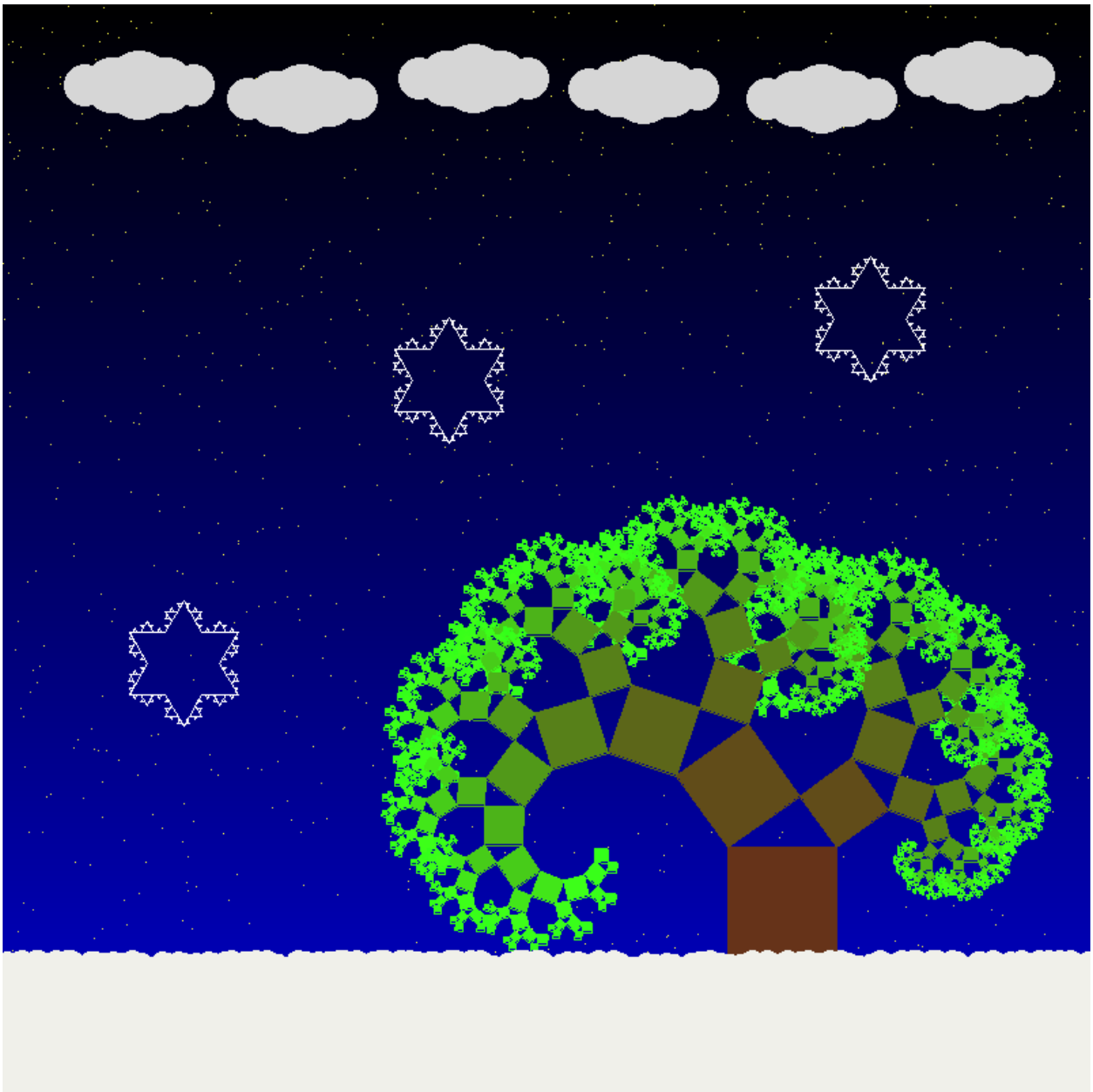
Jacob Landstrom
Christian Torralba
CS-410P-094

Final Portfolio

Recursive – Pythagoras Tree & Koch Snowflake

Piece

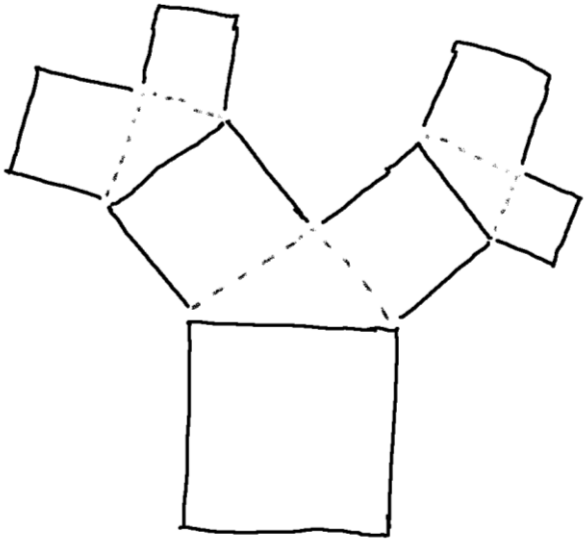


Information

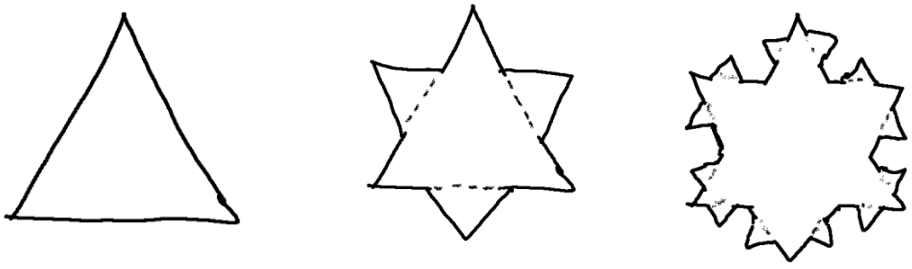
This design is set in the late evening outside, consisting of a Pythagoras Tree as the main fractal, with three Koch Snowflakes that you can click to place. The Pythagoras Tree fractal was invented by the Dutch mathematics teacher Albert E. Bosman in 1942[1]. The Koch Snowflake fractal is based on the Koch Curve fractal, which was first described by Helge von Koch in 1904[2]. The fractals used are generated using a recursive algorithm.

Image Generation

First, the Pythagoras Tree fractal is constructed from squares – one square as the base, then recursively adding two additional squares to the design. A gradient was applied starting at dark brown to green, which represents the green leaves on the tree branching off the brown trunk.



Next, the Koch Snowflake fractal is constructed from triangles – one triangle as the base, then recursively adding triangles to new edges created.

