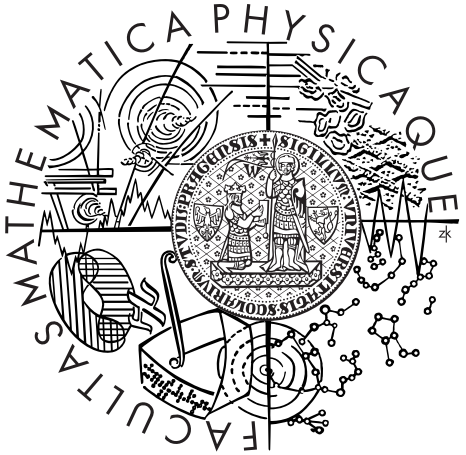
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Marek Fišer

L-systems online

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. Josef Pelikán
Study program: Computer Science
Specialization: Programming

Prague 2012

Typesetting by XeLaTeX

I would like to express my thanks to my supervisor RNDr. Josef Pelikán for guiding me through this Bachelor thesis, to Milan Straka for his great help with F# stuff (especially the lexer and parser) and to my friends and colleagues who have given me valuable advice and feedback. Last but not least, I would like to thank my family, especially my father, for their support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: L-systémy online

Autor: Marek Fišer

E-maiolvá adresa autora: malsys@marekfiser.cz

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán

E-maiolvá adresa vedoucího: pepca@cgg.mff.cuni.cz

Klíčová slova: Lindenmayerovy systémy, L-systémy, modelování rostlin, fraktál, systém komponent, SVG, WebGL

Abstrakt

L-systém je v nejjednodušší podobě varianta bezkontextové gramatiky. Byl vyvinut a používá se hlavně pro modelování růstu rostlin, ale s jeho pomocí se také dají vytvářet obecné fraktály, modely měst nebo dokonce hudba. Pokud někoho L-systémy zaujmou a chce s nimi experimentovat, je těžké najít aplikaci, která by mu to umožňovala. Cílem této práce bylo vytvořit online systém pro práci a experimentování s L-systémy pro široké spektrum uživatelů. Výsledné řešení se skládá ze dvou částí.

První část je univerzální, snadno rozšiřitelná knihovna pro zpracování L-systémů. Svou rozšiřitelnost dosahuje modularitou, vstup zpracovává prostřednictvím systému propojených komponent, které jsou specializované na konkrétní činnost. To také přispívá k přehlednosti a spolehlivosti celku. Navíc je knihovna zcela nezávislá a multiplatformní, lze ji tedy použít i v jiných aplikacích.

Druhá část je moderní webové rozhraní, které bylo navrženo tak, aby bylo srozumitelné pro nováčky a zároveň aby nabízelo pokročilé funkce pro náročnější uživatele. Součástí webu je i galerie L-systémů, do které může každý uživatel přispívat a tvořit tak komunitu. Webové rozhraní plně využívá schopnosti navržené knihovny a slouží tak i jako ukázka jejího použití.

Title: L-systems online

Author: Marek Fišer

Author's e-mail address: malsys@marekfiser.cz

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán

Supervisor's e-mail address: pepca@cgg.mff.cuni.cz

Keywords: Lindenmayer systems, L-systems, plant modelling, fractal, component system, SVG, WebGL

Abstract

An L-system in its simplest form is a variant of a context-free grammar. Originally, L-systems were developed and are still mainly used for modeling plant growth, though with L-systems it is possible to create general fractals, models of towns or even music. However, anyone interested in L-systems and wanting to experiment with them may have difficulty finding an appropriate application. The goal of this work was to create an online system, suitable for a wide range of users, for working and experimenting with L-systems. The resulting solution consists of two parts.

The first part is a universal, easily-expandable library for processing L-systems. Expandability is achieved thanks to its modularity. All input is processed through interconnected components that are specialized in particular activities. The specialization of the components also contributes to the clarity and reliability of the whole processing system. The library is independent and multiplatform and can thus be readily used in other applications.

The second part consists of a modern web interface designed to be understandable for beginners and yet also capable of providing advanced features for more advanced users. Part of the site is a gallery of L-systems to which each user can contribute and which thus helps to create a user-community. The web interface takes full advantage of the library and thus serves as an example of its use.

Contents

Introduction **5**

1 L-systems **9**
1.1 Formal definition of L-system 9
1.1.1 Rewriting principles of an L-system 9
1.1.2 Interpretation of L-system symbols 10
1.2 L-system types 12
1.2.1 Deterministic L-systems 12
1.2.2 Bracketed L-systems 12
1.2.3 Stochastic L-systems 14
1.2.4 Context-sensitive L-systems 15
1.2.5 Parametric L-systems 17
1.3 Related L-system generators 19
1.3.1 Web based generators 19
1.3.2 Desktop applications 20

2 Design **23**
2.1 Choice of development environment 23
2.2 L-system processing library 24
2.2.1 Input form 24
2.2.2 Input syntax 25
2.2.3 Source code compilation 28
2.2.4 Input processing 30
2.2.5 Components 30
2.2.6 Measuring pass 32
2.2.7 Utilities 34
2.3 Processing system 34
2.3.1 Basic component system 34
2.3.2 Component system extensions 35
2.3.3 Interpretation of a symbol as another L-system 36
2.3.4 Final component system 40
2.4 Web user interface 41
2.4.1 L-system processor 41
2.4.2 Gallery of L-systems 42
2.4.3 Help 42
2.4.4 Administration 42
2.4.5 Database 42

3 Implementation **47**
3.1 Solution structure 47
3.2 Input parsing 48
3.3 Compilation and evaluation 49
3.4 Components members 51
3.4.1 Documentation of members 53
3.4.2 Example 53

3.5 Input processing . . . 54
3.6 Immutable data structures as scoped storage . . . 55
3.7 Implemented components . . . 55
3.7.1 Symbol rewriter . . . 55
3.7.2 Turtle graphics interpreter . . . 58
3.8 Triangulation of 3D polygons . . . 59
3.9 Web user interface . . . 60
3.9.1 Data annotations . . . 62
3.9.2 Easy configurability . . . 63
3.9.3 Inversion of control . . . 64
3.9.4 Removal of literal strings with the T4MVC . . . 65
3.9.5 Generated help pages . . . 66
3.9.6 Caching and compression . . . 67
3.9.7 Error logging . . . 69
3.9.8 Cascading style sheets . . . 70
3.9.9 JavaScript . . . 71

4 Results **73**
4.1 L-system processing library . . . 73
4.1.1 Unit tests . . . 73
4.2 Web user interface . . . 74
4.2.1 Visitors and traffic . . . 74
4.3 Solution statistics . . . 76
4.4 Showcase of L-systems . . . 76

Conclusion **83**

List of Abbreviations **87**

List of Figures **87**

List of Tables **90**

List of Source codes **90**

Appendix A Contents of attached CD **93**

Appendix B About figures **95**

Appendix C User documentation **103**
C.1 How to process L-system . . . 103
C.2 Creation of the Pythagoras tree . . . 103

Appendix D Component implementation and usage **111**
D.1 Component implementation . . . 111
D.1.1 Static filtering . . . 111
D.1.2 Configurable filtering . . . 113
D.1.3 Logging of messages . . . 115
D.1.4 Usage in real process configuration . . . 116
D.2 Component documentation . . . 117

Appendix E Usage of L-system processing library **119**

Appendix F Publish on the server **123**
F.1 Creation of publish package . . . 123
F.1.1 Settings . . . 123
F.1.2 Compilation . . . 123
F.2 Configuration of the server . . . 123
F.2.1 Internet Information Services (IIS) . . . 124
F.2.2 Web platform installer . . . 124
F.2.3 F# . . . 125
F.3 Deploy of the application . . . 125
F.3.1 Creation of new Application pool . . . 125
F.3.2 Creation of new App Pool . . . 125
F.3.3 Copy files . . . 126
F.4 First run . . . 127
F.5 Server migration . . . 127

Appendix G Third-party libraries and services **129**
G.1 F# PowerPack . . . 129
G.2 HTML5 boilerplate . . . 129
G.3 Three.js . . . 129
G.4 jQuery . . . 129
G.5 Modernizr . . . 129
G.6 Code Contracts . . . 130
G.7 Autofac IoC container . . . 130
G.8 MvcContrib . . . 130
G.9 Elmah . . . 131
G.10 LESS css . . . 131
G.10.1 .LESS . . . 131
G.11 Data Annotations Extensions . . . 131
G.12 Yahoo! UI Library . . . 132
G.13 ReCaptcha . . . 132
G.14 Google Analytics . . . 132

Appendix H Input syntax reference **133**
H.1 Regular expressions . . . 133
H.2 Tokens . . . 133
H.2.1 Identifier . . . 133
H.2.2 Number . . . 134
H.2.3 Operator . . . 134
H.3 Input syntax . . . 134
H.3.1 Input . . . 134
H.3.2 Empty statement . . . 135
H.3.3 Constant definition . . . 135
H.3.4 Function definition . . . 135
H.3.5 L-system definition . . . 135
H.3.6 Process configuration definition . . . 137
H.3.7 Process statement . . . 138
H.3.8 Mathematical expression . . . 138

H.3.9 Common rules . . . 138

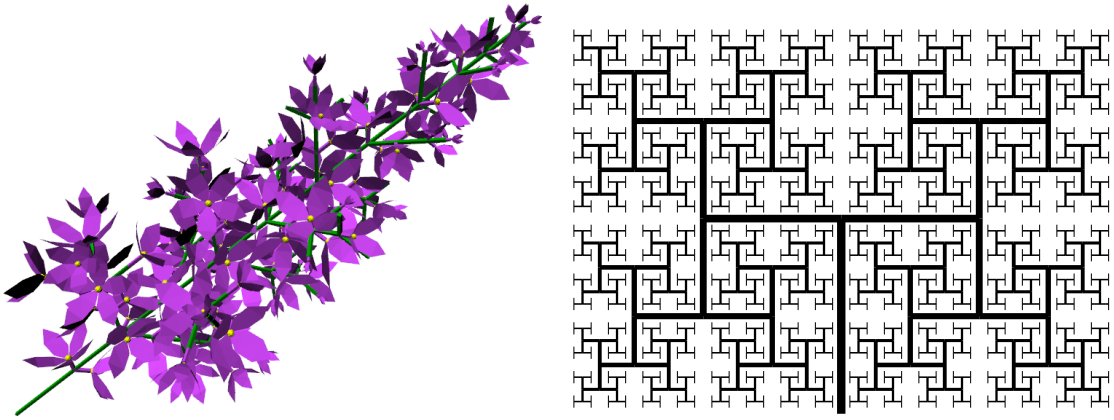
Appendix I Standard library source code **139**
I.1 General Constants . . . 139
I.2 Component specific constants . . . 139
I.2.1 Svg renderer . . . 139
I.2.2 ThreeJs renderer . . . 139
I.3 Abstract L-systems . . . 139
I.3.1 Standard L-system 2D . . . 140
I.3.2 Standard L-system 3D . . . 140
I.3.3 Branches . . . 140
I.3.4 Polygons and branches . . . 141
I.4 Process configurations . . . 141
I.4.1 Symbol printer . . . 141
I.4.2 Svg renderer . . . 141
I.4.3 ThreeJs renderer . . . 142
I.4.4 Hexagonal ASCII renderer . . . 142
I.4.5 Inner L-system process configuration . . . 143
I.4.6 Constant dumper . . . 143

Appendix J Components **145**
J.1 Legend . . . 145
J.2 Components . . . 145
J.2.1 2D SVG renderer . . . 145
J.2.2 3D renderer base . . . 146
J.2.3 3D Three.js renderer . . . 146
J.2.4 Axiom provider . . . 147
J.2.5 Constants dumper . . . 148
J.2.6 Hexagonal ASCII interpreter . . . 148
J.2.7 Inner L-system iterator . . . 149
J.2.8 Inner L-system processor . . . 150
J.2.9 Interpreter caller . . . 151
J.2.10 Memory-buffered iterator . . . 151
J.2.11 Random generator provider . . . 152
J.2.12 Symbol fileter . . . 153
J.2.13 Symbol provider . . . 154
J.2.14 Symbol rewriter . . . 154
J.2.15 Symbols saver . . . 155
J.2.16 Text renderer . . . 155
J.2.17 Turtle interpreter . . . 155

Appendix K Process configurations **159**
K.1 Legend . . . 159
K.1.1 SvgRenderer . . . 159

Introduction

An L-system (also called a Lindenmayer system) is a mathematical formalism that was developed by Aristid Lindenmayer in 1968 for modeling plant growth [Lin68]. An example of a plant modeled by an L-system is shown in Figure 1a. In its simplest form an L-system is a variant of a regular or context-free grammar. By rewriting (deriving) an initial string of symbols (also called an axiom) with some rewrite rules from a grammar, an L-system produces a new string of symbols which can be interpreted in many different ways. In the first L-systems by used Lindenmayer there symbols were to be interpreted as cells of algae. Later, different approach was adopted by Przemyslaw Prusinkiewicz who interpreted the L-system symbols using Logo-like turtle drawing system[1] [Pru85]. With this method he obtained more plant-like structures and fractals [CD93]. In Figure 1b you can see a H-tree fractal created by an L-system.



(a) Model of lilac panicle (b) H-tree fractal

Figure 1: Examples of models created by an L-system

Over time L-systems began to used in many diverse areas. For example they were used to generate rivers in fractal mountains [PH93], streets in virtual cities [PM01] and to describe the subdivision of curves [Pru*03]. L-systems can be used in fields other than computer graphics: for example, in music generation [HCJ99; Man06]. They are still used in plant modeling. Plant models generated with L-systems are used in modern video games or films: for example, they were used to generate many plants and trees for the famous film Avatar [Wor08; Dun10]. [2]

L-systems have a wide variety of interesting applications but it is not easy to find a place to experiment with them. Esentially there are two basic types of L-system generators: web-based and desktop applications. Web-based L-system generators are easily accessible but they are often too primitive to offer much more than the generation of simple fractals (see section 1.3.1). Some of them do not even work in the most-used browsers.

[1] Logo a is computer programming language developed for use in the education of programming for children. Logo controls a cybernetic turtle which does the drawing on a 2D canvas.

[2] Šťava et al. presented a reverse method – the automatic generation of L-systems from a 2D model [Šťa*10].

Desktop applications generally offer more options than web-based ones but most of them are also quite simple and do not offer advanced types of L-systems. There are some complex applications that offer pretty good sets of features but these are expensive, not easy to control, and/or they are old and no longer maintained (see section 1.3.2). A problem with desktop applications is also their compatibility with a user's operating system, its version, and the libraries installed.

The overall goal of this work is to take the best from both of the two main approaches and an create online, feature-rich, development environment for anybody who wants to experiment with L-systems. The development environment will be divided into two parts: a web user interface and an L-system processing library.

The user interface will be web a site that offers great accessibility. Anyone from around the world will be able to use it from any device connected to the Internet , such as: computers, laptops, tablets or smart phones. The interface should be user-friendly to new users and also offer advanced features for experienced users. The primary output format of the web-based L-system processor will be 2D images but it will also be possible to create and display 3D outputs using modern HTML5 WebGL[3] technology directly in the browser. In Figure 2 is a print-screen of a Menger Sponge model displayed by WebGL. Part of the web site will be a gallery of L-systems. Any registered user can add his own L-systems to the gallery along with a description and then others can rate it. This will help to create a community of active users and it can also serve as a learning tool for new users.

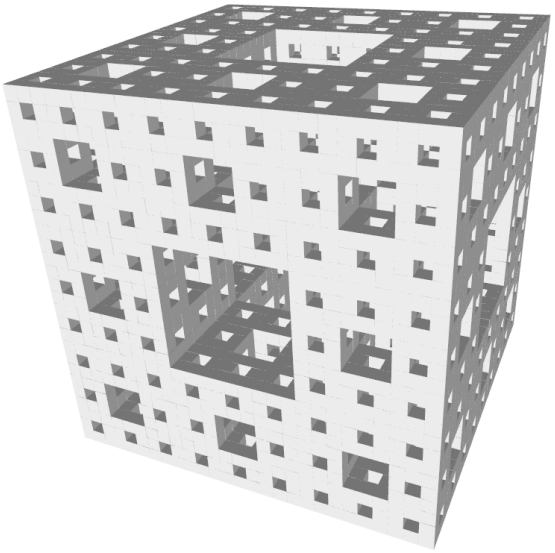


Figure 2: Menger sponge created by an L-system

The second part of the application will be the L-system processing library. Although it will be designed to support the demands of a web interface, it will be independent and should be usable in other applications. During the design of the library, great emphasis will be placed on its ease of extensibility to make it as universal as possible. It should be possible to extend the library by the users themselves.

A new syntax for input will be designed to improve the user experience especially for new users. The syntax should be clean, easy to understand and to remember.

[3]WebGL (Web-based Graphics Library) is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES 2.0, exposed through the HTML5 Canvas element as Document Object Model interfaces. WebGL code executes on a computer display card's GPU (graphics processing unit).

Structure of the thesis

In the first chapter a formal definition of L-systems is given and principles for their rewriting and interpretation are explained. Then follows some descriptions of L-system types and their properties. At the end of the first chapter is a list of some related L-system generators.

The second chapter is devoted to the design of the solution. There is described how L-system processing library and web user interface works.

Implementation details of the project are discussed in the third chapter. Sections in this chapters explains individual problems and their solutions. The text accompanies actual source code snippets and diagrams for better explanation.

The fourth chapter summarizes the results. Part of this chapter is showcase of images of generated L-systems.

All the source codes of L-systems used in this thesis are in a syntax designed as a part of this work. A reference to this syntax can be found in attachment H. It is possible to process all the source code on the web. More information about the figures in this thesis together with additional information and their source codes is given in attachment B.

1. L-systems

A brief history of L-systems has already been mentioned in the introduction. In this chapter are L-systems described more formally. Follows explanation of the rewriting and interpretation principles of L-systems. The main focus of this chapter is to describe various L-system types. At the end of the chapter is a list of related applications.

1.1 Formal definition of L-system

L-system L is formally a triplet $L = (\Sigma, \omega, R)$, where

- Σ is an *alphabet*, a non-empty set of symbols, Σ^* is a set of all the words[1] which can be created from the alphabet Σ, Σ^+ is a set of all non-empty words which can be created from the alphabet Σ,
- $\omega \in \Sigma^+$ is an *axiom* (also called seed), a word defining the initial state of the L-system,
- $R \subset \Sigma \times \Sigma^*$ is a finite set of *rewrite rules* (production rules), a rewrite rule defines rewriting of a symbol $s \in \Sigma$ to a word $w \in \Sigma^*$ is written as $s \to w$.

For any symbol $s \in \Sigma$ which does not appear on the left-hand side of any rewrite rule in R, the identity rewrite rule $s \to s$ is assumed. These symbols are called constants or terminals.

The formal definition of an L-system is similar to a deterministic context-free grammar but there are a few differences. In such a grammar we distinguish between terminal and non-terminal symbols, but in L-systems we do not define them explicitly (rather we define the identity rewrite rule for terminal symbols in L-systems). The next difference is in the initial string. In the grammar there is only one symbol as an initial state but the L-system allows a non-empty word. The biggest difference, however, is in the rewriting principles which are described in the following section.

1.1.1 Rewriting principles of an L-system

Starting with the initial axiom (0th iteration), in each iteration *all* symbols are rewritten with rewrite rules to form next iteration. All symbols can be rewritten because every symbol is on the left side of some rewrite rule. There is only one way to rewrite symbols in an iteration, thus rewriting is deterministic. The result depends only on the axiom.

The rewriting of symbols is parallel (all symbols are rewritten simultaneously during each iteration). This means that when some symbols are rewritten, the resulting symbols are not rewritten again in the same iteration.

The described rewriting principles distinguish an L-system from a formal grammar. In the grammar it is not mandatory to rewrite all possible symbols (the derivation of start state can result in several different derivations). Thus, L-systems are a strict subset of languages.

[1] A word is a sequence of symbols.

The L-system in Source code 1.1 produces strings as shown in Table 1.1. The L-system starts with an axiom `A` and two rewrite rules `A` → `B` and `B` → `A, B`. In the first iteration the axiom `A` is rewritten by the first rewrite rule to `B`. In the second iteration `B` is rewritten with the second rewrite rule to symbols `A, B`. In the third iteration the first symbol `A` is rewritten to `B` and the second symbol `B` rewritten to `A, B` which gives string `B, A, B` and so on.

```
lsystem RewritingExample {
    set symbols axiom = A;
    set iterations = 6;
    set interpretEveryIteration = true;
    rewrite A to B;
    rewrite B to A B;
}
process all with SymbolPrinter;
```

Source code 1.1: A simple L-system as an example of rewriting principles

Iteration	String of symbols
0	A
1	B
2	A B
3	B A B
4	A B B A B
5	B A B A B B A B
6	A B B A B B A B A B B A B

Table 1.1: Result of the L-system in Source code 1.1

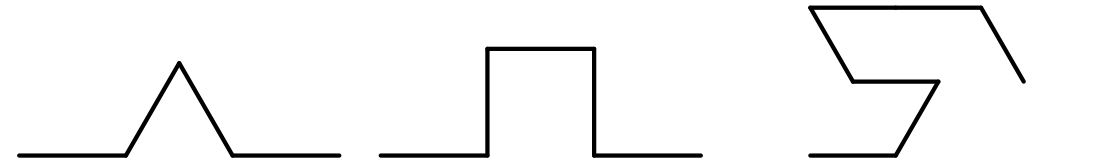
1.1.2 Interpretation of L-system symbols

The result of an L-system rewriting is a string of symbols. As mentioned in the Introduction, we can interpret a string of symbols in many ways: for example, as computer graphics or music.

The simplest and most common interpretation of L-system symbols is to interpret them as 2D graphic elements like lines or polygons. This interpretation is often called *turtle graphics* and it will be used to interpret most of the L-systems in this thesis. This approach can be easily extended into 3D.

Let symbol `F` be interpreted as *draw line forward*, `+` as *turn left* and `-` as turn right. Figure 1.1 shows the strings of symbols interpreted using turtle graphics. The initial direction is to the right.

A slightly more complex string of symbols, as an example of interpretation, is generated by the L-system in Source code 1.2, where symbol `F` is interpreted as *draw line forward*, symbol `+` is interpreted as *turn left* by 85 degrees and symbol `-` as *turn right* by 85 degrees (equally as *turn left* by −85 degrees). The result of interpretation of the first, second and fourth iteration is in shown Figure 1.2.



(a) $F + F - -F + F$, turning angle: 60° (b) $F+F-F-F+F$, turning angle: 90° (c) $F+F++F-F--FF-F$, turning angle: 60°

Figure 1.1: Examples of interpretation of simple string of symbols

```
lsystem InterpretationExample {
    set symbols axiom = F;
    set iterations = 4;
    interpret F as DrawForward(10);
    interpret + as TurnLeft(85);
    interpret - as TurnLeft(-85);
    rewrite F to F + F - - F + F;
}
process all with SvgRenderer;
```

Source code 1.2: Another symbol interpretation example

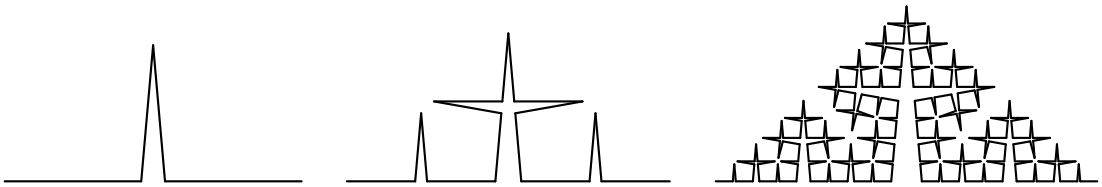


Figure 1.2: The first, second and fourth iteration of the Cesaro curve (Source code 1.2)

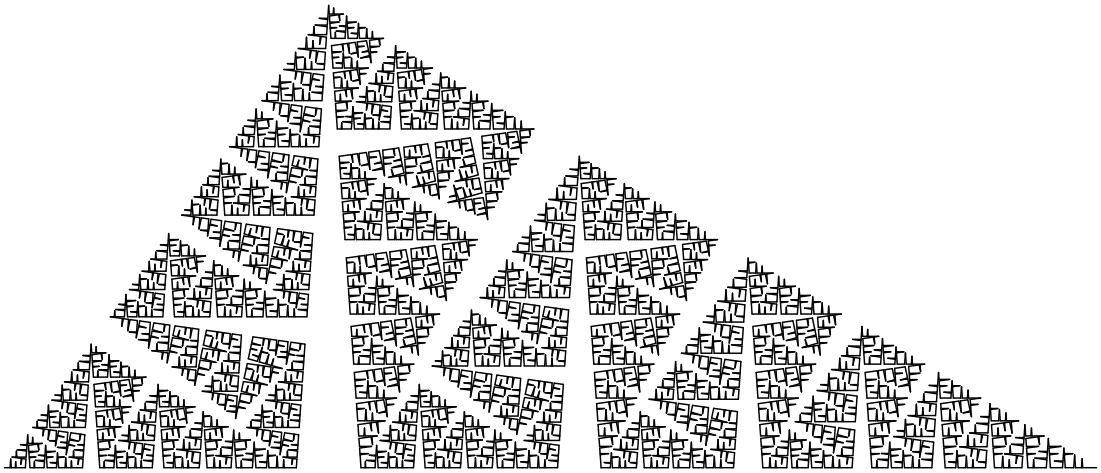


Figure 1.3: Enhanced Cesaro curve from Figure 1.2 [PL91, p. 48]

1.2 L-system types

In this section is described different types of L-systems. Some types may require an extension to the described formal definition of an L-system but this will be omitted here.

The L-systems described so far are called *deterministic L-systems* because their rewriting system is deterministic. *Bracketed L-systems* allow to save and load a state of the interpretation; this can be used to model branches of plants more easily. *Stochastic L-systems* can randomize a result model to suppress its artificiality. *Context-sensitive L-systems* allow to rewrite symbols depending on their context (the neighboring symbols around them). Symbols in *parametric L-systems* can hold any number of arguments that can be used while rewriting or interpreting symbols.

Any of the above-described types can be combined together.

1.2.1 Deterministic L-systems

The basic L-system type described by the previous formal definition is called a D0L-system[2]. *D* means that the rewriting is deterministic and *0* means it is context-free. The result of a D0L-system depends only on the initial string of symbols.

This type of L-system is often used to generate fractal curves. With the D0L-system in Figure 1.4 we can generate the Dragon curve that you can see in Source code 1.3.

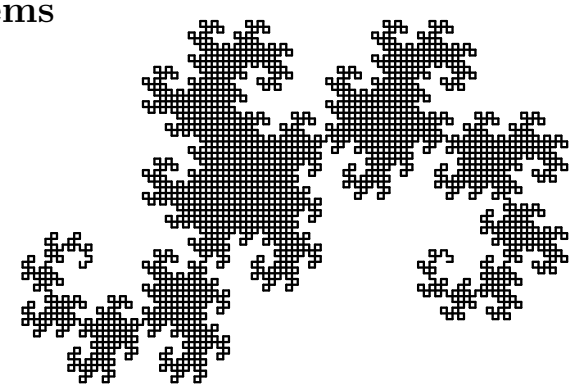


Figure 1.4: Dragon curve

```
lsystem DragonCurve {
    set iterations = 12;
    set symbols axiom = L;
    interpret R L as DrawForward(5);
    interpret + as TurnLeft(90);
    interpret - as TurnLeft(-90);
    rewrite L to L + R +;
    rewrite R to - L - R;
}
process all with SvgRenderer;
```

Source code 1.3: D0L-system for the generation of the Dragon curve (Figure 1.4)

1.2.2 Bracketed L-systems

A bracketed L-systems [PL91, p. 24] extends basic D0L-system with a branching system. Branching is such a fundamental feature that Bracketed L-systems are often just called L-systems.

[2] A D0L-system is also just called a dL-system [Žár*04].

A branching system brings two new commands to the symbol interpretation system: *start branch* and *end branch*. These commands are nearly always represented as bracket symbols (from which bracketed L-systems got their name). An open bracket "`[`" as a start branch and close bracket "`]`" as a close branch.

The start branch command saves the state of interpretation, which can then be loaded by end the branch command later. In turtle graphics, the interpretation state is the position, orientation and drawing color of the turtle. More than one state can be saved at the same time, and the last saved state will be loaded first. This behavior seems natural and could be compared to a pairing of brackets.

Branching extends a linear string of symbols to a tree structure. Individual branches do not affect each other nor their root. This allows plants to be modeled more easily and to create more complex models.

The bracketed L-system in Source code 1.4 demonstrates a use of the branching system to produce a plant-like model as can be seen in Figure 1.5. Note that the color of segments indicates their type and age. Black segments are drawn with the symbol `F` and they represent segments from the previous iteration. Green segments are drawn with the symbol `A` and they are new compared to the previous iteration.

```
lsystem PythagorasTree {
    set symbols axiom = A;
    set initialAngle = 90;
    set iterations = 4;
    interpret A F as DrawForward(16);
    interpret + as TurnLeft(45);
    interpret - as TurnLeft(-45);
    interpret [ as StartBranch;
    interpret ] as EndBranch;
    rewrite A to F [ + A ] [ - A ] F A;
    rewrite F to F F;
}
process all with SvgRenderer;
```

Source code 1.4: A bracketed L-system that which creates a plant-like model (Figure 1.5)

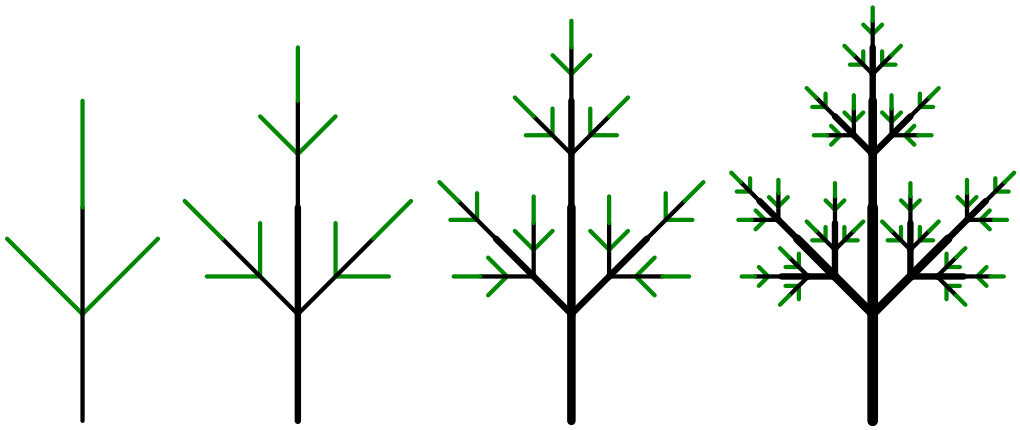


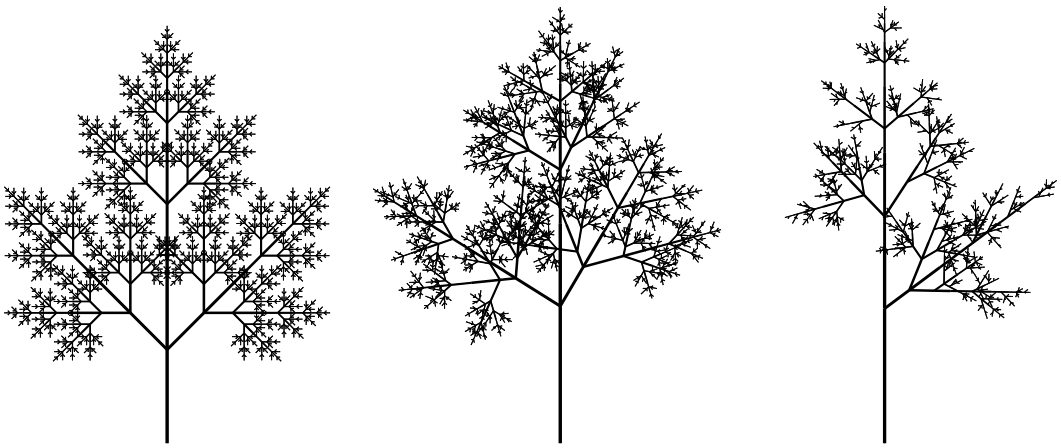
Figure 1.5: The first four iterations of the bracketed L-system in Source code 1.4

1.2.3 Stochastic L-systems

All plant models generated by the same deterministic L-system are identical. However, a forest made by trees which are all identical looks artificial and can not be used in films or video games. Stochastic L-systems solve this problem because they can produce a randomized model. Stochastic L-systems are called 0L-system where 0 means they are context-free.

Randomization of a model produced by stochastic a L-system can be done in two places, in the rewrite rules or in the interpretation of symbols (or in both). Randomization in interpretation can only change the properties of such interpreted symbols as lengths of lines or turning angles, while the topology of the model remains unchanged. This is in contrast to rewrite rule randomization that can also change the topology of a model. Rewrite rule randomization is achieved by defining more replacements for one rewrite rule. The rewriting system will pick a random replacement if the rewrite rule is applied. Each replacement can have a different probability of being picked.

In Figure 1.6, three models of a plant generated by stochastic L-systems are shown. The first image (1.6a) was generated without any randomization. The second image (1.6b) was generated with interpretation randomization of line lengths and angles. For the last image (1.6c) was used the rewrite rule randomization which changed the topology of the model (Source code 1.5).



(a) No randomization (b) Angles, lengths randomized (c) Also topology randomized

Figure 1.6: A comparison between a non-randomized and randomized plant model

```
lsystem StochasticLsystemExample {
    set symbols axiom = X;
    set iterations = 8;
    set initialAngle = 90;
    interpret F(age) as DrawForward(1.8^age*random(0.5,1.5), age/2);
    interpret + as TurnLeft(45 + random(-20, 20));
    interpret - as TurnLeft(-45 + random(-20, 20));
    interpret [ as StartBranch;
    interpret ] as EndBranch;
    rewrite F(age) to F(age + 1);
    rewrite X
        to F(1) [ + X ] [ - X ] F(1) X  weight 4 or
        to F(1) [ + X ]         F(1) X  weight 1 or
        to F(1)         [ - X ] F(1) X  weight 1;
}
process all with SvgRenderer;
```

Source code 1.5: Stochastic L-system with randomized interpretation of symbols and rewrite rule replacements

1.2.4 Context-sensitive L-systems

The rewriting of symbols in 0L-systems is context-free; the rewrite rules are applied to the symbols regardless of their context (the symbols around them). However, the rewriting of a symbol can also depend on its context. This is useful in simulating the flow of signals (nutrients or hormones) in a plant model that, for example, attempts to demonstrate natural plant growth [PL91].

Formally there are two types of context-sensitive L-systems, 1L-systems and 2L-systems. The rewrite rules of 1L-systems checks the context only to one side (left or right), whereas the rewrite rules of 2L-systems checks the context on both sides. Since 1L-systems are just 2L-systems with one context empty we will consider context-sensitive L-systems as 2L-systems.

The context-sensitive L-system in Source code 1.6 shows a simulation of signal propagation in a string of symbols; the result is given in Table 1.2.

```
lsystem RewritingExample {
    set symbols axiom = B A A A A A;
    set iterations = 6;
    set interpretEveryIteration = true;
    rewrite {B} A     to B;
    rewrite     B {A} to A;
}
process all with SymbolPrinter;
```

Source code 1.6: Context-sensitive L-system simulating signal propagation

Context-sensitive bracketed L-systems

If we add context-sensitive rewrite rules to bracketed L-systems the situation becomes more difficult. The context-matching procedure must take into account the branches. The following rules define the natural behavior of context between branches:

Iteration	String of symbols
0	B A A A A A
1	A B A A A A
2	A A B A A A
3	A A A B A A
4	A A A A B A
5	A A A A A B
6	A A A A A A

Table 1.2: An axiom and the first 6 iterations of an L-system in Source code 1.6 showing signal propagation in the given string of symbols

1. two symbols are neighbors even if there are some branches between them,
2. the left neighbor of the first symbol in a branch is a symbol before the branch,
3. the last symbol in a branch does not have a right neighbor,
4. unmatched symbols at the end of a branch are ignored,
5. the order of branches is insignificant.

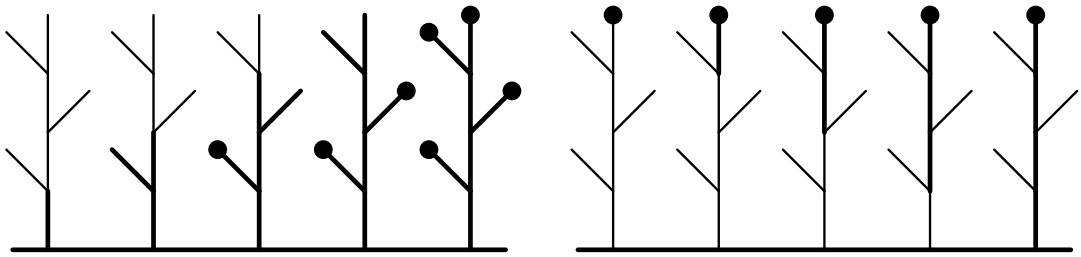
In Table 1.3 is a few examples of how a symbol with its context will match (or not) a given string of symbols with respect to the context-matching rules mentioned above.

Left ctx.	Symbol	Right ctx.	Symbol string	Match	Rule
	X	Y	A B **X** [A [B]] [C] **Y**	yes	1
	X	Y	A B X [Y B] C Y	no	
Y	X		A B **Y** [**X** A B] C	yes	2
Y	X		A B **Y** [[**X** A] B] C	yes	2
	X	Y	A [B X] Y	no	3
	X	[Y]	A B **X** [**Y** A B] A	yes	4
	X	[[Y]]	A B **X** [[**Y** A B] C]	yes	4
	X	[Y]	A B **X** [A B] [**Y**] A	yes	5
	X	[Y] [Z]	A B **X** [**Z**] [**Y**] A	yes	5

Table 1.3: Examples of context matching in bracketed L-systems

Context in bracketed L-systems can be used for the propagation of signals through tree structures. There are two basic types of signals: the first is the *acropetal* signal which spreads from the root to branches; and the second signal is *basipetal* which spreads in the opposite way i.e. from branches to root. This can be very useful in plant modeling.

Figure 1.7 shows a simulation of acropetal (1.7a) and basipetal (1.7b) signals in a static plant-like structure. Each figure shows the first 5 iterations and segments with the signal marked as a bolder line. The L-system in Source code 1.7 simulates acropetal signal propagation and its result is in Figure 1.7a (image) and Table 1.4 (symbols).



(a) Acropetal signal propagation (b) Basipetal signal propagation

Figure 1.7: Signal propagation simulated with context-sensitive bracketed L-systems

```
lsystem AcropetalSignal extends Branches {
    set symbols axiom = B [ + A ] A [ - A ] A [ + A ] A;
    // ignore + and - symbols in context search
    set symbols contextIgnore = + -;
    set iterations = 3;
    // interpret every iteration to see signal propagation
    set interpretEveryIteration = true;
    set initialAngle = 90;
    interpret A as DrawForward(50, 2);
    interpret B as DrawForward(50, 4);
    interpret + as TurnLeft(45);
    interpret - as TurnLeft(-45);
    rewrite { B } A to B;
}
process all with SvgRenderer;
```

Source code 1.7: The L-system simulating acropetal signal propagation (Figure 1.7a)

Iteration	String of symbols
0	B [+ A] A [- A] A [+ A] A
1	B [+ B] B [- A] A [+ A] A
2	B [+ B] B [- B] B [+ A] A
3	B [+ B] B [- B] B [+ B] B

Table 1.4: The result of the L-system simulating acropetal signal propagation in Source code 1.7

1.2.5 Parametric L-systems

Symbols in parametric L-systems can hold any number of arguments. Arguments are often floating point numbers, but they can be much more complicated structures. Arguments can be used in interpretation definition to send values like color or length of line to an interpretation routine. Arguments can also be used in rewrite rules to determine whether to rewrite a symbol or not, and to determine new arguments for rewritten symbols. In context 2L-systems it is also possible to get arguments from symbols in context and use them in rewrite rules.

The L-system in Figure 1.8 shows an example of how the parameters of symbols can be used in interpretation methods and in rewrite rules together with the result.

```
lsystem Circles {
    set symbols axiom = [ X ] +
        [ X ] + [ X ] + [ X ];
    set iterations = 7;
    interpret F as MoveForward;
    interpret K as DrawCircle;
    interpret + as TurnLeft(90);
    interpret - as TurnLeft(-90);
    interpret [ as StartBranch;
    interpret ] as EndBranch;
    rewrite K(n) to K(2*n);
    rewrite F(n) to F(2*n);
    rewrite X to K(2) F(3)
        [ + X ] [ - X ] X;
}
process all with SvgRenderer;
```

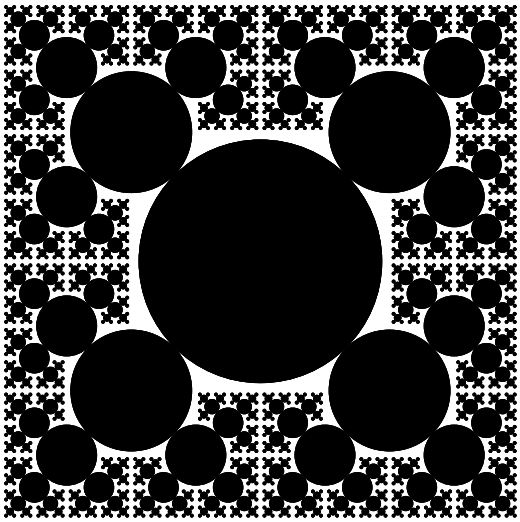


Figure 1.8: Parameters usage in L-system interpretation methods and in rewrite rules along with the result

In Figure 1.9 is more complicated model, the Pythagoras tree. Detailed instructions for its construction with L-systems are described in appendix C.

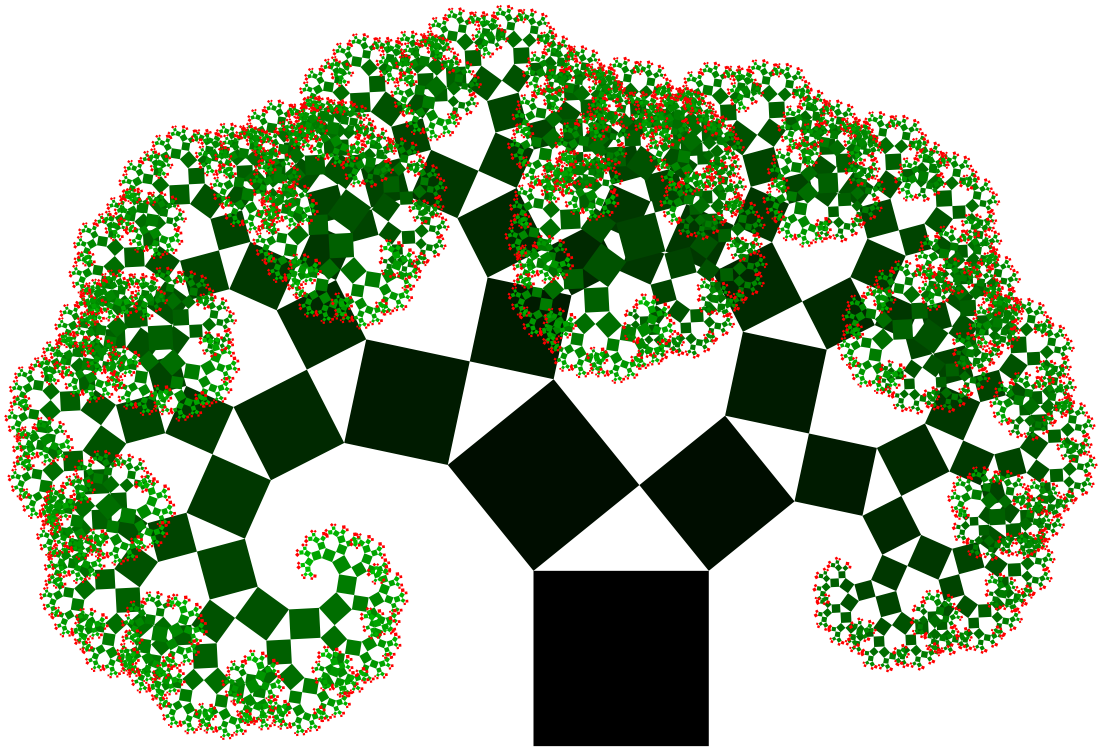


Figure 1.9: Pythagoras tree created with parametric L-system

1.3 Related L-system generators

In this section a list is given of other computer programs or web pages that allow the processing of L-systems and eventually their interpretation them in most cases as an image.

1.3.1 Web based generators

L-system generator by Michael Norris
`http://www.michaelnorris.info/software/l-system-generator.html`

A simple script which allows one to set some basic properties of an L-system: namely, number of iterations, axiom and up to 15 rewrite rules. The result is a list of strings of symbols from all iterations (it does not interpret symbols).

This site can be used to familiarize oneself with the rewriting principles of L-systems but it offers no additional functionality.

Lindenmayer power by MadFlame Software
`http://madflame991.blogspot.com/p/lindenmayer-power.html`

An L-system generator which allows the setting of some basic properties of an L-system and the interpretation of each symbol. Symbols can be interpreted using turtle graphics or they can define or modify the value of a variable. All iterations are listed as text and also drawn on screen.

The possibility to work with variables makes it a relatively powerful system, but it is only possible to draw with a thin black line. Also, the syntax is not very user-friendly and the user interface is hard to use (as well as the script not being very stable). The size of the output window is only 500×500 pixels and output cannot be saved other than using print-screen.

L-system generator by Nolan Carroll
`http://nolandc.com/sandbox/fractals/`

L-system generator has a nice looking interface where it is possible to set an L-system's basic properties. Interpretation of symbols is fixed. the last iteration of an L-system is drawn on the screen using animation (line by line from the starting position to the end).

The interface is user-friendly but the only interpretation of a symbol by drawing is a black line. There is no help nor examples; thus, it is hard to use for an inexperienced user. Output is drawn on a canvas which fills the entire area of the web browser but it cannot be saved other than using print-screen.

