**КОНТРОЛЬНОЕ ДОМАШНЕЕ ЗАДАНИЕ**

**ПО ТЕМЕ «ФРАКТАЛЫ»**

Пояснительная записка

Проект выполнен студентом ВШЭ ФКН ПИ группы БПИ181(1)

Степановым Евгением

Вариант №159

Москва 2018

**СОДЕРЖАНИЕ**

1. **Условие задачи3**
2. **Функции разрабатываемого приложения4**

2.1 Варианты использования4

1. **Структура приложения5**

3.1 Описание классов, их полей и методов5

3.2 Алгоритм рисования8

1. **Описание интерфейса пользователя10**
2. **Контрольный пример и описание результатов12**
3. **Сообщения пользователю16**
4. **Код программы17**
5. **Список литературы88**
6. **Приложение 189**
7. **УСЛОВИЕ ЗАДАЧИ**

Вариант №159.

Разработать оконное приложение WPF, позволяющее:

1. Отрисовывать три вида фракталов, которые перечислены в индивидуальном варианте
2. Предоставлять пользователю выбор текущего фрактала для отрисовки
3. Предоставлять пользователю возможность устанавливать количество шагов рекурсии (её глубину - количество рекурсивных вызовов). При изменении глубины рекурсии фрактал должен быть перерисован при нажатии клавишы Enter.
4. Автоматически изменять размер фрактала при изменении размеров окна. Главное окно должно быть масштабируемым.
5. Предоставлять пользователю возможность выбора начального и конечного цвета. В дальнейшем цвет каждой итерации определяется по формуле линейного градиента.
6. Сообщать о некорректном вводе данных, противоречивых или недопустимых значениях данных и других нештатных ситуациях во всплывающих окнах.
7. Сохранять полученный фрактал в изображение формата PNG
8. Изменять масштаб фрактала. Должна быть пердусмотрена возможность шестикратного увеличения
9. В программе должно быть реализовано два режима рисования: один режим, в котором фркатал рисуется мгновенно, второй режим, в котором фрактал рисуется поэтапно
10. Перемещать полученное изображение фрактала
11. В интерфейс могут быть добавлены дополнительные возможности на усмотрение разработчика

В оригинальное задание были внесены следующие уточнения:

1. Вместо автоматической перерисовки фрактала реализована перерисовка фрактала по нажатию клавишы Enter. Это связано с тем, что в приложении существует два режима рисования, один из которых может занимать достаточно много времени, поэтому автоматически запускать длительное рисование фрактала в таком режиме без прямого желания пользователя – неудобно для него.
2. При изменении размера окна фрактал не «перерисовывается», так как для перерисовки фрактала на больших уровнях рекурсии требуется больше секунды, и программа зависла бы при постоянном изменении размеров окна (в моей реализации). Вместо этого реализована адаптивность холста под размеры окна и постоянное находждение фрактала в центре этого холста.

Также при создании проекта был установлен ряд ограничений:

1. В программной реализации не использовать вспомогательные компоненты и сторонние библиотеки, не входящие в стандартную библиотеку.
2. Не использовать массивы типа object[].
3. **ФУНКЦИИ РАЗРАБАТЫВАЕМОГО ПРИЛОЖЕНИЯ**
   1. **Варианты использования**

Данное приложение может использоваться в образовательной сфере для изучения фракталов.

1. **СТРУКТУРА ПРИЛОЖЕНИЯ**
   1. **Описание классов, их полей и методов**

**Класс InitializeMainGrids** – этот класс отвечает за расположение, размер всех элементов в окне. Методы данного класса используются при открытии окна и далее вызываются при любом изменении размеров этого окна.

Поля: MainWindow – хранит ссылку на главное окно, Dispatcher – диспетчер текущего потока

Конструктор инициализирует MainWindow и задает начальные размеры окна.

Методы: DoGridInitialization инициализирует размер и позицию всех основных Grid. DrawMenuItems рисует с помощью методов DrawQCMenuFractal, DrawSCMenuFractal, DrawNFMenuFractal меню выбора фракталов. DoParamsGridInitialization инициализирует размер и позицию холста, меню настроек фрактала и меню работы с холстом.

**Класс NFFractal** наследник Fractal, реализует Н-Фрактал. В этом классе есть все необходимое для инициализации параметров фрактала и его рисования.

Поле MainCanvas – ссылка на холст, nfFractalPointsList – лист из всех элементов этого фрактала.

Конструктор инициализирует MainCanvas и вызывает конструктор базового класса (Fractals).

Метод FindDots заполняет nfFractalPointsList. Метод Draw рисует фрактал. Метод GetFractalPoints возвращает точки для следующей итерации.

**Класс QCFractal** наследник Fractal, реализует квази-клевер. В этом классе есть все необходимое для инициализации параметров фрактала и его рисования.

Поле MainCanvas – ссылка на холст, pointsList – лист из всех элементов фрактала.

Конструктор инициализирует MainCanvas и вызывает конструктор базового класса (Fractals).

Метод FindDots заполняет pointsList. Метод Draw рисует фрактал. Метод GetNewSidesDrawability показывает какие стороны фрактала рисовать во время следующей итерации. Метод GetNewPoint возвращает центр следующего эллипса (в следующей итерации)

**Класс SCFractal** наследник Fractal, реализует треугольник Серпинского. В этом классе есть все необходимое для инициализации параметров фрактала и его рисования.

Поле MainCanvas – ссылка на холст, pointsList – лист из всех элементов фрактала.

Конструктор инициализирует MainCanvas и вызывает конструктор базового класса (Fractals).

Метод InitialDraw рисует контур фрактала. Метод FindDots заполняет pointsList. Метод Draw рисует фрактал.

**Класс CreateFractalMenu –** базовый класс для классов CreateNFFractalMenu, CreateQCFractalMenu, CreateSCFractalMenu. Этот класс служит для создания базовой части меню настроек фрактала.

Свйоства: LengthTextBoxText – стартовый текст TextBox, куда вводится стартовая длина,

RecDepthBoxText – стартовый текст TextBox, куда вводится глубина рекурсии, StartColorBoxText – стартовый текст TextBox, куда вводится стартовый цвет, EndColorTextBox – стартовый тект TextBox, куда вводится конечный текст.

Метод CreateMenu создает базовые для всех фракталов части меню настройки фрактала.

**Класс CreateNFFractalMenu** – наследник CreateFractalMenu. Метод этого класса создает кнопку начала рисования фрактала.

Переопределенный метод CreateMenu создает части меню настроек фрактала, которые характерны для Н-Фрактала.

**Класс CreateQCFractalMenu** – наследник CreateFractalMenu. Метод этого класса создает кнопку начала рисования фрактала и кнопку дополнительных настроек для квази-клевера.

Переопределенный метод CreateMenu создает части меню настроек фрактала, которые характерны для квази-клевера.

**Класс CreateSCFractalMenu** - наследник CreateFractalMenu. Метод этого класса создает кнопку начала рисования фрактала.

Переопределенный метод CreateMenu создает части меню настроек фрактала, которые характерны для треугольника Серпинского.

**Класс DrawingParameters** – базовый класс для QCDrawingParams, SCDrawingParams, NFDrawingParams. Этот класс реализует тип данных, который необходим для отрисовки элемента фрактала.

Свойства: CurrentLength – текущая длина фрактала, RecursionLevel – текущий уровень рекурсии, CurrentCoords – координаты текущей точки.

Конструктор инициализирует все эти значения.

**Класс QCDrawingParams** – наследник DrawingParameters, расширяет этот тип данных, добавляя данные, присущие только квази-клеверу.

Свойства: SidesDrawability – булевский массив, показывающий какие стороны квази-клевера, нужно рисовать.

Конструктор – инициализирует это свойство и все свойства родительского класса.

**Класс NFDrawingParams** – наследник DrawingParameters, представляет собой параметры, необходимые для рисования Н-Фрактала.

Конструктор – инициализирует свойства родительского класса.

**Класс SCDrawingParams** – наследник DrawingParameters, представляет собой параметры, необходимые для рисования треугольника Серпинского.

Конструктор – инициализирует свойства родительского класса.

**Класс Fractal** – базовый абстрактный класс для QCFractal, NFFractal, SCFractal. Реализует свойства, которые характерны для всех фракталов (текущая длина рекурсии не включена в этот класс, так как она входит в DrawingParameters).

Свойства: StartLength – начальная длина фрактала, StartColor – начальный цвет линейного градиента, EndColor – конечный цвет линейного градиента, MaxRecusrionLevel – конечная, максимальная глубина рекурсии.

Конструктор инициализирует все эти свойства.

Абстрактный метод Draw который производит операцию рисования. О рисовании подробно сказано ниже.

**Класс CheckColorString** – в этот класс вынесена проверка строки цвета, вводимой пользователем.

Статическое свойство AllowedSymbol – строка из допустимых символов в объявлении цвета в шестнадцатеричной системе счисления.

Метод Check который проверяет является ли строка, введенная пользователем представлением цвета в шестнадцатеричной системе счисления.

**Класс GetCurrentColor** служит для определения цвета, в который должен быть окрашен элемент фрактала на конкретном уровне рекурсии

Метод GetDelta возвращает дельту, которая используется при расчете текущего цвета, метод Get возвращает цвет, в который должен быть покрашен текущий элемент фрактала. Цвет вычисляется по формуле линейного градиента.

**Класс PointData** –тип данных необходимый для хранения информации о каждом элементе фрактала.

Свойства: Length – длина данного элемента фрактала, CurrentBrush – цвет в который будет покрашен элемент фрактала, Coords – координаты данного элемента фрактала.

Конструктор инициализирует все эти свойства.

**Класс NFFractalPointData** наследник PointData, расширяет класс PointData для Н-Фрактала.

Свойства: PointsArr – массив точек, каждая пара точек которого представляет координаты прямой, которая является частью Н-Фрактала.

Конструктор – инициализирует это свойство и все свойства родительского класса.

**Класс StartDraw** – класс через который запускается процесс рисования всех фракталов на холсте.

Свойства: MainCanvas – ссылка на главный холст, на нем происходит рисование в режиме показа процесса рисования, TempCanvs – холст на котором рисуется фрактал при обычном режиме, CurrentFractal – номер текущего фрактала, StartColor – начальный цвет в линейном градиенте, EndColor – конечный цвет в линейном градиенте, GoBtn – ссылка на кнопку начала рисования, DispatcherPriority – приоритет диспетчера, QCSidesDrawability – булевский массив, показывающий какие стороны квази-клевера нужно рисовать.

Методы: EnableGoBtn – включает кнопку рисования, DisableGoBtn – выключает кнопку рисования, Enable – производит все операции включения, Disable – производит все операции выключения,

StartDraw – запускает рисование в одном их двух режимов, определяя это с помощью свойства DispatcherPriority.

**Класс MainWindow** – здесь реализована логика главного окна.

Поля: initialize – объект класса InitializeMainGrids, startMenuDefaultColor – начальный цвет неактивного элемента меню, endMenuDefaultColor – конечный цвет неактивного элемента меню, startMenuActiveColor – начальный цвет активного элемента меню, endMenuActiveColor – конечный цвет активного элемента меню.

Свойства: CurrentFractal – номер текущего фрактала, StartDraw – объект класса StartDraw, TokenSource – источник токена отмены рисования, CurrentScale – текущее увеличение, IsButtonEventSet – установлено ли событие клика у кнопки начала рисования, ScopeCoeff – коэффициент увеличения, MaxScopeEnlargement – максимальное увеличение, MaxLength – максимальная начальная длина текущего фрактала, InitialScrollMargin - начальный отступ scrollViewer, в котором находится холст, QCSidesDrawability – булевский массив, показывающий состояние дополнительных настроек квази-клевера, NumberOfQCSides – количество выбранных эллипсов в дополнительных настройках квази-клевера, MaxRecDepth – максимальная глубина рекурсии для текущего фрактала, MainCanvas – ссылка на текущий главный холст, CurrentLength – значение длины перед перерисовкой меню, CurrentDepth – значение глубины рекурсии перед перерисовкой меню,CurrentStartColor – значение начального цвета при перерисовке меню, CurrentEndColor – значение конечного цвета при перерисовке меню, IsDrawingGoing –булевская переменная, показывающая идет ли сейчас рисование (в обычном режиме).

Методы: EnlargeIconMouseEnter – подсвечивает иконку увеличения при наведении на нее мышкой, ReduceIconMouseEnter - подсвечивает иконку уменьшения при наведении на нее мышкой, CancelDrawingMouseEnter – подсвечивает иконку отмены рисования при наведении на нее мышкой, ExportPngFirstEnter – подсвечивает иконку экспорта фрактала, SelectEllipseMouseEnter – подсвечивает эллипс в дополнительных настройках квази-клевера при наведении мышки, MenuCanvasMouseEnter – подсвечивает иконку фрактала в меню при наведении мышкой. EnlargeIconMouseLeave, ReduceIconMouseLeave, CancelDrawingMouseLeave, ExportPngSecondLeave, SelectEllipseMouseLeave, MenuCanvasMouseLeave – возвращают своим иконкам исходный цвет, когда курсор выходит за их пределы.

Методы: MainWindowSizeChanged – меняет размер и позицию некоторых элементов при изменении размеров окна, SelectEllipseMouseDown – выбирает эллипс из дополнительных настроек квази-клевера, EnlargeIconMouseDown – увеличивает изображение на холсте, ReduceIconMouseDown – уменьшает изображение на холсте, CancelCurrentDrawingMouseDown – отменяет рисование во втором режиме (режиме показа рисования), ExportPngSecondClick – запускает процесс сохранения фрактала в PNG файл, MenuCanvasMouseDown – выбирает фрактал для отрисовки, DrawDractalEventDown – этот метод запускает рисование в первом или втором режиме, Scroll – прокручивает ползунки ScrollViewer на заданные значения, EnterKeyDown – при нажатии Enter запускает процесс рисования.

