

# Lindenmayer systems - a C++ implementation

Steffen Knoblauch

August 6, 2020

Lindenmayer Systems, short L-systems, are the result of **research from Lindenmayer et al.**<sup>1</sup> about the geometric features of plants. L-systems are a concept to mathematically/formal describe and model the growth processes of plant development. They are not only restricted to the plant based developments, but can also be used to generate fractals.

L-systems have an initial state and use rules, like a formal grammar, to transform or rather rewrite the current state to create the next state of the development from a plant or a fractal. It is therefore possible to successive calculate each state of the development. Such a state of a L-system can be interpreted as commands for a turtle graphic, which creates the opportunity to draw the created fractals or plant states.

Goal of this paper is to design an architecture for L-systems, which includes an implementation for L-systems, their creation and an interface for a turtle graphic. The interface should enable the polymorphic use of different turtle graphic implementations and enable drawing of the L-system state.

---

<sup>1</sup>ZITIEREN

# 1 Introduction

L-systems are a formal way to describe plant or fractal development and interpret the result as a graphic. In order to provide a general understanding of L-systems, this paper is organised in several topics. Section 2 is a short introduction to the general idea, based around object rewriting, the grammar of L-systems and the interpretation of a L-system as graphic. After discussing the architecture and possible implementation steps, a final concept for an implementation is proposed. The code for this implementation is available via my github repository.<sup>2</sup>

Finally, there is a conclusion and an outlook for possible future extensions.

## 2 Lindenmayer systems

### 2.1 History

"[L-systems] were introduced in 1968 by Lindenmayer as a theoretical framework for studying the development of simple multicellular organisms [...]"<sup>3</sup> and were later used in computer graphics to generate visuals of organisms and fractals.

On the beginning the focus of L-systems theory was based on larger plant parts and the graphical interpretation used chains of rectangles to display a L-system. Further research into L-system extended the interpretations, resulting in a interpretation of a L-system state with a LOGO-style Turtle. These extensions make it possible to model more complex plants and fractals and display them in a graphical way.<sup>4</sup>

### 2.2 General idea

The general idea of a L-system is the use of a rewriting system based on a formal grammar. The shape of a plant or a fractal consists of geometric pieces, for example a branch of a tree has several subbranches. "When each piece of a shape is geometrically similar to the whole, both the shape and the cascade that generate it are called self-similar."<sup>5</sup> The self-similarity makes it possible to create a formal description for the plant or fractal generation as a formal grammar, further discussed in section 2.3. The rewriting uses this formal description to generate the different states of the development. "In general, rewriting is a technique for defining complex objects by successively replacing parts of a simple initial object using a set of rewriting rules or productions."<sup>6</sup>

For example this concept can be used to rewrite an initial string, called axiom, with defined rewriting rules. A simple example is the following grammar, which consists of only two nonterminals, A and B, and two production rules. The first rule is  $A \rightarrow AB$ , the second rule is  $B \rightarrow A$ . The arrow ' $\rightarrow$ ' symbolises the replacement, the rewriting, of the object on the left with the object on the right of the arrow. The L-system has as axiom the value 'A' and will be expanded with these rules, creating the results in figure 1.

The first step is to use rule one, which replaces the nonterminal A with AB, resulting in the first generation. The result of the first generation ('AB') will be used to generate the second

---

<sup>2</sup><https://github.com/frozenshooter/LSystems>

<sup>3</sup>Zitiert aus abop Preface - Abschnitt Modeling of Plants

<sup>4</sup>Zitat abop Vgl. Seite 6 Chapter 1.3

<sup>5</sup>Zitate The Nature of... chapter 6 page 34

<sup>6</sup>Zitiert von abop Chapter 1 - 1.1 Rewriting systems

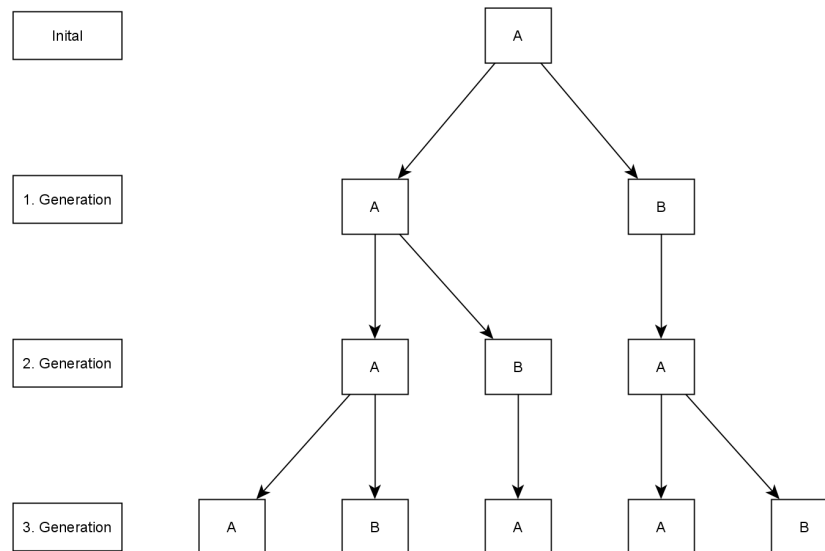


Figure 1: Simple L-system

generation. For all nonterminals the productions will be parallel applied. In this case the non-terminals A and B will be replaced, because both have existing production rules. This results in the second generation with 'ABA' as result.

