CS410P: Exploring Fractals

A Mathematical and Graphical Programming Portfolio
by Alex Staley

Table of Contents

Table of Contents 1

Snowy Night 2
- Fractal Description 3
- Design Paradigm & Mathematical Description 3
- Artistic Description 4

Nautiliteratorus 5
- Fractal Description 6
- Design Paradigm & Mathematical Description 6
- Artistic Description 6

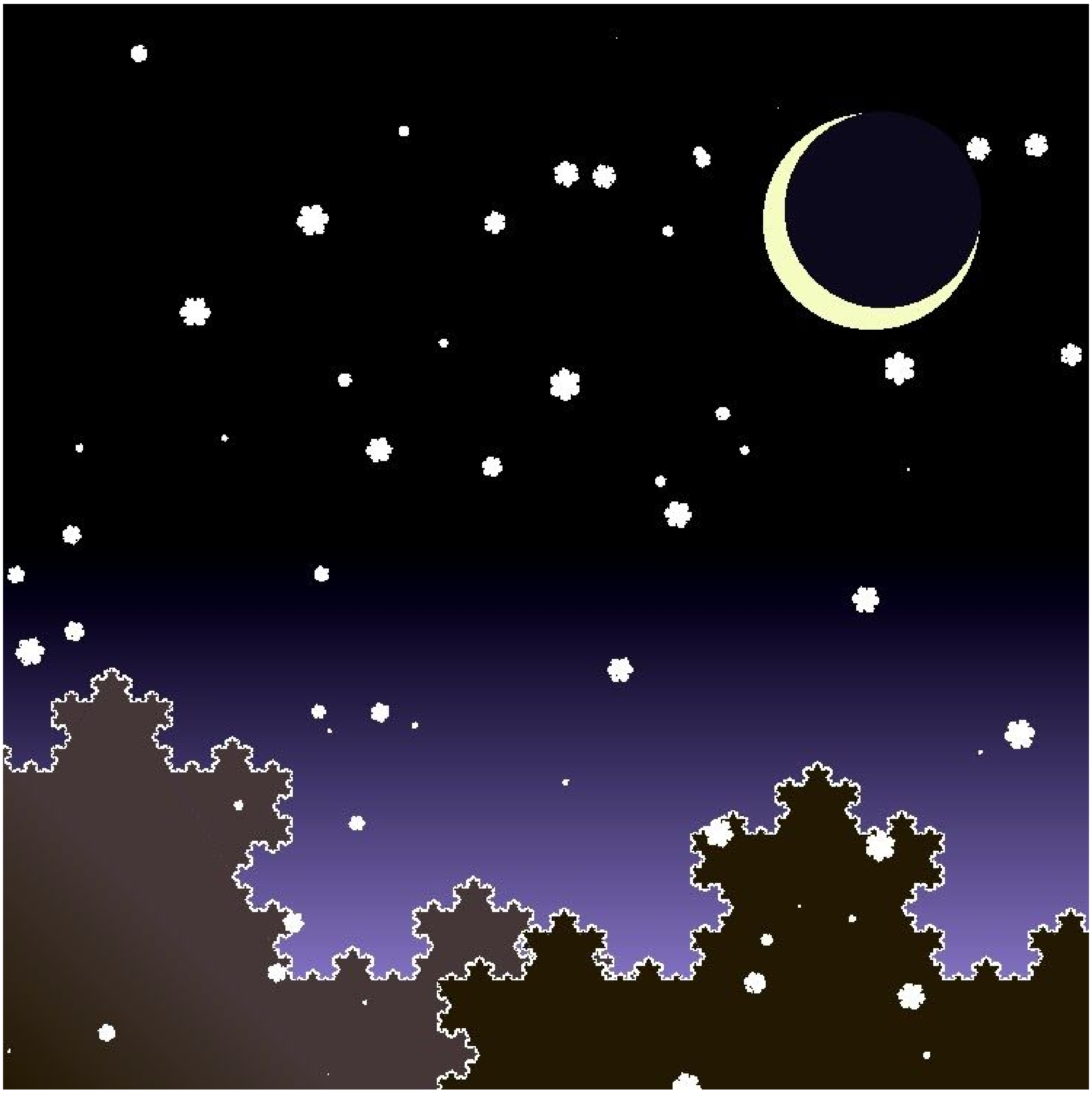
Mandelbrot's Ghost 7
- Fractal Description 8
- Design Paradigm & Mathematical Description 8
- Artistic Description 9

Baphomet 10
- Fractal Description 11
- Design Paradigm & Mathematical Description 11
- Artistic Description 12

Desert Life 14
- Fractal Description 15
- Design Paradigm & Mathematical Description 15
- Artistic Description 16

Source Code 17
- Snowy Night 17
- Nautiliteratorus 21
- Mandelbrot's Ghost 24
- Baphomet 26
- Desert Life 31

Snowy Night

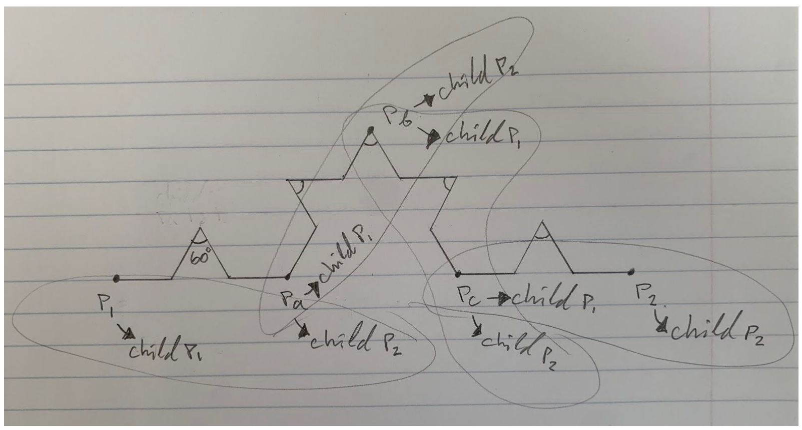


Fractal Description

The Koch Curve is a fractal first described by Swedish mathematician Helge von Koch in 1904. It is described by the transformation of the middle third of a line segment into two legs of an equilateral triangle, where the missing third leg is equal to the middle third taken from the original line segment. The Koch Curve can be iterated over the three legs of an equilateral triangle to form a Koch Snowflake.

Cited: https://en.wikipedia.org/wiki/Koch_snowflake

Design Paradigm & Mathematical Description



Both mountain and snowflake instances of the fractal were generated in the recursive paradigm. Each iteration of the `kochCurve()` function is given two points `p1` and `p2`, representing the endpoints of the parent line segment. The ⅓ and ⅔ points `pa` and `pc` are calculated via linear blending of `p1` and `p2`:

$$pa = p1 + \tfrac{1}{3}(p2 - p1) \qquad \text{(S1)}$$

$$pc = p1 + \tfrac{2}{3}(p2 - p1) \qquad \text{(S2)}$$

Equations (S1) and (S2) are of course implemented component-wise. The “top” vertex of the equilateral triangle being described is denoted `pb` and is calculated as a rotation of 60° about `p1`:

$$pb_x = pa_x + pt_x \cdot cos(60°) - pt_y \cdot sin(60°) \qquad \text{(S3)}$$

$$pb_y = pa_y + pt_y \cdot cos(60°) + pt_x \cdot sin(60°) \qquad \text{(S4)}$$

where $pt = pa - p1$ together with the addition of `pa` describes rotation about the origin.

The Koch Curve is iterated three times over three line segments forming an equilateral triangle in the `kochFlake()` function which, given two points, calculates the third point necessary for an equilateral triangle using the rotation mechanism from equations (S3) and (S4) and then calls the `kochCurve()` function for each leg of the resulting triangle.

