PH3170 - Fractal Visualisation Project Report

Guy Brown

December 12, 2018

Abstract

The aims of this assignment were to gain an understanding of fractal visualisation techniques and use these in practical applications. This was done by using the skills learned in the PH3710 C++ course to write a computer program which was able to read data in from a file, store the data within the program and use appropriate logic to create an algorithm to manipulate it. The altered data was written to a new file and visualised by the EasyBMP package.

1 Introduction

The aim of this project was to produce a C++ program to analyse and produce multiple geometries. The program enables the user to perform some more advanced studies on the nature of the structures that were produced. The program utilises the visualisation library EasyBMP, which produces a .BMP output of the pixels that are generated on a 2D image plane. An iterated function system (IFS) is defined as a finite set of contraction maps w_i for $i = 1, 2, \ldots, N$. Each contraction map has a contractivity factor between 0 and 1. The system maps a compact metric space onto itself. [1]

2 The Chaos Game

The Chaos Game is a simple way of producing fractals. It is used to produce an image based on an algorithm that takes into account the number of vertices of the desired polygon and the distance of the current point on the image plane to a randomly selected vertex for a finite number of iterations.

Initially, the code was written to perform this operation inside a triangle. The three vertices were pre-defined and their coordinates were stored. One of these vertices was selected as the starting point. Using a random number generator, one of the three vertices was selected. The average coordinates of these two points were calculated in order to place the next pixel half way between them. This process was repeated for a sufficient number of iterations to enable a structure to be resloved.

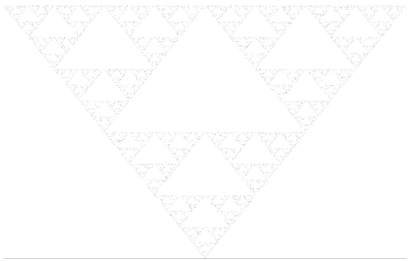

Figure 1: Sierpinski triangle for 10,000 iterations

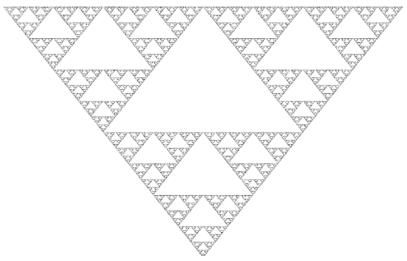

Figure 2: Sierpinski triangle for 100,000 iterations

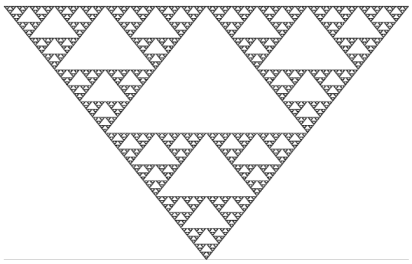

Figure 3: Sierpinski triangle for 1,000,000 iterations

The structure that is resolved in Fig. 1, 2 and 3 is called the Sierpinski triangle. It is an example of an attractor, which is the final state that a dynamical system such as an IFS will approach with increasing numbers of iterations. A structure begins to become visible after about 10,000 iterations. The clarity of the structure increases significantly with each order of magnitude increase of the number of iterations.

Once the Sierpinski triangle had been established, the code was generalised to allow the generated polygon to be user defined, for between 3 and 6 vertices. The program allows the user to produce IFS images of triangles, rectangles, pentagons and hexagons. The coordinates of each polygon's vertices are defined depending on the number of vertices. This is so the output

is always a regular polygon. One problem with the inclusion of this feature is that altering the code for polygons with even more vertices could be messy, as each shape is defined in the code by a set of if/else if statements which has to be expanded each time a vertex is added. This could however be generalised with a loop, and a formula would need to be used to calculate the coordinates of the vertices for each case.

The user can also choose the factor by which the new point moves towards a selected vertex. Playing around with this feature allows several different patterns to be formed within each polygon. The range for the step size was set to be only between 0.2 and 0.5 Outside of this range the IFS struggled to generate a visible attractor.