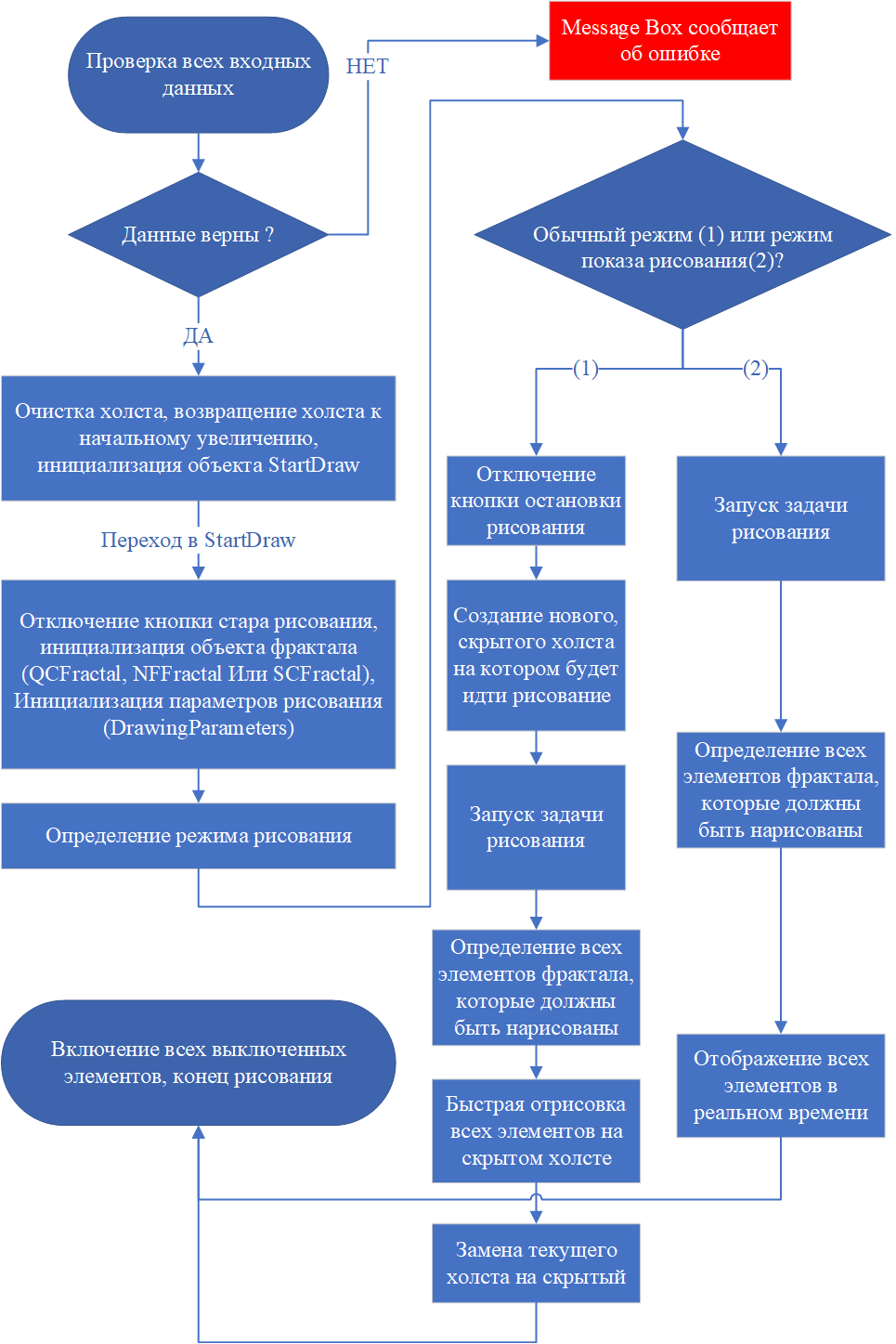
Каждому классу соответствует один файл в проекте.

* 1. **Алгоритм рисования**

На схеме ниже (Рисунок 1) представлен алгоритм рисования.

Стоит отметить, что в приложении есть два режима рисования: обычный режим, когда весь фрактал рисуется за 1-2 секунды и выводится на экран и режим показа рисования, когда пользователь может наблюдать весь процесс рисования. Кнопка отмены рисования работает только во втором режиме (режиме показа рисования).

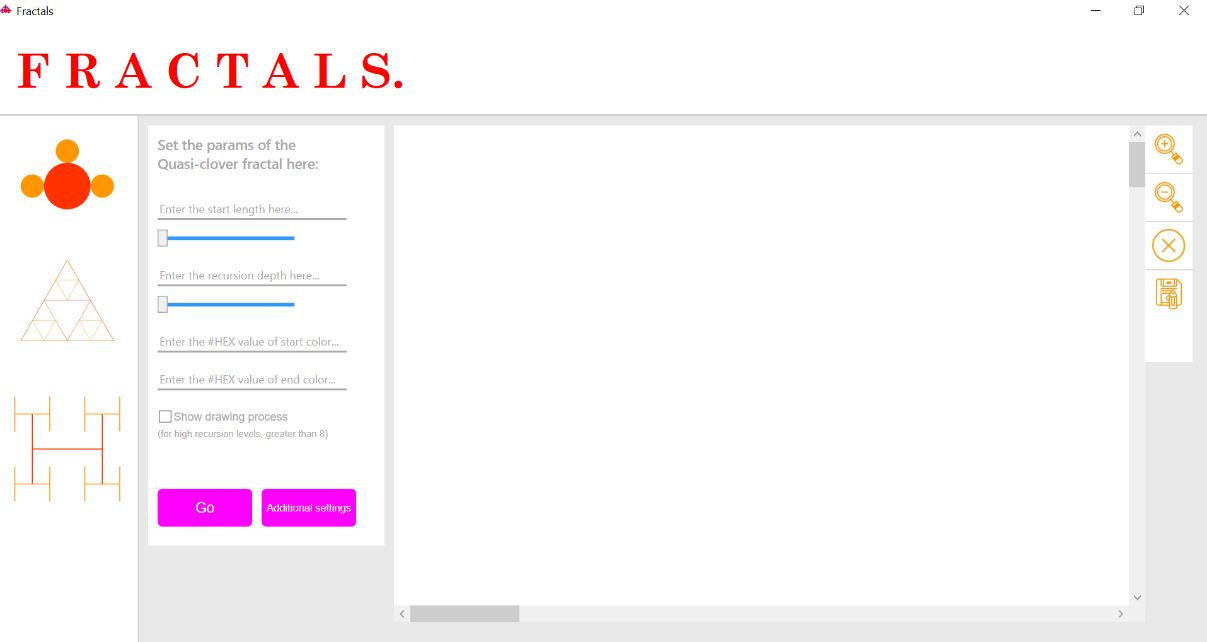
Рисунок 1 (Алгоритм рисования фрактала)



1. **ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ**

Интерфейс пользователя представлен на Рисунке 2.

Рисунок 2 (Интерфейс пользователя)



**Меню выбора фрактала**

Слева расположено меню выбора фрактала. По клику на изображение фрактала пользователь может выбрать фрактал для отрисовки.

**Меню настройки фрактала**

После клика на один из фракталов перед пользователем загрузиться меню настроек, в котором пользователь может задать начальную длину фрактала (либо с помощью ввода с клавиатуры, либо с помощью ползунка). Минимальное значение – 1, а максимальное задается для каждого фрактала, при наведении мышки на поле ввода появляется специальная подсказка, в которой указано максимальное значение.

Аналогичным образом работает ввод глубины рекурсии (поле для ввода и ползунок сразу после поля ввода и ползунка для ввода начальной длины). Ниже этих элементов расположены два поля ввода для ввода значения начального и конечного цвета в шестнадцатеричной системе счисления, которые используются при отрисовке фрактала.

Далее расположен элемент CheckBox, с помощью которого пользователь может выбрать режим рисования: прорисовывать фрактал сразу или прорисовывать фрактал, при этом показывая процесс рисования.

В конце этой панели расположены кнопки начала рисования и дополнительных настроек.

Кнопка начала рисования есть в меню настроек у всех фракталов, кнопка дополнительных настроек есть только у квази-клевера.

**Холст**

Холст занимает центральную часть экрана. На холсте рисуются все фракталы.

**Меню работы с холстом**

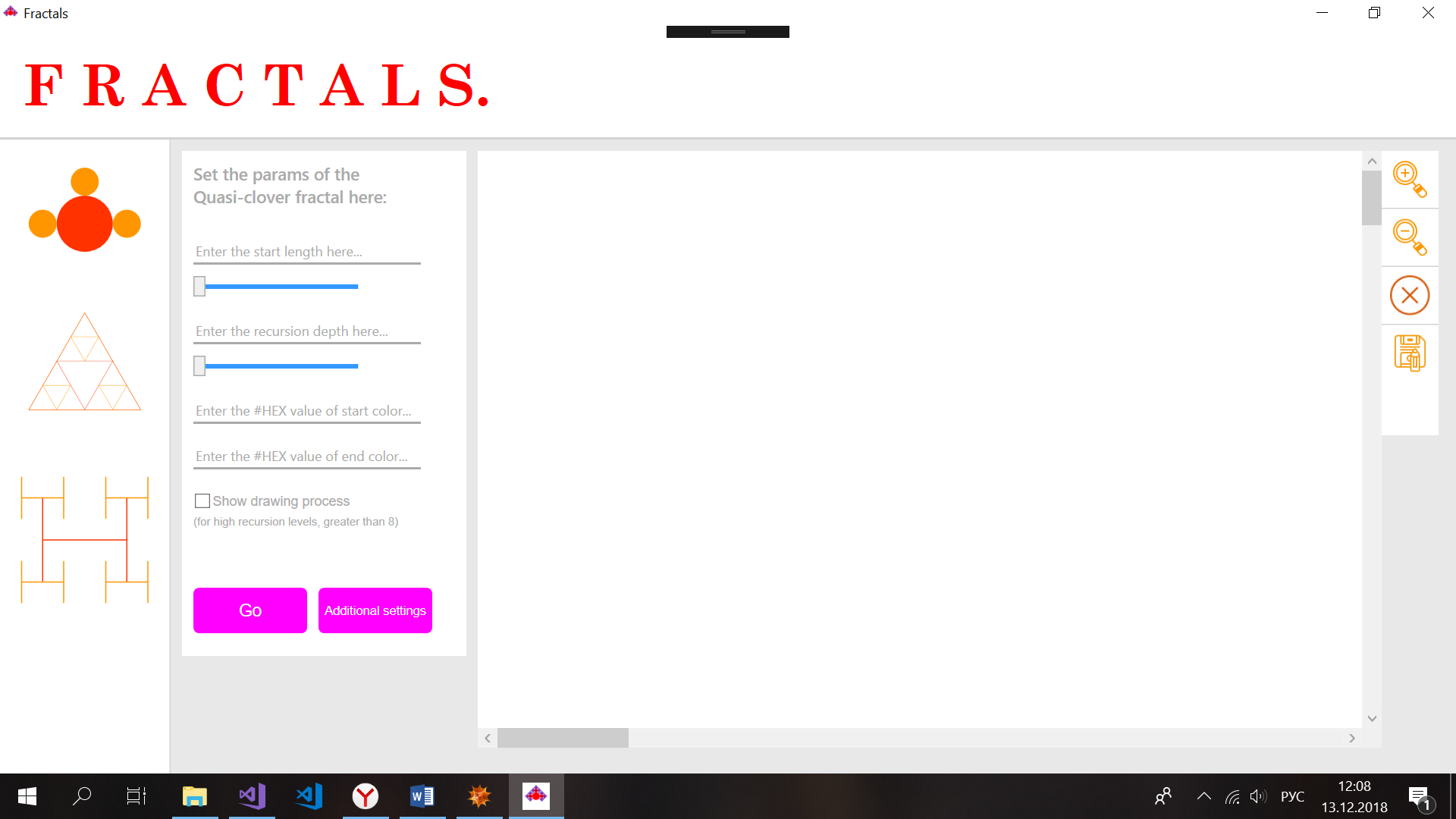
Меню работы с холстом расположено справа от холста. В этом меню есть функция увеличения фрактала (первая иконка сверху), функция уменьшения фрактала (вторая иконка сверху), функция отмены процесса рисования (третья иконка сверху, работает только в режиме показа процесса рисования) и функция сохранения фрактала в PNG файл (четвертая иконка сверху). Размер сохраненного фрактала зависит от приближения, таким образом можно выбирать подходящий размер.

1. **КОНТРОЛЬНЫЙ ПРИМЕР И ОПИСАНИЕ РЕЗУЛЬТАТОВ**

Далее описаны шаги, с помощью которых можно проверить функционал этой программы.

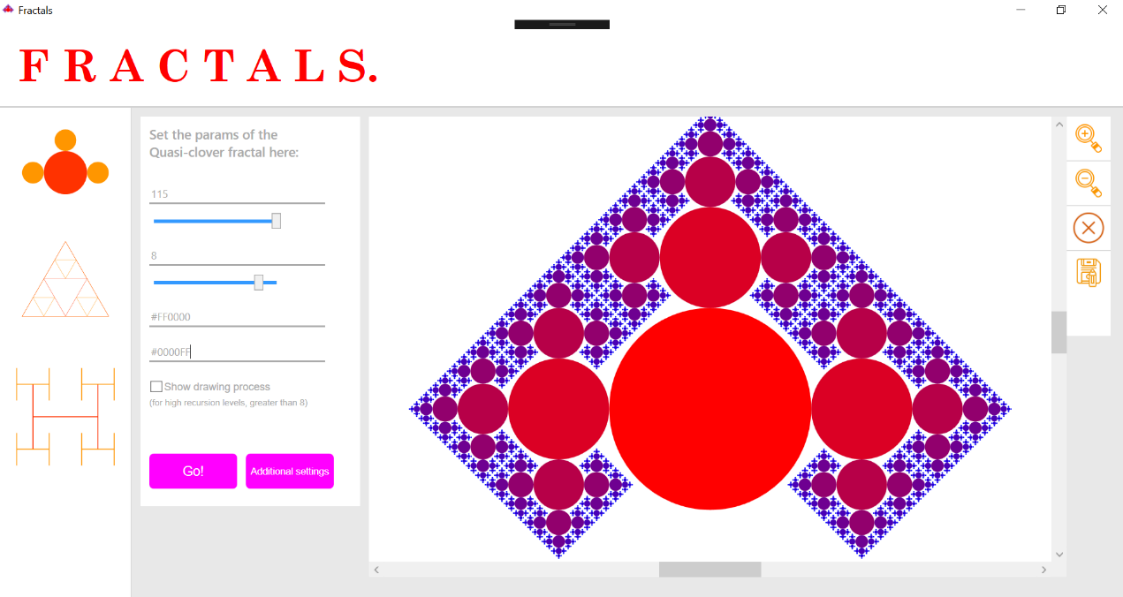
1. Запустить программу.
2. Выбрать квази-кдевер для отрисовки. Перед пользователем должна быть следующая картина (Рисунок 3):

Рисунок 3 (Выбор квази-клевера для отрисовки).



