THESIS

Olugbenga Tolulope Oluwaseun

Debrecen

2017

University of Debrecen

Faculty of Informatics

Visual Representation Of Simple Simulated Processes

Supervisor

Dr. Imre Varga

Associate Professor

Student

Olugbenga Tolulope Oluwaseun

Computer Science Engineering

Debrecen

2017

Table of Contents

1 Introduction

I got to my 5th semester of my study in the University of Debrecen, and it was time for me to choose a Thesis topic, as this is a criteria needed before one could graduate from this great university. There were a list of topics uploaded to my faculty's website; students are expected to choose from this list, there were lot of topics with various lecturers for various departments as well.

After crosschecking the list over and over again, I found no topic or research work that caught my interest, I wanted to do something more, something that's would bring joy and excitement, I wanted something that could be implemented with the use of C programming that I learnt while offering the course, programming languages 1 in my 2nd semester.

After the decision was made by me to do something using C programming, I knew that the only way this can be achieved is by getting a supervisor that offers a course involving C programming. I had a couple of lecturers to choose from, I decided to start with the lecturer that took the course "Introduction to Informatics" in my first semester Bíró Piroska, She is one of my favorites and a very good lecturer at that, After my email was sent, I was informed that she has not done Thesis for an international student.

I emailed another of my favorite lecturer, who took me programming languages 1, and developed my interest in programming, Dr Kocsis Gergely, I was informed that he would not be available for Thesis, as he hasn't had a foreign student doing their research work with him, I decided to drop my quest with him.

My last option was my lecturer that took me Hardware programming 1 and 2 in my 4th and 5th semester respectively, Dr. Varga Imre. I told him, I would love to do a research work that had to do with the implementation of my knowledge in C programming as this was my favorite programming language, he promised to get back to me after he has gotten a topic. Few days later I got his email that he would love to set up a meeting with me and discuss about the topic he had in mind. He informed me that we could do something with the use of bmp files, Pov-ray and some other sophisticated applications.

I was skeptical at first, as everything seemed like a new thing to me, I didn't have an idea about what I was doing, I had to learn a lot of new things, both in programming and about bmp files, after he explained the basic concept of the topic, it looked very challenging. I was given some days to think about my final decision for the topic. After a few days I told Dr Varga that I accept the topic, the name of the chosen topic is "Visual representation of simple simulated processes".

1.1 Overview of The Topic

The topic is very unique, in a sense that it could be similar to something that we see and use every day but we don't take note of them; Have you ever taken some time to wonder how they are created, how they come to play. Take for example a gif message you send on a Facebook, this is just a simple animation that comes through the combination of multiple images or multiple pictures. Or let's take for example a moving point like particle; this can be done by taking snapshots of the movement and making a video out of it.

I chose this topic because of the challenge involved, I knew it would require a lot of hard work and dedication to see it through; there were entirely new concepts to be learnt and implemented. At the time the topic was chosen, among all the elements to be used in the research work, I only knew a bit of the C programming.

This topic is not a popular topic people talk about, due to the fact that they are too simple and well understood problems. They were studied and simulated in the 80's, and I want to implement them to serve as a base of graphics according to my own visual expectations.

I wanted to create beautiful animations as well as simulations using Bit-Map pictures and using Pov-ray to illustrate the 3d representations in beautiful colors. My research work is divided into two separate parts, with little similarities between them. These topics are Random Walk and Diffusion Limited Aggregation respectively.

Random walk describes a physical process, which is the Brownian motion. The term "random" comes from the fact, that there is no way to predict the next step from knowledge of

the previous step, and the term “walk” because it takes a series of “steps” for the particle to move from here to there.

Diffusion-limited aggregation (DLA) is the process whereby particles undergoing a random walk due to Brownian motion cluster together to form aggregates of such particles, due to these particles moving in random motion, beautiful aggregates can be formed.

2 Simulated Processes

In this chapter, we are going to go through the definition, theoretical background, literature, history, mathematical formulas for both the “Random Walk” and the “Diffusion Limited Aggregation” respectively. But first we are going to talk briefly about the "Brownian Motion".

2.1 Brownian Motion

Brownian motion can be taken as a classic example of the random walk theory, let’s talk about how the Brownian motion came about. Brownian motion also known as pedesis can be defined as the random motion of particles suspended in a liquid or gas, this results from their collision with the fast-moving atoms or molecules in the gas or liquid.

This transport Phenomenon got it name after a botanist called “Robert Brown”, in 1827 while he looked through the eyes of a microscope at particles trapped in cavities inside pollen grain of water, he noticed that the particles were moving through the water, at this point he was not able to state precisely, what mechanisms could have caused the motion.

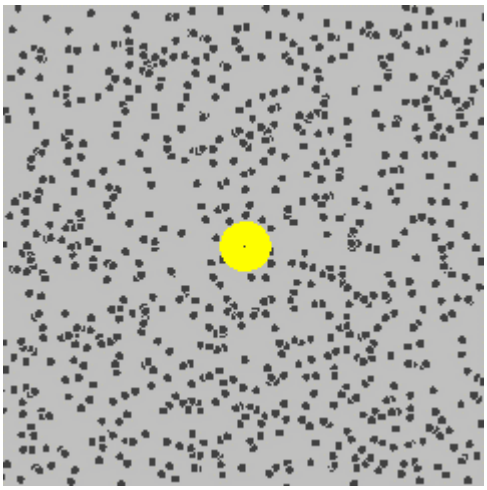

Figure 1:- A snapshot of the Brownian motion of a big particle (dust particle) that collides with a large set of smaller particles (molecules of a gas) which move with different velocities in different random directions.

In 1905, the well know mathematician and physicist “Albert Einstein” published a paper which explained in detail, that the motion Brown had observed was caused by a result of the pollen being moved by individual water molecules. This created solid evidence that, molecules and atoms actually exist, which Jean Perrin verified in 1908.

2.2 Random Walk

A random walk is an object, also known as a random process, which describes a path of random steps, on a mathematical space (e.g. integers). Imagine a completely drunk man, trying to get home with no one to support him, or hold his hands, he would walk in series of random steps, random walk can be seen in this way as well.

An example of random walk, could be the movement on number line, it’s initial position starts from 0 and at each step, there is a possibility of moving left(-1) or right(+1) with equal probability. Other example could be the path of a molecule traveling in liquid or gas, the price of a fluctuating stock.

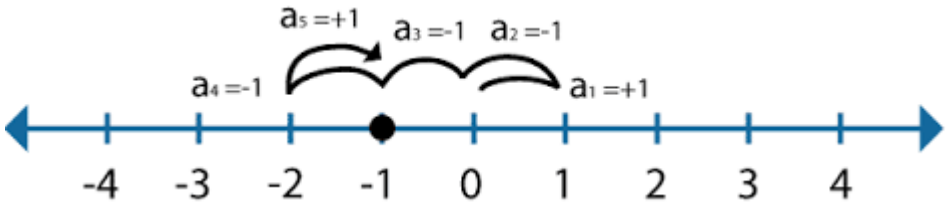

Figure 2:- Shows a number line, with equal probabilities to move left or right

The random walk can be seen in stock market, stock price changes are independent, having a common distribution, this means the previous movement of a stock price doesn’t determine the future movement/changes in the stock price or market, and therefore we could say the stock market applies the theory of “Random Walk”.

2.2.1 One-dimensional random walk

Let’s take a chalk, and draw a number line similar to the one above, we stand at a x=0 on the number line, we toss a fair coin with equal probability of ½, if we get an head, a step of length d

(e.g. 1), is taken to the right, at this point x becomes the value of d (x=d). While if a tail comes up, a step of same length d is taken to the left, x now becomes –d. There is an equal probability of ½, that you are standing on either x=d or x=-d, to move to a new position the whole process is repeated again.

When a coin is tossed, a head means a step of length d is moved to the right and a tail means a step of length d is being moved to the left. According to the below figure, the probability of going to either position x=2d or x= -2d, is ¼, but the probability of being back at the original position is ½. As the process is repeated over and over, because it is a fair coin, we cannot predict where you would be standing on next, so we can therefore say that you are taking a random walk, which is shown on the binary tree in the figure below, children to the right refers to Heads, and to the left refers to Tails.

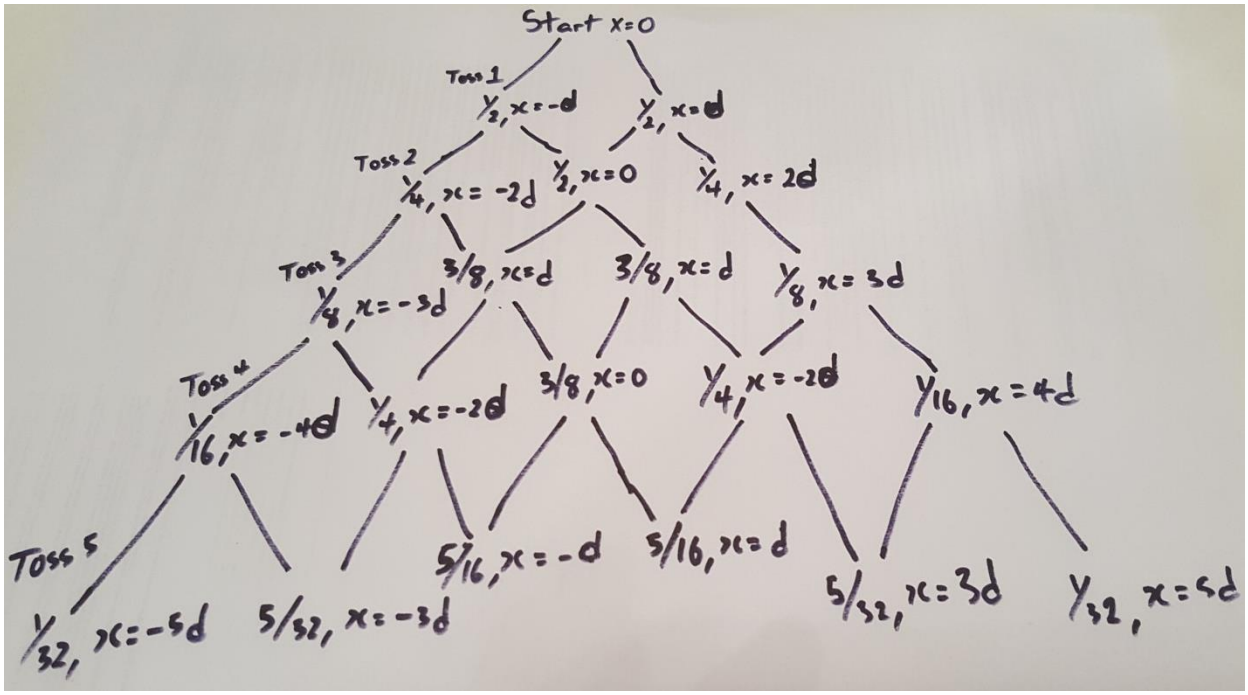

Figure 3:- Shows a binary tree, with probabilities for movement after 5 tosses of a coin

Starting from x =0, what could be said after N tosses of coin, the maximum location you could reach is x=Nd of x=-Nd, it's very rare to come even close to this value, because we are in a

system obeying random walk. This experiment can be repeated severally with a fair coin; at every repetition we would most likely follow a different route, because the steps are random.

2.2.2 Two-dimensional random walk

The two dimensional random walk, unlike the case of the number line, we have four possibilities of movement, up, right, left and down. Examples of 2d object are squares, rectangles, triangles, circles, and many more, these object can be drawn using only the x and y axis.

Imagine you are in the middle of the big city of Budapest, you are coming back from a party, and you are drunk but trying to find your way home. You have four directions you could move in, because you are drunk the probability of you to move in either directions, can be assumed to be ¼, either you know your way home or not, you would have problems getting home(that's if you actually do get home), the steps taken would be a series of random steps.

A similar case could be a blind man, finding his way home. The probability of getting to the final destination, is a really small value, and this value reduces and could become next to nothing, when the dimensions gets higher, this means there would be more directions/possibilities for movement, this slows or reduces the chances of finally getting home.

2.3 Diffusion Limited Aggregation (D.L.A.)

According to Wikipedia "Diffusion-limited aggregation (DLA) is the process whereby particles undergoing a random walk due to Brownian motion cluster together to form aggregates of such particles". The terms "Diffusion" is used because the aggregates forming the structure, first move about in a random motion before clustering themselves to the previous particles.

It's referred to as limited ("Diffusion-Limited") because there is always one particle moving in the system, the structure grows one at a time, so the particles do not touch each other. There are various ways of implementing this theory using a computer system, Paul Bourke wrote an article listing some examples of the diffusion limited aggregation, one of them he calls it's the point attractor, which he refers to as the most common implementation used.

In the implementation of the point attractor, he starts with a pure white image, with a single black pixel in the center, he introduces new points at the borders, this points move randomly until they get close enough, to stick with an existing black pixel. The figure below shows an image to describe this implementation, if a point during its random movement reaches an edge of the image, there are two possibilities, it either bounces off that edge or the image is periodical, which means that the point comes out from the opposite edge.

