Lindenmayer systems - a flexible C++ implementation

Steffen Knoblauch
August 13, 2020

Lindenmayer Systems, short L-systems, are the result of research from Lindenmayer et al. about the geometric features of plants [1]. L-systems are a concept to mathematicaly/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 inital state and use rules, like a formal grammar, to transform or rather rewrite the current state to create the next state of the plant or fractal development. Because of that, it is possible to successive calculate each state of the development. This state 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 calculation and an interface for a turtle graphic. The interface should enable the polymorphic use of different turtle graphic implementations and enable to draw a state.

1 Introduction

L-systems are a formal way to describe plant or fractal development and it is possible to interpret the result as a graphic. This paper aims not only to provide an overview of Lsystems, but also includes the discussion about a possible architecture and is organised as follows. Section 2 briefly introduces 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 a github repository.[1]

The remainder of the paper provides some examples and a conclusion with future enhancements.

2 Lindenmayer systems

2.1 History

"[L-systems] were introduced in 1968 by Lindenmayer as a theoretical framwork for studying the development of simple multicellular organisms [...]'[1, Preface, p. VI] and were later used in computer graphics to generate visuals of organisms and fractals.

In the beginning the focus of the L-system theory was based on larger plant parts. The graphical interpretation used chains of rectangles to display a L-system. Further research into L-system resulted in new interpretations and extensions. For example an 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. [1, Cf. Chapter 1.3, p. 6].

2.2 General idea

The use of a rewriting system, based on a formal grammar, is the general idea of a L-system. 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."[2, Chapter 6, p. 34] The self-similaritiy makes it possible to create a formal description for the plant or fractal generation as a formal grammar. The formal grammar is further discussed in section 2.3. Rewriting uses this formal description to generate the diffferent 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."[1, Chapter 1.1, p. 1].

This concept can be used to rewrite an inital string, called axiom, with specific rewriting rules. A simple example is the following grammar, which consits of only two nonterminals, A and B, and two productions. The first production is A→AB, the second one is B→A. The arrow '→' symbols 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, creating the results in figure 1.

As first step, using rule one, the nonterminal A is replaced with AB, resulting in the first generation. The result of the first generation process ('AB') will be used to determine the second generation. For all nonterminals the productions can be parallel applied. In this case, the nonterminals A and B will be replaced, because both have existing productions. This results in the second generation with 'ABA' as result.

[1]See https://github.com/frozzenshooter/LSystems

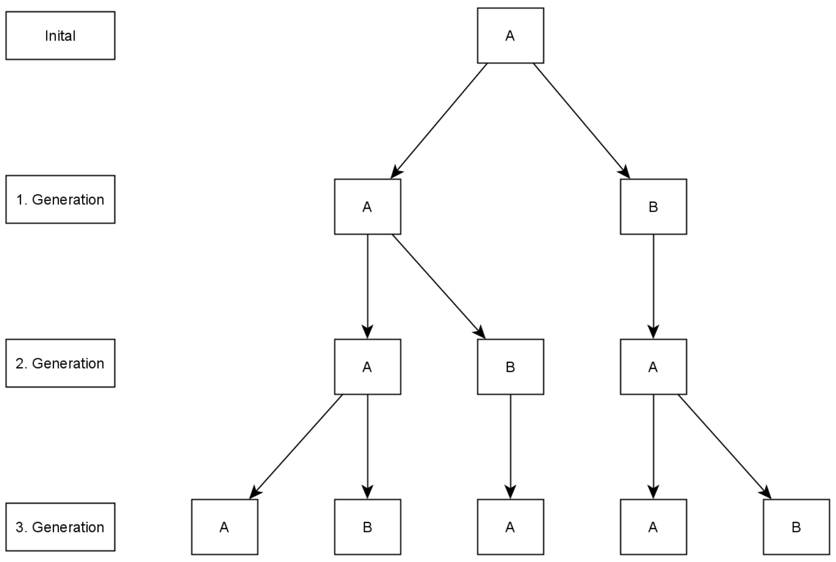


Figure 1: Simple L-system

This process can be successively and recursively repeated for an arbitrary amount of generations and create a fractal or plant with self-similar parts. 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.[1, Cf. Chapter 1.1, p. 3]
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 formaly described by a tuple:

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

V: set of symbols as alphabet of the L-system - consisting of terminals and nonterminals
ω: 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 replacment of the nonterminal in the next generation. To guarantee the deterministic characteristic of a L-system, it is relveant that there is only one production for each nonterminal of the alphabet.The identity production is implicit a part of the set of productions.[1, Cf. Chapter 1.2, p. 4f.]

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 the result further.

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

- Stochastic grammars: the system isn't deterministic anymore, because there can be multiple productions with a probaility for the same nonterminal. This results in a randomisation of a generation.
- Context senstive grammars: productions don't only look for the nonterminal, they rather look for the symbols before and after the current symbol (the context).
- Parametric grammars: it is possible to set additional parameters for a member of teh alphabet 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 L-system state, this 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.[3, Cf. Chapter 10, p. 179]
A state of a turtle can be described as a tuple[1, Cf. Chapter 1.3, p. 6ff]:

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

x and y are Cartesian coordinates
α 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. It is possible to think of it as a real turtle with a pen attached to it, walking on a paper. While moving thourgh the coordinate system or the paper, the turtle can draw lines. Given that, the turtle can receive different commands. The commands alter the current state and let the turtle walk on the paper or the coordinate system, drawing a line. For now it will be restricted 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 θ , the difference to the current faceing direction, is needed to calculate the next state of the turtle. These values will be used by the following general commands: [4, Cf. Chapter 2]

- 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 θ
- Left-turn: turns to the left by the angle θ

Additional to the turtle state, with position and facing direction, there is a state for the pen. The pen state is part of a turtle and consist for example of a color and or a line width.