VRML L-system generator by Patrick Murris
`http://www.alpix.com/vrml/lsys.htm`

An L-system generator which can generate a 3D VRML model. Basic properties of the L-system and interpretation can be set and output can be produced into VRML 1.0, 2.0 or string.

The problem is that a VRML plugin is needed for displaying 3D models.

L-system generator by John Snyders
`http://hardlikesoftware.com/projects/lsystem/lsystem.html`

At first sight this is a sophisticated L-system generator which can rewrite symbols with parameters and do context-sensitive rewriting. Results can be drawn on a page as an animation of the development and a progress bar shows its status. The biggest drawback is that L-systems are hard-coded in JavaScript and it is only possible to change the number of iterations.

This L-system generator contains many examples and output is rendered as image which can be saved, but examples are the only thing that it can produce.

WWW L-system Explorer by Zdík Kudrle
`http://zdeeck.borg.cz/wlse/l-system.php`

An L-system generator with a well-arranged user interface where it is possible to set the basic properties of the generated L-systems. The interpretation of symbols is fixed and it uses an unusual set of interpretation methods like *pen up* and *pen down* instead of the traditional *draw line* and *move forward*. The last iteration is drawn as an image by server-side PHP script thus output can be downloaded easily. It is possible to set line color (even to color gradient) and background color of the image. Size of the output image can be set freely.

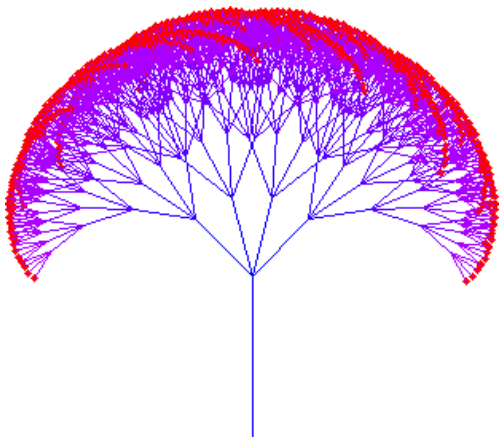


Figure 1.10: Image produced by WWW L-system Explorer

This web-based generator is the best among the generators listed in this section. It contains a well-written help section and several examples. However it cannot do context rewriting and symbols can't hold any parameters. The length of drawn lines or turns can only be adjusted by increasing *depth level* and setting the change ratio. An example of a plant-like model produced by WWW L-system Explorer is shown in Figure 1.10.

1.3.2 Desktop applications

L-systems explorer by James Matthews
`http://www.generation5.org/content/2002/lse.asp`

A simple desktop application which renders L-systems in the application window. Basic properties of an L-system and its interpretation can be edited in a dialog window but interpretation of individual symbols cannot be changed. It is possible to move and zoom the model with a mouse. L-systems can be saved or loaded into a text file and the drawn image can be saved to the clipboard.

L-systems explorer can be used for generating of simple models but it is not possible to do context rewriting or use symbol parameters; even line thickness cannot be changed. The user interface for editing an L-system is very simple (it is only possible to show rewrite rules for one symbol at a time).

L-system Vector Generator by Dmitry Malutin

`http://xaraxtv.at.tut.by/lsvg.htm`

A similar application to L-systems explorer by James Matthews but with a better user interface and it is also possible to randomize line lengths or turn angles. a nice feature is the *angle wizard* which displays grid a of L-systems each with a different setting of turning angle - allowing the user to easily make their choice. Drawn lines can be automatically closed to form polygons. It is possible to save an image as AI (Adobe Illustrator) or WMF (Windows Metafile) which are both not very common formats.

This application contains hundreds of examples but it lacks any advanced types of L-systems or interpretation settings. Application window size is about 700×550 pixels and it cannot be resized. One of the built-in examples with randomized angles and line lengths is shown in Figure 1.11.

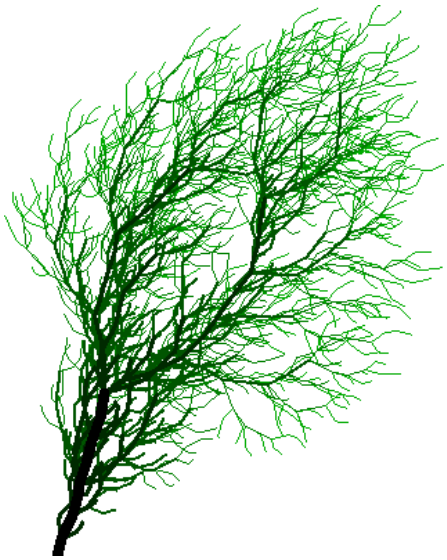


Figure 1.11: A plant example from L-system Vector Generator

L-system 4 by Timothy Perz

`http://www.oocities.org/tperz/L4About.htm`

L-system 4 is a relatively advanced tool for generating models with L-systems. Besides all the basic functionality it is possible to create 3D models with custom textures. Models can be saved as raster images (BMP or JPEG) or they can be exported to AutoCAD DXF format. Interpreting capabilities are quite good but it can only do deterministic rewriting with a limited usage of parameters.

A table of symbol interpretations (which are not changeable) can be displayed at right-hand side of the application: a nice feature. L-system 4 has good capabilities for producing 3D output but the input syntax is very compact and hard to read. Also, more advanced L-system types like, context-sensitive or parametric L-systems are, not supported.

L-studio by Przemysław Prusinkiewicz et. al

`http://algorithmicbotany.org/lstudio/`

L-studio is probably one of the best applications designed for modeling plants with L-systems. L-studio is not a single program but it is a suite of program modules that consists of many tools. L-studio can process all the types of L-systems described in section 1.2 and also produce the animation of plant growth. With L-studio it is possible to model 3D models of plants with regard to environmental factors such as wind, gravity, the space around a plant, sunlight, etc. The output model can be saved in many formats such as Wavefront OBJ, Postscript, or BMP, or it can be rendered with its built-in ray-tracer to produce photo-realistic images.

There are even many examples of plant models and an extensive help section, though certainly it is not easy to start using it. The syntax is very compact and lack clarity for a new user.

The application is not free-ware but a demo version can be downloaded. After an evaluation period it is still possible to use it but it is not possible to export images and previews have a watermark. In Figure 1.12 is one of the most beautiful examples from L-studio, the Lily.

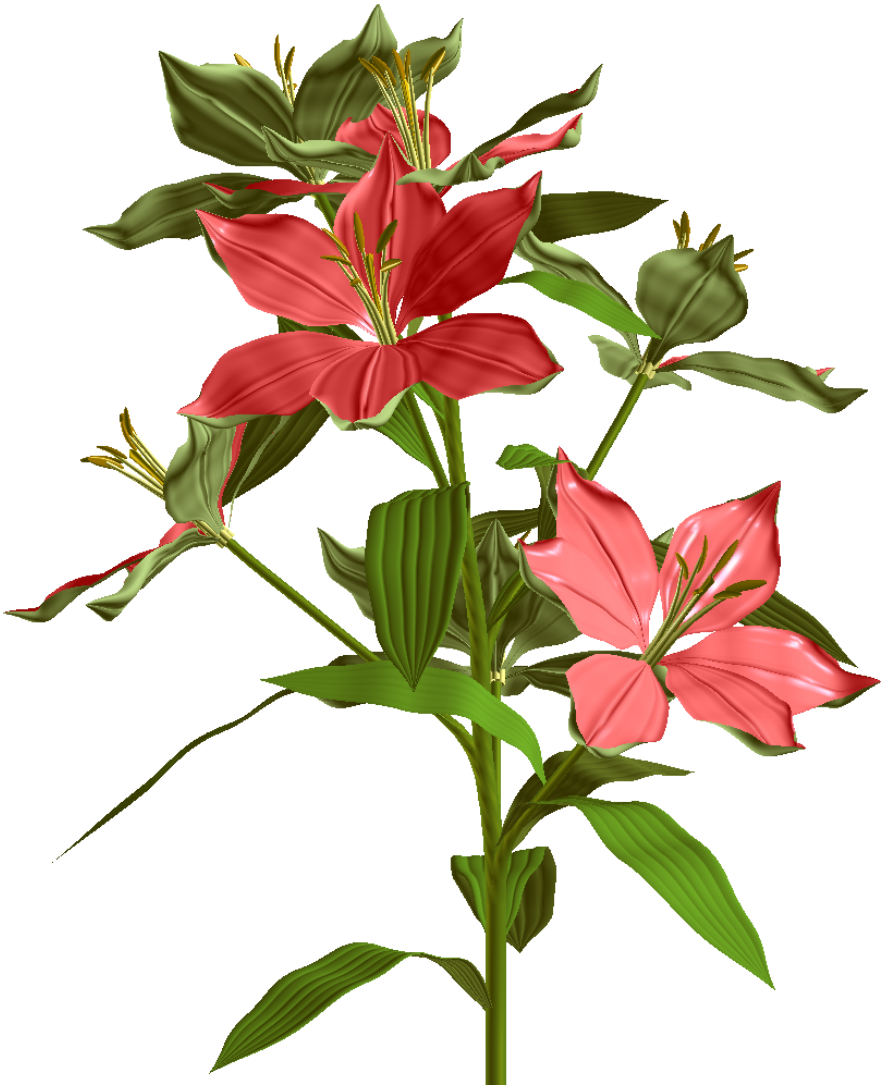


Figure 1.12: Model of Lily produced by L-studio

2. Design

In this chapter the design of my solution for an online feature-rich development environment is described and the decisions that were made explained. Implementation details are described in chapter 3.

In the first section (2.2) the design of the L-system processing library is described. The library processes input with a component-based approach. The core of the library is responsible for creating system a of connected components (components graph) but processing of the L-system itself is fully under the control of the components. Components can be created by the user, thus bringing freedom to the L-system processing.

The library contains predefined components to make it possible to process L-systems without need the for creating custom components. The design of these components is described in the second section (2.3). Predefined components also serves as an example for users who want to implement their own components or whole processing system.

In the third section (2.4) is described the design of the online web user interface. It uses the library and components to process input so it also serves as an example of the usage of the library.

2.1 Choice of development environment

As development environment was chosen the .NET framework because of following reasons.

Multiplatformity Thanks to the Mono project[1] .NET libraries and executables can be used not only on Windows but also on Linux, Mac and many other operating systems.

Development tools Visual Studio 2010 is powerful integrated development environment (IDE) with many integrated tools (like inteli-sense, NuGet package system or T4 templates) and useful downloadable plugins. Visual Studio has built-in support for unit testing which helps to test especially non-runnable code like libraries easily.

Reflection Reflection is the ability to examine types and work with meta-data, properties and functions of an object at runtime. Reflection can be used to load various plugins or data at runtime and help extensibility in great way.

Parser generator FsLex and FsYacc are lexer and parser generators written in F# with good support by Visual Studio. Generated lexer and parser are also in F# thus they can be easily used in any .NET project.

Web framework ASP.NET MVC is a lightweight presentation framework for creating web applications in .NET. ASP.NET MVC 3 is using the Razor view engine which helps to do the web very easily.

[1] Mono is an open source implementation of Microsoft's .NET Framework (`http://www.mono-project.com`).

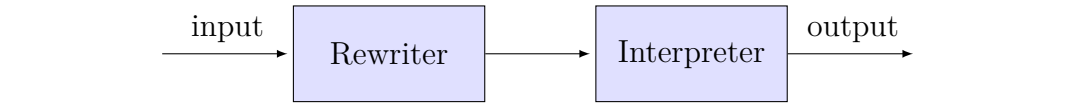
Database and object mapping MsSQL server offers to create database stored as a file directly in the application folder. Access to the database can be done using ADO.NET Entity Framework (EF) which can do an object-relational mapping (ORM) of the database.

2.2 L-system processing library

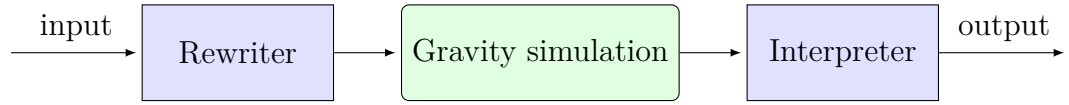
The main design goal of the L-system processing library was that it should be simple to extend. It should be possible to alter the processing of an L-system without needing to change the whole processing system. This behavior is achieved by its modular design – input is processed within a set of connected components. Each component is specialized for one particular activity: for example, symbol rewriting or the rendering of an image. Both components and connections are definable by the user.

A component-based modular system has many advantages over a monolithic system. Probably the biggest advantage, one already discussed, is its ease of extensibility. It is simple to implement the extension of a component and include it into the system. It is also possible to improve existing components and extend system capabilities. A simple illustration of how this component system extension works is shown in Figure 2.1. The first system (Fig. 2.1a) was extended by a gravity simulation component being added to give the second system (Fig. 2.1b). The possible results are shown in Figure 2.2.

Another advantage lies the in specialization of its components. Specialized components are easier to implement and the outcome will most likely have less bugs. Specialization also helps to make a system more robust because individual components can be tested separately. Tests for a single component are easier to write and they can test situations which cannot be tested for on the whole system.



(a) Simple component-based system



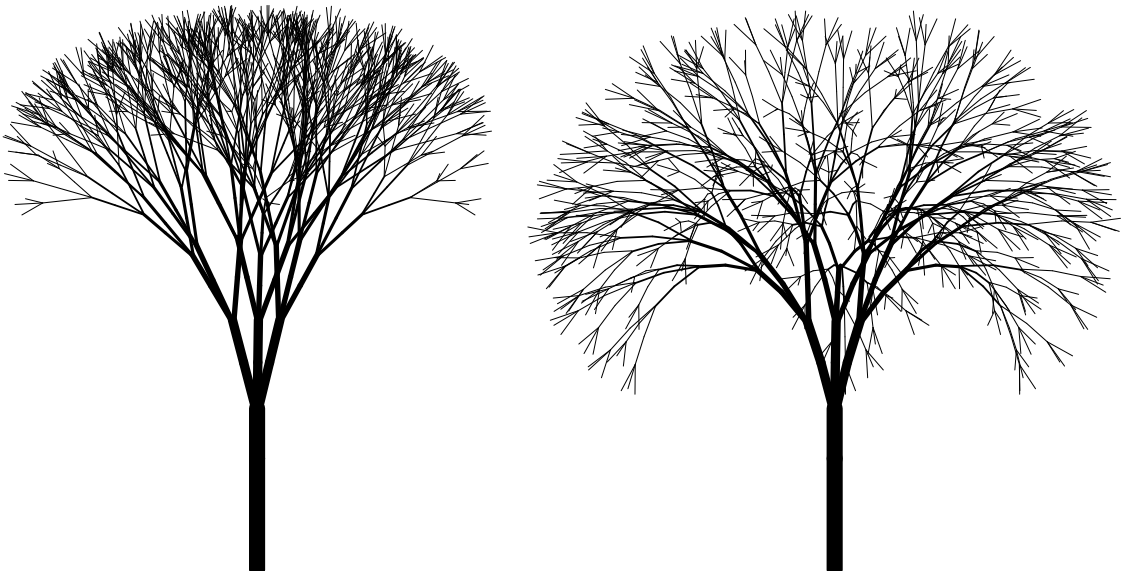
(b) Extended component-based system

Figure 2.1: The extension of component-based processing system

2.2.1 Input form

Input is an important part of an application. L-systems have no standardized input: for example, like programming languages. Every implementation of an L-system processor uses its own variant of the input.

The main goals for input design are simplicity and universality. With simple input there is lower threshold for a new user to start using an application. Complicated input might well discourage many potential users. Universal input means



(a) Original tree model (no effect of gravity) (b) Tree model with simulated gravity

Figure 2.2: Possible outputs from process systems in Figure 2.1

that *everything* is possible to define by input, including definitions of L-systems, configuration of components, etc.

Generally there are two basic types of input, the graphic interface and source code. Source code was chosen to be he input because it is better for saving, sharing and versioning. Statements can be easily commented upon, thus the ideas behind the code can be saved with it. Parts of the code can be copy-pasted and the syntax can be extended. Parsing of source code is quite complex but the input interface can be just one text area.

To achieve good readability of input source code the syntax will have to be rich in keywords. This should ensure that even new users will understand the statements. With source code input it is still possible to create a second type of input – the graphic interface. This can be achieved by source code designers but this will be left to a future extension.

2.2.2 Input syntax

Everything can be described by input. By everything is meant L-system definitions, configuration of components, definitions of component systems, which L-system to process with which component system etc. This is important for easy saving and sharing.

The following list describes concrete entities which are possible to describe with source code.

- Global constant
- Global function
- L-system definition
  - Local constant
  - Local function

  - Component property assignment
  - Component symbol property assignment
  - Symbol interpretation – defined interpretation for one or more L-system symbols
  - Rewrite rule
- Process configuration – definition of component system
  - Component
  - Container – components in container a can be reassigned when an L-system is processed by process configuration
  - Connection – defines the connection between two components
- Process statement – defines processing of the L-system with process configuration
  - Container component reassign – change of component in container
  - Additional L-system statements – additional L-system statements which can alter the processed L-system

A full reference of the input syntax is given in appendix H. A small example of input source code together with the result is shown in Figure 2.3.

```
lsystem SierpinskiGasket {
    set symbols axiom = + R;
    set iterations = 7;

    interpret L R as DrawForward(
        2 ^ -currentIteration
        * 700);
    interpret + as TurnLeft(60);
    interpret - as TurnLeft(-60);

    rewrite L to R + L + R;
    rewrite R to L - R - L;
}
process all with SvgRenderer;
```

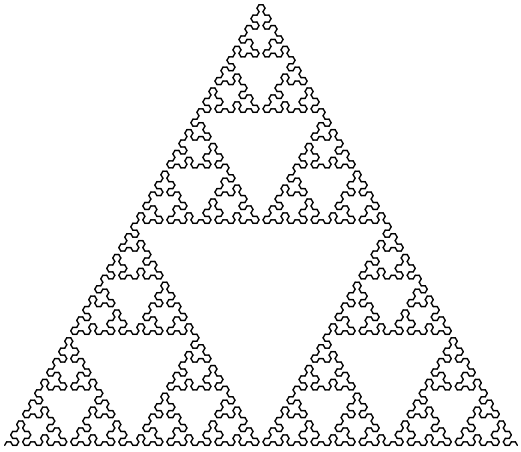


Figure 2.3: Example of source code along with the result – Sierpinski gasket

Value types

The L-system processing library allows to use two types of values for constants or for configuration of components. The first type is a *number* and it is stored as floating point value with about 15 significant digits. The numbers can be specified in 5 formats:

- floating-point format like 24, -8.1, 2e-10, 3141.59e3,
- binary format with prefix *0b* like 0b100101 or 0b0111011,
- octal format with prefix *0o* like 0o54607, 0o776,

- hexadecimal format with prefix *0x* like 0xFFBF00, 0x13a4F,
- hexadecimal format with prefix # like #332211 (this is handy for colors).

The second type is a *array*. The elements in the array can be both the constants and the arrays. The array is started and ended by a brace and the values are separated by a comma. The syntax of the array is illustrated in Source code 2.1.

```
let const = 0xAF + 1.2e1;
let array = {1, 2, {3.1, 3.2, 3.3}, const, 5, {}};
```

Source code 2.1: Example of array syntax.

L-system inheritance

The L-system can *inherit* (extend) all features from some other L-system. This feature should minimize code repetition and allows to define base L-systems in the standard library to simplify the work with L-systems (see appendix I.3).

The pre-existing L-system is called *base* or *ancestor* and the new L-system is called *derived* L-system or *child* L-system. Inheritance L-system statements follows a simple rule: the derived L-system will redefine definitions of the base L-system. The result of Source code 2.2 is `B B B B B A` (the first process statement) and `X X A` (the second process statement).

```
lsystem BaseLsystem {
    set symbols axiom = A;
    set iterations = 5;
    rewrite A to B A;
}
lsystem DerivedLsystem extends BaseLsystem {
    set iterations = 2;
    rewrite A to X A;
}
process BaseLsystem with SymbolPrinter;
process DerivedLsystem with SymbolPrinter;
```

Source code 2.2: Example L-systems inheritance.

It is possible to inherit more than one L-system. Their statements are redefined in the same order as stated in the definition. The result of Source code 2.3 is `Y Y A` (the first process statement) and `X X A` (the second process statement).

```
lsystem BaseLsystemX {
    set symbols axiom = A;
    set iterations = 5;
    rewrite A to X A;
}
lsystem BaseLsystemY {
    rewrite A to Y A;
}
lsystem DerivedLsystemXY extends BaseLsystemX, BaseLsystemY {
    set iterations = 2;
}
lsystem DerivedLsystemYX extends BaseLsystemY, BaseLsystemX {
    set iterations = 2;
}
process DerivedLsystemXY with SymbolPrinter;
process DerivedLsystemYX with SymbolPrinter;
```

Source code 2.3: Example of the array syntax.

2.2.3 Source code compilation

Source code compilation has two steps as shown in Figure 2.4. The first step is parsing the of source code to the *abstract syntax tree* (henceforth called AST). The syntax parser is generated by the *FsYacc* parser generator which will guarantee a robust and extensible syntax paring (for details see section 3.2). Each node of the AST will contain information about its position in the original source code. At this point the AST can be used for syntax highlighting or source code formatting.

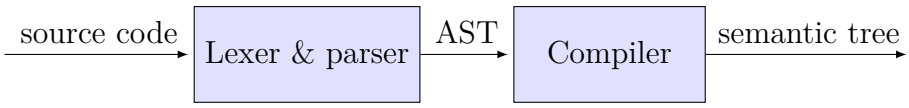


Figure 2.4: Source code compilation system

The abstract syntax tree (AST) for Source code 2.4 is shown in Figure 2.5. You can see that expressions are not parsed as a tree, they are placed in a linear list under the *Expression* node. This is because operators (and their precedences) can be defined by the user, thus parser can not parse expressions where operator precedence is needed (like $2+3*4$). Expressions will be parsed to the expression tree by the compiler in the next step.

```
let angle = 90 + 5 * random();
```

Source code 2.4: Constant definition statement for example of AST

The next step of source code processing is compilation of the AST into the *semantic tree* (henceforth called ST). Unlike the AST the semantic tree contains only data (no keywords or metadata about position). However the nodes of the semantic tree have a reference to the corresponding AST nodes to make it possible to get the metadata, for example, for reporting the locations of compilation errors. The semantic tree in Figure 2.6 is created by compiling the AST in Figure 2.5. A more complex semantic tree obtained from Source code 2.5 is shown in Figure 2.7.

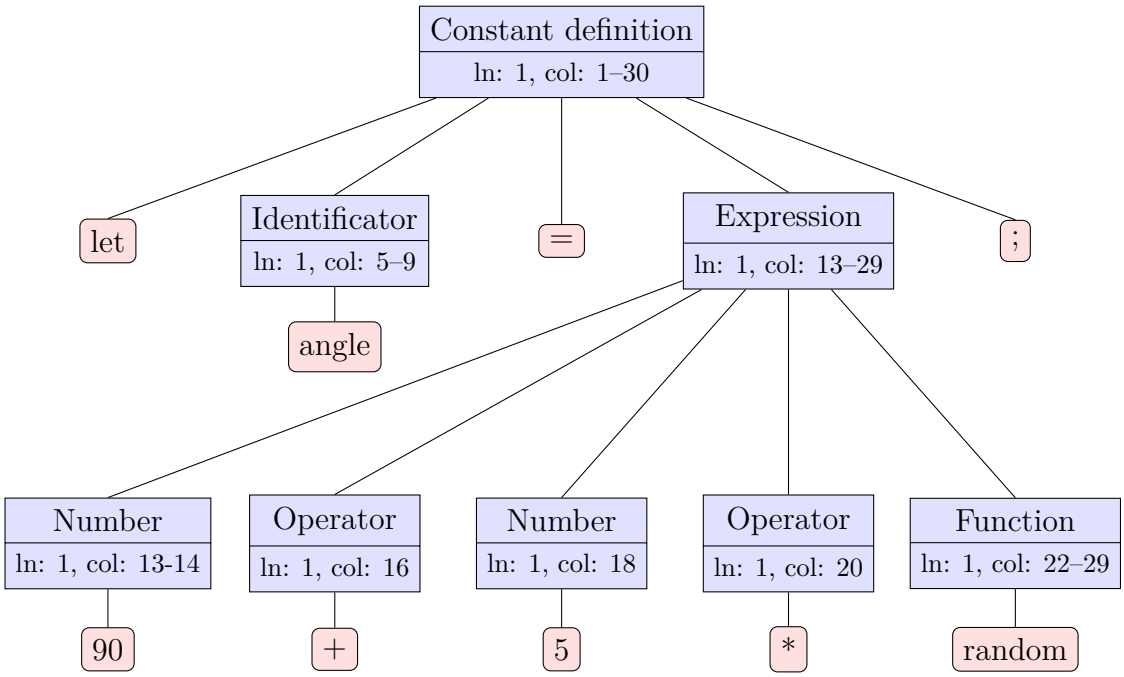


Figure 2.5: Abstract syntax tree parsed from Source code 2.4

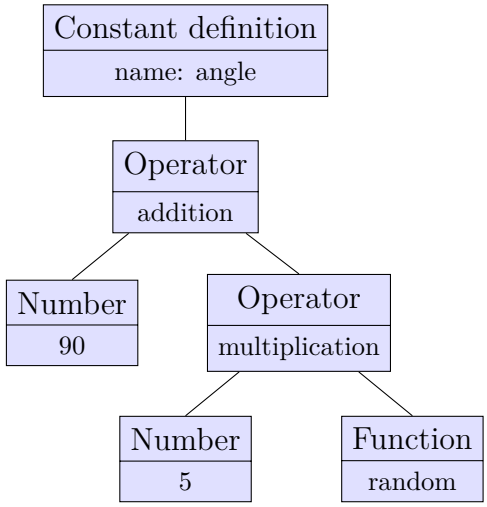


Figure 2.6: Semantic tree created by compilation of the AST in Figure 2.5

```
let iterBase = 2;
lsystem Octahedron {
    set symbols axiom = F;
    set iterations = iterBase + 1;
    interpret F as DrawForward(100, 2);
    interpret + as TurnLeft(45);
    rewrite F to F + F;
}
process all with SvgRenderer;
```

Source code 2.5: This source code results in the semantic tree shown in Figure 2.7

2.2.4 Input processing

Evaluation of the ST follows after compilation of the input. *Process statements* that define which L-system to process with which component system are chosen from the evaluated ST. The *process manager*[2] creates the appropriate component system, configure it and supply it all the data needed for processing. Then the control over processing is handed to the component system and the results are produced. The processing of the L-system is fully under the control of the component system. The described procedure is shown in Figure 2.8.

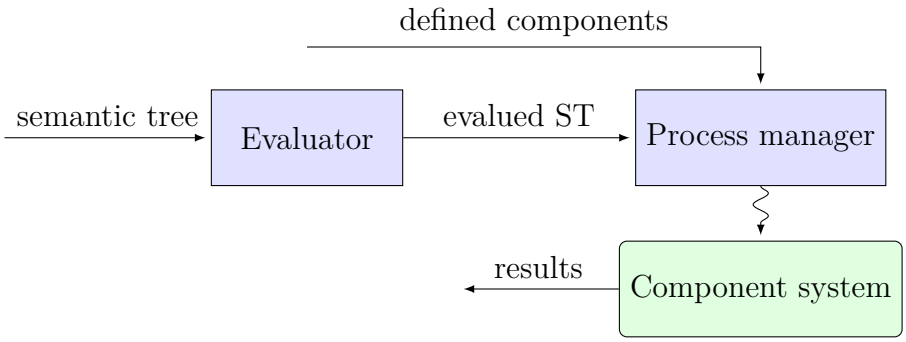


Figure 2.8: Input processing scheme

All components have access to *process context*. Process context contains all the properties of the processed L-system, current components graph, output provider and some other data which can be used in processing. A component can also provide *values* or *functions* to other components or for use in the input L-system (for example, in the rewrite rules or interpretation methods).

2.2.5 Components

The components are configurable by the input. The *settable properties* allow setting of values (numbers or arrays) whereas the *settable symbol properties* allow setting of L-system symbols. Other components can be connected to the *Connection properties*. Components can provide the *gettable properties*, the *callable functions* and the *interpretation methods*.

[2]The process manager is part of the L-system processing library responsible for the processing of the input.

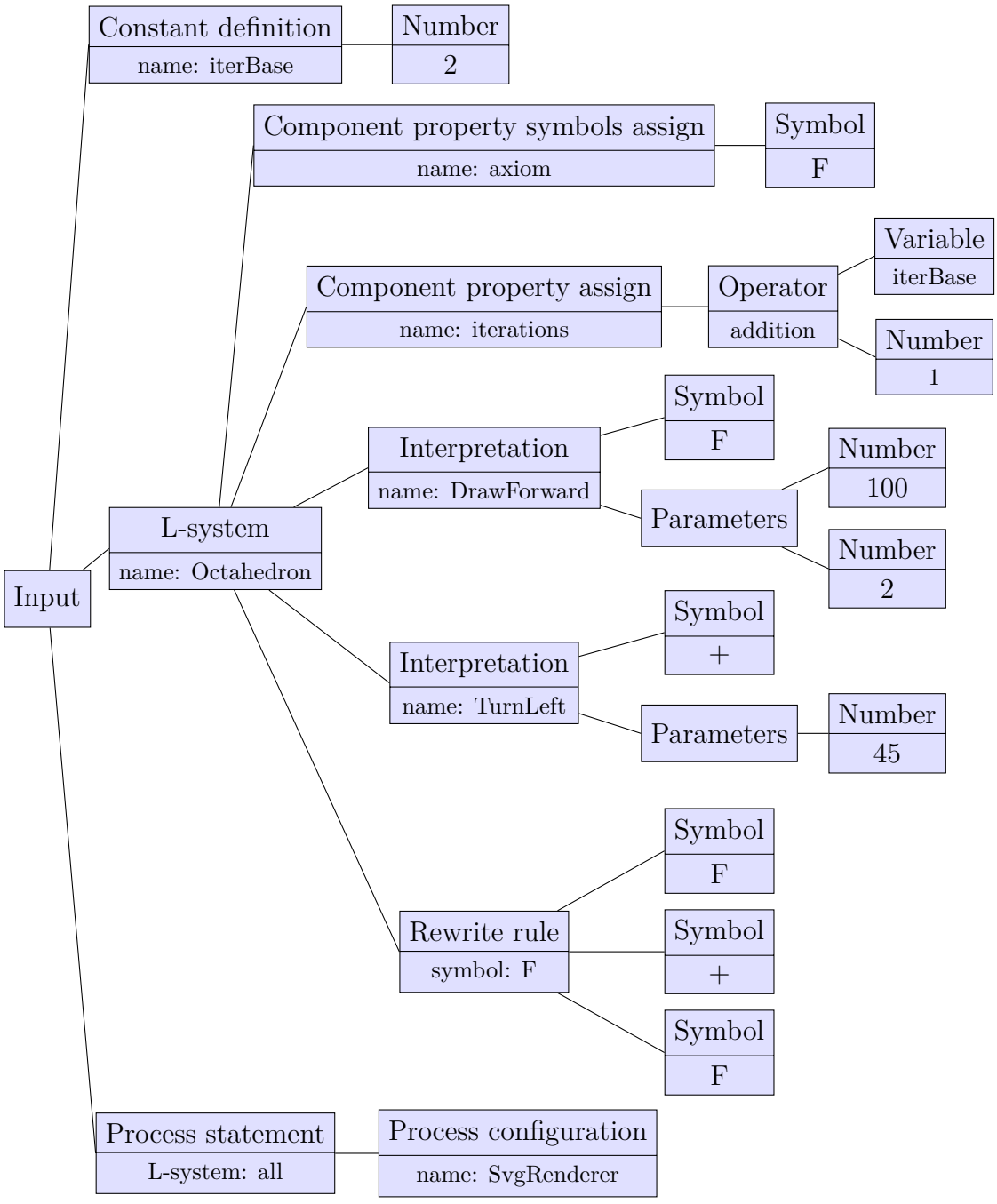


Figure 2.7: A more complex semantic tree of Source code 2.5

The component is .NET class. The settable and gettable properties are *properties* of the .NET class marked with special attributes. Callable functions and interpretation methods are methods of the .NET class also marked with the attributes. Concrete details about attribute types can be found in the section 3.4 and appendix D describes the implementation of a simple component.

2.2.6 Measuring pass

Some components may need to know some information about the processed model even if the model has not yet been completed. For example, when some renderer component is producing an image its dimensions may be needed before any drawing can be started. Also, if we want to continuously color all lines with some gradient we must know the total number of drawn lines.

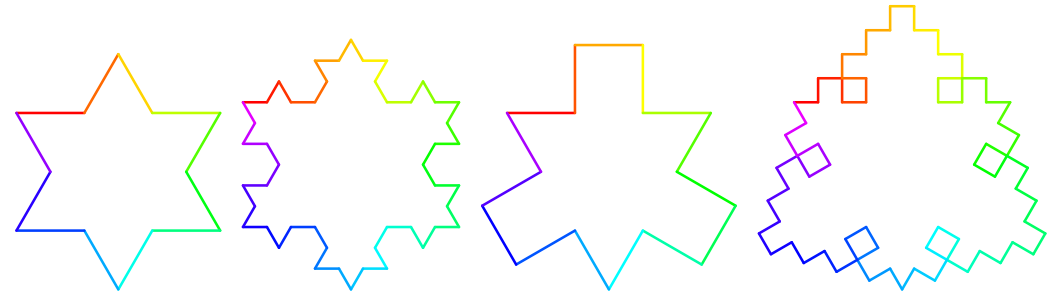
The only way how a component could achieve this is to cache all input and count the needed metadata, and after all the input has been supplied produce the output. However, this approach raises the complexity of components and it can lead to significant increases in memory demands. Caching also prevents communication between components about the current state of the L-system. Imagine that some renderer wants to communicate with a rewriter about concrete rewriting options and some component in their way caches all the data. At the moment when the renderer is ready to start to render the image, the rewriter has already done all the rewriting.

The library uses another way to allow components to pre-count some metadata. It is called the *measure pass*. If any component needs to pre-count some metadata, the process manager invokes processing of the whole system twice. The first pass is the *measure pass* where all components can count metadata and no output is produced. The second pass is the ordinary processing of the L-system but the the components now have the metadata already counted. This way does not prevent communication between components about the current state of the L-system.

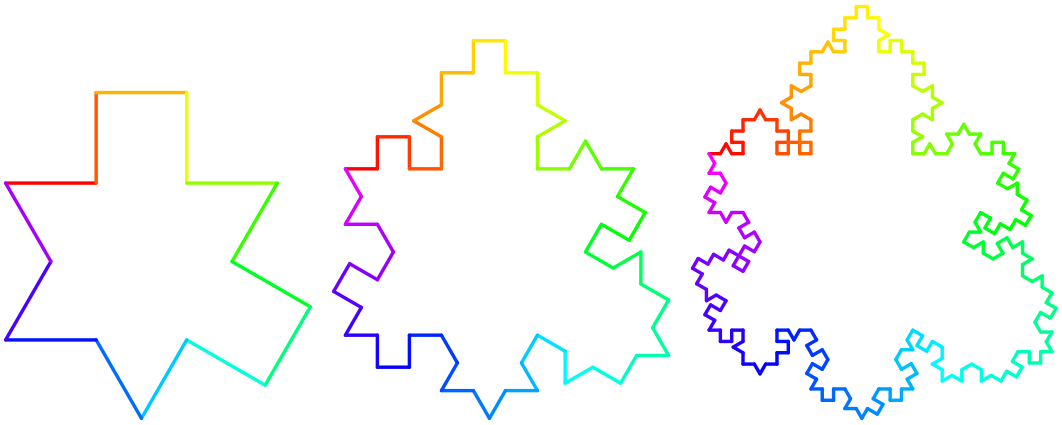
It is important to ensure that both passes will be equal. For example, problems can occur if a component is using a random generator for a randomizing of processing. The library provides a unified approach for random numbers generation with the *Random provider* component. The pseudo-random generator of the Random provider is reset after each pass to the same value. If the value of the random seed is not provided by the user it is generated randomly but it will be the same for both passes. The actual value of generated the seed is supplied to the user via the message system to make it possible to reproduce the output.

Figure 2.9 demonstrates the usage of the measure pass with the continuous coloring of line segments with a rainbow gradient for the L-system in Source code 2.6. The axiom of the L-system is an equilateral triangle. In every iteration, the rewrite rule rewrites every line segment to line segment with a triangle or square a on it with the same probability of 50%. The effect of the ”triangle” part of the rewrite rule is shown in Figures 2.9a and 2.9b, the effect of the ”square” part is shown in Figures 2.9c and 2.9d. Figure 2.9h shows randomized combination of the previous two rules. The first three and fifth iterations of the L-system in Source code 2.6 are in Figures 2.9e, 2.9f, 2.9g and 2.9h respectively.

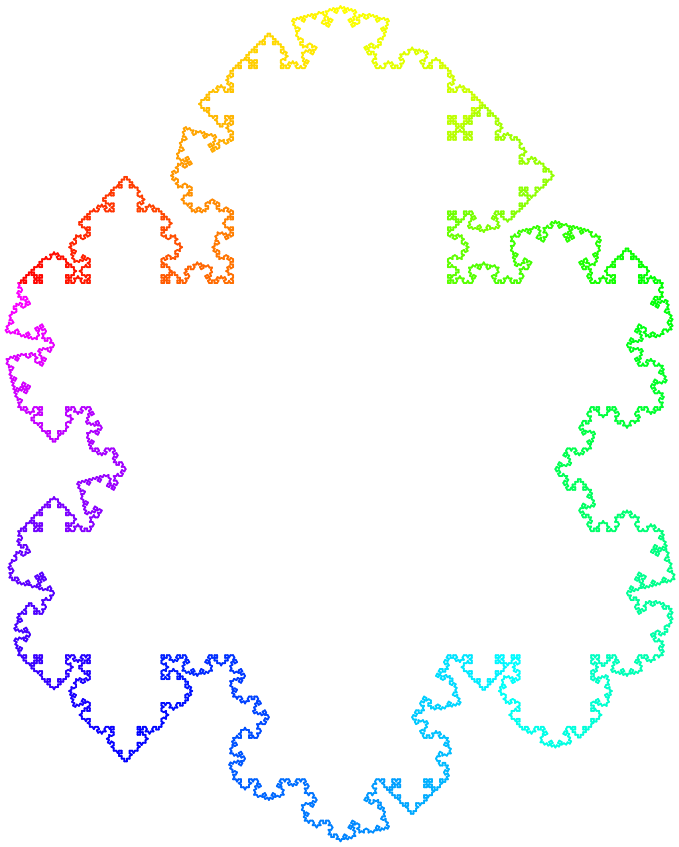
Notice that even tough the number of colored lines of the L-system is random,



(a) Triangles 1st (b) Triangles 2nd (c) Rectangles 1st (d) Rectangles 2nd



(e) Stochastic 1st iter. (f) Stochastic 2nd iter. (g) Stochastic 3rd iter.



(h) Correctly colored 5th iteration of stochastic L-system

Figure 2.9: Example of stochastic L-system which is correctly colored by color gradient even if total number of colored line segments is random

the rainbow gradient is applied correctly (Figure 2.9h). It is because in the measure pass the number of lines has been counted and the total sum used in the ordinary pass to distribute the gradient correctly.

```
lsystem WeirdKochCurve {
    set symbols axiom = F +(-120) F +(-120) F;
    set iterations = 5;
    set randomSeed = 2;
    set continuousColoring = true;

    interpret F as DrawForward(16);
    interpret + as TurnLeft;

    rewrite F
        to F +(60) F     +(-120)     F +(60) F  // triangle
        to F +(90) F +(-90) F +(-90) F +(90) F; // square
}
process all with SvgRenderer;
```

Source code 2.6: Stochastic L-system with a variable number of line segments

2.2.7 Utilities

The library contains some useful functionality – especially for implementation of components.

One large part is functions available for working with 3D. The most important one is the utility for the triangulation of 3D objects defined by their perimeter (the 3D version of 2D polygons). It is able to triangulate any object in space, but the triangulation of 3D objects defined by their perimeter is ambiguous so the triangulation strategy is configurable by the user. Part of the 3D utilities are functions for the manipulation of points, vectors and quaternions.

The next utility serves for source code printing. It is possible to print the abstract syntax tree as well as the semantic tree to the source code.

2.3 Processing system

As discussed in the previous chapter, the processing system of the library relies on the components. The core of the library is responsible for just creating the component graph. The processing of an L-system and production of results is fully under the control of the component graph. This gives absolute freedom to the user in implementing the process system.

However it is hard to design and implement the whole L-system processing system from scratch. The library contains a rich set of predefined components from which can be assembled many different component graphs. The predefined components have a general interface which allows the user to reuse or extend them in order to add new functionality with a minimum of effort.

2.3.1 Basic component system

The component system designed in this section is primarily used for processing L-systems to produce 2D and 3D graphics in the web interface. However, the

component system is designed to be extensible to any output type.

L-systems are generally processed in two phases. The first phase is rewriting where the axiom (the initial string of symbols) is rewritten by the rewrite rules, and the second phase is interpreting the resulting string of symbols. This can be done with two components, the Rewriter – which is responsible for rewriting the L-system to a given iteration – and the Interpreter – which is responsible for interpreting symbols and producing output (Fig. 2.10).

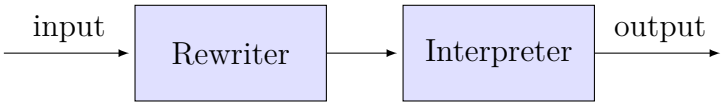


Figure 2.10: Simple L-system processing system

However the components in the system in Figure 2.10 have too many tasks to do, and thus they will be complicated to implement and hard to extend and test.

The system in Figure 2.11 was created by a subdivision of the previous system. The Rewriter component was split to the Rewriter and the Iterator. The (new) Rewriter will just do the rewriting of some given symbols and the Iterator will control the iterating of the L-system (repetitive rewriting). The Interpreter component was split to the Interpreter and the Renderer. The (new) Interpreter will handle the interpreting of symbols: which means keep position of the virtual "turtle" in space, saving and loading of states, etc. The Renderer will just produce the output. If we need to create a different output type we only have to implement the new renderer component and the rest of the system will remain unchanged.

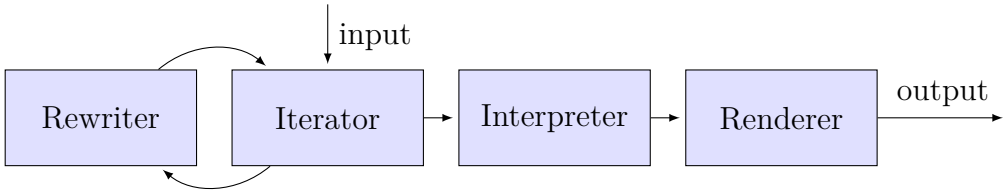


Figure 2.11: Subdivided L-system processing system

2.3.2 Component system extensions

The system in Figure 2.12 can be enhanced even more. Every component that interprets L-system symbols needs to translate symbols to interpretation methods. The translation can be implemented by every component individually. However, the translation can be done by a specialized component called the *Interpreter caller*. This component can be smart enough to explore all the components in the system, find all the interpretation methods of all components and do translation automatically. This causes an automatic "connection" of all interpreters to the interpreter caller.

More interpreters can be used to advantage: for example, for processing L-systems which interact with themselves or their Environment [MP96]. One interpreter actually creates the result model and the second interpreter simulates the environment.

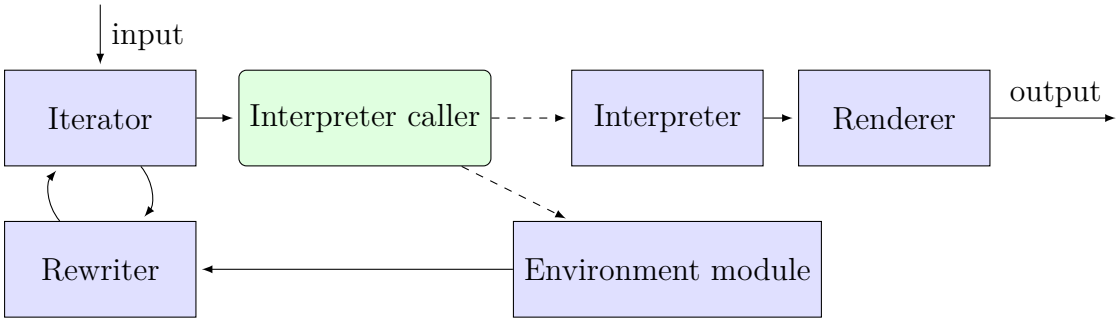


Figure 2.12: The Interpreter caller which automatically calls interpretation methods of any components

The next necessary component is called the *Random provider*. It provides controlled behavior for random number generation as described in section 2.2.6. This component provides a function which returns a random number and it can be called by other components or by the user in the L-system definition. This component is connected to the iterator to correctly the reset random seed at every pass.

The *axiom provider* is the next extending component and it provides the axiom to the Iterator. The axiom provider is only a "wrapper" around a single symbol property called the axiom. The Iterator is designed generally to take the axiom from any component implementing *ISymbolProvider* interface so it is possible to connect, for example, another rewriter as the axiom provider (Fig. 2.13).

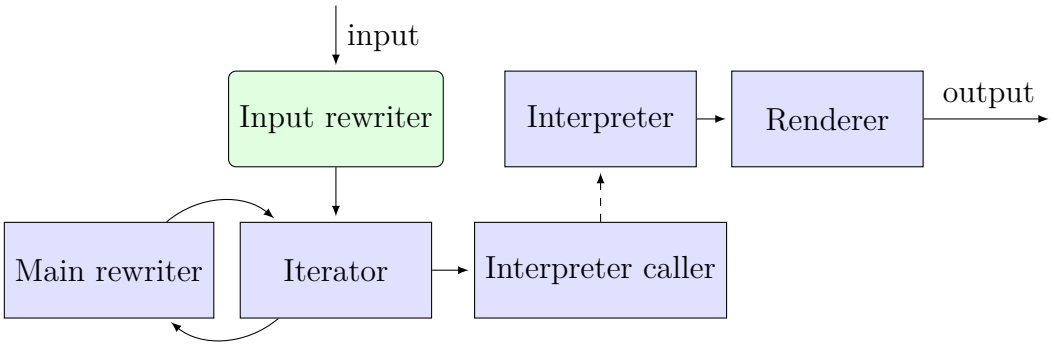


Figure 2.13: Input for the iterator can be supplied by another component

2.3.3 Interpretation of a symbol as another L-system

In some situations it can be handy to interpret a symbol as another L-system. The component system of the library is very versatile and it allows the creation of a specialized component which will be responsible for just this feature.

