Chapter 5
Grammars and L-systems with applications to vegetation and levels (DRAFT)

Julian Togelius, Noor Shaker and Joris Dormans

5.1 Plants are everywhere

Just like most games that feature physical movement include some form of terrains, very many games feature some vegetation in some form. Grass, trees, bushes and a myriad other forms. Vegetation seems like the perfect case for PCG: we need to create a huge number of artefacts (there are many trees in the forest and many straws of grass in the lawn) that are similar to each other, recognisable, but also slightly different from each other. Just copy-pasting trees won't cut it[1] as players will spot it quickly. Further, in most roles vegetation is of little functional significance, meaning that a botched plant will not make the game unplayable, just look a bit weird.

And in fact, vegetation is one of the success stories of PCG. Very many games use procedural vegetation generation, and there are many software frameworks available. For example, the *SpeedTree* middleware has been used in dozens of AAA games.

It turns out that one of the simplest and best ways to generate a tree or bush is to use a particular form of *formal grammar* called an *L-system*, and interpret its results as drawing instructions. This fact is intimately connected to the "self-similar" nature of plants, i.e. that the same structures can be found on both micro- and macro-levels. For an example of this, take a look at a branch of a fern, and see how the shape of the branch repeats in each sub-branch, and then in each branch of the sub-branch. Or look at a romanesco broccoli, which consists of cones on top of cones of top of cones... (see figure 5.1). As we will see, L-systems are naturally suited to reproducing such self-similarity.

In this chapter, we will introduce formal grammars in general, L-systems in particular and how to use a graphical interpretation of L-systems to generate plants. We will also give examples of how L-systems can be used as a representation in search-based PCG, allowing you to evolve plants. However, it turns out that plants

[1] In William Gibson's *Neuromancer*, portal novel of the cyberpunk movement, one of the main characters is busy copy-pasting trees in one of the early chapters.

Fig. 5.1: Romanesco broccoli. Note the self-similarity.

are not the only thing for which formal grammars are useful. In the rest of the chapter, we will explain how grammar-based systems can be used to generate quests and dungeon-like environments for adventure games such as Zelda, and levels for platform games such as Super Mario Bros.

5.2 Grammars

A (formal) *grammar* is a set of *production rules* for rewriting strings, i.e. turning one string into another. Each rule is of the form (symbol(s)) $\rightarrow$ (other symbol(s)). Here are some example production rules:

1. $A \rightarrow AB$
2. $B \rightarrow b$

Using a grammar is as simple as going through a string, and each time a symbol or sequence of symbols that occurs in the left hand side (LHS) of a rule is found, those symbols are replaced by the right hand side (RHS) of that rule. For example,

if the initial string is "A", in the first rewriting step the A would be replaced by B by rule 1, and the resulting string will be "AB". In the second rewriting step, the A would again be transformed to AB and the B would transformed to Bb using rule 2, resulting in the string "ABb". The third step yields the string "ABbb" and so on. A convention in grammars is that upper-case characters are nonterminal symbols, which are on the LHS of rules and therefore rewritten further, whereas lower-case characters are terminal symbols which are not rewritten further.

Formal grammars were originally introduced in the 1950's by the linguist Noam Chomsky as a way of modelling natural languages [3]. However, they have since found widespread application in computer science, as basically any computer science problem can be cast in terms of generating and understanding strings in a given language. Many results in theoretical computer science and complexity theory are therefore expressed using grammar formalisms. There is a rich taxonomy of grammars which we can only hint at here. Two key distinctions that are relevant for the application of grammars in procedural content generation are whether the grammars are deterministic, and the order in which they are expanded.

Deterministic grammars have exactly one rule that applies to each symbol or sequence of symbols, so that for a given string, it is completely unambiguous which rules to use to rewrite it. In nondeterministic grammars, several rules could apply to a given string, yielding different possible results of given rewriting step. So, how would you decide which rule to use? One way is to simply choose randomly. In such cases, the grammar might even include probabilities for choosing each rule. Another way is to use some parameters for deciding which way to expand the grammar — we will see an example of this in the section on grammatical evolution towards the end of the chapter.

5.3 L-systems

The other distinction of interest here is in which order the rewriting is done. *Sequential* rewriting goes through the string from left to right and rewrites the string as it is reading it; if a production rule is applied to a symbol, the result of that rule is written into the very same string before the next symbol is considered. In *parallel* rewriting, on the other hand, all the rewriting is done at the same time. Practically, this is implemented as new string being implemented at a separate memory location containing only the effects of applying the rules, and the original string is left unchanged. Sometimes, the difference between parallel and sequential rewriting can be major.

L-systems are a class of grammars whose defining feature is parallel rewriting, and which was introduced by the biologist Aristid Lindenmayer in 1968 explicitly to model the growth of organic systems such as plants and algae [9]. The following is a simple L-system defined by Lindenmayer to model yeast growth:

1. $A \rightarrow AB$
2. $B \rightarrow A$

Starting with the axiom A (in L-systems the seed strings are called axioms) the first few expansions look as follows:

1. A
2. AB
3. ABA
4. ABAAB
5. ABAABABA
6. ABAABABAABAAB
7. ABAABABAABAABABAABABA
8. ABAABABAABAABABAABABAABAABABAABAAB

There are several interesting things about this sequence. For one thing, the obvious regularity, which is more complex than simply repeating the same string over and over, and certainly seems more complex than is warranted by the apparent simplicity of the system that generates it. But also note that the rate of growth of the strings in each iteration is increasing. In fact, the length of the strings is a Fibonacci sequence: 1 2 3 5 8 13 21 34 55 89... This can be explained by the fact that the string of step n is a concatenation of the string of step $n-1$ and the string of step $n-2$.

Clearly, even simple L-systems have the capacity to give rise to highly complex yet regular results. This seems like an ideal fit for PCG. But how can we move beyond simple strings?

5.3.1 Graphic interpretation of L-systems

One way of using the power of L-systems to generate 2D (and 3D) artefacts is to interpret the generated strings as instructions for a turtle in *turtle graphics*. Think of the turtle as moving across a plane holding a pencil, and simply drawing a line that traces its path. We can give commands to the turtle to move forwards, and to turn left or right. For example we could use the following key to interpret the generated strings:

- F: move forward a certain length (e.g. 10 pixels)
- +: turn left 90 degrees
- -: turn right 90 degrees

Such an interpretation can be used in conjunction with a simple L-system to give some rather remarkable results. Consider the following system, consisting only of one rule:

1. $F \rightarrow F+F-F-F+F$

Starting this system with the axiom F, it would expand into $F+F-F-F+F$ and then into $F+F-F-F+F+F+F-F-F+F-F+F-F-F+F-F+F-F-F+F+F+F-F-F+F$ etc. Interpreting these strings as turtle graphics

instructions, we get the sequence of rapidly complexifying pyramid-like structures shown in figure 5.2, known as the Koch curve.

