Journal of Computer Graphics Techniques Vol. 1, No. 1
http://jcgt.org 2012

The Deco Framework for Interactive Procedural Modeling

Radomír Měch and Gavin Miller
Adobe



Figure 1. We created these complex patterns with a few strokes of Deco's flower and tree brushes.

Abstract

In this paper we introduce *Deco*, a powerful framework for procedural design that shipped in Adobe Flash Pro CS4 through CS6 and in Adobe Photoshop CS6. The Deco framework generates complex patterns from a small number of input parameters and models encoded as JavaScript objects, all stored in text files called *scriptals*. The object's methods define local growth and behavior or the rendering of the pattern. A collection of libraries simulate global interactions between the resulting structure and the environment. In addition, an artist can interactively control both the procedural growth and the resulting pattern or structure.

1. Introduction

Current, more powerful, computers enable the processing and visualizing of increasingly complex data. However, for artistic applications, the process of explicitly creating all of the detail in such data by hand in a modeling or

drawing package can be tedious and expensive. Consider the task of an artist modeling and placing each object in an outdoor scene with millions of grass blades and thousands of trees and their leaves, or the task of painting individual bricks, rust, cracks, and vines for a texture map of an old building. The challenge of modeling such detailed scenes are amplified when an animated sequence is required.

Procedural generation of models is a way of leveraging increasingly powerful computation to empower the content author as well as the content consumer. It is well known that complex structures and behaviors can emerge from repetitive application of a small set of rules that control the local behavior of the structure [Prusinkiewicz and Lindenmayer 1990]. This approach has previously been applied to procedural modeling and design of natural phenomena. Common examples include natural phenomena, such as grasses, trees, clouds, and smoke, as well as artificial structures, such as roads and cities, texture patterns, and ornaments.

We created Deco, a modeling framework that allows the user to create rich procedural geometry and included it in the Adobe Flash Pro authoring tools and in Adobe Photoshop to facilitate design of such complex structures. Deco supports selectable levels of input and interaction, because the artist can either let the modeled structure or pattern grow automatically, or he can guide the growth. Deco is made up of a procedural engine and a set of dynamically loaded environmental libraries. The procedural engine creates a structure or a pattern by repeatedly invoking procedural rules that are defined in a text file called a *scriptal* that contains a set of JavaScript objects.

The methods of the objects in the scriptal control the design and behavior of a pattern or a structure at the local level. To simulate global interaction of the structure and the environment, in which the structure grows, the script methods can invoke methods implemented in environmental libraries. Environmental libraries perform global tasks such as determining collisions between parts of the structure or between the structure and user-defined boundaries; they also can provide the procedural engine with the user input. The framework contains a set of predefined libraries, and it is easy to create new ones. The use of scripts and dynamic libraries makes it possible to create a wide range of procedural models.

The technical contributions and advantages of Deco described in this article are:

- Deco provides a general and extensible procedural modeling framework based on a widely-used programming language (JavaScript plus dynamic libraries).
- The interoperability of JavaScript and C++ allows us to expose utility C++ classes as objects in the scripts, making it easy for others to use the framework for performing complex modeling tasks or to define complex geometric primitives, for example, generalized cylinders.
- Deco provides several different schemes for a user to interactively control the pattern produced by a procedural model.
- Deco provides a general approach to expressing structural topology by using labeled ordered connections.

2. Background

2.1. Survey of Procedural Systems

Because the goal of Deco is to generalize and provide user control over previous procedural modeling frameworks, we briefly survey the history of such models and frameworks to provide context.

Procedural modeling can be traced back to the cellular automata introduced by von Neumann [1966]. Cellular automata are dynamic systems operating on a regular grid of cells. The state of a cell is updated synchronously according to a set of rules defining the new state based on the state of the cell and its neighbors. Toffoli and Margulos [1987] later introduced a Cellular Automata Machine, a framework with a special-purpose high-level programming language. Fleisher and Barr [1994] proposed a modeling framework for the simulation of growth of multi-cellular organisms in a continuous medium. The simulation testbed consists of a set of discrete cells and C-like functions describing a continuous environment. Hart [1994] proposed an alternative framework. He defined procedural geometric instancing as an extension of traditional instancing in which each instance was associated with a procedure to be executed on instantiation. The procedure could alter the object's parameters and transformations.

Lindenmayer [1968] introduced a powerful formalism for structural models. A Lindenmayer system (also commonly known as an L-system) consists of an alphabet of symbols, an initial string, and a set of rewriting rules. Each symbol represents a part of the modeled structure and the rules describe the local development of each part over time. Lindenmayer's original systems captured only the topology of the modeled structure. Prusinkiewicz [1986] proposed a graphical interpretation of L-system strings based on the "turtle" drawing object in the LOGO programming language. His system creates geometry by parsing a program string containing relative commands such as forward, backward, rotate, etc. and ones for creating geometric primitives. Prusinkiewicz and colleagues extended L-systems in many other ways, yielding a framework that they have used for modeling not only plants [Prusinkiewicz and Lindenmayer 1990], [Prusinkiewicz and Rolland-Lagan 2006], but also fractals, subdivision curves [Prusinkiewicz et al. 2003], and cities [Parish and Müller 2001].

Despite its expressive power, the L-system formalism has its limitations, and other procedural frameworks were developed later to address them. Wong et al. [1998] developed a custom-rule language for modeling floral ornaments with a specific communication with the environment, in which the ornament grows. Smith et al. [2003] introduced vertex-vertex systems, a framework in which a C-like code defines rules operating on vertices of a connected mesh; vv-systems were used to implement various subdivision schemes. Aliaga et al. [2007] defined procedural rules to generate floors in virtual buildings. Müller et al. [2006] introduced CGA shape, a grammar for modeling buildings and later used a simple recursive grammar for facade generation [Müller et al. 2007].

All of these frameworks were designed to address specific modeling domains, and therefore lack a lot of generality that L-systems provide. The Deco framework overcomes many such limitations, and we show that it is sufficiently general to provide results like those in the previous frameworks in many cases.

Procedural models often do not allow the user much control, aside from defining the initial parameters. Due to unexpected properties of procedural models, it is hard to predict the final shape of the structure from the initial parameters. There are, however, techniques, allowing the modeler to specify the global shape of the modeled structure, either by editing growth curves [Lintermann and Deussen 1999; Prusinkiewicz et al. 2001; Reeves and Blau 1985; Weber and Penn 1995] or by sketching [Anastacio et al. 2009; Boudon

et al. 2003; Chen et al. 2008; Okabe et al. 2006; Wither et al. 2009]; the model parameters determined by these methods are then fixed during the model growth.

Ijiri et al. [2006] present sketched L-systems, in which the user can sketch the shape of the main branch while the model is running. Palubicky et al. [2009] extend this concept further and allow the user to control the growth of the model using a brush that distributes particles near the brush. The model grows by consuming the particles thus, growing only in the areas specified by the brush. Lipp et al. [2008] allow the user to modify the grammar used to model a building or a facade by directly manipulating the resulting structure. The user interaction with Deco models is most closely related to the work by Palubicki et al. [2009].

3. Design Goals

In this section, we present an overview of the detailed system design goals and decisions used in creating Deco. Our goal was to develop a modeling framework that allows the user to create rich procedural geometry with various levels of input. We wanted a framework in which we would create procedural tools that users of creative design application can utilize to automate tedious parts of the design.

First, we wanted our framework to be more general than existing frameworks. For example, we did not want the topology of the model to be limited to branching structures, as in the case of L-systems or to be constrained to shapes, as in the case of shape grammars.

Second, we wanted to provide the user with a variety of ways, in which he can interact with the procedural model. Most users will use the tools as is, with their predefined parameters, and use the mouse to position the model or brush parts of the model. Many, however will modify the model parameters using sliders in the menu. A smaller group of users will download new tools from the web, and the most advanced users and developers will author new tools.

In addition, we wanted our framework to be able to connect to various creative design applications and not be limited by the type of user interactions, data types, and types of geometry or images the particular application uses.

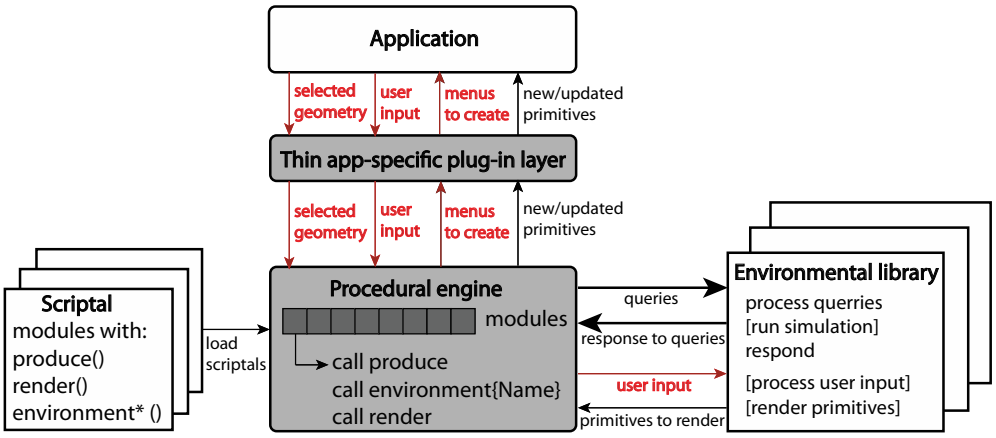


Figure 2. Schematic model of Deco framework.

3.1. Key Design Decisions

All existing frameworks are somewhat limited by the type of models that they target. Yet, we felt that L-systems have a rich formal base, and that it would be possible to build upon some crucial concepts we learned when developing an L-system framework in the past [Měch and Prusinkiewicz 1996].

The Deco framework consists of a procedural engine, a set of dynamically loaded environmental libraries, and a plug-in layer that connects the framework to a given application (Figure 2).

One important aspect of a modeling framework is the language used for defining the model. Usually, it is a special-purpose scripting language that can be interpreted during the execution of the algorithm. The use of a scripting language is crucial, since we do not want to hard-code a set of default behaviors that the user cannot change. The language used in L-systems is very compact because parts of the modeled structure are represented by letters with a set of parameters, making the rules very concise. Over time, though, additional extensions allowed C code to be included in the rules, making the rules less compact. In addition, the predefined turtle interpretation in L-systems is also limiting since the user cannot change the graphical representation of a module, a part of the modeled structure. Karwowski et al. [2003] took a different approach. They introduced L+C systems, in which they define the procedural rules in C and then compile the code into a library. The library is dynamically loaded by the modeling framework. The advantage of this approach over an interpreted language is speed.

Our approach is similar to the one by Wong et al. [1998] and also to L+C systems. The advantage of the Deco framework is that it uses a programming language based on JavaScript. Since the model is expressed in a programming language, the writer of the model has control over the rendering of the objects.

Similarly to Open L-systems [Měch and Prusinkiewicz 1996], we use external libraries written in C or C++ to simulate processes happening around the growing structure. Not only is it a nice conceptual differentiation between the model and the space it grows in, but it also can be used to define parts of the simulation that would be tedious to write in the script, or is common for many models, for example, collision detection (see Section 4.2).

To limit the dependency of our framework on the application in which it is used, we incorporated several constructs into the framework. For user interaction, we convert the mouse and key events to the local data types, we define menus in a simple linear fashion, and we collect the changes to the menu in the application-specific layer (Section 8.1). The framework can access the elements defined by the application. The elements that are to be used by the framework are encapsulated in the *Geometry* object (Section 8.2) or in the *Image* object and the framework can manipulate them (move them, scale them, or rotate them) without having to know their internal representation in the application. Rendering of the model is done by streaming drawing commands to the plug-in layer, which then decides whether the geometry is drawn directly to the screen or whether it is stored in the applications data structures, for example, in a scene in Flash Pro.

4. System Details

4.1. Procedural Engine

The core of the Deco framework is a procedural engine that loads the model description, runs the simulation, distributes the user input and geometry coming from the application, facilitates the communication with environmental libraries, and finally, sends the resulting geometry to the application.

