Lindenmayer systems - a flexible C++ implementation

Steffen Knoblauch
August 8, 2020

Lindenmayer Systems, short L-systems, are the result of research from Lindenmayer et al.[1] about the geometric features of plants. 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 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.

1 Introduction

L-systems are a formal way to describe plant or fractal development and interpret the result as a graphic. In order provide a general understanding of Lsystems is this paper 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.[1]

Finaly, 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 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.

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 extension 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 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."[2, Chapter 6, p. 34]The self-similartiy 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 rewritng 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].

For example this concept can be used to rewrite a inital string, called axiom, with defined rewriting rules. A simple example is the following grammar, which consits of only two nonterminals, A and B, and two production rules. The first rule is A→AB, the second rule is B→A. The arrow '→' symbols the replacement, the rewriting, of the object on the left with the object of 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 generation. For all nonterminals the productions will be parallel applied. In this case the nonterminals A and B will be replaced, because both have existing production rules. This results in the second generation with 'ABA' as result.

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

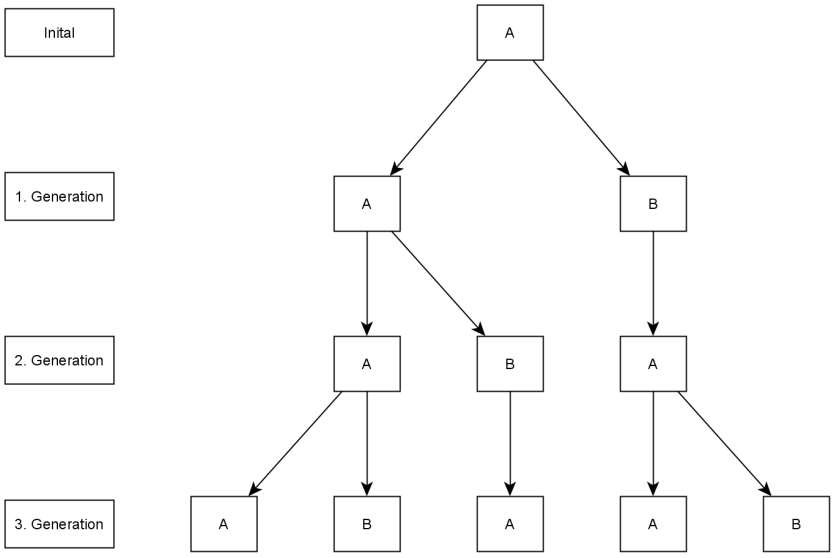

Figure 1: Simple L-system

This process can be successively repeated recursive for a 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.[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 - consiting of terminals and non terminals
ω: axiom - nonempty word of the alphabet, which should contain at least one nonterminal
P: set of productions

A production coniststs of a predecessor, a nonterminal symbol of the alphabet, and a successor, the replacment 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.[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 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 possiblities 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 probaility which results in a randomisation of the generation
- Context senstive grammars: production not only look for the nonterminal, they rather look for the symbols bevor 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 points 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.[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.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 θ is needed to calculate the next state of the turtle. There are these 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 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. [2]

2.5.1 Hilbert curve

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

- L→+RF-LFL-FR+
- R→-LF+RFR+FL-

The turtle will be initalized with an angle of 90° 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→YF+XF+Y
- Y→XF-YF-X

The turtle will be initalized with an angle of 60° and length d of 3 for a 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)

