Evolving 3D Models of Trees Using Genetic Programming

Guillaume Poncin
Gates Building 3-362
Stanford University
Stanford, California 94305
gponcin@cs.stanford.edu

ABSTRACT

This paper describes a technique to evolve 3D models of plants based on deterministic automata. It uses cellular encoding to obtain these automata from trees evolved in standard genetic programming software. The fitness is a combination of high-level characteristics of the model, including overall shape, number of branches and geometric characteristics. We introduce a grid-based constraint evaluation as a way for the designer to shape a tree. This approach can be extended to modeling more than plants.

1. Introduction

As 3D Graphics push the limits of virtual reality in movies and games, it is more and more the case that 3D artists want to model complex outdoors scene including trees and biological structures. These plants have to look realistic but also have to integrate smoothly in their environment. Conventional approaches can produce reasonably detailed trees by randomly varying parameters on a hand-modeled template. However, they usually require modeling all the trees individually.

It is desirable, though, not to ask human designers to create templates branch by branch but to let them specify high-level descriptions such as density of leaves and overall shape. In addition, realistic scenes often require generating plants that respond to constraints imposed by the environment. For instance, a large object could block a tree from developing on one side, or ivy will naturally grow against a wall.

We will describe in this paper a novel technique based on genetic programming to produce very diverse models. They all try to match the criterions specified by the designer as well as possible. Section 2 reviews previous work in the field and provides background on automata. Section 3 of this paper states the problem, and section 4 describes the method. In section 5 and 6, we state and discuss the results. Finally, section 7 states the conclusion. Section 8 discusses future work.

2. Background

2.1. L-Systems

A common way of describing plants is through Lindenmayer systems also called L-systems (Lindenmayer 1968, Prusinkiewicz and Lindenmayer 1990). This term regroups a class of descriptive grammars where rewriting rules are used to bring an individual from one time step to the next. Once interpreted they act as LOGO commands, driving a turtle to draw the geometry of the trees and occasionally add flowers or leaves. Several attempts have been made to use genetic algorithms to evolve L-systems: Jacob (1995), Mock (1998) and Ochoa (1998).

2.2. Automata

The approach we chose is not based on L-systems and rewriting rules but on a similar construct developed currently by N. Lambert at Stanford University and inspired from the work of P. de Reffye (1988).

In this model, the plant is seen as a directed graph, where each node represents a physical part of the plant, like an aggregate of cells. Each node contains the type of geometry, such as piece of stem or piece of leaf, and a list of automata that are in charge of the growth process. For each time step that corresponds to a cell growth, the nodes are executed in parallel. To execute a node we execute the associated list of automata in order (see Figure 1 for a simple example).

We use a simpler version of the above algorithm. In our system, each automaton has a local geometric frame plus a set of variables it can use as counters or for any other purpose. Two variables are privileged; they designate the current length and thickness of the portion of tree we are building. The possible states of an automaton are:

- Conditionals (if-then-else), test automaton variables against fixed values to determine the execution path
- `RotateH`, `RotateL`, `RotateU`, rotate the local HLU frame of the automaton along one axis.
- `Wait`, pauses the execution of the automaton
- `Killme`, ends the automaton execution
- `CreateAutomaton`, creates a new instance of an automaton designed by its number (it could also designate a leaf or a flower)
- `Assign`, assigns a value to one of the variable.

- `Add`, `Multiply`, add or multiply a variable with a fixed value
- `Push`, `Pop`, save the local coordinates and variables of the automaton and restore them.

When interpreting a set of automata to in order to grow a plant, we start by instantiating the root automaton. Then, we execute the following algorithm:

```
for each iteration do
  for each automaton still alive do
    repeat
      currentstate:=currentstate.next
      if (currentstate == Killme
            OR end of automaton reached)
        [Kill this automaton]
      end if
      [Execute currentstate,
       Possibly instantiate new automata]
    until (currentstate == Wait)
    [Move one step in H direction]
    [Record new position of the turtle]
  end for
end for
```

Each automaton keeps a local turtle to draw its own branch. After one step, all the automata go forward in the H direction of their local frame and their new positions are recorded. To draw the skeleton of the tree, we simply join each new position to the previous position as we run the algorithm. The growth ends when all automata are dead or after a maximum number of iterations.

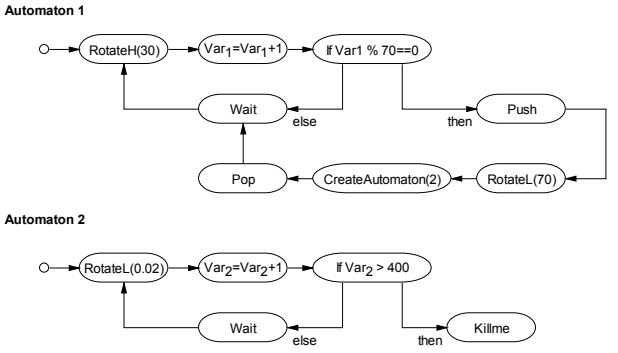

Figure 1. Two automata to describe a tree

Figure 1 shows a set of two automata resulting in the helix shaped tree of Figure 2. We used 1000 iterations to produce this very simple example.

For the first automaton, one time step ends when we return to the `Wait` state. The first `RotateH` performs the rotation around the axis of the trunk. The `else` branch of the `if` state does not do anything, which means we just move the turtle forward a little bit every time. Every 70 time steps, the `then` loop starts a new branch at a 70 degrees angle from the trunk by creating a new Automaton 2. Since we have a `Push/Pop` pair, the coordinate system of Automaton 1 is not affected by this creation.

