " + patternSize.y)
   ```

5. Go to pattern fill, and select *My Fill* on the first default input pattern. Before the preview dialog appears, you will get the following message.



You can see the message sent by the scriptal after the text *Info:*.

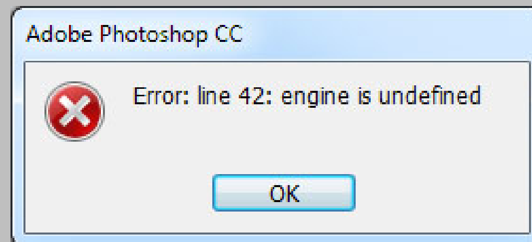
Please note that this functionality is intended only for purposes of learning about script writing and for debugging. It is not intended to be used during the normal operation of the scripted patterns.

A similar message is printed when there is a parsing error while executing the script, see Task 3.

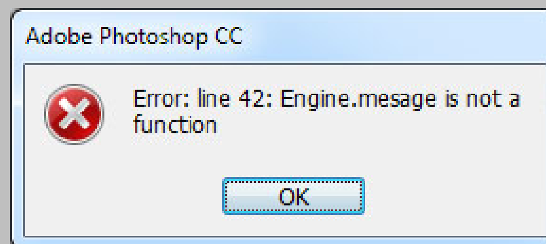
Task 3: Learn about error messages

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*.

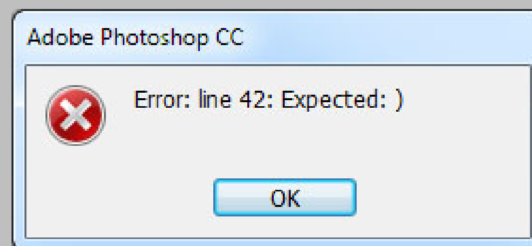
   engine.message("pattern size: ", patternSize.x, " x ", patternSize.y)
3. If you apply the *My Fill* now, you will see the following error message:



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 scriptal. The input custom pattern will be placed one after another in a grid layout. We can achieve this 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 with a given 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 command 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.

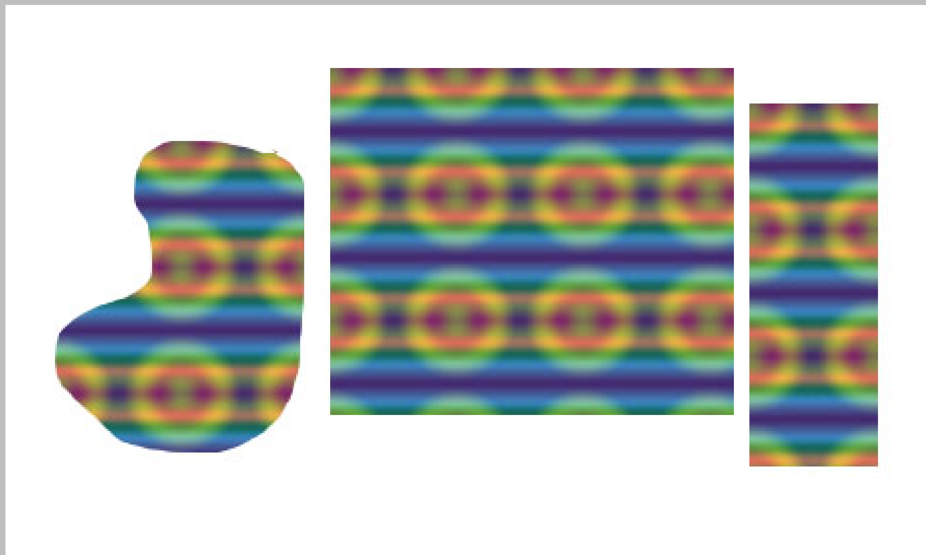
Task 4: Implement default pattern fill (almost)

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:

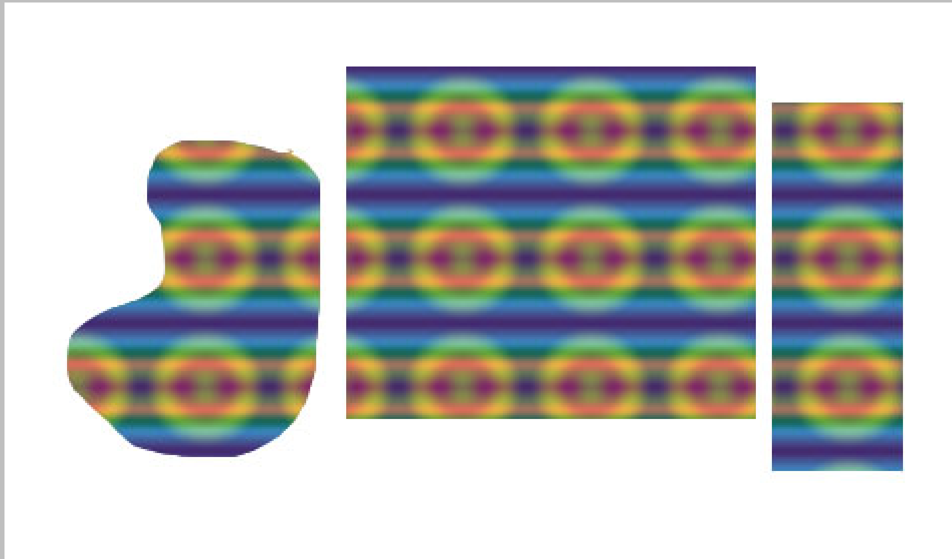
```
RenderAPI.translate ( -(origin.x % patternSize.x), -(origin.y % patternSize.y))
```

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)
```

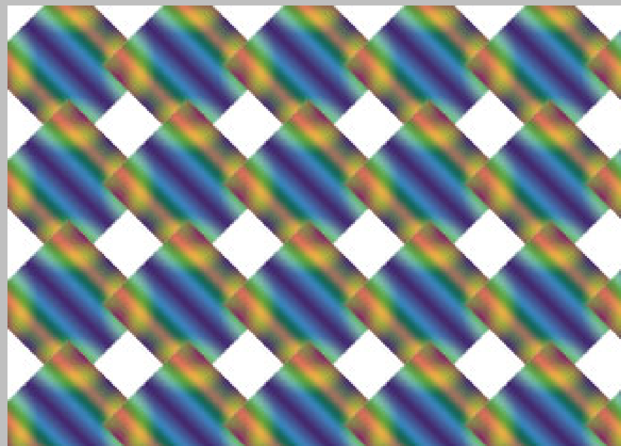
where the angle is specified in degrees. See Task 6 for an example.

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



.

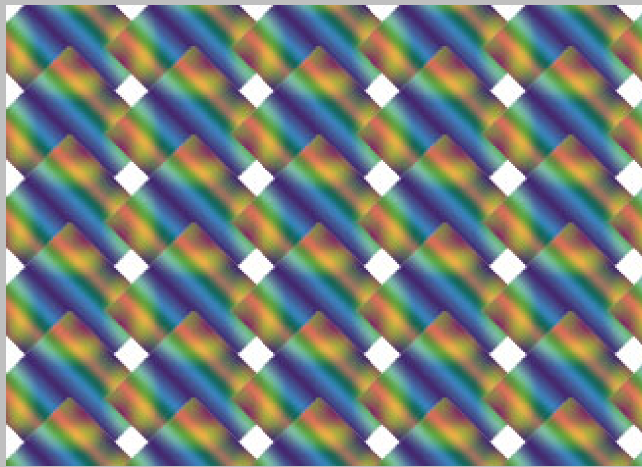
As you can see, the patterns in Task 6 overlap each other based on the order in which they were placed. The Fill started in the first row, left to right, then the second row, etc. This corresponds to the Photoshop's *Normal* blend mode. If the pattern is transparent, the Deco engine will use the transparency 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 Controlling Model Parameters

You are probably thinking would not it be nice to be able to change the rotation angle or spacing without having to edit the script? It was not possible in Photoshop CS6, but it is possible in Photoshop CC. You can encapsulate some parameters in an object called *modelParameters*, define a dialog structure, move drawing commands to a run method, and invoke a predefined script *_Deco Menu.jsx* from your script. The *_Deco Menu* script will build a dialog with a preview panel and once you choose desired parameter it will create the desirable fill pattern.

