INCREMENTAL RULES FOR GROWING PLANTS

Andrew Davison
Department of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand
E-mail: dandrew@ratree.psu.ac.th

KEYWORDS

3D worlds, rules-based programming, animation tools, 3D authoring tools, toolboxes for moving graphics.

ABSTRACT

L-systems are widely used for plant modeling and simulation, with remarkable results. However, we argue that the mathematical formalism underpinning L-systems encourages inefficient rendering of plants which grow and change over time. We propose new types of rules which emphasize the incremental nature of change in a plant's elements, and highlight an element's relationships with other components (e.g. a plant limb has a parent, children, and occurs at a certain level in the overall structure). We have implemented a Java 3D prototype using this approach, and compare it with code using a standard L-system.

INTRODUCTION

Lindenmayer systems (L-systems), consisting of rewrite rules, have been widely used for plant modeling and simulation, due to the direct mapping between the string expansions of the rule system and a visual representation of a plant (Prusinkiewicz and Lindenmayer, 1990; Prusinkiewicz and Hanan 1990; Prusinkiewicz et al. 1993; Prusinkiewicz et al. 1999). For example, the following bracketed L-system contains a single start string 'F', and the rewrite rule:

$$F \rightarrow F\ [-F]\ F\ [+F]\ F$$

The visual characterization is obtained by thinking of each 'F' symbol as a *limb* (or *module*) of the plant. The bracket notation can be viewed as a branch-creation operator; the '-' as a rotation to the right for the branch, '+' a left rotation.

Since each limb is an 'F' symbol, rewriting can continue, creating longer strings, and more complex plant-like shapes.

Each rewrite causes all the limbs (modules) of the current tree to be replaced in a parallel derivation step, reflecting the simultaneous passage of time in all parts of the tree. Time is viewed as discrete, represented by the sequence of derivation steps.

GROWTH AND L-SYSTEMS

Our application domain is a networked 3D virtual world, which changes over time – day turns to night, and plants/trees grow. The scenery is not photo-realistic; more emphasis is placed on the fast rendering of a large number of relatively simple shapes representing different kinds of trees. At certain times, these trees need to grow extremely rapidly.

The application is implemented in Java 3D, a high-level 3D graphics library for Java (see `http://java.sun.com/products/java-media/3D/`). In the first version of the application, a L-system generated strings which were passed to a rendering component to be turned into trees made from groups of coloured cylinders and other objects. Growth was represented by having the L-system pass the current strings to the renderer at the end of each derivation step. The renderer would dispose of the old 3D trees, generate new trees using the strings, and add them to the scene.

This approach proved to be very slow, and quickly ran out of memory when more than about ten medium-size trees were placed in the scene. The slowness was partly caused by Java 3D's slow run-time removal and addition of scene objects, and the subsequent garbage collection of the discarded trees. Also, as the L-system strings became larger, the renderer required increasingly large amounts of memory to recursively parse the strings and create the 3D shapes.

Part of the problem is due to Java 3D, but we also identified four problems with the L-system: two fundamental ones present in the rule formalism, and two minor ones that are language-related. We discuss the two serious issues first, then the lesser two.

Rewriting = Replacement

Each rewrite of a L-system string creates a more complex tree, but it is hard to see how the new tree has 'grown' out of the simpler one. For instance, what part of the current tree is new wood, which is old wood that has grown a little?

A L-system represents growth as a new structure completely replacing the old one. That is unimportant when the structure is a mathematical abstraction, but has serious consequences when implementing a growth algorithm. The natural approach, and the most disastrous from an efficiency point of view, is to discard the current structure at the start of a rewrite and generate a new one matching the new string expansion. There is no simple alternative to this since the L-system does not distinguish between old elements (either changed or unchanged) in the structure and the new parts.

No Tree Relationships

Another drawback of the L-system notation is its lack of tree nomenclature. For example, it is not possible to talk about the parent of a node, its children, or its level in the tree. To be fair, some of these capabilities can be programmed by using parameterized L-system rules. However, we believe that a production system for plant modeling should contain intrinsic ways of talking about the branching structure that it represents.