The component is called the *Inner L-system processor*. It is connected to the Interpreter caller and when the caller needs to interpret a symbol as an L-system it will call the Inner L-system processor to take care of this. A component graph with the Inner L-system processor component is shown in Figure 2.14.

The Inner L-system processor works internally in a similar way to that of the Process manager (see section 2.2.4). For every processed symbol it builds a new components graph for processing the inner L-system. The components graph

can be specified by a special process configuration which needs to be defined in the input[3]. The interpreter caller in the inner component graph is automatically connected to all interpreters in the original component graph (see section 2.3.2), therefore the inner L-system is interpreted by the same interpreter as the main L-system. The inner interpreter caller is also connected to the Inner L-system processor: thus it is possible to interpret a symbol as another L-system even in the inner L-system.

The creation of the inner component graph is a relatively complex operation. The created and used component graphs are cached and reused later which improves the performance.

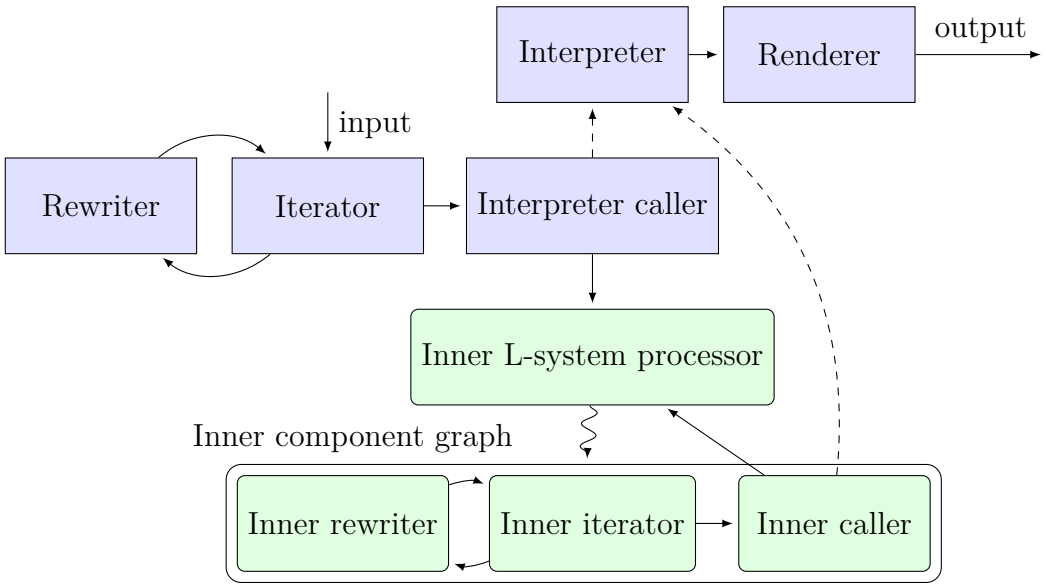
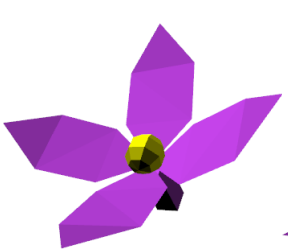


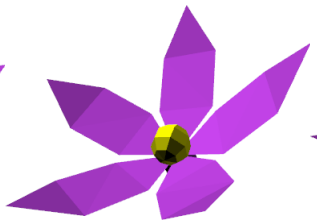
Figure 2.14: Component system for interpretation of a symbol as another L-system

The interpretation of symbol as another L-system is demonstrated in Figure 2.15. The Pythagoras tree is made of Menger sponges: number of iterations of each Menger sponge depends on its size. The smallest Menger sponge (zero iteration) has an extra blossom as a demonstration of an interpretation symbol as an L-system in the inner L-system. The iteration of the Blossom L-system determines the number of leaves and it is randomly selected from 4 to 6.

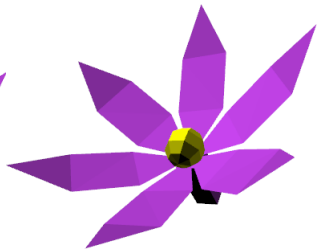
[3] The only implementation of the Inner L-system processor the *LsystemInLsystemProcessor* component uses the process configuration called *InnerLsystemConfig* for creating the inner component graph. This process configuration must be defined (see the definition in the Standard library I.4.5).



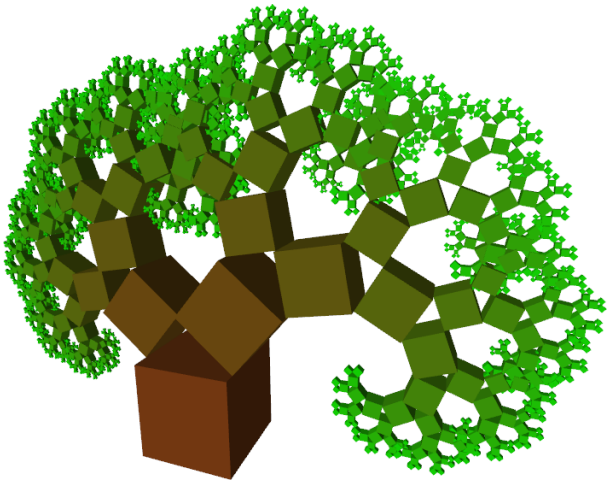
(a) 4th iteration



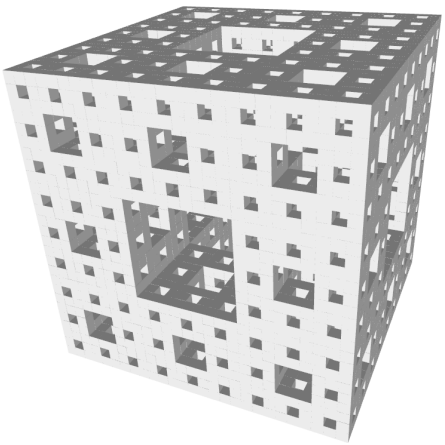
(b) 5th iteration



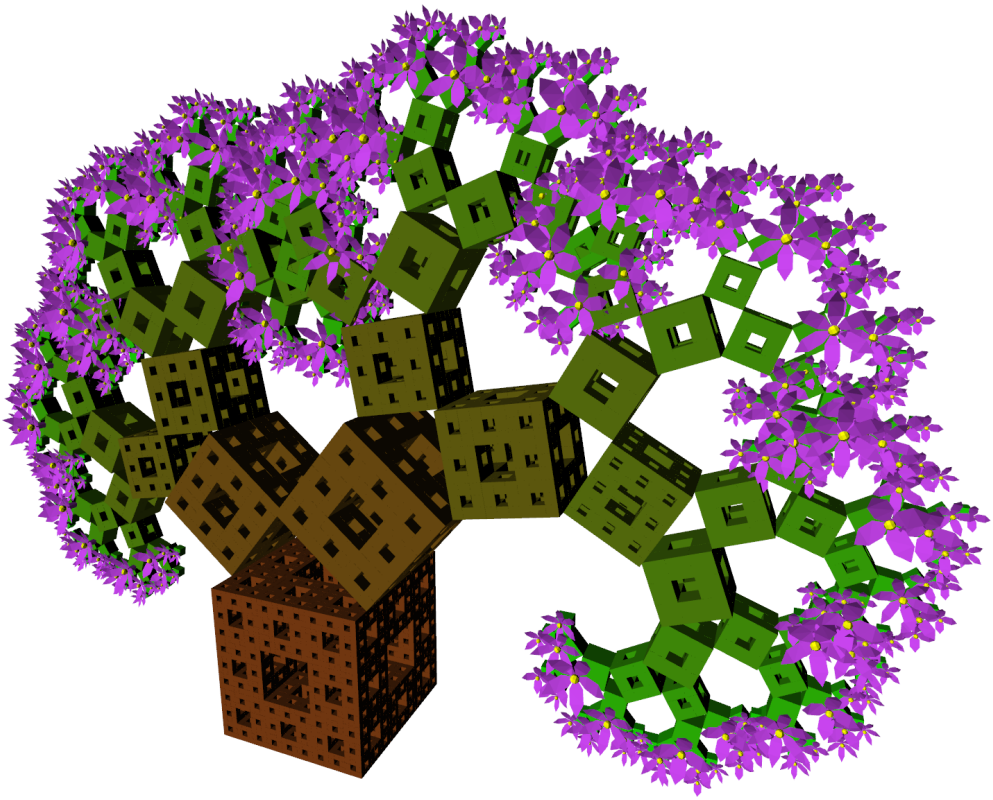
(c) 6th iteration



(d) 11th iteration of the Pythagoras tree



(e) 3rd iteration of the Menger sponge



(f) The Pythagoras tree made of the Menger sponges with blossoms at the smallest cubes

Figure 2.15: Example of interpreting s symbol as another L-system

```
lsystem HybridPythagorasTree(angle = 50) extends Branches {
    let angleComp = 90 - angle;  // angle complement
    let sinAngle = sin(deg2rad(angle));
    let sinAngleComp = sin(deg2rad(angleComp));
    set iterations = 8;
    set symbols axiom = F(64, 0);
    // interpret E(x) as DrawForward(x, x);  // cube
    interpret E(x) as lsystem MengerSponge(x);  // Menger sponge
    interpret m as MoveForward;
    interpret + as Yaw(angle);
    interpret - as Yaw(-angleComp);
    rewrite F(x)
        with left = x * sinAngle, right = x * sinAngleComp
        to E(x) [ + m(left / 2) F(right) ] - m(right / 2) F(left);
}

lsystem MengerSponge(size = 1) extends StdLsystem3D {
    let iters = if(size > 50, 2, if(size > 10, 1, 0));
    let cubeSize = size * (1/3)^iters;
    let renderBlooms = iters == 0;
    // add iteration to render blooms
    let iters = iters + if(renderBlooms, 1, 0);
    set iterations = iters;
    set symbols axiom = F;
    interpret F as DrawForward(cubeSize, cubeSize, #EEEEEE);
    interpret f as MoveForward(cubeSize / 2);
    interpret B as lsystem Bloom(cubeSize);  // renderes bloom
    rewrite F where renderBlooms to F [ ^ f B ];
    rewrite F to - f f + & f f ^ F F F +f+f- F F +f+f- F F +f+f- F
        -f+f+f^f F F &f&f^ F F &f&f^ F ^ ^ f f f & + f F F &f&f^ F
        ^ ^ f f f & + f F F &f&f^ F ^ ^ f f f & + f F f & f f ^ +
        + f f - f f f f f;
    rewrite f to f f f;
}

lsystem Bloom(size = 1) extends Polygons {
    let color = #d649ff;
    let leafCount = floor(random(4, 7));
    let angle = 150 / leafCount;
    set iterations = leafCount;
    set symbols axiom = F [ G(size/8) K ] leaf;
    interpret F as DrawForward(size * 0.5, size * 0.2, color);
    interpret G as MoveForward(size * 0.5);
    interpret K as DrawSphere(size / 6, #FFFF00);
    interpret + as Yaw(angle);
    interpret - as Yaw(-angle);
    interpret / as Roll;
    interpret ^ as Pitch(-15);
    rewrite leaf to /(360 / leafCount) [ ^(90) <(color) .
        + ^ G . - ^ G . - ^ G . + +(180) + G . - ^ G .  > ] leaf;
}

process HybridPythagorasTree with ThreeJsRenderer;
```

Source code 2.7: Source code of L-system (Fig. 2.15f) demonstrating use of an interpreting symbol as another L-system

2.3.4 Final component system

The final component system uses all the described functionality. The component graph is shown in Figure 2.16. Two main *process configurations* defined in the standard library use this scheme, namely the *SvgRenderer* and the *ThreeJsRenderer* (see appendix I.4).

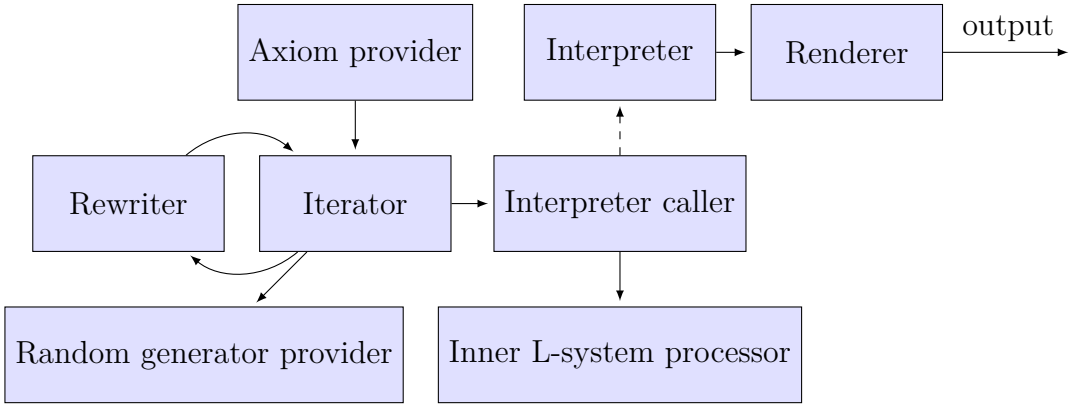


Figure 2.16: Final component system

2.4 Web user interface

The user interface is a very important part of the whole project. Two basic forms of the user interface were considered: desktop application and web site. The web was chosen because for the following reasons.

Accessibility The Web is accessible on a wide range of operating systems where desktop applications cannot be ported easily. Besides the usual desktop systems it is possible to browse it on mobile devices such as smart phones or tablets.

No installation The end-user does not have ti install anything: the application does not depend on the user's OS. The solution is not easy to setup because it has many dependencies to third-party libraries. The Web application is installed by experienced an administrator and everything is then set up properly.

Community Users can share and discuss their work at the same place where they create it. This helps to create a community which is important to all projects.

Up to date The web user interface is always up to date. All updates are instantly applied and available to all users. Errors can be logged and the administrator can fix them as soon as possible.

The web user interface also serves as a comprehensive example of the L-system processing library and its use and capabilities. The Sunflower model in Figure 2.17 was produced by the web site and because of its shape, which fits into a rectangle and its good recognizability even as a 32×32 pixels image, it was chosen as the logo of the web page.

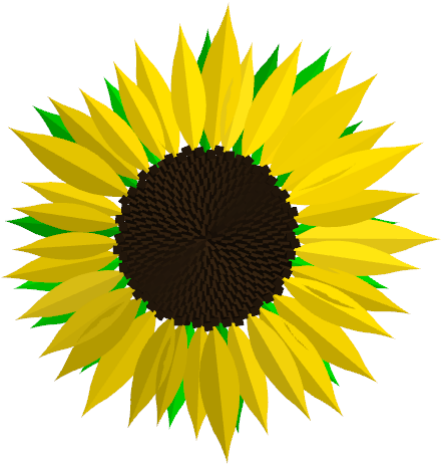


Figure 2.17: Logo of the web

The web page has four main parts. The first three parts, namely the *L-system processor*, *Gallery of L-systems* and *Help* are accessible to everyone. The fourth part is the *Administration* and it is accessible only to administrators.

2.4.1 L-system processor

The main functionality of the web is the processing of a user's input (source code) and showing the results. For this purpose there is a web page with a big text area where the source code can be written. There are three possibilities how to submit the source code.

The first is processing the source code and showing all the results (or list of errors). If there are too many outputs they are packed into one ZIP file. All the results can be downloaded.

The second possibility is to just compile the source code and see the compiled source code (no results are shown). This is intended for the debugging of errors in the input.

The last possibility, which is only available for registered users, is to save the source code. To be able to save the source code successfully it must be without compilation errors. For each saved source code a unique identifier is generated and with it is possible to access the saved input (by a permanent link). Saved inputs can be published in gallery.

2.4.2 Gallery of L-systems

The gallery will serve as a showcase of the capabilities of L-systems for new users as well as learning material. All entries in the gallery will have their source code included and anybody can try to process and customize it. Registered users can rate other gallery entries.

Every registered user may contribute to the gallery with their own creation. To enter an L-system into the gallery a user has to save and publish the source code via the L-system processor. It is possible to alter the thumbnail L-system over the original L-system. This allows the simplification of images into thumbnails and show complex models in the detail view.

Tags can be assigned to each L-system in the gallery. A tag is a short keyword, term or abbreviation which helps to describe the L-system and allows it to be found again. A list of all tags can be listed and a list of L-systems filtered by a specific tag can be shown. A tag can contain a short description of its meaning. The description can be edited only by a special user group.

L-systems can also be filtered by user name.

2.4.3 Help

An important part of the web is the help section. Help contains a few basic topics and FAQs (frequently asked questions) for new users. Then there is list of predefined components, process configurations, constants, functions and operators. The last part of the help is a detailed syntax reference.

2.4.4 Administration

The administration section of the web is accessible to a restricted group of users. There is more than one administrators group, every one with different privileges.

The main administrators group is able to manage roles for all users, manage user groups (roles) and explore error logs.

The next group is able to explore all processed inputs on the site, see all saved inputs and export the input database to a text file.

The last group can see a list of the submitted feedbacks and if a new feedback is submitted all users from this group will receive it via e-mail.

2.4.5 Database

The database will serve for saving all necessary data. Figures 2.18 and 2.19 shows the database scheme (*PK* after name means primary key and *FK* foreign key).

In the left part of the scheme shown in Figure 2.18 are the tables *User* and *Roles* with the relation n to n (any user can be in any number of roles). Both

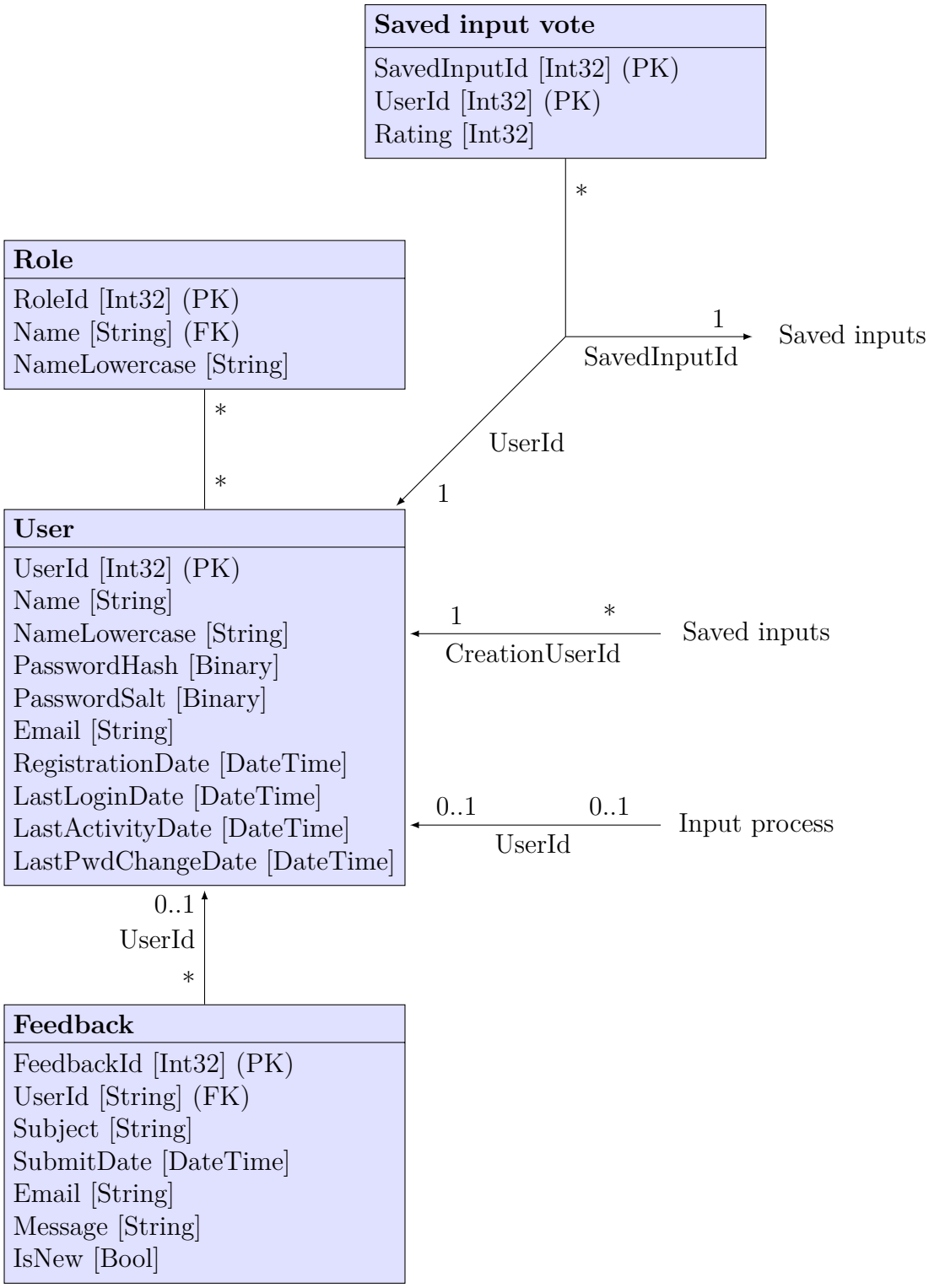


Figure 2.18: First half of the database scheme of the web

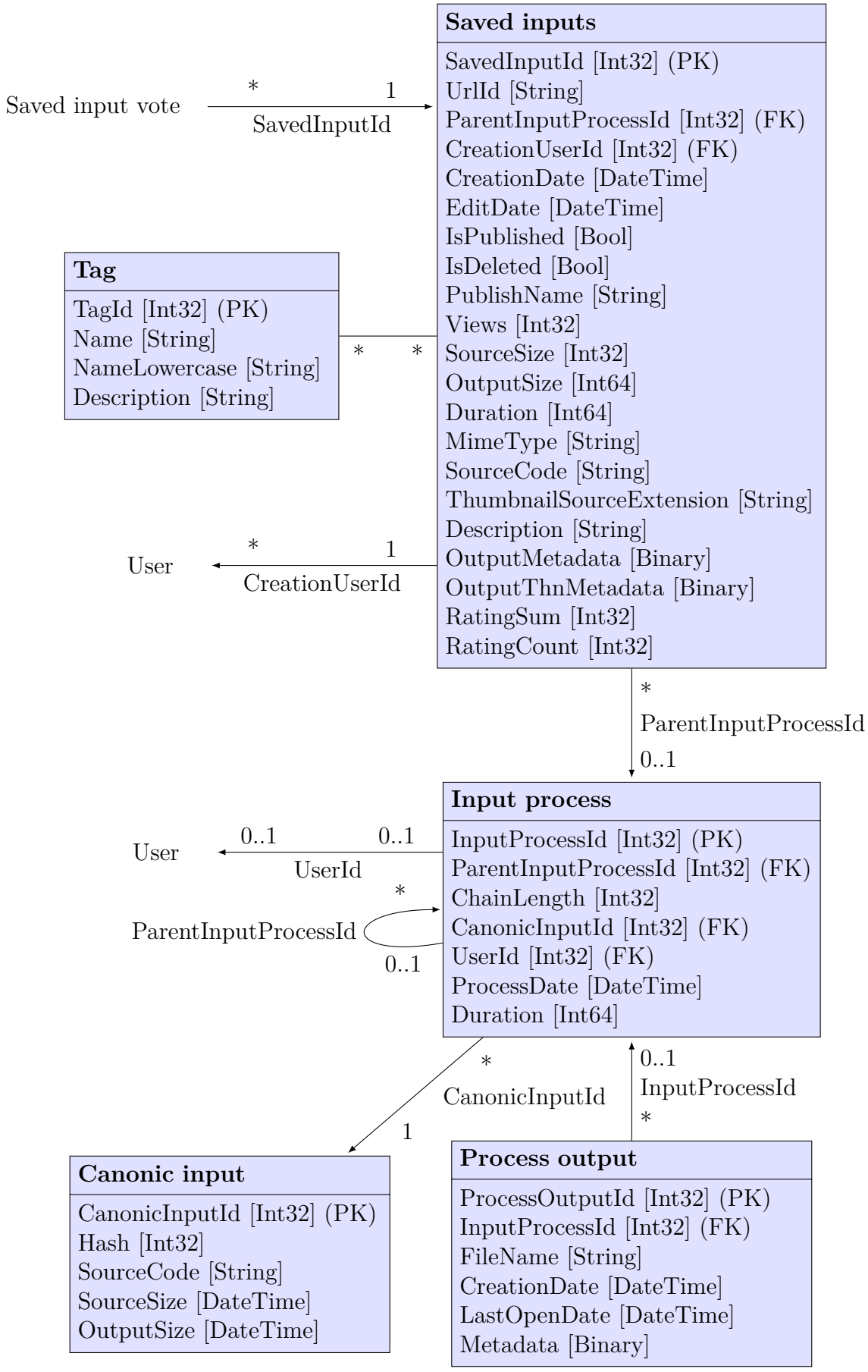


Figure 2.19: Second half of the database scheme of the web

tables table contains column called *NameLowercase* for canonical representation of the user names for easier searching. The *Feedback* table for saving posted feedbacks have a foreign key to *Users* (if a registered user submits a feedback).

The right part of the scheme shown in Figure 2.19 is more complicated. Every processed input is saved to the *Input process* table. To optimize the size of the database the source code is not saved for every input process but it is canonicalized and saved to the *Canonic input* table. The hash is counted for every saved canonical input to speed up lookup for identical inputs. This system ensures that two identical source codes will not be saved in the database. One might thing that the probability of processing two identical source codes is very low but it is not true. The most users trying to process the L-systems from the gallery and do minor changes to them like changing the number of iterations.

The results of processing (like images) are not saved directly into the database because may be relatively large. The result are saved to the hard disk to the *working directory* (which can be configured). Each produced file is saved into the *Process output* table to ensure effective cleaning of the physical files in the working directory.

LastOpenDate entry (in the *Process output* table) is updated every time the user views processed file. If the number of stored files exceeds the maximum (which can be configured) the files with the longest time before last opening are deleted. This mechanism allows to keep alive "old but viewed" files with no need for saving them permanently (for example for sharing with a friend).

Moreover, the new files are saved with the *LastOpenDate* lowered by one minute over the *CreationDate*. This will cause that deletion of non-viewed files is likely than the viewed ones. It can protect wiping all files by some bot that will process many inputs but do not open them.

Lets go back to the saving of all processed inputs to the *Input process* table. The creation of an L-system is iterative process. At the beginning is simple L-system which is gradually improved by the user. To keep track over this iterative process the "parent" input processes is saved for each processed input (if any exists).

The processes forms chains. The longer the chain of processes is, the better L-system can be expected. The length of the chain can be counted by searching the database an resolving the *ParentInputProcessId* column. However this process can take very long time because the *Input process* table will likely have many rows. To be possible to easily find the longest chain the chain length is counted for each row (column *ChainLength*).

If new input is about to save to the *Input process* table and it do not have the parent (for example the first process after opening the page), the *Canonic input* table is searched for corresponding input. If the canonical input it is found, the oldest[4] corresponding input process is selected as the parent.

Saved inputs

Registered users can save their inputs. Inputs are saved to the *Saved inputs* table which also serves as table for the gallery. For every saved input is generated

[4]More input process entries can share one canonical input entry.

unique ID stored in the *UrlId* column. The ID is used in the permanent link which allows permanent access to all saved inputs.

The saved inputs can be edited by the owner but, more importantly, they can be published to the gallery. The inputs in the gallery can be rated. Ratings are stored in the *Saved input vote* table. The primary key to this table is a pair of the *SavedInputId* and *UserId* allowing each user to vote for every input just once (of course the the vote can be changed later).

The published entries in the *Saved inputs* table are sorted by average rating taking into account total number of the votes (the more votes, the better). To speed up the sorting and eliminate joining with the *Saved input vote* table, the sum and the count of votes is stored directly in the *Saved inputs* table.

The source code of saved inputs is saved as is (without any canonicalization) to preserve the comments and formatting. The last output of processed L-system is saved as the result (image or thumbnail) which allows to generate the thumbnails effectively by adding the thumbnail source extension (the *ThumbnailSourceExtension* column) to the end of the actual source code (the *SourceCode* column). In the thumbnail source extension can be used the process statement to generate thumbnail easily.

3. Implementation

In this chapter are described implementation details of the project. Sections in this chapters explains individual problems and their solutions. The text accompanies actual source code snippets and diagrams for better explanation.

Initially, the project was named as *Malsys* which stands for *Mark's L-systems* and this name preserved till now.

3.1 Solution structure

The solution is divided into 6 projects: the L-system processing library (Malsys), the web user interface (Malsys.Web), the abstract syntax tree (Malsys.Ast), the syntax parser (Malsys.Parsing), the common functionality (Malsys.Common) and the project with tests (Malsys.Tests).

The main reason why the solution do not contain lower amount of projects is because the syntax parser is written in the F# which is language from .NET family as well as C# but it is not possible to compile the F# and C# into single DLL. The abstract syntax tree (AST) is separated from the parser because the AST will be compiled by the compilers written in the C# and it is desirable to have the AST data structures written in the C#. It is also more comfortable to design the AST classes in the C# because the F# is functional language and the syntax of classes definition is quite complex. The common functionality is separated into single project because it will be needed in all projects and the solution can not have circular dependencies of projects. The Web project is separated from main project intentionally to allow usage of the L-system processing library independently. And finally the test project is separated to be possible to test all projects independently. The dependencies of projects in solution are shown in the Figure 3.1. The *Malsys.Tests* project has dependencies to all other projects.

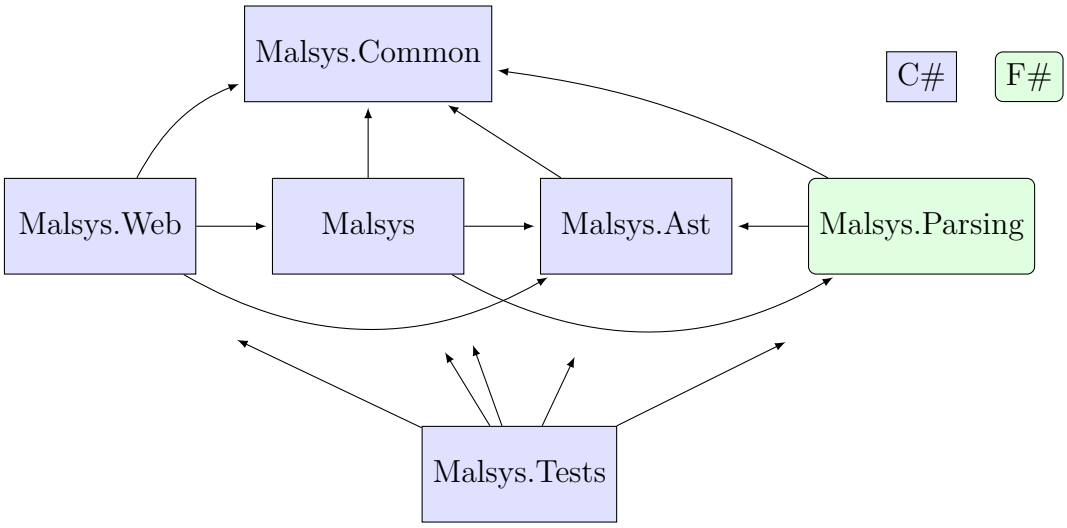


Figure 3.1: The dependencies of projects in the solution

3.2 Input parsing

Input parsing is in the *Malsys.parsing* project and it have two phases: lexing and parsing. The lexing phase uses the lexer to convert the input source code to a stream of *tokens* (basic blocks of input). The lexer is generated by the *FsLex* tool [G.1]. Rules for the *FsLex* are written using regular expressions and the F# code. Source code 3.1 shows an example of the lexer definition of the *FsLex* tool. Full definition is in the file *Lexer.fsl* in the *Malsys.parsing* project.

```
let whitespace = [' ' '\t']
let digit = '\Nd'  // unicode group for digits
// uppercase, lowercase, titlecase, modifier, other, number (letter)
let letter = '\Lu' | '\Ll' | '\Lt' | '\Lm' | '\Lo' | '\Nl'
// punctuation (connector), nonspacing, spacing, other (format)
let specialChar = '\Pc' | '\Mn' | '\Mc' | '\Cf'
let idFirstChar = letter | '_'
let idChar = letter | specialChar | digit | ['\'']
let id = idFirstChar idChar*

rule tokenize args = parse
    | whitespace { tokenize args lexbuf }  // ignore whitespaces
    | id { match keywords.TryFind(lexeme lexbuf) with
        | Some(token) -> token  // keyword
        | None -> ID(lexeme lexbuf) }  // identifier
    | digit+ { parseInt args lexbuf ConstantFormat.Float }
    ...
```

Source code 3.1: Example of the definition file for the *FsLex* tool

The next phase is called parsing. The parser is generated by the *FsYacc* tool [G.1] from a definition file (*Parser.fsy* in the *Malsys.parsing* project). The parser is written to parse the input to the abstract syntax tree (defined in *Malsys.ast*). All data structures in the AST are immutable[1] which helps to make the project more robust. It is impossible to change a value of some AST node by mistake but immutable data structures will not allow this. Source code 3.1 shows an example of the parser definition.

Both, the *FsLex* and the *FsYacc* tools are run automatically on the build of the project in the Visual studio.

```
// constant definition
ConstDef:
    | LET Id EQUALS Expression SEMI
      { new ConstantDefinition($2, $4, getPos parseState) }
 // function definition
FunDef:
    | FUN Id OptParamsParens FunBody
      { new FunctionDefinition($2, $3, $4, getPos parseState) }
FunBody:
    | LBRACE FunStatementsList RBRACE
      { new ImmutableListPos<IFunctionStatement>(
            $2, getPos parseState) }
FunStatementsList:
    |
      { new ResizeArray<IFunctionStatement>() }
    | FunStatementsList FunStatement  { $1.Add($2); $1 }
```

[1] Immutable data structures can not be changed after its creation.

```
FunStatement:
    | ConstDef
      { $1 :> IFunctionStatement }
    | RETURN Expression SEMI
      { $2 :> IFunctionStatement }
// identifier
Id:
    | ID
      { new Identifier($1, getPos parseState) }
```

Source code 3.2: Example of definition file for *FsYacc*

3.3 Compilation and evaluation

The compilation of the abstract syntax tree is done by a set of compilers defined in the *Compilers* namespace in the *Malsys* project. There are defined specialized compilers for each part of the AST. The compilers uses each other to compile the AST. For example, the *Input compiler* uses the *L-system compiler* which uses the *Constant definition compiler*, etc. Figure 3.2 shows hierarchy of the compilers. To keep the figure clear, arrows to the *Expression compiler* was shortened (every compiler uses the *Expression compiler*).

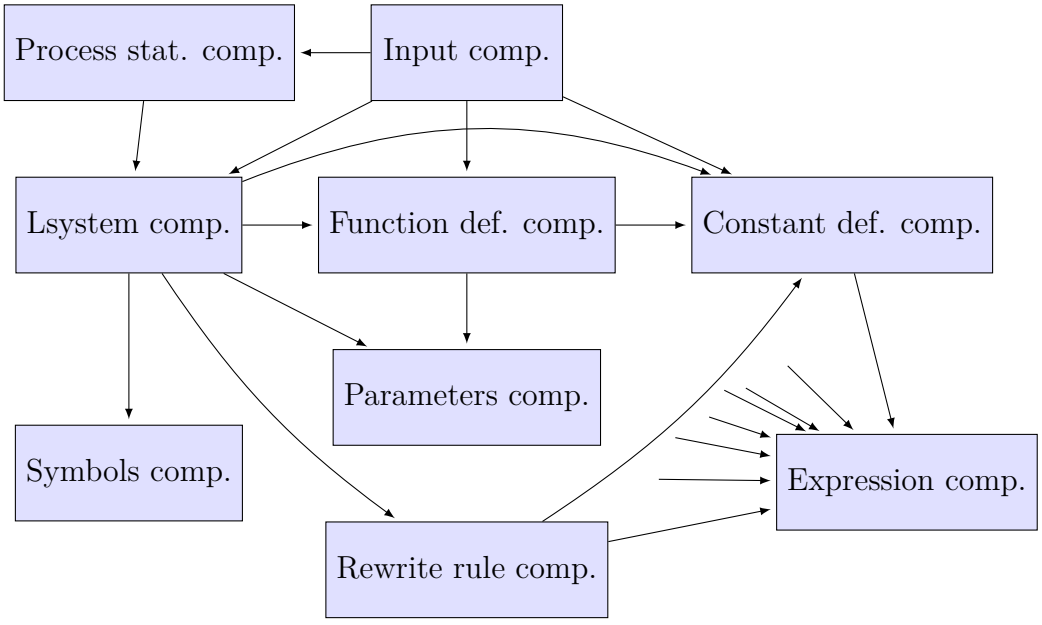


Figure 3.2: Hierarchy of the compilers

The compilers are not bound to each other statically. All compilers implements some general interface and they requires other compilers through that interfaces (Source code 3.3). Each compiler takes all dependent compilers as parameters of the constructor.

```
// general interface for simplifying definition of compilers interfaces
public interface ICompiler<TSource, TResult> {
    TResult Compile(TSource obj, IMessageLogger logger);
}

// constant definition compiler compiles Ast.ConstantDefinition to ConstantDefinition
public interface IConstantDefinitionCompiler
    : ICompiler<Ast.ConstantDefinition, ConstantDefinition> {}

// concrete implementation of constant definition compiler interface
public class ConstantDefCompiler : IConstantDefinitionCompiler {
        // constructor
        public ConstantDefCompiler(IExpressionCompiler expressionCompiler) { ... }
        // compile method
        public ConstantDefinition Compile(Ast.ConstantDefinition constDefAst,
            IMessageLogger logger) { ... }
}
```

Source code 3.3: General interface for the compilers, interface for the constant definition compiler and its implementation

An inversion of control (IoC) container is used to instantiate all compilers [G.7]. All types of compilers are registered to the IoC container. Then it is possible to resolve instances of compilers and all theirs dependencies are resolved by the IoC container. This approach also brings great simplicity and extensibility to the solution. Source code 3.4 shows the implementation of the compilers container and its possible usage.

```
public class CompilersContainer : ICompilersContainer {
    protected IContainer container;  // the IoC container

    public CompilersContainer() {
        var builder = new ContainerBuilder();
        builder.RegisterType<InputCompiler>().As<IInputCompiler>().SingleInstance();
        ...  // registration of all other compilers
        container = builder.Build();
    }

    public T Resolve<T>() {
        return container.Resolve<T>();
    }
}

// possible usage
var inputContainer = new CompilersContainer().Resolve<IInputCompiler>();
```

Source code 3.4: General interface for compilers and interface for the expression compiler

The result of compilation is a semantic tree (ST). The semantic tree is as well as the AST immutable.

All compilation errors are logged with the *IMessageLogger* class. No exceptions are thrown which helps the performance and error recovery (compiling can continue even after non fatal errors).

Evaluation

Evaluation of the semantic tree is implemented in the same way as the compilation. There is set of evaluators and an IoC container that links them together.

3.4 Components members

A component is the .NET class. All components must implement the *IComponent* interface (Source code 3.5) and they must have a parameter-less constructor.

```
public interface IComponent {
    IMessageLogger Logger { set; }
    void Initialize(ProcessContext context);
    void Cleanup();
}
```

Source code 3.5: Interface of the *ProcessManager* class

According to section 2.2.5, any component can have: settable properties, settable symbol properties, gettable properties, connection properties, callable functions and interpretation methods. All listed members are marked with special attributes to be possible to distinguish from other class properties and methods. The access name of all members is the same as their real name, however, the *AccessName* attribute can be used to change it.

Component lifetime

The *IComponent* have two basic methods: the *Initialize* and the *Cleanup*. Following list shows the order of individual operations during creation of the component graph.

- instantiation (using required parameter-less constructor),
- set of *Logger* property
- for each processed L-system:
  - reset (cleanup) with the *Cleanup* method, it should be used for setting the component to a default state
  - connecting of other components (setting the connection properties)
  - setting of the settable (symbol) properties
  - initialization with the *Initialize* method
  - processing of the L-system
- cleanup with *Cleanup* method.

Settable properties

The settable properties (and the settable symbol properties) are actual properties of the component class marked with the *UserSettable* (*UserSettableSybols*) attribute. The properties must have a public setter (a getter is not required). The settable properties must have their type assignable to the *IValue* which covers both base types, the numbers (*Constant* type) and the arrays (*ValuesArray* type). The settable symbol properties must have *ImmutableList<Symbol<IValue>>* type.

By default, all settable (symbol) properties are ”optional” which means that their value may not be set. By setting the *IsMandatory* property of the *UserSettable* (*UserSettableSybols*) attribute to true, the property is marked as mandatory and an error will be thrown if no value is set to it.

The setter of the settable (symbol) properties can throw the *InvalidUserValueException* if the supplied value is invalid. The text of the exception will be shown as an error to the user.

Gettable properties

Similarly as the settable properties, the gettable properties are actual properties of the component class marked with the *UserGettable* attribute. The properties must have a public getter (a setter is not required) and their type must assignable to the *IValue* type.

By default, values of the gettable properties can be get after the initialization of the component. In that time all the statements from processed L-system are already evaluated, thus the value of the gettable properties can not be used in them. However, it is possible to set the *IsGettableBeforeInitialiation* property of the *UserGettable* attribute to allow getting oft the property value before the initialization and to use the value in the L-system statements.

Connection properties

The connection properties are for connecting other components. They are actual properties of the component class marked with the *UserConnectable* attribute. Properties must have a public setter (a getter is not required) and their type must be assignable to the *IComponent* type.

Connection of some component to the connection properties is, by default, mandatory but it is possible to set the *IsOptional* property of the *UserConnectable* attribute to allow leaving the property unconnected (*null*). By default, only one component can be assigned (connected) to each connection property but this can be changed by the *AllowMultiple* property of the *UserConnectable* attribute.

The setter of connection property can throw the *InvalidConnectedComponentException* if supplied value of a component is invalid. The text of the exception will be shown as an error to the user.

Callable functions

Callable functions serves to allow calling of component methods in L-system code (in statements, rewrite rules etc.). Callable functions are method of the component class marked with the *UserCallableFunction* attribute. The underlying method must have two parameters parameter of types *IValue[]* and *IExpressionEvaluatorContext*. The return type of the method must be assignable to the *IValue* type.

Similarly as gettable properties callable functions can be by default called after the initialization of the component but it is possible to set the *IsCallableBeforeInitialiation* property of the *UserCallableFunction* attribute to allow calling the function before the initialization.

Interpretation methods

The interpretation methods are specialized methods used for interpretation of L-system symbols. The interpretation methods must be marked with the *Sym-*

bolInterpretation attribute, they must have a parameter of type *ArgsStorage* and *void* return type.

3.4.1 Documentation of members

To simplify the documentation of a component and their members the standard *XmlDoc* with custom elements is is used for documentation. The documentation is loaded automatically if the XML file with the documentation is included with the DLL where the component is located.

Standard *summary* tag is used to document the component. The *user readable name* for the component can be written in the *name* tag and for the name of *group* to which the component belongs is the tag called *group*. Example of the documentation of a component is in Source code 3.6.

```
/// <summary>
/// Provides symbol property called Axiom which serve as
/// initial string of symbols of L-system.
/// </summary>
/// <name>Axiom provider</name>
/// <group>Common</group>
public class AxiomProvider : SymbolProvider {
    ...
}
```

Source code 3.6: Example of usage the XmlDoc for documentation of a component

Settable (symbol) properties

Basic documentation is of a settable property is loaded from the *summary* tag. Expected value description can be put into the *expected* tag and for the default value of a settable property is the *default* tag.

For settable symbol properties is loaded only the *summary* tag

Gettable and connection properties

Documentation for gettable and connection properties is loaded from standard *summary* tag.

Callable functions and interpretation methods

In addition to standard *summary* tag for general documentation, callable functions (and interpretation methods) can be documented with the *parameters* tag where on each line should be documentation for each parameter.

3.4.2 Example

The example of implementation and the documentation of a component is in appendix D.

3.5 Input processing

Component graph creation is the core of the L-system processing library. The class responsible for that is called the *ProcessManager* (Source code 3.7).

It uses the class *ProcessConfigurationBuilder* for creation and configuration of the component graph. For resolving component types the *ProcessManager* uses the *IComponentMetadataResolver*. It works in the similar way as the IoC container but the components do not have any dependencies. The components are registered by the user directly using the *IComponentMetadataResolver* implementation or with the helper class called *MalsysLoader*.

```
public class ProcessManager {

    public ProcessManager(ICompilersContainer compIoc,
        IEvaluatorsContainer evalIoc, IComponentMetadataResolver compResolver) {

    public InputBlockEvaled CompileAndEvaluateInput(string sourceCode,
        string sourcName, IMessageLogger logger) { ... }

    public void ProcessInput(InputBlockEvaled inBlock, IOutputProvider outProvider,
        IMessageLogger logger, TimeSpan timeout) { ... }
}
```

Source code 3.7: Interface of the *ProcessManager* class

The *ProcessManager* does following steps in the method *ProcessInput*. Notice the similarities with the compoennt lifetime (section 3.4).

For each process statement in the input do:

1. instantiate all components in the Process configuration (specified by the process statement),
2. save the gettable properties and the callable functions from all components which can be get and called before the initialization,
3. for each L-system (specified by the process statement) do:
   (a) reset (clean) all components,
   (b) connect all components to the graph (resetting may disconnect them),
   (c) evaluate the L-system,the gettable properties and the callable functions from step 2 can be used to evaluate the L-system statements,
   (d) evaluate additional L-system statements from the process statement,
   (e) set the settable properties of all components (from evaluated L-system statements),
   (f) save rest of the gettable properties and the callable functions from all components,
   (g) initialize all components,
   (h) find the starter component and start it,
4. reset (clean) all components.