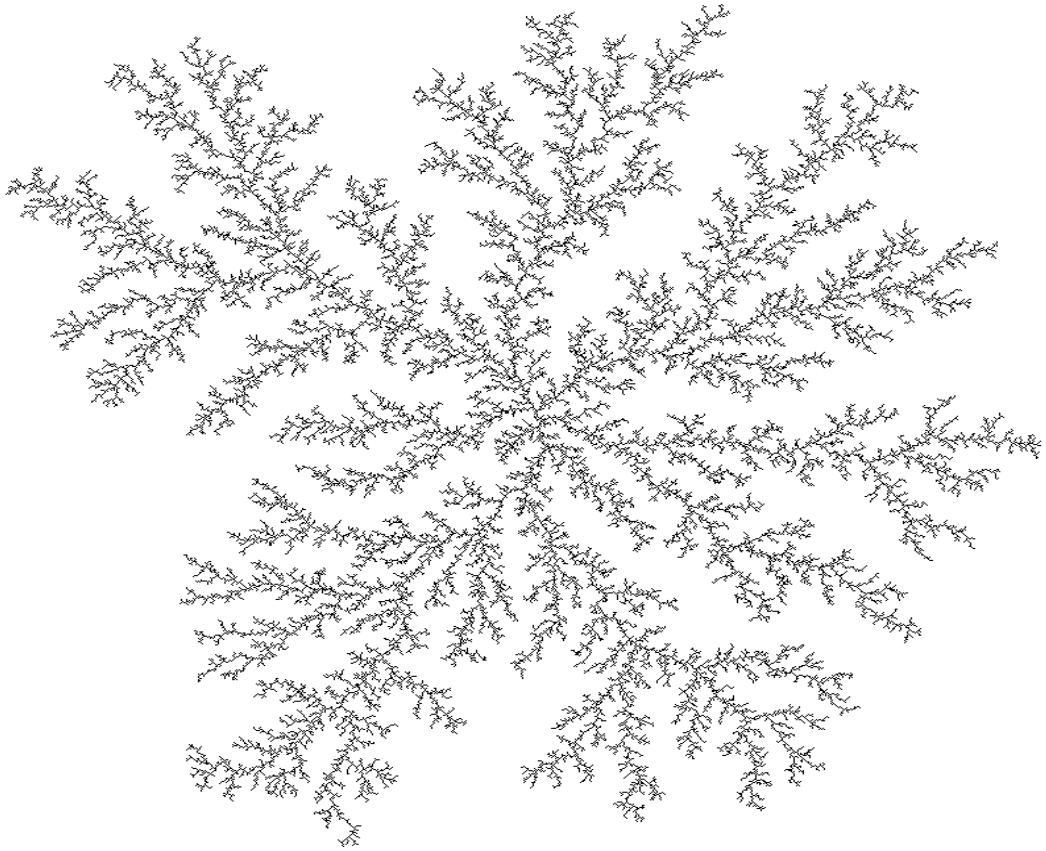

Figure 4:- Shows a beautiful structure created, by the aggregation of several particles to the center particle

New points can be initialized anywhere on the image, but it's advisable not to initialize them close to the borders, only if there is a significant difference. 3d representation follow the same

rules as the 2d, just we would have more directions for random walk, and some other little differences.

The 2d implementation in a later chapter would be more like a line attractor on the x axis, all other points attach to this line or other previous elements which have been attached earlier; the 3d representation would be more like a square on the x and z axis, all other points attach to this square or previously attached squares, we would speak more in detail on a later chapter.

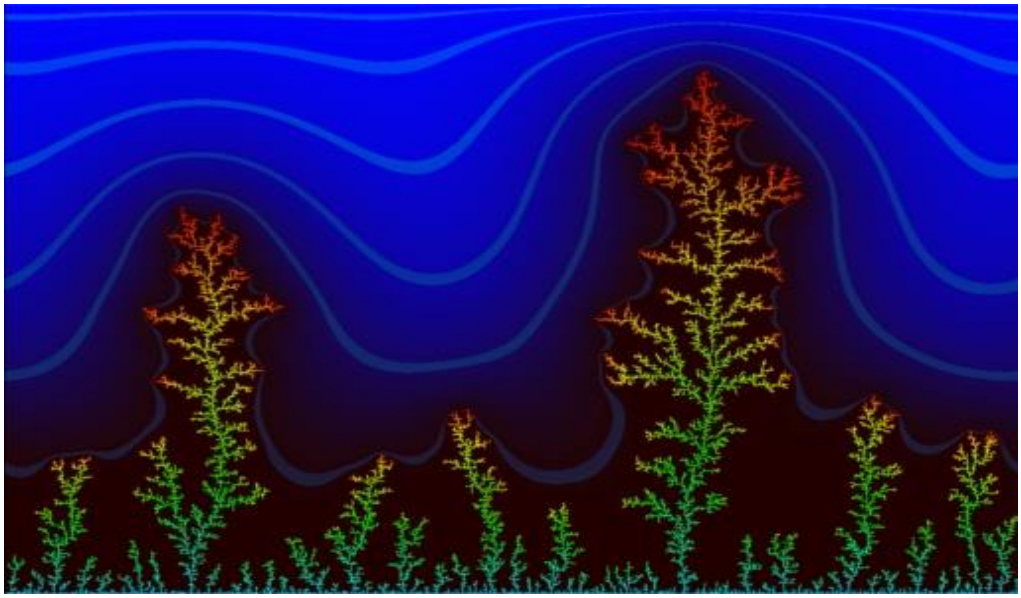

Figure 5:- Shows an example of a line attractor

3 Bit Map (BMP)

Firstly, we have to look at and understand the concept of bmp image files. According to Wikipedia “The BMP file format, also known as bitmap image file or device independent bitmap (DIB) file format or simply a bitmap, is a raster graphics image file format used to store bitmap digital images, independently of the display device (such as a graphics adapter), especially on Microsoft Windows and OS/2 operating systems”.

3.1 Applied BMP Format

The figure below shows the structure of a standard BMP format, for all implementations in this chapter, the Linux operating system is being used, Ubuntu to be precise. The first section of the bmp format is called the bitmap header, which is 14 bytes in length, these block of bytes are used to identify the file, most applications read this block first to ensure it’s a bmp file and the file is not damaged. The bitmap header consists of the Signature, which contains the character ‘B’ followed by the character ‘M’ ASCII encoding; these characters are 0x42 and 0x4D respectively in hexadecimal.

The next four bytes is the size of the bmp file, the four bytes following the file size are the unused bytes, usually zero, but the actual value could depend on the application that is used to create the image, another 4 bytes is been allocated to the pixel array offset, which is the starting address where the bitmap image data (pixel array) can be found, the value is usually 54. All integer values are stored in little-endian format, which means the least-significant bit comes first.

Bitmap header	Signature ("BM")	2 bytes
	File size in bytes	4 bytes
	Unused (0)	4 bytes
	Pixel Array offset (54)	4 bytes
DIB header	DIB header size (40)	4 bytes
	Image width in pixel	4 bytes
	Image height in pixel	4 bytes
	Planes (1)	2 bytes
	Bits/pixel (24)	2 bytes
	Compression (0)	4 bytes
	Image size (0)	4 bytes
	Horizontal pixel/meter (5900)	4 bytes
	Vertical pixel/meter (5900)	4 bytes
	Colors in palette (0)	4 bytes
	Used palette colors (0)	4 bytes
Pixel Array	Bottom row first pixel Blue	1 byte
	Bottom row first pixel Green	1 byte
	Bottom row first pixel Red	1 byte
	...	
	Bottom row last pixel Blue	1 byte
	Bottom row last pixel Green	1 byte
	Bottom row last pixel Red	1 byte
	Padding	1-3 bytes
	...	
	Top row first pixel Blue	1 byte
	Top row first pixel Green	1 byte
	Top row first pixel Red	1 byte
	...	
	Top row last pixel Blue	1 byte
	Top row last pixel Green	1 byte
	Top row last pixel Red	1 byte
	Padding	1-3 bytes

Figure 6:- The Image above shows the structure of a BitMap file

The next section is the DIB header section, this block is 40 bytes long, the block gives the application detailed information about the image and this information would be used to display the image on the screen. The first 4 bytes of this header is used to store the size of the DIB header, next 8 bytes stores the image width and image height in pixel respectively, 4 bytes for each, 2 bytes for the number of color planes(this value must be 1), 2 bytes for the bits per pixel, which is the color depth of the image, good values for the color depth could be 1,4,8,16,24, and 32; but throughout this chapter we would be using only a value of 24, 4 bytes for the compression method, best practice is using the value of zero(none), 4 bytes for the image size(0), 8 bytes for the horizontal and vertical resolution respectively 4 for each, 4 bytes for colors in palette(0), 4 bytes for used palette colors(0).

The last section is the pixel array, whose length is the file size excluding the number of bytes used by the BitMap header and the DIB header. A pixel is made up of three bytes, one byte each for the blue, green and red color depth. The pixel array starts from the bottom row all the way to the top row, at the end of each row there is a possible for a padding which could be up to 3 bytes of padding or no padding at all, these padded bytes would contain a value of zero, there is a padding if the number of bytes in the row is not divisible by 4, if it is divisible by 4, there is no padding needed, when it is no divisible by 4, a number of padded bytes are added to the end of each row, the plan is to make the row divisible by 4.

In order to properly understand the structure of the bmp files, how they work, how they could be manipulated, how they can be made, I would advise we go through some questions and get a closer look at the bmp file handling and structure.

3.2 Tasks/ Practice Questions

Before we look at the questions and solution, let’s go through a series of codes, which shows some properties of a bmp picture or a bmp file. These properties are the size of bmp file (in bytes), the resolution of the picture (pixel x pixel), color depth (in bits). What does it mean? We have a bmp named “proc1.bmp”, we read the properties of this bmp file, and we output the properties to the screen.

First we consider the size, the second bmp header field is the size, which is stored in 4 bytes (from the 3rd to the 6th byte) this information is stored in little-endian byte order. The size is a binary integer (int) value, how and what can be read in.

```
…
int size;                                   // to store the size
char sig[2];                                // to store the signature
int f=open("proc1.bmp", O_RDONLY);  //open a file for read only
read(f, sig, 2);                    // read the first two bytes, the signature, "BM" as a string
read(f,&size,4);                            // read the second field, the size, as integer
printf("The size of BMP is %d. \n", size);   // print it to the screen in normal (decimal) form
…
```

We then read the width and the height of the bmp file, according to the bmp applied format before the width and height, there are some bytes that are not so important at the moment, we skip these bytes by reading them, but not doing anything with the variables.

```
…
int temp;
int W,H;
read(f,&temp,4); // unused filed
read(f,&temp,4); // Pixel Array offset
read(f,&temp,4); // DIB header size
read(f,&W,4); // width in pixel (as binary integer)
read(f,&H,4); // height in pixel (as binary integer)
printf("The resolution is %d x %d. \n",W, H );
…
```

Then the next two fields are just 2 bytes long, so we use ‘short’ variables, the color depth means that, we use 24 bits which is equivalent to 3 bytes to store each pixel, 1 byte for each color.

```
…
short int tmp;
short int Color;
read(f,&tmp,2); // planes
read(f,&Color,2); // color depth (bits per pixel)
printf(“The color of each pixel is encoded in %d bit.\n”,Color);
…
```

We are ready with the basics of how the bmp file format works, We ask ourselves a question, “How much red, green and blue colors are in the bottom left pixel?”, according to the bmp format, these is the first 3 bytes in the pixel array, which is the green part in the picture above, each of these values are independently 1 byte long, so we have to use one-byte-long unsigned integer variables(unsigned char). In order to get to the pixel array, we have to skip the last 24 bytes in the DIB header, by overwriting it with the default value according to the bmp format figure.

```
…
unsigned char R,G,B;
// read intermediate fields temporarily
read(f,&B,1);   //Blue color is always stored, it goes as follows
read(f,&G,1);   //Green
read(f,&R,1);   //Red
printf(“ Red: %d, Green: %d\n Blue: %d\n”, R,G,B);
close(f);
…
```

We have gained the experience to how bmp files work and how bmp files are formed using practical examples, we can now proceed to making and manipulating bmp files.

3.2.1 Task 1

We are given an image named “proc1.bmp”, and our task is to decrease the brightness of this image. In order for the brightness of an image to decrease, the color intensity has to reduce, the three colors making up one pixel has to reduce, a simple way this could be done is by dividing the value of each color by 2, but how could this be done, we would see how soon, we cannot divide the pixels of the same picture that we are reading from by 2, I mean the “proc1.bmp”, the best logical and practical step is to store the new values in a separate bmp image, that we create using our program.