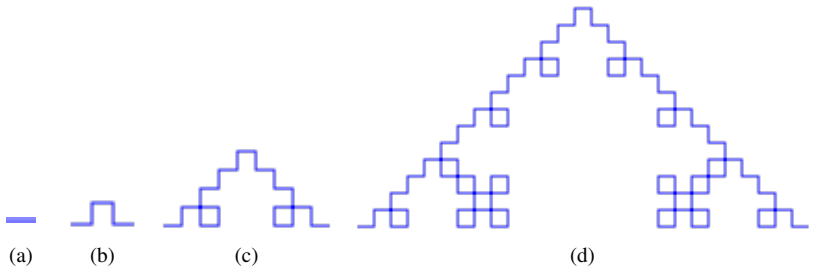


Fig. 5.2: Koch curve generated by the L-system $F \to F+F-F-F+F$ after 0, 1, 2 and 3 expansions.

5.3.2 Bracketed L-systems

While interpreting L-system-generated strings as turtle instructions allows us to draw complex fractal shapes, we are fundamentally limited by the constraint that the figures must be drawable in one contiguous line – the whole shape must be drawn "without lifting the pencil". However, many interesting shapes cannot be drawn this way. For example, plants are branching and requires you to finish drawing a branch before returning to the stem to draw the next line. For this purpose, *bracketed* L-systems were invented. These L-systems have two extra symbols, [and], which behave like any other symbols when rewriting the strings but act as "push" and "pop" commands to a stack when interpreting the string graphically. (The stack is simply a first-in, last-out list.) Specifically, [saves the current position and orientation of the turtle onto the stack, and and] retrieves the last saved position from the stack and resets the turtle to that position – in effect, the turtle "jumps back" to a position it has previously been at.

Bracketed L-systems can be used to generate surprisingly plant-like structures. Consider the L-system defined by the single rule $F \to F[-F]F[+F][F]$. This is interpreted as above, except that the turning angles are only 30 degrees rather than 90 degrees as in the previous example. Figure 5.3 shows the graphical interpretation of the L-system after 1, 2, 3 and 4 rewrites starting from the single symbol F. Minor variations of the rule in this system generate different but still plant-like structures, and the general principle can easily be extended to three dimensions by introducing symbols that represent rotation along the axis of drawing. For a multitude of beau-

tiful examples of plants generated by L-systems see the book "The Algorithmic Beauty of Plants" by Prusinkiewicz and Lindenmayer [15].

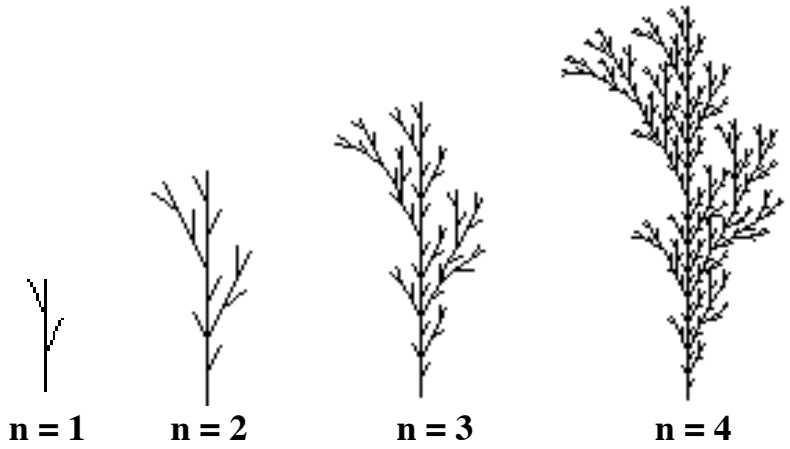


Fig. 5.3: Four rewrites of the bracketed L-system $F \rightarrow F[-F]F[+F][F]$.

5.4 Evolving L-systems

Like any parametrisable PCG method, L-system expansions can be used as genotype-to-phenotype mapping in search-based PCG. An early paper by Ochoa presents a method for evolving L-systems to attain particular 2D shapes [10]. She restricts herself to L-systems with the simple alphabet used above ($F + -[]$), the axiom F and a single rule with the LHS F. The genotype is the RHS of the single rule. Ochoa used a canonical genetic algorithm with crossover and mutation together with a combination of several evaluation functions. The fitness functions all relate to the shape of the phenotype, namely the height ("phototropism"), bilateral symmetry, exposed surface area ("light gathering ability"), structural stability and proportion of branching points. By varying the contributions of each fitness function, she showed that it is possible to control the type of the plants generated with some precision. Fig. 5.4 shows some examples of plants evolved with a combination of fitness functions, and Fig. 5.5 shows some examples of organism-like structures evolved with the same representation but a fitness function favouring bilateral symmetry.

Fig. 5.4: Some evolved L-system plants.

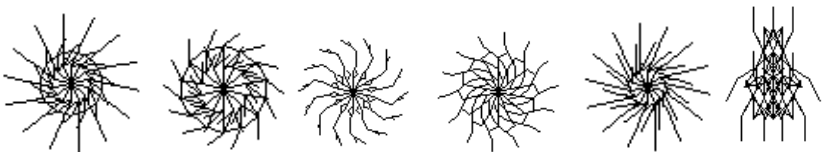


Fig. 5.5: Some L-system structures evolved for bilateral symmetry.

5.5 Generating missions and spaces with grammars

A game level is not a singular construction, but rather a combination of two interacting structures: a mission and a space [4]. A mission describes the things a player can or need to do to complete a level, while the space describes the geometric layout of the environment. Both mission and space have their own structural qualities. For missions it is important to keep track of flow, pacing and causality, while for the space connectedness, distance and sign posting are critical dimensions. To successfully generate levels that feel consistent and coherent it is important to use techniques that can generate each structure in such way that strengthens their individual qualities while making sure that the two structures are interrelated and work together. This section discusses how different types of generative or transformative grammars can be used to achieve this.

