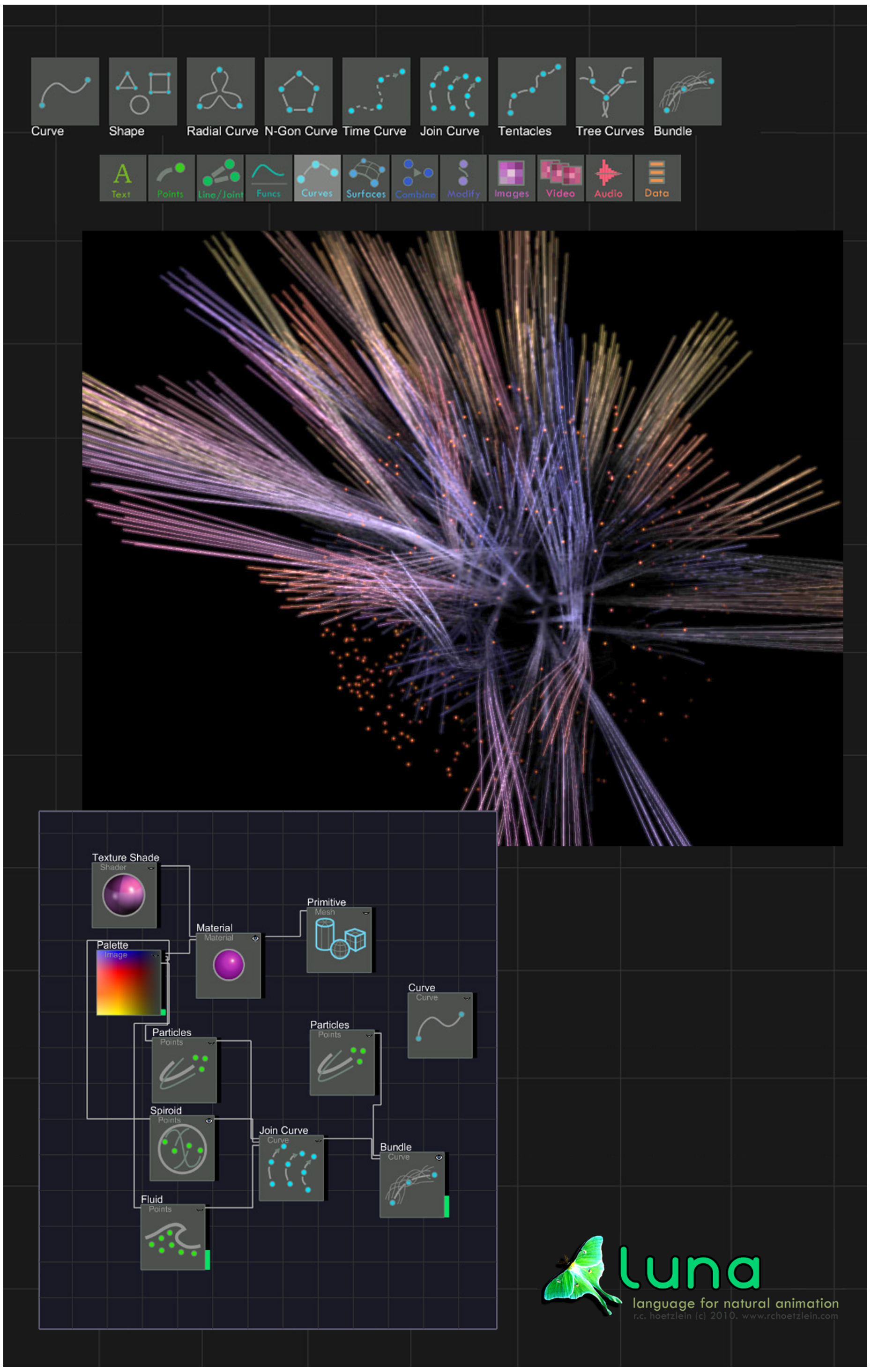
Chapter 3

LUNA: A Puzzle-Based Metaphor for Procedural Modeling

3.1 Introduction

Creative tools for digital artists have evolved considerably over the years. Since early systems such as Sutherland's Sketchpad [Sutherland, 1988] presented the first opportunity to directly interact with computers, users have been able to paint textures directly on surfaces [Blinn and Newell, 1976] [DeBry et al., 2002], and to interactively sculpt three-dimensional objects themselves [Lawrence, 2004]. These kinds of direct interactions, similar to physical artistic practices, are shifting more recently toward interaction in augment reality [Bandyopadhyay et al., 2001] [Ryokai et al., 2004] [Jacucci et al., 2005]. Yet tools for conceptual artists, who may view the art object as a dynamic system or model, have evolved more slowly.

While digital interfaces for the plastic arts are now common, such as painting with Photoshop or sculpting with Zbrush, visual interfaces for conceptual artists in the form of visual data flow languages are still relatively new and still evolving. Early prototypes


Curve
Shape
Radial Curve
N-Gon Curve
Time Curve
Join Curve
Tentacles
Tree Curves
Bundle
Text
Points
Line/Joint
Funcs
Curves
Surfaces
Combine
Modify
Images
Video
Audio
Data
Texture Shade
Shader
Material
Material
Primitive
Mesh
Palette
Image
Curve
Curve
Particles
Points
Particles
Points
Spiroid
Points
Join Curve
Curve
Bundle
Curve
Fluid
Points
luna
language for natural animation
r.c. hoetzlein (c) 2010. www.rchoetzlein.com

such as ConMan [Haeberli, 1988] introduced the notion of dataflow interaction for graphical objects, while IBM's Visualization Data Explorer [IBM, 1999] provided a wide range of tools for processing scientific data. Educational systems such as Alice and Scratch employ a visual data flow metaphor, but these are designed largely to teach programming concepts rather than to simplify artistic development [Conway and Pausch, 1997] [Resnick et al., 2009]. Commercial systems such as Houdini enable content creation for artists in the entertainment industry [Bannink, 2009], using procedural methods in an offline setting to develop physical simulations and special effects. However, there is relatively little current academic research on novel designs for visual data flow languages in comparison to augmented interfaces for physical manipulation.

For media artists working with live performance, visual data flow languages such as Max/MSP/Jitter, 'vvvv', and Soundium offer an interactive node-based interface. These systems have been employed in major international live exhibition artworks. Max/MSP/Jitter was founded on digital signal processing, with objects that can process audio signals in real-time, and can transform these signals into graphic primitives [Jones and Nevile, 2005]. Soundium is a visual language that allows for interactive composition of visual elements, such as shapes, images and video, during a live performance [**?**] while 'vvvv' provides for graphical interaction with support for multiple display rendering and geometric primitives such as mesh objects [Meso, 1998]. While each of these systems have different affordances, none of them employs a procedural modeling paradigm to allow for complex geometries. Although systems for media artists have evolved to support live performance they have remained relatively low-level in comparison to the model complexity offered by offline procedural tools.

Studies examining how conceptual artists interact, or would like to interact, with procedural data flow interfaces are not as common as studies of painting or drawing interfaces. In a cognitive study of visual dataflow programming, Green defines and explores several aspects of visual interfaces: 1) *commitment*, when the language requires early decisions, 2) *progressive evaluation*, the ability to see intermediate results, 3) *expressiveness*, how easy it is to say what you want, 4) *viscosity*, how much the interface resists change, and 5) *visibility*, how easily you can see what you're creating [Green and Petre, 1996]. While Green admits it may be difficult to establish quantitative measures of these, his criteria establish guidelines for evaluating visual languages.

To understand interface issues that are directly relevant to media artists, several dimensions of creativity of interest to this group were examined prior and during software development. These dimensions, 1) programming, 2) modality, 3) live performance, 4) dynamics and behavior, 5) structure and surface, and 6) image and idea, were found to represent common themes that media artists working with visual forms seek to explore (see Chapters 5 and 6 for a detailed discussion of these dimensions). *Programming* refers to the desire of some artist to engage in programming, while others seek to explore ideas visually, without programmatic knowledge. *Modality* is the ability to engage with different types of media, such as images, surface, video, and audio. *Live performance* refers to the desire of some artists to have immediate, real time feedback, with high quality output in full screen for installation and performance. *Dynamics* refers to certain artist's interest in exploring motion and behaviour, while *Structure and Surface* refers to artists interested in expressing geometric forms and their material appearance. Finally, *Image and Idea* refers to media artists desire to work with the image as a conceptual object with semantic content.

These dimensions establish the basis on which interface decisions were made to create LUNA, a novel visual data flow language for procedural modeling. LUNA is inspired by the board game Scrabble, where the ability to express a wide range of words comes from a combinatorial rearrangement of only a few tiles. The dimensions of creative exploration for media artists are used to inform the design decisions of the language, resulting in a minimalist approach which supports real time interaction for complex procedural models. Result consists of a number of cross-disciplinary projects, and comparisons of user interactions performed in Luna and Houdini.

3.2 Interface Design

Design of the visual data flow language for LUNA is motivated by the board game Scrabble, specifically the power of expression resulting from a combinatorial arrangement of minimally designed tiles. While similar systems use graphs to allow for such combinations, they are often superimposed with a number of other interface elements and widgets which must be read by the user in order to be understood. This need to "read" the interface, we would suggest, reduces the rate at which different conceptual designs may be explored by the artist. LUNA is developed using a minimalist approach, in which the combinatorial aspects of expression provide a top level interaction which is then continually refined through more specific interactions. Design decisions are motivated toward eliminating the act of reading in favour of visual metaphor.

3.2.1 Design Decisions

Support for creative exploration influenced the primary design decisions in the development of LUNA. Among these was the desire to develop a tool that builds on the dimensions of creativity established earlier by informing the goals of the language. The dimensions explored, and their impacts on language interface design, are as follows:

1. Programming - Mathematical knowledge should be optional to the artist. The primary mode of interaction should be conceptual and exploratory, suggesting a visual data flow language with a minimalist design aesthetic.

2. Modality - The language itself should offer a range of different media types. These are supported through a tool set that includes points, curves, surfaces, images, and materials. The interface should make the media types, which are the primary objects being operated on, apparent to the user while working.

3. Live Performance - The interface should enable live performance by allowing for full screen, high quality output, with real time design changes. This aspect informs the overall visual design of LUNA, resulting in an inverted window layout that floats the

menus and interface elements *above* a full screen output canvas, rather than surrounding the canvas by interaction tools.

4. Dynamics and Behaviour - The system should allow users to interact with dynamic objects and receive immediate feedback. This aspect informs LUNA through the use of property panels that resemble mixing boards, with large, intuitive sliders that modify on-screen behavior.

