Web GUI for Iterative Function Systems

Chiciuc Nicușor

July 21, 2014

Contents

1 Introduction

The following paper will discuss about Iterated Function Systems using a program written by the author.

1.1 General Iterative Function Systems

Iterated Function Systems or IFS are a method for constructing fractals. The fractals are made of the union of several copies of itself, each copy being transformed by a function (hence function system).

2 Demonstration

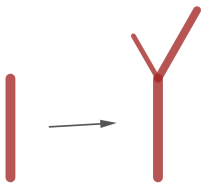
2.1 An colorful approach

Suppose we have the following transformation (Figure 1a).

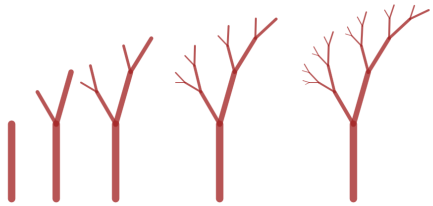
That is, transform a segment into a branch composed of three segments. This process can be applied to the two smaller branches. Repeating this process some numbers of times reveals a beautiful, fractal pattern (Figure 1b). This pattern is very similar to the *Barnsley fern* [7].

Figure 1: First steps.

(a) A simple transformation.

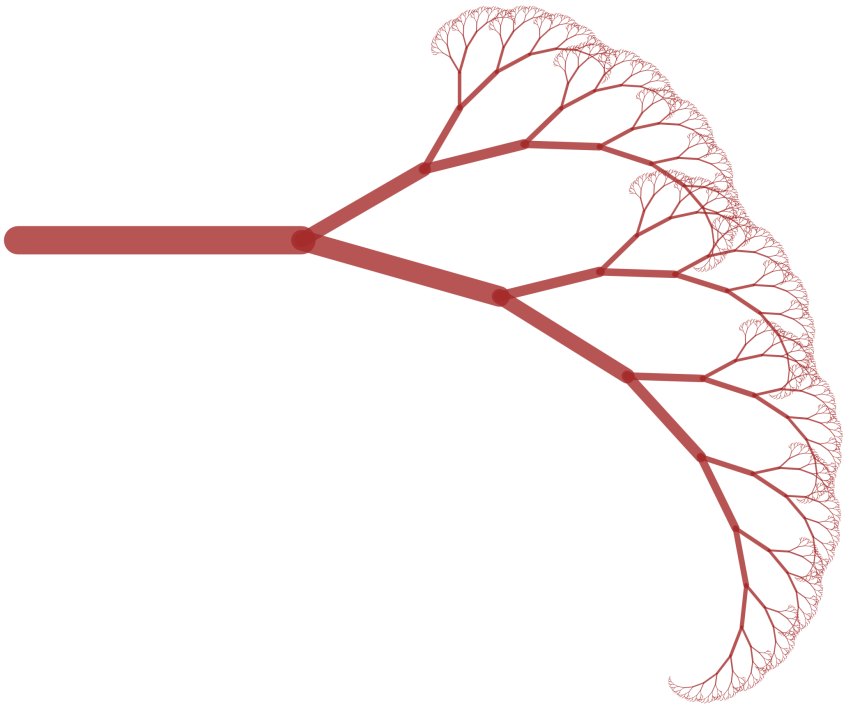


(b) Several level of recursion.



A much more detailed rendering can be observed in Figure 2

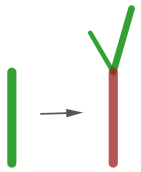
Figure 2: A deeper recursion level.



Suppose then, that one would like to define several such functions, and combine them together. In order to accomplish that, there should be a way to tell apart different kinds of segments. A very

straightforward method would be to give each segment a different color. In such a situation the previous example would translate to this (Figure 3).

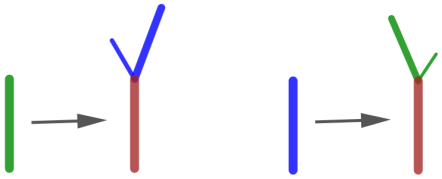
Figure 3: Different Colors



In this case the red segments will remain a simple segment while the green ones will transform further. Of course, it would be much clear if I would use arrows instead of simple segments because it might not be clear in which direction the segment should transform. This will be discussed further in the paper.

The fact that we use different colors allows not only to define a transformation in a much clear way, it also allows to implement much more interesting cases. Suppose that we have two functions defined in the following way.

Figure 4: Two transformation



As one can see (Figure 5), this produces a fractal that is surprisingly similar to a real tree, even though the rules used are fairly simple. If we start with the blue segment instead of the green one, we get a slightly different tree, Figure 6.

Figure 5: A simple fractal tree.

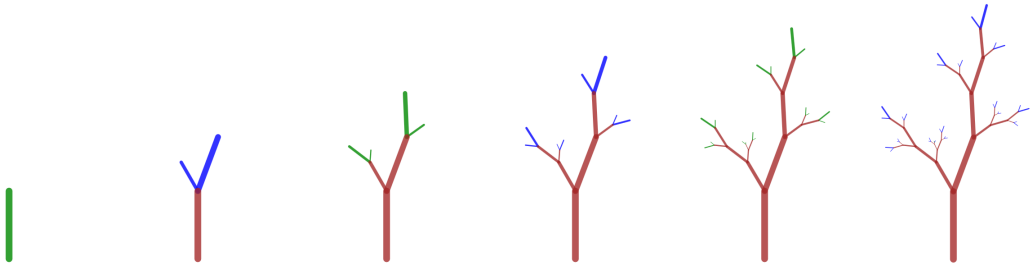


Figure 7 shows the fractals in bigger detail.

Figure 6: Another simple fractal tree.

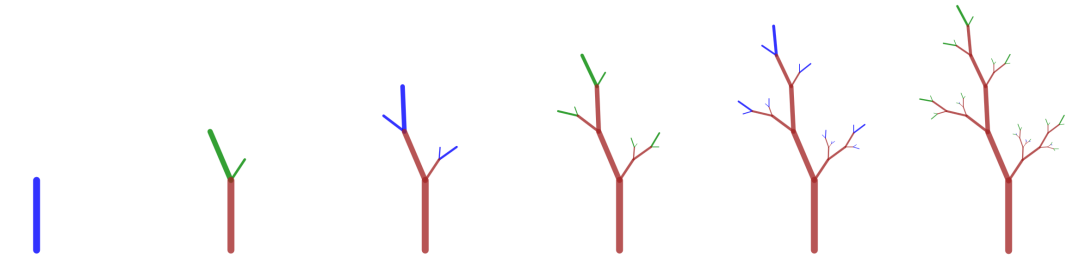
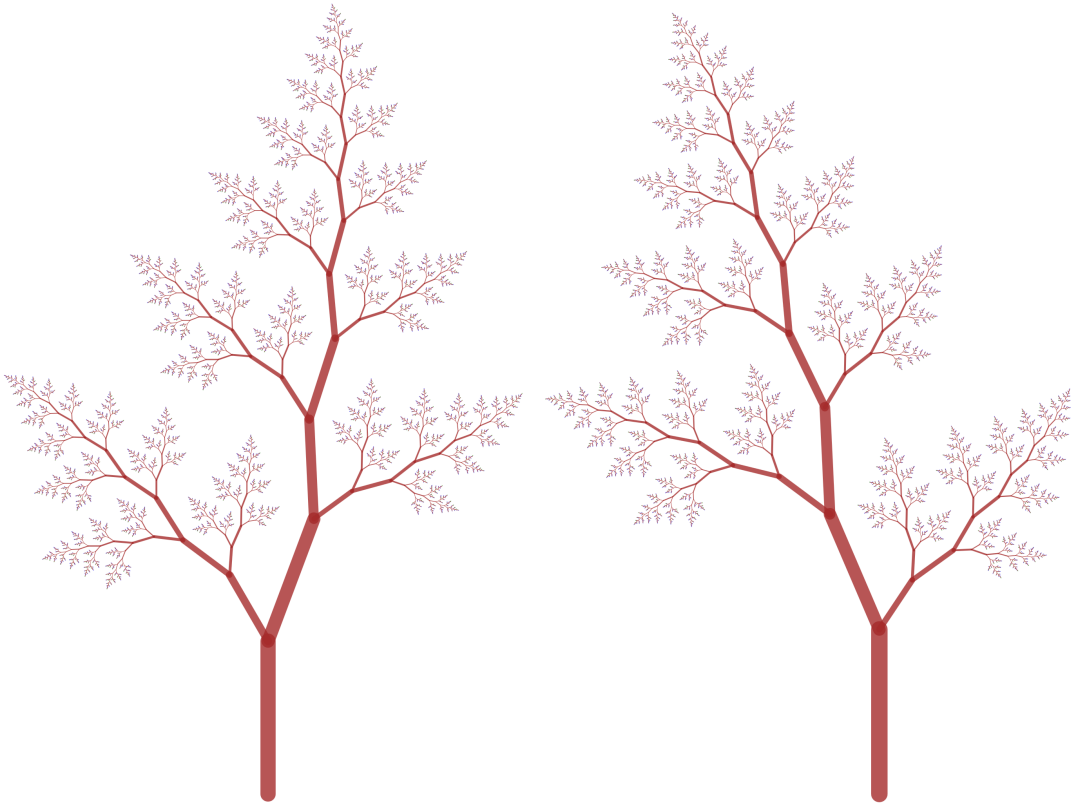


Figure 7: Detailed Trees.



At this moment, most of the readers might understand the basic idea of this project. In the next chapter I will try to explain how the GUI works and how the reader can create fractals by himself.

2.2 First steps

As I mentioned earlier, I started the work on the project with the idea that the program will be easy to use. There might exist other tools that allow the creation of iterated fractals, but most of them require advanced mathematical knowledge to understand and are a little cumbersome to use.

2.2.1 Basic controls

The way in which the user can create and manipulate a transformation is by dragging some circles. The circles can be dragged around by pressing and moving the *left mouse button*. To create a new circle a user can press the *right mouse button* on the background. To delete a circle the user should click the *middle mouse button*.

To create and arrow, one should drag a circle with the *right mouse button*. To delete a segment, one should click the segment with the *middle mouse button*. To change the type of the segment, one should click the segment with the *left mouse button*.

The concept are this:

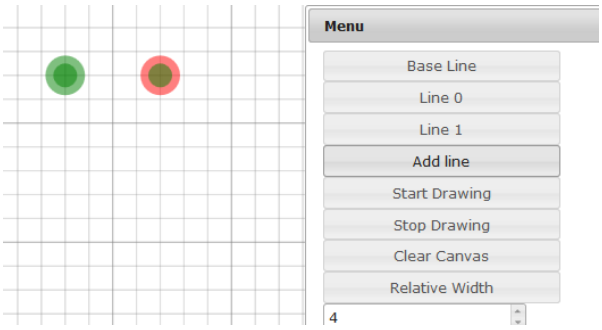
1. left button = move/change
2. right button = create
3. middle button = delete

These were all the basic instructions. Knowing how to use them is enough to create and edit a segment transformation.

2.2.2 The base layer and simple layers

A transformation can be create inside a layer. A new layer can be create by pressing the *Add Layer* button. Figure 8 shows how a new layer looks.

Figure 8: A new layer.



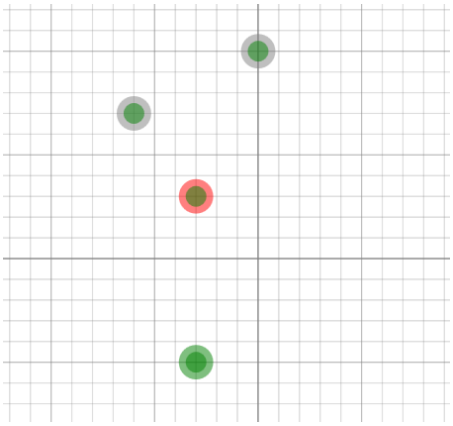
The inner color of the circles represents the color of the transformation. The outer color represents the type of specific circles inside a layer. The circle with the outer color green is the start point and the circle with the outer color red is the end point.

Lets try to implement the tree showed previously. First lets move the circles so that they stand vertically and also add two circles by pressing the *right mouse button* on the background.

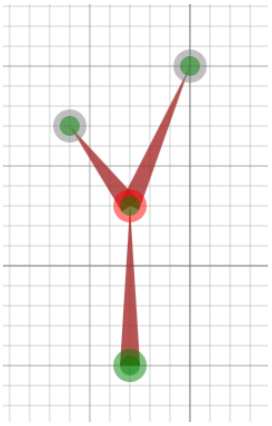
In order to create a segment the user should drag from a disk to another using the right mouse button (Figure 9b).

Figure 9: Points and segments.

(a) Two new points.



(b) The first segments.

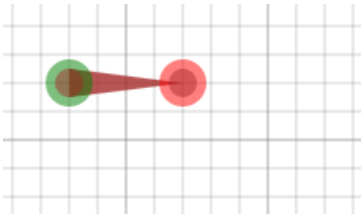


It can be observed that all the arrows have the same color, red (it could be any other color). But we didn't define a red type of transformation, so were do we have it.