We store the value of the width and height in variables, so we could access them easily. We have to include some extra libraries in our code apart from the default one, The <sys/stat.h> header talks about the structure of the data that is returned by the functions fstat(), lstat(), and stat(),The <fcntl.h> header defines the requests and arguments for use by the functions fcntl() and open().We need this headers in order to use low level file handling. We need the <malloc.h> header, in order to allocate memory to variables when needed, but we always need to deallocate when variables are not in use anymore.

```
…
int f=open("proc1.bmp", O_RDONLY);
int g=open("tina.bmp",O_WRONLY|O_CREAT|O_TRUNC|O_BINARY,S_IRUSR|S_IWUSR);
…
close(f); close(g); // all file pointers should be closed before the end of the program
```

We initialize a file pointer, to open “proc1.bmp”, which is the bmp file that we would be decreasing it brightness, it’s opened for read only (O_RDONLY), g points to a new file that is created called “tina.bmp”, it is opened for writing only, in order to create a this file a third parameter is needed, if O_CREAT is being used. We need to read the first file and copy its

contents into the second file, except the pixel array, we use read(, , ,) for copying and the write(, , ,) for writing into the new file.

```
…
read(f, sig, 2); // read the first two bytes, the signature, “BM” as a string
write(g, sig, 2);
read(f,&size,4); // read the second field, the size, as integer
write(g, &size, 4);
…
int RowSize = 3*W + (W%4);
unsigned char *p=(unsigned char*)malloc(RowSize);
…
```

The first parameter is the file pointer, the second parameter is the variable, and then we have the size in bytes. The size of each row, is the product of 3 and the width of the image, but if the width is not divided by 4, there is some padding of the width modulus 4 (Wmod4). The next row allocate memory for p, as large as to store every pixel, p represents the pixel array, when the pixel array is not used anymore we have to use “free(p)”, to deallocate, let’s see the next set of codes.

```
…
int n=0;
while (n<H){
   read(f,p,RowSize);
   for (i=0;i<(3*W);i++){
      p[i]/=2;
   }
   write(g,p,RowSize);
   n++;
}
```

…

The variable n represents the number of rows we have passed through until we reach the height, we read a row, the values are stored in the pixel array, we go through each value and divide it by 2, and we store the new pixel array in the new file's pixel array. The figures below show the result of the implementation of this code.

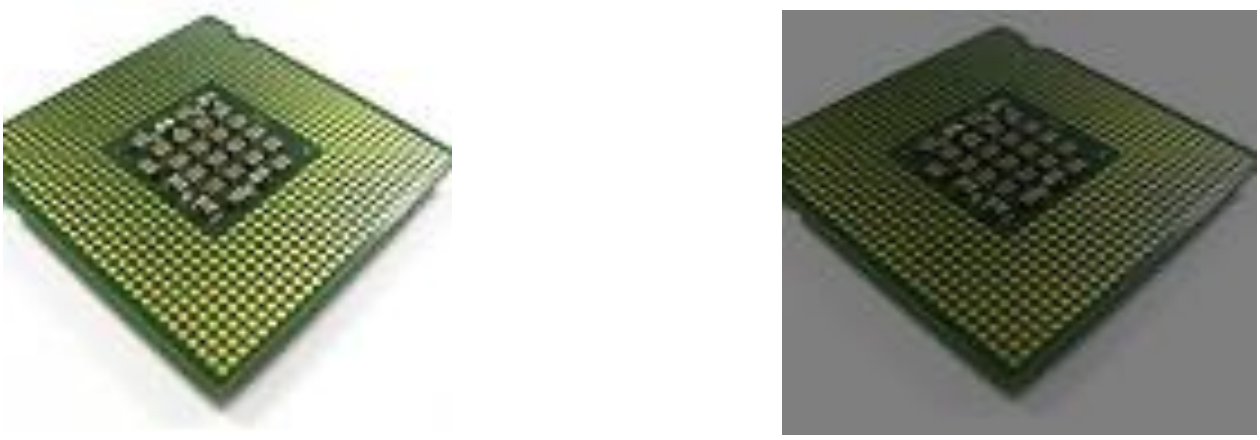

Figure 7:- Shows the before and after effect done by the implementation of the code above

3.2.2 Task 2

We are given an image named "proc2.bmp", and our aim is to convert this image into gray scale. When we ponder for a solution to this problem, we realize that every pixel that is gray scale has a particular feature in common, that is there is an equal number for all the pixel colors in RGB(red,green,blue), the two common gray scale colors, which are white and black have (255,255,255) and (0,0,0) respectively.

A way this could be solved, is to find a solution that makes all pixel colors even, in order to do this and at the same time not lose the intensity of the pixel, we could add all the pixel colors and divide the result by 3, we place the answer as a value for each pixel color, therefore making it even, the result would be a gray scale pixel.

…

```
int n = 0;
int RowSize = 3*W+W%4;
unsigned char *p=(unsigned char*)malloc(RowSize);
while (n < H){
  read(f,p,RowSize);
  for (i=0; i<3*W; i+=3){
    temp   = p[i] + p[i+1] + p[i+2];
    temp  /= 3;
    p[i]   = temp;
    p[i+1] = temp;
    p[i+2] = temp;
    }
  write(g,p,RowSize);
  n++;
  }
 …
```

First we read and write from the image “proc2.bmp” until we reach the pixel array, the row size would be the product of 3 and the width with the addition of padding if necessary, the color depth of the image is 24bits, which is 24/8 bytes, which is 3 bytes, so we have three bytes to store each pixel. We allocate memory to a pixel array, for the storage of each row, we read each row one at a time.

We use a for loop to go through the row elements which are now present in the pixel array, the condition takes until only the length of the actual width, excluding if there was any padding done, we increment the i variable by 3, so we get to the first color of the next pixel, we find the average of all pixel colors and we set the intensity of each color to the value of the average, we store the new pixel array in a new file named “tina2.bmp”, we repeat the process for all rows until we reach the last row, which is the height of the image.

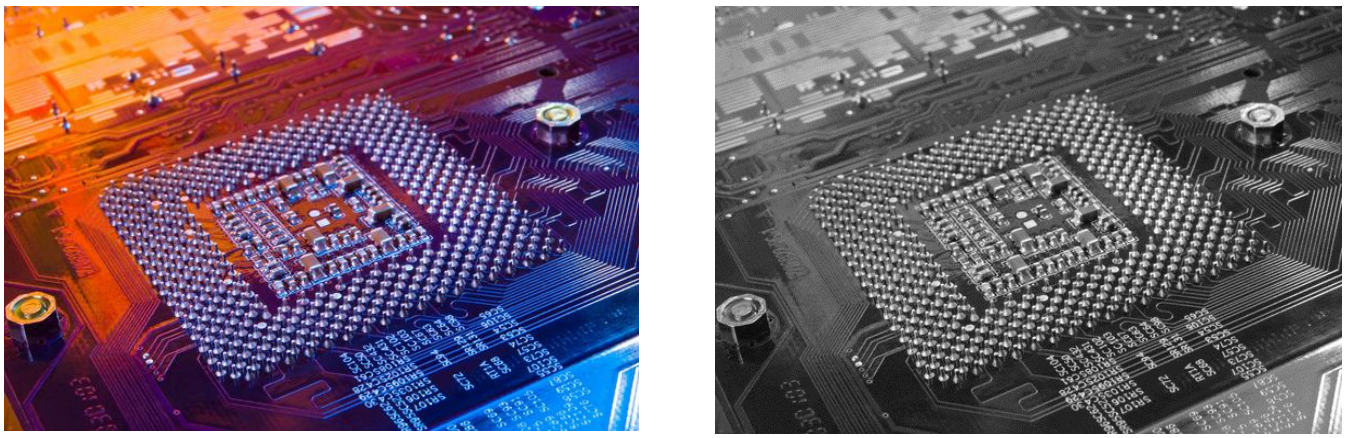

Figure 8:- Shows the before and after effect, due to the implementation of the code

3.2.3 Task 3

We are given a task to perform on "proc1.bmp" image, and our aim is to increase the resolution of this image to double. How could this be done?

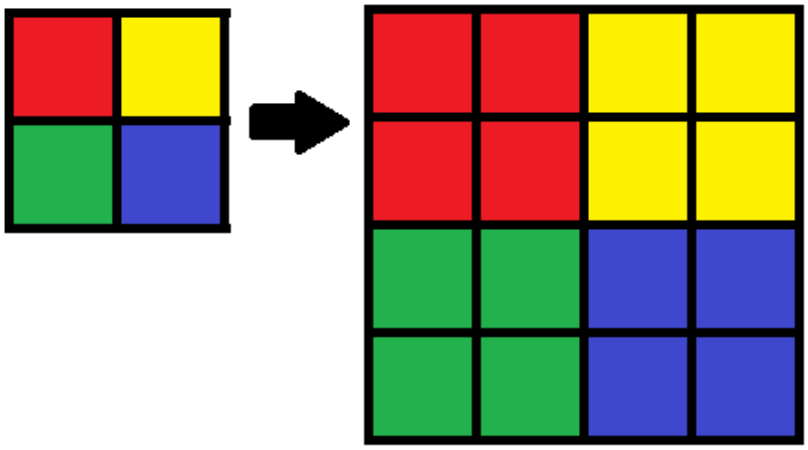

Figure 9:- Shows a sample of a picture whose resolution is doubled

This could be done by, doubling every pixel in a row, and repeating the row again, just like we see on the above figure. Increasing resolution to double means that if the picture has resolution 100x100, we modify it to 200x200 by copying all the pixels 4times (2x2). The final result would be named in a bmp file "tina3.bmp". The size, width, height values from the bmp header, should be doubled before writing to the new image, we repeat the same process of reading and writing to the bmp header, dib header, until we reach the pixel array, but not forgetting to modify the size, height and width to a double amount, in order to cater for the new amount of pixels.

```
…
int n=0;
int RowSize = 3*W + W%4;
int nRowSize = 3*nW + nW%4;
unsigned char *p=(unsigned char*)malloc(nRowSize);//Array to store the new row of pixels
unsigned char *l=(unsigned char*)malloc(RowSize); //Array to store the initial row of pixels
while (n < H){
    read(f,l,RowSize);
    int j=0;
    for (i=0;i<3*W;i+=3){
      p[j] = l[i];
      p[j+1] = l[i+1];
      p[j+2] = l[i+2];
      //Repeating each pixel
      p[j+3] = l[i];
      p[j+4] = l[i+1];
      p[j+5] = l[i+2];
      j+=6; //start writing again after every 6 bytes
    }
    write(g,p,nRowSize); //repeating the new row twice
    write(g,p,nRowSize);
    n++; //increment n to go to the next row
    }
…
```

RowSize represents the initial row being read, and its pixel array is l, nRowSize represents the newly modified and higher resolution image and its pixel array is p. We read the initial row and store its value in l, we then use a for loop, with conditions as before, we copy each pixel colors

twice and store it in pixel array p, the value j is an index for values in the p array, we continue adding the pixels from j+=6, which is a new pixel. When the for loop is done, we store the new row with pixel array ‘p’, twice in the new image file.

Now that we now know how to change and manipulate bmp files, we know how the bmp header, dib header and pixel array work. Let’s now make our own bmp files, these experiments are going to be referred to as task 4 and task 5.

3.2.4 Task 4

Our aim is to write a C program, that creates a bmp file with these attributes: resolution 200x200, color depth is 24 bit, it would contain 4 100x100 quarters of different colors (red, green, blue, yellow). The image would be similar to the 2x2 sample image for task 3, it would be a bigger version.

```
…
int g = open("tina4.bmp",O_WRONLY|O_CREAT|O_TRUNC|O_BINARY,S_IRUSR|S_IWUSR);
…
int size = 200*200*3 + 54;
…
int W= 200, H = 200;
…
```

In this case we would have to insert our own values, there would be no file to read from, we would use the default values from the figure of applied bmp format, and our own values as well. We have only one file pointer, we open a file “tina4.bmp” for writing only.

The file would be a quarter of 100x100 for each color which makes the width and height a value of 200, therefore the size of the file would be the product of the row size (200*3) and the height(200) with an addition of the bmp and dib header which is 54 bytes. All other values would be filled with their defaults until we reach the pixel array.

```
…
```

```
int i;
int RowSize = 3*W + W%4;
unsigned char *p = (unsigned char*)malloc(RowSize);
int n=0; int d;
while (n<H){
   d=0;
   for (i=0;i<3*W;i+=3){
      if ((d<100) && (n<100)){ //lower left
         p[i] = 0;
         p[i+1] = 255; //Green
         p[i+2] = 0;
         d++;
      }else if ((d>=100) && (n<100)){ //lower right
         p[i] = 255; //Blue
         p[i+1] = 0;
         p[i+2] = 0;
         d++;
      }else if ((d>=100) && (n>=100)){ //upper right
         p[i] = 0;
         p[i+1] = 255; //Green +
         p[i+2] = 255; //Red
         d++;
      }else if ((d<100) && (n>=100)){ //upper left
         p[i] = 0;
         p[i+1] = 0;
         p[i+2] = 255; //Red
         d++;
```

```
        }}
    write (g , p, RowSize);
    n++;
    }
    …
```

We have our pixel array p, to store every row, we introduce a variable d to tell us when we have passed the middle of each row, so we continue with another color. We use the same for loop as before to go through each pixel, when we are at the lower left we fill with the green color, when we are at the lower right we fill with blue, upper right we fill with yellow(Red + Green), upper left we fill with red, we write the new row to the file, and we continue the loop until we reach the height of the image.

Note:- Pixel colors in bitmap are stored in BGR(Blue,Green,Red) not RGB(Red,Green,Blue).

