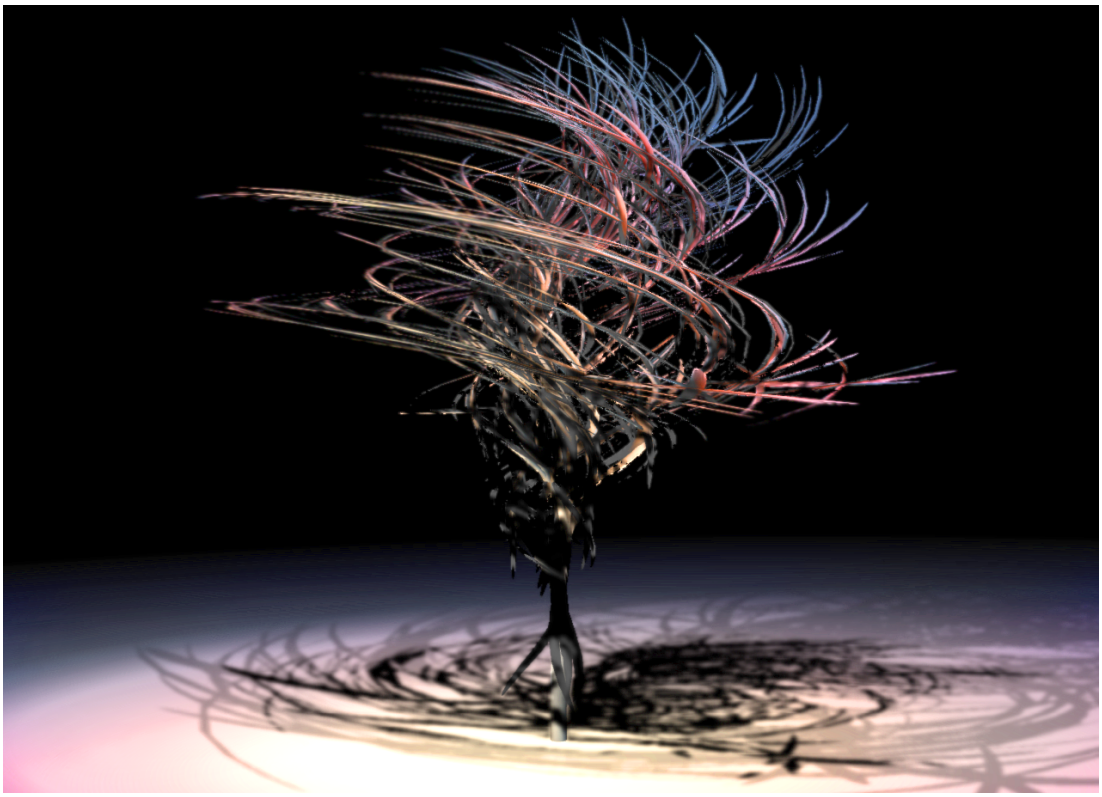
Chapter 4

Procedural Modeling of Complex Objects using the GPU



4.1 Introduction

Procedural modeling involves the use of functional specifications of objects to generate complex geometry from a relatively compact set of descriptors, allowing artists to build detailed models without manually specifying each element, with applications to organic modeling, natural scene generation, city layout, and building architecture. Early systems such as those by Lindenmayer and Stiny used grammatic rules to construct new geometric elements. A key feature of procedural models is that geometry is generated as needed by the grammar or language [Lindenmayer, 1968] [Stiny and Gips, 1971]. This may be contrast with *scene graphs*, which were historically used in systems such as IRIS Performer to group static geometric primitives for efficient hardware rendering of large scenes [Strauss, 1993] [Rohlf and Helman, 1994]. Scene graphs take advantage of persistent GPU buffers, and reordering of graphics state switches, to render these scenes in real time, and are ideally suited to static geometry with few scene changes. A more recent trend is to transmit simplified geometry to graphics hardware and allow the GPU to dynamically build detailed models without returning to the CPU. This technique has been successfully applied to mesh refinement, character skinning, displacement mapping, and terrain rendering [Lorenz and Dollner, 2008] [Rhee et al., 2006] [Szirmay-Kalos and Umenhoffer, 2006]. However, it remains an open question as to how to best take advantage of the GPU for generic procedural modeling.

Several early procedural systems developed from the study of nature. L-systems, introduced by Lindenmayer and Prusinkiewicz, use grammars based on string substitu-

tion to model plants [Lindenmayer, 1968]. Kawaguchi models shells, fossils and branches by iteratively and recursively constructing complete models from transformed shapes [Kawaguchi, 1982]. A method related to fractal geometry is employed by Stiny, who constructs models of painted regions using *shape grammars* that replace shape patterns with other shapes. Grammars have also been applied to the modeling of building architectures [Müller et al., 2006], whereby large volumes such as walls are replaced with windows, doors and framing. While grammars have a range of applications, they are often limited to a certain class of objects, and not easily animated, and it is difficult to determine how their evaluation might be parallelized.

An alternative to grammars is to rely on programming languages themselves to generate complex forms. Synder uses a C-interpreter to model volumes and surfaces for CAD objects, with *opreators* that can perform a variety of tasks, including constraint solving, integration and spatial deformation [Snyder, 1992]. In the animation language AL, S. May uses a Scheme interpreter to dynamically generate and animate complex objects, such as generative architectures and anatomical muscle models. AL uses functional operations that arbitrarily transform, deform, or generate new geometry. The commerical software Maya includes MEL, a interpreted scripting language that can be used to generate geometry in relation to efficient, underling C/C++ object models [Gould, 2002]. Although interpreted languages offer a great deal of flexibility, user interaction with such systems is difficult to define, and low-level interaction with graphics APIs makes it difficult to efficiently group geometry for hardware acceleration.

Visual dataflow langauges, VDFLs, provide another solution to procedural modeling. An early VDFL for procedural modeling is ConMan, a system by Paul Haeberli that uses a directed acyclic graphs (DAG) of behavioral nodes to generate objects. These nodes are similar to the language Squeak, whose inventor Luca Cardelli describes a "fragments of behavior", which perform actions on various data [Cardelli, 1985]. Xfrog is a natural modeling system in which *p-graphs*, a layout of labeled verticies, are converted into an *i-tree* by replacing verticies with primitives that are then rendering interactively [Deussen and Lintermann, 2004]. The commerical software Houdini uses VDFLs to describe and animate graphs consisting of surface operators (SOPs), particle operators (POPs), dynamic operators (DOPs), and render operators (ROPs), among others. In addition to developing an integrated VDFL framework for procedural modeling (discussed later), Ganster provides an overview of the features and drawbacks of several other procedural languages [Ganster and Klein, 2007]. VDFLs provide a more natural interface and also a great deal of flexibility as nodes may perform any operation, but it is not entirely clear how these function graphs should be connected to performance-based scene graphs.

While procedural languages have generally increased in expressiveness over time, scene graphs represent geometric models differently, and have evolved to meet other needs. The pre-cursor to the scene graph is a hierarchical description of object transformations, introduced in early systems such as GKS and PHIGS+ in the 1980s [van Dam, 1998]. In 1993, Strauss and Rohlf introduced IRIS Inventor and IRIS Performer, systems

which defined a scene graph as a DAG with nodes that include "shapes, properties, and groups" [Strauss, 1993] [Rohlf and Helman, 1994]. These classifications are introduced to enable efficient state management of graphics hardware, such as sorting by material properties, or spatial culling by groups. Building on this, OpenSceneGraph and OpenSG sought to bring these enhancements to consumers by expanding to different platforms (from SGI IRIX to Linux) and new rendering APIs (such as Microsoft's Fahrenheit) [Harrison, 2007]. Over time, the term *scene graph* has come to signify a wide range of approaches to storing geometric objects. In a panel discussion at SIGGRAPH 1999, graphics professionals were found to have differing views on the definition of scene graphs [Bethel, 1999]. A central issue to organizing geometric objects is that they have competing needs in terms of spatial organization (for culling), material properties (for state switching), functional behavior (animation), and generative abilities (procedural modeling). Avi Bar-Zeev collects and discuss several of these competing goals [Bar-Zeev, 2007].

The problem addressed here is how to integrate the performance benefits of scene graph organizations with the flexibility of procedural modeling languages, to develop a system for real time procedural animation. We build on visual dataflow languages to develop LUNA, an intuitive, high performance, real time system for procedural modeling. Our system is used to interactively model complex organic objects with a deferred shading engine for high quality rendering. We also introduce a procedural reference

model, and use it to do performance comparison with Houdini (the only other generic, modern procedural dataflow language for procedural modeling we are aware of).

