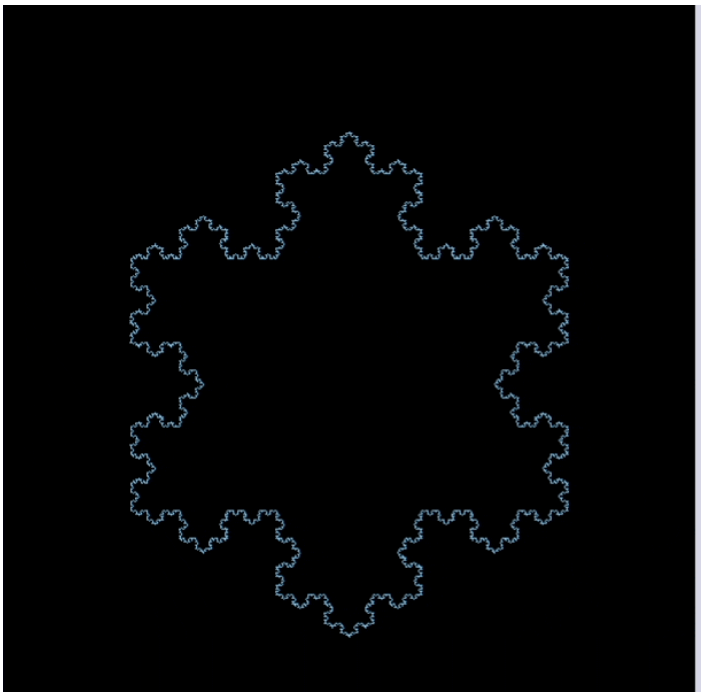
Nicki Wahlers
Exploration of Fractals
Summer 2022
Final Portfolio

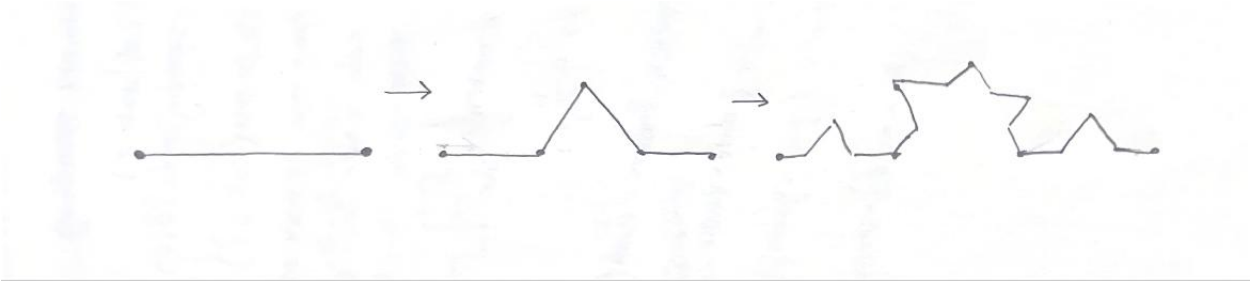
1. Recursive – The journey of a Koch snowflake

For the recursive entry of this portfolio, we will look at a snowflake made of Koch curves.



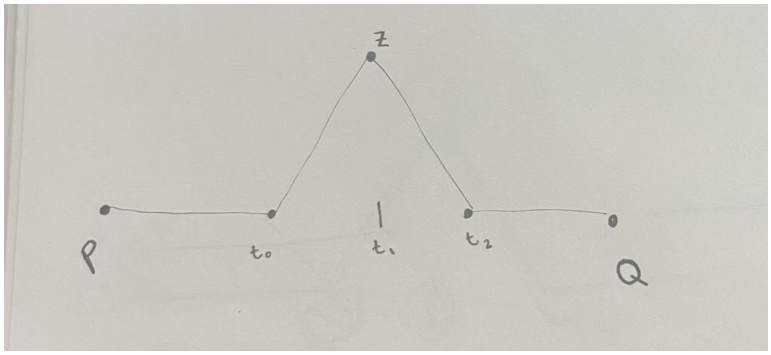
1.1 Background:

The Koch curve and the resulting Koch snowflake were originally described by Helge von Koch in 1903. The curve is made originally by one line, broken into 3 equal parts, and then an equilateral triangle being made by the middle part. Using recursion we can repeat this process with every line segment made by the previous iteration.



The above diagram shows repeating this process until we get to a depth of 3. To achieve the full Koch Snowflake, this process is done three times on an equilateral triangle with each side being its own original Koch Snowflake.

1.2 Design Paradigm & Mathematical Description:



To achieve the Koch curve, we can think in a few steps. We begin with the two end points of the line: P and Q, and then find the midpoint, the ⅓ point and the ⅔ point of these two points with

Dx = Qx - Px;
Dy = Qy - Py;

T0x = Px + ⅓(Dx);
T0y = Py + ⅓(Dy);

Similar for T1 and T2, but replace the fraction with ½ and ⅔ to get the midpoint and ⅔ point. You get it.

Now to find the z point, the tip of the equilateral triangle we will use the relationship of similar triangles.

We can use f = ½(√3a) where a is the length of one side of the equilateral triangle. We know that one side of our equilateral triangle is ⅓ of the entire Length from P to Q, so

F = ½ * ⅓ L *√3 = √3/6 L

So using this F as our scale factor, we can find the Z point with:

Zx = T1x - (f*Dy);
Zy = T1y + (f*Dx);

And in the most simple case as shown above, draw the lines between the points P, T0, Z, T2, and Q.

Now to add in the Recursive part of this problem, we want a function that does all of the above calculations and calls itself on all of the babies. An example of a baby is the line from P to T0. We do this until we have reached the depth of recursion we define above. Once we arrive at the depth of recursion we defined, it is time to draw just one line segment between the smallest points, and then back out of the recursive call, drawing the

lines as we back out. This ensures there is no line along the bottom of the equilateral triangle defined by T0 to T2.

To make the final snowflake image, I made three Koch curves following the outline of an equilateral triangle, ensuring that the points were being passed in the correct order so that all of the curves curve outward rather than into the triangle.

1.3 Artistic Description

For the artistic enhancement, I made a moving image that is like an overhead view of a snowflake as it falls from the clouds to the Earth, from the point of view of the cloud. For other paradigms we covered in class after the Recursive one, we learned about transformations such as scale, rotate, and translate. I borrowed these functions, modified them a bit to fit the format of this recursive problem, and then called them in succession to make the snowflake appear like it is falling away from the viewer.