5. Image and Idea - The language should allow complex, semantic transformations to be performed easily without lengthy interactions. For example, setting up motion capture or computer vision processes (e.g. transforming and image into semantic labels), should be possible without a length set up process to define parameters. This criteria influenced the decision in LUNA to require that each node provide intuitive default behaviors as soon as it is first placed.

The expressive power of a procedural modeling language is influenced by both its interface and the underlying structures that supports it. In the development of LUNA interface design decisions have had direct effect on the structure and design of the modeling language. This language structure, discussed in detail in Chapter 4, consists of geometric and media elements that flow through an abstract graph, similar to systems such as ConMan and Houdini. The primary contributions explored in this chapter are the graphical user interface features that directly enable the abstract goals described above.

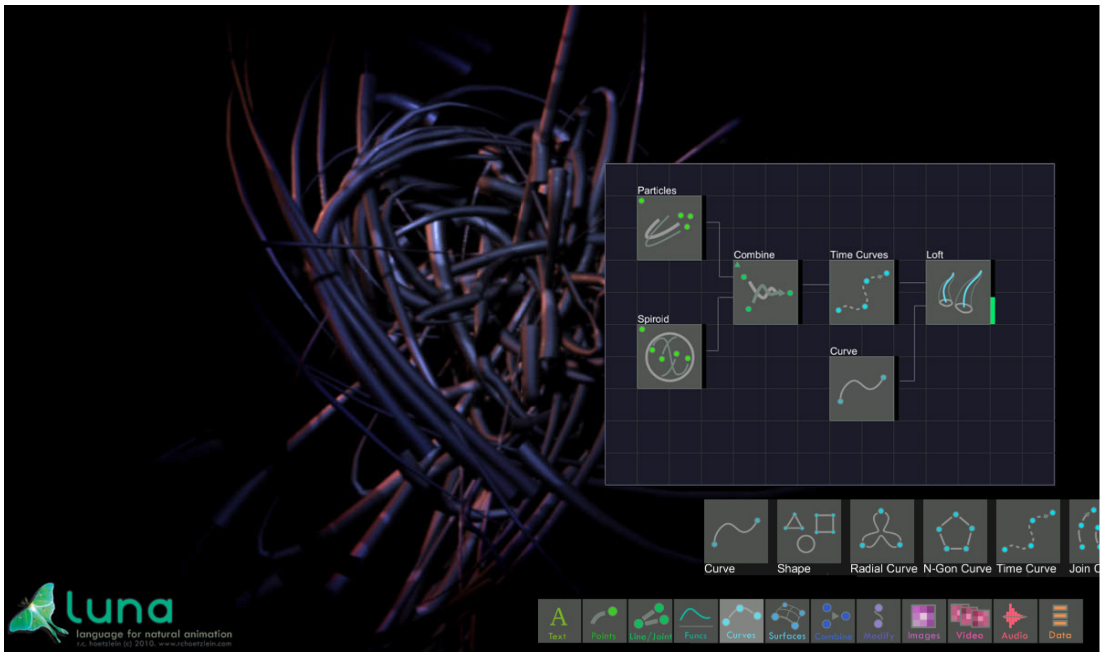


Figure 3.1: LUNA interface with full-screen background rendering and floating foreground elements. Interface elements may be hidden, or moved to a second display, to allow for live performances.

3.2.2 Workspace Layout

The general design of LUNA follows a workspace model, but one which has been inverted from the common layout (see Figure 3.9). Typically, the work flow of a procedural modeling tool employs a central view surrounded by menus and interaction panels. In the interest of live performance, and in order to emphasize the result of aesthetic explorations, this layout is inverted by using the entire display area as the space for the output, while floating the interaction panels above the output. Rather than consider the viewport as an intermediate result, as is common in other commercial packages (e.g. Maya, Houdini), it is designed as a primary output window with high quality rendering using deferred shading techniques typically found in gaming. Of course, the layout is flexible and all windows, in including the deferred shading output window, are resizeable and movable.

3.2.3 Object Icons

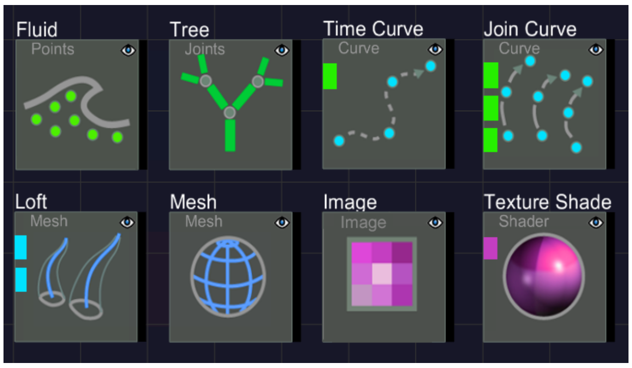


Figure 3.2: Symbolic icons in LUNA with dominant color used to indicate the base output type. The symbol uses the majority of the icon space.

In many visual graph languages, a tool bar with an iconic depiction of the object is used to quickly identify objects of interest. However, once the object is placed on the graph, the icon is removed and replaced by a textual description, object or class name. The LUNA interface retains the symbolic icon, and enlarges it, with minimal text above the graph object to describe its type, Figure 3.2. This allows the user to see, at a glance, the visual meaning of each object in the graph. Careful design of the icons gives a strong impression of the actual output the graph will produce.

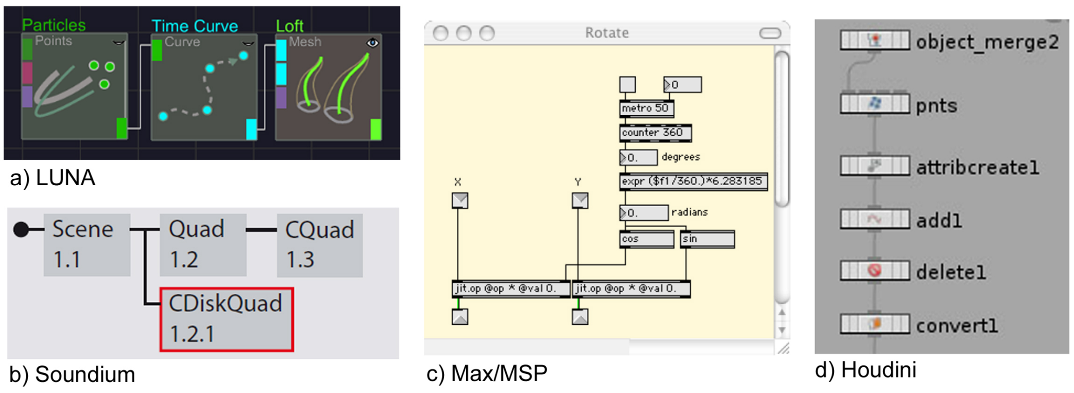


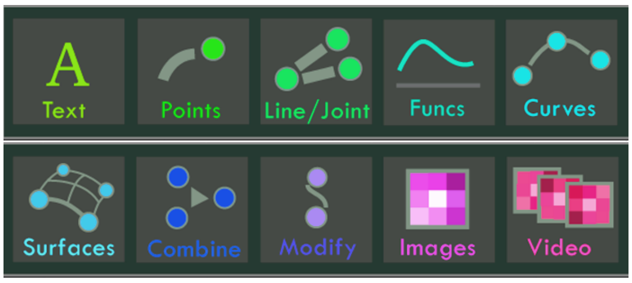
Figure 3.3: Colored tabs in LUNA (a) indicate media types as they flow through the graph, with green for points and blue for curves, in comparisons with other data flow interfaces from b) Soundium, c) Max/MSP, and d) Houdini.

3.2.4 Colored Inputs

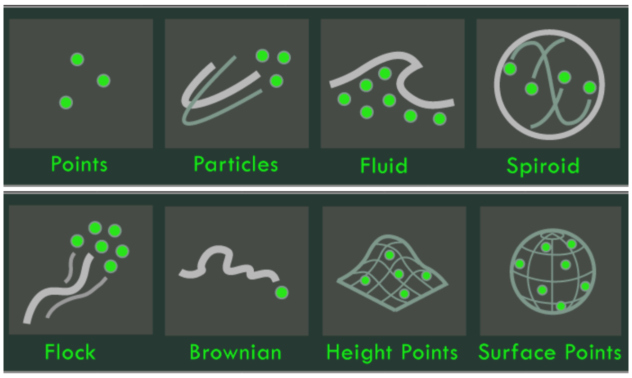
Procedural graphs often use nodes with input and output tabs to represent the arguments to a function. When constructing graphs, it can be difficult to determine which nodes are compatible with which inputs and outputs, often necessitating a help reference in order to construct syntactically valid graphs. In order to alleviate this problem, LUNA uses colored tabs to identify compatible input nodes, shown in Figure 3.3. This allows the artist to quickly see what *modality* they are working with throughout the design process. Objects without input tabs are generator objects, while objects colored grey are modifiers which accept several media types as input.

3.2.5 Toolbar Design

The design of toolbars for large numbers of objects in procedural systems is an on-going challenge. In many systems, objects are categorized according to *workflow*



(a) Primary toolbar with categories for geometry output types



(b) Secondary toolbar showing behavioral models, or functional variants, of a given primary type. Each object also has parameter inputs which are specific to it.

Figure 3.4: Two-level tool bar design intended to reflect the structure and function of procedural objects. In this example, the secondary tool bar shows all behavioral objects whose output type is a Point set (all are colored green).

categories of modeling, animation, characters, dynamics and rendering. In LUNA, a two-level system is introduced. The primary tool bar represents the *structural* objects of discrete geometry. These include: points, lines, curves, surfaces, images and video. Selection of a geometry type at level one exposes a set of objects in the secondary tool bar. The secondary bar shows all the different *behavioral* choices available for a given primary type. These objects are interchangeable, such that any behavior which outputs Points can be input to any node accepting points. For example, Particles, Fluids, Spiroids, Brownian, and Flocking all generate unique behaviors for a time-evolving point set. Any of these may be input into other objects which accept points as input (e.g. TimeCurves). This ability to quickly interchange behaviors is a major advantage in creative explorations.