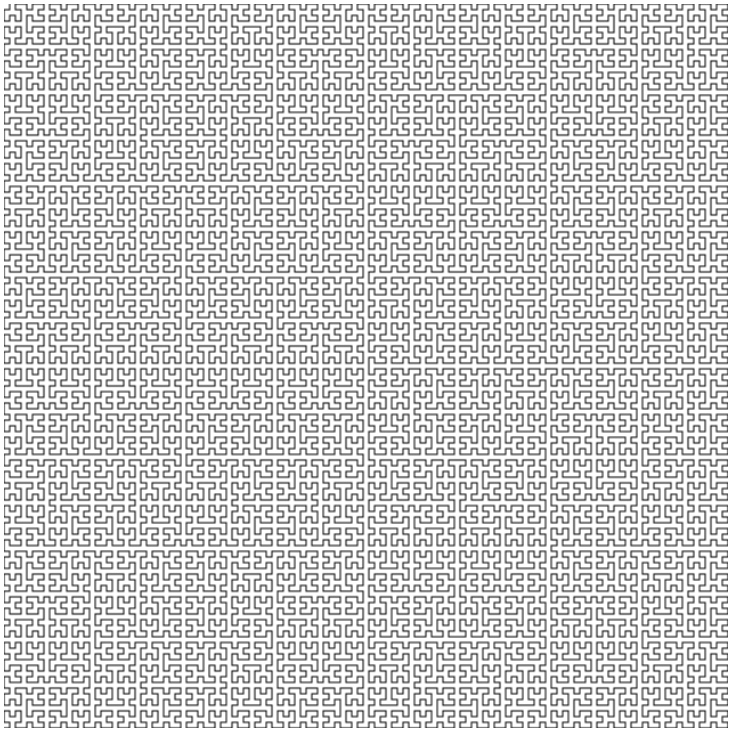

Figure 2: Hilbert L-system - 7 generations

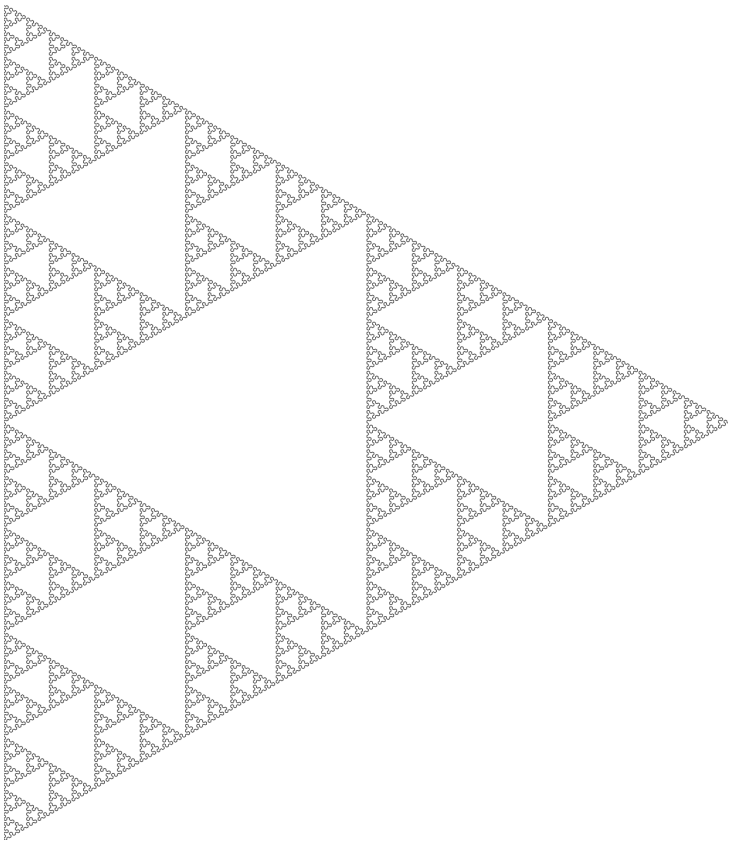

Figure 3: Sierpinski L-system - 9 generations

3 Architecture

The primary goal of this paper is 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 itfself, 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 dissussion 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 addtional 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 seperation between datastructure and processing of the data. It is not possible to simply exchange the datastructure or the function without the seperation, which affects flexibility of the L-system negative. 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 datastrutcture 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.

