An investigation into the theory and production of fractal images described as “Fractal Flames”

- Tina Nischan -

Innovations Project
BA (Hons) Computer Visualisation and Animation
Year 3, 2005 / 2006
National Centre for Computer Animation
Bournemouth University

Content

Abstract

The fractal images known as “Fractal Flames” make use of the aesthetics of a specific algorithm that is member of the Iterated Function System (IFS) class, but differing from the classical definition in some essential aspects. The set of functions is non-linear, the appearance is based on a log-density display instead of a linear map and colour is allocated by structure.
The “Fractal Flames” algorithm results in a maximised preservation of information in order to optimise the aesthetics of the result.
The subsequently described project shows investigations in the background of Iterated Function Systems and the chaos game, the specifics of the “Fractal Flame” algorithm and the attempt to implement the theory in a program using Open GL and C++.

1 – Introduction

Long before the term *fractal* was introduced, people where observing forms and figures in nature, trying to break down descriptively what they saw. Artists like Albrecht Dürer or philosophers like Leibniz (Wikipedia) described self-similarity and recursion.

But it was not earlier than 1975 that Mandelbrot coined the word *fractal* (Mandelbrot, 1975, 1982; Wikipedia; Schwebinghaus and Großer, 2000).

Going back to the observation of nature, mathematicians realised that specific forms couldn’t be described in the usual way. There was where Mandelbrot linked in.
Schwebinghaus and Großer (2000) cite Mandelbrot (1975) with a text that Mandelbrot himself has called a manifesto:

> *Clouds are no spheres, mountains are no cones, coastlines are no circles. The bark is not smooth - even lightning doesn’t blaze his trail straight… The existence of such forms challenges us to study, what Euklid leaves aside as formless and leads us to the morphology of the amorphous. So far the mathematicians evaded this challenge. Through the development of theories that no longer have a correlation to the visual world, they depart from nature. Answering to this, we will develop a new geometry of nature and proof it’s advantage in several areas. This new geometry describes a lot of these irregular and broken forms around us – namely a family of figures, which we will call fractals …*
> *(rough translation of the German source by Tina Nischan)*

Introducing the term *fractal* can be seen as the birth of a specific subject of mathematics.

Schwebinghaus and Großer (2000) explain further, that the word *fractal* is derived from the Latin word *fractus* = broken, which refers to topological dimensions as well as to the actual appearance.

The most ostentatious characteristic of fractals is their self-similarity or recursive construction. This means, that each form is constructed by smaller forms which look – partially or exactly - identical to the bigger part. Examples for that are ferns or the Sierpinski Triangle (see image1-1).

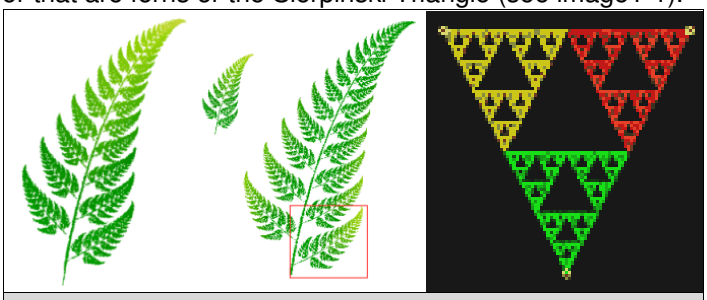


Image1-1. Schwebinghaus and Großer (2000): examples for self-similarity. When the red framed part of the fern (left) is rotated, mirrored and scaled it proofs to be identical to the original. The same applies for the Sierpinski Triangle (right).

Since Mandelbrot's ground braking definition a lot of sub-classes of fractal systems have been established.
In addition it was soon discovered, that fractal algorithms can be used as tools to create specific images for their aesthetic appearance – fractal images as art.

This project looks into the algorithms of the Iterated Function System (IFS) and their use to produce specific, aesthetical pleasing images called "Fractal Flames".

2 – Exploring the chaos game

The different sources (Draves, 2005; Schwebinghaus and Großer, 2000; Wikipedia;) describe the functionality of the chaos game with more or less details on the mathematics. Essentially it can be broken down to some basic aspects.

Following Schwebinghaus and Großer (2000) the basic concept is a feedback loop.
To begin with, lets assume a function $f(x)$, a set of numbers D and a start value x_1 which is part of D. A specific sequence of numbers can be built by inserting x into the function, taking the result, feeding it back into the function, and so on.

$X_2 = f(x_1)$
$X_3 = f(x_2)$
$X_4 = f(x_3)$
…

This concept is called recursion. One step is called iteration.

Does the equation reach a point where f(x) = x, meaning, the starting value is equal to the result, then this number is called a fixed point or attractor (Wikipedia).

Instead of using only one function a whole set of them can be accessed, one after the other. The sequence of the access depends on a random choice, called the chaos game.
“Chaos” is a misleading expression for it is not chaotic at all, but really means randomness. Strictly speaking even not that, because after enough iterations every point of the solution set will be generated eventually.
The set of functions used in the calculations built the Iterated Function System (IFS).

The chaos game can be summarised as followes:
A start value of a specific range is chosen and fed into a calculation.
This calculation is one of a set of functions – the Iterated Function System (IFS) set – which are accessed randomly.
The result is then fed back into the functions and so on as often as the iteration is specified.

3 – About fractal flames

This chapter is based mainly the original paper by Scott Draves (2005). It is not meant as a replacement for the detailed description of the original paper, but outlines the idea and summarises the information, that was taken out to work on for the project.

There are some major differences between a classical Iterated Function System and the “Fractal Flame” algorithm.
To start with, the functions used in the chaos game are non-linear.
Further differences refer especially to the method of displaying which is essential for the specific aesthetics of these images.
Instead of using a binary, hence black and white, representation or a linear greyscale, the colour is based on a log density system. In addition a colouring by structure instead of density is introduced.
All factors of the appearance of an image make use of these specifics.

Unfortunately Draves gives a lot of results without explaining, how he achieved them. But it seams to be a fair guess, that he went through a lot of experimental sessions to find the best approach. In other words: trial and error.

3.1 – Scott Drave’s fractal flame algorithm

The classical Iterative Function System (IFS), as described by Barnsly (1988) and cited by Draves (2005), is a set of recursive linear functions. Technically they are affine as each is capable of expressing scale, rotation, translation and shear.

The normal algorithm for solving the equations is called the chaos game, which can be discribed in pseudocode as follows:

(x,y)= a random point in the bi-unit square
iterate {

i = a random integer from 0 to n-1 inclusive
$(x,y) = F_i(x,y)$
plot (x,y) except during the first 20 iterations
}
where bi-unit square means, that x and y lie in [-1,1].
An example for this classical approach is the Sierpinski Triangle.

The “Fractal Flame” algorithm is a variation to this approach, using of a set of non-linear functions.
To guarantee that the points come together, contractive functions are used. It turned out, that an on average contractive set works best.
Actually Draves (2005) uses a lot more, but in his paper he gives sixteen examples of his function set. Images are derived by a combination of two, three or more of them.
The result of the so installed chaos game is plotted on a plain.

Image3.1-1. Draves (2005) Selected examples of the function set given in his paper. From left to right: linear, sinusoidal, polar, heart, swirl and diamond.

Special attention has to be taken to how the results are plotted, for to preserve and expose as much of the information as possible means to maximise the aesthetics.
Throughout the chaos game the points of the solution set will usually be hit more than once and by different functions. If the points were plotted in a binary, hence black and white image, all this detailed information would be lost. A grey scale would be an improvement. But looking to the histogram, it is obvious, how uneven the number of hits is. The densest points are much denser than the average density, thus with a linear map most of the image is very dark, and information is lost.