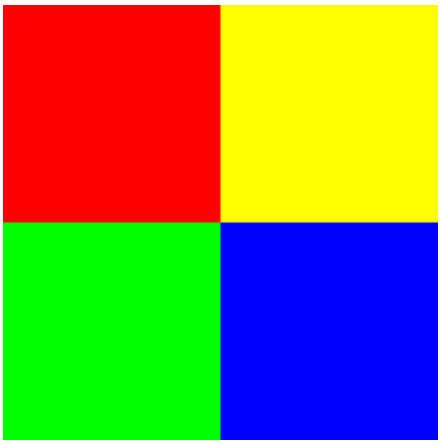

Figure 10:- Shows a created bmp file by the implementation of the above code

3.2.5 Task 5

Our aim is to write a C program, that creates a bmp file with these attributes:- resolution is 256x256, color depth is 24 bit, each line of the image represents 1 shade of red (from pure red to black), this means the top of the image is (light) red and then we darker and darker red shade as we move downward, and finally the last row is black.

The size of this image would be, the row size (256*3) added to the multiplication of padding for each row(256*(256%4)), everything multiplied by the height(256) of the image, and the addition for bmp and dib header(54) is made. So size = 256*(256*3 + 256*(256%4)) + 54.

```
…
unsigned char R,G,B;
R=0; //Red
G=0; //Green
B=0; //Blue
while (n<H){
   for (i=0;i<3*W;i+=3){
      p[i]= B;
      p[i+1] =G;
      p[i+2] = R;
      }
write (g , p, RowSize);
R++; n++; }
…
```

We use variable R,G,B to store the values of the pixel color, we use the same for-loop, to get to the next pixel, after every for-loop run, the value of red(R) is incremented by 1 until we reach the height of the image.

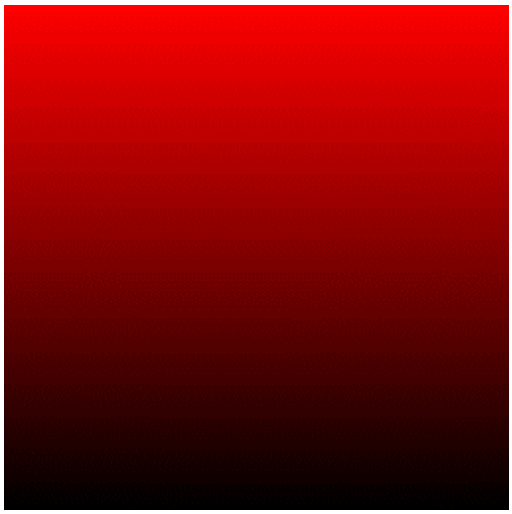

Figure 11:- Shows a created bmp image, due to the implementation of the above code

At the stage, bmp creation, manipulation, bmp headers, dib header, pixel array, are no longer strange topics to us. Learning how bmp works is important, for this project, because we want to simulate different processes and their time-evolution (snapshots) will be represented by the use of bmp images.

4 Random Walk Implementation

This subchapter would be dealing with implementation done, based on the knowledge of random walk theory, we would be using bmp files to shows snapshots at different time intervals of multiple random walkers moving in a state space, or a defined area. This implementation will be based on the two dimensional type of random walk, where we have both the x and y axis.

A randomly moving particle is illustrated by a square of side length 2S+1. To illustrate the latest positions of the particle a kind of trace is also visualized representing the older states by lighter and lighter gray shade. So the states which are older than a threshold (given by parameter hmax) are not visible (black on a black background). Thus the final animations contain wriggling snakes, however the motion of single point-like particles are simulated. The white square which behaves as the “head of snake” is the real particle that obeys random walk. In this chapter we may refer to “Random Walkers” as “Snakes”, but in the real essence we are talking about a Random Walker, obeying the laws of Random Walk.

4.1 Implementation

All properties of the random walk are stored in a parameter file called “param.txt”, one of the importance of the file is to allow easy manipulation of the random walkers. This parameter files contains these variables exactly as listed below, we would see how each variable works later: The total length of squares in the grid(lenofs, e.g. the length and height is 44 squares long), value of S(side length of each square is 2S+1, e.g. S=2, 5), coloring method(cm), length of the random walker’s head(hmax), number of random walkers in the defined area(snakenum),time of simulation(tmax), and the fileID.

```
…
fp=fopen("param.txt", "rw");//Opening the Parameter file for reading
fscanf(fp,"%d %d %d %d %d %d %s",&lenofs,&S,&cm,&hmax,&snakenum,&tmax,fileID);
//Reading from the parameter file
close(fp); //Close the parameter file
…
```

The variable name “fp”, is the file pointer to our parameter file, we open this file for reading and writing, we store the values in the file into variable names that we would use in our program (lenofs,S,cm,hmax,snakenum,tmax,fileID), these variables are read and stored according to how they are written in the parameter file, all values are integer values excluding the fileID, the fileID gives us the ability to keep the previously created bmp files, without overwriting them with the new created one, the fileID used in this implementation is “RW”, which represents it’s a random walk.

```
…
if (lenofs%2==1) lenofs--;
L= (2*S+1)*lenofs;
pixelsize = L*(L*3+L%4);
moveby = 2*S+1;
…
```

We make sure the total number of squares in a row and column of the grid is even, this makes it easier if we would like to get the middle of the grid directly. The area coverage is a square and it’s length is given by L, the side length L of the image in pixels is determined by the size of a square(2S+1) and the number of squares along the axis of the grid(lenofs), the variable “moveby” , if we are at the lower left corner of a square, this is the distance it takes us to reach the next square’s lower left corner as well.

```
…
 int x[snakenum][hmax], y[snakenum][hmax];
memset(x, 0, sizeof (x)); memset(y, 0, sizeof (y));
srand(time(NULL));
…
```

We use a two dimensional array to store the location for each random walker, we assume snakenum is i and hmax is j, x[i][j] is the x coordinate of the jth square of the ith random walker,

and y[i][j] would be the y coordinate of the jth square of the ith random walker, the memset(,,,) to set all the values of the x and y array to zero.

When random numbers are generated in C programming, these numbers are not truly random, if the source code is run several times, and the random numbers are printed out, we notice that we get the same values over and over again. The way random numbers are generated is, the random number generator algorithm starts from a seed, and moves a certain way and picks a number, so it always starts from the same seed over and over, therefore we get the same number.

There is a function that could be used to change where this algorithm starts from, "srand()" if a constant is kept as the parameter, it would still be the same number over and over again, except the seed is changing, one thing we know not to be constant is time, so we could keep time value as the seed, therefore we get a random number every time "srand(time(NULL))" .

```
…
for (i=0;i<snakenum;i++){
  rnum= rand()%(lenofs);
  x[i][0]=(2*S+1)*rnum;
  rnum= rand()%(lenofs);
   y[i][0]=(2*S+1)*rnum;
 }
…
```

We have two dimensional arrays, x and y which store the coordinate of the head of every snake, in the for loop above we store the initial position of the head of the snakes, as generated random numbers. We pick a random number from 0 still the lenofs, and any number we get would be stored in rnum, the x or y initial coordinate would be the product of the size of a square and the random number generated.

```
  …
  Head = (unsigned char*) malloc(54);
```

```
createBmp();
…
```

We allocate memory for an array that would store, the bitmap and dib header values, this is done because the values is going remain constant, only the pixel array changes during the program run, we call the createBmp() function, let's see what it does.

4.2 The createBmp() function

The aim of the create a bmp file, in this file all the values of the pixel array is zero, so we are just creating a black image, we are going to read the header of this file during the run of our program, this value would be used for the creation of further bmp files illustrating random walk, the createBmp() is a method so we are using the void type.

```
…
int t=0;
int size;
char sig[2] = "BM";
char name[100];
sprintf(name,"%s%04d.bmp", fileID,t);
int g = open(name,O_RDWR|O_CREAT|O_TRUNC|O_BINARY,S_IRUSR|S_IWUSR);
write (g, sig, 2);
size = L*(L*3 + L%4) + 54;
write (g, &size, 4);
…
unsigned char *p = (unsigned char*)calloc(pixelsize, sizeof(unsigned char));
…
free(p);
close(g);
…
```

We are using the applied bmp format, which we learnt earlier, to create this file and fill the header fields, t=0 is used to indicate that we have not started the time run, this is just a created file, so we get the header fields. The sprintf function prints the value to a string called name, the value would be “RW0000”, this is the value the string stores, a file pointer g is opened with the string value, the height and width of the image is L.

We create the file following the applied bmp format which was learnt earlier, the calloc function is used as an easier way to set all the values of the pixel array to zero, we free the pixel array as well as not forgetting to close the file.

The final implementation would have the possibility of using two coloring method, one which the intensity of a snake head decreases with a depth and another which the color just remains constant all through the snake head, according to the figure below

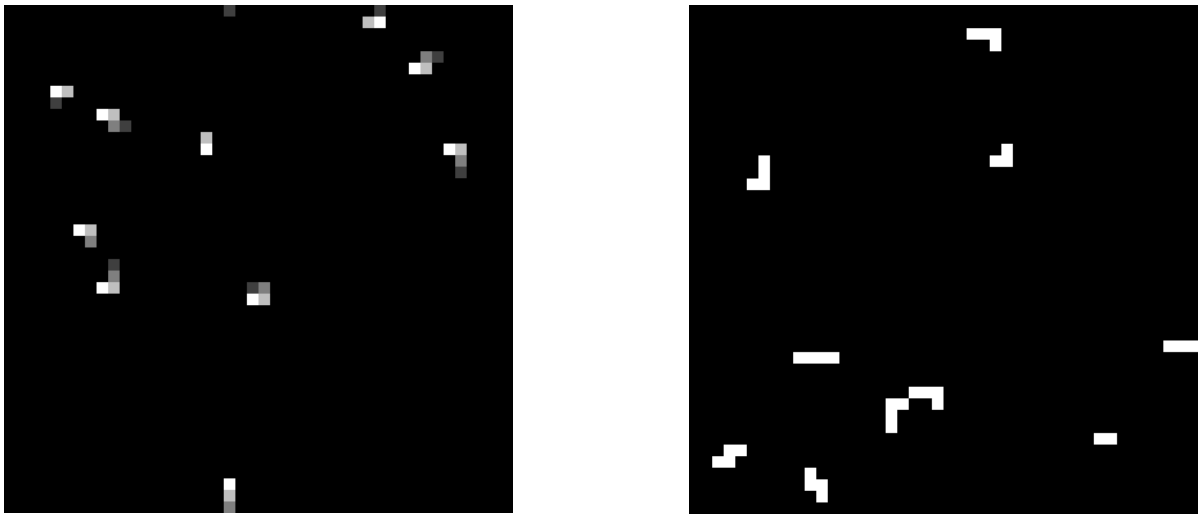

Figure 12:- Shows snapshots of bmp 2 bmp files, illustrating the two coloring methods

4.3 Implementation(Continuation)

…

```
t=0;
sprintf(name,"%s%04d.bmp", fileID,t);
g = open(name, O_RDWR);
read(g,Head,54); //read only the head
close(g);
…
```

We store the name of the just created black bmp file in the a string called name, we use a file pointer g to read from this file, the bmp and dib header of the file is stored in the “Head” array, anytime a file is opened, we have to close the file after it’s no longer in use.

```
…
float c[hmax];  //to store color shades
c[0]=255;
for (i=1;i<hmax;i++){//change the values of the remaining
   if(cm==1)
      c[i]= c[i-1] - 255/(float)hmax; //So the value is always decreasing
   if(cm==2)
      c[i]= c[i-1]; //So the value is constant
}
…
```

Earlier we spoke about the coloring method which is in the parameter file, it’s value is stored in variable “cm”, an array is created to store the colors shade of the head of the snake, depending on the coloring method, if cm=1 then the tip of the head of the snake is the brightest shade of grey, and the color intensity reduces until the length of the head is reached, if cm =2, then the head of the snake has the same color in all its squares.

4.4 Random Walk Simulation

In order to make snapshots of the random walkers, we need to go from t=1 still we reach the time of simulation which was read from the parameter file(tmax), we use a while loop, "while(t<tmax)", everything would be done in this while loop.

```
…
 PA = (unsigned char*) calloc(pixelsize, sizeof(unsigned char));
for (i=0;i<snakenum;i++){
   for (j=hmax-1;j>=0;j--){
       ….
}}
…
```

We are in the while loop for the simulation, we allocate memory for the pixel array, all the values of the pixel array are initialized to a value of zero. We need to address each snake and each head square for the snake specifically, independent of the other coordinates, we need a for loop with variable i representing the index of the snake, and another for loop with variable j representing the index of the head of the snake, the second for loop prints backward so the tip of the head of the snake could always be since, if there are other coordinates of the snake with the same value as it.

```
…//Inside the two for loops
tmpx=x[i][j];tmpy=y[i][j];
 while (1){              //for the column
     while(1){           //for the row
          PA[tmpy*(L*3 + L%4) + tmpx *3 + 0 ] = c[j];  //Blue Pixel
          PA[tmpy*(L*3 + L%4) + tmpx *3 + 1 ] = c[j];  //Green Pixel
          PA[tmpy*(L*3 + L%4) + tmpx *3 + 2 ] = c[j];  //Red Pixel
          if (tmpx == (x[i][j] + moveby -1)){
```

```
                tmpx = x[i][j];
                break;
            }//If
            tmpx++
        }//While
        if (tmpy == (y[i][j] + moveby -1)) break;
        tmpy++;
    }//While
…
```

In order not to change the values of the initial coordinate of the head of the snake, we store this values in temporary variables, tmpx and tmpy. The first while loop that follows it is for the column, and the second one is for the rows, in the second while loop we store the color of the head coordinate according to the color method stored in the array c[j], we amend the appropriate color in the pixel array “PA”, “tmpy*(L*3 + L%4)” is the number of bytes below the coordinate on other rows, “tmpx *3” is the number of bytes before the coordinate on the same row, adding 0, 1, 2, we get to the Blue, Green and Red pixel respectively.