Automaton 2 is responsible for the branches: it just moves forward for 400 iterations. The `RotateL` state slightly bends the branch downwards. Each branch is built by a different instance of Automaton 2. They all work in parallel to build the tree.

The two variables *`Var1`* and *`Var2`*, initialized to zero at the beginning of the simulation, serve as counters and are updated in a while-loop fashion.

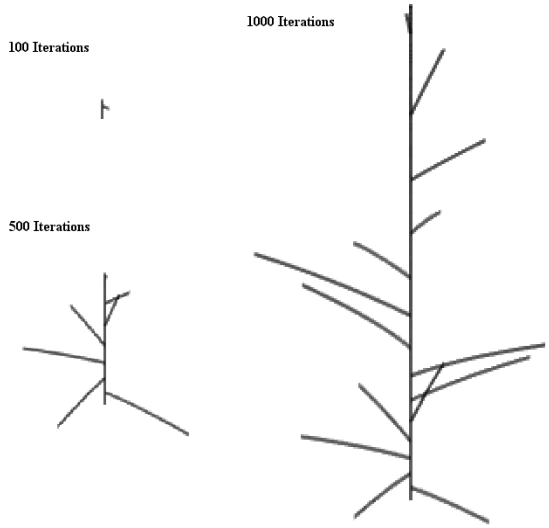

Figure 2. 3D helix produced by the automata in Fig. 1

This scheme can be easily extended with new operators to account for additional effects. The main problem is that it is extremely difficult to design complex plants by hand. We introduce below a mechanical way to build the automata.

3. Statement of the problem

Given a high level description of the shape and density of a tree, we want to be able to obtain a 3D model that looks reasonably like a tree and matches the designer description.

The specifications that we are given can be:

- *Shape,* surface where the endpoints of all the branches should lie
- *Number of branches*, approximate number of branches and optionally sub-branches that we want
- *Level of branch recursion,* 1 corresponds to just having the trunk, 2 to having branches, 3 to having sub-branches and so on…
- *Presence or absence of leaves*
- *Favorable / forbidden zones*, regions of the space where we want more or fewer branches/leaves

If some of the characteristics are not specifically defined, we consider them as free variables that the algorithm can experiment with to come up with the best possible model.

Additional implicit constraints need to be specified in order to get tree-like shapes and not random geometric

forms. We want the endpoints of the branches to be as distant as possible from each other. We also impose that the tree grows upwards in the beginning and that the trunk automaton can only create branch automata, the branch automata can only create sub-branch automata, etc. Leaves can only be added by the last level of sub-branches.

4. Methods

We make use of genetic programming techniques (Koza 1992) to evolve trees in the form of automata. Starting from simple individuals it is possible to obtain refined individuals that fit the description of a particular tree.

4.1. From an individual to a model of tree

We have seen how to represent trees as automata. But how can we make this representation friendly to genetic programming? Figure 3 shows the data flow from individuals to the fitness evaluation. What we call *individual* is a tree with operators, as usually employed in genetic programming algorithms.

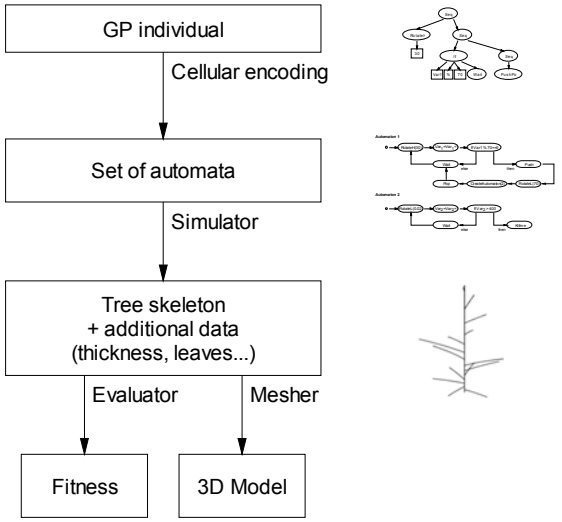

Figure 3. Operations leading from a GP individual to a 3D model with a fitness

We explain in section 4.2 how to go from the individual to the set of automata. The details of the genetic programming evolution are given in section 4.3. We cover the evaluation of the fitness in section 4.4. Finally, we give more insight on the meshing of a tree skeleton into a full 3D model in section 4.5.

4.2. Cellular Encoding

Individuals are represented by strongly typed trees containing several automata. Each automaton, one for the trunk and one for each level of branches, is described using cellular encoding techniques, similar to those described by F. Gruaut (1994) for neural networks.

We employ a collection of operators in our GP trees. All of them are of the same type *`TreeNode`*. We have 3 types for the input parameters of these operators: variable number, value between *-10* and *+10* and automaton number. We group the operators in three classes.

Some of the operators are in direct correspondence with the automata operators defined earlier. They form our first class of operators:

- `Rotate_H`, `Rotate_L`, `Rotate_U` each take a value parameter
- `Wait`, *`Killme`* with no parameter
- *`CreateAutomaton`* taking an automaton number parameter
- *`Assign`*, *`Add`*, *`Multiply`* with a variable number and a value parameter.

We can see on Figure 4 an example of the transformation from the tree representation to the automaton representation. Each state in the automaton has an incoming transition and an outgoing transition.