See Section 5.3 for more information and check out the listing on the following page.

```
modelParameters = {
    angle : 45,      // rotation angle
    offset : 100,    // offset between rows, between 0 and 100%.
}

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

function run (api, parameters, scale)
{
   var offset = parameters.offset/100
   api.translate (patternSize.x/2, patternSize.y/2 * offset)
   for (var y = 0; y < outputSize.y + patternSize.y;  y+= patternSize.y * offset)
   {
      api.pushMatrix()
      for (var x = 0; x < outputSize.x + patternSize.x;  x+= patternSize.x)
      {
         api.translate ( -(origin.x % patternSize.x), -(origin.y % patternSize.y))

         api.pushMatrix()
         api.rotate(parameters.angle)
         pattern.render(api)
         api.popMatrix()

         api.translate(patternSize.x, 0)
      }
     api.popMatrix()
     api.translate(0, patternSize.y * offset)
   }
}

var decoMenu = {    //  an object that defines the menu
  menuTitle : "Grid Fill",
  panels : [
   { panelName : "",
     panelMenu : [
      { itemName : "Rotate angle:",  itemUnit : "degrees",  itemType : 'slider',
        itemValue : modelParameters.angle, itemMin : -180, itemMax : 180, itemStep : 1,
        varName : 'angle'  },
      { itemName : "Offset:" , itemUnit : "%",  itemType : 'slider',
        itemValue : modelParameters.offset, itemMin : 0, itemMax : 100, itemStep : 0.1,
        varName : 'offset'  }
      ] }
  ]  // end of panels
 }; // end of menu

Engine.evalFile ("_Deco Menu.jsx") // Call Photoshop Script that creates the dialog

if (typeof skipRun == 'undefined' || !skipRun)
   run(RenderAPI, modelParameters, 1)
```

Table: Version of the Grid Fill script that uses a dialog to control the angle and the offset.

3.2.5 Changing Blend Mode

You can control the blend mode used when patterns or geometric primitives are placed. Due to limitations of OpenGL accelerated drawing, only 3 out of numerous Photoshop blend modes are supported: Normal, Multiply and Screen. A blend mode can be changed at any time using command

```
RenderAPI.setParameter(kpsBlendMode, mode)
```

Where mode is one of *kpsBlendNormal*, *kpsBlendMultiply*, and *kpsBlendScreen*.

If you really need to use other blend modes, it is possible, but only for patterns (not for other geometric primitives). You have to disable the OpenGL rendering for patterns using

```
RenderAPI.setParameter(kpsUseOpenGL, 0)
```

This has to be done at the beginning of the script, before any drawing happens. In case of using a Deco dialog and run method, it has to be done outside the run method. The flag is copied to the preview api automatically.

Once you disable OpenGL drawing, 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. Also, by disabling OpenGL the drawing will be slower.

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 two lines before the first for loop:

```
RenderAPI.setParameter(kpsUseOpenGL, 0)
pattern.setParameter (kpsPatternBlendMode, kpsBlendLighterColor)
```

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

Compare the result with the one from Task 6.

.

3.2.6 Modifying Color – Color Blend Mode

You may notice that the Deco scripted fill patterns shipped with Photoshop CC 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 using the command:

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

specifying the value for the red, green, and blue channels. The color of the pattern will be multiplied by this color. Note that the values of the color components used to multiply the pattern color can be above 1 (where 1 is the normal intensity).

To use other Photoshop blend modes to modify the color of the input pattern you need to disable OpenGL again as in the previous section. 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 3.2.5.

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.

3.2.7 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)
```

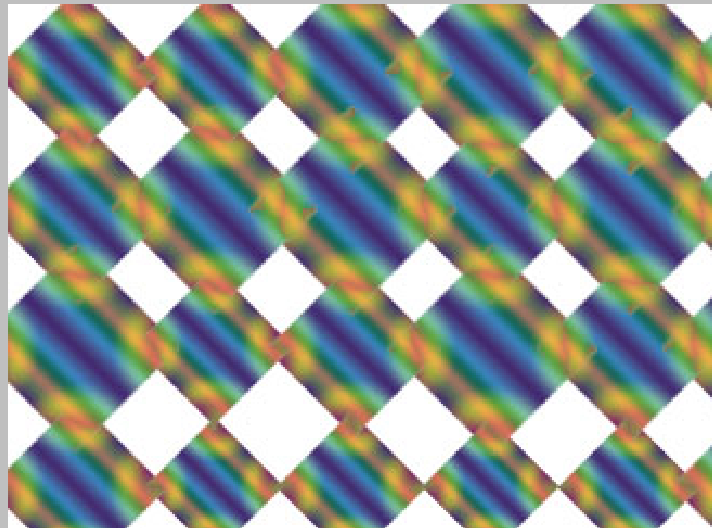
Keep in mind that the commands *translate*, *rotate*, and *scale* are additive, thus if you rotate twice by 20 degrees, for example, the resulting rotation will be 40 degrees.

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:



3.2.8 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 *Engine.rand*, 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 the following seed:

```
seed = (row * 214013+ c * 2531011) % 0x7fffffff
```

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 6 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 5, on the left.
4. Add the following two lines just before *pattern.render*:

```
RenderAPI.scale(Engine.rand()*0.1 + 1)
RenderAPI.rotate(-4 + Math.floor(Engine.rand()*60) / 7.5)  // 60 distinct rotations
```

5. Run the fill again and you will get the result from Figure 5, on the right.

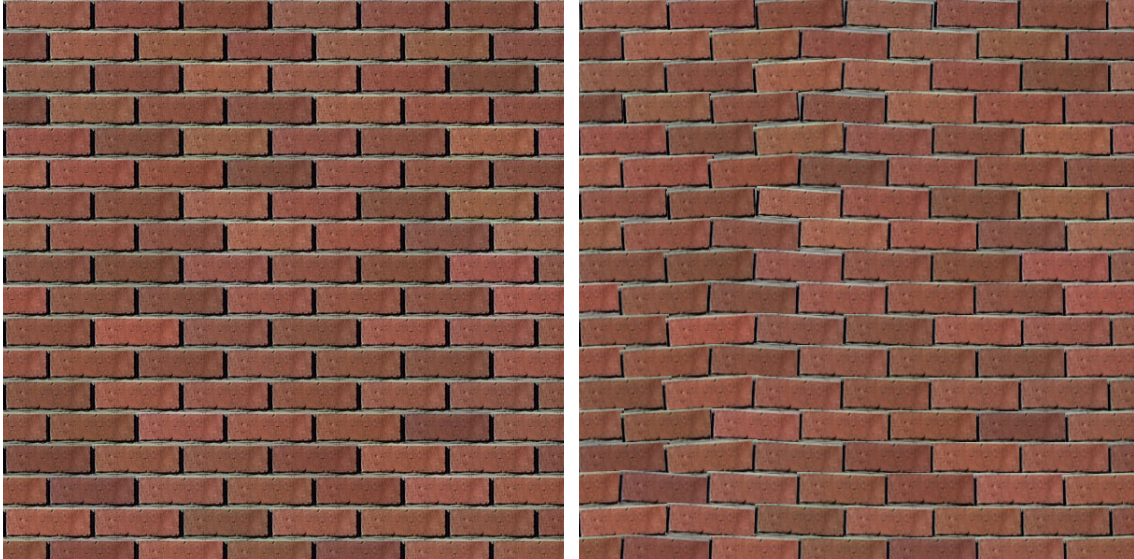


Figure 5: An effect of random scaling and rotation (on the left) applied to the default *Brick Fill* (on the right).

4 Drawing Geometric Primitives in Scripts

Since OpenGL is used for rendering the patterns internally, it is possible to create much more diverse results than those in Photoshop CS6. The differences are enumerated in this section.

The Deco drawing API in Photoshop CC supports the following new features:

- The user can draw 2D primitives, such as lines, Bezier curves, polygons, or even 3D shapes (that are flattened for display);
- The placement of patterns is hardware accelerated resulting in significant speed improvements compared to CS6.

Geometric primitives can be defined in two ways. First, it is possible to draw them directly using methods of the RenderAPI (which is visible as api in the run method if a dialog is defined). It is possible to draw circles, arcs, points, lines, polygons (filled or not), and Bezier curves. See more details in Appendix D.1 Drawing Methods of RenderAPI.

The second approach is to create a *DecoGeometry* object, specify add the primitive to it and then draw the object. See more details in Appendix G. Object DecoGeometry.

Following is an example of a script that draws several 2d primitives:

```
Engine.setSceneBBox(0,10,0,10)

RenderAPI.Color(kStrokeColor, 0.8, 0, 0)
RenderAPI.Line (1,1, 9,9)
var pt1 = new Vector2(1,9)
var pt2 = new Vector2(9,1)
RenderAPI.Line(pt1, pt2)

RenderAPI.Bezier(pt1, new Vector2(4,6), new Vector2(6,6), new Vector2(9,9))

RenderAPI.Color(kFillColor, 0, 0.8, 0)
RenderAPI.Polygon([ pt1, new Vector2(5,5), new Vector2(1,1) ])

var geom = new DecoGeometry
geom.addPolygon ([ pt2, new Vector2(5,5), new Vector2(9,9) ])
geom.addBezier(new Vector2(1,1), new Vector2(4,4), new Vector2(6,4), pt2)

geom.render(RenderAPI)
```

This script draws 2 lines, 2 Bezier curves, and 2 polygons. One polygon and one Bezier curve are defined using a *DecoGeometry* object. The first command, *Engine.setSceneBBox* sets the range of the coordinates used by the script – this is an option to using the default size in pixels. Note that it is possible to set two specific colors, the stroke color and the fill color.

As mentioned above, it is possible to create a 3D geometry as well. In fact, all points specifying the geometry primitives mentioned above can have the third coordinate that controls the distance of the point from the view plane. In addition, it is possible to define meshes (see *DecoGeometry* in Appendix G. Object DecoGeometry) and generalized cylinders, which are formed by 2d curves swiped in 3D along another curve (see Appendix H. Generalized Cylinders). The results are flattened to the 2D layer, Deco will not create a 3D layer in Photoshop. See Section 5.6 for an example.

5 Advanced topics

In this section we will cover more advanced topics related to use and operation of Deco pattern fills.

5.1 Use of Paths

The script writer can get information about Photoshop paths that are selected. You can get the selected path as an array of *DecoGeometry* objects (see Appendix G) as shown below

```
var paths = api.getParameter (kpsSelectedPaths)
```

The *api* is the *RenderAPI* in case the preview dialog is not used or the parameter of the *run* method if the dialog is used (see Section 5.3.3).

You can trace each path by using the following methods:

```
getValue (kGetGeometryLength)
getValue (kGetPointAlongGeometry, distanceAlongPath)
getValue (kGetNormalAlongGeometry, distanceAlongPath)
```

Once you obtain a length of each path, you can get any point along it, with the normal (a vector perpendicular to the path at that point). A path in Photoshop can be discontinuous – each path is a collection of strokes made by the pen tool, until a new path is selected in the path panel. If you need to process only continuous strokes of a pen, make each stroke in a separate path or add a script code that will detect discontinuities as you process each path in your *paths* array.

Here is an example of placing circles along all selected paths:

```
var paths = api.getParameter (kpsSelectedPaths)

for (var p = 0; p < paths.length, p++)

{
   var len = getValue (kGetGeometryLength)
   for (var dist = 0; dist < len; dist += 10)
   {
      var pt = getValue (kGetPointAlongGeometry, dist)
      // normal is not needed

      api.Circle(pt, 5) // point and radius
   }
}
```

5.2 Instancing

In case you are drawing the same geometric primitive or group of primitives multiple times you should use instancing. Since each instance is kept in the graphics hardware memory you can achieve significant speedups.

The following methods are used to define, test, and delete an instance:

```
api.defineInstance (instanceID1, instanceID2, …)
api.endInstance ()
api.drawInstance (instanceID1, instanceID2, …)
api.instanceExists (instanceID1, instanceID2, …)
api.deleteInstance (instanceID1, instanceID2, …)
```

The *api* is the *RenderAPI* in case the preview dialog is not used or the parameter of the *run* method if the dialog is used (see Section 5.3.3). Each instance is identified by one or more identifiers that are internally concatenated together. Each identifier should be a string or a number.

After you call *defineInstance* and it returns 1, you can define your primitives. Any call made to api will be recorded, until *endInstance* is called. An instance is drawn using the

call *drawInstance*. Note that you can nest instance by using *drawInstance* with a different id when defining another instance. You cannot nest *defineInstance* calls.

Here is an example:

```
drawSpirals = fuction (api)
{
   // define instances
   for (var index = 0; index < numSpirals; index++)
   {
      if (api.instanceExists("spiral", index))
         api.deleteInstance("spiral", index); // delete if exists

      if (api.defineInstance("spiral", index))
      {
         // define spiral
         …
         api.endInstance()
       }
    }

    // draw instances
   for (var index = 0; index < numSpirals; index++)
   {
      if (api.instanceExists("spiral", index))
         api.drawInstance("spiral", index)     // draw a spiral
   }
}
```

Since all *api* calls between *defineInstance* and *endInstance* are stored, you can not only specify primitives, but also their color or transformation. These are 'baked' into the instance. If you want to define an instance without color and then change the color before calling the *drawInstance*, there is a caveat. Since internally instances are stored as OpenGL display lists and OpenGL supports only one color, it is not possible to define two different colors (for fill and stroke) and then draw the instance. When an instance is being drawn only one color is used. If you want to have an instance with different fill and stroke colors, you should define two instances, one with the fill only and the other one using line primitives for the boundaries.

5.3 Defining Control Dialogs

Deco scripts allow the developer to create parametric procedural models. It is important for the user to be able to modify the parameters without having to know about the location of the script and having to edit the script. On the other hand, each script can have different sets of parameters, which cannot be captured by one static dialog.

Deco scripts provide a compromise. Each script can define its own dialog that is built by the scripting engine. The advantage is that the user can modify the script parameters and

see a preview of the result. The disadvantage is that not all features of static Photoshop dialogs are available and the Deco dialog has a slightly different feel.

5.3.1 Script Parameters to be Modified

The parameters that you want the dialog to control should be all in one object, named *modelParameters*. For example, here are the parameters for the Brick Fill script, with their default values:

```
modelParameters = {
   // Offset between rows of pattern expressed in percent of pattern width.
   // For example 50% is half the width.
   offset : 50,    // use a value between 0 and 100. The default is 50.

   // Spacing between patterns in pixels.
   // For example,1 creates 1 pixel gap between patterns
   spacing : 0,    // use a value between -10 to 20. The default is 0.

   // Variation of color of the pattern.
   // For example, value of 0.2 means that each of the red, green, and blue color
   // will be multiplied by a DIFFERENT random value from interval 0.8 and 1.2.
   // Set to 0 if you do not want to modify the pattern color.
   colorRandomness : 0.05,    // use a value between 0 and 1. The default is 0.05.

   // Variation of pattern brightness.
   // For example, value of 0.6 means that each of the red, green, and blue color
   // will be multiplied by THE SAME random value from interval 0.4 and 1.6.
   // Set to 0 if you do not want to modify the pattern brightness.
   brightnessRandomness : 0.1,   // use a value between 0 and 1. The default is 0.1.

   // Rotation of individual patterns.
   rotateAngle : 0 // Use a value between -180 and 180. The default is 0.
}
```

5.3.2 Dialog Definition

To define a dialog's menu panel, in which you can allow the user to modify selected parameters of the script, you define its name, size, and individual menu items. Menu items are displayed in the order they are specified.

The object used to define the menu has the following structure:

```
  var decoMenu = {
     menuTitle : 'My Menu',
     menuBackground : [0.93, 0.93, 0.93, 1],
     previewBackground : [1, 1, 1, 1],
     panels : [
      { panelName : 'Panel 1',
         leftColumnWidth : 180,
         editTextWidth : 35,
```

```
        unitsWidth : 65,
        dropdownlistWidth : 160,
        panelMenu : [
         { itemName : 'Item 1 (range 1, 10)',  itemUnit : 'pixels', itemType : 'edittext',
           itemValue : 5, itemMin : 1, itemMax : 10, varName : 'var1' },
          …
        ] }
    ] // end of panels
  } // end of menu
```

You can keep most of the data as shown, just change the *menuTitle* and provide your own list of menu items.

There are several types of parameter control that you can expose in the menu:

1. Text input

This item allows the user to enter a number or a text. The item is specified as follows:

```
 { itemName : 'Item 1 (range 1, 10)',  itemUnit : 'pixels', itemType : 'edittext',
   itemValue : 5, itemMin : 1, itemMax : 10, varName : 'var1' }
```

The *itemName* is the text that will be displayed to the left of the text input box. The *itemUnit* is the text displayed to the right of the text input box. The *itemType* should be 'edittext'. The *itemValue* is the initial value (often you can use *modelParameter.variable*). The *itemMin* and *itemMax* define the range in case the input is a number. The variable name is the name of the variable in the *modelParameters*. For example if you want to modify *modelParameters.density, varName* would be 'density'. Note that all values of the item have to be specified.

2. Drop down list

This item allows the user to select from a pull down list of options. The item is specified as follows:

```
{ itemName : 'List name',  itemUnit : '', itemType : 'dropdownlist',
  itemList : ['selection 1', 'selection 2', {item: 'selection3', image: 'filename'}],
  itemValue : 2, itemMin : 0, itemMax : 0, varName : 'var2',
  disableItems : [ // optional
      [0, [2,3]], // gray out the third and fourth menu item for 'selection 1' (indexed from 0)
      [1, [3]]     // gray out the fourth menu item for 'selection2'
     ]  },
```

The *itemName* is the text that will be displayed to the left of the pull down list. The *itemUnit* is ignored. The *itemType* should be 'dropdownlist'. The itemList is an array of strings that appear in the pulldown list. Instead of a string, you can define an object with a string *item* and an image filename – in this case the string will be preceded by the image. The system does not scale the image so it is recommended to use a low resolution icon only.

The *itemValue* is the initial selection – indexed from 0. The *itemMin* and *itemMax* are ignored. Optionally, you could specify which items in the current panel will be disabled (grayed out) for a specific selection. The item *disableItems* is an array of arrays. Each array specifies a pulldown list item index (from 0) and an array of indices of menu items that are to be grayed out when the selection is made.

3. Checkbox
This item allows the user to select a binary value using a checkbox. The item is specified as follows:

```
{ itemName : 'Checkbox name',  itemUnit : '', itemType : 'checkbox',
  itemValue : true, itemMin : 0, itemMax : 0, varName : 'var3'  },
```

The *itemName* is the text that will be displayed to the right of the pull down list. The *itemUnit* is ignored. The *itemType* should be 'checkbox'. The *itemValue* is the initial value, *true* of *false*. The *itemMin* and *itemMax* are ignored.

4. Colorpicker
This item allows the user to select a color. The item is specified as follows:

```
{ itemName : 'Color', itemUnit  : '',  itemType : 'colorpicker',
  itemValue : [1, 1, 1], varName : 'color1'  },
```

The *itemName* is the text that will be displayed to the right of the color swatch. The *itemUnit* is ignored. The *itemType* should be 'colorpicker'. The *itemValue* is the initial value, an array of red, green, and blue color components, each in the interval [0,1].

5. Slider
This item allows the user to enter a number either directly by typing it in or by moving a slider. The item is specified as follows:

```
{ itemName : 'Slider name', itemUnit  : 'degrees',  itemType : 'slider', itemValue : 0,
  itemMin : -45, itemMax : 45, itemStep : 1, varName : 'angle1'  }
```

The *itemName* is the text that will be displayed to the left of the text input box. The slider will be placed below the text and the input box. The *itemUnit* is the text displayed to the right of the text input box. The *itemType* should be 'slider'. The *itemValue* is the initial value. The *itemMin* and *itemMax* define the range and the slider is built based on these two values. The *itemStep* specifies the step in which the number changes when the slider is moved.

Optionally, you can link two sliders together by using *itemLEQitem : item_index* and *itemGEQitem: item_index*. In the first case, the current slider value is forced to be less or equal to value in a menu item *item_index* (indexed from 0), in the second case the value is forced to be greater or equal to the referenced menu item. This mechanism can be used when two sliders control the minimum and the maximum value of some range.

5.3.3 Run Method for Preview

Once you define the menu object, you give control to the script *_Deco_Menu.jsx* that opens the dialog with the menu and allows the user to select the input values. Once the user is satisfied with the selection the object *moduleParameters* is updated and the control comes back to the calling script:

```
Engine.evalFile ("_Deco Menu.jsx")
```

While the dialog is up, as parameter values are being changed, a preview image is computed (on mouse up only). To facilitate that you have to define a method *run* where you do all the model definition and drawing. The parameters of the method are *api*, specifying the output rendering api, *parameters*, specifying the model parameters, and *scale*, specifying additional scale. You need to use these values inside the run function, do not use the global *RenderAPI* and *modelParameters*. The preview image is generated using a different parameter object and using a different api than the final run.

If you are only drawing directly to the rendering api, you just define the run method:

```
function run (api, parameters, scale)
{
   // get the size of the output area – you have to do it inside run method
   var outputSize = api.getParameter(kpsSize)
   // get the location of the top left corner of the bounding rectangle around the selection
   var outputOrigin = api.getParameter(kpsOrigin)

   … // define the pattern
}
```

If you are using the simulation loop (see Section 5.7), you have to take care of the simulation loop and one *render* call at the end of your *run* function.

```
function run (api, parameters, scale)
{
   // get the size of the output area – you have to do it inside run method
   var outputSize = api.getParameter(kpsSize)
   // get the location of the top left corner of the bounding rectangle around the selection
   var outputOrigin = api.getParameter(kpsOrigin)

   … // create the simulation modules, compute stepsNeeded

   if (parameters == previewParameters) // only for preview run
   {
      for (var step = 0; step < stepsNeeded; step++)
         Engine.produce();

      Engine.render (api)
    }
}
```

The scale parameter is used to scale the preview up and down. It may be desirable to show the whole work area in the preview but this would be not only prohibitively slow for large areas, but the size of the patterns in the preview may be so small that the user may not discern the local structure of the result (in case of the Tree script, for example, the local preview does not make sense and that is why the preview is pre-computed for a predetermined set of input values). That is why by default the preview is scaled so only a local part of the results is seen. Generally, the size is determined so that the preview contains about 3-5 input pattern at scale 1 along the width. If you scale the pattern down, the preview will contain more of it. This default behavior can be overridden and you can set scale to whatever you see fit in your scripts.

Note that all modules added to the Deco engine using *addModule* are deleted after the run method is called during the preview. You do not have to remove them from the engine yourself.

Since all model definition is done inside the run method, you need to call it at the end, after the _Deco_Menu script is executed and the user selects the desired parameter values. If the user cancels the selection, a skipRun variable is set to true, thus you should test it before calling the final run:

```
if (typeof skipRun == 'undefined' || !skipRun)  // run unless we exited the preview window
   run(RenderAPI, modelParameters, 1)
```

5.4 Pattern Subregions

Any image represented as *Image* object, which includes also the user selected pattern, can be cropped into one or more regions using a command

```
var image1 = pattern.getSubregion(left, right, top, bottom)
```

The coordinates of the cropped region are in pixels, with 0,0 being on top left.

5.5 Troubleshooting OpenGL

If you have some issues with OpenGL in Scripted Patterns, you can try to trouble shoot them using a script *_Deco Settings.jsx* placed in the same directory as other scripts. The script is loaded before OpenGL is initialized both for main rendering and also for preview rendering and you can disable various parts or get additional debug information. Here is an example of such a script, with all values set to the default values:

```
var RenderAPI_settings =
{
        UseOpenGL : true,
        MaxOpenGLTileSize : 1024,
        UseMultiSampling : true,
        MaxSamples : 256,
        MaxBufferDepth : 32,
        OptimizeTileRendering : true,
        PerformGPUTest : true,
        DebugMessages : false
```

```
}

var DecoPattern_settings =
{
        UseOpenGL : true
}
```

For example, you can set the *UseOpenGL* flag to false to see whether OpenGL is indeed a culprit. Note that you cannot use that if you are actually defining primitives. You need OpenGL in that case, you can only disable it if your script places the custom pattern.

You can try to reduce the tile size to 512 or even 256 on cards with small amount of graphics memory, you can disable multisampling (although the results will be aliased), or set the maximum number of samples (to 8, for example).

5.6 3D Geometry

The following script is an example of drawing a 3D geometry primitive.

First a mesh is constructed from points, normals, and faces and added to a *Deco Geometry* object. Then we define the range of the visible area and whether the camera is orthographics or perspective. Then we define material properties of the surface and the light position and color. We set the light and the material, enable lighting, set the proper transformations and draw the cube.