5.5.1 Graph grammars

Generative grammars typically operate on strings, but they are not restricted to that type of representation. Grammars can be used to generate many different types of structures: graphs, tile maps, two or three dimensional shapes, and so on. In this

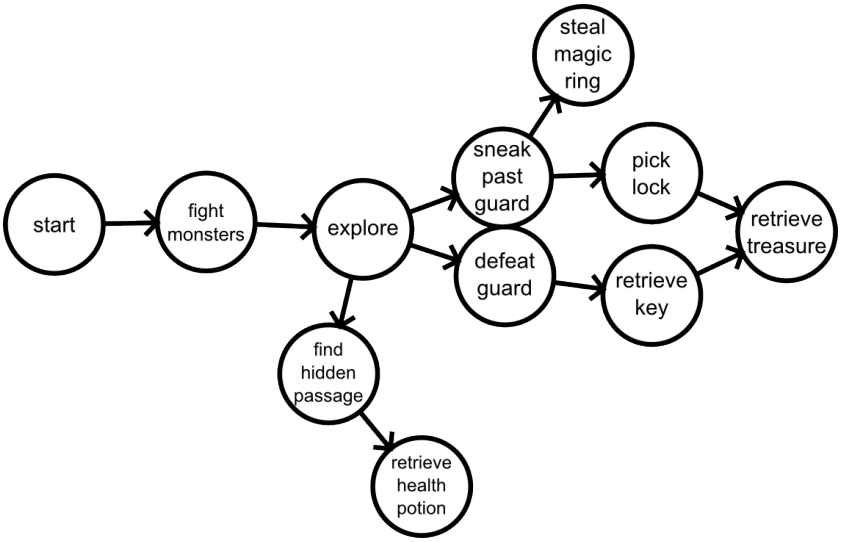


Fig. 5.6: A mission structure with two paths.

section and the following section, we will explore how grammars can be used to generate graphs and tile maps. These structures are useful ways to represent game missions and game spaces that combine into game levels.

Graph are more useful than strings to represent missions and spaces for games, especially when these missions and spaces need to have a certain level of sophistication. For example, a completely linear mission (which might be represented by a string) might be suitable for simple and linear games, but for explorative adventure games such as RPG dungeons you will want missions to contain lock and key puzzles, bonus objectives, and possibly multiple paths to lead to the level goal. Graphs can express this type of structures more easily. For example, Fig. 5.6 contains a mission that can be solved in two different ways.

Graph grammars work quite similar to string grammars; graph grammar rules also have a left hand part that identifies a particular graph construction that can be replaced by one of the constructions in the right hand part of the rule. However, to make the transformation, it is important to identify each node in the left hand individually and to match them with individual nodes in each right hand part. Fig. 5.7 represents a graph grammar rule and uses numbers to identify each individual node. When using this rule to transform a graph, follow 5 steps (as illustrated by Fig. 5.8)[2]:

[2] in simple graph transformations there is no need to identify and transform individual edges in the same way as nodes are identified and transformed. However, a more sophisticated implementation that requires edges to be transformed rather than removed and added for each transformation can be realised by identifying and replacing edges in the same way as nodes.

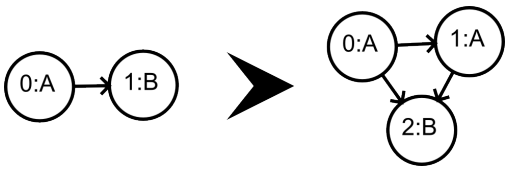


Fig. 5.7: A graph grammar rule.

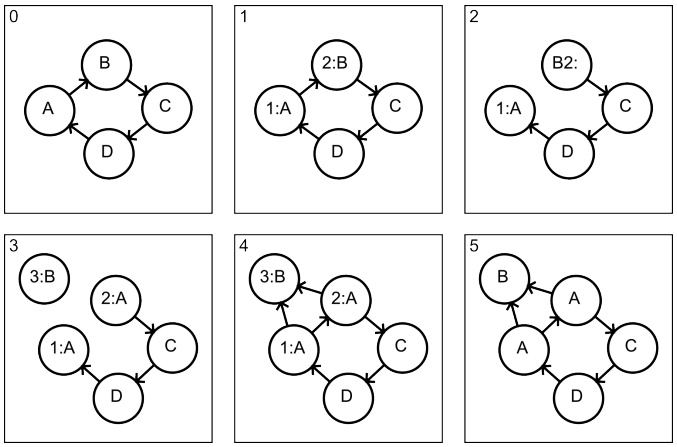


Fig. 5.8: Graph grammar transformation.

1. Find a subgraph in the target graph that matches the left hand of the rule and mark that subgraph by coping the identifiers of the nodes.
2. Remove all edges between the marked nodes.
3. Transform the graph by transforming marked nodes into their corresponding nodes on the right hand side, adding a node for each node in the right hand that has no match in the target graph, and removing any nodes that have no corresponding node in the right hand side[3].
4. Copy the edges as specified by the right hand side.
5. Remove all marks [16].

[3] Take into account that the removal of nodes only works when the node to be removed is only connected to nodes that have been marked. This is something to take into account when designing graph grammar rules.

5.5.2 Using graph grammars to generate missions

To generate a simple mission using graph grammars, it is best to start defining the alphabet the grammar is designed to work with. In this case the alphabet consists of the following nodes and edges:

- Start (node marked S): the start symbol from which the grammar generates a mission (the axiom).
- Entrance (nodes marked e): the starting place of the player.
- Tasks (nodes marked t): arbitrary, unspecified tasks (here be monsters!).
- Goals (nodes marked g): an task that finishes the level when successfully completed.
- Locks (nodes marked l): a tasks that requires a key to perform successfully.
- Keys (nodes marked k).
- Nonterminal task nodes (nodes marked T)
- Normal edges (represented as solid arrows) connecting nodes and identifying which task follows which.
- Unlock edges (represented as solid arrows marked with a dash) connecting keys to locks.

With this alphabet we can construct rules that generate missions. For example, the rules in Fig. 5.12 were used to generate the sample missions in Fig. 5.10[4].

One thing you might notice from studying these rules is that graph grammars can be hard to control. In the case of the rule set represented in Fig 5.12, the number of task generated (by the application of the "add task" rule) can be as low as one and has no upper limit. As soon as the Start node is removed from the graph, the number of tasks no longer grows. One way to get a better grip on the generated structures is not to apply rules indiscriminately, but to specify a sequence of rules so that each rule in the sequence is applied once to one possible location in the graph. For example, if we split up the "add task" rule from Fig. 5.12 into two rules (see Fig. 5.11), the missions in Fig. 5.12 are generated by apply the following sequence of rules[5]:

- start rule (x1),
- add task (x6),
- add boss (x1),
- define task (x6),
- move lock (x5).

[4] The rules use a special wildcard node (marked with a*) to indicate a match with any node. Wildcards in the right hand side of a rule never change the corresponding node in the graph being transformed. An alternative to these wildcards is to allow rules to have edges without origin or target node.

[5] Obviously, the sequence of rules might be generated by a string grammar.

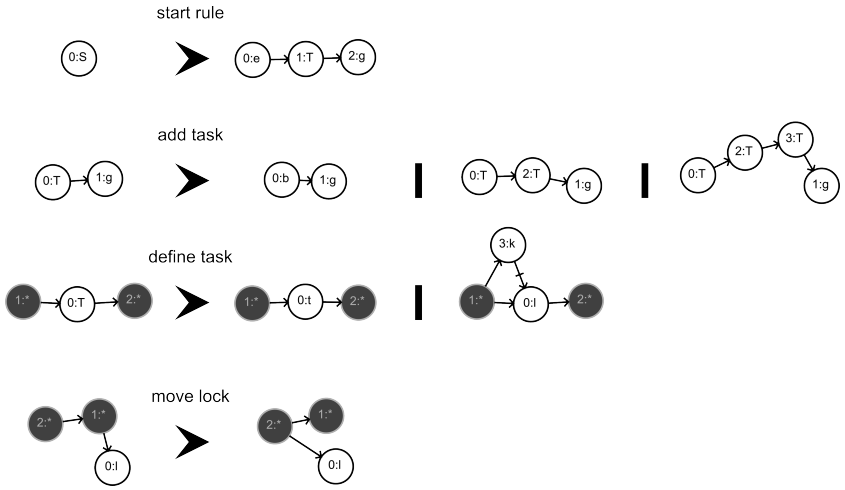


Fig. 5.9: Mission rules.