4.2 Language Design

4.2.1 Overview

Our approach is an integrated one that combines the modeling advantages of procedural dataflow languages with the performance advantages of scene graph memory organization and GPU kernel execution. The LUNA system consists of two directed acyclic graphs that include a *procedural graph*, representing behavioral nodes, and a *scene graph*, representing geometry and scene state. Our general design principles over these two graphs are:

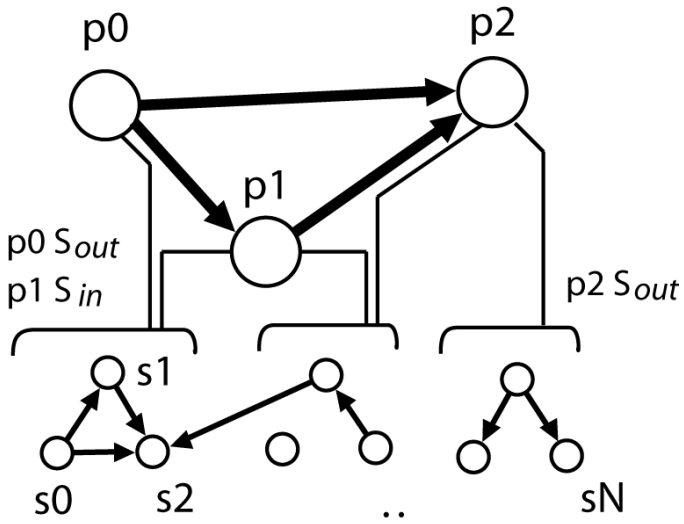
1) *Procedural graphs generate multiple scene sub-graphs.* A procedural node represents the behavior of a high-level concept, and is capable of taking multiple scene nodes as input, and generating multiple scene nodes as output. For example, in a classical design patter, a Mesh class would maintain geometry and allow loading from a file. In our system, *mesh* object may be a scene node storing geometry, but a MeshFile is a procedural node which can generates any number of mesh scene nodes. This distinction allows us to define many procedural behaviors for each geometry class.

2) *Scene graphs use API-dependent proxy objects to manage rendering.* At present a scene node specifies a pivot, material, and geometry buffers. A procedural node groups its output scene graph into a set of objects of similar type, which are rendered using

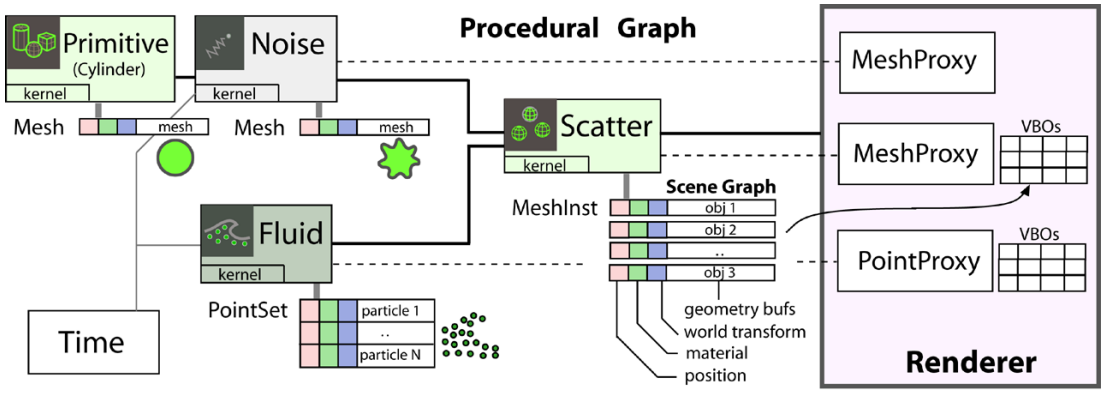
an API-dependent *proxy object*. The proxy object can take advantage of the pivot to perform spatial culling, the material to group for graphics texture state, and the geometry to group by object type, similar to the performance enhancements found in IRIS Performer.

3) *Execution takes advantage of scene graph organization, hardware rendering, and GPU kernel execution.* Our system takes advantage of the GPU in three ways: 1) The behavior of procedural nodes can be optionally executed on the GPU using GPU-specific node kernels, 2) Scene graph organization is optimized through grouping of geometry types mentioned earlier, 3) Rendering is optimized for quality and performance using a graphics hardware-based deferred shading engine.

An explicit distinction between behavior and structure can be found in several newer visual dataflow languages. In the integrated system by Ganster and Klein, a *model graph* represents functional operations that generate structure [Ganster and Klein, 2007]. Although the system is flexible, complex models such as trees are generated through iteration which occurs at the model level, reducing the performance of the system considerably. In LUNA, procedural nodes represent high level operations, and iteration inside their kernels produces multiple outputs. Forbes develops a system in the area of information visualization that models behaviors and structures as distinct graphs that can operate on one another [Forbes et al., 2010]. This system creates complex transitions for visual data but does not deal with generative geometric structures.



(a)



(b)

Figure 4.1: Structure of the LUNA graphs in which, a) a *procedural graph* operates on subsets of a *scene graph*, and a specific example b) in which four procedural nodes are used to generate multiple scene nodes to render the image shown in Figure 4.2. In the actual storage of the graph, all nodes are maintain by a single graph system, and their usage and behavior is distinguished by object semantics.

4.2.2 Formal Definition

Formally, LUNA is defined by two directed acyclic graphs, a procedural graph (P) that modifies input and output subsets of a scene graph (S). Each graph is a set of nodes and edges, while the P verticies additionally reference subsets of S:

$$P = \{P_v, P_e\} \tag{4.1}$$

$$S = \{S_v, S_e\} \tag{4.2}$$

Each of the nodes and edges of these graphs have a particular interpretation. A procedural node, $P_v \in P$, defines behavior that consists of a CPU kernel, an optional GPU kernel, which operate on an input subset of the scene graph, $S_{in} \subseteq S$, to produce an output subset, $S_{out} \subseteq S$. Procedural nodes also include a *proxy object*, P_x, which is referenced by the node but managed by renderer, and maintains the vertex buffers and graphics state for the object's output sub-graph. A procedural edge (P_e) connects two P-nodes and defines the functional inputs to a behavior (such as an image and a mesh being the input to a displacement).

$$P_v = \{S_{in}, S_{out}, P_x, kernel(CPU/GPU)\} \tag{4.3}$$

$$P_e = \{P_a, P_b\} \tag{4.4}$$

A scene node, $S_e \in S$, defines a media object, which may be an image, shape, mesh, sound, or other structure. For geometric objects, a scene node maintains a pivot which defines local and world transformations, references to material nodes that define shaders and textures, and geometry buffers which maintain verticies, edges and faces. Scene edges, S_v, define relationships between two geometric objects, the most common

example of which is an *instance* relation (MeshInst refers to a Mesh), described more later.

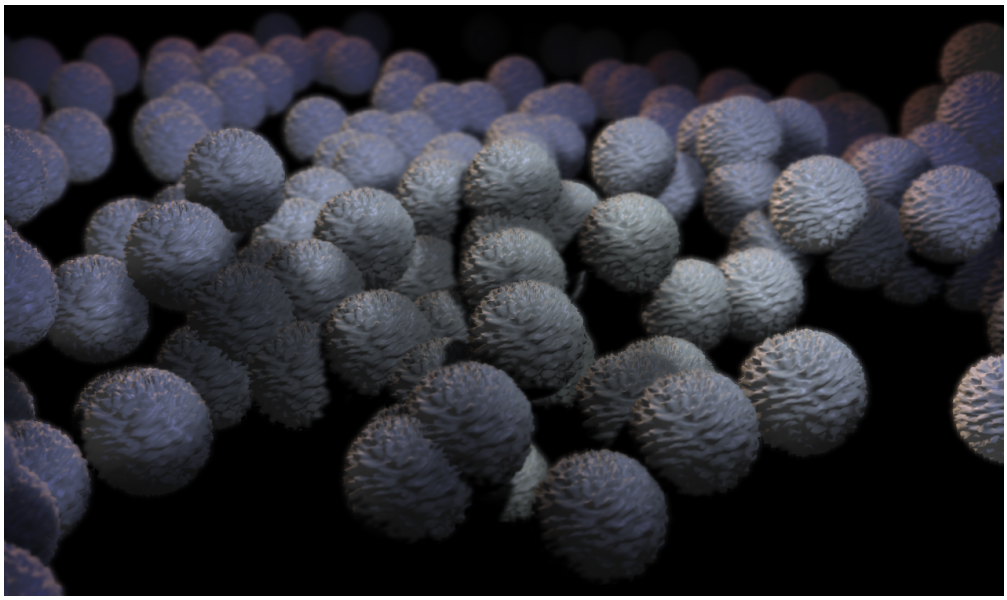
$$S_v = \{\text{pivot, material, data buffers}\} \tag{4.5}$$