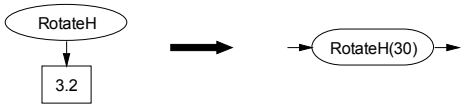

Figure 4. A simple operator made into an automaton

The second class of operators contains architecture modifying functions (see example in Figure 5):

- `Seq` juxtaposes two `TreeNodes` in the automaton. During the construction of the automaton we simply link the two states by a pointer.
- `Loop` creates a loop back transition in the automaton. There is no outgoing transition in this case, which may be a problem if we want to juxtapose a loop with some other state. That is why we did not allow this construct in the end.
- `If` allows for conditionals. The `then` transition is considered as the outgoing transition of this state.

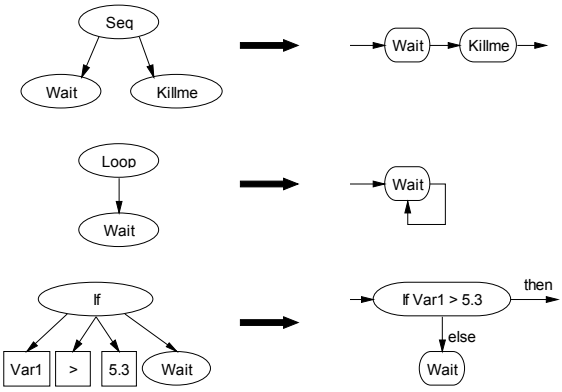

Figure 5. Complex operators translated into automata. Each `Wait` or `Killme` could be replaced by a subtree.

The third class of operators is formed of compounds based on the previous operators. They help GP to find a solution in a reasonable amount of time by introducing

interesting sub-blocks that would otherwise need a fairly large quantity of generations to be selected as efficient. In our tests they only appeared in a viable form after 100 generations or so. Given a larger computational power, we should not need these compounds.

- *Push/Pop* embeds a subtree between a Push-Pop pair. It prevents unmatched *Push* and *Pop* states and guarantees we only have well-formed automata.
- *Repeat* allows for the repetition of the same subtree for a number of times given as parameter. The *Count* variable is an additional variable of the automaton, independent for each *Repeat* loop.

Figure 6 gives examples of these compounds.

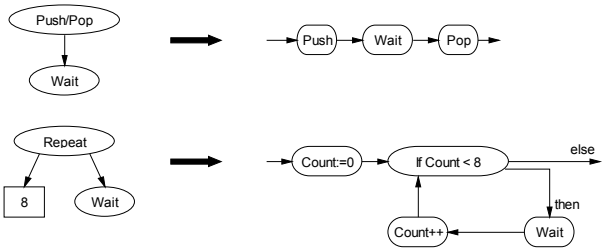

Figure 6. Compound operators.

4.3. Genetic Programming

It is now clear how to get automata from trees of operators. Using cellular encoding, we make sure that the automata we produce are viable and that all their states can be reached. We do not need to perform checks before executing the simulator.

Table 1 shows the characteristics of the genetic programming runs we are doing. We have been trying several combinations of the possible functions and parameters, so this table only reflects one particular run.

Our implementation is based on Lilgp with strong-typing support. The tree-modifying operations such as mutations and crossovers are the standard GP operations for strong-typed trees. We usually have *2%* mutations, *88%* crossover and *10%* reproduction during the breeding.

We do not include automatically defined functions in our runs because it is not clear how that kind of reuse would be helpful to the genetic programming algorithm. After several attempts we did not notice any use of ADFs in the top individuals of a generation. The repeat loop and the separation between several automata running in parallel already guaranties some reuse, which seems sufficient.

Depending on the severity of the constraints, we can obtain acceptable results with 200 individuals and 50 generations or we may need 60000 individuals and 500 generations (see Section 5). One individual is usually made of twenty to five hundred nodes. The average individual we observed is around 100 nodes.

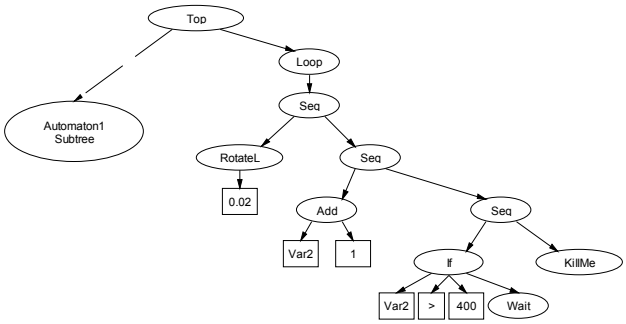

Figure 7. Tree representing the second automaton in figure 1. We did not detail automaton 1.

4.4. Fitness Evaluation

The expressivity of the system of automata that we use is extremely large. Thanks to cellular encoding we can restrict ourselves to the space of fully viable automata. But we want to limit the search space even more, to the shapes that look like trees. That is why we impose the restriction that an automaton can only create an automaton that is considered as a sub-branch, i.e. which appears later in the list of children for the `Top` node. It would not make sense to allow a branch to recreate a trunk, at least in our first model.

The fitness is probably the cornerstone of our system. From a very abstract goal "get a plant that looks nice", we need to set precise goals to GP to get interesting results. We have two classes of terms linearly combined in our fitness expression. Some terms try to solve the problem given by the user. The others are there to make sure we get plant-like shapes. We detail below each component of the fitness.

