PH3170 Fractal Modelling Project

Nickolas Theodoulou - 100794934

December 2016

Abstract

Using C++ and the EasyBMP graphics library, fractals can be observed. This can be achieved by using simple algorithms to create Chaos Games such as a Sierpinski Triangle. This can then be generalised to an n-sided polygon and then further generalised to create even more complicated structures such as Barnsley's Fern. This was all achieved by the code I created. It is also possible to use the Collage theorem to create an image using a set of contraction mapping whose union is near the original image which was also attempted using a maple leaf.

1 Introduction

Fractals are structures that appear throughout nature from seashells all the way to the structure of human lungs. Most fractals involve chance and both their regularities and their irregularities are statistical [1]. They all have a never ending pattern that repeats itself on different scales otherwise known as being self-similar. This makes it relatively easy to create fractals such as the Sierpinski Triangle by repeating a process over and over again in a for loop. This makes them relatively easy to model using a computer using the concept of iterated function system (IFS).

For this project, a C++ programme was created that was able to generate a Sierpinski Triangle. I then generalised this code to allow me to vary the number of sides of the polygon, the factor by which to move towards a selected vertex and the probability of selecting a vertex. A second, more general programme was then created based on the first one which allows the user to define an arbitrary number of contraction mappings. This was also generalised to allow the user to select the number of transformations to have available, the probability of each transformation being selected and the number of iterations to be performed. Barnsley's Fern was then created using this code.

For the code submitted in the zip file to run correctly, all the cpp files created by myself and the EasyBMP library included must be added to the directory of a project. Then, for a given code

such as *chaosgame.cpp* to compile, all the other files not in the EasyBMP library must be excluded from the project. It is also important to note that *advancedchaosgame.cpp* will not run without the $IFSFern.txt$ file which defines the transformations used within the code.

2 The Chaos Game

2.1 The Sierpinski Triangle

To generate a Sierpinski Triangle, three vertices are defined and then a random point within these three vertices is selected. This was done by putting the x and y co-ordinates into two separate arrays and iterating over them using the given algorithm to generate a Sierpinski Triangle. As each x co-ordinate corresponds to a value from 0 to 1920 and each y co-ordinate responds to a value from 0 to 1200, each new point generated corresponded to a pixel on the screen, so within the iteration the new point in terms of the co-ordinates could be set to a black dot on the screen which would produce a fractal after a large amount of iterations.

2.2 Generalising The Sierpinski Triangle

To generalise the code allowing the user to vary the number of sides of the polygon, the co-ordinates were placed in vectors instead. This made it possible to select any element within a vector randomly as the probability of selecting a vertex could be defined as $1/vector.size()$ making it possible to generate an n-sided polygon by adding more elements in the vector for the x and y co-ordinates. This allows the user to generate a variety of shapes such as pentagons and hexagons.

To vary the factor by which to move towards a selected vertex, the factor by which to move to the selected vertex was replaced by a vector. This vector contained elements equal to the number of vertices allowing the distance to be moved to a selected vertex to differ from any other. This allows various other patterns to be created. An example of this is a fractal that resembles a snowflake. This can be achieved by setting the vertices as the points on a regular pentagon and setting the distance between a vertex and the new point is changed to 1/3 as seen in Figure 1.

To vary the probability of selecting a vertex, a forth vector that contained the probability of selecting the i^{th} vertices was created. The random number generated to select a vertex was modified to equal the sum of the vector containing the probabilities. This allows a cumulative probability to be defined within the iteration loop so that the i^{th} vertices corresponds to the i^{th} probability correctly. Varying

this probability isn't particularly useful in most instances as it makes the algorithm to generate fractals such as the Sierpinski Triangle less efficient as it takes more iterations to randomly make the correct pixels black.

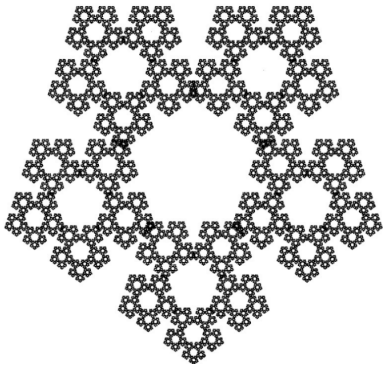

Figure 1: Fractal generated from a regular pentagon and setting the new point to a distance of 1/3

3 Generalised Chaos Game

The Chaos game can be further generalised to create even more complicated fractals.An interesting characteristic of this system is that no matter what initial image we begin with, the iterates of the system will always converge to the same final image [2]. This can be done by using Affine transformations which preserve co-linearity under transformation. An affine transformation w is expressed as:

$$w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{1}$$

where x and y are the co-ordinates and the elements with subscripts represent transformations and translations which are carefully defined to create fractals. To create Barnsley's Fern, contraction mappings are used which are transformations that reduce the size of the transformed object relative to the original. It is possible to check if the maps are valid by creating an if condition that returns 0 if the conditions defined in the script are not met and thus terminate the programme before running into errors. A convenient way to define the contractions is to read in the elements $a_{11}...b_2$ from a text file and then put all the elements as well as the probability of selecting a transformation into separate vectors. This allows the user to define as many transformations as desired by adding more rows in the text file. It is also possible for the user to change the number of iterations.