The explained turtle concept makes it possible to interpret the result of a L-system. Therefore, a mapping between the symbols in the alphabet and the commands, which should be executed,

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.
Alphabet V, a basic version of a L-system, is used for the mapping:

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

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. [2]

2.5.1 Hilbert curve

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

- L→+RF-LFL-FR+
- R→-LF+RFR+FL-

The turtle will be initalized with an angle of 90° and length $d = 5$ for a single 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→YF+XF+Y
- Y→XF-YF-X

The turtle will be initalized with an angle of 60° and length $d = 3$ for a single step. The result is displayed in figure 3.

[2]The grammars and further examples can be found under: R. Dickau, *Two-dimensional l-systems*, 1997. [Online]. Available: `http://mathforum.org/advanced/robertd/lsys2d.html` (visited on 08/05/2020)

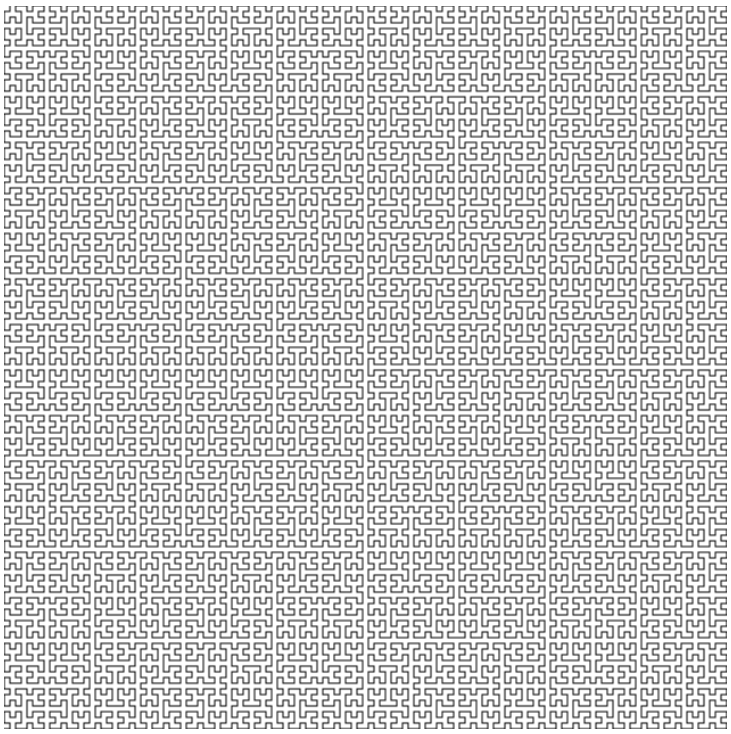


Figure 2: Hilbert L-system - 7 generations

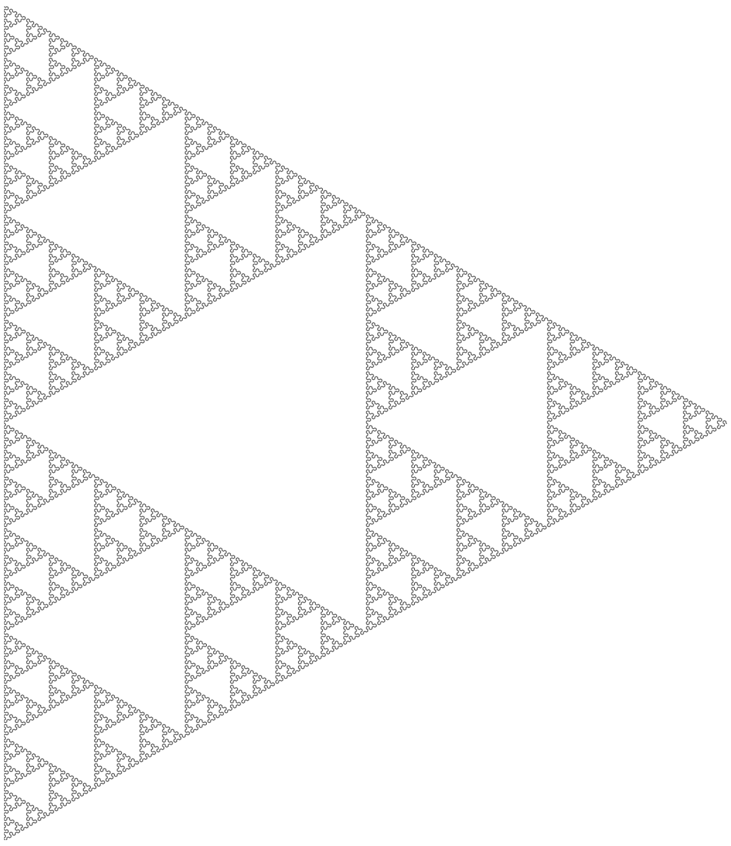


Figure 3: Sierpinski L-system - 9 generations

3 Architecture

The primary goal of this paper is an architecture and an implementation for a L-system. The focus of the architecture is 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 core requirements for an architecture with a flexible L-system and a 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 a configured grammar and grant access to the 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 flexible. 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 turning angle or line width.

A mapping between the turtle commands and the L-system alphabet is an important part. Because of that, 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 dissussion of possible implementations of the L-system and a final proposal.

A simple and rather naive idea is an object which holds the L-system generation as a string. Aditional it holds 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.