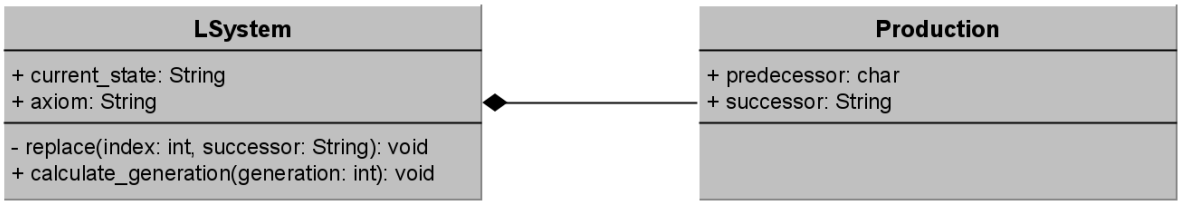

Figure 4: Naive L-system

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 →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-

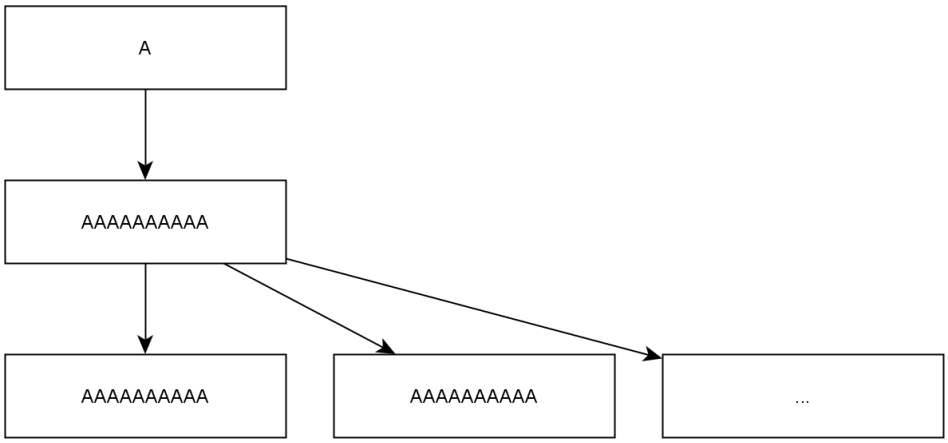

Figure 5: Memory reduction

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 →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 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