All the fitness functions we compute lie between *0* and *+infinity*. They are mapped between 0 and 1 in the form of adjusted fitness by Lilgp: *adj. fitness = 1 / (1+std. fitness)*, closer to 1 being better.

4.4.1. Solving the problem

We have three terms to get a plant as close from the goal specifications as possible. All the values in the following formulas are arbitrary but have been shown to work well.

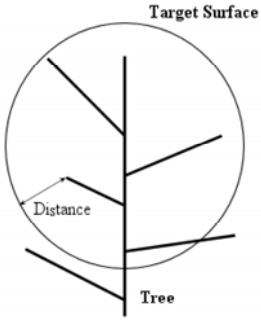

Figure 8. Branch-target distancefor F_{pos}

Table 1. Tableau for a typical plant generation problem

Objective	Find a tree encoding a given number of automata to solve the plant generation problem
Terminal set	`Wait, Killme` for `TreeNode` type of nodes. Also: automaton number, fixed point values between *-10.0* and *+10.0* and variable number (including special variables `Length` and `Thickness`).
Function set	`Top`, to start the tree `RotateH`, `RotateL`, `RotateU`, `CreateAutomaton`, `Assign`, `Add`, `Multiply` `Seq`, `If` and variations on the If operator to handle different types of comparisons `Push/Pop`, `Repeat` and variations on the repeat operator to use the index of the current automaton to initialize the counter.
Fitness case	All branch end points
Raw Fitness	Average for each fitness case of the distance between the endpoint and the target surface
Standardized fitness	Raw Fitness, modified to account for all the additional fitness parameters (see Section 4.4)
Hits	N/A
Wrapper	Simulator and 3D meshing to get a 3D model
Parameters	M = 10000, G = 150 usually. Number of automata to generate, usually 2 or 3.
Success predicate	None… (see Section 4.4)
Rules of construction	• Root is always a `Top` node, which has one child for each automaton we have to generate. • Remainder of the tree does not contain any `Top` nodes • An automaton can only create an automaton using `CreateAutomaton` if the new automaton appears later in the list of children of `Top`

First, the position fitness F_{pos}, is computed as the average over all the endpoints of all the automata instantiated during a simulation. The goal is to come as close as possible from a predefined shape. We are using a sphere in most cases, centered at *(0,200,0)* and of radius *150* in most cases. Figure 8 shows an example of this fitness for one branch.

$$F_{pos} = \underset{endpo\,\mathrm{int}\,s}{Average} \left| 150 - \left\| Pos_{endpo\,\mathrm{int}} - (0,200,0) \right\| \right|$$

The second term F_{Num} is based on the number of endpoints. It is desirable to have a limited number of endpoints; otherwise we end up with computations that take too long. Moreover, this fitness term encourages the creation of branches i.e. the use of Automata 2 by Automata 1. Typically, if we use two automata, we want around 30 branches, and if we use three automata, around 150 sub-branches.

$$F_{num} = \left| 30 - Num_{endpo\,\mathrm{int}\,s} \right|$$

To be more specific, we may define a fitness term like this one for each level of branches.

4.4.2. Making sure we have a tree

Even though we may match the user expectations we have to make sure that we find a tree, which is much more restrictive than the space of possible structure described by our automata.

The main property we want to have is the dispersion of the branches. In nature, most trees have branches all around their trunk, not just in one plane. This can be justified by weight repartition considerations, symmetry in the process that creates the branches, etc. These concepts are hard to incorporate in a fitness evaluation. So we will only consider a function that favors distant endpoints for the branches. It is fairly fast to evaluate and combined to the other terms, we get the expected results.

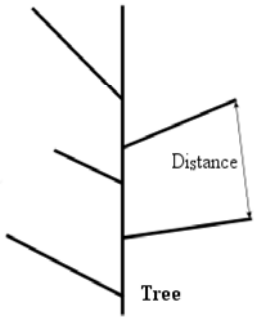

Figure 9. Branch-branch distance for F_{dist}

We have two ways to do the computation: either by evaluating the distance between the endpoints of all the branches F_{dist} or by evaluating the angle between two branches F_{angle} (see Figure 9).

$$F_{dist} = \underset{\substack{A \in endpo\,\mathrm{int}\,s \\ B \in endpo\,\mathrm{int}\,s}}{Average} \left(\frac{200}{\left\| Pos_A - Pos_B \right\| + 5} \right)$$

The expression of F_{angle} is very similar to that of F_{dist} except it is based on the angles. These two fitness

functions share the same issue. They can take up to N^2 comparisons if we want to evaluate each branch against each other, where N is the number of branches. We resort to a randomized evaluation: we only take *max(3N, 10)* pairs of points. Proceeding this way, we get approximately *20%* error. Since the fitness does not need to be exact, the significant speedup is worth the loss in precision.

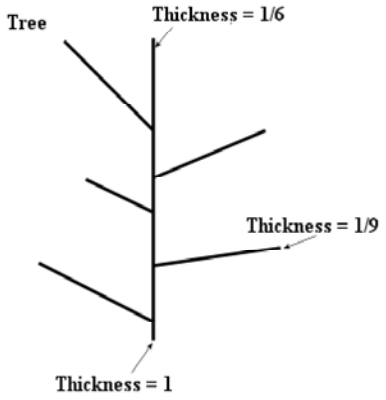

Figure 10. Ideal thickness for F_{thick}

The third term F_{Thick} evaluates how much thinner the ends of the branches are compared to the base of the trunk. This term does not need to be computed if we do not care about the thickness output. The thickness of the trunk is set to *1* at its base (enforced during the creation of the root automaton). Then we may want the trunk to end with fitness *1/6*, first-level branches to end with thickness *1/9*, second-level sub-branches to end with thickness *1/12* and so on (see Figure 10). These numbers are arbitrary and are easily modifiable by the user as an input parameter.

