Procedural Graphics Model and Behavior Generation

J.L. Hidalgo, E. Camahort, F. Abad, and M.J. Vicent

Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, 46021 Valencia, Spain

Abstract. Today's virtual worlds challenge the capacity of human creation. Trying to reproduce natural scenes, with large and complex models, involves reproducing their inherent complexity and detail. Procedural generation helps by allowing artists to create and generalize objects for highly detailed scenes. But existing procedural algorithms can not always be applied to existing applications without major changes. We introduce a new system that helps include procedural generation into existing modeling and rendering applications. Due to its design, extensibility and comprehensive interface, our system can handle user's objects to create and improve applications with procedural generation of content. We demonstrate this and show how our system can generate both models and behaviours for a typical graphics application.

1 Introduction

Many application areas of Computer Graphics require generating automatic content at runtime. TV and movies, console and computer games, simulation and training applications, and massive on-line games also require large numbers of object models and complex simulations. Automatic content generation systems are also useful to create a large number of members of the same class of an object with unique attributes, thus producing more realistic scenes. With the advent of high-definition displays, simulation and game applications also require highly detailed models.

In many situations, it is not enough to procedurally generate the geometric models of the actors in the scene. And it is not practical to create their animations by hand, so automatic modeling behavior is another problem to solve.

Our goal in this paper is to provide a unified approach to the generation of models for simulation and computer games. To achieve this goal we implement a system that combines procedural modeling with scripting and object-oriented programming. Procedural models may be of many different kinds: fractals, particle systems, grammar-based systems, etc. Ours are based on L-systems, but it can be used to implement all the other models. Our system supports features like parameterized, stochastic, context-sensitive and open L-systems.

Moreover, we want our system to be as flexible as possible, and to allow the user to embed it in her own applications. Thus, we provide a method to use our procedural engines in many application domains. Also, our system is able

M. Bubak et al. (Eds.): ICCS 2008, Part II, LNCS 5102, pp. 106–115, 2008.

to combine different types of objects (both system- and user-provided) within a single framework.

Geometry generators based on L-systems are usually targeted at specific applications with fixed symbols and tools based on LOGO's turtle metaphor. They are highly dependable on the rewriting engine, thus preventing grammar improvement, language extensions, and code reuse. These generators can not generate different models for the same application. They require multiple L-systems that are difficult to integrate within the same application framework.

To overcome these problems we introduce a procedural model generator that is easily extensible and supports different types of representations and application areas. It stems from a generator based on modular L-systems, L-systems that generate models using a rewriting engine witten in C/C++ and using Lua [1] as scripting language for grammar programming.

We show how we can implement procedural, growth, image-based and other types of models using our system. This paper is structured as follows. The next section reviews previous work in modeling and automatic model generation. The following sections present our system implementation and several results obtained with it. We implement three different improvements on L-system: stochastic, context-sensible and open L-systems. Finally, we finish our paper with some conclusions and directions for future work.

2 Background

Procedural generators are typically based on techniques like fractals [2], particle systems [3], and grammar-based systems [4]. One may also find generators of simple primitives, subdivision surfaces, complex geometries [5] and Constructive Solid Geometry. All these generators allow the creation of texture images [6], terrain models, water and rain, hair, and plants and trees, among others.

Historically, the most expressive procedural models have been the grammar-based technique called L-systems. It was introduced by Lindenmayer to model cellular interaction [7]. L-systems use a set of symbols, an axiom and a set of rewriting rules. The axiom is rewritten using the rules. Then, an output symbol string is generated and interpreted. The result of the interpretation is a freshly generated model of a tree, a building or any other object.

Initially L-systems were used to create plant ecosystems [4]. Subsequently, they have been used for shell texturing [8], virtual urban landscaping [9],[10],[11], and geometry mesh generation [5]. L-systems have also been used for behavior modeling.

Early L-systems were later modified to improve their expressiveness. First, they were parameterized, allowing arithmetic and boolean expressions on the parameters during the rewriting process. Later, stochastic L-systems introduced random variables into parameter expressions to support modeling the randomness of natural species [4]. Finally, context-sensitive rules and external feedback were added to L-systems to support interaction among generated objects and

between objects and their environment [12]. These systems are all based on the turtle metaphor [6].