The greatest challenge faced in getting the code to work correctly in producing Barnsley's Fern was scaling up the x and y values to fit on the canvas. The first step taken to achieve this was to iterate over

the affine transformations and their associated probability and collect all the x and y co-ordinates in two separate vectors. The max and min values of the vectors were then found. From these four values, an appropriate scale factor can be determined by using: $scalefactor = x_{pixel}/x_{max}$ and a similar equation for the y co-ordinates. The smalles value from these two is then chosen as the scale factor to enlarge the image without distorting it. If x_{min} or $y_{min} < 0$ then a translation: $x_{min} \cdot scalefactor$ or $y_{min} \cdot scalefactor$ is added to the new co-ordinates to account for any negative values. This algorithm was used to produce the Barnsley's Fern as seen in Figure 2.

Figure 2: Barnsley's Fern generated from generalised chaos game

4 The Collage Theorem

Creating code that could reproduce an image of a real world object was a large challenge faced in this project. The EasyBMP documentation had an example of how to access a pixel within a BMP image which was then generalised to access any point within the image. As there is a white background in the given maple leaf image, the code was then generalised to colour all the pixels that were not white red to ensure that the correct pixels can be selected. This was possible as white has an RGB colour of red: 255, green: 255, blue: 255 so a simple if condition was sufficient to access the coloured pixels in the original maple leaf image. After failed attempts of generalising this code to make use of the advance chaos game code, placing the Sierpinski Triangle on top of the maple leaf was attempted and worked as seen in Figure 3. This code will also be submitted.

Once this was created it gave me a general idea on how to edit the value of pixels. Contraction

Figure 3: Sierpinski's Triangle generated over Maple leaf

mappings of the maple leaf covering most of the area of the maple leaf were then created by trial and error as seen in Figure 4. This was created by finding 6 contraction mappings. Due to time constraints the code was not generalised further. The contractions created somewhat resembled the original image once placed on top of each other and it became increasingly more difficult to create mappings that resembled the smaller details of the original image.

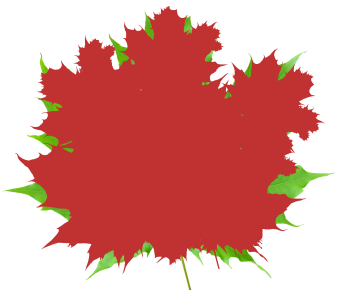

Figure 4: Contraction mappings over Maple leaf

5 Fractal Image Compression

A useful advanced technique combining the concepts that have been discussed thus far is fractal image compression. This technique is extremely useful as an uncompressed 512 pixels by 512 pixels grey-scale image takes up to 260,000 bytes. This is a significant amount of data that could take up to a minute for an average PC to load. However, if the image is compressed, the time taken to load the image can be significantly reduced.

One way to achieve this with a black and white image is to split the image into a number of square segments called range blocks and split these into even smaller segments known as domain blocks. The range blocks can then be iterated over using different transformations and the result that best matches the domain block can be stored. If a good match is found then the transformations can be stored as

well as the domain block to apply them to. This method of storing information requires much less memory.

6 Conclusion

The code created for the chaos game and generalised chaos game both have all the functionality described in the handout and work relatively well. An improvement for the generalised chaos game would be to allow the user to choose between reading a text file or define the transformations within the code itself. The scale factor logic could also be made more robust as there were a few edge cases that would cause the programme to crash which were solved by multiplying the scale factor by 0.9999. If possible a more reliable method could be implemented.

The Collage Theorem code is also somewhat incomplete due to time constraints. If time were not a factor, the code would have been generalised in the following way: first creating a separate vector for all the elements that transform the coordinates. This would then allowed the user to read in the elements from a text file the same way the generalised chaos game did. The next step would be to have a range of mappings predefined in a text file. This would allow the user to have the option to add a mapping one by one if it was suitable and ignore it if it wasn't suitable. The useful mappings could then be stored into a separate text file for later use. This could then be implemented for other images however, it would be extremely unlikely for the mappings to be accurate as they are essentially being guessed. A way of automating the mappings is possible using fractal image compression. Overall, I feel the code created especially for the chaos game is robust however, given more time I would have liked to attempt the fractal image compression section to further understand a real world application of fractals.

7 Bibliography

References

[1] Mandelbrot, B. B. *The Fractal Geometry of Nature.* New York, NY: W. H. Freeman and company, pp.1.

[2] Lackroix, B. (1998). *Fractal Image Compression.* Ottawa, National Library of Canada, pp.10.