This process can be successively repeated recursive for an arbitrary amount of generations and create a fractal or plant with self-similar pieces. The more generations are calculated, the more detailed is the resulting state.

## 2.3 Grammar

The definition of an L-system can be done similar to a Chomsky grammar, but there are some general differences. In Chomsky grammars the productions are applied sequentially, whereas in L-systems they can be applied parallel. This has some consequences for the formal properties of an L-system, for example a context-free L-system can produce a language which cannot be produced by a context-free Chomsky grammar.<sup>7</sup>

This paper will only focus on a class of L-systems, the DOL-systems, which can be used for string based rewriting systems. This class is deterministic (D) and context-free (O) and can be formally described by a tuple:

$$G = (V, \omega, P)$$

V: set of symbols as alphabet of the l-system - consisting of terminals and non terminals

$\omega$ : axiom - nonempty word of the alphabet, which should contain at least one nonterminal

P: set of productions

A production consists of a predecessor, a nonterminal symbol of the alphabet, and a successor, the replacement of the nonterminal in the next generation. To guarantee that the l-system is deterministic, there only can be one production for each nonterminal of the alphabet. The identity production is implicit a part of the set of productions.<sup>8</sup>

<sup>7</sup>Zitat abop vgl Seite 3 Abschnitt L-Systems

<sup>8</sup>Zitat vgl Seite 4 abop

If in the process of rewriting a terminal symbol is found, there will be no explicit production applied, but rather the identity production is applied. Terminals will therefore remain in future generations and won't extend an L-system with a production.

As mentioned in section 2.1 there are other extensions of a basic L-systems. The extensions introduce more possibilities for the generation process, like a non-deterministic behaviour. These extensions result in new grammars which can represent much complexer plants or fractals:

- Stochastic grammars: the system isn't deterministic anymore, because there can be multiple production rules for the same nonterminal with a probability which results in a randomisation of the generation
- Context sensitive grammars: production not only look for the nonterminal, they rather look for the symbols before and after the current symbol to process, the context.
- Parametric grammars: it is possible to set additional parameters for a symbol to influence the generation or the evaluation of the data

## 2.4 Interpretation as turtle commands

The L-systems introduced to this point are capable of creating a string based on a grammar. In order to create a graphic of a state of a L-system, the resulting state can be interpreted as commands of a turtle.

A turtle is a concept introduced by the language Logo as a tool for computer graphics.<sup>9</sup>

A state of a turtle can be described as a tuple<sup>10</sup>:

$$S = (x, y, \alpha)$$

$x$  and  $y$  are Cartesian coordinates

$\alpha$  is the direction in which a turtle is facing, called heading

A turtle can move in the Cartesian coordinate system by altering the current state. You can think of it as a real turtle with a pen attached to it, walking on a paper. While moving in the coordinate system, or on the paper, the turtle can draw lines. Given this concept the turtle can receive different commands. The commands let the turtle walk on the paper or the coordinate system by altering the current state and drawing a line. For now I'll restrict this to a two dimensional coordinate system, but it is also possible to enhance it for a three dimensional coordinate system.

The walking path of the turtle is controlled by commands. Therefore a defined step size  $d$  and an angle  $\theta$  is needed to calculate the next state of the turtle.

- Draw: moves one step in the current facing direction drawing a line
- Move: moves one step in the current facing direction without drawing a line
- Right-turn: turns to the right by the angle  $\theta$
- Left-turn: turns to the left by the angle  $\theta$

---

<sup>9</sup>Zitat VGL. Seite 179 Chapter 10 Logo book

<sup>10</sup>Zitat vgl. abop Seite 6 ff

Additional to the state of the turtle the state there is a state for the pen. The pen state consist of the color and its width, which will result in more colorfull or different pictures.

With the given turtle concept is it possible to interpret the result of an L-system. Therefore a mapping between the symbols in the alphabet and the commands which should be called is needed. This could be done by an arbitrary mapping between a nonterminal or a terminal and a command. For this paper the following mapping will be used, but the concept for the architecture includes the possibility to use other mappings in the future.

For the mapping alphabet V is used, which is a simple basic version of a L-system:

$$V = (F, f, +, -)$$

This alphabet will be mapped with these turtle commands:

Symbol	Turtle interpretation
F	Draw a line in the facing direction
f	Move in the facing direction
+	Turn right
-	Turn left

This interpretation enables to draw the result of the L-system by iterating over every terminal and nonterminal of the result state and calling the mapped command. If no command is mapped the symbol will be skipped.

## 2.5 Examples

This section will present some examples for L-system grammars which create fractals. They were created with the implementation, described in section 4 under the use of the command mapping from section 2.4.<sup>12</sup>

### 2.5.1 Hilbert curve

This fractal will be generated with the simple axiom 'L' and tow productions:

- $L \rightarrow +RF-LFL-FR+$
- $R \rightarrow -LF+RFR+FL-$

The turtle will be initalized with an angle of  $90^\circ$  and length  $d$  of 5 for a step. The result is displayed in figure 2.