4.1.1. Scriptals

The procedural engine first loads a *scriptal*, a JavaScript file containing the definition of the procedural model. A scriptal contains:

- a set of JavaScript objects, each representing a module, a part of the modeled structure. The object's methods specify the behavior of each module during the simulation and rendering.
- commands setting up the initial state of the model (initial module or modules and their parameters);
- commands loading environmental libraries;
- commands specifying parameters of the simulation or rendering;
- commands defining the model's menu structure that appears in the application.

The procedural engine exposes various C++ classes as JavaScript objects; we call them system objects. These objects facilitate the communication with the procedural engine or they can be used for various modeling tasks (Section 5). When the scriptal is loaded, it is executed (interpreted). First, the definitions of model objects are loaded into the JavaScript execution environment. Then, instances of a selected module or modules are created and sent to the procedural engine using the method addModule of the system object Engine (see Section 4.4). These modules form the initial state of the simulation. The system object Engine can be also used to set various simulation parameters.

4.1.2. Simulation Loop

The initial modules sent to the procedural engine by executing the scriptal are stored in a linked list. The engine then runs the simulation by repeatedly performing produce, environment, and render passes over the modules stored in the list. During each pass, the methods produce, environment, or render, respectively, defined on each module in the scriptal is executed. Figure 3 illustrates the flow of control in the Deco framework.

The produce method controls the behavior of the module during the simulation. It can update the module's parameter or create new modules. New modules are sent to the procedural engine using the Engine command addModule.

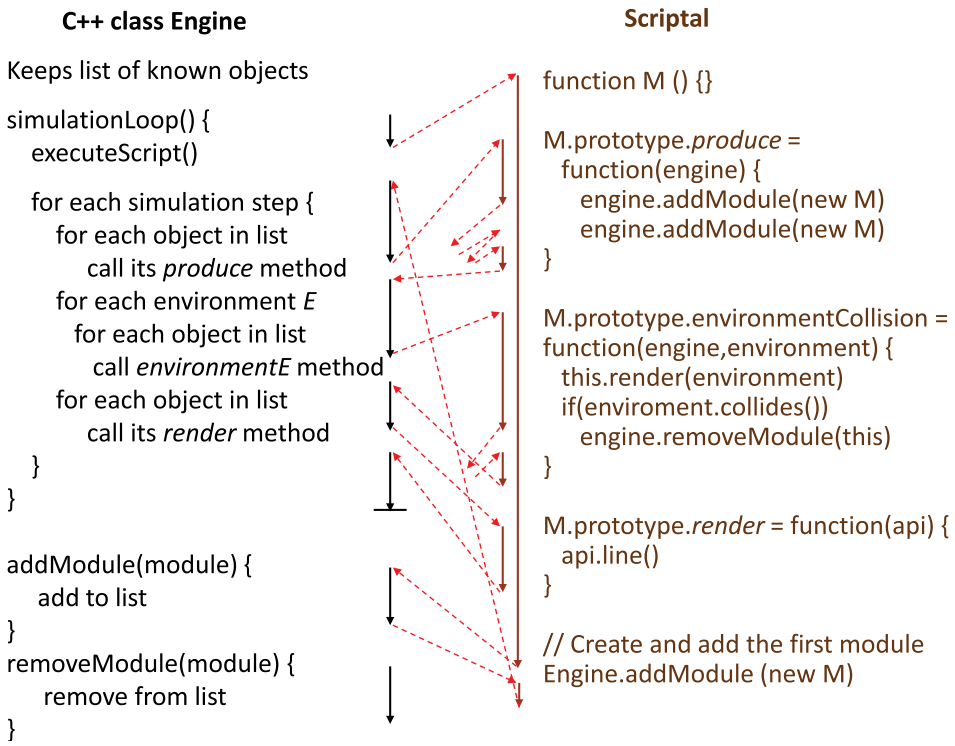


Figure 3. Example of the flow of control in the Deco framework during a simulation.

The environment method can send information about the module to the environment and collects the response. For example, if a Collision environment is loaded to the procedural engine during the scriptal execution, the engine calls a method environmentCollision on each module (if it exists). This method can send the geometry representing the module to the environment. It gets back information if that geometry collides with any other module. If it does the module can be removed.

The render method defines the geometry that is used to visualize the model. It receives a RenderAPI object as a parameter and it can call its methods to send primitives representing the module to the application (see more details in Section 7). Note that during the render pass the environment's render method is called as well, thus the environments themselves can add geometry to the result of the simulation.

4.2. Environmental Libraries

Conceptually, processes simulated by any procedural framework can be divided into internal and external processes. Internal processes control the development of individual modules locally, and external processes control how the structure interacts with the surrounding environment at the global level. Many procedural frameworks reviewed in Section 2.1 recognize the importance of the environment in the model's growth. An important extension to the L-system formalism are Open L-systems introduced by Měch et al. [1996] in order to simulate plants interacting with their environment. Wong et al. [1998] in their framework for modeling floral ornaments use the environment to search for the biggest empty circle into which new structures can grow. Müller et al. [2006] use the environment to determine visibility of facades and to snap tiles to common lines.

In the Deco framework, internal processes can be simulated inside the produce method. The information about neighbors can be obtained using the *Topology* system object (Section 5.1). External processes are captured by environmental libraries that are specific to a given task to be simulated. These tasks can be simple, for example, to determine collisions between parts of the growing structure. They can also be complicated, for example, to simulate real physical properties.

An environmental library is a C++ library that is loaded dynamically as needed using an Engine command addEnvironment. The communication with the environment is facilitated using predefined methods send and query or using additional methods that the environment allows. (see Section 6 for more information). To inform the environment about the module's shape, the user calls the module's render method with the environment object as a parameter. This is possible because the class defining an environment is a subclass of the RenderAPI class (see Appendix D). Thus, the mechanism of the RenderAPI class for passing the geometry to the application can be also used to pass the geometry to the environment.

Although the communication with the environment can be done in the produce or render methods, there is a convenience method environmentName, called for each module by the procedural engine, where Name is the name of the environment.

4.3. Plug-in Layer

The plug-in layer acts as the interface between the procedural engine and the application that is using it. The reason for having a plug-in layer is that we wanted Deco to be independent of any application that is providing the user input and that visualizes the results.

The role of the plug-in layer is to collect the user input in the form of mouse events, key strokes, and user-selected geometry from the application and convert it into the representation needed by the Deco framework (Figure 2). In addition, the application queries the plug-in layer about the items that should appear in a menu that is specific to each script, it builds the menu, and when any item in the menu changes, it informs the plug-in layer about it (Section 8.1).

Conversely, the plug-in layer takes care of converting the geometry being created by the procedural engine in the render pass into the format understood by the application. There are two main modes of rendering. At present, we have connected Deco to a simple 3D modeling application based on wxWidgets, to a Flash Authoring application, Version CS4, and to Photoshop, Version CS6. The 3D application uses OpenGL rendering that needs continuous updates, and the render pass needs to be performed every time the window is refreshed. In Flash Authoring, on the other hand, the rendering pass adds geometry into the Flash scene, and it is desirable to call the rendering pass only on the geometry that has changed to avoid frequent updates of the scene. See Section 7.3 for more information about incremental rendering. In Photoshop, the rendering is done by placing scaled and rotated patterns (images) into a scratch buffer, which is then blended with the current layer.

4.4. Example of a Deco Scriptal

The full listing of a scriptal as outlined in Figure 3 is given in Listing 1. The scriptal generates a sequence of tree-like structures shown in Figure 4.

The first method initializes the module ModuleLine. A single parameter frame is stored with the module. The parameter frame is of type Frame2, which is a Deco system object (Appendix B); frame specifies a transformation applied to the module.

The method produce creates two new modules, places them at the end of the existing module, one to the left and one to the right. The produce method returns the constant kDontCallAgain; thus it will not be called again.

The method render draws a single line, which by default goes from the

```
function ModuleLine(frame) {
    this.frame = frame
}

ModuleLine.prototype.produce = function (engine) {
    // branch 1
    var newframe1 = new Frame2 (this.frame)
    newframe1.advance (1.0)
    newframe1.rotateDeg (-25)
    engine.addModule (new ModuleLine (newframe1))

    // branch 2
    var newframe2 = new Frame2 (this.frame)
    newframe2.advance (1.0)
    newframe2.rotateDeg (35)
    engine.addModule (new ModuleLine (newframe2))

    return kDontCallAgain
}

ModuleLine.prototype.render = function (renderapi) {
    renderapi.Line ()
}

ModuleLine.prototype.environmentCollision =
   function (engine, environment) {
    this.render (environment)
    if (environment.collides ())
        engine.removeModule (this)
    return kDontCallAgain
}

// Initialization
var initialModule1 = new ModuleLine (new Frame2())
Engine.addModule (initialModule1)
Engine.addEnvironment ("Collision")
Engine.setSceneBBox (-5, 5, 0, 10)
```

Listing 1. Listing of the scriptal used in Figure 4

point $(0,0)$ to the point $(0,1)$ in the local coordinate frame specified by the parameter frame. By default, the frame is applied automatically before executing the method render.

The method environmentCollision draws the line to the environment and tests for a collision. If the line collides with another object already stored in the collision environment, the current module is removed.

The following sections provide more details on the system objects used in a scriptal.

5. Predefined Objects

The Deco framework contains a set of predefined objects that can be used by the scriptal author to communicate with the procedural engine or to perform various tasks that are often needed in a simulation:

- Engine: facilitates communication with the procedural engine;
- RenderAPI: sends geometry representing modules to the application for rendering;
- Vector2, Vector3, and Vector4: vector object with overloaded operators;
- Frame2 and Frame3: two- and three-dimensional frame, represented, respectively, by a 3×3 or 4×4 transformation matrix;
- Geometry: a container for primitives;
- Image: a container for images;
- Topology: stores connections between modules;
- Symmetry: used to create various topologies.

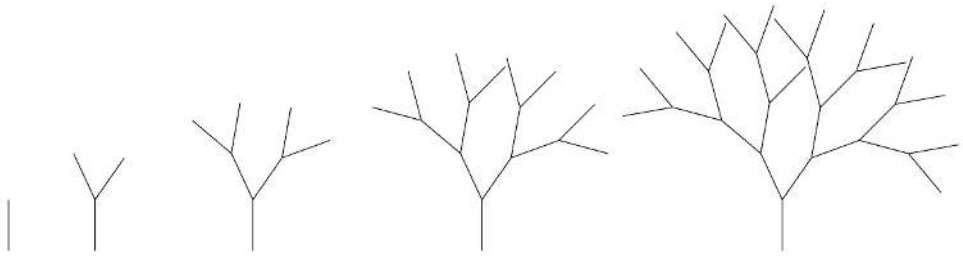


Figure 4. Developmental sequence of a simple branching structure created by the scriptal from Section 4.4.

The objects Topology and Symmetry are conceptually important; they are described in this section. The other objects are included in the Appendices.

In addition, the users can load user objects from a dynamic library. Currently, there are three user objects:

- MeshLoader: loads object from an obj file into a mesh;
- Stroke: creates variable width strokes;
- RenderToImage: renders objects into an off-screen buffer.

However, users can develop their own objects, similar to how they can create environmental libraries.

5.1. Topology

Since the operations in procedural models are defined locally, it is important to define a topology for the modules. In L-systems, for example, the topology is given by the order of the modules in the L-system string. Branching structures can be created by using special symbols "[" and "]". It is assumed that a right neighbor of a module before [can be either the symbol just after [or just after the matching]. In vv-systems (Section 9.1.2), the modules represent vertices of a mesh, and each vertex stores an ordered list of connections to its neighbors.

In both cases, the topology is limited. In the Deco framework, the connections between modules can form an arbitrary graph. Each connection can be labeled by an integer or a string label. Several connections can share a label, in which case they are ordered in a cyclical array.

