



**POLITECHNIKA ŚLĄSKA**

**WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI**

**KIERUNEK TELEINFORMATYKA**

Projekt zaliczeniowy

Generowanie Fraktali IFS

Autor: Bartłomiej Gad

Kierujący pracą: dr inż. Adam Gudys

Gliwice, styczeń 2018

## Wstęp

Napisać program generujący fraktal na podstawie wyznaczników funkcji i ich prawdopodobieństw wczytywanych z pliku wejściowego.

## 1. Analiza zadania

### 1.1 Analiza problemu, podstawy teoretyczne

Zagadnienie przedstawia problem generowania fraktali IFS. IFS, czyli system funkcji iterowanych. Niniejszy projekt poświęcony jest przypadkowi generowania fraktali w przestrzeni dwuwymiarowej. Wykorzystane funkcje są przekształceniami afinicznymi płaszczyzny dwuwymiarowej wykorzystujące poniższe przekształcenia:

$$\begin{aligned}x_{n+1} &= ax_n + by_n + e \\ y_{n+1} &= cx_n + dy_n + f\end{aligned}$$

Sposób generowania fraktala z takiego zbioru funkcji przebiega następująco:

- dla aktualnego punktu (x,y) wybieramy losowo funkcję ze zbioru i wykonujemy przekształcenie otrzymując nowy punkt
- Zaznaczamy punkt na płaszczyźnie, a następnie znów losujemy funkcję i dokonujemy kolejnego przekształcenia powtarzając ten proces określoną ilość razy
- Początkowa wartość punktu jest wybierana losowo lub domyślnie startuje od (0;0)
- Oprócz zbioru funkcji definiuje się również prawdopodobieństwo z jakim dana funkcja powinna zostać wylosowana, a suma prawdopodobieństw musi być równa 1

Jednym z głównych elementów prawidłowego działania programu jest odpowiednie stworzenie bitmapy. Format BMP został zaprojektowany przez firmę Microsoft, do przechowywania obrazów tzw. Grafiki rastrowej, czyli zdjęć, wysokiej jakości tekstur i innych skomplikowanych obrazów.

Głównymi zaletami formatu BMP są:

- prostota formatu
- duża szybkość przetwarzania skomplikowanych obrazów
- powszechność formatu na wielu platformach sprzętowych

Wady formatu BMP:

- duża pamięciożerność
- mała lub brak kompresji obrazu wewnątrz pliku

Istnieją cztery rodzaje plików BMP: pliki 1, 4, 8 i 24 bitowe. W tym programie będziemy produkować bitmapę 8-bitową.

Implementacja struktury pliku BMP w języku C++ wykonana jest w następujący sposób:

- **BITMAPFILEHEADER** – jest to struktura obejmująca parametry nagłówka pliku BMP
- **BITMAPINFOHEADER** – struktura obejmująca parametry bitmapy
- **RGBQUAD** – tablica kolorów
- Tablica bajtów odwzorująca obraz w pikselach

Definicje powyższych struktur zostały zaimplementowane oraz opisane w części poświęconej specyfikacji wewnętrznej programu.

## 1.2 Struktury danych

W programie wykorzystano jednowymiarowe tablice dynamiczne bajtów, w których każdy bajt zawiera informacje na temat piksela w bitmapie. Taka struktura danych wykorzystywana będzie do przechowywania macierzy współczynników (funkcji iterowanych IFS) oraz ich prawdopodobieństw.

Taka struktura danych umożliwia łatwy dostęp do nich, a ta cecha jest ważna z uwagi na wielokrotny dostęp do danych jaki wymusza algorytm działania programu. Z racji tego, że do reprezentacji macierzy współczynników używane są tego typu struktury, konieczne jest zapewnienie odpowiedniego dostępu do elementów tablicy.

Jednowymiarowe tablice dynamiczne zostały wykorzystane ze względu na uporządkowaną postać danych alokowanych dynamicznie w pamięci komputera.

## 1.3 Algorytmy

W programie wykorzystano algorytm tworzący fraktale z funkcji iterowanych IFS. Dane wyprodukowane z fraktali IFS zapisywane są do tablicy danych, a na ich podstawie uzupełniane są informacje w wektorze danych zawierający informacje o nasyceniu koloru danego piksela.