### 2.5.2 Sierpinski triangle

This fractal will be created with 'F' as axiom and two productions:

- $X \rightarrow YF+XF+Y$
- $Y \rightarrow XF-YF-X$

The turtle will be initalized with an angle of  $60^\circ$  and an arbitrary length  $d$  for a step. The result is displayed in figure 3.

<sup>11</sup>Zitat vlg cad book seitev 2 unteres drittel

<sup>12</sup>The grammars and further examples can be found under: **examples1997**



Figure 2: Hilbert L-system - 7 generations



Figure 3: Sierpinski L-system - 9 generations

## 3 Architecture

The primary goal of this paper is creating an architecture and an implementation of a L-system as described in section 2. The focus of the architecture will be on flexibility and expandability and therefore is the following section splitted into several parts with discussions about different aspects of the final architecture.

### 3.1 Requirements overview

The architecture for a L-system, as introduced in section 2, has several requirements. This section will introduce the needed core requirements for an architecture with a flexible L-system and turtle.

The core of the concept is the L-system itself, which is the base to guarantee a flexible use. The L-system holds relevant data like the grammar, consisting of productions and an axiom. The L-system should offer a way to configure a grammar and guarantee the usage in different architectures, without hardcoded grammars. The L-system has to be able to successively generate the next states with configured grammar and grant access to this generated result.

The turtle is another key component, which will be used to interpret the result of a L-system. The turtle has to offer the typical turtle commands as introduced in section 2.4. In order to provide a turtle that is as extensible as possible, it should be possible to implement it flexibly. Therefore an interface should be offered, that enables a fast exchange of implementations. Depending on the implementation of a turtle, it should be possible to configure the turtle and change properties, like the color or line width.

A mapping between the turtle commands and the L-system alphabet is an important part, so it is possible to call the correct turtle commands. In order to create an image of a plant or a fractal development state, it is required that the mapped commands get called, after interpreting the L-system result.

### 3.2 L-system

This section provides a discussion of possible implementations of the L-system and a final proposal.

A simple and rather naive idea would be an object which holds the L-system as a string and additional the grammar, consisting of productions and an axiom. This L-system could offer a function that calculates the next generation by iterating over every char in the string. If there is a production for the current char, the char will be replaced according to the production with the successor. This function could be called for n generations to create the n-th generation of the L-system, as show in figure 4.

This idea has several design flaws resulting in a bad performance and unflexible architecture. The first flaws exist because of the lack of separation between datastructure and processing of the data. It is not possible to simply exchange the datastructure or the function without the separation, which affects flexibility of the L-system negative. This can be solved by extracting the functionality into a separate function and the datastructure remains as "dumb" object. The separated function can be used to manipulate the current state in the datastructure itself and also creates the opportunity to exchange the underlying datastructure, as long as the datastructure fullfills some formal properties, like the support of access to the string and grammar.



Figure 4: Naive L-system

For a more flexible design additional changes could be done, because the current idea bases on chars as alphabet of the grammar. The L-system could be enhanced by allowing arbitrary objects as symbols of the alphabet, as long as they provide certain functions, like a operator to compare them. Instead of a string, the objects could for example be saved as a list.

A further flaw is the bad performance, because of the use of a string to save the current state/-generation. There are two reasons why the performance is bad, especially for larger generations. The first reason is the size of the string itself, which will grow very fast because of the nature of a L-system. Every nonterminal will be often replaced by severall symbols, which enormously increases the total size of the string for each generation. Additional to the large amount of space needed for the string, the replace in a string needs to be done for each nonterminal, which can result in a big overhead. Because of this problems, another more efficient design is needed.

This implementation only works for simple L-systems where only a few generations are needed.

In order to solve this problems and to create a more efficient and flexible architecture, let's recap the nature of L-systems. The generation of a L-system is based on a rewriting process. This process iterates over every object in the current state and calcualtes the resulting generation by applying the productions. Because of the restriction to DOL-systems, the rewriting is not context-sensitive and can be done independent from other objects of the current state. If a object gets replaced by the successor of the production, the result of this replacment can be handled independent. Consequently, the calculation of the n-th generation can be done for each object on their own and can simply be done in a recursive manner.

There is only a limited number of productions in a grammar of a L-system. Each of these productions is choosen deterministic when comparing the current nonterminal with the predecessor. The next generation is created by rewriting the current state, when a nonterminal is often in the current state, the same production can be used. If the same production is used muliple times, the resulting generation has a lot of similar objects or more specific the same string muliple times. The repeating data and the recursive calculation can be used to improve the naive idea.

The simple example in figure 5 shows the possible reduction of data using the following grammar:

Axiom: A

Production:  $A \rightarrow AAAAAAAAAA$

This rather ideal example shows how it would be sufficient to save just parts of the resulting string to represent the whole generation of a L-system. For this example it could be possible to save "AAAAAAAAAA" and how often it is needed. Because of the repetitions, this is partly even for more complex grammars possible.