In order to get a big square of length 2S+1, we have to write the row over and over until we reach the height of 2S+1, the if condition helps us to know if we have reached the edge of the square, if yes, we reset the value of tmpx to the beginning edge of the square, so we could write the row again, then we break the while loop for the row, if we haven’t reached the edge, we just increment the value of tmpx. When we have reached the edge we check if we reached the column limit of the square as well, if yes we break from the while loop of the column as well, if the answer is no, we just increment the value of tmpy and continue amending the pixel, this gives us a visible square of length 2S+1 instead of a really small pixel. We leave the last two, for loops and continue with our implementation.

```
…
sprintf(name,"%s%04d.bmp", fileID,t);
```

```
g = open(name,O_RDWR|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(g,Head,54);
write(g, PA, pixelsize);
close(g);
…
```

In order to create a new bmp file with the amendment from the present time simulation, we store the name of the files in a string called “name” (a sample of names RW0001.bmp, RW0002.bmp, …), a bmp file is created, the header fields “Head” are added to the file, as well as the pixel array “PA”.

```
…
for (i=0;i<snakenum;i++){
   for(j=hmax-1;j>0;j--){
       x[i][j]=x[i][j-1];// x[i][1]=x[i][0]
       y[i][j]=y[i][j-1];
      }}
for (i=0;i<snakenum;i++){
      …
}
…
```

After the image has been created, we aim to shift all coordinates for each snake to the right by one step, all coordinates on the last/oddest when shifted are lost. We have a for loop for every random walker in the system, the next for loop does the shifting of coordinates, therefore x[i][1] will become x[i][0]. A new for loop is created in order to change the current coordinate for each snake (x[i][0],y[i][0]), this for loop is a fundamental to get the head of the snakes to obey the law of random walk.

```
…//In the for loop
rnum=(float)rand()/RAND_MAX;
```

```
if( rnum < P_up) {
   y[i][0]+= moveby; //Move Up
    if(y[i][0] ==L){
       y[i][0]=0;
}}//if
else if(rnum >= P_up && rnum <P_up+P_down) {
    y[i][0]-= moveby; //Move down
    if(y[i][0] == 0-moveby){
        y[i][0]=L-moveby;
}}//else if
 else if(rnum >= P_up+P_down && rnum <P_up+P_down+P_left){
    x[i][0]-= moveby; //Move left
   if(x[i][0] == 0-moveby){
      x[i][0]=L-moveby;
}}//else if
else if(rnum >=P_up+P_down+P_left /*&& rnum<P_up+P_down+P_left+P_right*/) {
    x[i][0]+= moveby; //Move Right
    if(x[i][0] ==L){
       x[i][0]=0;
 }}//else if
…
```

A random number is generated between 0 and 1, an if condition checks if rnum means we should move up, if yes the y coordinate of the snake is incremented by moveby, which takes us to the lower left corner of the upper square, if our incrementation takes us to position L, then we continue from 0. If it means we should move down, the value of the y coordinate is decremented by moveby, if our decrementation took us to 0-moveby we start from L-moveby.

If it means we should move left, the value is of the x coordinate is decremented by moveby, if our decrementation took us to 0-moveby we start from L-moveby. And if it means we should move right, the value is of the x coordinate is incremented by moveby, if our incrementation took us to L we start from 0. Therefore we would have a periodical motion, if a random walker reaches any of the borders it comes out from the other side, we repeat the same procedure for all the snakes.

After we leave the for loop, we must increment the value of t, so we go to the next time simulation in the while loop.

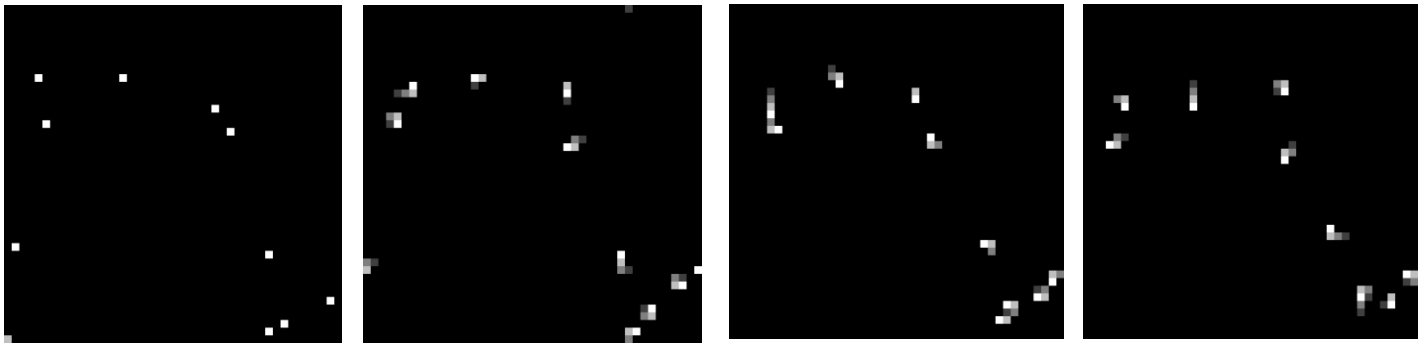

Figure 13:- Shows snapshots of 10 random walkers, taken at different time intervals RW0001, RW0010, RW0015, RW0020

5 Diffusion Limited Aggregation Implementation

Diffusion-limited aggregation (DLA) is basically a process whereby particles, obeying random walk due to Brownian motion cluster together to form aggregates of such particles. In this chapter we are going to implement both the 2d and 3d version of the diffusion limited aggregation, we would also look at the concept of ray tracing using the pov-ray application, which would be in the 3d version.

5.1 D.L.A. In Two Dimension

Basically, what we want to create, let's imagine we have a square image, as a base for the random walk of several particles to take place. The bottom layer of the square contains sticky particles that have been inserted earlier, therefore the coordinates of these particles are contained in the list of particles already.

We insert a new random walker, at a position where, the y coordinate is the maximum height of the system plus a constant value(e.g. 10), the x coordinate is between 1 and the width of the square. The particle starts obeying the law of random walk, if the particle reaches an upper limit, the particle dies and a new particle is initialized the same way as described above.

During the random walk if the particle is one of the neighbors to any of the previous particle in the list, the particle coordinate is added to the list, we also check if it height is higher than the maximum height of the system, if yes, it's y coordinate becomes the new height, then a new particle is initialized in the same way until we reach the desired number of particles needed in the system. In the final result, the color of the particle depends on when it is added to the list of particles.

In the last figure chapter 2, it shows a two dimension example for diffusion limited aggregation(line attractor), what we are going to implement would be very similar to what we have in the image, a major difference would be that, our implementation would not create the big blue sky like drawing above the particles in the system. When we have reached the maximum number of particles in the system, we create a bmp file with the new pixel array containing the colors of the particles.

5.1.1 Implementation

A structure is created, so the coordinates of every particle could be stored easily, and reference to the particles and their coordinates could be done easily, that means the values can be reached in no time, this structure would be called particle.

```
…
struct particle{
    int x;
    int y;
};
…
```

We use similar variable names like was used for the bmp implementation, as well as the random walk implementation, so as to avoid no confusion.

```
…
pixelsize = L*(L*3+L%4);
PA = (unsigned char*) calloc(pixelsize, sizeof(unsigned char));
srand(time(NULL));
//
int N=L*10;
struct particle part[N];
int hmax=0;
…
```

Most of all the variables used are declared as global variable, to be safe if we would like to use the same variable and manipulate its value in multiple functions. The code above is in the main function, pixelsize refers to the number of pixels needed for this implementation, and can be calculated like we see above, we initialize a pixel array (PA) to store this amount of pixels.

Because we would be dealing with the law of random walk, we need to make sure the random number generation algorithm is random, using the “srand()” function like we studied earlier.

The variable N is the number of particles needed for this implementation, its value should be a very large number much greater than L (e.g. L*10). Instead of creating one object for the structure we need an array which contains the x and y coordinates of N particles, there we use “part[N]”. The variable hmax in this implementation would represent the maximum height of the present particles, or the y coordinate of the highest particle in the system, its initial value would be zero, as there are no particles at the moment.

5.1.2 For-Loop, D.L.A. Process

We need a for loop which runs until we reach the desired number of particles in the system, which is represented by the variable “N”.

```
…
for (i=0; i<N; i++){
    bon=1;
    …
    }
…
```

The variable bon is created to be a binary number, which can have a value of either ‘0’ or ‘1’, it would be used as an indicator to break a do-while which we would be implementing in this for loop.

```
    …
    if ( i < L ){
      part[i].x = i; part[i].y=0;
    }//If
    else{
    …
```

```
}//else
…
```

As we can remember, initially at the bottom layer, we need particles at every pixel position, that's means we insert particles from 0 still we reach the image width 'L', when we passed the image width, this is when we start inserting particles at a dynamic height.

```
…
part[i].x = rand()%L; part[i].y= hmax + 1;
tmpy=hmax+1+2;
do{
…
}while(bon)
…
```

We are in the "else" condition, at the moment, part[i].x refers to the x coordinate of a particle with index i, which is a random number between 0 and L, part[i].y refers to the y coordinate of a particle with index i, which is the addition of the maximum height of existing particles and a constant(e.g. 1).

The variable "tmpy" is the upper limit, where the particle cannot pass, where the particle dies, and a new one enters, the value of "tmpy" is the addition of the position where the particle was initialized and another constant (e.g. (hmax+1)+2). We need a do-while loop, whose main function is for the movement and sticking of the particles to existing neighbors, the bon condition in the while loop, tells us when to stop the movement of a particle, when to break out of the loop.

```
…
rnum=(float)rand()/RAND_MAX;
if (rnum < P_up) part[i].y+=1;
else if (rnum >= P_up && rnum < P_up+P_down) part[i].y-=1;
else if (rnum >= P_up+P_down && rnum < P_up+P_down+P_left ) part[i].x-=1;
```

```
else part[i].x+=1;
…
```

Now we are in the do-while loop, we focus on the movement of the initialized particle, a particle can move in 4 different directions (up, down, left or right), with 4 different probabilities which sum up to a value of 1. A random number is chosen, then we check this number using the probabilities, the result gives a movement of either up, left, right or down.

```
…
if (part[i].y >= tmpy || part[i].y<=0 || part[i].x<=0){
    part[i].x = rand()%L; part[i].y= hmax+1;
    continue;
}
…
```

If the particle reaches the upper limit, a new particle is initialized, with the same way as we did earlier, and the do-while loop is started again, with the help of the “continue” keyword, some other conditions were added to avoid bugs or glitches during the program run.

```
…
for (j=0;j<i;j++){
  if ((part[i].x + 1 == part[j].x  && part[i].y==part[j].y)||
      (part[i].x - 1 == part[j].x  && part[i].y==part[j].y)||
      (part[i].y + 1 == part[j].y  && part[i].x==part[j].x)||
      (part[i].y - 1 == part[j].y  && part[i].x==part[j].x)){
          if (part[i].y > hmax) hmax=part[i].y;
             bon=0;
             break;
  }//If
}//for
```

…

The for-loop above helps to check for neighbors, by going through all the previously added particles in the list, the if condition compares each particle with the present particle, to check if they are neighbors. A particle that moves in four directions, has a probability of being neighbors with the an upper particle, a lower particle, a left hand particle or a right hand particle, so we have four possibilities of finding its neighbor.

A neighbor can be found on the right if an increment in the x coordinate, the y coordinate remains constant, we get a particle from the list; it could be found on the left if a decrement in the x coordinate, the y coordinate remains constant, we get a particle from the list; it could be found on the upper part if an increment in the y coordinate, the x coordinate remains constant, we get a particle from the list; it could be found on the lower part if a decrement in the y coordinate, the x coordinate remains constant, we get a particle from the list.