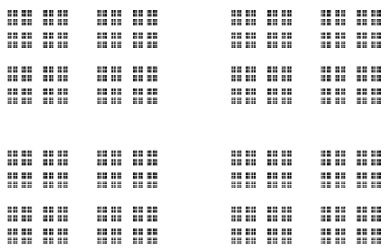

Figure 4: Rectangle with step size = 0.4

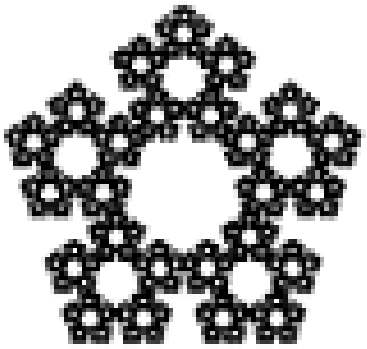

Figure 5: Pentagon with step size = 0.4

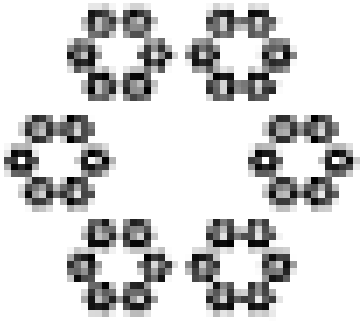

Figure 6: Hexagon with step size = 0.3

The final generalised feature of the code was the enablement of the user to choose the probability of selecting a vertex, where the effect is tonal grey-scaling in the image. This occurs because there is a segment of code that loops over all the elements of the 2D pixel array to work out which pixels were selected most often, and therefore appear the darkest. The entire image is then re-normalised based on the difference between the lightest and darkest pixels to ensure the best contrast. Therefore, the pixels surrounding one of the vertices can be chosen to appear lighter or darker. Choosing a vertex probability equal to $1/n$, where n is the number of vertices, will give no effect as the probabilities of any of the remaining vertices being selected are equivalent.

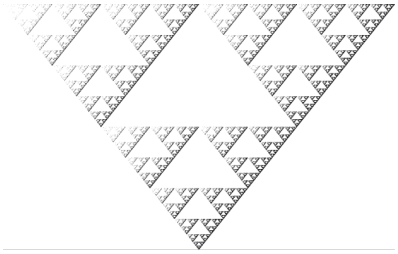

Figure 7: Triangle with vertex probability weighting = 0.1

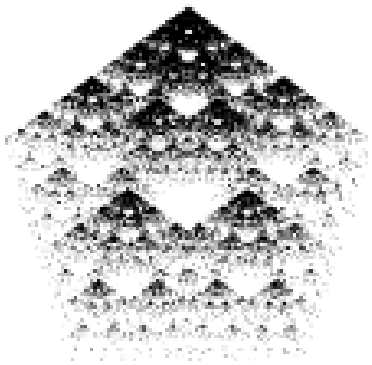

Figure 8: Pentagon with vertex probability weighting = 0.8

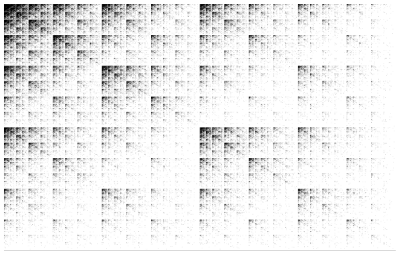

Figure 9: Rectangle with vertex probability weighting = 0.7

3 Generalised Chaos Game

Some more sophisticated transformations can be used to rotate, scale and translate each point. Randomly applying one transformation in a set with each iteration is a more generalised technique used to generate an IFS attractor. These transformations are known as affine transformations. Such transformations preserve co-linearity under transformation[2].