With this knowledge multiple improvements can be discussed. The first possible improvement is storing the data not in a simple list, like a string, but instead in a more sophisticated datas-





Figure 5: Memory reduction

tructure. For this case a graph as datastructure is better suited. Such a graph enables a better performance for replacements and reduction of memory space.

The graph has a special sturture to achive the improvements for a L-system. The following grammar is used to demonstrate the possible improvements:

Axiom: A

1. Production:  $A \rightarrow BBB$

2. Production:  $B \rightarrow AAA$

A natural improvement of a graph is the replace function. This could be done by simply creating new childnodes with the replaced data. In figure 6 is a example with the given grammar. Each generation has its own nodes with data of the current state. A level of this graph represents a whole generation and can be accessed with the depth of the nodes.

This graph still has the problem of the redundant data, because every note contains its own data. For example in the second generation the part "AAA" is stored three times. This can be improved with another indirection of the data. This concept results in a datastructure containing a graph to store the dependencies and a container for the data itself. In figure 7 is an example for this indirection. Each node saves a pointer or id of the data it represents. In this graph a replace could also be done with the creation of a childnodes.

Both of the described graphs hold every calculated generation, which allows the access to an arbitrary generation with the restriction to the deepest calculated generation. To reduce the amount of storage needed even more, it is possible to neglect if all generations are needed or only the current generation. If only the last calculated generation is needed, the graph can be simplified even more. To discuss the simplification, it is necessary to look further in the different ways to do the replacement. It is possible to iterate over the node data and replace for each symbol of teh data at a time. For example with the current grammar the data "AAA", could be replaced by replacing each "A" at a time. Because of the focus of DOL-Systems, another possible method is to replace a node with all childnodes at once.

There seems to be no difference, but in combination with the proposed next step of the graph, it is relevant. If not all generations have to be stored, it is possible to remove a completely replaced parentnode. For example in figure 8, it is not needed to save the crossed out nodes, because they are completely replaced. In this simple example is it already possible save only four instead of

