

# Graficzny generator L-Systemów

## Cele projektu

Celem projektu było zapoznanie się z zagadnieniami związanymi z L-Systemami od strony teoretycznej, przegląd literatury naukowej dotyczącej tej tematyki oraz ostatecznie stworzenie aplikacji umożliwiającej generowanie grafik będących wizualizacją L-Systemów i podsumowanie naszych działań w postaci sprawozdania.

## 1. Wprowadzenie i analiza wstępna

L-Systemy są formalnym opisem szeregu przekształceń pewnego ciągu symboli w zakresie zasad lingwistyki formalnej. W standardowy opis L-Systemów wchodzi elementy: zbiór symboli terminalnych, zbiór symboli nieterminalnych, ciąg początkowych symboli będący bazą do dalszych przekształceń, zbiór produkcji.

Symbole terminalne to te, które na których nie można dokonać żadnych podstawień, które byłyby opisane z zbiorze produkcji. Symbole nieterminalne to te, które produkują inne ciągi symboli.

W rozszerzonej wersji każdemu symbolowi przyporządkowana jest również dowolna ilość parametrów. Parametry te mogą brać udział w obliczeniach matematycznych opisujących poszczególne produkcje. Każda produkcja generując symbole wyjściowe bierze pod uwagę wartości parametrów symbolu wejściowego i może przeprowadzać na nich obliczenia uwzględniające dowolne poprawne wyrażenia matematyczne a wynikowe wartości służą do opisu nowych wartości parametrów wyjściowych symboli.

Algorytm generowania wyjściowych symboli odbywa się iteracyjnie poczynając od zadeklarowanego ciągu symboli początkowych. Jeżeli istnieje produkcja generująca inny ciąg z tego symbolu wykonywane jest podstawienie. Następnie otrzymany nowy ciąg symboli traktowany jest jako wejściowy do kolejnej iteracji algorytmu.

Aby z ciągu symboli uzyskać graficzną interpretację konieczne jest przypisanie symbolom akcji do wykonania w 2 lub 3-wymiarowej przestrzeni. Akcjami tymi mogą być na przykład:

- narysowanie linii
- przesunięcie kursora rysującego w konkretnym kierunku
- narysowanie innego kształtu
- zmiana koloru rysowania
- wypełnienie kształtu
- zmiana grubości linii
- itp.

Dzięki temu po przetworzeniu produkcji wynikowy ciąg symboli interpretowany jest pod kątem wykonania kolejnych akcji i są one iteracyjnie aplikowane w przestrzeni 2 lub 3-wymiarowej tworząc ostatecznie obraz – graficzną interpretację przekształceń w danym L-Systemie.

## 2. Wykorzystana literatura

1. Przemysław Prusinkiewicz, Mark Hammel, Jim Hanan, Radomir Mech (1996) *L-systems: from the Theory to Visual Models of Plants*
2. Rozenberg G., Salomaa A. (2001), *L-systems*
3. Przemysław Prusinkiewicz, Aristid Lindenmayer (2004), *The Algorithmic Beauty of Plants*
4. H. Abelson, A. A. (1982) *Turtle geometry*. M.I.T. Press, Cambridge
5. Przemysław Prusinkiewicz, Faramarz Samavati, Colin Smith, Radosław Karwowski, *L-System Description of Subdivision Curves*,
6. T. W. Chien, H. Jurgensen. (1992) *Parameterized L systems for modelling: Potential and limitations*. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*
7. P. Prusinkiewicz. (1986) *Graphical applications of L-systems*. In *Proceedings of Graphics Interface*
8. M. Zamir (2001) *Arterial Branching within the Confines of Fractal L-System Formalism*

### Odniesienie do literatury

Główne źródło informacji stanowiła książka (3) Przemysława Prusinkiewicza i Aristida Lindenmayera - *The Algorithmic Beauty of Plants*, w której znaleźliśmy wiele potrzebnych informacji przedstawionych w rzetelny i uporządkowany sposób oraz artykuł (1) *L-Systems: From the theory to visual models of plants*. Dały one podstawę teoretyczną do dalszego zagłębiania się w tematykę L-Systemów. Formalny opis zaimplementowanych L-Systemów jest zgodny z tym, które podają źródła.

Poznaliśmy konkretne przykłady L-Systemów jako formalne opisy gramatyki wraz z reprezentacją graficzną, co stanowiło punkt odniesienia w testowaniu działania projektu. Zaproponowana składnia opisu gramatyk jest jednak naszym pomysłem.

Przyjęty został sposób generowania wizualizacji jako tzw. *turtle graphics* czyli sposób tworzenia grafiki wektorowej oparty na przesuwaniu *żółwia* jako kursora rysującego oraz wykonywanie operacji geometrycznych na tym kursorze. W kolejnych krokach punkt startowy operacji jest punktem zakończenia operacji poprzedniej, zarówno jak inne parametry stanu kursora, np. rotacja. Szczegóły tego sposobu rysowania zostały opisane szczegółowo w (4) *Turtle Geometry*.

Pozostałe pozycje stanowiły dodatek rozszerzający spojrzenie na tematykę i pozwoliły przyrzeć się zaproponowanym rozwiązaniom, aplikacjom i próbom wykorzystania L-Systemów w sposób praktyczny w świecie rzeczywistym. Dały też ogłęd na podobieństwa świata rzeczywistego do wirtualnych L-Systemów i ogólnie na zależności matematyczne rządzące rzeczywistością.

Implementacja była w dużej części niezależna od technik, które można znaleźć w literaturze. W trakcie tworzenia projektu zastanawialiśmy się samodzielnie jak rozwiązać dany problem albo jaką reprezentację danych przyjąć, jak przyspieszyć obliczenia, itp. Pozycje naukowe dały jednak niezbędne podłoże do formalnego zrozumienia tematu i poznania podstawowych metod reprezentacji geometrycznej rezultatów prowadzonych symulacji. Stanowiły też źródło poprawnych przykładów, które posłużyły jako punkt odniesienia przy walidacji rezultatów i debugowaniu aplikacji.