3.6 Immutable data structures as scoped storage

In many places in the L-system processing library there is need for temporary addition of constants to a collection. As an example can be evaluation of a function. The parameters of the function are added as constants to the a collection but this addition can overlay already defined constants. After the evaluation, the collection should be in the same state as before (parameters and local variables must not remain). This problem can be solved by complex mutable data structures or by defensive copying of the collection which can lead to serious performance issues (there can be hundreds of defined constants).

Immutable data structures offers clean solution to this problem. An immutable data structure is data structure which can not be changed (muted) such as the *string* in the C#. They can not be changed, thus there is no need for creation of defensive copies. If a method passes an immutable collection of defined constants to the function evaluation method the collection can not be changed by it.

But how is possible to add new items to immutable data structures if they can not be changed? It is not possible because they are immutable but it is possible to create a new copy with added item. Interestingly enough, addition to an immutable tree has the same asymptotic complexity as insertion to a mutable one.

Each node of the immutable tree is immutable. If we want to add new node we will find the place in the tree where it should be. Now we need to connect it to the parent but the parent is immutable so we will create new parent (copying value from old one) but with new connections. Old children (if any) can be connected without re-creating them. So we need to re-create only nodes on the path to the root so complexity is logarithmic ($log_2(n)$ new nodes). Insertion to the immutable tree is illustrated in Figure 3.3.

There is no need for implementation of immutable tree because the F# contains it as the *FSharpMap<TKey, TValue>* class and the *MapModule* class provides methods for the work with it.

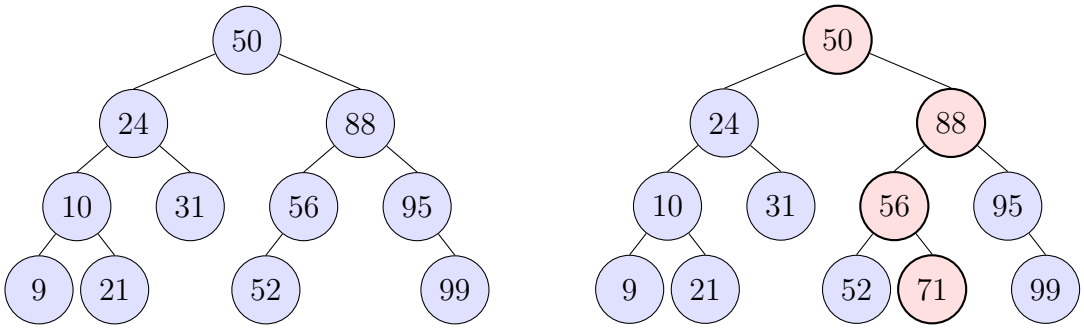
Described system ensures that global variables will be accessible and can be temporarily overlayed by more local definitions. It is used in may places such as evaluation of the L-systems (local constants and functions overlays global ones), evaluation of user-defined functions (parameters and local variables) and the rewrite rules (parameters of symbols are mapped as local variables).

3.7 Implemented components

There are many implemented components in the L-system processing library. In this section is described implementation details of some of them.

3.7.1 Symbol rewriter

The core of the L-system processing is the rewriting of L-system symbols. The *SymbolRewriter* component is responsible just for that. Its implementation supports all L-system types described in section 1.2.



(a) Original tree (b) Tree with inserted node 71

Figure 3.3: Example of the add operation to the immutable tree, red nodes are newly created

The most complicated of the *SymbolRewriter* component is the context checking (in context rewrite rules, see section 1.2.4). The context checking can not be implemented by naive approach which will search the context of symbol one by one because branches must be skipped and they can be very long. Source code 3.8 demonstrates an L-system where the naive implementation will not be able to work well. It uses symbol `S` to generate random color index as its parameter. Symbol `S` is "linked" using the context to every symbol `X` which is rewritten to new branches with color index from the base symbol `S`. This causes that all lines in one iteration have the same color but colors between iterations differs. First five iterations with the random seed set to 0 are shown in Figure 3.4.

```
lsystem LongContext extends Branches {
    set symbols axiom = S(0) X;
    set symbols contextIgnore = + - F;
    set iterations = 5;
    set initialAngle = 90;
    interpret F(c) as DrawForward(32, 2, c * #001100);
    interpret + as TurnLeft(20);
    interpret - as TurnLeft(-20);
    rewrite {S(c)} X to [ + F(c) X ] - F(c) X;
    rewrite S to S(floor(random(0,16)));
}
process all with SvgRenderer;
```

Source code 3.8: L-system

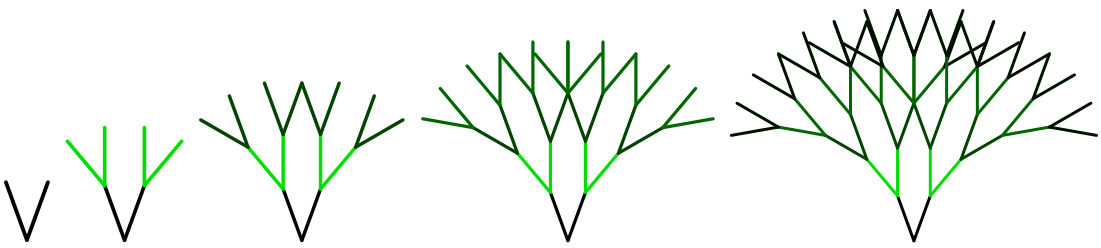


Figure 3.4: Result of L-system in Source code 3.8

In the following list are strings of symbols of the first five iterations of the L-system in Source code 3.8. Ignored symbols `-` and `F` are omitted. Every symbol

`X` needs to search for the very first symbol to match the context correctly. Now you can see why the naive implementation can not be used.

1. S(0) X
2. S(11) [X] X
3. S(13) [[X] X] [X] X
4. S(12) [[[X] X] [X] X] [[X] X] [X] X
5. S(8) [[[[X] X] [X] X] [[X] X] [X] X] [[[X] X] [X] X] [[X] X] [X] X

The *SymbolRewriter* component builds a tree from the symbols which allows to skip the branches quickly. The nodes of the tree at the same level are interlinked to allow search for the left and the right context effectively. The pointer to the current symbol is updated after each processed symbol. The tree is dynamically loaded as the right context needs more symbols. The tree node is represented by the *ContextListNode<T>* class. The *ContextListBuilder* class helps to build the tree correctly symbol by symbol.

The search tries all possible ways to match the context but number of possible ways is relatively low. Figure 3.5 shows an example of the context search in the tree built from the fourth iteration of the L-system in Source code 3.8. The red path shows search for the left context `S` of the symbol `X` and the green path shows the search for the right context `[X]` of the symbol `X`. The search methods are located in the *ContextChecker* class.

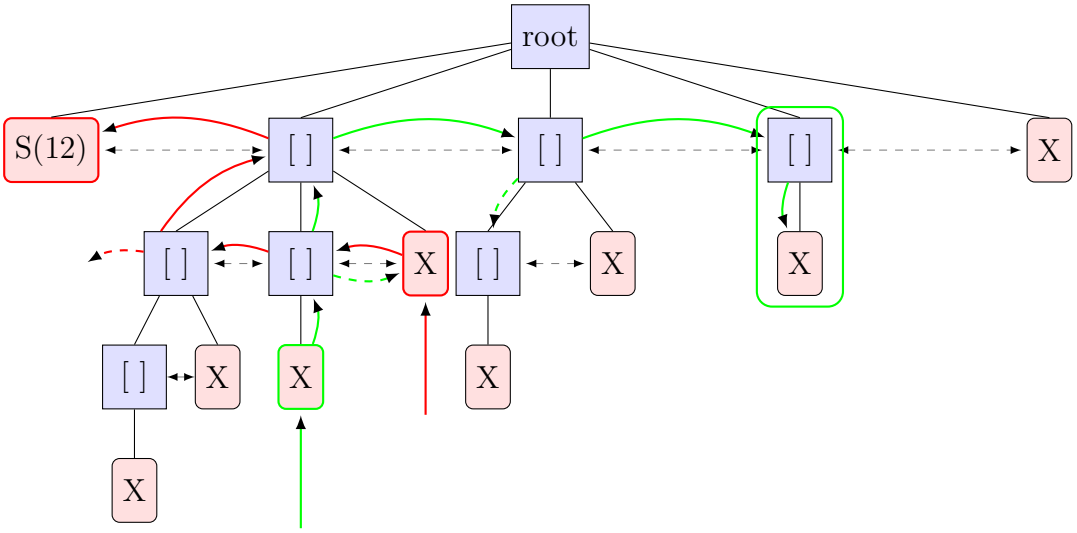


Figure 3.5: The context search tree built from the fourth iteration of the L-system in Source code 3.8

As a context matching method finds matching symbol it also maps its parameters as new constants. Because the context matching is done by the back-track and the mapped symbols can be invalid, immutable data structures are used for saving the mapped constants. Because of immutable data structures the roll-back to old values can be done with no cost (see section 3.6). Described algorithm is not optimal but it is sufficient for the most cases. Production L-system have rarely context longer than one.

The context checking is tested by many unit tests in the *Malsys.Tests* project which ensures correctness of implemented algorithm.

3.7.2 Turtle graphics interpreter

A turtle graphics interpreter is universal interpreter for both 2D and 3D renderers and it is implemented by the class *TurtleInterpreter*. It interprets L-system symbols as a *turtle graphics* in 3D but if 2D renderer is connected it omits the Z coordinate. This allows to use 3D commands even if 2D rendered is connected which brings many advantages. For example, the *Roll* (a rotation by the direction axis) by 180° can be used for the inversion of turning directions in 2D.

The interpreter allows to use three basic rotations: *pitch*, *yaw* and *roll*. The pitch operation turns "up" around right-hand vector, the yaw turns "left" around up vector axis and the roll operation rolls clock-wise around forward vector axis. The up, right and forward vectors can be set by the user using the settable properties. This allows some changes to the coordinate system for example, the forward and right vectors may not right-angled.

The rotation of the turtle is represented as the *Quaternion*. This has many advantages over "traditional" representation with a rotation matrix. Quaternions represent a rotation around some axis which is exactly what the basic operations do. In the contrast to representation by the rotation matrix, the quaternion allows to do pitch, yaw and roll by axes which are relative to current orientation easily. Also the storage size of the rotation represented by the quaternion is the smallest possible since the quaternion is represented as four numbers. Composition of two quaternion rotations is equal to multiplying the quaternions. Source code 3.9 shows a code snippet of the *Yaw* interpretation method of the *TurtleInterpreter*.

```
[SymbolInterpretation(1)]
public void Yaw(ArgsStorage args) {
    double angle = getArgumentAsDouble(args, 0);
    currState.Rotation *= new Quaternion(upVect, angle);
}
```

Source code 3.9: Implementation of the *Yaw* method of the *TurtleInterpreter*

The *TurtleInterpreter* can simulate the tropism[2]. This is easily done by quaternions too because the tropism is simulated as a physical force which is applied to the line as you can see in Source code 3.10 [PL91, p. 58].

```
private void rotateByTropism(Vector3D moveVector) {
    Vector3D axis = Vector3D.CrossProduct(moveVector, tropismVect);
    double angle = axis.Length * tropismCoef;
    if (angle.EpsilonCompareTo(0) == 0) {
        return;
    }
    axis.Normalize();
    currState.Rotation = new Quaternion(axis, angle) * currState.Rotation;
}
```

Source code 3.10: Implementation of the tropism which is applied after each line

[2]A tropism is a biological phenomenon, indicating growth or turning movement of a biological organism, usually a plant, in response to an environmental stimulus (`http://en.wikipedia.org/wiki/Tropism`).

3.8 Triangulation of 3D polygons

Turtle graphics interpretation of the L-systems can generate polygons which are objects in the space that are defined the points on its. These objects are regular polygons in 2D nad they are is easy to render even if its shape is complex (crossing edges, etc.). But in 3D the situation is much more complex. Lets call the object specified by points on its perimeter *3D polygon* even if it is not 100% technically correct. The 3D polygon is specified by the points on its perimeter but they does not say anything about the shape in the middle.

The easiest way how to render general shapes in 3D is to compose them from triangles. The conversion process from 3D polygon to triangles is called triangulation. There is more possible triangulations even of basic 3D polygons (see the situation for four points is in Figure 3.6).

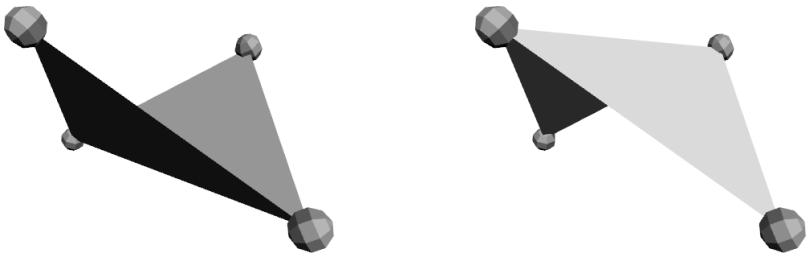


Figure 3.6: Ambiguous triangulation of four points

Because of the ambiguity in the triangulation there can not exist an ideal triangulation algorithm. With this in the mind was written the trianguler in the L-system processing library. It is possible to configure triangulation strategies.

The triangulation algorithm works on the *cutting-ear* base. From each point of the 3D polygon can be created a triangle (called *ear*) by connecting the point to its two neighbors. Then one ear is picked (by strategy which is given by the user) and *cut off* which means that the triangle is sent to the output. Remains a polygon with one less point and the cutting is repeated until only 3 points remains.

The only thing which can be affect by user is the order of the cutting but it is quite enough. This algorithm is implemented in the *Polygon3DTrianguler* class and the triangulation method takes as an argument (besides actual points) *triangulation parameters* which specifies: *a*) the evaluation delegate which is used for evaluation of ears, *a*) the ordering of ears' scores (ascending or descending) which specifies whether minimal or maximal score is the best, *a*) the recount mode which can be set to *never*, *neighbors* or *all*; it specifies what ears are reevaluated after the cutting, and *a*) attached multiplier, after the cutting the evaluation score of neighboring ears are multiplied with it.

The algorithm also have support for detection of planar polygons because the turtle graphic tends to produce planar polygons in 3D space. It tries to find a plane where the polygon is planar by finding some plane and counting the coefficient of variation of distance from the plane to all points (the ratio of the standard deviation σ to the mean μ). If the coefficient is under the threshold (given by the user) the polygon is projected on found plane and standard Delaunay triangulation algorithm is used for robust triangulation.

The triangulation algorithm is very versatile but to simplify its usage there is defined 5 strategies.

As in input – cuts ear one by one from the first to the last point without any sorting

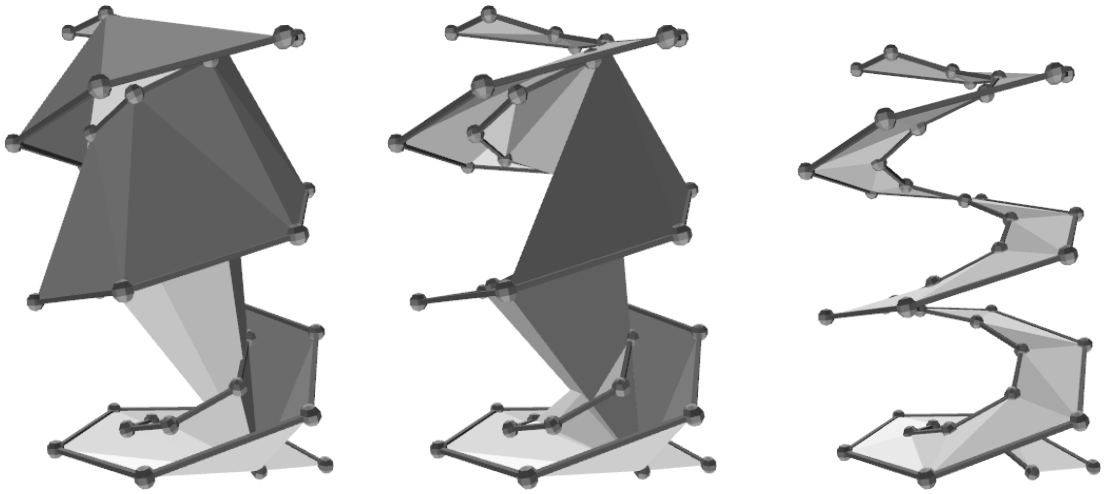
Minimal angle – triangles with minimal angles are triangulated first

Maximal angle – triangles with maximal angles are triangulated first

Maximal distance – triangles with maximal distance from all points are triangulated first

Maximal distance from non-triangulated – triangles with maximal distance from non-triangulated points are triangulated first

The user can pick one of the strategies that gives the best results to their polygon type. The choice can be done by setting the *polygonTriangulationStrategy* property of the *ThreeJsSceneRenderer3D* component. The standard library contains constants which can be used instead of "magic" numbers to keep the code readable (appendix I.2.2). Figure 3.7 shows complex 3D polygon triangulated with three different strategies.



(a) Maximal angle (b) Minimal angle (c) Maximal distance from non-triangulated

Figure 3.7: Complex 3D polygon triangulated with three different strategies

3.9 Web user interface

As an web framework is used the ASP.NET MVC 3. It works on the model-view-controller (MVC) design pattern.

Controller translates user input into operations on the model and view,

Model represents the application logic and data structures,

View generates output to the users (Figure 3.8).

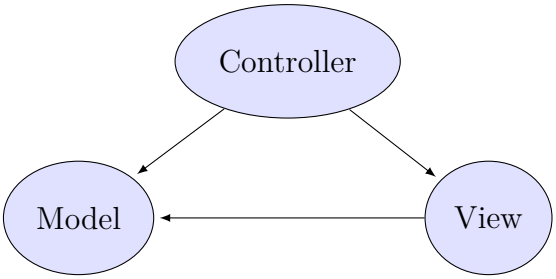


Figure 3.8: MVC design pattern

The model in our case is the L-system processing library and the database access layer. Views are written in the Razor view engine which allows to mix the HTML and C# code, thus to generate pages effectively. The controllers are classes and theirs methods represent the actions. The routing engine of the MVC 3 automatically translates HTTP requests to the actions by given rules. Example of routes definition is in Source code 3.11. The first rule called *Permalink* translates the URLs in the format of "permalink/id" to the call of the *Index* method of the *Permalink* controller (class). The second rule is called *default*. It is used in the most cases to translate URLs like "http://malsys.cz/Gallery/Detail/7qe7iF9P" to the call of the *Detail* method of the *Gallery* controller with the *id* parameter set to *7qe7iF9P* (Source code 3.12). The view is called at the end of the action method (Source code 3.13).

```
routes.MapRoute("Permalink",
    "permalink/{id}",
    new { controller = "Permalink", action = "Index" }
);

routes.MapRoute("Default",
     "{controller}/{action}/{id}",
     new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Source code 3.11: Example of routes definition

```
public class GalleryController : Controller {
    ...
    public ActionResult Detail(string id) {
        var model = malsysInputRepository.InputDb.SavedInputs
            .Where(input => input.UrlId == id && !input.IsDeleted)
            .SingleOrDefault();
        ...
        return View(model);
    }
    ...
}
```

Source code 3.12: The *Detail* method of the *Gallery* controller

```
@model InputDetail

<div class="right">permalink: @Html.InputPermaLink(Model.Input.UrlId)</div>
<h2>@Model.Input.PublishName</h2>
...
@if (Model.Input.Tags.Count > 0) {
    <h3>Tags</h3>
    foreach (var tag in Model.Input.Tags) {
        @Html.Tag(tag.Name)
    }
}
...
```

Source code 3.13: Gallery *detail* view demonstrating the Razor syntax

3.9.1 Data annotations

Data annotations are used for automatic generation and validation of HTML forms on both, client and server sides. For advanced data annotations are used the Data Annotations Extensions [G.11]. Source code 3.14 shows a model class for new user with the data annotations as attributes, highlighted code in Source code 3.15 will render the form shown in Figure 3.9.

```
public class NewUserModel {
    [Required]
    [Display(Name = "User name")]
    [StringLength(64, MinimumLength=4)]
    public string UserName { get; set; }

    [Required]
    [Email]
    [Display(Name = "E–mail address")]
    public string Email { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.",
        MinimumLength = 8)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password",
        ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

Source code 3.14: Model class with data annotations

```
@using (Html.BeginForm()) {
    <fieldset>
        <legend>Account Information</legend>
        @Html.EditorForModel()
        <br />
        @ReCaptcha.GetHtml(theme: "clean")
        <p><input type="submit" value="Register" /></p>
    </fieldset>
}
```

Source code 3.15: Part of user registration view

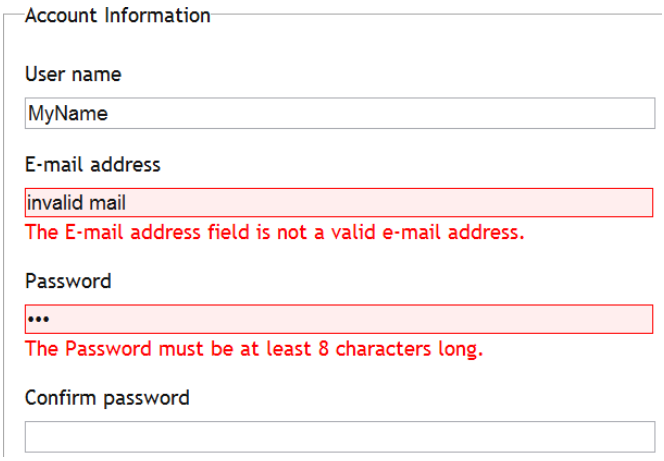


Figure 3.9: Rendered registration form with incorrectly entered e-mail address and too short password

3.9.2 Easy configurability

All the important settings are configurable in the *Web.config* file. The file is well commented to allow simple changing of the settings.

Note that the *Web.config* file have release transformation which is changing values for deploy.

Process times

It is easy to create an L-system which takes an eternity to process it thus, it is important to limit the processing time, especially for non-registered users. There are more settings for different user roles. Unregistered users have by default only 2 seconds. Registered users have 5 seconds and users in the *Trusted users* rule 10 seconds. Process time for gallery entries is separated because gallery image can be generated by any request (even by non-registered user). Example of the process times settings are shown in Source code 3.16.

```
<appSettings>
    <add key="ProcessTime_Unregistered" value="2" />
    <add key="ProcessTime_Registered" value="5" />
    <add key="ProcessTime_Trusted" value="10" />
    <add key="ProcessTime_Gallery" value="8" />
</appSettings>
```

Source code 3.16: Process time settings in the *Web.config*

Working directories

The outputs of processed L-systems are saved as files. The web application needs some working directories for saving them. A working directory for the L-system processing is separated from the gallery's. These directories can be cleared by the administrator at any time because files in the processing working directory are temporary and files in the working directory for the gallery are re-created when they are needed. This also allows easy migrating of the web site.

The maximum number of files in the processing work directory can be also set. Source code 3.17 shows the settings of the working directories and their limits.

```
<appSettings>
    <add key="WorkDir" value="~/WorkDir" />
    <add key="GalleryWorkDir" value="~/GalleryWorkDir" />
    <add key="Process_AutoPackTreshold" value="8" />
    <add key="WorkDir_MaxFilesCount" value="4096" />
    <add key="WorkDir_CleanAmount" value="32" />
</appSettings>
```

Source code 3.17: The working directories settings in the *Web.config* file

Additional setting

In *Web.config* file can be set additional setting such as public and private keys for the ReCaptcha [G.13] or the directory for saving error logs (Source code 3.18).

```
<appSettings>
    <add key="ReCaptcha_PublicKey" value="xxx–enterYourPublicKeyHere–xxx" />
    <add key="ReCaptcha_PrivateKey" value="xxx–enterYourProvateKeyHere–xxx" />
</appSettings>
<elmah>
        <errorLog logPath="~/ErrorLogs" ... />
        ...
</elmah>
```

Source code 3.18: Additional setting in the *Web.config* file

3.9.3 Inversion of control

The controllers are dependent on so called dependencies, i.e. the models and other instantiated classes. Every controller can instantiate all dependencies on its own but this approach statically binds the dependencies to the controllers and it is not possible to share the instances of the dependent classes between the controllers. For example, the model for database access should be instantiated only once per an HTTP request and it should be shared between all entities who wants the DB access.

The ASP.NET MVC 3 framework has built-in support for the inversion for control (IoC) container which can resolve the dependencies for the controllers. The big advantage of this approach is that the IoC container can control the lifetime of individual dependent classes. Some can be shared as a single instance between all controllers and some can be shared only within one HTTP request.

Concrete implementations of the dependent classes are ”hidden” under interfaces, thus change of some implementation can be done at one place where dependencies are registered to the IoC container.

As the IoC container is used the Autofac with the ASP.NET MVC 3 integration [G.7]. Source code 3.19 shows registration of the Autofac IoC container as default dependency resolver for the MVC and its configuration.

```
protected void Application_Start() {
    var resolver = buildDependencyResolver();
    DependencyResolver.SetResolver(resolver);
    ...
}

private IDependencyResolver buildDependencyResolver() {
    var builder = new ContainerBuilder();
    // registers all MVC controllers in this assembly
    builder.RegisterControllers(typeof(MvcApplication).Assembly);

    builder.RegisterType<StandardDateTimeProvider>()
        .As<IDateTimeProvider>().SingleInstance();
    builder.RegisterType<Sha512PasswordHasher>()
        .As<IPasswordHasher>().SingleInstance();

    builder.RegisterType<MalsysDb>()
        .As<IUsersDb>()
        .As<IInputDb>()
        .As<IFeedbackDb>()
        .InstancePerHttpRequest();
    ...
    return new AutofacDependencyResolver(builder.Build());
}
```

Source code 3.19: Registration of the dependency container and its configuration

The IoC also allows to test the controllers more easily. Test methods can pass special implementations of dependencies to tested controllers and simulated desired behavior.

For example, instead of static *DateTime* class for getting current date is used the *IDateTimeProvider* which is by default just wrapper around the static *DateTime* class but any test method can pass a special implementation which returns always the same date (for example 29th of February) to test the behavior for that concrete date. Also, simulation of the database is simpler. Source code 3.20 shows the controller which has four dependencies as parameters of the constructor.

```
public class GalleryController : Controller {

    public GalleryController(IMalsysInputRepository malsysInputRepository,
            IAppSettingsProvider appSettingsProvider,
            LsystemProcessor lsystemProcessor,
            IDateTimeProvider dateTimeProvider) {
        ...
    }
    ...
}
```

Source code 3.20: Registration of dependency container and its configuration

3.9.4 Removal of literal strings with the T4MVC

The ASP.NET MVC 3 framework contains many literal strings (also called ”magic” strings). Those strings are used for referring the controllers, actions and views but also the static files. The problem is that these magic strings must exactly match to the names of class members or files. Big problem occurs when those names are changed. The values of literal strings are not checked by the compilation but the run-time error will occur. Also when literal strings have to be written by hand, there is no intelli-sense for them and it is hard to write them correctly in larger application. Also it is easy to misspell a literal string.

The *MvcContrib* project offers the T4 template called *T4MVC* which solves

this problem [G.8]. The T4 template generates hierarchy static classes in the *MVC* and *Links* namespaces. They contain constants for all literal strings. Figure 3.10 shows code snippets with literal strings and in Figure 3.11 are replaced by generated equivalents.

```
public virtual ActionResult Edit(string id, EditSavedInputModel model) {
    ...
    return RedirectToAction("Detail", input.UrlId);
}
```

```
routes.MapRoute("Permalink",
    "permalink/{id}",
    new { controller = "Permalink", action = "Index" }
);
```

```
<p>... is the @Html.ActionLink("gallery", "Index", "Gallery") ...</p>
```

```
@Content.Css("~/Css/style.less.css")
```

```
<div class="logonBox">
    @Html.Partial("~/Views/Shared/LogOnPartial.cshtml")
</div>
```

Figure 3.10: Code snippets showing literal strings in the ASP.NET MVC 3

```
public virtual ActionResult Edit(string id, EditSavedInputModel model) {
    ...
    return RedirectToAction(Actions.Detail(input.UrlId));
}
```

```
routes.MapRoute("Permalink",
    MVC.Permalink.Name.ToLower() + "/{id}",
    new { controller = MVC.Permalink.Name, action = MVC.Permalink.ActionNames.Index }
);
```

```
<p>... is the @Html.ActionLink("gallery", MVC.Gallery.Index()) ...</p>
```

```
@Content.Css(Links.Css.style_less_css)
```

```
<div class="logonBox">
    @Html.Partial(MVC.Shared.Views.LogOnPartial)
</div>
```

Figure 3.11: Code snippets with literal strings replaced by generated equivalents

3.9.5 Generated help pages

The web site contains extensive help which is crucial for processing L-systems. The most of the pages are written by hand but the reference pages which lists all

the defined components, process configurations, defined constants and functions are generated automatically. This is possible because the defined members are documented directly in the code using the XmlDoc (see section 3.4.1).

For example, the help page with all the defined components is generated from all the loaded components. This have an advantage over the static help because the generated help pages describe exactly what can user use. If new component is defined it automatically appears in the help. Source code 3.21 shows two action methods of the *PredefinedController* which lists all the defined functions and components.

```
public class PredefinedController : Controller {
    public ActionResult Functions() {
        return View(expressionEvaluatorContext.GetAllStoredFunctions());
    }
    public ActionResult Components() {
        var components = componentContainer.GetAllRegisteredComponents();
        ...
        return View(components);
    }
    ...
}
```

Source code 3.21: Example of two action methods which lists all the defined functions and components

Note that appendices J and K was also generated by the web with special views which generate LaTeX source. They are accessible on similar URLs as traditional help (only with *Latex suffix*): */Help/Predefined/ComponentsLatex* and */Help/Predefined/ConfigurationsLatex*.

3.9.6 Caching and compression

Caching and compression is important to minimize amount of transferred data and to minimize the load of the server. This causes faster response and shorter loading times for the user.

Server-side caching

Server caches all the pages with static content. This is important because some of the static pages takes relatively long time to generate (for example the generated help).

The static pages are not the same if some user is logged in. In the header of the web page is user's name and other user related buttons. This is why the cache must vary by logged user.

Setting up the caching in the ASP.NET MVC 3 framework is done by marking the action or controller with the *OutputCache* attribute. The cache profile can be specified which is configured in the *Web.config*. Figure 3.12 shows snippets of described caching setup. The last snippet is the implementation of varying the cache by logged user.

Client-side caching

Some static files are not necessary to be transferred more than once to the user. These are for example the CSS definitions, JavaScript scripts and the images.

```
[OutputCache(CacheProfile = "HelpCache")]
public class PredefinedController : Controller { ... }
```

```
<outputCacheProfiles>
    <add name="HelpCache" duration="86400" varyByParam="user" />
</outputCacheProfiles>
```

```
public override string GetVaryByCustomString(HttpContext context, string custom) {
    if (custom == "user") {
        if (context.User.Identity.IsAuthenticated) {
            return context.User.Identity.Name.ToLower();
        }
        else {
            return "";
        }
    }
    return base.GetVaryByCustomString(context, custom);
}
```

Figure 3.12: The controller marked with the cache attribute, configuration of the cache profile and the implementation of varying cache by logged user

Web is configured to send these files with headers which says that client should cache them for up to 30 days. This minimizes number of requests to the server.

This heavy caching is possible because as URL suffix is automatically placed the hash from the static file's date of the last change. If the static file changes its URL will also change and the user will immediately download newer version.

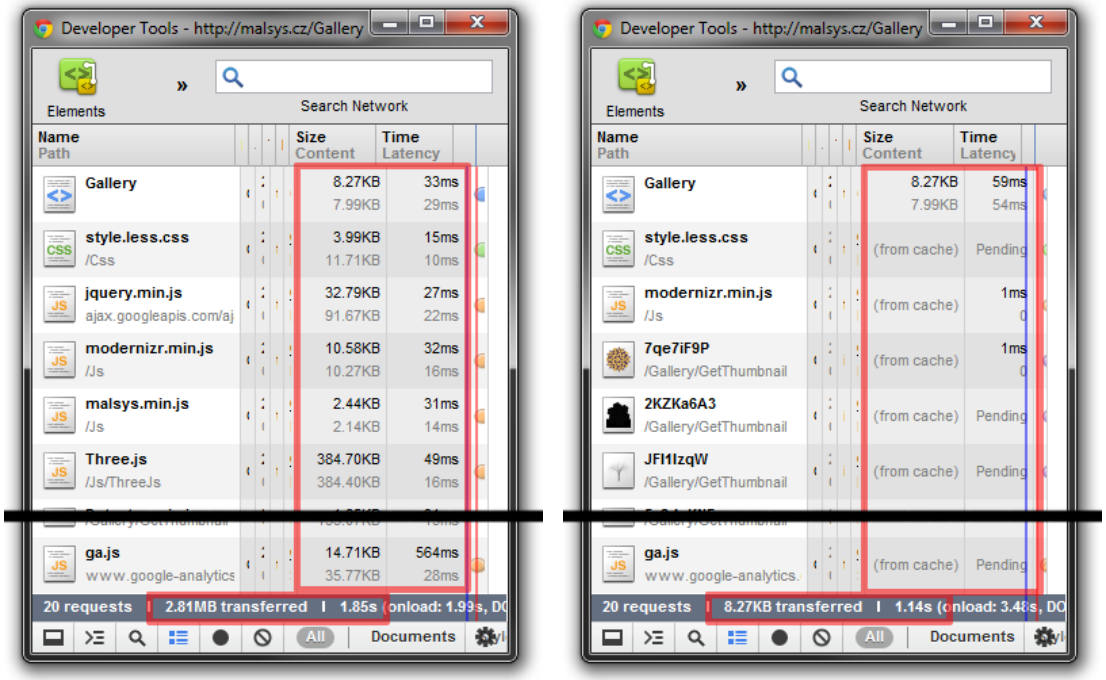
The cache is especially important in the gallery where are many static files. In Figure 3.13 is shown print-screen of the Google Chrome developer tools. On the left image is the first-time loaded page, on the right is showed subsequent loading of the same page. The bottom bar shows total number of downloaded bytes, you can see that for the first time (the user's cache is empty) the page needs 2.8 MB to fully load but with cache it loads only 8.3 KB (that's about 350× less). Also only 1 connection to the server was established instead of 20.

Compression

To minimize amount of data transferred by from the server, static files are automatically compressed by *GZip* compression (Source code 3.22).

```
<httpCompression minFileSizeForComp="1024">
<scheme name="gzip" dll="%Windir%\system32\inetsrv\gzip.dll" />
    <staticTypes>
        <add mimeType="text/*" enabled="true" />
        <add mimeType="message/*" enabled="true" />
        <add mimeType="application/javascript" enabled="true" />
        <add mimeType="*/*" enabled="false" />
    </staticTypes>
</httpCompression>
```

Source code 3.22: Compression part of the *Web.config*



(a) First-time loading

(b) Subsequent loading

Figure 3.13: Google Chrome developer tools showing difference between first and subsequent loading of the page

3.9.7 Error logging

For error logging is used third-party library called *Elmah* [G.9]. The Elmah is configured for automatically logging all unhandled exceptions to the XML files. The error log contains all information needed to trace the exception including the stack trace, url, and all other data about HTTP request like GET and POST data. The Elmah offers an HTTP module accessible on URL the */Elmah.axd* with a list of errors (Figure 3.14). This page is accessible only on the localhost or by users in the Administrators role.

The unhandled exceptions raised by L-system processing library are caught and logged using Elmah manually. This ensures that user do not looses the processed input (because of redirect to the error page in the case of the unhandled exception), just error message is showed.

```
public bool TryProcess(string sourceCode, IMessageLogger logger, ...) {
    try {
        var input = processManager.CompileAndEvaluateInput(sourceCode, ...);
        ...
        processManager.ProcessInput(input, logger, ...);
    }
    catch (Exception ex) {
        ErrorSignal.FromCurrentContext().Raise(ex);  // log exception by Elmah
        logger.LogMessage(Message.ExceptionWhileProcessingInput, ex.GetType().Name);
        return false;
    }
}
```

Source code 3.23: ...

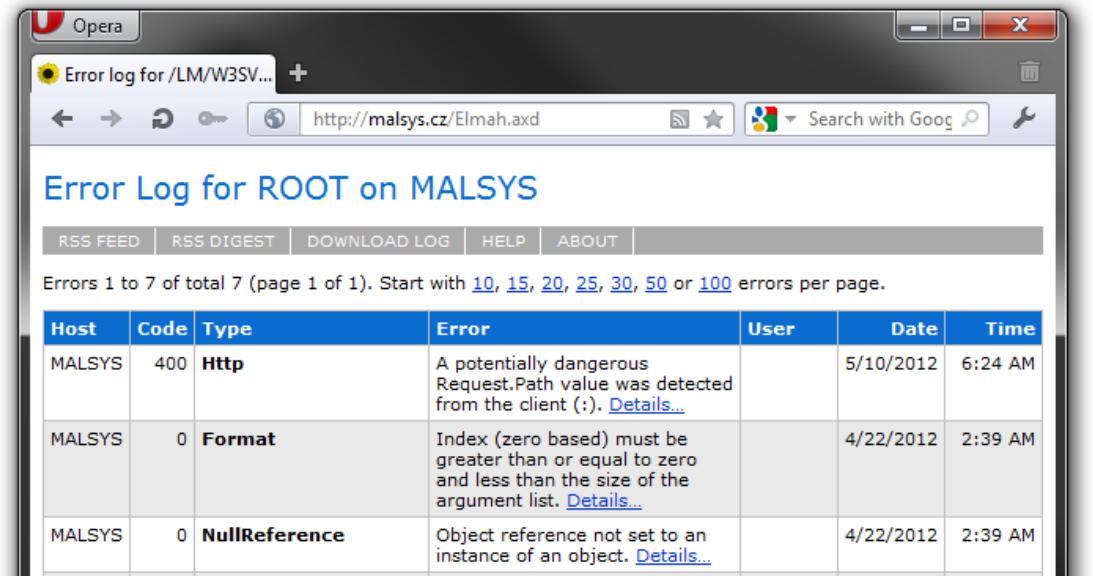


Figure 3.14: Error log provided by Elmah

3.9.8 Cascading style sheets

The *LESS* library [G.10] was used to simplify work with the Cascading style sheets (CSS). The LESS extends the CSS with dynamic behavior such as variables, mixins, operations and functions. This allows to write more simple, maintainable and clear definitions of the CSS.

Source code 3.24 shows the LESS source code and in Source code 3.25 shows the same code compiled to the CSS.

```
@themeColor: #0F4D92;
@baseWidth: 960px;
...
.box-shadow-inset (@x: 0, @y: 0, @blur: 1px, @spread: 0, @color: #000) {
    box-shadow: @arguments inset;
    -moz-box-shadow: @arguments inset;
    -webkit-box-shadow: @arguments inset;
}
...
body { min-width: @baseWidth; ... }
...
#header {
    margin: 0 10px;
    .navigation {
        float: right;
        a {
            font-size: 1.25em;
            &:hover { .box-shadow-inset(0, 0, 8px, 0, #FFF); ... } ...
        } ...
    } ...
}
```

Source code 3.24: Example of the LESS source code

```
body { min-width: 960px; }
#header { margin: 0 10px; }
#header .navigation { float: right; }
#header .navigation a { font-size: 1.25em; }
#header .navigation a:hover {
    box-shadow: 0 0 8px 0 #ffffff inset;
    -moz-box-shadow: 0 0 8px 0 #ffffff inset;
    -webkit-box-shadow: 0 0 8px 0 #ffffff inset;
}
```

Source code 3.25: Compiled LESS code (Source code 3.24) to the CSS

The compilation of the LESS is automatic thanks to the HTTP handler from the library *.LESS* (pronounced dot-less). The handler implicitly compiles the LESS code into the CSS, the configuration is showed in Source code 3.26.

```
<system.web>
    <httpHandlers>
        <add path="*.less.css" verb="GET"
            type="dotless.Core.LessCssHttpHandler, dotless.Core" />
    </httpHandlers>
</system.web>
<dotless minifyCss="true" cache="true" web="false" />
```

Source code 3.26: Configuration of implicit LESS files compilation in *Web.config*

3.9.9 JavaScript

All JavaScript files are minimized with the Yahoo! UI Library [G.12] using a custom T4 template which automates the minimalization process. Thanks to the T4MVC template the original (non-minimized) JavaScript files are used in the debug mode but minimized JavaScript files are used in the release mode.

4. Results

This section summarizes the results which are the L-system processing library and the web user interface. At the end of this chapter are examples of rendered images of L-systems produced by created program.

4.1 L-system processing library

Originally, the aim of this project was to create an online L-system processing interface. However, during the work the L-system processing library showed to be very universal and robust. It is written in the .NET framework and thanks to the Mono project it is multiplatform and it can be used in other projects too.

The library is unique by its component-based processing of L-systems. Components are connected to larger groups which allows to extend the system. The connections are defined by input and can be redefined easily. The component system is even capable to process other things than L-systems but it is limited by input syntax which is specialized for L-systems.

Components can be implemented by users and configured by the input. Example of implementation of a component can be found in appendix D. The library provides a many utilities for component implementation which makes it easier.

The new syntax for L-systems was created and it is relatively simple to read and understand. The syntax parser and compilers are extensible, thus syntax can be improved or extended with minimal effort.

The example of processing of input with the library is in appendix E.

4.1.1 Unit tests

The functionality of the L-system processing library is tested by nearly 200 unit tests. The tested parts are parsing, compilation and evaluation, processing of the L-systems and also individual components like rewriting correctness of the symbol rewriter.

Table 4.1 shows test coverage of the main projects. Note that some parts are very hard to test (for example L-system renderers) thus the are not covered.

Project	Coverage
Malsys	65%
Malsys.Ast	63%
Malsys.Common	42%

Table 4.1: Unit tests code coverage of main projects

4.2 Web user interface

The web user interface was deployed and it is accessible at address `http://malsys.cz`. The main function of the web is to process L-systems. Detailed instructions can be found in appendix C. Web site includes the L-system gallery and the help section.

Any user can register and gain some advantages. Registered users can save and publish their L-systems to the public gallery and they have longer time limit for processing of L-systems.

Web page is displayed correctly in the most common web browsers namely Google Chrome, Opera, Firefox, Internet Explorer and Safari. It is possible to browse it in smart phones or tablets. Figure 4.2 shows print-screens of the first page of the gallery on various operating systems.

Figure 4.1: `http://malsys.cz`

If the browser window is wider than 1900 px the layout of the L-system process page splits into two columns to allow to see the source code and the result simultaneously. This feature is done purely in CSS 3.

The web page is supports pinning to the Windows taskbar (Figure 4.3).

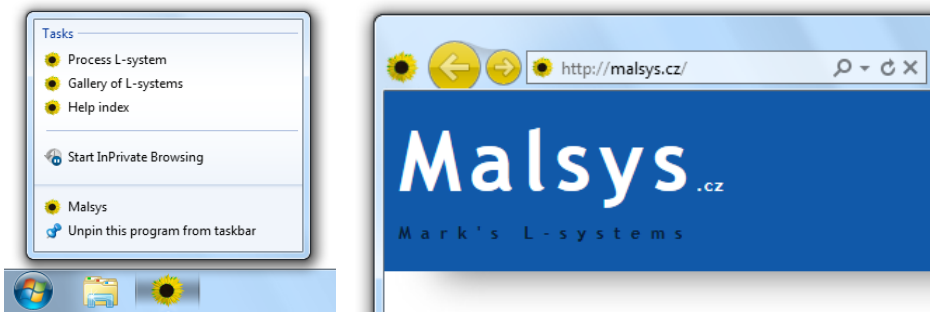
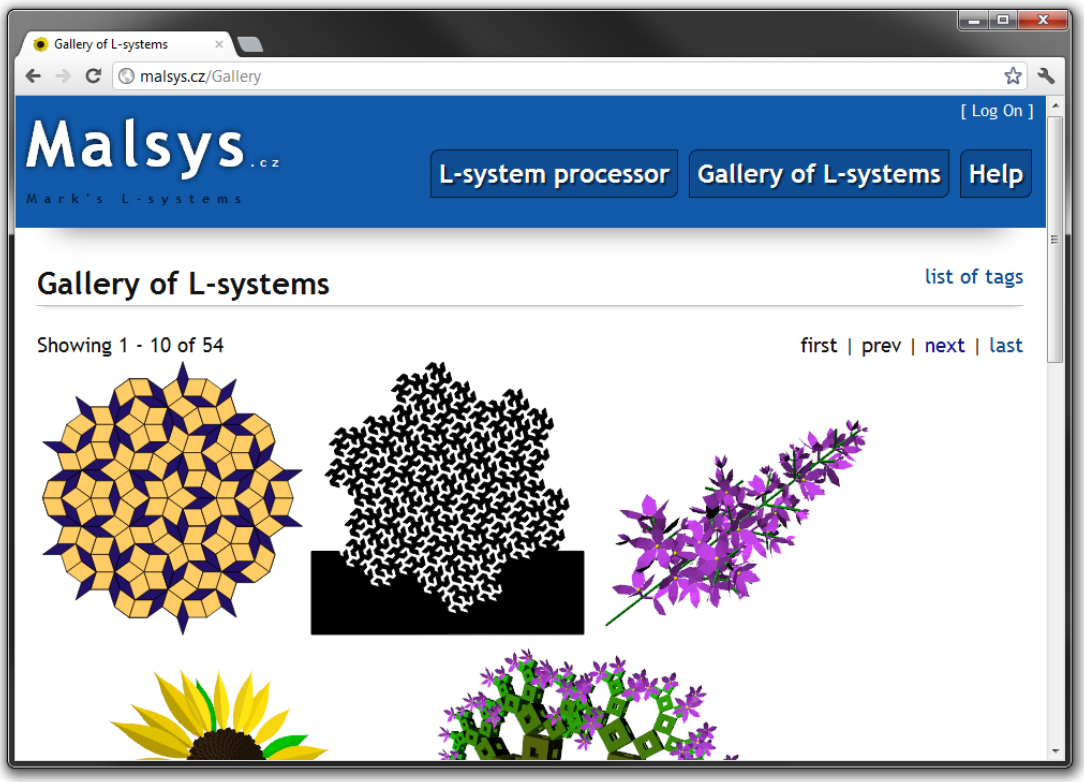


Figure 4.2: Jump-list of pinned site and header of opened Internet Explorer 9 using pinned shortcut

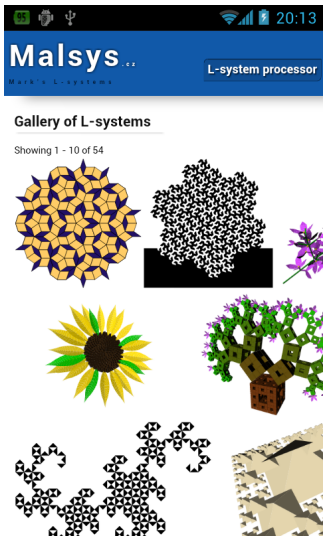
4.2.1 Visitors and traffic

The web was officially released on 15 April 2012. Two days later were posted some notifications on the Twitter and Facebook. That day the number of visitors peeked at 142 but the most of them just checked the gallery and next day the the messages on social networks were lost.

