Fractals Portfolio

Forest Pearson

June 2023

1 Entires

1.1 Complex Choice: Burning Ship

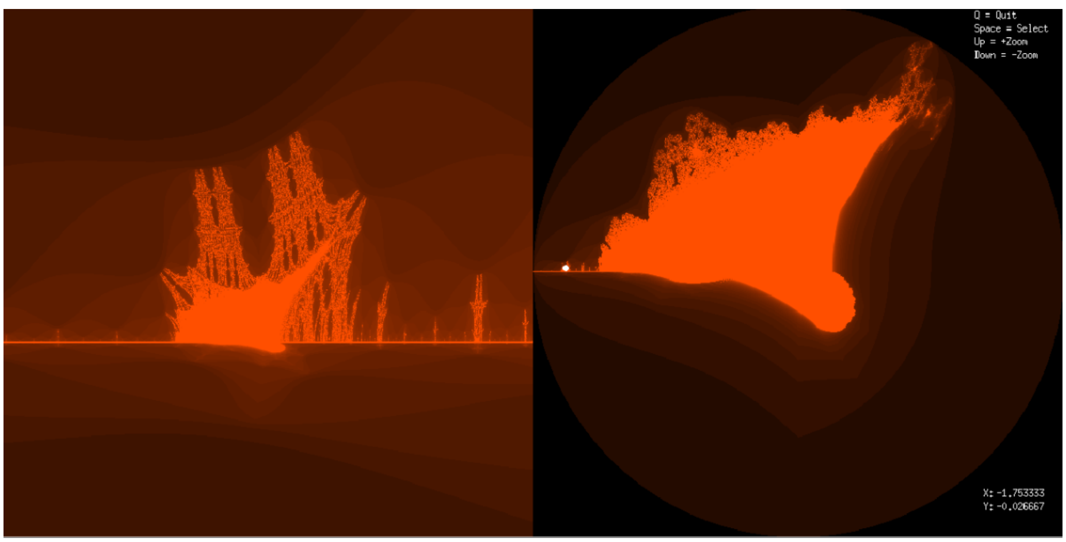

Figure 1: Burning Ship Fractal

Paradigm & Mathematical Description: The burning ship itself is a fractal described by Michael Michelitsh and Otto E Rossler in 1992 where like the mandelbrot it makes use of both real and imaginary numbers for its complex values. The difference here being that the iterating function is $z = abs(z)^2 + c$ where the imaginary values are set to their absolute values instead of $z = z^2 + c$ and diveregence is checked by $cabs(z) > 2.0$. The initial complex number set of cz, zy is obtained by mapping each (x, y) point on the screen from it's respective position with the functons:

$cx = 2 * ((x - (swidth/4.0)) * (swidth/4.0))$ and $cy = 2 * ((y - (sheight/2.0)) * (sheight/2.0))$

The screen itself is broken up into two halves, with the right containing the full fractal and the left containing the zoomed location around a chosen point. This zoom is created by four x and y max/min variables combined with a fed in zoom value from 0-1. This is used to create the real and imaginary values in conjunction with the previous position function. Ending up with $dx = (xmax - xmin)/(swidth/2.0)$ and $real = xmin + x * dx$ within the loop for x while a corresponding counterpart is created for y. This then loops through the designated space to create the zoomed in affect.

Artistic Description: For this fractal I kept the design simple to show of the complexity created in the burning ship fractal set. I focused on setting the red-orange color for divergence for the burning theme it's named after and spent my time focusing on improving the interactiveness of the fractal. Part of the fun I found is exploring this fractal as if it's a new world, zooming in and traveling across it. These capabilities are what I enabled with the controls seen in figure 1 where it is currently zoomed in on the most iconic part of the burning ship seen on the left antenna.

The coloring for depth here is done by multiplying set red and green values between 0-1 against the value $sf = 1.0 * k/reps$ where reps is the max depth and k is the current diverging or set depth. This allows it to naturally progress from one shade to another then eventually to black as it fully diverges for both the main fractal and the zoom.