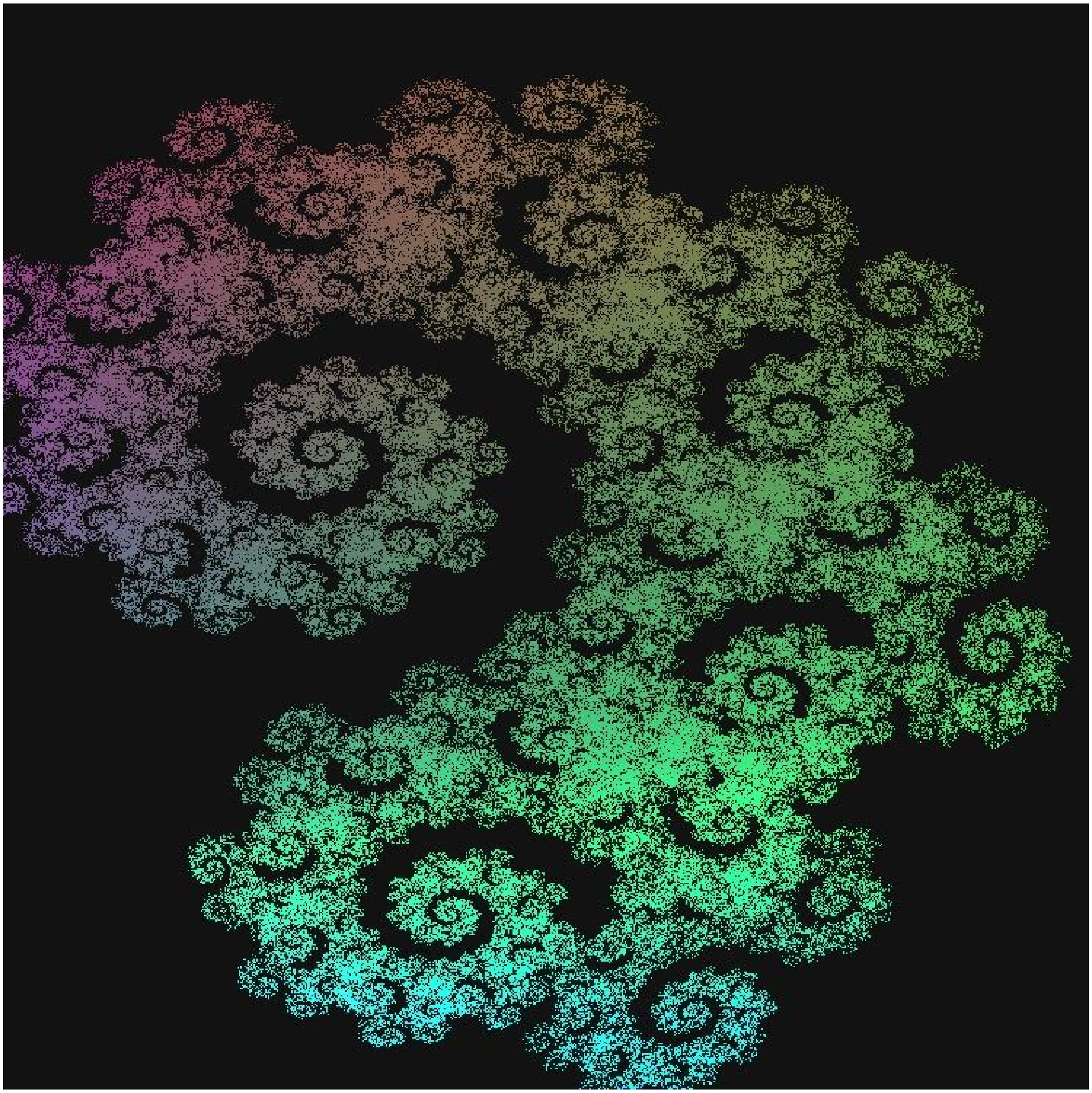
Artistic Description

The snowflakes are located, sized and oriented by selecting random numbers within specified bounds. The mountains are drawn in white at the deepest iteration of the Koch Curve to give the impression of snowcaps, but their overall brown colors were chosen to complement the purple of the twilight sky and shaded to give the impression of relative depth.

The color fading in the sky and in the bottom left corner of the image is accomplished by drawing successive lines in colors that scale incrementally from a dark to a light shade via mathematics similar to the linear blending used in the construction of the fractal.

The moon is constructed using two partially overlapping circles, the smaller of the two drawn in a color barely lighter than that part of the sky to give the impression of the shaded part of the moon.

Nautiliteratorus



Fractal Description

The Nautiliteratorus, while owing its origins to the Barnsley Fern fractal first described by British mathematician Michael Barnsley in 1988, is a unique design. It is made of three parts, each of which describes a nautilus- or torus-like spiral design, and is implemented using an iterative function.

Cited: https://en.wikipedia.org/wiki/Barnsley_fern

Design Paradigm & Mathematical Description

The Nautiliteratorus is designed in the iterative function system paradigm. A number between 0 and 1 is chosen at random; its value determines according to which of three rules a point will be processed. Each rule consists of a combination of linear transformations, scaling operations, and rotations. The processed point is then taken through a general transformation before being fed back into the next iteration of the function.

I cannot provide a more rigorous mathematical description of the shape of the Nautiliteratorus, since the design arises from the hard-coded parameters, rather than from deliberate mathematical formulae.

Artistic Description

I discovered this fractal by starting from the Barnsley Fern design and playing with parameters until I arrived at something fun and paisley-like; I then fine-tuned the parameters to where they are. The spirals repeating around the outer edge of the fractal are a bit squished, helping to give the illusion of depth.

The colors are functions of the points' locations on the screen and were chosen to enhance the depth aspect.

Mandelbrot’s Ghost

Fractal Description

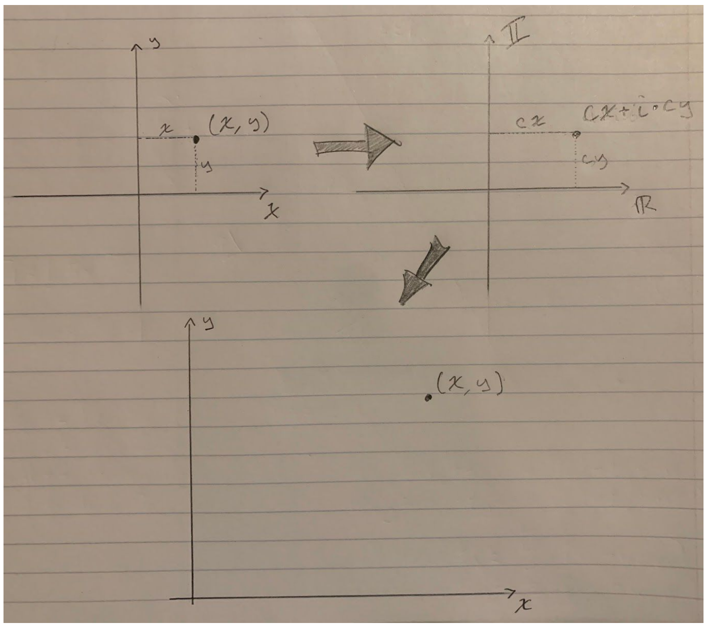
The visualization of the Mandelbrot Set was first depicted by Robert W. Brooks and Peter Matelski in 1978. The set itself is defined as the set of complex numbers c for which the function

$$z \longrightarrow z^2 + c \tag{M1}$$

does not diverge when iterated from $z = 0$. When mapped onto the (x, y) plane, the Mandelbrot Set creates one of the most famous fractal images in the world.

Cited: https://en.wikipedia.org/wiki/Mandelbrot_set

Design Paradigm & Mathematical Description



The fractal is generated using the complex number paradigm. Each point in the (x, y) plane is mapped to the complex plane as the point (cx, cy), accounting for scaling and translation when fitting the image to the screen:

$$cx = \frac{3x}{sw} - 2.15 \tag{M2}$$

$$cy = \frac{3y}{sh} - 1.5 \tag{M3}$$

Then equation (M1) is iterated a given number `depth` of times and the result tested for divergence. The point (x, y) is plotted according to the result.

After every point in the screen has been plotted in this way, the whole process repeats with `depth` having been incremented by 1, resulting in a more precise rendering of the set being layered over the previous rendering in a lighter shade.