The formula for the fitness F_{thick} is:

$$F_{thick} = \underset{A \in endpoints}{Average} \left| Thick_A - Thick_{reference} \right|$$

Since the thickness is treated as a variable and we have an `add` operator with negative constants, the thickness may end up being negative. We give a penalty every time it occurs.

4.4.3. Combining all the fitness terms

The four fitness terms described previously all have to be combined into one fitness evaluation. We use a linear combination such as:

$$F = w_{pos} \cdot F_{pos} + w_{num} \cdot F_{num} + w_{dist} \cdot F_{dist} + w_{thick} \cdot F_{dist}$$

The weights have been experimentally set to give fast results. The current weights are:

$w_{pos}=5$, $w_{num}=50$, $w_{dist}=500$, $w_{thick}=1000$

They do not exactly bring the four fitness functions on an equality standpoint; rather they force the distance and number of branches to be correctly set first. Only when these two measures approach zero do the other terms become prominent. When this happens, they allow for more accurate matching of the endpoint positions against the target surface and for research of the most balanced fitness in the tree.

4.4.4. Adding constraints

Up until this point, we can get "good-looking" trees in a reasonable amount of time (see Section 5 for actual results), which is only half of our original goal. Recall that we also want to be able to specify constraints on the geometric form of the tree. More than just reaching a surface, we want to be able to set regions where the plant should not be. This could correspond to external objects which prevent the plant from expanding uniformly in every direction (i.e. a wall). We could also specify regions where the plant should try to be, corresponding to well-lit areas in a room for example.

Once again, if we could model the mechanical environment perfectly, as well as the energy process that take place in the cells of the tree, we would automatically attain that kind of result. To keep computation time reasonable, we have to find a simple model that is easy to evaluate and accounts for all of these effects.

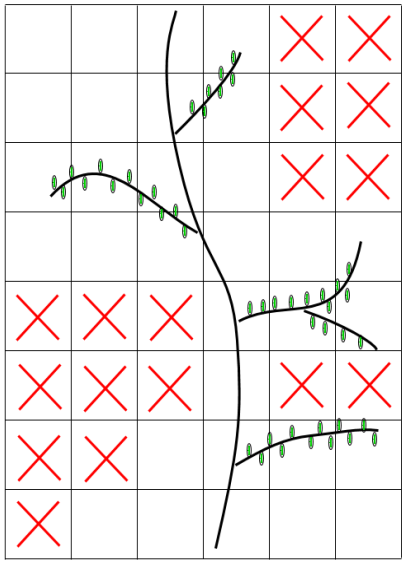

Figure 11. Schematic with a 6x8 grid in 2D. Crossed cells are forbidden: the plant grows around them. Higher leaf densities appear in high-weight cells

One of our major contributions in this paper is a density grid construct. We divide the space around the plant into a grid with a $N = N_x \,.\, N_y \,.\, N_z$ cells. Each cell has a target number of leaves that it should contain. Each cell also has a weight to indicate how important it is: the larger the weight, the more it matters that the plant matches the target number of leaves in this cell. If the weight is negative, it means the cell is forbidden and no leaf should be present. A stronger negative weight gives more importance to a particular forbidden cell.

Leaves are produced by the last level of branches. The target number for each cell helps regulate their density. We derived formulas to express the fitness of a particular cell

F_{cell}. The fitness values for all the cells are averaged to calculate grid fitness. Given a weight w for the cell and a target number of leaves N_{target}, we have for $w > 0$,

$$F_{cell} = \begin{cases} w_{cell} \cdot \left(1 - \frac{\left|N_{leaves} - N_{target}\right|}{8 * N_{goal}}\right) & if \quad w > 0 \\ w_{cell} \cdot \left(1 + \frac{N_{leaves}}{4 * N_{goal}}\right) & if \quad w < 0 \end{cases}$$

These formulas mean that if $w>0$, the fitness is best when the number of leaves is close to the target. If it goes higher than $8 * N_{goal,}$ then we impose a penalty for the cell. If $w<0$, the fitness is best when there is no leaf in the cell. We use $N_{target}=15$.

Since the grid fitness is not critical in creating a viable tree we only let it play a role for the individuals which already have sufficiently good fitness. For the others we set it to its maximum value, which helps reduce computing time.

4.5. Conversion to a 3D model

The fitness is evaluated from the skeleton. If we actually want to display the results or to integrate the plant in a 3D environment we need to transform this skeleton into a shape with textures and volume. A simple path-extrusion is performed: we create cylindrical surface along the trunk and the branches with a thickness given by the data stored with the skeleton. The leaves are generated randomly and added to the model at the positions recorded during the execution of the algorithm. For better results, we may want to impose nearly horizontal leaves. This is not currently included in the fitness, but it is easy to do as a post-processing step, before displaying the model.

5. Results

In the end, the only evaluation that matters is the visual aspect for a human observer, which is difficult to capture with formulas. That is why we do not expect our algorithm to come up with the best tree but rather to propose reasonable solutions to the problem. In this respect, we have generated good results.

5.1 Performance

Performing one generation consists of transforming each tree into automata, running the automata in the simulator and evaluating the fitness for the individual. We also have to breed the population based on the fitness evaluation.