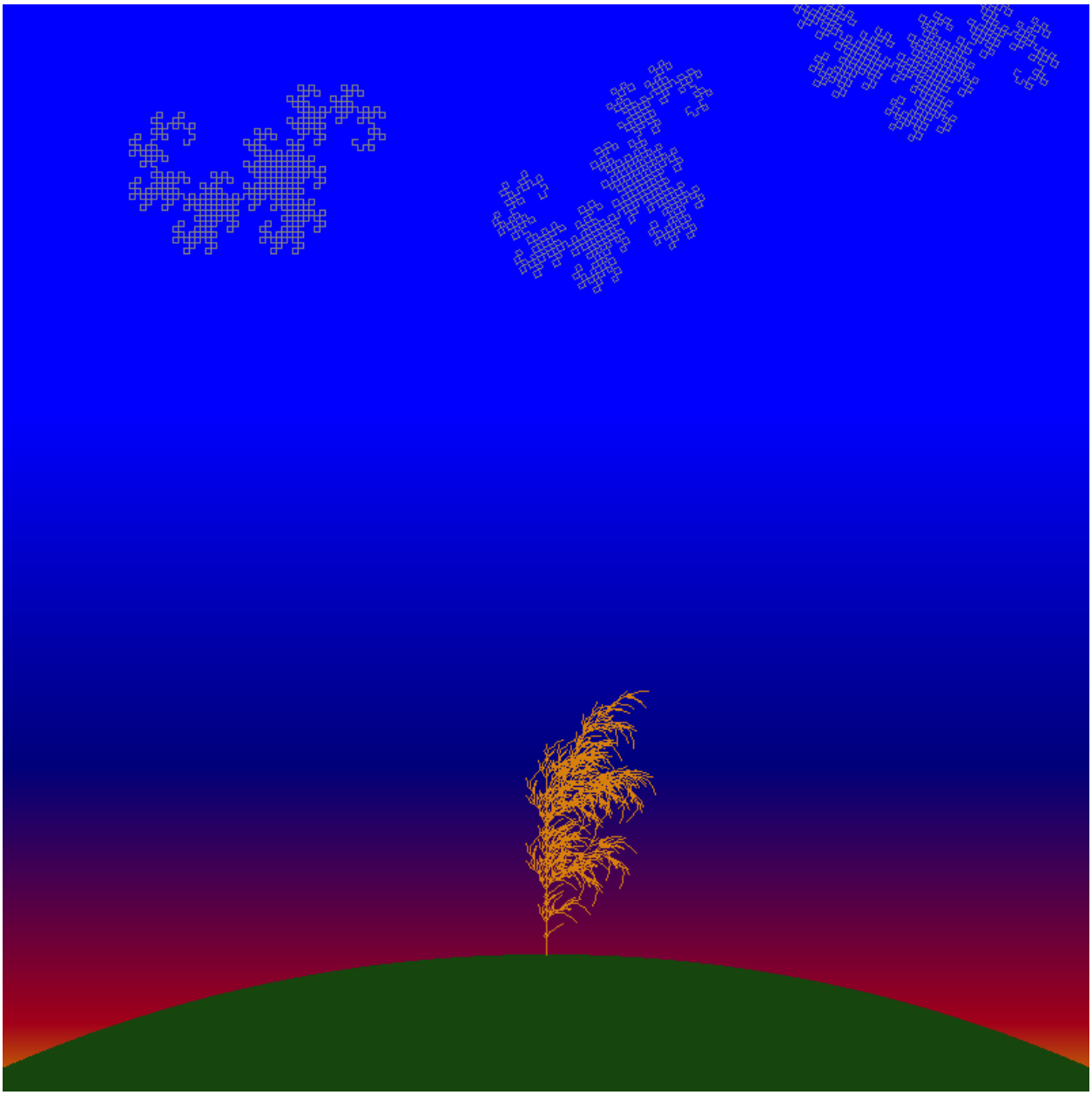


Lastly, a snowy ground was added with random points in the sky representing stars, with some clouds above created with five overlapping circles each. The sky was created using multiple lines to form a linear gradient.

[1] https://en.wikipedia.org/wiki/Pythagoras_tree_(fractal)
[2] https://mathworld.wolfram.com/KochSnowflake.html

L-Systems – Dragon Curve & Fern

Piece



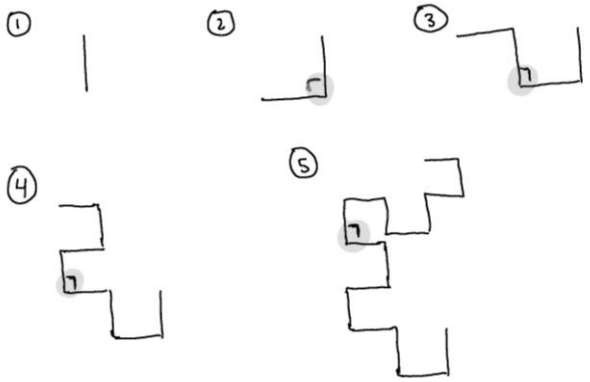
Information

This design is set in the morning, with the sun starting to rise, awakening the day. There's a lone fern, affected by the morning chills flowing towards the right. In the sky are some abstract clouds formed using the Dragon Curve. The Dragon Curve was first discovered in June 1966 a physicist named John Heighway who worked at NASA[3]. The fern grammar used was provided by Vexlio[4], which listed many different unique fern design grammars. The fractals used are generated using a Lindenmayer System, or "L-System", which takes an axiom to start with and consists of grammar rules to generate a string used for drawing[5].

Image Generation

The clouds in the sky are formed with the Dragon Curve fractal. This fractal starts with the axiom FX, and $+/-$ represents a 45-degree rotation, positive and negative respectively. F, X, and Y all move forward the same distance. The fractal string is generated using the following grammar rules:

$F \rightarrow Z$
$X \rightarrow +FX - -FY +$
$Y \rightarrow -FX + +FY -$

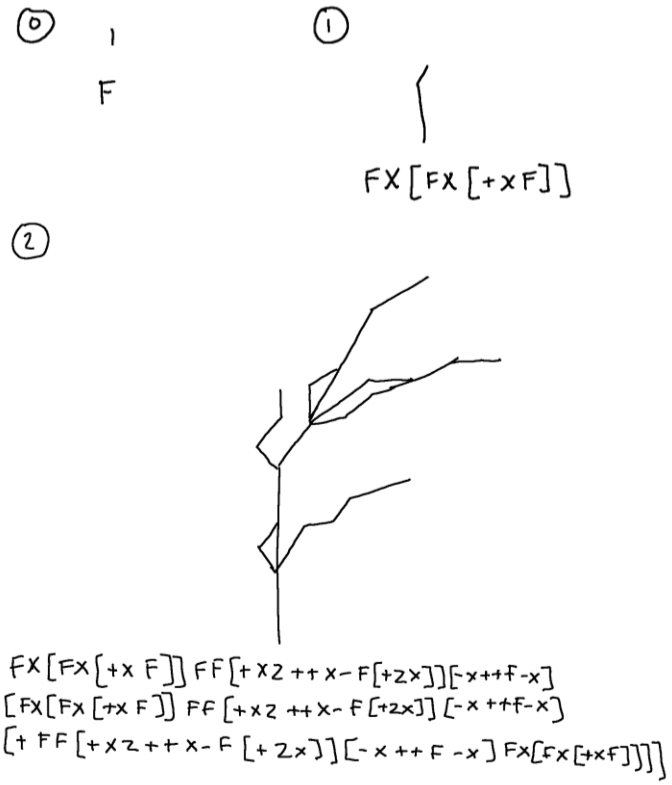


The dragon curve starts with a base line segment, then each segment is replaced by two segments with a right angle and with a rotation of 45 degrees. The replacement process is repeated until the maximum depth specified.