The idea of labeling connections and ordering of connections of the same label combines approaches of group-based fields, described by Giavitto and Michel [2001] and the vv programming language, introduced by Smith et al. [2003]. In a group-based field, each connection to another module is labeled. On the other hand, in the vv language, connections are not labeled but they are ordered in a cyclical array. Our approach can be viewed as a set of labeled vv connections.

5.1.1. Adding and Removing Connections

The connections are stored in a global object Topology that provides basic functions for setting and accessing connections. A connection with a label "label" between modules m1 and m2 is added using the following method:

```
Topology.add (m1, label, m2)
```

where label is an integer or a string. Connections of the same label are ordered, and the new connection is added to the end. It is possible to add a new connection to a specific position among existing connections using one of the following methods: addFirst, addLast, addBefore, or addAfter.

We can remove a single connection, all connections of a given label, or all connections of a module using the following method:

```
Topology.remove (m1 [,label [,m2]])
```

To be able to replace a connection without modifying the order we can call:

```
Topology.replace (m1, label, oldModule, newModule)
```

5.1.2. Querying Connections

To get a module connected to a given module, we can use any of the methods Topology.get, getFirst, getLast, getPrevious, or getNext. The parameters are the same as the corresponding add methods.

We can test if two modules are connected using:

```
Topology.isConnected (m1, label, m2).
```

5.1.3. Delayed Updates

Sometimes it may be desirable not to modify the current topology until the produce method is called for all modules. For this purpose, the user can set the kDelayedUpdates parameter using the method Topology.setParameter. If the parameter is set to 1, all changes to connections are stored in separate data structures and the changes are applied after all modules are processed in a produce pass. In addition, the removal of modules is also delayed.

Sometimes, we need to modify the parameters of a module while keeping the old version around. For example, when subdividing a mesh, as we are building the connections between new vertices we still need to query the old

connectivity while processing the existing vertices. We can create a new module and move the connections from the existing module using the following method:

```
Topology.move (moduleTo, moduleFrom)
```

When a connection is queried, the current connections are returned. To query the topology as it was at the beginning of the simulation step, the user can add the word Old to all get methods, such as getOld, getFirstOld, etc. and to the isConnected method. See the example in Appendix H.

5.2. Symmetry

The Deco framework supports various symmetries that can be applied globally to all modules or locally to selected modules. Symmetries can also be nested.

The framework defines a system object *Symmetry*. The object can be instanced using new Symmetry and placed among modules specifying the structure using Engine.addModule. Each instanced Symmetry object stores a list of matrices to be applied to the rendered geometry. The matrices are created automatically according to the type of symmetry, which is specified using the following method:

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

There are different types of symmetries, and the parameters specify a line symmetry, a point symmetry, a rotational symmetry, etc. Deco also supports a variety of tiling symmetries (see an example in Figure 20), in which case the number of matrices could be infinite. Thus the Symmetry object also receives the information about the size of the window and the bounding box of the modules affected by the symmetry so that a proper subset of the tiling matrices can be computed.

Modules that are affected by the symmetry are connected to the instanced Symmetry object using the predefined object SymmetryTopology. The connections are marked using the label kSymmetryLabel; they are bi-directional, that is, the symmetry module is connected back to all modules it affects.

The produce method of the module Symmetry is empty by default. The render method (defined in the procedural engine) loops over all matrices and, for each matrix it calls the render method for all modules that are connected to the module Symmetry using SymmetryTopology.

As the structure grows, every new module produced by an existing module should be added to the proper symmetry set. For this purpose a topology object

and a label can be registered with the Engine object. As a new module is added from within the produce method of another module (a parent module), the new module is connected to the same module as its parent (see kUpdateTopology in Appendix A).

The mechanism for automatic updates of connections makes it possible to add symmetries to existing models or parts of a model with only a minimal modification, as shown in the following example:

```
var symmetry = new Symmetry
symmetry.set(kSymmetryRotation, 4)
Engine.addModule (symmetry)
Engine.addModule (ourModule)
symmetry.addModule (ourModule)
```

Before adding a module ourModule to the procedural engine, we define a symmetry module and add it to the engine. Then, we add the module to the symmetry. If the module is the top-most, the symmetry is applied globally.

There are two ways of handling a response to environmental effects, such as a detected collision. The environment should be queried from the module's render method using direct communication (see Section 6.1). One could choose to not display the particular symmetry instance of the module by returning from the render method before the rendering is done. This would, however, visually break the symmetry of the result. To keep the symmetry intact, we instead can remove the module if any collision is detected in the render method, and all the symmetry instances will disappear.

6. Simulating Environments

As mentioned in Section 4.2, simulating the environment is an important part of a procedural modeling framework. In Deco, an environment is a C++ library that is loaded dynamically as needed and exposed in the script as an object. To load an environment, the user needs to give the name of the library to the Engine object using the following method:

```
Engine.addEnvironment ("Name")
```

Once the environment is loaded, an object EnvironmentName is created, for example, EvironmentCollisions.

6.1. Direct Communication with the Environment

There are two ways of directly querying the environment. In the first approach, a module's produce or render method can communicate with the environment using methods:

```
EnvironmentName.send (envParam1, ..., envParamN)
EnvironmentName.query (envParam1, ..., envParamN)
```

Both methods send a set of parameters to the environment. In the case of the query method, the environment sends back an array of values as the response. These values are converted into a JavaScript object. If a value in the array returned by the environment is preceded by a string, the string is used as a name of the value in the object. For example, if the environment returns an array of values: [name1, param1, name2, param2], we can access them as follows:

```
var ret = EnvironmentName.query(envParam1, envParam2)
var p1 = ret.name1
var p2 = ret.name2
```

If the environment returns only one value it can be accessed directly without indexing in the array or using names.

In addition to these default methods, each environment can define its own methods. The environment library can register methods of a given fixed format that become methods of the object EnvironmentName. For example, the scriptal in Section 4.4 is using the method collide.

Some environments do not need to know about the location and shape of each module. For those that do, it is possible to send the information about the geometry using the send method, but since the environment class is a subclass of the RenderAPI class (see Appendix D), we can also send the shape to the environment using standard rendering calls.

In the second approach to querying the environment, the Engine class invokes the method environmentName for each module, where Name is the name of the environment. This happens after the produce method has been called for all applicable modules.

6.2. Delayed Communication with the Environment

Let us look at the example from Section 4.4. If we create two new modules that collide, the simulation will always remove the one that is processed

later. We can introduce some variation by randomizing the order in which the method environmentCollision is called for modules by setting a parameter kRandomEnvironment of the Engine class (see Appendix A).

Sometimes even randomization is not sufficient, and it is necessary for the environment to collect all queries before it can respond. To support this important functionality, the environment class supports another type of query:

```
delayedQuery (envParams, object, methodName,
                methodParam1, ...)
```

In the case of a delayed query, the environment does not respond immediately; once it collects all necessary input, it calls the specified method on the given object. There are two sets of parameters, those sent only to the environment, stored in an array envParams, and those sent to the method that the environment calls (methodParams).

In our example, we can implement a delayed query as follows:

```
ModuleLine.prototype.environmentCollision =
  function (engine, environment)
{
    this.render (environment)
    environment.delayedQuery (this, this,
                              "removeModule", engine)
    return kDontCallAgain
}
ModuleLine.prototype.removeModule (engine)
{
    engine.removeModule (this)
}
```

Notice that both methods above are not module specific. Thus we can create a default commonQuery method of the environment object that is called if a module does not have its own environmentCollision method. Moreover, we can use an environment's method for removing the module.

```
EnvironmentCollision.prototype.commonQuery
                          = function (module, engine)
{
    module.render (this, 1);
        this.delayedQuery (module, this, "removeModule",
                          engine, module)
    return kDontCallAgain; // Stop calling the method
    }
EnvironmentCollision.prototype.removeModule
                                  (engine, module)
{
    engine.removeModule (module)
}
```

Using the commonQuery method is a very powerful mechanism since we can then add an environment without having to make any changes to the existing modules.

7. Rendering

Rendering of the procedural model is done by calling the engine's render method, which calls each module's render. The term rendering is used loosely in the context of the Deco Framework; by rendering we mean piping of geometry to an output, which could be a display, but also a scene graph, environmental library, or an output file. The render method uses a RenderAPI class that can be sub-classed by the plug-in layer to provide various rendering targets. The RenderAPI class is exposed in the scriptal as a base Render API object. The base object is used to send geometric primitives to a target, which can be an application's renderer or the application's scene graph (Section 7.3). An object sub-classed from the RenderAPI can be an exporter saving the simulated structure into a file, or an environmental library (Section 6).

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 line in the example of Section 4.4.

Rendering is very straightforward, but there are a few optimizations that we discuss in the following sections.

7.1. Bounding Shapes

Many environmental processes need information about the position and the shape of modules to determine collisions or proximity to obstacles. Often, it is sufficient to provide the environment with a tight bounding shape or a bounding box. We usually add a second parameter to the module's render method and, if that parameter is set to 1 by the environmentName method, we render only a bounding rectangle.

7.2. Instancing

In many models, the geometry created by certain modules does not change throughout the simulation. We can store a single graphical instance for each module and reuse it at each subsequent render call.

Some modules may need specific rendering, though, and we need a mechanism to inform the RenderAPI object about what needs to be instantiated. For this purpose, we can use the methods instantiate and endInstantiate.

The first method creates an instance specified by a collection of string and number parameters. Once the instance is created, it returns 1. The second method ends the instance. The instance is defined by all RenderAPI calls between those two methods. The render method can look as follows:

```
   Module.prototype.render = function (api, byenv) {
        // Render stem (don't instantiate)
        ...

        // Render flower (instantiate)
        if (api.instantiate ("flower"))  return 1;

        // Render the geometry
        ...
        api.endInstantiate()
}
```

7.3. Rendering by Adding Geometry into a Scene

Many applications do not provide direct access to their renderers. In addition, to be able to print the model or perform application-specific operations on the resulting geometry, it is often necessary to add the geometry to the application's scene graph. This can be easily achieved by creating an exporter class

(a subclass of the RenderAPI class) that adds geometry to the application's scene graph instead of rendering it on the screen.

In such an exporter, we need to handle the fact that the render method is called at each simulation step for each module and environmental library. We need to replace the existing geometry in the application's scene with the new one. Thus, the procedural engine needs to identify the module or the environment before calling its render method using the method setParameter(kModuleID, module/environment) of the RenderAPI. In addition, the user may decide to identify pieces of the rendered geometry individually inside the render method by providing an additional geometry id using the method setParameter(kGeometryID, id). The exporter class needs to keep a hash table of geometry handles associated with each module and geometry id. When an existing module or id is found in the table and it is the first time it appears during the rendering pass (it may appear more than once under a symmetry module), we remove the geometry associated with this module or id from the application's scene graph.

Replacing geometry at each rendering step is very inefficient when parts of the model do not change over time. For that reason, an exporter sets a parameter kIncrementalRender on the procedural engine. Once a render method is called for a module, it will not be called again, until the rendering is enabled again using the method setModuleParameter(module, kCall, kRender, 1). Thus, the author of the script needs to set this parameter when a module changes its shape or position.

We also took advantage of the incremental rendering mode in the OpenGL renderer. For the complex examples from Figures 19 and 20, we rendered the geometry incrementally into the front buffer as long as the camera did not change.

8. User Input

As mentioned in Section 1, the framework supports several levels of user input. We now describe how the user input is processed by the model and the environmental libraries.

8.1. Including the Model's Parameters in a Menu

The scriptal author not only specifies the model's behavior and rendering, but he can choose which parameters of the model are included in a menu. Each

menu is built from an array of items, where each item specifies the name of the variable to be modified, its initial value, the name of the menu item, and the range for the variable. This array is sent to the Engine object using the makeMenu method that forwards it to the plug-in layer. The plug-in layer is then responsible for creating the menu in the application and for informing the Engine class about any changes to the parameters.