If we have found a neighbor to this particle (part[i].x, part[i].y), another if condition is needed to check if the height of the particle "part[i].y" is greater than the current maximum height of particles "hmax", if it is, the y coordinate of this particle becomes the new "hmax"; "bon" is set to zero to indicate that a neighbor has been found we should stop the movement loop the "do-while loop" and initialize the next particle in the for loop, the "break" key leaves the current for loop.

```
…
if (i < L){
   …
}else{
      …
     do{
        …
     }while(bon);
 }
```

```
…
```

The variable “bon” has been set to zero, this breaks the do-while loop, in the next run of the for-loop, bon is set to one again, so the movement of the next particle occurs until we find a neighbor bon becomes zero again, then we leave the loop.

```
…
PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 0 ] = 128;
PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 1 ] = 0;
PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 2 ] = (i/(float)N) * 255;
…
```

When we leave the “else” bracket, we have to store each particle added to the list on the pixel array as well, so that a bmp picture could be created with the coordinates and the color depends on the time the particle was added to the list. How the colors are implemented, according to the above code, the blue pixel remains constant, and the red pixel is dependent on the value of i, the value of any pixel cannot be more than 255.

```
...
for (i=0; i<N; i++){
  …
}
createBmp();
…
```

This function creates an image, using the values in the pixel array just like, the implementation used during the random walk, the name of the created file would be “RW0000.bmp”. The first 54 bytes containing the “bmp header” and “dib header” are written, before the pixel array “PA” is written to the file.

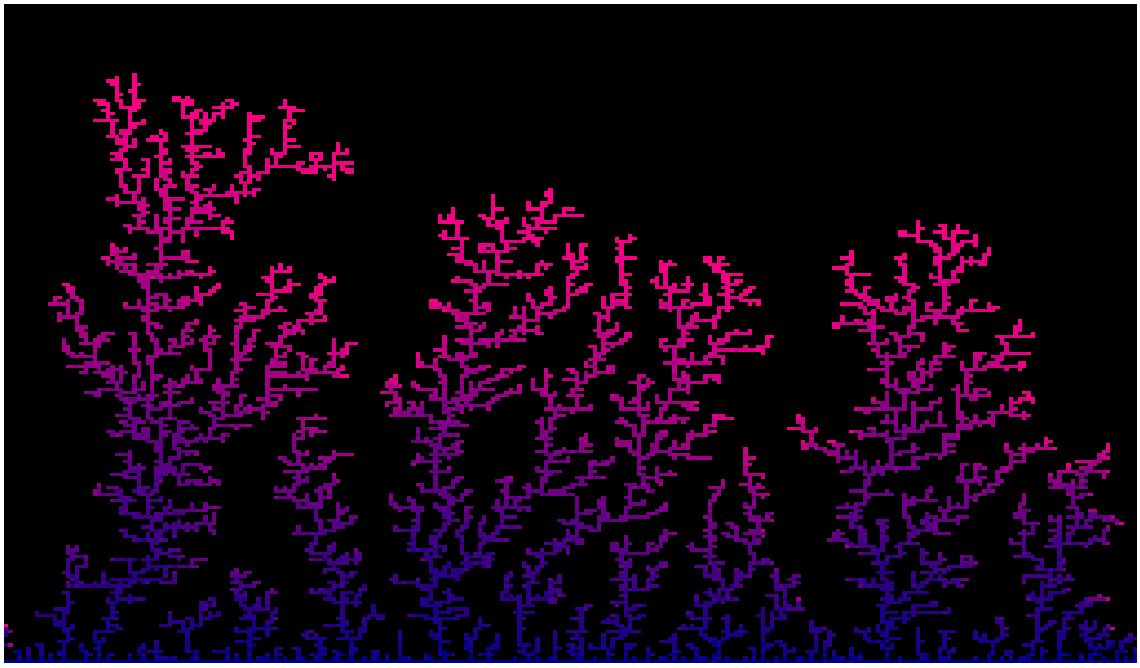

*Figure 14:- Shows the result of the code, with N=L*30, and probabilities 0.23,0.28,0.26,0.23, for P_up, P_down, P_left, P_right respectively*

5.2 D.L.A. In Three Dimension

Now we have familiar with the 2d implementation for the D.L.A., right now the focus would be on the 3d version. For this implementation we would need a 2d sticky square, it would be a bottom surface and a 3d space for random walk would be needed. Basically we would update the program code for to the 2d implementation, to 3d, the following must be taken into consideration.

The random movements in the case of 3d, would be in 6 possible directions, instead of 4 (up, down, left, right, forward, backward), unlike the case of 2d. The [0.0 , 1.0] interval is divided into 6 parts to choose a random direction with "P_up, P_down, P_left, P_right, P_forward, P_backward" values and their sum must be 1.0. The particles need 3 coordinates: x (left-right), y (up-down), z (forward-backward); the x and z coordinates must behave periodically (when the particles reach the edges, they come from the other edge).

The initial sticky bottom layer is a square (e.g. 20x20), automatically containing sticky sediments. Above this surface with a given distance (similar to the case of 2d), we can start the 3D random walkers one-by-one separately, if a particle touches one of the previous particles in the list, it gets stuck, if it's y coordinate is greater than hmax, it becomes the new hmax.

The sticking condition would have also 6 different directions and possibilities (up, down, left, right, forward, backward), if a particle reaches a very high position, this position would be changing with a change in the maximum height of the particles, whose value would be the position where a new particle is initialized with the addition of a constant.

This implementation would not be created and simulated with the use of bmp files, an easier way is by the use of the "Pov-ray" application, our aim is to create beautiful 3d images using this sophisticated tool. The possibility of using bmp files to simulate this process is possible, it's much more difficult and would require a lot of mathematics, computer graphics and more. In the case of bmp files we used raster graphics, but we use ray tracing in the case of pov-ray.

5.2.1 Pov-ray Format

According to Wikipedia, "The Persistence of Vision Ray Tracer, or POV-Ray, is a ray tracing program which generates images from a text-based scene description, and is available for a variety of computer platforms", Pov-ray is a free and open-source software, anyone could download of the internet and use.

Pov-ray can create three dimensional, photo-realistic pictures, its uses a rendering technique which is known as ray-tracing. Ray-tracing may not be a fast process, but it's known to produce very high quality images containing realistic reflections, perspective, shading and other effects.

In order to understand the format of the pov-ray application, we take a look at the source code of an image created with the use of this file. This image illustrates another process/structure, it's not about the Diffusion Limited Aggregation, it is the microstructure of an aerogel sample, we are just going to implement the same format for our 3d representation of D.L.A.

Figure 15:- Shows begin and end parts of the source code of a simple pov file

The figure above contains the beginning and end parts of a simple pov file, from which we would use as a base of reference for our 3d implementation using the pov-ray application. The first include statement reads in definitions for various useful colors, pov-ray is made up of many standard include files, reads predefined shapes,textures,metals.

The camera location is set, where we see the image from, the look at specifies the focal point of the image, in our implementation the camera location code would be at the last part of the code; the background is set to white. The union binds two or more shapes together, each particle can be taken as a sphere when written in pov-ray, while including it's correct coordinates, specify the color intensity to be used as well.

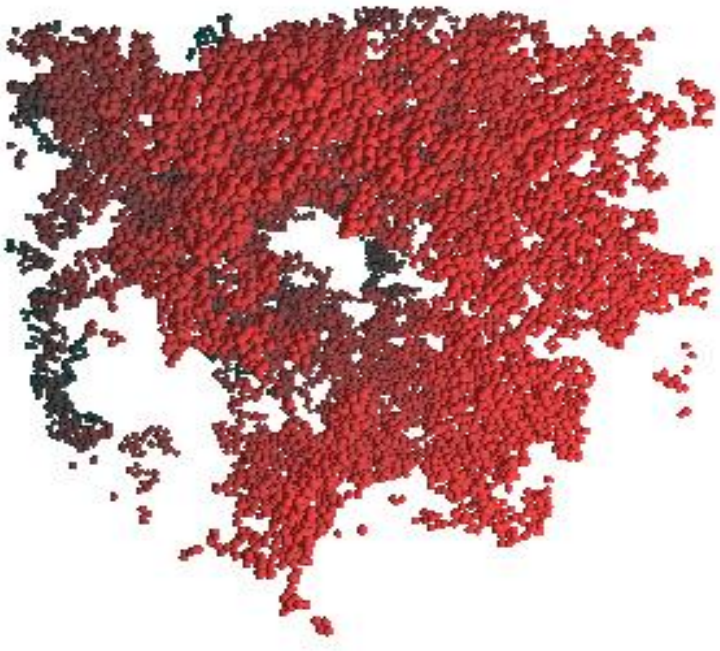

Figure 16:- Shows an image created by the implementation of the code in the figure above

5.2.2 Implementation

In this subchapter, we would look at the implementation of the 3d D.L.A., the changes, new directions for movement, probabilities and many more, we would use ray tracing technique with the help of the pov-ray application.

```
…
struct particle{
   int x;
   int y;
   int z;
};
…
```

Unlike the case of 2d, that only 2 coordinates were needed for “up, down, left and right” movement, to implement in 3d, an additional coordinate would be needed for the “forward and backward” movement.

```
…
```

```
int N=L*5;
…
for (i=0;i<SL;i++){
      for (j=0;j<SL;j++){
         part[count].x=j; part[count].y=hmax=0; part[count].z=i;
         count++;
      }
}
…
```

We have to introduce a variable “SL”, this variable represents the side length of the square, which lies on the x and z axis, the variable count refers to the number of elements added in the list. The two for-loop stores the coordinates of the sticky bottom square which lies on the x-z axis in the list.

```
…
P_up=P_down=P_left=P_right=P_forward=P_backward= 1.0/6.0;
for (i=count;i<N;i++){
          …
}
…
```

Equal probabilities are used for each movement, the for-loop is used to aid the addition of new particles to the list, until we reach the max N, i goes from count still N.

```
…//In the for-loop
bon=1;
part[i].x = rand()%SL; part[i].y= hmax + 5; part[i].z= rand()%SL;
tmpy=hmax+5+4;
do{
```

```
        …
}while(bon);
…
```

Similar to the 2d, the bon variable would be used to stop the do-while loop, the x and z coordinate are initialized between 0 and SL, the y coordinate is initialized at the maximum height of the particles in the list with the addition of a constant, the variable “tmpy” refers to the upper limit where a particle could not pass, its value is the y coordinate with the addition of another constant. The do-while is also known as the movement loop like in the case of 2d implementation.

```
…//While in the do-while loop
rnum=(float)rand()/RAND_MAX;
if (rnum < P_up) part[i].y+=1;
else if (rnum >= P_up && rnum < P_up+P_down) part[i].y-=1;
else if (rnum >= P_up+P_down && rnum < P_up+P_down+P_left ) part[i].x-=1;
else if (rnum >= P_up+P_down+P_left && rnum < P_up+P_down+P_left+P_right) part[i].x+=1;
else if (rnum >= P_up+P_down+P_left+P_right && rnum < P_up+P_down+P_left+P_right+P_forward)
part[i].z+=1;
else if (rnum >= P_up+P_down+P_left+P_right+P_forward &&
rnum<P_up+P_down+P_left+P_right+P_forward+P_backward) part[i].z-=1;
…
```

We are in the do-while loop, we choose a number random number, and we determine which direction the particle would move, similar to the case of 2d, but there are two extra conditions in this case for forward and backward movement.

```
        …
        if (part[i].y >= tmpy || part[i].y<0){
            part[i].x = rand()%SL; part[i].y= hmax+5; part[i].z= rand()%SL;
            continue;
```

```
}
…
```

After the movement of the particle, if it is found to reach the upper limit, a new particle is initialized, with the same parameter as it was done earlier, the do-while loop is restarted, with the "continue" key word.

```
…
if (part[i].x >= SL) part[i].x=0;
else if (part[i].x < 0) part[i].x=SL-1;
if (part[i].z >= SL) part[i].z=0;
else if (part[i].z < 0) part[i].z=SL-1;
…
```

The particles has to be periodical on both the x and z axis, if the particles reaches any on the edges on this two axis, it has to come out from the opposite edge, the condition above makes sure of that, so the movement is not distorted.

```
…
for (j=0;j<i;j++){
    if ((part[i].x + 1 == part[j].x  && part[i].y==part[j].y && part[i].z==part[j].z)|| //Right
        (part[i].x - 1 == part[j].x  && part[i].y==part[j].y && part[i].z==part[j].z)||  //Left
        (part[i].y + 1 == part[j].y  && part[i].x==part[j].x && part[i].z==part[j].z)||  //Up
        (part[i].y - 1 == part[j].y  && part[i].x==part[j].x && part[i].z==part[j].z)||  //Down
        (part[i].z + 1 == part[j].z  && part[i].x==part[j].x && part[i].y==part[j].y)||  //Forward
         (part[i].z - 1 == part[j].z  && part[i].x==part[j].x && part[i].y==part[j].y)){  //Backward
              if (part[i].y > hmax) hmax=part[i].y;
              bon=0;
              break;
            }//If
```

```
            }
        …
```

During the movement of each particle, the particles are compared with all the previous particles in the list, to confirm if they are neighbors to any of the previous particles. This is similar to the case of 2d, but in this case there are 6(3d) instead of 4(2d) possibilities of finding neighbors, this is due to the fact that a particle could move in 6 possible directions, do there it could find neighbors during any of its movement.

The same explanation that was used in 2d, can be applied here, for example, a particle finds a neighbor on the right if an increment in the x coordinate, all other coordinate remains constant, gives us an element from the list, and the same goes for all other five conditions used in the code above.