The fern was drawn in a similar way. The fractal starts with the axiom F, and $+/-$ represents a 22.5-degree rotation, positive and negative respectively. F, X, and Z all move forward the same distance. The fractal string is generated using the following grammar rules:
$F \rightarrow FX[FX[+XF]]$
$X \rightarrow FF\big[+XZ + +X - F[+ZX]\big][-X + +F - X]$

[3] http://5010.mathed.usu.edu/Fall2021/SFurfaro/Dragon%20Curve.html
[4] https://www.vexlio.com/blog/drawing-simple-organics-with-l-systems/
[5] https://en.wikipedia.org/wiki/L-system

$Z \rightarrow [+F - X - F][++ZX]$

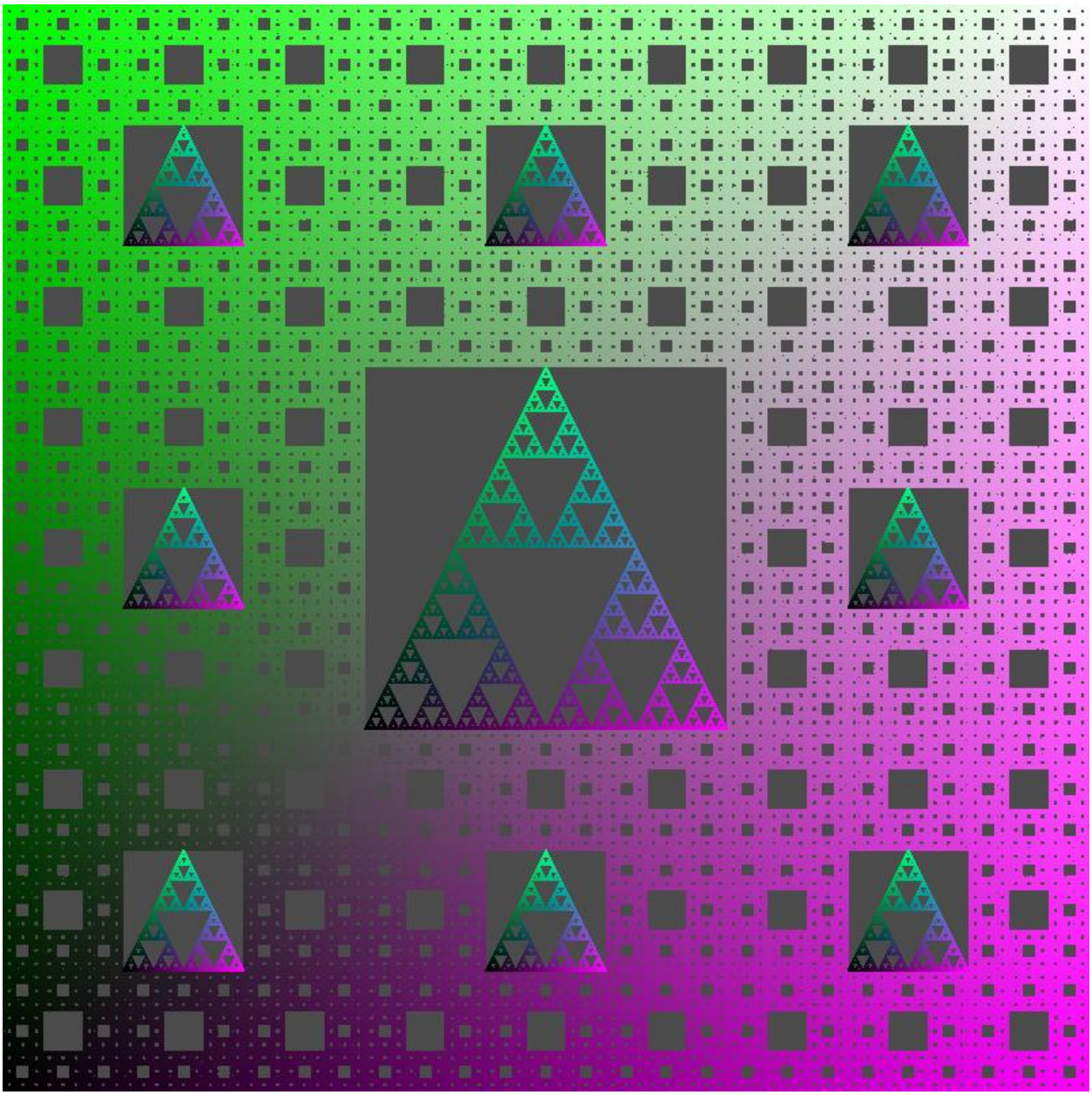


These steps are repeated until the maximum depth specified.

Finally, the morning sky was drawn using a gradient of lines, and the mountain the fern is on using a circle.

IFS – Sierpiński Carpet & Triangle

Piece

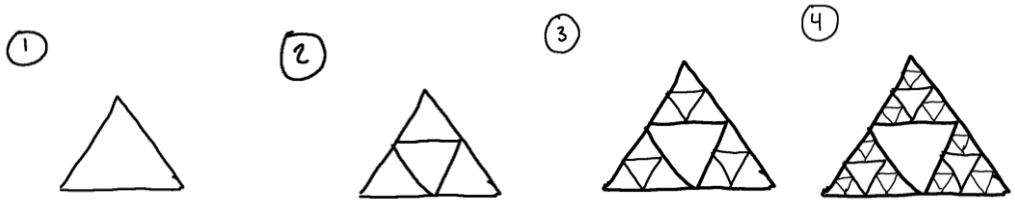


Information

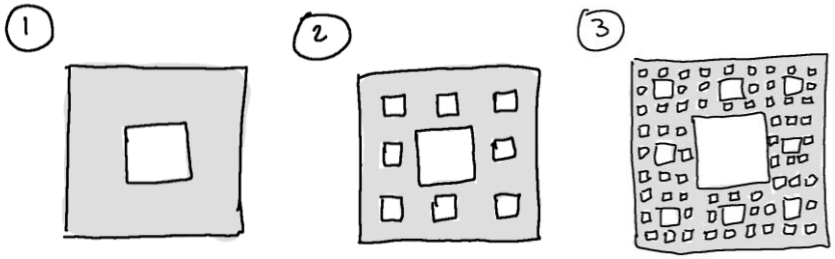
This is an abstract piece consisting of the Sierpiński Carpet fractal and nine Sierpiński Triangle fractals. The Sierpiński Carpet is a plane fractal first described by Wacław Sierpiński in 1916[6], and the Sierpiński Triangle is from 1915[7]. Sierpiński is well known for many different mathematical contributions and is well known for these fractals[8]. This abstract art piece was inspired by his contributions to not just fractals, but mathematics as a whole. These are drawn using an iterated function system, or "IFS" for short, which repeatedly applies a set of transformations and scaling to a point depending on the rules specified[9].

Image Generation

The triangles in the image are formed using the Sierpiński Triangle fractal, which is a very simple pattern of equilateral triangles. It starts with a single triangle, then three sub-triangles are created in the original created triangle. This step is repeated until the maximum specified depth is reached, where three sub-triangles are created in the previously created triangles. This fractal is displayed a total of nine times in the image: one larger triangle in the center, and eight smaller triangles around it.



The background portion of the image is formed using the Sierpiński Carpet fractal. This fractal is generated by drawing a square in the center, then eight squares are formed around it. This process is repeated until the specified maximum depth is reached, where eight squares are formed around the previously created squares.