```
cube = new DecoGeometry(new Frame3)
vertices = new Array(0)
normals = new Array(0)

vertices.push(new Vector3(-0.5, -0.5, -0.5))
vertices.push(new Vector3( 0.5, -0.5, -0.5))
vertices.push(new Vector3( 0.5, -0.5,  0.5))
vertices.push(new Vector3(-0.5, -0.5,  0.5))
vertices.push(new Vector3(-0.5,  0.5, -0.5))
vertices.push(new Vector3( 0.5,  0.5, -0.5))
vertices.push(new Vector3( 0.5,  0.5,  0.5))
vertices.push(new Vector3(-0.5,  0.5,  0.5))

normals.push(new Vector3(0,-1,0))
normals.push(new Vector3(0,1,0))
normals.push(new Vector3(-1,0,0))
normals.push(new Vector3(1,0,0))
normals.push(new Vector3(0,0,-1))
normals.push(new Vector3(0,0,1))

faces = new Array(0)

faces.push({ vertices: [4,3,2,1], normals: [1,1,1,1]}) // bottom
faces.push({ vertices: [5,6,7,8], normals: [2,2,2,2]}) // top
faces.push({ vertices: [1,2,6,5], normals: [5,5,5,5]}) // front
faces.push({ vertices: [4,3,7,8], normals: [6,6,6,6]}) // back
faces.push({ vertices: [2,3,7,6], normals: [4,4,4,4]}) // left
faces.push({ vertices: [4,1,5,8], normals: [3,3,3,3]}) // right

cube.addMesh({ vertices: vertices, normals: normals, faces:faces })

Engine.setSceneBBox (-1, 1, -1, 1, -1, 4) // orthographics projection
//Engine.setSceneBBox (-1, 1, -1, 1, 1.5, 4) // perspective projection

// set light and material - after the bounding box is defined
var light = new Light
light.setValue (kLightPosition, -2, 4, 10, 0)
light.setValue (kLightColor, 1,1,1)
var material = new Material
material.setValue (kColorAmbient, 0.2, 0.2, 0.0, 1)
material.setValue (kColorDiffuse, 1, 1, 0.2, 1)

RenderAPI.setLight (0, light)
RenderAPI.setMaterial (kMatFront, material)
RenderAPI.setParameter(kpsLighting, 1)

RenderAPI.translate(0,0,-2.5)
RenderAPI.rotate(45, new Vector3(1,1,1))
cube.render(RenderAPI)
```

5.7 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 7.

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 6 (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 */))
```

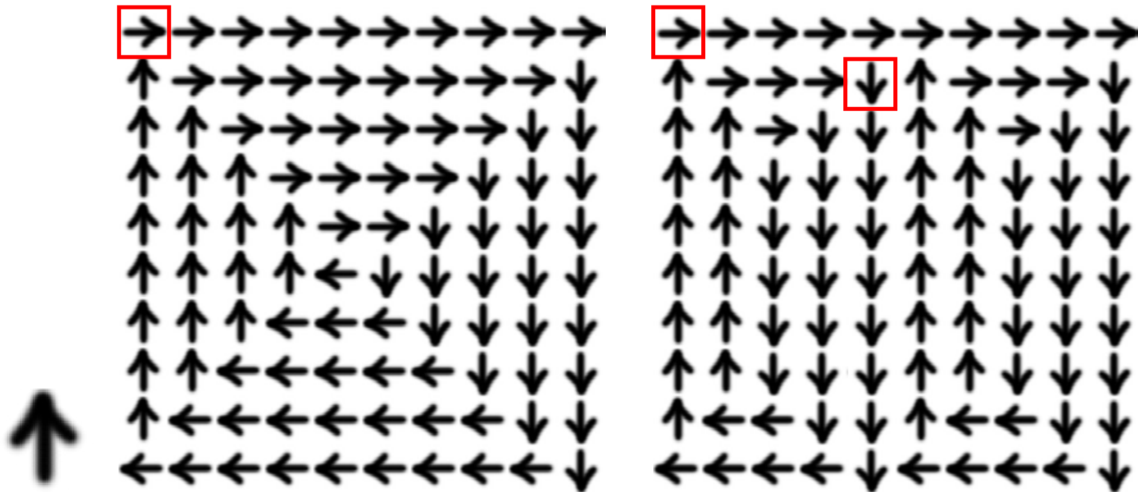


Figure 6: A more complicated pattern defined by custom scripts (not shipped with CS6). A simulation loop is executed on modules that place the input pattern (the arrow) in a row until an occupied element is reached and then they turn right. The pattern on the left started with one module and the pattern 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 completly 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.

5.8 Operation of Deco Scripts

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 7). 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 5.7).

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 7 illustrates the flow of control in the Deco framework.

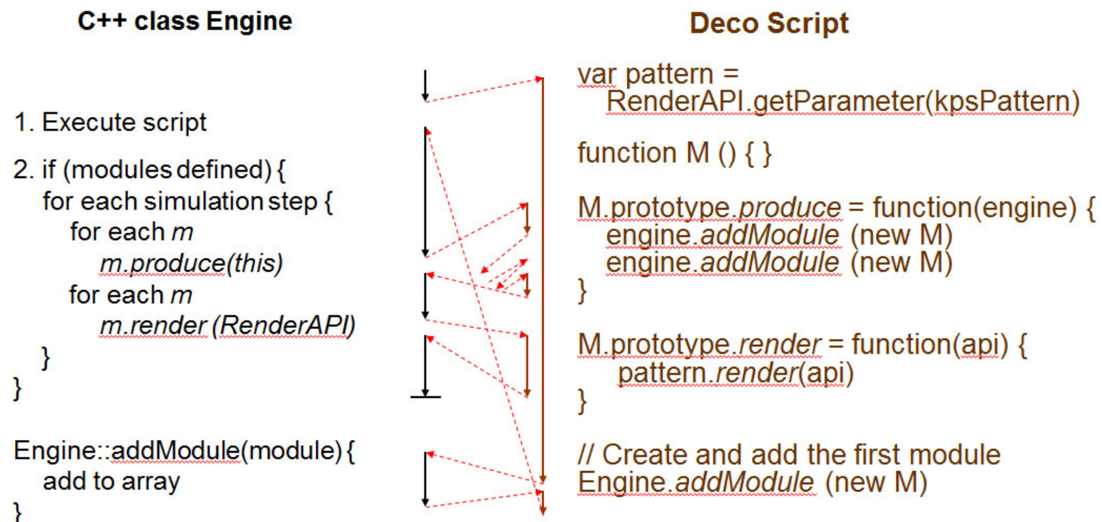


Figure 7: Example of the flow of control in the Deco framework during a simulation.

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.

5.8.1 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.

5.8.2 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.

5.8.3 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.

6 Performance

The performance of the scripted pattern fills depends on two components, the complexity of calculations needed to compute the size and position of all elements placed on the screen and the complexity of drawing all those elements.

Since the drawing is hardware accelerated using OpenGL and the drawing times are usually low. For very large documents the result is divided into tiles that fit into the graphics card memory and only objects overlapping a given tile are drawn. Of course if you define a very large document covered by many polygons of the document size, the drawing will slow down. As mentioned above it is still recommended to use instancing if an element is drawn multiple times (Section 5.2).

In more complex scripts such as the Picture Frame or Tree most of the time is spent in building the structure and by defining the geometry – that is then drawn quite fast.

Photoshop cannot know how long such a task would take without the script informing the engine about its progress. See the next section on how to control the progress bar from the script.

6.1 Controlling the Progress Bar

If a Photoshop task is expected to take more than a few second a progress bar appears to show progress. Since a script can take any time from a fraction of a second to tens of seconds, Photoshop does not know how to display the progress bar correctly.

If you know that your script will take longer than a few seconds, you should help Photoshop by informing it how the script execution is progressing. To do that you can wrap your functional parts with commands *kpsStartTask* and *kpsFinishTask* or *kpsStartSubTask* and *kpsFinishTask*.

For example, let us say you are creating 10,000 primitives in a loop and each primitive takes about the same time. You would use *kpsStartSubTask*, and additional two parameters specifying the current index of the subtask and the total number of subtasks.

```
for (var i = 0; i < 10000; i++)
{
  RenderAPI.command (kpsStartSubTask, i, 10000)
  // define your complex primitives
  …
  RenderAPI.command(kpsFinishTask)
}
```

This is good but we are not finished yet. This loop only defines the primitives, they are actually not drawn yet, even if you use RenderAPI calls like Line or Polygon etc. The drawing is happening later. As mentioned above, it usually takes longer to prepare the primitives than to draw them using OpenGL, but it still takes some time.