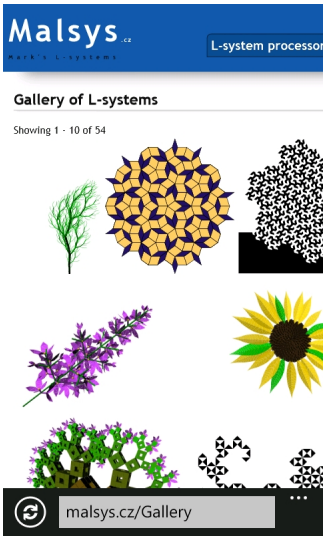
About week after the initial release a short newsflash was posted on the Czech server called `http://root.cz` which attracted 255 visitors that day. But visitors from the root.cz was not just looking in the gallery. In the contrast with the visitors from the social networks, users from the root.cz started to experiment with L-systems. This was probably because of fact that the root.cz is the a site



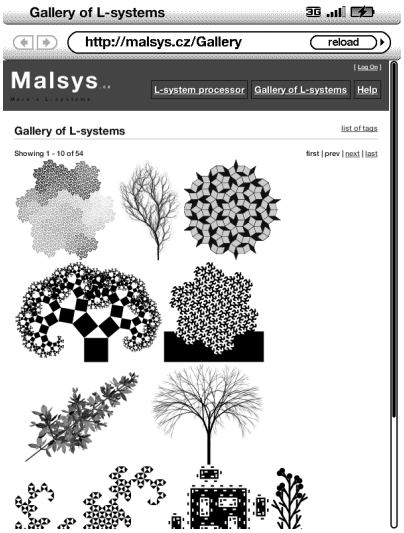
(a) Windows 7 (Google Chrome)



(b) Android ICS (default browser)



(c) Windows Phone (IE9 mobile)



(d) Amazon Kindle 3

Figure 4.3: The first page of the gallery displayed on various operating systems

about computer technologies, software and programming and users understood L-systems better.

At the end of May, one and half months after initial release the malsys.cz was seen by over 1000 unique visitors and they browsed over 9000 pages. Till the end of May unregistered users processed over 2000 L-systems.

4.3 Solution statistics

Table 4.2 shows number of lines of code based on file types. Listed numbers do not include generated code (if not stated otherwise). Also note that help pages with *predefined stuff* in the web are generated dynamically thus their content is not included in statistics of total line count.

Extension	Type	Line count	Comment
.cs	C#	> 30 000	
.fs, .fsy, .fls	F#	> 1 000	F# files together with lexer and parser definitions
.cshtml	Razor	> 10 000	views of the razor view engine
.generated.cs, .designer.cs	C#	> 5 000	automatically generated files

Table 4.2: Number of lines of code written by hand (if not stated otherwise) based on file types

4.4 Showcase of L-systems

The most L-systems are used in this thesis as figures illustrating described themes. In this section are images of some more L-systems.

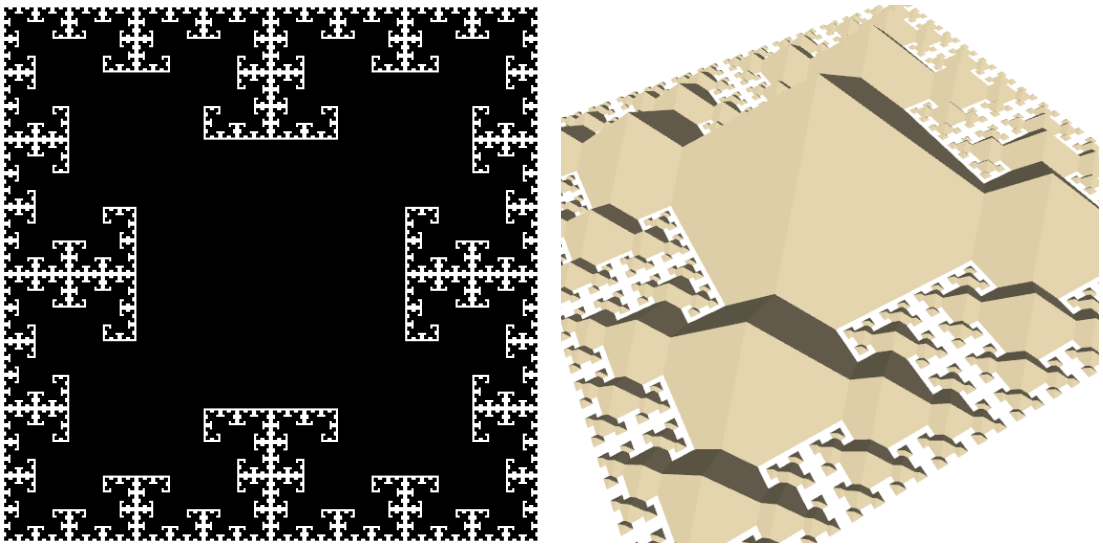


Figure 4.4: T-square fractal (left) and its generalization to 3D with pyramids instead of squares (right)

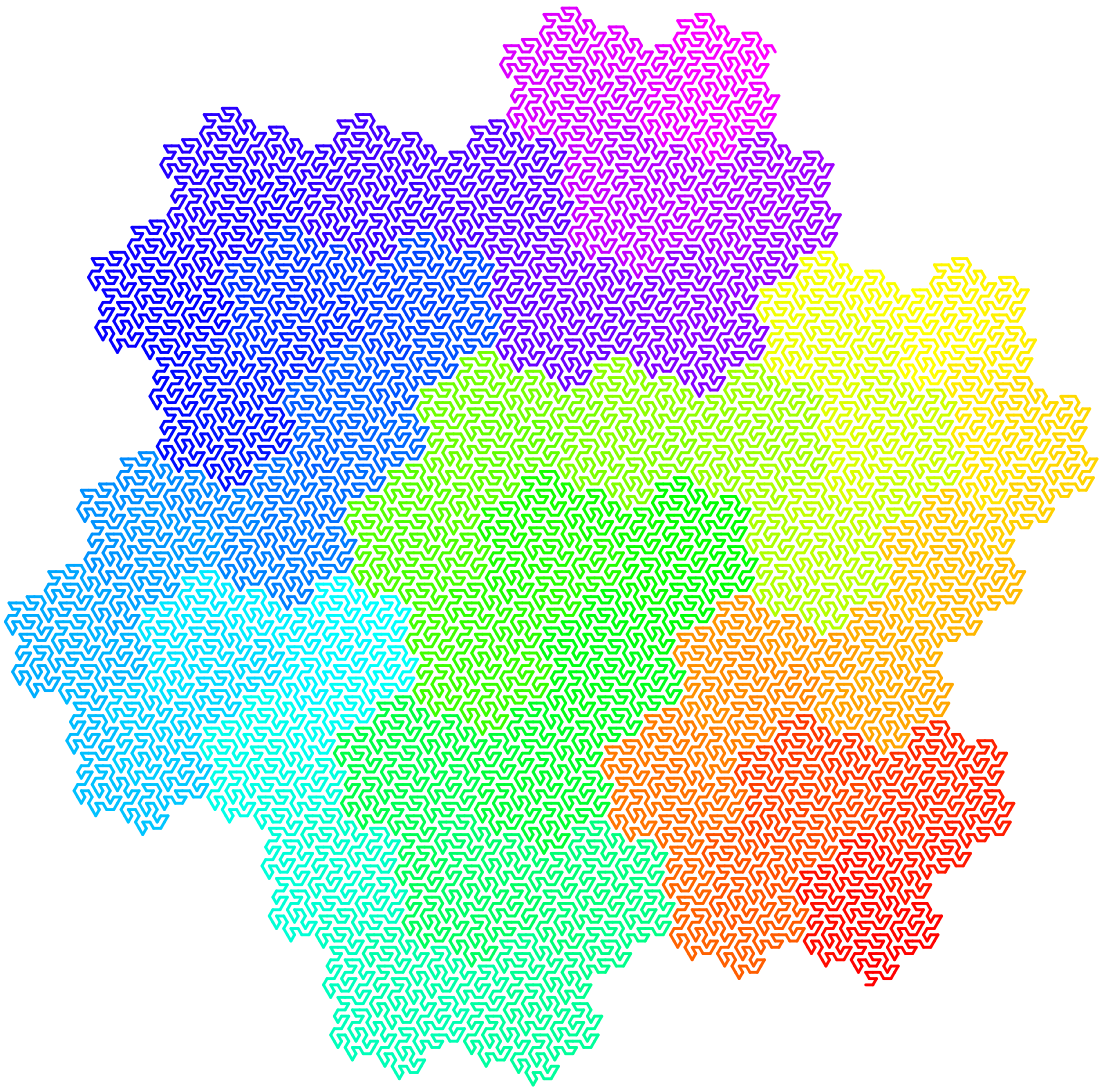


Figure 4.5: Hexagonal Gosper curve

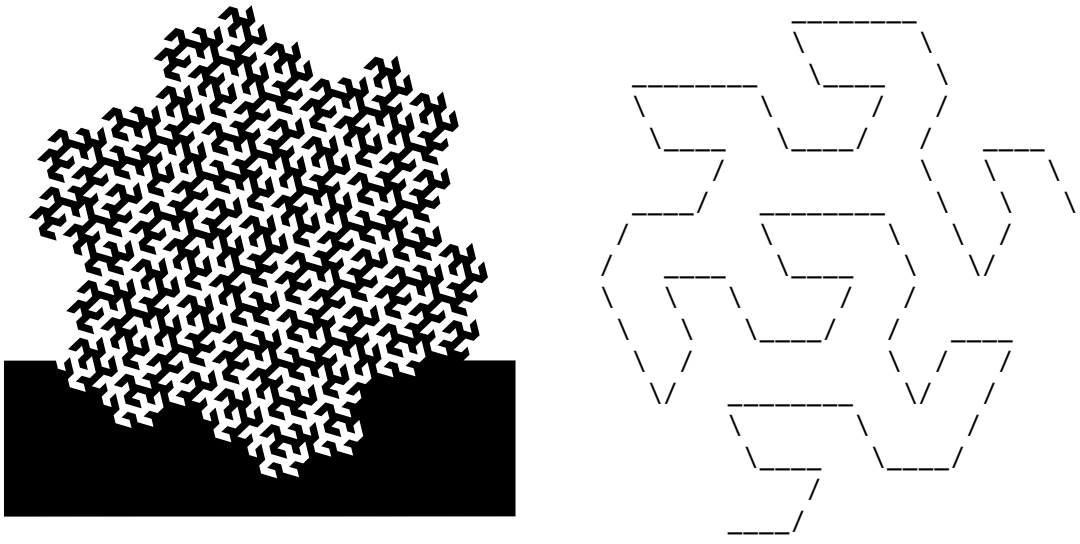


Figure 4.6: Hexagonal Gosper curve as polygon (left) and as ASCII art (right)

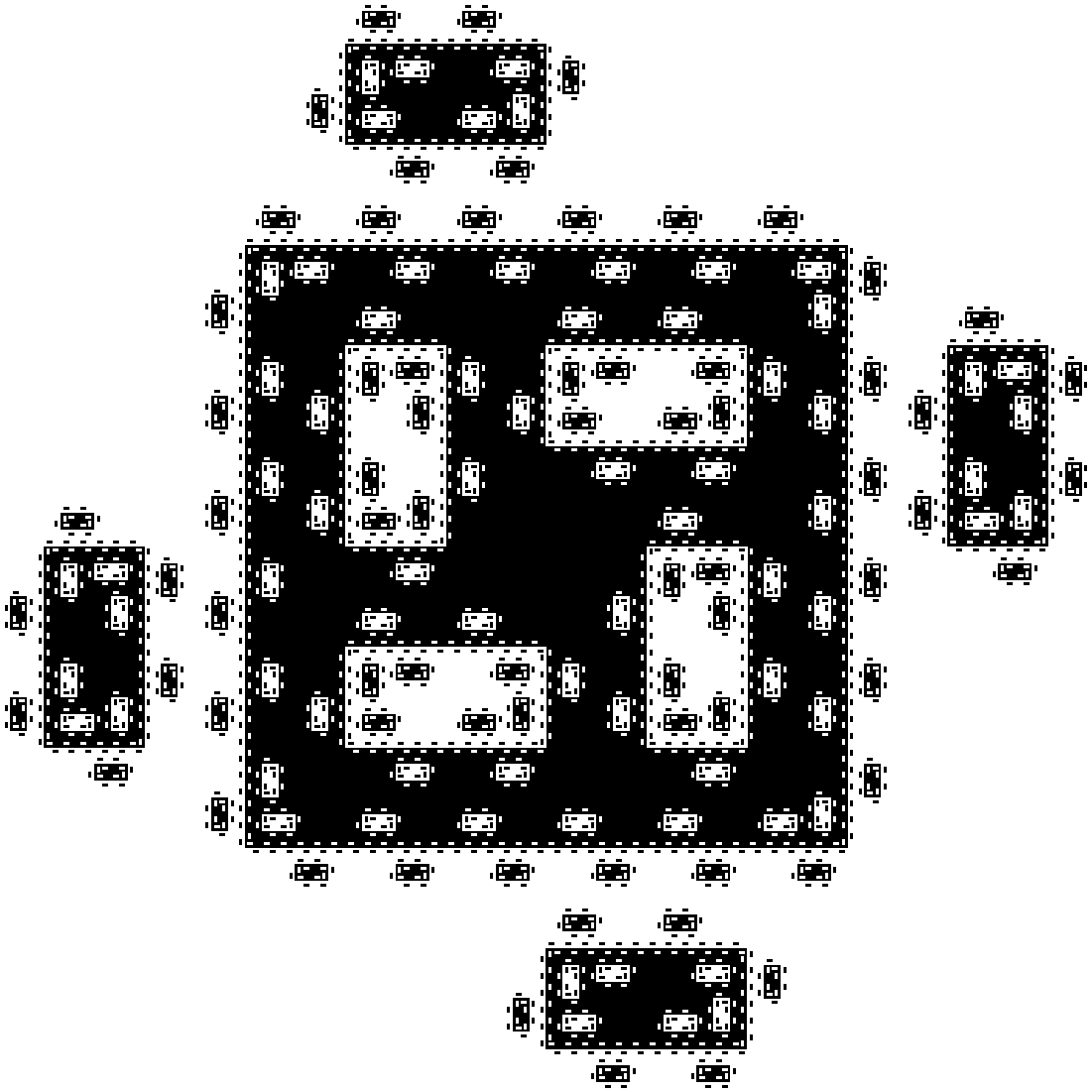


Figure 4.7: Islands and lakes

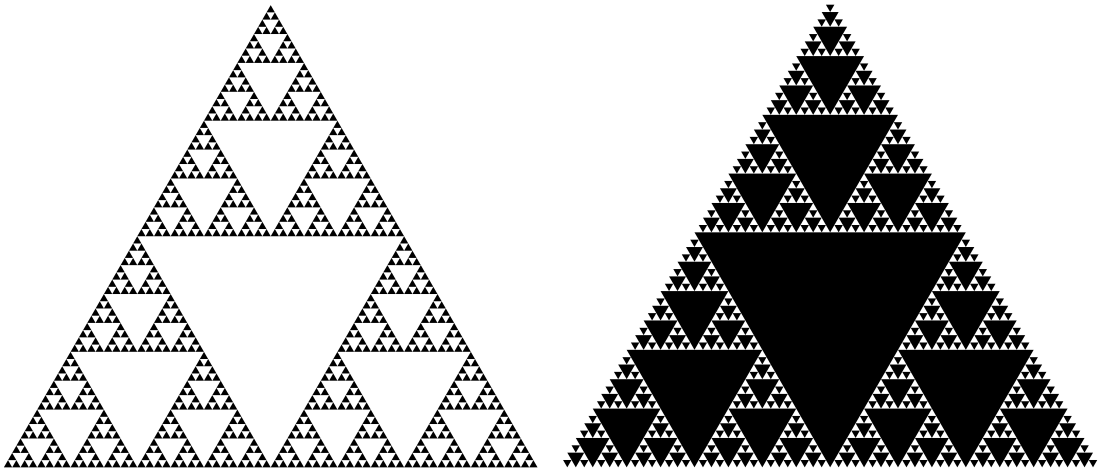


Figure 4.8: Basic (left) and inverted (right) Sierpinski triangles

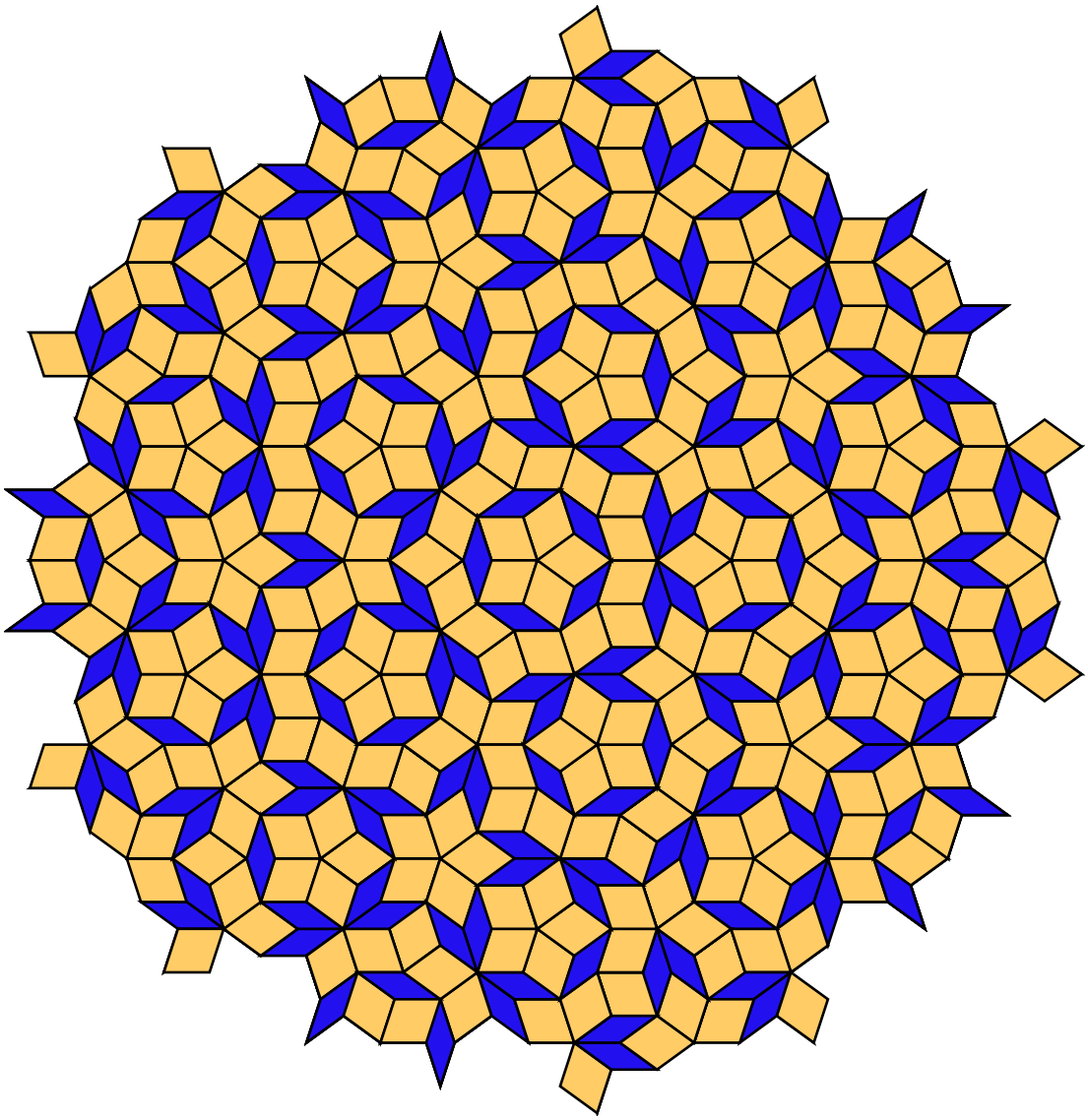


Figure 4.9: Penrose tiling

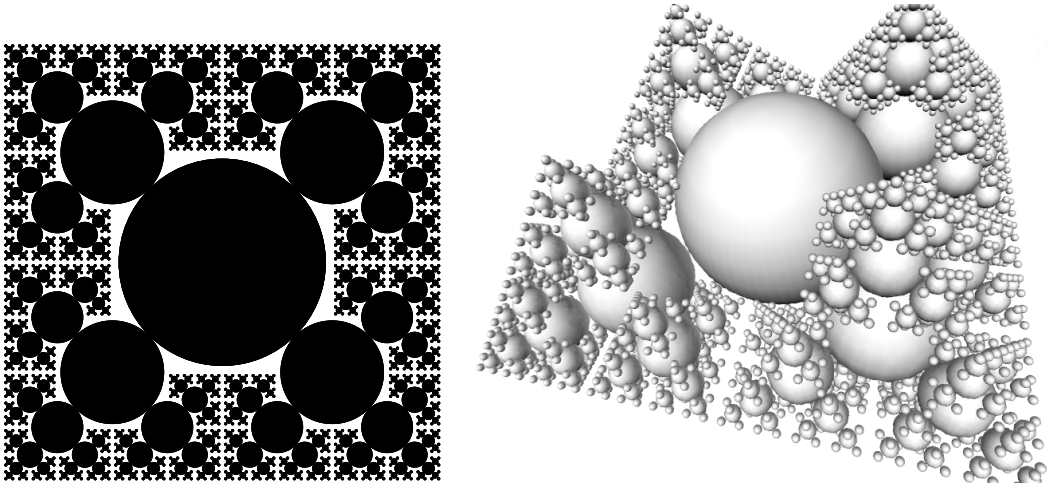


Figure 4.10: Circles (left) and its generalized version in 3D (right)

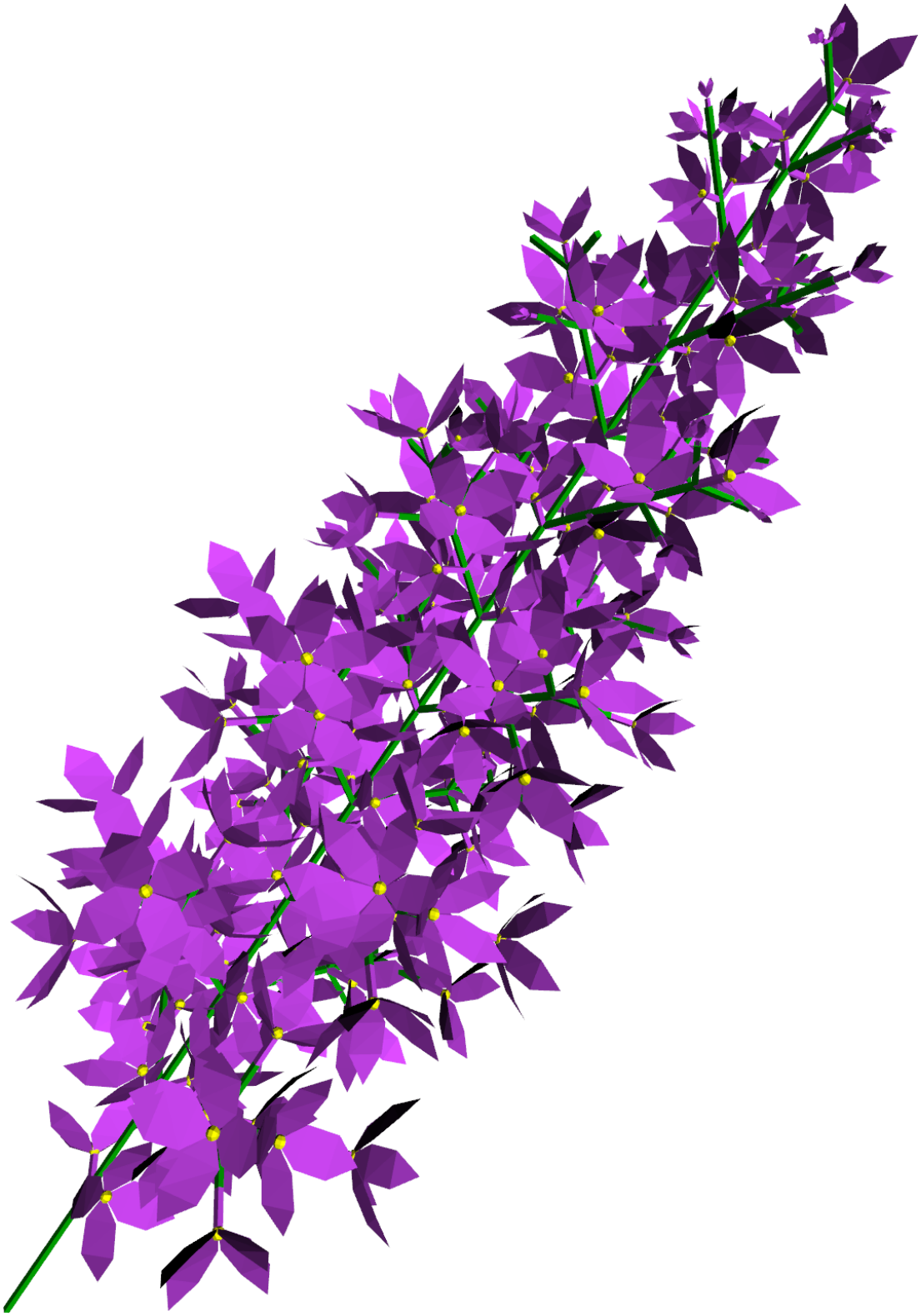


Figure 4.11: Lilac panicle

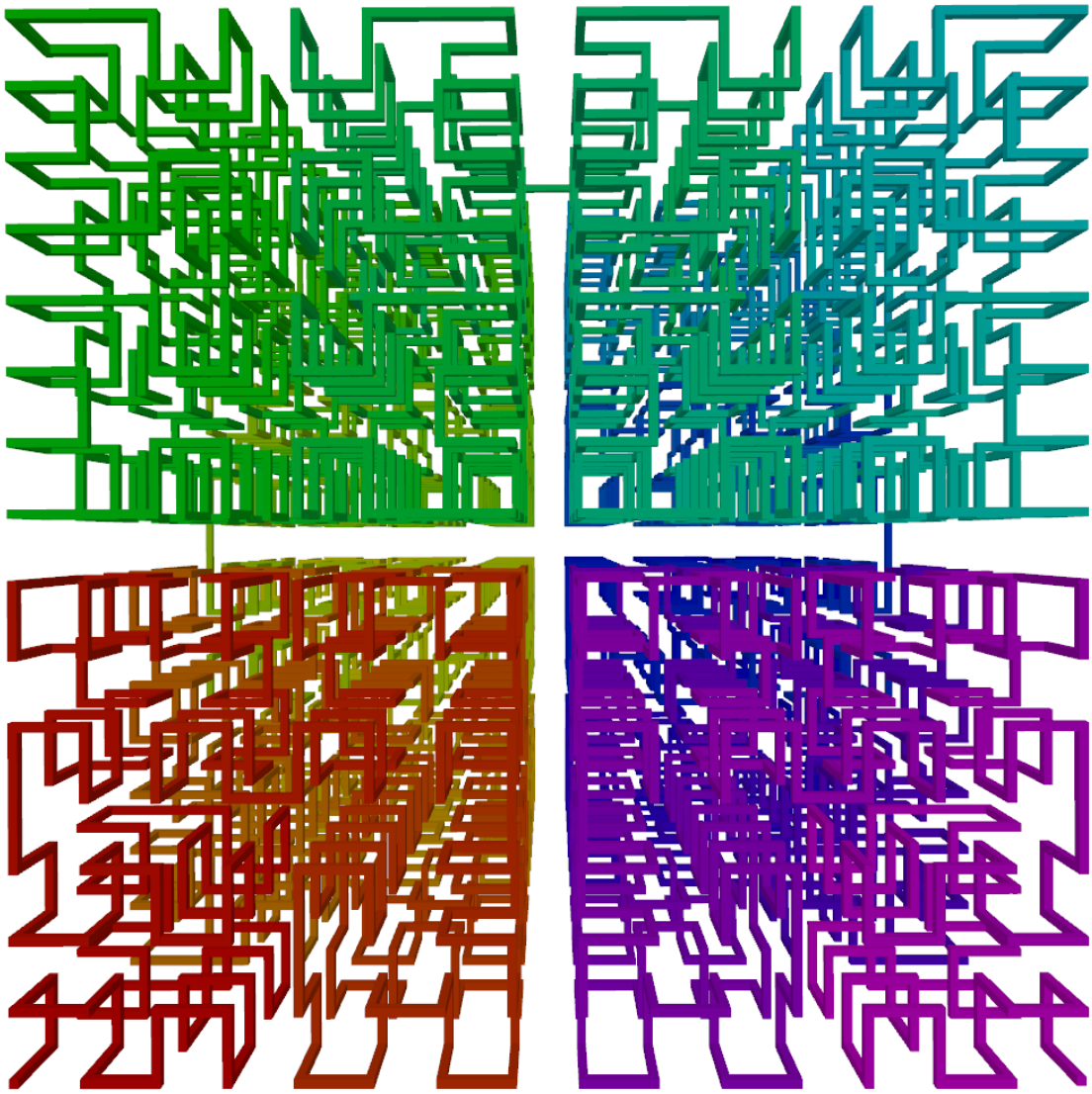


Figure 4.12: Hilbert curve 3D

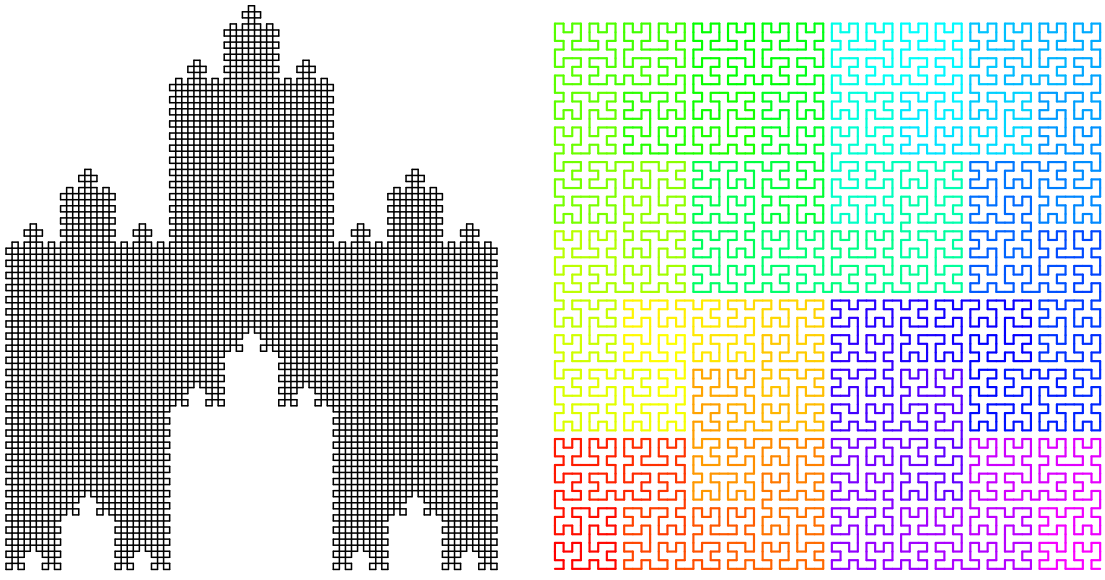


Figure 4.13: Dekking's chirch (left) and Hilbert curve (right)

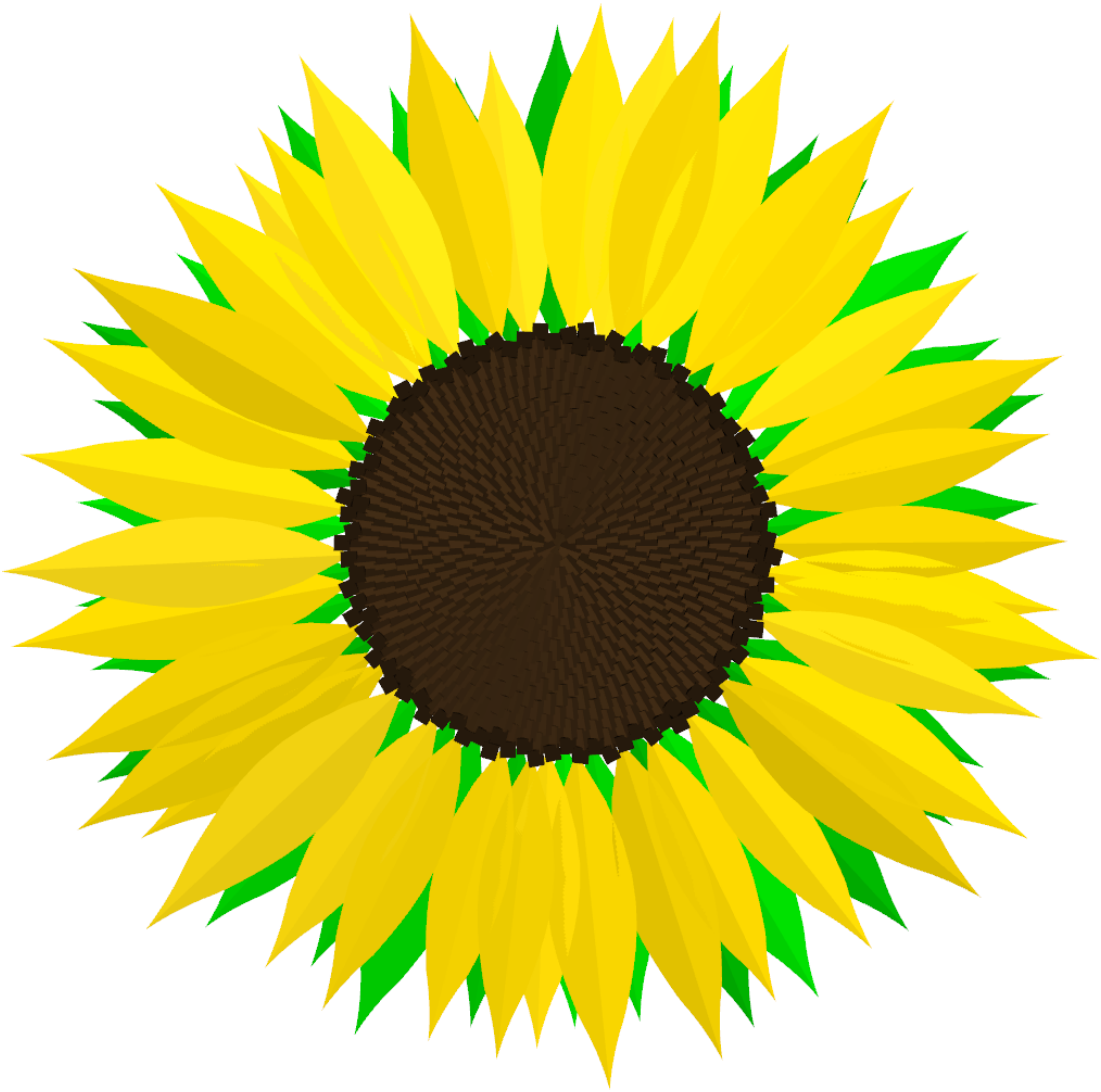


Figure 4.14: Sunflower

Figure 4.15: Models of plant-like structures with withered dandelion in the middle

Conclusion

The goal of this work was to an create online feature-rich development environment for anyone who wants to experiment with L-systems. This goal was achieved and the result can be seen at `http://malsys.cz`. However more than just a web-based L-system generator was created.

As part of the solution a standalone modular L-system processing library has been created which can process L-systems with component-based system. Component system and configuration of individual components is defined in the input together with L-systems. The components can be created or extended by the user which brings great extensibility to the L-system processing. Many components are already part of the library. They are used on the web to process L-systems and produce 2D images and 3D scenes or even ASCII art.

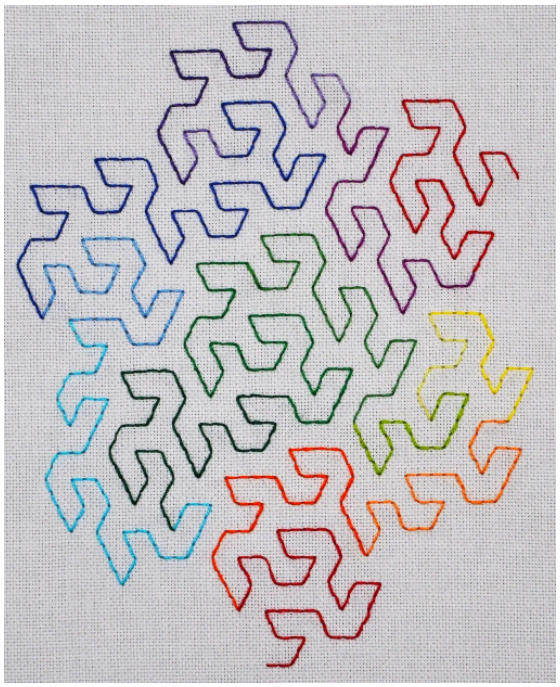


Figure 4.16: Needlework of Hexagonal Gosper curve

A component is a piece of the program and thus it can do anything. For example, it is possible to create a special component which will interpret L-system symbols as commands for some CNC[1] sewing machine which can sew a design as an ornament onto a T-shirt, carpet or curtain (Fig. 4.16). If a stochastic L-system would be used then no two T-shirts will have the same design on them. This example may seem relatively bizarre but it does reflect the extensibility of the library well.

Part of the created web interface is the gallery of L-systems with more than 50 inserted L-systems (at the time of publishing this thesis) and it is slowly becoming a unique database of all basic L-systems. Any registered user can save their L-systems and publish them on the gallery. Published L-systems can be rated by others.

Future work

The web user interface does not provide any way for some form of communication between users. A great improvement would be the possibility to add comments to the gallery entries and write personal messages to other users. Also some simple forum could be helpful.

[1] CNC stands for Computer Numerical Control and refers specifically to a computer *controller* which drives a powered mechanical device that, for instance, uses a number of different tools, drills, saws, etc., for fabricating items using materials like metal or wood.

The L-system processing library was written with an emphasis on functionality and simplicity, and not performance. The performance for processing L-system on the web is sufficient because it is not even possible to display large outputs in web browsers. However there are many areas where improvements could be made. For example, the compiler could optimize expression trees to eliminate static expressions ($1 + 2 \rightarrow 3$).

Because of the component-based design of the L-system processing library it is possible to extend it with a minimum of effort. The plan was to create a renderer component which renders the L-systems as a scene with the PovRay ray-tracer but there was insufficient time to implement this.

The syntax parser has poor error recovery which should be improved. Some syntax errors even do not show their position.

Bibliography

[CD93] K. Culik and S. Dube. *L-systems and mutually recursive function systems*. In: Acta Informatica 30.3 (1993), pp. 279–302. ISSN: 0001-5903 (cit. on p. 5).

[Dun10] R. Dunlop. *Avatar*. May 2010. URL: `http://cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/avatar` (visited on 03/2012) (cit. on p. 5).

[Khr12] Khronos Group. *WebGL – OpenGL ES 2.0 for the Web*. 2012. URL: `http://www.khronos.org/webgl/` (visited on 03/2012).

[HCJ99] C. Hazard, K. Catherine, and D. Johnson. *Fractal Music*. 1999. URL: `http://www.tursiops.cc/fm/` (visited on 03/2012) (cit. on p. 5).

[Lin68] A. Lindenmayer. *Mathematical models for cellular interactions in development*. In: Journal of theoretical biology Parts I and II 18.3 (1968), pp. 280–315 (cit. on p. 5).

[Man06] S. Manousakis. *Musical L-systems*. In: Unpublished master thesis, Institute of Sonology, The Hague (2006) (cit. on p. 5).

[MP96] R. Měch and P. Prusinkiewicz. *Visual models of plants interacting with their environment*. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM. 1996, pp. 397–410 (cit. on p. 35).

[PH93] P. Prusinkiewicz and M. Hammel. *A fractal model of mountains and rivers*. In: Graphics Interface. Vol. 93. Canadian Information Processing Society. 1993, pp. 174–180 (cit. on p. 5).

[PL91] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. In: (1991) (cit. on pp. 11, 12, 15, 58, 95–98, 100).

[PM01] Y.I.H. Parish and P. Müller. *Procedural modeling of cities*. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM. 2001, pp. 301–308. ISBN: 1-58113-374-X (cit. on p. 5).

[Pru*03] P. Prusinkiewicz et al. *L-system description of subdivision curves*. In: International Journal of Shape Modeling 9.1 (2003), pp. 41–59 (cit. on p. 5).

[Pru85] P. Prusinkiewicz. *Graphical applications of L-systems*. Department of Computer Science, University of Regina, 1985 (cit. on p. 5).

[Šťa*10] O. Šťava et al. *Inverse Procedural Modeling by Automatic Generation of L-systems*. In: Computer Graphics Forum. Vol. 29. 2. Wiley Online Library. 2010, pp. 665–674 (cit. on p. 5).

[Wor08] D. Worrall. *Lyndenmeyer Systems Tutorial*. May 2008. URL: `http://worrall.avatar.com.au/courses/Lsystems/index.html` (visited on 03/2012) (cit. on p. 5).

[Žár*04] J. Žára et al. *Moderní počítačová grafika*. Computer press, 2004 (cit. on p. 12).