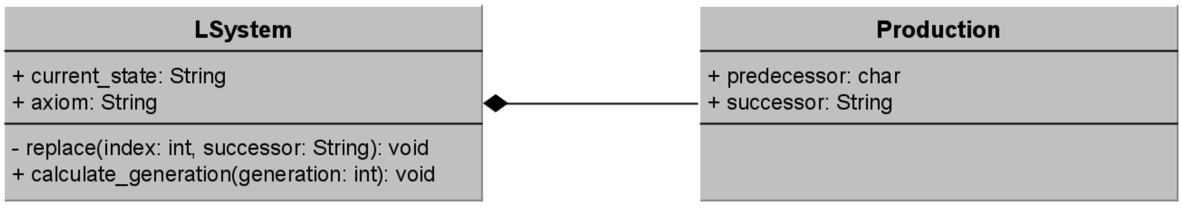


Figure 4: Naive L-system

This idea has several design flaws resulting in a bad performance and unflexible architecture. The lack of seperation between datastructure and processing of the data defines the first flaw. It is not possible to simply exchange the datastructure or the function without the seperation, which negatively affects the flexibility of the L-system. This can be solved by extracting the functionality into a seperate function and the datastructure remains as "dumb" object. The seperated function can be a used to manipulate the current state in the datastructure itself. This also creates the opportunity to exchange the underlying datastructure, as long as the datastructure fullfills some formal properties, like the access to the string and grammar.
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 several 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. Therefore, this naive concept works only for simple L-systems where only a few generations are needed.

In order to solve these 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 calculates 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 handeled 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 is used often. If the same production is used multiple 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 →AAAAAAAAAA

This rather perfect 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 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, which is basically a list of chars, but instead

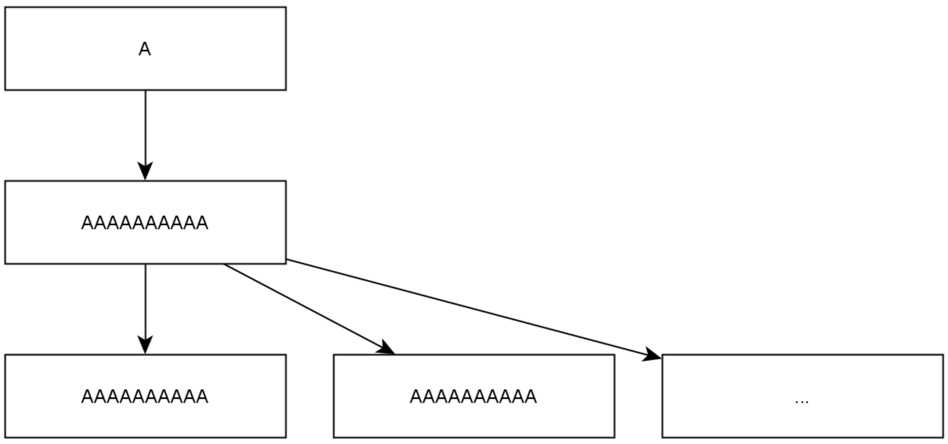


Figure 5: Memory reduction

in a more sophisticated datastructure. 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 →BBB
2. Production: B →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 an 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.

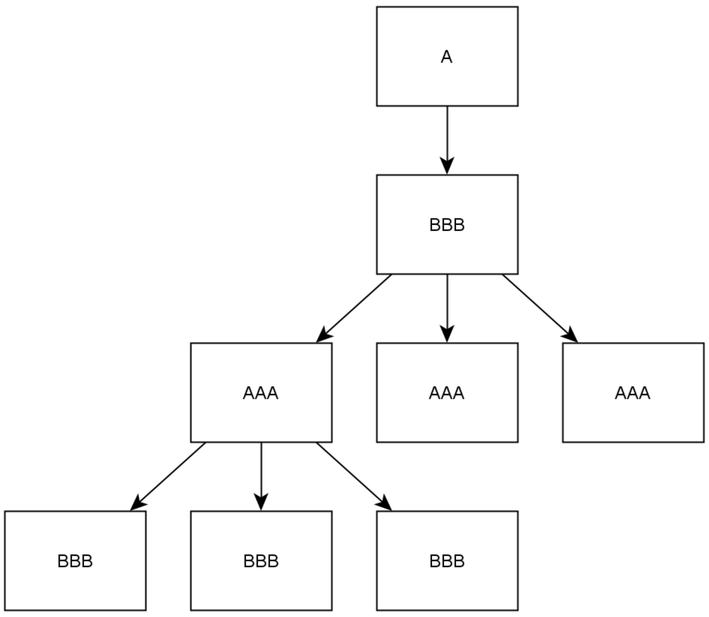


Figure 6: L-system as graph

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.

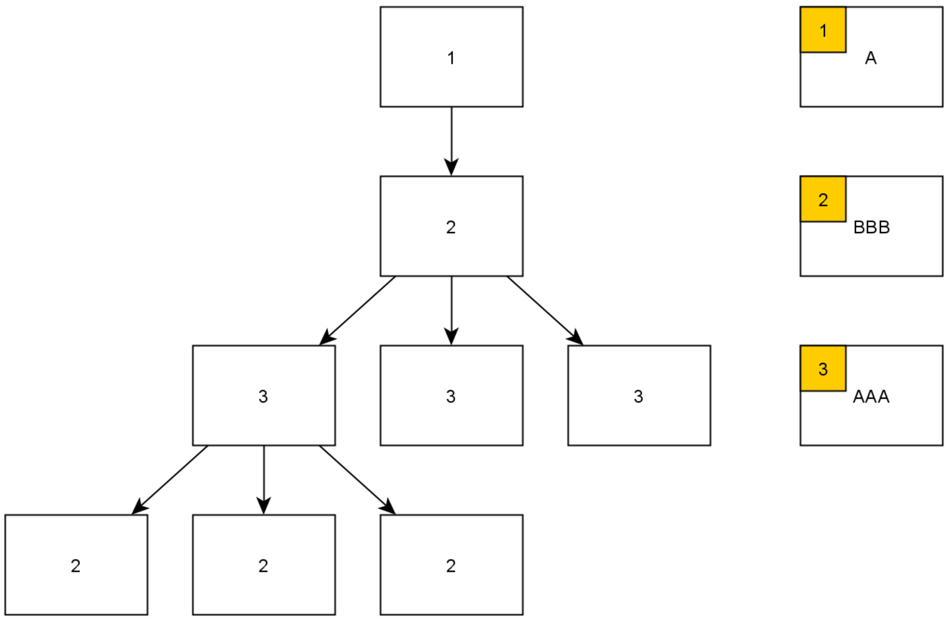


Figure 7: L-system as graph with improved memory consumption

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 each symbol 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 and therefore, the whole node.
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 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 replacment is done with the all at once method, the parentnode can be deleted directly. If the step by step replacement is done, it is nessecary to alter the parentnode data for each replacement. Only when the complete data and the parentnode is replaced, this node can be deleted. The complete replacement is simpler to implement, because the altering of parentnode datacan be omitted.