1.2 Complex: Mandelbrot and Juliet Exploration

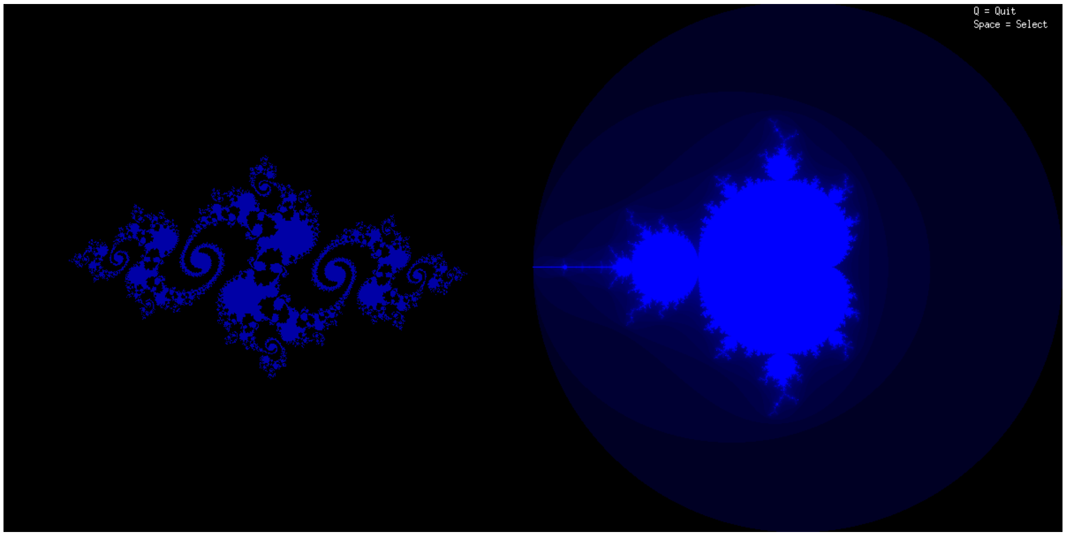

Figure 2: Mandelbrot And Juliet Fractal

Paradigm & Mathematical Description: The mandelbrot fractal is a set of complex numbers discovered by Benoit Mandelbrot in 1980 while working at IBM, wheras the Juliet set was discovered by Gaston Julia and Pierre Fatou where they were then popularized by Benoit Mandelbrot due to their close nature. For mathmatics the mandelbrot takes a simplified version of what I did in the burning ship with the iterating function being $z = z^2 + c$ while still using $cabs(z) > 2.0$ to check for divergence. The variables cz and cy are still obtained in the same position functions:

$cx = 2*((x-(swidth/4.0))*(swidth/4.0))$ and $cy = 2*((y-(sheight/2.0))*(sheight/2.0))$

The Juliet here though takes in the position of a single (x, y) pixel decided by the cursor into the cy and cx functions to create a complex number $c = mx + my * I$. This with a passed in depth and zoom is then fed into the julia which recursively travels through the left hand half of the screens width and height while iterating with $z = z^2 + c$ where c is a constant not recalculated.

Artistic Description: For this fractal I once again kept the design simple to show of the complexity created here in the Mandelbrot and Juliet together. I setup a gentle blue color for divergence spent my time focusing on improving the interactiveness across both the fractals. I found it mesmerizing to explore the edges of the Mandelbrot and see the vastly different Juliet sets created, which you can see in Figure 2 where I discovered a spiraling pattern different from a dozen others I had previously seen.

The coloring for depth here is once again done by multiplying a set blue color value between 0-1 against the value $sf = 1.0 * k/reps$ where reps is the max depth and k is the current diverging or set depth. This allows it to naturally progress from one shade to another then eventually to black as it fully diverges.