If a neighbor is found, we check if the y coordinate of the particle is greater than the maximum height of elements in the list, if this is true, this value becomes the new maximum “hmax”, bon is set to zero, so the do-while loop breaks and we move to initializing the next element.

```
    …
   if (i%200==0){
        sprintf(name,"Pictures Folder\\Sim%d.pov",i);
        fp=fopen(name, "w");
       fprintf(fp, "#include \"colors.inc\"\n#include \"shapes.inc\"\n#include \"textures.inc\"\n#include \"metals.inc\"\n");
        fprintf(fp, "background{White}\n");
        fprintf(fp, "union{light_source{<28.94,28.94,-61.38> color White shadowless}\n");
       for (j=0;j<=SL*SL;j++)  fprintf(fp, "sphere { <%d,%d,%d>,0.600\ntexture{ pigment{ color rgb<0.0,0.2,0.0>}\nfinish{ phong 1 metallic}}}\n",part[j].x,part[j].y,part[j].z);
      for (   ;j<=i    ;j++)  fprintf(fp, "sphere { <%d,%d,%d>,0.700\ntexture{ pigment{ color rgb<%f,0.0,0.5>}\nfinish{ phong 1 metallic}}}\n",part[j].x,part[j].y,part[j].z,1.0-(j-SL*SL)/((float)N-SL*SL));
      fprintf(fp,"rotate y*0\ntranslate <0,0,0>\n}\n");
```

```
        fprintf(fp,"camera{\nlocation <%d,%d,%d>\nlook_at  <%d,%d,%d>}\n" ,2*SL,hmax,-(2*SL+10)
,SL/2 -2,hmax/2 -2,SL/2 -2);

        fclose(fp);

         }

         …
```

After a particle has been added to the list, if the value of i is divisible by a constant (e.g. 200), our aim is to use the present amount of particles in the list to create an independent pov file, following the format we learnt early, adjusting the camera location, look at, colors and more , in order to get the best and beautiful image for our eyes.

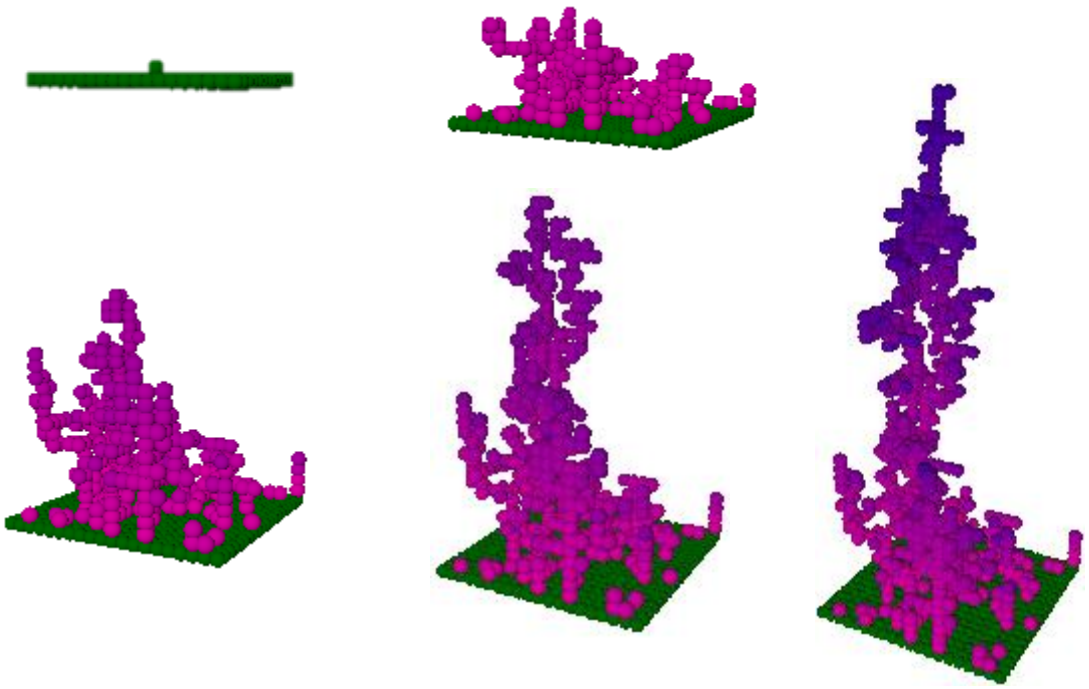

Figure 17: Shows a 3d representation of D.L.A. taken at i equals 400, 600, 800, 1000, 1200 respectively

We open a file for writing, the name of this file depends on the value of i, the file is stored in a folder called “Pictures Folder”, folder is located in the same directory as the source file. The

color of the 2d sticky square which lies on the x-z axis would be green, according to the value above. The camera location and look at are added to the end of the file, we close this file using the fclose() function. That’s it, Basically.

6 Summary

This research work is also a very good example of the application of random walk theory, first we had to learn the concept of bmp files, how they work, the applied format, how they are created, how they can be manipulated. It was important to get familiar with its concepts, so we could have the technical know-how to create and manipulate these files in the later implementation.

When we were done with learning the concepts, it was necessary to implement the random walk, whereby we have a given number of walkers, moving in random motion, the visible part of the walkers would be a fixed length, when any of the walkers reaches an edge, it comes out from the opposite side. All necessary values are being read from a parameter file, to allow easy manipulation, without any need to change the source code. Each walker was made up of squares with a fixed amount for all of them, this is due to the fact that representing only one pixel turned out to be too small, so squares are used instead with a length of 2S+1, snapshots are taken at different time simulations.

Diffusion limited aggregation is simply a process whereby particles obeying the random walk, cluster together to form structures, we could get nice and beautiful images from the coagulation of this particles. In the 2d representation, there is a sticky base line on the x axis, particles are inserted into the system at a dynamic distance, particles begin to move in random motion, if the particle passes the upper set limit, it dies, a new particle is placed in the system.

If the particle reaches any of the edges on the x axis, it comes out from the opposite side, if the particle touches the base line or any of the previous particles in the list, it gets stuck, if it's y coordinate is greater than the maximum height of the elements in the list, this becomes the new maximum, the particle is added to list, another particle is initialized until we reach the desired number of particles. At the end we get a beautiful tree like image that is made by the cluster of particles that were inserted in the system, the color of each particle is dependent of the time, it joined the list.

In the 3d representation, the concept is the same but in this case, there would be 6 directions for movement instead of 4 like the case of 2d, there would be 6 probabilities as well. In order to show the results of the 3d implementation, using an easy method, the concept of ray tracing is being using with the help of the pov-ray application. Initially there is a square shape on the x-z axis of a fixed length, particles are initialized at a dynamic height above the sticky square, if a particle passes the upper limit, it dies, and a new particle is initialized, if a particle reaches any of the edges on the x or z axis it comes out from the opposite edge.

During the random movement of the particle, if it touches the sticky bottom square of any other particles previously in the list, the particle gets stuck, if the y-coordinate of this particle is greater than maximum height of particles in the list, it becomes the new maximum, a new particle is initialized and the process repeats itself again until we reach the desired number of particles. At the multiple of a given number, a new pov file is created and all necessary parameters are put in place, this gives a snapshot of the particles in the list, relating to their coordinate at this time step. At the end we get beautiful/lovely sets of images created by the implementation and being able to be assessed by the help of the pov-ray application.

Random walk theory is a very important concept and can be applied in our daily lives, our choice of career, from the stock market; the movements of a drunkard, the prediction of a fair die, gambling. Other places the random walk theory can be applied in are physics, financial economics, computer networks, computer science, medicine, brain research, physiology and so much more.

7 References

- Applied BMP Format:- https://en.wikipedia.org/wiki/BMP_file_format (Last Visited 14.11.2017)
- Diffusion Limited Aggregation:- https://en.wikipedia.org/wiki/Diffusion-limited_aggregation (Last Visited 14.11.2017)
- Pov-Ray Application:- https://en.wikipedia.org/wiki/POV-Ray (Last Visited 14.11.2017)
- Brownian Motion:- https://en.wikipedia.org/wiki/Brownian_motion (Last Visited 14.11.2017)
- Random Walk:- https://en.wikipedia.org/wiki/Random_walk (Last Visited 14.11.2017)
- One-Dimensional Random Walk:- https://people.smp.uq.edu.au/Infinity/Infinity%2014/Random_Walks.html (Last Visited 14.11.2017)
- Diffusion Limited Aggregation(Paul Bourke):- http://paulbourke.net/fractals/dla/ (Last Visited 14.11.2017)
- Varga, F. Kun: “Computer method for modeling the microstructure of aerogel”, 19th International Conference on Computer Methods in Mechanics, p. 503, (2011)
- Random walk in random and non-random environments (2nd edition) by Pal Revesz, published by World Scientific in 2005
- Elements of random walk by Joseph Rudnick and George Gaspari, published by Cambridge University Press in 2004
- A Random Walk Through Fractal Dimensions (2nd edition) by Brian H. Kaye, published by Wiley-VCH in 1994
- Intersections of Random Walks by Gregory F. Lawler, published by Modern Birkhäuser Classics, the 2013 edition
- The Diffusion Limited Aggregation Model for Fractal Growth by Boaz Kol in 2002

8 Appendix

8.1 Random Walk

```
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include<fcntl.h> //In order to use low level file handling
#include<malloc.h> //In order to use the malloc function
//
FILE *fp;       //Parameter File //Variables are defined respective to file
int lenofs;      //length and height is 44 squares long
int S;           //Side length of each square is 2*S + 1 //If S=2 side is 5x5
int cm;          //Coloring Method e.g. 1
int hmax;       //The length of the head of the snake
int snakenum;   //How many snakes in the garden
int tmax;      //Time Of Simulation
char fileID[10]; //So we can change the file id in every run instead of just
"RW...."
//
int L;          //each square is 2S+1 long
int pixelsize;  //Changing the value of the global variable;
int moveby;     //for to get to the lowerleft corner of the next square, if
S=1 , moveby is 3
//Other Variables
char name[100]; //To store the name of the file
int i,j,k,t,g;
unsigned char *Head; //To store header fields
unsigned char *PA ;  //For the pixel array
float rnum; //random number
float P_up=0.26,P_down=0.25,P_left=0.26,P_right=0.23;
int tmpx, tmpy;
//
void createBmp(){ //To create the first black image
int t=0;
int size; // to store the size
char sig[2] = "BM"; // to store the signature
char name[100];
sprintf(name,"%s%04d.bmp", fileID,t);//printing it for each timeline
int g = open(name,O_RDWR|O_CREAT|O_TRUNC|O_BINARY,S_IRUSR|S_IWUSR); //(Read
https://linux.die.net/man/3/open)
write (g, sig, 2); //BM
size = L*(L*3 + L%4) + 54;
write (g, &size, 4);//size
//printf("%d",size);
int temp;
temp = 0;
write (g, &temp, 4);//Unused
temp = 54;
write (g, &temp, 4);//Pixel Array Offset
temp = 40;
```

```
write (g, &temp, 4);//DIB header size
int W= L, H = L;
write (g, &W, 4);
write (g, &H, 4);
short int tmp; //for 2 bytes  char
tmp = 1;
write (g, &tmp, 2); //Planes
tmp = 24;
write (g, &tmp, 2); //Bits per Pixel
temp =0;
write (g, &temp, 4);//Compression
write (g, &temp, 4); //Image size
temp = 5900;
write (g, &temp, 4); //Horizontal pixel/meter
write (g, &temp, 4); //Vertical pixel/ meter
temp = 0;
write (g, &temp, 4); //Colors in palette
write (g, &temp, 4); //Used palette colors
//
unsigned char *p = (unsigned char*)calloc(pixelsize, sizeof(unsigned char));
//all elements of the pixel array are set to zero
//
free(p);
close(g);
return;
}
//
int main(){
    fp=fopen("param.txt", "rw");//Opening the Parameter file for reading
    fscanf(fp,"%d %d %d %d %d %d
%s",&lenofs,&S,&cm,&hmax,&snakenum,&tmax,fileID); //Read from the param. file
    close(fp); //Close the parameter file
    //
    if (lenofs%2==1) lenofs--; //It works better if it is an even number
    L= (2*S+1)*lenofs; //each square is 2S+1 long
    pixelsize = L*(L*3+L%4);//Changing the value of the global variable;
    moveby = 2*S+1; //so we get to the lowerleft corner of the next square
    //
    int x[snakenum][hmax], y[snakenum][hmax];
    memset(x, 0, sizeof (x)); memset(y, 0, sizeof (y));//initialize all to 0
    srand(time(NULL));//we use time because it constantly changing
    for (i=0;i<snakenum;i++){ //Setting the snake head tip to random
        rnum= rand()%(lenofs); //Choose a random number, 0 - lenofs
        x[i][0]=(2*S+1)*rnum;    //Got a random square, now we store it
        rnum= rand()%(lenofs);
        y[i][0]=(2*S+1)*rnum;
    }
    //
    Head = (unsigned char*) malloc(54);//allocate memory to Head
    createBmp(); //To create the black file
    sprintf(name,"%s%04d.bmp", fileID,t);//Same name with the one created
    g = open(name, O_RDWR);
    read(g,Head,54); //read only the head
```

```
close(g);
//
float c[hmax];  //to store color shades
c[0]=255;
for (i=1;i<hmax;i++){//change the values of the remaining
    if(cm==1)
        c[i]= c[i-1] - 255/(float)hmax; //So the value is decreasing
    if(cm==2)
        c[i]= c[i-1]; //So the value is constant
}
//
t=1;
while (t<=tmax){//Until we reach the maximum time of simulation
    PA = (unsigned char*) calloc(pixelsize, sizeof(unsigned char));
    //Clearing the pixel array at every simulation
   for (i=0;i<snakenum;i++){//representing the particular snake
     for (j=hmax-1;j>=0;j--){
   //we print from the back, so the head is always visible
        tmpx=x[i][j];tmpy=y[i][j];
        while (1){//for the column
         while(1){//for the row
            PA[tmpy*(L*3 + L%4) + tmpx *3 + 0 ] = c[j]; //Blue Pixel
            PA[tmpy*(L*3 + L%4) + tmpx *3 + 1 ] = c[j];  //Green Pixel
            PA[tmpy*(L*3 + L%4) + tmpx *3 + 2 ] = c[j]; //Red Pixel
            if (tmpx == (x[i][j] + moveby -1)){
              //If we have reached the right edge
                tmpx = x[i][j];
                break;
            }//If
            tmpx++;//increment by 1, it is just the coordinate
            }//While
         if (tmpy == (y[i][j] + moveby -1)) break;
        //if we reach the upper edge
         tmpy++; //If we are not at the edge we increment
        }//While
    }//for
    }//for
    //
    sprintf(name,"%s%04d.bmp", fileID,t);
  //To create a new file with the amendment
    g = open(name,O_RDWR|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
    write(g,Head,54);
    write(g, PA, pixelsize);
    close(g);
    //
    for (i=0;i<snakenum;i++){
      for(j=hmax-1;j>0;j--){//Shift every element to the right by one step
        x[i][j]=x[i][j-1];// x[i][1]=x[i][0]
        y[i][j]=y[i][j-1];
    }}
    //
for (i=0;i<snakenum;i++){ //Amend the zeroth coordinate of all snakes
  rnum=(float)rand()/RAND_MAX; // 0 <= x <=1
```

```
    if( rnum < P_up) {
      y[i][0]+= moveby; //Move Up
      if(y[i][0] ==L){////Not affected the same way as left or down
         y[i][0]=0;
        }}//if
    else if(rnum >= P_up && rnum <P_up+P_down) {
      y[i][0]-= moveby; //Move down
      if(y[i][0] == 0-moveby){ //We check for lowerleft corner
         y[i][0]=L-moveby;//we continue from the lowerleft
      }}//else if
    else if(rnum >= P_up+P_down && rnum <P_up+P_down+P_left){
      x[i][0]-= moveby; //Move left
      if(x[i][0] == 0-moveby){//Same as the statement for down
         x[i][0]=L-moveby;
      }}//else if
    else if(rnum >=P_up+P_down+P_left ) {
      x[i][0]+= moveby; //Move Right
      if(x[i][0] ==L){ //Not affected the same way as left or down
         x[i][0]=0;
      }}//else if
   }//For
      //
   t++;
   }//While
  return 0;
  }
```