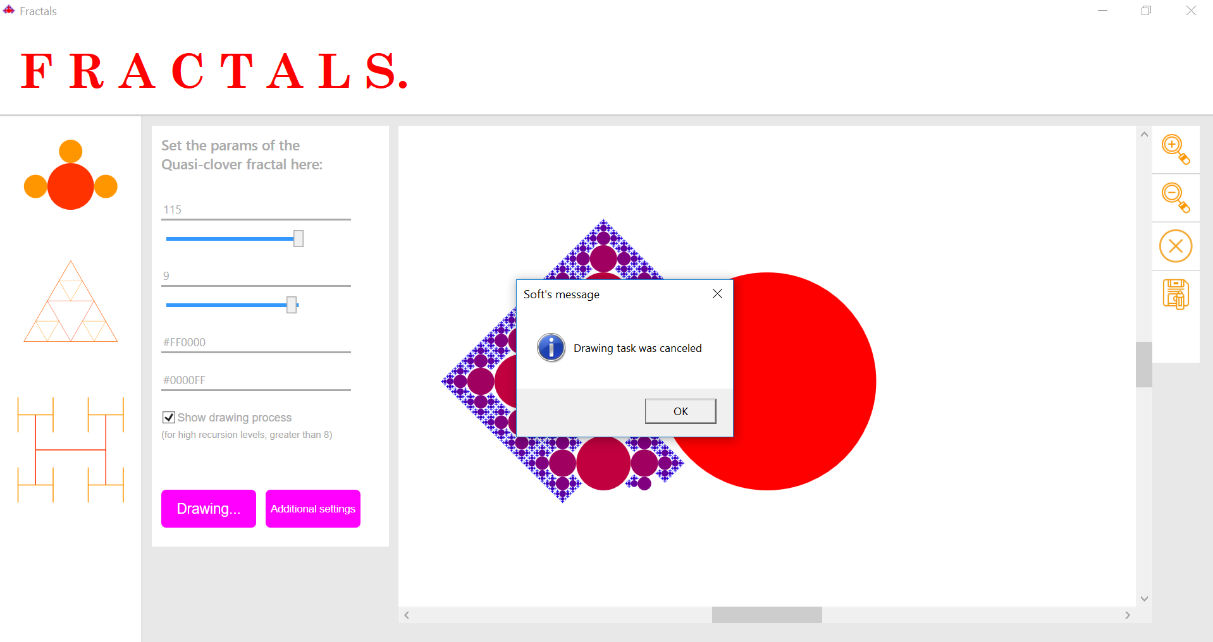
1. Задать следующие стартовые параметры: стартовая длина – 115, глубина рекурсии – 8, начальный цвет - #FF0000, конечный цвет - #0000FF.
2. Нажать кнопку “Go” или клавишу Enter. Квази-клевер будет нарисован (Рисунок 4)

Рисунок 4.



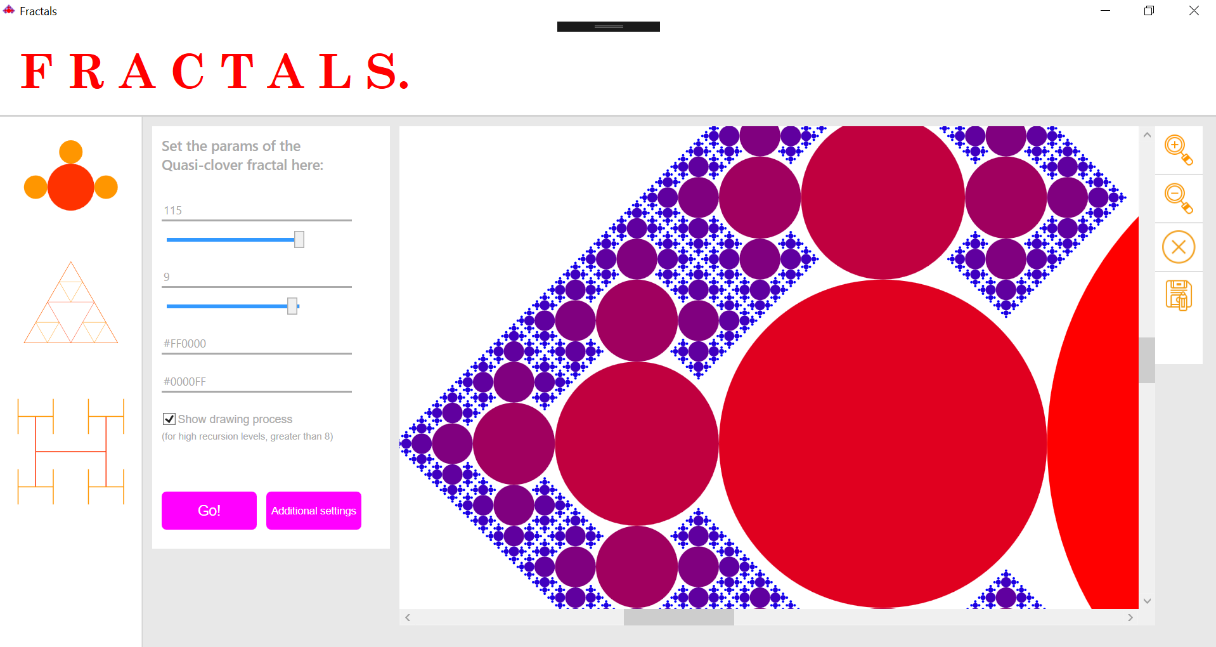
1. Нажать несколько раз на кнопки увеличения и уменьшения, так фрактал можно будет подробнее изучить.
2. Изменить глубину рекурсии на 9, и поставить галочку в CheckBox. Таким образом запускается второй режим рисования.
3. Нажать кнопку “Go” или клавишу Enter. Фрактал начнет рисоваться. В процессе рисования можно приближать и отдалять фрактал, изучая его построение. Через несколько секунд нажать кнопку отмены рисования (Рисунок 5).

Рисунок 5.



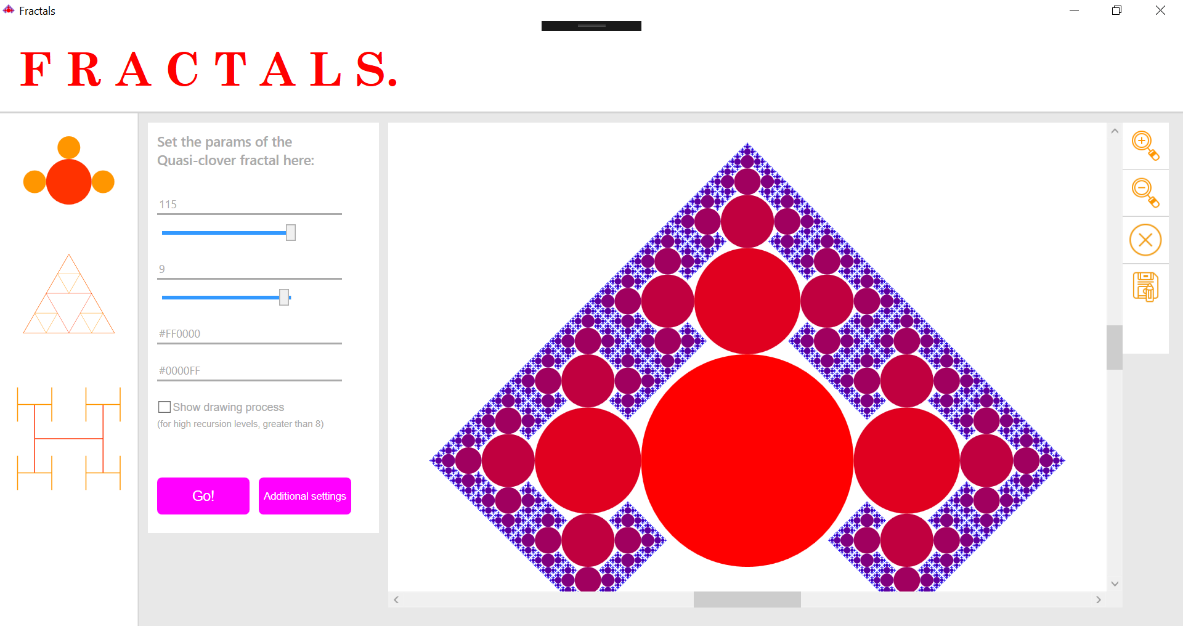
1. Опять нажать кнопки увеличения и уменьшения фрактала, изучить недорисованный фрактал (Рисунок 6).

Рисунок 6..



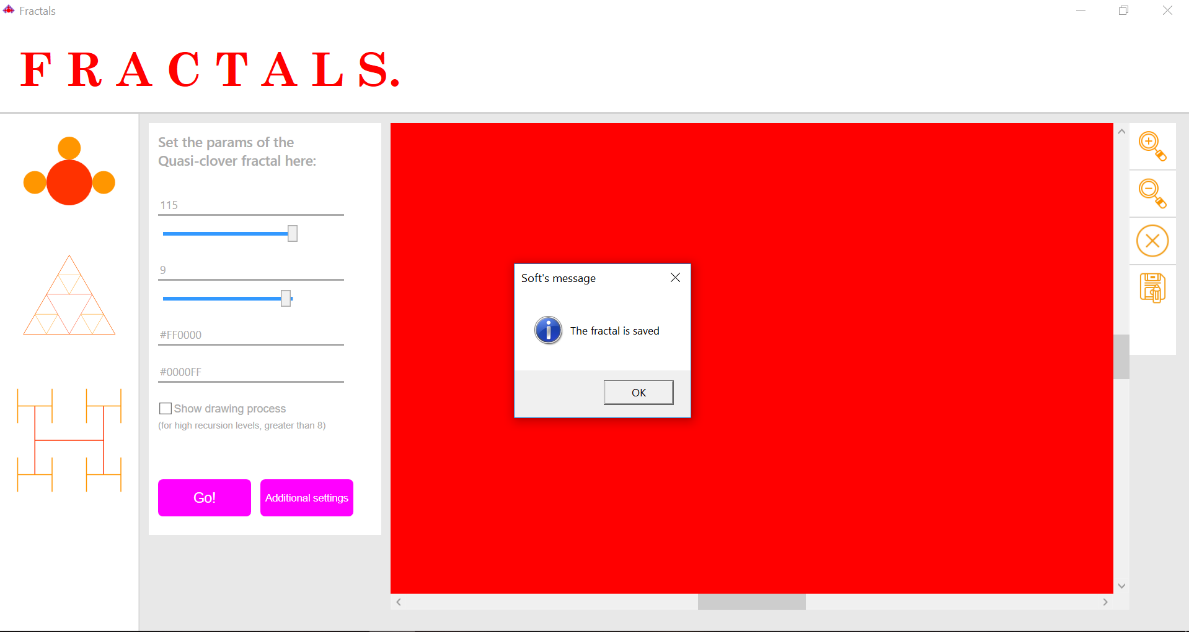
1. Убрать галочку из CheckBox и нажать Enter. Фрактал полностью нарисуется (Рисунок 7).

Рисунок 7.



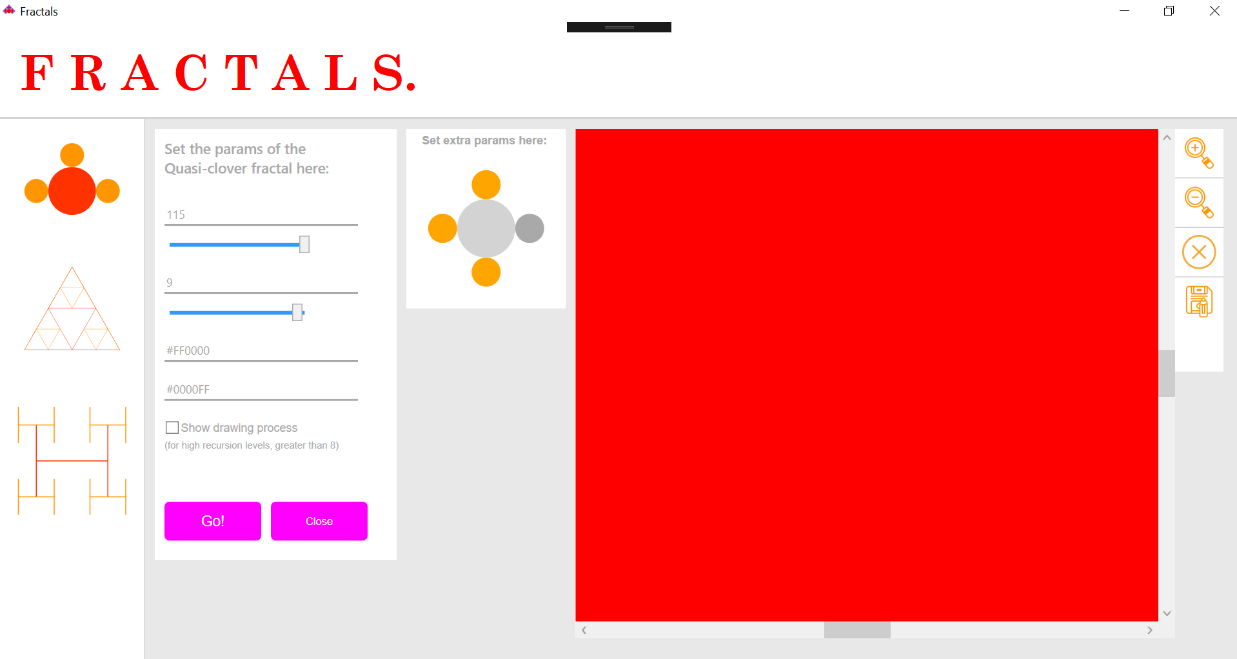
1. Увеличить фрактал до максимального значения, и нажмите кнопку сохранения. Выбрать место сохранения и сохранить фрактал. Появится сообщение об успешном сохранении (Рисунок 8).

Рисунок 8.



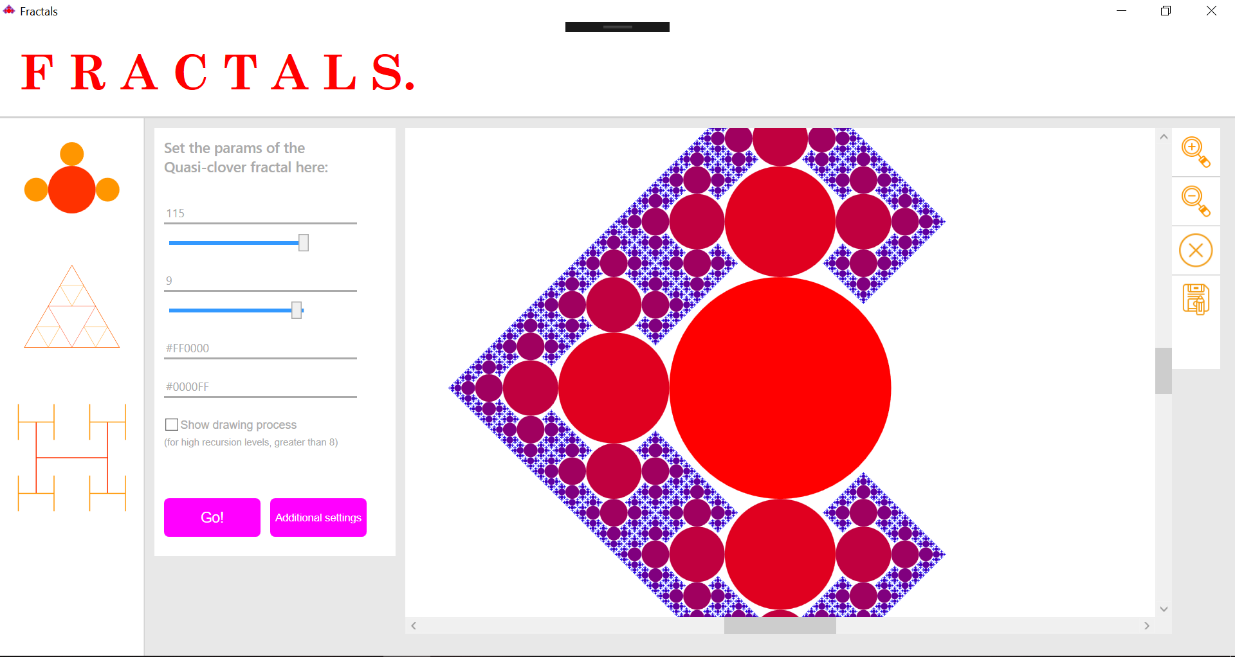
1. Нажать кнопку “Additional settings”. Выбрать другую конфигурацию фрактала (Рисунок 9).