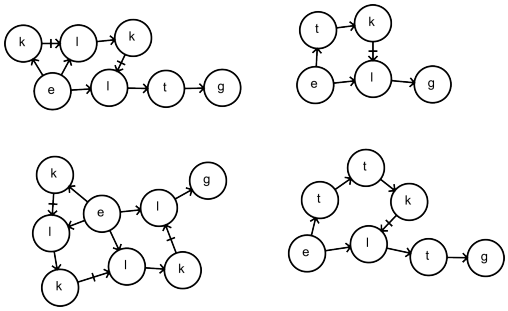


Fig. 5.10: Generated Missions.

Fig. 5.11: Two new rules to replace the old “add task”.

5.5.3 Breaking the process down into multiple generation steps

So far, the graph grammars are relatively simple. However, to generate anything resembling the complexity of the mission in Fig. 5.6, many more rules are required. Designing the grammars to achieve such results takes practice and patience. A key strategy to design successful grammars is to break down the process into multiple

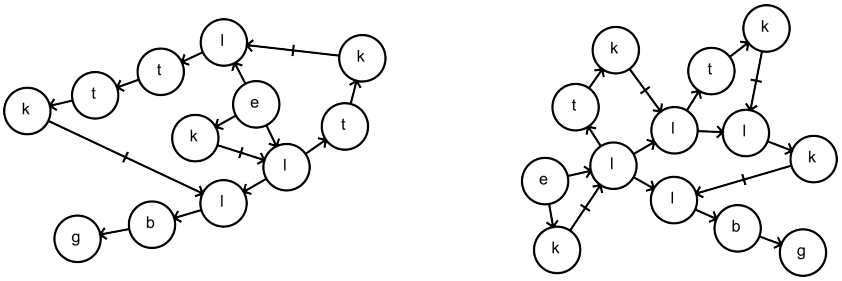


Fig. 5.12: Missions generated from the same sequence of rules.

steps. Trying to generate everything at once using only one grammar is a daunting task, and next to impossible to debug and maintain[6].

When breaking down the generation process into multiple steps, it useful to think of each step as a simulation of the design process. One step might generate the overall specifications of the mission, while the next might flesh out those specifications. In game design, a successful design strategy is to start from a set of random set of requirements and use your creativity to shape that random collection into a coherent whole. Following a similar approach for breaking down the generation procedure and designing individual grammars yields good results. In particular, designing one simple step to create a highly randomised graph and use a second step the restructure that graph into something that makes sense from the game's perspective, is a very effective strategy to create very expressive generation procedures [6].

For example, we can use a single step to generate a mission of a specified length and randomly choose between locks, keys and other tasks to fill in the spaces between the entrance and the goal. Although in this case we also make sure that the first task is always a key and the last task is always a lock. Fig. 5.13 and Fig. 5.14 represent the rules and a sample mission built using those rules. Note that although locks and keys are placed, no relationship between them is established.

The next step is to extract lock and key relationships. Based on the spread of the locks and keys over the tasks, multiple keys can be assigned to a single lock, and vice versa. This would represent multiple levers that need to be activated to open a single door, or a special weapon that can be used multiple times to get past a special type of barrier. Fig. 5.15 represents the rules to add these relationships, and Fig. 5.16 is a sample configuration created from the sample set in Fig. 5.14.

Next steps could include the movement of locks through the graph (as we have seen in the example above), generating more details of the nature of the locks and keys, or adding tasks of a different type. One of the advantages of using these two steps is that two relatively simple grammars can create a large variety of different

[6] Breaking down the generation into multiple steps is in line with the approach to software engineering and code generation suggested by model driven engineering. When done right, this approach leads to a flexible generation processes that allow you to generate spaces from missions or vice versa, and creates opportunities to design generic, reusable generation steps [1, 5].

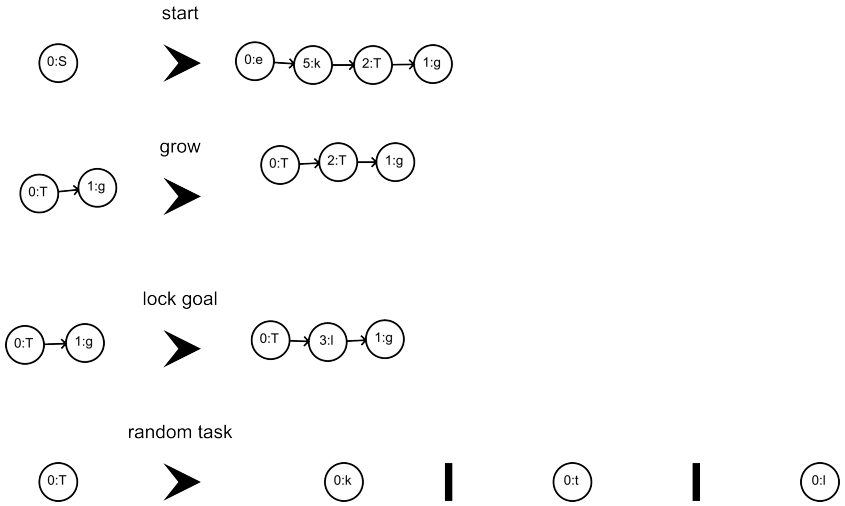


Fig. 5.13: Rules to create random set.

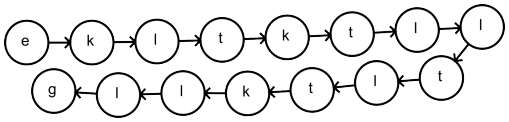


Fig. 5.14: Sample random set.