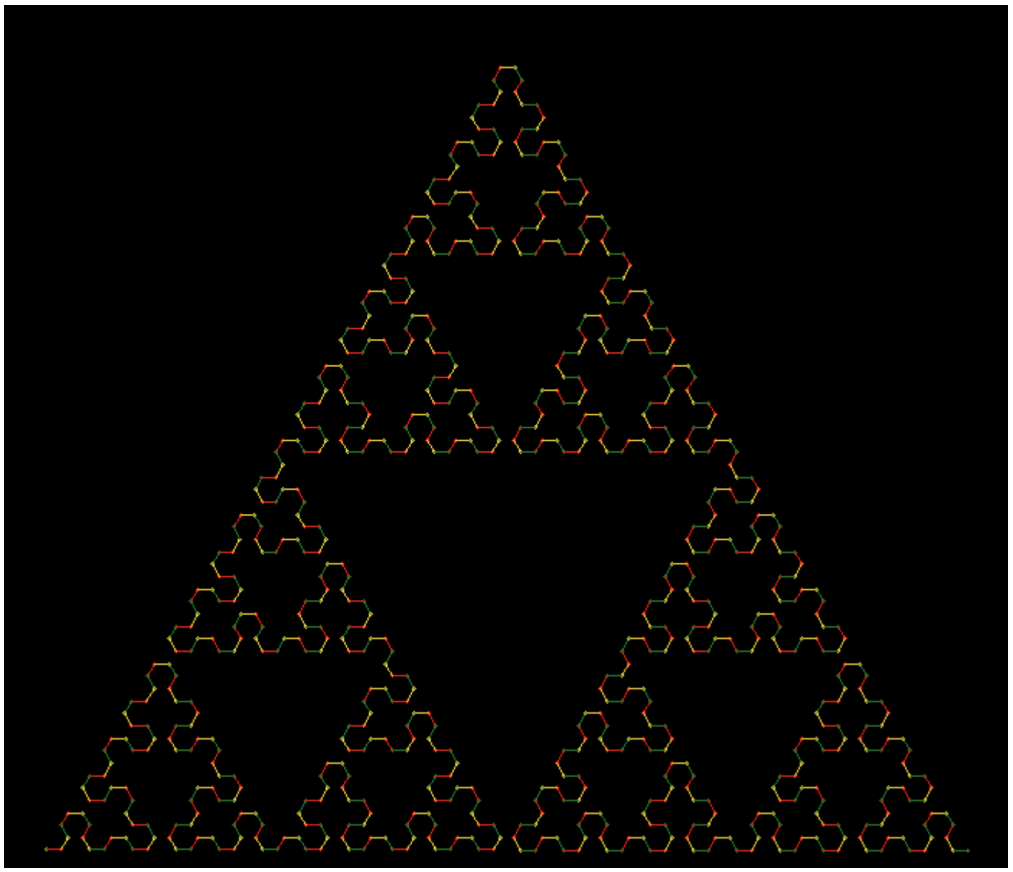
1.4 Code:

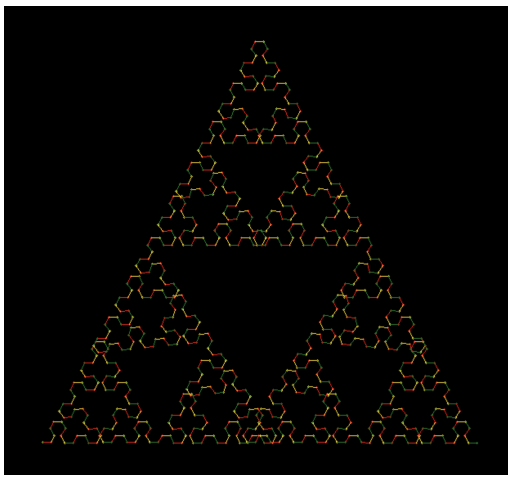
```
#include "FPToolkit.c"

double p[2], q[2];
double t[2], z[2];
int Wsize = 600;


void koch(int depth, double p0[], double p1[]) {
	if (depth == 0) {
		G_line(p0[0], p0[1], p1[0], p1[1]);
		return;
	}

	double dx = p1[0] - p0[0];  // x distance between original points
	double dy = p1[1] - p0[1];  // y distance between original points
	double t0[2], t1[2], t2[2]; // 1/3 and 2/3 and midpoint
	double z[2];                      // the top point

	double f = 0.5; // half

	t0[0] = p0[0] + (1.0 / 3) * dx; // 1/3 point x
	t0[1] = p0[1] + (1.0 / 3) * dy; // 1/3 point y val

	t1[0] = p1[0] - (1.0 / 3) * dx; // 2/3 point x
	t1[1] = p1[1] - (1.0 / 3) * dy; // 2/3 point y

	t2[0] = p1[0] - (1.0 / 2) * dx;
	t2[1] = p1[1] - (1.0 / 2) * dy; // midpoint

	f = sqrt(3) / 6;
```

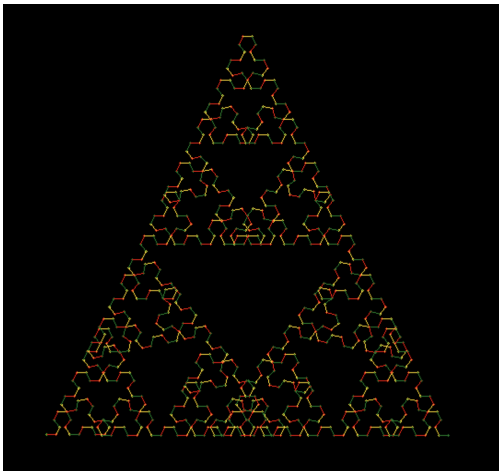
```
        z[0] = t2[0] - (f * dy);
        z[1] = t2[1] + (f * dx);

        koch(depth - 1, p0, t0);
        koch(depth - 1, t0, z);
        koch(depth - 1, z, t1);
        koch(depth - 1, t1, p1);
};

void translate(double dx, double dy) {
        p[0] = p[0] + dx;
        p[1] = p[1] + dy;

        q[0] = q[0] + dx;
        q[1] = q[1] + dy;

        z[0] = z[0] + dx;
        z[1] = z[1] + dy;
}

void rotate(double deg) {
        double r, a;
        double t = deg * (M_PI / 180);
        double temp;
        double c, s;
        c = cos(t);
        s = sin(t);

        temp = (p[0] * c) - (p[1] * s);
        p[1] = (p[1] * c) + (p[0] * s);
        p[0] = temp;

        temp = (q[0] * c) - (q[1] * s);
        q[1] = (q[1] * c) + (q[0] * s);
        q[0] = temp;

        temp = (z[0] * c) - (z[1] * s);
        z[1] = (z[1] * c) + (z[0] * s);
        z[0] = temp;
}

void scale(double s) {
        p[0] *= s;
        p[1] *= s;
        q[0] *= s;
        q[1] *= s;
        z[0] *= s;
        z[1] *= s;
}


int main() {
 char fname[100] ;

        G_init_graphics(Wsize, Wsize); // interactive graphics
```