### **3. Model**

#### **Wejście**

Wejściem modelu jest opis gramatyki w formie tekstu, zawierający:

- początkowy zbiór symboli
- zbiór produkcji
- ewentualne parametry przyporządkowane symbolom
- działania matematyczne opisujące zmiany wartości parametrów w kolejnych przekształceniach
- przypisane do symboli akcje w przestrzeni 2-wymiarowej

Oprócz tego ustalane są następujące parametry mające wpływ na przebieg symulacji:

- ilość iteracji
- punkt startowy generowania grafiki względem początku układu współrzędnych

#### **Wyjście**

Wyjściem modelu są:

- ciąg symboli wyjściowych wraz z wartościami ich parametrów
- grafika wyrenderowana w oparciu o zdefiniowane akcje geometryczne, wyświetlona bezpośrednio w aplikacji
- opcjonalnie grafika w formie pliku graficznego
- opcjonalnie plik tekstowy zawierający ostateczny ciąg symboli wraz z ich parametrami

#### **Założenia**

Wykorzystane algorytmy są całkowicie deterministyczne. Za podstawie tych samych

danych wejściowych otrzymujemy zawsze ten sam wynik. Nie występują czynniki losowe, nie odbywa się generowanie liczb pseudolosowych. Przekształcenie wejścia w wyjście modelu jest jednoznaczne, jednak z różnych opisów gramatyk można wygenerować tą samą graficzną interpretację.

### **Etapy działania**

- wprowadzenie danych, czyli wspomnianego opisu gramatyki i ewentualna zmiana dodatkowych parametrów symulacji
- zatwierdzenie wejścia
- parsowanie tekstu opisującego gramatkę, przekształcanie na reprezentację możliwą do dalszego procesowania
- przetwarzanie produkcji w kolejnych iteracjach i generowanie ciągów symboli
- analiza ostatecznego ciągu symboli pod kątem przypisanych do nich akcji graficznych
- zaaplikowanie akcji graficznych w przestrzeni 2-wymiarowej
- renderowanie grafiki
- wyświetlenie wyników

## **4. Implementacja**

### **Wykorzystane technologie**

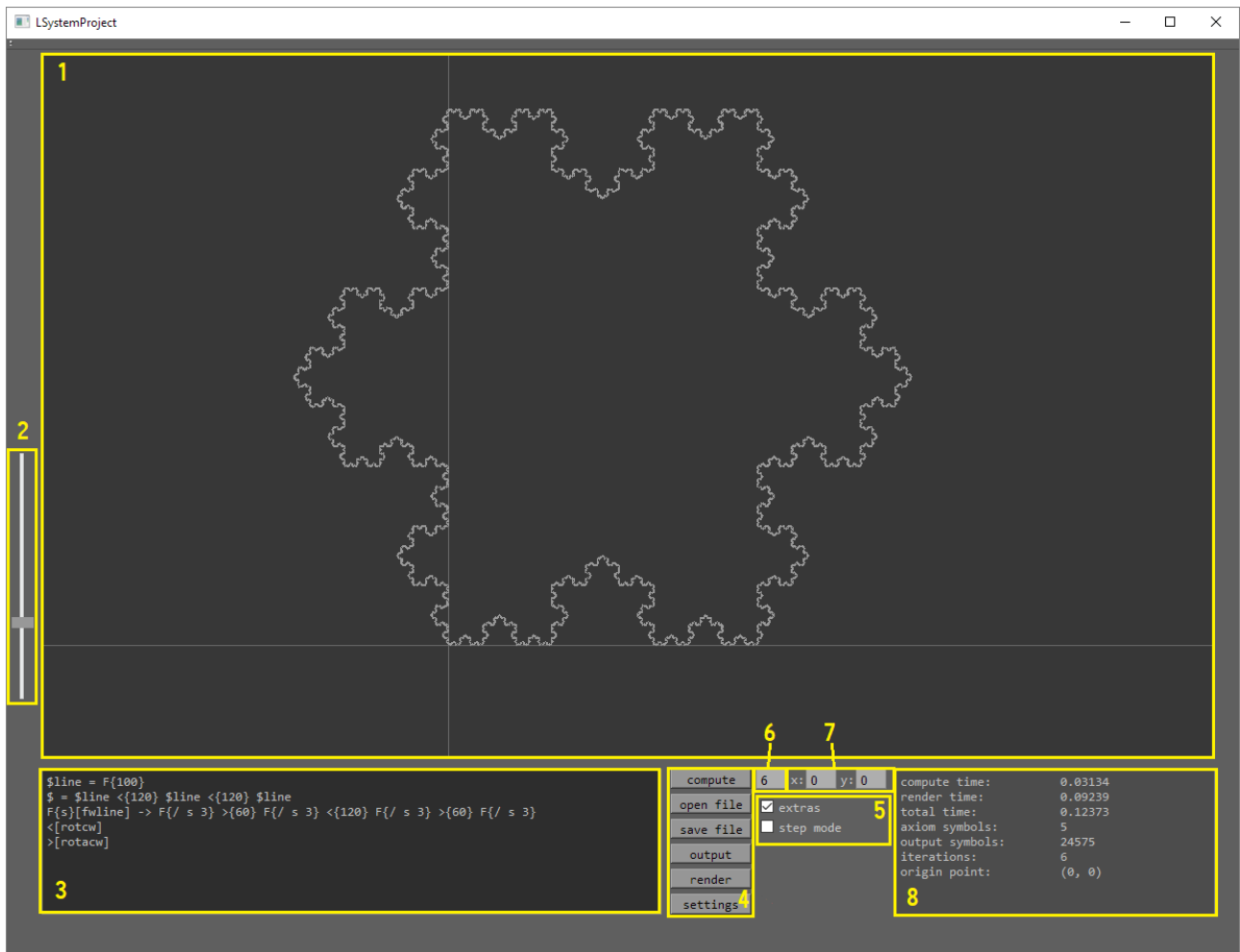
Jako że projekt wymaga dużej ilości obliczeń oraz optymalnego renderowania grafiki jako język programowania został obrany C++, gdyż pozwala na wiele niskopoziomowych zabiegów optymalizacyjnych przy jednoczesnym zachowaniu koncepcji programowania obiektowego i w konsekwencji bardziej wysokopoziomowych abstrakcji. Poza tym C++ jest bardzo powszechnie używany w przypadkach implementacji programów obliczeniowych i graficznych i daje możliwość skorzystania z szeregu bibliotek graficznych.