2.2.3 Identity Layer

If we'll switch to Layer 0 by clicking the appropriate button we will see this (Fig. 10). You can observe that the inner color of the circles is different, that's because we switched to a different layer.

Figure 10: The identity layer.

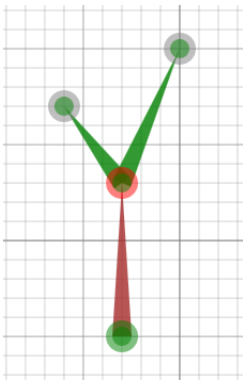


The identity layer is a little bit different from other layers. First of all, even though we can drag the circles around, we cannot add more circles or change the type of the arrow.

The identity layer draws itself, that is, it does not transform to something different. Because of this there are some optimizations that can be implemented. For example instead of recursing further when finding a identity layer, the program will just draw the line because it can infer that the segment will not transform to something different.

Coming back to Layer 1, we can click each of the upper arrows in order to change their colors (Fig. 11).

Figure 11: Different types of arrows.



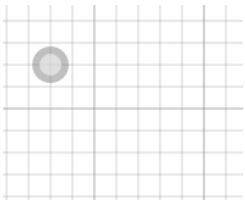
The new color is green, that is, the color of the current layer. If we would click the arrows one more time they would, once again, change their color to the color of the identity layer.

Even though we basically define all that is required we don't see anything happening.

2.2.4 The Base Layer

The uppermost Layer (Fig. 12), called the base layer is different from all the other layers. First of all it does not have a initial green and red circle.

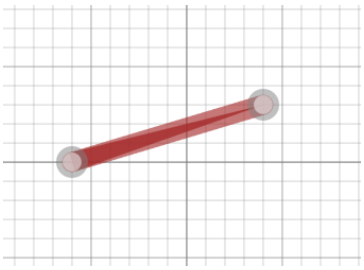
Figure 12: The Base Layer.



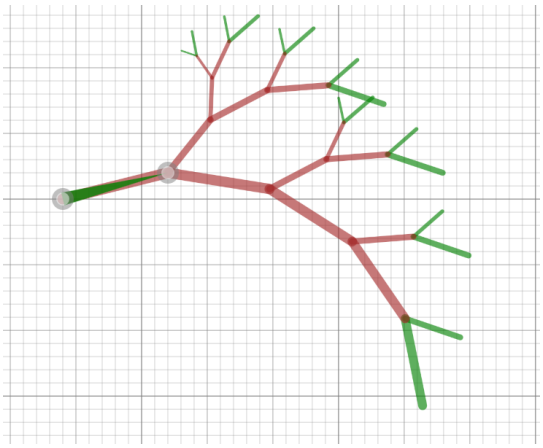
The base layer is used not to define a transformation but to apply them. Lets create another point and connect these two points with an arrow.

Figure 13: Changing the color of the arrow.

(a) First try.



(b) Second try.

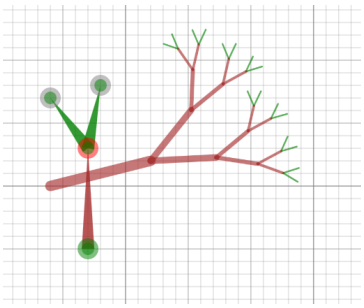


Something interesting happens. Just as we connected the circles, an thick arrow appeared on the background. Because the arrow is of identity type it just created a line. Lets try to press the arrow in order to change the type.

Figure 13b shows that we created our first, own fractal tree. If we move the circles around the tree will move along. This is because the *Dynamic Drawing* check-box is on, we'll talk about this feature later.

Another interesting feature is that if we switch to Layer 1 and move the circles there, the tree will also change (Fig. 14). It is important to not get confused at this point. For some it might be strange that the the root of the tree does not coincide with the identity arrow in Layer 1. This is because Layer are used to define transformation that are applied to the Base Layer. Only the Base Layer defines the position of the initial segments.

Figure 14: Perspective from another layer.



2.3 Common fractals

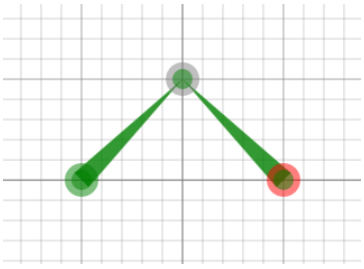
During this and the subsequent chapters I will show some fractals and other geometric structures that can be crated this soft. Also, in this way I will try to give some insight and useful information about the GUI. I think that a hands-on approach would be much more insightful than any other method of explanation.

2.3.1 The Dragon Curve

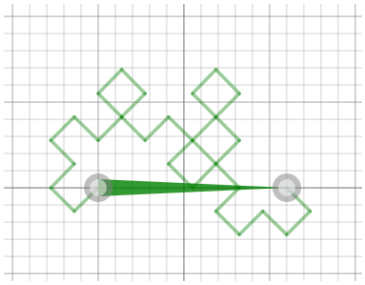
The Dragon Curve is very easy to create. This setup (Fig. 15a) is required in a new layer.

Figure 15: The Dragon Curve.

(a) The setup.



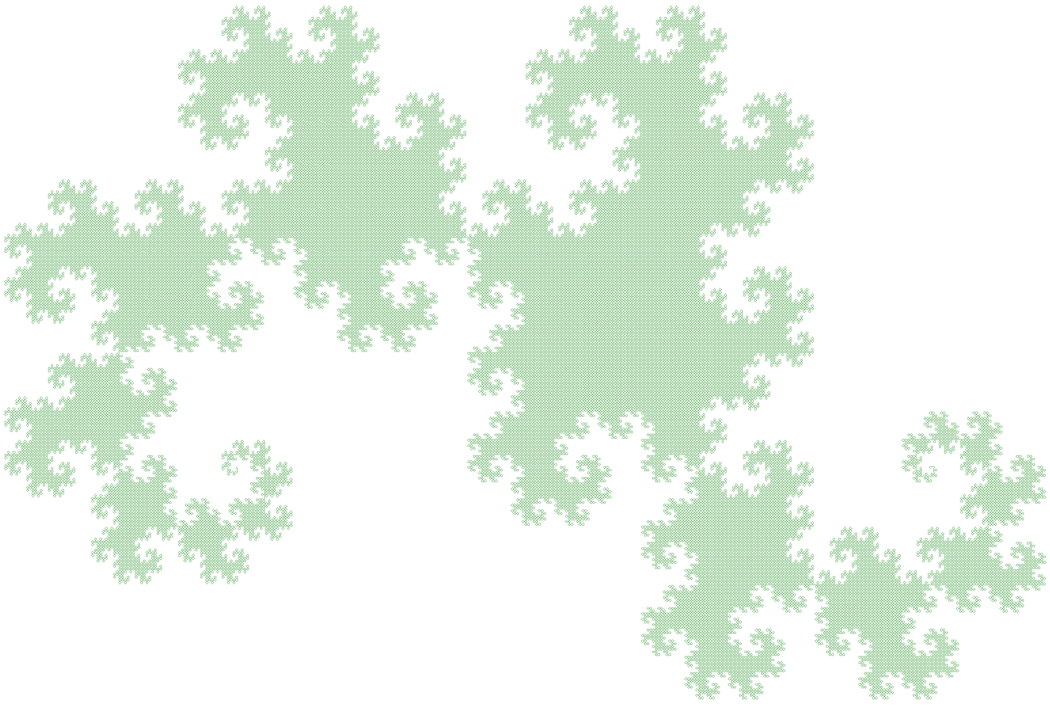
(b) Dragon Curve after 4 steps.



Adding a arrow in the base layer we get this (Fig. 15b).

To increase the recursion level the user must use the spinner in the main dialog. This is what can be created (Fig. 16).

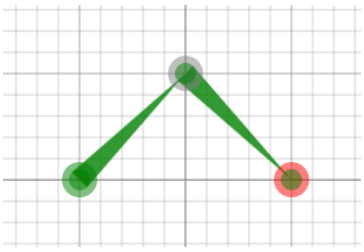
Figure 16: The Dragon Curve.



2.3.2 Levy C Curve

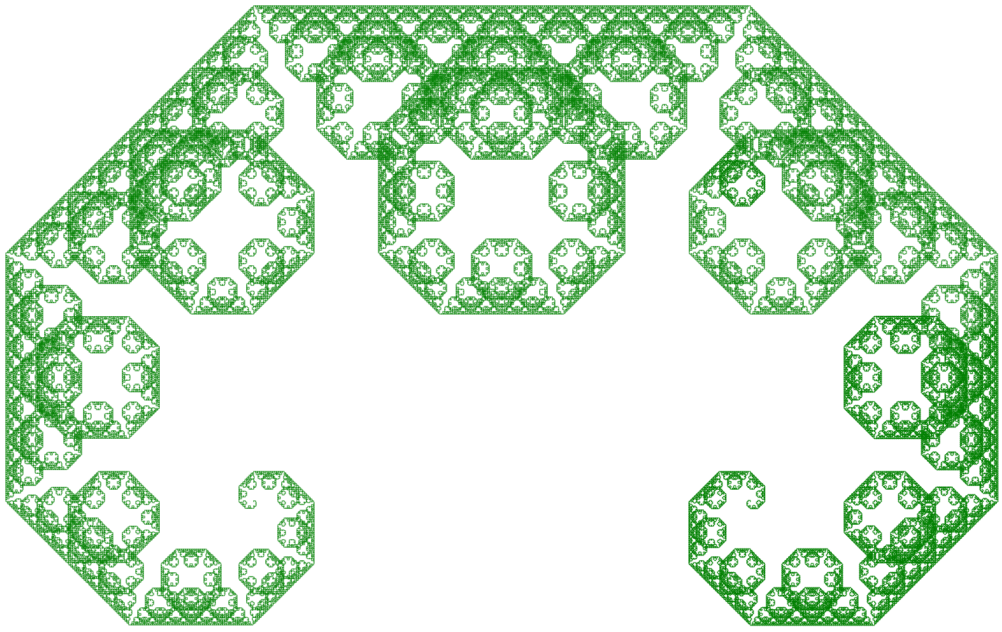
The setup for the Levy C Curve is very similar to the Dragon Curve.

Figure 17: The setup.



It is interesting that changing the direction of the arrow can have such an impact.

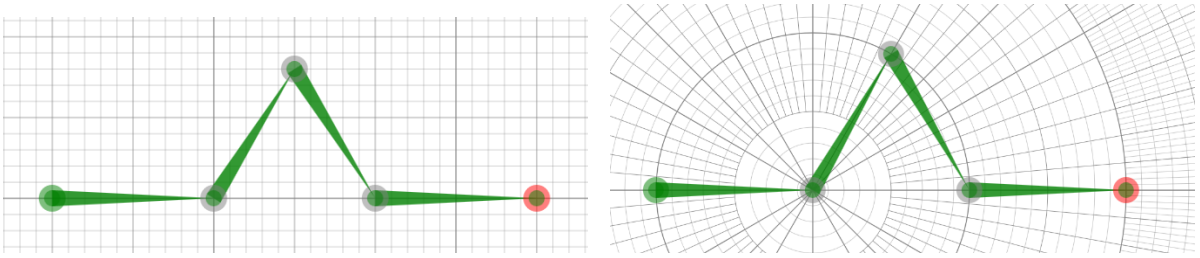
Figure 18: Levy C Curve.



2.3.3 Koch Snowflake and the Polar Grid

Lets try to create the Koch Snowflake.

Figure 19: Koch Snowflake.



(a) Cartesian grid (b) Polar grid

The problem, as one can see, is that it is basically impossible to create an angle of 60 degrees using the cartesian coordinates. To solve this problem I came up with an interesting idea. I implemented a polar grid. To switch to it the user has to press the *Polar Grid* button. The center of the polar grid will be *the last pressed circle*. Below is the new setup.

The setup in the Base Layer can be the following (Fig. 20).

The drawing is in Figure 21.

Figure 20: Base Layer setup.