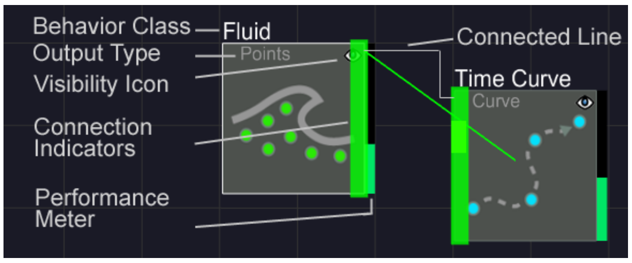


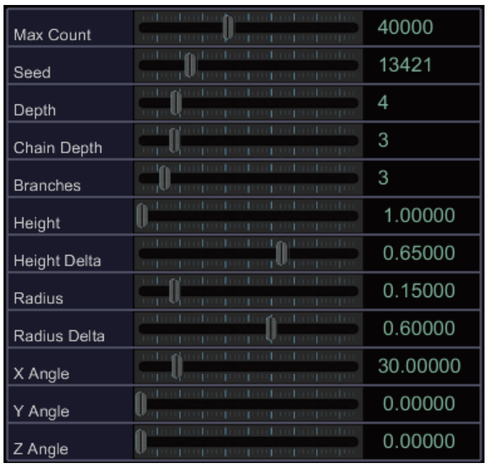
Figure 3.5: Objects have a base behaviour, printed above the icon, and an output type, shown inside it. Smart connection allows the user to drag a line from one object to another without precisely touching an input tab (the green line goes from object to object). Also shown are vertical performance bars. A visibility icon (eye), allows the user to see intermediate results.

3.2.6 Smart Connections

Typically, in order to connect two nodes, it is necessary to know the type and meaning of arguments to both. In many cases, however, there is only one combination of input types possible. For example, a generative terrain object may only require a two-dimensional image representing height. We distinguish between primary, required, inputs and secondary optional ones. When connecting objects, LUNA detects the semantics of incoming objects and, wherever possible, directly connects these objects without having to specify a specific input and output tab (Figure 3.5). With a single click-drag motion, it is possible to quickly connect many objects in this way. In the future, for more refined control, the user might hover over an input tab to specify a particular input.

3.2.7 Property Panel

To further enable live performance, and to provide precise control over node parameters, a property panel is introduced with an aesthetic based on sound mixing. While the interface appearance is reconfigurable, this design encourages large sliders with clear labels over numeric entry. The property panel is optionally accessed by clicking on a graph node, a top-down approach that places emphasis on the graph, where high level decisions are made first, rather than on more exacting parameter changes that can be made later. While only sliders are available currently, in the future the panel may be extended to support other controllers.



Max Count	40000
Seed	13421
Depth	4
Chain Depth	3
Branches	3
Height	1.00000
Height Delta	0.65000
Radius	0.15000
Radius Delta	0.60000
X Angle	30.00000
Y Angle	0.00000
Z Angle	0.00000

Figure 3.6: Property panel in LUNA with parameter sliders shown for the Tree object.

3.2.8 Required Defaults

While implementing LUNA, it was realized that a drawback of some systems is that inputs often must be exactly defined in order to produce any output. One design goal, reflected in the design dimension of image/idea, is that complex operations should not be prohibitively difficult to specify to get a minimal result. Therefore, every node in LUNA is required to have default behavior that produces output as soon as all required inputs are connected. This often means that a generative node must be capable of resampling or reducing the input size to a meaningful level to avoid stalling the system. Creating any node, and connecting its visible input tabs, produces and immediate result.

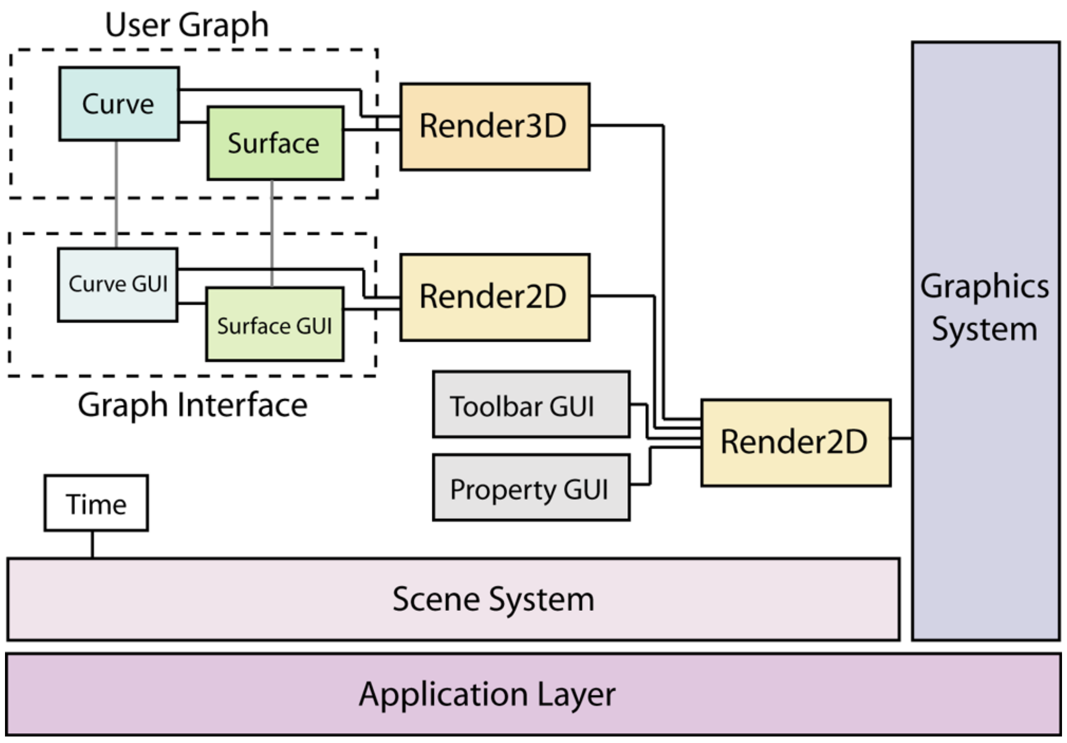


Figure 3.7

3.3 Graphical Language Implementation

The structure of LUNA consist of a procedural modeling language and a graphical user interface combined into a single graph architecture. The scene graph is a directed graph which contains objects for behavior, geometry, and interface. Unlike traditional model-view-controller designs, in which the scene DAGs is kept separate from the interface graph, LUNA combines these objects into a single graph and uses rich connection semantics to keep them organized. Thus, multiple sub-graphs may overlap in the same graph. The *primary graph* is a set of input/output connections that describe how to render both GUI elements and three-dimensional objects, see Figure 3.7. This allows the graphical interface and procedural models to be rendered by the rendering sub-system in a uniform way. Of course, these objects must be handled differently during rendering, so any object can indicate the style of evaluation it requires: 1) *self-draw*, used by GUIs to draw themselves and their contents, 2) *proxy*, used by geometry to request that the renderer build vertex buffers on the GPU, and 3) *resources*, used by images and materials to request persistent data available to multiple objects. The method of evaluation differs for procedural models and graphical interface components. In general, the generation of complex procedural models are discussed in detail in Chapter 4, while this chapter focuses on the evaluation on GUI elements, that is the visual and interactive combination two and three-dimensional components in the system.

The overall architecture of LUNA consists of multiple sub-systems, including graphics, video, networking, and input. These sub-systems are responsible for hardware or device-level interactions with the scene graph, and each may communicate with the graph in different ways. The graphics system, for example, renders any desired object and keeps track of graphics state and GUI buffers for performance. The renderer traverses the graph, conceptually, from right-to-left starting from the top level 2D desktop GUI node, Render2D, which covers the workspace. This node may have multiple Render2D and Render3D nodes connected to it as well, allowing for nested views and three-dimensional windows. The input sub-system handles user interface events, and traverses the same scene graph from root to leaf, but using a different set of functions for event handling.

Time, i.e. motion, is handled by the application by inserting a global Time node into the graph. The time node is unique in that it traverses the graph from the opposite direction, triggering any behavioral nodes which accept time as an input. The causes notifications to travel *up* the graph, informing any objects whose geometries or interfaces must be updated on the next render frame. With this overall design, it is possible to combine multiple semantics into a single graph architecture, simplify the need of maintaining separate graphs for model, view and control.[1]

The idea of rendering in two and three-dimensions is implemented using nodes also found in the scene graph. While a single graphics sub-system handles actual rendering, the presence of rendering nodes allows the graph to invoke different coordinate spaces, views, and windows as the graph is traversed during rendering. This allows both the graphical interface and the user output to be generated by the same rendering evaluation model. In Figure 3.7, for example, the Render3D node prepares the graphics system to generate geometry for the Curves and Surfaces connected to it, while the Render2D

[1]While model, view and control are still present, the objects which represent these different facets of the application are combined in the same graph through the diverse functionality present in each object.

node prepares a local two-dimensional canvas on which the GUI boxes representing these objects are drawn. If the user interacts with the three-dimensional curve itself (moving a vertex), this event passes down the Render3D portion of the graph from the root, while if the user moves the two-dimensional box representing the curve, this event passes down the Render2D portion of the graph.

3.4 Interaction Study and Evaluation

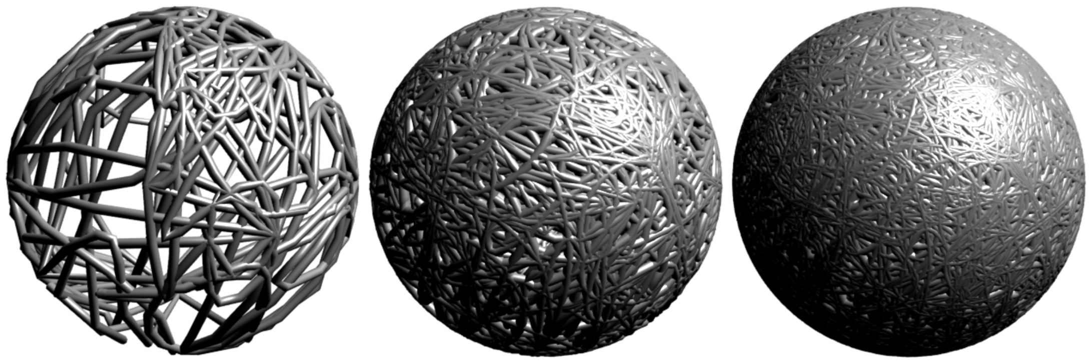
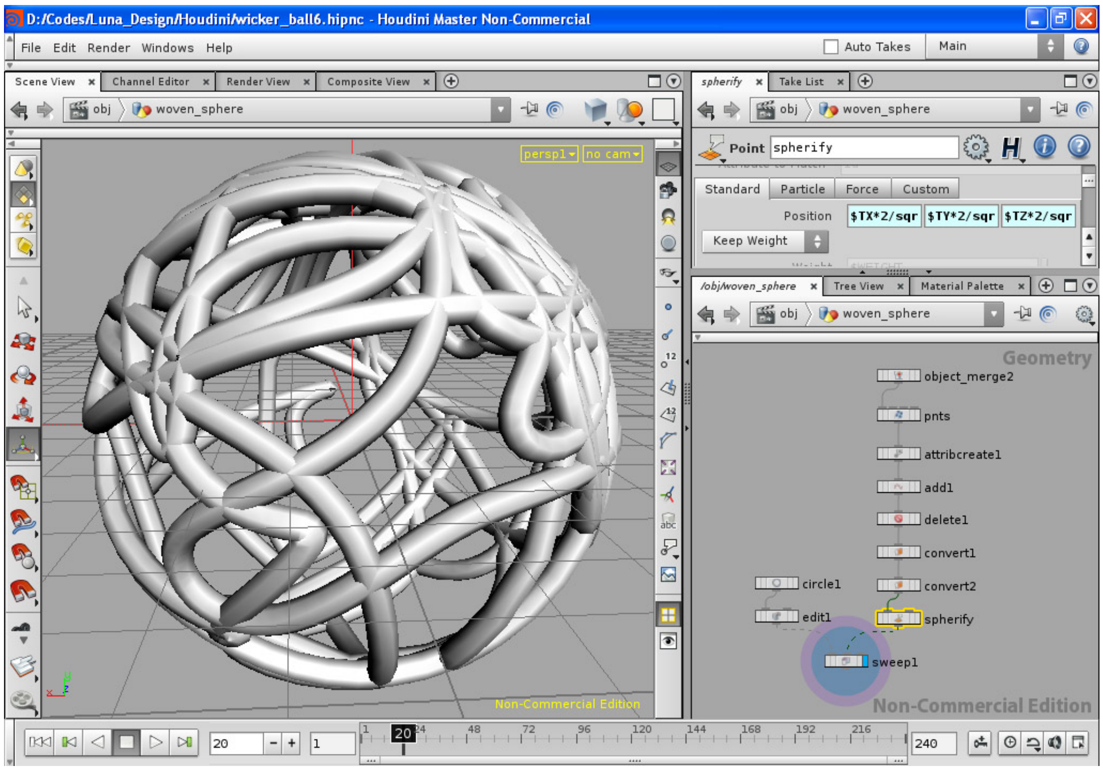