Рисунок 9.



1. Нажать кнопку “Close”. Нажать кнопку “Go”. Появится фрактал в новой конфигурации (Рисунок 10).

Рисунок 10.



1. Пункты 2-10 повторить для треугольника Серпинского и Н-Фрактала (в случае Н-Фрактала в пункте (3) задать глубину рекурсии 6, а в пункте (6) – 7). Для лучшего изучения фрактала можно изменить начальную длину в обоих случаях (в треугольнике Серпинского рекомендуется установить начальную длину 700, а в Н-Фрактале 325).
2. **СООБЩЕНИЯ ПОЛЬЗОВАТЕЛЮ**

Ниже приведен список сообщений, которые могут появится при работе с программой.

1. При отмене задачи рисования: “Drawing task was canceled”
2. При неправильном вводе: “Error in the input data”
3. При неправильном выборе начальной конфигурации квази-клевера: “Select three ellipses to draw…”
4. При некорректном вводе данных до нажатия кнопки “Go” или клавишы “Enter” нижняя граница TextBox будет подсвечена красным цветом
5. При отсутствии любого файла, необходимого для запуска программы, программа не запуститься. Список всех файлов для успешного запуска приведен в Приложении 1.
6. При OutOfMemoryException: “Insufficient memory to continue the execution of the program”
7. При NullReferenceException: “Object reference not set to an instance of an object”
8. При IndexOutOfRangeException: “Index was outside the bounds of an array”
9. При StackOverflowException: “Operation caused a stack overflow”
10. При InvalidCastException: “Specified cast is not valid”

При возникновении других исключений пользователю будет выведено базовое сообщение, характеризующее это исключение.

1. **КОД ПРОГРАММЫ**

**MainWindow.xaml.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Windows.Media.Animation;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

using Graphics;

using System.IO;

using Microsoft.Win32;

namespace Fractals

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

#region Private variables/properties

private InitializeMainGrids initialize;

//start and end colors for passive and active menu images

private Color startMenuDefaultColor = Color.FromRgb(255, 50, 0);

private Color endMenuDefaultColor = Color.FromRgb(255, 150, 0);

private Color startMenuActiveColor = Color.FromRgb(255, 100, 0);

private Color endMenuActiveColor = Color.FromRgb(255, 200, 0);

//current fractal, 0=nothing, 1=QC, 2=SC, 3=NF

private int CurrentFractal { get; set; } = 0;

//start draw object (initializes the main drawing process)

private StartDraw StartDraw { get; set; }

//tokern source, which token is used to stop the drawing process

private CancellationTokenSource TokenSource { get; set; } = new CancellationTokenSource();

private double CurrentScale { get; set; } = 1;

private bool IsButtonEventSet { get; set; } = true;

private double ScopeCoeff { get; set; } = 1;

private int MaxScopeEnlargement { get; set; } = 6;

//the max start length of the fractal

private int MaxLength { get; set; }

//the default margin of the scroll (when additional settings of QC are closed)

private Thickness InitialScrollMargin { get; set; }

//indicates which sides of the QC will be drawn

private bool[] QCSidesDrawability { get; set; } = { true, true, true, false };

//the number of selected ellipses in additional settings of QC

private int NumberOfQCSides { get; set; } = 3;

//the maximum recursion depth of the current fractal

private int MaxRecDepth { get; set; }

private Canvas MainCanvas { get; set; }

//current values of a textboxes in a params grid

private string CurrentLength { get; set; } = string.Empty;

private string CurrentDetpth { get; set; } = string.Empty;

private string CurrentStartColor { get; set; } = string.Empty;

private string CurrentEndColor { get; set; } = string.Empty;

//if the drawing is going

private bool IsDrawingGoing { get; set; } = false;

#endregion

public MainWindow()

{

try

{

InitializeComponent();

//set this culture in order to make all messages display in English

Thread.CurrentThread.CurrentUICulture = new System.Globalization.CultureInfo("en-US");

initialize = new InitializeMainGrids(mainWindow);

//initialize three main grids and two line grids

initialize.DoGridInitialization(headerGrid, leftGrid, mainGrid,

toplineGrid, leftLineGrid);

initialize.DrawMenuItems(qcFractal, nfFractal, scFractal,

startMenuDefaultColor, endMenuDefaultColor);

//initialize the parametres' grid (grid where all fractals params are set)

initialize.DoParamsGridInitialization(mainGrid, paramsGrid,

canvasOptionsGrid, mainCanvasScroll, qCExtraParamsGrid);

InitialScrollMargin = mainCanvasScroll.Margin;

MainCanvas = mainCanvas;

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}

#region Mouse Enter Events

private void EnlargeIconMouseEnter(object sender, MouseEventArgs e)

{

enlargeGridFirst.Visibility = Visibility.Collapsed;

enlargeGridSecond.Visibility = Visibility.Visible;

}

private void ReduceIconMouseEnter(object sender, MouseEventArgs e)

{

reduceGridFirst.Visibility = Visibility.Collapsed;

reduceGridSecond.Visibility = Visibility.Visible;

}

private void CancelDrawingMouseEnter(object sender, MouseEventArgs e)

{

cancelDrawingFirst.Visibility = Visibility.Collapsed;

cancelDrawingSecond.Visibility = Visibility.Visible;

}

private void ExportPngFirstEnter(object sender, MouseEventArgs e)

{

exportPngFirst.Visibility = Visibility.Collapsed;

exportPngSecond.Visibility = Visibility.Visible;

}

private void SelectEllipseMouseEnter(object sender, MouseEventArgs e)

{

Ellipse ellipse = (Ellipse)sender;

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 215, 0));

}

private void MenuCanvasMouseEnter(object sender, MouseEventArgs e)

{

Canvas canvas = (Canvas)sender;

switch (canvas.Name)

{

case "qcFractal":

initialize.DrawQCMenuFractal(canvas, startMenuActiveColor,

endMenuActiveColor);

break;

case "nfFractal":

initialize.DrawNFMenuFractal(canvas, startMenuActiveColor,

endMenuActiveColor);

break;

case "scFractal":

initialize.DrawSCMenuFractal(canvas, startMenuActiveColor,

endMenuActiveColor);

break;

}

}

#endregion

#region Mosue Leave Events

private void EnlargeIconMouseLeave(object sender, MouseEventArgs e)

{

enlargeGridFirst.Visibility = Visibility.Visible;

enlargeGridSecond.Visibility = Visibility.Collapsed;

}

private void ReduceIconMouseLeave(object sender, MouseEventArgs e)

{

reduceGridFirst.Visibility = Visibility.Visible;

reduceGridSecond.Visibility = Visibility.Collapsed;

}

private void CancelDrawingMouseLeave(object sender, MouseEventArgs e)

{

cancelDrawingFirst.Visibility = Visibility.Visible;

cancelDrawingSecond.Visibility = Visibility.Collapsed;

}

private void ExportPngSecondLeave(object sender, MouseEventArgs e)

{

exportPngSecond.Visibility = Visibility.Collapsed;

exportPngFirst.Visibility = Visibility.Visible;

}

private void SelectEllipseMouseLeave(object sender, MouseEventArgs e)

{

Ellipse ellipse = (Ellipse)sender;

switch (ellipse.Name)

{

case "leftEllipse":

if (QCSidesDrawability[0])

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

else

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

break;

case "topEllipse":

if (QCSidesDrawability[1])

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

else

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

break;

case "rightEllipse":

if (QCSidesDrawability[2])

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

else

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

break;

case "bottomEllipse":

if (QCSidesDrawability[3])

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

else

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

break;

}

}

/// <summary>

/// Change the color again to it's original value when the mouse leaves

/// </summary>

private void MenuCanvasMouseLeave(object sender, MouseEventArgs e)

{

Canvas canvas = (Canvas)sender;

switch (canvas.Name)

{

case "qcFractal":

initialize.DrawQCMenuFractal(canvas, startMenuDefaultColor, endMenuDefaultColor);

break;

case "nfFractal":

initialize.DrawNFMenuFractal(canvas, startMenuDefaultColor, endMenuDefaultColor);

break;

case "scFractal":

initialize.DrawSCMenuFractal(canvas, startMenuDefaultColor, endMenuDefaultColor);

break;

}

}

#endregion

#region Size Changed Events

/// <summary>

/// Change elements' position and size when the window is resized

/// </summary>

private void MainWindowSizeChanged(object sender, SizeChangedEventArgs e)

{

try

{

if (WindowState == WindowState.Maximized)

{

initialize = new InitializeMainGrids(mainWindow);

}

initialize.DoGridInitialization(headerGrid, leftGrid, mainGrid,

toplineGrid, leftLineGrid);

initialize.DoParamsGridInitialization(mainGrid, paramsGrid,

canvasOptionsGrid, mainCanvasScroll, qCExtraParamsGrid);

Scroll(mainCanvasScroll.ScrollableWidth / 2, mainCanvasScroll.ScrollableHeight / 2);

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

#endregion

#region Mouse Down/Click

/// <summary>

/// This event accurs when one ellipse in additional settings in QC

/// Refreshes the QCSidesDrawability and the NumberOfQCSides

/// </summary>

private void SelectEllipseMouseDown(object sender, MouseEventArgs e)

{

Ellipse ellipse = (Ellipse)sender;

switch (ellipse.Name)

{

case "leftEllipse":

if (QCSidesDrawability[0])

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

NumberOfQCSides--;

}

else

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

NumberOfQCSides++;

}

QCSidesDrawability[0] = !QCSidesDrawability[0];

break;

case "topEllipse":

if (QCSidesDrawability[1])

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

NumberOfQCSides--;

}

else

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

NumberOfQCSides++;

}

QCSidesDrawability[1] = !QCSidesDrawability[1];

break;

case "rightEllipse":

if (QCSidesDrawability[2])

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

NumberOfQCSides--;

}

else

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

NumberOfQCSides++;

}

QCSidesDrawability[2] = !QCSidesDrawability[2];

break;

case "bottomEllipse":

if (QCSidesDrawability[3])

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(169, 169, 169));

NumberOfQCSides--;

}

else

{

ellipse.Fill = new SolidColorBrush(Color.FromRgb(255, 165, 0));

NumberOfQCSides++;

}

QCSidesDrawability[3] = !QCSidesDrawability[3];

break;

}

}

private void EnlargeIconMouseDown(object sender, MouseButtonEventArgs e)

{

if (CurrentScale <= MaxScopeEnlargement)

{

double verticalOffSet = mainCanvasScroll.ContentVerticalOffset;

double horizontalOffSet = mainCanvasScroll.ContentHorizontalOffset;

ScaleTransform transform = new ScaleTransform

{

ScaleX = CurrentScale + ScopeCoeff,

ScaleY = CurrentScale += ScopeCoeff,

CenterX = MainCanvas.Width / 2,

CenterY = MainCanvas.Height / 2

};

MainCanvas.RenderTransform = transform;

Scroll(horizontalOffSet, verticalOffSet);

}

}

private void ReduceIconMouseDown(object sender, MouseButtonEventArgs e)

{

if (CurrentScale - ScopeCoeff >= 1)

{

double verticalOffSet = mainCanvasScroll.ContentVerticalOffset;

double horizontalOffSet = mainCanvasScroll.ContentHorizontalOffset;

ScaleTransform transform = new ScaleTransform

{

ScaleX = CurrentScale - ScopeCoeff,

ScaleY = CurrentScale -= ScopeCoeff,

CenterX = MainCanvas.Width / 2,

CenterY = MainCanvas.Height / 2,

};

MainCanvas.RenderTransform = transform;

Scroll(horizontalOffSet, verticalOffSet);

}

}

/// <summary>

/// Cancels the current drawing task with the token source

/// </summary>

private void CancelCurrentDrawingMouseDown(object sender, MouseButtonEventArgs e)

{

TokenSource.Cancel();

TokenSource = new CancellationTokenSource();

}

/// <summary>

/// Saves the image of a fractal as a PNG file

/// </summary>

private void ExportPngSecondClick(object sender, MouseButtonEventArgs e)

{

SaveFileDialog saveFileDialog = new SaveFileDialog

{

DefaultExt = "png",

Filter = "\*PNG(\*.png)|\*.png",

OverwritePrompt = true,

Title = "Save the fractal as a PNG file"

};

try

{

double horizontalOffSet = mainCanvasScroll.HorizontalOffset;

double verticalOffSet = mainCanvasScroll.VerticalOffset;

saveFileDialog.ShowDialog();

if (saveFileDialog.FileName != string.Empty)

{

Transform transform = MainCanvas.LayoutTransform;

//reset current transform (in case it is scaled or rotated)

MainCanvas.LayoutTransform = null;

Size size = new Size(MainCanvas.Width, MainCanvas.Height);

MainCanvas.Measure(size);

MainCanvas.Arrange(new Rect(size));

RenderTargetBitmap rtb = new RenderTargetBitmap((int)size.Width,

(int)size.Height, 96d, 96d, PixelFormats.Pbgra32);

rtb.Render(MainCanvas);

// Create a file stream for saving image

using (FileStream outStream = (FileStream)saveFileDialog.OpenFile())

{

// Use png encoder for our data

PngBitmapEncoder encoder = new PngBitmapEncoder();

// push the rendered bitmap to it

encoder.Frames.Add(BitmapFrame.Create(rtb));

// save the data to the stream

encoder.Save(outStream);

}

// Restore previously saved layout

MainCanvas.LayoutTransform = transform;

Scroll(mainCanvasScroll.ScrollableWidth / 2, mainCanvasScroll.ScrollableHeight / 2);

MessageBox.Show("The fractal is saved", "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}

catch (FileNotFoundException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}

/// <summary>

/// Starts the process of drawing

/// </summary>

private void MenuCanvasMouseDown(object sender, MouseButtonEventArgs e)

{

try

{

//get all text boxes out of the params grid

List<TextBox> pararmsTextBoxes = paramsGrid.Children.OfType<TextBox>().ToList();

Canvas canvas = (Canvas)sender;

if (CurrentFractal != 0)

{

CurrentLength = pararmsTextBoxes[0].Text;

CurrentDetpth = pararmsTextBoxes[1].Text;

CurrentStartColor = pararmsTextBoxes[2].Text;

CurrentEndColor = pararmsTextBoxes[3].Text;

}

switch (canvas.Name)

{

case "qcFractal":

MaxLength = 115;

MaxRecDepth = 10;

CreateQCFractalMenu createQCMenu = new CreateQCFractalMenu();

createQCMenu.CreateMenu(paramsGrid, mainWindow,

"Quasi-clover fractal", MaxLength, MaxRecDepth, CurrentLength,

CurrentDetpth, CurrentStartColor, CurrentEndColor);

CurrentFractal = 1;

break;

case "scFractal":