Wykorzystany został framework Qt, który jest jednym z najbardziej profesjonalnych i najlepiej rozwiniętych frameworków stworzonych w celu między innymi pisania interaktywnych aplikacji okienkowych. Posiada też odpowiednie narzędzia do renderingu w przestrzeni 2 i 3-wymiarowej, więc uznaliśmy, że będzie idealnie odpowiadać naszym potrzebom.

Jako IDE użyliśmy Visual Studio 2017 z wbudowanym kompilatorem C++ firmy Microsoft – Visual C++. Kod był kompilowany w zgodności ze standardem C++17. Aby połączyć funkcjonalności Visual Studio i Qt zainstalowana została wtyczka umożliwiająca tworzenie okienkowych projektów Qt bezpośrednio w Visual Studio. Do edycji okien aplikacji użyliśmy równoległego programu Qt Designer, który umożliwia graficzne planowanie okien i łączenie ich elementów z funkcjonalnościami programu.

W kodzie wykorzystaliśmy przede wszystkim dostarczone z Qt biblioteki w zakresie: obsługi tekstu, podstawowych kontenerów, wyrażeń regularnych, renderowania grafiki, obsługi plików. Do tego użyte było kilka elementów z biblioteki standardowej.

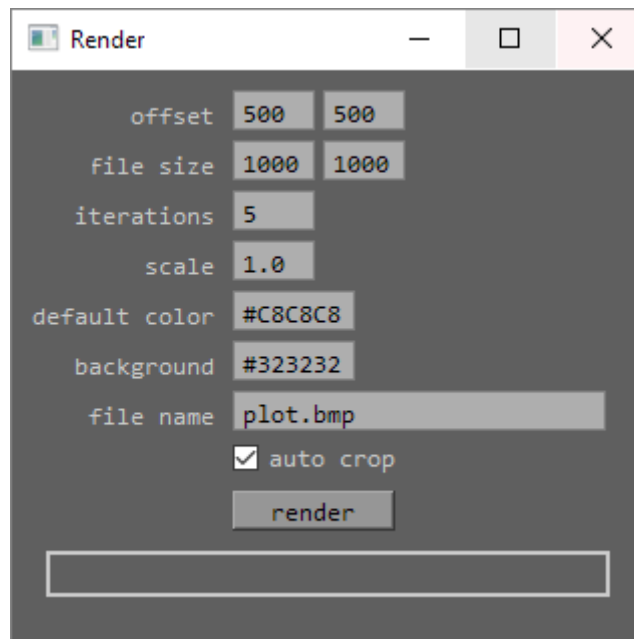
## Główne okno aplikacji



1. Główne pole, w którym wyświetlane są wyniki generowania L-Systemów. Okno Widok można przesuwąć przy pomocy lewego przycisku myszy oraz przybliżać i oddalać przy użyciu kółka myszy.
2. Pasek służący do ręcznego przybliżania i oddalania widoku.
3. Edytowalne pole tekstowe do opisu gramatyki
4. Główne przyciski do obsługi działania programu:
  - compute: uruchamia proces obliczeń i renderingu
  - open file: otwiera plik tekstowy z opisem gramatyki
  - save file: zapisuje plik tekstowy z opisem gramatyki
  - output: zapisuje plik tekstowy z wygenerowaną listą symboli wyjściowych
  - render: otwiera okno renderowania do pliku graficznego
  - settings: otwiera okno globalnych ustawień programu

5. Checkboxy ustawień:
  - extras: włącza / wyłącza rysowanie elementów pomocniczych
  - step mode: włącza / wyłącza rysowanie krok po kroku
6. Pole ustawienia ilości iteracji
7. Pole ustawienia punktu początkowego względem układu współrzędnych
8. Nieedytowalne pole z wygenerowanymi informacjami o ostatniej generacji

## Okno renderowania

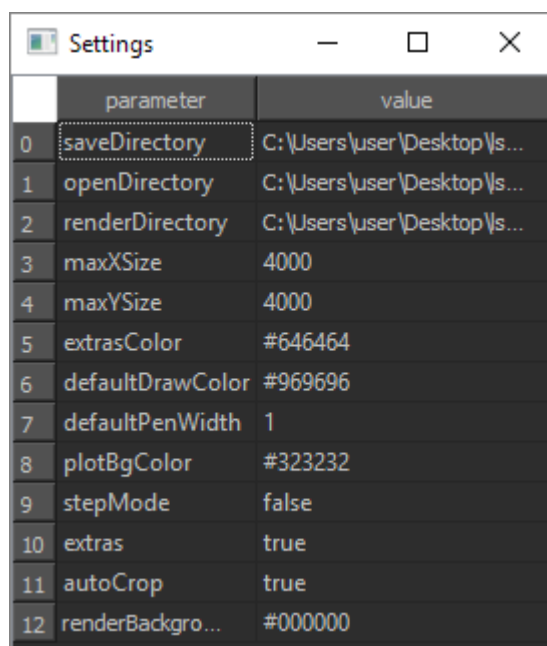


Jest otwierane po naciśnięciu przycisku renderowania. Pozwala na ustawienie kolejno:

- offset: punktu początkowego renderowania względem punktu (0,0) w pliku
- file size: rozdzielczość pliku
- iterations: ilość iteracji przy renderowaniu
- scale: mnożnik skali
- default color: kolor startowy przy rozpoczęciu rysowania
- background: kolor tła
- file name: nazwa pliku wyjściowego
- auto crop: opcja automatycznej minimalizacji rozmiaru pliku wyjściowego czyli dopasująca go do rozmiaru grafiki

Przycisk *render* rozpoczyna proces renderowania. Pasek na samym dole to pasek postępu renderowania. Pliki są zapisywane w formacie bmp.