Figure 3.8: Reference object for interface testing, a woven sphere, is described in detail in Appendix A.

To demonstrate the usability of LUNA in a practical context, a series of interaction studies were performed by the author. Although there are few procedural data flow languages with similar capabilities, these comparisons are done against Houdini using a procedural reference object. As there are no common reference models for interactions with procedural data flow systems, a novel object is introduced here. The object used for testing is a woven sphere, shown in Figure 3.8, which consists of a simple system of moving particles sampled to generate Bezier curves, normalized to a sphere, and lofted to create a set of tubes lying on the sphere. This model is described in further detail in Appendix A. The woven sphere is a suitable object for interface testing because it represents several specific steps which are unique to procedural systems. It includes an animated system, intermediate objects of different types (points, curves and meshes), and steps which must be introduced at the correct stages in the model graph to produce the correct output. In addition, this object is uniquely procedural, and cannot be constructed using traditional modeling techniques.



Point spherify

Position $TX*2/sqr $TY*2/sqr $TZ*2/sqr

Figure 3.9: Reference model created in the Houdini interface. The model uses a Particle SOP for initial point locations, a Point SOP /w a normalize expression to generate curves and map these curves to the surface of a sphere, a Circle SOP (set to 'polygons') to define the loft cross-section, and a Sweep SOP to build swept surfaces from these curves. The Copy Stamping method is used to generate different random instances of curves, with point inputs scaled to (0,0,0) so that the curves are not translated in space.

To perform interface testing, the author constructed the reference model in both Luna and Houdini [2] A log was kept of the challenges and problems encountered during model construction, as well as a record of the time at each phase, which are reported in Appendix B. Figure 3.9 shows the reference model being constructed in Houdini. In general, the model requires knowledge of points, curves, and surfaces as it is being constructed. However, these transformations in media type are not entirely clear in the Houdini graph. Other interactions were also found to be difficult in Houdini. A certain stages, detailed in Appendix B, it was necessary to know a particular, specific feature of a node in order to correctly produce the output type needed for the next step. For example, to generate the loft surfaces required by the model (tubes), the circle object in Houdini must be changed from "primitive" output to a "polygon" output in order to generate swept surfaces from cross-sections, otherwise the output remains blank. This knowledge is generally found by referring to the reference documentation for particular objects, which further detracts from the work flow. Overall, four hours were required by the author to create the reference model in Houdini.

There are several ways to create this same model in LUNA. One may work from right-to-left, imagining the form of the final result and filling in pre-requisite nodes that are needed to activate it. Or, one may work from left-to-right, starting from the basic structure of the model and building it into a final form. This latter approach is taken in the following examples. Figure 3.10a shows the first of these steps, selecting Points type

[2]These interactions were the author's first experiences with Houdini, so no prior knowledge was assumed.

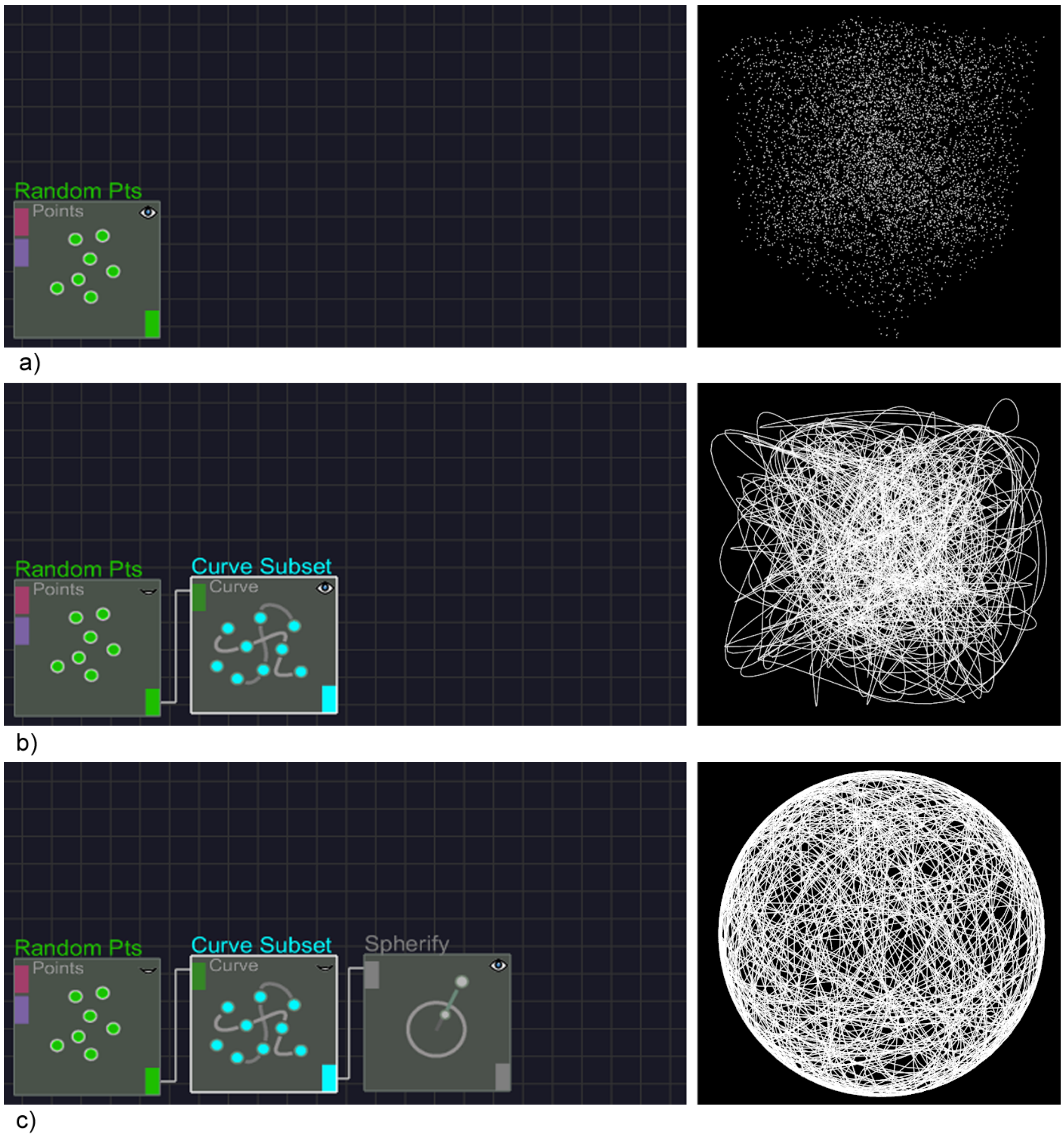


Figure 3.10: Steps in creating the reference model in LUNA include a) creating a set of Random Points as starting positions, b) producing curves by randomly sampling subsets of the input points using the Subset Curves object, and c) normalizing the curves to a sphere using a Spherify modifier.

from the main tool bar and dropping a Random Points node onto the canvas, producing the result shown.

The second step in this model, detailed in Appendix A, involves selection of several random subsets of these points to be used as the CV control points for Bezier curves. This is accomplished by dropping a Curve Subset object, and connecting the Random Points into it, shown in Figure 3.10b. This produces a set of curves which randomly fill the space occupied by the points. Graph connections are made using a single click-drag motion from input to output. The third step is to map these curves onto a sphere. This is accomplish in LUNA using the Spherify modifier, Figure 3.10c, which transforms any object onto a sphere (by normalizing its points), in this case the curves are spherified. In general, modifiers in LUNA are able to operate on any object, and their output takes on the type of the input connected to them, thus the output of Spherify in this case is another set of curves.

The final step in this example involves constructing swept surfaces from these curves. The Loft object in LUNA performs this function, and takes two curves as required inputs. The first specifies the cross-sectional shape of the surface, in this case a circle is connected at the top. The second input is the curve set which represents multiple paths along which this section will be swept. As the Spherify modifier outputs a set of curves, this is used to express the paths we wish to loft, producing the final results shown in Figure 3.11a. Using these nodes, the reference model can be created in this interface with literally ten clicks (5 to select objects, 5 to drop them), and four click-drag motions (connecting each node to the next).