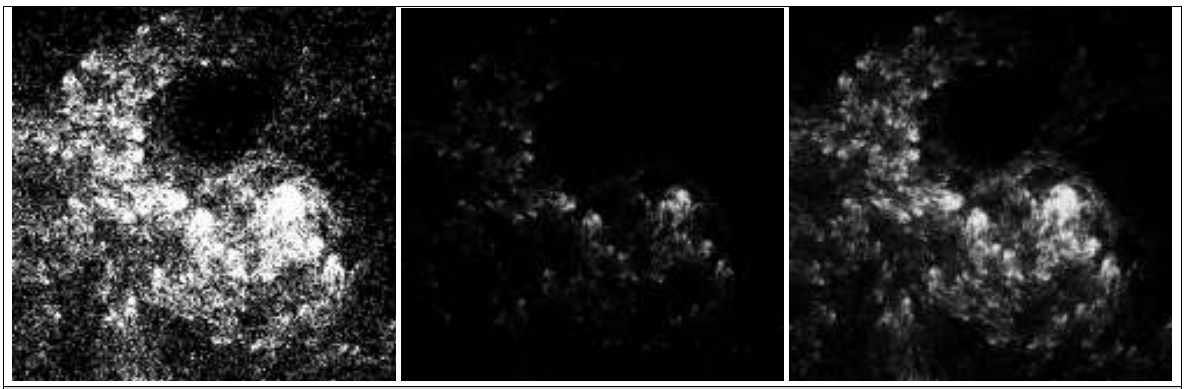


Image3.1-2. Draves (2005). From left to right: Displaying as binary map, linear grey scale, logarithmic grey scale.

A solution to this is the use of a logarithmic map from density to brightness, which take this differences in account.

The next step is to introduce colour. Therefore three counters are used to increase the rgb-values with every iteration plus a fourth one for the alpha value.
If the colour was allocated to the density again, all information about where the value came from would be lost. But it is in essence this specific value, the recognisability of a specific function, which helps to give

the illusion of three dimensions. While animating, the different elements of the used set, meaning the functions, can be followed.

Based on this essential characteristic, Draves (2005) describes a couple of additional calculations to improve the aesthetics.

The next step he suggests is to take a colour correction into account. He refers to the non-linear responses of screen phosphors to voltage in normal Cathode Ray Tube (CRT), where the different phosphors for red, green and blue react inhomogeneous. Another value for independent calculations is suggested and called *vibrancy*.

Draves (2005) also emphasises the importance of symmetry. He found out, that rotational or dihedral symmetries give optimal results. Of course to achieve this again another function has to be added. Two way symmetry results in 180° rotation, three way symmetry in 120° rotation. Dihedral symmetry refers to inverting the x-coordinate. A combination of both creates snowflake like structures.
Special attention has to be taken to the balancing of the functions.

Similar to other pixel based images anti-aliasing is an important topic. Unfortunately it is not easily achieved here. Especially using motion blur while animating is a complex calculation because of the non-linearity of log-density display.

Image3.1-3. Examples of “Fractal Flames”. Available from: http://www.flam3.com/

The main work of the “Fractal Flame” algorithm lies in the complex task to make the images beautiful through a series of calculations and variables that can be influenced.

3.2 – Electric Sheep

Electric Sheep is the name of a distributed screensaver, that makes use of the “Fractal Flame” algorithm. The name is a reference to Philip K. Dick's novel "*Do Androids Dream of Electric Sheep?*". Draves (2004) describes the project as a realisation of the collective dreams of sleeping computers from all over the internet.
Clients render JPEG frames and upload them to a server. After all frames have been calculated, the server compresses them to a MPEG file that shows a looped animation.
These animations can be downloaded and voted for or against. By voting, selected images, saying “sheep” live longer and reproduce more often. Through this interaction a specific aesthetic crystallises.

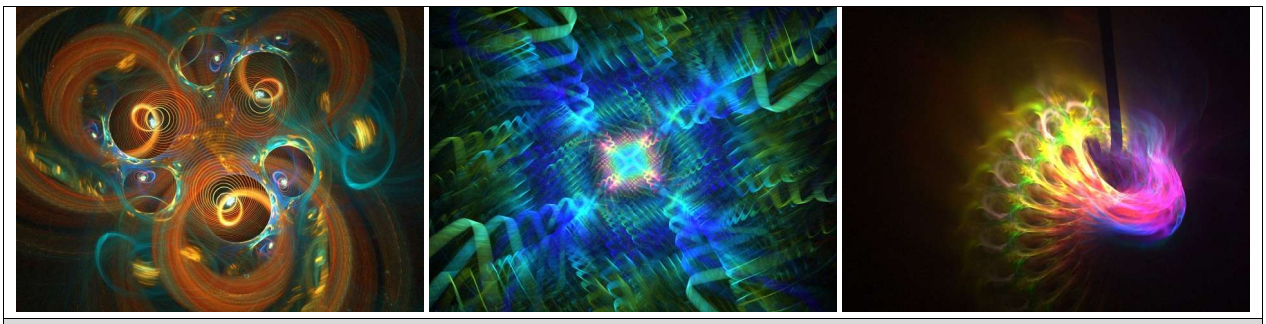


Image3.2-1. Examples of Electric Sheep. Avaiable from: http://electricsheep.org/

4 - The project

4.1 – Aim

This project's target was to research an existing solution and trying to implement it with the available possibilities. Therefore the aim was to learn to break down a complex task into several smaller, manageable steps and to translate it into code.
The aesthetics of the "Fractal Flames" served as a motivation but not as the final aim.

4.2 – Approach

As most other projects this one started with the research on the "Fractal Flame" algorithm. Therefore it was necessary to look into Iterated Function Systems (IFS) and the chaos game in general.
It became clear quite soon, that the complex calculations on the beauty of the "Fractal Flames" can't be achieved in the given time.

The finally set task was framed out like follows:

- create a possibility for displaying an image. Open GL was chosen because of former projects.
- Implement a structure to save image pixel information.
- Set an Iterative Function System (IFS) with linear functions, here the Sierpinski Triangle, and display it in black and white.
- Expand the system to a chaos game and display the result in black and white. The chosen example was Draves cloud-like structure.

Every step further was not part of the task.
Though, another example pattern by Scott Draves and two additional, experimental patterns were implemented. Also, a logarithmic density map for grey scale as well as colour schemes have been added.

4.3 – Details

4.3.1 – Manual

The program is designed for a LINUX platform.

For starting the application run *TinaFractal*.

After welcoming, the user is ask to specify variables for the creation of the image.

Iterations:

The more iterations the clearer becomes the pattern. On the other hand, more iterations need more time to calculate.
While experimenting it turned out, that the colour types “rainbow” and “all blue” with their multiple steps gain from more iterations.
A range of 1000,000 – 100,000,000 iterations turned out to be sufficient, more can be done for the named colour types.

Pattern:

1- Sierpinsky Triangle. Though this is strictly speaking not a member of the “Fractal Flame” algorithm it was left in the selection for documentary reasons. Anything else but a black and white map is a waist on this pattern.

The following patterns can be used with all colour types equally:

2- The cloud-like pattern, an example given by Scott Draves and shows exactly what its named and can be used with all colours.
3- “linear & spiral & swirl” is the second example provided by Scott Draves (for examples see appendix).
4- Is a first test to create a pattern new…
5- … the same counts for this collection of functions.

All patterns after that are there only for documentary reason. These are the examples of Scott Draves called separately. Because these functions were never meant for a singular use in the chaos game, they don’t result in sufficient images.

Colour type:

1- Black and white. Again this remains for documentary reason.
2- A grey scale based on logarithmic density map (the linear grey scale was left out because hardly anything was visible).
3- “Traffic light colours” ...
4- “Rainbow” and…
5- “All blue” works with a logarithmic density map as well.

The last three colour types differ mainly in the number of steps over the range (see below).

After all variables are set, the rendering is initiated. After a calculation time depending on the number of iteration, a window is opend and the image displayed.