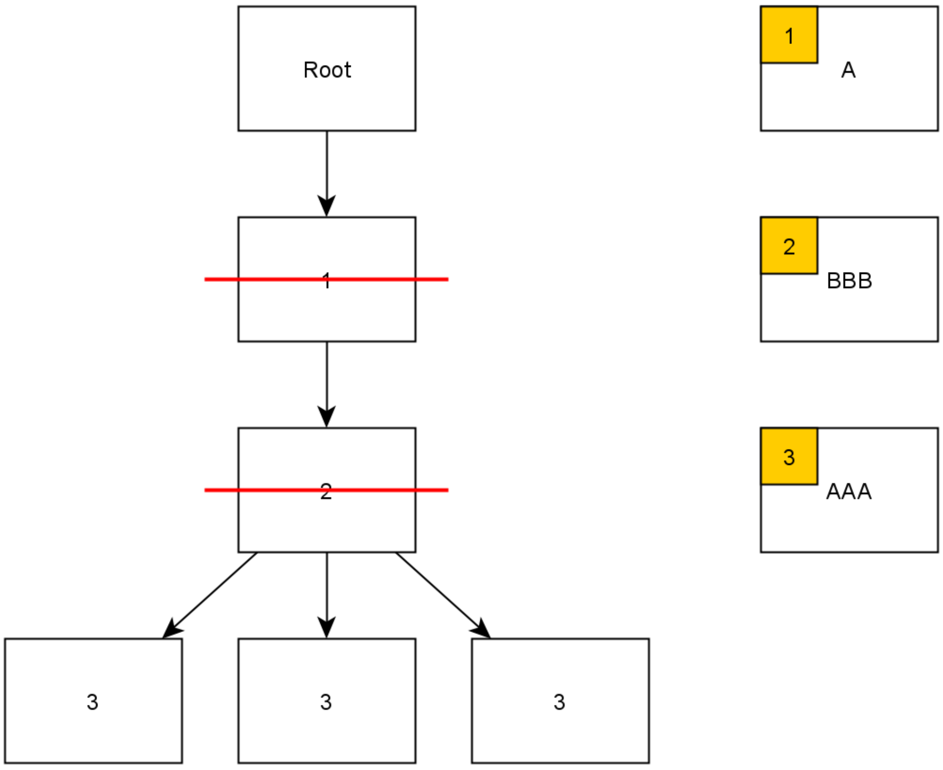


Figure 8: L-system as graph with one generation

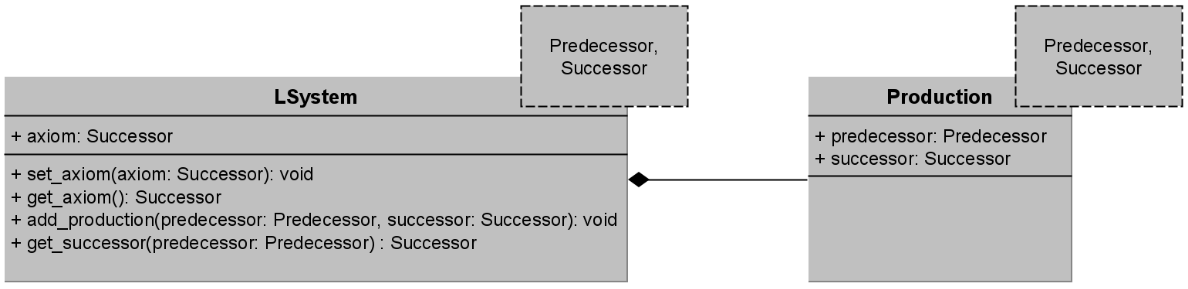
As consequence a graph is a faster and more memory saving approach in comparison to the naive L-system.

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 an additional recursive function, that will return the generation. Clearly an advantage is the reduction of needed storage space, but this reduction is only achived at the cost of recalculation for each generation. Whenever a generation is needed, it is nessecary 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 step forwards and backwards in the result, a solution could be to store the result in another object. This calculations also guarantees 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 even more flexible by allowing a simple exchange of a L-system datastructure.

This results in two main components, the L-system datastructure and the recursive 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.

As discussed, there are two general concepts 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.



Predecessor, Successor
LSystem
+ axiom: Successor
+ set_axiom(axiom: Successor): void
+ get_axiom(): Successor
+ add_production(predecessor: Predecessor, successor: Successor): void
+ get_successor(predecessor: Predecessor) : Successor
Predecessor, Successor
Production
+ predecessor: Predecessor
+ successor: Successor

Figure 9: L-system proposal

The proposed L-system, which will be given to the recursive function, just holds the grammar. This L-system can be used in a flexible way, because it can be configured without an arbitrary grammar of a DOL-system. Additional, there is only a general type restriction for the axiom and the prodcutions. The types called Successor and Predeccessor can be choosen freely, but they need to fullfill formal requirements. The self-similarity and ergo the rewriting need to be represented by these 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 axiom has also from type Successor, in order to guarantee that it is splittable in smaller parts and cen be rewritten.

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

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 smart iterator can then interpret the objects as turtle commands and call them. Nevertheless, the iterator has no restrictions on what to do with the data. For example the iterator may be also used to save data in an arbitrary datastructure or print it in a file.

Overall this concept allows a flexible field of operation for a possible user. 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.

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 Framworks, for example OpenGL or the Cairo graphics library. An 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 use cases, 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.