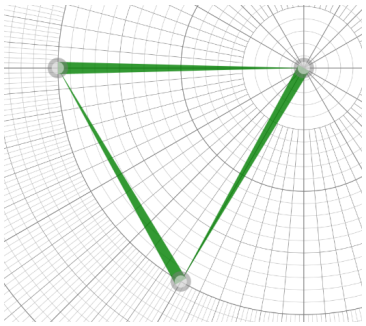
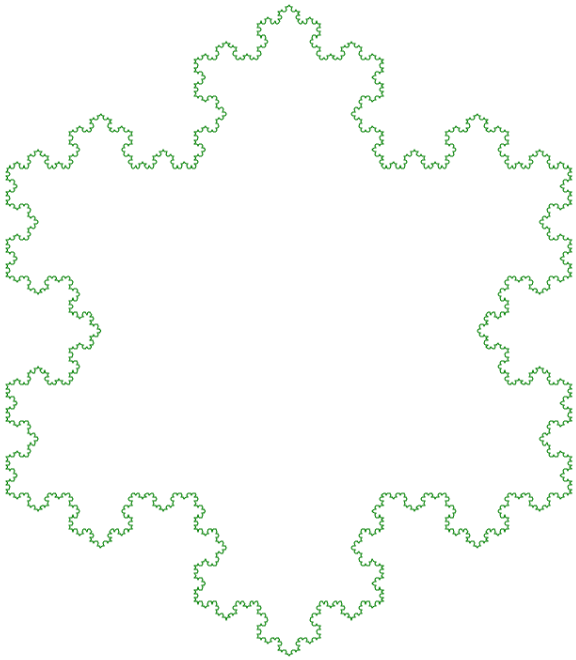


Figure 21: Koch Snowflake.



2.3.4 Some Quadratic curves

Following Koch's concept other types of curves were introduced.

Figure 22: Quadratic type 1.

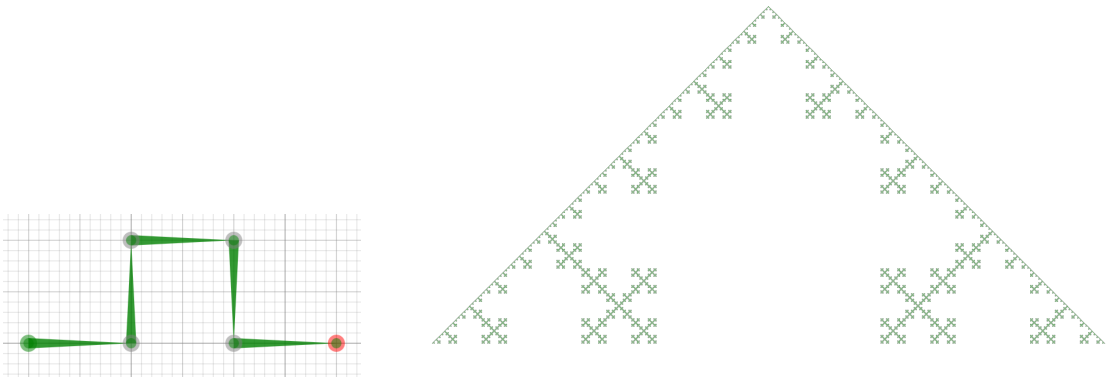
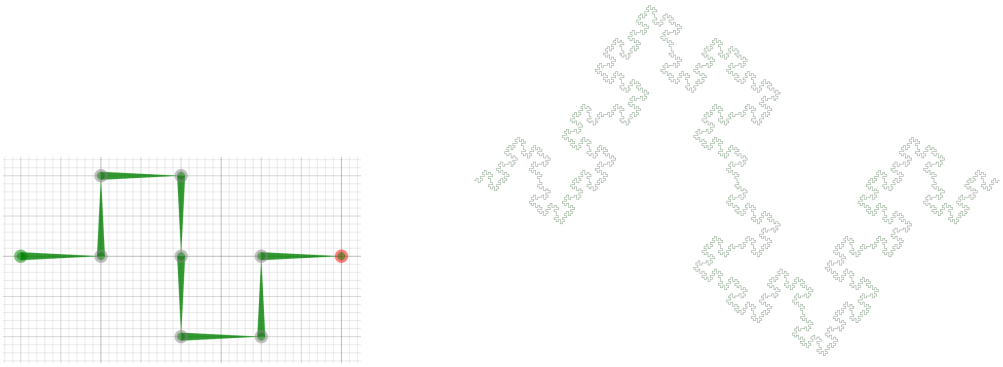


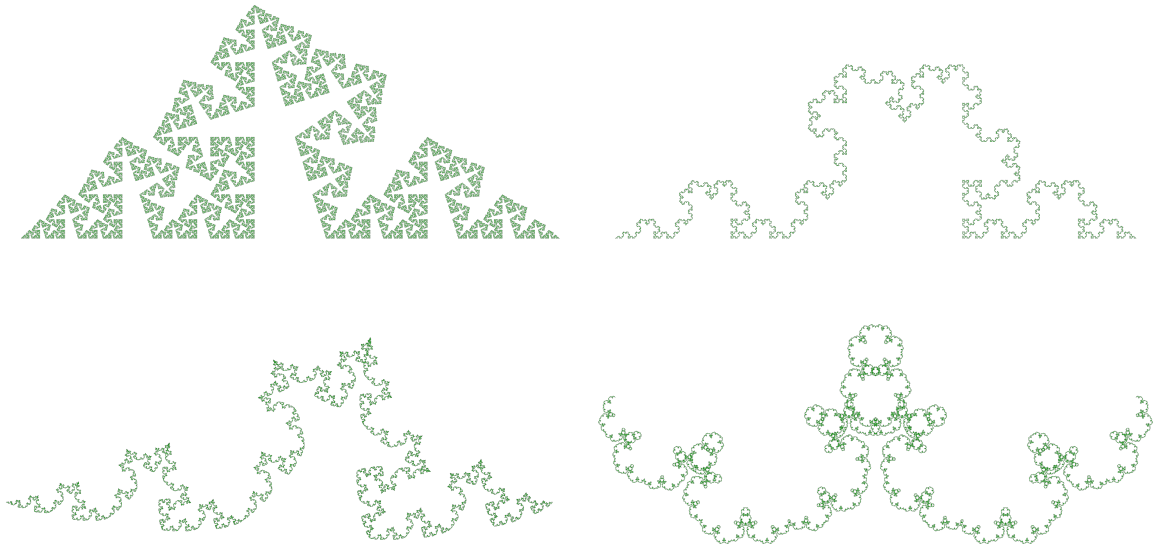
Figure 23: Quadratic type 2.



2.3.5 Variations

At this moment it will probably be already clear that the program is not bound to create rigorously defined fractals. The user can choose to create a lot of different variations with ease. The fact that the lines are drawn dynamically make this even more interesting. The pictures in Figure 24 show just a very few of the types of Koch curve variations that can be created. I will come back to Koch variations in later chapter.

Figure 24: Koch Curve variations



3 Simple Techniques

3.1 Simple Spirals

Spirals can be created by a simple pattern (Fig. 25). First a identity segment is drawn then, a single segment recurses further. We can call the segment that recurses further, *the recursive segment*. The type of the spiral depends on the placement of the recursive segment (green in this case).

Figure 25: Setup of a spiral.

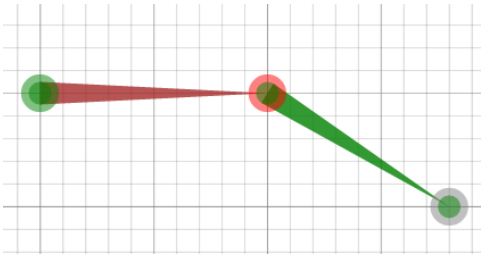
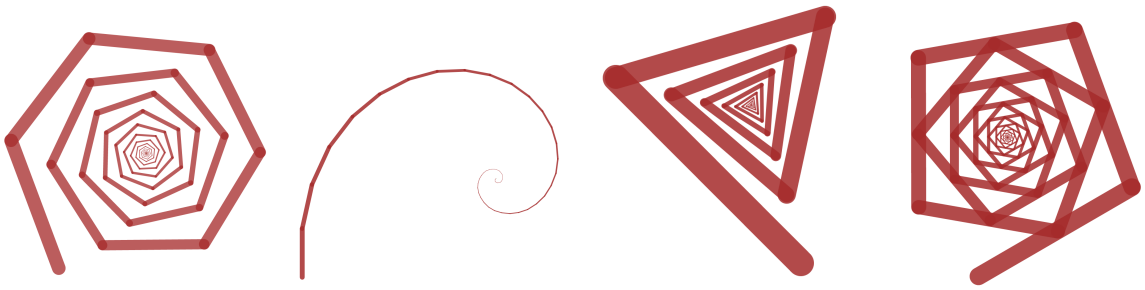


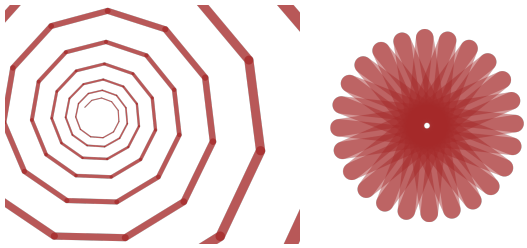
Figure 26: Simple spirals.



3.1.1 Types of simple spirals

The most straightforward way to classify spirals is by the length of their recursive segment. Therefore, there are 3 type of spirals. The ones which were discussed earlier are the most common type — *convergent spiral*. Convergent spiral are spirals in which the length of the recursive segment is smaller than the length of the identity segment. Because of the aforementioned condition, convergent spirals always converge.

Figure 27: An example of an divergent and a stationary spiral.

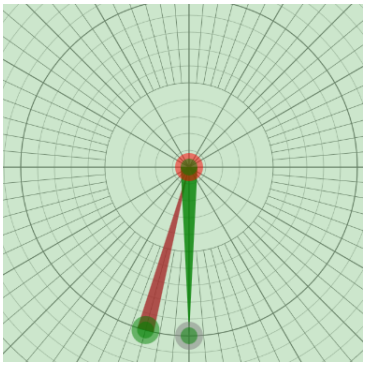


Besides this, there are of course two other types. There are divergent and stationary spirals. Usually divergent and stationary spirals are not as interesting as convergent ones, yet, they still can create

beautiful figures, as shown in Figure 27.

As a side note, the best method to create stationary spirals is by using the polar coordinates grid like in Figure 28. In this way the length of the segments remain the same and the user has to change only the angle, which is trivial.

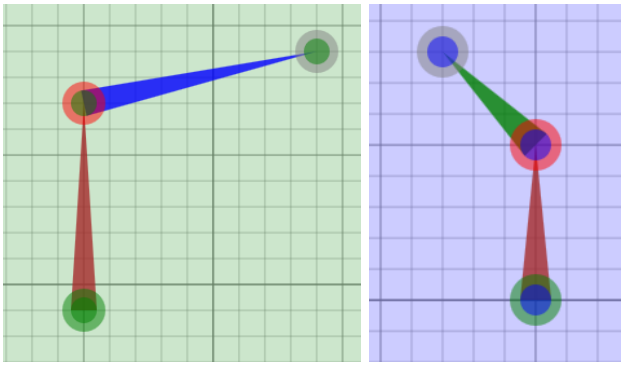
Figure 28: The setup of a stationary spiral.



3.1.2 More interesting spirals

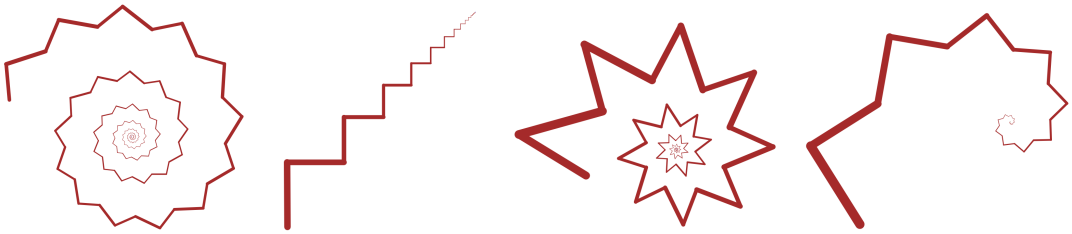
An interesting technique is to use two transformations in order to create a single spiral. The main one will go in the direction of the rotation of the spiral, as usual, but the second one will go in the opposed direction. In this way a zig-zag pattern can be observed.

Figure 29: A more interesting spiral setup.



As it can be observed, the green transformation has a quite long recursion segment. This will compensate for the fact that the blue transformation goes in the opposing direction. Figure 30 shows how such spirals can look. One of the figures doesn't even look like a spiral (more like a zig-zagging straight path) but keeping the convention, it will be named like one.

Figure 30: More interesting spirals.