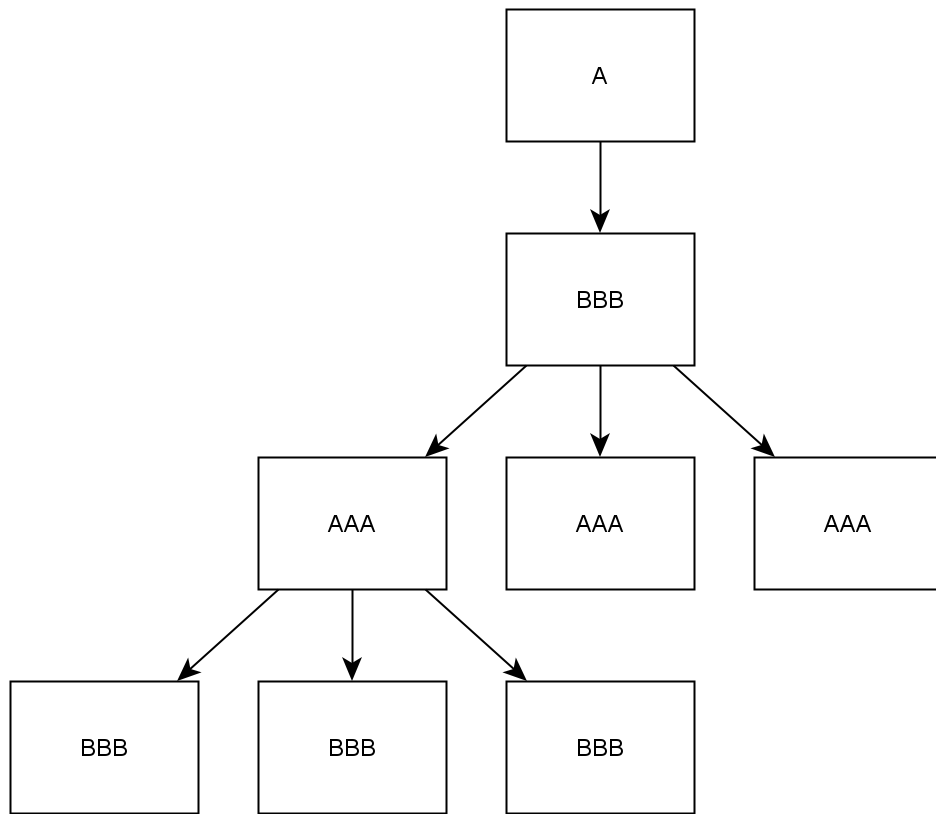


Figure 6: L-system as graph

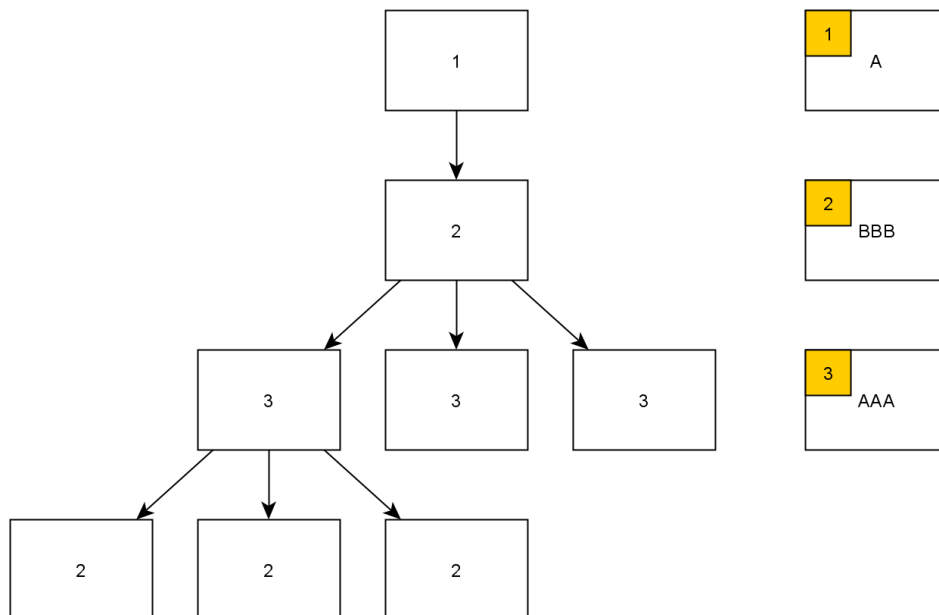


Figure 7: L-system as graph with improved memory consumption

six nodes and reduce the data. This results in a flat graph with only a few nodes depth, but

only the current state of a L-system. For this graph, it is important how the replacement is done. If the replacement is done with the all at once method, the parent node can be deleted directly. If the step by step replacement is done, it is necessary to alter the parent node data for each replacement. Only when the complete data and the parent node is replaced, this node can be deleted. The complete replacement is simpler to implement, because the altering of parent node data is not needed.

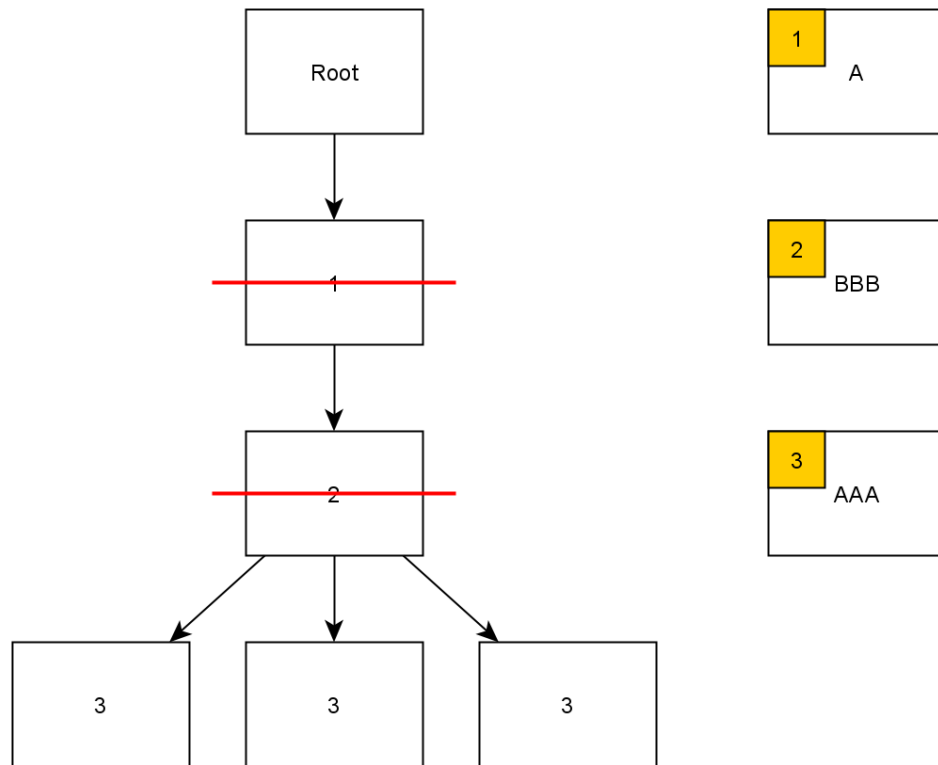


Figure 8: L-system as graph with one generation

A graph is as consequence a faster and memory saving approach in comparison to the naive L-system.

Another possible simplification is to neglect storing the generations of a L-system at all. Instead of saving the generation of a L-system in a datastructure, the result can be calculated on demand and never stored. The only data such a L-system would hold, would be the grammar itself. The calculation can be done with a recursive function, that will return the generation. Clearly an advantage is the reduction of needed storage space, but this reduction is only achieved at the cost of recalculation for each generation. Whenever a generation is needed, it is necessary to calculate again and it is not possible to use a previous result. If a generation is needed multiple times or if it is needed to setp forwards and backwards in the result, a solution could be to store the result in another object. This calculations also guarantees a flexibility, because you can either only calculate the generation once and use the result directly, like in this case to call the turtle commands, or save it for later usage.