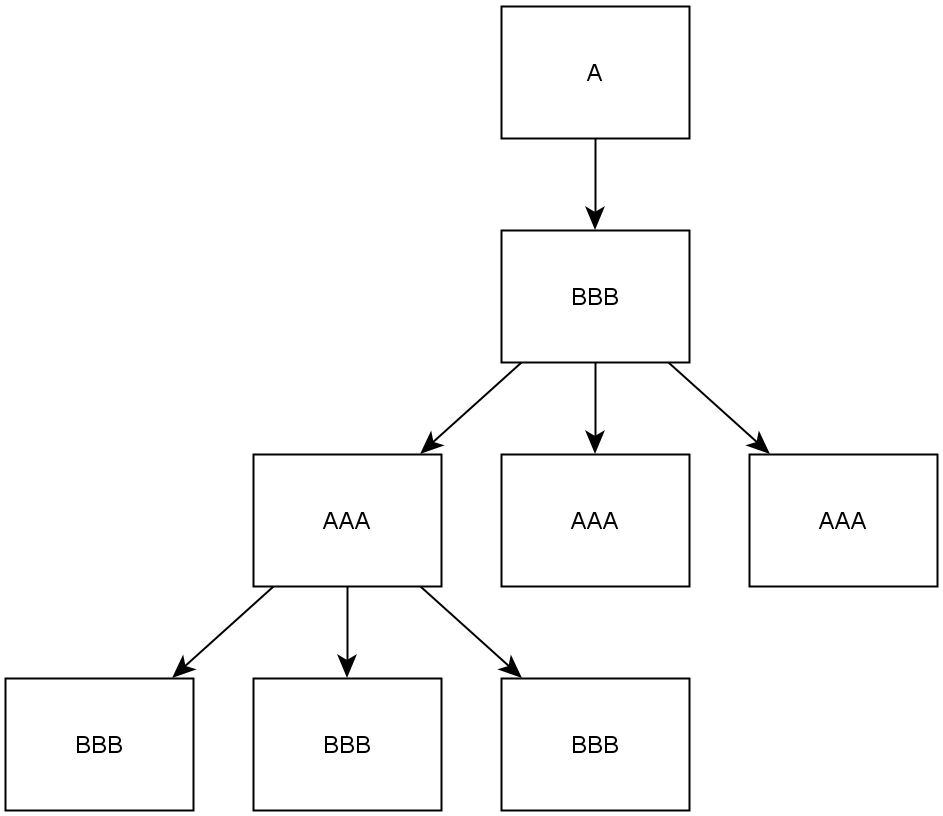

Figure 6: L-system as graph

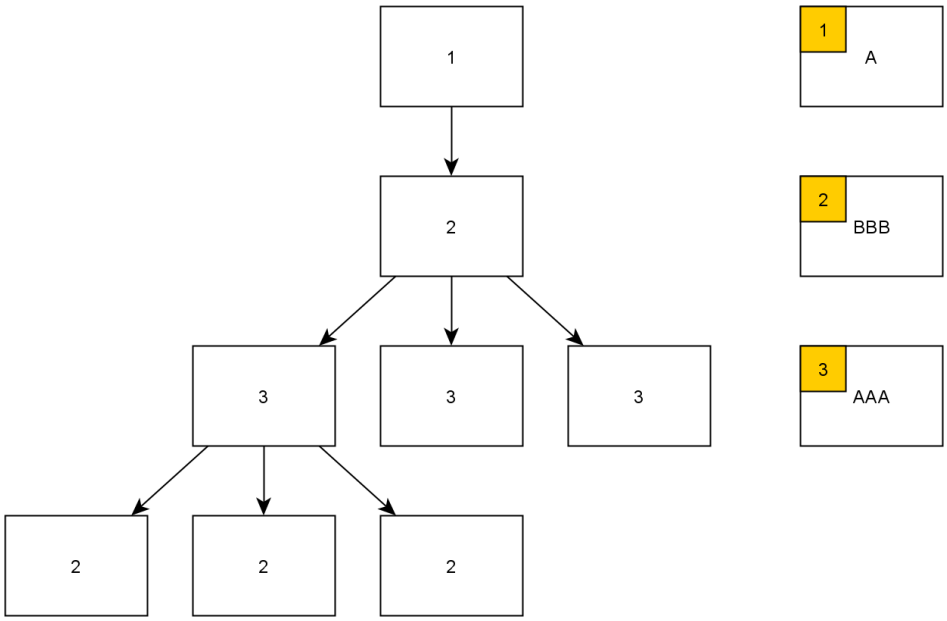

Figure 7: L-system as graph with improved memory consumption

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 replacment 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 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 data is not needed.