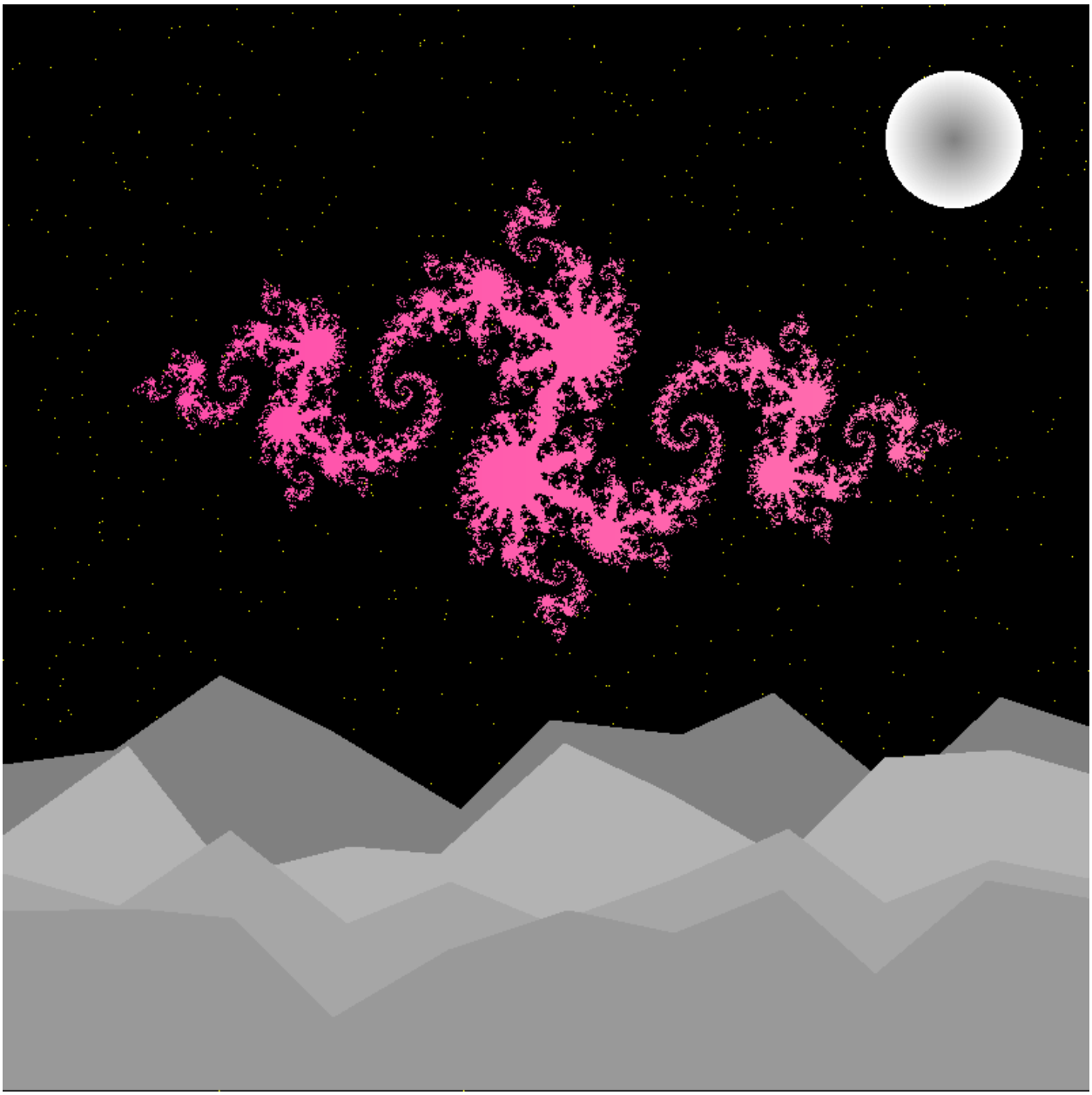
To create the color pattern, the x and y positions of the points being drawn are utilized in the RGB value calculation.

---

[6] https://en.wikipedia.org/wiki/Sierpi%C5%84ski_carpet
[7] https://protonstalk.com/triangles/sierpinski-triangle/
[8] https://en.wikipedia.org/wiki/Wac%C5%82aw_Sierpi%C5%84ski
[9] https://en.wikipedia.org/wiki/Iterated_function_system

Complex Number – Julia Set

Piece



Information

This design is inspired by the Julia Set, which is named after the French mathematician Gaston Julia who investigated their properties in 1915[10]. This set resembles a galaxy and was inspiration for an outdoor mountain landscape scene which contains the Julia Set in the sky.

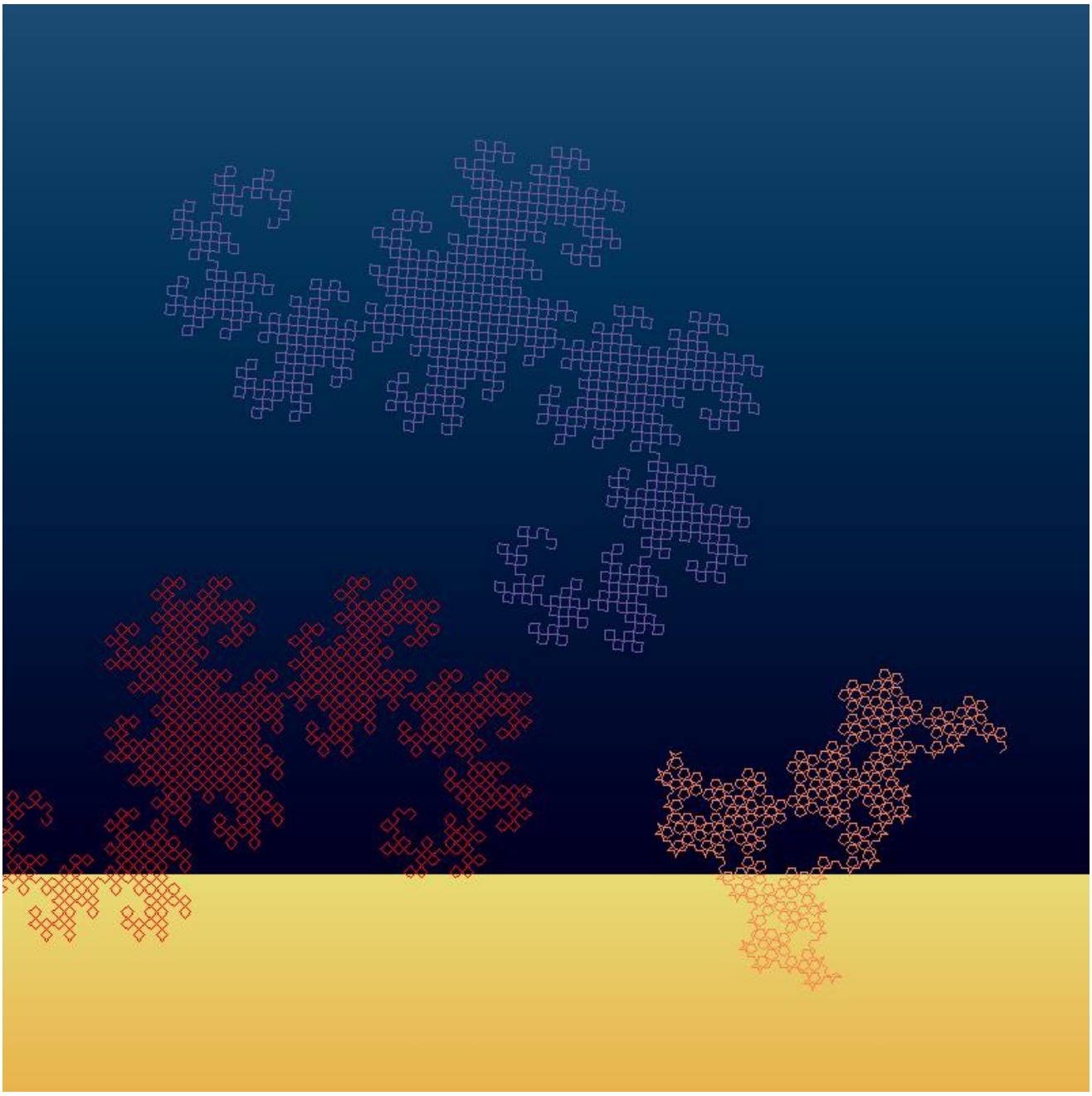
Image Generation

The Julia set is based on the complex equation: $f(z) = z^2 + c$, where z and c are complex numbers. The value of z gets changed every iteration, while the c value stays constant.