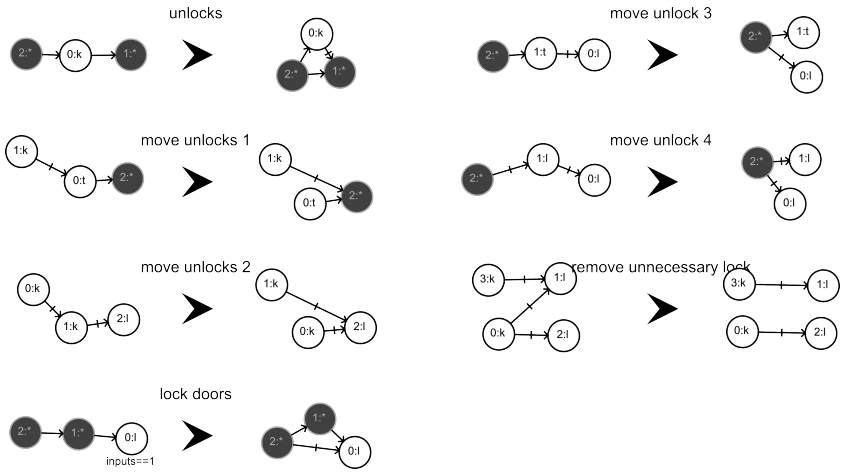


Fig. 5.15: Rules to add lock and key relationships.

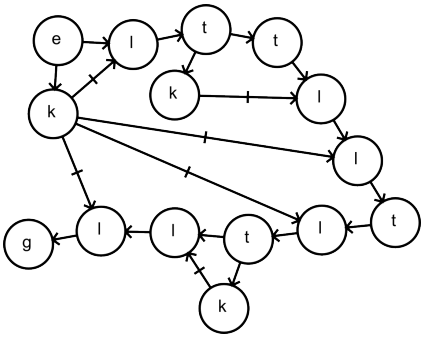


Fig. 5.16: Generated lock and key relationships.

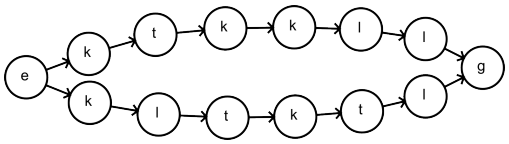


Fig. 5.17: Two paths to a single goal.

relationships (two keys to a single lock, or keys that are reused). Getting the same level of variation using explicit rules that create an X number of keys to a single lock would require many more rules which are much harder to maintain. In addition, the second step can also be executed on graphs that have been built to different specifications. For example, the same rules can be used to create lock and key relationships for a dungeon that has two separate paths (see Fig. 5.17).

5.5.4 Generating spaces to accommodate a mission

Having a representation of a mission itself is only one step towards the generation of levels for a game. Missions need to be transformed into spaces that the player can actually traverse. Transforming from mission to space is one of the hardest steps in this process. The problem comes down to generating two different, independent but linked structures: an abstract mission that details the things a player needs to do, and a concrete space that creates the world where the player can do these things. Below you find three strategies to deal with the problem of generating the two structures:

1. Transform from mission to space. The transition from Fig. 5.14 to Fig. 5.16 reflects the gradual transition from abstract missions to a more concrete representation of a game space. Although in this case, the game space is still highly abstract. However by using automatic graph layout algorithms and sampling the

results into a tile map, you are able to generate usable level geometry. This approach works well for games such as action adventure games or games with a strong narratives, where mission coherence and pacing is important. The disadvantage of this approach is that the difficulty of going from mission to space is most pronounced.
2. Transform a mission to a set of instructions to build a space. Instead of transforming a mission structure into a space directly you can transform the mission into a set of building instructions that can be used to build a space to match the requirements. This approach has the advantage that the transition from graphs to tiles or shapes is much easier, it also comes at a cost: it is very difficult to generate spaces that have multiple paths leading to the same goal or location. So this approach works best for very linear games like platformers or certain story driven games.
3. Build level geometry and distill a more abstract representation of the game space to generate the missions from. This approach reverts the problem by generating level geometry first and set up missions for that geometry. You can do this by generating a geometry using cellular automata, grammars, evolution, or any other technique, then analysing the geometry to create an abstract graph representation of that same space which you can transform into suitable mission structures. This approach works well for strategic games, levels that take place in locations that require some consistent architecture (such as castles, dwarf fortresses, police stations, or space ships), and for levels that the player is going to visit multiple times. The downside of this approach is that it is critical that the geometry is generated with enough mission potential (are there doors to be locked, bottleneck to set up traps, and so on) and that you have far less control over the mission than with the other two approaches.

When choosing between these strategies, or when trying to come up with another strategy, it is important to think like a designer. The most effective way of generating levels using a multistep process and different representations of missions and spaces is to model the real design process. Ask yourself, how would you go about designing a level by hand? Would you start by listing mission goals, or by sketching out a map? What sort of changes do you make and can those changes be captured by transformational grammars?

5.5.5 Extended Example: 'Dules

An extended example following the third strategy concludes this section. The example details the part of the PCG for the game 'Dules, which is currently in development. In this game, players control futuristic combat vehicles (tanks, hovercraft, and so on) in a post-apocalyptic, alien-infested world. The players can choose missions from a world map, after which the game generates an environment to match the location on the map and sets up a mission based on the affordances of the environment

and specifications dictated by the current game state (who controls the environment, is the player trying to take over, or defending from alien incursion, and so on).

The content generation of 'Dules makes use of transformation grammars that operate on strings, graphs, and tiles. Tile grammars are very simple. They also consist of rules with one left and one or more right hands where the left hand can be replaced by one of the right hand constructions. Like graph grammars, the tile grammars used in 'Dules can work with wildcards to indicate that certain tiles can be ignored. In contrast to string and graph grammars, tile grammars cannot change the number of tiles. In addition, tile grammars can be made to stack tiles onto each other instead of replacing them.

The procedural content generation procedure roughly follows the steps as outlined Fig. 5.18. In this case taking the tile-based world map as an input (1), the particular location is selected (2). Based on the presence of particular tiles indicating vegetation, elevation, buildings, and such, a combination of tile grammars and cellular automata are used to create the terrain (3-7). The terrain is analysed and transformed into an abstract representation (8). At the same time, mission specifications are generated using a string grammar (9), which are used as building instructions to plot a mission onto the space graph (10)[7]. Finally, some extra enemies are added to the mission (11), all the mission specific game objects are placed onto the same tile map (12) and combined with the terrain to create the complete mission (13).

Almost all steps in the process are handled by grammars. Tile grammars are used to generate the terrain, tile grammars are even used to specify different cellular automata. String grammars are used to create the mission specification and graph grammars are used to create the mission itself. The translation of the terrain into the space graph is done using a specialised algorithm that distinguishes between walkable terrain, impassible terrain, and bodies of water. Each node in (8) represents around 100 tiles, and a reference between the node and the tiles is kept to be able to place the game objects in the right area during (12).

5.6 Grammatical evolution for Infinite Mario Bros level generation

Grammatical Evolution (GE) is an evolutionary algorithm based on Grammatical Programming (GP) [13]. The main difference between GE and GP is the genome representation; while a tree-based structure is used in GP, GE relies on a linear genome representation. Similar to general Genetic Algorithms (GAs), GE applies fitness calculations for every individual and it applies genetic operators to produce the next generation.

[7] In this case certain graph nodes are represented to contain other nodes. This is just a representation, for the implementation and the grammars, such a containment is nothing but a special type of edge, that is rendered differently.