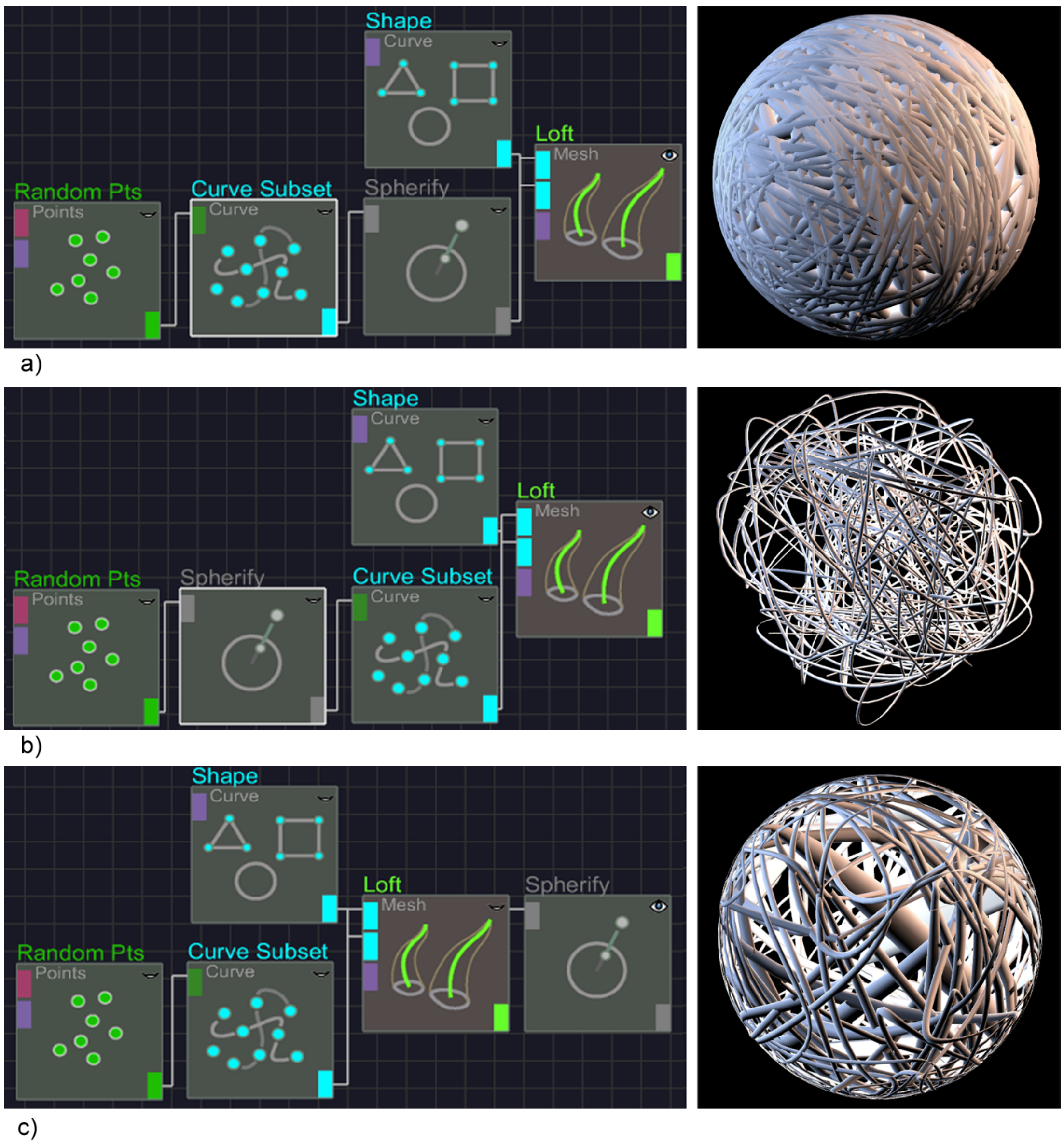


Figure 3.11: The final step in creating the reference model, a) connects the spherified curves to a Loft object to generate swept surfaces. Alternative models are easily explored by changing the order in which these high level actions are performed, such as b) where the spherify operation acts directly on points, and c) where spherify is applied to a mesh object.

Admittedly there are several advantage given to LUNA here. First, many of the nodes used here do not exist in Houdini. For example, the Spherify operation is not found in Houdini by default, and it was necessary (see Appendix B) to write an expression to perform the spherify operation in Houdini, which reduces performance. In addition, the author's familiarity with LUNA suggests that the choice of nodes, and their order of operation, may not be obvious to a new user of the language. The issue of language familiarity, however, is partly addressed by the ease with which one can discover different models in LUNA using very simple interactions. As the Spherify node operates any geometry type, it can be connected at different stages in the graph. In building the reference model, for example, the user may have thought to spherify the points *before* generating curves. The result of this is shown in Figure 3.11b. Since the Subset Curve object uses the points as control vertices in Bezier curves, the resulting curves themselves may penerate into or protrude away from the sphere surface, producing an incorrect model (relative to the reference goal). Using LUNA, the user can transition from this model to the correct one with only three click-drag motions. Thus, a unique contribution of LUNA is the ease with which new models can be generated through its interface.

This basic interface, which favours combinatorial rearrangement over detailed parameter controls, allows one to rapidly explore the power of LUNA as a language for procedural modeling. By connecting the spherify operation to the outcome of the loft node, on gets a different result altogether (Figure 3.11c). This causes the verticies of the tube meshes to be spherified, which distorts their cross-sectional geometry and volume in interesting ways. Modifiers may be connected to other modifiers, providing a limitless source of possible outcomes. While languages such as Houdini focus on the detailed control of each node - which is also possible in LUNA using the property panel - LUNA does not require this kind of detailed work flow in order create valid output, or to explore new results. This makes LUNA particularly suited to its original goal of serving creative exploration by media artists.

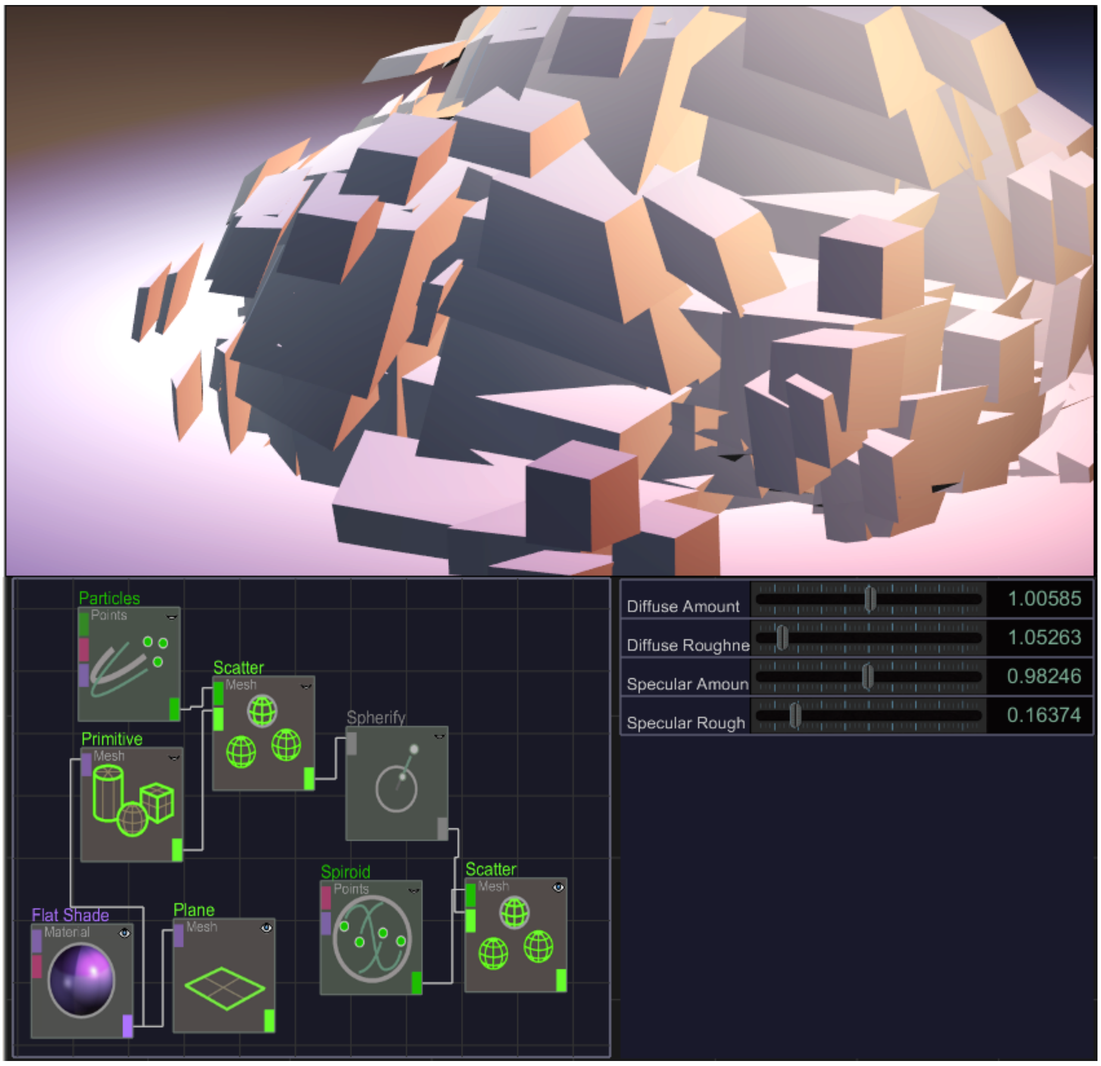
This brief demonstration was intended to show how one can quickly create novel objects in LUNA. However, a more thorough interface study could be designed to reveal more detailed results. To create a fair analysis, both Houdini and Luna would be provided with the same object set by implementing custom nodes in Houdini. Although it is difficult to find a task that represents overall creative exploration, one could ask users to create *any* object that meets certain criteria, such as incorporating points, curves and surfaces together. A detailed study might also reveal how these systems balance expressive power and flexibility. These are future areas for possible examination. In any case, there are few generic visual data flow languages for procedural modeling[3].

Prior to such user-based interaction studies we can evaluation the LUNA interface according to Green's criteria for visual data flow langauges defined earlier. Early decisions made in LUNA can be easily modified by reconnecting objects, so its level of *commitment* is lower than that of Houdini. *Progressive evaluation* is supported in sev-

[3]Others include Xfrog and Groboto, but these are specialized systems. Artists tools like Soundium are generative, but do not have a procedural aspect. See Chapter 2 for a comparison of systems.

eral ways, by allowing the user to see intermediate results (using the 'eye' icon) and by giving direct feedback on parametric changes, features also available in Houdini, although LUNA's performance is better overall in this regard (see Chapter 4). The ability to say what you want, *expressiveness*, can be interpreted in two ways. First, the power to say what you want is potentially higher in Houdini as it is a more developed commercial application, with support for more complex objects. However, expressiveness can also mean the ease with which you can say what you want, and in that respect LUNA may provide a better experience as its interaction produces more immediate results. Regarding *viscosity* (resistance to change), LUNA was intentionally design to make it very easy to modify objects and ideas interactively, and the use of color to denote media type, large iconic representations of tasks, and overall layout give it a level of *visibility* which makes it easier to see what you are making in comparison to Houdini. LUNA is thus presented as a modern, dynamic, interactive alternative to current commercial procedural modeling systems.