$$S_e = \{S_a, S_b\} \tag{4.6}$$

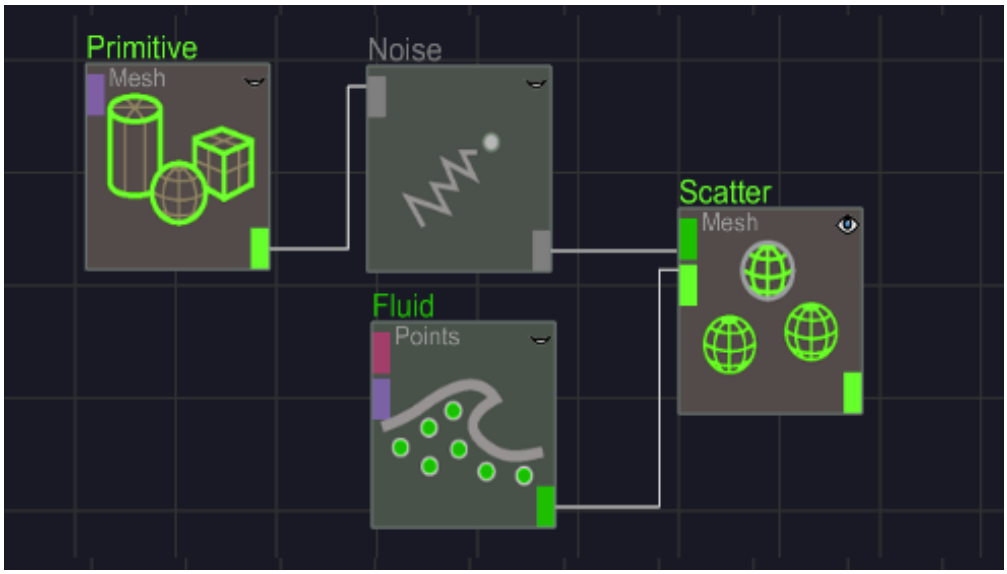
A graph demonstrating this structure is shown in Figure 4.1. Visually, our user interface shows the procedural graph while the scene graph remains hidden. As the system is procedural this is ideal since the output scene geometry can grow rapidly. Users create edges by click-dragging from an input node to an output node, as shown in Figure 4.2. In the graph structure, the Scatter P-node is connected to the renderer, and in our graphical interface this is indicated by the 'eye' icon. As such, any object in the procedural graph may be visualized by clicking on this icon, allowing multiple objects to be connected to the renderer. A special Time node is automatically introduced by the system to trigger time-dependent changes per frame.

4.2.3 Evaluation Model

The evaluation model consists of two basic steps. First, the procedural graph is traversed from the Time node outward to any downstream nodes that require updating (left-to-right), as shown in Figure 4.3a. In this example, it requests that all nodes except the primitive (sphere) be re-evaluated. The second step is to traverse the graph



(a)



(b)

Figure 4.2: The graphical interface in figure b) produces the visual results shown in a), by building the internal graph structure from Figure 4.1.

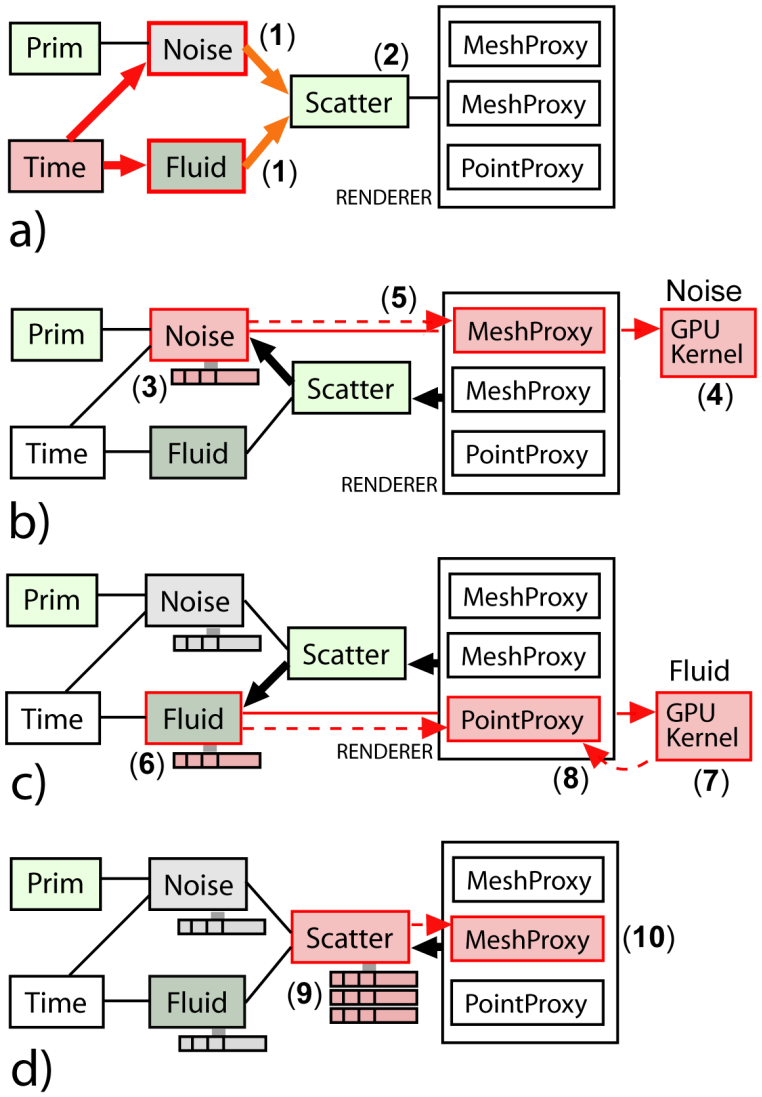


Figure 4.3: Sequence of steps in the evaluation model: a) The Time node requests updates by traversing graph dependencies, b) The Noise node is traversed in depth-first order to update a noise-animated mesh using a GPU kernel, and the result is stored on the GPU via a MeshProxy, c) The Fluid node is visited to update fluid particle locations using a GPU kernel, d) The Scatter node generates or updates the world transform of mesh instances, using the GPU mesh from step b as an instance that is scattered at the point locations of step c.

in depth-first order from any visible P-node connected to the renderer, to evaluate that node and update its geometry. Repeated evaluation is avoided by tagging the nodes that have already been evaluated on the current frame. For each visited node, its CPU or GPU kernel is called, which will generate the output sub-graph S_{out}. The proxy object P_x for that P-node is then called to update vertex buffers and to render the resulting sub-objects.

The example of Figure 4.1 uses a smoothed particle hydrodynamic simulation (SPH), a sphere primitive, and a noise generator, to generate a number of organic noise-animated spheres suspended in a fluid. The Primitive node creates a mesh scene output which is generated only once. Figures 4.3b through 4.3d show the results of the traversal process on each node in the example. The Noise node generates an animated noisy sphere which is updated on every frame, while the Fluid node generates a PointSet output which updates the fluid particle locations per frame.

The output scene graphs of Noise and Fluid, consisting of a mesh and a point set, are used as inputs to the Scatter node, which generates a list of MeshInst objects at each of the point locations. A MeshInst is a sub-class of a Mesh scene node which supports much of its functionality through a reference to another Mesh node. In this case, many MeshInst nodes with different transforms refer to a single input mesh, providing a way to represent geometry instancing. To avoid repeatedly generating the MeshInst output scene graph, the Scatter P-node detects changes in the number of elements of its

inputs, and only rebuilds when necessary. This allows nodes to generate more geometry as needed, or efficiently animate the geometry present.