Locating Limb (Module) State

Limb state includes information such as the present length of a limb, its current colour, and its age. Parameterized L-systems handle state by adding additional parameters to the rules, which tends to lead to large rules. Arguably, this solution places the data in the wrong place: state details for each limb should be located *inside* the particular limb rather than in the rules which are applied to all the limbs. This is really an argument for an object-based view of limbs, rather than a procedural one centered around the rules. Benefits include the ability to hide state, improved modularization, and cleaner abstractions.

Rule Reuse

Large groups of L-systems rules often contain very similar rules for the different plant elements. For example, most types of limbs will grow for a period of time (represented by a recursive, parameterized rule), followed by the appearance of child limbs as branches sprout (this is often called a decomposition rule).

Once an object view of limbs is utilized, it follows that plant node types should be represented by classes with their own data, methods, and rule behavior. Commonalties between the classes, whether in their state or rules, can be dealt with by subclassing.

INCREMENTAL RULES

Our principal change to the L-system formalism is to utilize rules which specify rewrites as *incremental changes* to existing limbs, such as a gradual increase in length or a deepening colour. Child branches can be spawned, but are defined in terms of how they are added to their parent limb. This requires the introduction of a tree notation so that the parent-child relationships can be stated.

We implemented our ideas in Java, so gaining the advantages of OOP. In our prototype system, a limb is represented by a `TreeLimb` class, which has over 20 public methods, roughly classified into five groups:

- scaling of the cylinder's radius or length;
- colour adjustment;
- parent and children methods;
- leaves-related;
- various others (e.g. accessing the limb's current age).

The system is activated every 100ms (the time interval between rewrites), and applies its rules to all the `TreeLimb` objects, affecting a parallel rewrite analogous to the L-system model. The difference lies in the incremental nature of the rules.

The rules have an if-then form, where the action is only carried out if the conditions evaluate true for the current limb.

The rest of this section contains descriptions of the simple ‘length’ and ‘thickness’ rules, and the slightly more complex ‘child limbs spawning’ rule.

The ‘length’ rule incrementally increases the length of a limb up to a maximum of about 1 unit:

```
if ((limb.getLength() < 1.0f) &&
      !limb.hasLeaves())
   limb.scaleLength(1.1f);
```

`limb` is the current `TreeLimb` object under consideration. The `hasLeaves()` part of the condition stops branches from growing any longer once they have leaves.

The 'thickness' rule mandates how a limb’s thickness should change:

```
if ((limb.getRadius() <=
     (-0.05f*limb.getLevel()+0.25f))
                   && !limb.hasLeaves())
   limb.scaleRadius(1.05f);
```

The equation `-0.05*limb.getLevel()+0.25` relates the maximum radius to the limb's level. For example, a limb growing directly out of the ground (level == 1) can have a larger maximum radius than a branch higher up the tree. This means that branches will get less thick the higher up the tree they appear, as in nature.

The ‘child limbs spawning’ rule creates at most two child limbs:

```
if ((limb.getAge() == 5) &&
     (treeLimbs.size() <= 256) &&
     !limb.hasLeaves() &&
     (limb.getLevel() < 10)) {
   if (Math.random() < 0.85)
     makeChild(randomRange(10,30), limb);
   if (Math.random() < 0.85)
     makeChild(randomRange(-30,-10), limb);
}
```

The four conditions only permit child limbs to appear if the parent is at least 5 time intervals old, the total number of limbs in the scene is less or equal to 256, the parent has no leaves, and the branch isn't too far up the tree.

`Math.random()` is employed to make it less certain that two children will be spawned. `randomRange()` returns a random number (in this case, an angle) in the specified range.