Recent improvements on L-systems include FL-systems and the L+C language. In the L+C language the symbols are C data structures and the right-hand sides of the rules are C functions from user developed libraries [13]. This improves on earlier systems by adding computation and data management to the rewriting process. Alternatively, FL-systems are L-systems that do not use the turtle metaphor [14]. Instead, they interpret the symbols of the derived string as function calls that can generate any geometry type. FL-systems have been used to generate VRML buildings.

3 Procedural Modeling and L-Systems

A general modeling system must support building many objects of many different types like, for example, crowd models made of people and city models made of streets and buildings. People may be represented using the same geometry, but each actual person should have slightly different properties, appearances and behaviors. Modeling such objects and behaviors by hand is impractical due to their complexity and large number of instances. We need systems to generate models automatically. Procedural modeling has been successfuly used to generate multiple instances of a same class of models.

Our system can generate procedural models and behaviors of many kinds. It was originally developed to generate geometry using L-systems [15]. We have now extended it to generate image-based, grammar-based, and growth-based models as well as behaviors. In this paper we will show how to implement in our system stochastic and context-sensitive L-systems, as well as systems that can interact with their environment.

The system's interface and programming are based on Lua [1], a scripting language. Lua is used to handle all the higher-level elements of the modeling like: rule definition, user code, plugins, ... Additionally, lower-level objects are implemented in C/C++ and bound to Lua elements. These objects are loaded during the initialization of the system. Our system includes the classes used for basic graphics modeling, and the framework to allow the user to provide his own classes.

This was designed with reusability in mind: objects, rules and bindings are organized in plugins that may be selectively loaded depending on the application. Object modeling is decoupled from string rewriting and offers a flexibility unavailable in other systems.

To generate a procedural model and/or behavior, the user must provide an axiom and a set of rules written in the scripting language. These rules can use both the modeling objects provided by the system as well as custom, user-provided modeling objects. Currently our system provides objects like extruders, metaball generators, 3D line drawers and geometry generators based on Euler operators.

Rewriting and interpretation rules are applied to the axiom and the subsequently derived strings. Internally the process may load dynamic objects, run code to create object instances, call object methods, and access and possibly modify the objects' and the environment's states. During the derivation process, any object in the system can use any other object, both system-provided or user-provided. This is why our L-systems are more flexible and more expressive than previous ones.

For example, our system supports the same features as FL-systems and the L+C language: we generate geometry without the turtle metaphor, we include a complete and extensible rewriting engine, and we allow rules that include arithmetic and boolean expressions, flow control statements and other imperative language structures. Using a scripting language allows us to perform fast prototyping and avoids the inconvenience derived of the compiling/linking process. Since we use plugins and C/C++ bindings to handle objects that are instances of Object Oriented Programming classes, we offer more expressiveness than the L+C system.

4 Derivation Engine and Programming

We illustrate the features of our approach by describing its basic execution and a few application examples. To create a model or behavior we need to select the supporting classes. These classes can be already available in the system or they have to be written by the user. Then we use plugins to load them into the system, we instantiate the required objects and we create the system's global state. Finally, the user has to provide the axiom and the rules (in Lua) that will control the derivation of the model.

Currently, our system only implements a generic L-system deriving engine. Other engines are planned to be implemented, like genetic algorithms, recursive systems, etc. Our engine takes an L-system made of a set of symbols, a symbol called axiom, and the two sets of rewriting and interpretation rules. Then it alternatively and repeatedly applies rewriting and interpretation rules to the axiom and its derived strings, thus obtaining new derived strings like any other L-system. The difference is what happens during rewriting and interpretation.

Our system is different because each time a rule is applied, a set of C/C++ code may be executed. Rewriting rules modify the derivation string without changing the C/C++ objects' state and the system's global state. Interpretation rules change the objects' state without modifying the derivation string.

Both types of rules have a left-hand side (LHS) and a right-hand side (RHS). The LHS has the form $AB < S > DE$ where S is the symbol being rewritten and AB and DE are the left and right contexts, respectively. These contexts are optional. To match a rule we compare the rewriting string with the LHS's symbol and its contexts. If the rule matches we run its RHS's associated code, a Lua function that computes the result of the rewriting process or does the interpretation.

5 Generating Models and Behaviors

To illustrate the power of our approach we show how to generate models and behaviors based on three types of improved L-systems. First, we implement stochastic L-systems, a kind of parametric L-system whose parameters may include random variables [4]. Then, we show how our system supports context-sensitive L-Systems, an improvement that allows modeling interaction of an object with itself. Finally, we implement *open L-systems*, an extension of context-sensitive L-systems that allows modeling objects and their interactions with each other and the environment [12].