A more complex interface example is shown in Figure 3.12. In this example, two particle systems and a cube primitive are used to generate complex arrangements using two Scatter nodes. A material node, Flat Shade, is used to change the visual appearance of both the overall object and the ground plane. The parameters to this material are shown in the Property panel to the right, with the results shown above.



Diffuse Amount 1.00585
Diffuse Roughne 1.05263
Specular Amoun 0.98246
Specular Rough 0.16374

Figure 3.12: Example of a complicated graph in LUNA, incorporating multiple surface objects, modifiers, and materials.

3.5 Project Results

3.5.1 *The Bones of Maria*, Organic Art

To explore the expressive range of LUNA several creative, interdisciplinary projects were developed using it. These include works with styles in digital and media arts. *The*

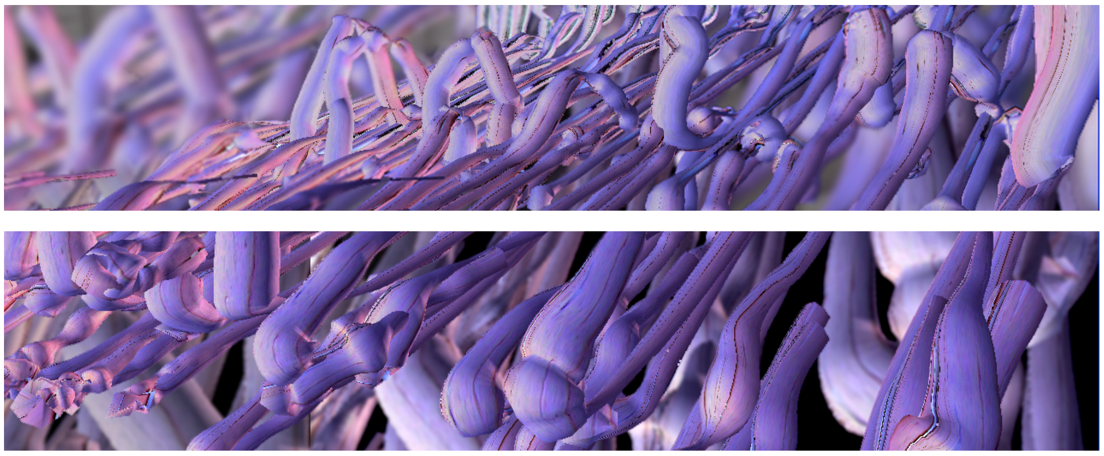


Figure 3.13: *The Bones of Maria.* Generative art. Exhibit online at The Cultor, IT. 2010.

Bones of Maria, shown in Figure 3.13, is a generative art project using a smoothed particle hydrodynamic (SPH) fluid system and time analysis to create organic, three-dimensional, textured forms. These were shown in an online exhibition at The Cultor, an arts and culture organization based in Torino, Italy. [4]

3.5.2 *Presence*, Interactive Art

Presence is an interactive, site-based installation exhibited at the University of California Santa Barbara's Davidson Library in 2009, Figure 3.14. A collaboration between R. Hoetzlein, Dennis Adderton, and Jeff Elings, *Presence* consists of a high resolution, virtual 360 degree panoramic photographs displayed on six screen. A camera detects the motion of passing library patrons and rotates the panorama as they walk by.

[4]http://www.cultor.it/Pinacoteca2.html

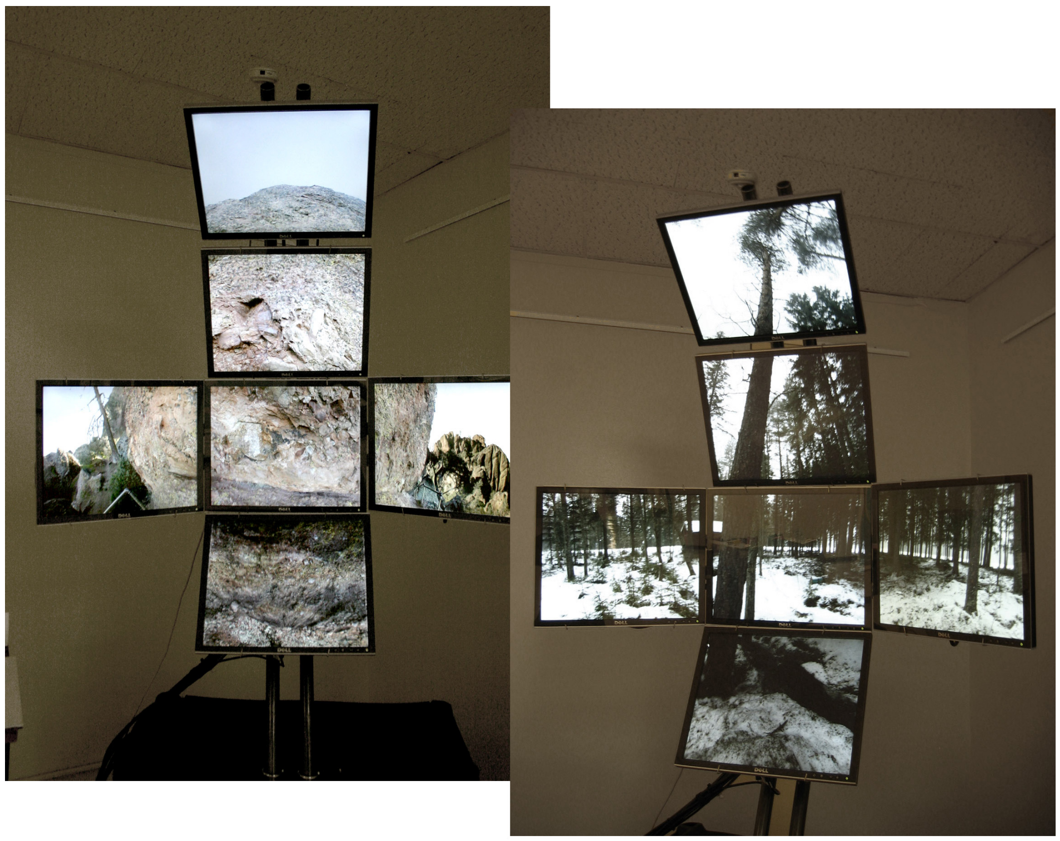


Figure 3.14: *Presence.* Interactive artwork exhibited at the University of California Davidson Library, 2009.

3.5.3 *Blocks*, Game Design

An experimental game project called *Blocks* was created with Mark Zifchock and Abraham Connelly using LUNA. *Blocks* is a universe of cubes where each has a unique function. Some blocks act like water flows, moving at right angles into lower spaces. Others are used to build terrain, bridges, and barriers. Logic blocks introduce computational and, or and not gates expressed in cubes, while special blocks allow for teleportation and wireless signaling. While *Blocks* was implemented using a custom

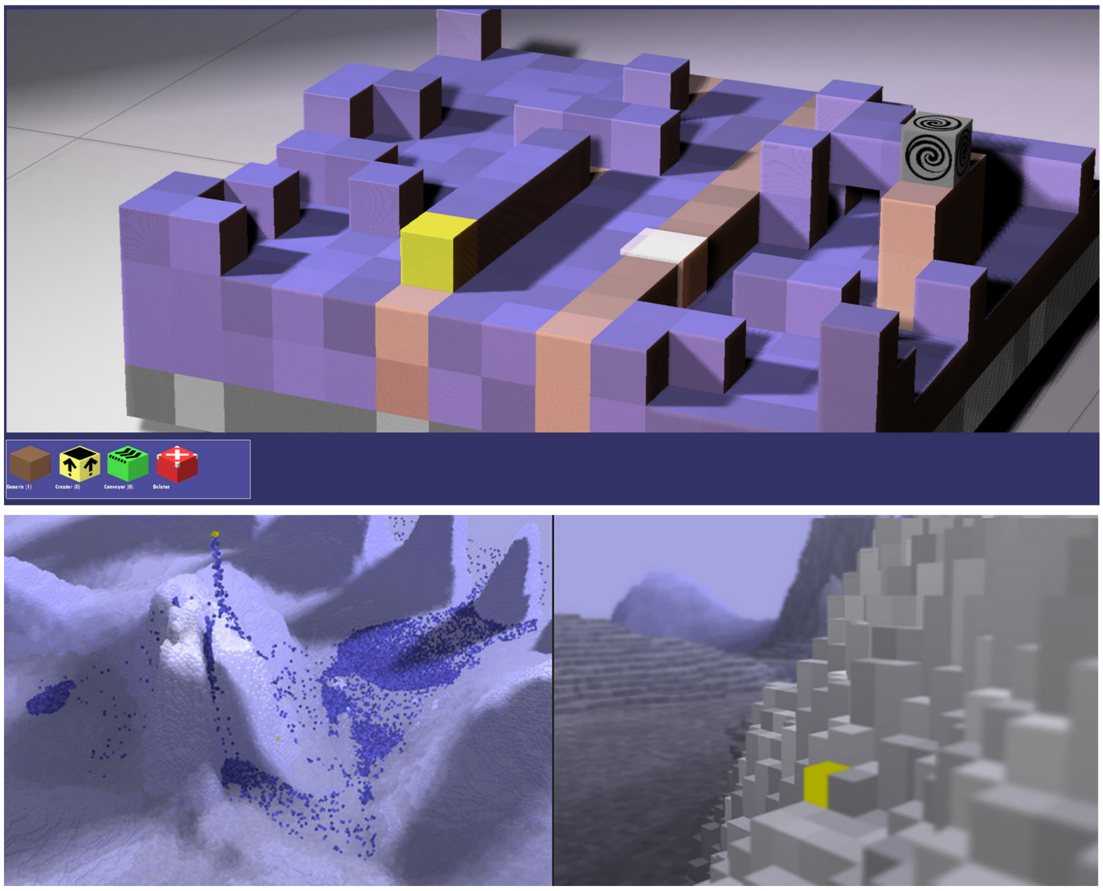


Figure 3.15: Blocks. Game design by Mark Zifchock and Rama Hoetzlein. Created using LUNA.

node in LUNA for the block-world simulation, the blocks universe may consists of millions of cubes, rendered using texture and Cg shader nodes in LUNA. A unique aspect of Blocks is its own graphical interface, which includes a custom tool bar for selecting block types. This interface was implemented using the same GUI graph architecture used for LUNA itself, and both GUI elements (Blocks tool bar and LUNA's object tool bars) are rendered in the same graph together.