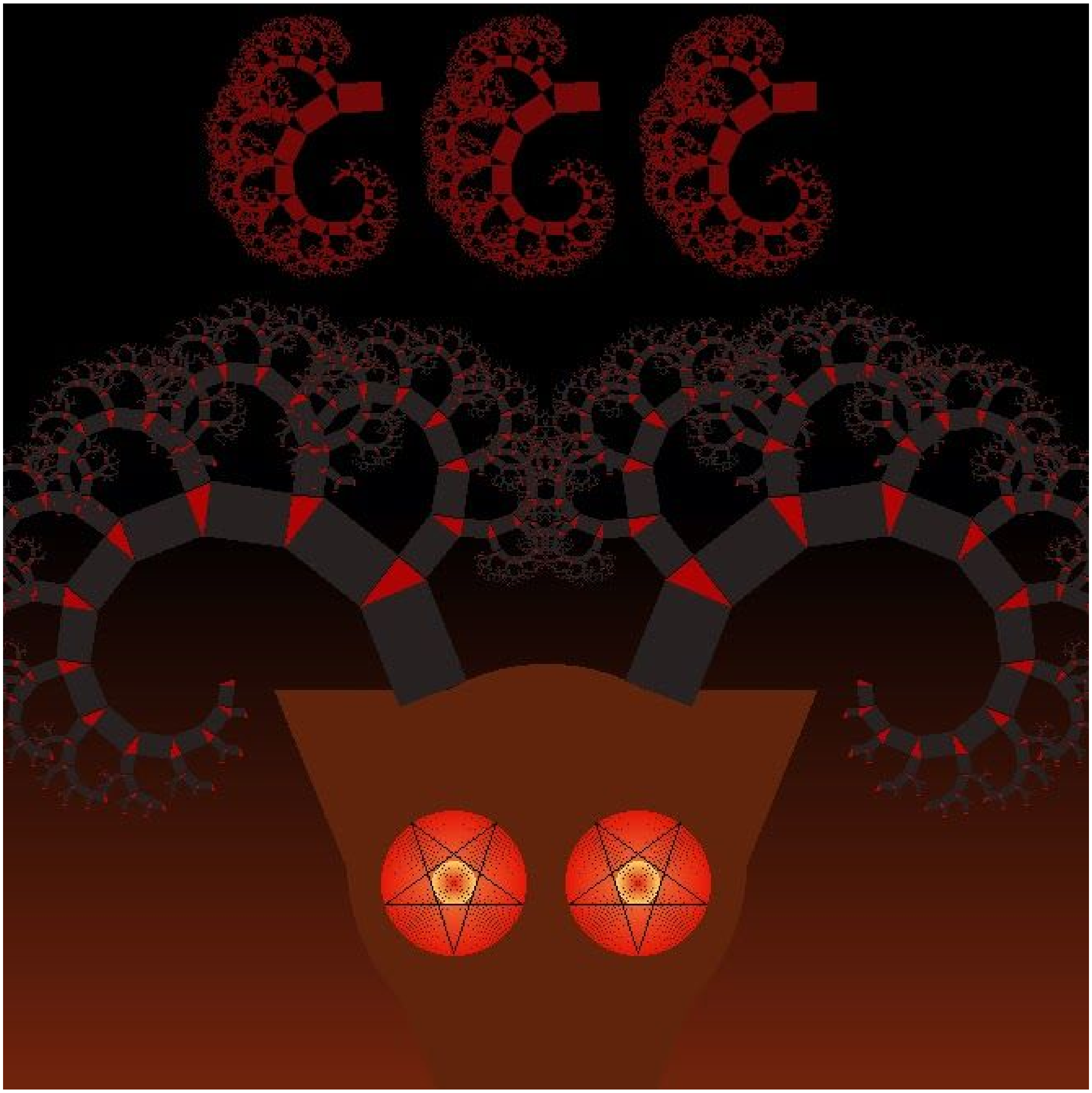
Artistic Description

I wanted to have a black-and-white entry in this portfolio, and I am pleased that I ended up filling that niche with the Mandelbrot set, since it is so often depicted in striking color. I also was pleased with the result of varying the shade according to the number of iterations of equation (M1), which created the dark “contour-map” effect around the shape, and also softened its edges as the value of `depth` was increased. The softness of the edges was especially refreshing after creating so many hard-line graphics using the recursion and L-system paradigms.

I did experiment with several more intricate color, shading and pattern ideas for both the background and the interior of the set, but nothing I was able to come up with was anywhere near as striking as this simple, soft grayscale ghost.

I turned it 90° because it’s more haunting that way.

Baphomet

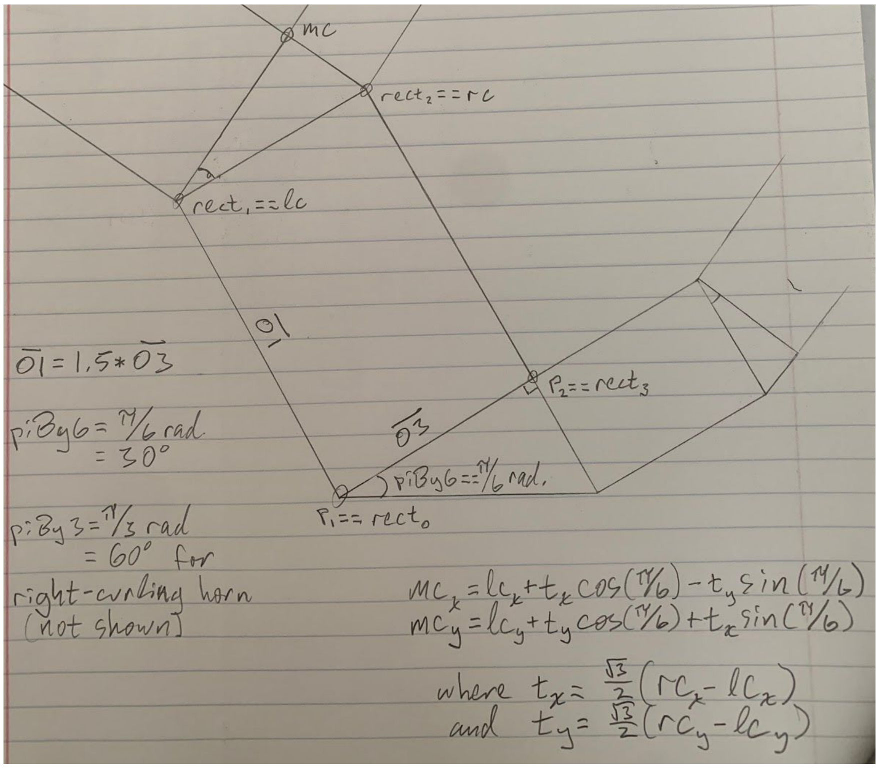


Fractal Description

The Pythagoras Tree is a fractal first described by Dutch mathematics teacher Albert E. Bosman in 1942. It is described by the perpendicular extension of a given line segment to form a rectangle, the construction of a right triangle whose hypotenuse is congruent with the “top” side of the rectangle, and the repetition of these two steps using each leg of the triangle as the line segment that seeds the next iteration.

Cited: https://en.wikipedia.org/wiki/Pythagoras_tree_(fractal)

Design Paradigm & Mathematical Description



The fractal was generated in the recursive paradigm. Given the endpoints of a line segment `p1` and `p2`, the other two points of the rectangle (x- and y-coordinates denoted `rectX` and `rectY`, respectively) is calculated using right-angle movement:

$$rectX_1 = p1 - leg2 \tag{B1}$$
$$rectX_2 = p2 - leg2 \tag{B2}$$
$$rectY_1 = p1 + leg1 \tag{B3}$$
$$rectY_2 = p2 + leg1 \tag{B4}$$

where $leg2 = 1.5 \cdot (p2 - p1)$ for the y-coordinates of `p1` and `p2` and $leg1 = 1.5 \cdot (p2 - p1)$ for the x-coordinates of `p1` and `p2`.

The third point of the right triangle is calculated using linear blending of the newly calculated rectangle vertices, combined with rotation about the “left” vertex:

$$mc_x = lc_x + t_x\,[cos(\alpha)] - t_y\,[sin(\alpha)] \qquad \text{(B5)}$$
$$mc_y = lc_y + t_y\,[cos(\alpha)] + t_x\,[sin(\alpha)] \qquad \text{(B6)}$$

where $t = f \cdot (rc - lc)$ component-wise in the linear blending portion of the calculation. The right-curling fractals use the values $f = 0.5$ and $\alpha = 60°$, while the left-curling fractals use the values $f = \frac{\sqrt{3}}{2}$ and $\alpha = 30°$.

Artistic Description

Extending the square part of the fractal into an $s \times 1.5 \cdot s$ rectangle and fixing the working angle at 30° gave the impression of an ibex horn, which inspired me to summon Baphomet. This decision forced an entirely red color spectrum, the shades chosen to evoke fire and brimstone. At first afraid to proceed in the dark, I consulted PSU Library’s copy of the Necronomicon for some hint of how the Summoning might best be effected, but after several terrifying nightmares I set that horrid grimoire aside and pressed on alone.