MaxLength = 700;

MaxRecDepth = 10;

CreateSCFractalMenu createSCMenu = new CreateSCFractalMenu();

createSCMenu.CreateMenu(paramsGrid, mainWindow,

"Serpinskiy's triangle", MaxLength, MaxRecDepth, CurrentLength,

CurrentDetpth, CurrentStartColor, CurrentEndColor);

CurrentFractal = 2;

//if the additional settings are still open - close them

if (mainCanvasScroll.Margin.Left != InitialScrollMargin.Left)

{

AnimateScroll(-175);

}

break;

case "nfFractal":

MaxLength = 325;

MaxRecDepth = 8;

CreateNFFractalMenu createNFMenu = new CreateNFFractalMenu();

createNFMenu.CreateMenu(paramsGrid, mainWindow,

"H-Fractral", MaxLength, MaxRecDepth, CurrentLength,

CurrentDetpth, CurrentStartColor, CurrentEndColor);

CurrentFractal = 3;

//if the additional settings are still open - close them

if (mainCanvasScroll.Margin.Left != InitialScrollMargin.Left)

{

AnimateScroll(-175);

}

break;

}

//attach a new button to the StartDraw property

if (StartDraw != null)

{

StartDraw.GoBtn = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1];

}

Button goBtn = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1];

//add a mouse down handler to the "Go" btn

if (IsButtonEventSet)

{

goBtn.Click += (obj, evArgs) =>

{

DrawFractalEventDown(true);

};

IsButtonEventSet = false;

}

//add an aditional settings event

if (CurrentFractal == 1)

{

Button additionalSettings = (Button)paramsGrid.Children[

paramsGrid.Children.Count - 2];

additionalSettings.Click += (s, ev) =>

{

if (mainCanvasScroll.Margin.Left == InitialScrollMargin.Left)

{

additionalSettings.Content = "Close";

AnimateScroll(175);

}

else

{

additionalSettings.Content = "Additional settings";

AnimateScroll(-175);

}

};

}

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (InvalidCastException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (ArgumentOutOfRangeException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}

private async void DrawFractalEventDown(bool showMsgBox)

{

try

{

IsDrawingGoing = true;

//start length & recursion depth

TextBox sLengthBox = (TextBox)paramsGrid.Children[1];

TextBox recDepthBox = (TextBox)paramsGrid.Children[3];

Point startPoint = new Point(MainCanvas.Width / (2),

MainCanvas.Height / (2));

//hex values of start and end color

string startColorText = ((TextBox)paramsGrid.Children[5]).Text;

string endColorText = ((TextBox)paramsGrid.Children[6]).Text;

if (double.TryParse(sLengthBox.Text, out double length) &&

int.TryParse(recDepthBox.Text, out int depth) && depth > 0

&& depth <= MaxRecDepth && length > 0 && length <= MaxLength &&

CheckColorString.Check(startColorText) && CheckColorString.Check(endColorText))

{

int numberOfTrues = 0;

foreach (bool b in QCSidesDrawability)

{

if (b)

numberOfTrues++;

}

if (numberOfTrues != 3 && CurrentFractal == 1)

{

MessageBox.Show("Select three ellipses to draw...",

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

else

{

//reset the canvas

Canvas newCanvas = new Canvas

{

Width = 5000,

Height = 5000,

Background = new SolidColorBrush(Color.FromRgb(255, 255, 255))

};

mainCanvasScroll.Content = MainCanvas = newCanvas;

//transform the canvas to it's original size

ScaleTransform transform = new ScaleTransform

{

ScaleX = CurrentScale = 1,

ScaleY = CurrentScale = 1,

};

MainCanvas.RenderTransform = transform;

Scroll(mainCanvasScroll.ScrollableWidth / 2, mainCanvasScroll.ScrollableHeight / 2);

//get the checkbox

List<CheckBox> checkBoxes = paramsGrid.Children.OfType<CheckBox>().ToList();

CheckBox checkBox = checkBoxes[0];

//intialize start draw object

StartDraw = new StartDraw

{

CurrentFractal = CurrentFractal,

MainCanvas = MainCanvas,

StartColor = (Color)ColorConverter.ConvertFromString(startColorText),

EndColor = (Color)ColorConverter.ConvertFromString(endColorText),

GoBtn = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1],

QCSidesDrawability = QCSidesDrawability

};

CancellationToken token = TokenSource.Token;

//set the priority

if (checkBox.IsChecked == false || checkBox.IsChecked == null)

{

cancelDrawingFirst.IsEnabled = false;

StartDraw.DispatcherPriority = DispatcherPriority.Send;

mainCanvasScroll.Content = MainCanvas = await StartDraw.DrawFractal(length, depth,

startPoint, Dispatcher, token);

cancelDrawingFirst.IsEnabled = true;

}

else

{

StartDraw.DispatcherPriority = DispatcherPriority.Background;

await StartDraw.DrawFractal(length, depth,

startPoint, Dispatcher, token);

}

}

}

else

{

if (showMsgBox)

MessageBox.Show("Error in the input data",

"Soft's message", MessageBoxButton.OK, MessageBoxImage.Information);

}

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (InvalidCastException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (OutOfMemoryException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (StackOverflowException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

finally

{

IsDrawingGoing = false;

}

}

#endregion

#region Animations

/// <summary>

/// Animates the scroll, offsetting it to the left

/// or to the right, depending on the offset's sign

/// </summary>

private void AnimateScroll(double offset)

{

DoubleAnimation scrollWidthAnimation = new DoubleAnimation

{

From = mainCanvasScroll.Width,

To = mainCanvasScroll.Width - offset,

Duration = TimeSpan.FromMilliseconds(100),

FillBehavior = FillBehavior.Stop

};

ThicknessAnimation scrollMarginAnimation = new ThicknessAnimation

{

From = mainCanvasScroll.Margin,

To = new Thickness(mainCanvasScroll.Margin.Left + offset,

mainCanvasScroll.Margin.Top, 0, 0),

Duration = TimeSpan.FromMilliseconds(100),

FillBehavior = FillBehavior.Stop

};

if (mainCanvasScroll.Margin.Left + offset >= InitialScrollMargin.Left)

{

mainCanvasScroll.BeginAnimation(MarginProperty, scrollMarginAnimation);

mainCanvasScroll.BeginAnimation(WidthProperty, scrollWidthAnimation);

mainCanvasScroll.Width -= offset;

mainCanvasScroll.Margin = new Thickness(mainCanvasScroll.Margin.Left + offset,

mainCanvasScroll.Margin.Top, 0, 0);

}

if (qCExtraParamsGrid.Visibility == Visibility.Collapsed)

qCExtraParamsGrid.Visibility = Visibility.Visible;

else

qCExtraParamsGrid.Visibility = Visibility.Collapsed;

}

#endregion

private void Scroll(double horizontalOffSet, double verticalOffSet)

{

mainCanvasScroll.ScrollToHorizontalOffset(horizontalOffSet);

mainCanvasScroll.ScrollToVerticalOffset(verticalOffSet);

mainCanvasScroll.LineUp();

}

private void EnterKeyDown(object sender, KeyEventArgs e)

{

if (CurrentFractal != 0 && e.Key == Key.Enter && !IsDrawingGoing)

DrawFractalEventDown(true);

}

}

}

**InitializeMainGrids.cs**

using System.Threading;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

/// <summary>

/// Class where all initial positions and sizes are set

/// (also this class is used when the size of the main window is changed)

/// </summary>

public class InitializeMainGrids

{

private Window MainWindow { get; }

Dispatcher Dispatcher { get; set; } = Dispatcher.CurrentDispatcher;

public InitializeMainGrids(Window mainWindow)

{

MainWindow = mainWindow;

MainWindow.Height = SystemParameters.MaximizedPrimaryScreenHeight;

MainWindow.Width = SystemParameters.MaximizedPrimaryScreenWidth;

MainWindow.Top = 0;

MainWindow.Left = 0;

}

/// <summary>

/// Initializes the position and size of three main grids

/// </summary>

public void DoGridInitialization(Grid headerGrid, Grid leftGrid, Grid mainGrid,

Grid topLineGrid, Grid leftLineGrid)

{

//main three grids

headerGrid.Width = MainWindow.Width;

leftGrid.Height = MainWindow.Height - headerGrid.Height;

mainGrid.Margin = new Thickness(leftGrid.Width, headerGrid.Height, 0, 0);

mainGrid.Width = MainWindow.Width - leftGrid.Width;

mainGrid.Height = MainWindow.Height - headerGrid.Height;

//devision lines grids

topLineGrid.Width = MainWindow.Width;

leftLineGrid.Height = leftGrid.Height;

}

/// <summary>

/// Draw fractals in the menu's cells

/// </summary>

public void DrawMenuItems(Canvas qcFractalCanvas, Canvas nfFractalCanvas,

Canvas scFractalCanvas, Color startColor, Color endColor)

{

DrawQCMenuFractal(qcFractalCanvas, startColor, endColor);

DrawNFMenuFractal(nfFractalCanvas, startColor, endColor);

DrawSCMenuFractal(scFractalCanvas, startColor, endColor);

}

public async void DrawQCMenuFractal(Canvas qcFractalCanvas, Color startColor,

Color endColor)

{

//draw qasiclover fractal in the first cell

qcFractalCanvas.Children.Clear();

QCFractal qcFractal = new QCFractal(qcFractalCanvas.Width / 6,

startColor, endColor, 2, qcFractalCanvas);

//start point

Point startPoint = new Point(qcFractalCanvas.Width / 2,

qcFractalCanvas.Height / 2);

QCDrawingParams drawingParams = new QCDrawingParams(qcFractal.StartLength, 1,

startPoint, new bool[] { true, true, true, false });

qcFractalCanvas = await qcFractal.Draw(Dispatcher, DispatcherPriority.Normal, CancellationToken.None,

drawingParams);

}

public async void DrawNFMenuFractal(Canvas nfFractalCanvas, Color startColor,

Color endColor)

{

//draw the H-Fractal in the third cell

nfFractalCanvas.Children.Clear();

NFFractal nfFractal = new NFFractal(nfFractalCanvas.Width / 2,

startColor, endColor, 2, nfFractalCanvas);

Point startPoint = new Point(nfFractalCanvas.Width / 2,

nfFractalCanvas.Height / 2);

NFDrawingParams nfDrawingParams = new NFDrawingParams(nfFractal.StartLength,

1, startPoint);

nfFractalCanvas = await nfFractal.Draw(Dispatcher, DispatcherPriority.Normal, CancellationToken.None,

nfDrawingParams);

}

public async void DrawSCMenuFractal(Canvas scFractalCanvas, Color startColor,

Color endColor)

{

//draw the Serpinskiy's triangle int the second cell

scFractalCanvas.Children.Clear();

SCFractal scFractal = new SCFractal(2.0 \* scFractalCanvas.Width / 3,

startColor, endColor, 2, scFractalCanvas);

Point startPoint = new Point(scFractalCanvas.Width / 2, scFractalCanvas.Height / 2);

SCDrawingParams scDrawingParams = new SCDrawingParams(scFractal.StartLength,

1, startPoint);

scFractal.InitialDraw(scDrawingParams);

scFractalCanvas = await scFractal.Draw(Dispatcher, DispatcherPriority.Normal, CancellationToken.None,

scDrawingParams);

}

public void DoParamsGridInitialization(Grid mainGrid, Grid paramsGrid,

Grid canvasOptionsGrid, ScrollViewer scroll, Grid qcExtraParamsGrid)

{

paramsGrid.Margin = new Thickness(10, 10, 0, 10);

paramsGrid.Height = 148 \* 3;

scroll.Width = mainGrid.Width - paramsGrid.Width - 50

- canvasOptionsGrid.Width - qcExtraParamsGrid.ActualWidth;

scroll.Height = MainWindow.Height - 170;

scroll.Margin = new Thickness(scroll.Margin.Left, scroll.Margin.Top,

scroll.Margin.Right, 30);

canvasOptionsGrid.Margin = new Thickness(scroll.Margin.Left + scroll.Width,

10, 10, 0);

}

}

}

**NFFractal.cs**

using System;

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace Graphics

{

public class NFFractal : Fractal

{

public Canvas MainCanvas { get; set; }

private List<PointData> nfFractalPointsList = new List<PointData>();

public NFFractal(double startLength, Color startColor, Color endColor,

int maxRec, Canvas canvas) : base(startLength, startColor,

endColor, maxRec)

{

MainCanvas = canvas;

}

/// <summary>

/// Creates a list of all points, which are used in the fractal

/// </summary>

private void FindDots(DrawingParameters drawingParameters, Dispatcher dispatcher)

{

try

{

NFDrawingParams nfDrawingParams = (NFDrawingParams)drawingParameters;

Point[] tPointsArr = GetFractalPoints(nfDrawingParams.CurrentCoords,

nfDrawingParams.CurrentLength);

SolidColorBrush brush = GetCurrentColor.Get(StartColor, EndColor,

nfDrawingParams.RecursionLevel - 1, MaxRecursionLevel - 1);

nfFractalPointsList.Add(new NFFractalPointData(tPointsArr, brush));

if (nfDrawingParams.RecursionLevel < MaxRecursionLevel)

{

//define new drawing params for four new elements and find the data about them

NFDrawingParams newDrawingParams = new NFDrawingParams(

nfDrawingParams.CurrentLength / 2, nfDrawingParams.RecursionLevel + 1,

tPointsArr[2]);

FindDots(newDrawingParams, dispatcher);

newDrawingParams.CurrentCoords = tPointsArr[3];

FindDots(newDrawingParams, dispatcher);

newDrawingParams.CurrentCoords = tPointsArr[4];

FindDots(newDrawingParams, dispatcher);

newDrawingParams.CurrentCoords = tPointsArr[5];

FindDots(newDrawingParams, dispatcher);

}

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

}

/// <summary>

/// Draws the farctal itself, based on the list of points we created before

/// </summary>

public override async Task<Canvas> Draw(Dispatcher dispatcher, DispatcherPriority priority,

CancellationToken token, DrawingParameters drawingParameters)

{

try

{

await dispatcher.BeginInvoke(new Action(() => FindDots(drawingParameters, dispatcher)));

for (int i = 0; i < nfFractalPointsList.Count; i++)

{

if (token != CancellationToken.None && token.IsCancellationRequested)

{

throw new TaskCanceledException();

}

dispatcher.Invoke(new Action(() =>

{

try

{

Line horizontalLine = new Line

{

Stroke = nfFractalPointsList[i].CurrentBrush,

X1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[0].X,

Y1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[0].Y,

X2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[1].X,

Y2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[1].Y,

};

Line leftVerticalLine = new Line

{

Stroke = nfFractalPointsList[i].CurrentBrush,

X1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[2].X,

Y1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[2].Y,

X2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[3].X,

Y2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[3].Y,

};

Line rightVerticalLine = new Line

{

Stroke = nfFractalPointsList[i].CurrentBrush,

X1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[4].X,

Y1 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[4].Y,

X2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[5].X,

Y2 = ((NFFractalPointData)nfFractalPointsList[i]).PointsArr[5].Y,

};

