<Programming Assignment #2>

21800325 Hayun Park
21800001 Narin Kang
21500359 Suegeun Song
21800601 HyunGyu Lee
21800511 MinJae Yi

1. Fractal

A fractal is a geometric object that has an infinitely complex pattern whose structure is identical to its own components. It is created by repeating a pattern in an ongoing loop.

With javascript in html, the following problems are drawing fractals. Finding the repeated pattern is the most crucial step toward the solution in order to put the pattern in a recursion.

More details about each step is as follows:

1. Analyze a result figure which is given through the instruction.
2. Find the repeated a pattern.
3. Implement the pattern into a recursion form.

1.1 Fractal 1: Intersection

The fractal could be divided into four sections: right, left, bottom, and top, and its base shape is as Figure I.

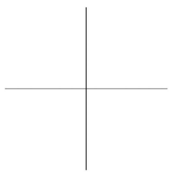

Figure I *a base shape of Fractal 1*

The base shape is intersection of two lines. If the depth is increased by one, four lines are added in each section. Then, if it is increased by one more, 12 lines are added and then 36 lines as shown by Figure 2. By analyzing the direction of each basic unit, it also has a pattern for each section. Therefore, I divided the whole figure into four sections and then divide these sections into another three parts depending on each direction in order as follows:

```
drawUpper(x1, yU, dx/2, dy/2, ratio, depth - 1);
drawBottom(x1, yB, dx/2, dy/2, ratio, depth - 1);
drawLeft(xL, y1, dx/2, dy/2, ratio, depth - 1);
```

The above functions are codes that are exploited in the drawLeft function. As shown through the code, each direction of the whole figure contains three small directions again, excluding its reversed direction.

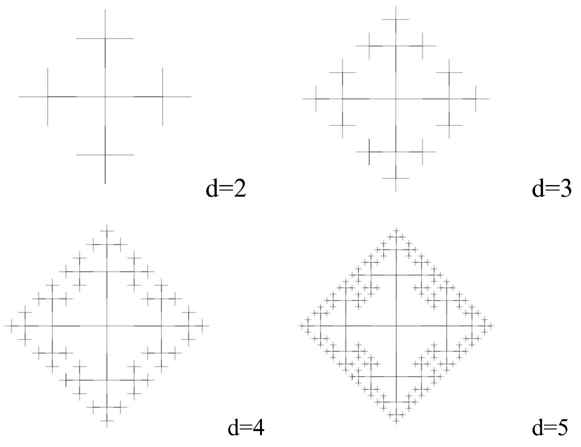

Figure II *Fractal 1 when depth is 2, 3, 4, and 5 (demonstration)*

1.2 Fractal 2: Flakes

The base shape of this fractal is composed of four lines. Unlike the first fractal, it requires measuring an angle between two lines.

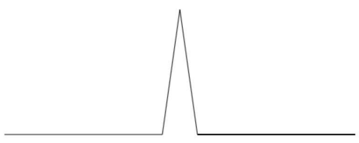

Figure III *a base shape of Fractal 2*

After some trials, the analyzed recursive patterns are as follows:

- From the center of the line, the length is measured as 9:10 so that 1:10 is left out from the center for each direction.

```
var x3 = (9*centerX+x1)/10;
var y3 = (9*centerY+y1)/10;
```

- The angle is measured based on the midpoint where atan2() function is used.

```
var angle = Math.atan2(-dy,dx) + Math.PI/2;
var x5 = centerX + len * Math.cos(angle);
var y5 = centerY - len * Math.sin(angle);
```

- Every drawn line includes another base shape on it, so four recursions are called (one recursion for one line).

```
fractal(x1, y1, x3, y3, depth - 1,  len/2);
fractal(x3, y3, x5, y5, depth - 1,  len/2);
fractal(x5, y5, x4, y4, depth - 1, len/2);
fractal(x4, y4, x2, y2, depth - 1, len/2);
```

  - 'len' is the length of each line which is cut in half in each recursion.

- Since there should be no line below a vertex, drawLine() function is activated from when the depth becomes one. In another word, when depth is greater than one, the recursions are called without drawing.

```
if (depth !== 1){
    fractal(x1, y1, x3, y3, depth - 1,  len/2);
            ...
  } else{
    drawLine(x1, y1, x3, y3);
            …
}
```

In order to solve the problem, the above approach is taken. Despite of a trial of getting an angle from the input, it is failed due to the miscalculation of a length based on each angle. The demonstration of the problem is shown through Figure IV.