5.1 Stochastic L-Systems

We implement stochastic L-systems by allowing parameters containing random variables. Fig. 1 shows code to generate the set of three buildings in the back of Fig. 3. Symbol *City* is rewritten as itself with one building less, followed by a translation and a *Building* symbol. *Building* is rewritten as a set of symbols that represent the floor, the columns and the roof of a building. The interpretation rules for those symbols generate geometry using a 3D extruder implemented in C/C++. The extruder can create 3D objects by extruding a given 2D shape and a path.

```
obj:RRule("City", function(c)
    if c[1] > 0 then
        return {
            City{c[1]-1},
            T{building_column_separation*building_width_columns+10,0},
            Building{},
        }
    end
end)
```

Fig. 1. A rule for creating a city made of copies of the same building

The code of Fig. 1 is deterministic and always generates the same set of buildings. To generate different types of buildings we parameterize the *Building* symbol with random variables representing building size, number of columns, number of steps, etc. Depending on the values taken by the random variables, different number of symbols will be created during rewriting. For example, building and column dimensions will determine how many columns are generated for each instance of a building. This is illustrated in the front row of buildings of Fig. 3.

In practice, adding this stochastic behavior requires adding random number generator calls to the rules' code (see Fig. 2). Fig. 3 shows an scene generated by this code. Note that we do not have to change our rewriting engine to support the improved L-System.

```
obj:RRule("RandCity", function(c)
    if c[1] > 0 then
        return {
            RandCity{c[1]-1},
            T{building_column_separation*building_width_columns+10,0},
            Building{
                width = rand_int(5)+3,
                length = rand_int(8)+4,
                column_height = rand_range(5,12),
                column_separation = rand_range(5,12),
                roof_height = rand_range(2,4),
                steps = rand_int(4) + 1
            },
        }
    end
end)
```

Fig. 2. A rule for creating a city made of different, randomly-defined buildings

5.2 Context-Sensitive L-Systems

The modeling of fireworks is another example that illustrates both model and behavior generation. We use it to describe the context-sensitive L-system feature of our system. We start with a C++ line generator object that creates a line given a color and two 3D points. Then, we define a L-system with five symbols: A, B, E, F, and L.

Fig. 4 shows how the symbols are used to generate the fireworks. Symbol A is responsible for creating the raising tail. Symbol B defines the beginning of the raising tail and symbol L defines one of its segments. The end of the tail and its explosion are represented by symbol E, which is rewritten with a number of arms (symbol F) in random directions.

Fig. 4 bottom shows an example derivation. When L is dim it becomes a B. When two Bs are together, the first one is deleted using a context-sensitive rule, thus eliminating unnecessary symbols and speeding up derivation.

Symbols contain parameters representing position, direction, timestamp, etc. (parameters have been removed for clarity). They are used to compute the parabolas of the fireworks. They can also be used to add wind and gravity effects and to change the speed of the simulation. Fig. 5 shows three frames of an animation showing the output of the L-System. A video of the animated fireworks can be found in [16]. Note that the particle model and its behavior are both generated using a simple 3D line generator together with an L-system of a few rules. This illustrates the expressiveness of our system.

5.3 Open L-Systems

Open L-systems allow modeling of objects that interact with other objects and/or the environment. We are primarily interested in spatial interactions, even

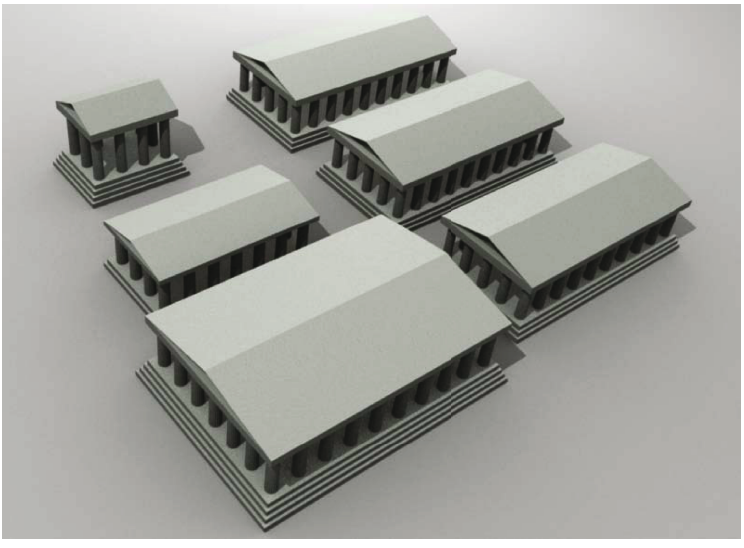

Fig. 3. In the back three copies of the same building are shown, generated with the rule of Fig. 1. The buildings in the front are generated with the rule of Fig. 2. Note how a single parameterized rule can be used to generate different building instances.