Colour type “traffic light”: Left column: range. Right column: rgb-values left from 0 to 255, right from 0 to 1.	
0.75 – 1.0	0 – R – 0.000 255 – G – 1.000 255 – B – 1.000
0.5 – 0.75	0 – R – 0.000 255 – G – 1.000 0 – B – 0.000
0.25 – 0.5	255 – R – 1.000 255 – G – 1.000 0 – B – 0.000
0.0 – 0.25	0 – R – 0.000 255 – G – 1.000 0 – B – 0.000

Colour type “rainbow”: Left column: range. Right column: rgb-values left from 0 to 255, right from 0 to 1.	
0.9 – 1.0	255 – R – 1.000 0 – G – 0.000 255 – B – 1.000
0.8 – 0.9	128 – R – 0.500 0 – G – 0.000 255 – B – 1.000
0.7 – 0.8	0 – R – 0.000 0 – G – 0.000 255 – B – 1.000
0.6 – 0.7	0 – R – 0.000 128 – G – 0.500 255 – B – 1.000
0.5 – 0.6	0 – R – 0.000 255 – G – 1.000 255 – B – 1.000
0.4 – 0.5	0 – R – 0.000 255 – G – 1.000 0 – B – 0.000
0.3 – 0.4	255 – R – 1.000 255 – G – 1.000 0 – B – 0.000
0.2 – 0.3	255 – R – 1.000 128 – G – 0.500 0 – B – 0.000
0.1 – 0.2	255 – R – 1.000 0 – G – 0.000 0 – B – 0.000
0.0 – 0.1	255 – R – 1.000 0 – G – 0.000 128 – B – 0.500

Colour type “all blue”: Left column: range. Right column: rgb-values left from 0 to 255, right from 0 to 1.	
0.9 – 1.0	0 – R – 0.000 255 – G – 1.000 199 – B – 0.780
0.8 – 0.9	0 – R – 0.000 255 – G – 1.000 239 – B – 0.937
0.7 – 0.8	0 – R – 0.000 223 – G – 0.875 255 – B – 1.000
0.6 – 0.7	0 – R – 0.000 167 – G – 0.655 255 – B – 1.000
0.5 – 0.6	0 – R – 0.000 120 – G – 0.469 255 – B – 1.000
0.4 – 0.5	0 – R – 0.000 64 – G – 0.251 255 – B – 1.000
0.3 – 0.4	24 – R – 0.094 0 – G – 0.000 255 – B – 1.000
0.2 – 0.3	88 – R – 0.345 0 – G – 0.000 255 – B – 1.000
0.1 – 0.2	144 – R – 0.563 0 – G – 0.000 255 – B – 1.000
0.0 – 0.1	231 – R – 0.906 0 – G – 0.000 255 – B – 1.000

4.3.2 – Through the code

After running the executable *TinaFractal* the user is first of all asked to specify variables necessary for calculating the images (main.cpp). For details on the dialog see chapter 4.3.1 – Manual.
The variables iIterations, iPattern and iColourtype are defined in the calc header file and hold integers.
Though for most of the calculations doubles or floats are more suitable – calculations have to be done without losing details in rounding – for the user defined variables integers are more than enough. They are only used for counting iterations or specifying a choice.
In case the specification of the user defined variables fails, they have been predefined with valid values by the calculation class constructer *Ccalculation()*.

Assuming everything worked fine, the program steps on to the initialisation of Open GL, creates a window according to the specifications and calls the initial calculation of the chaos game in the calc.cpp file.

The *initial calculation function* deals first of all with the randomisation.
Also member of the calculation class is a randomisation function called *randomNum*.
Random choices are essential for the chaos game. But a computer doesn't really know random choices, but bases a sequence of choice always on the same start up. Therefore this starting point, *the seed*, has to be unrepeatable to secure randomness. Here the internal clock was used as a seed.

Back in the initial calculation function, the feedback loop of the chaos game is started by choosing for a x-value as well as for a y-value a random number between –1 and 1 and passed into a loop that calls the actual calculation function as often as the number of iterations was defined.

The *actual calculation function* provides all patterns that are available. Each (proper) pattern here contains a set of two or three functions. Depending on the choice, only one set is called.
Each pattern calculation starts with choosing a random number which is necessary to decide which function of the set is calculated. Depending on the result, x and y are passed into the appropriate function, which results in a new x and y.

After having calculated a *new x and y* value, for the time being they are passed on inside the actual calculation function.

The next step is a test on the actual value.
Here initially a mistake was made. Keeping in mind that the window width and height define the range of –1 to 1 for displaying the calculated functions and that, while following the chaos game iterations, the x and y values might very well go outside –1 to 1, it was originally assumed that after one iteration the values have to be cropped. That turned out to be completely wrong or at least at this position.
In fact the whole chaos game has to be allowed to go over the full range (as long as the number doesn't go close to infinity) to enable the full pattern, e.g. a whole circle, from which later on maybe only a small part is visible.
The test here is not a test for the –1 to 1 range but for infinity (or something near).

After that the x and y values are passed into the function, that converts to screen coordinates. This function's only use is to count hits. Depending on the calculated coordinates, these values are passed to a three-dimensional array defined in the screen class, the first and second value used for the coordinates, the third for incrementing each time it is hit.
In addition the maximum hits are counted, for this value will be used later on when calculation the colour.

With finishing the maximum hit count function the one iteration of the actual calculation function has been fulfilled and the new x and y value are given back to the loop in the initial calculation function, where they are directly fed into the same pattern again as often as the iteration count increases.

So far nothing has been displayed, only information stored.

When the iteration loop is fulfilled, the function is called, that calculates the colour based on a logarithmic density map.

The *calculateHitsToColour* function makes use of an array, that is stored in the screen class as well. This array is meant to store the information for displaying a pixel of the image. So to say, each and every pixel is stored with its coordinates and rgba-values.
What the function has to calculate is, what these values are based on the array, that counted the hits for each pixel.

After entering a loop that goes through the complete array, the calculateHitsToColour function moves on to the chosen colour type, depending on the user's choice. The logarithmic calculation is executed and the results for rgba are stored in the displaying array.

Having fulfilled that, the function jumps back to the initial calculation, which is fulfilled as well, so the information is given back to the main function.

Now, after the complete calculation is done and the results stored, the glut main loop calls the function, that draws to the screen.
The array is traversed for the information of each and every pixel that is displayed.

4.3.3 – Continuative ideas

The way the code is written so far, the single steps are still traceable, which surely helps for the understanding. But thinking about how many variables, counters, patterns, colours etc. Drave implemented, it surely would gain from sourcing out code to additional files or classes.

So far the colour is based on density, which has nice results as well, but is not what is suggested for the "Fractal Flame" algorithm. The next step here might be, to implement additional counters for rgba separately and implement a check, from which calculation the resulting hit comes from.

A proper user interface would be nice, as well as a possibility to store the resulting images.

Having progressed in the set task so far, the possibility of animating the "Fractal Flames" doesn't seem to be so far away any more.
Images could be stored, their values alternated from image to image.

The possibilities of Open GI could be used more, especially for adding functionality like anti-aliasing.

5 – Summery

The "Fractal Flame" algorithm divers from the classical Iterated Function System (IFS) by introducing non-linear functions into the chaos game and in addition concentrating on implementing calculations which optimise the aesthetics of the images. Essential is the use of a logarithmic density map and a colouring based on the used function.

Choosing this topic as a project turned out to be a pleasing choice, because it provided aspects of two sides, technique and art.

Following Scott Draves paper as a guideline was helpful.

This project started with few hopes to end up with a visible result. The biggest lesson that was learned, was how to break down big problems in smaller ones, which are easier to handle.

Once the connections between the different aspects of calculating and displaying where established, it was easier to add bits and pieces.

Open GL and C++ seem to be a good choice for implementing. The only flaw is, that the handling of images is not as easy achieved.

Though finally the aesthetics of “Fractal Flames” where not achieved, the project has developed to a level, where it is possible to play around and try new combinations and new colours and therefore achieving new looks.

Apart from all technical knowledge and newly achieved know-how, it was this pleasure in seeing the result developing which will stick permanently.

Acknowledgment

My special thanks goes to Andy Cousins who's help was invaluable, especially in spotting my mistakes while following my line of thought, pushing me into the right direction, advising me on syntax and encouraging me whenever I struggled.

Thanks to Eike Anderson I had an optimal start in implementing my code with a framework in Open GL. Also he proofed to be a unfailing source for advice and reference material.

Thanks go to John Macey as well, because he allowed us to use his code examples. I happily accepted to use his detailed makefile.

Last but not least, I would like to thank Anargyros Sarafopoulos for his support and help through the development of this project and encouraging me to go for it.

References

BARNSLEY, M., 1988. *Fractals everywhere*. San Diego: Academic Press.

DRAVES, S., 2004. *The Interpretation of Dreams: An Explanation of the Electric Sheep Distributed Screen-saver* [online]. San Francisco, USA. Available from: http://electricsheep.org/ [Last access 8 March 2006].

DRAVES, S., 2005. *The Fractal Flame Algorithm* [online]. (no information about where it was published). Available from: http://www.flam3.com/index.cgi?&menu=math [Last access 8 March 2006].