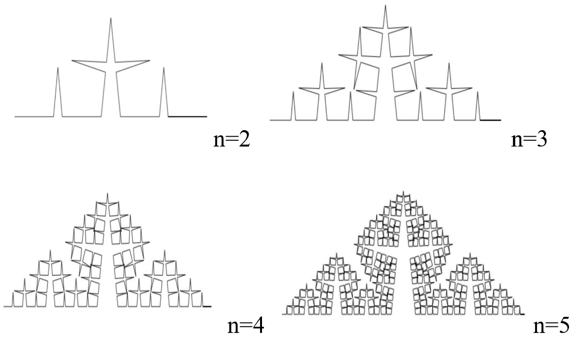

Figure IV *demonstration of Fractal 2*

1.3 Fractal 3: Creative Fractal

The base shape of the third Fractal is composed of six hearts which are centered on a single point as Figure V.

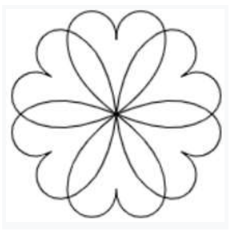

Figure V *a base shape of Fractal 3*

To draw this, we declare a function named drawHeart and receive the heart's base point (x,y) and the heart's width(w) and height(h) as variables. Then draw a heart using the quadraticCurveTo(cpx, cpy, x, y) method and bezierCurveTo(cpx1, cpy1, cpx2, cpy2, x, y) method which are secondary curved methods as follows:

```
context.quadraticCurveTo(x-w/2, y-h/3, x-w/2,
y-2*h/3);                  //left bottom of heart
context.bezierCurveTo(x-w/2, y-h, x, y-h, x,
y-2*h/3);                  //left top of heart
context.bezierCurveTo(x, y-h, x+w/2, y-h, x+w/2,
y-2*h/3);                  //right top of heart
context.quadraticCurveTo(x+w/2, y-h/3,  x, y);
                           //right bottom of heart
```

After drawing one heart, turn the canvas 60 degrees using rotate() method. One heart is drawn each time the drawHeart() function is executed. We used recursion to create a base shape by adding a node called 'leaf' to the function and reducing the leaf value by one. When leaf is zero, recursive execution of functions end.

To draw hearts with different sizes, we declare an additional doRecursion() function containing the drawHeart() function.

Each time the depth is reduced by one, the width and height of the heart are multiplied by 0.8 respectively. The change in overall shape according to the depth is shown through Figure VI.

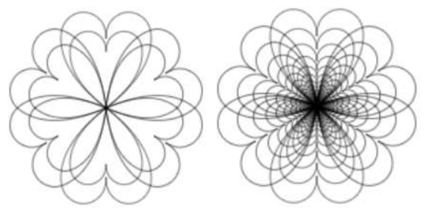

Figure VI *demonstration of Fractal 3*

After implementing the basic shape of the fractal 3, we used the fill method to make the fractal more beautiful. Each time the doRecursion(x, y, w, h, depth) function executes, it randomly generates a six-digit hexadecimal string and makes color code by adding a hash (#) at the front of the 6 digits. Then fills the shape with its color code. The results are as follows:

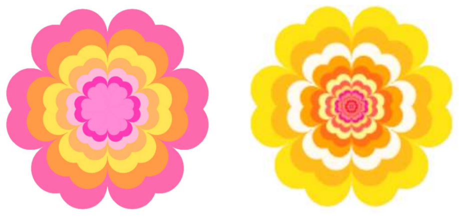

Figure VII *colored version of Fractal 3*

2. Triangulation

Using Javascript, judging simple polygon and do triangulation. For these tasks, more details are as follows:

~~1.~~ Whether Simple polygon it is
2. Draw all outlines
3. Do triangulation in polygon

2.1 Simple polygon

The simple polygon is closed polygonal chain of line segments that do not intersect each other. It means that there is no meet among outlines. So, a meet between two lines indicate that the polygon is not simple.

For finding existence of a meet, we need two lines, which are from four points. Let indicate them as

P1(x1, y1) P2(x2, y2)
P3(x3, y3) P4(x4, y4)

The line through P1,P2 can be written as

(y-y1)(x2-x1)-(x-x1)*(y2-y1)=0*

Let P3 and P4 assign, multiple value of them. If the sign of value is negative, there is a meet. So it indicate that the polygon is not simple.

For n points, we have to perform it from n-2th point to 2nd point. If it is simple, Then we can draw outlines.

2.2 Outline

The points are stacked and connected sequentially. Choosing two points recursively, draw outlines.

2.3 Triangulation

The triangulation is the division process of a map or a plan into triangles. By drawing lines, the outcome is made. There are many ways to implement it. But, Calculating the angle of polygon is our choice.

Here is a process.
- Choose three points sequentially.
- Calculating the inner angle.
- Draw lines to triangulate.

Convex polygon is a polygon that has all interior angles less than 180°. Contrary, concave polygon is a polygon that has at least one interior angle which is more than 180° and less than 360°.

If the polygon is convex polygon, it is easy to triangulation. From one point, draw lines to points not connected to the point directly.