Let us say we did some testing and we know that it usually takes 85% of the total time to prepare the objects (you can use an *alert* after your loop to determine that). In this case you can wrap your loop with a *kpsStartTask* and *kpsFinishTask* commands. The additional parameter of *kpsStartTask* command specifies the operation of this task with respect to the remaining estimated time. In our case it would be 0.85 then:

```
RenderAPI.command(kpsStartTask, 0.85)
for (var i = 0; i < 10000; i++)
{
  RenderAPI.command (kpsStartSubTask, i, 10000)
  // define your complex primitives
  …
  RenderAPI.command(kpsFinishTask)
}
RenderAPI.command(kpsFinishTask)
```

Photshop performs its own timing thus if your estimates are way off, the progress bar may not appear at all - Photoshop may assume from the first few subtasks that all subtasks will be done within a few seconds – or the progress bar is jerky. The second issue is really hard to avoid since it is difficult to estimate how long will parts of the script take on different machines. You may experience the effects when running the Tree script for example.

Note that the progress bar is disabled during the preview operation when the preview dialog is up.

6.2 Rotating Patterns when OpenGL is Disabled

If you disable OpenGL drawing (for example, to get more advanced blend modes), the software is used to draw patterns and then it matters whether the input pattern is large or whether it is rotated or scaled. If you scale each placed pattern (in the script), there will be an impact to performance. Rotating the input pattern can be even costlier 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.

7 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 8.

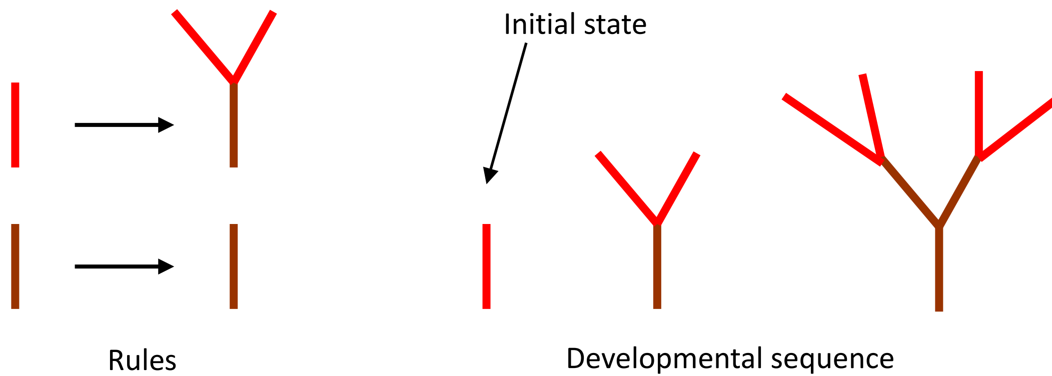


Figure 8: Example of a simple procedural model with two rules

Rules similar to those in Figure 8, with additional clipping when the branch reached out of a predefined shape have been used to generate branching structures in Figure 9. The leaves were added using additional rules.

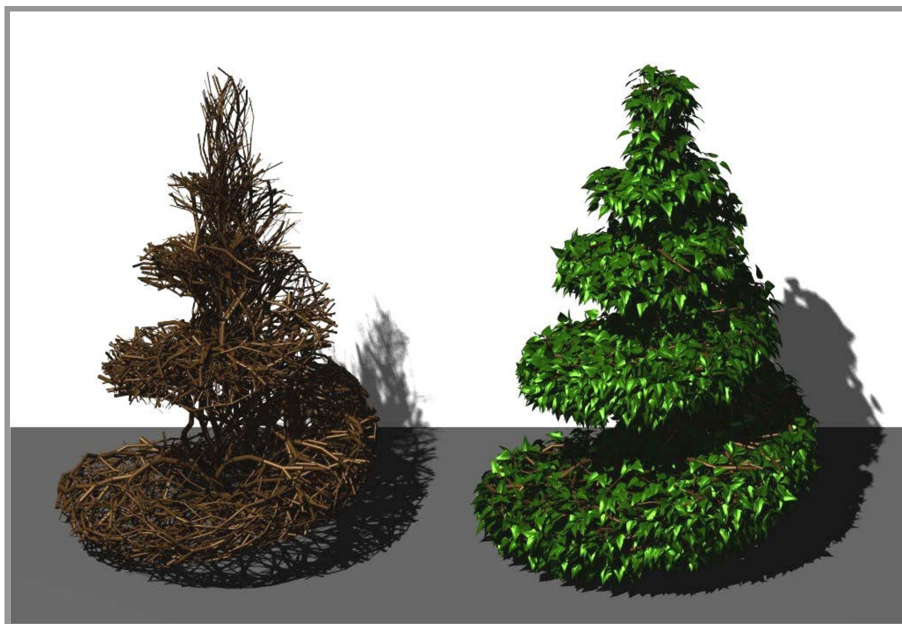


Figure 9: Procedurally generated branching structure clipped to a spiral shape *(from Siggraph '94 paper by Prusinkiewicz et al.).*

Procedural models can generate other structures than trees and bushes. Figure 10 shows several procedural ornaments, illustrating the potential of using procedural models in design applications.

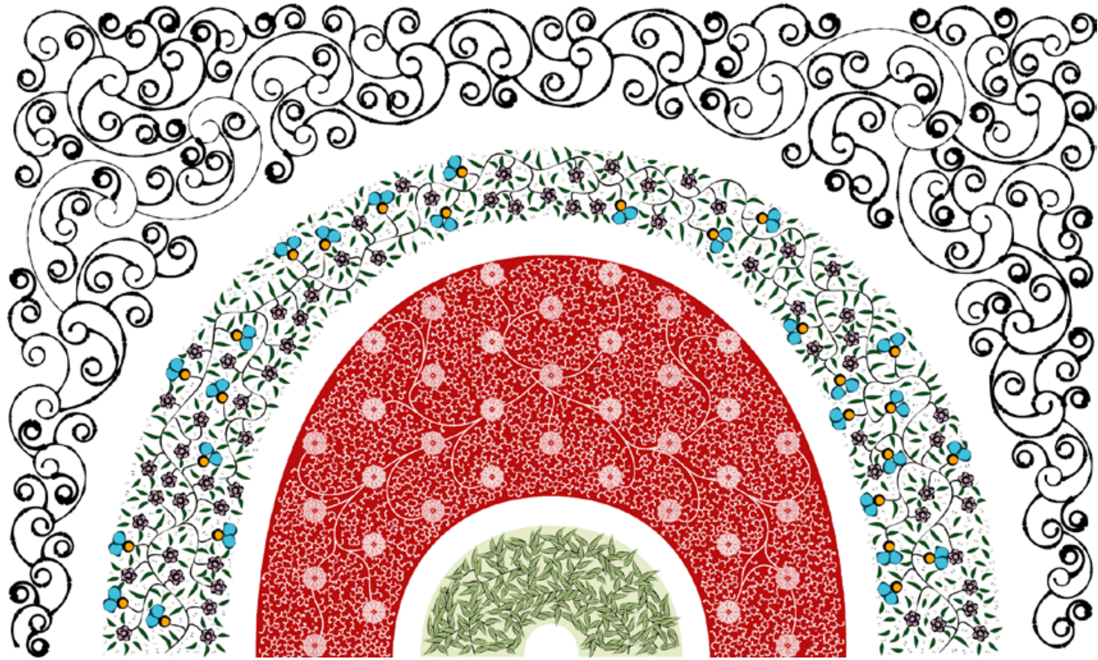


Figure 10: Procedurally generated ornamental patterns *(from Siggraph'98 paper by Wong et al.).*

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 11.

The pattern in Figure 11 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.