MANDELBROT, B. B., 1982 (1975). *The fractal geometry of nature*. New York : Freeman.

SCHWEBINGHAUS, U. AND GROßER, B., 2000. *Fraktale und Chaosspiel – Mit Würfeln zu Fraktalen Welten* [online]. University of Wuppertal / Germany. Available from: http://www.matheprisma.uni-wuppertal.de/index.htm [Last access 8 March 2005].
(Translated by Tina Nischan)

WIKIPEDIA – The Free Encyclopedia: *Chaos game* [online]. Available from: http://en.wikipedia.org/wiki/Chaos_game [Last access 8 March 2006].

WIKIPEDIA – The Free Encyclopedia: *Fractal* [online]. Available from: http://en.wikipedia.org/wiki/Fractals [Last access 8 March 2006].

WIKIPEDIA – The Free Encyclopedia: *Fractal flame* [online]. Available from: http://en.wikipedia.org/wiki/Fractal_flame [Last access 8 March 2006].

Appendix

Main.cpp–file: mainly setting the Open GL framework for being able to display something and providing choices for the user.

```
//------------------------------------------------------------------------
Open GL framework with help from Eike Anderson
//------------------------------------------------------------------------

#include <GL/glut.h> // enough to deal with opening window
#include "screen.h"
#include "calc.h"
#include <iostream>
using namespace std; // using c++

//------------------------------------------------------------------------

screen myScreen;
int iScreenWidth=800;  //never go over 1000, see array size
int iScreenHight=800;  //never go over 1000, see array size
CCalculation myCalc;

//------------------------------------------------------------------------

void preset(void);  // declare function for settings of window
void display(void); // declare function for displaying window

//------------------------------------------------------------------------



int main(int argc, char *argv[]) // commandline arguments because of glut s. next
{
  //dialog to set user defined variables
 cout<<endl;
 cout<<endl;
 cout<<"-----------------------------------------------"<<endl;
 cout<<"--Welcome to Tina's tests on fractal flames.--"<<endl;
 cout<<"-----------------------------------------------"<<endl;
 cout<<endl;
 cout<<endl;
 cout<<"Please choose the number of iterations"<<endl;
 cout<<"(recommended between 100000 and 100000000,"<<endl;
 cout<<"for the colours 'rainbow' and 'all blues' even more)."<<endl;
 cout<<endl;
 cin>>myCalc.iIterations;
 cout<<endl;
 cout<<endl;
 cout<<"Please choose a pattern from the following list:"<<endl;
 cout<<"1 - Sierpinsky's Triangle, functions by Scott Draves."<<endl;
 cout<<"2 - Cloud-like , example by Scott Draves."<<endl;
 cout<<"3 - linear & spiral & swirl , example by Scott Draves."<<endl;
 cout<<"4 - Tina's try out 1."<<endl;
 cout<<"5 - Tina's try out 2."<<endl;
 cout<<"The following are the original functions provided by Scott Draves."<<endl;
 cout<<"The used coefficients are taken from the above examples."<<endl;
 cout<<"These functions are not meant to be used separately,"<<endl;
 cout<<"so the result is not very pleasing."<<endl;
 cout<<"6 - Scott Drave's - linear -."<<endl;
 cout<<"7 - Scott Drave's - sinusoidal -."<<endl;
 cout<<"8 - Scott Drave's - spherical -."<<endl;
 cout<<"9 - Scott Drave's - swirl -."<<endl;
 cout<<"10 - Scott Drave's - horseshoe -."<<endl;
 cout<<"11 - Scott Drave's - polar -."<<endl;
 cout<<"12 - Scott Drave's - handkerchief -."<<endl;
 cout<<"13 - Scott Drave's - heart -."<<endl;
 cout<<"14 - Scott Drave's - disc -."<<endl;
 cout<<"15 - Scott Drave's - spiral -."<<endl;
 cout<<"16 - Scott Drave's - hyperbolic -."<<endl;
```

```
 cout<<"17 - Scott Drave's - diamond -."<<endl;
 cout<<"18 - Scott Drave's - ex -."<<endl;
 cout<<"19 - Scott Drave's - fisheye -."<<endl;
 cout<<"20 - Scott Drave's - cosine -."<<endl;
 cout<<endl;
 cin>>myCalc.iPattern;
 cout<<endl;
 cout<<endl;
 cout<<"Please choose the type of colouring from the following list:"<<endl;
 cout<<"1 - Black & white."<<endl;
 cout<<"2 - Logarithmic grey scale."<<endl;
 cout<<"3 - Traffic light colours."<<endl;
 cout<<"4 - Rainbow colours."<<endl;
 cout<<"5 - All blues."<<endl;
 cout<<endl;
 cin>>myCalc.iColourType;
 cout<<endl;
 cout<<endl;



 glutInit(&argc,argv); // pass in the addresses of the command-line argumets
 glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA); // Double-Buffered because calculating and drawing
                                                                                 at same time,

 glutInitWindowSize(iScreenWidth,iScreenHight); //square for later -1 - +1
                                                                    left - right, up - down
 glutInitWindowPosition(100,50);
 glutCreateWindow("Tina's Fractal Flame Test");

 glutDisplayFunc(display); // call the display function

 preset();

 myCalc.initCalculate(); //do the calculation once

 glutMainLoop(); // enter event loop

 return 0;
}



void preset(void)
{
  glClearColor(0.0,0.0,0.0,0.0); // set "background" colour to black
  glViewport(0,0,iScreenWidth,iScreenHight); // set viewport of 800x800 in
                                                                          middle of window
  glMatrixMode(GL_PROJECTION);  // set projection matrix
  glLoadIdentity(); // reset to Identity Matrix
  gluOrtho2D(0.0,iScreenWidth,0.0,iScreenHight); // create 2D projection
                                                              matrix, cause don't need 3D
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  //"beauty", so that the drawn pixel is really "in the middle" of the screen pixel
  glTranslatef(0.375,0.375,0.0);
}



void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT); // "clear board" before drawing
  preset();

  myScreen.drawScreen(); //draw what has been calculated

  glFlush(); // draw on screen
  glutSwapBuffers(); // swap because double buffers, not necessary in single buffer
```

```
}
```