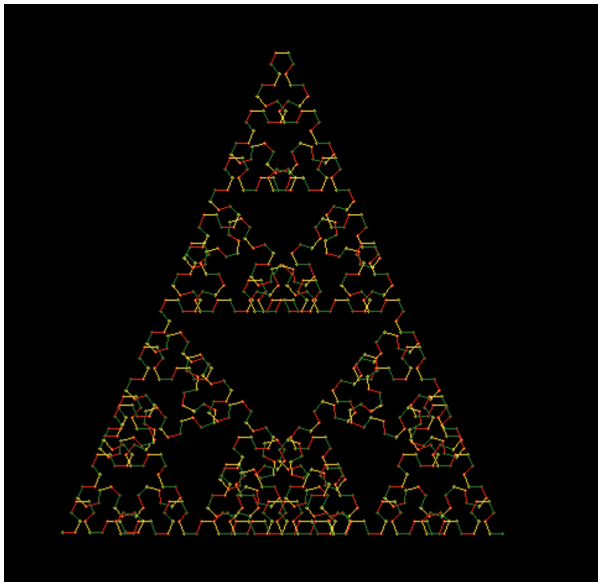
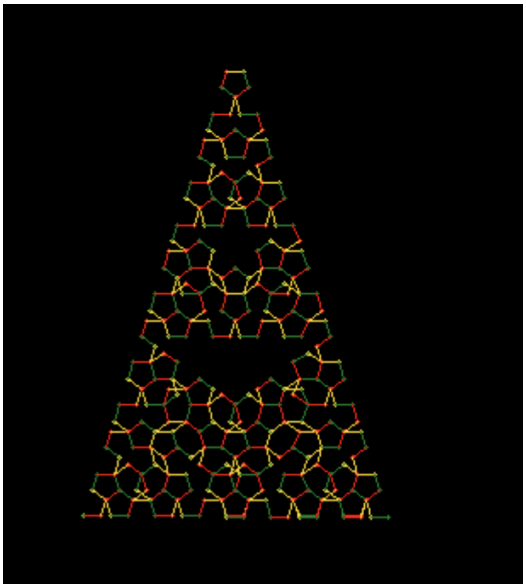
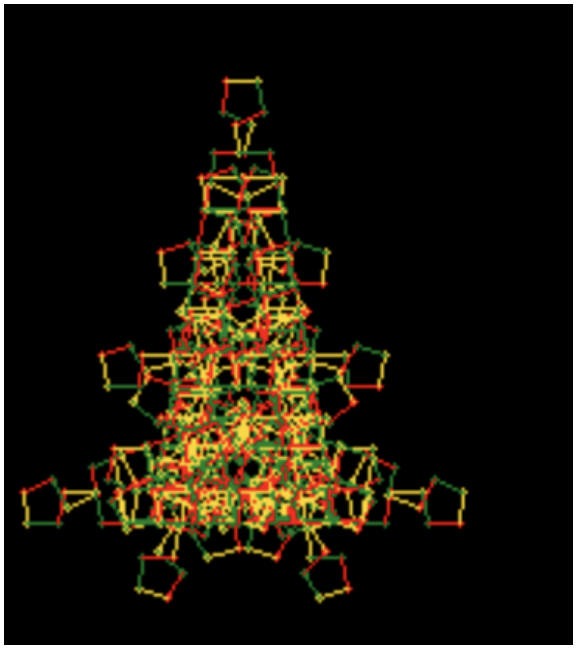
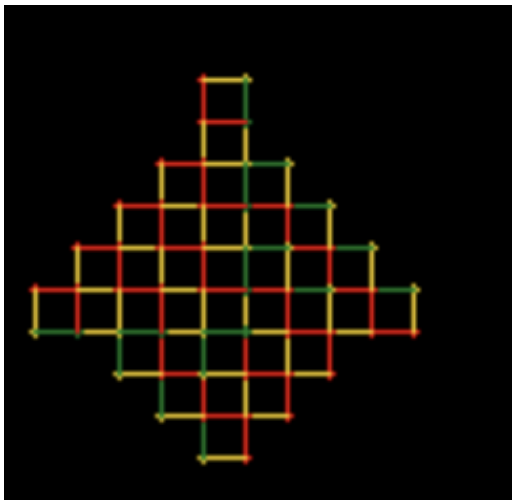
```
      G_rgb(0, 0, 0);
      G_clear();

      p[0] = 100; p[1] = 150;
      q[0] = 500; q[1] = 150;

      double dx = q[0] - p[0];
      double dy = q[1] - p[1];

      t[0] = p[0] + 0.5 * dx;
      t[1] = p[1] + 0.5 * dy;
      double f = sqrt(3) / 2;
      z[0] = t[0] - f * dy;
      z[1] = t[1] + f * dx;

      G_rgb(0, 0, 0);
      G_clear();

      for (int i = 0; i < 60; ++i) {
            translate(-Wsize / 2, -Wsize / 2);
            scale(0.95);
            rotate(2 * i);
            translate(Wsize / 2, Wsize / 2);

            G_rgb(0.53, 0.8, 0.98);

            koch(5, q, p);
            koch(5, z, q);
            koch(5, p, z);
   sprintf(fname, "sun%04d.bmp",i) ;
   G_save_to_bmp_file(fname) ;
            G_wait_key();
            G_rgb(0, 0, 0);
            G_clear();
      }

      int key;
      key = G_wait_key();
}
```

2. **L-System – Sierpinski with different angles**







2.1 Background:

Throughout this course, we returned over and over to the Sierpinski triangle in many of the different paradigms we discussed. It became a useful tool in understanding the different paradigms once we understood the math. Named after Polish mathematician Wacław Sierpiński, it can be described as one large equilateral triangle, and then repeatedly cutting equilateral triangles out of the middle by finding the center point of each line of the parent triangle, and forming the cut out by connecting those midpoints. This would be the method in the recursive paradigm, but the method of achieving this fractal with an L system is different, but with the same final result.

2.2 Design Paradigm & Mathematical Description:

The L system method of displaying a fractal is about using grammars which is a set of axioms and rules that define strings to produce a fractal. This method involves defining an Axiom to begin with, a set of rules for how to replace letters in that axiom, and then using these rules to form strings to direct our turtle in its path across the screen. The different characters in a string have different significances so the turtle knows where to walk so to speak. The rules we used as a class are,

a. Capital letters A-Z mean to move forward a defined number of units
b. + means to turn counterclockwise a defined angle
c. - means to turn clockwise by the same defined angle.