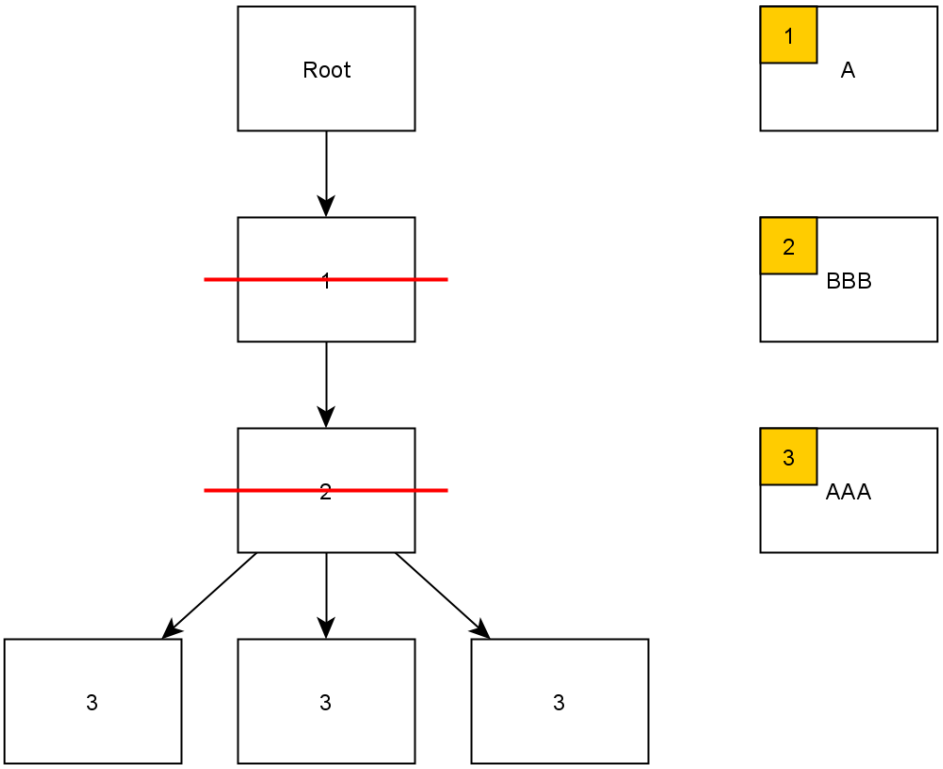

Figure 8: L-system as graph with one generation

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 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 setp forwards and backwards in the result, a solution could be to store the result in another object. This calculations also guarantees a flexiblity, 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.

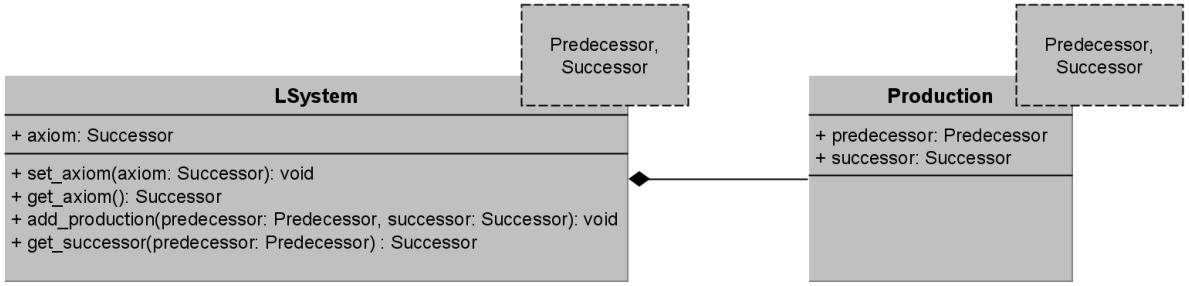

Figure 9: L-system proposal

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.Additonal, 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 requriements. 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 asume 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 than 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, consiting of the L-system datastructure and the recursive funtion, 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 Framworks, 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.

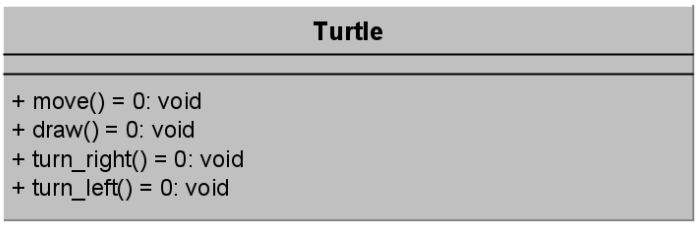

Figure 10: Minimal turtle

More specialiced versions can offer more functions, depending on other needs and completly undependent 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.