## 2. Specyfikacja zewnętrzna

### 2.1 Obsługa programu

Żeby uruchomić program, należy ustawić odpowiedni znacznik poprzez wejście w ustawienia całego projektu i dokonanie zmiany w zakładce Debugging w linii Command Arguments. Należy wprowadzić w to pole nazwę pliku wejściowego oraz odpowiedni przełącznik, przykładowa sekwencja wygląda następująco:

*-i input.txt*

Program po wykonaniu wszystkich zaimplementowanych zadań zapisze do folderu **FractalsGenerator** plik o rozszerzeniu .bmp.

### 2.2 format danych wejściowych oraz wyjściowych

Program odczytuje dane wejściowe, w których dane umieszczone muszą być dokładnie w takiej sekwencji jak ta zadana poniżej:

8BIT //typ bitmapy

Bitmapa.bmp //nazwa pliku bitmapy

100000 // liczba iteracji

1024 //szerokość bitmapy

1024 //wysokość bitmapy

-5.0 //dane do rozdzielczości ustawiane w zależności od generowanego fraktala

5.0

10.5

0.0 //na podstawie wartości powyżej wyliczany jest rozrzut (ratio) w celu poprawy ustawienia punktów na bitmapie

0.0 0.0 0.0 0.16 0.0 0.0 0.01 // współczynniki funkcji wraz z prawdopodobieństwem podanym w ostatnim punkcie

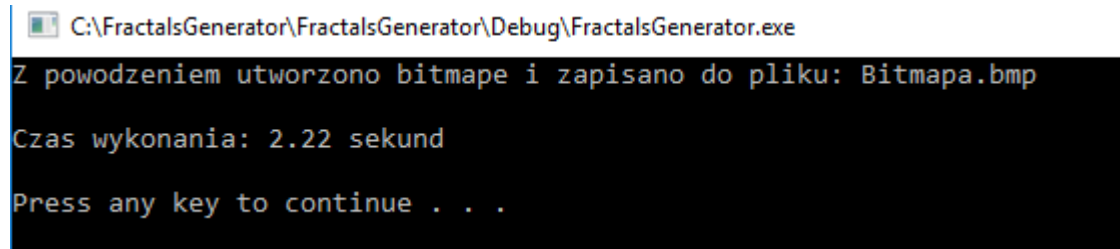
0.85 0.04 -0.04 0.85 0.0 1.6 0.85

0.2 -0.26 0.23 0.22 0.0 1.6 0.07

-0.15 0.28 0.26 0.24 0.0 0.44 0.07

Po wykonaniu algorytmu program zapisuje utworzoną bitmapę do folderu, którym znajdują się wszystkie pliki .cpp oraz nagłówkowe.

Jeśli kompilacja powiedzie się pomyślnie zostanie wyświetlony komunikat taki jak poniżej:



```
C:\FractalsGenerator\FractalsGenerator\Debug\FractalsGenerator.exe
Z powodzeniem utworzono bitmapę i zapisano do pliku: Bitmapa.bmp
Czas wykonania: 2.22 sekund
Press any key to continue . . .
```

## 2.3 Obsługa błędów

Jeżeli format danych wejściowych jest niepoprawny lub program wychwyci jakikolwiek inny błąd to kończy on swoje działanie, mówi o wykrytym błędzie i po naciśnięciu dowolnego klawisza trzeba go uruchomić ponownie.

W pierwszej kolejności dokonywane jest sprawdzenie czy zostały podane przełączniki, a jeżeli tak, to czy zostały podane poprawnie. W sytuacji gdy program wykryje jeden z wymienionych błędów zostanie wyświetlony jeden z poniższych komunikatów:

```
Bład!
FractalsGenerator: Nie podano przełączników!

Bład!
FractalsGenerator: Podano za dużo przełączników!

Bład!
FractalsGenerator: Nie podano nazwy pliku po przełączniku -i !

Bład!
FractalsGenerator: Podano niepoprawny przełącznik!
```

W momencie gdy przełączniki zostały wprowadzone poprawnie, program przechodzi do odczytania danych z pliku. W przypadku, gdy plik okaże się pusty lub dane wejściowe będą w niepoprawnym formacie zostaną wyświetlone (w zależności od wykrytego błędu) następujące komunikaty:

```
Bład!
FileParser: Plik jest pusty!

Bład!
FileParser: Błędne dane wejściowe! (bmpType)

Bład!
FileParser: Błędne dane wejściowe! (bmpType)
```