In this program, a specific subset of affine transformations is used. The members of this subset of transformations are called contraction mappings. Such a transofrmation reduces the size of a transformed object relative to the original. An affine transformation can be expressed as

$$w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

The matrix $\mathbf{A}$ $(a_{11} \ldots a_{22})$ represents rotation and scaling and matrix $\mathbf{B}$ (b_1, b_2) represents translation. The requirements for an affine transformation to be valid are [2]

$$a_{11}^2 + a_{21}^2 < 1$$

$$a_{12}^2 + a_{22}^2 < 1$$

$$a_{11}^2 + a_{12}^2 + a_{21}^2 + a_{22}^2 - [det(\mathbf{A}]^2 < 1$$

The program reads in a set of transformations and their probabilities from a text file. It then checks whether the transformations are valid by computing the three requirements. If all the transformations are valid, the mappings are applied to a point on a plane in an iterative process. The code works for an arbitrary number of transformations, i.e. however many lines are in the file. A transformation is added by appending another line with 7 values to the file. These values are the numbers making up matrices **A** and **B** and the probabilities associated with the relevant transformation. The probability of each transformation being selected can be altered by changing the 7th value on any line in the input file. The number of iterations performed is determined as a user input using std::cin. The attractor formed by 9 arbitrary transformations is shown for different numbers of iterations in Fig. 10, 11, 12. The structure becomes clear after about 1,000,000 iterations. A small group of the same pixels have to be selected many times for an image to be dark enough to see.

Figure 10: Randomly applying the arbitrary transformations for 10,000 iterations

Figure 11: Randomly applying the arbitrary transformations for 100,000 iterations

Figure 12: Randomly applying the arbitrary transformations for 1,000,000 iterations

Changing the probabilities of different transformations being selected allow different versions of the same image to be formed. Certain pixels are selected more often than others depending on how many times each transformation is selected. Therefore, the darker and lighter parts of the image appear in different places when the probabilities are altered. The algorithm used in this section made it difficult to keep the image within the array EasyBMP uses to define the image. To amend this, the results of the algorithm were stored in a separate structure and then mapped onto the 2D grid after re-scaling the pixel coordinates.

When trialling random transformations, it was difficult to come up with a set that produced a

distinguishable image within a reasonable number of iterations. This may have been because the transformations caused the pixels to be placed outside of the 2D grid very frequently. Because the area of the 2D grid over which pixels could be selected was greater, there was a lower probability of any pixel to be selected more than once. Consequently, there was little contrast between regions of the image where the pixels were either selected or not selected when the image was re-scaled.

Being given the IFS code for the attractor for Barnsley's fern, the transformations and their probabilities were read in from a text file to produce the output in Fig. 13.

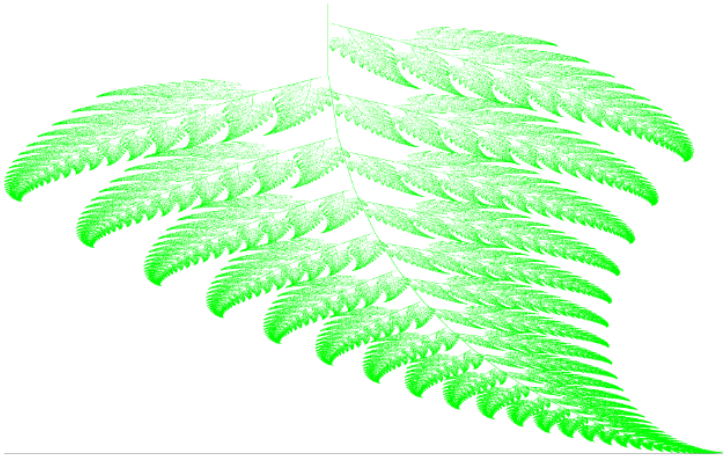

Figure 13: The attractor produced by the IFS code for Barnsley's fern over 1,000,000 iterations.

4 The Collage Theorem

The Collage Theorem states that to find the attractor for an IFS which looks like another image one must find the set of contraction mappings whose union, or collageis near that of the original image [2]. Applying this concept to the IFS program already written, a Maple leaf was modelled by the IFS code. An image of a leaf was read into the program, and a set of 6 transformations were applied to create smaller images of the original leaf. The transformations were modified so the smaller images were mapped over the area of the original, to estimate which transformations would provide best representation of the initial object, seen in Fig. 14. The corresponding transformations were then used in the IFS code to produce a rough model of the original maple leaf image.