The recursive function can be designed flexible in order to allow a simple exchange of a L-system datastructure. This results in two main components, the L-system datastructure and the function, which accepts a L-system datastructure and calculates the generation. The L-system,

which the function receives, has to fullfill some requirements, like to offer access functions. This will be described in further detail in the implementation in section 4.

There are the two possible ways to handle the L-system. The storage as a datastructure, based on a graph, and the recursive calculation. For this paper it is completely sufficient to calculate the generation on demand, because it is only necessary to call the turtle command once for each object of the L-system generation. As consequence no implementation of the graph is needed. Other use cases in the future might need to store the data. This would even be possible with the recursive function, when the result is stored in another object, for example a graph. The proposal for the L-system is shown in figure 9.

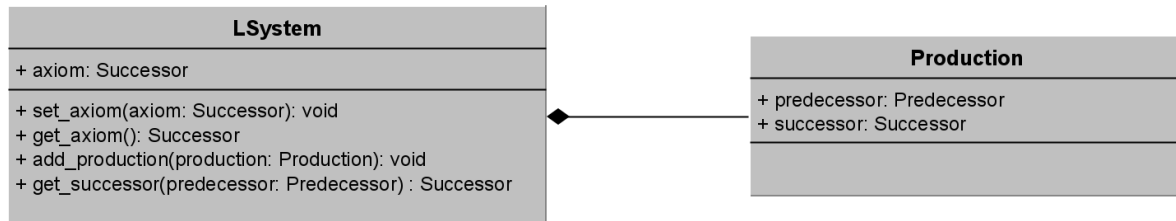


Figure 9: L-system proposal

The proposed L-system will just hold the grammar for a L-system, which will be given to the recursive function. This L-system can be used in a flexible way, because it can be configured without a restriction of the grammar. Additional, there is only a general type restriction for the axiom and the productions. The types called Successor and Predecessor can be chosen freely, but they need to fullfill formal requirements. The self-similarity and ergo the rewriting need to be represented by the types. The Successor type must consist of Successor objects itself, which can be splitted into objects of the Predecessor type. Each Predecessor object must be comparable to another Predecessor to be able to rewrite the L-system. The type of the axiom and the successor have the same type, in order to guarantee that they are splittable in smaller parts.

To this point the discussion doesn't include how the turtle commands can be called. As already mentioned there must be mapping between the commands of turtle and the alphabet of the L-system. The mapping will be discussed further in section 3.4 and for now we just assume a mapping exists in some way. When using a graph, it would be possible to call the turtle commands by accessing a generation data and interpret the objects.

Here the implementation is based on a recursive function. This function can call the turtle commands for a generation or more specific when the recursion depth is reached. How this will be done in detail is described in section 4, but is based around the idea of an output iterator. The recursive function will receive an arbitrary output iterator and instead of interpreting the data in the function itself it will just hand over the objects of the generation to the iterator. This iterator can then interpret the objects as turtle commands and call them. The iterator has no restrictions on what to do with the data. For example the iterator may be used to save data in an arbitrary datastructure or print it in a file.

Overall allows this concept, consisting of the L-system datastructure and the recursive function, a possible user a flexible field of operation. The recursive function and the L-system datastructure can be used completely independent. For this concept, these components will be used together, in combination with other components, like a turtle and a turtle command mapping.

### 3.3 Turtle

A quiet important component of the architecture is the turtle. The turtle has to offer the typical commands as described in section 2.4. A turtle should be so flexible that it is possible to implement it for different Frameworks, for example OpenGL or the Cairo graphics library. An abstract interface, with a minimal set of commands, will allow to implement this. A turtle like this is not only bound to a special graphics framework, but can be even implemented for other usecases, like generating text on the console or a file. The most general or abstract version of a turtle offers the four minimal functions of a turtle, as shown in figure 10.

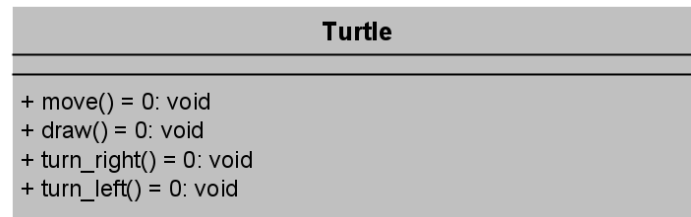


Figure 10: Minimal turtle

More specialiced versions can offer more functions, depending on other needs and completly independent of the paper usage of the turtle. For example an OpenGL implementation can offer more commands to represent a three dimensional turtle, like rotate around an axis. Additional to more commands, also different ways to configure a turtle can be offered for each specialization. Some possible implementations, in connection with this paper, will be described in section 4.

Consequently the general interface is not only sufficient for this paper, but also for other usecases in the future.

### 3.4 Turtle command mapping

It is nessecary to map a command to an object in order to interpret the result of a L-system. As mentioned in section 3.2 a recursive function will be used to calculate the result of a L-system. Until this point we just assumed there was a mapping between the members of the alphabet of the L-system and the commands which will be called. In this scenario the recursive function receives an iterator, which can handle the generated data.