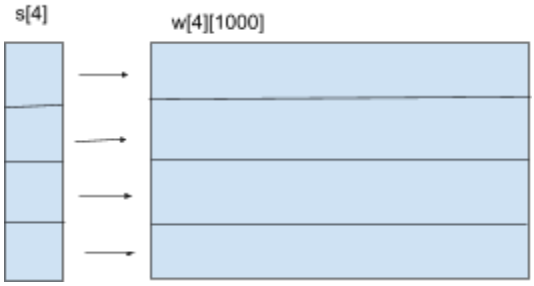
The grammar for Sierpinski's triangle is as follows:

Axiom: A

A → B-A-B

B → A+B+A

Once we have defined the grammar, we can run through this n number of times to form a final string of characters that work as directions for the turtle to follow. To do this, we store the letters to replace in one array s[], and then have a corresponding 2D array w[][] with the string to replace the character with. The following drawing demonstrates how these arrays match up, the final arrays in my program might be slightly different sizes.



There is also a very large u[] array that holds the final string of instructions for the turtle. Looping through the u[] array, we can check for the presence of letters to replace by comparing with s[] and when a match is found, hold the corresponding w[] in a temp, and then finally copy the resulting string back into u[]. If no match is found, the current u[i] just gets added to the temporary array that is holding this iteration through this string interpolation function.

Finally, when the string has been properly created, the turtle can follow the directions to create Sierpinski's Triangle, with the turn angle set to 60 degrees and a 10 pixel unit of movement. To play around, I ran this process of the turtle drawing the triangle over and

over with changing the turn degree, subtracting 2 from it every iteration to see what Sierpinski's triangle would look like morphed by the angle.

2.3 Artistic Description:

The colors I chose are the same as my choice L-system fractal that I will describe below. The effort I made here was to try rendering this image with the angle changing, rather than including a video I have just added a few stills above of the same grammar running a few times with different angles. In the program itself there is a loop that runs, and at each iteration the angle gets increased by 2 degrees. As the degree increases, the triangle somewhat collapses in on itself.

2.4 Code

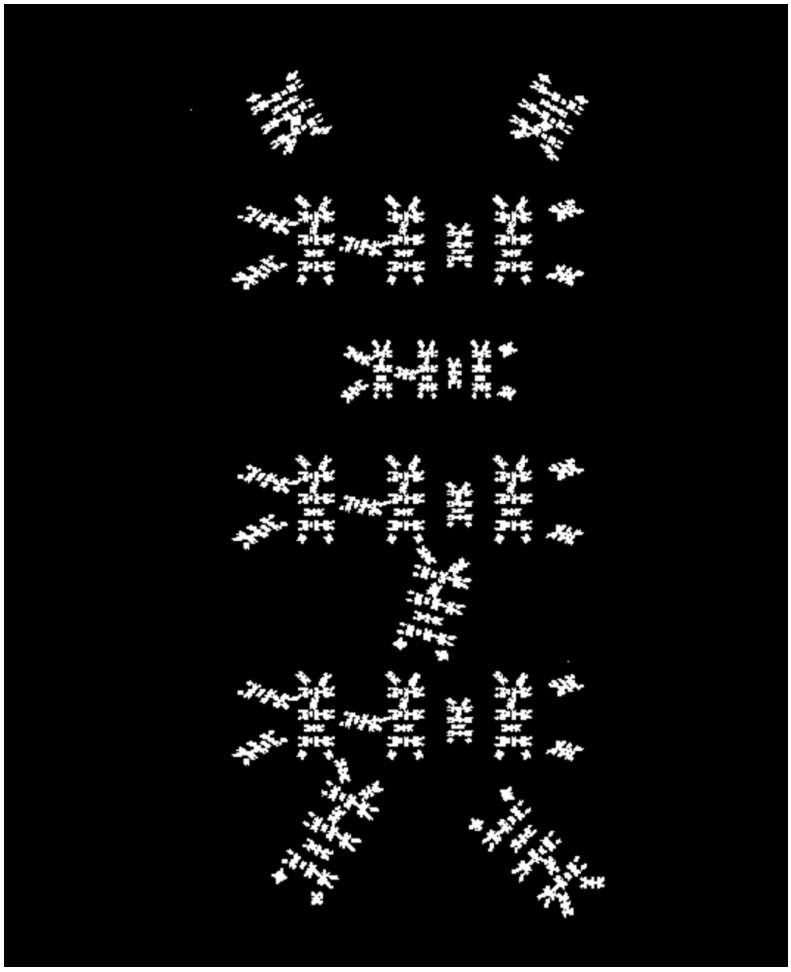
```
#include "FPToolkit.c"

char u[10000000];
char temp[10000000];
char s[10];
char w[10][1000];
int n = 0; // number of rules in Grammar

void function(int depth) {
	if (depth == 0) {
		return;
	}
	bool flag = false;
	temp[0] = '\0';
	char dumb[2];
	for (int i = 0; i < strlen(u); ++i) {	// GO THROUGH U
		for (int j = 0; j < strlen(s); ++j) { // GO THROUGH AND COMPARE WITH S
			if (u[i] == s[j]) {
				strcat(temp, w[j]);
				flag = true;
				break;
			} else {
				flag = false;
			}
		}
		if (flag == false) {
			dumb[0] = u[i];
			dumb[1] = '\0';
			strcat(temp, dumb);
		}
	}
	strcpy(u, temp);
	function(depth - 1);
}

void color(int i) {
	switch (i % 3) {
```

```
	case 0: // RED
		G_rgb(1, 0, 0);
		break;
	case 1: // GREEN
		G_rgb(0, 0.5, 0.15);
		break;
	case 2: // ORANGE
		G_rgb(1, 0.84, 0);
		break;
	case 3: // YELLOW
		G_rgb(1, 1, 0);
		break;
	defaut:
		break;
	}
}

int main() {
	int swidth, sheight;

	double deg = 58;
	double unit = 10;

	double turtle[2];
	double potential[2];
	double hold[2];
	double x, y;
	bool flag = false;

	swidth = 800;
	sheight = 700;
	// double init = swidth /2;
	double initx = 50;
	double inity =50;

	// GET GRAMMAR
	printf("Enter Axiom: ");
	scanf("%s", u);
	printf("Enter how many rules: ");
	scanf("%d", &n);
	for (int i = 0; i < n; ++i) {
		printf("Rule # %d ", i);
		printf("Letter to replace: ");
		scanf("%s", &s[i]);
		printf("\nReplace with: ");
		scanf("%s", &w[i][0]);
	}

	function(6);

	G_init_graphics(swidth, sheight); // interactive graphics
for(double j = 0; j < 10; j-= 0.1) {
    // unit -= 0.2;
  deg += 2;
	G_rgb(0, 0, 0);
```

```
G_clear();
// INITIALIZE TURTLE
turtle[0] = initx;
turtle[1] = inity;
potential[0] = turtle[0] + unit;
potential[1] = turtle[1];

// COLOR OF TURTLE!!
G_rgb(0, 1, 0);
G_fill_circle(turtle[0], turtle[1], 1);
G_wait_key();

// LETS START MOVING
double c, s;

double theta;
double temp;
theta = deg * M_PI / 180;
c = cos(theta);
s = sin(theta);

       for (int i = 0; i < strlen(u); ++i) {
              color(i);
              if (u[i] == '+') { //+

                     potential[0] = potential[0] - turtle[0];
                     potential[1] = potential[1] - turtle[1];

                     hold[0] = (potential[0] * c - potential[1] * s);
                     hold[1] = (potential[0] * s + potential[1] * c);

                     potential[0] = hold[0] + turtle[0];
                     potential[1] = hold[1] + turtle[1];
              } else if (u[i] == '-') { //+

                     potential[0] = potential[0] - turtle[0];
                     potential[1] = potential[1] - turtle[1];

                     hold[0] = (potential[0] * c) + (potential[1] * s);
                     hold[1] = -(potential[0] * s) + (potential[1] * c);

                     potential[0] = hold[0] + turtle[0];
                     potential[1] = hold[1] + turtle[1];
              }

              else if (u[i] <= 'Z' && u[i] >= 'A') { // uppercase letter
                     // G_wait_key();

                     G_line(turtle[0], turtle[1], potential[0], potential[1]);
                     hold[0] = potential[0] - turtle[0];
                     hold[1] = potential[1] - turtle[1];

                     turtle[0] = potential[0];
                     turtle[1] = potential[1];
```

```
                        potential[0] += hold[0];
                        potential[1] += hold[1];

                        G_fill_circle(turtle[0], turtle[1], 1);
                    } else {
                        continue;
                    }
                }
  int key;
       key = G_wait_key();
       }

       int key;
       key = G_wait_key();
}
```