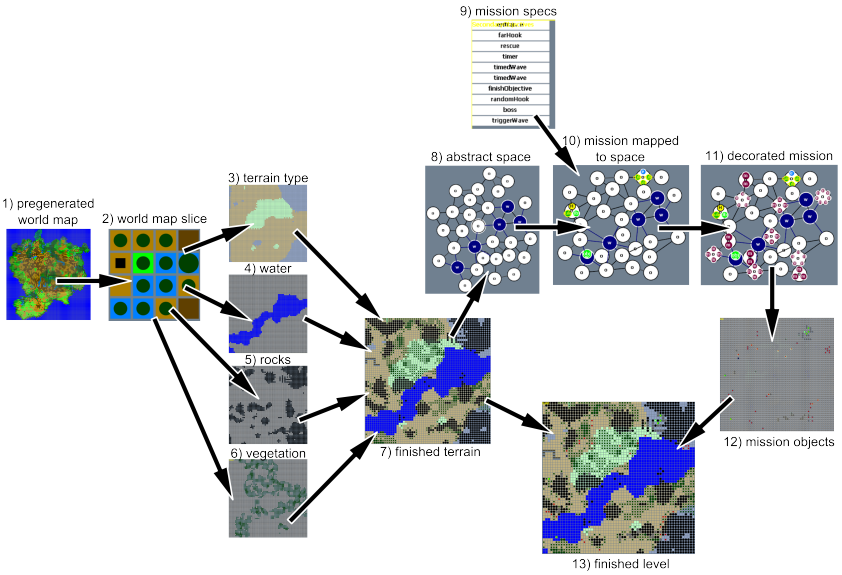


Fig. 5.18: The generation steps to create a level for ‘Dules.

The population of the evolutionary algorithm is initialised randomly consisting of variable-length integer vectors; the syntax of possible solution is specified through a context-free grammar. GE uses the grammar to guide the construction of the phenotype output. The context-free grammar employed by GE is usually written in Backus Naur Form (BNF). Because of the use of a grammar, GE is capable of generating anything that can be described as a set of rules such as mathematical formulas [18], programming code, game levels [17] and physical and architectural designs [2, 14]. GE has been used intensively for automatic design [8, 2, 14, 7, 11], a domain where it has been shown to have a number of strengths over more traditional optimisation methods.

5.6.1 Backus Naur Form

Backus Naur Form (BNF) is a set of production rules usually used to express a grammar. A BNF grammar $G = \{N, T, P, S\}$ consists of terminals, T, non-terminals, N, production rules, P and a start symbol, S. Such as in any grammar, non-terminals can be expanded into one or more terminals and non-terminals through applying the production rules. An example BNF to generate valid mathematical expressions is given in Fig. 5.19.

```
(1) <exp> ::= <exp> <op> <exp>
            | ( <exp> <op> <exp> )
            |  <var>
(2) <op> :: = + | - | * | /
(3) <var> ::= X
```

Fig. 5.19: Illustrative grammar for generating mathematical expressions.

Each chromosome in GE is a vector of codons. Each codon is an integer number used to select a production rule from the BNF grammar in the genotype-to-phenotype mapping. A complete program is generated by selecting production rules from the grammar until all non-terminal rules are mapped. The resulted string is evaluated according to a fitness function to give a score to the genome. To better understand the genotype-to-phenotype mapping, we will give a brief example.

Consider the grammar in Fig. 5.19 and the individual genotype integer string $(4,5,8,11)$. We begin the processing of the mapping from the start symbol $< exp >$. In this case there are three possible productions, to decide which production to choose, we use the first value in the input genome and apply the mapping function $4\%3 = 1$, where 3 is the number of possible productions, the result from this operation indicates that the second production should be chosen, and $< exp >$ is replaced with $(< exp >< op >< exp >)$. The mapping continues by using the next integer with the first unmapped symbol in the mapping string, the mapping string then becomes $(< var >< op >< exp >)$ through the formula $5\%3 = 2$. At this step $< var >$ has only one possible outcome and there is no choice to be made, hence, X is inserted without reading any number from the genome. The expression becomes $(X < op >< exp >)$. Continuing to read the codon values from the example, individual's genome $< op >$ is mapped to $+$ and $< exp >$ is mapped to X through the two formulas, $8\%4 = 0$ and $11\%3 = 2$, respectively. This results in the expansion $(X + X)$.

During the mapping process, it is possible for individuals to run out of genes, in this case GE either declares the individual as invalid by assigning it with a penalty fitness value or it wraps around and reuses the genes.

5.6.2 Grammatical evolution level generator

In the work done by Shaker et al. [17] grammatical evolution was adopted to generate content for Infinite Mario Bros (IMB) motivated by the number of advantages it provides over more traditional optimization methods [12]: it maintains a simple way of describing the structure of the levels; it enables an open-ended structure where the design and model size are not known a priori; it enables the design of aesthetically pleasing levels by exploring a wide space of possibilities since the exploratory process is not constrained or biased by imagination or known solutions; it allows an easy incorporation of domain knowledge through its underlying grammatical repre-

sentation permitting level designers to maintain greater control of the output and it makes possible to easily generalize to different types of games.

The following section summarises the work done by Shaker et al. [17]. We start by presenting the design grammar used by GE to specify the structure of IMB levels, after that we present how GE was employed to evolve playable levels for the game.

5.6.2.1 Design grammar for content representation

As mentioned earlier, GE uses a Design Grammar (DG), written in BNF, to specify the representation of solutions (in our case a level design). Several methods can be followed to specify the structure of the levels in a design grammar, but since the grammar employed by GE is of context-free nature, this limits the possible solutions available. To accommodate for this constraint, and to keep the grammar as simple as possible; the implementation proposed is to add a game elements to the 2D level array regardless of the positioning of the other elements. With this solution, however, arises a number of conflicts in level design that should be resolved. Section 5.6.2.2 discusses this issue and the proposed solution in details.

The internal representation of the levels in IMB is a two-dimensional array of objects, such as brick blocks, coins and enemies. The levels are generated by placing a number of chunks in the two-dimensional level map. The list of chunks that was considered includes platforms, gaps, stairs, piranha plants, bill blasters, boxes (blocks and brick blocks), coins, goombas and koopas. Each of these chunks has a distinguishable geometry and properties. Fig. 5.20 presents the different chunks that collectively constitute a level. The level initially contains a flat platform that spans the whole x-axis, this explains the need of defining gaps as one of the chunks.

A design grammar was specified that takes into account the different chunks. In order to allow more variations in the design, platforms and hills of different types were considered such as a blank platform/hill, a platform/hill with a bill blaster, and a platform/hill with a piranha plant.

Variations in enemy placements were achieved by (1) constructing the physical structure of the level, (2) calculating the possible positions on which an enemy can be placed (this includes all positions where a platform was generated) and (3) placing each generated enemy in one of the possible positions.