1.3 Lsys: A peaceful night

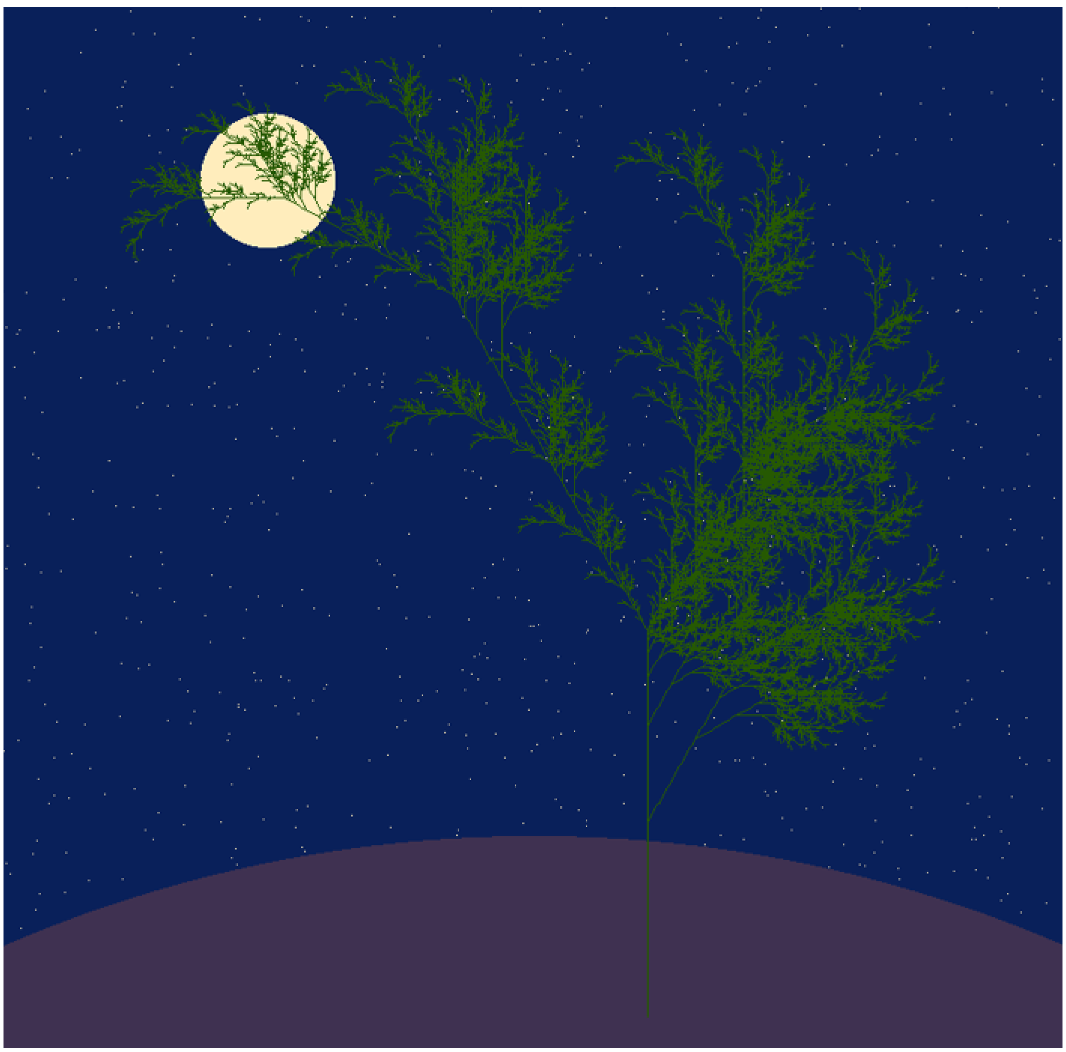

Figure 3: Lsys Fern Fractal

Paradigm & Mathematical Description: For this fractal the Lindenmayer Systems was discovered by Aristid Lindenmayer in 1968 during his time as a botanist at the University of Utrecht. The Lindenmayer system creates a formal grammer production system making use of rules to recursively build a progressively larger string dependent on depth. Here this sytem was used with an axiom A and two rules $A- > F-[A]--A++[++A]$ and $F- > FF$ where $+$ and $-$ move the angle of the system clockwise or counterclockwise respectively, [and] push and pop the current vector data, then finally if a character such as F or A is encounted the direction is moved along a preset length directed by the mentioned angle values.

Artistic Description: For this fractal I created a solitary fern bush among a starry night and a moon shining through the background. I did this in order to highlight the natural and organic style that can be generated by such a simplistic formal grammar. Showcasing how such natural designs seen in nature all around us can be recreated using pure mathematics, lending thought to how the world around us may take inspiration from similar mathematical principles in its own way.
Beyond that the stars are simply randomly generated G_points for a set amount between the screens width and height while the moon and planet are offset filled circles of various sizes.

1.4 Recursive: A snowy wonderland

Figure 4: Recursive Koch Snowflake Fractal

Paradigm & Mathematical Description: For this fractal the snowflake is an exploration of the Koch Curve discovered by Helge von Koch in a constructible geometry paper written in 1904. The koch curve here is generated using a recursive function from two points in essence on a single line (p_1, p_2), these points then create a a third segment with points p_3, p_4 evenly between them and a additional point p_3 to create an equilatoral triangle between points (p_3, p_4, p_5). The point p_5 here can be calculated by making use of sin and cos in the function $p_5x = p_3x + (p4_x - p3_x) * cos(PI/3.0) - (p_4y - p_3y) * sin(PI/3.0)$ and $p_5y = p_3y + (p4_x - p3_x) * sin(PI/3.0) - (p_4y - p_3y) * cos(PI/3.0)$. This can then be repeated recursively for each segment between two points to create the geometric Koch pattern we know. For the snowflake itself the Koch curve is started three times upon an equilateral triangle consisting of three passed points, leading to the design seen above.

Artistic Description: For this fractal I envisioned a late snowy night here in Portland, staring out into the dark night with large clumpy flakes visible only due to the light peaking through the window with you. A simple time where you can simply bask in the nature around you.

For how this is setup the initial equilateral triangles and then snowflakes are given random locations and sizes across the screen where they are then generated for a depth of 5. This is combined with a subtle blue gradient taking the height of the screen as a modifier to the rgb 0-1 values.