Here is an example of a menu specified in one of the scripts in Flash Authoring CS4:

```
// [variable name, initial value, menu item name, range,
// unit ]
var menu1 = [ "Initial␣parameters",
 ["vcolor", Initial.vcolor, "", [0,1]],
 ["iangle", Initial.iangle, "Branch␣angle:", [-180, 180],
    "CW"],
 ["sdiv", "divider", "␣", [0,0]],
 ["fsceneScale",Initial.fsceneScale ,"Pattern␣scale:",
    [0.5, 3], "%"],
 ["sdiv", "divider", "␣", [0,0]]
]
var menu2 = [    "Module␣Line␣parameters",
 ["flength", ModuleLine.flength, "Segment␣length:",
     [0, 5], "px"],
 ["sdiv", "divider", "␣", [0,0]],
 ["iAnimatePatern", 0, "Animate␣Pattern", [0, 1]],
 ["iframeStep", ModuleLine.iframeStep, "Frame␣step:",
     [0, 100]]
]
Engine.makeMenu ("Initial", menu1, "ModuleLine", menu2)
```

The single-letter prefix in the variable name controls the type: integer, float, or vector. The resulting menu is shown in Figure 5. When a parameter is changed in the menu, an optional method Initial.variableUpdated or Module.variableUpdated is called. The parameter of the method is the name of the variable being changed. This way the scriptal author can define a response to the changes made by the user.

We can apply the same idea to environmental libraries as well. The environment API includes a method getMenu that returns the same array as the one created by the model's script (see Appendix I). When a parameter changes, the framework calls the method setParamValue with the parameter name and a new value.



Figure 5. Deco menu specified by a scriptal in Flash Authoring.

8.2. Exposing Geometry in the Menu

The scriptal author can also expose a selected geometry module to the Engine that sends it to the plug-in layer. Those objects appear in the application menu as objects that can be modified. For example, in Flash Authoring CS4 these objects can be replaced by the library symbols (see Figure 6).

When such an object is modified, the plug-in layer directly updates the geometry object or, it can call a callback method with the new geometry as a parameter.

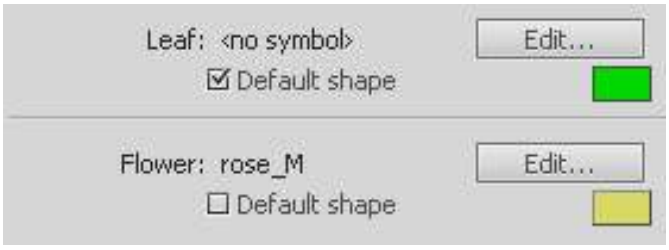


Figure 6. Part of a Deco menu in Flash Authoring exposing parts of a model that can be replaced by library symbols.

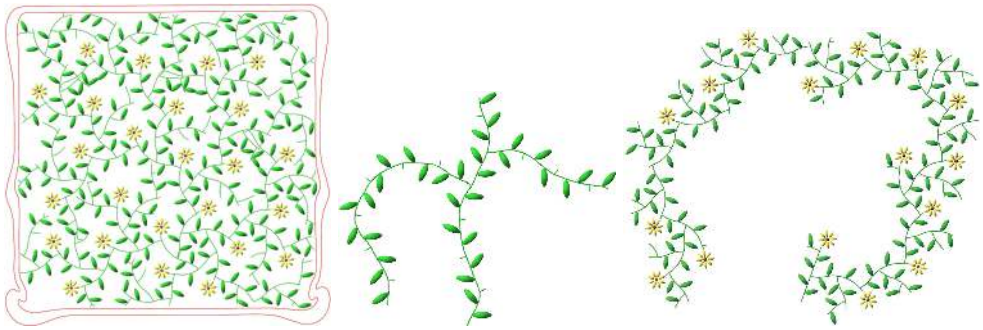


Figure 7. Different ways of controlling the Deco models. The user can click inside a shape to fill it (left), brush the stems directly (middle), or let the procedural model grow near the brush stroke (right).

8.3. Controlling the Procedural Growth

The techniques described above represent more traditional techniques of user interaction with procedural models. The user selects parameters of the model and lets the simulation run. In Deco, we added interactivity to the modeling process. We use three concepts:

1. The user can initiate the growth by clicking the mouse.
2. The user can paint parts of the model directly with other parts being added procedurally.
3. The user can paint areas, to which the procedural growth is constrained.

Figure 7 illustrates these concepts. The model on the left of Figure 7 is created by clicking inside a preloaded boundary. Once a mouse click is detected, the model creates a waving stem, terminated with a flower. As the stem grows, it creates leaves or new stems on each side. Collision detection is done by sending the lines representing the stems or polygonal outlines for leaves and flowers to the Collision environment. If a collision is detected, the model removes the colliding segment and the growth in this place is terminated[1] Thus, the model automatically fills the shape without any further user interaction.

The model in the middle of Figure 7 illustrates the use of direct brushing. It is similar to the previous example, but the stem growth is controlled by a

[1]A more elaborate model could detect a proximity of an obstacle before the collision happens and adjust the growth direction to avoid the collision.

mouse. After placing a short part of the stem, the stem module tests whether the mouse cursor is within a certain radius. If the cursor is near and if the button is down, the stem grows towards the mouse by adding one stem module. Leaves and initial branch segments are placed automatically. Each branch segment tests the mouse proximity in a manner similar to the main stem. Thus, the branches are drawn by the mouse, yet the procedural model places the leaves and initial branch segments automatically. This is a combination of direct control and some procedural growth on elements placed by the user (see also Figure 1).

The model on the right of Figure 7 shows an example where a procedural model grows only within a certain distance from the mouse position. This can be achieved by a simple modification of the initial model, where we insert a test to the produce method and grow only when the mouse is within the predefined distance and the mouse button is down. This way, the user can control the location of the procedural growth but not the position of individual branches. A similar approach of brushing the area in which a procedural model can grow was used in [Palubicki et al. 2009].

Once a structure is created, it is possible to continue to run the simulation, collect user input, and perform various editing tasks. One approach is to define a set of handles in the scriptals and detect when the user manipulates them. We used this approach in the Symmetry tool in Flash Authoring CS4 (see Figure 8). The handles can increase or decrease the number of elements in rotational or translational symmetry (the dot with the plus symbol), rotate the frame (dots with arrows) or move the group of elements (the central dot).

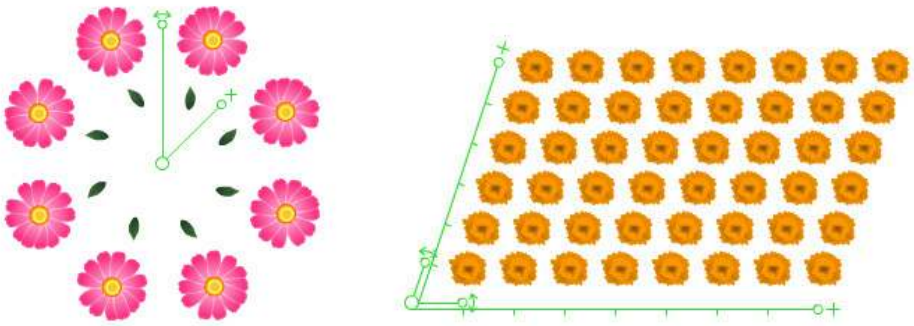


Figure 8. Symmetry model with handles drawn and controlled by the scriptal.

A different approach is used in one of the spray models we developed. In that model, the user may edit an existing spray path by selecting an edit mode in the menu and then moving the tablet pen over the path midsection. By applying varying pressure, the density of the distributed particles in that location is changed. Similarly, there can be a mode for editing the stroke width or various other parameters of the procedural model.

These modes of operation can be combined; together they provide a rich set of controls. The main advantage of the Deco framework over existing procedural frameworks is the variety of ways in which the user can control the model. This is an important aspect necessary for a smooth incorporation into many creative tools.

9. Results

Our framework can express many existing frameworks and can be used for expressing various procedural models. This section includes some examples.

9.1. Implementing Other Frameworks in Deco

In this section we show how to implement four existing frameworks in Deco.

9.1.1. L-systems

To express an L-system model in our framework, we convert each L-system symbol into a scriptal object. The set of productions for a given symbol is implemented by a single produce method.

For example, the following simple L-system:

```
Axiom: A(0)
A(n) -> [-(25) F A(n+1)] +(35) F A(n+1)
```

would be converted into the scriptal given in Listing 2.

Since by default new modules are added to the end of the linked list in the procedural engine, we need to set the parameter kKeepInPlace.

It is obvious that the L-system notation is much more compact, but there are limitations to what the rules can express, and there are assumptions being made about graphical representation of individual modules with which the user has to be familiar. A scriptal, on the other hand, is more verbose, yet the users can create more complex productions, and they have better control

```
// Modules
function ModuleMinus (a) { this.angle = a }
function ModulePlus (a) { this.angle = a }
function ModuleF () {}
function ModuleLeftBr () {}
function ModuleRightBr () {}
function ModuleA (n) { this.n = n }

// Produce methods
ModuleA.prototype.produce = function (engine) {
    engine.addModule (new ModuleLeftBr ())
    engine.addModule (new ModuleMinus (25))
    engine.addModule (new ModuleF ())
    engine.addModule (new ModuleA (this.n + 1))
    engine.addModule (new ModuleRightBr())

    engine.addModule (new ModulePlus (35))
    engine.addModule (new ModuleF ())
    engine.addModule (new ModuleA (this.n + 1))
    engine.removeModule (this);
}
// Render methods
ModuleMinus.prototype.render =
    function (api) { api.rotate (-this.angle, 0,0,1) }
ModulePlus.prototype.render =
    function (api) { api.rotate (this.angle, 0,0,1) }
ModuleLeftBr.prototype.render =
    function (api) { api.pushMatrix() }
ModuleRightBr.prototype.render =
    function (api) { api.popMatrix() }
ModuleF.prototype.render =
    function (api) { api.Line (0,0, 0,1) }

Engine.setParameter (kInsertModule, kKeepInPlace)
```

Listing 2. Scriptal version for a simple L-system.

over the rendering of modules. The scriptal in Section 4.4 expresses the same procedural model in a different way, more suitable for the Deco framework.

In context-sensitive L-systems, the selection of a production depends on the neighboring modules. In our framework, we would connect the modules using the Topology object and query the neighbors using its methods. In environment-sensitive L-systems, after productions are applied, an environ-

mental pass goes through the modules and interprets them, as it does during rendering. When special symbols are encountered, the engine either stores the current position and orientation in those modules or exchanges information with an external environmental process. We can reproduce that by calling the render method for each module from the module's enviromentName method, similar to the example from Section 4.4.

Thus, the powerful formalism of L-systems can be captured in the Deco framework.

9.1.2. Vertex-Vertex Systems

Vertex-vertex systems (or vv-systems) were introduced by Smith et al. [2003]; vv-systems operate on meshes. Each vertex stores an ordered list of neighboring vertices and operations are defined on this local neighborhood. We represent each mesh vertex as a module and store the connections to all neighbors of the vertex as Topology connections with the label "vertices." See Appendix H for the example implementation of Loop subdivision in the Deco framework.

In a vv-system, it is not possible to mark edges as normal, creases, or boundaries. In the Deco framework, as we load a file using the MeshLoader user-defined object, we create another set of connections labeled "crease" and "boundary." Then, in the scriptal, we use the method Topology.isConnected to test if an edge is a boundary or a crease and choose the subdivision rules accordingly. Figure 9 illustrates the use of the Loop subdivision scriptal, extended to consider boundaries.

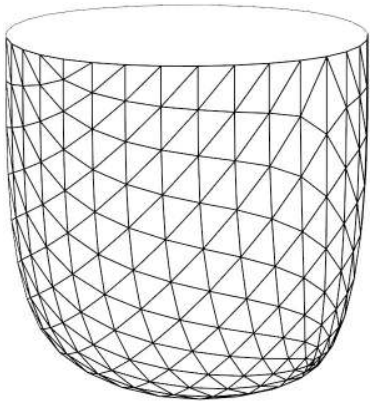


Figure 9. Example of a cube without a top subdivided using Loop subdivision.