The hard part is to triangulate the concave polygon. If we detect the concave part of polygon, we have to draw lines from the concave points. By drawing from them, we do triangulate.

3. CNF Converter

3.1 Summary

In this problem, we created a program that converts the normal formula "(operator operand operand ..)" into CNF formula by using binary tree structure and recursive functions. There are three major steps to convert the normal formula into CNF formula.

1. Read and insert data into binary tree structure.
2. Apply De Morgan's Laws and convert the tree into NNF.
3. Use distributive law to convert NNF into CNF.

3.2 Problem analysis

3.2.1

Each of node in a binary tree contains data, and data can be either an operator or an operand. Operator has three value AND, OR, and NOT. If the parent node is an operator, the child is define as an operand of the parent. By this structure we can apply any propositional statement as binary tree structure. we implemented such task with function makeTree(). In makeTree(), we recursively call the function whenever we read left parenthesis to make the subsequent part to be the child of current node.

3.2.2

In NNF formula NOT operator can only be placed in front of the operand. To exclude NOT operator from other operator adjust De Morgan's Laws. To exclude NOT operation from the node, every child of the NOT operation should be the opposition of itself. For example AND should be changed into Or, and a0 in to -a0. When the number of NOT is even, there is no change among childs

3.2.3

CNF formula should be a conjunction of one or more clauses, where a clause is a disjunction of operand. To implement CNF formula, we must apply distributive law. For instance, in P v Q form, P must have the form P1 ^ P2 ^ … ^ Pm, whereas Q must have the form Q1 ^ Q2 ^ … ^ Qn. After that P v Q becomes (P1 ^ P2 ^ … ^ Pm) v (Q1 ^ Q2 ^ … ^ Qn). Then we can change it into

(P1 v Q1) ^ (P1 v Q2) ^ … ^ (P1 v Qn)
^ (P2 v Q1) ^ (P2 v Q2) ^ … ^ (P2 v Qn)
…
^ (Pm v Q1) ^ (Pm v Q2) ^ … ^ (Pm v Qn).

3.3 Solution Overview

3.3.1.makeTree() : makeTree is designed for convert Input Text into binary tree structure

```
tree_node *makeTree(){
  while(1){
    ...
    if(str=='(') {
    child =  makeTree()
    …} else if(str is ')') return ptr;
 }}
```

3.3.2.tree_node * cvt2NNF(tree_node*,int) : Convert Binary Tree to NNF

```
tree_node * cvt2NNF(tree_node* ptr ,int
change){ // change is initialized as 0
if(check both child is NULL){
  if(change is odd) { convert . . . }
  return ptr
}
if(change is odd){ convert. . . }
if(ptr->data == not){
   change++
 if(change == 0){
   convert. . .
}
  return(ptr->leftchild,change)
}
ptr -> leftchild = cvt2NNF(ptr->leftchild,
change)
ptr -> rightchild = cvt2NNF(ptr->leftchild,
change)
}
```

3.3.3. NNF2CNF() : Convert NNF to CNF

```
NNF2CNF(){
  check=checkIfItisOnlyBuiltByOrOperation()
  //check whether the child and itself is only
consist with operand or and OR operator
  if(check==0) return ptr;
  left_child = NNF2CNF(left_child);
  right_child = NNF2CNF(right_child);
  …..
  //operate distribution
  return ptr;
  }
```

3.4 Demonstration

```
input : (or a1 (not (or (not (or a2 a3)) a4)))
output: 1 2 3
        1 -4
input:
output:
```

4. Discussion

1. Fractal

On the first trial of Fractal 1, we considered a complex form as a base shape, so there was some hard-coding. After we realized what the base shape was for the problem, we could remind the definition of a fractal.

While writing the code, we built four sub recursive functions and one recursion that combined them, but we think there would be a better way not to recursively call other functions but just find a pattern in it which we could not find in a simple way.

In Fractal 2 although we used Cosx and Sinx to position a vertex, one factor that we neglected was that the vertex should be placed on the midpoint of the line.

2. Triangulation

The detection of intersection and draw outlines recursively are fun for implement. By assigning discrete number sequentially, it is much helpful.

Here are if-statements what if. What if the number of points are not listed? Surely, It'll be much more harder than before.

3. CNF Converter

While struggling through this problem, we could acknowledge the functionality of recursion in depth. In specific, It was confusing when we had to check the output values of the tree since we utilized tree traversal to check the output. It was easy to check one by one when the tree was in small size, but once it became a little bigger, it was really confusing to check the location of the nodes. We should follow the process of recursion one by one to make sure. Through this experience we could observe how recursion works by simply checking the results out. Also, we observed that we could have the same value for AND and -2, Or and -1 since we represented the data of AND and OR to be -2 and -1. To resolve this problem, we created a function that determines whether the value is operand and operator.