## Okno ustawień



	parameter	value
0	saveDirectory	C:\Users\user\Desktop\ls...
1	openDirectory	C:\Users\user\Desktop\ls...
2	renderDirectory	C:\Users\user\Desktop\ls...
3	maxXSize	4000
4	maxYSize	4000
5	extrasColor	#646464
6	defaultDrawColor	#969696
7	defaultPenWidth	1
8	plotBgColor	#323232
9	stepMode	false
10	extras	true
11	autoCrop	true
12	renderBackgro...	#000000

Jest otwierane po naciśnięciu przycisku ustawień. Pozwala na ustawienie zmiennych globalnych programu:

- saveDirectory: domyślny folder zapisu plików gramatyki
- openDirectory: domyślny folder otwierania plików gramatyki
- renderDirector: domyślny folder zapisu renderowanych plików graficznych
- maxXSize: maksymalny rozmiar x przechowywanych w danym momencie danych graficznych
- maxYSize: maksymalny rozmiar y przechowywanych w danym momencie danych graficznych
- extrasColor: kolor rysowania elementów pomocniczych
- defaultDrawColor: domyślny kolor rysowania
- defaultPenWidth: domyślna grubość rysowanej linii
- plotBgColor: kolor tła w głównym obszarze rysowania
- stepMode: włączenie / wyłączenie trybu rysowania krok po kroku
- extras: włączenie / wyłączenie rysowania elementów pomocniczych
- autoCrop: włączenie / wyłączenie opcji automatycznego dopasowywania rozmiaru pliku graficznego
- renderBackground: domyślna wartość tła przy renderowaniu do pliku

## Szczegóły implementacji

### Interfejs użytkownika

Okna aplikacji zostały zaimplementowane w oparciu o framework Qt z użyciem Qt Designera. Klasy im odpowiadające dziedziczą po głównej klasie widgetów w oknach Qt – QWidget. Jest w nich zaimplementowana obsługa zdarzeń równoznacznych z interakcją użytkownika przez np. wciśnięcie przycisku. Zdarzenia obsługiwane są przez system sygnałów i slotów, będących dodatkiem do języka pochodzącym z Qt.

### Dynamiczne wyrażenia arytmetyczne

Klasa Expression i dziedziczące po niej są abstrakcją na budowane w trakcie parsowania wyrażenie odpowiadające obliczeniom wykonywanym na parametrach symboli przy procesowaniu produkcji. Constant odpowiada za wartość stałą w wyrażeniu, Variable za pojawiające się w wyrażeniu odniesienie do parametru, Function za wywołanie funkcji. Wynikowe wyrażenie arytmetyczne ma strukturę drzewiastą, reprezentującą zagnieżdżenie.

### Parsowanie

Odbywa się przez podział tekstu opisującego gramatykę na podstawie wyrażeń regularnych wyszukujących odpowiednie fragmenty. Następnie ze wstępnie podzielonego tekstu wydobywane są konkretne informacje również przy pomocy wyrażeń regularnych.

Wyrażenia arytmetyczne parsowane są metodą normalnej notacji polskiej. Za parsowanie odpowiedzialne są klasy GrammarParser, NPNExpressionParser i ProductionParser.

### Reprezentacja symboli, produkcji i gramatyk

Pojedynczy obiekt symbolu zawiera odpowiadający mu znak, liczbę parametrów oraz tablicę z parametrami. Zrezygnowaliśmy w tym przypadku z użycia gotowych kontenerów takich jak vector do przechowywania parametrów, aby w pewnym stopniu przyspieszyć obliczenia. Jeżeli symbol nie ma parametrów, tablica jest pusta.

Produkcja zawiera symbol stojący po lewej stronie oraz wynikowe symbole wraz z ewentualnymi działaniami arytmetycznymi na parametrach oraz opcjonalnie przypisaną akcję graficzną.

Klasa odpowiadająca gramatyce zawiera zbiór symboli początkowych, zbiór produkcji oraz mapę akcji graficznych. Zarówno produkcje jak i mapa akcji są 256-elementowymi tablicami, co w tym ograniczonym przypadku operowania na znakach ASCII znacznie przyspiesza obliczenia. Nie trzeba szukać odpowiednich produkcji ani akcji graficznych odpowiadających symbolom, ani używać dużo bardziej kosztownych



implementacji standardowych map. Operacje odwołania się do przypisanej akcji czy produkcji są stałymi w czasie operacjami pobrania elementu z tablicy na znanym indeksie. Mapa akcji geometrycznych jest tworzona dla konkretnej gramatyki na etapie parsowania.

## Memory Pool

Została wykorzystana koncepcja Memory Pool, czyli alokowania zawczasu większej ilości pamięci do późniejszego wykorzystania. Pamięć ta jest używana do przechowywania parametrów symboli, gdyż są to tablice dynamiczne, a operacje dynamicznej alokacji i dealokacji pamięci są stosunkowo kosztowne. Decyzja ta była kierowana bardzo dużą liczbą generowanych symboli, z których większość potrzebowała dynamicznej alokacji pamięci. Ten krok spowodował około dwukrotne przyspieszenie obliczeń.

## Rysowanie

Klasa Painter jest odpowiedzialna za rysowanie generowanych grafik, dziedziczy po QPainter i zawiera metody dla odpowiednich akcji graficznych, w których wykorzystuje narzędzia i funkcje rysujące dostarczone z Qt.

Każdej akcji graficznej odpowiada również pojedyncza klasa, która interpretuje tablicę parametrów danego symbolu i przekazuje je do odpowiedniej metody Paintera. Klasy te dziedziczą po bazie GeometricAction.

Zaimplementowane akcje graficzne obejmują (w nawiasach podane są odpowiadające im skróty służące do przypisywania do symboli w opisie gramatyki:

- przesunięcie kursora naprzód o zadaną odległość (*fw*)
- translacja o wektor (*trans*)
- narysowanie linii o zadanej długości bez przesunięcia (*line*)
- narysowanie linii o zadanej długości przesuając kursor na koniec linii (*fwline*)
- zmiana grubości linii (*width*)
- zmiana koloru rysowania (*color*)
- obrót o kąt w kierunku zgodnym z ruchem wskazówek zegara (*rotcw*)
- obrót o kąt w kierunku odwrotnym do ruchu wskazówek zegara (*rotacw*)
- narysowanie okręgu o zadanym promieniu (*circle*)
- narysowanie kwadratu o zadanym boku (*square*)
- narysowanie prostokąta o zadanych bokach (*rect*)
- ustawienie pozycji (*move*)
- ustawienie na stos całego stanu Paintera (*push*)
- przywrócenie ze stosu stanu Paintera (*pop*)

## Składnia opisu gramatyk

### Symbole

Po lewej stronie produkcji zawsze pojawia się symbol postaci bezparametrowej:

`S`

lub z parametrami:

`S{a1;a2;a3}`

gdzie `S` – znak symbolu, `a1`, `a2`, `a3` – parametry.