Bład!  
FileParser: Bledne dane wejscowe! (nazwa pliku wyjsciowego)

Bład!  
FileParser: Bledne dane wejscowe! (liczba iteracji)

Bład!  
FileParser: Bledne dane wejscowe! (szerokosc)

Bład!  
FileParser: Bledne dane wejscowe! (wysokosc)

Bład!  
FileParser: Bledne dane wejscowe! (minX)

Bład!  
FileParser: Bledne dane wejscowe! (maxX)

Bład!  
FileParser: Bledne dane wejscowe! (minY)

Bład!  
FileParser: Bledne dane wejscowe! (maxY)

Bład!  
FileParser: Nie podano zadnych wspolczynn timerow funkcji!

Jeżeli nie zaalokowano danych do wskaźnika typu Data zostanie wyświetlony następujący komunikat:

Bład!  
DataFactory: Wskaznik data = nullptr!

Zaimplementowano również informacje o błędach w sytuacji źle stworzonej bitmapy:

Bład!  
FractalsGenerator: Podany typ bitmapy nie jest obsługiwany!

Bład  
FractalsGenerator: Bład przy tworzeniu bitmapy!

### 3. Specyfikacja wewnętrzna

Program został napisany w języku C++, za pomocą kompilatora Microsoft Visual Studio. Zgodnie z założeniami program został napisany obiektowo.

#### 3.1 Klasy

##### Klasa BitMap

Klasa BitMap jest to klasa, która reprezentuje wszystkie dane potrzebne do stworzenia bitmapy.

```
class BitMap
{
private:
    std::size_t width_;
    std::size_t height_;
    BITMAPFILEHEADER bitMapFileHeader_;
    BITMAPINFOHEADER bitMapInfoHeader_;
    std::vector<RGBQUAD> rgbQuads;
    Data* data_;

protected:
    BITMAPFILEHEADER& getBitMapFileHeader();
    BITMAPINFOHEADER& getBitMapInfoHeader();
    std::vector<RGBQUAD>& getRgbQuads();
    Data* getData();

public:
    BitMap(const FileParser& fileParser, Data* data);
    virtual void saveBitMap(const std::string outFileName) = 0;

    virtual ~BitMap();

    std::size_t getWidth() const;
    std::size_t getHeight() const;
};
```

Klasa BitMap jest klasą abstrakcyjną, ponieważ posiada czysto wirtualną metodę saveBitMap.

Pola klasy BitMap:

```
Private:
    std::size_t width_; //szerokosc bitmapy
    std::size_t height_; // wysokosc bitmapy
    BITMAPFILEHEADER bitMapFileHeader_; //FileHeader bitmapy
    BITMAPINFOHEADER bitMapInfoHeader_; //InfoHeader bitmapy
    std::vector<RGBQUAD> rgbQuads; //wektor kolorow
    Data* data_; // wskaznik do danych bitmapy
```

```
protected:
    BITMAPFILEHEADER& getBitMapFileHeader();
    BITMAPINFOHEADER& getBitMapInfoHeader();
    std::vector<RGBQUAD>& getRgbQuads();
    Data* getData();
```

Powyżej zaimplementowano składowe informacji o tworzonej bitmapie, które używane są w klasach dziedziczących. Modyfikują one wartość.

```
public:
    BitMap(const FileParser& fileParser, Data* data); //konstruktor parametryczny

    virtual void saveBitMap(const std::string outFileName) = 0;
    virtual ~BitMap();
    std::size_t getWidth() const;
    std::size_t getHeight() const;
```

W części public klasy BitMap umieszczono czysto wirtualną metodę zapisującą BitMapę do pliku, destruktor oraz przekąźniki.

### Klasa BitMapEightBitWhiteBlack

Klasa BitMapEightBitWhiteBlack jest klasą reprezentującą bitmapę 8-bitową z 2 kolorami domyślnie zdefiniowanymi jako czarny i biały. Gdzie biały to kolor wypełnienia tła, a czarny to kolor przedstawiający wygenerowany fraktal.

```
class BitMapEightBitWhiteBlack : public BitMap
{
public:
    BitMapEightBitWhiteBlack(const FileParser& fileParser, Data* data);
    void BitMapEightBitWhiteBlack::saveBitMap(const std::string outFileName);
};
```

W tej klasie zdefiniowano konstruktor parametryczny wywołujący konstruktor parametryczny z klasy bazowej.