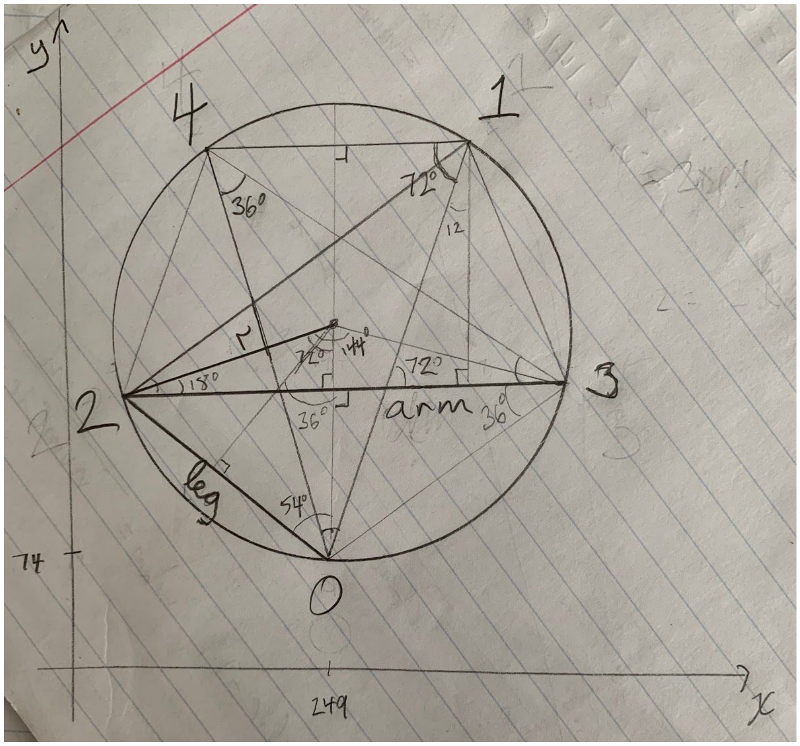
Baphomet’s head is described by an “upside-down” isosceles triangle and layered circle. Three arcs of the circle are visible outside the triangle, the top arc intersecting with the triangle’s base at points covered by the roots of the horns, giving the impression of a continuous head shape. At this point the Daemon’s power became manifest, and in his horrible Presence I was compelled by eldritch Forces to inscribe the mark of the beast above the creature using three more (upside-down) left-curling fractals.

The eyes, which required (upside-down) pentagrams and blazing color-faded circles, proved to be the most mathematically intensive part of the Summoning. The key vertices (x_1, y_1) and (x_3, y_3) of the pentagram (see points ‘1’ and ‘3’ in the diagram below) are defined by rotating two line segments of lengths `arm` and `leg` about the lower vertex (x_0, y_0) of the pentagram, which has coordinates given by the center coordinates and radius `r` of the eyes. Considering (x_0, y_0) as known:

$$x_1 = x_0 + arm \cdot cos(72°) \qquad \text{(B7)}$$
$$y_1 = y_0 + arm \cdot sin(72°) \qquad \text{(B8)}$$
$$x_3 = x_0 + leg \cdot cos(36°) \qquad \text{(B9)}$$
$$y_3 = y_0 + leg \cdot sin(36°) \qquad \text{(B10)}$$

where $arm = 2r \cdot sin(72°)$ and $leg = 2r \cdot sin(36°)$. The remaining vertices are calculated by copying (for the y-coordinates) and subtracting `arm` and `leg` from (for the x-coordinates) the known vertices:

$$x_2 = x_3 - arm \qquad \text{(B11)}$$
$$x_4 = x_1 - leg \qquad \text{(B12)}$$



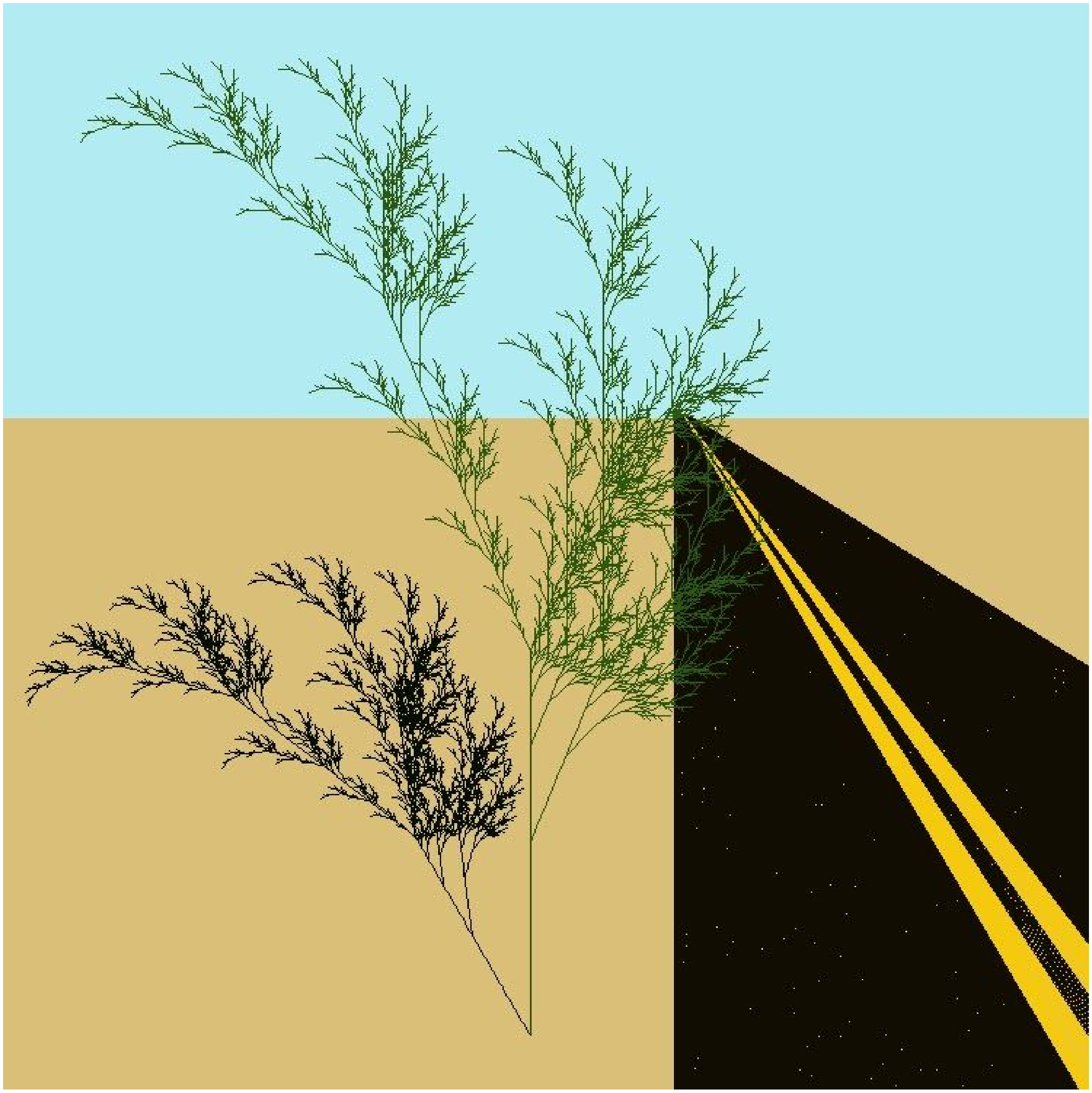
The diagram's proportions are not faithful — it was drawn during the heat of the Summoning, when my cowering mind had been perverted by the impossible geometries of some terrible unseen Cosmos.

The eyes themselves are described by red circles circumscribed around the outer vertices of the pentagram, along with a yellow incircle of the inner pentagon formed by the pentagram, which fades to a slightly brighter red at the center of the eye.

I had initially intended to use linear blending to fade the outer red into the yellow incircle as well as fading the yellow into the center red, but at a point during implementation I accidentally coded the existing eye colors, which do not fade from the circumscribed outer circle to the incircle. At that point, using some dark and terrible magicks, Baphomet wrested control of the project from me, and I have been unable to change the outer eye colors ever since.

I got the sense he was disappointed in me for avoiding the problem of his mouth by extending his face below the bottom of the screen, giving him a somewhat goofy peeking-over-the-fence aspect. Blending of the yellow incircle to the red eye center, apparently approved by Baphomet, was successful.

Desert Life



Fractal Description

This plant fractal is a well-known example of the capability of Lindenmayer systems. I copied the two production rules from [Paul Bourke](http://paulbourke.net/fractals/lsys/), who has a number of L-system examples posted on his website. [Lindenmayer systems](https://en.wikipedia.org/wiki/L-system) themselves were developed by the Hungarian biologist Aristid Lindenmayer in 1968. They comprise a formal grammar, using production rules to derive strings from a nonterminal starting character, then applying movement and drawing rules to each character in the derived string.

Cited: http://paulbourke.net/fractals/lsys/
Cited: https://en.wikipedia.org/wiki/L-system

Design Paradigm & Mathematical Description

The fractal is generated in the Lindenmayer Systems paradigm. Given the starting character `A`, two production rules are defined:

```
A -> B-[[A]+A]+B[+BA]-A        (D1)
B -> BB                        (D2)
```

The `-` and `+` characters indicate right and left turns of 22.5°, the `]` and `[` characters indicate "pop" and "push" operations on a stack representing the current state (position and heading), and the letter characters `A` and `B` indicate forward motion for a specified length `length`.