//add the lines to the canvas

MainCanvas.Children.Add(horizontalLine);

MainCanvas.Children.Add(leftVerticalLine);

MainCanvas.Children.Add(rightVerticalLine);

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (OutOfMemoryException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}), priority);

}

return MainCanvas;

}

catch (TaskCanceledException)

{

dispatcher.Invoke(() => MessageBox.Show("Drawing task " +

"was canceled", "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (IndexOutOfRangeException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

}

/// <summary>

/// Get the six points, which then will be used when drawing the base element

/// </summary>

private Point[] GetFractalPoints(Point startPoint, double currLength)

{

return new Point[]

{

//horizontal line

new Point(startPoint.X - currLength / 2, startPoint.Y),

new Point(startPoint.X + currLength / 2, startPoint.Y),

//two vertical lines, firstly left line

new Point(startPoint.X - currLength/2, startPoint.Y - currLength/2),

new Point(startPoint.X - currLength/2, startPoint.Y + currLength/2),

//right line

new Point(startPoint.X + currLength/2, startPoint.Y - currLength/2),

new Point(startPoint.X + currLength/2, startPoint.Y + currLength/2),

};

}

}

}

**QCFractal.cs**

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using System.Threading;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace Graphics

{

public class QCFractal : Fractal

{

public Canvas MainCanvas { get; set; }

List<PointData> pointsList = new List<PointData>();

public QCFractal(double startLength, Color startColor, Color endColor,

int maxRec, Canvas canvas) : base(startLength, startColor,

endColor, maxRec)

{

MainCanvas = canvas;

}

/// <summary>

/// Creates a list of all points, which are used in the fractal

/// </summary>

private void FindDots(DrawingParameters drawingParameters, Dispatcher dispatcher)

{

try

{

QCDrawingParams qcDrawingParams = (QCDrawingParams)drawingParameters;

double currX = drawingParameters.CurrentCoords.X - qcDrawingParams.CurrentLength;

double currY = drawingParameters.CurrentCoords.Y - qcDrawingParams.CurrentLength;

double length = qcDrawingParams.CurrentLength;

SolidColorBrush brush = GetCurrentColor.Get(StartColor, EndColor,

drawingParameters.RecursionLevel - 1, MaxRecursionLevel - 1);

pointsList.Add(new PointData(length, brush, new Point(currX, currY)));

QCDrawingParams newQCParams = new QCDrawingParams(qcDrawingParams.CurrentLength / 2,

qcDrawingParams.RecursionLevel + 1, new Point(0, 0), new bool[] { });

if (drawingParameters.RecursionLevel < MaxRecursionLevel)

{

bool[][] newSidesDrawability = GetNewSidesDrawability(qcDrawingParams.SidesDrawability);

for (int i = 0; i < qcDrawingParams.SidesDrawability.Length; i++)

{

if (qcDrawingParams.SidesDrawability[i])

{

newQCParams.CurrentCoords = GetNewPoint(qcDrawingParams.CurrentCoords,

qcDrawingParams.CurrentLength, i);

newQCParams.SidesDrawability = newSidesDrawability[i];

FindDots(newQCParams, dispatcher);

}

}

}

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

}

/// <summary>

/// Draws the farctal itself, based on the list of points we created before

/// </summary>

public async override Task<Canvas> Draw(Dispatcher dispatcher, DispatcherPriority priority,

CancellationToken token, DrawingParameters drawingParameters)

{

try

{

await dispatcher.BeginInvoke(new Action(() => FindDots(drawingParameters, dispatcher)));

for (int i = 0; i < pointsList.Count; i++)

{

if (token != CancellationToken.None && token.IsCancellationRequested)

{

throw new TaskCanceledException();

}

dispatcher.Invoke(new Action(() =>

{

try

{

//draw. firstly create an ellipse with the given size

Ellipse ellipse = new Ellipse

{

Width = pointsList[i].Length \* 2,

Height = pointsList[i].Length \* 2,

Fill = pointsList[i].CurrentBrush

};

//set the position of the created ellipse and add attach to the parent

Canvas.SetLeft(ellipse, pointsList[i].Coords.X);

Canvas.SetTop(ellipse, pointsList[i].Coords.Y);

MainCanvas.Children.Add(ellipse);

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (OutOfMemoryException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}), priority);

}

return MainCanvas;

}

catch (TaskCanceledException)

{

dispatcher.Invoke(() => MessageBox.Show("Drawing task " +

"was canceled", "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (IndexOutOfRangeException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

}

/// <summary>

/// Creates four new arrays with new sides drawability for each side

/// </summary>

/// <param name="sidesDrawability">

/// Old drawability boolean array

/// </param>

public bool[][] GetNewSidesDrawability(bool[] sidesDrawability)

{

bool[][] newArr = new bool[4][];

if (sidesDrawability[0])

{

newArr[0] = new bool[] { true, true, false, true };

}

if (sidesDrawability[1])

{

newArr[1] = new bool[] { true, true, true, false };

}

if (sidesDrawability[2])

{

newArr[2] = new bool[] { false, true, true, true };

}

if (sidesDrawability[3])

{

newArr[3] = new bool[] { true, false, true, true };

}

return newArr;

}

/// <summary>

/// Gets a point where the next fractal base element will be drawn

/// </summary>

public Point GetNewPoint(Point oldPoint, double oldLength, int elementNum)

{

switch (elementNum)

{

case 0:

double x = oldPoint.X - 1.5 \* oldLength;

return new Point(x, oldPoint.Y);

case 1:

double y = oldPoint.Y - 1.5 \* oldLength;

return new Point(oldPoint.X, y);

case 2:

x = oldPoint.X + 1.5 \* oldLength;

return new Point(x, oldPoint.Y);

case 3:

y = oldPoint.Y + 1.5 \* oldLength;

return new Point(oldPoint.X, y);

default:

MessageBox.Show("The fatal error, this should have never happened, " +

"restart the app.", "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

return new Point(0, 0);

}

}

}

}

**SCFractal.cs**

using System;

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace Graphics

{

public class SCFractal : Fractal

{

public Canvas MainCanvas { get; set; }

public List<PointData> pointsList = new List<PointData>();

public SCFractal(double startLength, Color startColor, Color endColor,

int maxRec, Canvas canvas) : base(startLength, startColor,

endColor, maxRec)

{

MainCanvas = canvas;

}

/// <summary>

/// Draws the first main triangle

/// </summary>

public void InitialDraw(DrawingParameters drawingParameters)

{

SCDrawingParams scDrawingParams = (SCDrawingParams)drawingParameters;

Point[] tPointsArr = GetTrianglePoints(scDrawingParams.CurrentCoords,

scDrawingParams.CurrentLength);

PointCollection tPointsCollection = new PointCollection(tPointsArr);

Polyline line = new Polyline

{

Points = tPointsCollection,

StrokeThickness = 0.3,

Stroke = GetCurrentColor.Get(StartColor, EndColor,

scDrawingParams.RecursionLevel, MaxRecursionLevel)

};

MainCanvas.Children.Add(line);

}

/// <summary>

/// Creates a list of all points, which are used in the fractal

/// </summary>

private void FindDots(DrawingParameters drawingParameters, Dispatcher dispatcher)

{

try

{

SCDrawingParams scDrawingParams = (SCDrawingParams)drawingParameters;

double currX = drawingParameters.CurrentCoords.X;

double currY = drawingParameters.CurrentCoords.Y;

double side = scDrawingParams.CurrentLength;

Point centerPoint = new Point(currX, currY);

SolidColorBrush brush = GetCurrentColor.Get(StartColor, EndColor,

drawingParameters.RecursionLevel - 1, MaxRecursionLevel - 1);

pointsList.Add(new PointData(side, brush, new Point(currX, currY)));

SCDrawingParams newQCParams = new SCDrawingParams(scDrawingParams.CurrentLength / 2,

scDrawingParams.RecursionLevel + 1, new Point(0, 0));

if (scDrawingParams.RecursionLevel < MaxRecursionLevel)

{

SCDrawingParams newDrawingParams = new SCDrawingParams(side / 2,

scDrawingParams.RecursionLevel + 1,

new Point(centerPoint.X, centerPoint.Y -

side \* Math.Sqrt(3) / 8));

FindDots(newDrawingParams, dispatcher);

newDrawingParams = new SCDrawingParams(side / 2,

scDrawingParams.RecursionLevel + 1,

new Point(centerPoint.X - side / 4.0, centerPoint.Y +

side \* Math.Sqrt(3) / 8));

FindDots(newDrawingParams, dispatcher);

newDrawingParams = new SCDrawingParams(side / 2,

scDrawingParams.RecursionLevel + 1,

new Point(centerPoint.X + side / 4.0, centerPoint.Y +

side \* Math.Sqrt(3) / 8));

FindDots(newDrawingParams, dispatcher);

}

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

}

}

/// <summary>

/// Draws the farctal itself, based on the list of points we created before

/// </summary>

public async override Task<Canvas> Draw(Dispatcher dispatcher, DispatcherPriority priority,

CancellationToken token, DrawingParameters drawingParameters)

{

try

{

await dispatcher.BeginInvoke(new Action(() => FindDots(drawingParameters, dispatcher)));

await dispatcher.BeginInvoke(new Action(() => InitialDraw(drawingParameters)));

for (int i = 0; i < pointsList.Count; i++)

{

if (token != CancellationToken.None && token.IsCancellationRequested)

{

throw new TaskCanceledException();

}

dispatcher.Invoke(new Action(() =>

{

try

{

Point centerPoint = pointsList[i].Coords;

double side = pointsList[i].Length;

Point[] tPointsArr = new Point[]

{

new Point(centerPoint.X - side/4, centerPoint.Y - 0.3),

new Point(centerPoint.X + side/4, centerPoint.Y - 0.3),

new Point(centerPoint.X, centerPoint.Y + side \* Math.Sqrt(3) / 4 - 0.3),

new Point(centerPoint.X - side/4, centerPoint.Y - 0.3),

};

PointCollection tPointsCollection = new PointCollection(tPointsArr);

Polyline line = new Polyline

{

Points = tPointsCollection,

StrokeThickness = 0.3,

Stroke = pointsList[i].CurrentBrush

};

MainCanvas.Children.Add(line);

}

catch (NullReferenceException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (OutOfMemoryException ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information);

}

}), priority);

}

return MainCanvas;

}

catch (TaskCanceledException)

{

dispatcher.Invoke(() => MessageBox.Show("Drawing task " +

"was canceled", "Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (IndexOutOfRangeException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message));

return null;

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

}

public Point[] GetTrianglePoints(Point centerPoint, double sideLength)

{

double median = sideLength \* Math.Sqrt(3) / 2;

return new Point[]

{

new Point(centerPoint.X, centerPoint.Y - median / 2.0),

new Point(centerPoint.X - (sideLength / 2.0), centerPoint.Y + median / 2.0),

new Point(centerPoint.X + (sideLength / 2.0), centerPoint.Y + median / 2.0),

new Point(centerPoint.X, centerPoint.Y - median / 2.0)

};

}

}

}

**NFDrawingParams.cs**

using System.Windows;

namespace Graphics

{

public class NFDrawingParams : DrawingParameters

{

public NFDrawingParams(double currentLength, int recLvl,

Point currPoint) : base(currentLength,

recLvl, currPoint) { }

}

}

**QCDrawingParams.cs**

using System.Windows;

namespace Graphics

{

public class QCDrawingParams : DrawingParameters

{

public bool[] SidesDrawability { get; set; }

//initialize all properties

public QCDrawingParams(double currentLength,

int recLvl, Point currPoint,

bool[] sidesDrawability) : base(currentLength, recLvl, currPoint)

{

SidesDrawability = sidesDrawability;

}

}

}

**SCDrawingParams.cs**

using System.Windows;

namespace Graphics

{

public class SCDrawingParams : DrawingParameters

{

public SCDrawingParams(double currentLength, int recLvl,

Point currPoint) : base(currentLength,

recLvl, currPoint)

{ }

}

}

**CreateFractalMenu.cs**

using System;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

public class CreateFractalMenu

{

#region Private Variables

private string LengthTextBlockText { get; } = "Enter the start length here...";

private string RecDepthBlockText { get; } = "Enter the recursion depth here...";

private string StartColorBoxText { get; } = "Enter the #HEX value of start color...";

private string EndColorBoxText { get; } = "Enter the #HEX value of end color...";

#endregion

public virtual void CreateMenu(Grid paramsGrid, Window mainWindow,

string fractalName, int maxLength, int maxDepth,

string currentLength, string currentDepth, string currStartColor,

string currEndColor)

{

paramsGrid.Children.Clear();

#region Elements

//create elements

TextBlock headerTextBox = new TextBlock

{

Text = $"Set the params of the \n{fractalName} here: ",

TextAlignment = TextAlignment.Left,

TextWrapping = TextWrapping.Wrap,

Width = 200,

Height = 55,

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 10, 0, 0),

Foreground = new SolidColorBrush(Color.FromRgb(170, 170, 170)),

FontSize = 15,

FontWeight = FontWeight.FromOpenTypeWeight(600)

};

TextBox startLengthTextBox = new TextBox

{

Style = mainWindow.Resources["paramsTextBoxStyle"] as Style,

Text = (currentLength == string.Empty) ? LengthTextBlockText : currentLength,

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 80, 0, 0),

Height = 20,

ToolTip = $"Max Length less or equal {maxLength}",

Width = 200

};

if ((!int.TryParse(currentLength, out int cLength) ||

cLength <= 0 || cLength > maxLength) &&

startLengthTextBox.Text != LengthTextBlockText)

{

startLengthTextBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

Slider startLengthSlider = new Slider

{

Width = 150,

Height = 20,

Minimum = 1,

Maximum = maxLength,

Margin = new Thickness(10, 110, 0, 0),

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

IsSelectionRangeEnabled = true,

SelectionStart = 1,

SelectionEnd = maxLength,

ToolTip = $"Max Length less or equal {maxLength}"

};

if (int.TryParse(currentLength, out int length) && length <= maxLength)

{

startLengthSlider.Value = length;

}

TextBox recursionDepthTextBox = new TextBox

{

Style = mainWindow.Resources["paramsTextBoxStyle"] as Style,

Text = (currentDepth == string.Empty) ? RecDepthBlockText : currentDepth,

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 150, 0, 0),

Height = 20,

ToolTip = $"Max depth less or equal {maxDepth}",

Width = 200

};

if ((!int.TryParse(currentDepth, out int cDepth)

|| cDepth <= 0 || cDepth > maxDepth) &&

recursionDepthTextBox.Text != RecDepthBlockText)

{

recursionDepthTextBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

Slider recursionDepthSlider = new Slider

{

Width = 150,

Minimum = 1,

Height = 20,

Maximum = maxDepth,

Margin = new Thickness(10, 180, 0, 0),

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

IsSelectionRangeEnabled = true,

SelectionStart = 1,

SelectionEnd = maxDepth,

ToolTip = $"Max depth less or equal {maxDepth}"

};

if (int.TryParse(currentDepth, out int depth) && depth <= maxDepth)

{

recursionDepthSlider.Value = depth;