[10] https://complex-analysis.com/content/julia_set.html

Choice – Dragon Curve

Piece



Information

Our choice fractal was a combination of two, the McWorters Pentigree, and the Heighway Dragon. The Dragon Curve is often described as a sea dragon, paddling to the left. Both use a Lindenmeyer System to draw a fairly simple, but interesting shape. Closely resembling Coral, we chose both of these fractals to be close to the seabed to take on a minimalistic, yet abstract approach.

Image Generation

Heighway Dragon:

Rules:

Angle 45
F —> Z
X —> +FX−−FY+
Y —> −FX++FY−

Axiom FX

McWorter's Pentigree:

Rules:

Angle: 36
F —> +F++F−−−−F−−F++F++F−

Axiom: F

\+ Increases the angle
\- Decreases the angle

Code

```
/*
Complex
*/
#include "../FPToolkit/FPToolkit.c"

#define WIDTH 800.0
#define HEIGHT 800.0

#include <complex.h>
#include <stdio.h>
#include <math.h>

double swidth = 800, sheight = 800;


void julia(double rad, int mlimit) {
 double delta, x, y, xp, yp;
 complex c = (-0.8) + 0.156 * I;
 complex z;
 int reps;

 int divisions = 4;

 delta = 2 * rad / WIDTH;
 y = -rad;
 for (yp = 0; yp < WIDTH; yp++) {
 x = -rad;
 for (xp = 0; xp < WIDTH; xp++) {
 //the julia iteration:
 z = x + y * I;
 for (reps = 0; reps < mlimit; reps++) {
 if (cabs(z) > 2.0) {
 break;
 }
 z = z * z + c;
 }

 if (reps == mlimit) {
 G_rgb(0.988 + (xp / WIDTH) * 0.471, 0.275 + (xp / WIDTH) * 0.192, 0.667 + (xp / WIDTH) * 0.020);
 G_point(xp, yp + 100);

 }
 x = x + delta;
 }
 y = y + delta;
 }
}


void generateYValues(double yvals[12], double yOffset) {
 // create 10 y values
 for (int i = 0; i < 11; i++) {
```

```
 yvals[i] = rand() % 115 + yOffset;
 }

 // close the polygon
 yvals[10] = 0;
 yvals[11] = 0;
}

void generateXValues(double xvals[12]) {
 // create 10 x values between 0 and swidth,
 xvals[0] = 0;
 for (int i = 1; i < 10; i++) {
 xvals[i] = (swidth / 10) * i + rand() % 20;
 }

 // close the polygon
 xvals[10] = swidth * 2;
 xvals[11] = 0;
}

int main(int argc, char **argv) {

 // seed rand with the current time
 srand(time(NULL));

 int swidth, sheight;


 // must do this before you do 'almost' any other graphical tasks
 swidth = WIDTH;
 sheight = HEIGHT;
 G_init_graphics(swidth, sheight); // interactive graphics

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




 // draw stars throughout
 for (int i = 0; i < 1000; i++) {
 int x = rand() % (int) swidth;
 int y = rand() % (int) sheight;
 G_rgb(1, 1, 0);
 G_point(x, y);
 }

 // draw a moon at the top right using multiple circles of different sizes to create a gradient
 for (int i = 0; i < 50; i++) {
 // change color based on itteration to create a gradient
 G_rgb(1 - (i / 100.0), 1 - (i / 100.0), 1 - (i / 100.0));

 G_fill_circle(700, 700, 50 - i);
 }
```

```
julia(2, 100);

// change the color to gray
G_rgb(0.5, 0.5, 0.5);

// create 10 y values
double yvals[12];
double xvals[12];

for (int i = 200; i > 0; i -= 50) {
// create 10 x values between 0 and 800 randomly
generateXValues(xvals);
generateYValues(yvals, i);
G_fill_polygon(xvals, yvals, 12);

// pick a color based on i
G_rgb(0.5 + (i / 1000.0), 0.5 + (i / 1000.0), 0.5 + (i / 1000.0));
}


int key;
key = G_wait_key(); // pause so user can see results

// G_save_image_to_file("demo.xwd") ;
G_save_to_bmp_file("Demo.bmp");


exit(EXIT_SUCCESS);
}
```

```
/*
Recursive
*/
#include "../FPToolkit/FPToolkit.c"

#define WIDTH 800.0
#define HEIGHT 800.0
#define MAX_SNOWFLAKES 3

using namespace std;


/**
 * Draws a Pythagoreas Tree, starting at a current location up until a certain amount of iterations.
 * @param p0 Bottom left point of the tree
```

```
 * @param p1 Bottom right point of the tree
 * @param curr_iteration Current iteration of function
 * @param depth How many layers deep are we iterating into the tree
 * @param f Angle of triangle, ranges from 0.0 - .99
 */
void draw_tree(double *p0, double *p1, int curr_iteration, int depth, double f) {

 if(curr_iteration > depth)
 return;

 double p2[] = {0.0, 0.0};
 double p3[] = {0.0, 0.0};

 double x2 = p1[0] - (p1[1] - p0[1]);
 double y2 = p1[1] + (p1[0] - p0[0]);

 double x3 = p0[0] - (p1[1] - p0[1]);
 double y3 = p0[1] + (p1[0] - p0[0]);

 double xm = x3 + f * (x2-x3);