1.5 IFS: An initial card

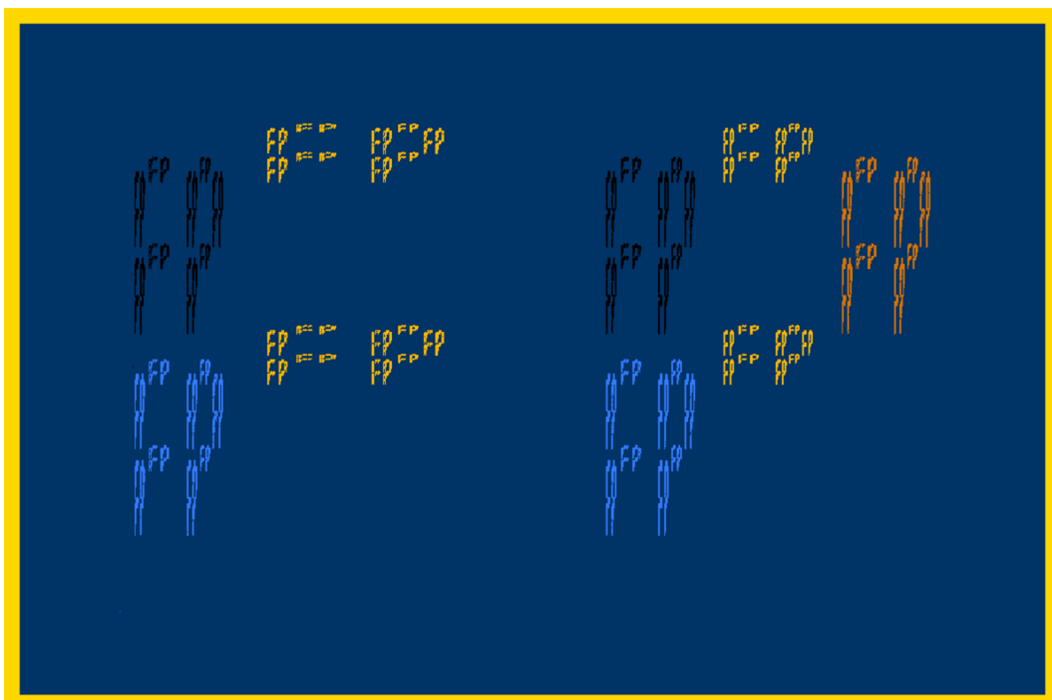

Figure 5: IFS Fractal

Paradigm & Mathematical Description: For this fractal the Iterated Function System were discovered formaly by John E. Hutchinson in 1981. This is a method of creating fractals that can be considered self-similar, where they themselves can be seen as the repeating pattern. For this fractal my initials were drawn down using a graph sheet to create percise fractions upon which I could generate the fractal. This is done by randomly selecting from a set of rules that will scale and translate a shared position where scale determine the size based upon a fraction given and translate will move the position by given fractions as well. These repeatedly processed rules will then generate a pattern based upon the scaling, translating, and rotating, setup among them.

Artistic Description: For this fractal it is a simple card with my initials upon it, with different rules containing unique colors to showcase the functionality of how the iterated function system works in its random selection of the rules. You can see how depending upon the (x, y) scaling letters appear streched or shrunked, allowing for a more percise fit into the letters themselves.All of this is then highlighted with a simple gold border.

2 Code

2.1 Complex Choice: Burning Ship

```
/*
#Forest Pearson
#Fractals course
#06/14/2023
*/
#include "FPToolkit.c"
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <tgmath.h>

const int swidth = 1200;
const int sheight = 600;

//burningShip variables
int reps = 50;
double cx,cy ;
complex c, z;
double sr, sg, sb ;
double er, eg, eb ;
double red, g, blue ;

//point coordinates
double p[2];

void burningShip() {
  sr = 0.0 ;  sg = 0.0 ;  sb = 0.0 ;
  er = 1.0 ;  eg = 0.31 ;  eb = 0.0 ;
  // iterate through each pixel of window
  for (int x = 0; x < swidth/2; x++){
    for (int y = 0; y < sheight; y++) {
```

```
      // map to coordinating complex number
      cx = 2*((x-(swidth/4.0))/(swidth/4.0)) ;
      cy = 2*((y-(sheight/2.0))/(sheight/2.0)) ;
      c = cx + cy*I ;
      z = 0;
      int k = 0;

      for(k = 0; k < reps; k++){
        z = (cabs(creal(z))+cabs(cimag(z))*I)*(cabs(creal(z))+cabs(cimag(z))*I)+c;
        if(cabs(z) > 2.0){
          break;
        }
      }
      double sf = 1.0*k/reps;
      sf = pow(sf,0.5);
      red = sr + sf*(er-sr);
      g = sg + sf*(eg-sg);
      blue = sb + sf*(eb-sb);
      G_rgb(red,g,blue);
      G_point(x+(swidth/2),sheight-y);
      //G_point(x+(swidth/2),y);
    }
  }
}
void zoom(double zoom, double a, double b){
  double xmin = a - zoom;
  double xmax = a + zoom;
  double ymin = b - zoom;
  double ymax = b + zoom;
  double dx = (xmax - xmin) / (swidth/2.0);
  double dy = (ymax-ymin) / (sheight);
  int x, y;
  for(x = 0; x < (swidth/2.0); x++){
    for(y = 0; y < sheight; y++){
      double real = xmin + x * dx;
      double imag = ymin + y * dy;
      c = real + imag * I;
      z = 0;
      int k = 0;
      for(k = 0; k < reps; k++){
        z = (cabs(creal(z))+cabs(cimag(z))*I)*(cabs(creal(z))+cabs(cimag(z))*I)+c;
        if(cabs(z) > 2.0){
          break;
        }
      }
      double sf = 1.0*k/reps;
      sf = pow(sf,0.5);
      red = sr + sf*(er-sr);
      g = sg + sf*(eg-sg);
      blue = sb + sf*(eb-sb);
      G_rgb(red,g,blue);
      G_point(x,sheight-y);
    }
  }
}
```

```c
int main(){
    G_init_graphics (swidth,sheight) ;
    G_rgb(1,1,1);
    G_clear();
    burningShip();
    zoom(1.0,0,0);
    int key;
    double zoomLevel = 0.1;
    p[0] = 900.00;
    p[1] = 284.00;
    G_rgb(1,1,1);
    G_draw_string("Q = Quit", swidth-100, sheight-15);
    G_draw_string("Space = Select", swidth-100, sheight-30);
    G_draw_string("Up = +Zoom", swidth-100, sheight-45);
    G_draw_string("Down = -Zoom", swidth-100, sheight-60);
    char str[100];
    while(1){
        G_draw_string("Q = Quit", swidth-100, sheight-15);
        G_draw_string("Space = Select", swidth-100, sheight-30);
        G_draw_string("Up = +Zoom", swidth-100, sheight-45);
        G_draw_string("Down = -Zoom", swidth-100, sheight-60);
        key =  G_wait_key();
        if(key == 65362){
          zoomLevel = zoomLevel + zoomLevel*0.1;
        }
        if(key == 65364){
          zoomLevel = zoomLevel - zoomLevel*0.1;
        }
        if(key == 32){
          G_wait_click(p);
        }
        if(key == 113){
          break;
        }
        G_rgb(1,1,1);
        G_clear();
        burningShip();
        G_rgb(1,1,1);
        G_fill_circle(p[0],p[1],3);
        double mx = 2*(((p[0]-(swidth/2))-(swidth/4.0))/(swidth
    /4.0));
        double my = 2*(((sheight-p[1])-(sheight/2.0))/(sheight/2.0)
    );
        zoom(zoomLevel, mx, my);
        G_rgb(1,1,1);
        sprintf(str, "%f", mx);
        G_draw_string("X:", swidth-90,45);
        G_draw_string(str, swidth-75,45);
        sprintf(str, "%f", my);
        G_draw_string("Y:", swidth-90,30);
        G_draw_string(str, swidth-75,30);

    }
  // save file
  G_save_to_bmp_file("Shipportfolio.bmp") ;

}
```