The command mapping iterator is an example of an output iterator, which will handle data interpretation and calls of turtle commands. In figure 11 is the a concept of this iterator displayed , which will handle a L-system based on strings and chars. Such an iterator receives data, interprets it according to the mapping and calls the specified command.

The recursive function will receive this iterator as parameter and just hands the result over to it. The command mapping iterator just has to select the mapped turtle command and call it.

The indirection in form of an iterator offers a clear separation between the calculation of the L-system generation and the handling of the data. In order to offer different mappings, new implementations of the iterator are nessecary. In the implementation, in section 4, there is the implementation of the iterator example in figure 11 based on the grammar introduced in previous sections.

The architecture is not only restricted to iterators comparable to command mapping iterator, but every output iterator can be used to receive the data and handle it differently.

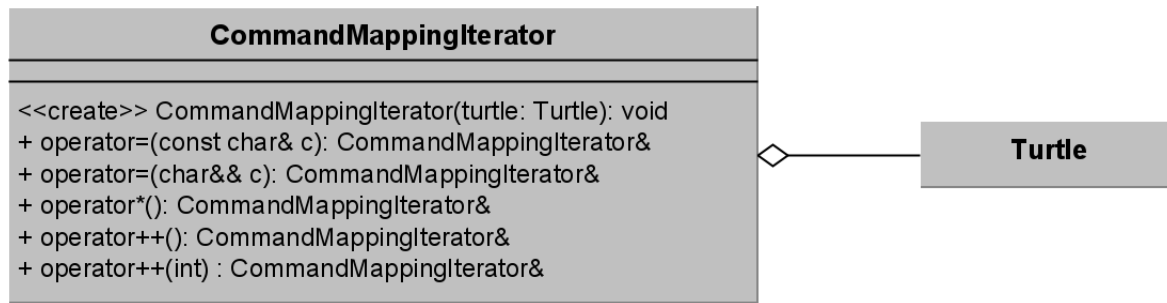


Figure 11: C++ interface command mappin iterator

### 3.5 Overview

This section will now give a short overview of the discussed components and their dependencies. The general approach for the design was a flexible use of the separated components. It should be possible to use each component on their own and even exchange a component and still guarantee the functionality. Under the premises, that exchanged components still offer the same functionality and data. In figure 12 a simplified overview of the dataflow is illustrated.

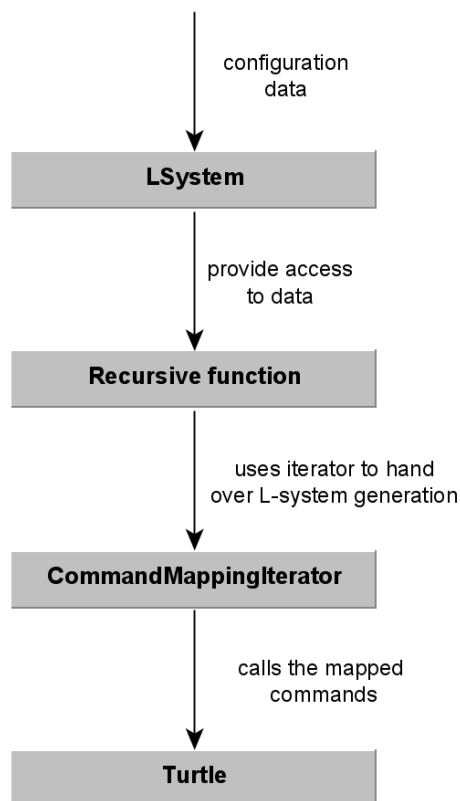


Figure 12: Simplified overview over the data flow

The loaded data will be used to initialize the L-system. The L-system itself needs only to hold the grammar and provide the access to it. The core of the L-system functionality is the recursive function, which will be used to calculate a L-system generation. This function offers a quiet flexible interface, because it should allow to use different L-system implementations, as long as the implementation provides a set of needed functions. These functions will be discussed in detail in section 4. Another key part of the recursive function is the support of an arbitrary output iterator to hand over the calculated generation. This introduces not only a flexible usage of the function itself, but integrates well with already existing containers.

The `CommandMappingIterator` is such an output iterator, which allows to use the provided interfaces to call the correct turtle command. Whenever data is handed over from the recursive function to the iterator, the iterator looks the data up and calls, if specified, the turtle command. The iterator here is implemented for a determined mapping, but can be replaced by another implementation, as long as the implementation provides the typical output iterator functionality.

The final component is the turtle, which is used as abstract interface. This interface enforces the absolute minimal set of functions a turtle has to fulfill. Each specialization of a turtle has to independently handle their configuration and additional dependencies.

As conclusion all the components can be exchanged with other objects, as long as they provide the needed interfaces. The needed interfaces will be described in detail in section 4 and additionally there are comments in the provided code.

---

<sup>13</sup>Configuration loader beschreiben

## 4 Implementation

This section provides the in detail description of the implementation for the components discussed in the previous sections. Additional to the components other aspects of the implementation, like the build system are discussed.