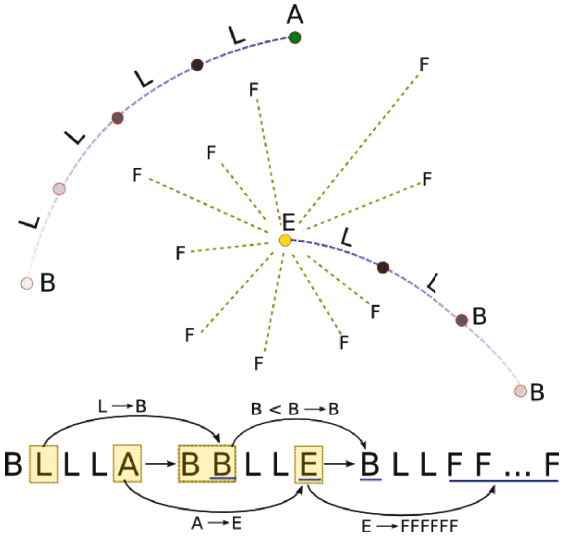

Fig. 4. Top: the raising tail of the palm grows from B to A, in segments defined by L. Middle: At the top of the palm, symbol E fires the F arms in random directions. Bottom: derivation example.

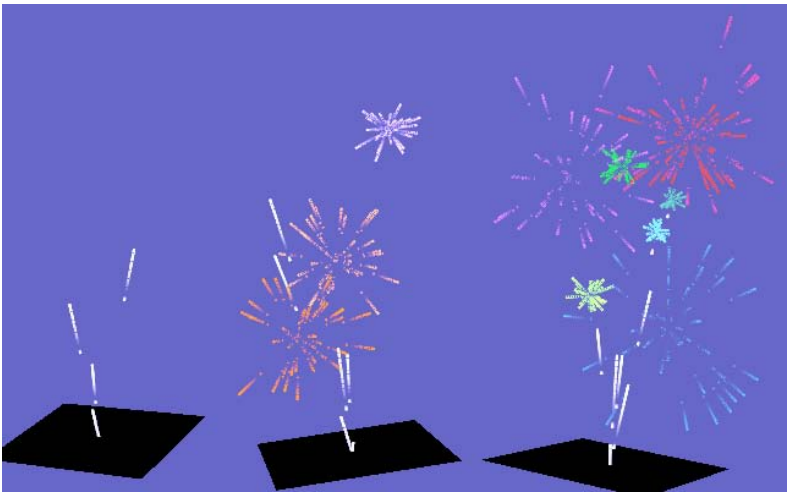

Fig. 5. Three frames of the firework animation generated using our context-sensitive L-system

if they are due to non-spatial issues, like plants growing for light or bacteria fighting for food.

In this section we present an example of an open L-system. The environment in represented by a texture, in which the user has marked several target regions with a special color. The axiom generates a number of autonomous explorers in random positions of the environment. The goal of these explorers is to find a target region and grow a plant. There are two restrictions: (i) only one explorer or one plant can be at any given cell at any given time, and (ii) no explorer can go beyond the limits of the environment.

The explorer is parameterized by a two properties: position and orientation. With a single rewriting rule, the explorer checks the color of its position in the environment map. If that color is the target color then, it is rewritten as a plant, else it updates its position. In each step, the explorer decides randomly if it advances using its current orientation, or it changes it. The explorer can not

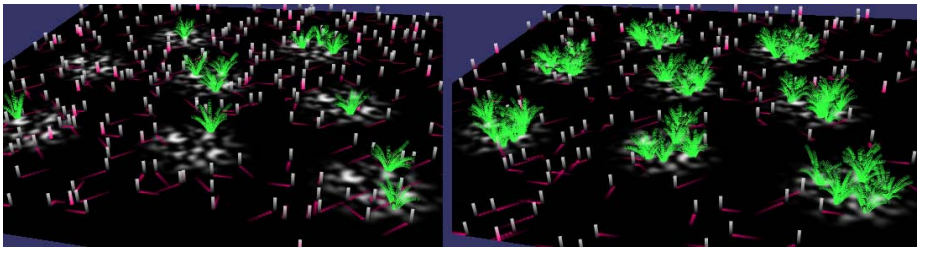

Fig. 6. Two frames of the animation showing the explorer's behavior

advance if the next position is outside of the map or there is another explorer or plant in that position.