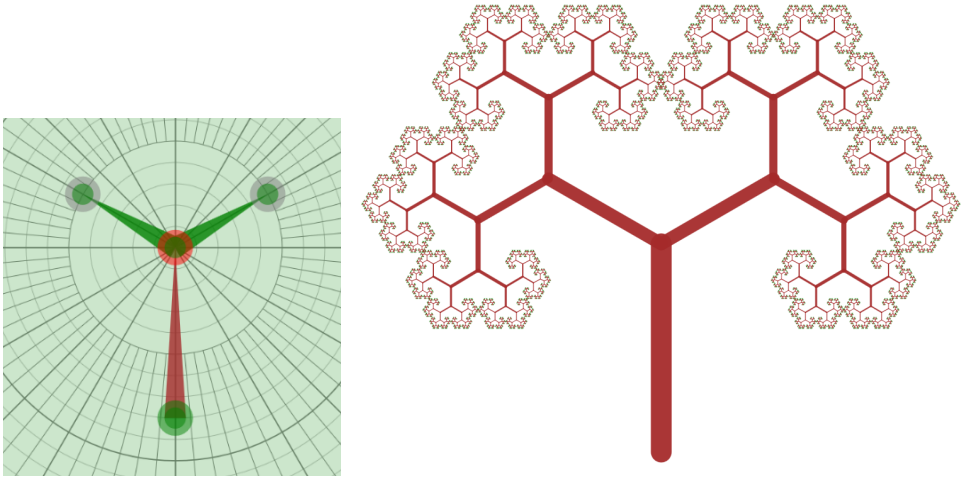
3.2 Simple Trees

By *Tree* I mean not only transformations that resemble trees but also any figure in which a tree-like structure can be distinguished. Just as spirals trees can be classified in convergent, divergent and stationary, yet I will only discuss convergent trees since the other two type do not have very interesting features.

3.2.1 Trees with two branches

Let's begin by creating a figure that resembles the Pythagoras Tree. For this particular tree I switched to Polar Coordinates grid so that I could put the branches at exactly 120°degrees.

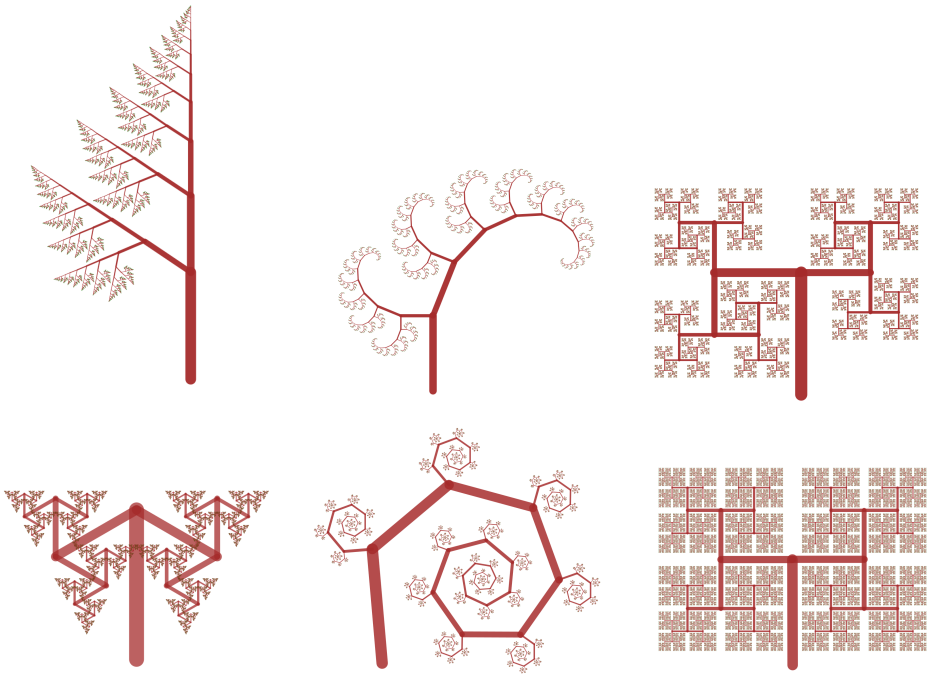
Figure 31: A geometric tree setup and its corresponding fractal.



Trees look like a natural progression from spirals. Spirals where bound to use just one recursion *branch* whereas trees can use two, or more, as we'll see further.

Figure 32 shows some possible types of trees.

Figure 32: Some trees.

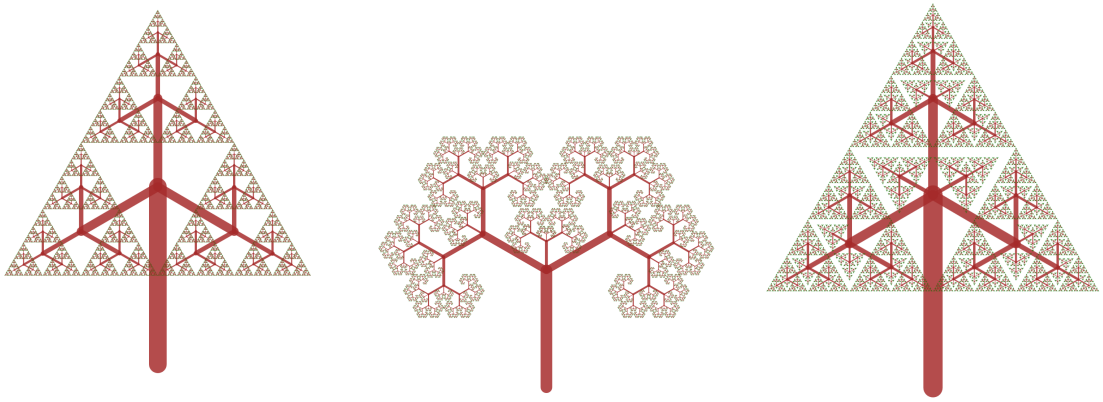


Some of these trees start to degenerate to spirals with an added node. I will come back to these types of spirals later.

3.2.2 Trees with multiple branches

Figure 33 shows some examples of trees with multiple branches. Usually having many branches results in getting a clustered fractal so choosing a good transformation is very important.

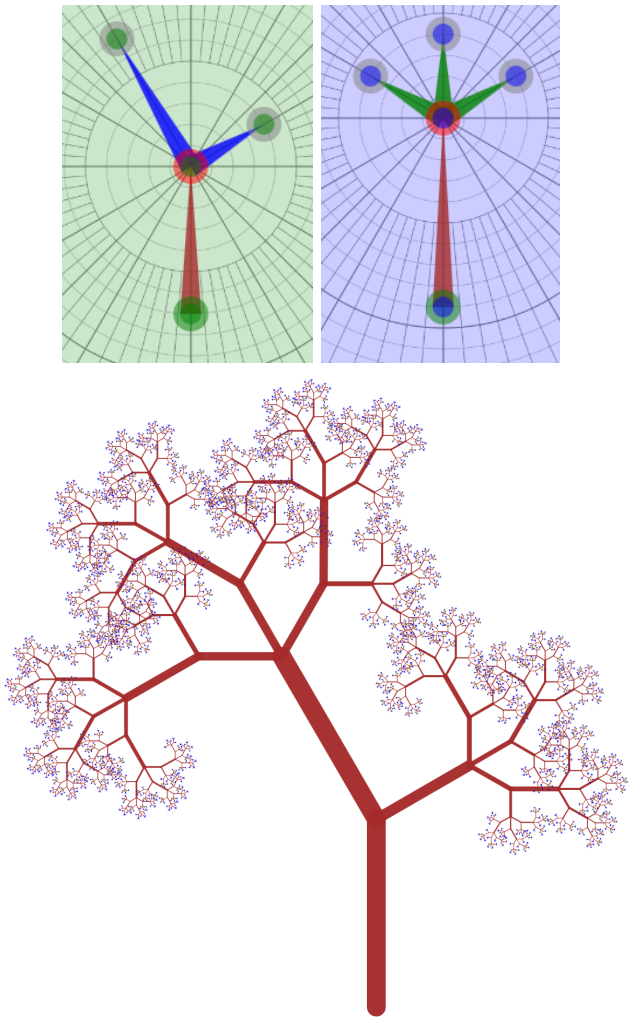
Figure 33: Trees with several branches.



3.2.3 Trees with multiple transformations

A much more interesting progression from simple trees is to use different transformations that call each other, just like in the beginning of the paper.

Figure 34: Setup of a tree with two transformations.



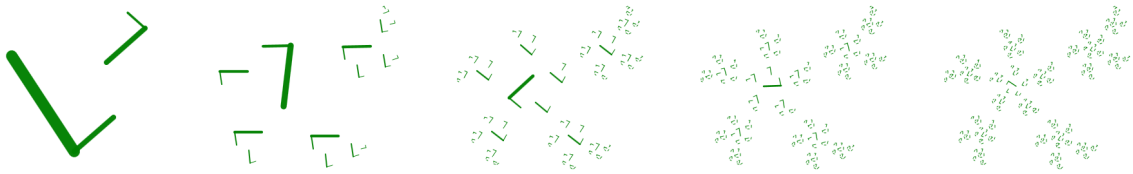
The reader should experiment and see whether he or she can find interesting types of trees using just these simple techniques.

3.3 Unconnected figures

Until now every transformation implied some kind of connection between segments. This is not necessary. We can easily create transformations without connections. Understanding the mathematical reasoning behind these type of transformations is very important.

Let's start with two adjacent segment, which are not connected to the start nor the end point. Figure 35 shows the setup. The progression shows how the lines evolve.

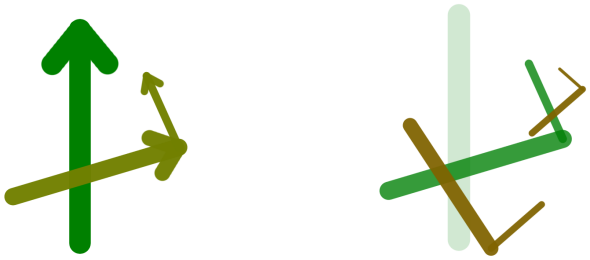
Figure 35: Two segments.



The reasoning behind these transformations may seem rather obscure so let's go through it step by step. We start with a segment with the direction showed by the arrow. The transformation is superimposed (Fig. 36).

Figure 36: The transformation step by step.

(a) (b)



Now, it should be clear how the mechanics of the transformation works. The arrows are very important, changing the direction of the arrow, usually, completely changes the visual aspect of the figure.

A major problem with unconnected transformations is that they can diverge to figures that are hard to observe since they are composed from very small parts. It is possible to create unconnected transformation with only one recursing segment but the behavior is trivial, so the minimum segments for constructing an unconnected transformation (and actually any type of transformation) is two.

When it comes to unconnected transformations the result is usually quite unexpected. It is hard to guess the final figure just by looking at the transformation. Figure 37 shows an example.

Sometimes, especially when dealing with unconnected transformations it is best to not iterate very deep. For example, in fig. 38 the iteration level is just 5. Also observe how unexpected the result is comparing to the setup. Some special cases of unconnected transformation will be discussed in further chapters.

Figure 37: A unconnected transformation with a triangle

(a) (b)

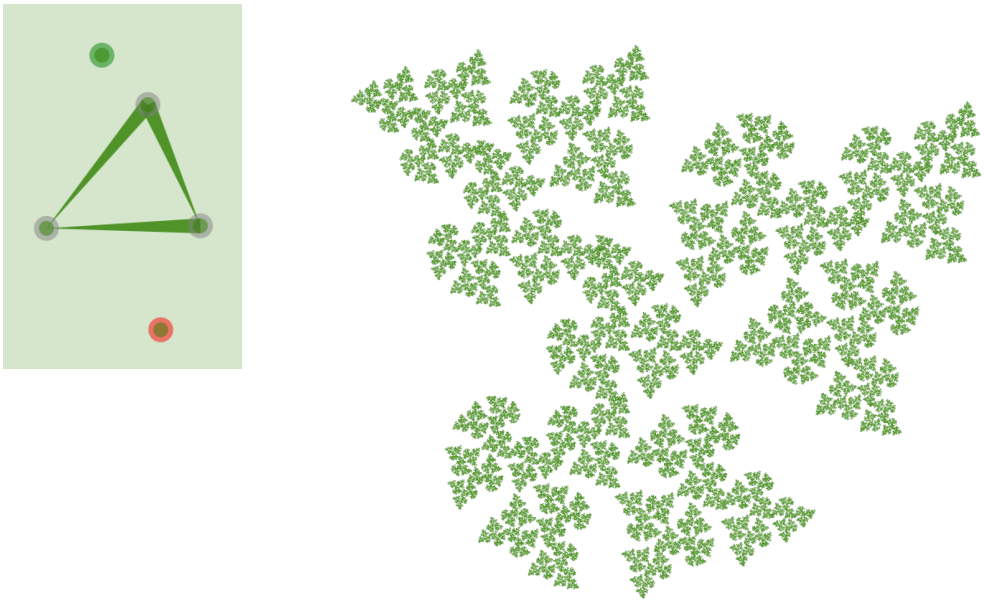
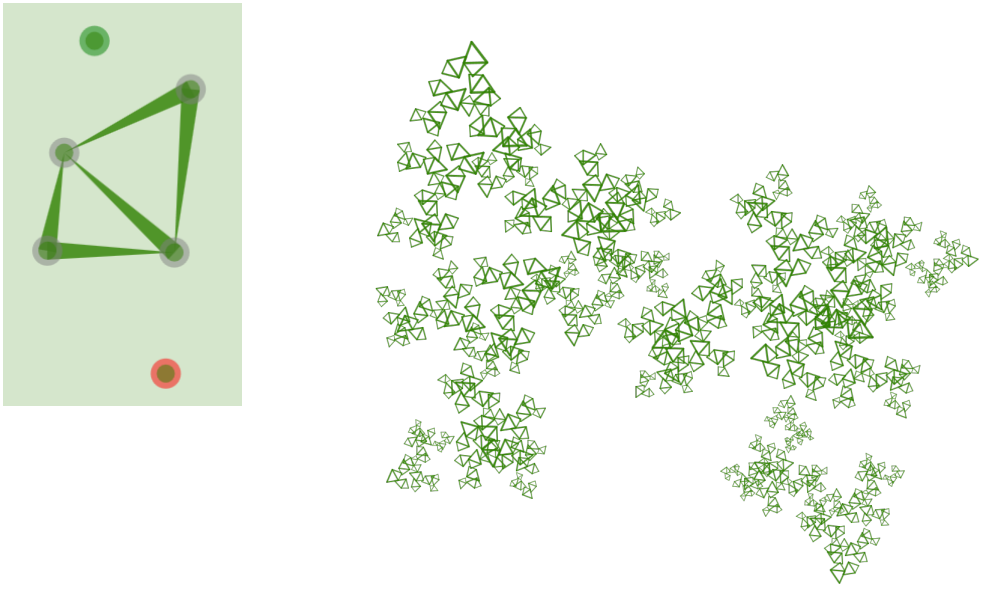


Figure 38: A unconnected transformation with a figure.