Calc.cpp–file: the “heart” of the code, calculations on chaos game and colour types.

```
#include <iostream>
#include <math.h>
#include <time.h>
using namespace std;
#include "calc.h"
#include "screen.h"

//------------------------------------------------------------------------

extern int iScreenWidth; //never go over 1000, see array size
extern int iScreenHight; //never go over 1000, see array size
extern screen myScreen; //myScreen is defined in main

//------------------------------------------------------------------------



CCalculation::CCalculation()
{
   dx = dy = iMaxHit = 0;
   //in case no valid input by user make it work at all
   iIterations=1000000;
   iPattern=1;
   iColourType=1;
}



double CCalculation::randomNum( double dMin, double dMax )
{
   double dTemp;
   double dRange = ( dMax - dMin );
   double dNewRand;

   dTemp = rand();

   dNewRand = ( dTemp / ( double )RAND_MAX ) * dRange; //typecast otherwise
                                                       it gives back NO fractions

   return ( dMin + dNewRand );
}



void CCalculation::initCalculate()
{
   srand( time( NULL ) ); //here, because it is counting seconds and I have
                                                    to give it time to count further

   dx = randomNum( -1.0, 1.0 );
   dy = randomNum( -1.0, 1.0 );

   cout<<"calculating......."<<endl;
   cout<<endl;
   cout<<endl;

   for ( int i = 0; i < iIterations; i++ )
   {
      actualCalculate();
   }
   calculateHitsToColour( iScreenWidth, iScreenHight );
}
```

```
void CCalculation::actualCalculate()
{
   double dPI = 3.14159265358979323846264338327795;

   //-------------------------------------------------------------------
   //all possible pattern
   //-------------------------------------------------------------------

   //-------------------------------------------------------------------
   //those provided by Scott Drave's paper
   //-------------------------------------------------------------------
   if ( iPattern == 1 )  //Test Sierpinsky Triangle, functions from Scott Drave's paper
   {
      int iFuncNum = ( int )randomNum( 0, 3 );
      if ( iFuncNum == 0 )
      {
         dx = dx / 2.0;
         dy = dy / 2.0;
      }
      else if ( iFuncNum == 1 )
      {
         dx = ( dx + 1.0 ) / 2.0;
         dy = dy / 2.0;
      }
      else
      {
         dx = dx / 2.0;
         dy = (dy+1)/2.0;
      }
   }
   else if ( iPattern == 2 ) //"cloud", functions and coefficients by Scott Draves
   {
      int iFuncNum = ( int )randomNum( 0, 2 );
      if ( iFuncNum == 0 ) //spherical
      {
         dVx = ( 0.562482 * dx ) + ( -0.539599 * dy ) + ( -0.42992 );
         dVy = ( 0.397861 * dx ) + ( 0.501088 * dy ) + ( -0.112404 );
         dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
         dx = dVx / ( dRadius * dRadius );
         dy = dVy / ( dRadius * dRadius );
      }
      else
      {
         dVx = ( 0.830039 * dx ) + ( 0.16248 * dy ) + ( 0.91022 );
         dVy = ( -0.496174 * dx ) + ( 0.750468 * dy ) + ( 0.288389 );
         dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
         dx = dVx / ( dRadius * dRadius );
         dy = dVy / ( dRadius * dRadius );
      }
   }
   else if ( iPattern == 3 ) //"linear&spiral&swirl", functions and coefficients by Scott Draves
   {
      int iFuncNum=(int)randomNum(0,3);
      if (iFuncNum == 0) //linear
      {
         dVx = ( 0.98396 * dx ) + ( 0.359416 * dy ) + ( -0.85059 );
         dVy = ( 0.298328 * dx ) + ( -0.583541 * dy ) + ( 0.378754 );
         dx = dVx;
         dy = dVy;
      }
      else if ( iFuncNum == 1 ) //spiral
      {
         dVx = ( -0.900388 * dx ) + ( 0.397598 * dy ) + ( 0.465126 );
         dVy = ( 0.293063 * dx ) + ( 0.0225629 * dy ) + ( -0.277212 );
         dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper
                                                                  p.12, root of x*x+y*y
         dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of
                                                                                  y/x
         dx = ( cos( dAngleTheta ) + sin( dRadius ) ) / dRadius;
```

```
      dy = ( sin( dAngleTheta ) + cos( dRadius ) ) / dRadius;
   }
   else //swirl
   {
      dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
      dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
      dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of y/x
      dx = dRadius * ( cos( dAngleTheta + dRadius ) );
      dy = dRadius * ( sin( dAngleTheta + dRadius ) );
   }
}

//-----------------------------------------------------------------------
//Tina try outs
//-----------------------------------------------------------------------
else if ( iPattern == 4 ) //Tina Test 1
{
   int iFuncNum = ( int )randomNum( 0, 2 );
   if ( iFuncNum == 0 ) //diamond
   {
      dVx = dx; //not enough time to try out more coefficients
      dVy = dy;
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
      dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of y/x
      dx = sin( dAngleTheta ) * cos( dRadius );
      dy = cos( dAngleTheta ) * sin( dRadius );
   }
   else //heart
   {
      dVx = dx;
      dVy = dy;
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
      dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of y/x
      dx = dRadius * ( sin( dAngleTheta * dRadius ) );
      dy = -1.0 * ( dRadius * ( cos( dAngleTheta * dRadius ) ) );
   }
}
else if ( iPattern == 5 ) //Tina Test 2
{
   int iFuncNum = ( int )randomNum( 0, 4 );
   if ( iFuncNum == 0 ) //discalone
   {
      dVx = dx;
      dVy = dy;
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y

      dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of y/x
      dx = ( dAngleTheta * ( sin( dPI * dRadius ) ) ) / dPI;
      dy = ( dAngleTheta * ( cos( dPI * dRadius ) ) ) / dPI;
   }
   else if ( iFuncNum == 1 ) //sinusoidal
   {
      dVx = dx;
      dVy = dy;
      dx = sin( dVx );
      dy = sin( dVy );
   }
   else  //spiral
   {
      dVx = dx;
      dVy = dy;
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) ); //see paper p.12, root of x*x+y*y
      dAngleTheta = atan( dVy / dVx ); //see paper p.12, arctan of y/x
      dx = ( cos( dAngleTheta ) + sin( dRadius ) ) / dRadius;
      dy = ( sin( dAngleTheta ) + cos( dRadius ) ) / dRadius;
   }
}
```

```
//-----------------------------------------------------------------------
//chaos game, functions from Scott Drave's paper, p.12
//coefficients from Scott Draves privious examples
//-----------------------------------------------------------------------
else if ( iPattern == 6 )  //linear
{
   dVx = ( 0.98396 * dx ) + ( 0.359416 * dy ) + ( -0.85059 );
   dVy = ( 0.298328 * dx ) + ( -0.583541 * dy ) + ( 0.378754 );
   dx = dVx;
   dy = dVy;
}
else if ( iPattern == 7 ) //sinusoidal
{
   dVx = ( -0.900388 * dx ) + ( 0.397598 * dy ) + ( 0.465126 );
   dVy = ( 0.293063 * dx ) + ( 0.0225629 * dy ) + ( -0.277212 );
   dx = sin( dVx );
   dy = sin( dVy );
}
else if ( iPattern == 8 ) //spherical
{
   dVx = ( -0.900388 * dx ) + ( 0.397598 * dy ) + ( 0.465126 );
   dVy = ( 0.293063 * dx ) + ( 0.0225629 * dy ) + ( -0.277212 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dx = dVx / ( dRadius * dRadius );
   dy = dVy / ( dRadius * dRadius );
}
else if ( iPattern == 9 ) //swirl
{
   dVx = ( 0.125503 * dx ) + ( -0.00125 * dy ) + ( 0.004588 );
   dVy = ( -0.0850061 * dx ) + ( -0.136802 * dy ) + ( 0.430655 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dRadius * ( cos( dAngleTheta + dRadius ) );
   dy = dRadius * ( sin( dAngleTheta + dRadius ) );
}
else if ( iPattern == 10 ) //horseshoe
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dRadius * ( cos( 2.0 * dAngleTheta ) );
   dy = dRadius * ( sin( 2.0 * dAngleTheta ) );
}
else if ( iPattern == 11 ) //polar
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dAngleTheta / dPI;
   dy = dRadius - 1.0;
}
else if ( iPattern == 12 ) //handkerchief
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dRadius * ( sin( dAngleTheta + dRadius ) );
   dy = dRadius * ( cos( dAngleTheta - dRadius ) );
}
else if ( iPattern == 13 ) //heart
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dRadius * ( sin( dAngleTheta * dRadius ) );
```

```
   dy = -1.0 * ( dRadius * ( cos( dAngleTheta * dRadius ) ) );
}
else if ( iPattern == 14 ) //disc
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = ( dAngleTheta * ( sin( dPI * dRadius ) ) ) / dPI;
   dy = ( dAngleTheta * ( cos( dPI * dRadius ) ) ) / dPI;
}
else if ( iPattern == 15 ) //spiral
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = ( cos( dAngleTheta ) + sin( dRadius ) ) / dRadius;
   dy = ( sin( dAngleTheta ) + cos( dRadius ) ) / dRadius;
}
else if ( iPattern == 16 ) //hyperbolic
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = ( sin( dAngleTheta ) ) / dRadius;
   dy = ( cos( dAngleTheta ) ) * dRadius;
}
else if ( iPattern == 17 ) //diamond
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.54759);
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = sin( dAngleTheta ) * cos( dRadius );
   dy = cos( dAngleTheta ) * sin( dRadius );
   }
else if ( iPattern == 18 ) //ex
{
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = dRadius * ( ( sin( dAngleTheta + dRadius ) ) * ( sin(
             dAngleTheta + dRadius ) ) * ( sin( dAngleTheta + dRadius ) ) );
   dy = dRadius * ( ( cos( dAngleTheta - dRadius ) ) * ( cos(
             dAngleTheta - dRadius ) ) * ( cos( dAngleTheta - dRadius ) ) );
}

//Julia doesnt work for me, there is no sqrt of a negative number
/*else if ( iPattern == 19 ) //julia
{
   double dOmega=0;  //see paper p.12, either 0 or PI
  double dTemp=(int) randomNum(0,2);
  if (dTemp==0)
   double dOmega=0;
  else
    double dOmega=dPI;
   dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
   dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
   dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
   dAngleTheta = atan( dVy / dVx );
   dx = sqrt( dRadius * ( cos( ( dAngleTheta / 2.0 ) + dOmega ) ) );
   dy = sqrt( dRadius * ( sin( ( dAngleTheta / 2.0 ) + dOmega ) ) );
}*/

     else if ( iPattern == 19 ) //in paper = fct 16, fisheye
{
```

```
      dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
      dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
      dAngleTheta = atan( dVy / dVx );
      dx = dx * ( 2.0 * dRadius / ( dRadius + 1.0 ) );
      dy = dy * ( 2.0 * dRadius / ( dRadius + 1.0 ) );
   }
   else if ( iPattern == 20 )//cosine
   {
      dVx = ( -0.329863 * dx ) + ( -0.369381 * dy ) + ( 0.977861 );
      dVy = ( -0.0855261 * dx ) + ( -0.858379 * dy ) + ( 0.547595 );
      dRadius = sqrt( ( dVx * dVx ) + ( dVy * dVy ) );
      dAngleTheta = atan( dVy / dVx );
      dx = cos( dx * dPI ) * cosh( dy );
      dy = -sin( dx * dPI ) * sinh( dy );
   }


   //the problem is infinity, not going over -1 to 1
   if ( ( dx > 1e10 || dx < -1e10 ) || ( dy > 1e10 || dy < -1e10 ) )
   {
      dx = randomNum( -1.0, 1.0 );
      dy = randomNum( -1.0, 1.0 );
   }
   else
   {
      //choose double for temporal variable for not loosing exact number
      dSx = dx;
      dSy = dy;

      // take functions x and y and convert to screen coordinates
      //convert only what lies within -1.0 - 1.0
      if ((dSx >= -1.0 && dSx<=1.0) && (dSy >= -1.0 && dSy<=1.0))
      {
         int iScreenX = 0; //initialise to zero, before using for actual calculation
         iScreenX = convertToScreen( dSx, iScreenWidth );
         int iScreenY = 0;
         iScreenY = convertToScreen( dSy, iScreenHight );

         //count hits instead of putting it directly onto screen
         myScreen.fScreenHits[iScreenX] [iScreenY] =
                                    myScreen.fScreenHits[iScreenX] [iScreenY] + 1;

         //count maximum hits for using value for scaled/logarithmic colour calculation
         if ( iMaxHit < myScreen.fScreenHits[iScreenX] [iScreenY] )
         {
            iMaxHit = myScreen.fScreenHits[iScreenX] [iScreenY];
         }
      }
   }
}



int CCalculation::convertToScreen( double dValue, int iScreenValue )
{
   int iCoord = ( dValue + 1.0 ) * ( ( float )iScreenValue / 2.0 ); //temporal
                                                  variable to calculate screen coordinate

   //check to stay in limits, !!!800 array positions BUT 0-799
   if ( iCoord < 0 )
      iCoord = 0;
   if ( iCoord >= iScreenValue ) iCoord = iScreenValue - 1;
      return ( iCoord );
}
```

```
void CCalculation::calculateHitsToColour( int iSizeWidth, int iSizeHight )
{
   for ( int i = 0; i < iSizeWidth; i++ )
   {
      for ( int j = 0; j < iSizeHight; j++ )
      {
         if ( myScreen.fScreenHits[i] [j] != 0 ) //test transfer data from hit count to display
array
         {
            if ( iColourType == 1 ) //Black & White
            {
               myScreen.fScreenArray[i] [j] [0] = 1.0;
               myScreen.fScreenArray[i] [j] [1] = 1.0;
               myScreen.fScreenArray[i] [j] [2] = 1.0;
               myScreen.fScreenArray[i] [j] [3] = 1.0;
            }

            //-----------------------------------------------------

            else if ( iColourType == 2 ) //logarithmic grayscale, result
                                         dependent on maximum hits at a point
            {
               myScreen.fScreenArray[i] [j] [0] = log10( ( double
                )myScreen.fScreenHits[i] [j] ) / ( log10( ( double )iMaxHit ) );
               myScreen.fScreenArray[i] [j] [1] = log10( ( double
                )myScreen.fScreenHits[i] [j] ) / ( log10( ( double )iMaxHit ) );
               myScreen.fScreenArray[i] [j] [2] = log10( ( double
                )myScreen.fScreenHits[i] [j] ) / ( log10( ( double )iMaxHit ) );
               myScreen.fScreenArray[i] [j] [3] = log10( ( double
                )myScreen.fScreenHits[i] [j] ) / ( log10( ( double )iMaxHit ) );
            }

            //-------------------------------------------------------------

            else if ( iColourType == 3 ) // traffic light colours
            {
               double dLog = log10( ( double )myScreen.fScreenHits[i] [j] ) / (
                                    log10( ( double )iMaxHit ) ); //as in grayscale

               float fRed = 0.0;
               float fGreen = 0.0;
               float fBlue = 0.0;
               float fAlpha = 1.0;

               if ( dLog >= 0.0 && dLog < 0.25 )
               {
                  fRed = ( dLog ) * 2.5;
                  fGreen = 0.0;
                  fBlue = 0.0;
                  fAlpha = 1.0;
               }
               else if ( dLog >= 0.25 && dLog < 0.5 )
               {
                  fRed = 1.0;
                  fGreen = ( dLog - 0.25 ) * 2.5;
                  fBlue = 0.0;
                  fAlpha = 1.0;
               }
               else if ( dLog >= 0.5 && dLog < 0.75 )
               {
                  fRed = 1.0 - ( ( dLog - 0.5 ) * 4.0 );
                  fGreen = 1.0;
                  fBlue = 0.0;
                  fAlpha = 1.0;
            }
               else //light blue is of course not in traffic light… ☺ … but I
                                                try to emphasise the rare high hits
               {
                  fRed = 0.0;
```

```
            fGreen = 1.0;
            fBlue = ( dLog - 0.75 ) * 4.0;
            fAlpha = 1.0;
            }
         myScreen.fScreenArray[i] [j] [0] = fRed;
         myScreen.fScreenArray[i] [j] [1] = fGreen;
         myScreen.fScreenArray[i] [j] [2] = fBlue;
         myScreen.fScreenArray[i] [j] [3] = fAlpha;
      }

      //-------------------------------------------------------------

      else if ( iColourType == 4 ) // rainbow colours
      {
         double dLog = log10( ( double )myScreen.fScreenHits[i] [j] ) / (
                                 log10( ( double )iMaxHit ) ); //as in grayscale

         float fRed = 0.0;
         float fGreen = 0.0;
         float fBlue = 0.0;
         float fAlpha = 1.0;

         if ( dLog >= 0.0 && dLog < 0.1 )  //black to redish pink
         {
            fRed =( dLog ) * 10.0;
            fGreen = 0.0;
            fBlue = ( dLog ) * 5.0;
            fAlpha = 1.0;
         }
         else if ( dLog >= 0.1 && dLog < 0.2 ) //redish pink to red
         {
            fRed = 1.0;
            fGreen = 0.0;
            fBlue = 0.5-((dLog-0.1)*5.0);
            fAlpha = 1.0;
         }
      else if ( dLog >= 0.2 && dLog < 0.3 ) //red to orange
      {
        fRed = 1.0;
        fGreen = ( dLog -0.2) * 5.0;
        fBlue = 0.0;
        fAlpha = 1.0;
      }
      else if ( dLog >= 0.3 && dLog < 0.4 ) //orange to yellow
      {
        fRed = 1.0;
        fGreen =(dLog-0.2) * 5.0;
        fBlue = 0.0;
        fAlpha = 1.0;
      }
      else if ( dLog >= 0.4 && dLog < 0.5 ) //yellow to green
      {
        fRed =1.0-((dLog-0.4)*10.0);
        fGreen = 1.0;
        fBlue = 0.0;
        fAlpha = 1.0;
      }
      else if ( dLog >= 0.5 && dLog < 0.6 ) //green to tourquoise
      {
        fRed =0.0;
        fGreen = 1.0;
        fBlue = (dLog-0.5)*10.0;
        fAlpha = 1.0;
      }
      else if ( dLog >= 0.6 && dLog < 0.7 ) //tourquoise to light blue
      {
        fRed =0.0;
        fGreen = 1.0-((dLog-0.6)*5.0);
        fBlue = 1.0;
```

```
    fAlpha = 1.0;
  }
  else if ( dLog >= 0.7 && dLog < 0.8 ) //light blue to blue
  {

    fRed =0.0;
    fGreen = 0.5-((dLog-0.7)*5.0);
    fBlue = 1.0;
    fAlpha = 1.0;
  }
  else if ( dLog >= 0.8 && dLog < 0.9 ) //blue to purple
  {
    fRed =(dLog-0.8)*5.0;
    fGreen = 0.0;
    fBlue = 1.0;
    fAlpha = 1.0;
  }
  else //purple to light purple
  {
    fRed = (dLog-0.8)*5.0;
    fGreen = 0.0;
    fBlue = 1.0;
    fAlpha = 1.0;
  }
  myScreen.fScreenArray[i] [j] [0] = fRed;
  myScreen.fScreenArray[i] [j] [1] = fGreen;
  myScreen.fScreenArray[i] [j] [2] = fBlue;
  myScreen.fScreenArray[i] [j] [3] = fAlpha;
}

 //-----------------------------------------------------------------

else if ( iColourType == 5 ) // all blues
{
  double dLog = log10( ( double )myScreen.fScreenHits[i] [j] ) / (
                           log10( ( double )iMaxHit ) ); //as in grayscale

  float fRed = 0.0;
  float fGreen = 0.0;
  float fBlue = 0.0;
  float fAlpha = 1.0;

  if ( dLog >= 0.0 && dLog < 0.1 )  //black to r=0.906, g=0.0, b=1.0
  {
      fRed =( dLog ) * 9.06;
      fGreen = 0.0;
      fBlue = ( dLog ) * 10.0;
      fAlpha = 1.0;
   }
  else if ( dLog >= 0.1 && dLog < 0.2 ) //.. to r=0.563, g=0.0, b=1.0
  {
    fRed = 0.906-((dLog-0.1)*3.43);
    fGreen = 0.0;
    fBlue = 1.0;
    fAlpha = 1.0;
  }
  else if ( dLog >= 0.2 && dLog < 0.3 ) //.. to r=0.345, g=0.0, b=1.0
  {
    fRed = 0.563-((dLog-0.2)*2.18);
    fGreen = 0.0;
    fBlue = 1.0;
    fAlpha = 1.0;
  }
  else if ( dLog >= 0.3 && dLog < 0.4 ) //.. to r=0.094, g=0.0, b=1.0
  {
    fRed = 0.345-((dLog-0.3)*2.51);
    fGreen =0.0;
    fBlue = 1.0;
    fAlpha = 1.0;
```

```
          }
          else if ( dLog >= 0.4 && dLog < 0.5 ) //.. to r=0.0, g=0.251, b=1.0
          {
            fRed =0.094-((dLog-0.4)*0.94);
            fGreen = (dLog-0.4)*2.51;
            fBlue = 1.0;
            fAlpha = 1.0;
          }
          else if ( dLog >= 0.5 && dLog < 0.6 ) //.. to r=0.0, g=0.469, b=1.0
          {
            fRed =0.0;
            fGreen = 0.251+((dLog-0.5)*2.18);
            fBlue = 1.0;
            fAlpha = 1.0;
          }
          else if ( dLog >= 0.6 && dLog < 0.7 ) //.. to r=0.0, g=0.655, b=1.0
          {
            fRed =0.0;
            fGreen = 0.469+((dLog-0.6)*1.86);
            fBlue = 1.0;
            fAlpha = 1.0;
          }
          else if ( dLog >= 0.7 && dLog < 0.8 ) //.. to r=0.0, g=0.875, b=1.0
          {
            fRed =0.0;
            fGreen = 0.655+((dLog-0.7)*2.2);
            fBlue = 1.0;
            fAlpha = 1.0;
          }
          else if ( dLog >= 0.8 && dLog < 0.9 ) //.. to r=0.0, g=1.0, b=0.937
          {
            fRed =0.0;
            fGreen = 0.875+((dLog-0.8)*1.25);
            fBlue =1.0-((dLog-0.8)*0.63);
            fAlpha = 1.0;
          }
          else //.. to r=0.0, g=1.0, b=0.780
          {
            fRed = 0.0;
            fGreen = 1.0;
            fBlue = 0.937-((dLog-0.9)*1.57);
            fAlpha = 1.0;
          }
          myScreen.fScreenArray[i] [j] [0] = fRed;
          myScreen.fScreenArray[i] [j] [1] = fGreen;
          myScreen.fScreenArray[i] [j] [2] = fBlue;
          myScreen.fScreenArray[i] [j] [3] = fAlpha;
        }
      }
    }
  }
}
```