Fig. 6 shows two frames of an animation created using this rules. In this example, two modeling objects are used: the image manipulator, and the line generator used in the fireworks example. The image manipulator is an object that is able to load images, and read and write pixels from that image. Both objects are used simultaneously, and they communicate (the explorers' traces are drawn onto the environment map, and the explorer check that map to decide whether they can move in certain direction).

6 Conclusions and Future Work

We present in this paper a new approach to procedural model and behavior generation. We propose a highly flexible and expressive tool to build L-systems using a scripting language and specifying rule semantics with an imperative object-oriented language. Our system combines the power of C/C++ objects with the simplicity and immediacy of Lua scripting.

We show how our tool can be used to implement stochastic, context-sensitive and open L-systems. We show how different types of objects can be combined to generate geometry (buildings), images, and behaviors (explorers, fireworks). Our system can be applied to virtually any Computer Graphics related area: landscape modeling, image-based modeling, and modeling of population behavior and other phenomena.

We expect to increase the functionality of our system by adding tools to generate models and behaviors based on: fractals and other iterative and recursive functions, genetic algorithms, growth models, particle systems and certain physics-based processes. We then want to apply it to the generation of virtual worlds for different types of games and for simulation and training applications.

Acknowledgments. This work was partially supported by grant TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science and by a doctoral Fellowship of the Valencian State Government.

References

1. Ierusalimschy, R.: Programming in Lua, 2nd edn. Lua.org (2006)
2. Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freeman, New York (1982)
3. Reeves, W.T., Blau, R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. In: SIGGRAPH 1985: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 313–322. ACM Press, New York (1985)
4. Prusinkiewicz, P., Lindenmayer, A.: The algorithmic beauty of plants. Springer, New York (1990)

5. Tobler, R.F., Maierhofer, S., Wilkie, A.: Mesh-based parametrized L-systems and generalized subdivision for generating complex geometry. International Journal of Shape Modeling 8(2) (2002)
6. Ebert, D., Musgrave, F.K., Peachey, D., Perlin, K., Worley, S.: Texturing & Modeling: A Procedural Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
7. Lindenmayer, A.: Mathematical models for cellular interaction in development, parts I and II. Journal of Theoretical Biology (18), 280–315 (1968)
8. Fowler, D.R., Meinhardt, H., Prusinkiewicz, P.: Modeling seashells. Computer Graphics 26(2), 379–387 (1992)
9. Parish, Y.I.H., Müller, P.: Procedural modeling of cities. In: SIGGRAPH 2001: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 301–308. ACM Press, New York (2001)
10. Hahn, E., Bose, P., Whitehead, A.: Persistent realtime building interior generation. In: sandbox 2006: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, pp. 179–186. ACM, New York (2006)
11. Müller, P., Wonka, P., Haegler, S., Ulmer, A., Gool, L.V.: Procedural modeling of buildings 25(3), 614–623 (2006)
12. Měch, R., Prusinkiewicz, P.: Visual models of plants interacting with their environment. In: SIGGRAPH 1996: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 397–410. ACM, New York (1996)
13. Karwowski, R., Prusinkiewicz, P.: Design and implementation of the L+C modeling language. Electronic Notes in Theoretical Computer Science 86(2), 141–159 (2003)
14. Marvie, J.E., Perret, J., Bouatouch, K.: The FL-system: a functional L-system for procedural geometric modeling. The Visual Computer 21(5), 329–339 (2005)
15. Hidalgo, J., Camahort, E., Abad, F., Vivo, R.: Modular l-systems: Generating procedural models using an integrated approach. In: ESM 2007: Proceedings of the 2007 European Simulation and Modeling Conference, EUROSIS-ETI, pp. 514–518 (2007)
16. http://www.sig.upv.es/papers/cggm08