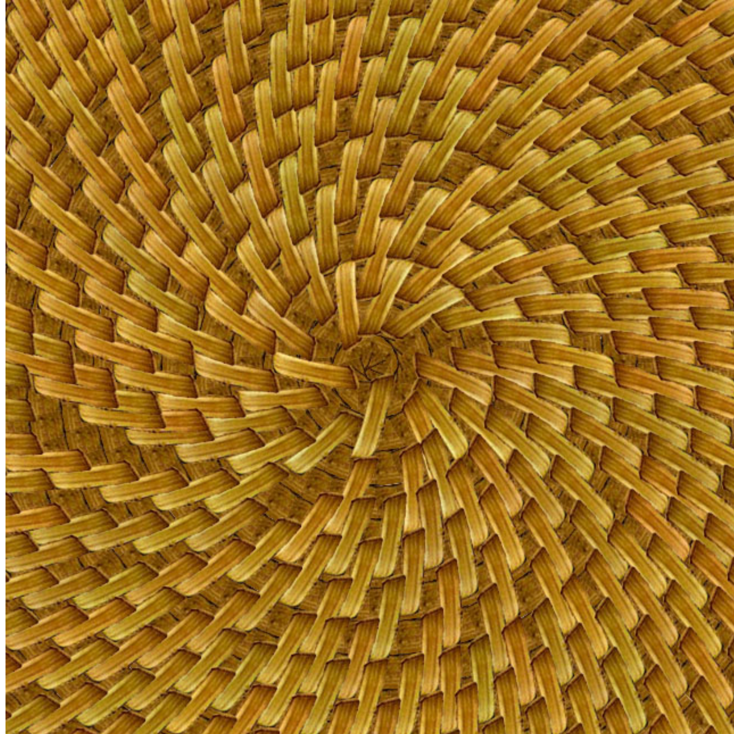


Figure 11: A more complicated pattern defined by a custom script (not shipped with CS6).

8 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 scriptals.
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.

9 Additional Resources

Paper: Měch, R. and Miller, G. The Deco Framework for Interactive Procedural Modeling. Journal of Computer Graphics Techniques (JCGT), 1(1):43—99 (Dec 28, 2012),

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, it 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 suing 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)
rotate (deg), rotate(deg, x,y,x), rotate(deg, vector)
translate(x,y) translate(x,y,z)
translateRel(x,y) translate(x,y,z])
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 uinform, same in *x* and *y*. The method **rotate** rotates the subsequently placed pattern by the given angle in degress. Optionally, you can specify the vector around which the rotation occurs (it is (0,0,1) by default). If you want to rotate around a point, you need to translate by –point, rotate and translate by +point. Note that a pattern without any rotation is pointing up (along positive y axis).

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 3.2.6 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 5.8.2).

D.1 Drawing Methods of RenderAPI

In Photoshop CC, the following *RenderAPI* methods can be used for drawing:

```
Circle (), Circle (radius), Circle (frame|point, radius)
Point (frame|point), Point (x,y)
Polygon (array_of_points)
Line (), Line (frame), Line (frame1, frame2),
```

```
  Line (point1, point2), Line (x1, y1, x2, y2)
Arc (radius, angle)
Bezier (frame1, frame2 [, mint, maxt]),
   Bezier (pt1, pt2, pt3, pt4 [, mint, maxt])

genCylinder (control_point1, control_point2)

defineInstance(id), endInstance(), drawInstance(id),
    instanceExists(id), deleteInstance(id)

translate(x,y,z|vector)
scale(x,y,z|vector)

setSceneBBox(minx, maxx, miny, maxy [,minz , maxz])
Color(kStrokeColor|kFillColor, red, green, blue [, alpha])
lineWidth(width)
setLight(index, lightObject)
setMaterial(face, materialObject)
```

The methods for rendering geometric primitives are selfexplanatory. By default a primitive is rendered at (0, 0, 0), unless a frame or a point does not specify the position. In case of lines and Beziers a frame can specify two points, one at its origin and one at Its end. In this case, the frame direction defines the tangents at the control points.

The starting point of an arc is point (0,0) and the center is point (radius, 0). This way it is easier to connect arcs in a branching structure.

The method *genCylinder* renders a generalized cylinder specified by two control points, *GenCylPoint* objects (see more information in Appendix H).

When the objects rendered are complex it is desirable to store them in an instance so that the repeated rendering is faster. A new instance is created by calling method *defineInstance(id)*, where the id is a sequence of strings and numbers (the first parameter should be a string). Any RenderAPI command afterwards is stored in the instance, until a method *endInstance()* is called. You can draw an instance using a method *drawInstance(id)*.

The method setSceneBBox is used to set the boundaries of the scene. By default it is from 0 to the pixel width and height but you can change it to your values. If you use 3D primitives, you also have to specify the range for z coordinates. The primitives will be clipped at the limits of the range. If zmin and zmax have oppositite sign an orthographics projection is set. If they have both the same sign, a perspective projection is set, with the near plane at zmin and the x and y range specifying the size of the vew rectangle at the near plane.

D.2 Lighting of 3D Primitives

By default the rendering is not shaded and each primitive has only a stroke and fill color associated with it. The color is specified by the method *Color*.

You can switch to shaded mode by setting a light. A new light is created by creating a new *Light* object. The parameters of the light are set by command:

```
setValue (valueType, r/x, g/y, b/z [, a/w])
```

where *valueType* is one of:

- *kLightColor*
- *kLightPosition*
- *kLightSpotDirection*
- *kLightSpotExponent*
- *kLightSpotCutoff*
- *kLightConstantAttenuation*
- *kLightLinearAttenuation*
- *kLightQuadraticAttenuation*.

A new light has to be set using *RenderAPI.setLight(index, light)*. Lights are indexed from 0.

A *Material* object specifies the property of the primitive's surface. The parameters of the material are set by command:

```
setValue (valueType, r, g, b [, a]])
```

where *valueType* is one of:

- *kColorAmbient*
- *kColorDiffuse*
- *kColorSpecular*
- *kColorEmission*
- *kMatShininess* – only one parameter after *valueType*

When a material is set using the method *RenderAPI.setMaterial (face, material)*, the face parameter is one of: *kMatFront*, *kMatBack*, or *kMatFrontAndBack*.

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)
load (filename [, alpha_image_filename])
save (filename)
clearCache ()

getParameter (type)
setParameter (type, value)
getSubregion (left, right, top, bottom)  // only in Version 2
```

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 **load** loads the pattern from a given file. If the path does not include the leading '/' or the drive letter the path is relative to *install_dir/Presets/Deco*. Currently, only *png* and *tga* image formats are supported in object *Image*. Optionally, you can load the rgb channels from one image and provide a second image whose first channel will be used as an alpha channel.

The method **save** can be used to save the image into a file. As above, the path is either absolute or relative to *install_dir/Presets/Deco.* For saving, only *png* image format is supported.

The method **clearCache** clears the cache of rotated patterns. The whole cache is cleared, not only cache related to the current pattern. See Section 6 for more detail on the use of pattern cache.

The method **getSubregion** can be used to select a part of the input pattern in Version 2. The method returns a new object image. This method can be used to define more than one input pattern, by combining several patterns into one pattern and then retrieve them by calling *getSubregion* repeatedly.

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 6 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 6 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 3.2.5).
- *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 3.2.5).
- *kpsMaxPatternCacheSize* – the value is the maximum allowed size of the pattern cache in bytes. See Section 6 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 6 for more details on the use of the pattern cache.

- *kpsUseOpenGL* – the value 0 or 1 specifies whether the pattern is drawn using OpenGL with hardware acceleration. Such patterns are drawn much faster but there are limits of the patterns size (depending on the graphics hardware, it could be up to 8k times 8k pixels). Also, only a basic blend mode is then available.

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

kSymmetrySetFrames
```

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.

Optionally, you can specify your own symmetry by using parameter *kSymmetrytSetFrames*, followed by an array of *Frame* objects, each specifying the symmetry. Usually, the first *Frame* is an identity.

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.

G. Object DecoGeometry

DecoGeometry is an object that contains a set of primitives. The object has the following methods:

```
load (filename)
addLineStrip(frame), addLineStrip(point, point),
addLineStrip([array of points])
addBezier(frame, frame), addBezier(pt1, pt2, pt3, pt3)
addArc(radius, angle)
addCircle(), addCircle(radius), addCircle(frame|point, radius)
addPolygon([array of points])

addMesh (meshObject)

setColor(kStrokeColor|kFillColor, red, green, blue [, alpha])

setFrame(frame)
multFrame(frame)
pushFrame()
popFrame()

render(renderAPI [, send_bbox])

smoothen(numEdgeTris)

instantiate(renderAPI)
deleteInstance(renderAPI)

getValue(parameter)
```

The method *load* can be used to load the geometry from a file. Currently, it is possible to load in svg and obj files.