4.3 Implementation

4.3.1 Discrete Geometry

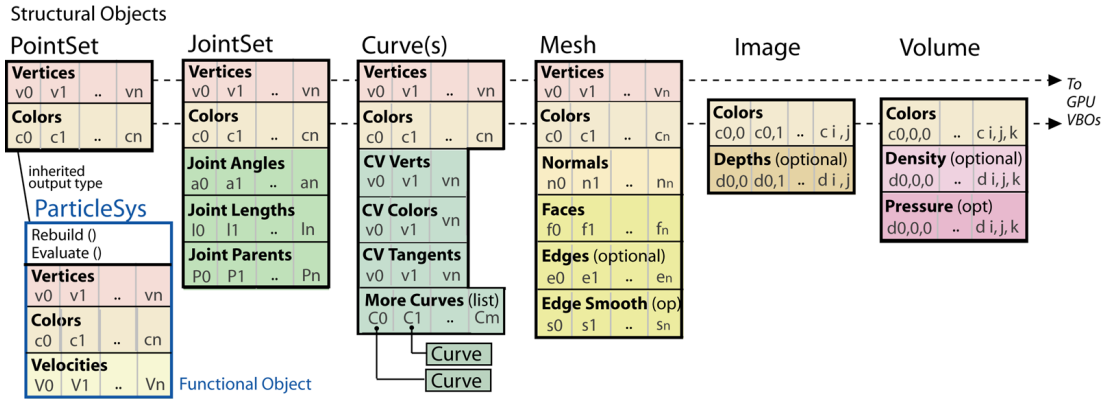


Figure 4.4: Objects are represented as discrete geometry in LUNA with uniform buffers. Buffers may vary in length, as in the example of CV Verts for the Curve object shown. Other buffers might store pointers to allow lists of the same object class (C0,C1,..,Cn). Data is sent to the GPU in a unified way using proxy objects. Some buffers are allocated by derived nodes, such as particle velocity. Other functional objects may operate across many object types. A bend transform, for example, works with any object that has a vertex buffer.

Scene nodes store and transmit geometry through the graph. Their represetation consists of discrete geometry stored in the *data buffers* of each node. These buffers have a direct mapping to GPU buffer objects and can be directly copied from CPU memory to the GPU. While a common method of storing objects uses a C-struct to describe individual elements, this makes it difficult to expand object attributes at run time, whereas performance and flexibility are improved by storing structures in buffers

of uniform types. Figure 4.4 shows the base classes for discrete sets and their similarities in buffer structure.

In creating geometric types for LUNA, the concept of uniform buffers is abstracted to an arbitrary number of named buffers, each with variable length, and fixed sized elements with in a given buffer that implement a particular discrete geometry. Their layout is designed to match graphics hardware buffer objects, yet store every per-element variable needed for high-level objects including hierarchical relationships. Trees for example, are stored in JointSet scene objects, which contain buffers that hold references to parent and child branches.

A key benefit of using uniform buffers is that broad classes of objects can be treated similarly both during evaluation and rendering, without the need to define new functions. Points, curves, joints and meshes are all defined with their three-dimensional vertices as the initial buffer. Thus, modifiers such as twist, bend and distort can be performed in a consist way across many different geometry types, including points, curves, and surfaces. The renderer requires vertex-face information and normals, which are available as additional mesh buffers.

4.3.2 Performance

The structure of LUNA supports procedural modeling by allowing multiple outputs for each behavioral operator, while its evaluation model supports several performance enhancements. First, only the portions of the graph that change over time are re-

evaluated per frame. Second, the proxy objects efficiently transfer only the sub-graph geometry that has changed from CPU to GPU, and also group that geometry to share render state transitions (e.g. materials). While the scene graph S formally represents the total visual output of the system, because of its generative nature it is useful to conceive of the system as a set of procedural nodes in which scene sub-graphs flow through the system. A procedural node can essentially create, modify, destroy, or transform an input scene graph to generate an output scene graph. All of these changes are found in the complete graph, as the scene graph S holds inputs and outputs of all procedural nodes. This introduces a potential problem common to VDFLs: the replication of data buffers at later points in the graph. In the current system, modifiers copy input objects to outputs, and then perform transformations on the output buffers. This resolves situations where a single input is modifier in two different ways, but is memory inefficient when there is only one output. In the future, the evaluation model could be modified to remove this inefficience.

The LUNA design takes advantage of the ability of modern graphics hardware to invoke re-entrant kernels. Giden et al. develop such a system consisting of graphs of CUDA kernels for the eXtensible Imaging Platform (XIP) of the National Cancer Institute [Giden et al., 2008]. In LUNA, each procedural nodes may have an optional CUDA GPU kernel that performs simulation, or geometry generation, on the GPU. Notice there can be multiple GPU kernels in a graph in Figure 4.3b, and these can be interspersed with rendering calls to allow several high level nodes to be evaluated on

the GPU while also using the GPU for rendering. At present, only the GPU kernel for the SPH fluid simulator is written, and has not yet been tested in the context of other GPU nodes although the design allows for it.

Data replication between CPU and GPU versions of an object are avoided through the proxy objects. At present, the CPU maintains the final copies of all data. For objects with single outputs, the proxy object transfers that data to the GPU and renders it, updating only changed buffers. In the future, for nodes evaluated with GPUs kernels, it should be possible to directly update the proxy object, as in Figure 4.3c, without returning the data to the GPU. By authoring multiple nodes for the GPU, this should allow complete graphs to be evaluated entirely on the GPU. We view the bus transfer from CPU-to-GPU as a flexible step based on which portion of the graph requires the data next.

Although CPU-GPU bus transfers incur a cost for dynamic geometry they also introduce a benefit in procedural modeling. In RenderAnts, Zhou et al. develop an adaptive parallel pipeline for geometry transfer to the GPU in a Reyes rendering environment. Dynamic loading of the GPU allows for more complex models since scene detail is not bound by GPU memory. This can be an advantage in procedural modeling as well, and LUNA proxy objects support this by *reusing* the same vertex buffer objects for different mesh geometies in the same render frame. Although not adaptive in the same sense as RenderAnts, each P-node generates an output sub-graph of varying size based on user requested detail levels, and the proxy can quicky load different meshes in a single set of

VBOs allowing for much greater scene complexity. This is demonstrated in particular by the Loft examples in the next section.

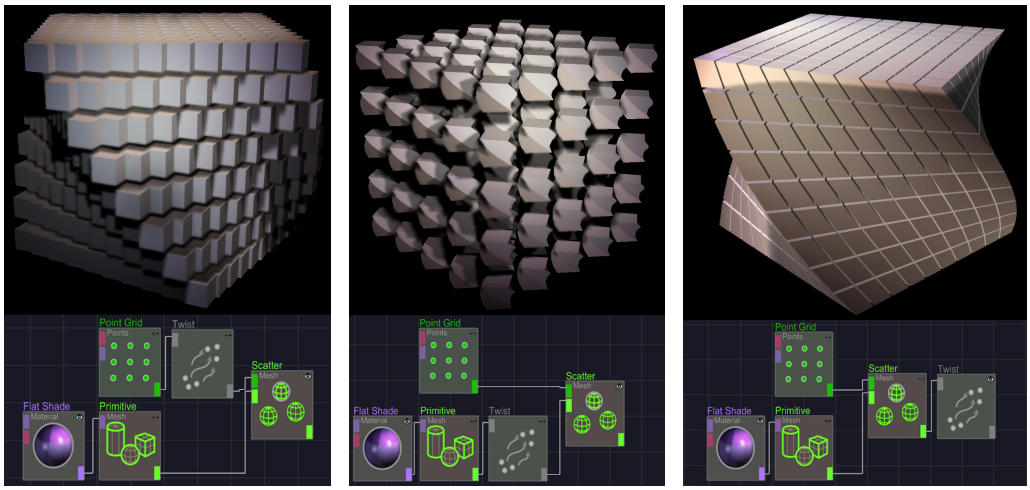
4.3.3 Rendering

The integration of procedural modeling with high quality real-time rendering is a novel research area. Although deferred shading is typically found only in game engines with objects of particular class types (terrain, etc), the LUNA rendering engine supports deferred shading and multi-pass rendering of procedural models over OpenGL. Individual objects in LUNA may have Cg shaders assigned to them which permit advanced visual effects on generated models. Cg shaders are included in procedural graphs as material inputs to nodes. At run time, the proxy object for a particular node runs any Cg shaders prior to primitive drawing. Screen space shaders allow for effects such as depth-of-field, deferred soft shadows, motion blur and light bloom. Increased performance due to geometry classes, lack of temporary objects, and streamlined buffer transfers allows procedural models to be generated and shaded in a real time, high quality rendering environment.