9.1.3. Wong's Framework for Ornaments

The framework of Wong et al. [1998] finds the largest circle that does not intersect any existing modules or the boundary. Only modules located near this circle may produce new modules. This mechanism can be easily expressed in our framework.

We converted Wong et al.'s original code into an environmental library called BiggestCircle. Instead of using a produce method for each module, we converted the rules of Wong et al.'s framework into a create method, because the productions are not initiated by the procedural engine but by the environment.

To initiate growth in each simulation step, we insert a special module ModuleStart as the first module. We use a delayed query for the environment and set the callback method to the create method of the ModuleStart.

At each simulation step, the produce method of ModuleStart initiates the search for the largest circle. Once the circle is found, the environment calls the ModuleStart.create method for each module near the circle. This method then invokes the create method for the module given as the parameter of the delayed query. As new modules are created, they are sent back to the environment.

Figure 10 shows one of Wong et al.'s published models reproduced in Deco. A snippet of the script, showing the methods of the module ModuleStart is given in Listing 3.

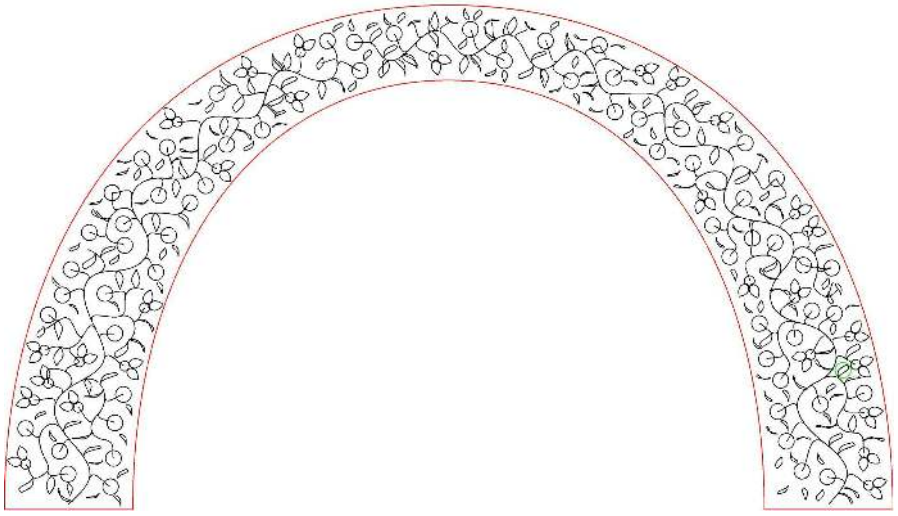


Figure 10. Sample model created by expressing Wong et. al's [1998] framework.

```
function ModuleStart() {
    EnvironmentBiggestCircle.delayedQuery(
        kBigCircleSetCreateMethod, this, "create")
}
ModuleStart.prototype.create =
                    function(module,goalC,goalRadius)
 {
    return module.create(goalC, goalRadius)
 }
ModuleStart.prototype.produce = function(engine) {
    EnvironmentBiggestCircle.send(kBigCircleFindCircle)
    return kCallAgain
}
```

Listing 3. Portion of the script showing the methods of ModuleStart.

```
function BoxShape(pos, size) {
   this.pos = pos
   this.size = size
}

BoxShape.prototype.Subdiv = function (vec) {
   Engine.addModule(new Shape(this.pos+vec, this.size-vec))
   if (vec.x > 0) this.size.x = vec.x
   else if (vec.y > 0) this.size.y = vec.y
   else if (vec.z > 0) this.size.z = vec.z
   return shape
}

BoxShape.prototype.Comp = function () {
   var f = new Array(4)
   f[0] = new Face(this.pos,new Vector3(this.size.x,
                     this.size.y,0))
   this.pos.x += this.size.x
   f[1] = new Face(this.pos,new Vector3(0,this.size.y,
                     this.size.z))
   this.pos.z += this.size.z
   f[2] = new Face(this.pos,new Vector3(-this.size.x,
                     this.size.y,0))
   this.pos.x -= this.size.x
   f[3] = new Face(this.pos,new Vector3(0,this.size.y,
                    -this.size.z))
   return f // side faces
}
```

Listing 4. Scriptal for a box-shaped object.

9.1.4. CGA Shape Grammars

CGA shape is a shape grammar for CG architecture, introduced by Müller et al. [2006]. It can be expressed in the Deco framework in the following way. Simple shapes, such as boxes, can be implemented directly in the scriptal while more complex shapes can be implemented as user-defined objects in dynamic libraries. An object representing a shape defines methods Subdiv and Comp that implement the subdivision operation and return the faces comprising the shape, respectively. Faces are similar to boxes with one component of size set to 0.

An example of a box-shaped object implemented directly in the scriptal is given in Listing 4. The parameters pos, size, and vec are of type Vector3.

The grammar rules can be converted directly into scriptals, with operations determining visibility or snapping tiles to given lines implemented in an environmental library.

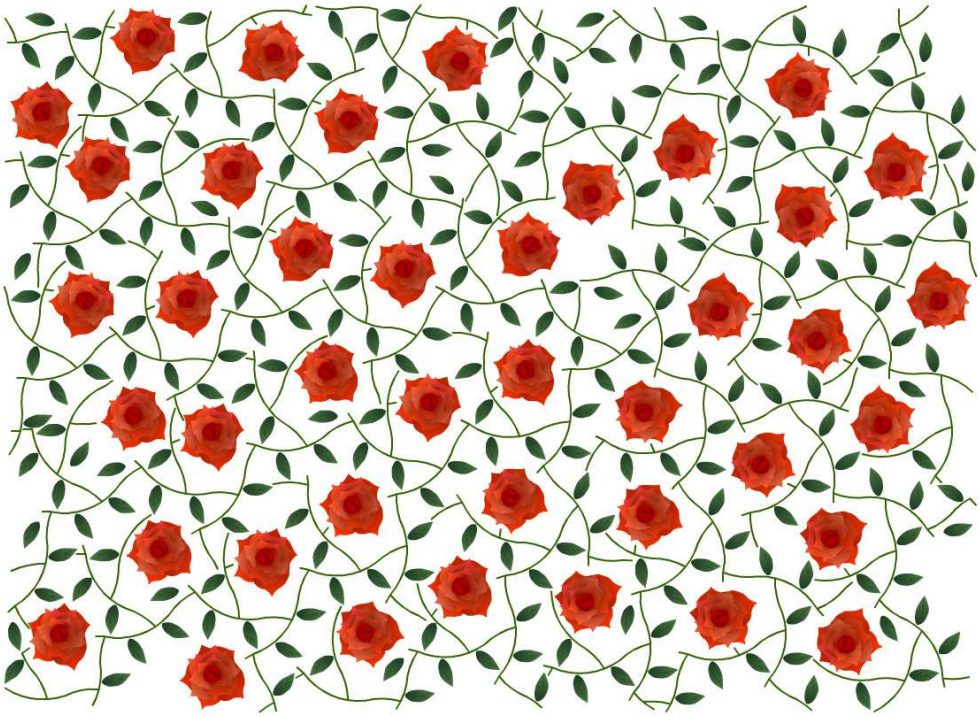


Figure 11. Space-filling flower pattern in Flash Authoring CS4.

9.2. Deco in Flash Authoring

The Deco framework shipped in Adobe's Flash Authoring versions CS4, CS5, and CS6. In version CS4, four tools were implemented using Deco: a spray brush, a flower-fill tool (see Figure 11), a grid-fill tool (an improved version was used in Figure 13 for making the wall), and a symmetry tool (see Figure 8). Seven additional tools shipped in version CS5, including a tree brush

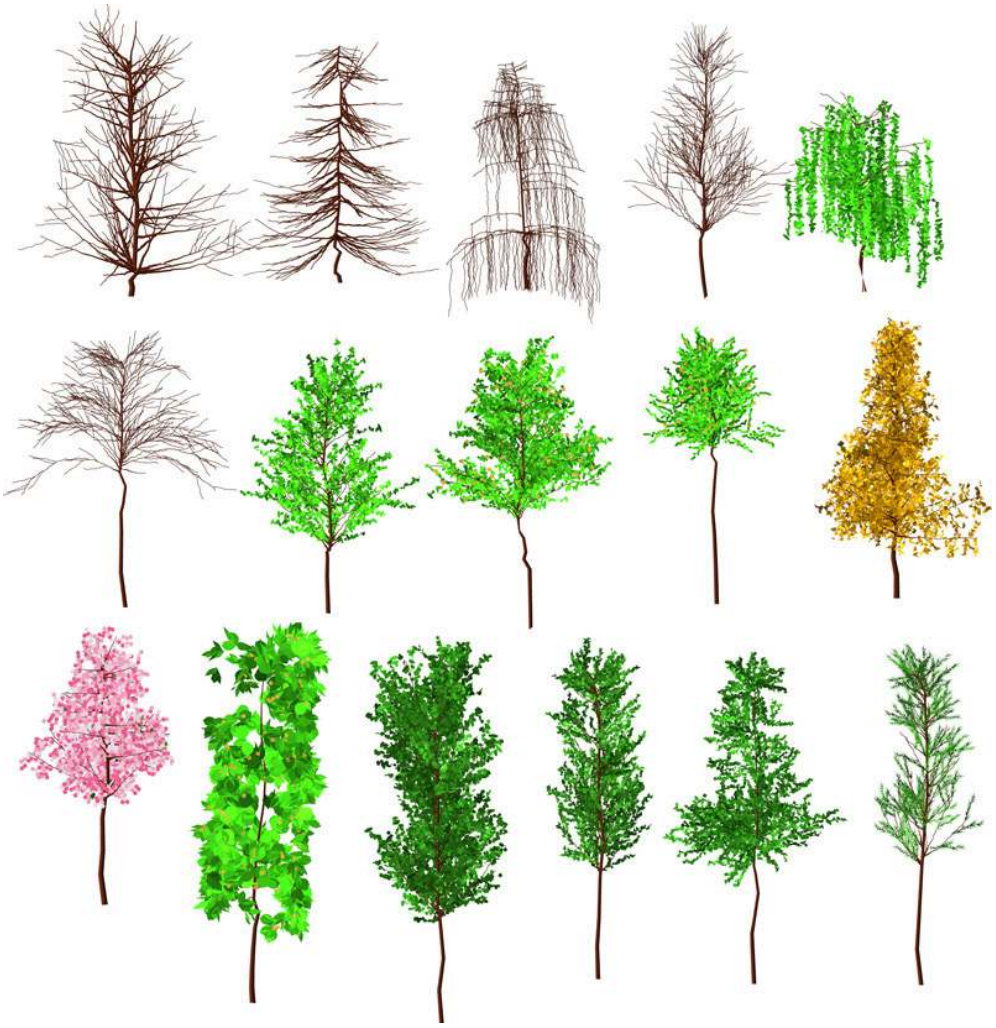


Figure 12. Trees created using the tree brush. A single stroke is guiding the growth of the main axis. The branches are created automatically based on a selection of a tree type in the menu.

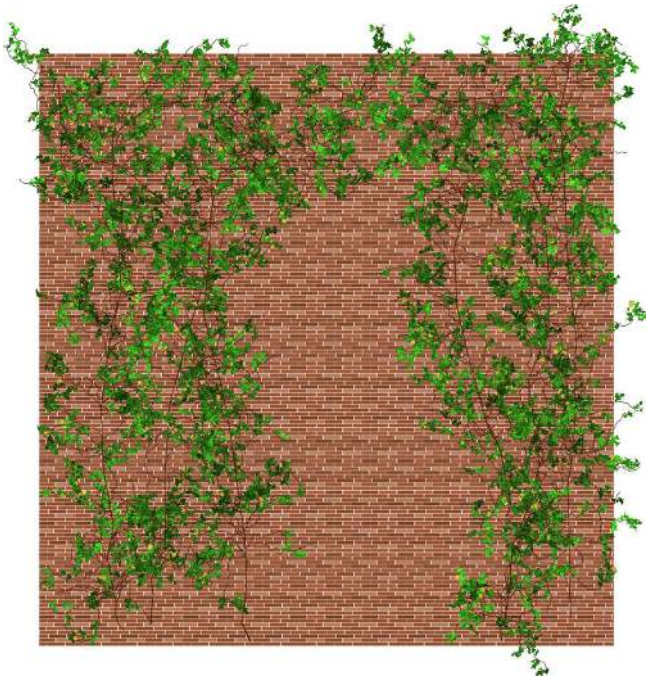


Figure 13. Model of a vine climbing a wall created using the flower brush and the grid-fill Deco tool.

and flower brush (see Figures 1, 12, 13, and 14), or lightning brush (Figure 15). Version CS6 included no new changes.