After the string has been derived, it is processed by the `autoFit()` function, which determines a bounding rectangle $dX = xMax - xMin$, $dY = yMax - yMin$ for the resulting image when $length = 1$. It then determines the scaling factor required to fit the long dimension (in this case `dY`) of the bounding rectangle to 90% of its corresponding screen dimension (`sheight`), and applies that scaling factor to determine the ideal value for `length`:

$$length = \frac{ndX}{dX} \qquad \text{(D3)}$$

where $ndX = dX \cdot (\frac{ndY}{dY})$ for $ndY = 0.9 \cdot sheight$.

`autoFit()` also calculates the ideal starting point to center the fractal in the screen by taking half of the difference between the screen dimension and the scaled bounding rectangle, keeping in mind the case of a negative value for `xMin`, as we happen to be have:

$$start_x = \frac{1}{2}(swidth - ndX) - xMin \cdot (\frac{ndX}{dX}) \qquad \text{(D4)}$$
$$start_y = \frac{1}{2}(sheight - ndY) \qquad \text{(D5)}$$

After determining the dimensions and placement, the fractal is shifted left of center to account for the road on the right side of the image.

Artistic Description

The countenance of this plant is both serene and austere. It spoke to me of standing alone in the desert near a sand-swept road. The shadow just needed another iteration interpreting the derived string, with a shorter length and a greater starting angle.

Random dots of brown represent sand blown across the road, slightly softening the severity of the image and drawing attention away from the failure of the road to completely cover the desert color between the center lines in the bottom-right corner of the image.

The straight lines and stark colors are meant to evoke the vastness and brightness of the desert. To emphasize these qualities I chose to avoid fading the colors of the sand and sky.

Source Code

Snowy Night

```
/*
Alex Staley - CS410P - July 2020

  KOCH SNOWFLAKES FALLING
  ON SNOWCAPPED KOCH MOUNTAINS
  UNDER A TWILIGHT SKY
*/

#include  "FPToolkit.c"
const double piBy3 = M_PI / 3.0;

void sky(int swidth, int sheight, double colorBot[], double colorTop[]);
void mountains(int swidth, int sheight, double colorDark[], double colorLite[]);
void cornerFade(double colDark[], double colLite[], double xMax);
void kochFlake(double p1[], double p2[], double color[], int depth);
void kochCurve(double p1[], double p2[], double color[], int curr, int dep);

int main() {
  int swidth, sheight;
  int flakes, depth, orient;
  double p1[2], p2[2], p3[2];
  double skyColorTop[3], skyColorBot[3];
  double mtColorDark[3], mtColorLite[3];

  // Set up display environment
  srand(time(0));
  flakes = 60;
  depth = 6;
  orient = 0;
  swidth = 746;
  sheight = 746;
  //G_choose_repl_display();
  G_init_graphics (swidth,sheight);

  // Night sky
  skyColorBot[0] = 166.0 / 255.0;
  skyColorBot[1] = 145.0 / 255.0;
  skyColorBot[2] = 242.0 / 255.0;
  skyColorTop[0] = 22.0 / 255.0;
  skyColorTop[1] = 15.0 / 255.0;
  skyColorTop[2] = 48.0 / 255.0;
  sky(swidth, sheight, skyColorBot, skyColorTop);

  // Mountains
  mtColorDark[0] = 35;
  mtColorDark[1] = 25;
  mtColorDark[2] = 3;
  mtColorLite[0] = 68;
  mtColorLite[1] = 55;
```

```
  mtColorLite[2] = 55;
  mountains(swidth, sheight, mtColorDark, mtColorLite);

  // Moon
  Gi_rgb(245, 252, 193);
  G_fill_circle(swidth*0.8, sheight*0.8, sheight*0.1);
  G_rgb(0.6*skyColorTop[0], 0.6*skyColorTop[1], 0.6*skyColorTop[2]);
  G_fill_circle(swidth*0.81, sheight*0.81, sheight*0.09);

  // Snowflakes
  double white[3] = {255, 255, 255};
  G_rgb(1,1,1);
  for (int i=0; i<flakes; ++i) {
    p1[0] = rand() % (swidth-10);
    p1[1] = rand() % (sheight-10);
    orient = rand();
    p2[0] = p1[0] + (i%20)*cos(orient);
    p2[1] = p1[1] + (i%20)*sin(orient);
    kochFlake(p1, p2, white, depth);
  }

  // Display and save image
  G_wait_key();
  G_save_to_bmp_file("snowyNight.bmp");

  return 0;
}

// Fade the sky's colors from bottom to top
void sky(int swidth, int sheight, double colorBot[], double colorTop[]) {
  double r, g, b, sf;
  double rShift, bShift, gShift;
  double bound = sheight / 2.5;

  rShift = colorTop[0] - colorBot[0];
  gShift = colorTop[1] - colorBot[1];
  bShift = colorTop[2] - colorBot[2];

  for(double k=0; k<=sheight; ++k) {
    sf = k/bound;

    r = colorBot[0] + sf*rShift;
    g = colorBot[1] + sf*gShift;
    b = colorBot[2] + sf*bShift;

    G_rgb(r,g,b);
    G_line(0,k, swidth,k);
  }
}

// Snow-capped Koch mountains
void mountains(int swidth, int sheight, double colorDark[], double colorLite[]) {
  double mt1[2], mt2[2], mt3[2], mt4[2];
  double darkFade[3], liteFade[3];

  // Calculate starting coordinates
  mt1[0] = swidth*(-0.4);
```

```
  mt1[1] = sheight*(0.1);
  mt2[0] = swidth*(0.6);
  mt2[1] = sheight*(0.1);
  mt3[0] = swidth*(0.4);
  mt3[1] = sheight*(0.1);
  mt4[0] = swidth*(1.1);
  mt4[1] = sheight*(0.1);

  // Convert colors to fraction for fade
  for (int i=0; i<3; ++i) {
    darkFade[i] = colorDark[i] / 255.0;
    liteFade[i] = colorLite[i] / 255.0;
  }

  // Draw mountains
  kochFlake(mt1, mt2, colorLite, 8);
  kochFlake(mt3, mt4, colorDark, 8);

  // Draw shadow
  cornerFade(darkFade, liteFade, 0.3*swidth);
}

// Fade color dark->lite from the origin to (xMax, yMax)
void cornerFade(double colDark[], double colLite[], double xMax) {
  double r, g, b, y, sc;
  double rShift, bShift, gShift;

  rShift = colLite[0] - colDark[0];
  gShift = colLite[1] - colDark[1];
  bShift = colLite[2] - colDark[2];

  y = 0;
  for (double x=0; x<=xMax; ++x) {
    sc = x / xMax;

    r = colDark[0] + sc*rShift;
    g = colDark[1] + sc*gShift;
    b = colDark[2] + sc*bShift;

    G_rgb(r,g,b) ;
    G_line(x, 0, 0, y);
    ++y;
  }
}

// Draw a solid hexagonal snowflake using Koch's curve
void kochFlake(double p1[], double p2[], double color[], int depth) {
  double p3[2], p4[2], p5[2], p6[2], pt[2];

  // Calculate p3
  pt[0] = p2[0] - p1[0];
  pt[1] = p2[1] - p1[1];
  p3[0] = p1[0] + pt[0] * cos(-piBy3) - pt[1] * sin(-piBy3);
  p3[1] = p1[1] + pt[1] * cos(-piBy3) + pt[0] * sin(-piBy3);

  // Draw snowflake
  kochCurve(p1, p2, color, 0, depth);
```

```
  kochCurve(p2, p3, color, 0, depth);
  kochCurve(p3, p1, color, 0, depth);

  // Fill in center
  Gi_rgb(color[0], color[1], color[2]);
  G_fill_triangle(p1[0], p1[1], p2[0], p2[1], p3[0], p3[1]);
}

// Recursively draw Koch's curve to a given depth
void kochCurve(double p1[], double p2[], double color[], int curr, int dep) {
  if (curr == dep) return;
  double pa[2], pb[2], pc[2], pt[2];

  // Calculate key points
  pa[0] = p1[0] + (1.0/3.0) * (p2[0] - p1[0]);
  pa[1] = p1[1] + (1.0/3.0) * (p2[1] - p1[1]);
  pt[0] = pa[0] - p1[0];
  pt[1] = pa[1] - p1[1];
  pb[0] = pa[0] + pt[0] * cos(piBy3) - pt[1] * sin(piBy3);
  pb[1] = pa[1] + pt[1] * cos(piBy3) + pt[0] * sin(piBy3);
  pc[0] = p1[0] + (2.0/3.0) * (p2[0] - p1[0]);
  pc[1] = p1[1] + (2.0/3.0) * (p2[1] - p1[1]);

  // Reinforce base line
  // and draw triangle
  if (curr+1 == dep) {
    G_rgb(1, 1, 1);  // snowcaps
  }
  G_line(p1[0], p1[1], p2[0], p2[1]);
  // Retain color for triangle
  Gi_rgb(color[0], color[1], color[2]);
  G_fill_triangle(pa[0], pa[1], pb[0], pb[1], pc[0], pc[1]);

  // Recurse
  kochCurve(p1, pa, color, curr+1, dep);
  kochCurve(pa, pb, color, curr+1, dep);
  kochCurve(pb, pc, color, curr+1, dep);
  kochCurve(pc, p2, color, curr+1, dep);
}
```