(a) (b)

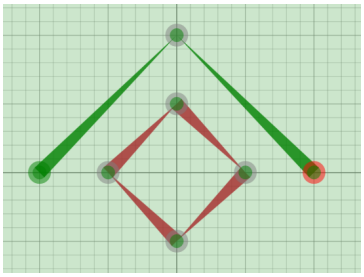


3.4 Singleton Pattern

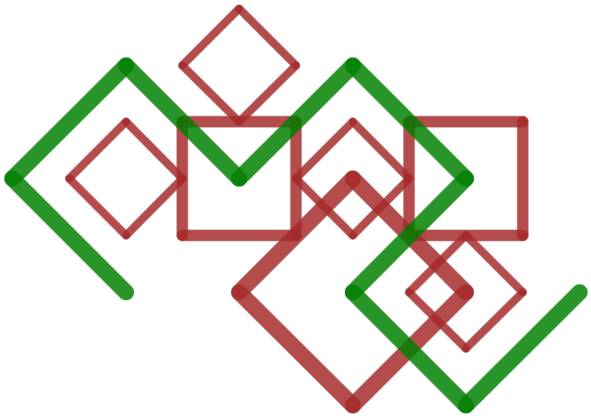
Suppose we have the following setup (fig. 39a). This is the dragon curve with a square inside. Looking at the fractal after some iterations (fig. 39b) it is clear that this is not a very aesthetically pleasing fractal. It would be much better if only the last squares were seen.

Figure 39: An example.

(a) The setup.

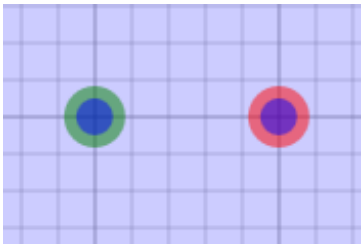


(b) After some iterations



The Singleton pattern solves this problem. First we need to create a new type of transformation (fig. 40). It may be strange that the transformation for a singleton pattern doesn't require any segments, but the power of this pattern will prove its value.

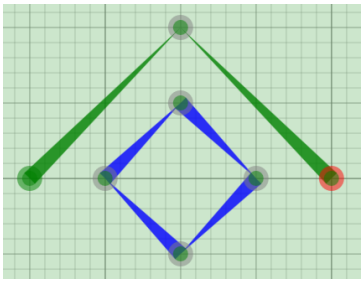
Figure 40: The singleton transformation.



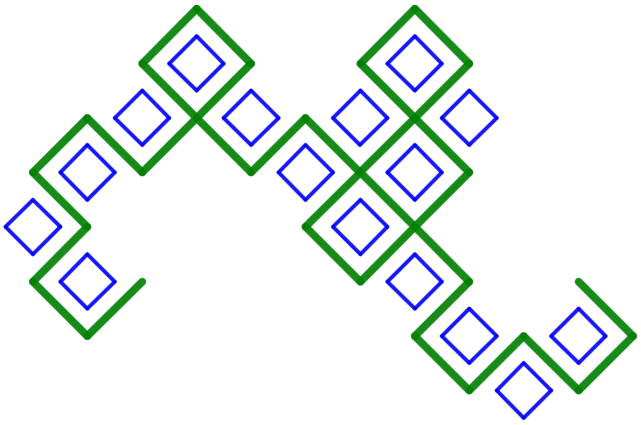
Let's try to change the dragon curve example on more time, this time using the blue layer (fig. 41a). In this case the final result looks like this (fig. 41b). The logic behind this is quite clear, on every iteration a new pair of squares forms but the last one iterates into the singleton transformation and disappears.

Figure 41: Using the singleton pattern.

(a) The setup.

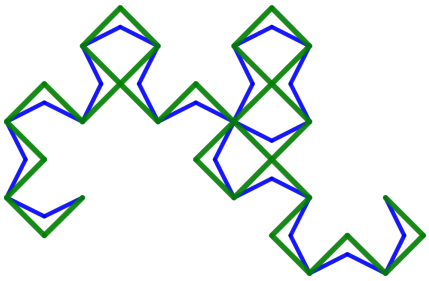


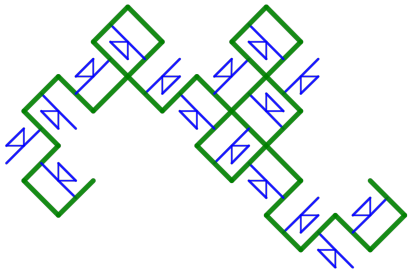
(b) After some iterations



Instead of just square we can use any type of figure (fig.42).

Figure 42: Some examples.



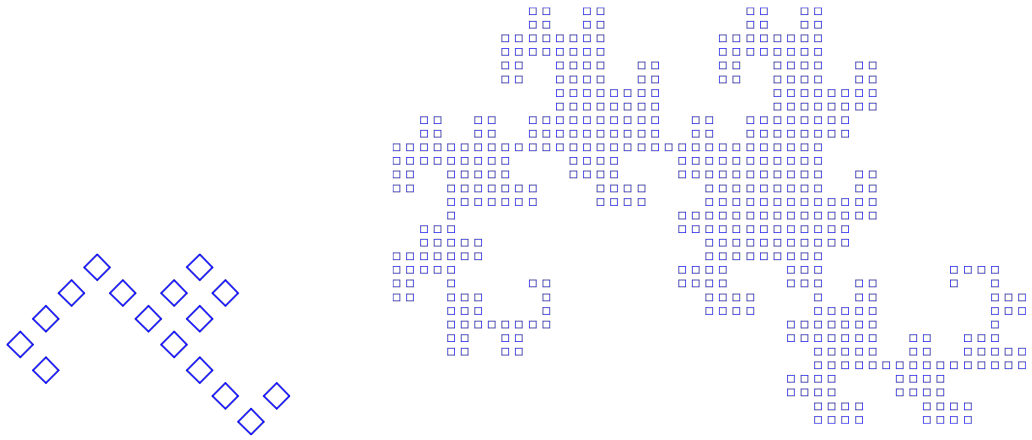


In the previous example only the dragon curve was used. This was due to the fact that it was easier to explain using it, nevertheless singleton pattern can be applied in a big variety of cases.

3.5 Decorator Pattern

The definition of the decorator pattern is more abstract than the definition of the previous patterns. The idea is to divide segments in two parts. One that is responsible for the recursion mechanics and the other one responsible for the visual setup.

Let's review the dragon curve example from the previous chapter but make some changes. We will use a singleton transformation and the setup will be the same (fig. 41a). This time, however, I will use a interesting property of the software. Inside the color-choosing window there is a gray square with an X inside. Pressing that square will make the segments of this transformation completely transparent. In this way, the green segments will not be seen, only the blue squares.



(a) (b)

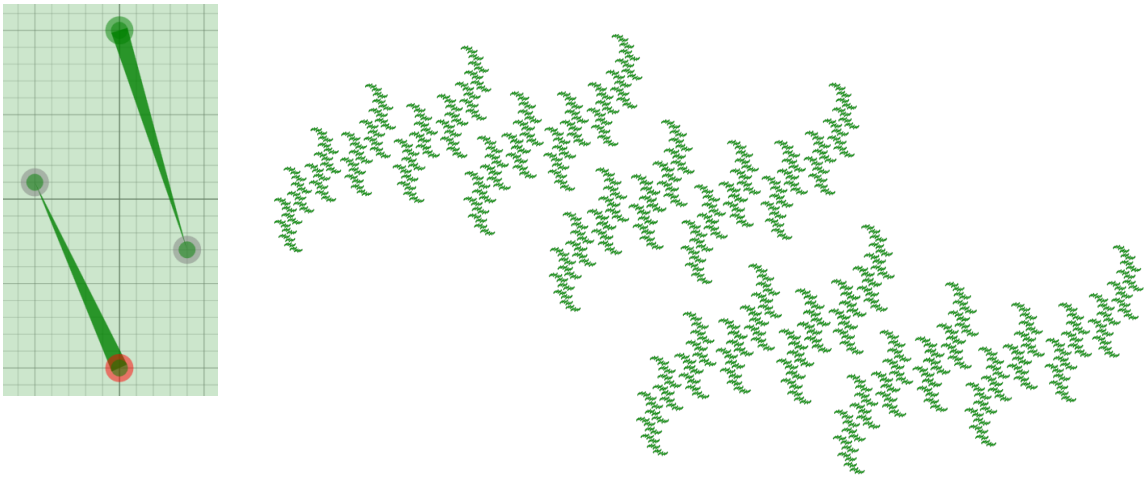
3.6 Case Studies

3.6.1 Two Rays

The title might be a misleading since we are not actually dealing with mathematical rays. It's deals with the visual aspect of the transformation we are about to discuss. Figure 44 shows the setup and the figure that it creates.

Figure 44: Two Rays

(a) The setup (b) The fractal.



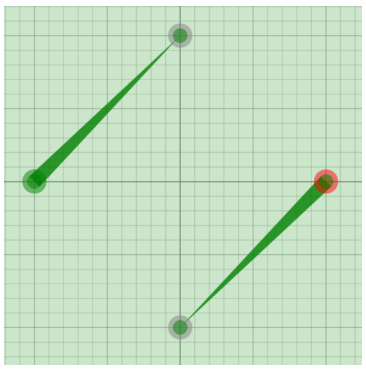
It's probably clear why the name "Rays" was chosen.

Since the segments are not connected to each other, the final fractal is unconnected. This doesn't mean that the segments cannot overlap, this is a very common feature of unconnected fractals.

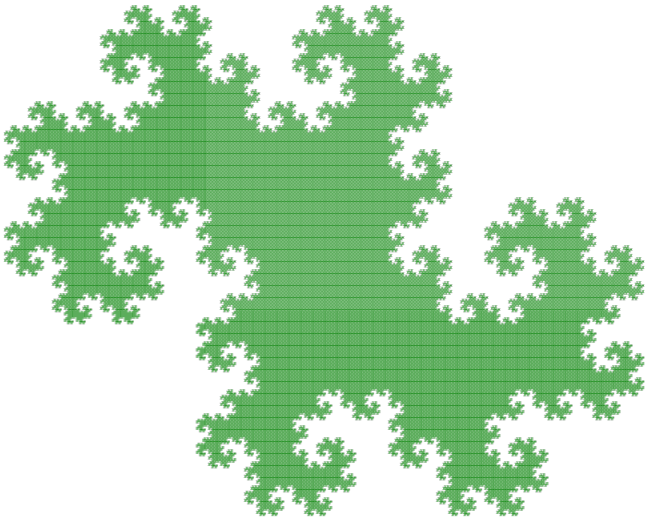
If we change the setup we can get an interesting fractal (fig.45). The fractal is similar to the Dragon Curve. Even though the segments are unconnected, this fractals shows a interesting type of connection.

Figure 45: The Ray Dragon.

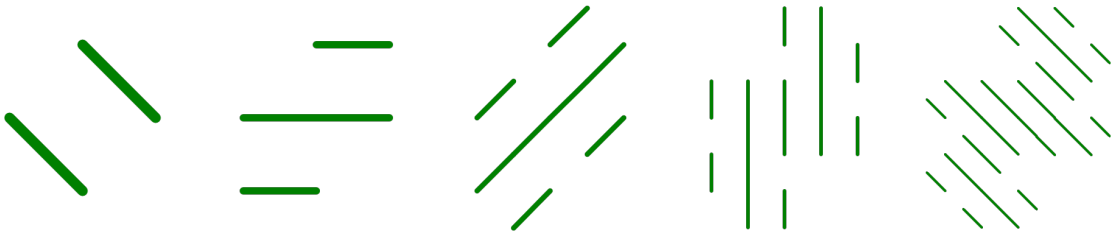
(a) The setup



(b) The fractal.