 double ym = y3 + f * (y2 - y3);

 double g = sqrt(f * (1-f));
 double xq = xm - g * (y2-y3);
 double yq = ym + g * (x2 - x3);

 G_rgb(.4 - ((double) curr_iteration / 50), .2 + ((double) curr_iteration / 10), 0.1);


 G_line(p0[0], p0[1], p1[0], p1[1]);
 G_line(p1[0], p1[1], x2, y2);
 G_line(x2, y2, x3, y3);
 G_line(p0[0], p0[1], x3, y3);

 // fill
 double x[] = {p0[0], p1[0], x2, x3};
 double y[] = {p0[1], p1[1], y2, y3};
 G_fill_polygon(x, y, 4);

 p2[0] = x2;
 p2[1] = y2;

 p3[0] = x3;
 p3[1] = y3;

 double q[] = {xq, yq};

 draw_tree(q, p2, curr_iteration + 1, depth, f);
 draw_tree(p3, q, curr_iteration + 1, depth, f);
}



/**
```

```
 *
 * @param p0
 * @param p1
 * @param depth
 */
void drawKochCurve(double p0[2], double p1[2], int depth) {
 if (depth <= 0) {
 return;
 }
 depth--;

 // pick a random color
 G_rgb(1.0, 1.0, 1.0);

 // calculate the 2 new points which are 1/3 of the way from the original points
 double p2[2], p3[2];
 p2[0] = p0[0] + (p1[0] - p0[0]) / 3;
 p2[1] = p0[1] + (p1[1] - p0[1]) / 3;
 p3[0] = p0[0] + (p1[0] - p0[0]) * 2 / 3;
 p3[1] = p0[1] + (p1[1] - p0[1]) * 2 / 3;

 // calculate the 3rd point of the triangle which is 60 degrees from the line
 double p4[2];
 p4[0] = p2[0] + (p3[0] - p2[0]) / 2 - (p3[1] - p2[1]) * sqrt(3) / 2;
 p4[1] = p2[1] + (p3[1] - p2[1]) / 2 + (p3[0] - p2[0]) * sqrt(3) / 2;

 // draw the 3 lines of the triangle
 G_line(p0[0], p0[1], p2[0], p2[1]);
 G_line(p2[0], p2[1], p4[0], p4[1]);
 G_line(p4[0], p4[1], p3[0], p3[1]);
 G_line(p3[0], p3[1], p1[0], p1[1]);

 // draw the other 3 lines of the triangle recursively
 drawKochCurve(p0, p2, depth);
 drawKochCurve(p2, p4, depth);
 drawKochCurve(p4, p3, depth);
 drawKochCurve(p3, p1, depth);
}



/**
 * This function takes a point on the screen and a radius essentially, then draws three koch curves around that point to
create
 * a snowflake.
 * @param point Location on the screen to draw the KochCurve
 * @param sideLength Radius of the snowflake.
 */
void draw_koch_at_point(double point[2], double sideLength) {


 double p0[2], p1[2], p2[2];

 p0[0] = point[0] - (sideLength / 2.0);
 p0[1] = point[1] - (sideLength * sqrt(3.0) / 4.0);
```

```
 p1[0] = point[0];
 p1[1] = point[1] + ((sideLength * sqrt(3.0)) / 4.0);

 p2[0] = point[0] + (sideLength / 2.0);
 p2[1] = point[1] - (sideLength * sqrt(3.0) / 4.0);

 drawKochCurve(p0, p1, 3);
 drawKochCurve(p1, p2, 3);
 drawKochCurve(p2, p0, 3);

}



/**
 * Draws a cloud at a location (x, y)
 * @param x X location to place cloud
 * @param y Y location to place cloud
 */
void draw_cloud(double x, double y) {
 G_rgb(.839, .839, .839);
 G_fill_circle(x, y, 25);
 G_fill_circle(x - 20, y, 20);
 G_fill_circle(x + 20, y, 20);
 G_fill_circle(x - 40, y, 15);
 G_fill_circle(x + 40, y, 15);
}



int main(int argc, char **argv) {
 int swidth, sheight;



 // must do this before you do 'almost' any other graphical tasks
 swidth = WIDTH;
 sheight = HEIGHT;
 G_init_graphics(swidth, sheight); // interactive graphics

 G_rgb(0.3, 0.3, 0.3); // dark gray
 G_clear();

 // Night sky gradiaent
 for(int i = 0; i < HEIGHT; i++) {
 G_rgb(0.0, 0.0, 0.0 + (double) i / 1000);
 G_line(0, 800 - i, 800, 800 - i);
 }



 // Star specs throughout whole screen
 G_rgb(.949, .937, .286);
 for(int i = 0; i < 1000; i++) {
 G_point(drand48() * WIDTH, drand48() * HEIGHT);
 }

 // Snow lines
 G_rgb(.94, .94, .916);
```

```
for(int i = 0; i < HEIGHT / 8.0; ++i) {
G_line(0, i, WIDTH, i);
}


// Drawing clouds around the top fo the screen
draw_cloud(100, 740);
draw_cloud(220, 730);
draw_cloud(346, 745);
draw_cloud(471, 737);
draw_cloud(602, 730);
draw_cloud(718, 747);



// Place tree
double p0[] = { WIDTH / 1.5, HEIGHT / 8.0};
double p1[] = { (WIDTH / 1.5) + 80, HEIGHT / 8.0};
draw_tree(p0, p1, 0, 10, .66);


// Drawing the snow covering the ground
G_rgb(.94, .94, .916);
for(int i = 0; i < 500; ++i) {
G_fill_circle(drand48() * WIDTH, HEIGHT / 8.0 - 5, 10 * drand48());
}


// Having user draw snowflakes
double click[] = {0.0, 0.0};
for (int i = 0; i < MAX_SNOWFLAKES; i++) {
G_wait_click(click);
draw_koch_at_point(click, 80);
}


int key;
key = G_wait_key(); // pause so user can see results
G_save_to_bmp_file("recursive.bmp");

exit(EXIT_SUCCESS);
}
```