List of Abbreviations

API application programming interface, page 6

AST abstract syntax tree, page 28

CSS Cascading style sheets, page 70

DB database, page 64

DLL dynamic-link library, page 53

EF Entity framework, page 24

FAQ frequently asked questions, page 42

GPU graphics processing unit, page 6

GZip GNU zip (software for file compression and decompression), page 68

HTML5 hypertext markup language, page 6

HTTP Hypertext transfer protocol, page 61

IDE integrated development environment, page 23

IE Internet Explorer, page 75

IoC inversion of control, page 50

MsSQL Microsoft SQL (server), page 24

MVC model-view-controller (design pattern), page 23

ORM object-relational mapping, page 24

OS operating system, page 41

PHP PHP: Hypertext preprocessor (scripting language), page 20

SQL structured query language, page 24

ST semantic tree, page 28

T4 text template transformation toolkit, page 23

URL uniform resource locator (a reference to an Internet resource), page 61

WebGL web graphics library, page 6

List of Figures

1 Examples of models created by an L-system 5
2 Menger sponge created by an L-system 6

1.1 Examples of interpretation of simple string of symbols 11
1.2 The first, second and fourth iteration of the Cesaro curve 11
1.3 Enhanced Cesaro curve 11
1.4 Dragon curve 12
1.5 The first four iterations of a bracketed L-system 13
1.6 A comparison between a non-randomized and randomized plant model 14
1.7 Signal propagation simulated with context-sensitive bracketed L-systems 17
1.8 Parameters usage in L-system interpretation methods and in rewrite rules along with the result 18
1.9 Pythagoras tree created with parametric L-system 18
1.10 Image produced by WWW L-system Explorer 20
1.11 A plant example from L-system Vector Generator 21
1.12 Model of Lily produced by L-studio 22

2.1 The extension of component-based processing system 24
2.2 Possible outputs from process systems in Figure 2.1 25
2.3 Example of source code along with the result – Sierpinski gasket . 26
2.4 Source code compilation system 28
2.5 Abstract syntax tree parsed from Source code 2.4 29
2.6 Semantic tree created by compilation of the AST in Figure 2.5 . . 29
2.8 Input processing scheme 30
2.7 A more complex semantic tree of Source code 2.5 31
2.9 Correctly colored stochastic L-system 33
2.10 Simple L-system processing system 35
2.11 Subdivided L-system processing system 35
2.12 Automated interpreter caller 36
2.13 Input for the iterator can be supplied by another component . . . 36
2.14 Component system for interpretation of a symbol as another L-system 37
2.15 Example of interpreting s symbol as another L-system 38
2.16 Final component system 40
2.17 Logo of the web 41
2.18 First half of the database scheme of the web 43
2.19 Second half of the database scheme of the web 44

3.1 The dependencies of projects in the solution 47
3.2 Hierarchy of the compilers 49
3.3 Example of the add operation to the immutable tree 56
3.4 Result of L-system in Source code 3.8 56
3.5 The context search tree 57
3.6 Ambiguous triangulation of four points 59

3.7 Complex 3D polygon triangulated with three different strategies . 60
3.8 MVC design pattern 61
3.9 Rendered registration form with incorrectly entered e-mail address and too short password 63
3.10 Code snippets showing literal strings in the ASP.NET MVC 3 . . 66
3.11 Code snippets with literal strings replaced by generated equivalents 66
3.12 Example of the usage of cache 68
3.13 Google Chrome developer tools showing difference between first and subsequent loading of the page 69
3.14 Error log provided by Elmah 70

4.1 QR code for http://malsys.cz 74
4.2 Jump-list of pinned site and header of opened Internet Explorer 9 using pinned shortcut 74
4.3 The first page of the gallery displayed on various operating systems 75
4.4 T-square fractal (left) and its generalization to 3D with pyramids instead of squares (right) 76
4.5 Hexagonal Gosper curve 77
4.6 Hexagonal Gosper curve as polygon (left) and as ASCII art (right) 77
4.7 Islands and lakes 78
4.8 Basic (left) and inverted (right) Sierpinski triangles 78
4.9 Penrose tiling 79
4.10 Circles (left) and its generalized version in 3D (right) 79
4.11 Lilac panicle 80
4.12 Hilbert curve 3D 81
4.13 Dekking's chirch (left) and Hilbert curve (right) 81
4.14 Sunflower 82
4.15 Models of plant-like structures with withered dandelion in the middle 82
4.16 Needlework of Hexagonal Gosper curve 83

B.1 Some iterations of the *MycelisMuralis* L-system 101

C.2 The branching in the Pythagoras tree with branches as squares . 103
C.1 L-system processing interface 104
C.3 The first square of the Pythagoras tree 105
C.4 Branching of the Pythagoras thee 106
C.5 Growing Pythagoras tree 107
C.6 Pythagoras tree rendered in 3D 109
C.7 Results of finished L-system of the Pythagoras tree 109

D.1 Testing process configuration 113
D.2 The result of processing 116
D.3 Extended *SymbolPrinter* process configuration with the filter component 116

F.1 Creation of publish the package in the Visual Studio 2010 123
F.2 Marked products to install in the *Web Platform Installer* 124
F.3 App pool settings dialogs 126
F.4 Filled *Add Web Site* dialog 126
F.5 Rights of the *App_Data* directory 127

List of Tables

1.1 Result of the L-system in Source code 1.1 10
1.2 An axiom and the first 6 iterations of an L-system in Source code 1.6 showing signal propagation in the given string of symbols 16
1.3 Examples of context matching in bracketed L-systems 16
1.4 The result of the L-system simulating acropetal signal propagation in Source code 1.7 17

4.1 Unit tests code coverage of main projects 73
4.2 Number of lines of code written by hand (if not stated otherwise) based on file types 76

H.1 Meaning of syntax of regular expressions 133

List of source codes

1.1 A simple L-system as an example of rewriting principles 10
1.2 Another symbol interpretation example 11
1.3 D0L-system for the generation of the Dragon curve (Figure 1.4) . 12
1.4 A bracketed L-system that which creates a plant-like model (Figure 1.5) 13
1.5 Stochastic L-system with randomized interpretation of symbols and rewrite rule replacements 15
1.6 Context-sensitive L-system simulating signal propagation 15
1.7 The L-system simulating acropetal signal propagation (Figure 1.7a) 17
2.1 Example of array syntax. 27
2.2 Example L-systems inheritance. 27
2.3 Example of the array syntax. 28
2.4 Constant definition statement for example of AST 28
2.5 This source code results in the semantic tree shown in Figure 2.7 30
2.6 Stochastic L-system with a variable number of line segments . . . 34
2.7 Source code of L-system (Fig. 2.15f) demonstrating use of an interpreting symbol as another L-system 39
3.1 Example of the definition file for the *FsLex* tool 48
3.2 Example of definition file for *FsYacc* 48
3.3 General interface for the compilers, interface for the constant definition compiler and its implementation 50
3.4 General interface for compilers and interface for the expression compiler 50
3.5 Interface of the *ProcessManager* class 51
3.6 Example of usage the XmlDoc for documentation of a component 53
3.7 Interface of the *ProcessManager* class 54
3.8 L-system 56
3.9 Implementation of the *Yaw* method of the *TurtleInterpreter* . . . 58
3.10 Implementation of the tropism which is applied after each line . . 58
3.11 Example of routes definition 61
3.12 The *Detail* method of the *Gallery* controller 61
3.13 Gallery *detail* view demonstrating the Razor syntax 62
3.14 Model class with data annotations 62
3.15 Part of user registration view 62
3.16 Process time settings in the *Web.config* 63
3.17 The working directories settings in the *Web.config* file 64
3.18 Additional setting in the *Web.config* file 64
3.19 Registration of the dependency container and its configuration . . 65
3.20 Registration of dependency container and its configuration 65
3.21 Example of two action methods which lists all the defined functions and components 67
3.22 Compression part of the *Web.config* 68
3.23 69
3.24 Example of the LESS source code 70
3.25 Compiled LESS code (Source code 3.24) to the CSS 71

3.26 Configuration of implicit LESS files compilation in *Web.config* . . 71
C.1 Final L-system of the Pythagoras tree 108
D.1 Filter component with static filtering 112
D.2 L-system code for testing the filter component 113
D.3 Filter component with static filtering 114
D.4 L-system code for testing improved filter component 115
D.5 L-system code for testing improved filter component 116
D.6 Test of extended *SymbolPrinter* process configuration with created filter component 117
D.7 Symbol filter component with documented members 118
E.1 Symbol filter component with documented members 120
E.2 Symbol filter component with documented members 121
G.1 Example of code contracts in the *Triangularize* method of the *Polygon3DTrianguler* class. 130
G.2 Usage of the *Grid* component to show list of user roles 131

A. Contents of attached CD

Contexts of attached CD are listed in the following directory tree.

src – contains the source codes

- **ExamplePlugin** – example plugin, contains a component whose implementation is explained in appendix D
- **Malsys** – L-system processing library
- **Malsys.Ast** – abstract syntax tree
- **Malsys.Common** – common functionality
- **Malsys.Parsing** – lexer and parser (in F#)
- **Malsys.Tests** – unit tests
- **Malsys.Web** – web user interface
- **packages** – third party libraries

doc – contains generated documentation and a PDF file with this thesis

B. About figures

All images of L-systems in this thesis (if not stated otherwise) are created in the created web by written L-system processing library. Some source codes in the thesis and may be simplified because of lack of the space. This appendix contains additional information about figures and their source codes.

Figure 1a, 4.11 [page 5, 80] 3D model of lilac panicle [PL91, p. 92]. Some blooms have 4 and some 5 leafs.

```
lsystem LilacInflorescences extends Branches {
    // A(energy, branchEnergy)
    set symbols axiom = F(50) A(12, 5);
    set iterations = 12;

    interpret F as DrawForward(10, 2, #00AA00);
    interpret K(age) as lsystem Bloom(age);
    interpret + as Pitch(60);
    interpret - as Pitch(-60);
    interpret / as Roll(90);

    rewrite A(energy) where energy <= 0 to K(1);
    rewrite A(energy, branchEnergy) to [ - / K(1) ] [ + / K(1) ]
        I(0, branchEnergy) / A(energy - 1, branchEnergy);
    rewrite I(t, energy) where energy <= 0 to nothing;
    rewrite I(t, energy) with e = energy - 1, be = energy where t==2
        to I(t + 1, e) [ - F F A(e, be) ] [ + F F A(e, be) ];
    rewrite I(t, e) to F I(t + 1, e - 1);
    rewrite K(age) to K(age + 1);
}
abstract lsystem Bloom(age = 4) extends Polygons {
    let color = #d649ff;
    let leafCount = round(random(3.5, 5.5));
    let angle = 150 / leafCount;
    let size = min(4, age);

    set symbols axiom = F [ G(1.5) K ] leaf;
    set iterations = leafCount;

    interpret F as DrawForward(size * 2.5, 1 + size / 4, color);
    interpret G as MoveForward(size * 2.5);
    interpret K as DrawSphere(size / 2, #ffff00);
    interpret + as Yaw(angle);
    interpret - as Yaw(-angle);
    interpret | as Yaw(180);
    interpret / as Roll;
    interpret ^ as Pitch(-15);

    rewrite leaf to /(360 / leafCount) [ ^(40 + 10*size) <(color) .
        + ^ G . - ^ G . - ^ G . + | +   G . - ^ G .  > ] leaf;
}
process all with ThreeJsRenderer;
```

Figure 1b [page 5] H-tree fractal [PL91, p. 50].

```
lsystem Htree(R = sqrt(2)) extends Branches {
    set symbols axiom = + A(1);
    set iterations = 11;
    set lineCap = none;

    interpret F(x) as DrawForward(R^x * 2 ^ -(currentIteration / 2) * 256, x);
    interpret + as TurnLeft(90);
    interpret - as TurnLeft(-90);

    rewrite A to F(1) [+A] [-A];
    rewrite F(x) to F(x + 1);
```

```
}
process all with SvgRenderer;
```

Figure 2 [page 6] Menger sponge.

```
lsystem MengerSponge {
    set iterations = 3;
    set symbols axiom = F;

    interpret F as DrawForward(10, 10, #FFFFFF);
    interpret f as MoveForward(5);
    interpret + as Yaw(90);
    interpret - as Yaw(-90);
    interpret ^ as Pitch(90);
    interpret & as Pitch(-90);

    rewrite F to - f f + & f f ^ F F F +f+f- F F +f+f- F F +f+f- F
        -f+f+f^f F F &f&f^ F F &f&f^ F ^ ^ f f f & + f F F &f&f^ F
        ^ ^ f f f & + f F F &f&f^ F ^ ^ f f f & + f F f & f f ^ +
        + f f - f f f f f;
    rewrite f to f f f;
}
process all with ThreeJsRenderer;
```

Figure 1.3 [page 11] Row of trees [PL91, p. 48].

```
lsystem RowOfTrees {
    set symbols axiom = F(1, 0);
    set iterations = 10;
    let p = 0.3;
    let q = 1-p;
    let h = (p*q)^0.5;

    interpret F(x) as DrawForward(x * 2 ^ -(currentIteration / 10) * 1024,1);
    interpret + as TurnLeft(86);
    interpret - as TurnLeft(-86);

    rewrite F(x,t) where t == 0 to F(x*p,2) + F(x*h,1) - - F(x*h,1) + F(x*q,0);
    rewrite F(x,t) to F(x,t-1) ;
}
process all with SvgRenderer;
```

Figure 2.2 [page 25] Tree model with simulated gravity [PL91, p. 60]. The tree is actually in 3D but it is rendered as 2D. It is possible to render 3D model using the ThreeJsRenderer.

Changing the *d1*, *d2*, *angle*, *l* and *w* parameters can be created different tree model.

```
lsystem Tree extends Branches {
    let d1 = 94.74; let d2 = 132.63;  // divergence angle 1 and 2
    let angle = 18.95;  // branching angle
    let l = 1.109; let w = 1.732;  // length and width increase rate

    set symbols axiom = /(45) F(100, 1) A;
    set iterations = 6;
    set initialAngle = 90;
    set tropismVector = {0, -1, 0};
    set tropismCoefficient = 0.15;

    interpret F as DrawForward;
    interpret f as MoveForward;
    interpret & as Pitch(-angle);
    interpret / as Roll;

    rewrite A to F(50, w) [ & F(50, 1) A ] /(d1)
```


```
        [ & F(50, 1) A ] /(d2) [ & F(50, 1) A ];
    rewrite F(length, width) to F(length * l, width * w);
    rewrite f(length) to F(length * w);
}
process all with SvgRenderer;
```

Figure 2.17, 4.14 [page 41, 82] Sunflower [PL91, p. 103]. The number of seeds and leafs is configurable.

```
lsystem Sunflower(seedCount = 300, altSeedCount = 50, greenLeafCount = 15,
        yellowLeafCount = 35) extends Branches {

    set symbols axiom = A(0);
    set iterations = seedCount + altSeedCount + greenLeafCount + yellowLeafCount;

    interpret f as MoveForward;
    interpret Seed as DrawForward(24, 18, #332211);
    interpret AltSeed as DrawForward(24, 18, #24180C);
    interpret GreenLeaf as lsystem Leaf(lighten(#00AA00, random(0, 0.1)));
    interpret YellowLeaf as lsystem Leaf(lighten(#E5C500, random(0, 0.1)));
    interpret + as Yaw(137.515);
    interpret / as Roll(45);
    interpret ^ as Pitch(90);
    interpret & as Pitch(-90);

    let altSeedTreshold = seedCount;
    let greenLeafTreshold = seedCount + altSeedCount;
    let yellowLeafTreshold = seedCount + altSeedCount + greenLeafCount;

    rewrite A(n) where n > yellowLeafTreshold
        to + [ f(n^0.5 * 10 - 20) ^(random(5, 15)) YellowLeaf ] A(n+1);
    rewrite A(n) where n > greenLeafTreshold
        to + [ f(n^0.5 * 10 - 20) & f(10) ^ ^(random(0, 5)) GreenLeaf ] A(n+1);
    rewrite A(n) where n > altSeedTreshold
        to + [ f(n^0.5 * 10) ^ f(-12) /(random(-20, 20)) AltSeed ] A(n+1);
    rewrite A(n) to + [ f(n^0.5 * 10 - 10) / Seed ] A(n+1);
}
abstract lsystem Leaf(color = #E5C500) extends Polygons {
    let la = 5; let ra = 1.1; let lb = 1; let rb = 1.2; let pd = 1;
    let angle = 60;
    set symbols axiom = [ [ +(angle) ^ B(0) <(color) . ] A(1, angle) . > ]
        [ [ +(-angle) ^ B(0) <(color) . ] A(1, -angle) . > ];
    set iterations = random(18, 20);

    interpret G as MoveForward;
    interpret + as Yaw(60);
    interpret - as Yaw(-60);
    interpret ^ as Pitch(10);

    rewrite A(t, angle) to . G(la, ra) . [ +(angle) ^ B(t) . > ]
        [ +(angle) ^ B(t) <(color) . ] A(t+1, angle);
    rewrite B(t) where t > 0 to G(lb, rb) B(t - pd);
    rewrite G(s, r) to G(s*r, r);
}
process all with ThreeJsRenderer;
```

Figure 3.7 [page 60] A spiral polygon demonstrating capabilities of 3D triangulizer.

```
lsystem Spiral3D extends Polygons {
    set symbols axiom = <(#AAAAAA) .  X  + F . + Y >;
    set iterations = 14;
    set polygonTriangulationStrategy = maxDistanceFromNonTriangulated;

    interpret F as MoveForward(1);
    interpret + as Yaw(60);
    interpret - as Yaw(-60);
    interpret ^ as Pitch(10);
```

```
    interpret & as Pitch(-10);

    rewrite X to ^ F F . & + X;
    rewrite Y to & & F . ^ ^ - Y;
}
process all with ThreeJsRenderer;
```

Figure 4.4 [page 76] 3D T-square fractal with pyramids instead of squares.

```
lsystem TPyramid extends Branches {
    let size = 64;
    set symbols axiom = F(size) f(-size/2) + f(size/2) + + [ X(size/2) ] f(size) +
         [ X(size/2) ] f(size) + [ X(size/2) ] f(size) + X(size/2);
    set iterations = 5;

    interpret F(x) as lsystem Pyramid(x);
    interpret f as MoveForward;
    interpret + as Yaw(90);

    rewrite X(s) with h = s / 2
        to F(s) f(-h) + f(h) + + [ X(h) ] f(s) + [ X(h) ] f(s) + X(h);
}
abstract lsystem Pyramid(size = 20, color = #F3E3B9) extends StdLsystem3D {
    let h = size / 2; let sq = h * sqrt(3); let a = 90 - rad2deg(asin(sqrt(2/3)));
    set symbols axiom = [ ^(90) f(h) &(90) +(45)
        [ <(color) . &(a) f(sq) . ^(a) +(135) f(size) . > ] +(90)
        [ <(color) . &(a) f(sq) . ^(a) +(135) f(size) . > ] +(90)
        [ <(color) . &(a) f(sq) . ^(a) +(135) f(size) . > ] +(90)
        [ <(color) . &(a) f(sq) . ^(a) +(135) f(size) . > ] ];
}
process all with ThreeJsRenderer;
```

Figure 4.5 [page 77] Hexagonal Gosper curve [PL91, p. 12].

```
lsystem HexagonalGosperCurve {
    set symbols axiom = L;
    set iterations = 5;
    set continuousColoring = true;

    interpret R L as DrawForward(4);
    interpret + as TurnLeft(60);
    interpret - as TurnLeft(-60);

    rewrite L to L + R + + R - L - - L L - R +;
    rewrite R to - L + R R + + R + L - - L - R;
}
process all with SvgRenderer;
```

Figure 4.7 [page 78] Islands and lakes (colored) [PL91, p. 9]

```
lsystem IslandsAndLakesColored extends Polygons {
    let darkColor = #000000;
    let lightColor = #FFFFFF;

    set symbols axiom = <(darkColor,0) . f. - f. - f. - f. >;
    set iterations = 2;
    set reversePolygonOrder = true;

    interpret f g as MoveForward(8);
    interpret + as TurnLeft(90);
    interpret - as TurnLeft(-90);

    rewrite f to f + g <(darkColor, 0) . f. - f. f. - f. - f. f. > + g + f f
        - g <(lightColor, 0) . f. + f. f. + f. + f. f. > - g - f f f;
    rewrite g to g g g g g g;
}
process all with SvgRenderer;
```


Figure 4.8 [page 78] Sierpinski triangles

```
lsystem SierpinskiTrangle extends Polygons {
    set symbols axiom = F + F + F;
    set iterations = 6;

    interpret F f as MoveForward(2 ^ -currentIteration * 600);
    interpret + as TurnLeft(120);
    interpret - as TurnLeft(-120);

    rewrite F to <(0,0) . F . + F . > + f + f F;
    rewrite f to f f;
    rewrite < to nothing;
    rewrite . to nothing;
    rewrite > to nothing;
}
process all with SvgRenderer;
```

Figure 4.9 [page 79] Penrose tiling.

```
lsystem PenroseTiling extends StdLsystem {
    set symbols axiom = [N] + + [N] + + [N] + + [N] + + [N];
    set iterations = 5;
    set reversePolygonOrder = true;
    let darkClr = #221166;  // dark blue
    let lightClr = #FFCC66;  // dark yellow

    interpret M N O P as MoveForward(2 ^ -(currentIteration / 2) * 200);
    interpret + as TurnLeft(36);
    interpret - as TurnLeft(-36);

    rewrite M to O + + <(darkClr,2,#0) . P . - - - - N . [ - O . - - - - M . > ] + +;
    rewrite N to + <(lightClr,2,#0) . O . - -  P . [ - - - M . - - N . > ] +;
    rewrite O to - <(lightClr,2,#0) . M . + +  N . [ + + + O . + + P . > ] -;
    rewrite P to - - <(darkClr,2,#0) . O . + + + + M . [ + P . + + + + N . > ] - - N;
}
process all with SvgRenderer;
```

Figure 4.10 [page 79] 3D version of Circles fractal. Bigger circles are made from more polygons (see the third parameter of the *DrawSphere* interpretation method).

```
lsystem Circles3D extends Branches {
    set symbols axiom = [ X(60) ] + [ X(60) ] + [ X(60) ] + X(60);
    set iterations = 3;
    set smoothShading = true;

    let scale = 3;
    interpret F as MoveForward;
    interpret K(n) as DrawSphere(n, #FFFFFF, n^(1/3));
    interpret + as Yaw(90);
    interpret - as Yaw(-90);
    interpret ^ as Pitch(90);
    interpret & as Pitch(-90);

    rewrite K(n) to K(2*n);
    rewrite F(n) to F(2*n);
    rewrite X to K(2 * scale) F(3 * scale) [ + X ] [ - X ] [ ^ X ] [ & X ] X;
}
process all with ThreeJsRenderer;
```

Figure 4.12 [page 81] 3D Hilbert curve.

```
lsystem HilbertCurve3D extends StdLsystem3D {
    set iterations = 4;
    set symbols axiom = X;
    set continuousColoring = true;
```

```
    interpret F as DrawForward(16,2);
    interpret f as MoveForward(-1);

    rewrite X to ^ \textbackslash X f F ^ \textbackslash X f F X - f F ^ / /
        X f F X & f F + / / X f F X - f F / X - /;
}

process all with ThreeJsRenderer;
```

Figure 4.13a [page 81] Dekkings church, *Advances in Math*, vol. 44, 1982, pp. 78-104. Works only for odd iterations.

```
lsystem DekkingsChurch {
    set symbols axiom = w x y z;
    set iterations = 7;

    interpret F as DrawForward(8);
    interpret + as TurnLeft(90);
    interpret - as TurnLeft(-90);

    rewrite F to nothing;
    rewrite w to F w + F - z F w - F + x;
    rewrite z to + + F - - y - F + x + + F - - y - F + x;
    rewrite y to + + F - - y + F - z;
    rewrite x to F w + F - z;
}
process all with SvgRenderer;
```

Growing plant Following L-system simulates the growth of the *Mycelis muralis* [PL91, p. 89]. Those lucky ones who have a printed version of this thesis can watch the animation of the growth in the bottom left corner of this thesis. Just turn the thesis with face down, open it, grab all pages at the bottom corner and slowly drop one page after another with a thumb. Figure B.1 shows some frames of the animation.

```
lsystem MycelisMuralis extends StdLsystem {
    set symbols axiom = I(20) F A(0);
    set iterations = 50;
    set initialAngle = 90;
    set scale = 4;
    set symbols contextIgnore = + / F W I K;

    interpret K as DrawSphere(3);

    rewrite {S} A to T V K;
    rewrite {V} A to T V K;
    rewrite A(t) where t > 0 to A(t-1);
    rewrite A(t)             to M [ +(30) G ] F /(180) A(2);
    rewrite {S} M     to S;
    rewrite     S {T} to T;
    rewrite {T} G     to F A(2);
    rewrite {V} M     to S;
    rewrite     T {V} to W;
    rewrite     W     to V;
    rewrite I(t) where t > 0 to I(t - 1);
    rewrite I                to S;
}
process all with SvgRenderer;
```

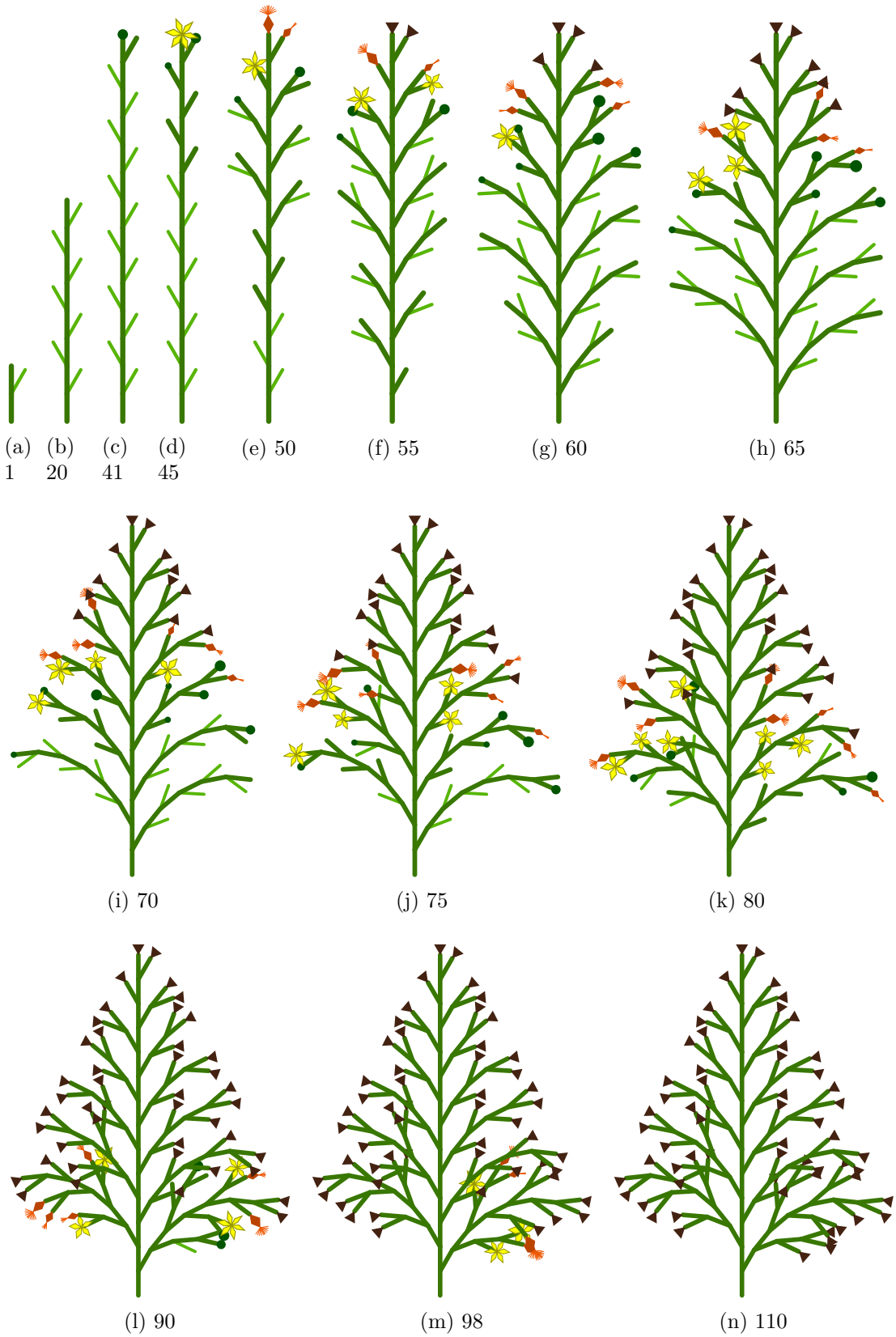



Figure B.1: Some iterations of the *MycelisMuralis* L-system

C. User documentation

In this appendix is explained how to process the L-systems in the web user interface. There is also a step-by-step tutorial how to create an L-system from scratch – the Pythagoras tree.

C.1 How to process L-system

Processing of the input on the web is fairly easy. Following explanation will be referring to Figure C.1.

To process the L-system click on the *L-system processor* link (A) in the main menu of the web, enter the the L-system code into the text area (C) and click the *Process & display results* button. Then you will see the results (D) and eventually some warnings or errors (B).

C.2 Creation of the Pythagoras tree

The creation of actual L-system is little bit harder. Lets start with explanation of the Pythagoras tree itself.

The Pythagoras tree is a binary tree which starts from the root[1]. Two other branches grows each branch (including the root). It is named by the Pythagoras because if we denote the length of the base branch as c and the length of branches raised from the base branch as a and b the Pythagorean theorem describes theirs relation as $c^2 = a^2 + b^2$. If the branches are drawn as squares, the formula also says that the area of the base square is equal to sum of the areas of its child squares. The relation applies to all the squares in the tree.

Pythagoras can be easily built from squares, the angle of branches determines the size of the squares. The branching setup is shown in Figure C.2. If we denote the left angle of the triangle as α and the length of the edge of the base square as c then the length of the edge of the left branch is equal to the $b = c \cdot \cos(\alpha)$ and similarly for the right branch ($a = c \cdot \sin(\alpha)$).

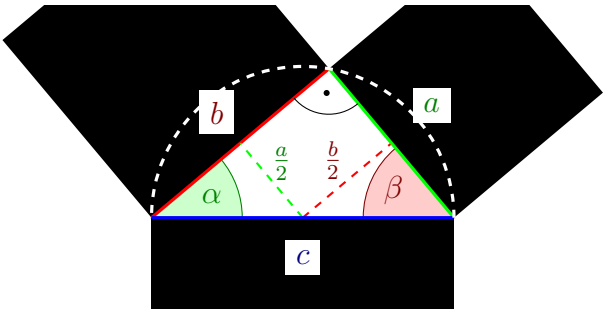


Figure C.2: The branching in the Pythagoras tree with branches as squares

Lets try to draw similar scheme using the L-system. For drawing we will use the *SvgRenderer* process configuration which renders the L-systems with 2D turtle graphics.

[1]The root is at the bottom (note for computer scientists).

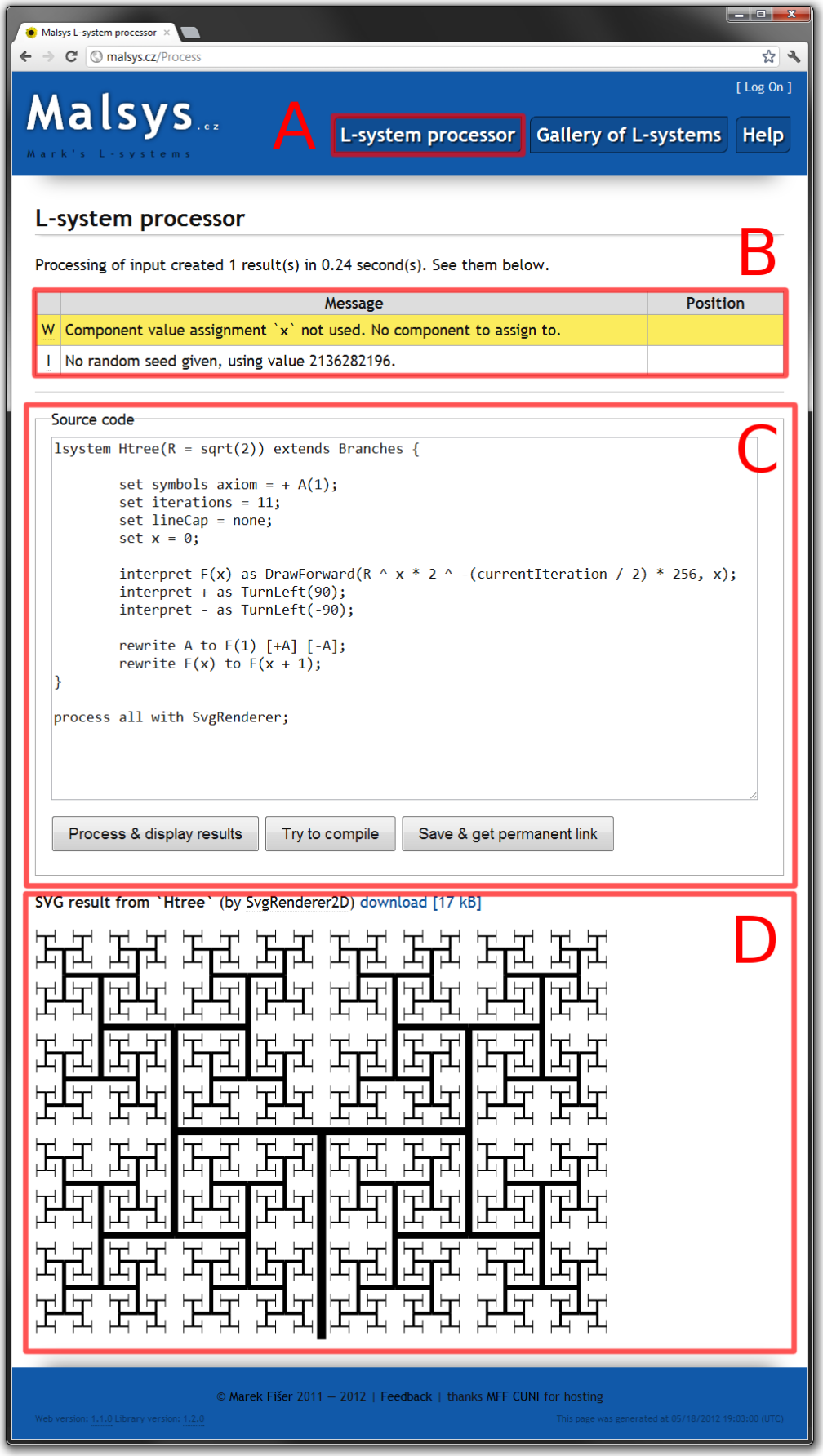


Figure C.1: L-system processing interface

Firstly, we need to draw a square. The simplest square is a line with the same width as length. If we look in the consolidated documentation of the *SvgRenderer* process configuration (appendix K) in the section *Interpretation methods* we can see that there is a method called *DrawForward* which is exactly what we need.

Lets write our first code. We start with the L-system called *PythagorasTree* with the axiom containing single symbol which will be interpreted as a square. Also we set the scale to 100 (to be able to see the result well) and initial angle to 90° to start to move up (instead of right). These properties can be found in the section *Settable properties* of mentioned documentation (appendix K). The code and its result is shown in Figure C.3

```
lsystem PythagorasTree {
    set symbols axiom = S;
    set scale = 100;
    set initialAngle = 90;
    interpret S as DrawForward(1, 1);
}
process PythagorasTree with SvgRenderer;
```

Figure C.3: The first square of the Pythagoras tree

The first problem is obvious. By default, lines have round caps. After a quick look into the documentation there is a property called *lineCap*. Its value 0 will remove the round caps. To keep the source code more readable we can use a predefined constant *none* for it (see appendix I.2.1).

```
set lineCap = none;
```

Now we need an angle for counting the sizes of the branches. We will define the angle as a local variable called *alpha* with vale of 90°. The value of the β angle (in Figure C.2) is obviously $90 - \alpha$ so we can define it as a local variable too.

```
let alpha = 30;
let beta = 90 - alpha;
```

To be able to turn by these angles we must define an interpretation methods. Lets define the symbol `+` as turn left by α degrees and symbol `-` as turn right by β degrees (equally as turn left by $-\beta$ degrees).

```
interpret + as TurnLeft(alpha);
interpret - as TurnLeft(-beta);
```

We need to be able to draw branches easily. For this are the *Bracketed L-systems* which allows saving and loading of the interpretation state. We can define the interpretation on our ow by following code.

```
interpret [ as StartBranch;
interpret ] as EndBranch;
```

However it is much easier to just inherit the *Branches* L-system which will do the trick.

```
lsystem PythagorasTree extends Branches {
```

To be able to draw squares with any size we will upgrade the interpretation rule to take single parameter from the interpreting symbol and use it as line length and width.

```
interpret S(size) as DrawForward(size, size);
```

The last thing we need to do is to skip the space between base square and the branch (mark as dotted line in Figure C.2). For this we have the interpretation method called *MoveForward*. We do not need to define explicit parameters because all parameters from interpreted symbol are automatically forwarded to the interpretation method.

```
interpret m as MoveForward;
```

Now we can draw the squares, move without drawing, we can turn and we can do the branching so lets put it all together. To draw a branch of the Pythagoras tree we need to *a*) start the branch, *a*) turn left, *a*) move without drawing by half of the size of the other branch than we are drawing, *a*) draw the square and *a*) end the branch. Likewise with the right branch.

```
set symbols axiom = S(1) // base
     // left branch
    [ + m(1 * sin(deg2rad(alpha)) / 2) S(1 * cos(deg2rad(alpha))) ]
     // right branch
    [ - m(1 * cos(deg2rad(alpha)) / 2) S(1 * sin(deg2rad(alpha))) ];
```

Figure C.4 shows result of putting everything together along with the result.

```
lsystem PythagorasTree extends Branches{
    let alpha = 30;
    let beta = 90 - alpha;

    set symbols axiom = S(1)
    [ + m(1 * sin(deg2rad(alpha)) / 2)
        S(1 * cos(deg2rad(alpha))) ]
    [ - m(1 * cos(deg2rad(alpha)) / 2)
        S(1 * sin(deg2rad(alpha))) ];

    set scale = 100;
    set initialAngle = 90;
    set lineCap = none;

    interpret S(x) as DrawForward(x, x);
    interpret m as MoveForward;
    interpret + as TurnLeft(alpha);
    interpret - as TurnLeft(-beta);
}
process PythagorasTree with SvgRenderer;
```

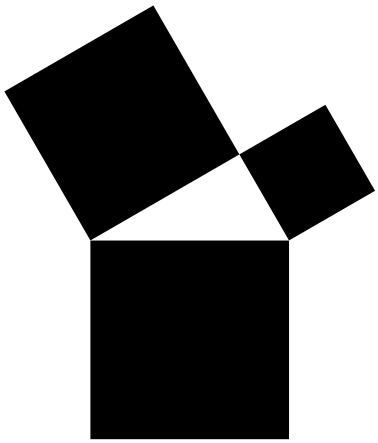


Figure C.4: Branching of the Pythagoras thee

This was the hard part. Now L-systems will do the hard work for us. We need to apply the creation of new branches to again and again.

Because we need to rewrite only the last level of branches we will define a new symbol `X` for already rewritten squares. We can add it to the existing

interpretation rule.

```
interpret S X (size) as DrawForward(size, size);
```

Now for the rewrite rule. All we need to do is to copy the axiom as the rewrite rule and use the parameter of rewritten symbol as the base size.

```
rewrite S(x) to X(x)
    [ + m(x * sin(deg2rad(alpha)) / 2) S(x * cos(deg2rad(alpha))) ]
    [ - m(x * cos(deg2rad(alpha)) / 2) S(x * sin(deg2rad(alpha))) ];
```

To simplify the rewrite rule we can define local variables.

```
rewrite S(x)
    with a = x * sin(deg2rad(alpha)), b = x * cos(deg2rad(alpha))
    to X(x) [ + m(a / 2) S(b) ] [ - m(b / 2) S(a) ];
```

To rewrite the L-system we need to set the number of iterations. We will use lower number like 2 to see if it is working.

```
set iterations = 2;
```

Voilà, the Pythagoras tree is growing.

```
lsystem PythagorasTree extends Branches{
    let alpha = 30;
    let beta = 90 - alpha;

    set symbols axiom = S(1);

    set scale = 100;
    set initialAngle = 90;
    set lineCap = none;
    set iterations = 2;

    interpret S X(x) as DrawForward(x,x);
    interpret m as MoveForward;
    interpret + as TurnLeft(alpha);
    interpret - as TurnLeft(-beta);

    rewrite S(x)
        with a = x*sin(deg2rad(alpha)),
             b = x*cos(deg2rad(alpha))
        to X(x) [ + m(a / 2) S(b) ]
            [ - m(b / 2) S(a) ];
}
process PythagorasTree with SvgRenderer;
```

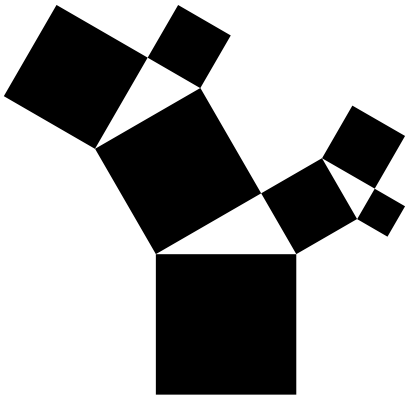


Figure C.5: Growing Pythagoras tree

We can even render it into 3D with minimal effort, just process it with the *ThreeJsRenderer* (and set little more iterations and green color), see Figure C.6.

```
process PythagorasTree with ThreeJsRenderer
    set initialColor = #00AA00
    set iterations = 10;
```

If we want to generate many Pythagoras trees with different angles we can add a parameter to the L-system. This parameter will be used as the α angle. Then we just remove the local variable.

```
lsystem PythagorasTree(alpha = 30) extends Branches{
```

To process more L-systems at once with different parameters we can add more process statements.

```
process PythagorasTree(30) with SvgRenderer;
process PythagorasTree(35) with SvgRenderer;
process PythagorasTree(40) with SvgRenderer;
process PythagorasTree(45) with SvgRenderer;
```

Final L-system is in Source code C.1 and its results are in Figure C.7

```
lsystem PythagorasTree(alpha = 30) extends Branches{
    let beta = 90 - alpha;
    set symbols axiom = S(1);
    set scale = 100;
    set initialAngle = 90;
    set lineCap = none;
    set iterations = 10;
    interpret S X(x) as DrawForward(x,x);
    interpret m as MoveForward;
    interpret + as TurnLeft(alpha);
    interpret - as TurnLeft(-beta);
    rewrite S(x)
        with a = x*sin(deg2rad(alpha)), b = x*cos(deg2rad(alpha))
        to X(x) [ + m(a / 2) S(b) ] [ - m(b / 2) S(a) ];
}
process PythagorasTree(30) with SvgRenderer;
process PythagorasTree(35) with SvgRenderer;
process PythagorasTree(40) with SvgRenderer;
process PythagorasTree(45) with SvgRenderer;
```

Source code C.1: Final L-system of the Pythagoras tree

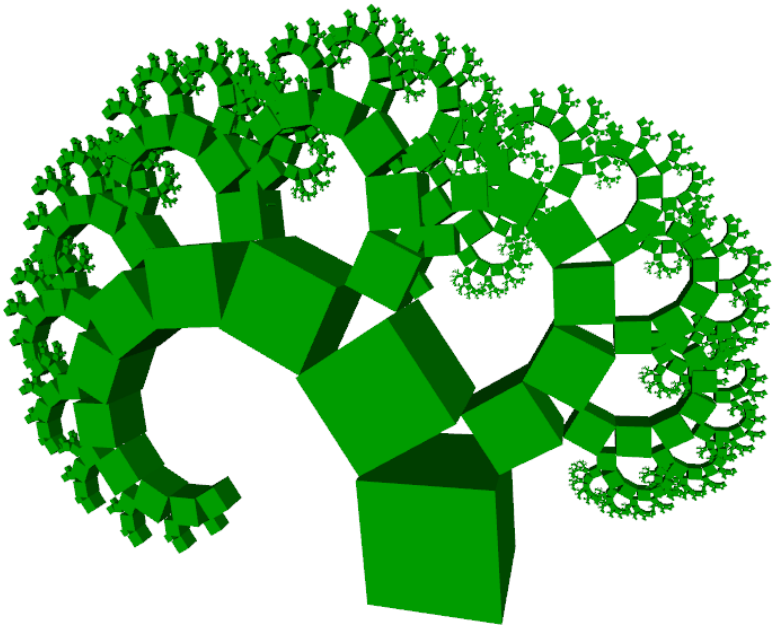
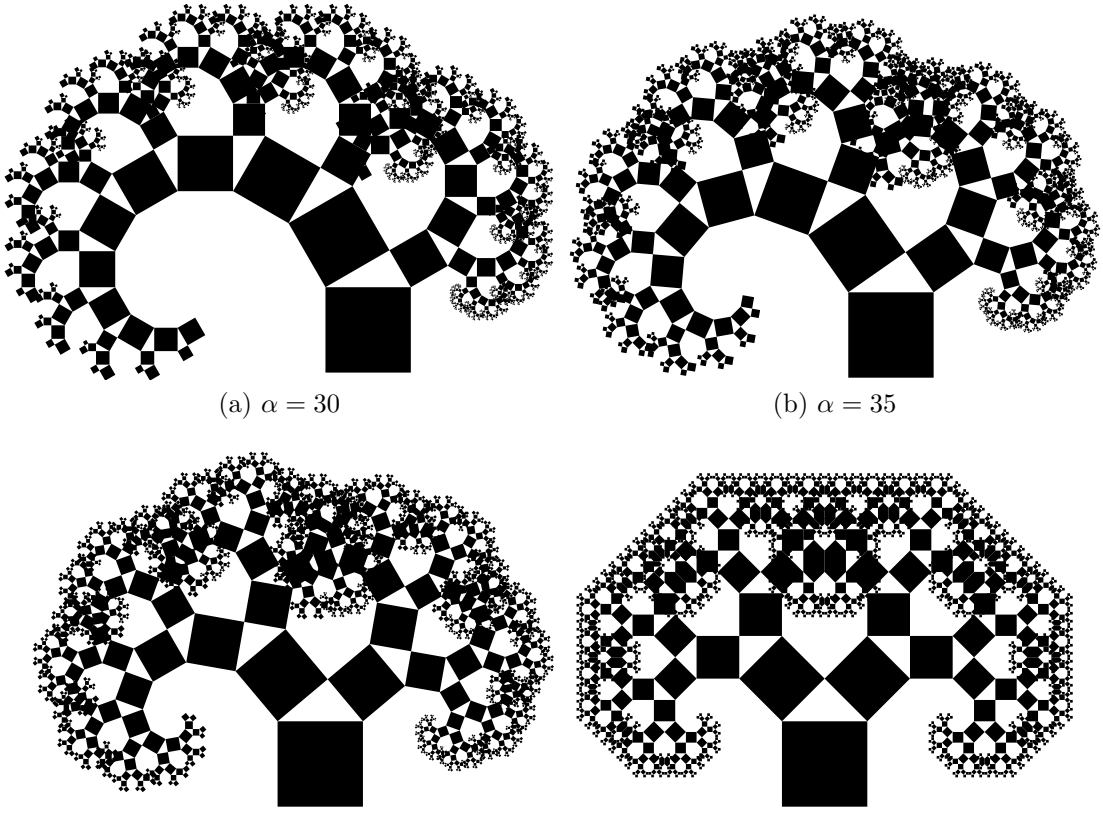


Figure C.6: Pythagoras tree rendered in 3D



(a) $\alpha = 30$ (b) $\alpha = 35$

(c) $\alpha = 40$ (d) $\alpha = 45$

Figure C.7: Results of finished L-system of the Pythagoras tree

D. Component implementation and usage

In this appendix is explained how to implement a component from scratch and use it together with other components in a custom component graph. At the end of this chapter is shown how to document created component correctly.

D.1 Component implementation

For purpose of this example is implemented a component for filtering L-system symbols. In the first part is implemented static (non-configurable) filtering of symbols. Then the component is extended to allow configurable filtering and to check the parameters of passed symbols.

D.1.1 Static filtering

The component is intended to be between the iterator and the interpreter components (see predefined process configurations in appendix I.4). If we look in the section *Connectable properties* in the documentation of the MemoryBufferedIterator we can see that the *OutputProcessor* connectable property accepts the *ISymbolProvider* type. We just start with an empty class implementing that interface.

```
public class SymbolFilter : ISymbolProcessor {
    // IComponent members
    public IMessageLogger Logger { set { throw new NotImplementedException(); } }
    public void Initialize(ProcessContext context) {
        throw new NotImplementedException();
    }
    public void Cleanup() {
        throw new NotImplementedException();
    }
    // IProcessComponent members
    public bool RequiresMeasure { get { throw new NotImplementedException(); } }
    public void BeginProcessing(bool measuring) {
        throw new NotImplementedException();
    }
    public void EndProcessing() {
        throw new NotImplementedException();
    }
    // ISymbolProcessor members
    public void ProcessSymbol(Symbol<IValue> symbol) {
        throw new NotImplementedException();
    }
}
```

The *ISymbolProcessor* interface is implementing other three interfaces, the *IComponent* interface which is base interface for all components and gives us members *Logger*, *Initialize* and *Cleanup*.

The *Logger* property is for logging of errors and messages, we can just auto-implement it.

```
public IMessageLogger Logger { get; set; }
```

The *Initialize* method is for first-time initialization of the component and we can leave it empty. The *Cleanup* method is for setting component to the initial state (prepared for processing), we will do any cleaning there but we will leave it empty for now.

```
public void Cleanup() { }
```

Next block of members is inherited from the *IProcessComponent* interface which allows repetitive processing within processing of the L-system.