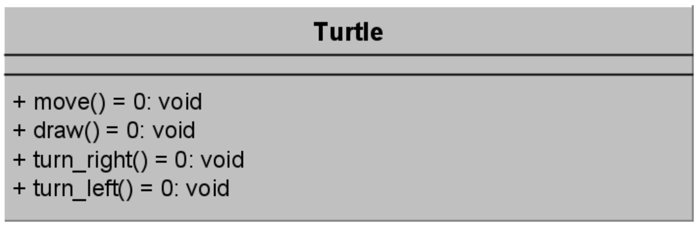


Figure 10: Minimal turtle

More specialized versions can offer more functions, depending on other needs and completly independent of the paper usage of a turtle. For example an OpenGL implementation can offer more commands to represent a three dimensional turtle, like rotate around an axis or other affine transformations. 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

In order to interpret the result of a L-system, it is nessecary to map a command to a symbol of a generation. 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 L-system alphabet and the commands. As discussed, the recursive function receives an output iterator, which handles the generated data.

For this paper an example output iterator is introduced, the CommandMappingIterator. This iterator will handle the data interpretation and calls turtle commands. In figure 11 is the concept of this iterator displayed. It handles a L-system based on strings and chars. It receives data(chars), interprets it according to the mapping and calls the specified command.

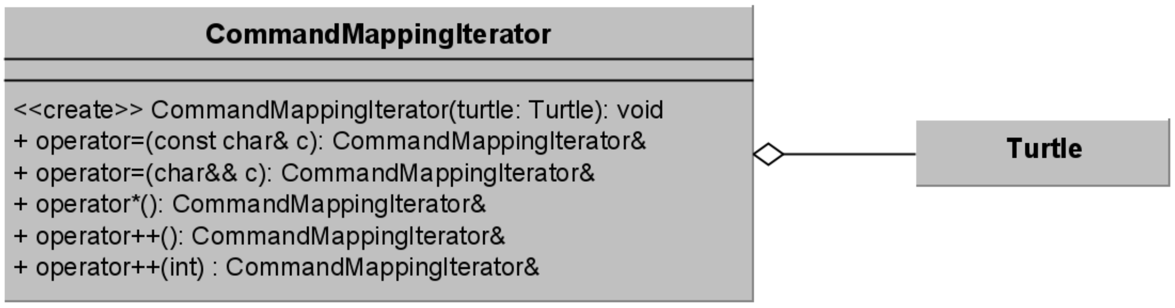

CommandMappingIterator
<<create>> CommandMappingIterator(turtle: Turtle): void
+ operator=(const char& c): CommandMappingIterator&
+ operator=(char&& c): CommandMappingIterator&
+ operator*(): CommandMappingIterator&
+ operator++(): CommandMappingIterator&
+ operator++(int) : CommandMappingIterator&
Turtle

Figure 11: C++ class for the command mapping iterator

The recursive function will receive this iterator as parameter and just hands the chars of the result over. 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 data handling. In order to offer different mappings, new implementations of an output iterator are nessecary. In section4 is the implementation for this example provided, based on figure 11 and the grammar introduced in previous sections.

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

3.5 Overview

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

At the beginning the L-system will be initalized. The L-system only holds the grammar and provides access to it. The recursive function will be used to calculate a L-system generation with the accessed data. 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 standard components.

The CommandMappingIterator is an output iterator, which allows to use the provided turtle interfaces to call the correct turtle command. Whenever data from the recursive function is handed over to the iterator, it interprets it and calls a turtle command. It 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 provided as interface. This interface enforces the absolut minimal set of functions a turtle has to offer. Each spezialisation of a turtle has to handle their configuration and additional dependencies on their own.

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

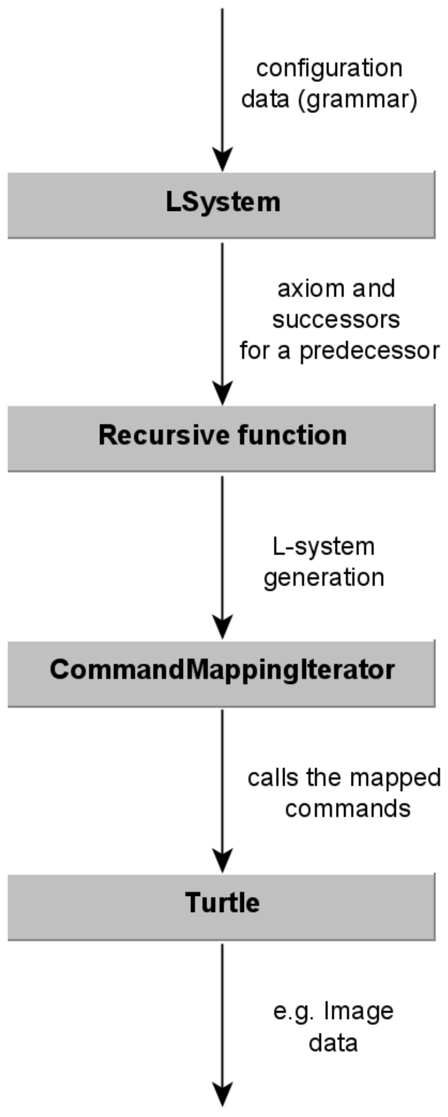

configuration data (grammar)
LSystem
axiom and successors for a predecessor
Recursive function
L-system generation
CommandMappingIterator
calls the mapped commands
Turtle
e.g. Image data

Figure 12: Simplified overview over the data flow

4 Implementation

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

4.1 Build System

The question what build system to choose or if a build system is even needed, is one of the first steps to define for an implementation. A build system provides a fast way to build a complexer architecture and even extend it with libraries. A big role for the question what build system to choose, is the choice of operating system, because not all build systems work on all platforms.

A typical build system for C++ projects are Makefiles, which include a file with what to build and how to build it. 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 of their choice.