2.2 Complex: Mandelbrot Exploration

```c
/*
#Forest Pearson
#Fractals course
#06/14/2023
*/
#include "FPToolkit.c"
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <tgmath.h>

const int swidth = 1200;
const int sheight = 600;

//mandelbrot variables
int reps = 50;
double cx,cy ;
complex c, z;
double sr, sg, sb ;
double er, eg, eb ;
double red, g, blue ;

//point coordinates
double p[2];

void mandelbrot() {
  sr = 0.0 ;  sg = 0.0 ;  sb = 0.0 ;
  er = 0.0 ;  eg = 0.0 ;  eb = 1.0 ;
  // iterate through each pixel of window
  for (int x = 0; x < swidth/2; x++){
    for (int y = 0; y < sheight; y++) {

      // map to coordinating complex number
      cx = 2*((x-(swidth/4.0))/(swidth/4.0)) ;
      cy = 2*((y-(sheight/2.0))/(sheight/2.0)) ;
      c = cx + cy*I ;
      z = 0;
      int k ;
      for (k = 0 ; k < reps ; k++) {
        z = z*z + c ;
        if (cabs(z) > 2) { // diverged
          break;
        }
      }
      double sf = 1.0*k/reps;
      sf = pow(sf,0.5);
      red = sr + sf*(er-sr);
      g = sg + sf*(eg-sg);
      blue = sb + sf*(eb-sb);
      G_rgb(red,g,blue);
      G_point(x+(swidth/2),y);
    }
  }
}
```

```c
int juliaRecursive(float x, float y, int r, int depth, int max, double complex c, double complex z){
    double sf = 1.0 - ((((max-depth)*(max-depth))%(max*max))/255);
    sf = pow(sf,0.3);
    if (cabs(z) > 2) {
        depth = 0;
    }
    if (sqrt(pow((x - (swidth / 2) / 2), 2) + pow((y - sheight / 2), 2)) > sheight / 2) {//Creates encompasing circle
        G_rgb(0,0,0);
        G_point(x,y);
    }
    if (depth < max / 4) {
        double test = 1.0*depth/max;
        test = pow(test, 0.3);
        blue = sb + test*(eb-sb);
        G_rgb(0,0,blue);
        G_point(x,y);
        return 0;
    }
    juliaRecursive(x, y, r, depth - 1, max, c, cpow(z, 2) + c);
}
void juliaset(int depth, int r){
    for (float x = (((swidth / 2) / 2)) - sheight / 2; x < (((swidth / 2) / 2)) + sheight / 2; x += 1) {
        for (float y = 0; y < sheight; y += 1) {
            juliaRecursive(x, y, r, depth, depth, c, (2 * r * ((x - (swidth/ 2) / 2)) / sheight) + (2 * r * (y - sheight / 2) / sheight) * I);
        }
    }
}
int main(){
    G_init_graphics (swidth,sheight) ;
    G_rgb(1,1,1);
    G_clear();
    mandelbrot();
    c = -0.766667 + 0.100000*I;
    juliaset(100,2);
    int key;
    G_rgb(1,1,1);
    G_draw_string("Q = Quit", swidth-100, sheight-15);
    G_draw_string("Space = Select", swidth-100, sheight-30);
    while(1){
        G_rgb(1,1,1);
        G_draw_string("Q = Quit", swidth-100, sheight-15);
        G_draw_string("Space = Select", swidth-100, sheight-30);
        key = G_wait_key();
        if(key == 113){
          break;
        }
        G_wait_click(p);
        G_rgb(1,1,1);
        G_clear();
        mandelbrot();
        G_rgb(1,1,1);
        G_fill_circle(p[0],p[1],3);
```

```
        double mx = 2*(((p[0]-(swidth/2))-(swidth/4.0))/(swidth/4.0));
        double my = 2*((p[1]-(sheight/2.0))/(sheight/2.0));
        c = mx + my*I;
        printf("'X, '%f\n", mx);
        printf("'Y, '%f\n", my);
        G_rgb(1,1,1);
        juliaset(100, 2);
    }
  //Save file
  G_save_to_bmp_file("MandelPortfolio.bmp") ;

}
```