Calc.h–file: variables and functions used for calculating chaos game, colour value, screen.

```
#include <GL/glut.h>



class CCalculation
{
   private:
      double dx;
      double dy;
      double dVx;
      double dVy;
      double dRadius;
      double dAngleTheta;
      double dSx;
      double dSy;
      int iMaxHit;

   public:
      CCalculation();
      double randomNum(double dMin, double dMax);
      void initCalculate();
      void actualCalculate();
      int convertToScreen(double dValue, int iScreenValue);
      void calculateHitsToColour(int iSizeWidth, int iSizeHight);

      //input by user
      int iIterations;
      int iPattern;
      int iColourType;


};
```

Screen.cpp–file: managing screen information.

```
#include "screen.h"

//--------------------------------------------------------------------------

extern int iScreenWidth;  //never go over 1000, see array size
extern int iScreenHight;  //never go over 1000, see array size

//--------------------------------------------------------------------------


screen::screen()  //constructor: set whole screen to black
{
   for(int i=0; i<1000; i++)
   {
      for (int j=0; j<1000; j++)
      {
         setPixelBW(i,j,BLACK);
      }
   }
}

void screen::drawScreen(void)  //exactly what it's named
{
   for(int i=0; i<iScreenWidth; i++)
   {
      for (int j=0; j<iScreenHight; j++)
      {
         glBegin(GL_POINTS);
         glColor4f(fScreenArray[i][j][0],
               fScreenArray[i][j][1],
               fScreenArray[i][j][2],
               fScreenArray[i][j][3]);

         glVertex2f(i,j);
         glEnd();
      }
   }
}

void screen::setPixelBW(int x, int y, int colour)  //kept concept from earlier test
{
   if(colour == WHITE)
   {
      fScreenArray[x][y][0]=1.0;
      fScreenArray[x][y][1]=1.0;
      fScreenArray[x][y][2]=1.0;
      fScreenArray[x][y][3]=1.0;
   }
   else if(colour == BLACK)
   {
      fScreenArray[x][y][0]=0.0;
      fScreenArray[x][y][1]=0.0;
      fScreenArray[x][y][2]=0.0;
      fScreenArray[x][y][3]=0.0;
   }

   fScreenHits[x][y]=0;
}
```