### Klasa Data

Klasa Data to klasa reprezentująca dane, które następnie zapisywane są w bitmapie. Zdefiniowane zostały w niej pola chronione (protected) oraz publiczne (public).

```
class Data
{
    //protected - aby było widoczne w klasie pochodnej
protected:
    std::size_t width_; //szerokość danych
    std::size_t height_; //wysokość danych

public:
    Data() : width_(0), height_(0) {} //konstruktor domyślny

    Data(const std::size_t width, const std::size_t height) : width_(width),
height_(height) {}
```



```

        virtual void setPixel(const int x, const int y, const int level) = 0;

        virtual ~Data(){}
};

```

Pola klasy Data:

```

protected:
    std::size_t width_;
    std::size_t height_;

```

W polu protected zadeklarowano wysokość danych oraz szerokość.

```

public:
    Data() : width_(0), height_(0) {}

    Data(const std::size_t width, const std::size_t height) : width_(width),
height_(height) {}

        virtual void setPixel(const int x, const int y, const int level) = 0;

        virtual ~Data(){}

```

W polu public zdefiniowano konstruktor przyjmujący jako parametry szerokość i wysokość bitmapy oraz ustawia je. Zadeklarowano również metodę czysto wirtualną co sprawia, że klasa Data, podobnie jak klasa BitMap, jest klasą abstrakcyjną. Metody czysto wirtualne deklarowane są w celu wykorzystania polimorfizmu.

Metoda setPixel jest odpowiedzialna za ustawienie danego piksela w zbiorze danych. Do metody setPixel przekazywane są parametry:

- współrzędne X, Y
- poziom nasycenia koloru Level

## Klasa DataEightBit

Klasa DataEightBit to klasa, która reprezentuje dane bajtowe (8bit) zapisywane w bitmapie 8-bitowej. W tym zbiorze danych każdy piksel reprezentowany jest przez wartość 1-bajtową. Im wyższa wartość danego bajtu, tym mocniejsze jest nasycenie koloru odpowiedzialnego za przedstawienie wygenerowanego fraktala.

```

class DataEightBit : public Data
{
private:
    std::vector<unsigned char> data_;
public:
    DataEightBit(){}
    DataEightBit(const std::size_t width, const std::size_t height);

```

```

        void setPixel(const int x, const int y, const int level);

private:
    std::vector<unsigned char>& getData();

public:

    friend std::ostream & operator<<(std::ostream &s, DataEightBit &data);};

```

Pola klasy DataEightBit:

```

private:
    std::vector<unsigned char> data_;

W polu prywatnym zdefiniowano wektor zawierający dane o poszczególnych pikselach w postaci bajtowej. Przyjmuje on wartości 0-255.

public:
    DataEightBit(){}

    DataEightBit(const std::size_t width, const std::size_t height);

    void setPixel(const int x, const int y, const int level);

    friend std::ostream & operator<<(std::ostream &s, DataEightBit &data);

```

W polu public znajduje się konstruktor parametryczny wywołujący konstruktor z klasy bazowej, który odpowiedzialny jest za rozmiar wektora data. Znajduje się tu również funkcja ustawiająca odpowiedni bit oraz przeciążony operator << zwracający strumień danych z obiektu DataEightBit.

## Klasa DataFactory

Klasa DataFactory to wzorzec fabryki static factory, czyli klasa posiadającą metodę statyczną do utworzenia produktu. Została stworzona w celu generowania różnych rodzajów danych dla bitmapy na podstawie danych z pliku. Można powiedzieć, że zostawia otwartą drogę do dalszego rozbudowywania programu.

```

class DataFactory
{
public:
    //Tworzy obiekt klasy pochodnej od Data i zwraca wskaźnik na niego typu Data*
    static Data* createData(const FileParser &fileParser);
};

```

## Klasa FileParser

Obiekt klasy FileParser zawiera informacje potrzebne do stworzenia BitMapy uzyskane z pliku wejściowego. Klasa ta dziedziczy po klasie InfoParser wszystkie pola.

```

class FileParser : public InfoParser
{
public:
    FileParser(const std::string &inFileName); }