The project setup is straightforward and consists mainly of the C++ version and the files to build the project. 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 in comparison to Unix systems. After some research, a sample from J. Preshing [6] provided a working setup for CMake and he even provided some precompiled library files [7], called dll on Windows. The setup can be found in the provided code and should work on different operating systems.[3]

The CMake file includes also a definition for a test executable, further descriebed in section 6.

4.2 LSystem

The different variants of L-system implementations are discussed in previous sections. 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 for a reason called Predecessor and Successor, like the members of a production.

```
template <typename Predecessor, typename Successor>
class LSystem {
    void set_axiom(Successor axiom) {
        ...
    }

    std::template shared_ptr<Successor> get_axiom() {
        ...
```

[3]The build process is only tested on a windows machine with some additional setup for convenience, but should also work on Unix systems

```
    }

    void add_production(Predecessor predecessor, Successor successor) {
        ...
    }

    std::template shared_ptr<Successor> get_successor(const Predecessor& predecessor) {
        ...
    }

private:
    std::shared_ptr<Successor> axiom_;
    std::unordered_map<Predecessor, std::shared_ptr<Successor>> productions_;
}
```

There are some general restrictions for the templates, 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 objects of the same type. 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. These parts are the predecessor value of a production and are needed for the rewriting. This might sound complex, but it should be straightforward with the concrete example of a string as Successor and char as Predecessor. A string can be splitted into smaller strings, fullfilling the first requriement. The string can be splitted into chars and therfore fullfills also the second requirment.

This L-system implementation holds the productions of a grammar in a map. A production is also templated, but is only a simple data holder without special functionality. The productions are saved in a map to gain fast access, because the recursive function will call get_successor() often. There will be a call for each element, each Predecessor object, of a L-system generation. Because of that the amount of calls will increase enormously with every generation.

4.3 Recursive function

The purpose of the recursive function, is to calculation of a L-system generation. In order to fullfill its purpose the function provides multiple template parameters.

```
template<template <typename, typename> class LSystem, typename Predecessor, typename
    Successor, typename OutputIterator>
void calculate_l_system_generation(LSystem<Predecessor, Successor>& l_system, unsigned int
    generation, OutputIterator& output_iterator, std::shared_ptr<Successor> current_value =
    nullptr) {
    if (current_value == nullptr) {
        // inital value
        current_value = l_system.get_axiom();
    }

    for (auto&& part : *(current_value)) {
        if (generation > 0) {
            // rewrite needed - get production result
            auto successor = l_system.get_successor(part);
12
13              if (successor == nullptr) {
14                  // no production found, it is therefore a terminal and can be handed over directly
15                  *output_iterator = part;
16                  ++output_iterator;
17              }
18              else {
19                  calculate_l_system_generation<LSystem, Predecessor, Successor, OutputIterator
                        >(l_system, generation − 1, output_iterator, successor);
20              }
21          }
22          else {
23              // max depth of recursion reached
24              *output_iterator = part;
25              ++output_iterator;
26          }
27      }
28  }
```

As shown in the listing there are several parameters needed for the calculation of a L-system. The first parameter is the L-system itself, which can be exchanged with any L-system implementation, as long as it provides general functionality. The L-system must provide at least two functions:

- std::shared_ptr<Successor> get_axiom()
- std::shared_ptr<Successor> get_successor(const Predecessor& predecessor)

These functions will be used in the recursion to get calculate the next generation. In order to guarantee the same type for all components, both functions are using a template. The Predecessor and Successor type are also used to specify the L-system itself. These types have some general restriction as described in section 4.2.
Additional to requriements from the L-system, there are requirments from the recursive function itself. The Successor type has to provide more functionality, because of the use in a ranged based for loop. For example, there has to be access to the begin and end, to iterate over the data. The part, compare line 8 in the listing, has to be a Predecessor object, because of the use with the get_successor function of the L-system.

The second parameter is the generation, which will be used to hand over the depth of the recursion. The value will be decremented and used for the next recursion step.

The OutputIterator type is self explanatory and has to fullfill all the requirments of an arbitrary output iterator. The iterator handles the L-system result in the each way it needs, for example to store it or to call a turtle function.

The last parameter is the current_value which will be used to hand over the results of previous recursion steps. On default it will be initalized with a nullptr and therefore start the calculation with the axiom of the L-system grammmar. Other use cases are also fullfilled, because it is possible to start a recursivce calculation on a arbitrary generation. The current_value could be an already stored generation of a L-system. In order to calculate more generations and get a more detailed result, this generation can be be handed over to the function as current_value . Therefore, it is possible to cache a generation and use it later again.

4.4 CommandMappingIterator

The CommandMappingItertator is an example implementation for an output itertator, which interprets the result of a L-system and calls the mapped commands.

```

class CommandMappingIterator {
public:

    explicit CommandMappingIterator(Turtle& turtle) noexcept : turtle_(std::addressof(turtle))
        {}

    CommandMappingIterator& operator=(const char& c) {
        handle(c);
        return *this;
    }

    CommandMappingIterator& operator=(char&& c) {
        handle(c);
        return *this;
    }

    CommandMappingIterator& operator*() noexcept { ... }
    CommandMappingIterator& operator++() noexcept { ...}
    CommandMappingIterator& operator++(int) noexcept { ... }

private:
    void handle(char c) {
        switch (c)
        {
        case 'F':
            turtle_->draw();
            break;
        case '-':
            turtle_->turn_left();
            break;
        case 'f':
            turtle_->move();
            break;
        case '+':
            turtle_->turn_right();
            break;
        default:
            // do nothing
            break;
        }
    }

    // only save pointer to allow different turtle implementations
    Turtle* turtle_;
};
```

This iterator receives a turtle implementation in the constructor and uses it later to call the mapped commands. This implementation is based on the simple grammar from previous sections and allows only the use of chars. The chars will be assigned and interpreted, in the handle function, as turtle commands. The use of diffferent turtle implementations is possible, in order to still provide the flexibility given by the polymorphism.