2.3 Lsys: A peaceful night

```
/*
#Forest Pearson
#Fractals course
#06/14/2023
*/
#include  "FPToolkit.c"
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <tgmath.h>
#define MAX_SIZE 1000000

typedef struct {
  char nonterminal;
  char rule[100];
} Production;

typedef struct {//Struct to track states
  double x[MAX_SIZE];
  double y[MAX_SIZE];
  double d[MAX_SIZE]; //Direction of turtle
  int xI;
  int yI;
  int aI;
} Stack;

Stack stack;
Production prods[10];
char axiom[2] = {'A', '\0'};
char derivation[MAX_SIZE] = {'\0'};
double direction = 0;
double cur[2];

void push() {
  if (stack.xI < MAX_SIZE-1) {
    stack.xI += 1;
    stack.x[stack.xI] = cur[0];
  }
  if (stack.yI < MAX_SIZE-1) {
    stack.yI += 1;
```

```
    stack.y[stack.yI] = cur[1];
  }
  if (stack.aI < MAX_SIZE-1) {
    stack.aI += 1;
    stack.d[stack.aI] = direction;
  }
}

void pop() {
  if (stack.xI >=0) {
    cur[0] = stack.x[stack.xI];
    stack.xI -=1;
  }
  if (stack.yI >= 0) {
    cur[1] = stack.y[stack.yI];
    stack.yI -= 1;
  }
  if (stack.aI >= 0) {
    direction = stack.d[stack.aI];
    stack.aI -= 1;
  }
}

void autoFit(int swidth, int sheight, double angle, double
    mainAngle, double * idealPosition) {
  double xMin=0; double yMin=0;
  double xMax=0; double yMax=0;
  double dX=0; double dY=0;
  double tempX = 0.9*swidth;
  double tempY = 0.9*sheight;
  double next[2];

  int i = 0;
  direction = mainAngle;
  cur[0] = 0;
  cur[1] = 0;

  while (derivation[i] != '\0') {
    if (derivation[i] == '[') {
      push();
    }
    else if (derivation[i] == ']') {
      pop();
    }
    else if (derivation[i] == '-') {
      direction -= angle;
    }
    else if (derivation[i] == '+') {
      direction += angle;
    }
    else if ((derivation[i] >= 'A' && derivation[i] <='Z') ||
    derivation[i] == 'f') {
      next[0] = cur[0] + cos(direction);
      next[1] = cur[1] + sin(direction);
      cur[0] = next[0];
      cur[1] = next[1];