3.5.4 Procedural Modeling

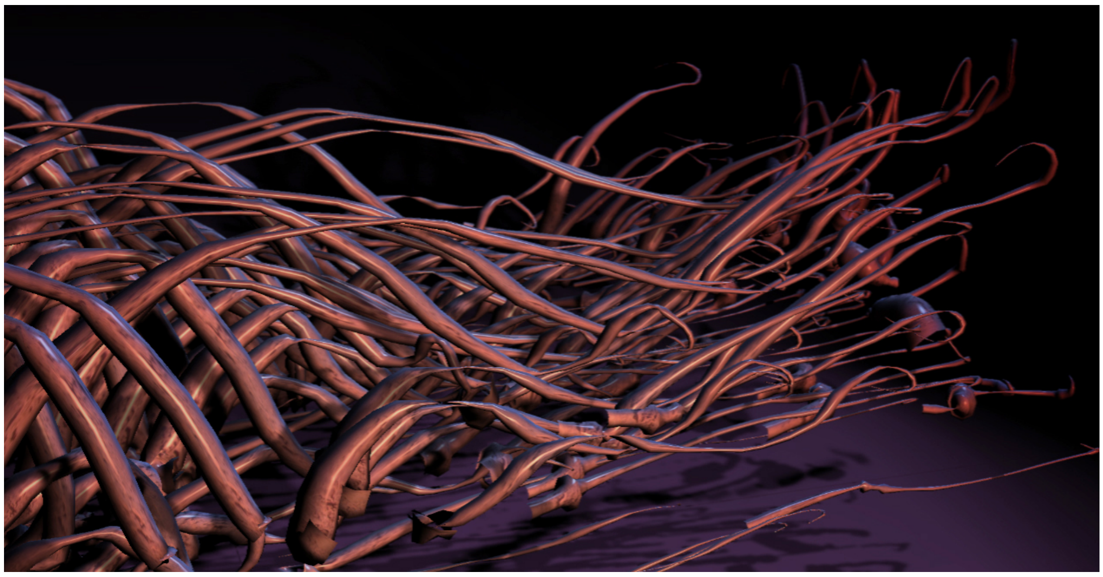


Figure 3.16: Loft surface with high quality rendering created in LUNA using shadows and depth-of-field

Other experiments with procedural modeling in LUNA are shown in Figures 3.16 and on the next page. In many cases, these images were constructed, generated and rendered in a matter of seconds. The dynamic nature of the interface makes it very easy to quickly replace an object with another node of a similar type. Thus, an artist is able to rapidly experiment with different dynamic behaviors as this sequence shows.

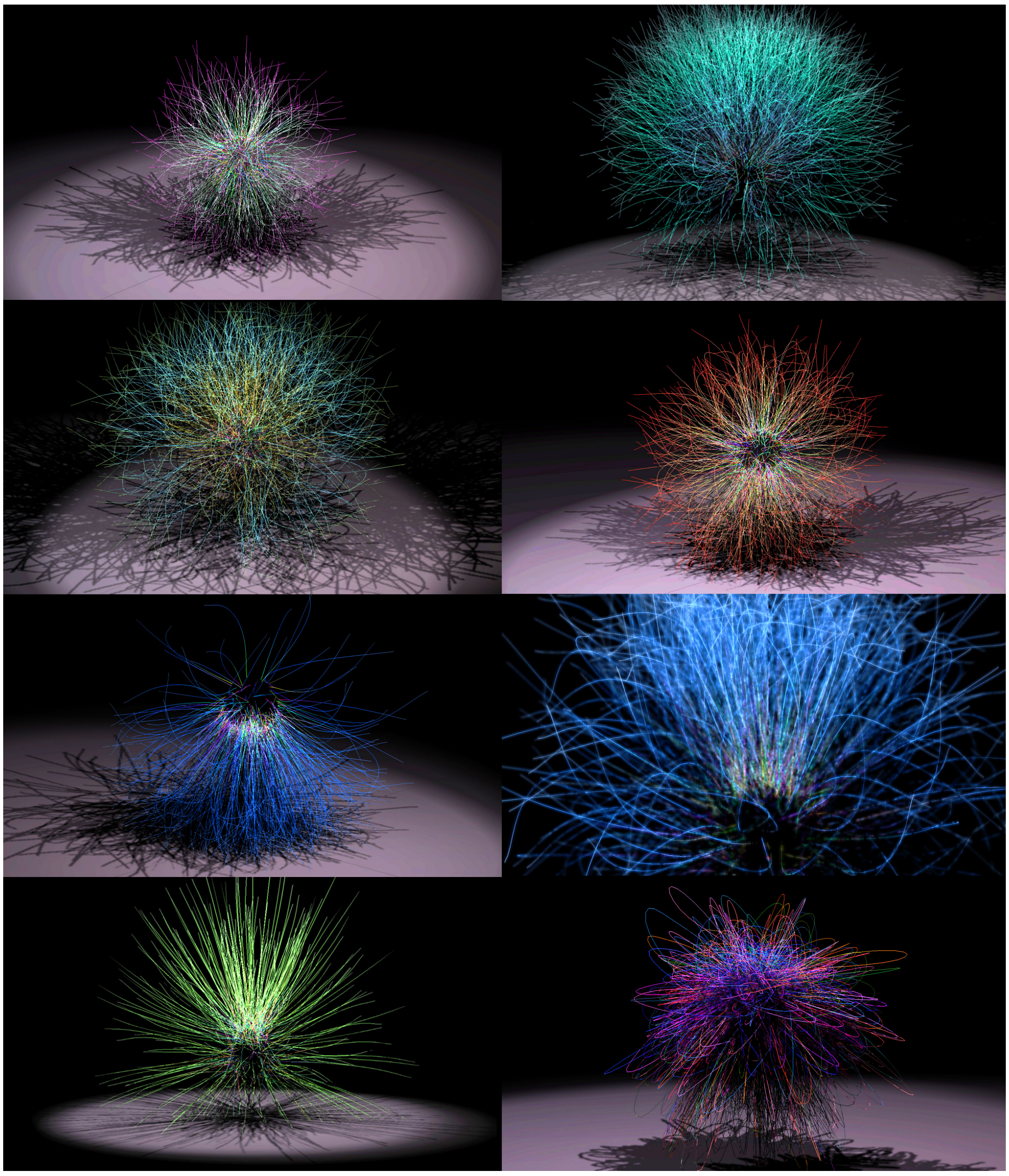


Figure 3.17

3.5.5 Biological Modeling

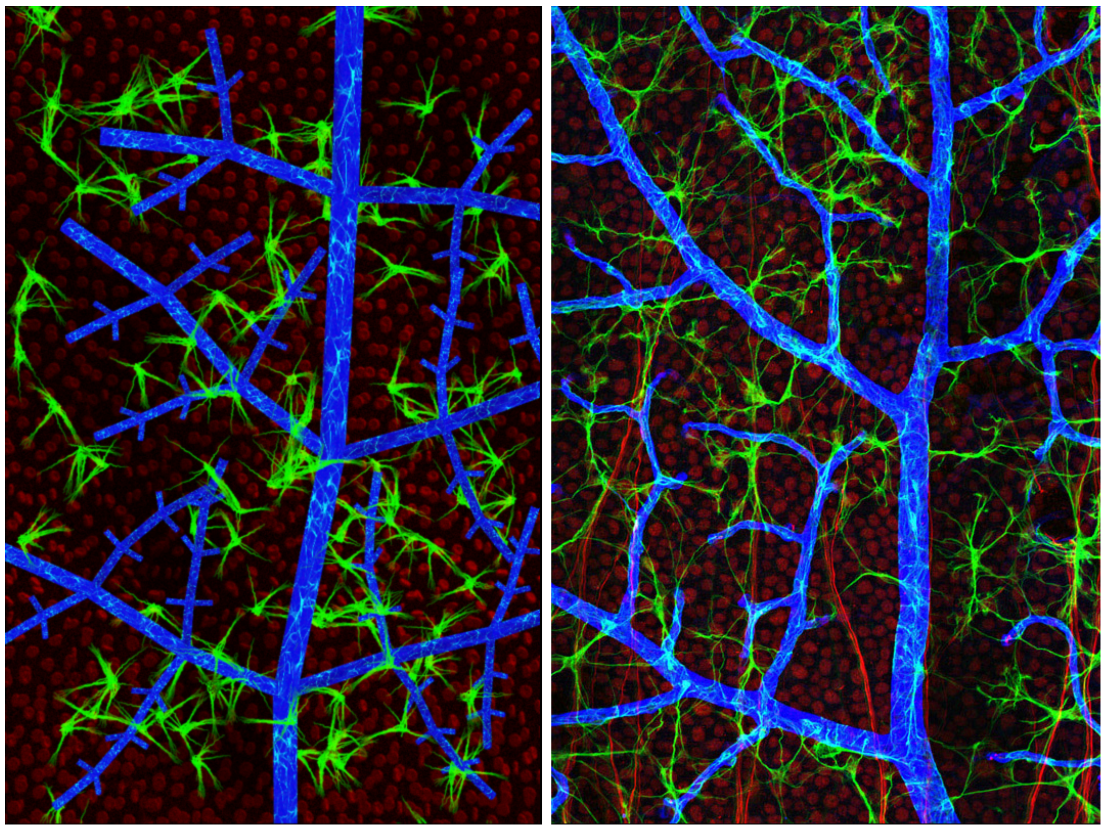


Figure 3.18: Synthetic rendering created in LUNA (left) compared to real astrocyte imagery of a rabbit retina. Blood vessels (blue) were modeled using a tree object, while astrocytes (green) were modeled as Bezier curves using a physically based spring-system with added noise.

These examples demonstrate that LUNA is able to achieve many distinct creative styles. In addition to creative projects, a scientific collaboration with Mock, Brian and Steve Fischer explored the use of LUNA to create synthetic models of real world biology. Figure 3.18 shows an astrocyte image of a rabbit retina. Blue blood vessels are shown next to green astrocyte cells. The synthetic model, at left, is the first example of a procedural model whose goal is to visually match the structure of a micro-cellular network whose biological organization is unknown. The motivation for this on-going project is to reproduce images sufficiently similar to real world microscopic slides that vision algorithms used to detect astrocyte cell centers and geometry could be evaluated on synthetic data with known ground truth.

3.6 Conclusions

A visual data flow language, LUNA, is presented for the creative exploration of procedural models using an intuitive, minimalist interface. Its design follows from a combinatorial approach influenced by a series of design goals established from creative dimensions that are of particular interest to media artists. Experiments with the interface show that it is possible to rapidly explore interesting, alternative designs by quickly connecting and arranging high level tiles representing procedural objects. The graphical interface in LUNA enables this by making specific use of layout, color, and connection behavior to meet these design goals. In addition, LUNA itself is capable of many different creative styles, including procedural and organic modeling, interactive art using video input, game design, and high quality rendering with deferred shading.