W przypadku symboli po prawej stronie produkcji mogą mieć również działania arytmetyczne wykonane na parametrach w postaci normalnej notacji polskiej:

`S{- a1 2; * + a2 a3 1.33}`

### Produkcje

Pojedyncza produkcja jest pojedynczą linią postaci:

`A -> BCD`

gdzie `A`, `B`, `C`, `D` – symbole.

Jeśli występują parametry, produkcja może mieć postać np:

`A{a;b;c} -> B{- a b} C D{* a c; c}`

### Akcje graficzne

Symbolom mogą być przypisane pojedyncze akcje graficzne podane w kwadratowych nawiasach w miejscu, gdzie symbol pojawia się po lewej stronie produkcji, np:

`F{s}[fwline] -> F{/ s 3}`

Jeżeli na danym symbolu nie występuje produkcja można i tak przypisać akcję pomijając prawą część produkcji w następujący sposób:

`<[rotcw]`

Wartości parametrów przekazywane do symbolu po prawej stronie produkcji automatycznie są aplikowane do przypisanej akcji, jeśli jest przypisana. W przeciwnym razie są jedynie zapamiętywane i przekazywane dalej. Można zadeklarować symbol z przypisaną akcją nie podając parametrów, a później tak czy inaczej przekazać do niego parametry po prawej stronie produkcji, tak jak w powyższym przykładzie, *rotcw* jest akcją obrotu o zadany kąt w kierunku zgodnym z ruchem wskazówek zegara, więc musi przyjąć 1 parametr.

Dopuszczalne akcje graficzne zostały podane wcześniej.

### Podstawienia

Są przetwarzane na początku parsowania i mają działanie podobne do makr w C++. Sygnalizowane są zarówno przy deklaracji jak i przy użyciu symbolem `$`

`$line = F{100}`

w każdym miejscu użycia tej nazwy zostanie podstawiony adekwatny tekst, a dopiero później przetworzony przez parser.

Zastosowanie podstawień powoduje ułatwienie zapisu przy jednoczesnym zwolnieniu z konieczności przetwarzania nadmiernych produkcji.

## Początkowy ciąg symboli

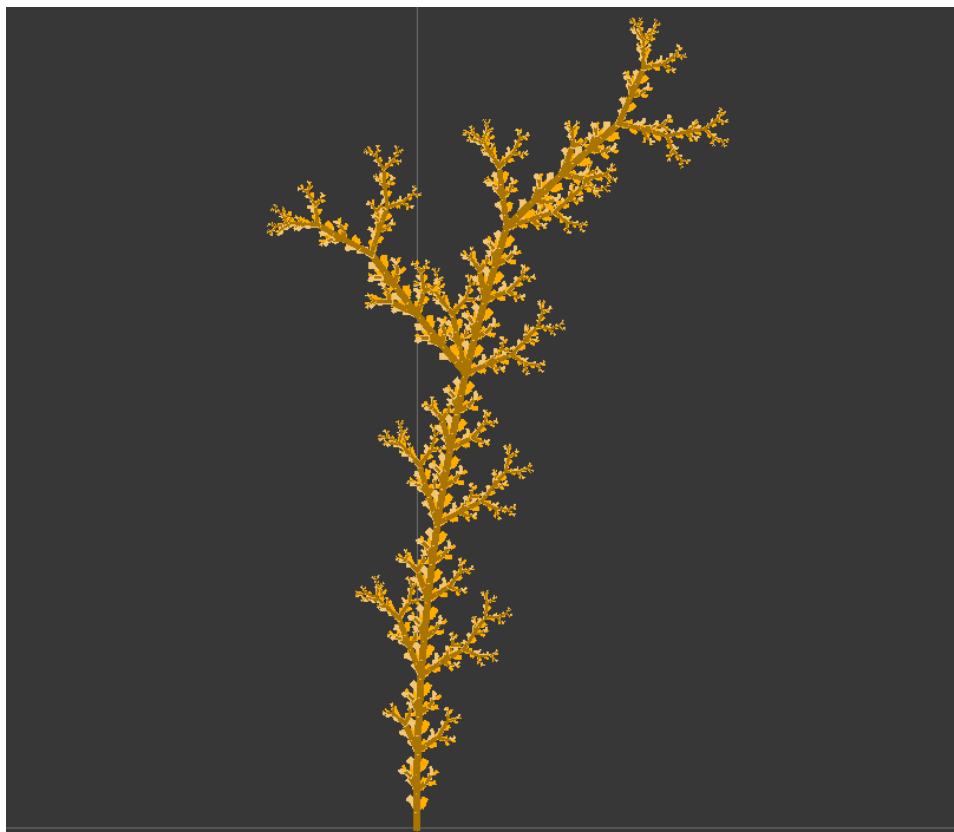
Oznaczany pustym podstawieniem, np:

```
$ = $line <{120} $line <{120} $line
```

## Pełny przykład

```
$p = #{165;110;0}  
$q = #{255;170;0}  
$r = #{240;195;110}  
$acw = <{26.7}  
$cw = >{27}  
$ = W{8}F{10} $p F{10}F{10} $acw [ $q $acw F{10}] $acw [ $r $cw  
    [ $cw F{10} $cw F{10}]]  
F{a}[fwline] -> W{* a 0.6} $p F{a}F{a} $cw [ $q $cw F{* a 0.65}  
    $acw F{* a 0.5}] $acw [ $r $acw F{* a 0.8} $acw F{* a 0.4}]  
W[width]  
#[color]  
<[rotacw]  
>[rotcw]  
[[push]  
][pop]
```

Powyższy kod powoduje wygenerowanie następującego drzewa przy 4 iteracjach:



## 5. Wyniki symulacji

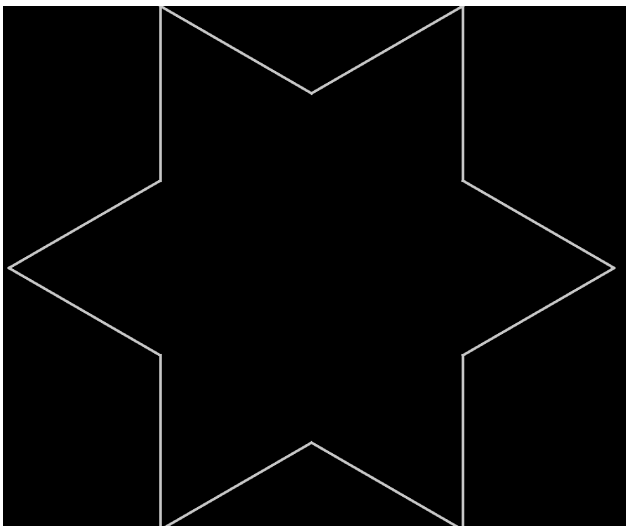
### Krzywa Kocha

```
$line = F{90}  
$ = $line <{120} $line <{120} $line  
F{s}[fwline] -> F{/ s 3} >{60} F{/ s 3} <{120} F{/ s 3} >{60} F{/ s 3}  
<[rotcw]  
>[rotacw]
```

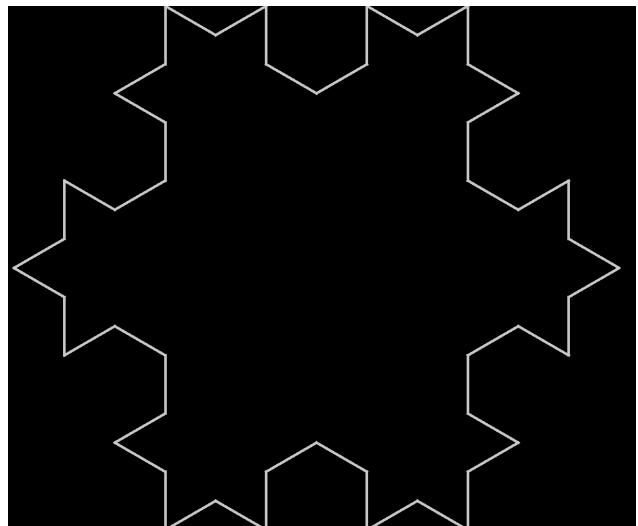
### Porównanie statystyk dla różnych ilości iteracji

compute time:	0.00005	compute time:	0.00064
render time:	0.08449	render time:	0.08781
total time:	0.08454	total time:	0.08845
axiom symbols:	5	axiom symbols:	5
output symbols:	383	output symbols:	6143
iterations:	3	iterations:	5

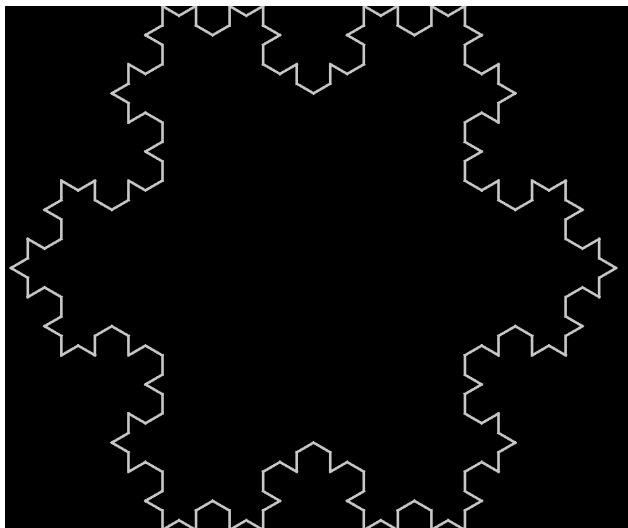
compute time:	0.04416	compute time:	0.71590
render time:	0.66304	render time:	8.94617
total time:	0.70720	total time:	9.66207
axiom symbols:	5	axiom symbols:	5
output symbols:	393215	output symbols:	6291455
iterations:	8	iterations:	10



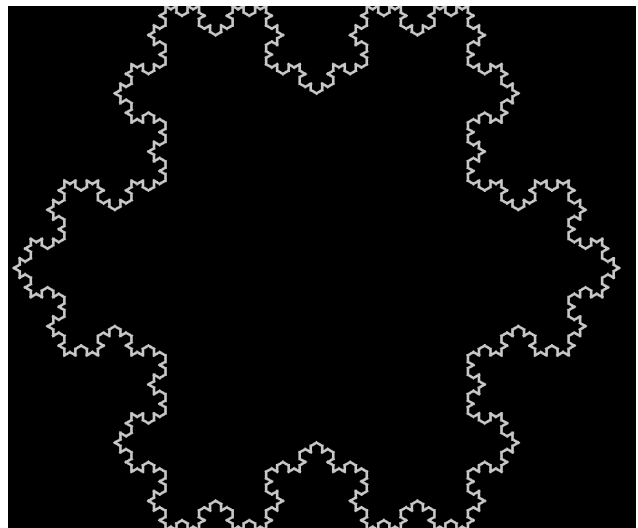
1 iteracja



2 iteracja



3 iteracja



4 iteracja

### Wynikowy ciąg symboli po 3 iteracji

```
F(10) >(60) F(10) <(120) F(10) >(60) F(10) >(60) F(10) >(60)
F(10) <(120) F(10) >(60) F(10) <(120) F(10) >(60) F(10) <(120)
F(10) >(60) F(10) >(60) F(10) >(60) F(10) <(120) F(10) >(60)
F(10) <(120) F(10) >(60) F(10) <(120) F(10) >(60) F(10) >(60)
F(10) >(60) F(10) <(120) F(10) >(60) F(10) <(120) F(10) >(60)
F(10) <(120) F(10) >(60) F(10) >(60) F(10) >(60) F(10) <(120)
F(10) >(60) F(10) <(120) F(10) >(60) F(10) <(120) F(10) >(60)
F(10) >(60) F(10) >(60) F(10) <(120) F(10) >(60) F(10) <(120)
F(10) >(60) F(10) <(120) F(10) >(60) F(10) >(60) F(10) >(60)
F(10) <(120) F(10) >(60) F(10)
```