```

```
      if (cur[0] < xMin) xMin = cur[0];//Builds outer parameters while comparing
      if (cur[0] > xMax) xMax = cur[0];
      if (cur[1] < yMin) yMin = cur[1];
      if (cur[1] > yMax) yMax = cur[1];
    }
    i++;
  }
  dX = xMax - xMin;//Create the bounding square
  dY = yMax - yMin;
  if (dY > dX) {
    tempX = dX * (tempY / dY);
  }
  else {
    tempY = dY * (tempX / dX);
  }
  idealPosition[0] = 0.5 * (swidth - tempX);
  if (xMin < 0) idealPosition[0] -= (xMin * (tempX/dX));
  idealPosition[1] = 0.5 * (sheight - tempY);
  if (yMin < 0) idealPosition[1] -= (yMin * (tempY/dY));
  idealPosition[2] = tempX / dX;
}

void stringInterpreter(int pos[2], double length, double angle, double mainAngle) {
  direction = mainAngle;
  cur[0] = pos[0];
  cur[1] = pos[1];
  double next[2];
  int i = 0;

  while (derivation[i] != '\0') {//Loop through the instructions without the need to determine bounds.
    if (derivation[i] == '[') {
      push();
    }
    else if (derivation[i] == ']') {
      pop();
    }
    else if (derivation[i] == '-') {
      direction -= angle;
    }
    else if (derivation[i] == '+') {
      direction += angle;
    }
    else if ((derivation[i] >= 'A' && derivation[i] <='Z')||derivation[i] == 'f') {
      next[0] = cur[0] + length * cos(direction);
      next[1] = cur[1] + length * sin(direction);
      G_line(cur[0], cur[1], next[0], next[1]);
      cur[0] = next[0];
      cur[1] = next[1];
    }
    i++;
  }
}
```

```c
void stringBuilder(int curr, int max) {
  if (derivation[0] == '\0') {
    strcpy(derivation, axiom);
  }
  if (curr == max){//Retrun condition for recursion
    return;
  }

  int rule=0;
  char cur[2];  cur[1] = '\0';
  char temp[MAX_SIZE];
  int i = 0;
  int j = 0;
  while (derivation[i] != '\0') {
    cur[0] = derivation[i];
    while (j < 2 && rule == 0) {//Checks rules
      if (derivation[i] == prods[j].nonterminal) {
        strcat(temp, prods[j].rule);
        rule = 1;
      }
      j++;
    }
    if (rule == 0) strcat(temp, cur);
    i++;
    j = 0;
    rule = 0;
  }
  strcpy(derivation, temp);
  stringBuilder(curr+1, max);//Next process of the rule upon itself
}

int main() {
  double length;
  int swidth = 800; int sheight = 800;
  G_init_graphics (swidth,sheight);

  G_rgb(0.039, 0.125, 0.352);//Generate the moon, ground, and stars.
  G_clear();
  G_rgb(255/255.0,237/255.0,188/255.0);
  G_fill_circle(swidth/4, sheight-sheight/6, 50);
  for(int i= 0; i < 1000; i++){
    G_point(rand()%swidth, rand()%sheight);
  }
  G_rgb(63/255.0,49/255.0,81/255.0);
  G_fill_circle (swidth/2, -825, 1000);

  //build the string and populate it
  prods[0].nonterminal = 'A';
  strcpy(prods[0].rule, "B-[[A]+A]+B[+BA]-A");
  prods[1].nonterminal = 'B';
  strcpy(prods[1].rule, "BB");
  stringBuilder(0, 7);
  stack.x[0] = '\0';
  stack.y[0] = '\0';
  stack.d[0] = '\0';
  stack.xI = -1;
```

```c
  stack.yI = -1;
  stack.aI = -1;

  int pos[2];
  double idealPosition[3];
  double mainAngle = M_PI/ 2.0; //Vertical
  autoFit(swidth, sheight, M_PI/6.0, mainAngle, idealPosition);//Determines Ideal position for demensions/placement
  pos[0] = idealPosition[0];
  pos[1] = idealPosition[1];
  length = idealPosition[2];

  G_rgb(0.15, 0.35, 0.01);//Draws the fern based upon ideal parameters.
  stringInterpreter(pos, length, M_PI/6.0, mainAngle);

  int key;
  key = G_wait_key();
  //Save to file
  G_save_to_bmp_file("lSysPortfolio.bmp");

  return 0;
}
```

2.4 Recursive: A snowy wonderland

```c
/*
#Forest Pearson
#Fractals course
#06/14/2023
*/
#include "FPToolkit.c"
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <tgmath.h>

#include  "FPToolkit.c"



void koch(double pOne[], double pTwo[], int curr, int dep) {
  double a[2], b[2], c[2], t[2];
  if (curr == dep){
    return;
  }

  a[0] = pOne[0] + (1.0/3.0) * (pTwo[0] - pOne[0]);//Determine a,t,b and c
  a[1] = pOne[1] + (1.0/3.0) * (pTwo[1] - pOne[1]);
  t[0] = a[0] - pOne[0];
  t[1] = a[1] - pOne[1];
  b[0] = a[0] + t[0] * cos(M_PI / 3.0) - t[1] * sin(M_PI / 3.0);
  b[1] = a[1] + t[1] * cos(M_PI / 3.0) + t[0] * sin(M_PI / 3.0);
  c[0] = pOne[0] + (2.0/3.0) * (pTwo[0] - pOne[0]);
  c[1] = pOne[1] + (2.0/3.0) * (pTwo[1] - pOne[1]);
```

```
  G_line(pOne[0], pOne[1], pTwo[0], pTwo[1]);
  G_fill_triangle(a[0], a[1], b[0], b[1], c[0], c[1]);

  koch(pOne, a, curr+1, dep);//Loop for angles
  koch(a, b, curr+1, dep);
  koch(b, c, curr+1, dep);
  koch(c, pTwo, curr+1, dep);
}
void snowFlake(double pOne[], double pTwo[], int depth) {
  double p3[2];//Determine p3 for the 2nd and third curves
  p3[0] = pOne[0] + (pTwo[0]-pOne[0]) * cos(-M_PI / 3.0) - (pTwo[1]-pOne[1]) * sin(-M_PI / 3.0);
  p3[1] = pOne[1] + (pTwo[1]-pOne[1]) * cos(-M_PI / 3.0) + (pTwo[0]-pOne[0]) * sin(-M_PI / 3.0);

  koch(pOne, pTwo, 0, depth);//Call the three parts to create the snowflake out of koch curves
  koch(pTwo, p3, 0, depth);
  koch(p3, pOne, 0, depth);
  G_fill_triangle(pOne[0], pOne[1], pTwo[0], pTwo[1], p3[0], p3[1]);
}

int main() {
  int swidth = 800; int sheight = 800;
  double pOne[2], pTwo[2], p3[2];
  G_init_graphics (swidth,sheight);

  G_rgb(0.039, 0.125, 0.352);
  G_clear();
  for(int i = 0; i < sheight; i++){
    for(int j = 0; j < swidth; j++){
      double gradient = (double)i/sheight;
      G_rgb(0.039*gradient, 0.125*gradient, 0.352*gradient);
      G_pixel(j,i);
    }
  }
  G_rgb(1,1,1);
  for (int i=0; i<100; ++i) {//White snowflake generation
    pOne[0] = rand() % (swidth);//Declared here to use twice.
    pOne[1] = rand() % (sheight);
    pTwo[0] = (pOne[0]) + (i%20)*cos(rand());
    pTwo[1] = (pOne[1]) + (i%20)*sin(rand());
    snowFlake(pOne, pTwo, 5);
  }
  int key;
  key = G_wait_key();
  //Save to file
  G_save_to_bmp_file("RecursivePortfolio.bmp");
  return 0;
}
```