Nautiliteratorus

```
/*
Alex Staley - CS410P - August 2020

  THE NAUTILITERATORUS:
  AN IFS FRACTAL DESIGN
*/

#include "FPToolkit.c"
void translate (double dx, double dy);
void scale (double sx, double sy);
void rotate (double degrees);
double getLength(double a, double b);

double x[1] = {0};
double y[1] = {0};
int n = 1;

int main()
{
  // Set up display environment
  //G_choose_repl_display(); //for repl
  int swidth = 746; int sheight = 746;
  G_init_graphics(swidth, sheight);
  G_rgb(0.067, 0.071, 0.067);
  G_clear();
  srand48(time(0));

  double momL = getLength(99, 25);
  double momA = atan2(99, 25);

  double mainL = getLength(83, 25);
  double mainA = atan2(83, 25);
  double mainD = mainA - momA;

  double leftL = getLength(33, 35);
  double leftA = atan2(35, -33);
  double leftD = leftA - momA;

  double rightL = getLength(30, 14);
  double rightA = atan2(14, 30) - M_PI/6.0;
  double rightD = momA - rightA;

  double r, factor, xCoord, yCoord;
  double red, grn, blu;
  int j = 0;

  while (j < 250000) {
    r = drand48();

    if(r < 0.75) { //1st child
      factor = mainL / momL;
      translate(-0.5, 0);
      rotate(mainD);
      scale(factor, factor);
```

```
      translate(0.5, 4.0/25.0);
    }
    else if(r < 0.95) { //2nd child
      factor = leftL / momL;
      translate(-0.5, 0);
      rotate(leftD);
      scale(factor, factor);
      translate(0.5, 4.0/25.0);
    }
    else { //3rd child
      factor = rightL / momL;
      translate(-0.5, 0);
      rotate(rightD);
      scale(factor, 0.25);
      translate(0.5, 1.0/25.0);
    }

    // "Every time" adjustments
    rotate(M_PI/4.0);
    translate(0.94, 0.02);
    xCoord = x[0]*swidth*0.66;
    yCoord = y[0]*sheight*0.48;
    red = 0.25*y[0] + 0.11/(x[0]+0.5);
    grn = 0.41/y[0] + 0.26*x[0];
    blu = 0.4/(y[0]+0.3) + 0.11/(x[0]+0.3);
    G_rgb(red, grn, blu);
    G_point(xCoord, yCoord);
    ++j;
  }

  // Display and save image
  G_wait_key();
  G_save_to_bmp_file("nautiliteratorus.bmp");
}

// Return the magnitude of the vector
double getLength(double a, double b) {
  return sqrt(a*a + b*b);
}

// Shift a point dx and dy
void translate (double dx, double dy) {
  for (int i=0; i<n; ++i) {
    x[i] = x[i] + dx;
    y[i] = y[i] + dy;
  }
}

// Scale a point by sx and sy
void scale (double sx, double sy) {
  for (int i=0; i<n; ++i) {
    x[i] = sx * x[i];
    y[i] = sy * y[i];
  }
}

// Rotate a point r radians about (0,0)
```

```
void rotate (double r) {
  double t;
  double c = cos(r);
  double s = sin(r);

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

Mandelbrot’s Ghost

```c
/*
Alex Staley - CS410P - August 2020

  THE GHOST OF THE MANDELBROT SET
  (IT'S IMAGINARY, BUT ALSO REAL)
*/

#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "FPToolkit.c"

#define SWIDTH 746
#define SHEIGHT 746

complex double mapComplex(double x, double y);
complex double iterate(complex double c, int reps);
void plotPoint(double x, double y, complex double z, double depth);

int main()
{
  double x,y,depth;
  complex double c,z;

  // Set up display environment
  //G_choose_repl_display(); //for repl
  G_init_graphics(SWIDTH, SHEIGHT);

  int startDepth = 2;
  int endDepth = 45;
  double range = endDepth - startDepth;

  // Process each point on the screen
  // for every integer depth in the range
  for (int d=startDepth; d<=endDepth; ++d) {
    for (x=0; x<=SWIDTH; ++x) {
      for (y=0; y<=SHEIGHT; ++y) {
        c = mapComplex(x, y);
        z = iterate(c, d);
        depth = (d - startDepth) / range;
        plotPoint(x, y, z, depth);
      }
    }
  }

  // Display and save image
  G_wait_key();
  G_save_to_bmp_file("mandelbrotGhost.bmp");
  return 0;
}

// Map (x,y) to the complex plane
complex double mapComplex(double x, double y) {
  double cx = 3*x/SWIDTH - 2.15;
```

```
  double cy = 3*y/SHEIGHT - 1.5;
  return cx + cy*I;
}

// Iterate z = z^2 + c
complex double iterate(complex double c, int reps) {
  complex double z = 0;
  for (int k=0; k<reps; ++k) {
    z = z*z + c;
  }
  return z;
}

// Plot (x,y) based on divergence of z
void plotPoint(double x, double y, complex double z, double depth) {
  if (cabs(z) < 1000) {
    G_rgb(depth,depth,depth);
    G_point(y, x);
  }
}
```

Baphomet

```
/*
Alex Staley - CS410P - July 2020

  HOW TO SUMMON BAPHOMET WITH
  A PYTHAGORAS TREE FRACTAL
*/

#include  "FPToolkit.c"

void background(int swidth, int sheight, double colorBot[], double colorTop[]);
void buildHead(int swidth, int sheight);
void buildHorns(int swidth, int sheight, int depth);
void buildEyes(int swidth, int sheight, int depth);
void eyeballs(double eyeColors[], double eyeCoords[], double radius);
void pentagrams(double eyes[], double radius);
void markOfTheBeast(int swidth, int sheight, int depth);
void pyTree(double p1[], double p2[], int dep, int orient, int color[]);
void getMidCR(double lc[], double rc[], double * mc);
void getMidCL(double lc[], double rc[], double * mc);

const double piBy3 = M_PI / 3.0;
const double piBy5 = M_PI / 5.0;
const double piBy6 = M_PI / 6.0;

int main()
{
   // Screen dimensions swidth and sheight
   // determine the dimensions of Baphomet
   int swidth = 666;  //if not using repl, please
   int sheight = 666; //initialize to 666 x 666. -B
   double colorBot[3] = {0.44, 0.14, 0.05};
   double colorTop[3] = {0.03, 0.01, 0.01};

   // Set up display
   //G_choose_repl_display(); //enable if repling
   G_init_graphics(swidth,sheight);

   // Draw the background
   background(swidth, sheight, colorBot, colorTop);
   markOfTheBeast(swidth, sheight, 15);

   // Draw Baphomet
   buildHead(swidth, sheight);
   buildHorns(swidth, sheight, 12);
   buildEyes(swidth, sheight, 10);

   // Display and save image
   int key =  G_wait_key();
   G_save_to_bmp_file("baphomet.bmp");

   return 0;
}

// Fade the background colors
```

```
void background(int swidth, int sheight, double colorBot[], double colorTop[]) {
  double r, g, b, scaleFactor;
  double rShift, bShift, gShift;
  double bound = 0.5 * sheight;

  rShift = colorTop[0] - colorBot[0];
  gShift = colorTop[1] - colorBot[1];
  bShift = colorTop[2] - colorBot[2];

  for(double k=0; k<=sheight; ++k) {
    scaleFactor = k/bound;

    r = colorBot[0] + scaleFactor*rShift;
    g = colorBot[1] + scaleFactor*gShift;
    b = colorBot[2] + scaleFactor*bShift;

    G_rgb(r,g,b);
    G_line(0,k, swidth,k);
  }
}

// A circle and a triangle
void buildHead(int swidth, int sheight) {
  Gi_rgb(96, 36, 12);
  G_fill_circle(0.5*swidth, (5.0/24.0)*sheight, (11.0/60.0)*sheight);
  G_fill_triangle(0.25*swidth, (11.0/30.0)*sheight, 0.75*swidth, (11.0/30.0)*sheight,
0.5*swidth, -0.25*sheight);
}