3. IFS - Kanji



3.1 Background:

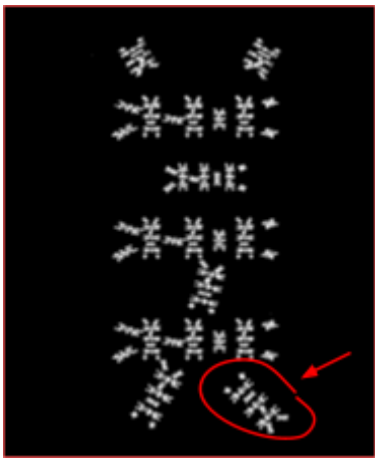
Iterated function systems is about looking at subcomponents of a picture as a translation of the bigger picture. In class we would talk about the greater picture as “mama” and the subcomponents as “babies.” So a completed image using the iterated function system paradigm would be a mama image made of babies that are smaller, translated versions of mama. In class we did our initials, which inspired me to do the exercise over with my name in Japanese. I ended up not doing my whole name 美生 (Mio), and just doing the first part the 美. This kanji character means “beautiful” and is extra special to me because my sister and my mother and I all share the same character at the beginning of our names.

3.2 Design Paradigm & Mathematical Description:

As described above, iterated function systems are about describing an entire image with smaller versions of these babies. In my image above, the large image is the 美 character made with 9 babies. Each baby has been scaled to be a smaller version of the big picture, rotated from the straight vertical of the mama picture to make the correct angle for its part, and then translated to the right part of the display window.

To form an image within the IFS paradigm, we randomly draw points throughout the entire window that eventually form the entire picture. To generate this random assortment of points, I run the function 50000000 times, which corresponds to 50000000 points drawn on the screen. For each iteration, generate a random floating point number from 0-1, and pass that number to the function that generates the babies. Using that random number from 0-1 and division, assign which rule will be followed on this iteration. For example, to get baby number 3, that is the case where $r < 0.3/9$.

Once we have decided which baby image to make, and the corresponding rule for that baby, we can do our transformations. Let's look at baby number 9:



The transformations for baby number nine are:

```
scale(2.0/12, 4.0/24)
rotate(45)
translate(15.0/24, 0.9/24)
```

The scale means to make it 2.0/12 the size of mama in the x direction and 4/24 the size of mama in the y direction. This was planned using graph paper, and deciding that these would be the appropriate ratios for this baby. Then one we have scaled, rotate the baby by 45 degrees so that it is at an angle rather than the straight up and down of mama, and then finally translate 15/24 of the larger image in the x

direction, and 0.9/24 in the y direction. Finally, once the transformations have taken place, to finally scale to the visible window, we scale the resulting x and y coordinates of each point by the height and width of the window and then plot the point. This example just showed baby number 9, so the rules will be slightly different for every baby, but the process remains the same.

Repeat this process of getting a random number to decide which baby this point will correspond to, transform, and then scale and plot the point 5000000 times.

3.3 Artistic Description

For this image, I decided to go a simple route of white on black. The artistic portion was to try to translate one character that has many lines into a visible and understandable image.

3.5 Code

```
double x[1] = {0.0};
double y[1] = {0.0};
int n = 1 ;
double width = 701, height = 701;


void translate (double dx, double dy)
{
  int i ;

  i = 0 ;
  while (i < n) {
    x[i] = x[i] + dx ;
    y[i] = y[i] + dy ;
    i = i + 1 ;
  }
}

void scale (double sx, double sy){

  for(int i = 0; i < n; ++i){
    x[i] *= sx;
    y[i] *= sy;
  }
}


void rotate(double deg){
  double r, a;
  double t = deg * (M_PI / 180);
  double temp;
  double c, s;
  c = cos(t);
  s = sin(t);
```

```
  for(int i = 0; i < n; ++i){
    temp = (x[i]* c) - (y[i] * s);
    y[i] = (y[i] * c) + (x[i] * s);
    x[i] = temp;
  }
}

void diagonal_line(double r ){
  if(r < 0.5){
    scale(0.5, 0.5);
  }
  else{
    scale(0.5, 0.5);
    translate(0.5, 0.5);
  }
}
```