We run our program on Windows or Linux on a 3GHz P4 machine with 1GB of RAM. With 2000 individuals, the breeding phase takes virtually no time. The simulator itself is extremely fast, on the order of 500 individuals per second. The series of operations necessary to transform individuals feed them to the simulator and compute the fitness runs at a rate of 10 to 200 individuals per second depending on the complexity of the population. A typical run of 5000 individuals and 100 generations takes in the order of one hour to complete. The automata are executed with a limit of 100 iterations most of the time.

We built a simple viewer in OpenGL to display the skeleton of the trees. It functions in real-time and displays all the details such as reference frame and grid. The meshing module outputs standard VRML 3D format from a record of points in a matter of seconds. Each file contains a few hundred thousand triangles.

5.2 Early results

The fitness functions were mostly added one by one, to correct aberrations in the results of the runs, until we found a good combination.

When we had only the position fitness, we did not observe any creation of branches. The best individual was usually only one trunk that would hit the target surface and obtain a perfect fitness value for that (see Figure 12).

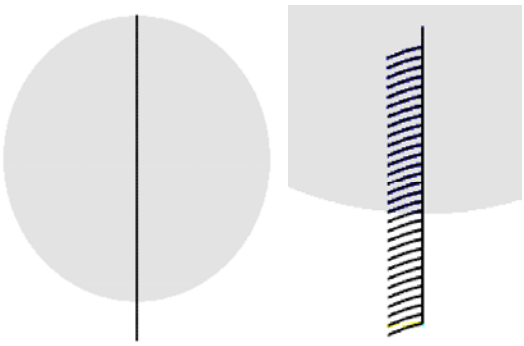

Figure 12. Gallery of monsters – the target is in gray
Left, only the position fitness: a single line
Right, with number of branches fitness: all in one plane

Then we added the fitness related to the number of branches, which helped correct this problem. The next issue was the fact that all the branches were created in the same plan. Since no extra rotation is needed, it is easier to do. (see Figure 12). Hence, we added the maximum-distance fitness, which was successful in spreading the branches. It forces the plant to start growing into the third dimension. (see Figure 13)

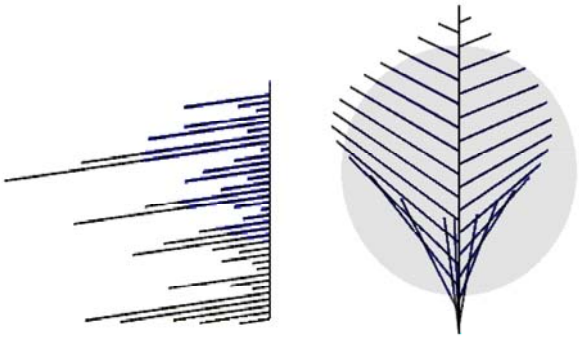

Figure 13. First results with distance fitness.
Left, G=20, M=200: still the same plane;
Right, G=30, M=500: first real 3D tree!

In order to keep the computation time low, we had to sample a subset of all the pairs of endpoints to do the computation, which resulted in some individuals being wrongly marked as very good. To solve this, we can evaluate more points of course, or at least set a minimum threshold on the number of sample points. We should also reevaluate the trees at every generation instead of caching their values.

It is interesting to notice that most plants pass through the previous 4 stages before evolving any further. With 1000 individuals, we often get one single line after the first generation, one line with starting points for branches after 2 or 3 generations, an evolved structured in the plane after 10 generations, a fully 3D structure after 25 generations. Only then does it start to resemble the final shape.

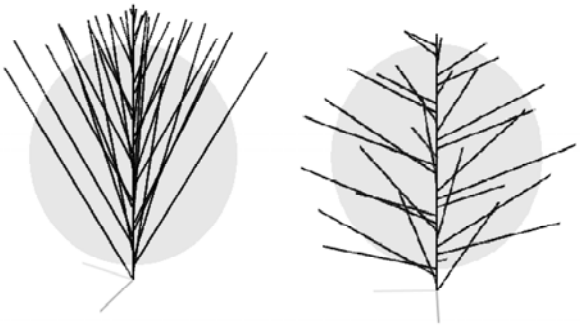

Figure 14. With G=40, M=1500, more complex results

5.2 More realistic trees

We started adding more individuals in the population and tweaking the parameters. Our goal was to produce tress with more than one level of recursion of branches.

We also played with the thickness (see Figure 17). It appeared that since we were trying to stay close to zero without ever touching zero, we could not control precisely the lower bound on the thickness. The solution is to remap the thickness values between 0.1 and 1 (instead of the usual 0.001 to 1 range that we get).

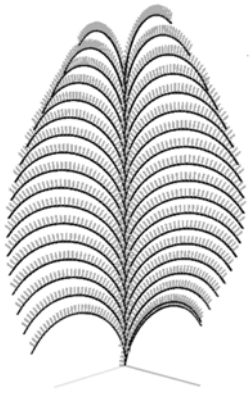

Figure 15. With G=50, M=3000 and flatter target shape. Light marks indicate the positions of the leaves

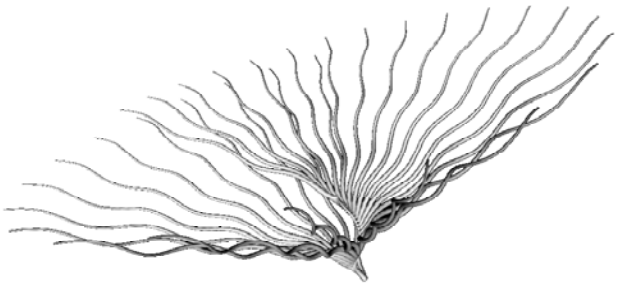

Figure 16. Surprising undulation in the branches. This model is meshed but bears no leaf.