/*
L-System
*/

```
/**
 * PythagorasTree.c
 */
#include "../FPToolkit/FPToolkit.c"
#include <stack>

#define WIDTH 800.0
#define HEIGHT 800.0
#define DEGREE_TO_RAD(x) ((x * M_PI) / 180)



void drawPlant(char *string, double distance, double angle, double *points);

void stringBuilderPlant(char *source, int depth);

void stringBuilderDragon(char *source, int depth);

void drawDragon(char *string, double distance, double angle, double *points);

int main() {

 double swidth, sheight;
// must do this before you do 'almost' any other graphical tasks
 swidth = 800.0;
 sheight = 800.0;
 G_init_graphics(swidth, sheight); // interactive graphics
// clear the screen with white
 G_rgb(.3, .3, .3);
 G_clear();
// ======================================================================




// Drawing the sky
 for (int i = 0; i < HEIGHT; ++i) {
 G_rgb(.8 - ((double) i / 300), .5 - ((double) i / 100), .2 * i / 100);
 G_line(0, i, WIDTH, i);
```

```
}

// Creating grass
G_rgb(.086, .271, 0.055);
G_fill_circle(WIDTH / 2.0, -900, 1000);


char str[1000000] = {'F'};
char drag[1000000] = {'F', 'X'};
stringBuilderDragon(drag, 10);
stringBuilderPlant(str, 4);


// Drawing clouds
double points[2] = {WIDTH / 6.0, HEIGHT - (HEIGHT / 8.0)};
G_rgb(.5, .5, .5);
drawDragon(drag, 2.0, 0.0, points);

points[0] += WIDTH / 6.0;
points[1] -= 40.0;
drawDragon(drag, 2.0, 30.0, points);

points[0] += WIDTH / 6.0;
points[1] += 90.0;
drawDragon(drag, 2.0, -30.0, points);


// Drawing lonely plant
points[0] = WIDTH / 2.0;
points[1] = 100.0;
G_rgb(.84, .5, .05);
drawPlant(str, 3.0, 90.0, points);


int key;
key = G_wait_key(); // pause so user can see results

G_save_to_bmp_file("turtle.bmp");
}


/**
 * Draws a line a certain length at a given angle
 * @param listOfPoints Location to draw from
 * @param lineLength Length of the line
 * @param angle Angle to draw at
 * @param increment If you want the function to increase the value of listOfPoints by lineLength
 * @param transparent True if you wish the drawing not to be seen
 */
void drawLineAtAngle(double *listOfPoints, double lineLength, double angle, bool increment, bool transparent) {
double radians = DEGREE_TO_RAD(angle);
double yLength = lineLength * sin(radians);
double xLength = lineLength * cos(radians);

double p2[2] = {listOfPoints[0] + xLength, listOfPoints[1] + yLength};
```

```
 if (!transparent) {
 G_line(listOfPoints[0], listOfPoints[1], listOfPoints[0] + xLength, listOfPoints[1] + yLength);
 }

 if (increment) {
 listOfPoints[0] += xLength;
 listOfPoints[1] += yLength;
 }
}


/**
 * This function draws a plant at a given location, it uses a make-shift stack to keep track of previously saved locations and angle so we create a
 * nice in depth photo
 * @param string Grammmar to draw from
 * @param distance Line length between each connecting piece
 * @param angle Angle between lines
 * @param points Starting point
 */
void drawPlant(char *string, double distance, double angle, double *points) {

 int arrSize = 0;
 double *storedX = nullptr;
 double *storedY = nullptr;
 double *storedAngle = nullptr;

 for (int i = 0; i < strlen(string); ++i) {
 if (string[i] == '[') {
 ++arrSize;
 }
 }

 storedX = new double[arrSize];
 storedY = new double[arrSize];
 storedAngle = new double[arrSize];
 int currentIndex = 0;


 for (int i = 0; i < strlen(string); ++i) {

 switch (string[i]) {
 case 'F':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case 'X':
 drawLineAtAngle(points, distance, angle, true, false);
 break;
 case 'Z':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case '[':
 storedX[currentIndex] = points[0];
```

```
 storedY[currentIndex] = points[1];
 storedAngle[currentIndex] = angle;
 ++currentIndex;
 break;

 case ']':
 --currentIndex;
 points[0] = storedX[currentIndex];
 points[1] = storedY[currentIndex];
 angle = storedAngle[currentIndex];

 break;
 case '+':
 angle -= 22.5;
 break;
 case '-':
 angle += 22.5;
 break;
 }
 }
}


/** function which creates a grammar rule in a char array of:
* Rules:
 * F -> FX[FX[+XF]]
 * X -> FF{+XZ++X-F[+ZX]][-X++F-X]
 * Z -> [+F-X-F][++ZX]
 * + -> Increase angle
 * - -> Decrease angle
 * [ -> Store location and angle
 * ] -> Go back to last location and angle
 * @param source Starting string
 * @param depth Number of iterations to do
 **/
void stringBuilderPlant(char *source, int depth) {

 char temp[1000000] = {'\0'};

 for (int i = 0; i < depth; ++i) {
 for (int j = 0; j < strlen(source); ++j) {
 switch (source[j]) {

 case 'F':
 strcat(temp, "FX[FX[+XF]]");
 break;

 case 'X':
 strcat(temp, "FF[+XZ++X-F[+ZX]][-X++F-X]");
 break;
 case 'Z':
 strcat(temp, "[+F-X-F][++ZX]");
 break;

 case '+':
 strcat(temp, "+");
```

```
 break;
 case '-':
 strcat(temp, "-");
 break;
 case '[':
 strcat(temp, "[");
 break;
 case ']':
 strcat(temp, "]");
 break;

 default:
 break;
 }
 }
 strcpy(source, temp);
 memset(temp, '\0', sizeof(temp));
 }

}


void stringBuilderDragon(char *source, int depth) {

 char temp[1000000] = {'\0'};

 for (int i = 0; i < depth; ++i) {
 for (int j = 0; j < strlen(source); ++j) {
 switch (source[j]) {

 case 'F':
 strcat(temp, "Z");
 break;

 case 'X':
 strcat(temp, "+FX--FY+");
 break;
 case 'Y':
 strcat(temp, "-FX++FY-");
 break;

 case '+':
 strcat(temp, "+");
 break;
 case '-':
 strcat(temp, "-");
 break;

 default:
 break;
 }
 }
 strcpy(source, temp);
 memset(temp, '\0', sizeof(temp));
 }
```

```
}


void drawDragon(char *string, double distance, double angle, double *points) {
 for (int i = 0; i < strlen(string) - 1; ++i) {
 switch (string[i]) {
 case 'F':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case 'X':
 drawLineAtAngle(points, distance, angle, true, false);
 break;
 case 'Y':
 drawLineAtAngle(points, distance, angle, true, false);
 break;
 case '+':
 angle -= 45.0;
 break;
 case '-':
 angle += 45.0;
 break;
 }
 }
}
```