4.4 Procedural Results

4.4.1 *Twist*: Modifiers and Order of Operation

Modifiers are procedural nodes that operate on *any* input geometry type. This is accomplished by copying the input sub-graph, which may be points, lines, curves, or



(a) (b) (c)

Figure 4.5: Modifiers operate on different geometry types. In these examples, a cube Primitive (bottom left) is instanced at PointGrid locations (top left) by a Scatter node (right). Placing a twist at different stages in the graph produces a) A twisted grid with untwisted cubes, b) Twisted cubes located at an untwisted grid, and c) A regular grid with regular cubes, the whole of which is twisted.

surface geometries, to the output sub-graph, and then uniformly modifying the point locations according to some global transform. The order of these operations is also important, the examples in Figure 4.5 show, where a cube is instanced at grid locations. Depending on what stage the Twist operation is performed produces different variations of twisted/untwisted objects. In this example, the grid locations, the cube mesh, or the entire object are twisted separately. Although a point cloud (PointSet) can be twisted by its verticies, to correctly twist a mesh requires transforming both verticies and normals. This is done by allowing the Twist node to inspect the type of its output and perform an additional transformation on the normal buffer.

Figure 4.6: *Twist*, R. Hoetzlein (2010). A tree is constructed from a JointSet hierarchy which is then built up using cylinders Primitives. The overall shape is then twisted to procedue these results, rendered in real time using deferred shading, shadows, and depth of field.

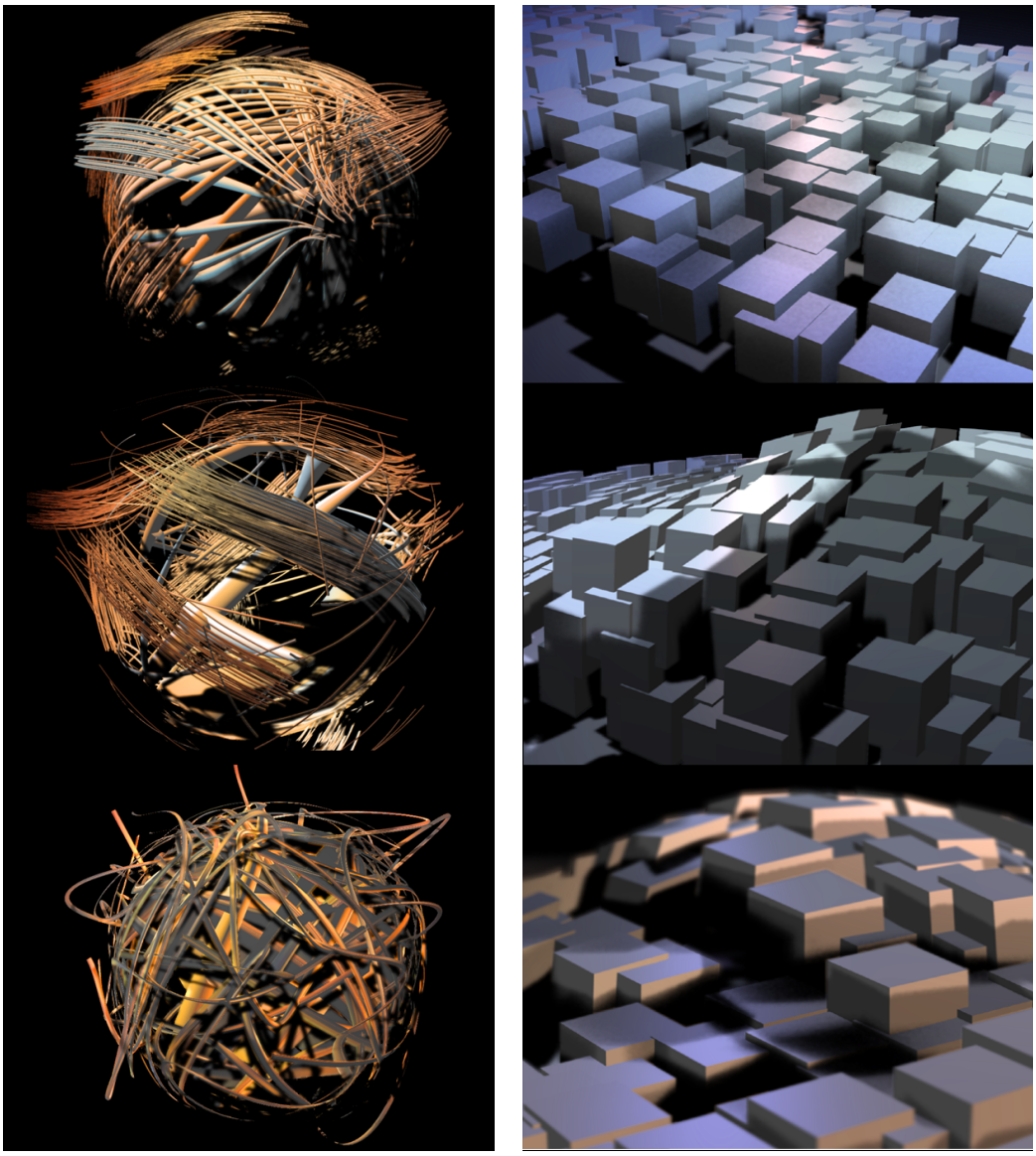
A more complex twisting example is shown in Figure 4.6. Here, a tree is constructed from a JointSet hierarchy, which is made into a solid object by scattering Cylinder primitives at the joint locations. Unlike an interpreted function, which might recursively traverse the tree, determine joint angles, and instance a cylinder as it goes, in LUNA the structure of the entire tree is generated first. After which, any object may be instanced at this tree structure separately from the tree definition. A twist is added to the final

result, distorting the cylinder shapes in addition to the tree structure, producing the image shown.

4.4.2 *Scatter*: Compound instancing

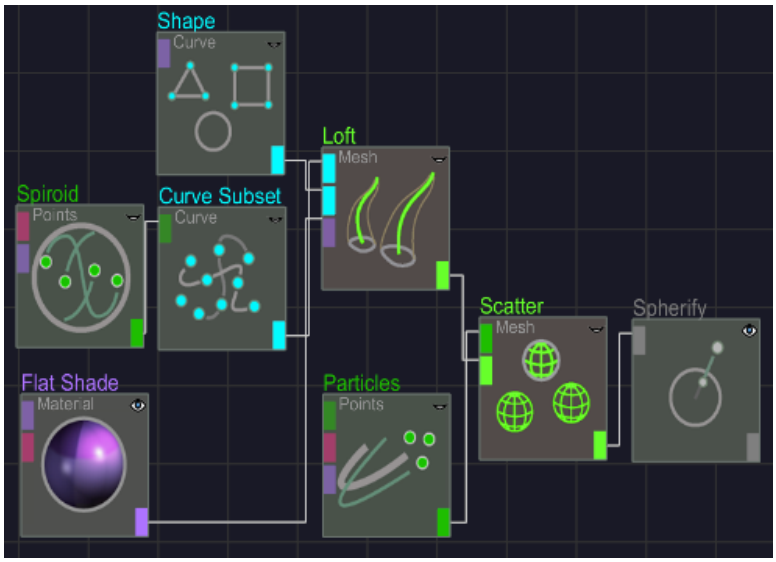
One of the key features of procedural modeling systems identified by Reeves is replication, the ability to generate instances of a model with subtle variation [Reeves et al., 1990]. The LUNA language can distinguish between *instancing*, which is the copying of a single mesh at multiple locations and orientations, and *replication*, the ability to repeatedly re-evaluate a procedural model so that each instance is different from the last. Currently, the node for LUNA to support replication is not yet written, as this requires repeated evaluation of procedural sub-graphs. However, LUNA does support nested or compound instancing, the ability to create instances of models that themselves contain instances.