4.5 TurtleGraphic

The base for turtle implementations is an interface, which provides only a minimal set of functions. With these functions it is able to draw for example a fractal, but doesn't restrict an underlying implementation.

```
class Turtle {
public:
    virtual ~Turtle() {};
    virtual void move() = 0;
    virtual void draw() = 0;
    virtual void turn_right() = 0;
    virtual void turn_left() = 0;
};
```

Based on this interface, there will be two implementations provided in this paper.

4.5.1 TestTurtle

The first implementation is a turtle called TestTurtle. This turtle is quiet simple and can be used to print the called functions to the console and test the made calls. It just implements the minimal set of functions from the interface and outputs the called command. For example, such a command is implemented like this:

```
...
void draw() override {
    std::cout << "[Draw-Call]: draw" << std::endl;
}
...
```

There are no further requirments, like a configuration, implemented and only the minimal functionality.

4.5.2 CairoTurtle

In contrast to the TestTurtle, the CairoTurtle has a more complex design with additonal needed functionality. This turtle is based on the graphics library Cairo.

"Cairo is a 2D graphics library with support for multiple output devices. Currently supported output targets include the X Window System (via both Xlib and XCB), Quartz, Win32, image buffers, PostScript, PDF, and SVG file output. Experimental backends include OpenGL, BeOS, OS/2, and DirectFB.

Cairo is designed to produce consistent output on all output media[...]."[8, Cf.]
Only a small part of the Cairo library will be used for this paper. The goal is to export a L-system fractal as PNG-file. In order to define the boundries of the implementation some general requirments need to be specified.

The implementation of the turtle interface is self explanatory and enables the drawing of a graphic. In the context of this paper, such a graphic can be influenced by the line length, the line width and the turning angle. The CairoTurtle should therefore provide functions to configure these values.

Furthermore, the turtle should be able to export a graphic as PNG-file. It would be possible to set a default size for a export file, but because of the dynamic generation is this not a good solution. In the dynamic generation can the size of the fractal vary and influence the size. There are severeal solutions for this, for example to cut off parts of the graphic to fit a certain file size. Cutting off data outside of the specified size looses data and reduces the quality of the result image. To avoid this, the size of the file is determined using a bounding box, in which case the file is dynamically larger or smaller.

The Cairo library offers a quiet complex drawing model, but only some general concepts are needed for this paper. The first concept or component is a surface, which is the object to drawn on. For example a surface migth be tied to an image format like PNG. The process of drawing will be done with a Cairo context. The context keeps track of the rendering state, for example about the color. The Cairo library provides a special surface, called recording surface. This surface records the draw calls done by the context, without binding it to a output target like an image. The recorded draw calls can later be applied to another surface, in our case to a image surface, which can be exported as PNG.

The Cairo context offers different functions to draw and create a path. A path is not a drawn object, but more like a blueprint for a line, which will be drawn with the stroke call. The context offers a line_to and a move_to function, to draw a line to or move to coordinates. There is no turn function to change the current facing direction about an angle. Because of that, the implementation has to hold the state of the turtle, to be able to calculate the next coordinates depending on the current facing direction. For this purpose an additonal class is introduced, the State. The current state of the turtle will be updated for each draw call and saves the current facing direction and positon. Whenever a draw call is made, the next sate will be calculated and the path extended. This results in the class illustrated in figure 13.
In this figure the the context and the surface from Cairo aren't mentioned yet. These objects will be stored as raw pointers, because they will always be used as such. Another reason to this, is that the creation and destruction is handled by the Cairo library and therefore smartpointers wouldn't add a real value.

The Cairo library offers a function to calculate the bounding box of a recording surface. The function should return the values of an rectangle, including the made draw calls on this surface, but the function didn't return the correct bouding box. In order to get the correct bouding box, an addtional class is introduced - the bounding box. The turtle will hold this class and update it on each draw call. The bounding box will be used in the saving process to return the size and the translation values to center the graphic for the output file.
At this state the turtle allows to draw multiple objects, but there is no possibility to delete previous draw calls. If a second object is drawn with the same turtle instance, both objects will be included in the final image. To prevent this, the reset function is offered, which reinitializes the turtle.

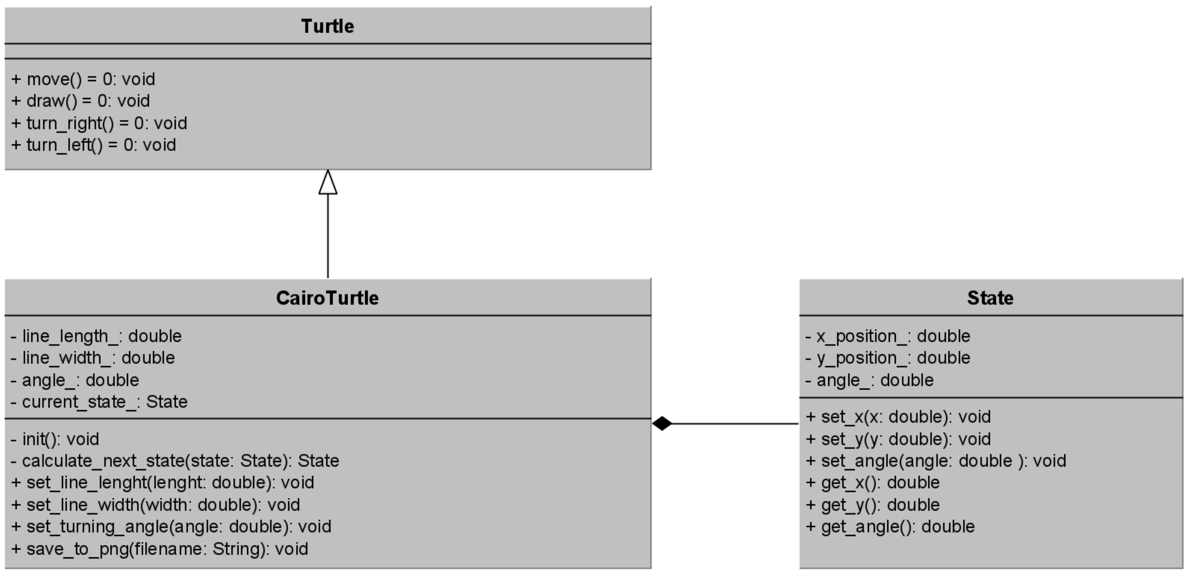


Figure 13: Cairo turtle concept

The complete implementation of the turtle can be found in the provided source code and at the github repository.[4]