8.2 Diffusion Limited Aggregation

8.2.1 2d version

```
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include <dirent.h>
#include<fcntl.h> //In order to use low level file handling
#include<malloc.h> //In order to use the malloc function
//
int L =256;
int pixelsize;
int i,j,t;
//unsigned char *Head; //To store header fields
unsigned char *PA ;//For the pixel array
float rnum; //random number
float P_up=0.23,P_down=0.28,P_left=0.26,P_right=0.23;
int tmpx, tmpy;
int bon;//if 1 don't break, Break if 0
//
void createBmp(){ //To create the first black image
int t=0;
int size; // to store the size
char sig[2] = "BM"; // to store the signature
```

```
char name[100];
sprintf(name,"%s%04d.bmp", fileID,t);//printing it for each timeline
int g = open(name,O_RDWR|O_CREAT|O_TRUNC|O_BINARY,S_IRUSR|S_IWUSR); //(Read
https://linux.die.net/man/3/open)
write (g, sig, 2); //BM
size = L*(L*3 + L%4) + 54;
write (g, &size, 4);//size
//printf("%d",size);
int temp;
temp = 0;
write (g, &temp, 4);//Unused
temp = 54;
write (g, &temp, 4);//Pixel Array Offset
temp = 40;
write (g, &temp, 4);//DIB header size
int W= L, H = L;
write (g, &W, 4);
write (g, &H, 4);
short int tmp; //for 2 bytes  char
tmp = 1;
write (g, &tmp, 2); //Planes
tmp = 24;
write (g, &tmp, 2); //Bits per Pixel
temp =0;
write (g, &temp, 4);//Compression
write (g, &temp, 4); //Image size
temp = 5900;
write (g, &temp, 4); //Horizontal pixel/meter
write (g, &temp, 4); //Vertical pixel/ meter
temp = 0;
write (g, &temp, 4); //Colors in palette
write (g, &temp, 4); //Used palette colors
//
write(g,PA,pixelsize);
//
close(g); free(PA);
return;
}
//
struct particle{ //Use to store the coordinates of particles
    int x;
    int y;
};
//
int main(){
    pixelsize = L*(L*3+L%4);// Total number of pixels
    PA = (unsigned char*) calloc(pixelsize, sizeof(unsigned char));
    //Initiallizing the pixel array to zero
    //
    srand(time(NULL));//we use time because it changes everytime
    int N=L*10; //Making sure N is a very large number than L
    struct particle part[N];//An array to store all particles, 0,1,2,until N
    int hmax=0; //Initial height is zero
```

```
    //
    for (i=0;i<N;i++){
      bon=1;                        //bon must be one before the do-while
      if (i<L){                     //Storing the bottom pixels
        part[i].x = i; part[i].y=0; //The y coordinate remains 0
      }//If
      else{
        part[i].x = rand()%L; part[i].y= hmax + 1;
        tmpy=hmax+1+2; //The maximum height a particle can reach before dying
        do{//Loop was created for the movement of a particle
            rnum=(float)rand()/RAND_MAX; // 0 <= x <=1 //Gets random number
            if (rnum < P_up) part[i].y+=1;
            else if (rnum >= P_up && rnum < P_up+P_down) part[i].y-=1;
            else if (rnum >= P_up+P_down && rnum < P_up+P_down+P_left )
part[i].x-=1;
            else part[i].x+=1;
            //
            if (part[i].y >= tmpy || part[i].y<=0 || part[i].x<=0){
            //If it has reached the upper limit
               part[i].x = rand()%L; part[i].y= hmax+1; //New Particle
               continue;                          //Starts do while loop again
            }
            //
            for (j=0;j<i;j++){//Checking if they are neighbors
               if ((part[i].x + 1 == part[j].x  && part[i].y==part[j].y)||
                   (part[i].x - 1 == part[j].x  && part[i].y==part[j].y)||
                   (part[i].y + 1 == part[j].y  && part[i].x==part[j].x)||
                   (part[i].y - 1 == part[j].y  && part[i].x==part[j].x)){
                  //Checking with the right, left, up and down respectively
                   if (part[i].y > hmax) hmax=part[i].y;
                        //if we have a new hmax
                   bon=0;
                   break; //Leave present for loop
                }//If
            }

        }while(bon); //bon is 0 so loop stops
        }//Else
        //
        //Store the new neighbour in the pixel array
        PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 0 ] = 128; //Blue Pixel
        PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 1 ] = 0;  //Green Pixel
        PA[part[i].y*(L*3 + L%4) + part[i].x *3 + 2 ] = (i/(float)N) * 255;
                                                                  //Red Pixel
    }//For
   //
    createBmp();//Printing the final product after all manipulation on a
picture
    return 0;
    }
```

8.2.3 3d version

```
#include <stdio.h>
#include <stdlib.h>
#include<sys/stat.h>
#include <dirent.h>
#include<fcntl.h> //In order to use low level file handling
#include<malloc.h> //In order to use the malloc function
//
int L =256;
int pixelsize;
int i,j,t;
char name[100];
float rnum; //random number
float P_up,P_down,P_left,P_right,P_forward,P_backward;//Sum must be equal to 1
int tmpx, tmpy,count=0;
int bon;//if 1 don't break, Break if 0
//
struct particle{ //To store the coordinates of particles
    int x;
    int y;
    int z;
};
//
int main(){
    FILE *fp; //The file pointer for the pov file
    //
    srand(time(NULL));//we use time because it changes everytime
    int N=L*5; //Making sure N is a very large number than L
    struct particle part[N];// To store all particles, 0,1,2, until N
    int hmax=0; //Initial height is 0
    //
    int SL=20; //The side length of the bottom square
    for (i=0;i<SL;i++){ //The square bottom layer in 2D 20x20 (0 still 19)
        for (j=0;j<SL;j++){
            part[count].x=j; part[count].y=hmax=0; part[count].z=i;
            count++;
        }
    }
    //
    P_up=P_down=P_left=P_right=P_forward=P_backward= 1.0/6.0; //Sum is 1
    for (i=count;i<N;i++){
        bon=1;                          //bon is initially 1
        part[i].x = rand()%SL; part[i].y= hmax + 5; part[i].z= rand()%SL;
        tmpy=hmax+5+4; //The maximum height a particle can reach before dying
        do{//Loop was created for the movement of a particle
            rnum=(float)rand()/RAND_MAX; // 0 <= x <=1 //Gets random number
            if (rnum < P_up) part[i].y+=1;
            else if (rnum >= P_up && rnum < P_up+P_down) part[i].y-=1;
            else if (rnum >= P_up+P_down && rnum < P_up+P_down+P_left )
part[i].x-=1;
            else if (rnum >= P_up+P_down+P_left && rnum <
P_up+P_down+P_left+P_right) part[i].x+=1;
```

```
            else if (rnum >= P_up+P_down+P_left+P_right && rnum <
P_up+P_down+P_left+P_right+P_forward) part[i].z+=1;
            else if (rnum >= P_up+P_down+P_left+P_right+P_forward &&
rnum<P_up+P_down+P_left+P_right+P_forward+P_backward) part[i].z-=1;
            //
            if (part[i].y >= tmpy || part[i].y<0){
            //If it has reached the upper limit
               part[i].x = rand()%SL; part[i].y= hmax+5; part[i].z= rand()%SL;
               //New Particle
               continue;                          //Starts do while loop again
            }
            //x and z coordinate are periodical
            if (part[i].x >= SL) part[i].x=0;
            else if (part[i].x < 0) part[i].x=SL-1;
            if (part[i].z >= SL) part[i].z=0;
            else if (part[i].z < 0) part[i].z=SL-1;
            //
            for (j=0;j<i;j++){//Checking if they are neighbors
               if ((part[i].x + 1 == part[j].x  && part[i].y==part[j].y &&
part[i].z==part[j].z)||(part[i].x - 1 == part[j].x  && part[i].y==part[j].y &&
part[i].z==part[j].z)||(part[i].y + 1 == part[j].y  && part[i].x==part[j].x &&
part[i].z==part[j].z)||(part[i].y - 1 == part[j].y  && part[i].x==part[j].x &&
part[i].z==part[j].z)||(part[i].z + 1 == part[j].z  && part[i].x==part[j].x &&
part[i].y==part[j].y)||(part[i].z - 1 == part[j].z  && part[i].x==part[j].x &&
part[i].y==part[j].y)){ //Check if neighbors with previous elements
                  if (part[i].y > hmax) hmax=part[i].y; //new hmax or not
                  bon=0;
                  break; //Leave present for loop
                }//If
            }//for
         }while(bon); //bon is 0 so loop stops
         //
      if (i%200==0){

            sprintf(name,"Pictures Folder\\Sim%d.pov",i);
            //Creating pov files after every given simulation
            fp=fopen(name, "w");
            fprintf(fp, "#include \"colors.inc\"\n#include
\"shapes.inc\"\n#include \"textures.inc\"\n#include \"metals.inc\"\n");
            fprintf(fp, "background{White}\n");
            fprintf(fp, "union{light_source{<28.94,28.94,-61.38> color White
shadowless}\n");

            for (j=0;j<=SL*SL;j++) fprintf(fp, "sphere {
<%d,%d,%d>,0.600\ntexture{ pigment{ color rgb<0.0,0.2,0.0>}\nfinish{ phong 1
metallic}}}\n",part[j].x,part[j].y,part[j].z);
            for (   ;j<=i    ;j++) fprintf(fp, "sphere {
<%d,%d,%d>,0.700\ntexture{ pigment{ color rgb<%f,0.0,0.5>}\nfinish{ phong 1
metallic}}}\n",part[j].x,part[j].y,part[j].z,1.0-(j-SL*SL)/((float)N-SL*SL));

            fprintf(fp,"rotate y*0\ntranslate <0,0,0>\n}\n");
            fprintf(fp,"camera{\nlocation <%d,%d,%d>\nlook_at  <%d,%d,%d>}\n"
,2*SL,hmax,-(2*SL+10) ,SL/2 -2,hmax/2 -2,SL/2 -2);
```

```
        fclose(fp);
    }//If
}//For
return 0;
}
```