Examples of compound instances are shown in Figure 4.7 and 4.8 using a Scatter node. In the first example, 4.7a, a Loft object is used to generate N swept surfaces as output scene sub-objects according to Subset Curves defined over a set of points. These sub-objects are then Scattered to produce N*M swept curves, whose parameters in this case give the appearance of three dimensional brush strokes. In the second example, 4.7b, a cube primitive is Scattered at regular locations on a point grid. The result is then Scattered again at moving particle locations to create N*M output cubes with more variation. As scattering multiplies its input by N point positions, nested scattering

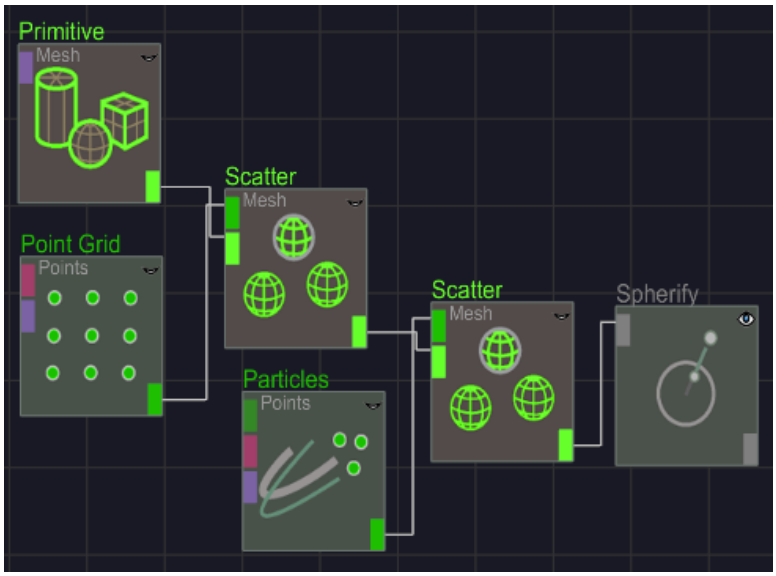


(a) (b)

Figure 4.7: Compound systems creating using a variety of objects, a) Swept surfaces are generated from a particle system, which is then scattered to create the appearance of three dimensional brushstrokes composed of tubes, b) A collection of cube primitive is scattered using a grid, then scattered again to produce variation. In both systems, space is distorted by using a spherify operation on the output.



(a)



(b)

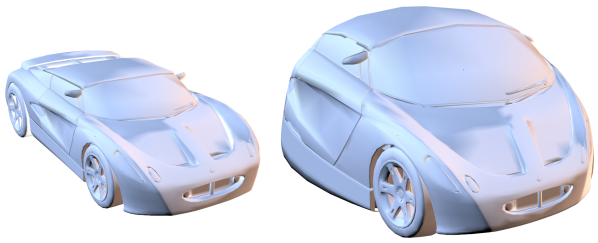
Figure 4.8: Graphs for the visual results of Figures 4.7a and 4.7b, showing a) a collection of loft surfaces which are scattered, and b) a collection of scattered cubes which are scattered again. In both cases the compound result is *spherified* to distort the final space.

can result in an exponential generation of mesh geometry growing as $O(N^M)$, assuming N particles per scatter, and M scatter operations. To prevent system stalling, a user controllable maximum count is present on each Scattering node, forcing a limit on the maximum output at each to step. In the future, it should be possible to allow the renderer itself to dynamically adjust these limits to meet performance or quality goals for real time or offline rendering.

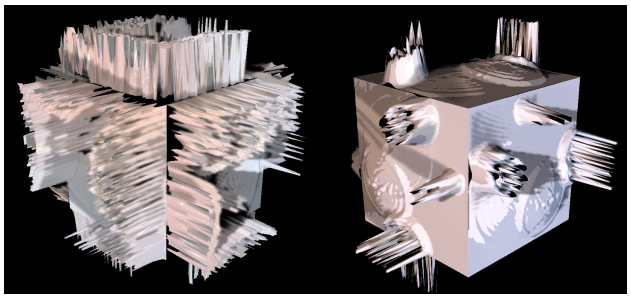
4.4.3 *Displace, Wave, Cube*: Other experiments

A number of other experiments are shown in Figure 4.9. These experiments include a) a spherified car mesh, b) a cube primitive with wave displacement rendering with toon shading, c) architectural forms created using a combination of swept and fun surfaces, with texturing, d) shells created by animated wave displacement of a sphere, rendered with environment mapping, and e) a planet-like form for rendered using curve subsets on a set of points surrounding a sphere.

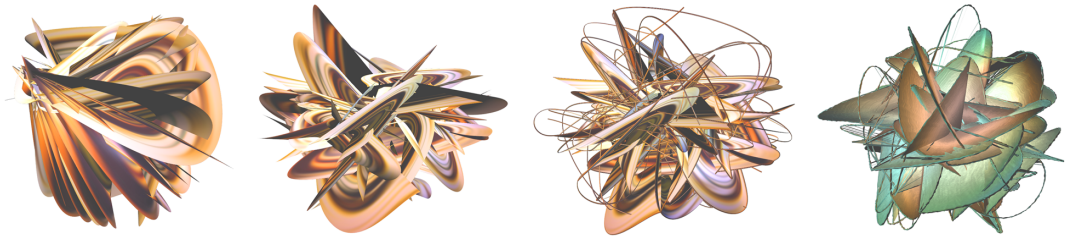
All objects in these examples are rendered at interactive rates in real time with soft shadows, depth of field, and texturing. Notice in example e), *planets*, the rendering system allows for combinations of curve and mesh primitives, showing different portions of the procedural graph simultaneously. Example b) and d) show the use of other media types in conjunction with surfaces, where an animated image is used to displace the surface points of a mesh along its normals. This is an ideal node for implementation



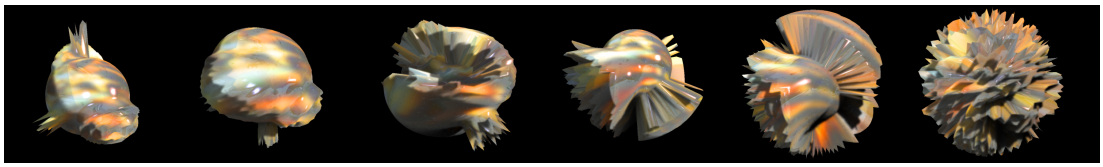
(a)



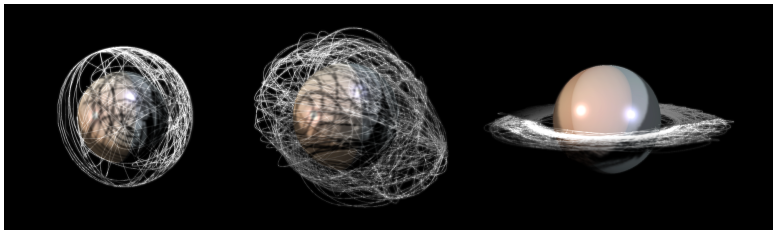
(b)



(c)



(d)



(e)

Figure 4.9

as a GPU kernel which, in the future, could perform mesh refinement and displacement in one step.

4.5 Performance Results

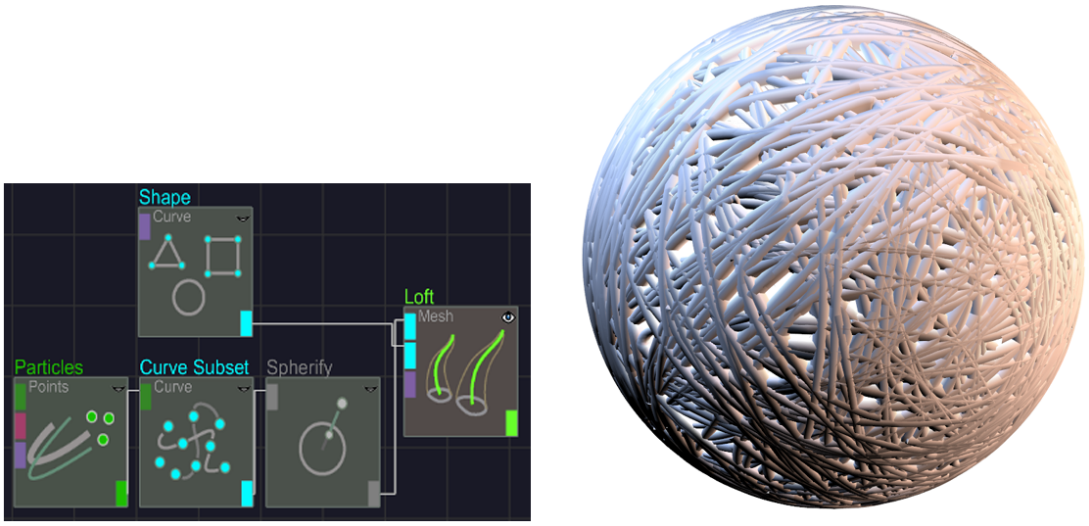


Figure 4.10: Procedural reference model introduced for performance comparisons with Houdini and a baseline OpenGL model. The object is specified as a set of curves defined from random subsets of points, normalizing the curves to a sphere, and lofting them to create tubes. See Appendix A for a detailed definition. The LUNA graph used to make this object is also shown.