Methods *addLineStrip*, *addBezier*, *addArc*, *addCircle*, and *addPolygon* add primitives to the *DecoGeometry* object. The parameters are similar to those in *RenderAPI* object.

Method *addMesh* can be used to specify a mesh. To define a mesh, create a javascript object with the following properties:

- *vertices* – this property contains an array of vertices specified as a *Vector3*
- *normal* – this property contains an array of normals specified as a *Vector3*
- *faces* – this property contains an array of faces stored in an object with the following properties:
    - *vertices* – array of indices of face vertices
    - *normals* – array of indices of normal
    - *multipleTriangles* – 0 (default, if the property is not present) if we have a single polygon face, 1 if the indices specify a triangular mesh and we have *vertices.length*/3 triangles stored in a single face (this is more efficient than having one face structure per each triangle, but there needs to be one normal per vertex, so *vertices.length* = *normals.length*).

Here is an example of defining mesh of a cube:

```
cube = new DecoGeometry(new Frame3)

vertices = new Array(0)
normals = new Array(0)

vertices.push(new Vector3(-0.5, -0.5, -0.5))
vertices.push(new Vector3( 0.5, -0.5, -0.5))
vertices.push(new Vector3( 0.5, -0.5,  0.5))
vertices.push(new Vector3(-0.5, -0.5,  0.5))
vertices.push(new Vector3(-0.5,  0.5, -0.5))  // 5
vertices.push(new Vector3( 0.5,  0.5, -0.5))  // 6
vertices.push(new Vector3( 0.5,  0.5,  0.5))
vertices.push(new Vector3(-0.5,  0.5,  0.5))

normals.push(new Vector3(0,-1,0))
normals.push(new Vector3(0,1,0))
normals.push(new Vector3(-1,0,0))
normals.push(new Vector3(1,0,0))
normals.push(new Vector3(0,0,-1))
normals.push(new Vector3(0,0,1))


faces = new Array(0)

faces.push({ vertices: [4,3,2,1], normals: [1,1,1,1]}) // bottom
faces.push({ vertices: [5,6,7,8], normals: [2,2,2,2]}) // top
faces.push({ vertices: [1,2,6,5], normals: [5,5,5,5]}) // front
faces.push({ vertices: [4,3,7,8], normals: [6,6,6,6]}) // back
faces.push({ vertices: [2,3,7,6], normals: [4,4,4,4]}) // left
faces.push({ vertices: [4,1,5,8], normals: [3,3,3,3]}) // right

cube.addMesh({ vertices: vertices, normals: normals, faces:faces })
```

Methods *setFrame*, *multFrame*, *pushFrame*, and *popFrame* are used to set a frame for primitives that are added afterwards. These functions have no effect once all geometry primitives are added.

The method *render* sends stored primitives to the given renderer. The optional second parameter indicates whether we are sending only the bounding box of each primitive. The method *instantiate* instantiates the primitives for the given API so that when the render method is called repetitively only the instance is invoked. The method *deleteInstance* deletes the instance for the given *DecoGeometry*.

The method *smoothen* smoothens the meshes stored in *DecoGeometry*. Each face is split into triangles and each triangle is approximated by a smooth surface based on the vertex normal. The smooth surface is drawn using *numEdgeTris*(numEdheTris+1)/2*. If *numEdgeTriangles* is set to 1, the mesh will not appear smoothened.

The method *getValue* returns the following information for different value of *parameter*:

- *kGetGeometryLength* – if the geometry contains only lines, Beziers, arcs and other primitives that have defined length (circumference of a circle, for example), the method returns the total length of all primitives in the **DecoGeometry** object.
- *kGetPointAlongGeometry*, *kGetNormalAlongGeometry* – once you query the geometry length you can trace it by querying points and the geometry normal at those points at a desired distance from the first point.
- *kGetBoundingBox* – returns the geometry bounding box.
- *kGetNumVertices*, *kGetNumFaces* – returns the total number of vertices and faces for all meshes stored inside the geometry, respectively.
- *kGetVertex*, *kGetFace* – the first parameter is followed by a second one, specifying the vertex or face index. Face is returned as an object with the following properties:
  - *vertices* – array of indices of face vertices
  - *normals* – array of indices of normal
  - *multipleTriangles* – 0 if we have a single polygon face, 1 if the indices specify a triangular mesh and we have *pts.length*/3 triangles stored in a single face (this is more efficient than having one face structure per each triangle).

H. Generalized Cylinders

Generalized cylinders are defined by a sequence of control points, represented by *GenCylPoint* objects. Each *GenCylPoint* object consists of a frame, a contour (a cross-section curve), and a set of profile curves.

The frames of two consecutive *GenCylPoints* points *P1* and *P2* define a Bezier curve that forms an axis of the generalized cylinder, along which the contour curve is swept. The Bezier curve is defined by points (0,0,0) and (0,1,0) in frame coordinates of *P1* and points (0,0,0) and (0,-1,0) in frame coordinates of *P2*.

H.1. Cross Section Curve

The cross section curve, a contour, is defined as a set of Bezier curves, line segments, and arcs. It is stored in an object *Curve*. The center of the cross section is at (0,0,0) and it is defined in plane $z = 0$. An object *Curve* can consist of several primitives, added one by one.

The object *Curve* has the following methods:

```
loadCurve (filename)
addLine (frame), addLine(point, point),
addBezier(frame, frame), addBezier(pt1, pt2, pt3, pt3)
addArc(radius, angle)


setNumSegments(num), setNumSegments(index,num)
```

The method *loadCurve* can be used to load the curve from a file. Currently, it is possible to load in svg files.

Methods *addLine*, *addBezier*, *addArc*, *addCircle*, and *addPolygon* add primitives to the *Curve* object. The parameters are similar to those in *RenderAPI* object.

Method *setNumSegments* specifies into how many straight line segment will the curve be split for rendering. One parameter specifies the number of segments for the whole curve. When the curve has sharp features, you may want to specify the number of segments per each added primitive, using the index of the primitive (in the order they were added to the curve) and the number of segments for that primitive (see example in the section *Rendering* below).

Here is an example of a simple contour curve:

```
var frame1 = new Frame2d()
frame1.setPosition (-0.5, 0.0)
var frame2 = new Frame2d()
frame2.setPosition (0.5, 0.0)
var contour = new Curve()
contour.addBezier (frame1, frame2)
```

A contour curve is added to a *GenCylPoint* object using its method *setContour*. If cross section curves at two subsequent control points of a generalized cylinder difier they are interpolated along the generalized cylinder.

H.2. Profile Curves

As the cross section curve is swept along the axis of a generalized cylinder a set of profile curves can adjust the width of the cross section. A profile curve is defined along the y axis. It starts at *(start radius, 0, 0)* and ends at *(end radius, y, 0)*. The curve is stretched in y axis to fit the length of the axis of the generalized cylinder between the two control points *P1* and *P2*. A profile curve is added to a *GenCylPoint* object using the method *addProfile*.

When more than one profile curve is specified, each curve has to be given a value between 0 and 1, indicating the normalized distance along the cross section curve. The value 0 is at the beginning of the cross section curve and the value 1 at the end. The distance is specified as the second parameter of the *addProfile* method. Profile curves along a cross section are interpolated.

H.3. Rendering

For rendering purposes contour curves and profile curves need to be divided into a number of straight segments. The segments are distributed evenly along the curve (even if it consists of several primitives - arcs, line segments, or Bezier curves) so that their lengths are the same.

In case you want to preserve sharp features or when you do not want to further tessellate line segments on the curve, you can specify the number of segments for each primitive on the curve, indexed in the order they were specified:

```
profile.addBezier(fr1, fr2)
profile.addLine (fr3)
profile.setNumSegments (0,8) // index 0 – Bezier
profile.setNumSegments (1,1) // index 1 - line
```

When a generalized cylinder is tessellated, contours at points *P1* and *P2* have to be split into the same number of segments. If not, the second contour is re-tessellated. Similarly, all profile curves of a *GenCylPoint* have to have the same number of segments.

Once contour and profile curves are tessellated the generalized cylinder is rendered as a set of *N* triangle strips, where *N* is the number of segments along each profile curve. The frames at *P1* and *P2* are interpolated, as are coordinates of corresponding contour points at point *P1* and *P2*. Along the strip, the values between profile curves (if there is more than one) are also interpolated, based on the distance along the contour curve.