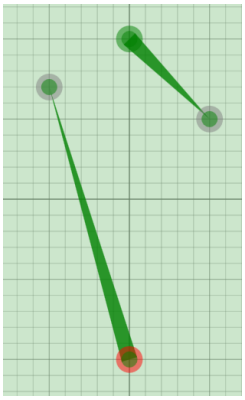
(c) First iterations.



Moving the segments around we can get other types of strange fractals. We can try to invert the arrows, that is, change their direction. There are 4 combination in total, below are all the combination applied to the same set of points.

Figure 46: Combination 1.

(a) The setup



(b) The fractal.

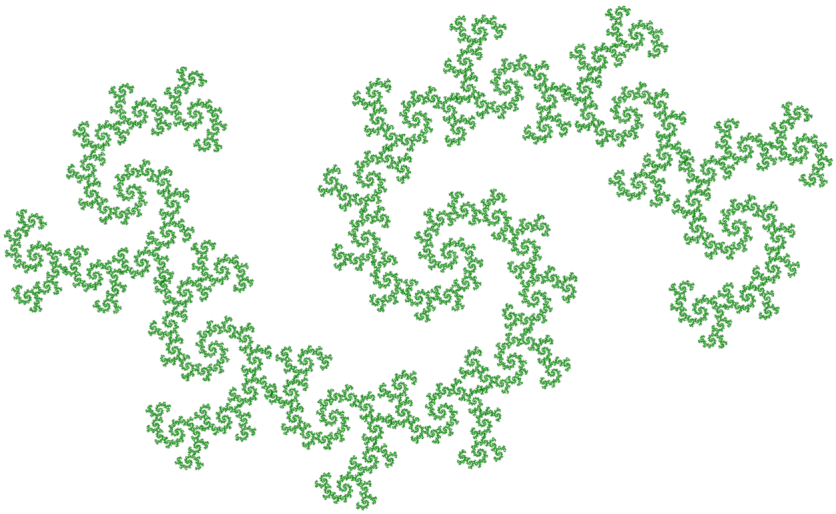
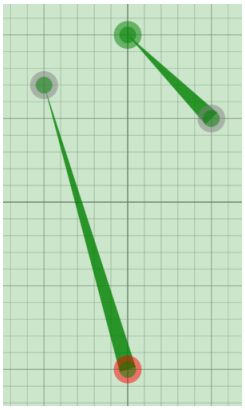


Figure 47: Combination 2.

(a) The setup



(b) The fractal.

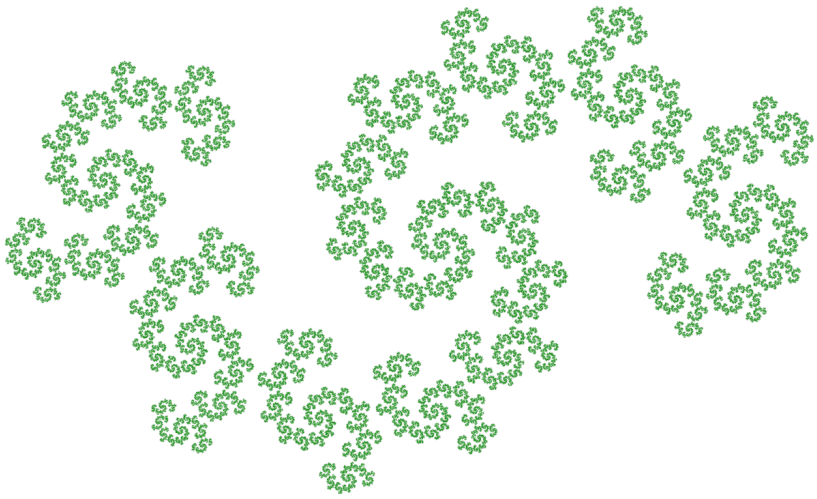
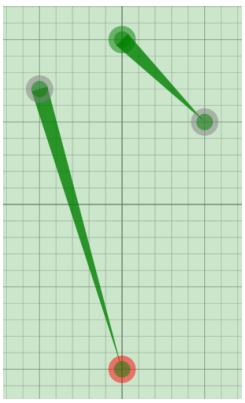


Figure 48: Combination 3.

(a) The setup



(b) The fractal.

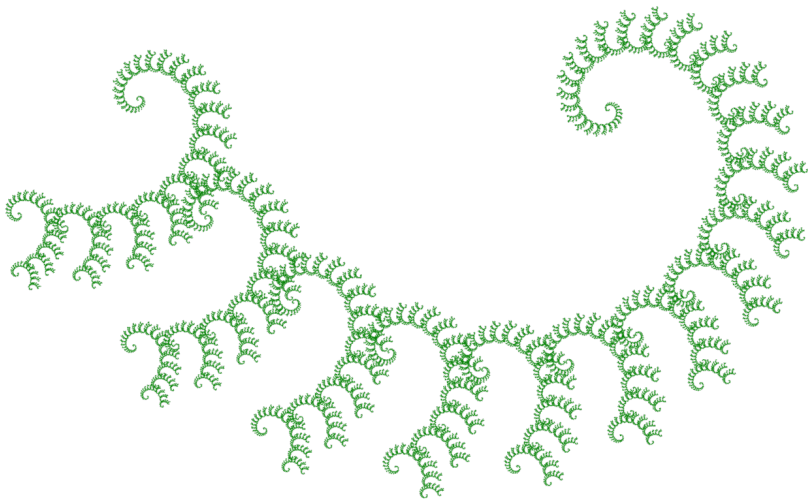
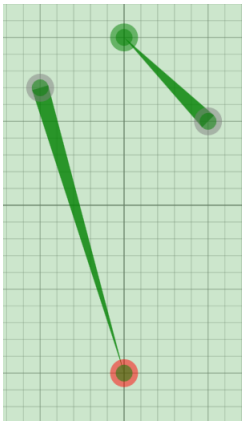
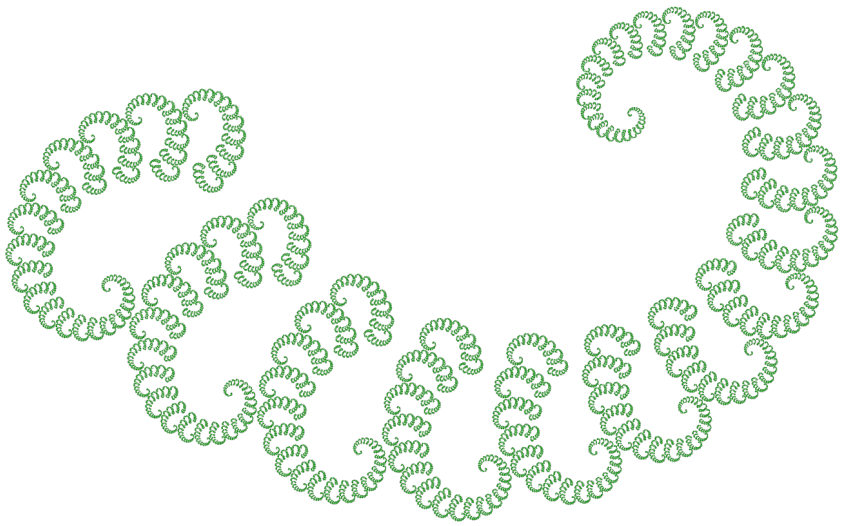


Figure 49: Combination 4.

(a) The setup



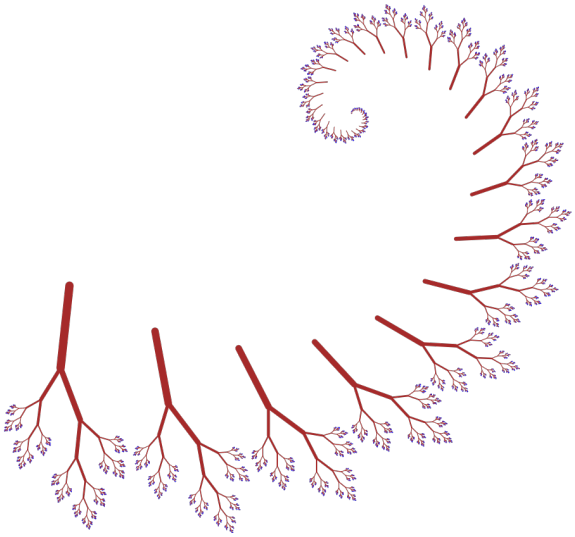
(b) The fractal.



As we can observe the long segment changes the general aspect of the fractal whilst the smaller ones changes the details of the fractal.

As a final note, the small segment can be substituted with anything else, for example a tree.

Figure 50: Tree in spiral.



4 Advanced Techniques

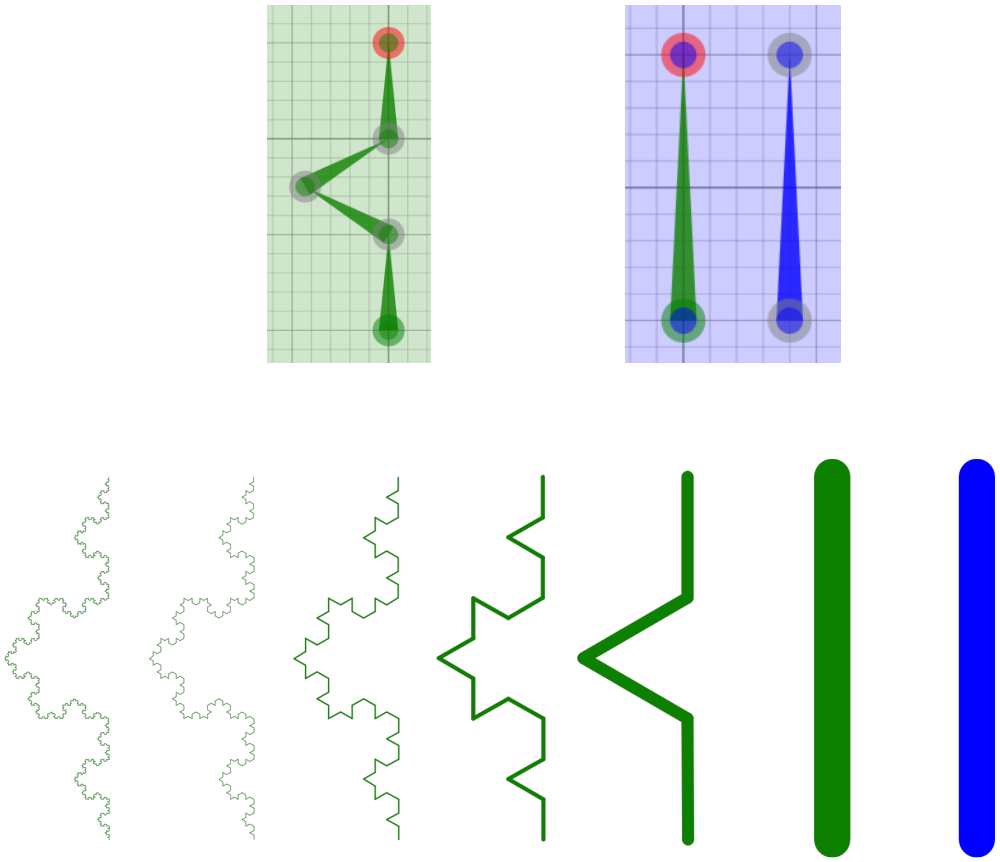
4.1 Step-by-step pattern

The Step-by-step pattern was used previously in the book. It is very useful to use it for observation purposes, that is, to observe how a fractal evolves over time.

Suppose that Layer 1 has a simple transformation, for example the Koch Snowflake. Then suppose that Layer 2 has the following transformation (fig.51).

Figure 51: Step-by-step pattern.

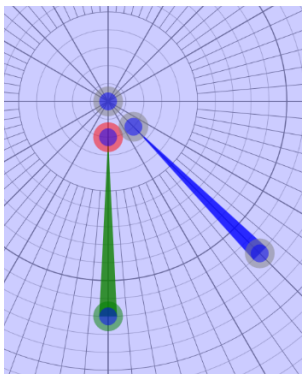
(a) Koch Snowflake. (b) Step-by-step pattern.