The design grammar constructed can be seen in Figure 5.21. A level is constructed by placing a number of chunks each assigned with two or more properties, the x and y parameters specify the coordinates of the chunk starting point position in the 2D level array and are limited to the ranges [5,95] and [3,5], respectively. These ranges are constrained by the dimension of the level map. The first and last five blocks in the x dimension are reserved for the starting platform and the ending gate, while the y values have been constrained in a way that insures playability (the existence of a path from the start to the end position) by placing all items in areas reachable by *Mario* by performing jumps. The w_g parameter specifies the width of gaps that insures the ability to reach the other edge, w stands for the width of a platform or a hill, w_c defines the number of coins, and h indicates the height of a tube

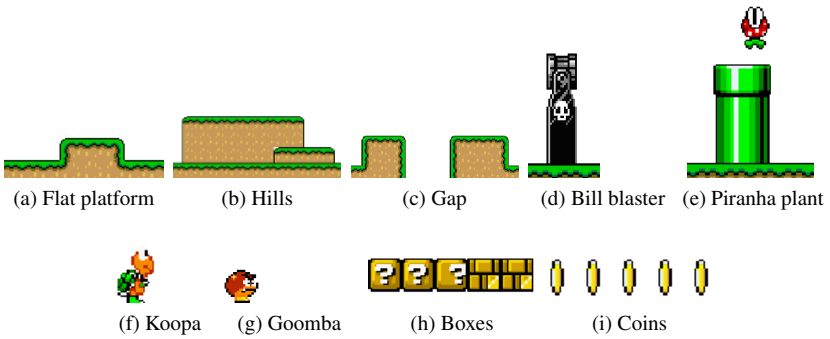


Fig. 5.20: The geometric representation of the different chunks used for constructing Infinite Mario Bros levels.

of the piranha plant or the height of a bill blaster. This height is also constrained to the range [3,4] assuring the possibility of jumping over tubes and bill blasters.

5.6.2.2 Conflict resolution and content quality

There are a number of conflicts inherent within the design grammar. According to the design approach, each chunk generated can be assigned any x and y values from the ranges [5,95] and [3,5], respectively, depending on the genotype without any restrictions. This means that it is very likely that there will be an overlap between the coordinates of the generated chunks. For example: $hill(65,4,5)$ $hill(25,4,4)$ $blaster\_hill(67,4,4,4,3)$ $coin(22,4,6)$ $platform(61,4,4)$ is a phenotype that has been generated by the grammar and contains a number of conflicts: e.g., $hill(65,4,5)$ and $blaster\_hill(67,4,4,4,3)$ were assigned the same y value, and overlapping x values; another conflict occurs between $hill(25,4,4)$ and $coin(22,4,6)$; as the two chunks also overlap on the $x-axes$.

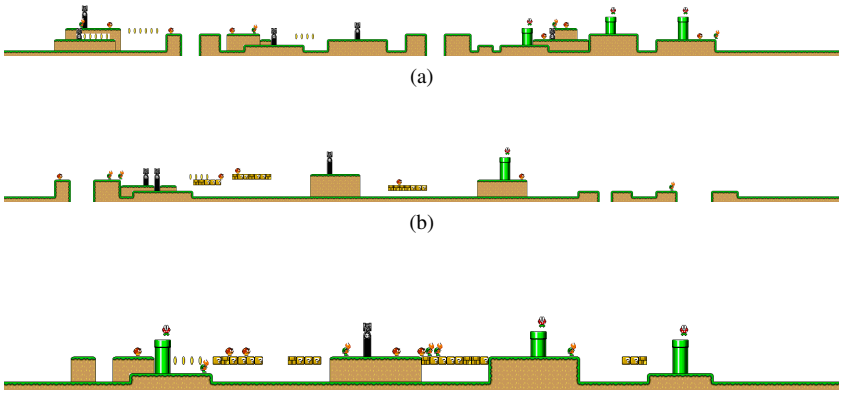
To resolve these conflicts, a priority value was manually defined and assigned to each of the chunks. Hills with bill blasters or piranha plants are given the highest priority followed by blank hills, platforms with enemies (bill blasters or piranha plants) come next then blank platforms and finally come coins and blocks with the lowest priority. After generating a genotype (with possible conflicts), a post-processing step is applied in which the chunks are arranged in a descending order according to their priorities, coordinates and type. The resulted ordered phenotype is then scanned and whenever two overlapping chunks are detected, the one with the higher priority value is maintained and the other is removed. Nevertheless, to allow more diversity, some of the chunks are allowed to overlap such as hills of different height (Figure 5.20. (b)), and coins or boxes with hills (hills here refer to all types

```
<level> ::= <chunks>  <enemy>
<chunks> ::= <chunk> |<chunk> <chunks>
<chunk> ::= gap(<x>,<y>, <w_g>,<w_before>,<w_after>)
  | platform(<x>,<y>,<w>)
  | hill(<x>,<y>,<w>)
  | blaster_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube_hill(<x>,<y>,<h>,<w_before>,<w_after>)
  | coin(<x>,<y>,<w_c>)
  | blaster(<x>,<y>,<h>,<w_before>,<w_after>)
  | tube(<x>,<y>,<h>,<w_before>,<w_after>)
  | <boxes>

<boxes> ::=  <box_type> (<x>,<y>)^2 | ...
  | <box_type> (<x>,<y>)^6

<box_type> ::= block_coin | block_powerup
  | brick_coin  | brick_empty

<enemy> ::= (koopa | goomba)(<pos>) ^2 | ...
  | (koopa | goomba)(<pos>) ^10
<x> ::= [5..95]
<y> ::= [3..5]
<w_g> ::= [2..5]
<w_before> ::= [2..5]
<w_after> ::= [2..5]
<w> ::= [2..6]
<w_c> ::= [2..6]
<h> ::= [3..4]
<pos> ::= [0..100000]
```

Fig. 5.21: The design grammar employed to specify the design of the level. The superscripts (2, 6 and 10) are shortcuts specifying the number of repetition.

of hills; blaster-hills, tube-hills and flat hills). Without this refinement, most levels would look rather flat and uninteresting.

To measure content quality, a relatively simple fitness function was implemented. The main objective of the fitness function is to allow for exploring the design space by creating levels with an acceptable number of chunks permitting for rich design and variability. Thus, the fitness function used is a weighted sum of two normalised measures; the first one, f_p, is the difference between the number of chunks placed in the level and a predefined threshold that specifies the maximum number of chunks that can be placed. The second, f_c, is the number of different conflicting chunks found in the design. Apparently, the two fitness functions partially conflict since optimising f_p by placing more chunks implicitly increases the chance of creating conflicting chunks (f_c). Some example levels generated are presented in Fig. 5.22.



(a)

(b)

(c)

Fig. 5.22: Example levels generated by the GE-generator using the design grammar in Fig. 5.21.