Standard reference models for procedural modeling do not yet exist. Baseline models for static geometry have been around since the Utah teapot [Torrence, 2006], the Stanford Bunny [Turk and Levoy, 1994], and the happy Buddha [Curless and Levoy, 1996]. Similar test objects are available for volumetric data [1]. We thus propose a test case for procedural modeling: a woven sphere composed of swept surfaces residing on a sphere,

[1]Several volumetric data sets are available from http://www.volvis.org/ with references to original authors of the data

Figure 4.10. A detailed description of the woven sphere is presented in Appendix A and also available online.

The woven sphere is a suitable reference for procedural modeling for several reasons. First, it is relatively simple, and although it contains physical animation it cannot be created from a physics simulation alone. Second, later construction steps are dependent on earlier time-dependent motions. Third, it requires evaluation of random point subsets to create curves, which is a combinatorial process that is efficient when done correctly. Fourth, it generates multiple loft surfaces which cannot be created through instancing or static geometry deformation, so it is unique to procedural systems. Finally, the meshes generated require vertex re-evaluation per frame which forces a CPU-to-GPU transfer, or direct evaluation on the GPU. Thus, as hardware support for procedural generation of dynamic geometry improves this object can be used to investigate bus transfer overhead during rendering. In addition to these specific features, the object is relatively simple from a procedural standpoint and its storage complexity can be computed theoretically.

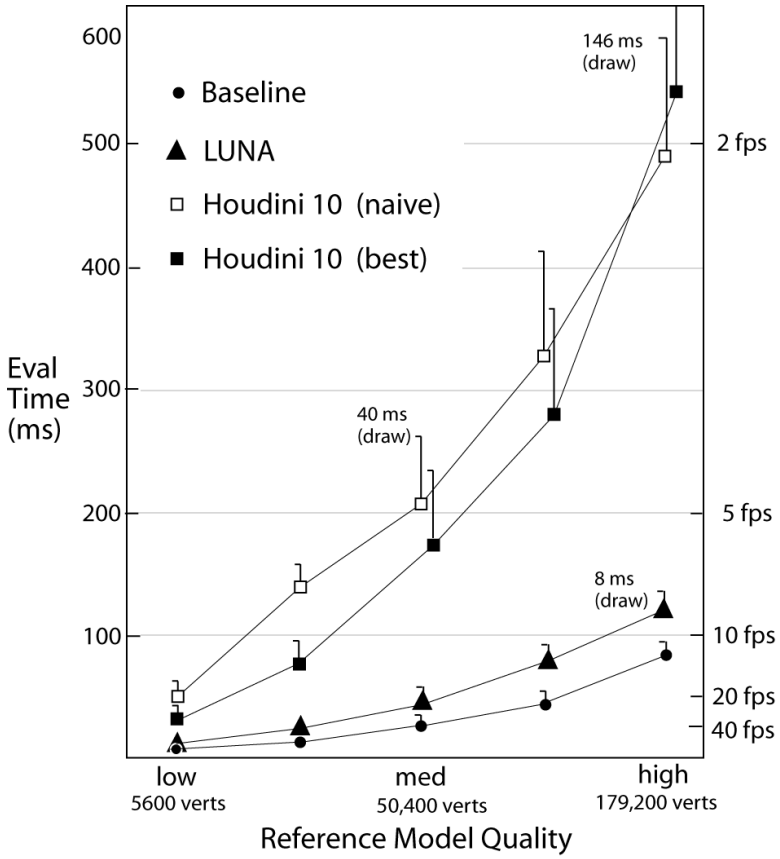
To advance the woven sphere as a test case in the graphics community, we implement a baseline model directly in C++ using GLUT without a procedural modeling language, and make this freely available. This provides an absolute reference for the best possible performance since it eliminates any overhead due to language evaluation. We also generate the woven sphere in both Houdini and in LUNA for comparison.

We recommend specific parameters for low, medium and high resolution reference models in Appendix A. For Houdini, we measure evaluation time and viewport drawing

cost using Houdini's "performance monitor". To guarantee the level of detail settings match our baseline, the number of sample vertices per curve are individually counted. We also check that the total number of vertices and faces are very close to the theoretical number for a given test (Houdini does not allow one to set resampled curve resolution exactly). For LUNA, we construct a graph for the woven sphere and set the sampling parameters to match the baseline.

Performance results for the woven sphere for the baseline, Houdini and LUNA models are shown in Figure 4.11. The LUNA reference model is implemented using five nodes: Particles, Subset Curves, Spherify, Curve (shape) and Loft, and can be constructed in the interface in under a minute with ten clicks and four click-drag motions. On average, LUNA is consistently 4x to 7x faster than Houdini and only 2x slower than the baseline model (a direct C implementation with no overhead). All LUNA models run in real time with the low res model at 150 fps (7 ms total) and the high res model at 8 fps (126 ms total)

The baseline model is implemented directly in C using GLUT for rendering. Our first, naive implementation in Houdini uses the Copy Stamping SOP to evaluate an expression to generate point subsets for curves. Repeated interpretation of string expressions is likely the cause of reduced performance here. Following discussions in online Houdini forums, a better method uses the AttribCreate OP to tag particles into groups and then the Add SOP to generate curves from the groups by attribute name. This Houdini graph requires ten nodes. Although performance is improved in most cases,



(a)

Model	Verts	Baseline (ms)		Houdini[1]		Houdini[2]		Luna (ms)	
		eval	draw	eval	draw	eval	draw	eval	draw
Low res	5k	2	<1	40	4	27	4	5	1
.	22k	10	<1	140	13	69	18	22	3
Med res	50k	24	1	203	39	183	47	45	4
.	89k	44	2	318	71	276	71	89	7
High res	179k	88	5	536	130	555	146	118	9

[1] Naive method. Created using Copy Stamping SOP and expressions.
[2] Suggested method. Created using AttribCreate SOP and Add SOP.

(b)

Figure 4.11: Performance comparison for graph evaluation and rendering time (in milliseconds) for the reference model in LUNA, Houdini 10, and OpenGL baseline.

this technique is slower than Copy Stamping for the high-res model and averages 10x slower than the baseline model overall.

Interestingly, although this object represents the worst case for bus transfer overhead due to dynamically generated geometry per frame, in the baseline test it represents only 4 to 6% of the total cost (5 ms out of 93 ms for the high-res model). In a game engine, however, a 5 ms cost for a 180k vertex object is unacceptable, and in such a context this object would be solved using fixed vertex shaders on the GPU while sacrificing flexibility. LUNA render times closely match the baseline. In Houdini, the viewport drawing cost averaged between 10 to 15% of the total cost, and we are unsure of the reasons for this additional overhead.