The *RequiresMeasure* property is used for indicating whether component needs the measure pass (see section 2.2.6) which we do not, we can return false.

```
public bool RequiresMeasure { get { return false; } }
```

The *BeginProcessing* and *EndProcessing* methods are for signaling individual process passes. Within them we must call our output processor but we don't have one yet so lets define it. The output processor will have *ISymbolProcessor* type to be possible to connect the same components as was connected to the original iterator component (to which we will be connected).

```
[UserConnectable]
public ISymbolProcessor Output { get; set; }
```

Now we can implement the *BeginProcessing* and *EndProcessing* methods properly by calling the output processor.

```
public void BeginProcessing(bool measuring) {
    Output.BeginProcessing(measuring);
}

public void EndProcessing() {
    Output.EndProcessing();
}
```

The last unimplemented method in our component is the *ProcessSymbol* method. This method is called only between the *BeginProcessing* and *EndProcessing* methods. Lets implement simple filtering based on the case of the first letter of passing symbols. We will send to output only symbols with lower-case first letter.

```
public void ProcessSymbol(Symbol<IValue> symbol) {
    if (char.IsLower(symbol.Name[0])) {
        Output.ProcessSymbol(symbol);
    }
}
```

So if we put everything together we will get something similar to Source code D.1;

```
public class SymbolFilter : ISymbolProcessor {
    [UserConnectable]
    public ISymbolProcessor Output { get; set; }

    public IMessageLogger Logger { get; set; }
    public void Initialize(ProcessContext context) { }
    public void Cleanup() { }

    public bool RequiresMeasure { get { return false; } }
    public void BeginProcessing(bool measuring) {
        Output.BeginProcessing(measuring);
    }
    public void EndProcessing() {
        Output.EndProcessing();
    }
```


```
    public void ProcessSymbol(Symbol<IValue> symbol) {
        if (char.IsLower(symbol.Name[0])) {
            Output.ProcessSymbol(symbol);
        }
    }
}
```

Source code D.1: Filter component with static filtering

Creation of process configuration

And that's is. For testing the the functionality we need to plug the filter component to some process configuration. We will do a very simple process configuration for testing purposes (Fig. D.1). The Iterator component have no rewriter component attached to it because we do not need to iterate (number of iterations is by default 0)

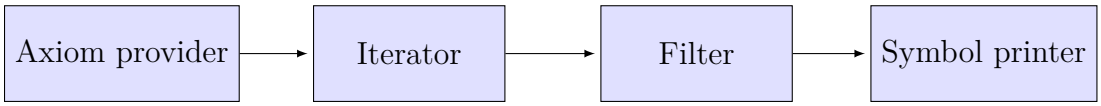


Figure D.1: Testing process configuration

Processing of Source code D.2 resulted to the expected output: `a cd g n oP`. Before running the processing make sure that our filter component is loaded to the component resolver together with other components from the standard library (see section 3.5).

```
configuration FilterTester {
    component AxiomProvider typeof AxiomProvider;
    component Iterator typeof MemoryBufferedIterator;
    component Filter typeof SymbolFilter;
    component SymbolsPrinter typeof SymbolsSaver;

    connect AxiomProvider to Iterator.AxiomProvider;
    connect Filter to Iterator.OutputProcessor;
    connect SymbolProcessor to Filter.Output;
}

lsystem TestLsystem {
    set symbols axiom = a B cd Ef g H I JK Lm n oP;
}

process TestLsystem with FilterTester;
```

Source code D.2: L-system code for testing the filter component

D.1.2 Configurable filtering

To allow the user to chose what symbols are filtered we can use the settable symbol property. Symbols set to this property will be ignored.

The *HashSet<string>* will be used for effective storing and queering for ignored symbols.

```
private HashSet<string> ignoredSymbols = new HashSet<string>();
```

Then we will define the symbol property for setting ignored symbols. In the setter are all given symbols saved to the *HashSet*.

```
[AccessName("ignore")]
[UserSettableSybols]
public ImmutableList<Symbol<IValue>> Ignore {
    set {
        ignoredSymbols.Clear();
        foreach (var sym in value) {
            ignoredSymbols.Add(sym.Name);
        }
    }
}
```

The component must be reusable so we need to clean ignored symbols after each use. For this is the *Clear* method that was mentioned earlier.

```
public void Cleanup() {
    ignoredSymbols.Clear();
}
```

Now we can alter the *ProcessSymbol* method to filter symbols from the *ignoredSymbols* field.

```
public void ProcessSymbol(Symbol<IValue> symbol) {
    if (!ignoredSymbols.Contains(symbol.Name)) {
        Output.ProcessSymbol(symbol);
    }
}
```

The complete source code of the filter component is in Source code D.3;

```
public class SymbolFilter : ISymbolProcessor {

    private HashSet<string> ignoredSymbols = new HashSet<string>();

    [AccessName("ignore")]
    [UserSettableSybols]
    public ImmutableList<Symbol<IValue>> Ignore {
        set {
            ignoredSymbols.Clear();
            foreach (var sym in value) {
                ignoredSymbols.Add(sym.Name);
            }
        }
    }
    [UserConnectable]
    public ISymbolProcessor Output { get; set; }

    public IMessageLogger Logger { get; set; }
    public void Initialize(ProcessContext context) { }
    public void Cleanup() {
        ignoredSymbols.Clear();
    }

    public bool RequiresMeasure { get { return false; } }
    public void BeginProcessing(bool measuring) {
        Output.BeginProcessing(measuring);
    }
    public void EndProcessing() {
        Output.EndProcessing();
    }

    public void ProcessSymbol(Symbol<IValue> symbol) {
        if (!ignoredSymbols.Contains(symbol.Name)) {
            Output.ProcessSymbol(symbol);
        }
    }
```


```
}
```

Source code D.3: Filter component with static filtering

To test new filer component we can use the same process configuration as in the previous part (Source code D.2). The result from Source code D.4 is `A B C A B C`.

```
lsystem TestLsystem {
    set symbols axiom = A + B - C X X Y A + B - C;
    set symbols ignore = X Y + -;
}
process TestLsystem with FilterTester;
```

Source code D.4: L-system code for testing improved filter component

D.1.3 Logging of messages

To further improve the filter component and to show how logging is done we will do a check of parameters of passed symbols. If they will contain any strange values like *NaN* or *infinity* we will report it.

The logic will be placed to the *ProcessSymbol* method. To log message we can use the *Logger* property of our component which was discussed earlier. For arguments with the value equal to *infinity* we will log a warning and for *NaN* values we will log an error. Note that the error will not abort the evaluation but at the end of processing the results will not be shown to the user. To aborting the evaluation can be thrown the *ComponentException*.

```
public void ProcessSymbol(Symbol<IValue> symbol) {
    if (ignoredSymbols.Contains(symbol.Name)) {
        return;  // symbol is ignored
    }
    foreach (var arg in symbol.Arguments) {
        if (!arg.IsConstant) { continue; /* ignore non-constant arguments */ }
        var c = (Constant)arg;
        if (c.IsNaN) {
            Logger.LogMessage("InvalidSymbolParameter", MessageType.Error,
                symbol.AstNode.TryGetPosition(),
                string.Format("Symbol `{0}` have invalid parameter value `{1}`.",
                    symbol.Name, c));
        }
        else if (c.IsInfinity) {
            Logger.LogMessage("StrangeSymbolParameter", MessageType.Warning,
                symbol.AstNode.TryGetPosition(),
                string.Format("Symbol `{0}` have strange parameter value `{1}`.",
                    symbol.Name, c));
        }
    }
    Output.ProcessSymbol(symbol);
}
```

To try newly implemented functionality we will, again, use the *FilterTester* process configuration. Figure D.2 shows the result of processing Source code D.5. Notice the correct positions of the messages (right part of the table in Figure D.2) achieved by the `symbol.AstNode.TryGetPosition()` call.

```
lsystem TestLsystem {
    set symbols axiom = A I(infinity - infinity) B(25)
        X(1/0) A(NaN) Y(25, infinity);
    set symbols ignore = A;
}
process TestLsystem with FilterTester;
```

Source code D.5: L-system code for testing improved filter component

L-system processor

Processing took 0 second(s).

	Message	Position
E	Symbol `I` have invalid parameter value `NaN`.	ln: 2 col: 23
W	Symbol `X` have strange parameter value `Infinity`.	ln: 3 col: 2
W	Symbol `Y` have strange parameter value `Infinity`.	ln: 3 col: 16

Source code

```
lsystem TestLsystem {
        set symbols axiom = A I(infinity - infinity) B(25)
                X(1/0) A(NaN) Y(25, infinity);
        set symbols ignore = A;
}
```

Figure D.2: The result of processing

D.1.4 Usage in real process configuration

New we can use the filter component in more complex process configuration. We will alter actual *SymbolPrinter* process configuration (see appendix I.4.1) by extending it with our filter component as shows Figure D.3.

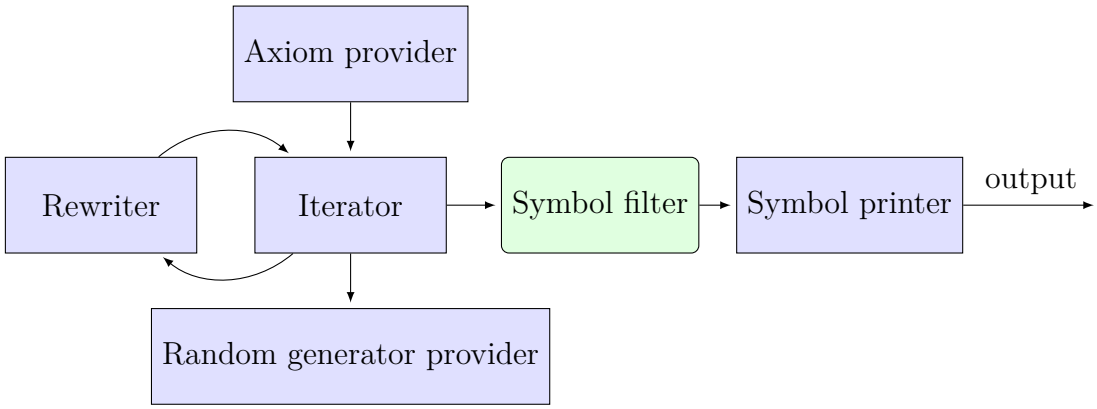


Figure D.3: Extended *SymbolPrinter* process configuration with the filter component

Processing of Source code D.6 will produce following output: `F(2) F(4) F(256) F(Infinity)` along with warning about the parameter of the symbol `F`.

```
configuration FilteredSymbolPrinter {
    component AxiomProvider typeof AxiomProvider;
    component RandGenProvider typeof RandomGeneratorProvider;
    component Filter typeof SymbolFilter;

    container Rewriter typeof IRewriter default SymbolRewriter;
    container Iterator typeof IIterator
        default MemoryBufferedIterator;
    container SymbolProcessor typeof ISymbolProcessor
        default SymbolsSaver;

    connect RandGenProvider to Iterator.RandomGeneratorProvider;
    connect AxiomProvider to Iterator.AxiomProvider;
    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect Filter to Iterator.OutputProcessor;
    connect SymbolProcessor to Filter.Output;
}

lsystem TestLsystem {
    set symbols axiom = X(2);
    set symbols ignore = X;
    set iterations = 4;
    rewrite X(n) to F(n) X((n ^ n));
}

process TestLsystem with FilteredSymbolPrinter;
```

Source code D.6: Test of extended *SymbolPrinter* process configuration with created filter component

D.2 Component documentation

The documentation of components is described in section 3.4.1. Source code D.7 shows the created filter component with documented members. Rest of the class is omitted. The result of generated documentation can be found in appendix J (SymbolFilter component).

```
/// <summary>
/// Filters symbol stream.
/// </summary>
/// <name>Symbol fileter</name>
/// <group>Plugin</group>
public class SymbolFilter : ISymbolProcessor {

    private HashSet<string> ignoredSymbols = new HashSet<string>();

    /// <summary>
    /// List of ignored symbols.
    /// </summary>
    [AccessName("ignore")]
    [UserSettableSybols]
    public ImmutableList<Symbol<IValue>> Ignore {
        set {
            ignoredSymbols.Clear();
            foreach (var sym in value) {
                ignoredSymbols.Add(sym.Name);
            }
        }
    }

    /// <summary>
    /// Components to which filtered symbols are sent.
    /// </summary>
    [UserConnectable]
    public ISymbolProcessor Output { get; set; }

    ...
}
```

Source code D.7: Symbol filter component with documented members

E. Usage of L-system processing library

The web user interface serves as an example of the usage of the L-system processing library however it is relatively complicated (it uses the IoC container). This appendix will show how to process a simple L-system from the string.

We have following L-system definition in the string which we want to evaluate.

```
string sourceCode = string.Join("\n",
    "lsystem Fibonacci {",
    "   set iterations = 6;",
    "   set interpretEveryIteration = true;",
    "   set symbols axiom = A(0) B(1);",
    "   rewrite          A(a) { B(b) } to A(b);",
    "   rewrite { A(a) } B(b)          to B(a + b);",
    "}",
    "process all with SymbolPrinter;");
```

First we need to load all components, functions, operators and other things from the *Malsys* project. If we did not load them we could not use any operators, functions or components. The class *MalsysLoader* will load everything for us, otherwise would have to load all ”stuff” separately.

```
var logger = new MessageLogger();

var knownStuffProvider = new KnownConstOpProvider();
IExpressionEvaluatorContext evalCtxt = new ExpressionEvaluatorContext();
var componentResolver = new ComponentResolver();

var loader = new MalsysLoader();
loader.LoadMalsysStuffFromAssembly(Assembly.GetAssembly(typeof(MalsysLoader)),
    knownStuffProvider, knownStuffProvider, ref evalCtxt, componentResolver, logger);

if (logger.ErrorOccurred) {
    throw new Exception("Failed to register Malsys stuff. "
        + logger.AllMessagesToFullString());
}
```

After loading all standard things we can load some additions or plugins. To demonstrate it we will add a new operator @ which will take two constants and add them (like the + operator). To reflect this in our L-system we will also change the replacement of the second rewrite rule to: `B(a @ b)`.

```
knownStuffProvider.AddOperator(new OperatorCore("@", 300, 320,
        ExpressionValueTypeFlags.Constant, ExpressionValueTypeFlags.Constant,
        (l, r) => ((Constant)l + (Constant)r).ToConst()));
```

Now we can instantiate the main class for L-system processing, the *ProcessManager*. It will need instances of compiler and evaluator containers which will be also created.

```
var compiler = new CompilersContainer(knownStuffProvider, knownStuffProvider);
var evaluator = new EvaluatorsContainer(evalCtxt);
var processMgr = new ProcessManager(compiler, evaluator, componentResolver);
```

Evaluating of the L-system input is just one lie of code.

```
var evaledInput = processMgr.CompileAndEvaluateInput(sourceCode, "testInput", logger);
if (logger.ErrorOccurred) {
    throw new Exception("Failed to evaluate input."+logger.AllMessagesToFullString());
}
```

Before we can process it we must join it with the standard library to be able to use predefined process configurations and other useful definitions. The source code of the standard library is stored as an resource in the *Malsys* project. First, we need to read it.

```
string stdLibResName = ResourcesHelper.StdLibResourceName;
string stdlibSource;
using (Stream stream = new ResourcesReader().GetResourceStream(stdLibResName)) {
    using (TextReader reader = new StreamReader(stream)) {
        stdlibSource = reader.ReadToEnd();
    }
}
```

Then we will compile it in the same way as the L-system input.

```
var stdLib = processMgr.CompileAndEvaluateInput(stdlibSource, "stdLib", logger);
if (logger.ErrorOccurred) {
    throw new Exception("Failed to build std lib. "+logger.AllMessagesToFullString());
}
```

Now we must join our input and the standard library together. It is important to add our input to the standard library, not in the opposite order.

```
evaledInput = stdLib.JoinWith(evaledInput);
```

Processing of the L-systems needs an output provider. The web user interface uses the *FileOutputProvider* class as an output provider. It saves the outputs as files to the file system. For our purposes is better to use the *InMemoryOutputProvider* class which keeps the outputs in the operating memory.

```
var outProvider = new InMemoryOutputProvider();
processMgr.ProcessInput(evaledInput, outProvider, logger, new TimeSpan(0, 0, 5));
if (logger.ErrorOccurred) {
    throw new Exception("Failed to process input. "+logger.AllMessagesToFullString());
}
```

And that's it. Now we can read all outputs from output provider. We will print them to the system console.

```
var encoding = new UTF8Encoding();
var outputs = outProvider.GetOutputs().Select(x => encoding.GetString(x.OutputData));
foreach (var o in outputs) {
    Console.WriteLine(o);
}
```

The output is in Source code E.1. The complete source code is in Source code E.2.

Source code E.1: Symbol filter component with documented members

```
A(0) B(1)
A(1) B(1)
A(1) B(2)
A(2) B(3)
A(3) B(5)
A(5) B(8)
A(8) B(13)
```


Did you noticed that all error reporting is done by the message logger? There is no need to catch exceptions at all. Only mistakes and bugs in the library will trow exceptions.

Complete source code

```
string sourceCode = string.Join("\n",
    "lsystem Fibonacci {",
    "   set iterations = 6;",
    "   set interpretEveryIteration = true;",
    "   set symbols axiom = A(0) B(1);",
    "   rewrite          A(a) { B(b) } to A(b);",
    "   rewrite { A(a) } B(b)          to B(a @ b);",
    "}",
    "process all with SymbolPrinter;");

var logger = new MessageLogger();
var knownStuffProvider = new KnownConstOpProvider();
IExpressionEvaluatorContext evalCtxt = new ExpressionEvaluatorContext();
var componentResolver = new ComponentResolver();

var loader = new MalsysLoader();
loader.LoadMalsysStuffFromAssembly(Assembly.GetAssembly(typeof(MalsysLoader)),
    knownStuffProvider, knownStuffProvider, ref evalCtxt, componentResolver, logger);

if (logger.ErrorOccurred) {
    throw new Exception("Failed to register Malsys stuff. "
        + logger.AllMessagesToFullString());
}

knownStuffProvider.AddOperator(new OperatorCore("@", 300, 320,
    ExpressionValueTypeFlags.Constant, ExpressionValueTypeFlags.Constant,
    (l, r) => ((Constant)l + (Constant)r).ToConst()));

var compiler = new CompilersContainer(knownStuffProvider, knownStuffProvider);
var evaluator = new EvaluatorsContainer(evalCtxt);
var processMgr = new ProcessManager(compiler, evaluator, componentResolver);


var evaledInput = processMgr.CompileAndEvaluateInput(sourceCode, "testInput", logger);

if (logger.ErrorOccurred) {
    throw new Exception("Failed to evaluate input."+logger.AllMessagesToFullString());
}

string stdLibResName = ResourcesHelper.StdLibResourceName;
string stdlibSource;
using (Stream stream = new ResourcesReader().GetResourceStream(stdLibResName)) {
    using (TextReader reader = new StreamReader(stream)) {
        stdlibSource = reader.ReadToEnd();
    }
}

var stdLib = processMgr.CompileAndEvaluateInput(stdlibSource, "stdLib", logger);
if (logger.ErrorOccurred) {
    throw new Exception("Failed to build std lib."+logger.AllMessagesToFullString());
}

evaledInput = stdLib.JoinWith(evaledInput);

var outProvider = new InMemoryOutputProvider();

processMgr.ProcessInput(evaledInput, outProvider, logger, new TimeSpan(0, 0, 5));

if (logger.ErrorOccurred) {
    throw new Exception("Failed to process input. "+logger.AllMessagesToFullString());
}

var encoding = new UTF8Encoding();
var outputs = outProvider.GetOutputs().Select(x => encoding.GetString(x.OutputData));
```

```
foreach (var o in outputs) {
    Console.WriteLine(o);
}
```

Source code E.2: Symbol filter component with documented members

F. Publish on the server

In this appendix are step-by-step instructions how to publish the web site on the Windows Server 2008 R2 Standard x64 SP1. At the end of this chapter is mentioned the migration of the server.

F.1 Creation of publish package

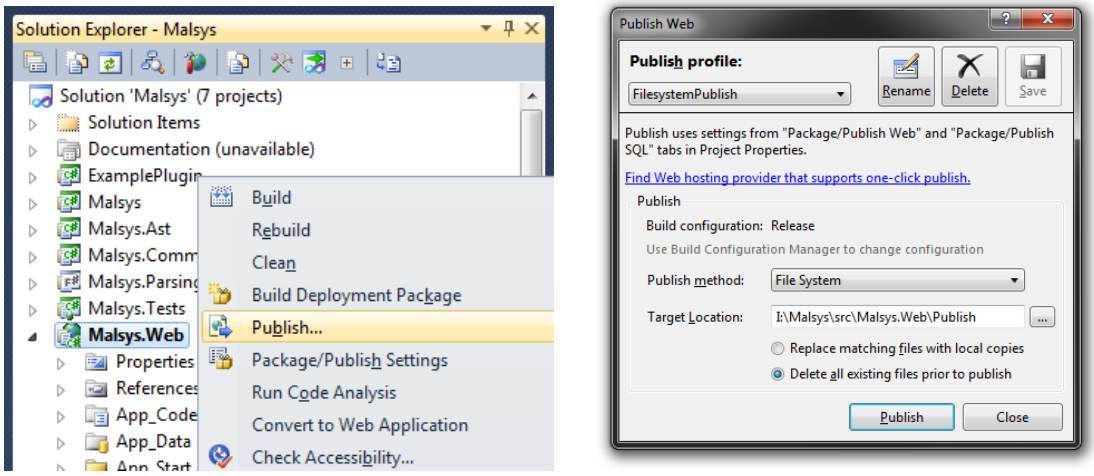
F.1.1 Settings

The web should be configured before the first deploy. The configuration settings are in the *Web.config* file and they are explained in section 3.9.2.

The most settings can be left as default but at leas a valid keys for the *ReCaptcha* should be set [G.13]. Also correct ID for *Google Analytics* [G.14] can be set in the Layout view (*/Views/Shared/_Layout.cshtml*).

F.1.2 Compilation

Creation of the publish package is simple. Open the solution in the Visual Studio 2010 and in the *Solution explorer* right click on the *Malsys.Web* project and chose *Publish* from the context menu (Fig. F.1a). Then appears a popup dialog with publish settings. In this tutorial we will use publishing to the file system as you can see in Figure F.1b. Then click on the *Publish* button and the publish package will be created in selected directory. Do not forget to set *release* configuration in the Visual Studio 2010 before publishing.



(a) (b)

Figure F.1: Creation of publish the package in the Visual Studio 2010

F.2 Configuration of the server

Following steps expects fresh installation of the Windows Server 2008 R2 Standard x64 SP1. Any step can be skipped if any described component is already

installed.

F.2.1 Internet Information Services (IIS)

To run the web we will need the IIS on the server.

To install it, run the *Server Manager* and in *Roles summary* click on the *Add Roles* button. Skip the *Before You Begin* page if it appears, select the *Web Server (IIS)* from the list and click the *Next* button. On *Role Services* screen select *ASP.NET* and *HTTP Redirection* and confirm installation of any dependencies as well. Then click the *Next* button and finish installation.

F.2.2 Web platform installer

For installation of needed programs we will use the Web platform installer.

Download the *Web Platform Installer* from `http://www.microsoft.com/web/downloads/platform.aspx` and run it. In the *Web Platform Installer* switch to the *Products* tab and mark to install following products (Figure F.2):

- Microsoft .NET Framework 4
- SQL Server Express 2008 R2
- ASP.NET MVC 3 (Visual Studio 2010)
- ASP.NET WebPages
- ASP.NET MVC Tools Update
- ASP.NET Web Pages Language Packs Language Packs
- URL Rewrite 2.0
- IIS: Logging Tools

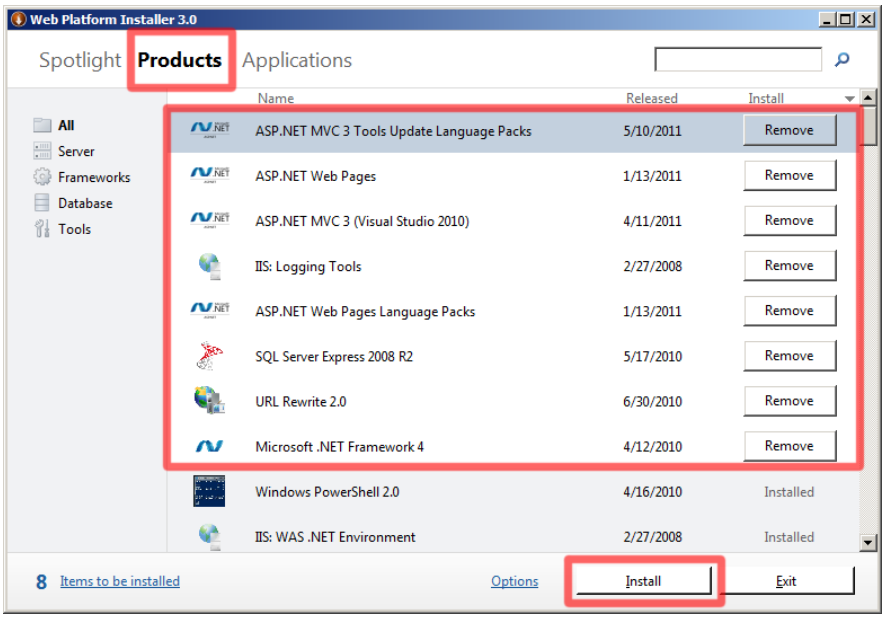


Figure F.2: Marked products to install in the *Web Platform Installer*

F.2.3 F#

F# Redistributable Package

To load the F# 4.0 libraries properly the *F# 2.0 Runtime SP1* must be installed. The redistributable package can be downloaded from `http://msdn.microsoft.com/en-us/library/ee829875.aspx`.

F# PowerPack

Standard F# distribution do not contain tolls like FsLex and FsYacc which are used in this project. They are in downloadable package called the *F# PowerPack* which can be downloaded from `http://fsharppowerpack.codeplex.com/`.

F.3 Deploy of the application

In this moment all needed software to run the web is installed on the server.

F.3.1 Creation of new Application pool

For running the web we will create a new Application pool (App pool) because default App pools are not configured as we need.

Open the *Internet Information Services (IIS) Manager*[1], in the left menu open tree node with the name of your server, right click on the *Application Pools* and choose the *Add Application Pool* from the context menu. Enter the name (*Malsys* in our case) of new App pool and as the *.NET Framework version* chose *.NET Framework v4.0* (Fig. F.3a) and confirm the dialog.

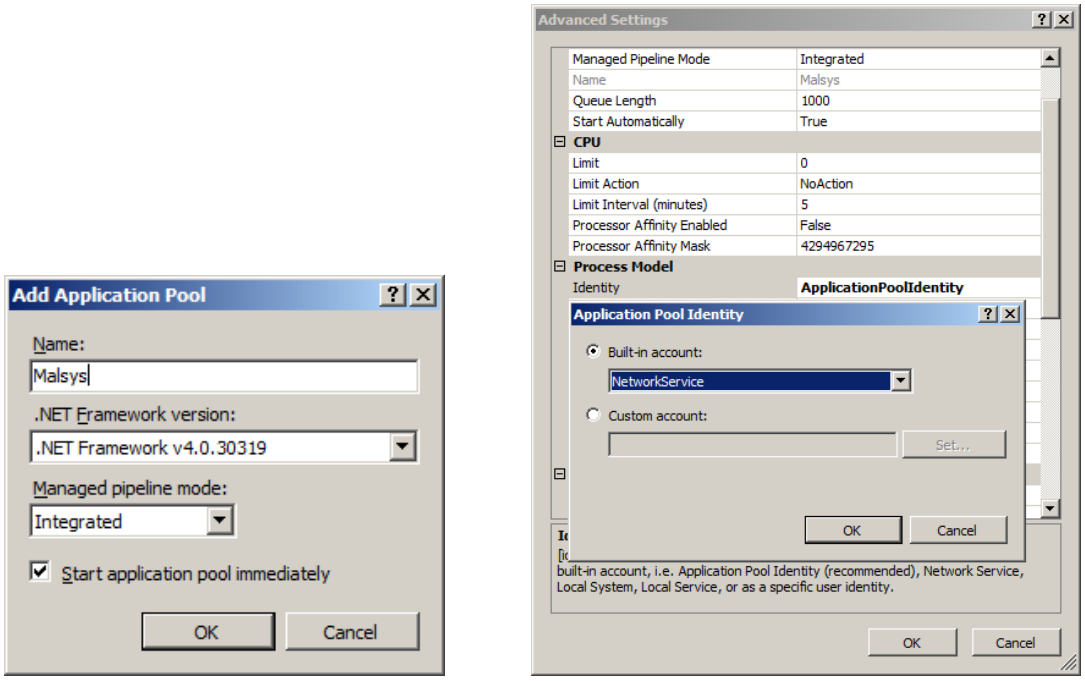
Then select *Application Pools* node (left menu), right click on newly created App pool and chose *Advanced settings*. In the *Advanced Settings* dialog box find the category called *Process Model*. The first row in the category will be the *Identity* row. Click on the *Identity* row and then click on the small button that shows on the right-hand side of the value cell. A dialog box called *Application Pool Identity* will popup. Within that dialog box make sure the first radio button titled *Built-in Account* is selected. In the dropdown box below the radio button choose *Network Service* for the identity (Fig. F.3b). Then confirm all dialogs with *Ok* buttons.

F.3.2 Creation of new App Pool

In the *Internet Information Services (IIS) Manager* click on *Sites* node on the left menu. If your installation of the IIS is fresh you can delete the default site called *Default Site*. Then right click *Sites* node and chose *Add Web Site*.

In the dialog, enter the *Site name* and chose newly created App pool from previous step. Then enter physical path of the web root (*C:\inetpub\WwwMalsys* in our case). Then you can configure *Binding*. If the server will host only this web site we can leave default settings (Fig. F.4).

[1]The *Internet Information Services (IIS) Manager* can be easily found typing the "IIS" in the start menu search bar.



(a) New app pool dialog (b) App pool settings dialog

Figure F.3: App pool settings dialogs

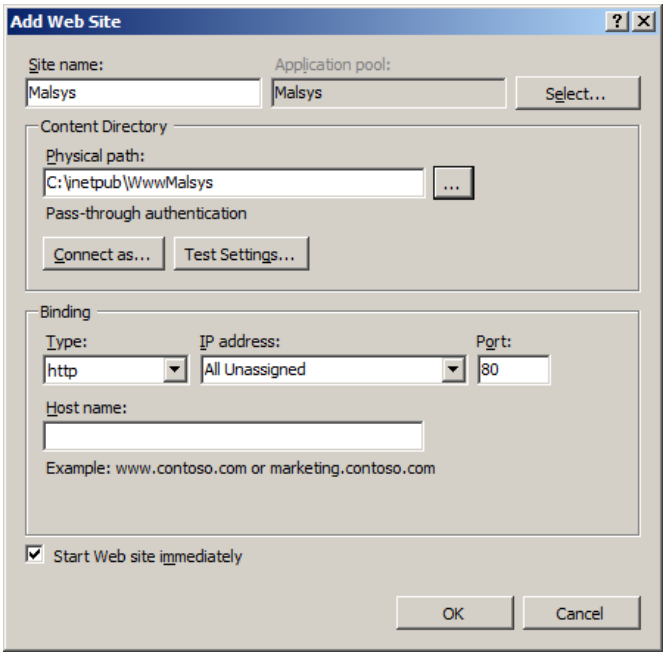


Figure F.4: Filled *Add Web Site* dialog

F.3.3 Copy files

Copy all files from the deployment package (created in section F.1) to the web site root. Database file is not included in deployment package to avoid rewriting "life" version while updating the web site. Empty database file is located in the *App_Data.empty.zip* archive in the *App_Data* directory of the *Malsys.Web* project. Extract both files from the archive to the *App_Data* directory of the

new web.

To allow access of database server to the database files set access to the *App_Data* to full control for users *NETWORK SERVICE* and *IIS_IUSRS* as you can see in Figure F.5.

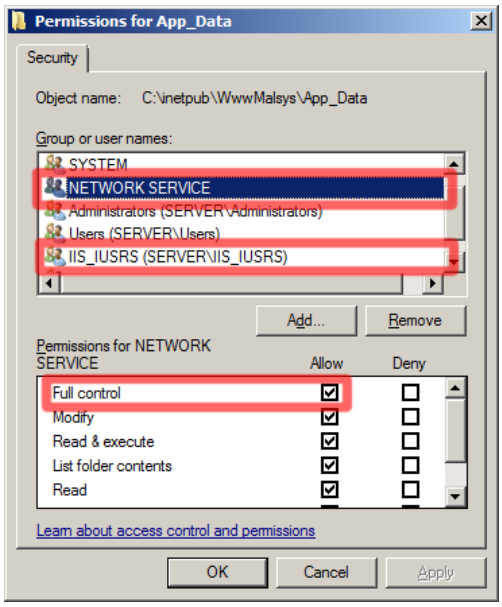


Figure F.5: Rights of the *App_Data* directory

F.4 First run

The web has detection for correctness of basic settings. One of them are working directories for processing and gallery configurable in the *Web.config* (see section 3.9.2). They must be created and accessible for reading and writing for the *IIS_IUSRS* user in order to run the web. The web application will try to create them but if it fail an exception is thrown. The same applies for directory for logging of errors which is also settable in the *Web.config*.

The database is also checked. The existence of following user roles is checked: *administrator*, *trusted*, *viewstats* and *viewfeedbacks*. Missing ones are recreated.

If there is no user in the DB a new user with name *Administrator* is created and automatically added to the *administrator* role. This user have password set to *malsys*. This is important because without administrator is not possible to add any users to user roles, thus it is not possible to promote any user to new administrator. The password of this default user should be changed immediately after deploying.

F.5 Server migration

The migration of the server is very simple. If the new server is prepared all what needs to be done is transfer of the database. Since DB is stored in single *.mdf* file it can be just copied and that's it.

Each image in the gallery will generate automatically on the first request for it.

G. Third-party libraries and services

G.1 F# PowerPack

`http://fsharppowerpack.codeplex.com/`

The F# PowerPack is a collection of libraries and tools for use with the F# programming language provided by the F# team at Microsoft. The PowerPack includes F# versions of lexer and parser generation tools (FsLex and FsYacc), along with the MSBuild tasks to incorporate them in the build process.

The FsLex and FsYacc are used for parsing of input (see section 3.2).

G.2 HTML5 boilerplate

`http://html5boilerplate.com/`

HTML5 Boilerplate is the professional frontend developers's base HTML/CSS/JS template for a fast, robust and future-safe site.

It is used as the base of the HTML and CSS in the web user interface.

G.3 Three.js

`http://github.com/mrdoob/three.js/`

Three.js is lightweight JavaScript 3D library (3D engine) for rendering 3D scenes directly in web browser.

It is used to render 3D models produced by the *ThreeJsSceneRenderer3D* component.

G.4 jQuery

`http://jquery.com/`

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.

All custom JavaScript win web user interface was written with the help of jQuery.

G.5 Modernizr

`http://modernizr.com/`

Modernizr is an open-source JavaScript library that helps to build the next generation of HTML5 and CSS3-powered websites.

Cool HTML5 and CSS3 features can be used in web user interface thanks to Modernizr.

G.6 Code Contracts

`http://research.microsoft.com/en-us/projects/contracts/`

Code Contracts provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants. Contracts act as checked documentation of your external and internal APIs. The contracts are used to improve testing via runtime checking, enable static contract verification, and documentation generation.

The Code Contracts are used in many methods across all projects. Contract checking is turned on while the solution is built on the debug settings but because of performance reasons they are turned off in the release.

In Source code G.1 are defined four preconditions checking validity of arguments and two postconditions ensuring output format. From the contracts is clear that the method must be supplied with at least 3 points in list and the number of returned indices will be divisible by 3 (without reminder). It makes sense because every 3 indices specifies a triangle.

```
public List<int> Triangularize(IList<Point3D> points,
        Polygon3DTriangulerParameters prms) {

    Contract.Requires<ArgumentNullException>(points != null);
    Contract.Requires<ArgumentNullException>(prms != null);
    Contract.Requires<ArgumentException>(points.Count >= 3);
    Contract.Requires<ArgumentException>(
        prms.TriangleEvalDelegate != null);

    Contract.Ensures(Contract.Result<List<int>>() != null);
    Contract.Ensures(Contract.Result<List<int>>().Count % 3 == 0);
    ...
}
```

Source code G.1: Example of code contracts in the *Triangularize* method of the *Polygon3DTrianguler* class.

G.7 Autofac IoC container

`http://code.google.com/p/autofac/`

Autofac is an IoC container for Microsoft .NET. It manages the dependencies between classes so that applications stay easy to change as they grow in size and complexity. This is achieved by treating regular .NET classes as components.

The Autofac have the ASP.NET MVC 3 integration which extends the Autofac by helper methods for simple registration of controllers.

G.8 MvcContrib

`http://mvccontrib.codeplex.com/`

MvcContrib project adds functionality and ease-of-use to Microsoft's ASP.NET MVC Framework.

The part of the MvcContrib is *T4MVC* which is a T4 template for ASP.NET MVC apps that creates strongly typed helpers that eliminate the use of literal strings when referring the controllers, actions and views (see section 3.9.4).

The MvcContrib contains many useful UI components like the *Grid* which is used for displaying data as table (Source code G.2).

```
@Html.Grid(Model).Columns(col => {
        col.For(x => x.RoleId)
            .Named("Id")
            .HeaderAttributes(style => "width: 40px;")
            .Attributes(@class => "center");
        col.For(x => x.Name);
        col.For(x => Html.ActionLink("Edit", MVC.Administration.Roles.Edit(x.RoleId)))
            .Attributes(@class => "center")
            .HeaderAttributes(style => "width: 50px;")
            .Encode(false);
    }).Attributes(@class => "w100")
```

Source code G.2: Usage of the *Grid* component to show list of user roles

G.9 Elmah

`http://code.google.com/p/elmah/`

ELMAH (Error Logging Modules and Handlers) is an application-wide error logging facility that is completely pluggable. It can be dynamically added to a running ASP.NET web application, or even all ASP.NET web applications on a machine, without any need for re-compilation or re-deployment. Once ELMAH has been dropped into a running web application and configured appropriately it is possible to log nearly all unhandled exceptions.

For usage of ELMAH in the web project see section 3.9.7

G.10 LESS css

`http://lesscss.org/`

LESS is dynamic stylesheet language. It extends CSS with dynamic behavior such as variables, mixins, operations and functions. LESS can run on both the client-side and server-side.

LESS was used for definition of stylesheets in the web (see section 3.9.8).

G.10.1 .LESS

`http://www.dotlesscss.org/`

.LESS (pronounced dot-less) is a .NET port of the LESS JavaScript libary. It allows implicit compilation of LESS files to CSS.

G.11 Data Annotations Extensions

`http://dataannotationsextensions.org/`

Data Annotations Extensions provides common validation attributes which extend the built-in ASP.NET DataAnnotations. The core library provides server-side validation attributes that can be used in any .NET 4.0 project.

For the the usage of the annotations see section 3.9.1.

G.12 Yahoo! UI Library

`http://yuicompressor.codeplex.com/`

Yahoo! UI Library is a .NET port of the Yahoo! UI Library's YUI Compressor Java project. The objective of this project is to compress any Javascript and Cascading Style Sheets to an efficient level that works exactly as the original source, before it was minified.

The Yahoo! UI Library is used for minimalization of JavaScript files to improve site performace section 3.9.9.

G.13 ReCaptcha

`http://recaptcha.net/`

reCAPTCHA is a free CAPTCHA[1] service that helps to digitize books, newspapers and old time radio shows.

reCAPTCHA is used at user registration and feedback pages to prevent spamming by bots.

G.14 Google Analytics

`http://www.google.com/analytics/`

Google Analytics (GA) is a free service offered by Google that collects visitors data and generates detailed statistics about them. The GA can track visitors from all referrers, including search engines, social networks, etc.

The GA is used for tracking the activity of users in the web user interface.

[1] A CAPTCHA is a program that can tell whether its user is a human or a computer.

H. Input syntax reference

This appendix contains formal definition of the designed syntax. The syntax is described with the regular expressions which are explained in the first section. The syntax is described from the most general parts to more concrete parts.

H.1 Regular expressions

Table H.1 explains the syntax of the regular expressions used for description of the input syntax.

Regexp	Definition	Example
`' '`	matches the text between quotes (and nothing else)	`'let'` matches only string `let`
`[ ]`	matches one of any characters enclosed in brackets	`[ab]` matches only `a` or `b`
`[ - ]`	matches single character between two specified characters (inclusive)	`[0-9]` matches any digit from `0` to `9`
`|`	matches regexp on the right *OR* on the left of pipe	`'gray'|'grey'` matches only `gray` or `grey`
`?`	preceding regexp must match zero or one times	`'colo' 'u'? 'r'` matches only `color` or `colour`
`+`	preceding regexp must match one or more times	`[0-9]+` matches any non-negative integer like `5` or `42`
`*`	preceding regexp must match zero or more times	`'b' 'e'*` matches `b`, `be` or `beee`

Table H.1: Meaning of syntax of regular expressions

H.2 Tokens

A token is atomic element of grammar. In the token can not be any white-space character however the white-space characters are often used to separate individual tokens. Names of the tokens will be upper-case to distinguish them from the grammar rules.

H.2.1 Identifier

```
ID = (ALPHA_CHAR | '_') (ALPHA_CHAR | DIGIT | '_')* '\''*
```

The `ID` token represents identifier which starts with alphabetic character (letter) or underscore and may also contain digits. At the end can be apostrophes to allow identifiers such as `a'` or `a''`.

Note that regular expression is simplified by the *ALPHA_CHAR* and *DIGIT* to avoid using characters groups in unicode. The ALPHA_CHAR matches any letter and the DIGIT matches any digit character.

H.2.2 Number

```
NUMBER = [0–9]+ ('.' [0–9]+)? ([eE] ('+'|'–')? [0–9]+)?
    | '0'[bB] [01]+
    | '0'[oO] [0–7]+
    | '0'[xX] ([0–9] | [a–f] | [A–F])+
    | '#' ([0–9] | [a–f] | [A–F])+
```

The `NUMBER` token represents a number literal. Numbers can be specified in five formats: floating-point, binary, octal and hexadecimal prefixed `0x` or `#`.

H.2.3 Operator

```
OPERATOR = (firstOpChar opChar*) | '==' | '/'

firstOpChar = [!$%&\<>@^|~:–] | '?' | '+' | '*'
opChar = firstOpChar | '=' | '/'
```

The `OPERATOR` token represents an operator in mathematical expression.

H.3 Input syntax

The syntax is described with regular expressions explained in section H.1. The regular expressions can contain literals, tokens or other regular expressions. The input grammar is white-space independent, between any two regular expression members can be any number of white-space characters.

In each subsection, the first line of formal specification is the regular expression for described statement. Next lines are describing regular expressions used in the main definition.

H.3.1 Input

```
input = inputStatement*

inputStatement = emptyStatement
    | constantDef
    | functionDef
    | lsystemDef
    | processStatement
    | processConfigDef
```

The *Input* rule is the start rule of the input syntax. The Input can contain constant, function and L-system definitions, process statements and process configuration definitions. Empty statement allows redundant semicolons between statements.

H.3.2 Empty statement

```
emptyStatement = ';'
```

Empty statement allows redundant semicolons in syntax.

H.3.3 Constant definition

```
constantDef = 'let' ID '=' expression ';'
```

Defines a constant with name represented by `ID` with value represented by `expression`.

H.3.4 Function definition

```
functionDef = 'fun' ID paramsDefValListParens
    '{' constantDef* 'return' expression ';' '}'
```

Defines a function with name represented by `ID`, parameters (with optional default values) `paramsDefValListParens`, local constants `constantDef` and return value `expression`.

H.3.5 L-system definition

```
lsystemDef = 'abstract'? 'lsystem' ID paramsDefValListParens?
    baseLsystems? '{' lsystemStatement* '}'

baseLsystems = 'extends' baseLsystemsList
baseLsystemsList = ID exprListParens? (',' baseLsystemsList)?
lsystemStatement = emptyStatement
    | constantDef
    | functionDef
    | componentPropertyAssign
    | symbolsInterpretationDef
    | rewriteRule
```

Defines an L-system with name represented by `ID`, optional parameters (with optional default values) `paramsDefValListParens`, base L-systems `baseLsystems` and L-system statements `lsystemStatement`. Arguments can be supplied to each base L-system. The L-system statement can be constant, function and symbols interpretation definition, component property assign and rewrite rule.

Component property assign

```
componentPropertyAssign = 'set' ID '=' expression ';'
    | 'set' 'symbols' ID '=' symbolExprArgs* ';'
```

Defines a component property assign of property with name represented by `ID` to value represented by `expression` (value properties) or to list of symbols `symbolExprArgs` (symbol properties).

Symbols interpretation definition

```
symbolsInterpretationDef = 'interpret'
    symbol+ paramsDefValListParens? 'as' ID exprListParens? ';'
```

Defines an interpretation method with name represented by `ID` to symbols `symbol`. If parameters `paramsDefValListParens` are specified values of arguments of interpreted symbols are matched to parameters and they can be used as variables in the interpretation method arguments `exprListParens`.

Rewrite rule definition

```
rewriteRule = 'rewrite' rrPattern rrConsts? rrCondition?
    'to' rrReplacement ';'
```

Defines a rewrite rule for symbol (and its context) specified in `rrPattern` to symbols `rrReplacement`. Optionally there can be specified local constants `rrConsts` and condition `rrCondition`.

Rewrite rule pattern

```
rrPattern = rrContext? symbolOptParams rrContext?

symbolOptParams = SYMBOL symbol_params?
symbol_params = '(' ( ID (',' ID)* )? ')'
rrContext = '{' symbolPptParams* '}'
```

Defines a pattern of the rewrite rule which defines rewriting of a symbol represented by `symbolOptParams`. The first `rrContext` represents left context of main symbol `symbolOptParams` and the second `rrContext` represents right context. Main symbol and every symbol in context can have specified parameters names. Actual arguments of matched symbols will be set to specified parameters.