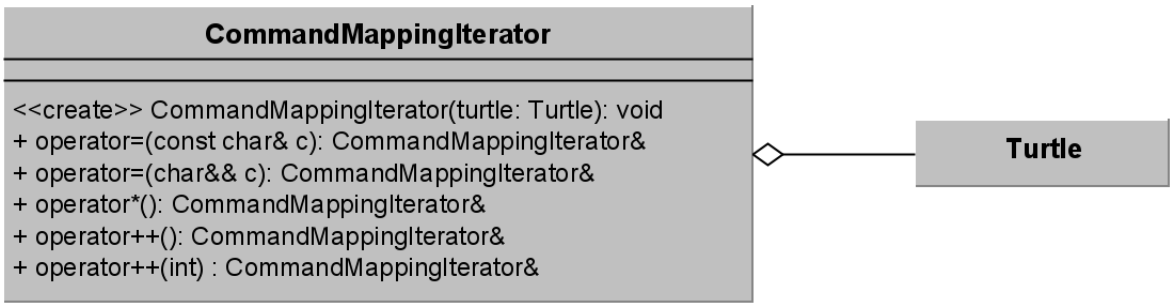

Figure 11: C++ interface command mappin iterator

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.

3.5 Overview

This section will now give a short overview of the disscuessed 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 functionlity. Under the premises, that exchanged components still offer the same functionality and data. In figure 12 a simplified overview of the dataflow is illustrated.

At the beginning the L-system will be initalized. The L-system itself needs only to hold the grammar and 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 form 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 mappping, 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 absolut minimal set of functions a turtle has to fullfill. Each spezialisation of a turtle has to independent handle their configuration and additonal 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 addtional there are comments in the provided code.

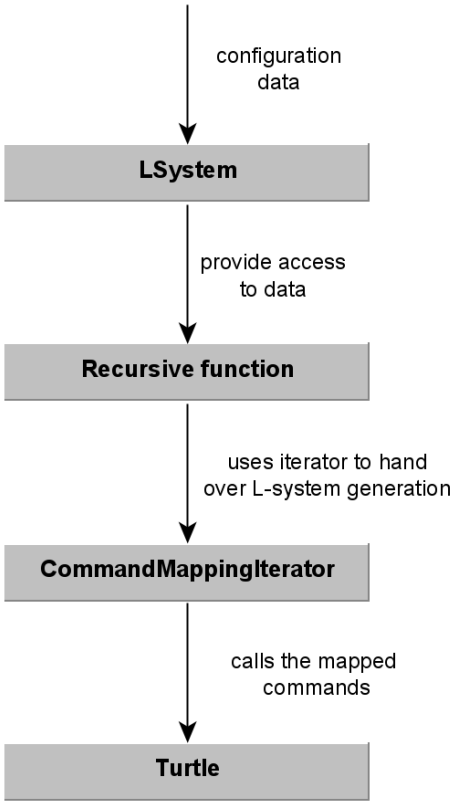

Figure 12: Simplified overview over the data flow

4 Implementation

This section provides the in detail 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 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, which also plays a big role for the question what build system to choose, is the choice of operating system.

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 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]

4.2 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 for a reason called Predecessor and Successor, like the members of a production.

```
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     }
```

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

```
10
11      void add_production(const Predecessor& predeccessor, const Successor& successor) {
12          ...
13      }
14
15      std::template shared_ptr<Successor> get_successor(const Predecessor& predecessor) {
16          ...
17      }
18
19  private:
20      std::shared_ptr<Successor> axiom_;
21      std::unordered_map<Predecessor, std::shared_ptr<Successor>> productions_;
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 are 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, fullfilling the first requriement. The string can be splitted into chars and therfore 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 th gain fast access, because the recursive function will call get_successor() often. This will happen 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 recursive function purpose is to calculate a generation of a L-system. In order to fullfill its purpose the function provides multiple template parameters.