```

## Klasa Function

Klasa Function jest odpowiedzialna za przechowywanie współczynników funkcji, które będą następnie iterowane, przechowuje równie prawdopodobieństwa. Ogólnie odpowiedzialna jest za przechowanie najważniejszych danych do wygenerowania fraktala z pliku wejściowego. Zadeklarowano w niej również konstruktor domyślny oraz konstruktor zainicjalizowany przekazanymi wartościami.

```
class Function
{
private:
    double a_; //parametr a
    double b_; // parametr b
    double c_; // parametr c
    double d_; // parametr d
    double e_; // parametr e
    double f_; //parametr f
    double probability_; //prawdopodobieństwo
public:
    Function() : a_(0), b_(0), c_(0), d_(0), e_(0), f_(0), probability_(0) {}

    Function(const double a, const double b, const double c, const double d, const
double e, const double f, const double probability) : a_(a), b_(b), c_(c), d_(d),
e_(e), f_(f), probability_(probability) {}

    double getA() const;
    double getB() const;
    double getC() const;
    double getD() const;
    double getE() const;
    double getF() const;
    double getProbability() const;
};
```

## Klasa InfoParser

Klasa InfoParser jest odpowiedzialna za przechowywanie informacji potrzebnych do wygenerowania bitmapy.

```
class InfoParser
{
public:
    enum class BMPType { ONEBIT, FOURBIT, EIGHTBIT, RGB }; //typ enum, wskazujący na
rodzaj bitmapy
protected:
    BMPType bmpType_; //typ bitmapy
    std::string outFileName_; //nazwa pliku bitmapy
    int nrIterations_; //liczba iteracji
    std::size_t width_; // szerokość bitmapy
    std::size_t height_; // wysokość bitmapy
    double minX_;
    double maxX_;
    double minY_;
    double maxY_;
    std::vector<Function> functions_; //funkcje iterowane
```

```

public:
    InfoParser(){}
    virtual ~InfoParser(){}

    BMPType getBmpType() const;
    std::size_t getWidth() const;
    std::size_t getHeight() const;
    double getMaxX() const;
    double getMaxY() const;
    double getMinX() const;
    double getMinY() const;
    std::string getOutFileName() const;
    std::vector<Function> getFunctions() const;
    int getNrIterations() const;
};

```

### Plik źródłowy DataFactory

W pliku źródłowym dataFactory zaimplementowany został generator losujący funkcje, mechanizm obliczania punktu oraz ustawienia go w danych do bitmapy.

```

random_device rd;
mt19937 gen(rd());
discrete_distribution<> d(probabilityArray.begin(), probabilityArray.end() );

```

Powyżej przedstawiony jest generator losowania funkcji.

```

for (int i = 0; i<fileParser.getHeight(); i++)
{
    for (int j = 0; j<fileParser.getWidth(); j++)
    {
        if (dataTemp[j + i*fileParser.getWidth()] > 0)
        {
            level = (log(dataTemp[j + i*fileParser.getWidth()]) / log(biggest)) * 0xff;

            data->setPixel(i, j, level);
        }
    }
}

```

Powyżej przedstawiono sposób w jaki piksel jest ustawiany w bitmapie. Zastosowano polimorfizm.

### Plik źródłowy FractalsGenerator

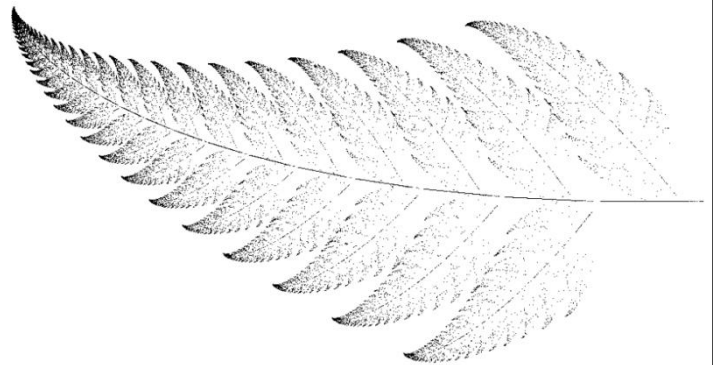
W tym pliku zaimplementowano funkcję main oraz mechanizm sprawdzający przełączniki. W pierwszej kolejności następuje weryfikacja prawidłowości przełącznika. W następnym kroku program rozpoczyna sekwencje weryfikacji danych zawartych w pliku źródłowym oraz tworzenia fraktala na ich podstawie.