//IFS FUNCTION

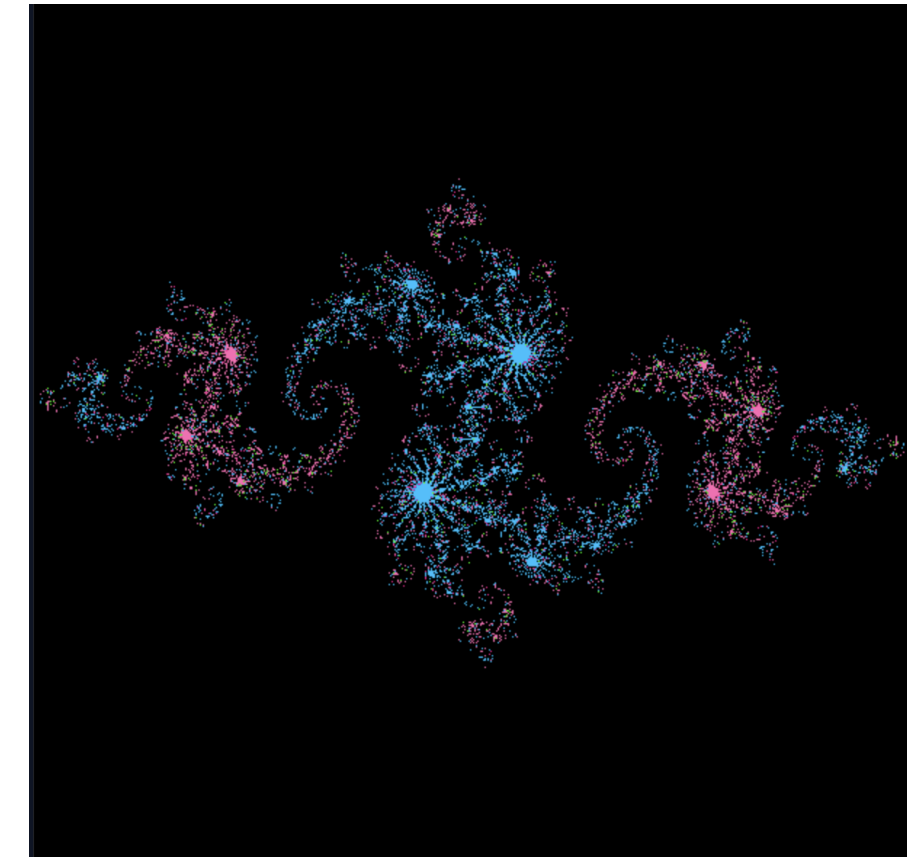
```
void mio(double r){
  if(r < 1.0 /9){
    scale(2.0/12, 2.0/24);
    rotate(30);
    translate(4.0/12, 19.0/24);
    G_rgb(1,0,0); //RED

  }
  else if(r < 2.0 / 9){
    scale(2.0/12, 2.0/24);
    rotate(330);
    translate(6.5/12, 21.0/24);
    G_rgb(1, .75, .79); //PINK
  }
  else if(r < 3.0 / 9){
    scale(3.0/12, 10.0/24);
    rotate(270);
    translate(3.5/12, 21.0/24);
    G_rgb(0.5,0,0); //MAROON
  }
  else if(r < 4.0 / 9){
    scale(2.0/12, 5.0/24);
    rotate(270);
    translate(5.0/12, 17.0/24);
    G_rgb(0,1,0); //GREEN
  }
  else if(r < 5.0 / 9){
    scale(3.0/12, 10.0/24);
    rotate(270);
    translate(3.5/12, 15.0/24);
    G_rgb(0,0,1); //BLUE
  }
  else if(r < 6.0 / 9){
    scale(2.0/12, 3.0/24);
    rotate(165);
    translate(7.50/12, 10.5/24);
```

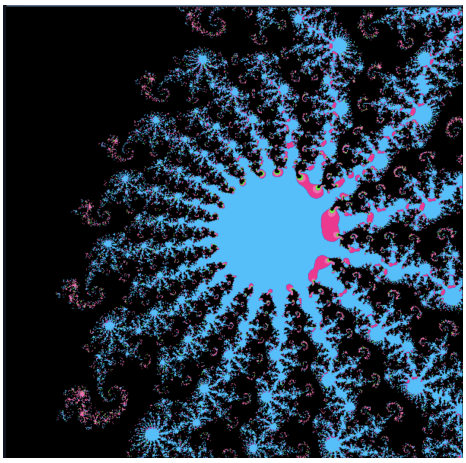
```
    G_rgb(0, .5, 0.5); //TEAL
  }
  else if(r < 7.0 / 9){
    scale(3.0/12, 10.0/24);
    rotate(270);
    translate(7.0/24, 10.0/24);
    G_rgb(0, 0.74, 0.5); // mint
  }
  else if(r < 8.0 / 9){
    scale(2.0/12, 4.0/24);
    rotate(150);
    translate(6.5/12, 5.0/24);
    G_rgb(.14, 0.5, 0.14); //DARK GREEN
  }
  else{
    scale(2.0/12, 4.0/24);
    rotate(45);
    translate(15.0/24, 0.9/24 );
    G_rgb(0.61, 0.19, 0.88); //purple
  }

}
```

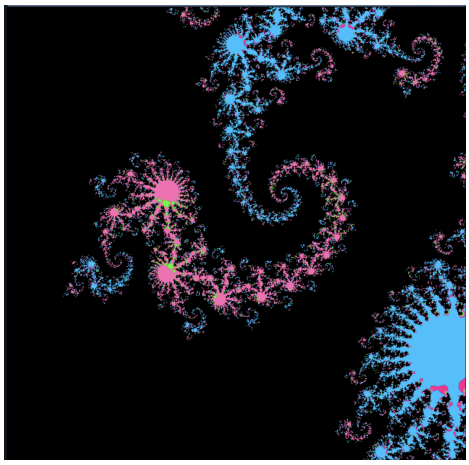
//MAIN

```
int main()
{
  int q ;

  G_init_graphics(width, height) ;
  G_rgb(0,0,0) ;
  G_clear() ;
  srand48(200);
  G_rgb(1,1,1);
  G_fill_rectangle(2,2,width-4, height-4);
  G_rgb(1,0,0);

  double r;

  for (int i = 0; i < 500000000; ++i){
    r = drand48();

    mio(r);
    G_rgb(0,0,0);
    G_point(x[0]* width, y[0]* width);
    // G_rgb(r,r*2,r);
    if(i % 1000 == 0){

    G_display_image();
    usleep(1000);
      }
    }

  q = G_wait_key() ;
}
```

3. Complex Number system. Julia set







4.1 Background:

The Julia set, defined by Gaston Julia who was significant in the study of complex number dynamics. There are many examples of common Julia sets and they can look very different with slight variations in the initializations of the numbers we send into our function.

4.2 Design Paradigm & Mathematical Description:

To get to the Julia set, first we described the Mandelbrot set. The Mandelbrot set is defined by a function

$$z \leftarrow z^2 + c$$

Where z is a constant 0, and c is changing depending on the point on the screen. In the world in which we are plotting points, we initialize a window, mine is initialized to 600x600, and every point on that window must be mapped to the complex plane, and then run through the function. The 600x600 pixel window has points xp and yp, and these must be mapped to the complex plane with

$$(x\text{-}rcen) / radius = (xp - 300) / 300$$
$$x = ((xp - 300) / 300 * radius) + rcen$$
and
$$(y\text{-}icen) / radius = (yp - 300) / 300$$
$$y = ((yp - 300) / 300 * radius) + icen$$

300 is half the window size

rcen, icen, and radius are predefined as the real and imaginary parts of the center of the image, and radius is the size of the window we are thinking in the complex plane we are mapping to.