// Two mirrored fractal ibex horns
void buildHorns(int swidth, int sheight, int depth) {
  int colorHorns[6] = {39, 33, 33, 175, 4, 4};
  double p1[2], p2[2], p3[2], p4[2];

  // X-coordinates of horn roots
  p1[0] = 0.365 * swidth;
  p4[0] = 0.635 * swidth;
  p2[0] = p1[0] + 0.0625 * swidth;
  p3[0] = p4[0] - 0.0625 * swidth;

  // Y-coordinates of horn roots
  p1[1] = 0.35 * sheight;
  p4[1] = 0.35 * sheight;
  p2[1] = p1[1] + 0.025 * sheight;
  p3[1] = p4[1] + 0.025 * sheight;

  pyTree(p1, p2, depth, 'L', colorHorns);
  pyTree(p3, p4, depth, 'R', colorHorns);
}

// Windows into Hell
void buildEyes(int swidth, int sheight, int depth) {
  double colorEyes[9] = {0.92, 0.51, 0.26, 0.9, 0.09, 0.03, 0.96, 0.85, 0.39};
  double eyes[4];  //centers of eyes
  double radius = (1.0 / 15.0) * sheight; //eyeball radius

  // Calculate eye coordinates
```

```
  eyes[0] = 0.415 * swidth;
  eyes[1] = 0.19 * sheight;
  eyes[2] = 0.585 * swidth;
  eyes[3] = 0.19 * sheight;

  // Draw eyes
  eyeballs(colorEyes, eyes, radius);
  pentagrams(eyes, radius);
}

// Fiery shaded circles
void eyeballs(double eyeColors[], double eyeCoords[], double radius) {
  double r, g, b, shadeScale;  //eye color fading utils
  double inRadius = 0.3 * radius;  //iris radius
  double rShift = eyeColors[3] - eyeColors[0];   //shading factor r
  double gShift = eyeColors[4] - eyeColors[1]; //shading factor g
  double bShift = eyeColors[5] - eyeColors[2];  //shading factor b

  // Outer color fade (NOT APPROVED BY BAPHOMET)
  for (double i=inRadius; i<radius; ++i) {
    shadeScale = i / radius;
    r = eyeColors[0] + shadeScale * rShift;
    g = eyeColors[1] + shadeScale * gShift;
    b = eyeColors[2] + shadeScale * bShift;

    G_rgb(r, g, b);
    G_circle(eyeCoords[0], eyeCoords[1], i);
    G_circle(eyeCoords[2], eyeCoords[3], i);
  }

  // Inner color fade
  rShift = eyeColors[6] - eyeColors[3];
  gShift = eyeColors[7] - eyeColors[4];
  bShift = eyeColors[8] - eyeColors[5];
  for(double j=0; j<=inRadius; ++j) {
    shadeScale = j / inRadius;
    r = eyeColors[3] + shadeScale * rShift;
    g = eyeColors[4] + shadeScale * gShift;
    b = eyeColors[5] + shadeScale * bShift;

    G_rgb(r, g, b);
    G_circle(eyeCoords[0], eyeCoords[1], j);
    G_circle(eyeCoords[2], eyeCoords[3], j);
  }
}

// Inscribe pentagrams in eyes
void pentagrams(double eyes[], double r) {
  double xPentL[5], yPentL[5], xPentR[10], yPentR[10];
  double arm = 2 * r * sin(2*piBy5);   //long segment
  double leg = 2 * r * sin(piBy5);    //short segment
  double face = eyes[2] - eyes[0]; //distance between eyes

  // Calculate pentagram coordinates for left eye
  xPentL[0] = eyes[0];
  xPentL[1] = xPentL[0] + arm * cos(2*piBy5);
  xPentL[3] = xPentL[0] + leg * cos(piBy5);
```

```
  xPentL[2] = xPentL[3] - arm;
  xPentL[4] = xPentL[1] - leg;
  yPentL[0] = eyes[1] - r;
  yPentL[1] = yPentL[0] + arm * sin(2*piBy5);
  yPentL[3] = yPentL[0] + leg * sin(piBy5);
  yPentL[2] = yPentL[3];
  yPentL[4] = yPentL[1];

  // Right eye
  for (int i=0; i<6; ++i) {
    xPentR[i] = xPentL[i] + face;
    yPentR[i] = yPentL[i];
  }

  // Draw pentagrams
  G_rgb(0, 0, 0);
  G_polygon(xPentL, yPentL, 5);
  G_polygon(xPentR, yPentR, 5);
}

// 666
void markOfTheBeast(int swidth, int sheight, int depth) {
  double p1a[2], p1b[2], p2a[2], p2b[2], p3a[2], p3b[2];
  int colorMark[6] = {115, 8, 8, 0, 0, 0};

  // Six hundred
  p1a[0] = 0.3 * swidth + 0.05 * swidth;
  p1a[1] = 0.9 * sheight;
  p1b[0] = p1a[0] - 0.002 * swidth;
  p1b[1] = p1a[1] + 0.027 * sheight;

  // Sixty
  p2a[0] = 0.5 * swidth + 0.05 * swidth;
  p2a[1] = p1a[1];
  p2b[0] = p2a[0] - 0.002 * swidth;
  p2b[1] = p1b[1];

  // Six
  p3a[0] = 0.7 * swidth + 0.05 * swidth;
  p3a[1] = p1a[1];
  p3b[0] = p3a[0] - 0.002 * swidth;
  p3b[1] = p1b[1];

  pyTree(p1a, p1b, depth, 'L', colorMark);
  pyTree(p2a, p2b, depth, 'L', colorMark);
  pyTree(p3a, p3b, depth, 'L', colorMark);
}

// Build a Pythagoras tree fractal
void pyTree(double p1[], double p2[], int dep, int orient, int color[]) {
  if (dep == 0) {
    return;
  }

  double leg1, leg2;  //baseline components
  double rectX[4], rectY[4];    //rectangle
  double lc[2], rc[2], mc[2];    //triangle
```

```
  leg1 = 1.5 * (p2[0] - p1[0]);
  leg2 = 1.5 * (p2[1] - p1[1]);

  // Define rectangle
  rectX[0] = p1[0];
  rectX[1] = p1[0]-leg2;
  rectX[2] = p2[0]-leg2;
  rectX[3] = p2[0];
  rectY[0] = p1[1];
  rectY[1] = p1[1]+leg1;
  rectY[2] = p2[1]+leg1;
  rectY[3] = p2[1];

  // Draw rectangle
  Gi_rgb(color[0], color[1], color[2]);
  G_fill_polygon(rectX, rectY, 4);

  // Define triangle
  lc[0] = rectX[1];
  lc[1] = rectY[1];
  rc[0] = rectX[2];
  rc[1] = rectY[2];
  if (orient == 'L') {
    getMidCL(lc, rc, mc);
  }
  else getMidCR(lc, rc, mc);

  // Draw triangle
  Gi_rgb(color[3], color[4], color[5]);
  G_fill_triangle(lc[0], lc[1], mc[0], mc[1], rc[0], rc[1]);

  // Recursed
  pyTree(lc, mc, dep-1, orient, color);
  pyTree(mc, rc, dep-1, orient, color);
}

// Define third triangle vertex (L/R)
void getMidCR(double lc[], double rc[], double * mc) {
  double t[2];
  t[0] = 0.5 * (rc[0] - lc[0]);
  t[1] = 0.5 * (rc[1] - lc[1]);
  mc[0] = lc[0] + t[0] * cos(piBy3) - t[1] * sin(piBy3);
  mc[1] = lc[1] + t[1] * cos(piBy3) + t[0] * sin(piBy3);
}

void getMidCL(double lc[], double rc[], double * mc) {
  double t[2];
  t[0] = 0.5 * sqrt(3) * (rc[0] - lc[0]);
  t[1] = 0.5 * sqrt(3) * (rc[1] - lc[1]);
  mc[0] = lc[0] + t[0] * cos(piBy6) - t[1] * sin(piBy6);
  mc[1] = lc[1] + t[1] * cos(piBy6) + t[0] * sin(piBy6);
}
```

Desert Life

```
/*
Alex Staley - CS410P - July 2020

  AN L-SYSTEM TREE GUARDING
  A WINDSWEPT DESERT ROAD
*/

#include  "FPToolkit.c"
const double piBy2 = M_PI / 2.0;
const double piBy3 = M_PI / 3.0;
const double piBy4 = M_PI / 4.0;
const double piBy6 = M_PI / 6.0;
const double piBy8 = M_PI / 8.0;

#define MAX_SIZE 1000000

void stringWrapper();
void stackBuilder();
void stringBuilder(int curr, int depth);
void stringInterpreter(int start[2], double length, double angle, double gangle);
void autoFit(int swidth, int sheight, double angle, double gangle, double * placement);
void pushState();
void popState();