2.5 IFS: An initial card

```
/*
#Forest Pearson
```

```
#Fractals course
#06/14/2023
*/
#include "FPToolkit.c"
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <tgmath.h>

int n ;
double current[2];
double square[2];
int swidth, sheight;
double lowlefthirdCornerX, lowlefthirdCornerY, width, height;

void scale(double x, double y){
  current[0] *= x;
  current[1] *= y;
}
void translate(double x, double y){
  current[0] += x;
  current[1] += y;
}
void rotate (double angle) {
  double temp ;
  double radians = angle*M_PI/180.0 ;
  temp = current[0]*cos(radians) - current[1]*sin(radians) ;
  current[1] = current[0]*sin(radians) + current[1]*cos(radians) ;
  current[0] = temp ;
}

void initials(){
  double k;
  double height = 4;
  double width = 10;
  double widthScale = swidth/width;
  double heightScale = sheight/height;
  const double s = 50.0;
  for(int i = 0; i < 1000000; i++){
    k = rand() % 9;
    //G_rgb((190.0/255.0), (59.0/255.0), (255.0/255.0));
    G_rgb((255.0/255.0), (184.0/255.0), (0.0/255.0));
    if(k == 0){
      G_rgb((50.0/255.0), (122.0/255.0), (255.0/255.0));
      scale((double)1/9,(double)3/7);
      translate((double)1/9,(double)1/7);

    }
    else if(k == 1){
      //G_rgb((51.0/255.0), (255.0/255.0), (138.0/255.0));
      G_rgb((0.0/255.0), (0.0/255.0), (0.0/255.0));
      scale((double)1/9,(double)3/7);
      translate((double)1/9,(double)3/7);
    }
    else if(k == 2){
      scale((double)2/9,(double)1/7);
      translate((double)2/9,(double)5/7);
```

```c
    }
    else if(k == 3){
      scale((double)2/9,(double)1/7);
      translate((double)2/9,(double)3/7);
    }
    else if(k == 4){
      G_rgb((50.0/255.0), (122.0/255.0), (255.0/255.0));
      scale((double)1/9,(double)3/7);
      translate((double)5/9,(double)1/7);
    }
    else if(k == 5){
      G_rgb((0.0/255.0), (0.0/255.0), (0.0/255.0));
      scale((double)1/9,(double)3/7);
      translate((double)5/9,(double)3/7);
    }
    else if(k == 6){
      scale((double)1/9,(double)1/7);
      translate((double)6/9,(double)5/7);
    }
    else if(k == 7){
      //G_rgb((255.0/255.0), (114.0/255.0), (118.0/255.0));
      G_rgb((216.0/255.0), (115.0/255.0), (0.0/255.0));
      scale((double)1/9,(double)3/7);
      translate((double)7/9,(double)3/7);
    }
    else if(k == 8){
      scale((double)1/9,(double)1/7);
      translate((double)6/9,(double)3/7);
    }
    G_fill_circle (swidth*current[0], sheight*current[1], .10);
  }
  return;
}
int main() {
  swidth = 1200;
  sheight = 800;
  G_init_graphics(swidth, sheight);


  //G_rgb((196.0/255.0), (221.0/255.0), (226.0/255.0));
  G_rgb((0.0/255.0), (51.0/255.0), (102.0/255.0));
  G_clear();
  G_rgb((255.0/255.0), (215.0/255.0), (0.0/255.0));
  G_fill_rectangle(0,0, 20, 800);
  G_fill_rectangle(1180,0, 20, 800);
  G_fill_rectangle(20,0,1300,20);
  G_fill_rectangle(0,780,1300,20);
  G_rgb(1.0, 1.0, 1.0);
  G_rgb(0.0, 0.0, 0.0);

  double pi = 3.14159265;
  double radian = pi /180.0;
  srand(time(NULL));
  double n = rand() / ((double) RAND_MAX);

  current[0] = swidth;
  current[1] = sheight;
```

```
  current[0] = n;
  current[1] = n;
  current[0] = 0.0;
  current[1] = 0.0;
  G_fill_circle (current[0] * swidth, current[1] * sheight, 1);

  initials();

  int wait;
  wait = G_wait_key();
  G_save_to_bmp_file("IFSPortfolio.bmp");
}
```