Figure 17. Pine tree (with leaves!). The thickness is taken into account: trunk and branches shrink progressively. The bending is due to simulated gravity.

5.3 Using the grid to get more specific trees

The grid was meant to replace and extend the maximum distance as well as the position fitness. Our experimentations showed that it was not a good idea to simply replace these two. The grid is too complex for simple trees and we do not get past the first stage of evolution, where usually we should observe the separation of branches and the passage to 3D.

As a result, we only start using the grid after the individuals become good enough. This intervenes after the fitness is low enough to show that the individual already looks like a tree. The purpose of the position fitness is not to define the shape of plant anymore but to favor its expansion during its early stages. That approach yielded interesting trees in 50 generations with 3000 individuals.

The tree in Figure 19 has been made by creating a grid with constraints matching the shape of the house and taking 100 generations for 2000 individuals.

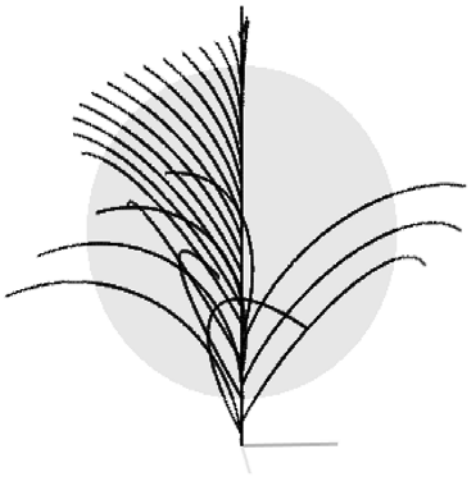

Figure 18. Skeleton view. The 5x5x5 grid favored the top left side and forbade the top right side

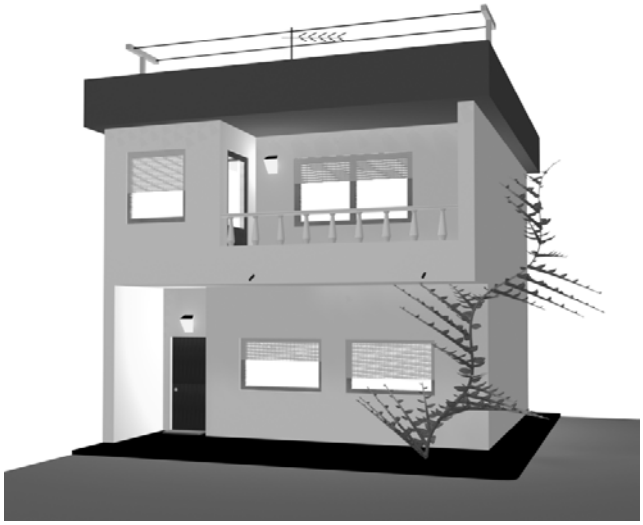

Figure 19. The plant fits nicely under the balcony.

5.4 A step-by-step evolution

As an illustration to all the concepts introduced above, figure 20 shows a step-by-step evolution. The goal was to get a fairly round shape (same target sphere as in section 4), but we wanted to limit the growth of the plant as if there was some object above it, figured by a box in the pictures. We designed a 5x5x5 grid to codify this constraint in the form of forbidden cells at the top of the grid.

Table 2. Step by step fitness evaluation

Best of Generation	2	8	25	100	**Weight**
Branches	36	20	30	30	-
F_{pos}	35	27	13	15	**5**
F_{num}	6	10	0	0	**50**
F_{dist}	5.4	2.9	1.9	0.7	**200**
F_{grid}	100	59.3	23.9	19.5	**10**
F_{total}	2555	1808	684	410	-

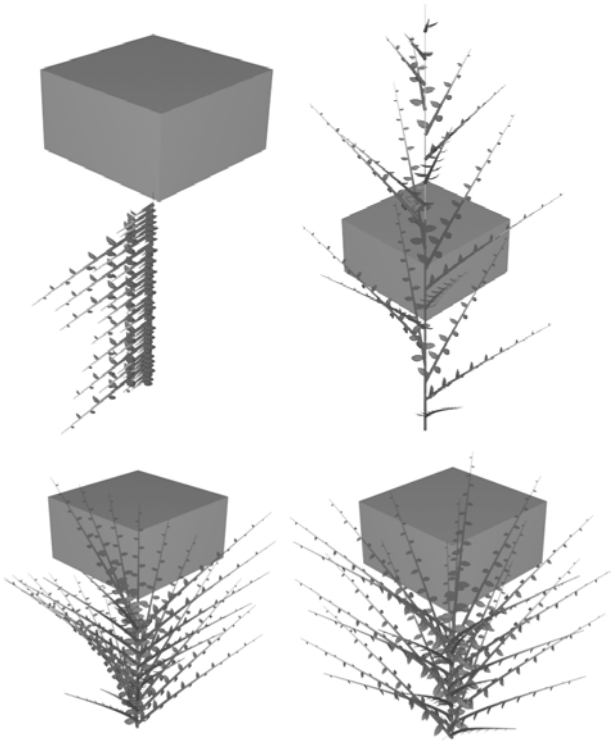

Figure 20. Four steps of an evolution with M=8000 Best individuals for generations 2 (top left), 8 (top right), 25 (bottom left) and 100 (bottom right). Only tree and view change; the box stays the same.