So to get the Mandelbrot set mapped from the complex plane to our large 600x600 visible window, run through every point to plot (xp, yp) in the 600x600 window, map the point to a, (x,y) in the complex plane with the above equations, and then define

$$c = x + y*I$$

Now we need to determine if each of these points diverges to infinity when run through the function, or else converges or ping pongs between values. Run through the mandelbrot function

$$z \leftarrow z^2 + c$$

a predefined number of times, which I have set to 256. Once we have run through this, we can check if the number diverges by seeing if it is bigger than a defined cutoff, say 100, but it can be smaller than that. If after running through 256 times, z is greater than the cutoff, we can assume that it goes to infinity, and color the pixel black. If z is not greater than the predefined cutoff, we can assume that z either converges on a point, or ping pongs back and forth between two values, and so we can color this point.

The Julia set is a very similar process, run through every pixel of our window, map the point to the complex plane, run through a function to see if the point converges or

diverges, and color the points, all with the same functions as the Mandelbrot set, but this time, the z is always changing based on the (x,y) point, and C remains constant. Many different common Julia sets can be produced by changing the c constant. For my version above, I chose to set

$c = -0.8 + 0.156*I$

So to check for convergence or divergence,

$z \leftarrow z^2 + -0.8 + 0.156*I$

4.3 Artistic Description

As said above in the math section, many different Julia sets can be defined by changing the value of c. I chose $c = -0.8 + 0.156*I$ from a list of common Julia sets because I thought it was prettiest, and also it reminded me of a galaxy. With all the recent news of the James Webb telescope taking the clearest oldest pictures of the stars, I guess it has been on my mind, and I like the starry night vibes of this specific Julia set. To begin, I just had diverging points be black, and converging points be blue, but to add a little more stars/ galaxy look to it, I added a few colors to add variation in the colors. To do this, instead of just black for diverge and blue for converge, I check the value of z against the cutoff value, and depending on different ratios of the cutoff, and assign colors based on how small the value is. Pink if bigger than ⅓ of the cutoff, orange if ¼ of the cutoff, etc. blue if smaller than the cutoff. The two bonus images are zoomed in portions of the bigger image.

4.4 Code

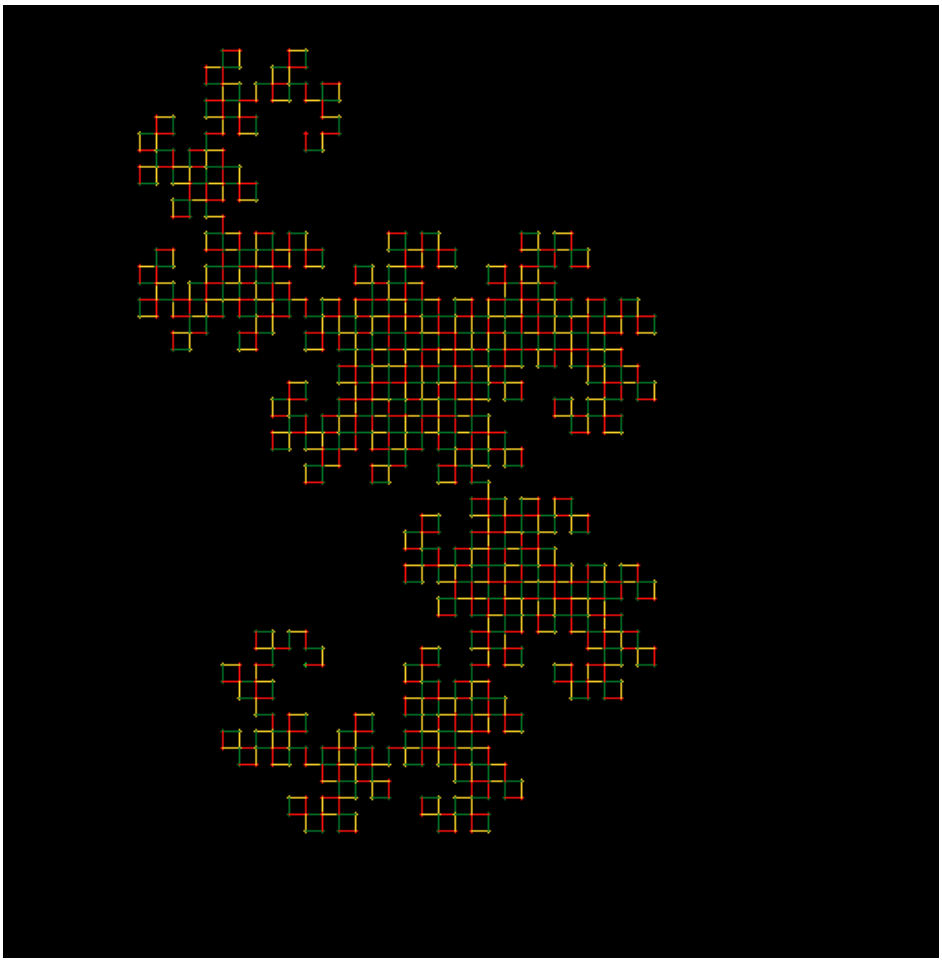
```
int Wsize = 600;

void print_complex(complex z) {
       printf("%lf + %lfi\n", creal(z), cimag(z));
}
void julia(double rcen, double icen, double radius, double cutoff, double mlimit){
  int xp, yp; //points to plot. 600x600 plane
       double x, y;  //corresponding points in complex plane
       complex z; //z to check for convergence
       complex c; //complex number

       for (yp = 0; yp < Wsize; yp++) {
              for (xp = 0; xp < Wsize; xp++) {

      x = (((xp - 300) / 300.0)* radius) + rcen;
      y = (((yp - 300) / 300.0)* radius) + icen;

      c = -0.8 + 0.156*I;
                     z = x + y * I;
                     for (int i = 0; i < mlimit; ++i) {
                            z = cpow(z, 2) + c;
```

```
                    }
                    if (cabs(z) > cutoff) { // going to infinity
                          G_rgb(0, 0, 0);           /// black
                    }
      else if (cabs(z) > cutoff/3.0){   // converging or ping ponging
                          G_rgb(1, 0.41, 0.7);            // pink

                    }
      else if (cabs(z) > cutoff/2.0){   // converging or ping ponging
                          G_rgb(0, 1, 0);           // green
                    }

      else if (cabs(z) > cutoff/4.0){   // converging or ping ponging
                          G_rgb(1, 0.07, 0.57);           // orange
                    }

      else{
        G_rgb(0, 0.75, 1 );
      }
                    G_point(xp, yp);
      // G_fill_circle(xp, yp, 1);
             }
       }
}

int main() {
       G_init_graphics(Wsize, Wsize); // interactive graphics
       G_rgb(0, 0, 0);
       G_clear();

       double cutoff = 2;
       double mlimit = 256;

       // go across bottom -2 to 2
       // go up -2 to 2

  // double rcen = 0.35;
  // double icen = 0.1;
  // double radius = 0.25;
  double rcen = 0;
  double icen = 0;
  double radius = 1.5;

  // mandelbrot(rcen, icen, radius,  cutoff, mlimit);
  julia(rcen, icen, radius,  cutoff, mlimit);
  double p[2], q[2];

  for(int i = 0; i < 3; ++i){
    G_wait_click(p); //center point
    G_wait_click(q); //determine radius

    double x0 = (((p[0] - 300) / 300.0)* radius) + rcen;
    double y0 = (((p[1] - 300) / 300.0)* radius) + icen;

    double x1 = (((q[0] - 300) / 300.0)* radius) + rcen;
```

```
    double y1 = (((q[1] - 300) / 300.0)* radius) + icen;

    rcen = x0;
    icen = y0;

    double dx = x1-x0;
    double dy = y1-y0;
    radius = sqrt(pow(dx, 2) + pow(dy, 2));

    mlimit -= 50;
   julia(rcen, icen, radius,  cutoff, mlimit);
  }

      int key;
      key = G_wait_key();
}
```