## 4. Testowanie

Program został przetestowany dla różnych danych wejściowych. Dokonywano w nich modyfikacji rozdzielczości tworzonej bitmapy, ilości wykonywanych operacji oraz wygenerowano fraktale różnego typu. Poniżej przedstawione zostaną wygenerowane fraktale. Względem ilości iteracji oraz ustawionej rozdzielczości dla bitmapy zmienia się również czas tworzenia fraktala.

Pierwszym z generowanych fraktali jest liść. Aby go wygenerować wprowadzono poniższe dane do pliku wejściowego:

```
8BIT
Bitmapa1.bmp
100000
1024
1024
-5.0
5.0
10.5
0.0
0.0 0.0 0.0 0.16 0.0 0.0 0.01
0.85 0.04 -0.04 0.85 0.0 1.6 0.85
0.2 -0.26 0.23 0.22 0.0 1.6 0.07
-0.15 0.28 0.26 0.24 0.0 0.44 0.07
```



```
C:\FractalsGenerator\FractalsGenerator\Debug\FractalsGenerator.exe
Z powodzeniem utworzono bitmapę i zapisano do pliku: Bitmapa1.bmp
Czas wykonania: 2.244 sekund
Press any key to continue . . .
```

Kolejnym z wygenerowanych fraktali jest trójkąt Sierpińskiego. Został on wygenerowany z poniższych wartości. Jak widać na rysunkach wzrósł czas generowania fraktala ze względu na zwiększenie rozdzielczości bitmapy. Warto również zauważyć, że wygenerowany fraktal jest słabo widoczny, a to za sprawą większej rozdzielczości przy relatywnie niskiej ilości iteracji.

8BIT

Bitmapa2.bmp

100000

2048

2048

-1.0

1.0

-1.0

1.0

0.5 0.0 0.0 0.5 -0.5 0.5 0.3333

0.5 0.0 0.0 0.5 -0.5 -0.5 0.3333

0.5 0.0 0.0 0.5 0.5 -0.5 0.3333



```
C:\FractalsGenerator\FractalsGenerator\Debug\FractalsGenerator.exe
Z powodzeniem utworzono bitmapę i zapisano do pliku: Bitmapa2.bmp
Czas wykonania: 5.997 sekund
Press any key to continue . . .
```

Tym razem dla odmiany ustawiono niską rozdzielczość oraz bardzo dużą liczbę iteracji. Daje to efekt bardzo szczegółowego fraktala, ale przekłada się na zdecydowanie dłuższy czas jego generacji. Po przekroczeniu pewnej liczby operacji, czas trwania generacji fraktala zaczyna wzrastać wykładniczo.

8BIT

Bitmapa3.bmp

10000000

512

512

-8.0

8.0

11.0

-1.0

0.787879 -0.424242 0.242424 0.859848 1.758647 1.408065 0.895652

-0.121212 0.257576 0.151515 0.053030 -6.721654 1.377236 0.052174

0.181818 -0.136364 0.090909 0.181818 6.086107 1.568035 0.052174



C:\FractalsGenerator\FractalsGenerator\Debug\FractalsGenerator.exe

Z powodzeniem utworzono bitmapę i zapisano do pliku: Bitmapa3.bmp

Czas wykonania: 83.864 sekund

Press any key to continue . . .

## Podsumowanie

Program realizuje założenia, czyli poprawnie tworzy fraktale z zadanego pliku wejściowego o odpowiednich parametrach. Głównym problemem programu jest fakt, że wykorzystany przestaje być efektywny z coraz większą liczbą iteracji. Po przekroczeniu pewnego progu czas wykonywania programu wzrasta wykładniczo. Wpływ na czas generowania fraktala ma również ustawienie rozdzielczości w jakiej ma zostać wygenerowana bitmapa – im większa tym dłuższy czas.

Program wymagał wiedzy z dziedziny programowania w języku C++, struktury budowy bitmapy oraz przeanalizowania zasady działania algorytmu odpowiedzialnego za tworzenie fraktali.

W folderze z plikami źródłowymi oraz nagłówkowymi znajdują się również przygotowane pliki wejściowe w celu przetestowania działania programu. Program po zakończeniu działania zapisuje pliki wyjściowe do tego samego folderu.