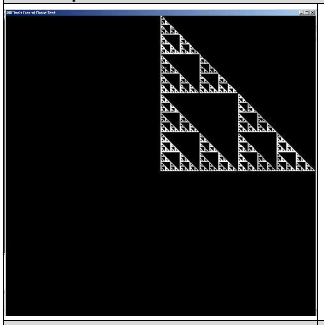
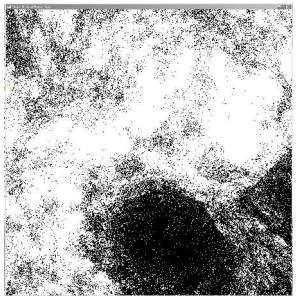
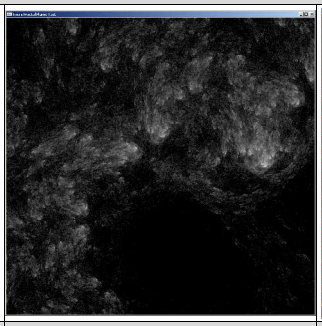
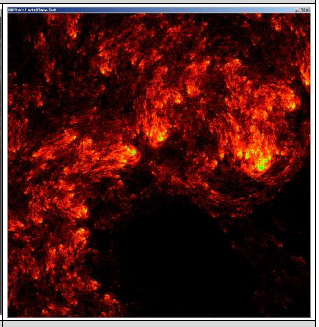
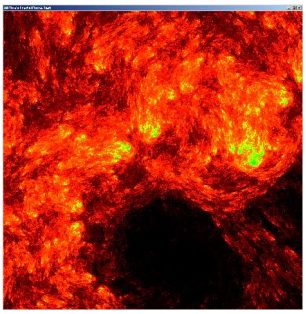
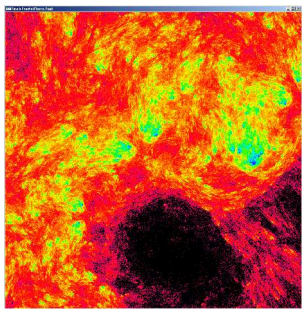
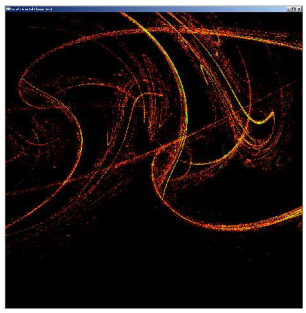
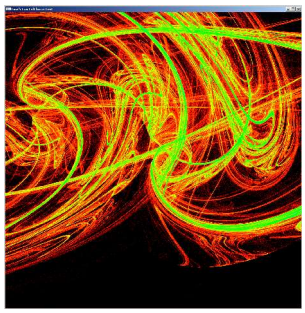
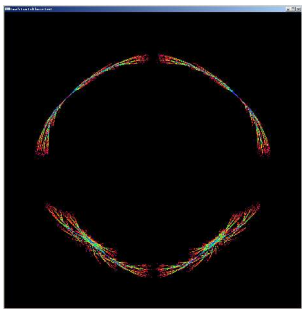
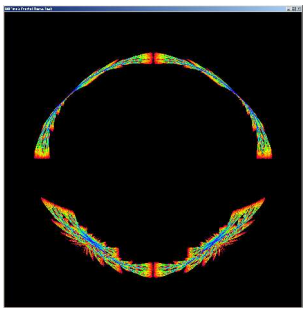
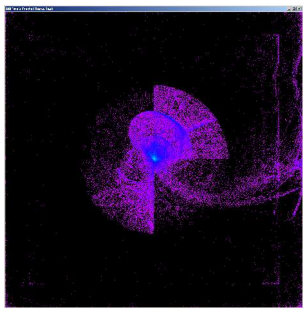
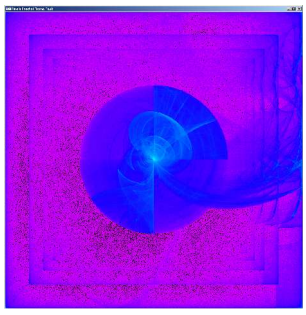
Screen.h–file: variables and functions used for displaying.

```
#include <GL/glut.h>

#define BLACK 0
#define WHITE 1
#define MAX_ARRAY_X 1000
#define MAX_ARRAY_Y 1000

//--------------------------------------------------------------------------



class screen
{

   public:
      screen();
      void setPixelBW(int x, int y, int colour);
      void drawScreen(void);

   public:
      float fScreenArray[MAX_ARRAY_X][MAX_ARRAY_Y][4];  //x, y and rgba,
         !!!array takes only constant number, can't use global screenSizes here
      float fScreenHits[MAX_ARRAY_X][MAX_ARRAY_Y];  //count how often the pixel
                                          is hit and make calculations on that number

};
```

John Macey's makefile, marked bold is the project specific data

```
#first we find what OS were on and define OSVER to allow the different versions of the code
OSVER := $(shell uname)
# on a mac so we define DARWIN
ifeq "$(OSVER)" "Darwin"
   ARCH=_DARWIN_
endif
#Linux box so we define _LINUX_
ifeq "$(OSVER)" "Linux"
   ARCH=_LINUX_
endif

#############################################################################
# INCDIR is the flag to set the default include paths for the system
#############################################################################
INCDIR = -I../include -I/usr/include

ifeq "$(OSVER)" "Linux"
   INCDIR+=
endif
ifeq "$(OSVER)" "Darwin"
   INCDIR+=-I/usr/local/
endif

#############################################################################
# CCFLAGS are the compile flags for compiling the code
# -g include debug info (for use with ddd or gdb) can be removed from exe using strip
# -Wall display all warnings
# -DUNIX - GraphicsLib define to set some UNIX options
# -funroll-loops unroll static loops (i.e) if loop has hard coded values these will be
#            replaced with more optimised procedural sections increases code size
#            and MAY increase speed
# -O3 turn on all optimisations
# Note :- for more options and flags type man g++ (there are a lot of arch specific options)
#############################################################################
CCFLAGS = -g -Wall -DUNIX -funroll-loops -O3 -DUNIX -lglut -lGLU -lGL
#############################################################################
# LIBS add library search path to the compiler
#############################################################################
LIBS=screen
ifeq "$(OSVER)" "Linux"
   LIBS =-L/usr/X11R6/lib
endif
ifeq "$(OSVER)" "Darwin"
   LIBS =
endif

#############################################################################
#############################################################################
# OBJECTS should contain a list of any object files to be created. These should be the same case
# as the corresponding .cpp files for the project. For example if the program has two source
files
# Main.cpp and DrawFunc.cpp the OBJECTS line should read OBJECTS= Main.0 DrawFunc.o
# if multiple lines of OBJECTS are required they should be seperated using a backslash as follows
#
# OBJECTS = Main.o (use a backslash here)
#        DrawFunc.o  note that the last line doesn't have a (backslash here )
# Note :-  can't actually put a single backslash in a Makefile as it messes up the parsing
#        but can put a double one to show what a single one looks like (\\) end rambling bit ;-)
#############################################################################
OBJECTS = main.o screen.o calc.o
#############################################################################
# XLIBS a list of the standard libs to include in the linking process
# to add libs use -l[LIBNAME] where LIBNAME is the name of the lib minus lib[LIBNAME].so
# e.g. to add libX11.so use -lX11
#############################################################################
```

```
XLIBS =
ifeq "$(OSVER)" "Darwin"
   XLIBS +=
endif

##############################################################################
# MATHS flag to add the default maths lib (some linux distros need this)
##############################################################################
MATHS =
# EXENAME flag to set the output executable name for the project. This is used with the
# -o flag of the linker as well as the make clean function to remove the exe
##############################################################################
LINK_TARGET = TinaFractal
##############################################################################
# this line links the created OBJECT files into the exe specified from EXENAME
# and includes the relevant libraries
##############################################################################
all   :  $(LINK_TARGET)
$(LINK_TARGET) : $(OBJECTS)
   g++ -o $(LINK_TARGET) $(CCFLAGS) $(LIBS) $(INCDIR)  $(MATHS) \
         $(OBJECTS)  $(XLIBS)
##############################################################################
# rule to build the .o files from the .cpp files
# this line will compile in turn each of the FILES from the OBJECTS variable (as a .cpp)
# into a .o file for the linking stage
##############################################################################
$(OBJECTS): %.o: %.cpp
   g++ -D$(ARCH) -c $(FLAGS) $(GRAPHICSLIB) $(CCFLAGS) $(INCDIR) $< -o $@

##############################################################################
# rule to remove and backup files, .o files and the exe for a clean build
##############################################################################
clean :
   rm -f *.o; rm -f $(LINK_TARGET);
##############################################################################
# end of Makefile
##############################################################################
```

Examples achieved with the above code by Tina Nischan			
			
Iterations: 1,000,000 Pattern: Sierpinski triangle Colour: black and white	Iterations: 1,000,000 Pattern: cloud (Draves) Colour: black and white	Iterations: 1,000,000 Pattern: cloud (Draves) Colour: grey scale	Iterations: 1,000,000 Pattern: cloud (Draves) Colour: traffic light
			
Iterations: 100,000,000 Pattern: cloud (Draves) Colour: traffic light	Iterations: 100,000,000 Pattern: cloud (Draves) Colour: rainbow	Iterations: 1,000,000 Pattern: swirl (Draves) Colour: traffic light	Iterations: 100,000,000 Pattern: swirl (Draves) Colour: traffic light
			
Iterations: 1,000,000 Pattern: Tina1 Colour: rainbow	Iterations: 100,000,000 Pattern: Tina1 Colour:rainbow	Iterations: 1,000,000 Pattern: Tina2 Colour: all blues	Iterations: 100,000,000 Pattern: Tina2 Colour: all blues