Figure 14: Six transformed Maple leaf images positioned on the orignal image, using the collage theorem

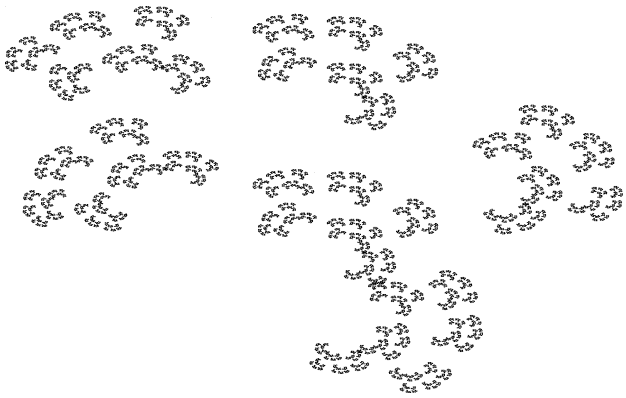

Figure 15: Attractor produced by IFS code with ten transformations, modelled on the Maple leaf

The reproduced image in Fig 15 has an overall shape that is representative of the original, in the sense that the five protrusions are somewhat distinguishable. In order to produce a uniform image, the propability of each transformation is equivalent, at $1/n$, where n is the number of transformations. Changing the probabilities can make different portions of the reproduced image appear lighter or darker, where the regions corresponding to the higher-weighted probabilities will appear darker. Applying more transformations will produce a more accurate reproduced image. Covering a larger portion of the original image's area with a greater number of smaller versions of the original will give a better set of transformations to use in the IFS. As the transformed images get smaller and greater in quantity, they converge to the size and orientations of singular pixels. The original image could effectively be mapped onto itself if the number of transformations is equal to the number of pixels in the image, and each transformation is scaled to produce an image that is the size of a single pixel.

Figure 16: Ten transformed Maple leaf images positioned on the orignal image, using the collage theorem

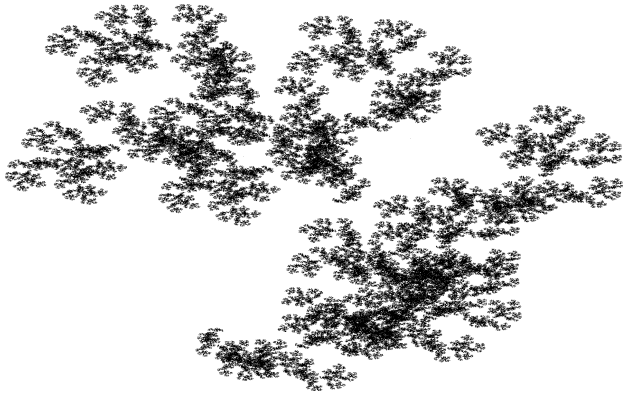

Figure 17: Attractor produced by IFS code with ten transformations, modelled on the Maple leaf

Fig. 17 shows the attractor for the ten-transformation IFS code. Visually it is a better represnetation of the original image than Fig. 15, but it is by no means a perfect depiction. It is not easy to determine the best positions of the smaller images to produce a good IFS approximation. Some more time for trial and error would have allowed the production of a more accurate attractor. On the EasyBMP grid, the origin is at the top-left corner. This means that the y-axis is inverted, so initially it took a little while to work out how to position the images when adding the translation vector into the transformations. Furthermore, the smaller images couldn't be translated by simply shifting the pixels in the transformation. Instead, the translations had to be defined by setting the boundary for the minimum position of the x and y coordinates on the grid. This was however not a problem when using the corresponding IFS code.

5 Conclusions

The aims of this assignment were to gain an understanding of fractal visualisation techniques and demonstrate them by using the C++ skills learned during the course. An understanding of iterated function systems and affine transformations were gained along with knowledge of

the conditions that makes valid contraction mappings. Matrix algebra was implemented by a combination of computing methods including control structures, loops, functions and standard template library containers. User inputs were used to control some of the parameters, data was read in from files, manipulated and written to a file containing information to generate 2D images. Exception handling was used for invalid inputs and files. Overall, the aims of the project were met.

References

[1] WolframMathWorld http://mathworld.wolfram.com/IteratedFunctionSystem.html

[2] Royal Holloway, University of London Physics Department, *PH3170 (C++ Programming) Fractal Modelling Project* (2018).