```
1   template<template <typename, typename> class LSystem, typename Predecessor, typename
        Successor, typename OutputIterator>
2   void calculate_l_system_generation(LSystem<Predecessor, Successor>& l_system, int generation,
        OutputIterator& output_iterator, std::shared_ptr<Successor> current_value = nullptr) {
3       if (current_value == nullptr) {
4           // inital value
5           current_value = l_system.get_axiom();
6       }
7
8       for (auto&& part : *(current_value)) {
9           if (generation > 0) {
10
11              // rewrite needed - get production result
12              auto successor = l_system.get_successor(part);
13
```

```
14                  if (successor == nullptr) {
15                      // no production found, it is therefore a terminal and can be handed over directly
16                      *output_iterator++ = part;
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              }
26          }
27      }
```

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 have at least these functions:

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

These functions will use the grammar of an arbitrary L-system to calculate the result generation. In order to guarantee the same type for all components, both functions have the template for a Predecessor and a Successor type, which will also be used to define the L-system itself. The Successor and Predecessor type have some general restriction as described in section 4.2.
There are additional requriements for the Successor type, because of the use in a ranged based for loop. For example, there has to be access to the begin and end of the type. The part has also 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. If these requirements are fullfille, it is possible that the iterator can handle the L-system result in each way it needs to be used. 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 a already stored generation of a L-system. This generation can be be handed over to the function in order to calculate more generations and get a more detailed result. As consequence it is possible to cache a generation and use it later again.

4.4 CommandMappingIterator

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

```
1
2  class CommandMappingIterator {
3  public:
4
5      explicit CommandMappingIterator(Turtle& turtle) noexcept : turtle_(std::addressof(turtle))
           {}
6
7      CommandMappingIterator& operator=(const char& c) {
8          handle(c);
9          return *this;
10     }
11
12     CommandMappingIterator& operator=(char&& c) {
13         handle(c);
14         return *this;
15     }
16
17     CommandMappingIterator& operator*() noexcept { ... }
18     CommandMappingIterator& operator++() noexcept { ...}
19     CommandMappingIterator& operator++(int) noexcept { ... }
20
21 private:
22     void handle(char c) {
23         switch (c)
24         {
25         case 'F':
26             turtle_->draw();
27             break;
28         case '-':
29             turtle_->turn_left();
30             break;
31         case 'f':
32             turtle_->move();
33             break;
34         case '+':
35             turtle_->turn_right();
36             break;
37         default:
38             // do nothing
39             break;
40         }
41     }
42
43     // only save pointer to allow different turtle implementations
44     Turtle* turtle_;
45 };
```

This iterator receives the obligatory turtle in the constructor and uses it later on to call the mapped commands. The focus of this implementation is based on the simple grammar from previous sections. Because of this the implementation only allows the use of chars and interprets them as turtle commands. This implementation allows the use of diffferent turtle implemenntations, in order to still provide flexibility.

4.5 TurtleGraphic

The central interface of the turtle is an abstract class, which provides only a minimal set of functions and is for example able to draw a fractal.

```
1 class Turtle {
2 public:
3     virtual ~Turtle() {};
4     virtual void move() = 0;
5     virtual void draw() = 0;
6     virtual void turn_right() = 0;
7     virtual void turn_left() = 0;
8 };
```

There will be two implementations of this interface 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 showcase 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:

```
1 ...
2 void draw() override {
3     std::cout << "[Draw-Call]: draw" << std::endl;
4 }
5 ...
```

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.]
For this paper only a small part of the Cairo library will be used, with the goal 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 and the turtle 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 the size of the fractal can vary and would influence the size. There are severeal solutions for this, for example to cut off parts of the graphic to fit a certain file size, which would also loose data. To avoid this, the size of the file is determined using a bounding box, in which case the file is dynamically larger or smaller.

Cairo offers a quiet complex drawing model, but only some general concepts are needed for this paper. The first concept is a surface, which is the object which will be drawn on. For example a surface migth be tied to a image format like png. The process of drawing will be done with a Cairo context, which keeps track of the rendering state. There is a special surface, called recording surface, this surface records the draw calls done by the context. 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 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, which only draw to coordinates and not only a step in the facing direction. The context also doesn't offer a function to turn and it is neseccary. In order to offer the turtle function ir is neseccary to hold the state of the turtle in the turtle itself. 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.

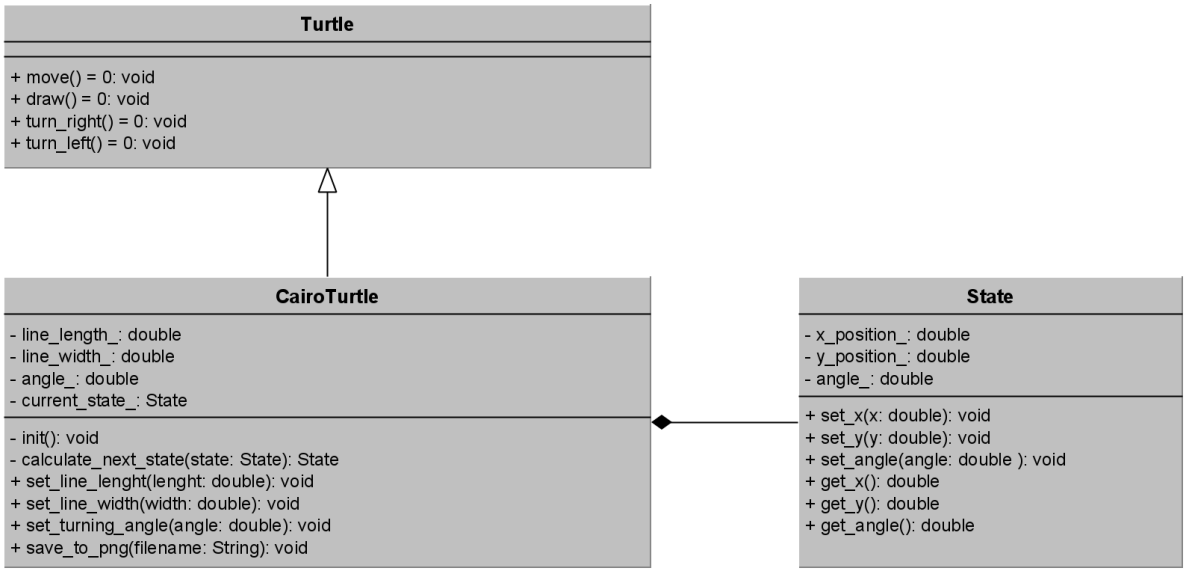

Figure 13: Cairo turtle concept

In this figure the the conetxt 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. Addtional is the creation and destruction handled by Cairo functions and therefore smartpointers wouldn't add an

advantage.

Cairo 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. When the save function is called the bounding box will return the size and the translation values to center the graphic for the output file.
At the moment 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.

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

4.5.3 Further implementations

The abstract turtle class 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 components. 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 transformation matrices. Overall the turtle interface should provide enough flexibility to implement an arbitrary trutle with any graphic framework.

5 Tests

6 Example usage

The provided sample code provides a simple example for the already shown examples in section 2.5. The example will use the introduced architecture, consiting of the cairo turtle, the command mapping iterator and the L-system with the recursive function.

7 Future enhancements

7.1 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
```

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

rule Y XF-YF-X

In this example the key is seperated 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 precessor 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.

With this enhancement, a dynamic way of using the architecture would be possible.

8 Conclusion

The introduced architcture enables a flexible use of all introduced components. With the formal restrictions different L-systems can be used and calculated. The iterator supports different ways of use for the result of a L-system, like the graphic generation with a turtle. The turtle 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 Cairo turtle is only a workaround for the not working function from the Cairo library. This bounding box also doesn't consider the width of the lines and because of that should include a dynamic padding. Another not ideal point is the static configuration of a L-system in teh code as mentioned in the further enhancements.
Overall the concept provides a flexible and easy extensible implementaion of L-systems.
[5]

[5] UML DIAGRAMME NOCHMALS ÜBERARBEITEN - nicht immer wirklcihes uml sondern tw zu sehr abhängig von der Implementierung

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).