4. L-System Dragon curve CHOICE

For my choice fractal, I looked up more L system examples, and landed on the Dragon curve because I liked the name. This implementation is the Heighway dragon.



5.1 Background:

This dragon curve is generated in the same way that I generated my previous L system fractal above, and the description of all of that remains here. The difference now is to set the turn angle to 90 and use a new grammar.

5.2 Design Paradigm & Mathematical Description:

Use 90 degrees and in my example above I set my units of forward motion to 10. The implementation for the dragon involves 2 rules.

Axiom: A

$$A \rightarrow A+B$$
$$B \rightarrow A-B$$

5.3 Artistic Description

For the artistic element of this fractal, I took inspiration from the name dragon. One of my favorite cities in the world is Hong Kong, where the dragon is a very important traditional symbol. There are even holes in some skyscrapers in Hong Kong that allow for dragons to fly through unimpeded. Hong Kong is also known for their neon lights, and from the few times I have visited I am left with memories of the glowing red, orange, yellow, and green lights. So these are the colors I used for my dragon curve above. For every character in the direction array generated by the grammar, I mod the index of the character array by 4 and assigned the colors based on the result of the mod operation.

https://archinect.com/news/article/150000479/dragon-proofing-why-skyscrapers-in-hong-kong-have-holes

https://i.ytimg.com/vi/mzbCgeMRy68/maxresdefault.jpg

5.4 Code:

```
#include "FPToolkit.c"

char u[10000000];
char temp[10000000];
char s[10];
char w[10][1000];
int n = 0; // number of rules in Grammar
```

```
void function(int depth) {
	if (depth == 0) {
		return;
	}
	bool flag = false;
	temp[0] = '\0';
	char dumb[2];
	for (int i = 0; i < strlen(u); ++i) {	// GO THROUGH U
		for (int j = 0; j < strlen(s); ++j) { // GO THROUGH AND COMPARE WITH S
			if (u[i] == s[j]) {
				strcat(temp, w[j]);
				flag = true;
				break;
			} else {
				flag = false;
			}
		}
		if (flag == false) {
			dumb[0] = u[i];
			dumb[1] = '\0';
			strcat(temp, dumb);
		}
	}
	strcpy(u, temp);
	function(depth - 1);
}

void color(int i) {
	switch (i % 3) {
	case 0: // RED
		G_rgb(1, 0, 0);
		break;
	case 1: // GREEN
		G_rgb(0, 0.5, 0.15);
		break;
	case 2: // ORANGE
		G_rgb(1, 0.84, 0);
		break;
	case 3: // YELLOW
		G_rgb(1, 1, 0);
		break;
	defaut:
		break;
	}
}

int main() {
	int swidth, sheight;

	double deg = 90;
	double unit = 10;

	double turtle[2];
	double potential[2];
```

```
double hold[2];
double x, y;
bool flag = false;

swidth = 600;
sheight = 600;
// double init = swidth /2;
double initx = 200;
double inity =200;

// GET GRAMMAR
printf("Enter Axiom: ");
scanf("%s", u);
printf("Enter how many rules: ");
scanf("%d", &n);
for (int i = 0; i < n; ++i) {
       printf("Rule # %d ", i);
       printf("Letter to replace: ");
       scanf("%s", &s[i]);
       printf("\nReplace with: ");
       scanf("%s", &w[i][0]);
}

function(10);

G_init_graphics(swidth, sheight); // interactive graphics

G_rgb(0, 0, 0);
G_clear();
// INITIALIZE TURTLE
turtle[0] = initx;
turtle[1] = inity;
potential[0] = turtle[0] + unit;
potential[1] = turtle[1];

// COLOR OF TURTLE!!
G_rgb(0, 1, 0);
G_fill_circle(turtle[0], turtle[1], 1);
G_wait_key();

// LETS START MOVING
double c, s;

double theta;
double temp;
theta = deg * M_PI / 180;
c = cos(theta);
s = sin(theta);

       for (int i = 0; i < strlen(u); ++i) {
              color(i);
              if (u[i] == '+') { //+

                     potential[0] = potential[0] - turtle[0];
                     potential[1] = potential[1] - turtle[1];
```

```
                        hold[0] = (potential[0] * c - potential[1] * s);
                        hold[1] = (potential[0] * s + potential[1] * c);

                        potential[0] = hold[0] + turtle[0];
                        potential[1] = hold[1] + turtle[1];
                  } else if (u[i] == '-') { //+

                        potential[0] = potential[0] - turtle[0];
                        potential[1] = potential[1] - turtle[1];

                        hold[0] = (potential[0] * c) + (potential[1] * s);
                        hold[1] = -(potential[0] * s) + (potential[1] * c);

                        potential[0] = hold[0] + turtle[0];
                        potential[1] = hold[1] + turtle[1];
                  }

                  else if (u[i] <= 'Z' && u[i] >= 'A') { // uppercase letter
                        // G_wait_key();

                        G_line(turtle[0], turtle[1], potential[0], potential[1]);
                        hold[0] = potential[0] - turtle[0];
                        hold[1] = potential[1] - turtle[1];

                        turtle[0] = potential[0];
                        turtle[1] = potential[1];

                        potential[0] += hold[0];
                        potential[1] += hold[1];

                        G_fill_circle(turtle[0], turtle[1], 1);
                  } else {
                        continue;
                  }
            }


      int key;
      key = G_wait_key();
}
```