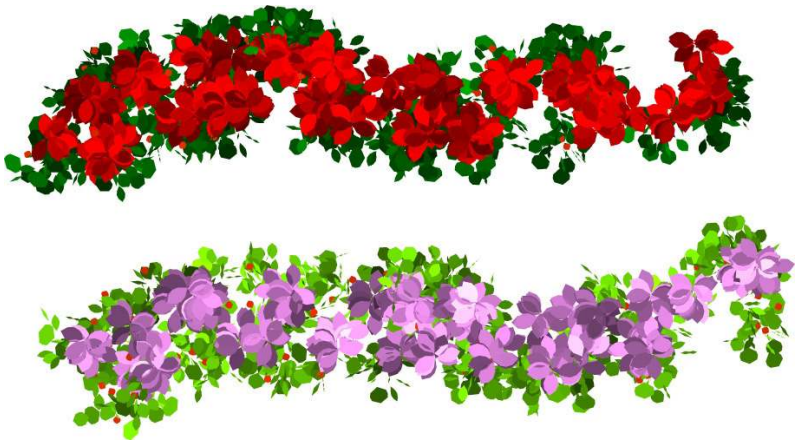


Figure 14. Two strokes created using the flower brush Deco tool.

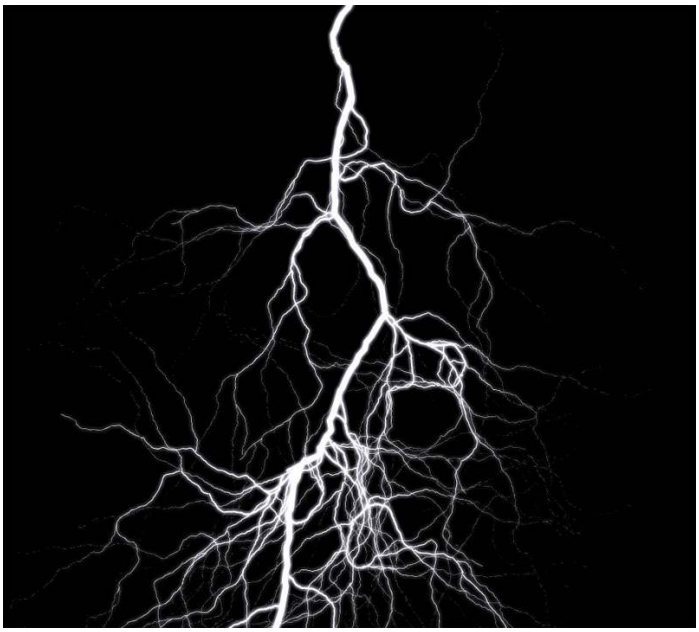


Figure 15. Lightning created using the Lightning Deco tool with added glow.

It is possible to modify the existing scriptals and to add new ones. In Windows, the scriptals can be found in a ProcScripts folder that is located at

```
C:\Users\YourName\AppData\Local\Adobe\Flash CS5\en_US
      \Configuration
```

On OS X, the folder path is

```
\\User\Library\Application Support\Adobe\Flash CS5\en_US
      \Configuration
```

To add new tools you can create a UserScripts folder in the ProcScript folder and place new scriptals there[2].

As we added new tools, we realized that tools created by non-artists (most of those in CS4) are more oriented towards general use and that they produce less visually appealing content. The tools provided in CS5 were created by an artist, who knew scripting and learned the syntax of Deco scriptals. The changes in the application or the Deco framework between CS4 and CS5 were

[2]See http://www.adobe.com/devnet/flash/articles/deco_intro.html

minimal; we improved frame advancing for animation scripts and allowed rendering below a specific object for the 3D brush, so that objects higher on the screen can be drawn behind objects that are placed lower on the screen.

9.3. Deco in Photoshop

The Deco framework has been also shipped with Photoshop CS6. It is used to create procedural fill patterns in the Photoshop's fill tool. Photoshop CS6 contains five scripted patterns that can be accessed by right clicking on a selection and choosing Fill, or by selecting Fill from the Edit Menu. Once the fill dialog box is up you choose Pattern in the Use selection box, and check the checkbox Scripted Patterns.

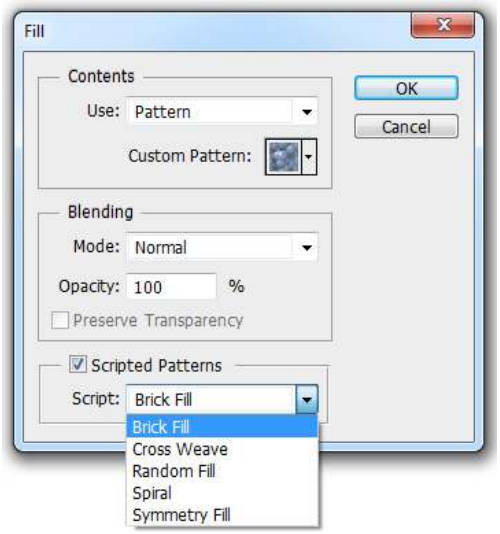


Figure 16. Acessing Deco scripted patterns in the Fill dialog in Photoshop CS6.

These five JavaScript files are located in the following directories for Windows 32-bit, Windows 64-bit, and OS X respectively:

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

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

/Applications/Adobe Photoshop CS6/Presets/Deco
```

There has been a vibrant community of users at the pre-release forums

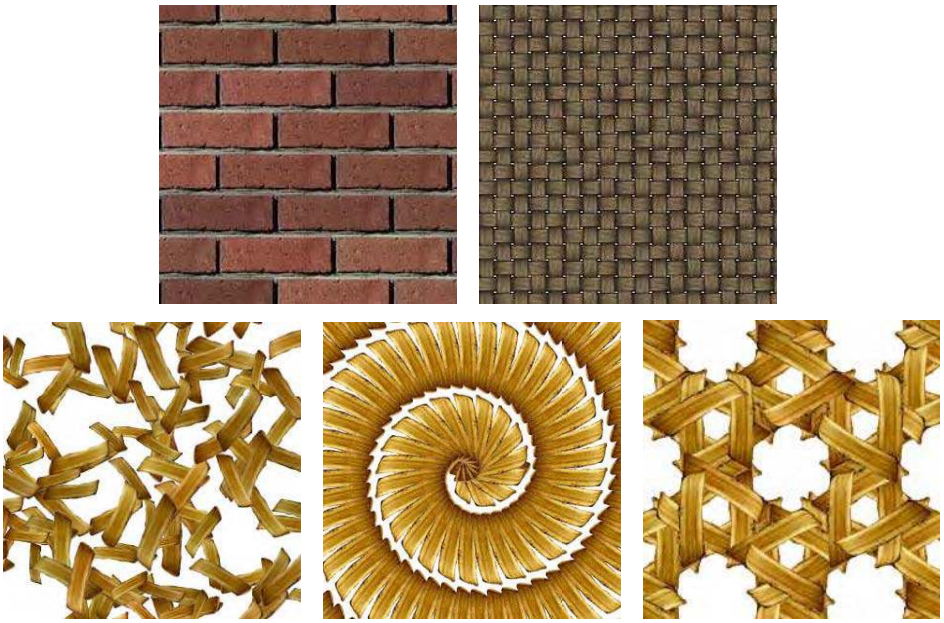


Figure 17. Five patterns generated by scripted Deco pattern fills in Photoshop CS6: Brick Fill, Cross Weave (top), Random Fill, Spiral, and Symmetry Fill (bottom).

already. A few users created new scripts and some managed to cleverly combine regular Photoshop scripts with Deco scripts to extend the functionality of the feature. Others then created beautiful patterns using these new scripts (Figure 18). The website Adobe website provides more information and more scripts at:

http://www.adobe.com/technology/projects/DecoScriptedPatternsInPhotoshop.html

Unlike in FlashPro, though, some functionalities of the Deco framework are not exposed to the script writer. The engine is not processing any user input and the environmental libraries are also not supported.

9.4. 3D Models

Flash Authoring is a 2D application; thus, for developing and testing 3D models we used an OpenGL-based prototype application, wxDeco.

Figure 19 shows two examples of the Streamer model, in which a 3D generalized cylinder is created by defining a few points that chase the cursor. The points maintain a constant speed and change their direction towards the cursor. Shapes that are more winding are created when the cursor moves slowly and less winding when it moves fast.

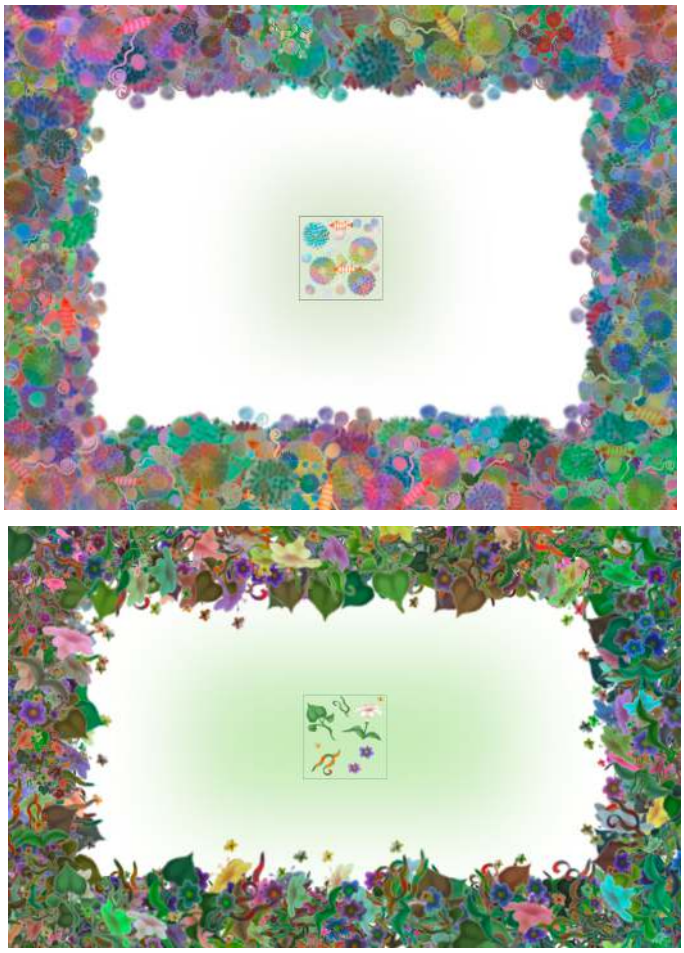


Figure 18. Frame patterns created by Jodi Frye, using random boundary script posted at `http://www.adobe.com/technology/projects/DecoScriptedPatternsInPhotoshop.html`.

The model in Figure 20 uses the Symmetry object to create tilings of streamers.The last example in Figure 21 shows a few frames from an animation of the development of a rose.

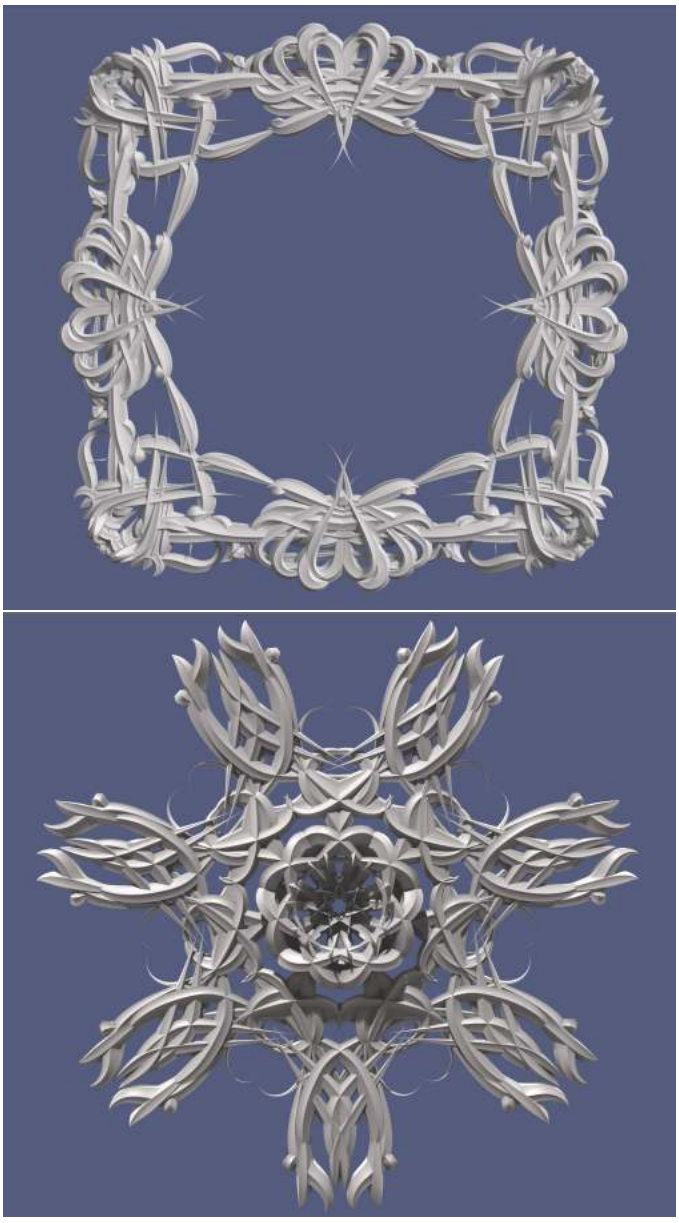


Figure 19. Models of 3D streamers with rotational symmetries.

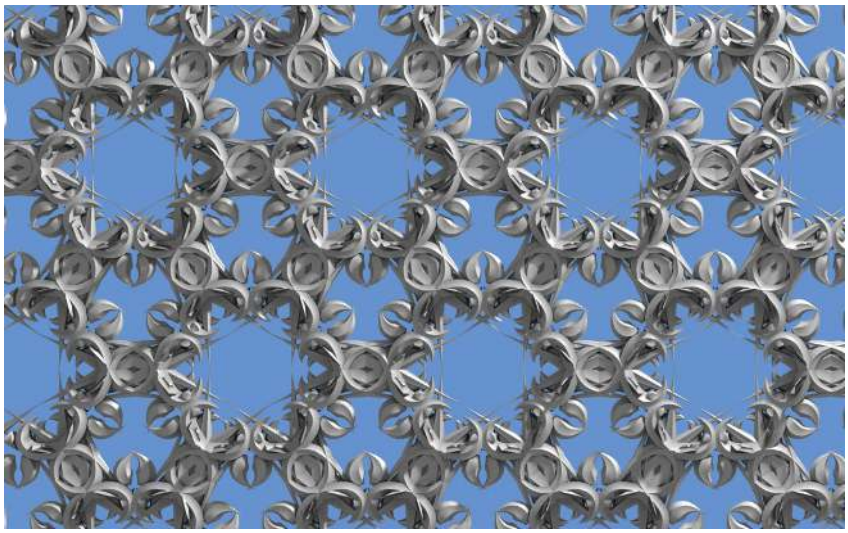


Figure 20. Model of 3D streamers with a tiling symmetry.

9.5. Performance

Performance is important since interactivity is a crucial part of working with tools implemented with the Deco framework. Scriptals are interpreted and if very complex code is executed often, it can impact the performance. We have observed that the performance of the models depends more on the amount of geometry the framework creates than on the complexity of the scriptals. Yet, if parts of the scriptal becomes too complex, we can move some of the code into an environmental library or a user-defined object, which is also loaded as a dynamic library (Section 5).

All examples that use continuous user input run at interactive speeds. Those that are initiated by a user with a mouse click may run slower; the flower-fill

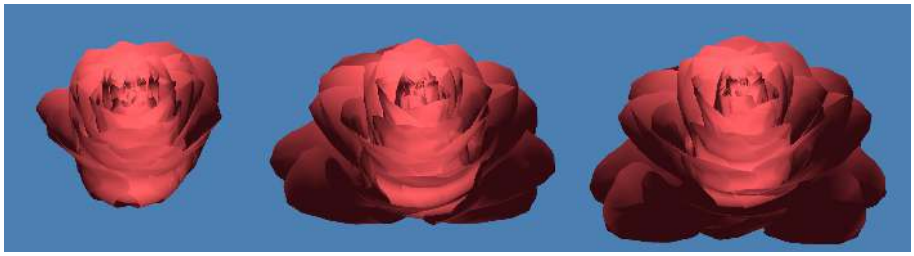


Figure 21. Three frames of a rose animation in wxDeco.

may take a few seconds to fill the screen, for example.

In the 3D prototype application, we can create more complex models since we avoid the overhead of inserting geometry into a scene graph and using the Flash renderer to render the model. In addition, we can make the rendering efficient through incrementally rendering the geometry into the front buffer. As illustrated in Figure 20, quite complex structures can be created interactively by brushing. Once the structures were created, they were output into an obj file and rendered using the Adobe ray tracer.

10. Conclusions

Deco is a powerful framework for interactive procedural modeling. It was designed so that it can express existing procedural models, that the models are easy to specify in a well-known scripting language with various functionality provided as predefined objects, and that it can support various methods for user interaction with the models. In addition, the framework can be deployed to a variety of applications by implementing a relatively thin application specific plug-in layer. Building the dynamic menus for each scriptal proved to be the only difficult part of implementing the plug-in layer. The framework itself is extensible by using environment libraries and user-defined objects.

The framework is accessible to the user at several levels. Only a small group of developers will create their own environmental libraries. Most users who are familiar with scripting can write their own scriptals or modify the existing ones. Most artists just choose from a set of predefined scriptals provided with the application as a list of Deco tools; they can modify various parameters of the model in a menu and control the procedural growth interactively.

Including the Deco framework in a commercially available product brought interesting challenges not previously considered in procedural modeling. One of them is undo and redo that is currently supported by the framework by tracking modules created between predefined undo points. Another was rendering into a scene graph, for which the incremental rendering mode was introduced. Another challenge is saving models into a file when the user exits an application with an active tool that still remains as a part of future work.

We hope to continue expanding the framework by introducing more built-in geometric primitives or by creating additional environmental libraries, as well as including Deco in other applications. We also realize that authoring scriptals is not easy for most artists; we are investigating various methods

that make it possible to create procedural models from a given example, or interactively.

A. Object Engine

The procedural engine can be accessed as an object Engine in the scriptal. The class includes the following methods:

```
addModule (object)
removeModule (object)

addEnvironment (Name [, objectName] [, sendMouseEvents])

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

loadGeometry (filename)
exposeGeometry (name, geometry/function)
loadUserObject (dllName [, objectName])
evalFile(scriptName [, init func parameters])

render (renderAPI)
moduleProcessed (module)
stopPass()

makeMenu()
message/warning/error (strings, variables)
```

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

The method addEnvironment loads an environmental library. Optional parameters specify the name of the corresponding javascript object (allowing the user to load the same library more than once).

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

- kRandomEnvironment: if set to 1, the module's environmental methods are called in random order.
- kApplyFrame: if set to 1 (default), the module's frame is automatically applied before a render method is called.

- kUpdateTopology: this parameter is followed by a topology object and a pair of corresponding parent-child labels. Afterwards, if a new module N is added to the procedural engine from the produce method of a module M, and the module M is connected to another module P using the given topology and labels, the module N is connected to the module P (see Section 5.1).
- kIncrementalRender: if this parameter is set to 1, the procedural engine keeps track of which module has been rendered and it does not call its render method again, even if the method returns kCallAgain.
- kRunSimulation: if this parameter is set to 1, the procedural engine informs the application that the simulation should be automatically started.

The method setModuleParameter controls various parameters of a given module. The second parameter of the method is one of the following constants:

- kModuleSaveRenderCalls: if set to 1 (default), the procedural engine saves all rendering calls made from the render method.
- kModuleProcessed: if set to 1, the procedural engine stops further processing of the module.
- kModuleApplyFrame: this parameter overrides the global parameter set in the procedural engine.
- kCall: this parameter is followed by one of kProduce, kRender, 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 loadGeometry loads geometry from a file and returns a Geometry object (see Appendix E). Currently, only obj and svg files are processed.

The method exposeGeometry informs the procedural engine about a geometry object that can be modified by the application (see Section 8.2).

The method loadUserObject loads a user-defined dynamic library representing an object. The dynamic library contains a single class derived from a simple basic class.

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.

The method render performs the rendering pass using the given renderAPI. In normal circumstances, the engine automatically calls render, but the user may want to use this method if it is desirable to render the scene using an additional renderer.

The method moduleProcessed returns true if the given module has been already processed— if its produce method has been called already in the current step—or if it is a newly created module for which the produce method is not being called.

The method stopPass terminates the current produce, render, or environment pass.

The method makeMenu defines the menu for the script (see Section 8.1).

Methods message, warning, and error can be used to print a message into some log window or issue an error notice. The parameters are strings and variables, separated by commas.

B. Object Frame

Frame is a 4×4 matrix specifying the model's position and orientation. There are two frame objects Frame2 and Frame3. 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 (),
heading (), up (), right (),
size (),
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)
```

Most of the methods are self-explanatory. The parameter vector is an instance of the object Vector (see Appendix C). The heading is represented by the direction $(0,1,0)$ and the up vector is $(0,0,1)$.

The method rotateDeg rotates the frame around the vector $(0,0,1)$. Methods rotatePitch, rotateYaw, and rotateRoll, rotate the frame around vectors $(1,0,0)$, $(0,0,1)$, 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, contained in the script, defining two- through four-dimensional vectors. The elements of a vector can be accessed using *.x*, *.y*, *.z*, and *.w*.

The object's methods length(), lengthSquared(), dot(vector), normalize(), cross(vector) 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 class is used as a base class for renderers, exporters, or even environmental libraries, basically any time the primitives representing the model are needed. The class has the following methods:

```
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)

Instantiate(id), endInstantiate()

translate(x,y,z|vector), scale(x,y,z|vector)
setFrame(frame), getFrame()
pushMatrix(), popMatrix()

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

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

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 Béziers, a frame can specify two points, one at its origin and one at $(0,1,0)$.

The starting point of an arc is the point $(0,0)$ and the center is given as the point $(\text{radius},0)$. 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 Appendix G).

The method Instantiate marks the beginning of a sequence of RenderAPI calls that will be instantiated until the endInstantiate method is called. The parameters of the method are used to identify the instance (see Section 7.2).

The method setFrame sets the current frame. In fact, it multiplies the existing frame with the new one; you need to use pushMatrix and popMatrix calls if you do not want this frame to persist. The method getFrame gets the current frame.

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. To do that, you create a Light object using new Light, set its parameters and then call setLight. Lights are indexed from 0.

E. Object Geometry

Geometry 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])
setColor(kStrokeColor|kFillColor, red, green, blue [, alpha])
setFrame(frame)
multFrame(frame)
pushFrame()
popFrame()
render(renderAPI [, send_bbox])
instantiate(renderAPI)
getValue(kGetGeometryLength | kGetPointAlongGeometry)
```

The load method 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 Geometry object. The parameters are similar to those in the RenderAPI object.

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 only the bounding box of each primitive is sent.

The method instantiate instantiates the primitives for the given API so that when the render method is called repeatedly only the instance is invoked. Not all renderers may support instantiation.