}

TextBox startColorTextBox = new TextBox

{

Style = mainWindow.Resources["paramsTextBoxStyle"] as Style,

Text = (currStartColor == string.Empty) ? StartColorBoxText : currStartColor,

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 220, 0, 0),

Height = 20,

Width = 200

};

if (!CheckColorString.Check(currStartColor) &&

startColorTextBox.Text != StartColorBoxText)

{

startColorTextBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

TextBox endColorTextBox = new TextBox

{

Style = mainWindow.Resources["paramsTextBoxStyle"] as Style,

Text = (currEndColor == string.Empty) ? EndColorBoxText : currEndColor,

VerticalAlignment = VerticalAlignment.Top,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 260, 0, 0),

Height = 20,

Width = 200

};

if (!CheckColorString.Check(currEndColor) &&

endColorTextBox.Text != EndColorBoxText)

{

endColorTextBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

CheckBox drawSpeedRadioBtn = new CheckBox

{

//Style = mainWindow.Resources["radioBtnStyle"] as Style,

Padding = new Thickness(1, 1, 1, 1),

Content = "Show drawing process",

FontFamily = new FontFamily("Arial"),

Foreground = new SolidColorBrush(Color.FromRgb(169, 169, 169)),

Height = 20,

Width = 200,

Margin = new Thickness(10, 300, 0, 0),

HorizontalAlignment = HorizontalAlignment.Left,

VerticalAlignment = VerticalAlignment.Top

};

TextBlock hintForCheckBox = new TextBlock

{

Text = "(for high recursion levels, greater than 8)",

FontFamily = new FontFamily("Arial"),

FontSize = 10,

Foreground = new SolidColorBrush(Color.FromRgb(169, 169, 169)),

Margin = new Thickness(10, 320, 0, 0),

};

#endregion

#region Event Handlers

//event handlers:

startLengthTextBox.GotFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (textBox.Text == LengthTextBlockText)

{

textBox.Text = string.Empty;

}

if (!int.TryParse(textBox.Text, out int x) || x <= 0

|| x > maxLength)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

};

recursionDepthTextBox.GotFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (!int.TryParse(textBox.Text, out int x) || x <= 0 || x > maxDepth)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

if (textBox.Text == RecDepthBlockText)

{

textBox.Text = string.Empty;

}

};

startLengthTextBox.LostFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (textBox.Text == string.Empty)

{

textBox.Text = LengthTextBlockText;

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

else

{

if (!double.TryParse(textBox.Text, out double x) || x <= 0

|| x > maxLength)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

}

};

recursionDepthTextBox.LostFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (textBox.Text == string.Empty)

{

textBox.Text = RecDepthBlockText;

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

else

{

if (!int.TryParse(textBox.Text, out int x) || x <= 0 || x > maxDepth)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

}

};

startColorTextBox.GotFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (textBox.Text == StartColorBoxText)

{

textBox.Text = string.Empty;

}

if (!CheckColorString.Check(textBox.Text))

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

};

startColorTextBox.LostFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

textBox.BorderBrush = new SolidColorBrush(

Color.FromRgb(170, 170, 170));

if (textBox.Text == string.Empty)

{

textBox.Text = StartColorBoxText;

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

else

{

if (!CheckColorString.Check(textBox.Text))

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

}

};

endColorTextBox.GotFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (textBox.Text == EndColorBoxText

)

{

textBox.Text = string.Empty;

}

if (!CheckColorString.Check(textBox.Text))

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

};

endColorTextBox.LostFocus += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

textBox.BorderBrush = new SolidColorBrush(

Color.FromRgb(170, 170, 170));

if (textBox.Text == string.Empty)

{

textBox.Text = EndColorBoxText;

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

else

{

if (!CheckColorString.Check(textBox.Text))

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

}

};

//if the input is incorrect then make the bottom border red to inform user

recursionDepthTextBox.KeyUp += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (!int.TryParse(textBox.Text, out int x) || x > maxDepth || x <= 0)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

recursionDepthSlider.Value = double.Parse(textBox.Text);

}

};

startLengthTextBox.KeyUp += (sender, e) =>

{

TextBox textBox = (TextBox)sender;

if (!double.TryParse(textBox.Text, out double x) || x <= 0 ||

x > maxLength)

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

startLengthSlider.Value = double.Parse(textBox.Text);

}

};

startLengthSlider.ValueChanged += (sender, e) =>

{

startLengthTextBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

startLengthTextBox.Text = ((int)startLengthSlider.Value).ToString();

};

recursionDepthSlider.ValueChanged += (sender, e) =>

{

recursionDepthTextBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

recursionDepthTextBox.Text = ((int)recursionDepthSlider.Value).ToString();

};

void BoxesKeyUp(object sender, EventArgs e)

{

TextBox textBox = (TextBox)sender;

if (CheckColorString.Check(textBox.Text))

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyle"] as Style;

}

else

{

textBox.Style = mainWindow.Resources["paramsTextBoxStyleWrong"] as Style;

}

}

startColorTextBox.KeyUp += BoxesKeyUp;

endColorTextBox.KeyUp += BoxesKeyUp;

#endregion

#region Add Children

paramsGrid.Children.Add(headerTextBox);

paramsGrid.Children.Add(startLengthTextBox);

paramsGrid.Children.Add(startLengthSlider);

paramsGrid.Children.Add(recursionDepthTextBox);

paramsGrid.Children.Add(recursionDepthSlider);

paramsGrid.Children.Add(startColorTextBox);

paramsGrid.Children.Add(endColorTextBox);

paramsGrid.Children.Add(hintForCheckBox);

paramsGrid.Children.Add(drawSpeedRadioBtn);

#endregion

}

}

}

**CreateNFFractalMenu.cs**

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

public class CreateNFFractalMenu : CreateFractalMenu

{

public override void CreateMenu(Grid paramsGrid, Window mainWindow,

string fractalName, int maxLength, int maxDepth, string currLength,

string currDepth, string currStartColor, string currEndColor)

{

Button oldButton = null;

if (paramsGrid.Children.Count != 4)

{

oldButton = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1];

}

base.CreateMenu(paramsGrid, mainWindow, fractalName, maxLength, maxDepth,

currLength, currDepth, currStartColor, currEndColor);

#region Elements

Button goBtn = new Button

{

Background = new SolidColorBrush(Color.FromRgb(255, 0, 255)),

Width = 100,

Height = 40,

Content = "Go",

FontFamily = new FontFamily("Arial"),

Foreground = new SolidColorBrush(Color.FromRgb(255, 255, 255)),

FontSize = 15,

VerticalAlignment = VerticalAlignment.Bottom,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 0, 0, 20),

Style = mainWindow.Resources["goBtnStyle"] as Style

};

#endregion

#region Event Handlers

goBtn.MouseEnter += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(210, 0, 210));

};

goBtn.MouseLeave += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(255, 0, 255));

};

#endregion

if (oldButton == null)

{

paramsGrid.Children.Add(goBtn);

}

else

{

paramsGrid.Children.Add(oldButton);

}

}

}

}

**CreateQCFractalMenu.cs**

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

public class CreateQCFractalMenu : CreateFractalMenu

{

public override void CreateMenu(Grid paramsGrid, Window mainWindow,

string fractalName, int maxLength, int maxDepth, string currLength,

string currDepth, string currStartColor, string currEndColor)

{

Button oldButton = null;

if (paramsGrid.Children.Count != 4)

{

oldButton = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1];

}

base.CreateMenu(paramsGrid, mainWindow, fractalName, maxLength, maxDepth,

currLength, currDepth, currStartColor, currEndColor);

#region Elements

Button goBtn = new Button

{

Background = new SolidColorBrush(Color.FromRgb(255, 0, 255)),

Width = 100,

Height = 40,

Content = "Go",

FontFamily = new FontFamily("Arial"),

Foreground = new SolidColorBrush(Color.FromRgb(255, 255, 255)),

FontSize = 15,

VerticalAlignment = VerticalAlignment.Bottom,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 0, 0, 20),

Style = mainWindow.Resources["goBtnStyle"] as Style

};

Button additionalSettings = new Button()

{

Background = new SolidColorBrush(Color.FromRgb(255, 0, 255)),

Width = 100,

Height = 40,

Content = "Additional settings",

FontFamily = new FontFamily("Arial"),

Foreground = new SolidColorBrush(Color.FromRgb(255, 255, 255)),

FontSize = 11,

VerticalAlignment = VerticalAlignment.Bottom,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(120, 0, 0, 20),

Style = mainWindow.Resources["goBtnStyle"] as Style

};

#endregion

#region Event Handlers

goBtn.MouseEnter += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(210, 0, 210));

};

goBtn.MouseLeave += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(255, 0, 255));

};

additionalSettings.MouseEnter += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(210, 0, 210));

};

additionalSettings.MouseLeave += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(255, 0, 255));

};

#endregion

paramsGrid.Children.Add(additionalSettings);

if (oldButton == null)

{

paramsGrid.Children.Add(goBtn);

}

else

{

paramsGrid.Children.Add(oldButton);

}

}

}

}

**CreateSCFractalMenu.cs**

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

public class CreateSCFractalMenu : CreateFractalMenu

{

public override void CreateMenu(Grid paramsGrid, Window mainWindow,

string fractalName, int maxLength, int maxDepth, string currLength,

string currDepth, string currStartColor, string currEndColor)

{

Button oldButton = null;

if (paramsGrid.Children.Count != 4)

{

oldButton = (Button)paramsGrid.Children[paramsGrid.Children.Count - 1];

}

base.CreateMenu(paramsGrid, mainWindow, fractalName, maxLength, maxDepth,

currLength, currDepth, currStartColor, currEndColor);

#region Elements

Button goBtn = new Button

{

Background = new SolidColorBrush(Color.FromRgb(255, 0, 255)),

Width = 100,

Height = 40,

Content = "Go",

FontFamily = new FontFamily("Arial"),

Foreground = new SolidColorBrush(Color.FromRgb(255, 255, 255)),

FontSize = 15,

VerticalAlignment = VerticalAlignment.Bottom,

HorizontalAlignment = HorizontalAlignment.Left,

Margin = new Thickness(10, 0, 0, 20),

Style = mainWindow.Resources["goBtnStyle"] as Style

};

#endregion

#region Event Handlers

goBtn.MouseEnter += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(210, 0, 210));

};

goBtn.MouseLeave += (sender, e) =>

{

((Button)sender).Background = new SolidColorBrush(

Color.FromRgb(255, 0, 255));

};

#endregion

if (oldButton == null)

{

paramsGrid.Children.Add(goBtn);

}

else

{

paramsGrid.Children.Add(oldButton);

}

}

}

}

**DrawingParameters.cs**

using System.Windows;

using System.Windows.Media;

namespace Graphics

{

/// <summary>

/// Parent class to all drawing parameters classes

/// which are transmited to the Draw method

/// </summary>

public class DrawingParameters

{

public double CurrentLength { get; }

public int RecursionLevel { get; }

public Point CurrentCoords { get; set; }

public DrawingParameters(double currentLength,

int recLvl, Point currPoint)

{

CurrentLength = currentLength;

RecursionLevel = recLvl;

CurrentCoords = currPoint;

}

}

}

**Fractal.cs**

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

abstract public class Fractal

{

#region Private variables

//length of the start element (radius, segment)

public double StartLength { get; }

public Color StartColor { get; }

public Color EndColor { get; }

public int MaxRecursionLevel { get; }

#endregion

public Fractal(double startLength, Color startColor, Color endColor, int maxRec)

{

StartLength = startLength;

StartColor = startColor;

EndColor = endColor;

MaxRecursionLevel = maxRec;

}

public abstract Task<Canvas> Draw(Dispatcher dispatcher, DispatcherPriority priority,

CancellationToken token, DrawingParameters drawingParameters);

}

}

**CheckColorString.cs**

namespace Graphics

{

public static class CheckColorString

{

private static string AllowedSymbols { get; set; } = "#1234567890ABCDEF";

public static bool Check(string input)

{

if (input.Length != 7)

{

return false;

}

for (int i = 0; i < input.Length; i++)

{

if (AllowedSymbols.IndexOf(input[i]) < 0)

{

return false;

}

}

return true;

}

}

}

**GetCurrentColor.cs**

using System.Windows.Media;

namespace Graphics

{

public static class GetCurrentColor

{

/// <summary>

/// Calculates delta which is then added to a minimum color (R, G, B) value

/// </summary>

/// <param name="cMax">

/// Max value of R or G or B in the "end color"

/// </param>

/// <param name="cMin">

/// Min value of R or G or B in the "start" color

/// </param>

private static int GetDelta(int cMin, int cMax, int currRec, int size)

{

return (int)((double)(cMax - cMin) \* currRec / size);

}

/// <summary>

/// Creates a new solid color brush which is initialized with the color

/// which is calculated with a linear gradient formula

/// </summary>

public static SolidColorBrush Get(Color startColor, Color endColor,

int currRecNum, int maxRecNum)

{

byte rAverage = (byte)(startColor.R + GetDelta(startColor.R, endColor.R,

currRecNum, maxRecNum));

byte gAverage = (byte)(startColor.G + GetDelta(startColor.G, endColor.G,

currRecNum, maxRecNum));

byte bAverage = (byte)(startColor.B + GetDelta(startColor.B, endColor.B,

currRecNum, maxRecNum));

return new SolidColorBrush(Color.FromRgb(rAverage, gAverage, bAverage));

}

}

}

**NFFractalPointData.cs**

using System.Windows;

using System.Windows.Media;

namespace Graphics

{

public class NFFractalPointData : PointData

{

public Point[] PointsArr { get; set; }

public NFFractalPointData(Point[] points, SolidColorBrush brush)

: base(0, brush, new Point())

{

PointsArr = points;

}

}

}

**PointData.cs**

using System.Windows.Media;

using System.Windows;

namespace Graphics

{

public class PointData

{

public double Length { get; set; }

public SolidColorBrush CurrentBrush{ get; set; }

public Point Coords { get; set; }

public PointData(double length, SolidColorBrush brush, Point point)

{

Length = length;

CurrentBrush = brush;

Coords = point;

}

}

}

**StartDraw.cs**

using System;

using System.Threading;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Threading;

using System.Windows.Controls;

using System.Windows.Media;

namespace Graphics

{

public class StartDraw

{

#region Properties

public Canvas MainCanvas { get; set; }

public Canvas TempCanvas { get; set; }

public int CurrentFractal { get; set; }

public Color StartColor { get; set; }

public Color EndColor { get; set; }

public Button GoBtn { get; set; }

public DispatcherPriority DispatcherPriority { get; set; }

public bool[] QCSidesDrawability { get; set; }

#endregion

/// <summary>

/// Enables go btn and set the text to "Go!"

/// </summary>

public void EnableGoBtn()

{

GoBtn.IsEnabled = true;

GoBtn.Content = "Go!";

}

/// <summary>

/// Disables go btn and set the text to "Drawing..."