### 4.1 Build System

The first important point for the implementation is the question what build system to choose or if a build system is even needed. A build system is needed or at least comfortable to use because of two points. The first point is a fast way to build a complexer design and even extend it with libraries in a fast way. The second point is the use on different operating systems, which also plays a big role for the question what build system to choose.

A typical build system for C++ projects are Makefiles, which include a description what and how to build. The problem with Makefiles is the restricted support on Windows, which results in either an inconvenient setup or to choose another build system. In this context CMake comes to mind, a generator for buildsystems. It is possible to define the project in a file and CMake can generate files for different build systems, on different operating systems. These resolves not even the problem with the use of Windows, but enables other developers to build it on the system their choice.

The project setup is straightforward and consists mainly of the C++ version and the files to build. A more complicated part is the setup to use a library. A library is needed for the implementation of a specilization of a turtle, the CairoTurtle. This brought up some problems, because of the devlopment on a windows machine, where the library handling is done a bit different. After some research, a sample from J. Preshing<sup>14</sup> provided a working setup for CMake and he even provided some precompiled library files<sup>15</sup>, called dll on Windows. The setup can be found in the provieded code and should work on different operating systems.<sup>16</sup>

### 4.2 ConfigurationLoader

Hier noch abklären in welchem umfang das gemacht werden soll

---

<sup>14</sup>`cairoinclude`

<sup>15</sup>`cairodll`

<sup>16</sup>Only tested on a windows machine with some additional setup for convenience



### 4.3 LSystem

The different variants of L-system implementations are discussed in previous sections and this chapter only focuses on the proposed concept, consisting of a L-system, holding a grammar.

The proposed L-system should be as flexible as possible and has therefore templates for the underlying types. The templates are called Predecessor and Successor, like the members of a production, for a reason. In order to guarantee a correct behaviour, they need to be the same type, because if they would differ, it wouldn't be possible to rewrite a L-system.

---

```
1 template <typename Predecessor, typename Successor>
2 class LSystem {
3     void set_axiom(const Successor& axiom) {
4         ...
5     }
6
7     std::template shared_ptr<Successor> get_axiom() {
8         ...
9     }
10
11    void add_production(const Production<Predecessor, Successor>& production) {
12        ...
13    }
14
15    std::template shared_ptr<Successor> get_successor(const Predecessor& predecessor) {
16        ...
17    }
18
19 private:
20     Successor axiom_;
21     std::unordered_map<Predecessor, Production<Predecessor, Successor>> production_rules_;
22 }
```

---

These templates have some general restrictions, because of the formal properties of a L-system, like the self-similarity. In order to enable the rewriting, the Successor type needs to be consisting of smaller parts which also can Successors objects. A Successor object needs also to be splittable into Predecessor objects, this has the reason, that the recursive function needs to split an Successor object into parts and use a production on it. This might sound complex, but with the concrete example of a string as Successor and char as Predecessor it should be straightforward. A string can be splitted into smaller strings, fulfilling the first requiriment. The string can be splitted into chars and therefore fullfills also the other requirment.

This L-system implementation uses the production object, to handle the different rewriting rules. A production is also a templated object, but only as a simple data holder without special functionality. The productions are saved in a map for fast access, because the recursive function will call *get\_successor*() for each element, each Predecessor object, of a L-system generation.

### 4.4 Recursive function

The recursive function will call

## 4.5 CommandMappingIterator

## 4.6 TurtleGraphic

### 4.6.1 TestTurtle

### 4.6.2 CairoTurtle

### 4.6.3 Further implementations

SVG implementation

## 5 Tests

## 6 Example usage

## 7 Problems and Restrictions

## 8 Outlook

### 8.1 Configuration

Some of the discussed components in this paper can be configured. The configuration could either be done in code, for example by setting it with constants, or in a dynamic way. A dynamic way would be possible by loading the configuration data from a stream, for example a filestream. For the configuration of the introduced components it is sufficient to use simple key value pairs. For example such file could contain the configuration of a L-system:

```
generations 14
axiom X
rule X YF+XF+Y
rule Y XF-YF-X
```

In this example the key is separated by the value with a space and in each row there is another key value pair. A problem is, that a value can not only consist of a single value, but can instead have of multiple values. For example a rule can consist of a predecessor and a successor value. A solution for the multi value problem could be the use of a JSON object. A JSON object would allow to encapsulate the values without a restriction, but for this paper a simpler approach should be sufficient. As solution for the multi value objects is a simple separation of the values with a space. This is already shown in the example above, where a rule is splitted into three values. When reading the data from a stream it is necessary to know what datatype the values have and therefore a mapping between the keys and the value type is needed. This can be done by a static mapping in the code, which will also allow to solve this in a flexible way. At least so flexible that the mapping can be set in a central place at the code.

<sup>17</sup>

18 19

---

<sup>17</sup>Hier muss man noch das genaue design beschrieben -> wie werden die daten übergeben bzw wie werden die daten gehalten oder werden die daten direkt übergeben

<sup>18</sup>FÜR ALLE UMLS DIE MEMBER VARIABLEN UM DEN UNTERSTRICH ERGÄNZEN

<sup>19</sup>prusinkiewicz.p.lindenmayera.2004