F. Object Image

The object Image is a container for images. The object has the following methods:

```
load (filename)
save (filename)
getSize()
setParameter(paramType, value)
getParameter(paramType)
render(renderAPI])
```

Methods load and save can be used to load the image from a file or save it into a file. Currently, it is possible to load and save tga and png files.

The method getSize returns the size of the image as a Vector object. Methods getParameter and setParameter querries and sets various parameters, respectively. In Photoshop CS6, the parameter type can be kpsColorBlendMode or kpsPatternBlendMode and the value is one of Photoshop blending modes: 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, and kpsBlendDivide.

The method render sends the image to the given renderer.

G. 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, P_1 and P_2, define a Bézier curve that forms an axis of the generalized cylinder, along which the contour curve is swept. The Bézier curve is defined by points $(0,0,0)$ and $(0,1,0)$ in frame coordinates of P_1 and points $(0,0,0)$ and $(0,-1,0)$ in frame coordinates of P_2.

G.1. Cross-Section Curve

The cross-section curve, a contour, is defined as a set of Bézier 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 the plane $z = 0$. An object Curve can consist of several primitives, added incrementally. Listing 5 gives 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)
```

Listing 5. An example of a simple contour curve.

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 differ they are interpolated along the generalized cylinder.

G.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 $(\text{start_radius}, 0, 0)$ and ends at $(\text{end_radius}, \text{y}, 0)$. The curve is stretched in the y-axis direction to fit the length of the axis of the generalized cylinder between the two control points P_1 and P_2. 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.

G.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 Bézier curves—so that their lengths are the same. In order to preserve sharp features or to avoid further tessellating line segments on the curve, the number of segments for each primitive on the curve can be specified, indexed in order:

```
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 P_1 and P_2 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 P_1 and P_2 are interpolated, as are coordinates of corresponding contour points at point P_1 and P_2. 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.

H. Scriptal for Loop Subdivision

The scriptal in Listing 6 implements Loop subdivision, based on the algorithm by Smith [2003].

The mesh is loaded using the MeshLoader user object that loads a specified obj file, creates a set of objects of a given name, in this case Vertex, and connects them using the Topology object. Note that a module is not removed from the engine until all modules are processed due to delayed updates of connections.

```
// Inserts new vertex between vertices p and q
function insert (v, p, q, label) {
  Topology.setAll (v, label, p, q);  // set neighbors of v
  Topology.replace (q, label, p, v); // q, an existing vertex
  Topology.replace (p, label, q, v); // p is a new vertex
}

function Vertex (pos) {
  this.pos = pos;
}

Vertex.prototype.produce = function (engine)
{
  // Get old neighbors
  // before any changes were made in this step
  var label = "vertex";
  var neighbors = Topology.getAllOld (this, label);
  var n = neighbors.length;


  // Weight of old position of neighbors on the new position
  var w = (5/8)/n - Math.pow (3/8 + 0.25*cos (2*PI/n), 2) / n
  var pos = this.pos * (1 - n * w)

  var newVertex = new Vertex (pos)
  // Add the new vertex to the engine
  Engine.addModule (newVertex)

  // Move new connections from this to newVertex
  Topology.move (newVertex, this)

  var i
  for (i = 0; i < n; i++) {
    var q = neighbors[i]
    newVertex.pos += w * q.pos
    if (Engine. moduleProcessed (q))
      continue // to avoid creating twice
```

```
    var vpos = 3/8*this.pos + 3/8*q.pos
      + 1/8*Topology.getPreviousOld(this, label, q).pos +
      + 1/8*Topology.getNextOld(this, label, q).pos

    var edgeVertex = new Vertex(vpos)
    engine.addModule (edgeVertex)
    insert (edgeVertex, newVertex, q, label)
  }

  // Get new mid egde neighbors
  var midEgdeNeighbors = Topology.getAll (newVertex, label)
  for (i = 0; i < newNeighbors.length; i++) {
     var midEdge = midEgdeNeighbors [i]
     // Get the neighbor at the end of the old edge
     var edgeNeighbor = Topology.getPrevious(midEdge,label,
                                                 newVertex);
     if (!engine.moduleProcessed (edgeNeighbor))
       continue // edgeNeighbor is not created yet

     Topology.setAll (x, label, newVertex,
         Topology.getPrevious (newVertex, label, midEdge),
         Topology.getNext (edgeNeighbor, label, midEdge),
                           edgeNeighbor,
           Topology.getPrevious (edgeNeighbor, label,
                                 midEdge),
           Topology.getNext (newVertex, label, midEdge));
  }
  engine.removeModule (this)
}

Engine.loadUserObject ("MeshLoader");
MeshLoader.load ("obj/cubeNoTop.obj", "Vertex", Engine,
                   Topology);
```

Listing 6. A scriptal that implements Loop subdivision.

I. C++ API for Environmental Libraries

Listing 7 lists the header file that defines the API, through which the procedural engine communicates with environmental libraries. Each environmental library is subclassed from the class Environment, defined here.

The included header files define a Variant class, a union of integer, float, char *, void *, Frame *, and Module *. It also defines vArgs as an array of Variants and a guiEvent for storing events coming from the main application.

```
#include "Variant.h"
#include "Event.h"
#include "RenderAPI.h"
#include "UserMenu.h"

typedef Variant (*queryCallback)(int object,
                   char *methodname, vArgs methodParams);

class Environment : public RenderAPI  {
   char *objectName;
public:
   Environment (char *n) :  objectName (n) {};
   virtual ~Environment () {};

   virtual char *getObjectName (void) { return objectName; }

   // Queries
   virtual int query (vArgs input, vArgs *output = NULL) = NULL;

   virtual int send (vArgs input) { return query (input); }

   virtual int delayedQuery (vArgs input,
                             queryCallback callback,
                             int object, char * methodname,
                             vArgs methodParams) { return 0; }

   // User input, return 0 if no events required
   virtual int sendEvent (const guiEvent *event) { return 0; }

   // Menus
   virtual UserMenu *getMenu (void) { return NULL; }
   virtual int setParamValue (const char *varname,
                              Variant value) { return 0; }

   // Called by the procedural engine
   // when a module is removed
   virtual void removeModule (Module *moduleID) {};

   // Rendering, return 0 if no rendering required
   virtual int render (RenderAPI *api) { return 0; /*stop */ }
};
```

Listing 7. Header file for C++ API.

References

ALIAGA, D. G., ROSEN, P. A., AND BEKINS, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics 13*, 4, 786–797. 46

ANASTACIO, F., PRUSINKIEWICZ, P., AND COSTA SOUSA, M. 2009. Sketch-based parameterization of L-systems using illustration-inspired construction lines and depth modulation. *Computers & Graphics 33*, 4, 440–451. 47

BOUDON, F., PRUSINKIEWICZ, P., FEDERL, P., GODIN, C., AND KARWOWSKI, R. 2003. Interactive design of bonsai tree models. *Computer Graphics Forum (Proc. Eurographics) 22*, 3, 591–599. 47

CHEN, X., NEUBERT, B., XU, Y.-Q., DEUSSEN, O., AND KANG, S. B. 2008. Sketch-based tree modeling using markov random field. In *ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 109:1–109:9. 47

FLEISCHER, K., AND BARR, A. H. 1994. A simulation testbed for the study of multicellular development: The multiple mechanisms of morphogenesis. In *Artificial Life III*, Addison-Wesley, Reading, MA, C. G. Langton, Ed., 389–416. 45

GIAVITTO, J.-L., AND MICHEL, O. 2001. Declarative definition of group indexed data structures and approximation of their domains. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, ACM, New York, NY, USA, 150–161. 56

HART, J. C. 1994. On efficiently representing procedural geometry. *Manuscript, Jul. 1994. (An early version of one chapter in D. Ebert, Ed., Texturing and Modeling: A Procedural Approach. Morgan Kaufmann, Nov. 2002.)*. 45

IJIRI, T., OWADA, S., AND IGARASHI, T. 2006. The sketch L-system: Global control of tree modeling using free-form strokes. In *Smart Graphics 2006*, ACM, New York, NY, 138–146. 47

KARWOWSKI, R., AND PRUSINKIEWICZ, P. 2003. Design and implementation of the L+C modeling language. *Electr. Notes Theor. Comput. Sci. 86*, 2. 48

LINDENMAYER, A. 1968. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology 18*, 3 (March), 280–315. 46

LINTERMANN, B., AND DEUSSEN, O. 1999. Interactive modeling of plants. *IEEE Comput. Graph. Appl. 19*, 1, 56–65. 46

LIPP, M., WONKA, P., AND WIMMER, M. 2008. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, ACM, New York, NY, USA, 1–10. 47

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, 614–623. 46, 52, 74

MÜLLER, P., ZENG, G., WONKA, P., AND GOOL, L. V. 2007. Image-based procedural modeling of facades. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, ACM, New York, NY, USA, 85. 46

MĚCH, R., AND PRUSINKIEWICZ, P. 1996. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 397–410. 48, 49, 52

OKABE, M., OWADA, S., AND IGARASHI, T. 2006. Interactive design of botanical trees using freehand sketches and example-based editing. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, 18. 47

PALUBICKI, W., HOREL, K., LONGAY, S., RUNIONS, A., LANE, B., MĚCH, R., AND PRUSINKIEWICZ, P. 2009. Self-organizing tree models for image synthesis. In *Proc. SIGGRAPH 2009*, ACM Press, New York, NY. 47, 68

PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 301–308. 46

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA. 44, 46

PRUSINKIEWICZ, P., AND ROLLAND-LAGAN, A.-G. G. 2006. Modeling plant morphogenesis. *Current opinion in plant biology 9*, 1 (February), 83–88. 46

PRUSINKIEWICZ, P., MÜNDERMANN, L., KARWOWSKI, R., AND LANE, B. 2001. The use of positional information in the modeling of plants. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 289–300. 46

PRUSINKIEWICZ, P., SAMAVATI, F. F., SMITH, C., AND KARWOWSKI, R. 2003. L-system description of subdivision curves. *International Journal of Shape Modeling 9*, 1, 41–59. 46

PRUSINKIEWICZ, P. 1986. Graphical applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 247–253. 46

REEVES, W. T., AND BLAU, R. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph. 19*, 3, 313–322. 46

SMITH, C., PRUSINKIEWICZ, P., AND SAMAVATI, F. F. 2003. Local specification of surface subdivision algorithms. In *Applications of Graph Transformations with Industrial Relevance (Second International Workshop, AGTIVE 2003)*, 313–327. 46, 56, 71, 92

TOFFOLI, T., AND MARGOLUS, N. 1987. *Cellular automata machines: a new environment for modeling*. MIT Press, Cambridge, MA, USA. 45

VON NEUMANN, J. 1966. *Theory of Self-Reproducing Automata*. University of Illionois Press, Champain, IL. 45

WEBER, J., AND PENN, J. 1995. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 119–128. 46

WITHER, J., BOUDON, F., CANI, M.-P., AND GODIN, C. 2009. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *Comput. Graph. Forum 28*, 2, 541–550. 47

WONG, M. T., ZONGKER, D. E., AND SALESIN, D. H. 1998. Computer-generated floral ornament. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 423–434. 46, 49, 52, 72

Author Contact Information

Radomír Měch
Gavin Miller
345 Park Ave
San Jose, CA 95110
rmech@adobe.com
gmiller@adobe.com

Radomír Měch and Gavin Miller, The *Deco* Framework for Interactive Procedural Modeling, *Journal of Computer Graphics Techniques (JCGT)* 1, 1, 43–99, 2012
http://jcgt.org/published/0001/01/04/

Received:	December 6, 2012		
Recommended:	December 21, 2012	Corresponding Editor:	Morgan McGuire
Published:	December 28, 2012	Editor-in-Chief:	Morgan McGuire

© 2012 Radomír Měch and Gavin Miller (the authors).
The authors provide this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at http://creativecommons.org/licenses/by-nd/3.0/. The authors further provide the first page as a separable document under the CC BY-ND 3.0 license, for use in promoting the work.