/// </summary>

public void DisableGoBtn()

{

GoBtn.IsEnabled = false;

GoBtn.Content = "Drawing...";

}

/// <summary>

/// Draws a fractal on the main Canvas using the several pararmeters

/// to initalize the drawing process

/// </summary>

public async Task<Canvas> DrawFractal(double length, int depth, Point startPoint,

Dispatcher dispatcher, CancellationToken token)

{

try

{

Disable();

switch (CurrentFractal)

{

case 1:

QCFractal qcFractal = new QCFractal(length,

StartColor, EndColor, depth, MainCanvas);

DrawingParameters qcParams = new QCDrawingParams(length, 1,

startPoint, QCSidesDrawability);

if (DispatcherPriority == DispatcherPriority.Send)

{

TempCanvas = new Canvas

{

Width = 5000,

Height = 5000,

Background = new SolidColorBrush(Color.FromRgb(255, 255, 255))

};

qcFractal.MainCanvas = TempCanvas;

return await Task.Run(() => qcFractal.Draw(dispatcher,

DispatcherPriority, token, qcParams));

}

else

{

await Task.Run(() => qcFractal.Draw(dispatcher,

DispatcherPriority, token, qcParams));

}

return null;

case 2:

SCFractal scFractal = new SCFractal(length,

StartColor, EndColor,

depth, MainCanvas);

DrawingParameters scParams = new SCDrawingParams(length, 1,

startPoint);

scFractal.InitialDraw(scParams);

if (DispatcherPriority == DispatcherPriority.Send)

{

TempCanvas = new Canvas

{

Width = 5000,

Height = 5000,

Background = new SolidColorBrush(Color.FromRgb(255, 255, 255))

};

scFractal.MainCanvas = TempCanvas;

return await Task.Run(() => scFractal.Draw(dispatcher, DispatcherPriority,

token, scParams));

}

else

{

await Task.Run(() => scFractal.Draw(dispatcher, DispatcherPriority,

token, scParams));

}

return null;

case 3:

NFFractal nfFractal = new NFFractal(length,

StartColor, EndColor,

depth, MainCanvas);

DrawingParameters nfParams = new NFDrawingParams(length, 1,

startPoint);

if (DispatcherPriority == DispatcherPriority.Send)

{

TempCanvas = new Canvas

{

Width = 5000,

Height = 5000,

Background = new SolidColorBrush(Color.FromRgb(255, 255, 255))

};

nfFractal.MainCanvas = TempCanvas;

return await Task.Run(() => nfFractal.Draw(dispatcher, DispatcherPriority,

token, nfParams));

}

else

{

await Task.Run(() => nfFractal.Draw(dispatcher, DispatcherPriority,

token, nfParams));

}

return null;

default:

MessageBox.Show("The fatal error happened, this should be never seen...");

return null;

}

}

catch (OutOfMemoryException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (StackOverflowException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (IndexOutOfRangeException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (NullReferenceException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (TaskCanceledException ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

catch (Exception ex)

{

dispatcher.Invoke(() => MessageBox.Show(ex.Message,

"Soft's message", MessageBoxButton.OK,

MessageBoxImage.Information));

return null;

}

finally

{

Enable();

}

}

public void Enable()

{

EnableGoBtn();

}

public void Disable()

{

DisableGoBtn();

}

}

}

**MainWindow.xaml**

<Window x:Class="Fractals.MainWindow"

x:Name="mainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d"

Title="Fractals"

MinHeight="600"

MinWidth="1024"

SizeChanged="MainWindowSizeChanged"

KeyDown="EnterKeyDown">

<Window.Resources>

<Style x:Key="selectRecDirectionBtnStyle" TargetType="Button">

<Setter Property="OverridesDefaultStyle" Value="True"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

<Border Name="border"

BorderThickness="0"

Padding="4,2"

BorderBrush="DarkGray"

CornerRadius="30"

Background="{TemplateBinding Background}">

<ContentPresenter HorizontalAlignment="Center"

VerticalAlignment="Center" />

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

<Style x:Key="goBtnStyle" TargetType="Button">

<Setter Property="OverridesDefaultStyle" Value="True"/>

<Setter Property="Margin" Value="5"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

<Border Name="border"

BorderThickness="0"

Padding="4,2"

BorderBrush="DarkGray"

CornerRadius="5"

Background="{TemplateBinding Background}">

<ContentPresenter HorizontalAlignment="Center"

VerticalAlignment="Center" />

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

<Style x:Key="paramsTextBoxStyle" TargetType="TextBox">

<Setter Property="Background" Value="White"/>

<Setter Property="Foreground" Value="#AAAAAA"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="TextBox">

<Border Name="Bd"

BorderThickness="0, 0, 0, 2"

BorderBrush="#AAAAAA"

SnapsToDevicePixels="true">

<ScrollViewer Name="PART\_ContentHost"

Background="#FFFFFF"/>

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

<Style x:Key="paramsTextBoxStyleWrong" TargetType="TextBox">

<Setter Property="Background" Value="White"/>

<Setter Property="Foreground" Value="#AAAAAA"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="TextBox">

<Border Name="Bd"

BorderThickness="0, 0, 0, 2"

BorderBrush="#FF0000"

SnapsToDevicePixels="true">

<ScrollViewer Name="PART\_ContentHost"

Background="#FFFFFF"/>

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</Window.Resources>

<Grid>

<!-- Header -->

<Grid x:Name="headerGrid"

Height="100"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Background="White">

<TextBlock x:Name="headBlock"

Text="F R A C T A L S."

VerticalAlignment="Center"

HorizontalAlignment="Left"

Margin="20, 2, 0, 0"

FontSize="50"

FontFamily="CENTURY SCHOOLBOOK"

FontStyle="Normal"

FontWeight="ExtraBold"

Foreground="Red"

TextAlignment="Center"/>

<Grid x:Name="toplineGrid"

Height="2"

HorizontalAlignment="Left"

VerticalAlignment="Bottom"

Background="LightGray"/>

</Grid>

<Grid x:Name="leftGrid"

Background="White"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Margin="0, 100, 0, 0"

Width="150">

<!--Three fractals options in the main left menu-->

<Canvas x:Name="qcFractal"

MouseEnter="MenuCanvasMouseEnter"

MouseLeave="MenuCanvasMouseLeave"

MouseDown="MenuCanvasMouseDown"

Width="148"

Height="148"

Background="White"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Margin="0.5, 0, 0, 0"/>

<Canvas x:Name="scFractal"

MouseEnter="MenuCanvasMouseEnter"

MouseLeave="MenuCanvasMouseLeave"

MouseDown="MenuCanvasMouseDown"

Width="148"

Height="130"

Background="White"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Margin="0.5, 130, 0, 0"/>

<Canvas x:Name="nfFractal"

MouseEnter="MenuCanvasMouseEnter"

MouseLeave="MenuCanvasMouseLeave"

MouseDown="MenuCanvasMouseDown"

Width="148"

Height="148"

Background="White"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Margin="0.5, 278, 0, 0"/>

<Grid x:Name="leftLineGrid"

HorizontalAlignment="Right"

VerticalAlignment="Top"

Width="1"

Background="LightGray"/>

</Grid>

<!--Main work place-->

<Grid x:Name="mainGrid"

Background="#E8E8E8"

HorizontalAlignment="Left"

VerticalAlignment="Top">

<Grid x:Name="paramsGrid"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Background="White"

Width="250">

<Rectangle Margin="10, 10, 10, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Width="230"

Height="30"

Fill="#DCDCDC"/>

<Rectangle Margin="10, 90, 10, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Width="230"

Height="2"

Fill="#DCDCDC"/>

<Rectangle Margin="10, 120, 10, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Width="230"

Height="2"

Fill="#DCDCDC"/>

<Rectangle Margin="10, 0, 10, 20"

VerticalAlignment="Bottom"

HorizontalAlignment="Left"

Width="100"

Height="40"

Fill="#FF00FF"/>

</Grid>

<Grid x:Name="qCExtraParamsGrid"

Width="165"

Height="185"

Background="White"

Margin ="270, 10, 0, 0"

HorizontalAlignment="Left"

VerticalAlignment="Top"

ToolTip="Select three ellipses"

Visibility="Collapsed">

<TextBlock VerticalAlignment="Top"

HorizontalAlignment="Center"

Text="Set extra params here: "

FontFamily="Arial"

Margin="0, 5, 0, 0"

FontSize="12"

FontWeight="Bold"

Foreground="DarkGray"/>

<Ellipse Width="60"

Height="60"

Margin="0, 20, 0, 0"

VerticalAlignment="Center"

HorizontalAlignment="Center"

Fill="LightGray"/>

<Ellipse x:Name="leftEllipse"

Width="30"

Height="30"

Fill="Orange"

VerticalAlignment="Center"

HorizontalAlignment="Center"

Margin="0, 20, 90, 0"

MouseEnter="SelectEllipseMouseEnter"

MouseLeave="SelectEllipseMouseLeave"

MouseDown="SelectEllipseMouseDown"/>

<Ellipse x:Name="rightEllipse"

Width="30"

Height="30"

Fill="Orange"

VerticalAlignment="Center"

HorizontalAlignment="Center"

Margin="90, 20, 0, 0"

MouseEnter="SelectEllipseMouseEnter"

MouseLeave="SelectEllipseMouseLeave"

MouseDown="SelectEllipseMouseDown"/>

<Ellipse x:Name="topEllipse"

Width="30"

Height="30"

Fill="Orange"

VerticalAlignment="Center"

HorizontalAlignment="Center"

Margin="0, -70, 0, 0"

MouseEnter="SelectEllipseMouseEnter"

MouseLeave="SelectEllipseMouseLeave"

MouseDown="SelectEllipseMouseDown"/>

<Ellipse x:Name="bottomEllipse"

Width="30"

Height="30"

Fill="DarkGray"

VerticalAlignment="Center"

HorizontalAlignment="Center"

Margin="0, 110, 0, 0"

MouseEnter="SelectEllipseMouseEnter"

MouseLeave="SelectEllipseMouseLeave"

MouseDown="SelectEllipseMouseDown"/>

</Grid>

<ScrollViewer x:Name="mainCanvasScroll"

Margin="270, 10, 0, 0"

Width="500"

Height="500"

HorizontalScrollBarVisibility="Auto"

VerticalScrollBarVisibility="Auto"

VerticalAlignment="Top"

Background="White"

HorizontalAlignment="Left">

<Canvas x:Name="mainCanvas"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Background="White"

Width="5000"

Height="5000"

Margin="0, 0, 0, 0">

</Canvas>

</ScrollViewer>

<Grid x:Name="canvasOptionsGrid"

Width="50"

Height="250"

Background="White"

HorizontalAlignment="Left"

VerticalAlignment="Top">

<Grid x:Name="enlargeGridFirst"

Width="50"

Height="50"

Margin="0, 0, 0, 0"

HorizontalAlignment="Left"

VerticalAlignment="Top"

MouseEnter="EnlargeIconMouseEnter">

<Image Source="Icons/enlargeFirst.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid x:Name="enlargeGridSecond"

Width="50"

Height="50"

Margin="0, 0, 0, 0"

HorizontalAlignment="Left"

VerticalAlignment="Top"

Visibility="Hidden"

ToolTip="Enlarge image"

MouseLeave="EnlargeIconMouseLeave"

MouseDown="EnlargeIconMouseDown">

<Image Source="Icons/enlargeSecond.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid Width="50"

Height="1"

Background="LightGray"

Margin="0, 50, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"/>

<Grid x:Name="reduceGridFirst"

Width="50"

Height="50"

Margin="0, 51, 0, 0"

MouseEnter="ReduceIconMouseEnter"

VerticalAlignment="Top"

HorizontalAlignment="Left">

<Image Source="Icons/reduceFirst.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid x:Name="reduceGridSecond"

Width="50"

Height="50"

Margin="0, 51, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Visibility="Hidden"

ToolTip="Reduce image"

MouseLeave="ReduceIconMouseLeave"

MouseDown="ReduceIconMouseDown">

<Image Source="Icons/reduceSecond.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid Width="50"

Height="1"

Background="LightGray"

Margin="0, 101, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"/>

<Grid x:Name="cancelDrawingFirst"

Width="50"

Height="50"

Margin="0, 102, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Visibility="Visible"

MouseEnter="CancelDrawingMouseEnter">

<Image Source="Icons/cancelFirst.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid x:Name="cancelDrawingSecond"

Width="50"

Height="50"

Margin="0, 102, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Visibility="Hidden"

ToolTip="Cancel PROCESS of drawing"

MouseLeave="CancelDrawingMouseLeave"

MouseDown="CancelCurrentDrawingMouseDown">

<Image Source="Icons/cancelSecond.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid Width="50"

Height="1"

Background="LightGray"

Margin="0, 152, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"/>

<Grid x:Name="exportPngFirst"

Width="50"

Height="50"

Margin="0, 153, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Visibility="Visible"

MouseEnter="ExportPngFirstEnter">

<Image Source="Icons/exportFirst.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

<Grid x:Name="exportPngSecond"

Width="50"

Height="50"

Margin="0, 153, 0, 0"

VerticalAlignment="Top"

HorizontalAlignment="Left"

Visibility="Hidden"

ToolTip="Export"

MouseLeave="ExportPngSecondLeave"

MouseDown="ExportPngSecondClick">

<Image Source="Icons/exportSecond.png"

Width="40"

Height="40"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Margin="0, 0, 0, 0"/>

</Grid>

</Grid>

</Grid>

</Grid>

</Window>

1. **СПИСОК ЛИТЕРАТУРЫ**
2. Metanit [Электронный ресурс] // https://metanit.com [сайт]. [2018]. URL: <https://metanit.com/sharp/tutorial/12.1.php>  (дата обращения: 1.12.2018).
3. Metanit [Электронный ресурс] // https://metanit.com [сайт]. [2018]. URL: https://metanit.com/sharp/tutorial/12.2.php  (дата обращения: 1.12.2018).
4. Metanit [Электронный ресурс] // https://metanit.com [сайт]. [2018]. URL: <https://metanit.com/sharp/tutorial/12.3.php>  (дата обращения: 1.12.2018).
5. Metanit [Электронный ресурс] // https://metanit.com [сайт]. [2018]. URL: <https://metanit.com/sharp/tutorial/12.4.php>  (дата обращения: 1.12.2018).
6. Metanit [Электронный ресурс] // https://metanit.com [сайт]. [2018]. URL: <https://metanit.com/sharp/tutorial/12.5.php>  (дата обращения: 1.12.2018).
7. ProfessorWeb [Электронный ресурс] // https://professorweb.ru [сайт]. [2018]. URL: https://professorweb.ru/my/WPF/documents\_WPF/level31/31\_2.php  (дата обращения: 1.12.2018).

**ПРИЛОЖЕНИЕ 1**

**СБОРКА ДЛЯ УСПЕШНОГО ЗАПУСКА ПРИЛОЖЕНИЯ.**

Список файлов, необходимых для успешного запуска программы:

1. Fractals.exe
2. Fractals.exe.config
3. Fractals.pdb
4. Graphics.dll
5. Graphics.pdb

На Рисунке 11 приведен скриншот – примерный вид того, как должна выглядеть папка со всеми этими файлами.

Рисунок 11