`makeChild()`'s definition:

```
private void makeChild(double angle,
                        TreeLimb par)
{ TransformGroup startLimbTG =
               par.getEndLimbTG();
  int axis = (Math.random() < 0.5) ?
                Z_AXIS : X_AXIS;
  TreeLimb child = new TreeLimb(axis,
              angle, 0.05f, 0.5f,
              startLimbTG, par);
  treeLimbs.add(child);
   // add new limb to tree limbs list
}
```

The first line gets the 'end point' of the parent limb, which becomes the place where the child is connected. `Math.random()` is used to randomize the child's orientation axis.

Figure 1 shows a sequence of screen shots of the application. Five trees grow from saplings, young green shoots turn brown, leaves sprout, all taking place over a period of a few seconds.

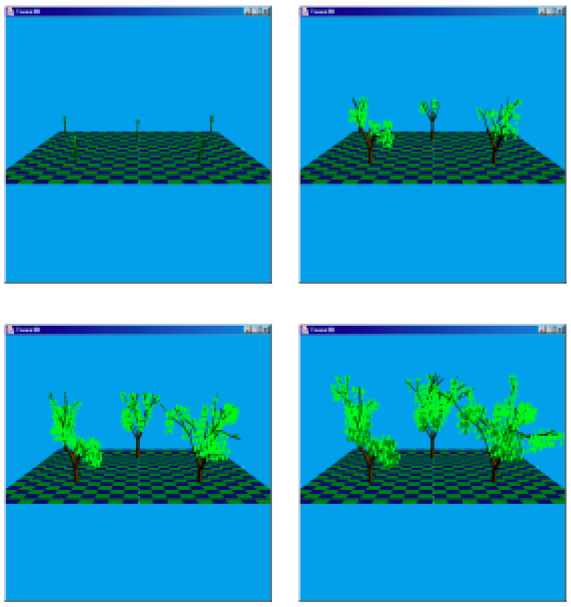

Figure 1: Growing Trees

DISCUSSION

The application carries out very little garbage collection, in contrast to the original L-system-based code, because the trees are not being repeatedly regenerated. In fact, no tree limbs are discarded at all.

The application's main control structure is a time-triggered loop through all the limb objects, applying the rules to each one. The original code uses recursion to generate all the limbs in each tree from scratch in every derivation step. It is hardly surprising that the new application has a much faster rendering time (sometimes twice as fast).

Over 4000 limbs can be created before the application needs additional heap space, compared to tens of limbs in the original code. This is due to the reduced garbage collection needs and the use of looping rather than recursion in the rendering.

Time is represented discretely, and growth is defined in incremental steps rather than as differential equations (e.g., as in dL-systems). The primary reasons for this choice was to make rule definition simpler: most users find the specification of differentials rather difficult. This approach also avoids the need for a run-time solver for the equations.

Java offers the advantages of OOP, and each limb is represented by its own object. The principal limb class is `TreeLimb`, which can be subclassed easily.

A drawback of our application is the complexity of the rules which refer to graphical elements of the 3D models. For example, `makeChild()` utilizes a Java 3D `TransformGroup` node to connect a child to its parent. This suggests the need for a higher-level notation which hides connection details.

The application shown in Figure 1, together with a more detailed explanation of its implementation can be found at `http://fivedots.coe.psu.ac.th/~ad/jg/ch178`.

REFERENCES

Prusinkiewicz, P. and Hanan, J. 1990. "Visualization of Botanical Structures and Processes using Parametric L-Systems", In *Scientific Visualization and Graphics Simulation*, D. Thalmann (ed.), pp.183-201, John Wiley & Sons.

Prusinkiewicz, P., Hanan, J., and Mech, R. 1999. "An L-System Plant Modeling Language", In *Proc. of the Int. Workshop AGTIVE'99*, M. Nagl, A. Schuerr and M. Muench (eds), Kerkrade, The Netherlands, September, LNCS 1779, Springer, pp.395-410.

Prusinkiewicz, P., Hammel, M.S., and Mjolsness, E. 1993. "Animation of Plant Development", *Computer Graphics*, Vol. 27, No. 3, pp. 351-360.

Prusinkiewicz, P. and Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*, Springer-Verlag, NY.