Although any visual data flow language requires some symbolic interaction, the features of LUNA are constructed to meet the needs to artists, allowing them to focus on the task of exploring creative possibilities. The dimension of *modality*, for example, is embedded in the two-level tool design and in the currently available media types, while the dimensions of *dynamics* and *structure* are embedded in the temporal and geometric behavior of objects as they are manipulated by the graph. The critical features of LUNA, established by these creative dimensions, are thus incorporated into both the interface and the structure of the language.

While LUNA presents interesting possibilities, it is a new systems which would benefit from further development and testing. Currently there are 29 nodes (as of Oct 2010) available in LUNA. One future goal is to expand this vocabulary to include audio, video, and device interaction. In the area of interface design, the issue of the temporality is not yet addressed as all nodes perform their actions continuously, making it difficult to script different behaviors over time. This suggest an interactive timeline in addition to the canvas area. Specific areas, such as the types of parameter controls in the property panel (currently only sliders are present), also deserves more attention. Finally, user studies may make it possible to establish real differences in expressive power between LUNA and other languages.

LUNA is presented here as a novel interface for the construction of dynamic, creative objects with interactive feedback, with specific examples showing how artists can use this visual language to quickly create new and interesting models. Six creative dimensions of interest to artists contribute to both the interface and structure of the language, result in a system which is intentionally designed to meet the needs of media artists, with the hope of unifying many of the diverse practices and techniques found in the digital visual arts.

Bibliography

[Bandyopadhyay et al., 2001] Bandyopadhyay, D., Raskar, R., and Fuchs, H. (2001). Dynamic shader lamps: Painting on movable objects. In *ISAR '01: Proceedings of the IEEE and ACM International Symposium on Augmented Reality (ISAR'01)*, page 207, Washington, DC, USA. IEEE Computer Society.

[Bannink, 2009] Bannink, P. (2009). Houdini in a games pipeline. In *SIGGRAPH '09: SIGGRAPH 2009: Talks*, pages 1–1, New York, NY, USA. ACM.

[Blinn and Newell, 1976] Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547.

[Conway and Pausch, 1997] Conway, M. J. and Pausch, R. (1997). Alice: easy to learn interactive 3d graphics. *SIGGRAPH Comput. Graph.*, 31(3):58–59.

[DeBry et al., 2002] DeBry, D., Gibbs, J., Petty, D. D., and Robins, N. (2002). Painting and rendering textures on unparameterized models. *ACM Trans. Graph.*, 21(3):763–768.

[Doerr and Kuester, 2010] Doerr, K.-U. and Kuester, F. (2010). Cglx: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints).

[Green and Petre, 1996] Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174.

[Haeberli, 1988] Haeberli, P. E. (1988). Conman: a visual programming language for interactive graphics. *SIGGRAPH Comput. Graph.*, 22(4):103–111.

[Hoetzlein and Adderton, 2009] Hoetzlein, R. C. and Adderton, D. (2009). Mint/vfx - a high-performance computing framework for interactive multimedia.

[IBM, 1999] IBM (1999). Ibm visualization data explorer.

[Jacucci et al., 2005] Jacucci, G., Oulasvirta, A., Salovaara, A., Psik, T., and Wagner, I. (2005). Augmented reality painting and collage: Evaluating tangible interaction in a field study. In *Human-Computer Interaction, INTERACT 2005*, pages 43–56, Heidelberg, Berlin. Springer.

[Jones and Nevile, 2005] Jones, R. and Nevile, B. (2005). Creating visual music in jitter: Approaches and techniques. *Comput. Music J.*, 29(4):55–70.

[Lawrence, 2004] Lawrence, J. (2004). A painting interface for interactive surface deformations. *Graph. Models*, 66(6):418–438.

[Manovich, 2001] Manovich, L. (2001). *The Language of New Media (Leonardo Books)*. The MIT Press.

[Meso, 1998] Meso (1998). Vvvv: A multipurpose toolkit. http://www.meso.net/vvvv, accessed June 2010.

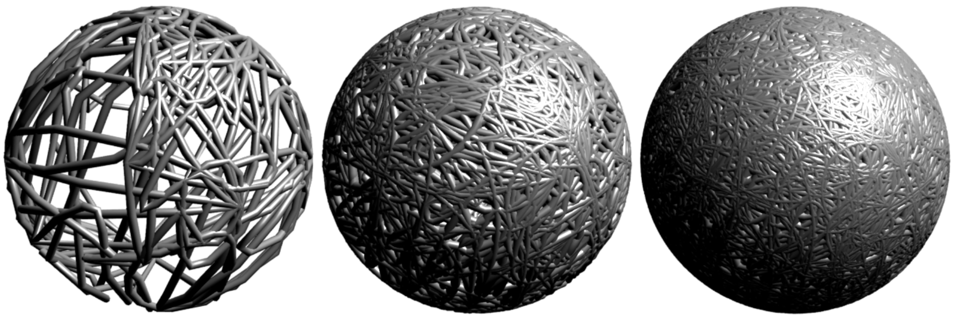
[Resnick et al., 2009] Resnick, M., Maloney, J., and Monroy-Hernandez, A. (2009). Scratch: Programming for all. *Commun. ACM*, 52(11).

[Ryokai et al., 2004] Ryokai, K., Marti, S., and Ishii, H. (2004). I/o brush: drawing with everyday objects as ink. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 303–310, New York, NY, USA. ACM.

[Sutherland, 1988] Sutherland, I. E. (1988). Sketchpad a man-machine graphical communication system. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 507–524, New York, NY, USA. ACM.

Appendix A

Reference Model



Model	P	C	K	V	U	Verts	Tris
Low res	50	25	8	28	8	5,600	9,450
Med res	150	100	8	42	12	50,400	90,200
High res	250	200	8	56	16	179,200	330,000

Figure A.1: Woven sphere reference model with parameter values for low, medium and high resolution models.

The woven sphere is a procedural model defined as follows. Input consists of a particle system with P points, generated randomly in a box from (-1,-1,-1) to (1,1,1) and moving with a uniform velocity of 0.0025 in a random direction (arbitrary units, time step is 1.0). As the points animate, they reflect off boundaries to remain inside the initial volume. Described in LUNA notation:

PSYS$_{points}$ (P, init_min $< -1, -1, -1 >$, init_max $< 1, 1, 1 >$, init_vel $< 0.0025 >$)

From these points, random subsets are selected in groups of K to become the CV control keys of C Bézier spline curves. The Bézier curves are sampled to a resolution of V total sample vertices per curve. The curve order is 3 (cubic). The function is:

SUBSET$_{curves}$ (POINTS$_{points}$, num_keys K, num_curves C, num_samples V)

This generates C curves with K keys and V sampled points in each. These curves are then spherified to a unit sphere (radius 1) by normalizing the points in each curve. Note that it is incorrect to normalize the CV keys as the resulting curve may still penetrate the sphere. The spherify function should operate on the final sampled points to guarantee the sampled curve lies on the sphere. In procedural modeling terms, the spherify function takes any geometric object (points, curves, meshes) and normalizes its verticies. It is a typeless function defined by p' $=|p|$:

SPHERIFY (OBJ)

Finally, loft surfaces are generated by sweeping a circle along the curves. A circle of radius 0.025, sampled with U verticies, is used as the cross-section. The paths are the spherified curves of the previous step. The loft surface has a cylindrical topology with only triangular faces, and no end caps. This produces a total of U*V verticies per loft, and C*U*V verticies for the entire woven sphere object, with 2(U-1)(V-1) triangles per loft, and 2(U-1)(V-1)C triangles for the whole object.

CIRCLE_{curve} (samples U)

LOFT_{mesh} (PATH_{curves}, SHAPE_{curve})

The total function is:

LOFT_{mesh} (SPHERIFY(SUBSET_{curves} (PSYS_{points}(P, init_vol, init_vel), K, C, V)), CIRCLE_{curve} (U))

Parameter values and sample representations for the low, medium and high-res models used in our tests can be found in Figure 3.8. For render performance testing in real-time systems, it should be rendered at 1024x768 using a single Phong light source and no shadows or anti-aliasing. When reporting results, ideally evaluation should be separated from render time. Animation of the underlying particle system causes the curves to gradually morph along the sphere surface.

Appendix B

Houdini Interaction Study

Results of the interface test in Houdini for the reference model are shown here. No prior knowledge of Houdini is assumed, although the author is familiar with procedural modeling concepts. In total, it took around 4 hours to create this model in Houdini.

Elps Time	Task Time	Description
0:02	2 min	Figure out how to create objects (must press enter)
0:08	6 min	Cannot use Source on Particles (only Fluids)
0:13	5 min	Source for Geometry used to emit particles. Needed to explore help docs to find that Emission type parameter can be set to Volume.
0:44	31 min	Trying to figure out how to build a curve from particles. No obvious function to generate curve from points. Found an online forum: "moving curves points to the particle locations using a Point SOP"
0:59	15 min	Time spent figuring out how to connect object subgraphs to one another. Incorrect assumption about how Houdini works.
1:36	37 min	Output: Now produces points moving on surface of a sphere. Created a point SOP to shrink points to a sphere. Learned that top-level graphs are not flow networks, but heirarchy networks. So it is not possible to connect object sub-graphs. Must copy nodes into an object's flow graph.

1:51	15 min	Moved the 'spherify' node after the curve input, to properly match reference model. Attempting to use the Copy Stamping method to generate many curve instances, after further reading of documentation.
2:06	15 min	Discovery that graphs in Houdini compute entire objects first. I should not generate multiple curves, but generate a complete curve-loft, then replicate.
2:18	12 min	Skin Output of the Sweep SOP is not working. Not sure why.
2:36	18 min	Found that Circle primitive type must be changed from Primitive to Polygon in order to generate swept surfaces.
2:56	20 min	Curve points are not spherified, only control keys. To spherify curve itself, necessary to add a Convert operator to make a Polyline.
3:01	5 min	Output: Now produces curves moving on surface of sphere. Determining relation between Level of Detail and number of points generated, as I cannot precisely control the curve sampling.
3:24	23 min	Found that instancing was incorrect because 'stamp' was not being used correctly. Took time to figure out it must be an expression of the form: point("particles", $PT +
3:34	10 min	Some time lost due to object path naming. Interface automatically inserts paths like "obj/group/particles/"
3:51	17 min	Output: Complete graph is working, with curves becoming loft tubes. Copy stamping is slow (expression parsing?), probably a better way to do this. Cannot stop it from translating curves to the particle locations.
3:54	3 min	Hack was used to solve Copy Stamping translation problem. Particles scaled to (0,0,0). This 0 point particle set used as input to the Copy to Points node.
3:54		Output: Produces results that match the reference model.