### Drzewo

```
$p = #{145;90;63}
$q = #{70;195;110}
$r = #{70;195;110}
$acw = <{26.7}
$cw = >{27}
$ = W{8}F{10} $p F{10} F{10} $acw [ $q $acw F{10}] $acw [ $r $cw [ $cw
    F{10} $cw F{10}]]
F{a}[fwline] -> W{* a 0.6} $p F{a} $cw [ $q $cw F{* a 0.65} $acw F{* a
    0.5}] $acw [ $r $acw F{* a 0.8} $cw F{* a 0.4} $acw F{* a 0.4} $acw
    F{* a 0.4} $cw F{* a 0.4}]
W[width]
#[color]
<[rotacw]
>[rotcw]
[[push]
][pop]
```

## Porównanie statystyk dla różnych ilości iteracji

compute time: 0.00003  
render time: 0.08460  
total time: 0.08463  
axiom symbols: 22  
output symbols: 166  
iterations: 1

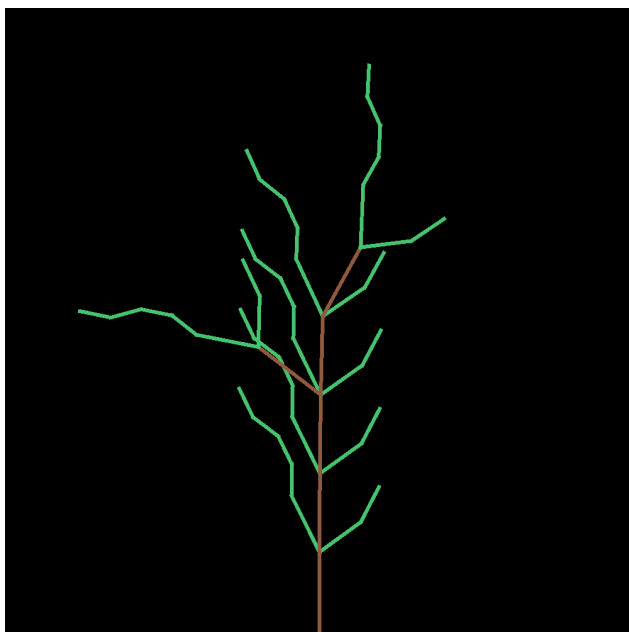
compute time: 0.00011  
render time: 0.08593  
total time: 0.08604  
axiom symbols: 22  
output symbols: 1318  
iterations: 2

compute time: 0.00082  
render time: 0.10144  
total time: 0.10226  
axiom symbols: 22  
output symbols: 10534  
iterations: 3

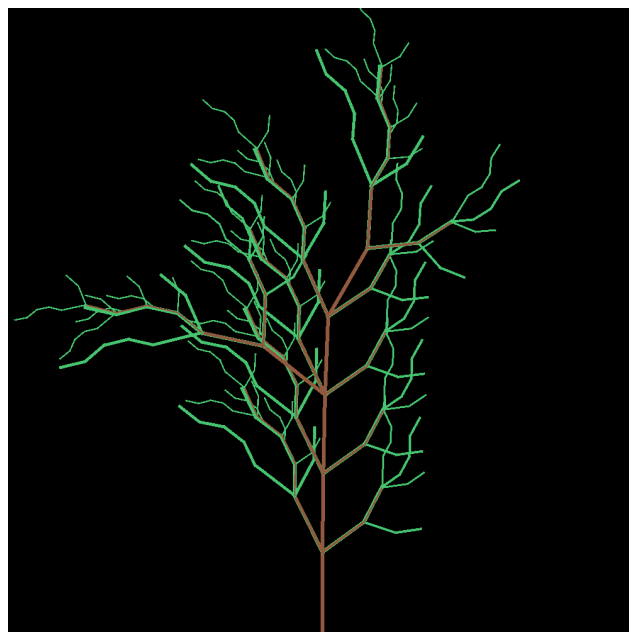
compute time: 0.00769  
render time: 0.20014  
total time: 0.20783  
axiom symbols: 22  
output symbols: 84262  
iterations: 4

compute time: 0.07510  
render time: 0.92525  
total time: 1.00035  
axiom symbols: 22  
output symbols: 674086  
iterations: 5

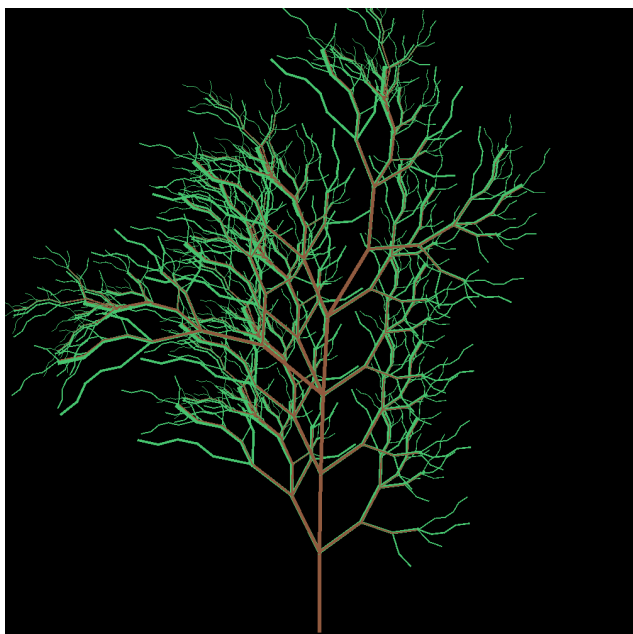
compute time: 0.50614  
render time: 6.49774  
total time: 7.00388  
axiom symbols: 22  
output symbols: 5392678  
iterations: 6