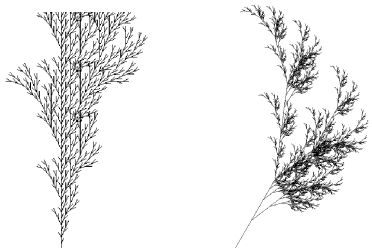
5.7 Lab exercise: create plants with L-systems

At this point, you should be able to implement a bracketed simple L-system to generate plants, which is what you will be doing for this lab exercise. Use an L-system to generate your plants and a turtle graphics program to draw them. You will be given a software package that contains three main classes: *LSystem*, *State* and *Canvas*. Your main work will be to implement the two main methods in the *LSystem* class:

```
public void expand(int depth)
public void interpret(String expression)
```

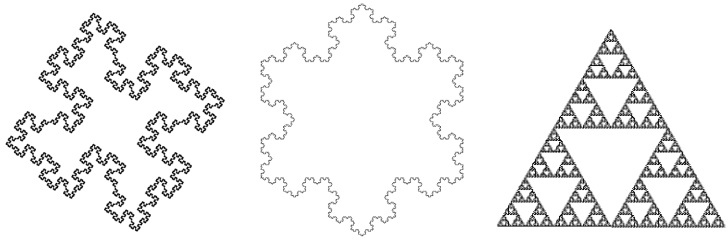
The L-system has an alphabet, axioms, production rules, a starting point, a starting angle, a turning angle and a length for each step. The expand method is used to expand the axiom of the L-system a number of times specified in the *depth* parameter. After expansion, the system processes the expansion and visualises it through the *interpret* method. The result of each step is drawn on the canvas. Since the L-system will be in a different number of states during expansion, a *State* class is defined to represent each state. An instance of this class is made for each state of the L-system and the variables required for defining the state are passed on from the L-system to the state; these include the x and y coordinates, the starting and turing angels and the length of the step. The L-system is visualised by gradually drawing each of its states.

The *State* and the *Canvas* classes are helpers, and therefore there is no need to do any modifications to them. The *Canvas* class has the methods required for simple drawing on the canvas and it contains the main method to run your program. In



(a) (b)

Fig. 5.23: Example trees generated with an L-system using different instantiation parameters.



(a) (b) (c)

Fig. 5.24: Example fractals generated with an L-system using different production rules.

the *main* method, you can instantiate your L-system, define your axiom, your production rules and the number of expansions. Fig. 5.23 presents example L-systems generated using the following rules: $(F,F,F \rightarrow [-F]F[+F][F])$ (Fig. 5.23.5.23a) and $(F,f,(F \rightarrow FF,f \rightarrow F-[[f]+f]+F[+Ff]-f))$ (Fig. 5.23.5.23b). (Note that the rules are written in the form $G = (A,S,P)$, where A is your alphabet, S is the axiom or the starting point and P is the set of production rules).

You can of course use the same software to draw fractal-like forms such as the ones presented in Fig. 5.24. Some example simple rules that you can use to create relatively complex shapes are the followings: $(F,F+F+F+F,(F+F+F+F \rightarrow F+F+F+F,F \rightarrow F+F-F-FF+F+F-F))$ (Fig. 5.24.5.24a), $(F,F++F++F,F \rightarrow F-F++F-F)$ (Fig. 5.24.5.24b) and $(F,f,(f \rightarrow F-f-F,F \rightarrow f+F+f))$ (Fig. 5.24.5.24c).

References

1. Brown, A.: An introduction to Model Driven Architecture (2004). URL http://www.ibm.com/developerworks/rational/library/3100.html
2. Byrne, J., Fenton, M., Hemberg, E., McDermott, J., O'Neill, M., Shotton, E., Nally, C.: Combining structural analysis and multi-objective criteria for evolutionary architectural design. Applications of Evolutionary Computation pp. 204–213 (2011)
3. Chomsky, N.: Three models for the description of language. Information Theory, IRE Transactions on **2**(3), 113–124 (1956)
4. Dormans, J.: Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In: Proceedings of the Foundations of Digital Games Conference Monterey CA, June 2010 (2010)
5. Dormans, J.: Level Design as Model Transformation: A Strategy for Automated Content Generation. In: Proceedings of the Foundations of Digital Games Conference, Bordeaux France, June 2011 (2011)
6. Dormans, J., Leijnen, S.: Combinatorial and Exploratory Creativity in Procedural Content Generation. In: Proceedings of the Foundations of Digital Games Conference Chania, Greece, May 2013 (2013)
7. Hemberg, M., O'Reilly, U.: Extending grammatical evolution to evolve digital surfaces with genr8. In: Proceedings of the 7th European Conference on Genetic Programming, (EuroGP), pp. 299–308. Springer-Verlag (2004)
8. Hornby, G., Pollack, J.: The advantages of generative grammatical encodings for physical design. In: Proceedings of the Congress on Evolutionary Computation, pp. 600–607. IEEE (2001)
9. Lindenmayer, A.: Mathematical models for cellular interactions in development i. filaments with one-sided inputs. Journal of theoretical biology **18**(3), 280–299 (1968)
10. Ochoa, G.: On genetic algorithms and lindenmayer systems. In: Parallel Problem Solving from NaturePPSN V, pp. 335–344. Springer (1998)
11. O'Neill, M., Brabazon, A.: Evolving a logo design using lindenmayer systems, postscript & grammatical evolution. In: IEEE Congress on Evolutionary Computation, pp. 3788–3794. IEEE (2008)
12. O'Neill, M., McDermott, J., Swafford, J., Byrne, J., Hemberg, E., Brabazon, A., Shotton, E., McNally, C., Hemberg, M.: Evolutionary design using grammatical evolution and shape grammars: Designing a shelter. International Journal of Design Engineering (1), 4–24 (2010)
13. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation (4), 349–358 (2001)
14. O'Neill, M., Swafford, J., McDermott, J., Byrne, J., Brabazon, A., Shotton, E., McNally, C., Hemberg, M.: Shape grammars and grammatical evolution for evolutionary design. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 1035–1042. ACM (2009)
15. Prusinkiewicz, P., Lindenmayer, A., Hanan, J.S., Fracchia, F.D., Fowler, D.R., de Boer, M.J., Mercer, L.: The algorithmic beauty of plants, vol. 2. Springer-Verlag New York (1990)
16. Rekers, J., Schürr, A.: A Graph Grammar Approach to Graphical Parsing. In: Proceedings of the 11th International IEEE Symposium on Visual Languages, Darmstadt Germany, May 1995, pp. 195–202 (1995)
17. Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J., ONeill, M.: Evolving levels for super mario bros using grammatical evolution. pp. 304–311 (2012)
18. Tsoulos, I., Lagaris, I.: Solving differential equations with genetic programming. Genetic Programming and Evolvable Machines (1), 33–54 (2006)