Rewrite rule constants definition

```
rrConsts = 'with' rrCostDefsList

rrCostDefsList = ID '=' expression  (',' rrCostDefsList)?
```

Defines local variables in the rewrite rule separated by comma. Syntax is similar to the constant definition but there is no *let* keyword at the beginning and no semicolon at the end.

Rewrite rule condition

```
rrCondition = 'where' expression
```

Defines a rewrite rule condition.

Rewrite rule replacement

```
rrReplacements = 'nothing'
    | symbolExprArgs* rrWeight? ('or'? 'to' rrReplacements)?

rrWeight = 'weight' expression
```

Defines one or more replacements for the rewrite rule. Each replacement can have probability weight.

L-system symbol

```
symbol = ID | OPERATOR | '[' | ']' | '.'
symbolExprArgs = symbol exprListParens?
```

H.3.6 Process configuration definition

```
processConfigDef = 'configuration' ID '{' processConfigStatement* '}'

processConfigStatement = emptyStatement
    | procConfComponentDef
    | procConfContainerDef
    | procConfConnectionDef
```

Defines a process configuration with name represented by `ID` with statements `processConfigStatement`. Statements can be component, container or connection definition.

Process configuration component definition

```
procConfComponentDef = 'component' ID 'typeof' typeId ';'
```

Defines a component with name represented by `ID` with type `typeId`.

Process configuration container definition

```
procConfContainerDef =
    'container' ID 'typeof' typeId 'default' typeId ';'
```

Defines a container with name represented by `ID` with type `typeId` (first) with default component with type `typeId` (second).

Process configuration connection definition

```
procConfConnectionDef = 'virtual'? 'connect' ID 'to' ID '.' ID ';'
```

Defines a connection of component with name represented by `ID` (first) to property with name `ID` (third) of component with name `ID` (second).

H.3.7 Process statement

```
processStatement = 'process' name exprListParens?
    'with' ID useComponents* ';'

name = 'all' | ID
useComponents = 'use' ID 'as' ID
```

Defines processing of one or *all* L-systems represented by `name` with parameters `exprListParens` with process configuration `ID`. At the end or process statement can be specified usage of extra components in containers in configuration.

H.3.8 Mathematical expression

```
expression = exprMember+

exprMember = NUMBER
    | ID
    | OPERATOR
    | exprIndexer
    | exprArray
    | exprFunction
    | '(' expression ')'
exprIndexer = '[' expression ']'
exprArray = '{' exprList? '}'
exprFunction = ID exprListParens
```

The expression consists of list of members. The `ID` token represents a variable. Meaning of the rest of members are obvious from their names. The grammar of expression is not strict, correctness of the expression will ensure the compiler. The parser can not parse expression as tree because the operators are not defined while parsing.

H.3.9 Common rules

Mathematical expression list

```
exprList = expression (',' expression)*
exprListParens = '(' exprList? ')'
```

List of parameters with default values

```
paramsDefValList = ID ('=' expression)?  (',' paramsDefValList)?
paramsDefValListParens = '(' paramsDefValList? ')'
```

Type identifier

```
typeId = ID ('.' ID)*
```

Represents a fully qualified type identifier.

I. Standard library source code

The Standard library was created for easier creation of the inputs. It contains useful constants, L-systems for inheritance and predefined process configurations. The standard library source code is prepended to all processed inputs in the web user interface.

The source code is divided into logical sections. Each section contains short comment and explanation of the source code.

I.1 General Constants

```
let pi = 3.14159265358979323846;
let π = pi;
let e = 2.7182818284590452354;
```

I.2 Component specific constants

Constants defined in this section helps to add a semantic meaning to the numeric values which are used for configuration of the components.

I.2.1 Svg renderer

Following constants represent line cap values of the *LineCap* property of the *SvgRenderer2D* component.

```
let none = 0;
let square = 1;
let round = 2;
```

I.2.2 ThreeJs renderer

Following constants represent triangulation strategies which are set to the *PolygonTriangulationStrategy* property of the *ThreeJsSceneRenderer3D* component.

```
let fanFromFirstPoint = 0;
let minAngle = 1;
let maxAngle = 2;
let maxDistance = 3;
let maxDistanceFromNonTriangulated = 4;
```

I.3 Abstract L-systems

L-systems defined in this section are intended to use as base L-systems for L-systems defined by the user (for inheritance). They are defined as *abstract* to exclude them from processing with the *all* keyword.

I.3.1 Standard L-system 2D

L-system called *StdLsystem* defines interpretation for usual symbols and correctly defines branches.

```
abstract lsystem StdLsystem {
    interpret A B C D E F G as DrawForward(8);
    interpret a b c d e f g as MoveForward(8);

    interpret + as TurnLeft(90);
    interpret -(x = 90) as TurnLeft(-x);
    interpret | as TurnLeft(180);
    interpret / as Roll(180); // switches meaning of + and - symbols

    interpret < as StartPolygon;
    interpret . as RecordPolygonVertex;
    interpret > as EndPolygon;

    set symbols startBranchSymbols = [;
    set symbols endBranchSymbols = ];

    interpret [ as StartBranch;
    interpret ] as EndBranch;
}
```

I.3.2 Standard L-system 3D

L-system called *StdLsystem3D* defines interpretation for usual symbols and correctly defines branches. The only difference between the `StdLsystem3D` and `StdLsystem` is in the interpretation of `+` and `-` symbols. 2D image must be rendered in the XY plane, thus `+` and `-` symbols must do a pitch but in 3D for pitch is more intuitive to use `^` and `&` symbols.

```
abstract lsystem StdLsystem3D {
    interpret A B C D E F G as DrawForward(8);
    interpret a b c d e f g as MoveForward(8);

    interpret + as Yaw(90);
    interpret -(x = 90) as Yaw(-x);
    interpret | as Yaw(180);

    interpret ^ as Pitch(90);
    interpret &(x = 90) as Pitch(-x);

    interpret / as Roll(90);
    interpret \(x = 90) as Roll(-x);

    interpret < as StartPolygon;
    interpret . as RecordPolygonVertex;
    interpret > as EndPolygon;

    set symbols startBranchSymbols = [;
    set symbols endBranchSymbols = ];

    interpret [ as StartBranch;
    interpret ] as EndBranch;
}
```

I.3.3 Branches

L-system *Branches* defines interpretation for branches correctly. To be able to do context rewriting correctly the Rewriter component must know what symbols start and end the branches. This should be the same symbol as supplied to interpreter.

```
abstract lsystem Branches {
    set symbols startBranchSymbols = [;
    set symbols endBranchSymbols = ];

    interpret [ as StartBranch;
    interpret ] as EndBranch;
}
```

I.3.4 Polygons and branches

L-system *Polygons* defines interpretation for polygons and branches.

```
abstract lsystem Polygons {
    interpret < as StartPolygon;
    interpret . as RecordPolygonVertex;
    interpret > as EndPolygon;

    set symbols startBranchSymbols = [;
    set symbols endBranchSymbols = ];

    interpret [ as StartBranch;
    interpret ] as EndBranch;
}
```

I.4 Process configurations

I.4.1 Symbol printer

Process configuration called *SymbolPrinter* prints rewrited symbols as text. It can be used to familiarize with L-system principles. Advanced users can use is while debugging some more complex L-systems.

```
configuration SymbolPrinter {
    component AxiomProvider typeof AxiomProvider;
    component RandomGeneratorProvider typeof RandomGeneratorProvider;

    container Rewriter typeof IRewriter default SymbolRewriter;
    container Iterator typeof IIterator default MemoryBufferedIterator;
    container SymbolProcessor typeof ISymbolProcessor default SymbolsSaver;

    connect RandomGeneratorProvider to Iterator.RandomGeneratorProvider;
    connect AxiomProvider to Iterator.AxiomProvider;
    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect SymbolProcessor to Iterator.OutputProcessor;
}
```

I.4.2 Svg renderer

Process configuration *SvgRenderer* interprets symbols with the *TurtleInterpreter* component and renders the with the *SvgRenderer2D* component to the SVG (Scalable Vector Graphics). However this process configuration is relatively universal and it can be used any 2D or 3D renderer component in the *Renderer* container. Note that L-system is interpreted in 3D but z-coordinate from data sent to the *SvgRenderer2D* are cut off.

```
configuration SvgRenderer {
    component AxiomProvider typeof AxiomProvider;
    component RandomGeneratorProvider typeof RandomGeneratorProvider;
```

```
    component LsystemInLsystemProcessor typeof LsystemInLsystemProcessor;

    container Rewriter typeof IRewriter default SymbolRewriter;
    container Iterator typeof IIterator default MemoryBufferedIterator;
    container InterpreterCaller typeof IInterpreterCaller default InterpreterCaller;
    container Interpreter typeof IInterpreter default TurtleInterpreter;
    container Renderer typeof IRenderer default SvgRenderer2D;

    connect RandomGeneratorProvider to Iterator.RandomGeneratorProvider;
    connect AxiomProvider to Iterator.AxiomProvider;
    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect InterpreterCaller to Iterator.OutputProcessor;
    connect LsystemInLsystemProcessor to InterpreterCaller.LsystemInLsystemProcessor;
    connect Renderer to Interpreter.Renderer;
}
```

I.4.3 ThreeJs renderer

Process configuration *ThreeJsRenderer* is very similar to the *SvgRenderer*. The only difference is in used renderer component in the *Renderer* container. This process configuration uses the *ThreeJsSceneRenderer3D* to render 3D scene for the Three.js engine [G.3].

```
configuration ThreeJsRenderer {
    component AxiomProvider typeof AxiomProvider;
    component RandomGeneratorProvider typeof RandomGeneratorProvider;
    component LsystemInLsystemProcessor typeof LsystemInLsystemProcessor;

    container Rewriter typeof IRewriter default SymbolRewriter;
    container Iterator typeof IIterator default MemoryBufferedIterator;
    container InterpreterCaller typeof IInterpreterCaller default InterpreterCaller;
    container Interpreter typeof IInterpreter default TurtleInterpreter;
    container Renderer typeof IRenderer default ThreeJsSceneRenderer3D;

    connect RandomGeneratorProvider to Iterator.RandomGeneratorProvider;
    connect AxiomProvider to Iterator.AxiomProvider;
    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect InterpreterCaller to Iterator.OutputProcessor;
    connect LsystemInLsystemProcessor to InterpreterCaller.LsystemInLsystemProcessor;
    connect Renderer to Interpreter.Renderer;
}
```

I.4.4 Hexagonal ASCII renderer

Process configuration *HexAsciiRenderer* interpret symbols with special *HexaAsciiInterpreter* which can move forward by fixed steps (not by any distance) and it can turn only by multiples of sixty degrees thus result will be in hexagonal grid. The *HexaAsciiInterpreter* can communicate only with is supposed to communicate with the *TextRenderer* component which is used to generate ASCII art-style output.

```
configuration HexAsciiRenderer {
    component AxiomProvider typeof AxiomProvider;
    component RandomGeneratorProvider typeof RandomGeneratorProvider;
    component LsystemInLsystemProcessor typeof LsystemInLsystemProcessor;

    container Rewriter typeof IRewriter default SymbolRewriter;
    container Iterator typeof IIterator default MemoryBufferedIterator;
    container InterpreterCaller typeof IInterpreterCaller default InterpreterCaller;
    container Interpreter typeof IInterpreter default HexaAsciiInterpreter;
    container Renderer typeof IRenderer default TextRenderer;
```


```
    connect RandomGeneratorProvider to Iterator.RandomGeneratorProvider;
    connect AxiomProvider to Iterator.AxiomProvider;
    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect InterpreterCaller to Iterator.OutputProcessor;
    connect LsystemInLsystemProcessor to InterpreterCaller.LsystemInLsystemProcessor;
    connect Renderer to Interpreter.Renderer;
}
```

I.4.5 Inner L-system process configuration

The *InnerLsystemConfig* process configuration is used by the *ILsystemInLsystemProcessor* component to interpret symbols as another L-system (2.3.3).

Last connection is virtual because the *ILsystemInLsystemProcessor* component will be artificially added to process configuration by itself (the *ILsystemInLsystemProcessor* component).

```
configuration InnerLsystemConfig {
    component Rewriter typeof SymbolRewriter;
    component Iterator typeof InnerLsystemIterator;
    component InterpreterCaller typeof InterpreterCaller;

    connect Iterator to Rewriter.SymbolProvider;
    connect Rewriter to Iterator.SymbolProvider;
    connect InterpreterCaller to Iterator.OutputProcessor;

    virtual connect LsystemInLsystemProcessor
        to InterpreterCaller.LsystemInLsystemProcessor;
}
```

I.4.6 Constant dumper

The *ConstantDumper* process configuration contains single component, the *ConstantsDumper* which just prints all defined global constants as text. This can be used to experiment with expressions.

Even though the *ConstantsDumper* do not need any L-systems to prints constants (L-systems are actually ignored), process system of the library can process only L-systems. To over come this restriction with no effort the *Constants* L-system is defined to be used in the process statement as follows:
`process Constants with ConstantDumper`

```
configuration ConstantDumper {
    component Dumper typeof ConstantsDumper;
}

abstract lsystem Constants { }
```

J. Components

This appendix contains list of all important[1] components with their comprehensive description. In order to save space in the printed version of the thesis the list of interfaces is available only int the web user interface.

All components are in the main project called *Malsys* int the namespace *Malsys.Processing.Components*. If some component is in sub-namespace, then the path from the main namespace is in the bracket after component type.

All listed components can be accessed in process configurations by type name or full name (type name with all namespaces). For example component called Turtle interpreter can be accessed by name `TurtleInterpreter` or full name `Malsys.Processing.Components.Interpreters.TurtleInterpreter`.

J.1 Legend

Explanation of tags which describes special properties of some members.

abstract Components marked as *abstract* can not be instantiated. They can be used in the same way as interfaces (only as container type).

run-time only Gettable properties (or callable functions) marked as *run-time only* can be get (called) only while L-system is processed (in rewrite rules or interpretation methods). Especially they can not be get (called) in L-system let or set statements.

mandatory Value of settable properties (and settable symbol properties) marked as *mandatory* must be set in L-system definition. Parameters of interpretation method marked as *mandatory* must be supplied to interpretation method.

optional Connectable properties marked as *optional* may not be connected by process configuration (by default they must be connected).

allowed multiple More components can be connected to connectable properties marked as *allowed multiple* (by default only one component can be connected).

J.2 Components

J.2.1 2D SVG renderer

Provides commands for rendering 2D image. Result is vector image in SVG (Scalable Vector Graphics, plain text XML). Result is by default compressed by GZip (svgz).

Type name SvgRenderer2D (Renderers.SvgRenderer2D)

[1] Components for debugging are omitted.

Assignable to interfaces IComponent, IProcessComponent, IRenderer, IRenderer2D

Settable properties of 2D SVG renderer

margin (accepts value or array) – Margin of result image.
Expected value: One number (or array with one number) for all margins, array of two numbers for vertical and horizontal margins or array of four numbers as top, right, bottom and left margin respectively.
Default value: 2

canvasOriginSize (accepts array) – When set it overrides measured dimensions of image and uses given values.
Expected value: Four numbers representing x, y, width and height of canvas.
Default value: none

compressSvg (accepts value) – If set to true result SBG image is compressed by GZip. GZipped SVG images are standard and all programs supporting SVG should be able to open it. GZipping SVG significantly reduces its size.
Expected value: true or false
Default value: true

scale (accepts value) – Scale of result image.
Expected value: Positive number.
Default value: 1

lineCap (accepts value) – Cap of each rendered line.
Expected value: 0 for no caps, 1 for square caps, 2 for round caps
Default value: 2 (round caps)

J.2.2 3D renderer base

Provides commands for rendering 3D scene and also some basic functionality common for all 3D renderers.

Abstract component (can not be instantiated)

Type name BaseRenderer3D (Renderers.BaseRenderer3D)

Derived components ThreeJsSceneRenderer3D

Derived interfaces IComponent, IProcessComponent, IRenderer, IRenderer3D

J.2.3 3D Three.js renderer

Provides commands for rendering 3D scene. Result is JavaScript script defining 3D scene in JavaScript 3D engine Three.js.

Type name ThreeJsSceneRenderer3D (Renderers.ThreeJsSceneRenderer3D)

Base components BaseRenderer3D

Assignable to interfaces IComponent, IProcessComponent, IRenderer, IRenderer3D

Settable properties of 3D Three.js renderer

smoothShading (accepts value) – If set to true, triangles will be shaded smoothly. This can improve quality of spheres or cylinders but it has no effect on cubes. Also it significantly reduces performance of rendering.
Expected value: true or false
Default value: false

polygonTriangulationStrategy (accepts value) – Polygon triangulation strategy.
Expected value: 0 for "fan from first point", 1 triangles with minimal angle are prioritized, 2 triangles with maximal angle are prioritized, 3 triangles with maximal distance from all other points are prioritized, 4 triangles with maximal distance from not-yet-triangulated points are prioritized
Default value: 2

cameraPosition (accepts array) – Camera position. If not set it is counted automatically.
Expected value: Array 3 numbers representing x, y and z coordinate of camera position.
Default value: counted dynamically

cameraUpVector (accepts array) – Camera up vector.
Expected value: Array 3 numbers representing x, y and z up vector of camera.
Default value: {0, 1, 0}

cameraTarget (accepts array) – Camera target. If not set it is counted automatically.
Expected value: Array 3 numbers representing x, y and z coordinate of camera target.
Default value: counted dynamically

J.2.4 Axiom provider

Provides a symbol property called Axiom which serves as an initial string of symbols of an L-system.

Type name AxiomProvider (Common.AxiomProvider)

Base components SymbolProvider

Assignable to interfaces IComponent, IProcessComponent, ISymbolProvider

Settable symbol properties of Axiom provider

axiom – Initial string of symbols. The value is provided to the connected component.

Symbols – Symbol string which is provided.

J.2.5 Constants dumper

Prints all defined constants from the global scope. To be able to use this component you will need to process some L-system with it. It is possible to define an empty L-system or you can use the Constants L-system from the standard library. The process statement may look like this: process Constants with ConstantDumper;

Type name ConstantsDumper (Common.ConstantsDumper)

Assignable to interfaces IComponent, IProcessStarter

Settable properties of Constants dumper

DumpAllConstants (accepts value) – Default behavior is to print only constants in main input. If this is set to true all constants will be printed.
Expected value: true or false
Default value: false

J.2.6 Hexagonal ASCII interpreter

Hexagonal ASCII interpreter interprets symbols as lines on hexagonal grid rendering them as text (ASCII art).

Type name HexaAsciiInterpreter (Interpreters.HexaAsciiInterpreter)

Assignable to interfaces IComponent, IInterpreter, IProcessComponent

Settable properties of Hexagonal ASCII interpreter

scale (accepts value) – Scale of result ASCII art. Value representing number of characters to draw per line.
Expected value: Positive number.
Default value: 1

horizontalScaleMultiplier (accepts value) – Horizontal scale multiplier is used to multiply number of characters per horizontal line. Default value is 2 because ordinary characters are 2 times taller than wider.
Expected value: Positive number.
Default value: 2

Connectable properties of Hexagonal ASCII interpreter

Renderer (connectable type: IRenderer) – Render for rendering of ASCII art. Connected renderer must implement ITextRenderer interface.

Interpretation methods of Hexagonal ASCII interpreter

Nothing – Symbol is ignored.
Parameters: 0

MoveForward – Moves forward (without drawing) by one tile in current direction.
Parameters: 0

DrawLine – Draws line (from characters) in current direction.
Parameters: 0

TurnLeft – Turns left by 60 degrees.
Parameters: 0

TurnRight – Turns right by 60 degrees.
Parameters: 0

TurnAround – Turns by 180 degrees.
Parameters: 0

StartBranch – Saves current state (on stack).
Parameters: 0

EndBranch – Loads previously saved state (returns to last saved position).
Parameters: 0

J.2.7 Inner L-system iterator

Specialized iterator for iterating inner L-systems. Axiom is directly in iterator as property to optimize number of components. AxiomProvider property is ignored.

Type name InnerLsystemIterator (RewriterIterators.InnerLsystemIterator)

Base components MemoryBufferedIterator

Assignable to interfaces IComponent, IIterator, IProcessComponent, IProcessStarter, ISymbolProvider

Gettable properties of Inner L-system iterator

currentIteration *run-time only* (returns value) – Number of current iteration. Zero is axiom (no iteration was done), first iteration have number 1 and last is equal to number of all iterations specified by Iterations property.

iterations, i *run-time only* (returns value) – Number of iterations to do with current L-system.

Settable properties of Inner L-system iterator

iterations, i (accepts value) – Number of iterations to do with current L-system.
Expected value: Non-negative number representing number of iterations.
Default value: 0

interpretEveryIteration (accepts value) – If set to true iterator will send symbols from all iterations to connected interpret. Otherwise only result of last iteration is interpreted.
Expected value: true or false
Default value: false

interpretEveryIterationFrom (accepts value) – Sets interprets all iteration from given number.
Expected value: true or false
Default value: false

interpretFollowingIterations (accepts array) – Array with numbers of iterations which will be interpreted.
Expected value: Array of numbers
Default value: {} (empty array)

Settable symbol properties of Inner L-system iterator

axiom *mandatory* – Axiom is directly in iterator to optimize number of components.

Connectable properties of Inner L-system iterator

AxiomProvider *optional* (connectable type: ISymbolProvider) – To allow not connecting AxiomProvider component.

SymbolProvider *optional* (connectable type: ISymbolProvider) – Iterator iterates symbols by reading all symbols from SymbolProvider every iteration. Rewriter should be connected as SymbolProvider and rewriters's SymbolProvider should be this Iterator. This setup creates loop and iterator rewrites string of symbols every iteration. When number of iterations is set to 0 (of left default as 0) only axiom is used and this that case this property can be left unconnected.

OutputProcessor (connectable type: ISymbolProcessor) – Result string of symbols is sent to connected output processor. It should be InterpretrCaller who calls Interpreter and interprets symbols.

RandomGeneratorProvider *optional* (connectable type: RandomGeneratorProvider) – Connected RandomGeneratorProvider's random generator is rested after each iteration if iterator is configured to do that (ResetRandomAfterEachIteration property is set to true).

J.2.8 Inner L-system processor

This is special component for interpreting an L-system symbol as another L-system. The symbol is processed by newly created component system but interpretation calls are processed with all the components in the original system.

Type name LsystemInLsystemProcessor (Common.LsystemInLsystemProcessor)

Assignable to interfaces ILsystemInLsystemProcessor, IComponent

J.2.9 Interpreter caller

Process symbols by calling interpretation methods on connected interpreter. For conversion are used defined interpretation rules in current L-system.

Type name InterpreterCaller (Interpreters.InterpreterCaller)

Assignable to interfaces IComponent, IInterpreterCaller, IProcessComponent, ISymbolProcessor

Settable properties of Interpreter caller

debugInterpretation (accepts value) – True if print debug information about interpretation converting.
Expected value: true or false
Default value: false

Connectable properties of Interpreter caller

LsystemInLsystemProcessor *optional* (connectable type: ILsystemInLsystemProcessor) – Specialized component to allow interpret L-system symbol as another L-system.

J.2.10 Memory-buffered iterator

Iterates L-system from connected symbol provider with connected rewriter. Buffers symbols from rewriter in the memory.

Type name MemoryBufferedIterator (RewriterIterators.MemoryBufferedIterator)

Derived components InnerLsystemIterator

Assignable to interfaces IComponent, IIterator, IProcessComponent, IProcessStarter, ISymbolProvider

Gettable properties of Memory-buffered iterator

currentIteration *run-time only* (returns value) – Number of current iteration. Zero is axiom (no iteration was done), first iteration have number 1 and last is equal to number of all iterations specified by Iterations property.

iterations, i *run-time only* (returns value) – Number of iterations to do with current L-system.

Settable properties of Memory-buffered iterator

iterations, i (accepts value) – Number of iterations to do with current L-system.
Expected value: Non-negative number representing number of iterations.
Default value: 0

interpretEveryIteration (accepts value) – If set to true iterator will send symbols from all iterations to connected interpret. Otherwise only result of last iteration is interpreted.
Expected value: true or false
Default value: false

interpretEveryIterationFrom (accepts value) – Sets interprets all iteration from given number.
Expected value: true or false
Default value: false

interpretFollowingIterations (accepts array) – Array with numbers of iterations which will be interpreted.
Expected value: Array of numbers
Default value: {} (empty array)

Connectable properties of Memory-buffered iterator

SymbolProvider *optional* (connectable type: ISymbolProvider) – Iterator iterates symbols by reading all symbols from SymbolProvider every iteration. Rewriter should be connected as SymbolProvider and rewriters's SymbolProvider should be this Iterator. This setup creates loop and iterator rewrites string of symbols every iteration. When number of iterations is set to 0 (of left default as 0) only axiom is used and this that case this property can be left unconnected.

AxiomProvider (connectable type: ISymbolProvider) – Axiom provider component provides initial string of symbols. All symbols are read at begin of processing.

OutputProcessor (connectable type: ISymbolProcessor) – Result string of symbols is sent to connected output processor. It should be InterpretrCaller who calls Interpreter and interprets symbols.

RandomGeneratorProvider *optional* (connectable type: RandomGeneratorProvider) – Connected RandomGeneratorProvider's random generator is rested after each iteration if iterator is configured to do that (ResetRandomAfterEachIteration property is set to true).

J.2.11 Random generator provider

This component offers both, random and pseudo-random generators. It provides a callable function called random which can be called even in the L-system definition (not only at run-time). The pseudo-random number generator is used by default. If no random seed is set it will be generated randomly and a message with its value will be sent to the user to be possible to reproduce generated result. Truly-random generation should be used only by experienced users because other

components will not be able to measure a generated model and results may be strange.

Type name RandomGeneratorProvider (Common.RandomGeneratorProvider)

Assignable to interfaces IComponent

Gettable properties of Random generator provider

trueRandom *run-time only* (returns value) – If set to true as random generator will be used true-random (cryptographic random) generator. For this random generator can not be set any seed and numbers are always unpredictably random. If set to false as random generator will be used pseudo-random generator.

randomSeed (returns value) – If set pseudo-random generator will generate always same sequence of random numbers. Do not work if TrueRandom property is set.

Settable properties of Random generator provider

trueRandom (accepts value) – If set to true as random generator will be used true-random (cryptographic random) generator. For this random generator can not be set any seed and numbers are always unpredictably random. If set to false as random generator will be used pseudo-random generator.
Expected value: true or false
Default value: false

randomSeed (accepts value) – If set pseudo-random generator will generate always same sequence of random numbers. Do not work if TrueRandom property is set.
Expected value: Non-negative integer.
Default value: random

Callable functions of Random generator provider

random (returns value) – Returns random value from 0.0 (inclusive) to 1.0 (exclusive).
Parameters: 0

random (returns value) – Returns random value within specified range.
Parameters: 2

1. The inclusive lower bound of the random number returned.
2. The exclusive upper bound of the random number returned.

J.2.12 Symbol fileter

Filters symbol stream.

Type name SymbolFilter (ExamplePlugin.Components.SymbolFilter)

Assignable to interfaces IComponent, IProcessComponent, ISymbolProcessor

Settable symbol properties of Symbol fileter

ignore – List of ignored symbols.

Connectable properties of Symbol fileter

Output (connectable type: ISymbolProcessor) – Components to which filtered symbols are sent.

J.2.13 Symbol provider

Standard implementation of ISymbolProvider interface. It provides all symbols set to the Symbols property regardless of the state of processing.

Type name SymbolProvider (Common.SymbolProvider)

Derived components AxiomProvider

Assignable to interfaces IComponent, IProcessComponent, ISymbolProvider

Settable symbol properties of Symbol provider

Symbols – Symbol string which is provided.

J.2.14 Symbol rewriter

Full featured symbol rewriter which rewrites symbols based on defined rewrite rules in the L-system. It is capable to rewrite symbol based all criteria of Malsys' rewrite rules. Rewriting is initiated by symbol request (by enumerator). Then rewriter takes as many symbols from connected symbol provider as is needed for rewriting the symbol. If contexts (or branches) are long it may load many symbols before returning single one.

Type name SymbolRewriter (Rewriters.SymbolRewriter)

Assignable to interfaces IComponent, IProcessComponent, IRewriter, ISymbolProvider

Settable symbol properties of Symbol rewriter

contextIgnore – List of symbols which are ignored in context checking.

startBranchSymbols – List of symbols which are indicating start of branch. This symbols should be identical to symbols which are interpreted as start branch.

endBranchSymbols – List of symbols which are indicating end of branch. This symbols should be identical to symbols which are interpreted as end branch.

Connectable properties of Symbol rewriter

SymbolProvider (connectable type: ISymbolProvider)

J.2.15 Symbols saver

Prints all processed symbols (with their parameters) as the text. Symbols are delimited with a space.

Type name SymbolsSaver (Common.SymbolsSaver)

Assignable to interfaces IComponent, IProcessComponent, ISymbolProcessor

J.2.16 Text renderer

Provides commands for rendering plain text ASCII art.

Type name TextRenderer (Renderers.TextRenderer)

Assignable to interfaces IComponent, IProcessComponent, IRenderer, ITextRenderer

J.2.17 Turtle interpreter

Turtle interpreter interprets symbols as basic 2D or 3D graphics primitives. Interpreting is always in 3D but if it is connected 2D renderer (component with interface IRenderer2D) the Z coordinate is omitted.

Type name TurtleInterpreter (Interpreters.TurtleInterpreter)

Assignable to interfaces IComponent, IInterpreter, IInterpreter2D, IInterpreter3D, IProcessComponent

Gettable properties of Turtle interpreter

origin (returns array) – Origin (start position) of "turtle".

forwardVector (returns array) – Forward vector of "turtle".

upVector (returns array) – Up vector of "turtle".

Settable properties of Turtle interpreter

origin (accepts array) – Origin (start position) of "turtle".
Expected value: Array of 2 or 3 numbers representing x, y and optionally z coordinate.
Default value: {0, 0, 0}

forwardVector (accepts array) – Forward vector of "turtle".
Expected value: Array of 3 numbers representing x, y and z coordinate.
Default value: {1, 0, 0}

upVector (accepts array) – Up vector of "turtle".
Expected value: Array of 3 constants representing x, y and z coordinate.
Default value: {0, 1, 0}

rotationQuaternion (accepts array)

initialAngle (accepts value) – Initial angle (in degrees) in 2D mode (angle in plane given by forward and up vectors).
Expected value: Number representing angle in degrees.
Default value: 0

initialLineWidth (accepts value) – Initial width of drawn line.
Expected value: Number representing width. Unit of value depends on used renderer.
Default value: 2

initialColor (accepts value or array) – Initial color of drawn line.
Expected value: Number representing ARGB color (in range from 0 to 232 - 1) or array of numbers (in range from 0.0 to 1.0) with length of 3 (RGB) or 4 (ARGB).
Default value: 0 (black)

continuousColoring (accepts value or array) – Continuous coloring gradient. If enabled all colors will be ignored and given gradient (or default gradient of rainbow) will be used to continuously color all objects.
Expected value: Boolean false disables continuous coloring, true uses default rainbow gradient to continuous coloring. Array representing color gradient can be also set (see documentation or examples for syntax).
Default value: false

reversePolygonOrder (accepts value) – Reverses order of drawn polygons from first-opened last-drawn to first-opened first-drawn. This in only valid when 2D renderer is attached (in 3D is order insignificant).
Expected value: true or false
Default value: false

tropismVector (accepts array) – Tropism vector affects drawn or moved lines to derive to tropism vector.
Expected value: Array of 3 constants representing x, y and z coordinate.
Default value: {0, 1, 0}

tropismCoefficient (accepts value) – Tropism coefficient affects speed of derivation to tropism vector.
Expected value: Number.
Default value: 0

Connectable properties of Turtle interpreter

Renderer (connectable type: IRenderer) – All render events will be called on connected renderer. Both IRenderer2D or IRenderer3D can be connected.

Interpretation methods of Turtle interpreter

Nothing – Symbol is ignored.
Parameters: 0

MoveForward – Moves forward in current direction (without drawing) by distance equal to value of the first parameter.
Parameters: 1 (1 mandatory)

1. Moved distance. (*mandatory*)

DrawForward – Draws line in current direction with length equal to value of first parameter.
Parameters: 4 (1 mandatory)

1. Length of line. (*mandatory*)
2. Width of line.
3. Color of line. Can be ARGB number in range from 0 to 232 - 1 or array with 3 (RGB) or 4 (ARGB) items in range from 0.0 to 1.0.
4. Quality of rendered line in 3D.

DrawCircle – Draws circle with center in current position and radius equal to value of the first parameter.
Parameters: 2 (1 mandatory)

1. Radius of circle. (*mandatory*)
2. Color of circle.

DrawSphere – Draws sphere with center in current position and radius equal to value of the first parameter.
Parameters: 3 (1 mandatory)

1. Radius of sphere. (*mandatory*)
2. Color of sphere.
3. Quality of sphere (number of triangles).

TurnLeft – Turns left by value of the first parameter (in degrees) (in X-Y plane, around axis Z).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Yaw – Turns left around up vector axis (default Y axis [0, 1, 0]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Pitch – Turns up around right-hand vector axis (default Z axis [0, 0, 1]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Roll – Rolls clock-wise around forward vector axis (default X axis [1, 0, 0]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

StartBranch – Saves current state (on stack).
Parameters: 0

EndBranch – Loads previously saved state (returns to last saved position).
Parameters: 0

StartPolygon – Starts to record polygon vertices (do not saves current position as first vertex). If another polygon is opened, its state is saved and will be restored after closing of current polygon.
Parameters: 3 (0 mandatory)

1. Color of polygon.
2. Stroke width.
3. Stroke color.

RecordPolygonVertex – Records current position to opened polygon.
Parameters: 0

EndPolygon – Ends current polygon (do not saves current position as last vertex).
Parameters: 0

K. Process configurations

In order to save space in the printed version of the thesis this appendix contains reference for the only process configuration called *SvgRenderer*. The reference also contains consolidated list of all members of all components used in the process configuration (default components are considered for containers).

K.1 Legend

Explanation of tags which describes special properties of some members.

abstract Components marked as *abstract* can not be instantiated. They can be used in the same way as interfaces (only as container type).

run-time only Gettable properties (or callable functions) marked as *run-time only* can be get (called) only while L-system is processed (in rewrite rules or interpretation methods). Especially they can not be get (called) in L-system let or set statements.

mandatory Value of settable properties (and settable symbol properties) marked as *mandatory* must be set in L-system definition. Parameters of interpretation method marked as *mandatory* must be supplied to interpretation method.

optional Connectable properties marked as *optional* may not be connected by process configuration (by default they must be connected).

allowed multiple More components can be connected to connectable properties marked as *allowed multiple* (by default only one component can be connected).

virtual Connections marked as virtual are not checked by compiler if they connects defined components.

K.1.1 SvgRenderer

Components

AxiomProvider AxiomProvider

RandomGeneratorProvider RandomGeneratorProvider

LsystemInLsystemProcessor LsystemInLsystemProcessor

Containers

Rewriter IRewriter (default SymbolRewriter)

Iterator IIterator (default MemoryBufferedIterator)

InterpreterCaller IInterpreterCaller (default InterpreterCaller)

Interpreter IInterpreter (default TurtleInterpreter)

Renderer IRenderer (default SvgRenderer2D)

Connections

- RandomGeneratorProvider **to** Iterator.RandomGeneratorProvider
- AxiomProvider **to** Iterator.AxiomProvider
- Iterator **to** Rewriter.SymbolProvider
- Rewriter **to** Iterator.SymbolProvider
- InterpreterCaller **to** Iterator.OutputProcessor
- LsystemInLsystemProcessor **to** InterpreterCaller.LsystemInLsystemProcessor
- Renderer **to** Interpreter.Renderer

Gettable properties of SvgRenderer

trueRandom of RandomGeneratorProvider *run-time only* (returns value) – If set to true as random generator will be used true-random (cryptographic random) generator. For this random generator can not be set any seed and numbers are always unpredictably random. If set to false as random generator will be used pseudo-random generator.

randomSeed of RandomGeneratorProvider (returns value) – If set pseudo-random generator will generate always same sequence of random numbers. Do not work if TrueRandom property is set.

currentIteration of MemoryBufferedIterator *run-time only* (returns value) – Number of current iteration. Zero is axiom (no iteration was done), first iteration have number 1 and last is equal to number of all iterations specified by Iterations property.

iterations, i of MemoryBufferedIterator *run-time only* (returns value) – Number of iterations to do with current L-system.

origin of TurtleInterpreter (returns array) – Origin (start position) of ”turtle”.

forwardVector of TurtleInterpreter (returns array) – Forward vector of ”turtle”.

upVector of TurtleInterpreter (returns array) – Up vector of ”turtle”.

Settable properties of SvgRenderer

trueRandom of RandomGeneratorProvider (accepts value) – If set to true as random generator will be used true-random (cryptographic random) generator. For this random generator can not be set any seed and numbers are always unpredictably random. If set to false as random generator will be used pseudo-random generator.
Expected value: true or false
Default value: false

randomSeed of RandomGeneratorProvider (accepts value) – If set pseudo-random generator will generate always same sequence of random numbers. Do not work if TrueRandom property is set.
Expected value: Non-negative integer.
Default value: random

iterations, i of MemoryBufferedIterator (accepts value) – Number of iterations to do with current L-system.
Expected value: Non-negative number representing number of iterations.
Default value: 0

interpretEveryIteration of MemoryBufferedIterator (accepts value) – If set to true iterator will send symbols from all iterations to connected interpret. Otherwise only result of last iteration is interpreted.
Expected value: true or false
Default value: false

interpretEveryIterationFrom of MemoryBufferedIterator (accepts value) – Sets interprets all iteration from given number.
Expected value: true or false
Default value: false

interpretFollowingIterations of MemoryBufferedIterator (accepts array) – Array with numbers of iterations which will be interpreted.
Expected value: Array of numbers
Default value: {} (empty array)

debugInterpretation of InterpreterCaller (accepts value) – True if print debug information about interpretation converting.
Expected value: true or false
Default value: false

origin of TurtleInterpreter (accepts array) – Origin (start position) of ”turtle”.
Expected value: Array of 2 or 3 numbers representing x, y and optionally z coordinate.
Default value: {0, 0, 0}

forwardVector of TurtleInterpreter (accepts array) – Forward vector of ”turtle”.
Expected value: Array of 3 numbers representing x, y and z coordinate.
Default value: {1, 0, 0}

upVector of TurtleInterpreter (accepts array) – Up vector of ”turtle”.
Expected value: Array of 3 constants representing x, y and z coordinate.
Default value: {0, 1, 0}

rotationQuaternion of TurtleInterpreter (accepts array)

initialAngle of TurtleInterpreter (accepts value) – Initial angle (in degrees) in 2D mode (angle in plane given by forward and up vectors).
Expected value: Number representing angle in degrees.
Default value: 0

initialLineWidth of TurtleInterpreter (accepts value) – Initial width of drawn line.
Expected value: Number representing width. Unit of value depends on used renderer.
Default value: 2

initialColor of TurtleInterpreter (accepts value or array) – Initial color of drawn line.
Expected value: Number representing ARGB color (in range from 0 to 232 - 1) or array of numbers (in range from 0.0 to 1.0) with length of 3 (RGB)

or 4 (ARGB).
Default value: 0 (black)

continuousColoring of TurtleInterpreter (accepts value or array) – Continuous coloring gradient. If enabled all colors will be ignored and given gradient (or default gradient of rainbow) will be used to continuously color all objects.
Expected value: Boolean false disables continuous coloring, true uses default rainbow gradient to continuous coloring. Array representing color gradient can be also set (see documentation or examples for syntax).
Default value: false

reversePolygonOrder of TurtleInterpreter (accepts value) – Reverses order of drawn polygons from first-opened last-drawn to first-opened first-drawn. This in only valid when 2D renderer is attached (in 3D is order insignificant).
Expected value: true or false
Default value: false

tropismVector of TurtleInterpreter (accepts array) – Tropism vector affects drawn or moved lines to derive to tropism vector.
Expected value: Array of 3 constants representing x, y and z coordinate.
Default value: {0, 1, 0}

tropismCoefficient of TurtleInterpreter (accepts value) – Tropism coefficient affects speed of derivation to tropism vector.
Expected value: Number.
Default value: 0

margin of SvgRenderer2D (accepts value or array) – Margin of result image.
Expected value: One number (or array with one number) for all margins, array of two numbers for vertical and horizontal margins or array of four numbers as top, right, bottom and left margin respectively.
Default value: 2

canvasOriginSize of SvgRenderer2D (accepts array) – When set it overrides measured dimensions of image and uses given values.
Expected value: Four numbers representing x, y, width and height of canvas.
Default value: none

compressSvg of SvgRenderer2D (accepts value) – If set to true result SBG image is compressed by GZip. GZipped SVG images are standard and all programs supporting SVG should be able to open it. GZipping SVG significantly reduces its size.
Expected value: true or false
Default value: true

scale of SvgRenderer2D (accepts value) – Scale of result image.
Expected value: Positive number.
Default value: 1

lineCap of SvgRenderer2D (accepts value) – Cap of each rendered line.
Expected value: 0 for no caps, 1 for square caps, 2 for round caps
Default value: 2 (round caps)

Settable symbol properties of SvgRenderer

axiom of AxiomProvider – Initial string of symbols. The value is provided to the connected component.

Symbols of AxiomProvider – Symbol string which is provided.

contextIgnore of SymbolRewriter – List of symbols which are ignored in context checking.

startBranchSymbols of SymbolRewriter – List of symbols which are indicating start of branch. This symbols should be identical to symbols which are interpreted as start branch.

endBranchSymbols of SymbolRewriter – List of symbols which are indicating end of branch. This symbols should be identical to symbols which are interpreted as end branch.

Connectable properties of SvgRenderer

SymbolProvider of SymbolRewriter (connectable type: ISymbolProvider)

SymbolProvider of MemoryBufferedIterator *optional* (connectable type: ISymbolProvider) – Iterator iterates symbols by reading all symbols from SymbolProvider every iteration. Rewriter should be connected as SymbolProvider and rewriters's SymbolProvider should be this Iterator. This setup creates loop and iterator rewrites string of symbols every iteration. When number of iterations is set to 0 (of left default as 0) only axiom is used and this that case this property can be left unconnected.

AxiomProvider of MemoryBufferedIterator (connectable type: ISymbolProvider) – Axiom provider component provides initial string of symbols. All symbols are read at begin of processing.

OutputProcessor of MemoryBufferedIterator (connectable type: ISymbolProcessor) – Result string of symbols is sent to connected output processor. It should be InterpretrCaller who calls Interpreter and interprets symbols.

RandomGeneratorProvider of MemoryBufferedIterator *optional* (connectable type: RandomGeneratorProvider) – Connected RandomGeneratorProvider's random generator is rested after each iteration if iterator is configured to do that (ResetRandomAfterEachIteration property is set to true).

LsystemInLsystemProcessor of InterpreterCaller *optional* (connectable type: ILsystemInLsystemProcessor) – Specialized component to allow interpret L-system symbol as another L-system.

Renderer of TurtleInterpreter (connectable type: IRenderer) – All render events will be called on connected renderer. Both IRenderer2D or IRenderer3D can be connected.

Callable functions of SvgRenderer

random of RandomGeneratorProvider (returns value) – Returns random value from 0.0 (inclusive) to 1.0 (exclusive).
Parameters: 0

random of RandomGeneratorProvider (returns value) – Returns random value within specified range.
Parameters: 2

1. The inclusive lower bound of the random number returned.
2. The exclusive upper bound of the random number returned.

Interpretation methods of SvgRenderer

Nothing of TurtleInterpreter – Symbol is ignored.
Parameters: 0

MoveForward of TurtleInterpreter – Moves forward in current direction (without drawing) by distance equal to value of the first parameter.
Parameters: 1 (1 mandatory)

1. Moved distance. (*mandatory*)

DrawForward of TurtleInterpreter – Draws line in current direction with length equal to value of first parameter.
Parameters: 4 (1 mandatory)

1. Length of line. (*mandatory*)
2. Width of line.
3. Color of line. Can be ARGB number in range from 0 to 232 - 1 or array with 3 (RGB) or 4 (ARGB) items in range from 0.0 to 1.0.
4. Quality of rendered line in 3D.

DrawCircle of TurtleInterpreter – Draws circle with center in current position and radius equal to value of the first parameter.
Parameters: 2 (1 mandatory)

1. Radius of circle. (*mandatory*)
2. Color of circle.

DrawSphere of TurtleInterpreter – Draws sphere with center in current position and radius equal to value of the first parameter.
Parameters: 3 (1 mandatory)

1. Radius of sphere. (*mandatory*)
2. Color of sphere.
3. Quality of sphere (number of triangles).

TurnLeft of TurtleInterpreter – Turns left by value of the first parameter (in degrees) (in X-Y plane, around axis Z).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Yaw of TurtleInterpreter – Turns left around up vector axis (default Y axis [0, 1, 0]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Pitch of TurtleInterpreter – Turns up around right-hand vector axis (default Z axis [0, 0, 1]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

Roll of TurtleInterpreter – Rolls clock-wise around forward vector axis (default X axis [1, 0, 0]) by value given in the first parameter (in degrees).
Parameters: 1 (0 mandatory)

1. Angle in degrees.

StartBranch of TurtleInterpreter – Saves current state (on stack).
Parameters: 0

EndBranch of TurtleInterpreter – Loads previously saved state (returns to last saved position).
Parameters: 0

StartPolygon of TurtleInterpreter – Starts to record polygon vertices (do not saves current position as first vertex). If another polygon is opened, its state is saved and will be restored after closing of current polygon.
Parameters: 3 (0 mandatory)

1. Color of polygon.
2. Stroke width.
3. Stroke color.

RecordPolygonVertex of TurtleInterpreter – Records current position to opened polygon.
Parameters: 0

EndPolygon of TurtleInterpreter – Ends current polygon (do not saves current position as last vertex).
Parameters: 0

Coloring

As a reward that you have have read this far you can color some L-systems! You can try to color the L-systems with the fewest number of colors so that no two adjacent cells will have same color.