4.5.3 Further implementations

The turtle interface allows further implementations, for example to export a SVG-file. This could be also done with the Cairo turtle, by including another export function, but a more lightweight solution would be also possible. For example by directly generating the SVG-tags.

Another possible enhancement would be the use of OpenGL and draw in a 3D environment. For this the turtle has to provide further functions, like the rotation around axis. For example by providing the needed afffine transformation matrices.

Overall the turtle interface should provide enough flexibility to implement an arbitrary turtle with any graphic framework.

5 Example usage

The sample code provides a simple example for the already shown results in section 2.5. The example uses the introduced architecture, consiting of the CairoTurtle, the CommandMappingIterator, the L-system and the recursive function. The example can be build with CMake, resulting in an exectuable generating both images.

6 Testing

There are test cases for the introduced components provided. The test cases provide a C0-coverage and are placed in an additional executable. The additional executable is also defined with CMake and provides an independent usage of the tests and the example usage.

[4]See https://github.com/frozzenshooter/LSystems

7 Future enhancements

The implementation of the different components has the potential of further extensions. Obviously the turtle allows new implementations, for example based on other frameworks. But also the already provided implementation of the CairoTurtle can be improved. For example, it would be possible to add a new function to alter the drawing colors. Another possible enhancment is the scaling of a fractal in the Cairo turtle. At the moment the size of the picture will be determined with the bouding box, but this has the disadvantage, that it can grow to an potential endless size. In order to prevent this it would be possible to use further affine transformations on the data and scale the image down to a maximal size.
Another part of the architecture is the L-system itself. The L-system implementation is just a dumb data holder and therefore, there is potential to provide further functionality. At the moment the L-system validates that there is only one production for a predecessor. The L-system doesn't check if with the given productions a generation is even possible. It would be possible to provide a function, which checks if the provided productions enable the rewriting of the L-system.
A futher enhancement would be additional test cases. At the moment only a C0 coverage is provided, but further test cases could provide a better testing result.
At the moment the implementation only provides a configuration in the code itself and is more or less static. A way more flexible and convenient way is the configuration with files. This makes it possible to configure a L-system without a code change, only by providing such a file. A dynamic configuration can include not only the L-system grammar but also configuration of a turtle and the recursivec function. Such a complete configuration can be done in a single file, but should be seperated to maintain clear separation. A simple structure of such a file are key-value pairs. This has still the downside of a bad separation of the configuration from the different components.

A solution for this is the use of JSON. This allows the seperation of the configuration for each component. A configuration file contains a JSON-object and allows the dynamic configuration. The JSON-object has several child objects containing the different component configurations. In order to be able to read the file the architecture can provide another component, reading an input stream and loading the configuration. Afterwards, this component provides the configuration datat for other components.
The mentioned extensions are only a small part of the possibilites given by the architecture.

8 Conclusion

The introduced architecture enables a flexible use of all introduced components. Different L-systems, with the formal restrictions, can be used and generated. The Support of an output iterator allow different ways to use a L-system result, like the graphic generation with a turtle. The turtle interface enables different implementations and supports therefore all kinds of frameworks.
Even with this flexible concept, there are remaining problems. The already mentioned bounding box in the CairoTurtle, is only a workaround for the not working function from the Cairo library. The bounding box itself has also issues, for example it doesn't consider the line width. This would for example require an additional dynamic padding.

Another not ideal point is the static configuration in code as mentioned in the further enhancements.

Even with these potential enhancements, provides the concept overall a flexible and easy extensible implementation of a L-system and corresponding components.

References

[1] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. 2004. [Online]. Available: `http://algorithmicbotany.org/papers/abop/abop.pdf` (visited on 07/16/2020).

[2] B. B. Mandelbrot, *The Fractal Geometry of Nature*. 1982. [Online]. Available: `https://www.semanticscholar.org/paper/Fractal-Geometry-of-Nature-Mandelbrot/d9c33f310c8af5dbdbc79d1609d3e1bc45180847` (visited on 08/06/2020).

[3] B. Harvey, *Computer Science Logo Style: Symbolic Computing*. 1997, vol. 1. [Online]. Available: `https://www.mobt3ath.com/uplode/book/book-24624.pdf` (visited on 08/06/2020).

[4] R. Goldman, S. Schaefer, and T. Ju, *Turtle geometry in computer graphics and computer-aided design*. [Online]. Available: `https://www.cse.wustl.edu/~taoju/research/TurtlesforCADRevised.pdf` (visited on 08/06/2020).

[5] R. Dickau, *Two-dimensional l-systems*, 1997. [Online]. Available: `http://mathforum.org/advanced/robertd/lsys2d.html` (visited on 08/05/2020).

[6] J. Preshing, *Cairosample*, 2018. [Online]. Available: `https://github.com/preshing/CairoSample` (visited on 08/06/2020).

[7] ——, *Here's a standalone cairo dll for windows*, 2017. [Online]. Available: `https://preshing.com/20170529/heres-a-standalone-cairo-dll-for-windows/` (visited on 08/06/2020).

[8] Unknown, *Cairo homepage*. [Online]. Available: `https://www.cairographics.org/` (visited on 08/07/2020).