/*
Iterated Function System
*/

```
#include "../FPToolkit/FPToolkit.c"

#define WIDTH 800.0
#define HEIGHT 800.0
#define DEGREE_TO_RAD(x) ((x * M_PI) / 180)

using namespace std;

double points[2] = {0.0, 0.0};
```

```
/**
 * Scales our global value points by both scale factors.
 * @param scaleFactorX
 * @param scaleFactorY
 */
void scale(double scaleFactorX, double scaleFactorY) {
 points[0] *= scaleFactorX;
 points[1] *= scaleFactorY;
}

/**
 * Translates our global value points by translation factors
 * @param transFactorX
 * @param transFactorY
 */
void translate(double transFactorX, double transFactorY) {
 points[0] += transFactorX;
 points[1] += transFactorY;
}



/**
 * Draws a Sierpinsky triangle on the screen restricted to a certain area with a certain scale factor.
 * @param min_x Restricts minimum location for triangle along X axis
 * @param min_y Restricts minimum location for triangle along Y axis
 * @param iterations Amount of points to place
 * @param scaleAmt Scale factor
 */
void drawTriangleIFS(double min_x, double min_y, int iterations, double scaleAmt) {

 for (int i = 0; i < iterations; ++i) {

 double random = drand48();

 if (random < .333) { // Draw bottom left quadrant
 translate(0.0, 0.0);

 G_rgb(1.0, 0.0, 0.0);
 } else if (random > .333 && random < .666) { // Top Left Quadrant


 translate(0.5, 0.0);

 G_rgb(0.0, 1.0, 0.0);

 } else {
 translate(.25, .5);

 G_rgb(0.0, 0.0, 1.0);
 }

 G_rgb(points[0] - points[1], points[1], points[0]);
 G_point((scaleAmt * points[0]) + min_x , (scaleAmt * points[1]) + min_y );
 scale(.5, .5);
```

```
 }

}

/**
 * Draws a Sierpinsky carpet across the entire screen
 * @param iterations Number of points to palce.
 */
void drawCarpet(int iterations) {
 for (int i = 0; i < iterations; ++i) {
 double random = drand48();
 if (random < (1.0 / 8.0)) { // Bottom left
 G_rgb(points[0], points[1], points[0] - points[1]);
 } else if (random < (2.0 / 8.0)) {
 G_rgb(1, 1, 0);
 translate(0.0, (1.0 / 3.0));
 } else if (random < (3.0 / 8.0)) {
 G_rgb(1, 0, 1);
 translate(0.0, (2.0 / 3.0));
 } else if (random < (4.0 / 8.0)) {
 G_rgb(0, 1, 1);
 translate((1.0 / 3.0), 0.0);
 } else if (random < (5.0 / 8.0)) {
 G_rgb(1, 0, 0);
 translate((1.0 / 3.0), (2.0 / 3.0));
 } else if (random < (6.0 / 8.0)) {
 G_rgb(0, 1, 0);
 translate((2.0 / 3.0), 0.0);
 } else if (random < (7.0 / 8.0)) {
 G_rgb(0, 0, 1);
 translate((2.0 / 3.0), (1.0 / 3.0));
 } else {
 G_rgb(0.5, 0, 1);
 translate((2.0 / 3.0), (2.0 / 3.0));
 }
 G_rgb(points[0], points[1], points[0]);
 G_point(WIDTH * points[0], HEIGHT * points[1]);
 scale((1.0 / 3.0), (1.0 / 3.0));
 }

}


int main(int argc, char **argv) {
 int swidth, sheight;

 // must do this before you do 'almost' any other graphical tasks
 swidth = WIDTH;
 sheight = HEIGHT;
 G_init_graphics(swidth, sheight); // interactive graphics

 G_rgb(0.3, 0.3, 0.3); // dark gray
 G_clear();
```

```
// Draw carpet across whole screen
drawCarpet(10000000);

for(int x = 1; x <= 9; x+=3) {
for(int y = 1; y <= 9; y+=3) {
// Skip the center square
if (x == 4 && y == 4) {
drawTriangleIFS(WIDTH / 3.0, HEIGHT / 3.0, 1000000, WIDTH / 3.0);
continue;
}
double minX = (WIDTH / 9) * x;
double minY = (HEIGHT / 9) * y;

drawTriangleIFS(minX, minY, 1000000, WIDTH / 9.0);

points[0] = 0.0;
points[1] = 0.0;
}
}

int key;
key = G_wait_key(); // pause so user can see results

G_save_to_bmp_file("l-system.bmp");


exit(EXIT_SUCCESS);
}
```

```
/**
 * Choice piece
 */
#include "../FPToolkit/FPToolkit.c"

#define WIDTH 800.0
#define HEIGHT 800.0
#define DEGREE_TO_RAD(x) ((x * M_PI) / 180)
```

```
/**
 * Draws a line a certain length at a given angle
 * @param listOfPoints Location to draw from
 * @param lineLength Length of the line
 * @param angle Angle to draw at
 * @param increment If you want the function to increase the value of listOfPoints by lineLength
 * @param transparent True if you wish the drawing not to be seen
 */
void drawLineAtAngle(double *listOfPoints, double lineLength, double angle, bool increment, bool transparent) {
 double radians = DEGREE_TO_RAD(angle);
 double yLength = lineLength * sin(radians);
 double xLength = lineLength * cos(radians);


 if (!transparent) {
 G_line(listOfPoints[0], listOfPoints[1], listOfPoints[0] + xLength, listOfPoints[1] + yLength);
 }

 if (increment) {
 listOfPoints[0] += xLength;
 listOfPoints[1] += yLength;
 }
}


/**
 * Draws a Heighway Dragon fractal on the screen
 * @param string Langauge to draw from
 * @param distance Line length
 * @param angle Current angle
 * @param points Starting position
 */
void drawCoral(char *string, double distance, double angle, double *points) {
 for (int i = 0; i < strlen(string); ++i) {
 switch (string[i]) {
 case 'F':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case 'X':
 drawLineAtAngle(points, distance, angle, true, false);
 break;
 case 'Y':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case '+':
 angle -= 45.0;
 break;
 case '-':
 angle += 45.0;
 break;
```

```
 }
 }
}


/**
 * Builds a grammar for the Heighway Dragon fractal
 * Rules:
 * F -> Z \n
 * X -> +FX--FY+ \n
 * Y -> -FX++FY- \n
 * + -> Increase angle by 45 degrees \n
 * - -> Decrease angle by 45 degrees \n
 * Axiom: "FX"
 * @param string String to write into
 * @param depth Number of iterations
 */
void coralStringBuilder(char *string, int depth) {

 char temp[1000000] = {'\0'};

 for (int i = 0; i < depth; ++i) {
 for (int j = 0; j < strlen(string); ++j) {
 switch (string[j]) {
 case 'F':
 strcat(temp, "Z");
 break;

 case 'X':
 strcat(temp, "+FX--FY+");
 break;
 case 'Y':
 strcat(temp, "-FX++FY-");
 break;
 case '+':
 strcat(temp, "+");
 break;
 case '-':
 strcat(temp, "-");
 break;

 default:
 break;
 }
 }
 strcpy(string, temp);
 memset(temp, '\0', sizeof(temp));
 }
}


/**
 * Draws a McWorter fractal to the screen. \n
 * @param string Grammar to draw
 * @param distance Distance of lines
 * @param angle Angle difference
 * @param points Starting position
```

```
 */
void drawMcworter(char *string, double distance, double angle, double *points) {

 for (int i = 0; i < strlen(string); ++i) {
 switch (string[i]) {
 case 'F':
 drawLineAtAngle(points, distance, angle, true, false);
 break;

 case '+':
 angle -= 36.0;
 break;
 case '-':
 angle += 36.0;
 break;
 }
 //G_wait_key();
 }
}

/**
 * Creates Creates a string containing necessary grammar to draw a McWorter Pentigree \n
 * Rules: \n
 * F -> +F++F----F--F++F++F- \n
 * + -> Increase angle 36.0 degrees \n
 * - -> Decrease angle 36.0 degrees \n
 * Axiom: \n
 * F \n
 * @param str String to write into
 * @param depth Number of iterations
 */
void buildMcworter(char *str, int depth) {
 char temp[1000000] = {'\0'};

 for (int i = 0; i < depth; ++i) {
 for (int j = 0; j < strlen(str); ++j) {

 switch (str[j]) {
 case 'F':
 strcat(temp, "+F++F----F--F++F++F-");
 break;

 case '+':
 strcat(temp, "+");
 break;
 case '-':
 strcat(temp, "-");
 break;

 default:
 break;
 }
 }
 strcpy(str, temp);
 memset(temp, '\0', sizeof(temp));
 }
```

```
}


int main() {
 double swidth, sheight;

 // must do this before you do 'almost' any other graphical tasks
 swidth = 800.0;
 sheight = 800.0;
 G_init_graphics(swidth, sheight); // interactive graphics

 // clear the screen with white
 G_rgb(.3, .3, .3);

 G_clear();

 // Dark ocean gradient
 for (int i = 0; i < HEIGHT; ++i) {
 double r = 0.1059 - (double) i / 2000;
 double g = 0.3020 - (double) i / 2000;
 double b = 0.4471 - (double) i / 2000;
 G_rgb(r, g, b);
 G_line(0, HEIGHT - i, WIDTH, HEIGHT - i);
 }

 // Draw sand

 for(int i = 0; i < HEIGHT / 5.0; ++i) {
 G_rgb(.91, .7 + (i / 1000.0), .3 + (i / 1000.0));
 G_line(0.0, i, WIDTH, i);

 }


 double coralPosition[2] = {300.0, 200.0};
 char coral[1000000] = {'F', 'X'};
 coralStringBuilder(coral, 11);
 G_rgb(1.0, 0.0, 0.0);
 drawCoral(coral, 3.0, 180.0, coralPosition);

 G_rgb(255.0 / 255.0, 127.0 / 255.0, 80.0 / 255.0);
 double worterPosition[2] = {500.0, 250.0};
 char worter[100000] = {'F'};
 buildMcworter(worter, 4);
 drawMcworter(worter, 5.0, 0.0, worterPosition);


 G_rgb(.467, .337, .651);
 coralPosition[0] = WIDTH / 2.0;
 coralPosition[1] = HEIGHT / 2.0;
 drawCoral(coral, 3.5, 130.0, coralPosition);


 int key;
 key = G_wait_key(); // pause so user can see results
```

```
 G_save_to_bmp_file("turtle.bmp");
}
```