After generation 2, the tree is in a plane. Branches are starting to get irregular. Generation 8 is the first generation where the best individual is not just in the plane. At that point the grid constraint has just kicked in, which explains why the best individual goes through the box. Around generation 25, the constraint is being considered by more than half of the individuals. The best tree of generation 25 does not seem to hit the box. But it is still very irregular and does not look all that nice. That can be explained by the distance maximization component of the fitness, which is still fairly high. After 100 generations, we get a tree that solves the problem: it does not hit the box at all. The branches have a significantly larger spread and more cells of the grid are covered than in generation 25, which yield a lower fitness.

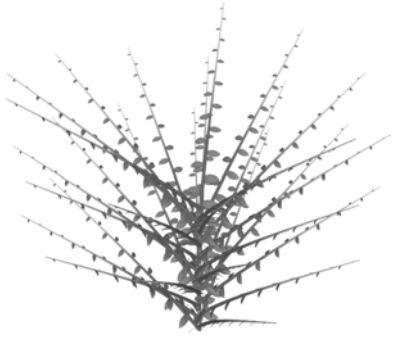

Figure 21. End result for this run, G=100.

6. Discussion of results

The early results we obtained were far from our best results. It took considerable tweaking of the fitness to get what we wanted. Especially since we use several numerical values, which are created once and for all before the first generation, we always ended up with a reduced set of values after a few generations. To correct that, we added 2% mutations to renew the pool of values without breaking too many good trees. We could have mutated only the values but it was also interesting to have some operators reappear after being evicted. For instance, as long as getting the right number of branches is predominant in the fitness, everything comes down to loops and `CreateAutomaton`. It is interesting to be able to reintroduce `multiply` operators after the plant is stable, in order to take care of the thickness. We finally have a fitness function that works well to grow realistic tree.

The feature we did not manage to fully implement was the multiple levels of branches. It was extremely hard to generate a tree with 30 principal branches but only 5 sub-branches per branch, let alone make the tree satisfy the other constraints at the same time. We think it is mostly a computation time issue. It takes longer evaluation with much more individuals to solve that kind of problems in our framework. This explains why we could not attain results that really look like common trees.

The extension to a grid of constraints works particularly well. Even though we could not simplify the fitness evaluation as we had hoped in the beginning, it gives powerful high-level control on the shape of the tree. To make best use of this technique, the design of the tree-specific part could be left to more standard approaches than evolution. The genetic programming algorithm could just act on an additional automaton.

7. Conclusion

We have described a way to apply genetic programming to a tree-modeling problem. Starting from high-level constraints of shape and density, we defined adequate fitness functions. We used an automata model to represent trees and we converted these automata into genetic programming tree of operators using cellular encoding. We evolved a very general fitness function from experimenting with the algorithms.

To extend our results we introduced a grid-based constraint description that lets us specify attractive and repulsive zones of the space in an easy way. We finally obtained the convoluted plants we expected. Since we do not look for the best tree but for a tree, which respects most of the specifications, genetic programming is very appropriate. It gives a variety of results that can be helpful for a designer.

8. Future Work

It seems that we only scratched the surface of an interesting approach to solve the problem of automatically designing 3D models. We could directly improve our results on trees by adding more precise fitness evaluation regarding the position and orientation of the leaves, or by allowing more freedom in the creation of the automata (more levels of branches, or different kind of organs).

Since we only used a small subset of the expressivity of the automata model, we could easily extend this work to a larger subset, accounting for external stimulations and forces instead of only considering static constraints. We could also let a certain kind of automata operate on the plant after it has been built. In particular, we could make flowers bloom or allow for aging. This would add significant complexity to the simulator and the automata but it is a path that I am currently pursuing with N. Lambert.

Our analysis could be extended to the production of leaves and flowers with no difficulty. It is even more generally applicable in general modeling problems where we want to design an object only through high level concepts. In particular the grid-based fitness is extremely flexible.

Acknowledgments

I would like to thank N. Lambert for providing the base code for the simulator and the 3D meshing system as well as for his support during the development of this project. The house 3D model has been designed by Alberto Garcia.

Bibliography

Gruau, F. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* Ecole Normale Supérieure de Lyon, France, 1994

Jacob, C. 1995. Genetic L-System Programming: Breeding and Evolving Artificial Flowers with Mathematica. *IMS'95, Proc. First International Mathematica Symposium*, Southampton, Great-Britain, UK, Computational Mechanics Publications: 215-222

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Lindenmayer, A. 1968. Mathematical models for cellular interaction in development. *Journal of theoretical biology* 18: 280–315.

Mock, K. J. 1998. Wildwood: The Evolution of L-Systems Plants for Virtual Environments. *Proc. International Conference on Evolutionary Computation ICEC'98.* Anchorage, Alaska, USA. IEEE Press.

Ochoa, G. 1998. On Genetic Algorithms and Lindenmayer Systems. *Proc. PPSN IV, Lecture Notes in Computer Science 1498.* Sringer-Verlag, Amsterdam: 335-343

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

Reffye P., Edelin C., Francon J., Jaeger M. and Puech C. 1988. Plant Models Faithful to Botanical Structure and Development. *Journal of Computer Graphics* 22(4): 151-158.