4.6 Conclusions

We demonstrate a visual dataflow language for procedural modeling, LUNA, that is efficient and flexible. This system integrates a procedural modeling graph which operates over input and output subsets of a scene graph. The scene graph is implemented using discrete geometry, selective updates, and material grouping, to allow for interactive rendering using a deferred shading engine. The interface allows novice users to quickly prototype objects without the need to understand detailed controls, and the performance of the system enables direct feedback on the structure, appearance, and surfacing of complex models. A demo version of LUNA is currently available online at

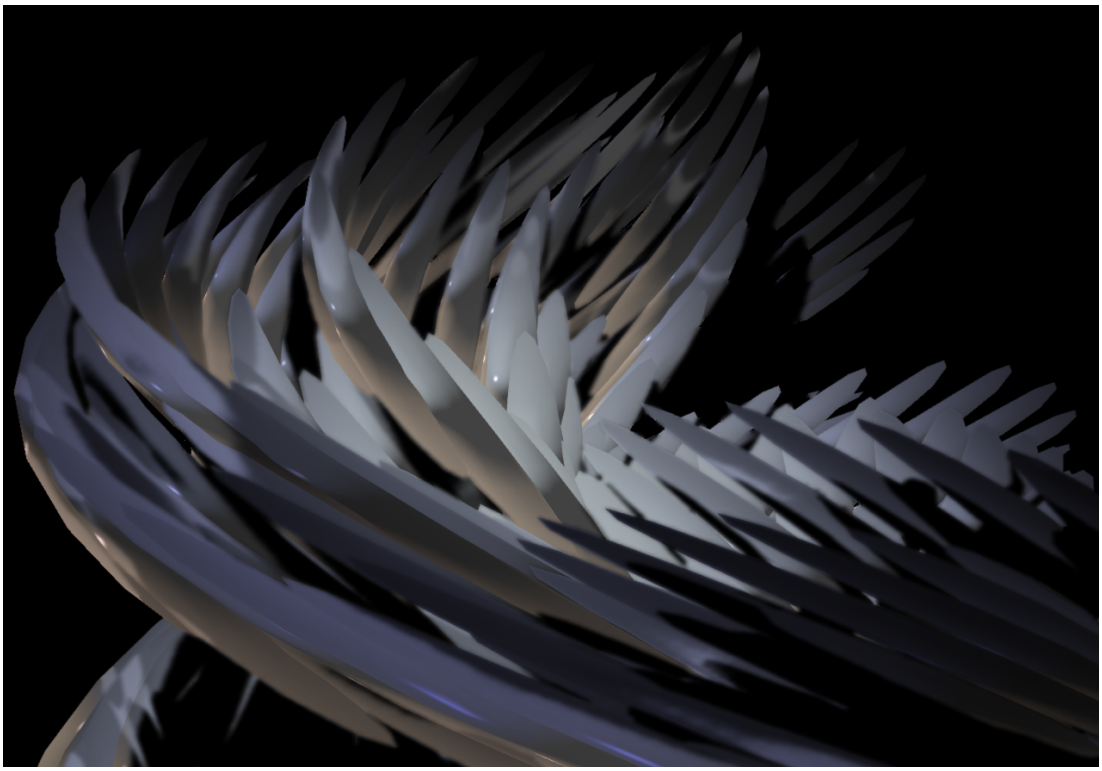


Figure 4.12: *Glass*, R. Hoetzlein (2010). Surfaces generated by twisting tubes, rendered with glass-like material properties.

http://www.rchoetzlein.com/luna/. Our system is tested and compared to others using a new reference model for procedural system.

There are several limitations still present in LUNA we hope to address in the future. First, although individual objects may contain heirarchies (e.g. JointSet, Tree), the scene graph is incomplete in the sense that output subsets are still primarily lists of scene nodes rather than transformation hierarchies. Cross-referencing using scene edges to connect scene nodes is present only during mesh instancing and material referencing, while hierarchical models such as trees are handled by propagating local transforms of joints to the world transformations of output objects. A more complete model would

implement a complete scene graph model for each procedural node, allowing for multiple P-node output types and more complex object relationships. For example, in the future character models should be possible using named scene nodes with joint relationship handled internally.

Several performance improvements are still possible. At present, LUNA takes advantage of scene graph performance primarily by grouping material and texture render state, and by intelligent updating of hardware geometry buffers. Although the information is present in LUNA, we have not yet implemented spatial culling or acceleration structures which would improve the system considerable for larger scenes. Dynamic overlapping geometry, such the woven sphere reference model, still present a difficult performance challenge which can only be addressed through better polygon-level rendering. However, it should be possible to accelerate disjoint objects using known techniques since some objects are naturally separated both spatially and conceptually as high level entities by the P-graph.

A benefit of the language model presented is that it allows for different interpretations. LUNA is simply defined as a procedural graph that makes generative changes to subsets of a scene graph, adding or modifying scene nodes as needed. While LUNA does not yet contain a detailed vocabulary for physical simulation, crowds, or character modeling, we believe that due to the structure of the langauge these objects should be relatively easy for communities to added in the future. To our knowledge, LUNA is the

first system to allow for high quality, deferred shading of complex procedural models with interactive feedback.

Bibliography

[Bar-Zeev, 2007] Bar-Zeev, A. (2007). Scenegraphs: Past, present, and future. http://www.realityprime.com/scenegraph.php, visited June 2010..

[Bethel, 1999] Bethel, W. (1999). Scene graph apis: Wired or tired? Panel discussion, SIGGRAPH 1999.

[Cardelli, 1985] Cardelli, L. (1985). Fragments of behavior. In *Personal Communication. DEC Systems Research Center*, Palo Alto, CA.

[Curless and Levoy, 1996] Curless, B. and Levoy, M. (1996). A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, New York, NY, USA. ACM.

[Deussen and Lintermann, 2004] Deussen, O. and Lintermann, B. (2004). *Digital Design of Nature: Computer Generated Plants and Organics*. SpringerVerlag.

[Forbes et al., 2010] Forbes, A., Hollerer, T., and Legrady, G. (2010). Behaviorism: A framework for dynamic data visualization. In *Proceedings of InfoVis 2010. Oct 24-29th.*, Salt Lake City, Utah. IEEE.

[Ganster and Klein, 2007] Ganster, B. and Klein, R. (2007). An integrated framework for procedural modeling. In *Spring Conference on Computer Graphics*, pages 150–157, Comenius University, Bratislava.

[Giden et al., 2008] Giden, V., Moeller, T., Ljung, P., and Paladini, G. (2008). Scene graph-based construction of cuda kernel pipelines for xip. *Proceedings of High-Performance Medical Image Computing and Computer Aided Intervation (HP-MICCAI) Workshop, Sept 2008.*

[Gould, 2002] Gould, D. (2002). Complete maya programming - an extensive guide to mel and the c++ api. Elsevier, San Francisco.

[Harrison, 2007] Harrison, D. (2007). Evaluation of open source scene graph implementations. Technical Report. Visualization & Virtual Reality Research Group.

[Kawaguchi, 1982] Kawaguchi, Y. (1982). A morphological study of the form of nature. In *Computer Graphics*, volume 16.

[Lindenmayer, 1968] Lindenmayer, A. (1968). Mathematical models for cellular interactions in development. In *Journal of Theoretical Biology*, volume 18, pages 280–315.

[Lorenz and Dollner, 2008] Lorenz, H. and Dollner, J. (2008). Dynamic mesh refinement on gpu using geometry shaders. In *In WSCH 2008 Full Papers Proceedings*, pages 97–104.

[Müller et al., 2006] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. (2006). Procedural modeling of buildings. In *Transactions on Graphics*, volume 25, page 614623.

[Reeves et al., 1990] Reeves, W. T., Ostby, E. F., and Lefler, S. J. (1990). The menv modelling and animation environment. In *Journal of Visualization and Computer Animation*, volume 1, pages 33–40.

[Rhee et al., 2006] Rhee, T., Lewis, J., and Neumann, U. (2006). Real-time weighted pose-space deformations on the gpu. In *Computer Graphics Forum*, volume 25, pages 439–448.

[Rohlf and Helman, 1994] Rohlf, J. and Helman, J. (1994). Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. *In Computer Graphics, Proceedings of ACM SIGGRAPH '94*, pages 381–394.

[Snyder, 1992] Snyder, J. (1992). Generative modeling for computer graphics and cad: symbolic shape design using interval analysis. In *Academic Press Professional, San Diego*.

[Stiny and Gips, 1971] Stiny, G. and Gips, J. (1971). Shape grammars and the generative specification of painting and sculpture. In *IFIP Congress 1971. North Holland Publishing.*

[Strauss, 1993] Strauss, P. S. (1993). Iris inventor, a 3d graphics toolkit. *SIGPLAN Not.*, 28(10):192–200.

[Szirmay-Kalos and Umenhoffer, 2006] Szirmay-Kalos, L. and Umenhoffer, T. (2006). Displacement mapping on the gpu - state of the art. In *Proceedings of Eurographics*, volume 25, pages 1–24.

[Torrence, 2006] Torrence, A. (2006). Martin newell's original teapot. In *SIGGRAPH '06: ACM SIGGRAPH 2006*, page 29, New York, NY, USA. ACM.

[Turk and Levoy, 1994] Turk, G. and Levoy, M. (1994). Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318, New York, NY, USA. ACM.

[van Dam, 1998] van Dam, A. (1998). Some personal recollections on graphics standards. In *The History of Computer Graphics Standards Development.*

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 A.1. 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 sub-graphs 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.