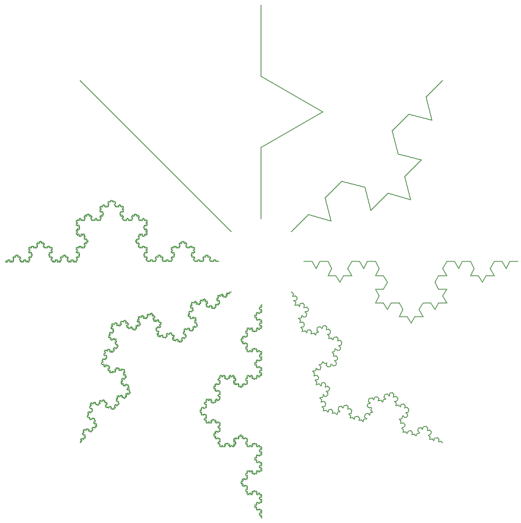
Layer 2 works by creating a desired iteration and then calling himself a little further. In this way the development of the fractal becomes very clear. Of course it is not obligatory that the segments in the Step-by-step transformation should be parallel to each other. We can choose a circular configuration. The blue color can be removed by pressing the X in color-choosing dialog.

Figure 52: Step-by-step pattern.

(a) The setup.



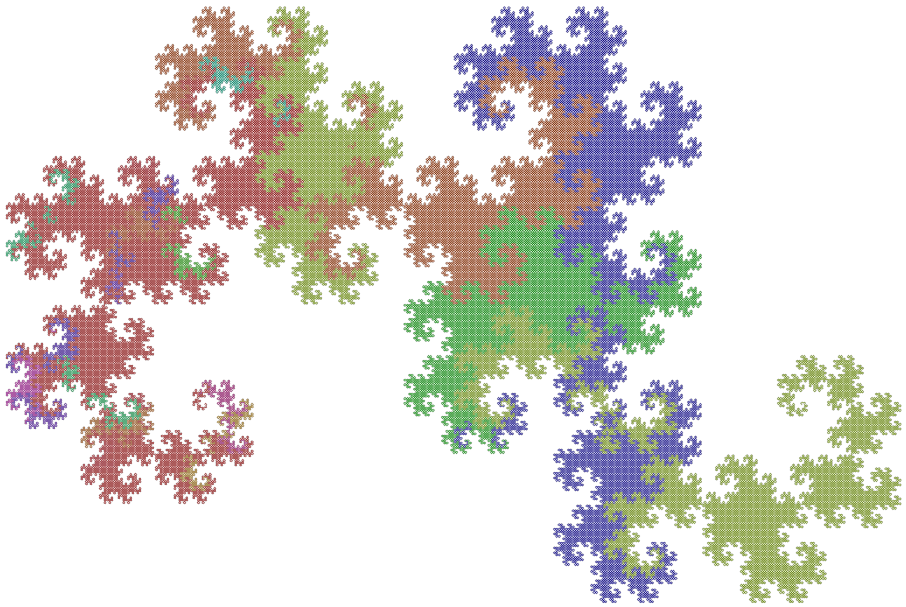
(b) The fractal.



4.2 Color Mixing

This is more of a trick than a relevant mathematical concept. While a complex figure is being drawn, the user can choose a different color for the segments. Figure 53 shows this technique applied to the dragon curve.

Figure 53: Colored Dragon



5 Randomness

Until now, every transformation had a deterministic figure. Usually, for fractals, this is a important behavior but sometimes some randomness would be useful. For example suppose we want to simulate some trees. Every tree created should have a different type of transformation. Even if we use different starting segments, the trees will have similar behavior. What to do then in order to create a forest of trees. Randomness is the answer.

In his book "A new Kind of Science", Stephen Wolfram discusses about the fact that randomness can arise from within a simple system without interaction from other systems (extern factors). For this he uses different types of finite automata.

In finite automata, the further development of a cell can be influenced by it's surroundings. His finite automata are placed in discrete space and can easy have access to their surroundings. I found it harder to reproduce this behavior for systems which are placed in continuous space.

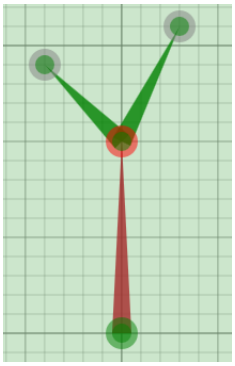
For example a segment could recurse further or stop depending on whether it crosses another segment. Or the type of the segment could be influenced by the number of segments in a certain proximity.

The implementation of these functionalities into the software would take away from its minimalism. That is the reason I chose (for now) some other methods for creating randomness. Unfortunately these methods rely on the outer world.

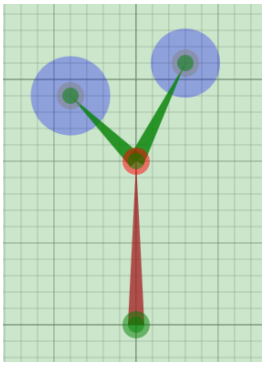
5.1 Node position variation

Suppose we have setup of a simple tree (fig. 54a). It will create a very basic fractal. In order to add some randomness the following should be done by the user. Press *CTRL* key while dragging the node. A blue disc appears around the node (fig. 54b). This blue disc represents the area in which the node can be placed. To delete the randomness disc press the *CTRL* key and the middle mouse button.

One can observe that the creating, deletion buttons retain their meaning, only the *CTRL* key is added.



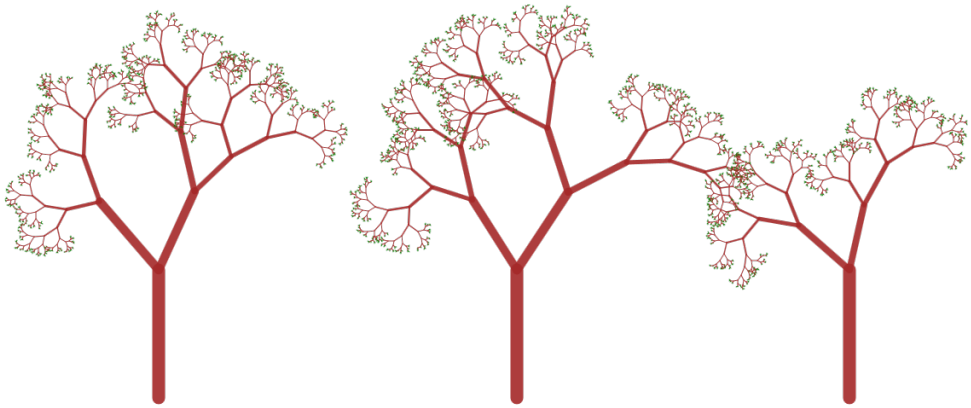
(a) A simple tree.



(b) A tree with added randomness.

Figure 55 shows some possible trees. All the trees were created using the same starting segment.

Figure 55: Some random trees.



Something more about the randomness functionality. The start point and the end point cannot have a random position. No node in the base layer can have a random position.

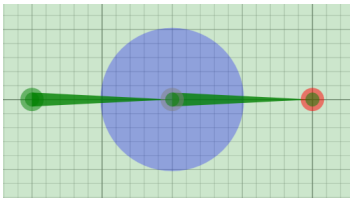
Let's look at some useful application.

5.1.1 Random Paths

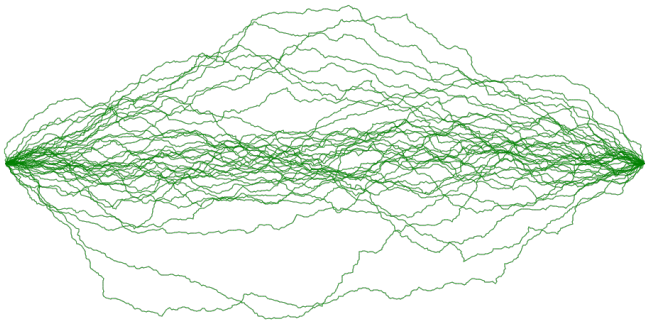
The setup of a very interesting example can be observed in figure 56a. Figure 56b shows some of the figures, superimposed.

Figure 56: Random path.

(a) The setup.



(b) Some paths.

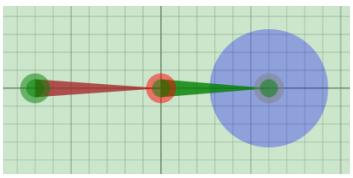


5.1.2 Random Spirals

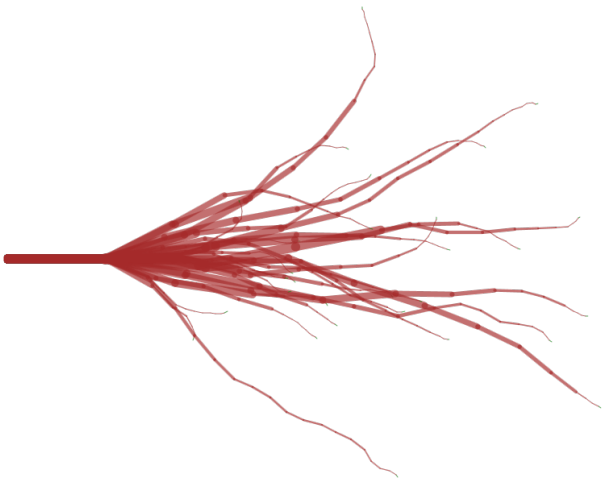
Due to the minimalistic nature of spirals, randomness is a very good fit for them. I'll start with spirals which do not rotate.

Figure 57: Straight spirals.

(a) The setup.



(b) Some paths.



Randomness can be applied to “real” spirals but that doesn’t always result in a beautiful picture.

Figure 58: Some spirals.

6 Used Technologies

I started creating this piece of software with the idea that it would be easily used online. That is the reason I used JavaScript, HTML and CSS as the main technologies. Below I will try to explain most of the details and the problem that arised during the development phase.

6.1 General HTML layout

The main HTML page has several layers stacked one over another. First of all there are 2 *canvas* elements. One is used for the grid and the other one is used for drawing the recursive image. After this there are several *SVG* elements (I call them *layers* in the code). These layers contain all the draggable elements (the circles, the arrows and the invisible background). Also, for the menu, I used *jQuery UI* framework. That is a simple to use HTML/CSS widget. This is basically all the setup.

6.2 The *Canvas* element

Wikipedia [3] give a very good and brief description of the canvas.
The canvas element is part of HTML5 and allows for dynamic, scriptable rendering of 2D shapes and bitmap images. It is a low level, procedural model that updates a bitmap and does not have a built-in scene graph.

The canvas element has several simple to use functions like `lineTo()`, `moveTo()`, `fill()`, `stroke()`, etc. Canvas is very good and fast at drawing lots of lines and this was exactly what I needed.

While beginning to work on the project I first implemented the recursive drawing using a SVG element (which I will talk about shortly) but it was to computationally expensive and slow that's why I used canvas.

6.3 The *SVG* element

The SVG element (Scalable Vector Graphics) is developed since 1999. It uses an XML syntax (just like HTML) to represent vector images. For example in order to represent an circle of radius $r = 50$ with the center in coordinates $x = 100$ and $y = 200$ one should simply write :

```
<?xml version="1.0"?>
<svg viewBox="0 0 120 120" version="1.1"
     xmlns="http://www.w3.org/2000/svg">

     <circle cx="100" cy="100" r="50"/>
</svg>
```

Actually all the text before `<circle />` is just telling what version of SVG standard we are using. Omitting most of those lines would usually not be a problem.

As it can be deduced SVG is great at, well, displaying vector graphics. Because it is written in XML it has a clear hierarchy of the objects inside it. Besides this it can trigger events that can be used by JavaScript code. This is a reason why I used SVG for drawing the draggable parts. For example in order to test whether the user has clicked inside the arrow (which is basically a triangle) I would had to use some geometric calculations. Using SVG I just used the `click()` event. The same is for the circles.

It should be noted that SVG is used not just in web browsers. There are some vector graphics editors which use SVG as the primary standard (Inkscape).

6.4 Snap SVG

Working with pure JavaScript functions for editing the SVG can be tedious and error prone. That is the reason I used a small framework, Snap.svg. Snap has a lot of functionality, at least more than I

needed. It can perform animation, matrix transformation, filtering, etc. But I used just a little part of them. The most useful function was `attr()`. That is a versatile function that can get or set attributes of SVG elements. For example instead of writing:

```
cirle.setAttribute('cx', 5);
cirle.setAttribute('cy', 89);
cirle.setAttribute('r', 34);
```

I could just write

```
circle.attr({
    'cx', 5,
    'cy', 89,
    'r', 34
});
```

which is easier to write and read.

Snap.svg has similar function for creating and deleting elements. These functions where widely used inside the code of the software.

6.5 jQuery UI

One of the most used JavaScript framework is jQuery. A very good explanation of what jQuery UI is can be found on their site [4].
jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library.

In order to use jQuery UI one should simply add an `div` or `input` element inside the HTML page then invoke some jQuery UI methods in order to apply the needed CSS transformation to that div. For example The code below transforms the div with the id *testing* into a button.

```
$('#testing').button();
```

Of course besides this, jQuery has similar methods for responding to events.

6.6 Spectrum.js

I needed some way to let the user to choose colors for different layers. Writing such a thing from scratch would not be efficient that's why I used a user-made widget for jQuery UI. The name of that widget is *Spectrum.js*. It provides some simple way of configuration and, generally, didn't require to much insight to use.