1 iteracja



2 iteracja



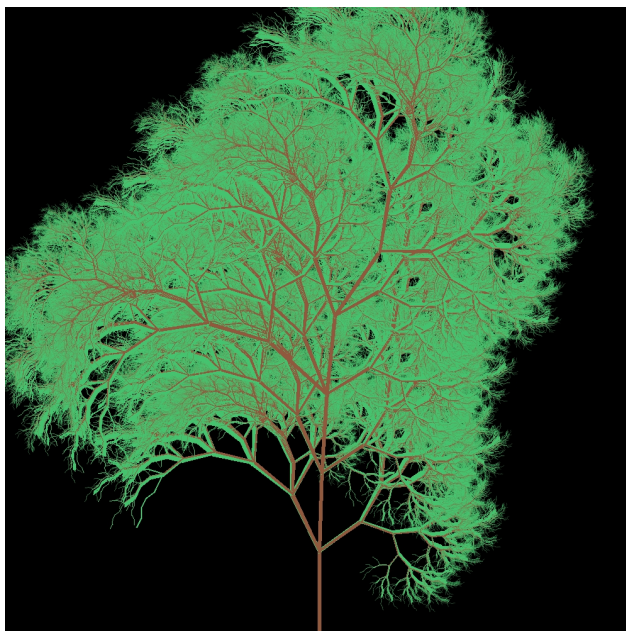
3 iteracja



4 iteracja



5 iteracja



6 iteracja

## **Komentarz do przeprowadzonych symulacji**

Wyniki działania aplikacji są zgodne z oczekiwaniami i przykładami z literatury. Renderowanie przebiega sprawnie, ale jak widać w powyższych statystykach generowanie bardziej skomplikowanych struktur wymaga do kilku sekund. Jest to jednak czas całkowicie akceptowalny. Mniej zaawansowane struktury generowane są w ciągu ułamków sekund, co sprawia, że w tych przypadkach również przybliżanie widoku jest dość płynne. Ze statystyk wynika również, że zdecydowanie więcej czasu pochłania proces renderowania niż samego przetwarzania gramatyki.

Generowanie plików graficznych dla kolejnych iteracji było ułatwione dzięki dodanemu menu renderowania. Przedstawione zostały przykłady, które naszym zdaniem są najbardziej znaczące w ocenie możliwości aplikacji. Wizualnie grafiki wynikowe nie odbiegają jakością od tych podstawowych, które można znaleźć w literaturze.

Bardzo istotnym w generowaniu drzew elementem były wprowadzone na wzór opisanych w źródłach naukowych operacje push i pop, które odpowiednio: wkładają na stos cały aktualny stan Paintera i zdejmują ze stosu ostatni stan przewracając go. Operacje te umożliwiają zapamiętanie stanu przy początku rozgałęzienia, kontynuowanie rysowania a następnie powrót do miejsca rozgałęzienia i rysowanie dalej w innym kierunku.

Statystyki pokazują również jak duża ilość symboli jest generowana stosunkowo nieskomplikowanymi opisami przy zaledwie kilku iteracjach.

## **16. Wnioski**

Cały proces pracy nad aplikacją i ostatecznie przeprowadzone symulacje dały do zrozumienia jak dużą rolę pełni optymalizacja w implementacji programów obliczeniowych. Sukcesem jest fakt, że program działa wystarczająco szybko. Nawet skomplikowane generacje nie trwają więcej niż kilka, maksymalnie kilkanaście sekund. Biorąc pod uwagę ogromną ilość przetworzonych symboli i ilość operacji, które należy wykonać w przypadku każdego z nich z pewnością czasy generowania ciągu symboli wyjściowych są zadowalające. Analizując wynikowe statystyki widzimy, że zdecydowanie dłużej od przetwarzania gramatyki trwał proces renderingu. Niestety zależy to od implementacji rysowania we frameworku Qt i nie mieliśmy na to dużego wpływu.

W trakcie tworzenia projektu pojawiło się kilka problemów optymalizacyjnych jednak w większości zostały rozwiązane. Rozwiązania optymalizacyjne zaproponowane przez nas nie są wzorowane na istniejących projektach, zostały przemyślane i zaimplementowane przez nas, więc jest to też sukcesem.



Biorąc pod uwagę nakład obliczeniowy, z pewnością język programowania został dobrany odpowiednio do problemu. Użycie frameworku Qt również przyczyniło się do ułatwienia procesu tworzenia aplikacji.

Użycie technik wielowątkowych napewno spowodowałoby jeszcze lepsze wyniki czasowe, jednak napotkaliśmy problemy w przypadku renderowania, które były związane z ograniczeniami frameworku Qt. Trudność sprawiło renderowanie wieloma wątkami na jednym obrazie. Wprowadzenie takiego usprawnienia byłoby możliwe, jednak wymagałoby znacznej ilości czasu i refaktoryzacji istniejącego kodu.

Nierozwiązanym problemem są też niektóre zabezpieczenia programu, nie byliśmy w stanie przewidzieć wszystkich możliwych scenariuszy użycia i w związku z tym nie każdy błąd został zdebugowany.

Najważniejszą trudnością tego projektu było zaplanowanie struktury programu i przede wszystkim sposobu reprezentacji danych związanych z symbolami, gramatykami, tak aby ich przetwarzanie było możliwie szybkie. Implementacja abstrakcji na wyrażenia arytmetyczne również było ciekawym zadaniem, tak jak i projektowanie składni opisu gramatyki. Proces parsowania byłby równie trudnym problemem, została zastosowana prosta metoda aby go rozwiązać. Metoda ta została opisana wcześniej i nie jest idealna, ale spełnia swoje zadanie.

Aplikację można byłoby w przyszłości rozwinąć o wspomniane techniki wielowątkowe, zastosować dodatkowe optymalizacje, zrefaktoryzować część kodu. Dodatkowymi atutami byłoby więcej statystyk i ustawień renderowania, jednak te istniejące całkowicie wystarczają do swobodnego i bezbolesnego korzystania z aplikacji i generowania struktur w bardzo szerokim zakresie. Rozwinąć można również gramatykę opisującą L-Systemy i wprowadzić produkcje kontekstowe oraz zamienić parsowanie wyrażeń arytmetycznych ze składni normalnej notacji polskiej na klasyczną.

Słowem podsumowania, projekt jest zgodny z formalnymi definicjami, pozwala na efektywne generowanie dobrej jakości grafik bazujących na L-Systemach. Po zakończeniu prac najważniejszy wydaje się udział implementacji optymalizacyjnych rozwiązań w korzyściach płynących z wykonania projektu. Ze względu na specyfikę zagadnienia stosunkowo mało czasu potrzebne było na debugowanie, co nie oznacza, że wszystkie ewentualne błędy zostały znalezione. Daje to do zrozumienia jak skomplikowane staje się zapanowanie nad rozbudowanym programem w długotrwałym procesie jego tworzenia.