// Production: nonterminal -> rule
typedef struct {
  char nonterminal;
  char rule[100];
} Production;

// Stacks to track current state
typedef struct {
  double x[MAX_SIZE]; //x location
  double y[MAX_SIZE]; //y location
  double a[MAX_SIZE]; //heading (radians)
  int xI;
  int yI;
  int aI;
} Stack;

// Defined globally:
Stack stack;
Production prods[10]; //array of <=10 productions
int numProds = 0;     //number of prods defined
char axiom[2] = {'A', '\0'};       //starting string
char derivation[MAX_SIZE] = {'\0'}; //derived string
double heading = 0; //current angle from zero in radians
double here[2];     //current (x, y) position


int main() {
  int start[2];
  double placement[3];
  double gangle = piBy2; //DEFINE GANGLE HERE
```

```
int depth;
double length;

// Set up
srand(time(0));
int key = 0;
int swidth = 746; int sheight = 746;
double horizon = sheight * 0.618;
double road = swidth * 0.618;
//G_choose_repl_display(); //for repl
G_init_graphics (swidth,sheight);

// Background
Gi_rgb(217, 191, 119);
G_clear();
Gi_rgb(178, 235, 242);
for (double i=horizon; i<=sheight; ++i) {
  G_line(0, i, swidth, i);
}

// Road
Gi_rgb(18, 13, 2);
for (double r=road; r<swidth*1.618; ++r) {
  if (r > swidth*0.99) Gi_rgb(245, 201, 13);
  if (r > swidth * 1.03) Gi_rgb(18, 13, 2);
  if (r > swidth*1.06) Gi_rgb(245, 201, 13);
  if (r > swidth * 1.10) Gi_rgb(18, 13, 2);
  G_line(road + r*0.01, horizon, r, 0);
}
G_line(road, 0, road, horizon);
G_line(road, 0, road+1, horizon);
G_line(road, 0, road+2, horizon);
G_line(road, 0, road+3, horizon);

int xSand, ySand;
Gi_rgb(217, 191, 119);
for (int i=0; i<700; ++i) {
  xSand = rand() % ((int)(swidth*1.618)) + swidth*0.6;
  ySand = rand() % ((int)horizon);
  G_point(xSand, ySand);
}

// Derive and build string
stringWrapper();
stackBuilder();

// Determine line dimensions
autoFit(swidth, sheight, piBy8, gangle, placement);
start[0] = placement[0] - (swidth / 9.0); //scoot for road
start[1] = placement[1];
length = placement[2];

// Draw tree
Gi_rgb(43, 88, 12);
stringInterpreter(start, length, piBy8, gangle);
```

```
  // Draw shadow
  length = length * 0.6;
  gangle = piBy2 + piBy6;
  Gi_rgb(6, 13, 2);
  stringInterpreter(start, length, piBy8, gangle);

  // Display and save image
  key = G_wait_key();
  G_save_to_bmp_file("desertLife.bmp");

  return 0;
}

// Populate the global prods[] array.
// DEFINE PRODUCTIONS HERE!
void stringWrapper(/*int numProds, Production prods[]*/) {
  // Define nonterminals and
  // associated derivations,
  // then increment numProds
  prods[0].nonterminal = 'A';
  strcpy(prods[0].rule, "B-[[A]+A]+B[+BA]-A");
  ++numProds;

  prods[1].nonterminal = 'B';
  strcpy(prods[1].rule, "BB");
  ++numProds;

  // Build string
  stringBuilder(0, 6);
}

void stackBuilder() {
  stack.x[0] = '\0';
  stack.y[0] = '\0';
  stack.a[0] = '\0';
  stack.xI = -1;
  stack.yI = -1;
  stack.aI = -1;
}

void pushState() {
  if (stack.xI < MAX_SIZE-1) {
    stack.xI += 1;
    stack.x[stack.xI] = here[0];
  }
  if (stack.yI < MAX_SIZE-1) {
    stack.yI += 1;
    stack.y[stack.yI] = here[1];
  }
  if (stack.aI < MAX_SIZE-1) {
    stack.aI += 1;
    stack.a[stack.aI] = heading;
  }
}

void popState() {
  if (stack.xI >=0) {
```

```
    here[0] = stack.x[stack.xI];
    stack.xI -=1;
  }
  if (stack.yI >= 0) {
    here[1] = stack.y[stack.yI];
    stack.yI -= 1;
  }
  if (stack.aI >= 0) {
    heading = stack.a[stack.aI];
    stack.aI -= 1;
  }
}

// Determine the length and starting point
// to ideally fit the screen, given its dimensions
void autoFit(int swidth, int sheight, double angle, double gangle, double * placement) {
  double xMin=0; double yMin=0;
  double xMax=0; double yMax=0;
  double dX=0; double dY=0;
  double ndX = 0.9*swidth;  //this,
  double ndY = 0.9*sheight; //XOR this
  double there[2];  //destination point container

  int i = 0;
  heading = gangle;
  here[0] = 0;
  here[1] = 0;

  // Loop over derived string
  while (derivation[i] != '\0') {
    if (derivation[i] == '[') { //push state
      pushState();
    }
    else if (derivation[i] == ']') { //pop state
      popState();
    }
    else if (derivation[i] == '-') { //turn clockwise
      heading -= angle;
    }
    else if (derivation[i] == '+') { //turn counterclockwise
      heading += angle;
    }
    else if ((derivation[i] >= 'A' && derivation[i] <='Z')
        || derivation[i] == 'f') { //move forward
      there[0] = here[0] + cos(heading);
      there[1] = here[1] + sin(heading);
      here[0] = there[0];
      here[1] = there[1];

      // Check current location against current extrema
      if (here[0] < xMin) xMin = here[0];
      if (here[0] > xMax) xMax = here[0];
      if (here[1] < yMin) yMin = here[1];
      if (here[1] > yMax) yMax = here[1];
    }

    ++i;
```

```
  }

  // Calculate scaling factors
  dX = xMax - xMin;
  dY = yMax - yMin;
  if (dY > dX) {
    ndX = dX * (ndY / dY);
  }
  else {
    ndY = dY * (ndX / dX);
  }

  // Calculate starting point and length,
  // accounting for negative min values
  placement[0] = 0.5 * (swidth - ndX);  //startX
  if (xMin < 0) placement[0] -= (xMin * (ndX/dX));
  placement[1] = 0.5 * (sheight - ndY); //startY
  if (yMin < 0) placement[1] -= (yMin * (ndY/dY));
  placement[2] = ndX / dX;          //length
}

void stringInterpreter(int start[2], double length, double angle, double gangle) {
  heading = gangle;
  here[0] = start[0];
  here[1] = start[1];
  double there[2];
  int i = 0;

  // Loop over derived string
  while (derivation[i] != '\0') {
    if (derivation[i] == '[') { //push state
      pushState();
    }
    else if (derivation[i] == ']') { //pop state
      popState();
    }
    else if (derivation[i] == '-') { //turn clockwise
      heading -= angle;
    }
    else if (derivation[i] == '+') { //turn counterclockwise
      heading += angle;
    }
    else if ((derivation[i] >= 'A' && derivation[i] <='Z')
        || derivation[i] == 'f') { //move forward
      there[0] = here[0] + length * cos(heading);
      there[1] = here[1] + length * sin(heading);
      G_line(here[0], here[1], there[0], there[1]);
      here[0] = there[0];
      here[1] = there[1];
    }

    ++i;
  }
}

// Derive a string to the given depth
// MUST DEFINE PRODS W/ stringWrapper() BEFORE CALLING!
```

```
void stringBuilder(int curr, int depth) {
  if (derivation[0] == '\0') {
    // Start with global axiom
    strcpy(derivation, axiom);
  }

  if (curr == depth) return;

  int sI=0;  int pI=0; //starting string, prod list indices
  int derived=0; //flag for nonterminal found this iteration
  char let[2];  let[1] = '\0'; //current letter as a string
  char temp[MAX_SIZE]; //temporary derived string container

  // Loop over existing string
  while (derivation[sI] != '\0') {
    let[0] = derivation[sI];
    // Loop over all possible productions
    while (pI < numProds && derived == 0) {
      // If a nonterminal is found,
      // apply the appropriate rule
      if (derivation[sI] == prods[pI].nonterminal) {
        strcat(temp, prods[pI].rule);
        derived = 1;
      }
      ++pI;
    }
    // If the character is a
    // terminal, copy it to temp
    if (derived == 0) strcat(temp, let);

    // Increment/reset loop indices
    ++sI;
    pI = 0;
    derived = 0;
  }

  // Copy derived string and recurse
  strcpy(derivation, temp);
  stringBuilder(curr+1, depth);
}
```