6.7 CoffeeScript

CoffeeScript is a little language that compiles into JavaScript. That's what their site claims. Their motto is "It's just JavaScript". The language does not implement some different concepts or any other things that would be strange for a JS developer. It just writes more beautiful and more readable code than JS. For example instead of writing.

```
test = function() {
    // something
}
```

in JavaScript, one could write

```
test = () ->
    # something
```

in CoffeeScript.

Of course this is a simple example but the lack of the `function` keyword was a big motivation for me to switch to CoffeeScript.

I will note that there are a lot of languages that compile to JavaScript. Another one worth to mention is TypeScript. TypeScript was developed by a Microsoft employee and provides classes and static typing.

7 Implementation details

Before starting the work on this project I did a somehow similar piece of software. That project gave me the idea to implement develop the project that I am currently talking about that I now write about.

I first wanted to draw the recursive line in a SVG element, I also wanted to draw a grid for every layer. The way the layers are structured in the final version was not thought from the start, it was a process of trial and error.

The main piece of code was written in a handful of days. It was written in pure JavaScript with very poor architecture. It could draw lines but it was clear that the bugs and the general code repartition would not allow space for further improvements.

After struggling some time I decided to use a different approach. I decided to rewrite all of my code in CoffeeScript. I never used it or anything alike till then. I had the wrong opinion that CoffeeScript would be to high-level and would hard to debug after transforming it to JavaScript. This was not the case.

CoffeeScript is a very thing layer over JavaScript. Most lines in CoffeeScript can be easily converted to JavaScript, without to much hassle. I talked about this before.

After a day or two I, finally, managed to transform all my code to CoffeeJS. The didn't mean that my code got better. It still lacked modularity and was generally speaking - bad code.

Using the CoffeeJS `class` construct I managed to implement a better structured OOP architecture.

7.1 The Vector class

During all the geometry transformations, lots of times, I had to transform cartesian coordinates to polar coordinates and vice-versa. I could create two functions that would transform to and from these coordinates systems. But it was clear that using such an approach would be tedious. This problem arose at my first project (I talked about it earlier).

I tried to resolve this by implementing a class that could store and return at the same time both types of coordinates. The class had 4 properties :
`orthX, orthY, polarRad, polarAng`

These were closure [5] variables so the user could not access them. I provided several methods for interaction, like:
`setX(), getAng(), getY(), setRad()`, etc.

Inside these functions, beside setting or returning the requested variable I would recalculate the coordinates. For example if a user used `setAng()`, I would set the angle to the given value and, after that, would recalculate the cartesian coordinates. In this way, every calculation happened without the knowledge of the user.

The problem with this was that it was still to cumbersome. That was when I found out about a very interesting and obscure feature of JavaScript [6].

7.1.1 Object.defineProperty

Basically this function allows to define all kinds of properties for variables. For example:

```
Object.defineProperty(this, 'y', {
    get : function() {
        return orthY;
    },
    set : function(val) {
        orthY = val;
        this.recalcPolar();
    }
});
```

This piece of code defines a property `y` that would return the private value `orthY` when used, for example:

```
var otherVariable = object.y;
```

will return the private variable `orthY` from inside the closure of `object`.

This is doesn't look very useful but the `get` function defines a more interesting behavior. When someone assigns a value to property `y` it not only changes the variable `orthY`, it also recalculates the polar coordinates. For example, the user can use

```
object.y = 5;
```

instead of

```
object.setY(5);
```

An even more interesting behavior can be observed below

```
object.y += 3;
```

Here we increase the `y` value the recalculate the points. Writing this using simple functions would look like this:

```
object.setY( object.getY() + 3 );
```

Now the real power of `Object.defineProperty()` can be observed.

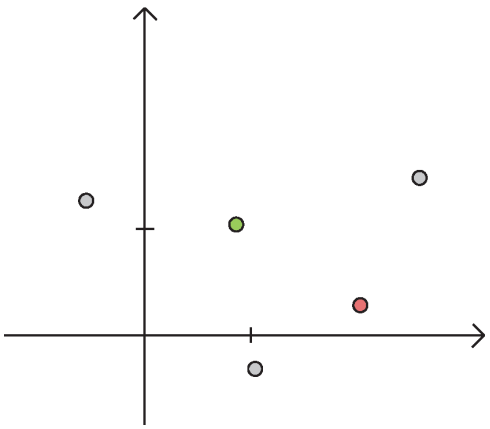
7.2 Basic Geometry and point transformation

After the user moves the points on the SVG layers I needed a way to transform that location of the points to a different line. One of the biggest fears that stopped me from starting to work on this project was that I was not sure how to implement the line transformations. After implementing the Vector class I used a very simple technique. This technique will be explained below.

7.2.1 Simple Transformation

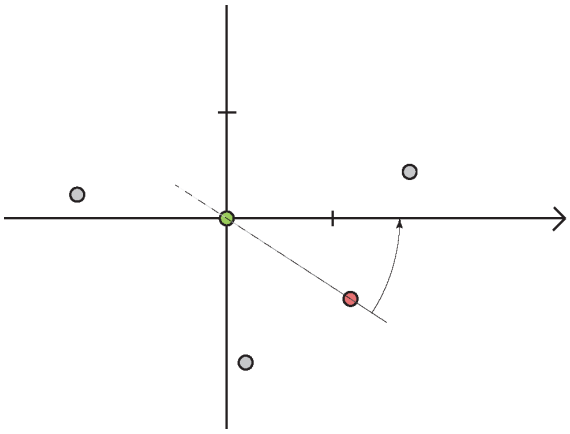
Suppose we have this setup:

Figure 59: Initial state of the points.



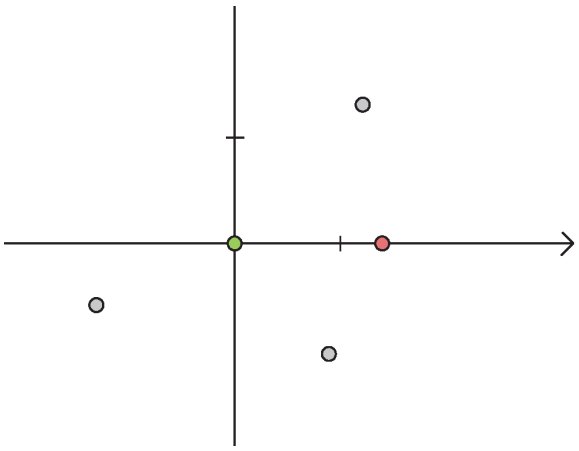
We have several points. The green one represents the start of the segment and the red one the end of the segment. I start by translating the points so that the start point coincides with the center of the coordinate system. At the same time I find the angle between the segment that links the start and end point and the X-axis.

Figure 60: After translation. The angle.



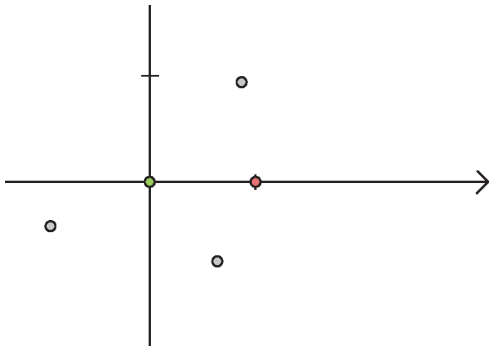
Now I rotate all the points with that angle so that the end point is on the X-axis. All the other point follow by.

Figure 61: After translation. The angle.



Finally, I scale all the points such that the end point is on (1, 0) on cartesian coordinates.

Figure 62: After scaling.



In such a way the vector that connect the start point with the end point is the X-axis unit vector.

All these transformation occur inside the function `toUnitVectors()`. This function is called from the method `getData()` of the `LineLayer` class. Basically the user can call the method on a particular SVG layer and the layer will calculate the data and will return it in a compact form. Of course the function `getData()` also returns data about the lines between the points but this is trivial task.

Here is the code

```
toUnitVectors: function(points){
    var ret = [];

    // There should exist at least two points
    if (!(points[1] && points[0])) return;

    // Vector from start point to end point
    var dir = new Vector(points[1].x - points[0].x, points[1].y - points[0].y);

    for (var i=0; i<points.length; ++i) {
        // vector from start point to point[i]
        var v = new Vector(points[i].x, points[i].y);

        // Translate the point
        v.x -= points[0].x;
        v.y -= points[0].y;

        // Change angle
        v.ang -= dir.ang;

        // Scale distance
        v.rad /= dir.rad;

        ret.push(v);
    }
    return ret;
}
```

7.2.2 A faster algorithm

One might argue that it would be a better idea to implement something that would not require so much computation and storage. For example to use simple matrix transformations and to not use a complex

data type like Vector but stick to a simple one, an array of two elements for example. This behavior will be probably implemented, but there are other, more important problems that need to be solved.

7.3 Recursive drawing

Another problem that I had to tackle is choosing an optimal recursive drawing algorithm. The core is clear, I iterate through the segment calling the function recursively till I get to deepness 0 when I draw the line.

If I would use this simple approach the program might resist for simple drawings but it would completely freeze if there a lot of functions. And I think when the number of calls gets to 300000, that can be called a lot of functions. Besides this, even if the program might succeed in drawing the segments the user will be unable to move anything while the program is drawing. That was a major drawback.

7.3.1 Asynchronous JavaScript

JavaScript is known as a language than can be easily used for asynchronous function callback. You give a function, give a number of milliseconds, and after that time passes the functions is executed. There are to basic functions that provide this functionality: `setTimeout(func, millis)` and `setInterval(func, millis)`. The first one executes the function just one time, the second executes the function indefinitely. The execution can be stopped by `clearTimeout(timeout)` and, respectively `clearInterval(interval)`.

There are a lot of myths about these two functions. First of all people tend to think that JavaScript uses multiple threads to handle asynchronous functions. Unfortunately it does not. As I side word there start to appear some functionality in JS that would permit multi-threading. This, however, is beyond the scope of this paper. The reader might want to search for *Javascript Web Workers*.

All the asynchronous functions are executed on the same thread. The exact implementation varies from browser to browser but I will try to give a simplified explanation. First of all there is a queue that catches all the asynchronous execution, be it the main program, a function called with `setTimeout()` or even `XMLHttpRequest`. I will talk only about timeouts and intervals, and the differences between the two.

If a timeout or an interval fires and there is nothing to be executed, the asynchronous function is just executed. If there is another process running the asynchronous function is queued. The interesting thing is that if an interval fires another time while the last occurrence was not executed, the new occurrence will be dropped.

There are two approaches for handling iterative asynchronous function calling. First there is setting up an interval. The second is to set up a timeout and reset that timeout inside the function that he fires. There is a slight difference between two approaches. More about this can be found in the book *Secrets of the JavaScript Ninja* [2], chapter 8 *Taming threads and timers*.

7.3.2 Asynchronous Recursive Drawing

With the knowledge from the previous paragraph the user might have observed a very good optimization. Instead of just recursively calling the function that deals with drawing the program might set a timeout. In that way the program will not freeze. It might take him a little more time but it will not freeze.

The problem with this approach was that it is to expensive. There were cases when I got more than 1 GB used by the browser.

Not only were a lot of variables declaration in every function call, also, there were a lot of functions firing at the same time. Imagine how was the browser feeling with more than 10 thousand timeouts firing at the same time. Expecting the browser to take care of the queue by himself is not what a gentleman would do.

7.3.3 An Asynchronous Iterative approach

For some it might have been clear from the beginning that the best approach would be to create a queue, pass all the data for the function in that queue and then take one element after another until the queue gets empty. I didn't adapt this technique from start because I was lazy.

First of all I changed a little bit the values that were passed to the function. On the first variant the function would get the starting points of the segment in form of Vector points, the type of the segment and the level of the recursion. The most obvious optimization was to not pass a Vector object but just two numbers for every point.

Every call to the function would populate the queue. From a different function I would control the execution of the queue. I used a timeout to execute the function. I soon realized that the timeout could execute the function more than one time. I introduced a variable that controlled how many time that function would get executed every timeout. Now it is set at around 300 times. This is a number that lets the user interact with the browser while the function is still drawing.

7.3.4 The last optimization

I observed another thing that could be optimized. Every function call would create some new variables. Letting the JavaScript garbage collector control when the variables were disposed was not the best idea. Also the fact of allocating a new variable was itself time consuming.

I decided to use function from outside the function. In this way every function would use the same functions, just like static functions in other languages.

Inside the function there are cases when I allocated a new Vector object. An optimization was to use the same object and only change his data.

In this way the drawing got more fast.

References

[1] *A new Kind of Science* Stephen Wolfram. 2002

[2] *Secrets of the JavaScript Ninja*. John Resing, Bear Bibeault. Manning Publications. 2013

[3] Wikipedia. Canvas Element

[4] jQuery UI

[5] MDN, JavaScript Closures

[6] MDN, Object.defineProperty()

[7] Wikipedia, Barnsley Fern